Datové typy

- jaká data jsou potřebné pro řešení problému?
- jak se budou data reprezentovat?
- jaké operace se budou nad daty provádět?

Datový typ

- rozsah hodnot, které může nabývat proměnná daného datového typu
- množina operací, které jsou pro daný datový typ povolené / definované
- nezávisí na konkrétní implementaci

Datové typy a struktury

jednoduchý (skalární) datový typ

data zabírají vždy konstantní (typicky malé) množství paměti, zpřístupnění hodnoty skalárního typu trvá konstantní čas

číselné a znakové typy, typ pravdivostních hodnot, výčtový typ

složený datový typ

implementace složeného datového typu se nazývá datová struktura

- **statický** - pevná velikost; časová složitost zpřístupnění prvku je konstantní
  k-tice, pole konstantní délky
- **dynamický** - neomezená velikost; časová složitost zpřístupnění prvku je funkcí závislou na velikosti
  seznam, zásobník, fronta, slovník, strom, graf

Dynamické datové typy

- množina objektů; v průběhu výpočtu můžeme do množiny prvky přidávat a odebírat resp. množinu jinak modifikovat (*tzv. dynamická množina)*
- každý prvek dynamické množiny je reprezentovaný jako objekt, jehož atributy můžeme zkoumat a modifikovat za předpokladu, že máme ukazatel / referenci na tento objekt
- jeden z atributů objektu je jeho identifikátor - klíč key
- jestliže všechny prvky mají různé klíče, často mluvíme o množině obsahující klíče

Dynamické datové typy - základní operace

$\textsc{Search}(S, k)$ pro množinu S a klíč k vrátí ukazatel x takový, že $x.key = k$ resp. $\textsc{Nil}$, když objekt s klíčem k není obsažen v množině S

$\textsc{Insert}(S, x)$ do množiny S vloží objekt s ukazatelem x

$\textsc{Delete}(S, x)$ z množiny S odstraní objekt s ukazatelem x

$\textsc{Maximum}(S)$ pro množinu S s úplně uspořádanými objekty vrátí ukazatel x na objekt, jehož klíč je maximální

$\textsc{Minimum}(S)$ pro množinu S s úplně uspořádanými objekty vrátí ukazatel x na objekt, jehož klíč je minimální

$\textsc{Successor}(S, x)$ pro množinu S s úplně uspořádanými objekty vrátí ukazatel na objekt, jehož klíč následuje bezprostředně za klíčem $x.key$, resp. hodnotu $\textsc{Nil}$ když x je maximální

$\textsc{Predecessor}(S, x)$ symetricky k $\textsc{Successor}$

Datové struktury

Problém rezervací

online rezervační systém
(*např. rezervace lékařského vyšetření, přistávací ranveje, ...*)

- množina rezervací R
- požadavek t na rezervaci
- rezervace může být potvrzena právě když v intervalu $(t-k, t+k)$ není žádná jiná rezervace (k je délka trvání události) a současně t je aktuální
- mazání realizovaných aktualizací

příklad: $R = \{21, 26, 29, 36\}$, $k = 3$, aktuální čas 20

rezervace 24 není validní (možná), protože 26 $\in R$
33 je OK
15 není validní, protože aktuální čas je 20

Problém rezervací - řešení

jaký datový typ je vhodný pro reprezentaci R a realizaci požadovaných operací???

uspořádaný seznam
ověření rezervace v čase $\mathcal{O}(n)$, záznam rezervace v čase $\mathcal{O}(1)$

uspořádané pole
ověření rezervace v čase $\mathcal{O}(\log n)$, záznam rezervace v čase $\mathcal{O}(n)$

neuspořádaný seznam / pole
ověření rezervace v čase $\mathcal{O}(n)$, záznam rezervace v čase $\mathcal{O}(1)$

minimová halda
ověření rezervace v čase $\mathcal{O}(n)$, záznam rezervace v čase $\mathcal{O}(\log n)$, aktuálnost rezervace v čase $\mathcal{O}(1)$

binární pole rezervace t je uložena v položce s indexem t – problém velikosti pole

existuje lepší řešení?? současně efektivní vyhledávání i vkládání!

Vyhledávací stromy

- umožňují efektivní implementaci operací SEARCH, MINIMUM, MAXIMUM, PREDECESSOR, SUCCESSOR, INSERT, DELETE
- operace nad vyhledávacím stromem mají složitost **úměrnou hloubce stromu**, tj. v nejhorším případě až lineární

- **binární vyhledávací stromy** - každý vrchol stromu má nejvýše 2 následníky, tj. strom může mít až hloubku n
- **vyvážené binární vyhledávací stromy** mají logaritmickou hloubku, tj. operace mají složitost $\mathcal{O}(\log n)$

Binární vyhledávací stromy (BVS)

- datová struktura, která využívá ukazatele
- každý vrchol (uzel) stromu představuje jeden objekt
- každý vrchol obsahuje
  - klíč
  - ukazatele *left*, *right* a *p* na levého syna, pravého syna a na otce; ukazatel má hodnotu *Nil* právě když vrchol nemá příslušného syna, resp. otce
  - případné další data

v binárním vyhledávacím stromu jsou klíče vždy uloženy tak, že platí

BVS vlastnost

jestliže x je vrchol BVS a

y je vrchol v levém podstromu vrcholu x, tak platí $y.key \leq x.key$

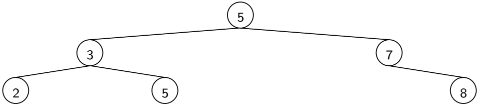
y je vrchol v pravém podstromu vrcholu x, tak platí $y.key \geq x.key$

BVS - procházení stromu

- cílem je projít strom tak, aby každý vrchol byl navštíven právě jednou
- využití: provedení operace nad každým vrcholem, výpis klíčů, kontrola vlastností stromu, ...

- strom procházíme rekurzivně
- začínáme v kořeni stromu
- (rekurzivně) navštívíme všechny vrcholy levého podstromu kořene
- (rekurzivně) navštívíme všechny vrcholy pravého podstromu kořene

BVS - výpis klíčů



klíče uložené v BVS můžeme vypsat v pořadí

inorder hodnotu klíče uloženého v kořeni vypíšeme **mezi** vypsáním klíčů uložených v jeho levém a pravém podstromě (2 3 5 5 7 8)

preorder hodnotu klíče uloženého v kořeni vypíšeme **před** vypsáním klíčů uložených v jeho levém a pravém podstromě (5 3 2 5 7 8)

postorder hodnotu klíče uloženého v kořeni vypíšeme **po** vypsání klíčů uložených v jeho levém a pravém podstromě (2 5 3 8 7 5)

Inorder

Inorder_Tree_Walk(x)

```
if x ≠ Nil
   then INORDER_TREE_WALK(x.left)
        print x.key
        INORDER_TREE_WALK(x.right)
fi
```

- $\textsc{Inorder\_Tree\_Walk}(T.root)$ vypíše klíče uložené v BVS T **od nejmenšího po největší**
- časová složitost je $\Theta(n)$, kde n je počet vrcholů stromu T
- $\textsc{BVS Sort}$ - časová složitost ???

BVS - vyhledávání ve stromu

- začínáme v kořeni stromu, postupujeme rekurzivně
- porovnáme hledaný klíč k s klíčem uloženým v navštíveném uzlu, jestliže se rovnají, tak vyhledávání končí úspěchem
- jestliže hledaný klíč k je **menší** než klíč $x.key$ uložený v navštíveném uzlu x, tak pokračujeme v **levém** podstromu uzlu x
- v opačném případě pokračujeme v pravém podstromu uzlu x
- vyhledávání končí neúspěchem právě když hledaný klíč není uložen ani v navštíveném listu

Tree_Search(x, k)

if $x = Nil \vee k = x.key$
    then return x **fi**
if $k < x.key$
    then return $\textsc{Tree\_Search}(x.left, k)$
     else return $\textsc{Tree\_Search}(x.right, k)$ **fi**

BVS - minimální a maximální klíč

- jestliže hledáme **minimální** klíč, tak v stromu postupujeme vždy **doleva**
- jestliže hledáme **maximální** klíč, tak v stromu postupujeme vždy **doprava**

Tree_Minimum(x)

while $x.left \neq Nil$ **do** $x \leftarrow x.left$ **od**
return x

Tree_Maximum(x)

while $x.right \neq Nil$ **do** $x \leftarrow x.right$ **od**
return x

BVS - předchůdce a následník

- předpokládáme, že všechny klíče uložené v stromě jsou vzájemně různé[1]
- **následníkem** uzlu x je uzel, který obsahuje **nejmenší klíč větší než** $x.key$ *(successor)*
- **předchůdcem** uzlu x je uzel, který obsahuje **největší klíč menší než** $x.key$ *(predecessor)*

[1] analogicky se operace definují i pro strom, který může obsahovat uzly se stejnými klíči

následníkem uzlu x je uzel, který obsahuje **nejmenší klíč větší než** $x.key$

- jestliže uzel x má neprázdný pravý podstrom, tak jeho následníkem je nejmenší klíč uložený v jeho pravém podstromu
- jestliže pravý podstrom je prázdný, tak
  - následníkem x je uzel y takový, že $x.key$ je největším klíčem v levém podstromu uzlu y
  - uzel y je prvním uzlem na cestě z x do kořene stromu takový, že $y.key > x.key$ (*jinými slovy x patří do levého podstromu uzlu y*)

Tree_Successor(x)

```
if x.right ≠ Nil
  then return TREE_MINIMUM(x.right) fi
y ← x.p
while y ≠ Nil ∧ x = y.right
      do x ← y
         y ← y.p
od
return y
```

BVS - přidání nového uzlu

- procházíme strom stejně jako kdyby jsme klíč nového uzlu vyhledávali
- hledáme vrchol, jehož příslušný podstrom je prázdný (levý podstrom když klíč nového uzlu je menší než klíč vrcholu, pravý podstrom když je větší) a nový uzel se stane jeho příslušným synem

Tree_Insert(T, z)

$y \leftarrow Nil$
$x \leftarrow T.root$
while $x \neq Nil$ **do**
    $y \leftarrow x$
    if $z.key < x.key$ **then** $x \leftarrow x.left$
                        else $x \leftarrow x.right$ **fi**
od
$z.p \leftarrow y$
if $y = Nil$ **then** $T.root \leftarrow z$
            else if $z.key < y.key$ **then** $y.left \leftarrow z$
                                    else $y.right \leftarrow z$ **fi**
fi

BVS - odstranění uzlu - případ 1

při odstraňování uzlu lz, můžou nastat 3 případy

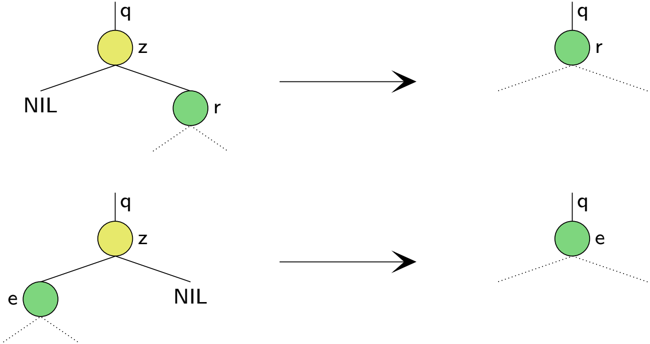
z nemá žádného syna

uzel odstraníme

BVS - odstranění uzlu - případ 2

z má jediného syna

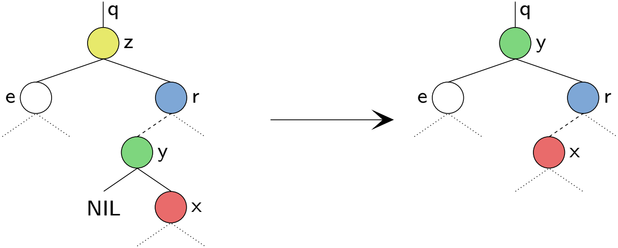
syna přesuneme na pozici uzlu z tak, že otec uzlu z se stane otcem jeho syna



BVS - odstranění uzlu - případ 3

z má dva syny

- potřebujeme najít uzel y, který nahradí uzel z
- vhodným kandidátem na y je následník uzlu z *(symetricky by jsme mohli využít předchůdce uzlu z)*
- protože pravý podstrom uzlu z je neprázdný, tak následník y uzlu z je nejmenším uzlem v pravém podstromě uzlu z
- y nemá levého syna proto ho můžeme ho přesunout na pozici z



BVS - přesun podstromů

- Transplant nahradí podstrom s kořenem u podstromem s kořenem v
- otcem uzlu v se stane otec uzlu u
- otec uzlu u bude mít uzel v jako svého syna

Transplant(T, u, v)

if $u.p = Nil$ **then** $T.root \leftarrow v$
 else if $u = u.p.left$ **then** $u.p.left \leftarrow v$
  else $u.p.right \leftarrow v$
 fi
fi
if $v \neq Nil$ **then** $v.p \leftarrow u.p$ **fi**

BVS - odstranění uzlu

Tree_Delete(T, z)

if $z.left = Nil$
  then $\textsc{Transplant}(T, z, z.right)$
  else if $z.right = Nil$
    then $\textsc{Transplant}(T, z, z.left)$
    else $y \leftarrow \textsc{Tree\_Minimum}(z.right)$
      if $y.p \neq z$ **then** $\textsc{Transplant}(T, y, y.right)$
        $y.right \leftarrow z.right$
        $y.right.p \leftarrow y$
      fi
      $\textsc{Transplant}(T, z, y)$
      $y.left \leftarrow z.left$
      $y.left.p \leftarrow y$
  fi
fi

Složitost

- všechny uvedené operace nad binárním vyhledávacím stromem mají složitost úměrnou hloubce stromu, tj. v nejhorším případě $\mathcal{O}(n)$, kde n je počet uzlů stromu
- při hledání předchůdce a následníka nemusíme vůbec porovnávat klíče
- operace se dají využít k seřazení klíčů např. tak, že najdeme minimální klíč a pak (rekurzivně) jeho následníka (složitost?!)

Vyvážené binární vyhledávací stromy

hloubka stromu je logaritmická

složitost operací je úměrná hloubce stromu

- AVL stromy
- 2 - 3 stromy
- 2 - 3 - 4 stromy
- B stromy
- červeno černé stromy

Modifikace datových struktur

- reálné situace, ve kterých potřebujeme datovou strukturu odlišnou od „*učebnicových struktur*“
- ??? účelnost návrhu úplně nové struktury
- možné řešení:
  - rozšíření některé známé struktury o nové informace
  - návrh nových operací nad takto rozšířenou strukturou
  - . . . při zachování efektivnosti původních operací

příklad: využití binárních vyhledávacích stromů pro reprezentaci množiny intervalů

Reprezentace intervalů

- číselný interval $\langle t_1, t_2 \rangle$
- objekt i s atributy $i.low$ a $i.high$
- intervaly i a i' se překrývají právě když $i.low \leq i'.high$ a současně $i'.low \leq i.high$
- pro libovolné dva intervaly i a i' platí právě jedna z možností
  - intervaly se překrývají
  - interval i je vlevo od i', tj. $i.high < i'.low$
  - interval i' je vlevo od i, tj. $i'.high < i.low$

hledáme datovou strukturu pro reprezentaci množiny intervalů nad kterou je možné efektivně implementovat operace

- $\textsc{Interval\_Insert}(T, x)$ – do množiny intervalů T přidá objekt reprezentující interval x
- $\textsc{Interval\_Delete}(T, x)$ – z množiny intervalů T odstraní objekt reprezentující interval x
- $\textsc{Interval\_Search}(T, i)$ – vrátí ukazatel na objekt, který reprezentuje interval překrývající se s intervalem i resp. hodnotu Nil, když takový objekt neexistuje

řešení 1

- seznam intervalů
- přidání intervalu v konstantním čase
- odebrání a vyhledání intervalu v čase $\mathcal{O}(n)$ *(n je počet intervalů v množině)*

řešení 2

- uspořádaný seznam intervalů
- všechny operace v čase $\mathcal{O}(n)$

intervalové stromy

- rozšíření binárních vyhledávacích stromů

Intervalové stromy

- binární vyhledávací strom
- **každý uzel má atributy $i.low$, $i.high$, a $i.max$**
- **jako klíč je použita hodnota $x.low$**

- $x.max$ je maximální hodnota krajního bodu intervalu uloženého v podstromu s kořenem x

$$x.max = \max\{x.high,\ x.left.max,\ x.right.max\}$$

Přidání nového intervalu

- postupujeme jako v BVS od kořene, vkládaný uzel se stane listem
- každému uzlu y na cestě z kořene do nového uzlu x aktualizujme hodnotu $y.max$ právě když $y.max < x.high$
- pro žádný vrchol neležící na cestě z kořene do nového uzlu se hodnota max nemění

Odstranění intervalu

- postupujeme jako v BVS
- na pozici odstraněného uzlu se přesune uzel y
- pro aktualizaci hodnot max procházíme cestu od původní pozice uzlu y do kořene a každému uzlu z na této cestě aktualizujeme hodnotu $z.max = \max\{z.left.max, z.right.max, z.high\}$
- složitost operace se navýší o $\mathcal{O}(n)$
- celková složitost operace odstranění intervalu zůstává asymptoticky stejná

Vyhledávání intervalu

Interval_Search(T, i)

```
x ← T.root
while x ≠ Nil ∧ intervaly i a (x.low, x.high) se nepřekrývají do
        if x.left ≠ Nil ∧ x.left.max ≥ i.low
           then x ← x.left
            else x ← x.right fi
od
return x
```

složitost

- vyhledávání začíná v kořeni
- po každé iteraci cyklu testujeme uzel, jehož hloubka je o 1 vyšší
- složitost je úměrná hloubce stromu

Vyhledávání intervalu

Interval_Search(T, i)

$x \leftarrow T.root$
while $x \neq Nil \wedge$ intervaly i a $\langle x.low, x.high \rangle$ se nepřekrývají **do**
    if $x.left \neq Nil \wedge x.left.max \geq i.low$
        then $x \leftarrow x.left$
        else $x \leftarrow x.right$ **fi od**
return x

korektnost - případ 1 - ve vyhledávání postupujeme doprava

- předpokládejme, že ve vyhledávání postupujeme z uzlu x doprava a levý podstrom **není** prázdný
- platí $x.left.max < i.low$ *(jinak by jsme postupovali doleva)*
- pro každý interval $\langle a, b \rangle$ z levého podstromu platí $b \leq x.left.max$ *(z definice hodnoty max)*
- $b < i.low$ znamená, že i se nepřekrývá se žádným intervalem v levém podstromu

Vyhledávání intervalu

Interval_Search(T, i)

$x \leftarrow T.root$
while $x \neq Nil \wedge$ intervaly i a $\langle x.low, x.high\rangle$ se nepřekrývají **do**
  if $x.left \neq Nil \wedge x.left.max \geq i.low$
    then $x \leftarrow x.left$
    else $x \leftarrow x.right$ **fi**
od
return x

korektnost - případ 2 - ve vyhledávání postupujeme doleva

- předpokládejme, že žádný interval levého podstromu se nepřekrývá s i
- v levém podstromu leží interval $\langle c, d\rangle$ takový, že $d = x.left.max$
- $i.high < c$ *(i a $\langle c, d\rangle$ se nepřekrývají)*
- pro každý interval $\langle a, b\rangle$ z pravého podstromu platí $c \leq a$ *(vlastnost BVS)*
- $i.high < a$ znamená, že i se nepřekrývá se žádným intervalem v pravém podstromu

Intervalové stromy - modifikace

- vyhledávání **všech** překrývajících se intervalů
- intervaly vyšší dimenze
- namísto obecného binárního vyhledávacího stromu můžeme použít vyvážený binární vyhledávací strom

Radix trees

- využití binárních vyhledávacích stromů pro lexikografické řazení binárních řetězců
- řetězce postupně vkládáme do vyhledávacího stromu
- po vložení všech řetězců strom prohledáme a klíče vypíšeme v pořadí preorder
- časová složitost je $\Theta(n)$, kde n je součet délek všech řetězců

- zobecnění pro řetězce nad libovolnou abecedou - použijeme stromy, jejichž arita je stejná jako velikost abecedy

Datové struktury

Červeno černé stromy

Červeno černý strom je binární vyhledávací strom, jehož každý uzel je obarvený červenou anebo černou barvou a splňuje podmínky

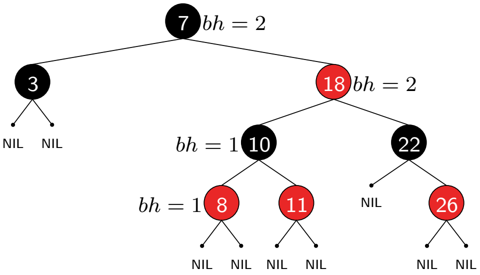
1. kořen stromu je černý
2. listy stromu nenesou žádnou hodnotu, tj. jsou označeny *Nil*, a mají černou barvu
3. když je uzel červený, tak jeho otec je černý
4. pro každý uzel x stromu platí, že všechny cesty z uzlu x do listů obsahují stejný počet černých uzlů

alternativně: oba synové červeného uzlu mají černou barvu

- každý uzel obsahuje atributy *key, color, left, right, p*
- jestliže uzel nemá některého syna anebo otce, tak příslušný atribut má hodnotu Nil

Výška uzlu a černá výška uzlu

- **výška** uzlu x je rovna počtu hran na nejdelší cestě z x do listu
- **černá výška** uzlu x, $bh(x)$, je rovna počtu **černých** uzlů na cestě z x do listu (uzel x nezapočítáváme)
  (díky vlastnosti 4 je černá výška dobře definovaná!)



Výška červeno černého stromu

Lema 1

Každý uzel s výškou h má černou výšku alespoň $h/2$.

z vlastnosti 4 plyne, že v nejhorším případě je každý druhý uzel na cestě červený

Lema 2

Pro každý uzel x platí, že podstrom s kořenem x má alespoň $2^{bh(x)} - 1$ vnitřních uzlů[2].

důkaz indukcí k výšce h uzlu x

$h = 0$ x je list $\Longrightarrow bh(x) = 0$ a současně počet vnitřních uzlů podstromu s kořenem x je 0

$h > 0$
- nechť x má výšku h a černou výšku $bh(x) = b$
- každý syn uzlu x má výšku $h - 1$ a černou výšku b anebo $b - 1$
- z indukčního předpokladu má podstrom každého syna alespoň $2^{bh(x)-1} - 1$ vnitřních uzlů
- podstrom s kořenem x má alespoň $2(2^{bh(x)-1} - 1) + 1 = 2^{bh(x)} - 1$ vnitřních uzlů

[2] vnitřním uzlem rozumíme uzel, který nese hodnotu, tj. list není vnitřním uzlem

Výška červeno černého stromu

Věta 3

Červeno černý strom s n vnitřními uzly má výšku nejvýše $2\log_2(n+1)$.

- nechť strom má výšku h a černou výšku b
- z předchozích lemmat plyne

$$n \geq 2^b - 1 \geq 2^{h/2} - 1$$

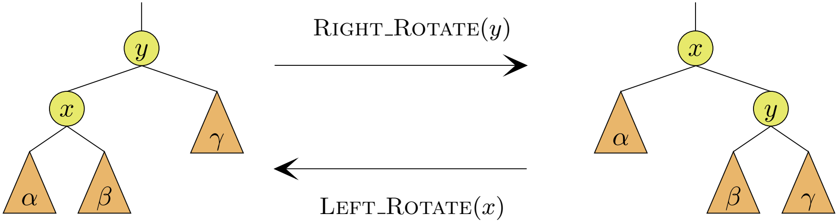
- po úpravě $\log_2(n+1) \geq h/2$, a tedy $h \leq 2\log_2(n+1)$

Červeno černé stromy - operace

- SEARCH, MIN, MAX, SUCCESSOR, PREDECESSOR se implementují stejně jako pro binární vyhledávací stromy
- vyjmenované operace mají složitost $\mathcal{O}(\log n)$

- INSERT a DELETE modifikují strom
- modifikace může porušit vlastnosti červeno černého stromu
- jsou potřebné další kroky, které vlastnosti obnoví
- základní operací, která vede k obnovení požadovaných vlastností, je rotace

Rotace



- rotace zachovává vlastnost binárního vyhledávacího stromu

$$a \in \alpha, b \in \beta, c \in \gamma \Rightarrow a \leq x \leq b \leq y \leq c$$

- časová složitost $\mathcal{O}(1)$

Rotace

Left_Rotate(T, x)

$y \leftarrow x.right$
$x.right \leftarrow y.left$
if $y.left \neq T.nil$
  then $y.left.p \leftarrow x$ **fi**
$y.p \leftarrow x.p$
if $x.p = T.nil$
  then $T.root \leftarrow y$
   else if $x = x.p.left$
        then $x.p.left \leftarrow y$
        else $x.p.right \leftarrow y$ **fi fi**
$y.left \leftarrow x$
$x.p \leftarrow y$

Přidání nového uzlu

- uzel x do stromu přidáme stejným postupem jako do binárního vyhledávací stromu
- jakou barvou máme obarvit nový uzel?
- obě možnosti mají za důsledek porušení některých vlastností červeno černého stromu

- řešení: obarvi uzel x **červenou** barvou
- vlastnosti
  1 *(černý kořen)* jestliže x je kořenem, tak vlastnost neplatí
  3 *(otec červeného uzlu je černý)* nemusí platit
  4 *(stejná černá výška)* zůstává v platnosti

 Červeno černé stromy

Přidání nového uzlu - schéma

RB_Insert(T, a)

```
TREE_INSERT(T, a)
a.color ← red
while a ≠ T.root ∧ a.p.color = red
      do if a.p = a.p.p.left
            then d ← a.p.p.right
                 if d.color = red
                    then  případ 1
                    else if a = a.p.right
                            then  případ 2
                            else  případ 3
                         fi
                 fi
            else  stejně jako THEN se záměnou left a right
         fi
od
T.root.color ← black
```

Přidání nového uzlu - schéma

případ 1

$a.p.color \leftarrow black$
$d.color \leftarrow black$
$a.p.p.color \leftarrow red$
$a \leftarrow a.p.p$

případ 2

$a \leftarrow a.p$
$\textsc{Left\_Rotate}(T, a)$

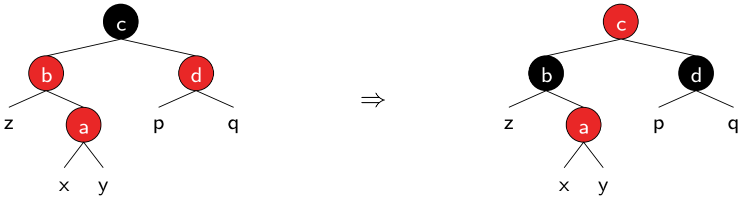
případ 3

$a.p.color \leftarrow black$
$a.p.p.color \leftarrow red$
$\textsc{Right\_Rotate}(T, a.p.p)$

Přidání nového uzlu - korekce - případ 1

- nově přidaný uzel a je červený
- jeho otec b je červený a je levým synem svého otce[3]
- strýc d uzlu a je červený
- praotec c uzlu a je **černý**

- obarvi otce (b) a strýce (d) uzlu a **černou** barvou
- obarvi praotce (c) uzlu a červenou barvou



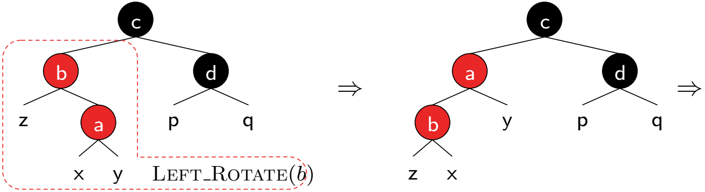
stromy z, x, y, p, q mají černý kořen a všechny mají stejnou černou výšku

[3] situace když b je pravým synem svého otce se řeší symetricky

Přidání nového uzlu - korekce - případ 2

- uzel a je červený a je pravým synem svého otce
- jeho otec b je červený a je levým synem svého otce
- strýc d uzlu a je **černý**
- praotec c uzlu a je **černý**

- proveď levou rotaci kolem otce (b) uzlu a
- pokračuj na případ 3

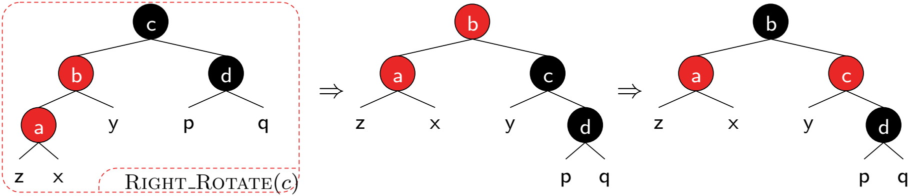


pokračuj na případ 3

Přidání nového uzlu - korekce - případ 3

- uzel a je červený a je levým synem svého otce
- jeho otec b je červený a je levým synem svého otce
- strýc d uzlu a je **černý**
- praotec c uzlu a je **černý**

- proveď pravou rotaci kolem praotce (c) uzlu a
- vyměň obarvení mezi otcem (b) uzlu a a jeho novým bratrem (c)



Složitost přidání nového uzlu

- případ 1: změna obarvení 3 uzlů
- případy 2 a 3: jedna nebo dvě rotace a změna obarvení 2 uzlů

- v případě 1 může změna barvy praotce (c) uzlu a způsobit nový konflikt a to když otec uzlu c má červenou barvou
- v popsaném případě musíme pokračovat další iterací a korigovat barvu uzlu c
- konečnost je garantována faktem, že každou iterací se zmenšuje vzdálenost korigovaného uzlu od kořene stromu
- celková složitost $\mathcal{O}(\log n)$

Odstranění uzlu

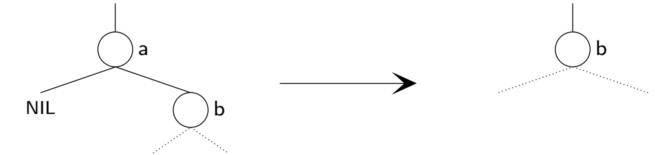
- uzel x ze stromu odstraníme stejným postupem jako z binárního vyhledávací stromu
- v případě, že odstraněný uzel měl červenou barvu, vlastnosti stromu zůstávají zachované
- v případě, že měl černou barvu, může dojít k porušení vlastnosti 4 (stejná černá výška)
- černou barvu z odstraněného uzlu přesouváme směrem ke kořenu tak, aby jsme obnovili platnost vlastnosti 4

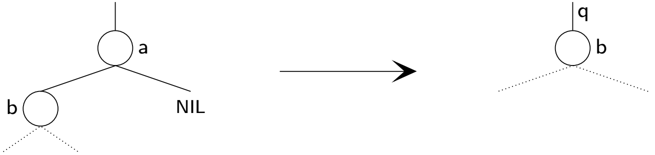
Odstranění uzlu a - případy 1 a 2

a **nemá levého syna**

- odstraň a a nahraď ho jeho pravým synem (b)
- jestliže po přesunu uzel b a jeho otec porušují vlastnost 3 (oba jsou červené), tak uzel b obarvíme černou barvou; tím zachováme černou výšku (a musel být černý)

a **nemá pravého syna** - symetricky



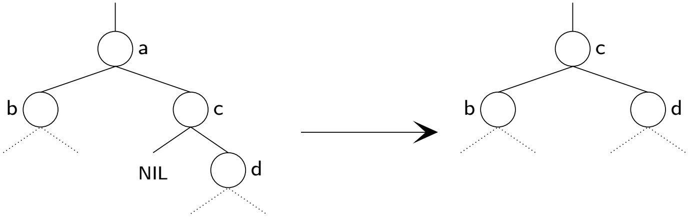


Odstranění uzlu a - případ 3

a **má dva syny, následník** (*successor*) **uzlu** a **je jeho pravým synem**

- odstraň a a nahraď ho jeho následníkem (c)
- levý syn uzlu a se stane levým synem následníka uzlu a

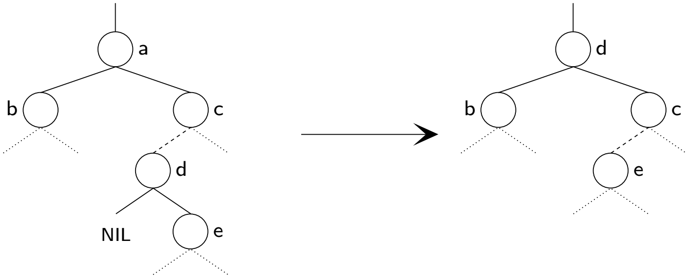
- po přesunu obarvíme následníka (c) barvou uzlu a
- jestliže následník měl původně černou barvu, tak černou barvu dostane jeho syn, tj. syn má dvě barvy (červenou a černou anebo černou a černou)
- problém dvou barev vyřešíme při korekci



Odstranění uzlu a - případ 4

a **má dva syny, následník** (*successor*) **uzlu** a **není jeho synem**

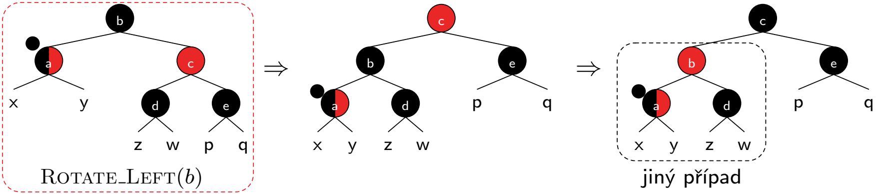
- následníka (d) nahraď jeho pravým synem (e)
- odstraň a a nahraď ho jeho následníkem (d), synové uzlu a se stanou syny následníka (d)
- po přesunu obarvíme následníka (d) barvou uzlu a
- jestliže následník měl původně černou barvu, tak černou barvu dostane jeho syn, tj. syn má dvě barvy (červenou a černou anebo černou a černou)
- problém dvou barev vyřešíme při korekci



Odstranění uzlu - korekce dvou barev - případ 1

uzel a má dvě barvy
bratr (c) uzlu a je červený

- proveď levou rotaci kolem otce (b) uzlu a
- vyměň barvy mezi otcem (b) a praotcem (c) uzlu a
- pokračuj některým z následujících případů

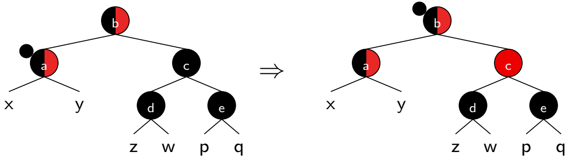


stromy x, y, z, w, p, q mají stejnou černou výšku, nemají žádný uzel s dvěma barvami a neporušují žádnou vlastnost červeno černého stromu

Odstranění uzlu - korekce dvou barev - případ 2

uzel a má dvě barvy
bratr (c) uzlu a stejně jako oba jeho synové (d, e) mají černou barvu

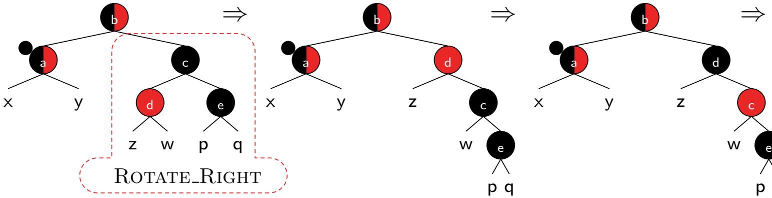
- vezmi jednu černou barvu z uzlu a a přesuň ji do jeho otce (b)
- bratr (c) uzlu a dostane červenou barvu *(aby se zachovala černá výška)*
- uzel se dvěma barvami se přesunul blíž ke kořenu, problém jeho dvou barev řešíme rekurzivně



Odstranění uzlu - korekce dvou barev - případ 3

uzel a má dvě barvy
bratr (c) uzlu a a jeho pravý syn (e) mají černou barvu, levý syn (d) je červený

- proveď pravou rotaci kolem bratra (c) uzlu a
- vyměň barvy mezi původním a novým bratrem uzlu a (d, c)
- pokračuj případem 4

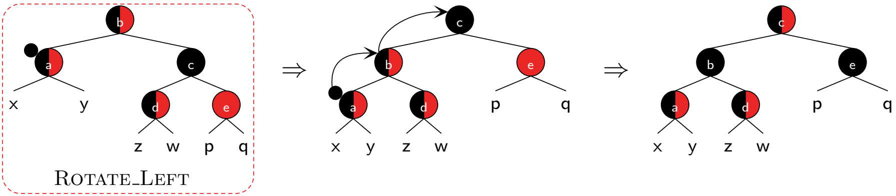


pokračuj případem 4

Odstranění uzlu - korekce dvou barev - případ 4

uzel a má dvě barvy
bratr (c) uzlu a má černou barvu, jeho pravý syn (e) má červenou barvu

- proveď levou rotaci kolem otce (b) uzlu a
- obarvi nového praotce (c) uzlu a barvou jeho otce (b)
- přesuň černou barvu z uzel a na jeho otce (b), otec (b) se stane černým
- uzel (e) se stane černým



Pořadí (rank) prvku

využití červeno černých stromů při určení ranku (pořadí) prvku a vyhledávání prvku s daným rankem

- množina A obsahující n vzájemně různých čísel
- číslo $x \in A$ má rank i právě když v A existuje přesně $i - 1$ čísel menších než x

možné řešení

- jestliže prvky A jsou uložené v poli, tak v čase $\mathcal{O}(n)$ můžeme
  - najít číslo s rankem i
  - určit rank daného čísla

existuje efektivnější řešení?

při použití červeno černých stromů dokážeme oba problémy vyřešit v čase $\mathcal{O}(\log n)$

Rozšíření červeno černých stromů

požadujeme

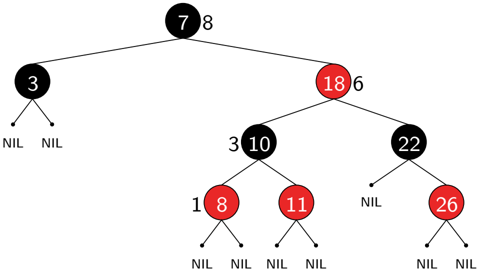
- efektivní implementaci standardních operací nad červeno černým stromem
- efektivní implementaci operace $\mathrm{RB\_S{\scriptsize ELECT}}(x, i)$, která najde i-ty nejmenší klíč v podstromě s kořenem x
- efektivní implementaci operace $\mathrm{RB\_R{\scriptsize ANK}}(T.x)$, která určí rank klíče uloženého v uzlu x

jestliže strom obsahuje uzly se stejnými klíči, tak rankem klíče je pořadí uzlu v INORDER *uspořádání uzlů stromu*

Princip

ke každému uzlu x přidáme atribut $x.size$ - počet (vnitřních) uzlů v podstromě s kořenem x, včetně uzlu x

$$x.size = x.left.size + x.right.size + 1$$



Vyhledání klíče s daným rankem

RB_Select(x, i)

```
r ← x.left.size + 1
if i = r then return x
         else if i < r then return RB_Select(x.left, i)
                       else return RB_Select(x.right, i − r) fi fi
```

korektnost

- z definice atributu $.size$ plyne, že počet uzlů v levém podstromu uzlu x navýšený o 1 (r) je přesně rank klíče uloženého v x v podstromě s kořenem x
- když $i = r$, tak x je hledaný uzel
- když $i < r$, tak i-ty nejmenší klíč se nachází v levém podstromě uzlu x a je i-tým nejmenším klíčem v tomto podstromě
- když $i > r$, tak i-ty nejmenší klíč se nachází v pravém podstromě uzlu x a jeho pořadí v tomto podstromě je i snížené o počet uzlů levého podstromu

Vyhledání klíče s daným rankem

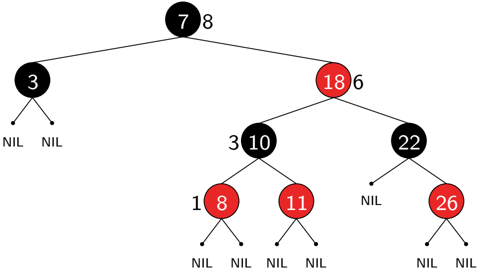
RB_Select(x, i)

$r \leftarrow x.left.size + 1$
if $i = r$ **then return** x
  else if $i < r$ **then return** $\text{RB\_SELECT}(x.left, i)$
    else return $\text{RB\_SELECT}(x.right, i - r)$ **fi fi**

složitost

- každé rekurzivní volání se aplikuje na strom, jehož hloubka je o 1 menší
- hloubka červeno černého stromu je $\mathcal{O}(\log n)$
- složitost RB_SELECT je $\mathcal{O}(\log n)$

Určení ranku daného prvku



rank prvku 11

- všechny uzly v levém podstromě uzlu 11
- sledujeme cestu od 11 do kořene
- jestliže uzel na cestě je levým synem, nemění rank prvku 11
- jestliže uzel na cestě je pravým synem, tak on sám jakož i jeho levý podstrom obsahují klíče menší než 11

Určení ranku daného prvku

RB_Rank(T, x)

$r \leftarrow x.left.size + 1$
$y \leftarrow x$
while $y \neq T.root$
  do if $y = y.p.right$
    then $r \leftarrow r + y.p.left.size + 1$ **fi**
  $y \leftarrow y.p$ **od**
return r

korektnost

invariant na začátku každé iterace **while** cyklu je r rovné ranku klíče $x.key$ v podstromě s kořenem y

inicializace na začátku je r rovné ranku $x.key$ v podstromě s kořenem x a $x = y$

RB_Rank(T, x)

$r \leftarrow x.left.size + 1$
$y \leftarrow x$
while $y \neq T.root$
    do if $y = y.p.right$
        then $r \leftarrow r + y.p.left.size + 1$ **fi**
    $y \leftarrow y.p$ **od**
return r

invariant na začátku každé iterace **while** cyklu je r rovné ranku klíče $x.key$ v podstromě s kořenem y

iterace

- na konci cyklu se vykoná $y \leftarrow y.p$
- po provedení cyklu proto musí platit, že r je rank $x.key$ v podstromě s kořenem $y.p$
- jestliže y je levý syn, tak všechny klíče v podstromě jeho bratra jsou větší než $x.key$ a r se nemění
- jestliže y je pravý syn, tak všechny hodnoty v podstromě jeho bratra jsou menší než $x.key$ a hodnota r se zvýší o velikost tohoto stromu plus 1 (klíč v uzlu $y.p$ je taky menší než $x.key$)

RB_Rank(T, x)

```
r ← x.left.size + 1
y ← x
while y ≠ T.root
      do if y = y.p.right
            then r ← r + y.p.left.size + 1 fi
      y ← y.p od
return r
```

invariant na začátku každé iterace **while** cyklu je r rovné ranku klíče $x.key$ v podstromě s kořenem y

ukončení

výpočet končí když $y = T.root$, z platnosti invariantu plyne korektnost algoritmu

složitost

- po každé iteraci se sníží vzdálenost y od kořene o 1
- hloubka červeno černého stromu je $\mathcal{O}(\log n)$
- složitost RB_Rank je $\mathcal{O}(\log n)$

Přidání nového uzlu

přidání uzlu postupujeme od kořene do listu, kde vytvoříme nový uzel, přitom se změní (o 1) pouze velikost podstromů těch uzlů, kterými procházíme

korekce stromu změna barvy uzlu nemění velikost podstromu

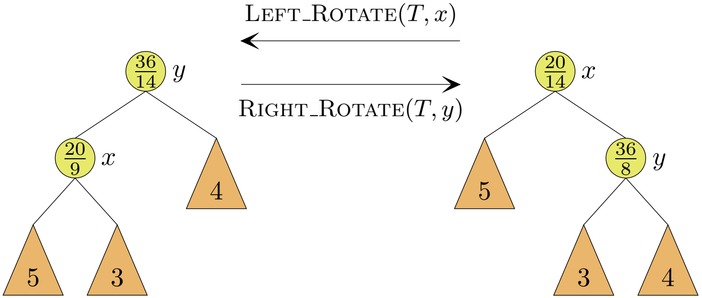
při rotaci se může změnit velikost podstromů

proceduru Left_Rotate doplníme o příkazy

$y.size \leftarrow x.size$

$x.size \leftarrow x.left.size + x.right.size + 1$

symetricky pro pravou rotaci



Odstranění uzlu

první fáze odstraní uzel ze stromu

- na pozici odstraněného uzlu se přesune uzel y
- pro aktualizaci hodnot $size$ procházíme cestu od původní pozice uzlu y do kořene a každému uzlu na této cestě snížíme hodnotu $size$ o 1
- složitost operace se navýší o $\mathcal{O}(\log n)$

korekce obarvení stromu

- ke změně velikosti podstromu může dojít při rotaci, aktualizace hodnot viz přidání nového uzlu
- počet rotací je nejvýše 3, složitost se navýší o $\mathcal{O}(1)$

složitost přidávání i odstraňování uzlu zůstává asymptoticky stejná

Datové struktury

B stromy

B stromy jsou zobecněním binárních vyhledávacích stromů

- B strom je balancovaný, všechny listy mají stejnou hloubku
- vnitřní uzel stromu obsahuje $t-1$ klíčů a má t následníků
- klíče ve vnitřních uzlech stromu zároveň vymezují t intervalů, do kterých patří klíče každého z jeho t podstromů

využití B stromů

- B stromy se typicky používají v databázových systémech a aplikacích, kde objem zpracovávaných dat není možné uchovávat v operační paměti
- počet klíčů uložených v uzlu (a tím i počet následníků) se může pohybovat od jednotek po tisíce; cílem je minimalizovat počet přístupů na disk
- v pseudokódu modelujeme přístupy operacemi $\textsc{Disk_Read}$a $\textsc{Disk\_Write}$
- existují různé varianty, podrobněji viz např. PV062
- Bayer, McCreight 1972

B stromy vs BVS a červeno černé stromy

- zachován princip vyhledávání
- všechny uzly mají stejnou hloubku
- uzly B stromů můžou mít víc následníků
- výška B stromu je $\mathcal{O}(\log n)$, díky většímu počtu následníků může být ale výrazně menší
- operace minimalizují průchod stromem

Stupeň B stromu

minimální stupeň stromu

číslo t, které definuje dolní a horní hranici na počet klíčů uložených v uzlu

- každý uzel (s výjimkou kořene) musí obsahovat alespoň $t-1$ klíčů
- jestliže strom je neprázdný, tak kořen musí obsahovat alespoň jeden klíč
- každý vnitřní uzel (s výjimkou kořene) musí mít alespoň t následníků

- každý uzel může obsahovat nejvýše $2t-1$ klíčů
- každý vnitřní uzel může mít nejvýše $2t$ následníků

- uzel, který má přesně $2t$ následníků, se nazývá *plný*

- nejjednodušší B strom má minimální stupeň 2
- každý jeho vnitřní uzel má 2, 3 anebo 4 následníky
- obvykle se označuje jako 2-3-4 strom

Výška B stromu

B strom s $n \geq 1$ klíči a minimálním stupněm $t \geq 2$ má hloubku nejvýše

$$h \leq \log_t \frac{n+1}{2}$$

- kořen obsahuje alespoň jeden klíč, každý vnitřní uzel alespoň $t-1$ klíčů
- strom má 1 uzel hloubky 0 (kořen), alespoň 2 uzly hloubky 1, alespoň $2t$ uzlů hloubky 2, alespoň $2t^2$ uzlů hloubky 3, obecně alespoň $2t^{h-1}$ uzlů hloubky h

$$\begin{aligned} n &\geq 1 + (t-1)\sum_{i=1}^{h} 2t^{i-1} = 1 + 2(t-1)\sum_{i=0}^{h-1} t^i \\ &= 1 + 2(t-1)\Big(\frac{t^h - 1}{t-1}\Big) = 2t^h - 1 \end{aligned}$$

- z toho $t^h \leq \frac{n+1}{2}$ a tedy $\log_t t^h \leq \log_t \frac{n+1}{2}$ $\square$

Definice B stromu

- každý **uzel** x má atributy
  - $x.n$ - počet klíčů uložených v uzlu x
  - klíče $x.key_1, x.key_2, \cdots, x.key_{x.n}$, které jsou uloženy v neklesajícím pořadí
  - $x.leaf$ - booleovská proměnná nabývající hodnotu je *true* právě když uzel x je listem stromu
- každý **vnitřní uzel** x obsahuje navíc $x.n+1$ ukazatelů $x.c_1, x.c_2, \ldots, x._{c_{x.n+1}}$
- klíče $x.key_i$ definují intervaly, z kterých jsou klíče uložené v každém z podstromů; jestliže k_i je klíč uložený v podstromě s kořenem $x.c_i$, tak platí

$$k_1 \leq x.key_1 \leq k_2 \leq x.key_2 \leq \ldots \leq x.key_{x.n} \leq k_{x.n+1}$$

- všechny **listy mají stejnou hloubku**

Operace nad B stromem

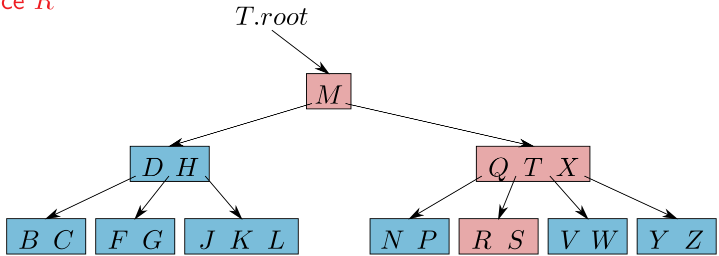
- vytvoření stromu; vyhledávání, přidání a odstranění klíče
- typické aplikace, které využívají B stromy, pracují s daty uloženými na externím disku
- před každou operací, která přistupuje k objektu x, se nejdříve musí vykonat operace $\text{DISK\_READ}(x)$, která zkopíruje objekt do operační paměti (za předpokladu, že tam není)
- symetricky operace $\text{DISK\_WRITE}(x)$ se použije pro uložení všech změn vykonaných nad objektem x
- předpokládáme, že kořen B stromu je vždy uložený v operační paměti a proto nad kořenem vykonáváme pouze operaci DISK_WRITE
- asymptotická složitost všech operací je úměrná hloubce stromu, tj. $\mathcal{O}(\log n)$, kde n je počet klíčů uložených v stromu

- z důvodu optimalizace počtu přístupů na externí disk jsou všechny operace navrženy tak, aby se uzel stromu navštívil nejvýše jednou, tj. všechny operace postupují směrem od kořene dolů a nikdy se nevracejí do již navštíveného uzlu

Vyhledávání

- analogicky jako v binárním vyhledávacím stromě, vybíráme jednoho z následníků uzlu
- argumentem operace je ukazatel $T.root$ na kořen stromu a hledaný klíč k
- jestliže klíč k je v B stromě, operace vrátí dvojici (y, i), kde y je uzel a i index takový, že $y.key_i = k$
- v opačném případě vrátí hodnotu Nil

vyhledání klíče R



Vyhledávání

B-Tree_Search(x, k)

```
1 i ← 1
2 while i ≤ x.n ∧ x.key_i < k do
3         i ← i + 1 od
4 if i ≤ x.n ∧ x.key_i = k
5   then return (x, i) fi
6 if x.leaf then return Nil
7              else DISK_READ(x.c_i)
8                   return B-TREE_SEARCH(x.c_i, k) fi
```

- počet DISK_READ operací je ohraničený hloubkou stromu h
- počet opakování cyklu 2 - 3 je nejvýše $2t$ (t je minimální stupeň B stromu)
- celková složitost je $\mathcal{O}(th) = \mathcal{O}(t \log_t n)$

Vytvoření prázdného stromu

B-Tree_Create(T)

$x \leftarrow$ ALLOCATE_NODE()
$x.leaf \leftarrow true$
$x.n \leftarrow 0$
DISK_WRITE(x)
$T.root \leftarrow x$

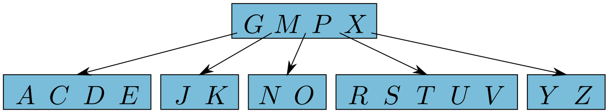
celková složitost $\mathcal{O}(1)$

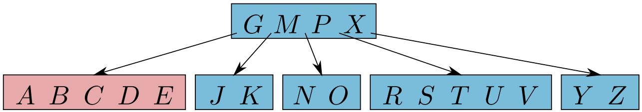
Přidání klíče

- podobně jako u BVS hledáme list, do kterého uložíme nový klíč
- nemůžeme vytvořit nový list (jako v BVS), protože by jsme porušili vlastnost minimálního počtu klíčů v uzlu
- klíč vložíme do existujícího listu
- když vložením klíče dojde k porušení vlastnosti maximálního počtu klíčů, tak list rozdělíme na dva nové listy
- rozdělením se zvýší počet následníků předchůdce původního listu
- pokud se tím poruší vlastnost maximálního počtu následníků, tak musíme (rekurzivně) rozdělit i předchůdce
- proces rozdělování uzlů se v nejhorším případě zastaví až v kořeni stromu

Přidání klíče B do listu, který není plný

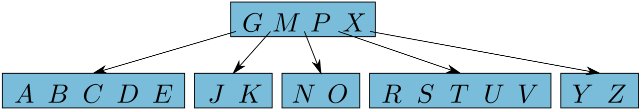
minimálny stupeň stromu je 3

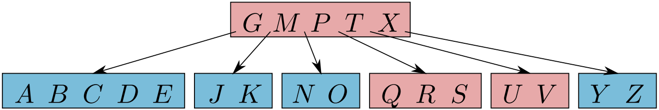




Přidání klíče Q do plného listu

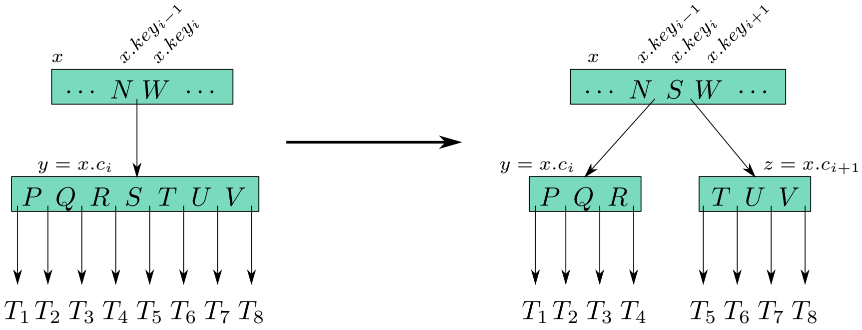
minimálny stupeň stromu je 3





Rozdělení uzlu - schéma

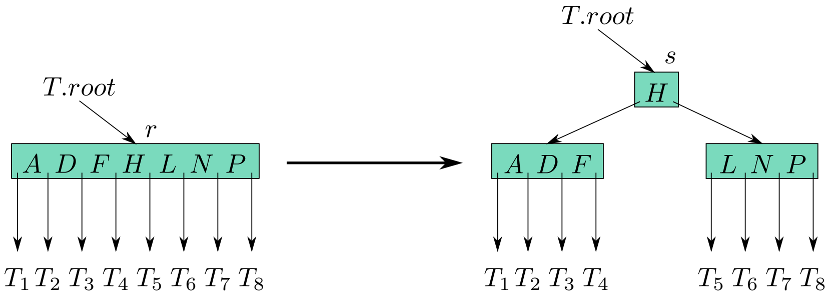
minimálny stupeň stromu je 4



argumentem operace B-TREE_SPLIT je

- vnitřní uzel x, který není plný
- index i takový, že $x.c_i$ je plný následník uzlu x

Rozdělení kořene - schéma



- když potřebujeme rozdělit kořen stromu, tak nejdříve vytvoříme nový, prázdný uzel, který se stane novým kořenem stromu
- rozdělení kořene způsobí navýšení hloubky stromu o 1

Rozdělení uzlu - implementace

B-Tree_Split(x, i)

$z \leftarrow$ ALLOCATE_NODE()
$y \leftarrow x.c_i$
$z.leaf \leftarrow y.leaf$
$z.n \leftarrow t - 1$
for $j = 1$ **to** $t - 1$ **do** $z.key_j \leftarrow y.key_{j+t}$ **od**
if $\neg y.leaf$ **then for** $j = 1$ **to** t **do** $z.c_j \leftarrow y.c_{j+t}$ **od fi**
$y.n \leftarrow t - 1$
for $j = x.n + 1$ **downto** $i + 1$ **do** $x.c_{j+1} \leftarrow x.c_j$ **od**
$x.c_{i+1} \leftarrow z$
for $j = x.n$ **downto** i **do** $x.key_{j+1} \leftarrow x.key_i$ **od**
$x.key_i \leftarrow y.key_t$
$x.n \leftarrow x.n + 1$
DISK_WRITE(y)
DISK_WRITE(z)
DISK_WRITE(x)

Rozdělení uzlu - složitost

- rozdělujeme uzel y (řádek 2)
- když y není list, tak má před rozdělením $2t$ následníků a po rozdělení počet jeho následníků klesne na t
- z je nový uzel (řádek 1) a jeho následníky tvoří t největších následníků uzlu y
- celková složitost je $\mathcal{O}(t)$
- počet operací $\textsc{Disk\_Write}$ a $\textsc{Disk\_Read}$ je $\mathcal{O}(1)$

Přidání klíče - optimalizace

základní varianta

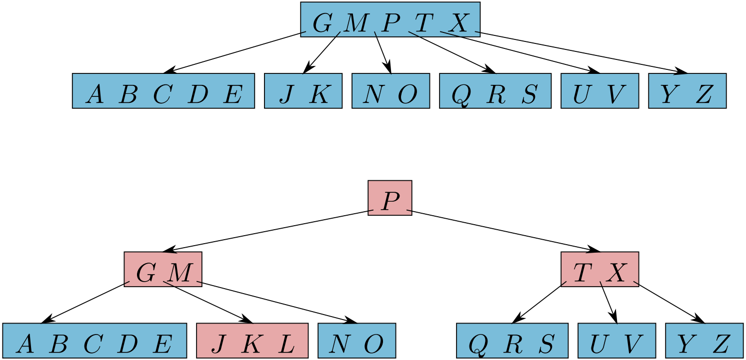
- rozdělení uzlu způsobí navýšení počtu následníků předchůdce rozdělovaného uzlu
- pokud se tím poruší vlastnost maximálního počtu následníků, tak musíme (rekurzivně) rozdělit i předchůdce
- proces rozdělování uzlů se v nejhorším případě zastaví až v kořeni stromu

optimalizace

- cílem je realizovat celou operaci přidání klíče při jednom průchodu stromu od kořene k listu *(optimalizace počtu přístupů na disk!!!)*
- rozdělování může nastat pouze u těch uzlů, které jsou plné, tj. obsahují maximální povolený počet klíčů $(2t - 1)$
- vždy, když procházíme přes plný uzel, rozdělíme ho na dva nové uzly a to tak, že každý ze dvou nových uzlů dostane $t - 1$ klíčů a jeden klíč se přesune do jejich otce
- korektnost postupu je garantována, protože předchůdce rozdělovaného uzlu není plný

Přidání klíče L - procházíme přes plný uzel

minimálny stupeň stromu je 3



Přidání klíče - implementace

B-Tree_Insert(T, k)

1 $r \leftarrow T.root$
2 **if** $r.n = 2t - 1$
3 **then** $s \leftarrow$ ALLOCAT_NODE()
4 $T.root \leftarrow s$
5 $s.leaf \leftarrow false$
6 $s.n \leftarrow 0$
7 $s.c_1 \leftarrow r$
8 B-TREE_SPLIT$(s, 1)$
9 B-TREE_INSERT_NONFULL(s, k)
10 **else** B-TREE_INSERT_NONFULL(r, k)
11 **fi**

- řádky 3 - 9 řeší plný kořen stromu
- na konci se volá procedura B-TREE_INSERT_NONFULL, která vloží klíč do stromu, jehož kořen není plný

Přidání klíče - implementace

B-Tree_Insert_Nonfull(x, k)

```
i ← x.n
if x.leaf
   then while i ≥ 1 ∧ x.key_i > k
                 do x.key_{i+1} ← x.key_i
                 i ← i − 1 od
        x.key_{i+1} ← k
        x.n ← x.n + 1
        Disk_Write(x)
    else while i ≥ 1 ∧ x.key_i > k do i ← i − 1 od
        i ← i + 1
        Disk_Read(x.c_i)
        if x.c_i.n = 2t − 1 then B-Tree_Split(x, i)
                                 if x.key_i < k then i ← i + 1 fi fi
        B-Tree_Insert_Nonfull(x.c_i, k)
fi
```

Přidání klíče - složitost

- počet operací Disk_Write a Disk_Read je $\mathcal{O}(h)$
  (*vždy jenom jedna mezi dvěma voláními* B-Tree_Insert_Nonfull)
- celková složitost je $\mathcal{O}(th) = \mathcal{O}(t \log_t n)$
- procedura B-Tree_Insert_Nonfull je tail - rekurzivní, a proto je počet uzlů, které musí být uloženy v operační paměti, konstantní

Odstranění klíče

odstranění probíhá analogicky jako u binárního vyhledávacího stromu

- jestliže se klíč určený k odstranění nachází v listu, odstraníme ho
- jestliže se klíč určený k odstranění nachází v uzlu, který není listem, nahradíme ho jeho následníkem (resp. předchůdcem) a následníka odstraníme z listu ve kterém se původně nacházel
- samotné mazání klíče se **vždy** realizuje v listu
- operace má stejnou asymptotickou složitost jako u BVS
- samotná implementace má ale několik speciálních případů, protože klíč může být odstraněn z libovolného uzlu
- v optimalizované variantě klíč odstraníme při jednom průchodu stromem od kořene dolů, s možnou výjimkou návratu do uzlu, ve kterém byl původně uložen odstraňovaný klíč

Odstranění klíče - základní varianta

odstranění klíče k z listu x

1. list x je současně kořenem stromu
   - klíč k odstraníme
2. list x není kořenem a obsahuje alespoň t klíčů
   - klíč k odstraníme
3. list x není kořenem a obsahuje přesně $t-1$ klíčů
   - vezmi toho bratra y listu x, který má více klíčů
   - vytvoř seznam obsahující klíče z listů x a y a navíc ten klíč z otce p listu x, který tvoří hranici mezi x a y
   - délka seznamu je $t-2$ ($=$ počet klíčů v x) $+$ 1 ($=$ klíč z otce) $+$ počet klíčů v $y \geq t-2+1+t-1$
   - rozlišujeme dva případy podle délky seznamu

Odstranění klíče - základní varianta - délka seznamu

3 A seznam obsahuje alespoň $2t - 1$ klíčů

- seznam rozdělíme na 3 části: *Left*, *Middle* a *Right*, kde *Middle* je medián seznamu, *Left* obsahuje klíče menší než medián a *Right* klíče větší než medián
- klíč *Middle* vrátíme do otce p, ze kterého jsme předtím odebrali hraniční klíč
- klíče *Left* a *Right* vložíme do dvou uzlů x a y
- uzly x a y mají alespoň $t - 1$ klíčů, počet klíčů v uzlu p zůstal nezměněný, hotovo

3 B seznam obsahuje právě $2t - 2$ klíčů

- uzly x a y nahradíme jediným uzlem obsahujícím všechny klíče seznamu
- nový list má povolený počet klíčů
- otec p má počet klíčů o 1 nižší než původně
- v případě, že počet klíčů v uzlu p klesl pod minimální hranici $t - 1$, opakujeme (rekurzivně) postup pro uzel p

Odstranění klíče - základní varianta

- po odstranění klíče z listu může klesnout počet klíčů v jeho uzlu pod minimální hranici
- musíme realizovat operace, které obnoví platnost podmínky minimálního počtu klíčů v uzlu
- může nastat situace, když se prochází strom od kořene k listu a potom zpátky od listu ke kořeni (*např. když všechny uzly na cestě od kořene do listu obsahujícího klíč mají stupeň přesně* t)
- podobně jako při vkládaní klíče optimalizujeme proces odstranění klíče tak, aby sme minimalizovali počet přístupů na disk

Odstranění klíče - optimalizace

- postupujeme od kořene směrem k listu
- vždy, když procházíme přes uzel, který má přesně $t - 1$ klíčů, tak uděláme takovou korekci, která zvýší počet klíčů v uzlu na t
- když narazíme na uzel, ze kterého potřebujeme odstranit klíč, máme garanci, že jeho otec má alespoň t klíčů
- když odstranění klíče z uzlu způsobí snížení počtu klíčů v jeho otci, nevznikne žádný problém

Optimální odstranění klíče - pravidla

odstraňujeme klíč k

1. když klíč k je v listu x, odstraň k z x

2. když klíč k je ve vnitřním uzlu x, tak
   a. jestliže syn y, který je před k v x, obsahuje alespoň t klíčů, tak najdi v podstromě s kořenem y předchůdce k' klíče k; nahraď v x klíč k klíčem k'; rekurzivně odstraň klíč k'

   b. jestliže syn y má méně než t klíčů tak, symetricky, prozkoumej syna z, který následuje za k v x; v případě že z obsahuje alespoň t klíčů, tak najdi v podstromě s kořenem z následníka k' klíče k; nahraď v x klíč k klíčem k'; rekurzivně odstraň klíč k'

   c. v případě, že synové y i z mají jen $t-1$ klíčů, tak do vrcholu y přesuň klíč k a všechny klíče z vrcholu z; z vrcholu x odstraň k a ukazatel na z; nový uzel y obsahuje $2t-1$ klíčů (mezi nimi i klíč k); rekurzivně odstraň k z y

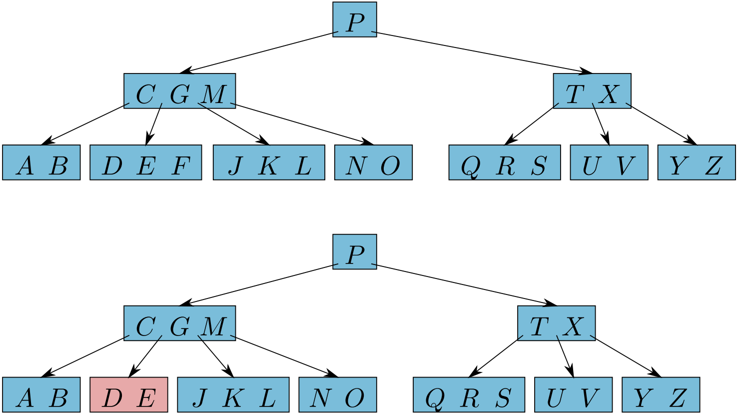
Odstranění klíče - pravidla, pokračování

3 když klíč k není ve vnitřním uzlu x , tak urči kořen $x.c_i$ stromu, který musí obsahovat k (za předpokladu, že k je v stromě); v případě, že uzel $x.c_i$ obsahuje jen $t-1$ klíčů, pokračuj body 3.a. anebo 3.b. které zaručí, že rekurzivní volání se aplikuje na uzel obsahující alespoň t klíčů; rekurzivně odstraň klíč k z vhodného následníka uzlu x

- **a.** v případě, že $x.c_i$ obsahuje jen $t-1$ klíčů, ale některý z jeho přímých bratrů obsahuje alespoň t klíčů, tak zvyš počet klíčů v $x.c_i$ a to tak, že přesuneš klíč z x do $x.c_i$, přesuneš klíč z bratra x a přesuneš příslušný ukazatel na následníka z bratra do uzlu $x.c_i$
- **b.** v případě, že $x.c_i$ i jeho jeho přímí bratři obsahují jen $t-1$ klíčů, tak přesuň do $x.c_i$ jeden klíč z x a všechny klíče z jednoho z bratrů

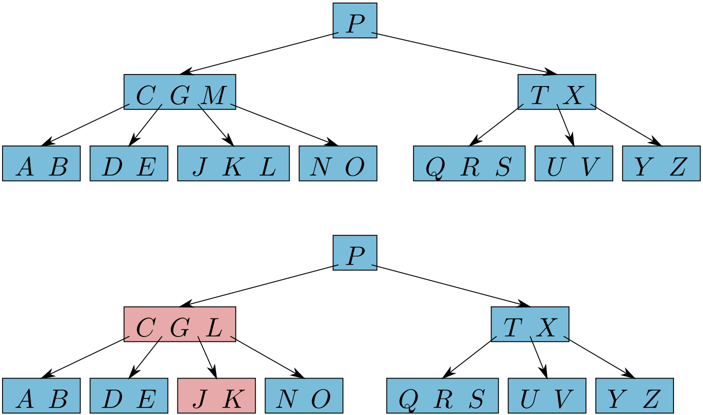
Odstranění klíče F – případ 1

$t = 3$



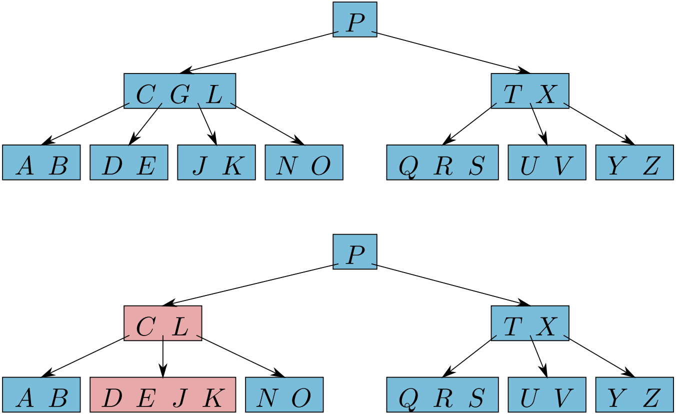
Odstranění klíče M – případ 2a

$t = 3$



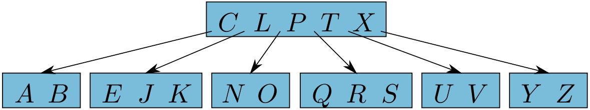
Odstranění klíče G – případ 2c

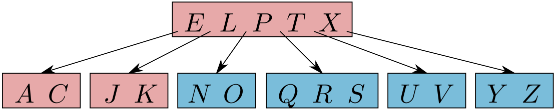
$t = 3$



Odstranění klíče B – případ 3a

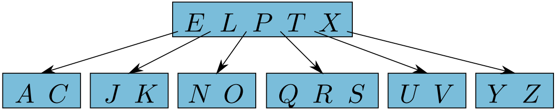
$t = 3$

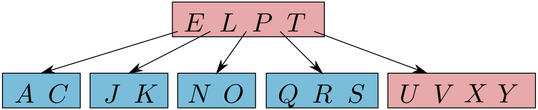




Odstranění klíče Z – případ 3b

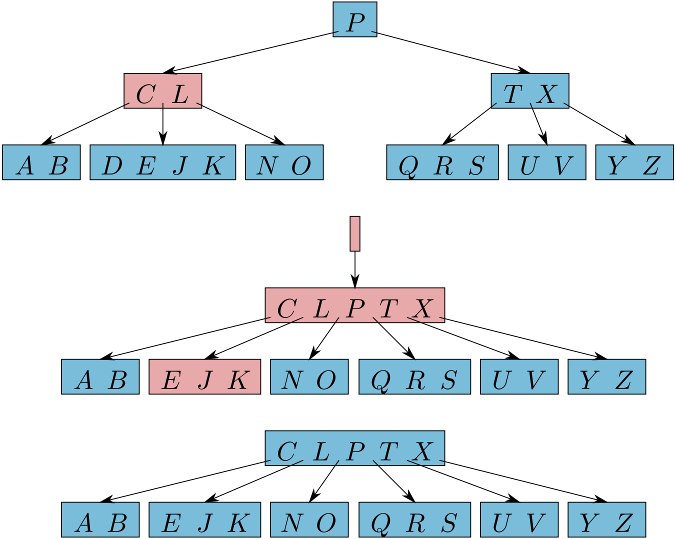
$t = 3$





Odstranění klíče D – případ 3b

$t = 3$



Odstranění klíče - složitost

- v případě, že se odstraňovaný klíč nachází v listu, procedura prochází od kořene k listu bez nutnosti návratu
- v případě, že se klíč nachází ve vnitřním uzlu, tak procedura postupuje od kořene k listu s možným návratem do vrcholu, ze kterého byl klíč odstraněn a nahrazen svým předchůdcem anebo následníkem (případy 2.a., 2.b.)
- mezi dvěma rekurzivními voláním se vykoná nanejvýš jedna operace $\textsc{Disk\_Write}$ a jedna operace $\textsc{Disk\_Read}$; jejich celkový počet je proto $\mathcal{O}(h)$
- celková složitost je $\mathcal{O}(th) = \mathcal{O}(t \log_t n)$

B+ stromy

- klíče jsou uloženy pouze v listech
- zřetězení listů zachovává pořadí klíčů
- vnitřní uzly B+ stromů indexují listy

výhody a nevýhody

- klíč v B stromě se najde před dosažením listu
- vnitřní uzly B stromů jsou větší, do uzlů se proto může uložit méně klíčů a strom je hlubší
- operace vkládaní a odstraňování klíče z B stromu jsou komplikovanější
- implementace B stromu je náročnější než implementace B+stromu

Datové struktury

Slovník

- dynamický datový typ pro reprezentaci množiny objektů
- podporované operace
  $\textsc{Insert}(S, x)$ do množiny S přidá objekt x
  $\textsc{Search}(S, x)$ zjistí, zda množina S obsahuje objekt x
  $\textsc{Delete}(S, x)$ z množiny S odstraní objekt x

vhodné datové struktury pro implementaci slovníku

seznam všechny operace mají složitost $\mathcal{O}(n)$ (n je mohutnost množiny S)

vyhledávací strom se dá použít za předpokladu, že objekty mají číselný klíč, při použití vyváženého stromu je složitost operací $\mathcal{O}(\log n)$

cíl: složitost všech operací

v nejhorším případě $\Theta(n)$

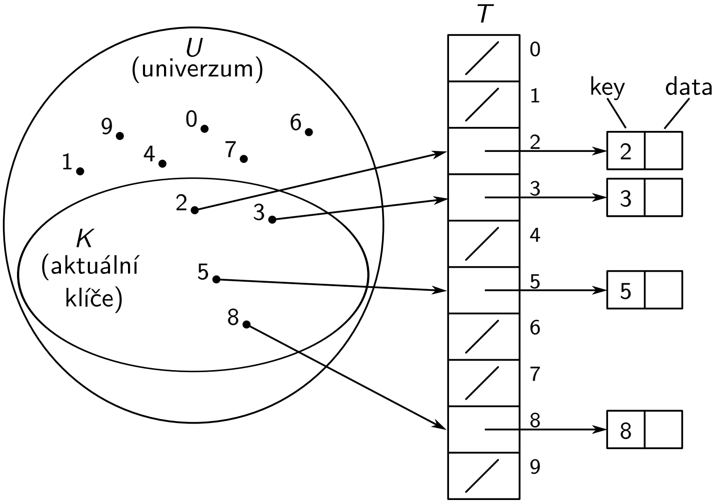
v očekávaném případě $\mathcal{O}(1)$

Přímé adresování

- každý prvek reprezentované množiny prvků má přiřazen klíč vybraný z univerza $U = \{0, 1, \ldots, m-1\}$
- **žádné dva prvky nemají přiřazený stejný klíč**

- pole $T[0 \ldots m-1]$
  - každý slot (pozice) v T odpovídá jednomu klíči z U
  - když reprezentovaná množina obsahuje prvek x s klíčem k, tak $T[k]$ obsahuje ukazatel na x
  - v opačném případě je $T[k]$ prázdné ($\textsc{Nil}$)
- složitost operací je konstantní

Přímé adresování - schéma



Výhody a nevýhody přímého adresování

výhody

- konstantní složitost všech operací
- jednoduchá implementace

nevýhody

- v případě, že univerzum U je veliké, tak uchovávání tabulky velikosti univerza je neefektivní resp. nemožné
- v případě, že množina aktuálně uložených klíčů je malá ve srovnání s velikostí univerza, tak větší část paměti alokované pro tabulku T je nevyužitá
- problém objektů se stejným klíčem

Hašovací tabulka

- v případě, že množina aktuálně uložených klíčů K je výrazně menší než U, využívá hašovací tabulka výrazně méně paměti, než tabulka s přímým přístupem
- potřebný prostor se dá redukovat až na $\Theta(|K|)$
- složitost operací zůstává konstantní avšak v *očekávaném* (a ne v nejhorším) případě

přímé adresování	prvek x s klíčem k uloží v tabulce na pozici $T[k]$
hašování	prvek x s klíčem k uloží v tabulce na pozici $T[h(k)]$

- h je funkce $h : U \longrightarrow \{0, 1, \ldots, m-1\}$
- h se nazývá hašovací funkce

Hašovací tabulka - problémy k řešení

1. řešení kolizí

kolize $\approx$ dva anebo více klíčů zahašujeme na stejnou pozici
pro $x \neq y$ *je* $h(x) = h(y)$, x *a* y *mají stejný otisk*

- zřetězené hašování *(chaining)*
- otevřená adresace *(open adressing)*

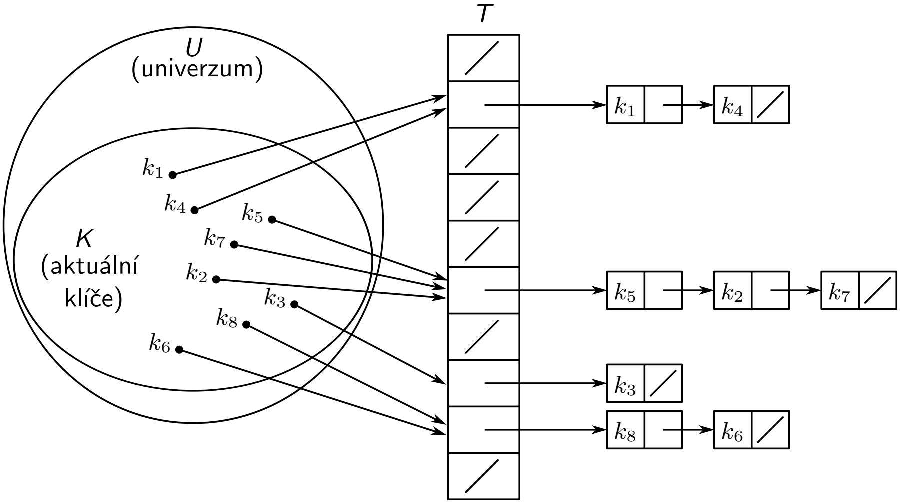
2. výběr hašovací funkce

- minimalizovat počet kolizí
- efektivní výpočet funkce

Zřetězené hašování

- každá položka tabulky obsahuje (ukazatel na) seznam prvků zahašovaných na stejnou pozici
- seznam je prázdný právě když žádný prvek nebyl zahašovaný na danou pozici
- vkládání prvku x do hašovací tabulky T se realizuje jako přidání prvku na začátek seznamu $T[h(x.key)]$
- prvek x vyhledáváme v seznamu $T[h(x.key)]$
- prvek x odstraníme vymazáním ze seznamu $T[h(x.key)]$

Zřetězené hašování - schéma



Zřetězené hašování - složitost

složitost v nejhorším případě

Insert konstantní (*za předpokladu, že vkládaný prvek není v tabulce*)

Search úměrná délce seznamu; v nejhorším případě $\Theta(n)$, kde n je počet prvků uložených v tabulce

Delete (asymptoticky) stejná jako složitost $\textsc{Search}$ (*za předpokladu dvousměrného seznamu*)

složitost v průměrném případě

záleží od výběru hašovací funkce

Zřetězené hašování - průměrná složitost

- předpokládáme, že hašovací funkce je **jednoduchá uniformní** *(simple uniform)*, tj. že pro každý prvek univerza je pravděpodobnost jeho zahašování na kterýkoliv index tabulky stejná (a nezávislá od toho, kam jsou zahašovány zbylé prvky univerza)

- složitost operací se vyjadřuje vzhledem k **faktoru naplnění** *(load factor)*
- pro danou tabulku s m pozicemi, ve které je uložených n prvků, definujeme faktor naplnění α předpisem $\alpha = n/m$, tj. průměrný počet prvků zahašovaných na stejnou pozici
- pro $j = 0, 1, \ldots, m-1$ nechť n_j označuje délku seznamu $T[j]$
- pro jednoduchou uniformní hašovací funkci platí, že **očekávaná délka seznamu $T[j]$ je**
$$E[n_j] = \alpha = n/m$$
- předpokládáme, že výpočet hodnoty funkce má konstantní časovou složitost

Zřetězené hašování - průměrná složitost

V hašovací tabulce, ve které jsou kolize řešeny zřetězením a ve které se používá jednoduchá uniformní funkce, má operace neúspěšného vyhledávaní prvku průměrnou časovou složitost $\Theta(1+\alpha)$.

V hašovací tabulce, ve které jsou kolize řešeny zřetězením a ve které se používá jednoduchá uniformní funkce, má operace úspěšného vyhledávaní prvku průměrnou časovou složitost $\Theta(1+\alpha)$.

- v případě, že počet pozic v tabulce je proporcionální počtu prvků v tabulce, $n = \mathcal{O}(m)$, platí $\alpha = n/m = \mathcal{O}(m)/m = \mathcal{O}(1)$
- vyhledávaní prvku má konstantní průměrnou složitost
- samotné vložení prvku a odstranění prvku ze seznamu má konstantní složitost
- **všechny operace mají za daných předpokladů konstantní průměrnou složitost**

Výběr hašovací funkce

jak vybrat dobrou hašovací funkci?

- funkce by měla mít vlastnosti jednoduché uniformní funkce: každý klíč je zahašován na všechny pozice se stejnou pravděpodobností
- v praxi je těžké ověřit podmínku uniformity, protože nepoznáme rozložení klíčů (a navíc jsou často na sobě závislé)
- v praxi využíváme při volbě hašovací funkce znalosti rozložení klíčů s cílem, aby se často společně se vyskytující klíče zahašovali na různé pozice
- **příklad:** když klíče jsou vybírány náhodně s uniformním rozdělením z intervalu $\langle 0, 1)$, tak hašovací funkce $h(k) = \lfloor k \cdot m \rfloor$ je jednoduchou uniformní funkcí

Klíče jako přirozené čísla

- většina hašovacích funkcí je navržená pro univerzum - množinu přirozených čísel $\mathbb{N}$
- když klíče nejsou přirozená čísla, můžeme je interpretovat jako přirozená čísla použitím vhodného kódování

příklad

- znakový řetězec interpretujeme jako číslo (ve vhodně zvolené číselné soustavě)
- řetězec CLRS
- ASCII hodnoty: C = 67, L = 76, R = 82, S = 83
- máme 128 ASCI hodnot, volíme proto číselnou soustavu se základem 128
- CLRS interpretujeme jako $(67 \cdot 128^3) + (76 \cdot 128^2) + (82 \cdot 128^1) + (83 \cdot 128^0)$

Hašovací funkce - metoda dělení

$$h(k) = k \mod m$$

příklad $m = 20$, $k = 91 \Longrightarrow h(k) = 11$

výhody rychlost

nevýhody špatné chování pro některé m

- pro $m = 2^p$ je hodnota $h(k)$ vždy p nejpravějších bitů z k
- když k je znakový řetězec interpretovaný při základě 2^p, tak hodnota $m = 2^p - 1$ není vhodná, protože po permutaci řetězce se hodnota hašovací funkce nezmění
- dobrou volbou pro m je prvočíslo

Hašovací funkce - metoda binárního násobení

- předpoklad: univerzum je U množina binárních čísel délky w
- předpoklad: velikost m tabulky je mocninou dvojky, $m = 2^p$
- cílem je zahašovat w-bitové čísla na p-bitové čísla

- zvolíme libovolnou konstantu A, $0 < A < 1$

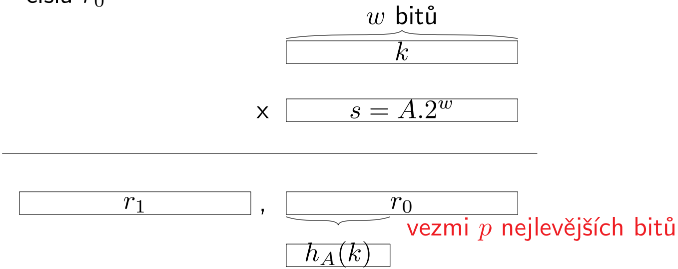
$$h_A(k) = \lfloor m\ (k\ A \mod 1) \rfloor$$

postup výpočtu

1. vynásob klíč k konstantou A a ze součinu vezmi desetinnou část
2. výsledek vynásob číslem m a ze součinu vezmi celou část

$$h_A(k) = \lfloor m \; (k \; A \quad \mathrm{mod}\; 1) \rfloor$$

- zvolíme A tvaru $s/2^w$
- vynásobíme čísla k a s
- výsledkem násobení je $2w$ bitové číslo, kde r_1 je celočíselná část součinu kA a r_0 je desetinná část součinu *(viz obrázek)*
- pro další výpočet potřebujeme pouze r_0
- potřebujeme celou část součinu čísel r_0 a m
- vzhledem k tomu, že $m = 2^p$, násobení znamená posun o p bitů doleva
- ve skutečnosti nemusíme vůbec násobit a stačí vzít p nejvýznamnějších bitů čísla r_0



Metoda binárního násobení - příklad

- $w = 5, m = 8, w = 3$, tj. hašujeme 5 bitové čísla, velikost tabulky je $8 = 2^3$ a chceme hašovat na 3 bitové čísla
- hašujeme klíč $k = 21$
- vybíráme konstantu A tvaru $s/2^w$ a takovou, aby $0 < A < 1$ – proto musí platit $0 < s < 2^5$, vybereme $s = 13 \Longrightarrow A = 13/32$

výpočet $h_A(k)$ podle vzorce $h_A(k) = \lfloor m \ (k \ A \mod 1) \rfloor$

- $kA = 21 \cdot 13/32 = 8\frac{17}{32}$
- $kA \mod 1 = 17/32$
- $m(kA \mod 1) = 8 \cdot 17/32 = 4\frac{1}{4}$
- $\lfloor m(kA \mod 1) \rfloor = 4 = h_A(k)$

implementace

- $ks = 21 \cdot 13 = 273 = 8 \cdot 2^5 + 17$
- $r_1 = 8, r_0 = 17$; bitový zápis r_0 je 10001
- vezmeme $p = 3$ nejvýznamnější bity r_0, tj. 100 (4 v desítkové soustavě)
- $h_A(k) = 4$

Univerzální hašování

scénář ani nejlepší hašovací funkce negarantuje dobré chování hašování v případě, že klíče určené k zahašování jsou vybrány tím nejhorším možným způsobem *(můžeme si představit útočníka, který pozná náš hašovací program a hašovací funkci a na základě toho dokáže vybrat takové klíče, které se zahašují na stejnou pozici, viz analogii s výběrem pivota pro Quicksort)*

řešení při každém použití hašovacího programu vybereme náhodně jinou hašovací funkci
(když útočník neví, jaká hašovací funkce bude vybrána, nemůže záměrně vybírat vstupy, které povedou k špatnému chování)

výběr funkce samotný fakt náhodného výběru funkce ještě negarantuje efektivitu hašování; je potřebné vybírat z vhodných kandidátů

Univerzální hašování

Definice 4

Nechť $\mathcal{H}$ je konečná množina hašovacích funkcí, které mapují univerzum klíčů U na m pozic. $\mathcal{H}$ je **univerzální množinou hašovacích funkcí** *právě když pro každou dvojici klíčů $k, l \in U$, $k \neq l$, je počet hašovacích funkcí $h \in \mathcal{H}$, pro které $h(k) = h(l)$, nejvýše $|\mathcal{H}|/m$.*

Věta 5

Předpokládejme, že hašovací funkce, náhodně vybraná z univerzální množiny hašovacích funkcí, je použita pro zahašování n klíčů do tabulky s m pozicemi. Pak pro klíč k platí, že když

- *k není v tabulce, tak očekávaná délka seznamu, do kterého se zahašuje k, je nejvýše $\alpha = n/m$*
- *k je v tabulce, tak očekávaná délka seznamu, který obsahuje k, je nejvýše $\leq 1 + \alpha$.*

Univerzální hašování - složitost

Důsledek 6

Libovolná posloupnost n *operací* INSERT, SEARCH *a* DELETE, *z nichž nejvýše* $\mathcal{O}(m)$ *operací je typu* INSERT, *má očekávanou časovou složitost* $\Theta(n)$ *za předpokladu použití zřetězeného hašování, univerzální množiny hašovacích funkcí a tabulky s* m *pozicemi.*

Důsledek 7

Použitím univerzálního hašování a řešení kolizí řetězením v tabulce s m *pozicemi zabere očekávaný čas* $\Theta(n)$ *jakákoliv posloupnost* n *operací* INSERT, SEARCH *a* DELETE, *která obsahuje* $\mathcal{O}(m)$ *operací* INSERT.

Konstrukce univerzální množiny hašovacích funkcí

příklad univerzálního hašování

- zvolíme prvočíslo p takové, že žádný klíč není větší než p
- pro libovolná čísla $a \in \{1, 2, \dots p-1\}$ a $b \in \{0, 1, \dots p-1\}$ definujeme hašovací funkci předpisem

$$h_{ab}(k) = ((ak + b) \mod p) \mod m)$$

- množina funkcí

$$\mathcal{H}_{pm} = \{h_{ab} | a \in \{1, 2, \dots p-1\}, b \in \{0, 1, \dots p-1\}$$

  je univerzální množinou hašovacích funkcí
- výběr prvočísla umožňuje efektivní implementaci operací $\mod$

přesné důkazy tvrzení jako i další podrobnosti týkající se univerzálního hašovaní jsou v literatuře, např. v monografii T. Cormen, Ch. Leiserson, R. Rivest, C. Stein: Introduction to Algorithms. *Third Edition. MIT Press, 2009*

Otevřená adresace

- všechny klíče ukládáme přímo do tabulky, počet klíčů nemůže přesáhnout velikost tabulky
- při vyhledávání se systematicky zkoumají pozice tabulky, dokud není nalezen hledaný klíč nebo není jasné, že v tabulce není
- nepotřebujeme seznamy a ukazatele, místo nich se počítá sekvence pozic v tabulce, které mají být prozkoumány (tzv. sondování)

Otevřená adresace - vyhledávání

- hašovací funkce je typu $h : U \times \{0, 1, \ldots, m-1\} \rightarrow \{0, 1, \ldots, m-1\}$
- pro každý klíč potřebujeme posloupnost $\langle h(k, 0), h(k, 1), \ldots h(k, m-1) \rangle$, která je permutací posloupnosti $\langle 0, 1, \ldots, m-1 \rangle$
- každá pozice tabulky obsahuje buď klíč, anebo hodnotu $\textsc{Nil}$
- při hledání klíče k
  - proměnná i je rovna pořadovému číslu testu, iniciální hodnota i je 0
  - vypočítáme hodnotu $h(k, i)$ a testujeme obsah pozice $h(k, i)$
  - když pozice $h(k, i)$ obsahuje klíč k, vyhledávání je úspěšné
  - když pozice $h(k, i)$ obsahuje hodnotu $\textsc{Nil}$, vyhledávání je neúspěšné (tabulka neobsahuje klíč k)
  - když pozice $h(k, i)$ obsahuje neprázdnou hodnotu různou od k, tak zvýšíme pořadové číslo testu a vypočítáme novou pozici v tabulce jako funkci k a pořadového čísla testu a klíč hledáme pomocí této nové hašovací funkce

Otevřená adresace - vkládání

- analogicky jako při vyhledávání najdeme volnou pozici v tabulce
- vkládání skončí úspěchem když je nalezena volná pozice, na kterou se klíč vloží
- když počet testů dosáhne m, tak vkládání končí neúspěchem

Otevřená adresace - odstranění klíče

- vyhledáme klíč k v tabulce, nechť se nalézá na pozici j
- může nastat situace, že po odstranění klíče k budeme v tabulce vyhledávat klíč k', který je v tabulce uložen) a v průběhu jeho vyhledávání budeme zkoumat i pozici j
- když by jsme na pozici j vložili hodnotu NIL, tak by jsme při následném vyhledávání klíče k' dostali nesprávný výsledek

řešení

- místo hodnoty NIL použijeme speciální hodnotu DELETED
- operace INSERT považuje pozici s hodnotou DELETED za prázdnou
- operace SEARCH považuje pozici s hodnotou DELETED za obsazenou, ale obsahující jinou hodnotu než hledaný klíč

Otevřená adresace - výpočet sekvence sond

nejčastěji se používají k výpočtu sekvence sond tři techniky

- lineární adresace *(linear probing)*
- kvadratická adresace *(quadratic probing*
- dvojité hašování *(double hashing)*

Otevřená adresace - lineární

využívá pomocnou hašovací funkci $h' : U \longrightarrow \{0, 1, \ldots, m-1\}$

$$h(k, i) = (h'(k) + i) \mod m$$

- pro daný klíč je nejdříve prozkoumána pozice $T[h'(k)]$, pak pozice $T[h'(k)+1]$, ..., $T[m-1]$ a pak zase od $T[0]$ až k $T[h'(k)-1]$
- problémem je tzv. primární shlukování, které může výrazně zvýšit složitost operací

Otevřená adresace - kvadratická

využívá pomocnou hašovací funkci $h' : U \longrightarrow \{0, 1, \ldots, m-1\}$ a pomocné konstanty $c_1, c_2 \neq 0$

$$h(k,i) = (h'(k) + c_1 i + c_2 i^2) \mod m$$

- pro daný klíč je nejdříve prozkoumána pozice $T[h'(k)]$, dále pak pozice posunuta o offset závislý kvadratickým způsobem na pořadí sondy
- kvadratická adresace je obvykle lepší než lineární
- problémem je vhodný výběr konstant c_1 a c_2 a velikosti tabulky m
- když dva klíče jsou primárně zahašovány na stejnou pozici protože $h'(k_1) = h'(k_2)$, tak mají stejnou celou posloupnost sond - tzv. sekundární shlukování

Otevřená adresace - dvojité hašování

využívá dvě pomocné hašovací funkce h_1, h_2

$$h(k, i) = (h_1(k) + ih_2(k)) \mod m$$

- pro daný klíč je nejdříve prozkoumána pozice $T[h_1(k)]$, následující pozice je posunuta o offset $h_2(k) \mod m$
- hodnota $h_2(k)$ musí být nesoudělná s velikostí hašovací tabulky m, aby byla prohledána celá tabulka
- vhodnou volbou je vzít m jako mocninu 2 a navrhnout h_2 tak, že výsledkem bude vždy liché číslo, nebo
- zvolit m jako prvočíslo a navrhnout h_2 tak, že výsledkem bude vždy kladné číslo $< m$
- dvojité hašování je lepší než kvadratické, protože generuje $\Theta(m^2)$ posloupností sond místo $\Theta(m)$ jako kvadratická adresace

Otevřená adresace - složitost

Věta 8

Pro hašovací tabulku s otevřenou adresací s faktorem naplnění $\alpha = n/m < 1$ *je očekávaný počet sond při neúspěšném hledání nejvýše* $1/(1-\alpha)$ *a to za předpokladu uniformního hašování.*

Věta 9

Pro hašovací tabulku s otevřenou adresací s faktorem naplnění $\alpha = n/m < 1$ *je očekávaný počet sond při úspěšném hledání nejvýše* $\frac{1}{\alpha}\ln\frac{1}{1-\alpha}$ *a to za předpokladu uniformního hašování.*

uniformní hašování je takové, že každý klíč má jako posloupnost sond se stejnou pravděpodobností libovolnou z $m!$ permutací $\langle 0, 1, \ldots, m-1 \rangle$

Kukaččí hašování (Cuckoo hashing)

- pro hašování se používají **dvě tabulky** velkosti m a **dvě hašovací funkce** $h_1, h_2 : U \longrightarrow \{0, 1, \ldots m-1\}$
- každý klíč k je zahašovaný buď na pozici $h_1(k)$ v první tabulce, anebo na pozici $h_2(k)$ v druhé tabulce

- **hledání klíče** má konstantní složitost, protože stačí otestovat dvě pozice

- **odstranění klíče** má konstantní složitost, analogicky jako jeho hledání

- při **vkládání nového klíče** k se použije *hladová strategie*: nejdříve se pokusíme vložit klíč k na pozici $h_1(k)$
- když je pozice $h_1(k)$obsazena, tak klíč y uložený na pozici $h_1(k)$ přesuneme do druhé tabulky na jeho alternativní pozici $h_2(y)$
- proces opakujeme a přepínáme se mezi tabulkami dokud nenajdeme volnou pozici, anebo se proces zacyklí

R. Pagh, F. Rodler: Cuckoo hashing. Journal of Algorithms 51 (2004) 122 - 144

Dokonalé hašování (Perfect hashing)

- hašování, které má konstantní složitost i v nejhorším případě
- předpokladem je statická množina klíčů
- využívá dvě úrovně hašování

první úroveň

v podstatě stejná, jako zřetězené hašování

druhá úroveň

- místo seznamů použijeme sekundární hašovací tabulky S_j s asociovanou hašovací funkcí h_j, přičemž vhodným výběrem můžeme zajistit, aby na druhé úrovni nebyly žádné kolize
- velikost m_j tabulky S_j je kvadratická vůči počtu klíčů zahašovaných na pozici j
- hašovací funkce na první úrovni se vybírá z univerzální množiny hašovacích funkcí $\mathcal{H}_{pm}$, na druhé úrovni z univerzální množiny $\mathcal{H}_{pm_j}$