Petr Osička

Poznámky k předmětu

ALGORITMY 2

s 53 obrázky

katedra informatiky

OBSAH

ÚVOD

Pojmem *datová struktura* myslíme systematický způsob organizace dat v programu, především jejich uložení v paměti a algoritmy pro efektivní přístup k takto uloženým datům.

Práce s daty a datovými strukturami je velmi důležitou součástí programování a návrhu a implementace algoritmů. Vhodně zvolená datová struktura vede k efektivnějšímu algoritmu, případně k přehlednějšímu a efektivnějšímu programu. Důležitosti datových struktur se týká i řada citátů známých informatiků, například

> *Bad programmers worry about the code. Good programmers worry about data structures and their relationships.*
>
> **Linus Torvalds**

> *Data dominates*
>
> **Rob Pike**

Niklaus Wirth, držitel Turingovy ceny a autor Pascalu, pojmenoval svoji knihu *Algorithms + data structures = programs*.

Datová struktura je konkrétní realizací či implementací *abstraktní datové struktury*. Ta popisuje rozhraní mezi uživatelem struktury a strukturou samotnou. Typicky je nutno uvést

- popis datových položek, které bude struktura obsahovat,
- popis operací, které struktura podporuje,
- další požadavky, například očekávaná složitost vybraných operací

Jako příklad uvažme abstraktní datovou strukturu `set` pro množinu celých čísel. Uvedeme tedy, že datovou položkou je celé číslo, pořadujeme operace pro přidání do množiny, odebrání z množiny a pro test toho, jestli se dané číslo v množině vyskytuje.

Abstraktní datovou strukturu `set` můžeme popsat i obecněji, můžeme například chtít, aby datové položky tvořili množinu hodnot s rovností, tj. stačí abychom uměli pro dvě hodnoty určit, jestli jsou stejné. Datovou struktura pro tuto obecnější definici `set` lze použít i pro specifičtější případ z předchozího odstace, opačně to ovšem být nemusí. Datové struktury vytvořené specificky pro celá čísla existují, a nelze je použít jiný typ datových položek.

Přirozeným rozšířením `set` je `dictionary`. Datové položky jsou zde tvořeny dvojicemi `(klíč, data)`, `dictionary` tak uchovává asociace mezi klíči a libovolnými daty, na která neklademe žádné další požadavky. Klíče oproti tomu musíme umět porovnat: testovat je na rovnost, nebo dokonce zjistit, jestli je nějaký klíč menší než jiný klíč, či dokonce požadujeme aby klíče tvořily lineárně uspořádanou množinu. `dictionary` je primárně určeno pro vyhledávání: pro daný klíč chceme najít data, která jsou s ním asociována.

V kurzu budeme probírat převážně datové struktury, které lze použít k implementaci `dictionary`. Přitom budeme pro jednoduchost uvažovat pouze klíče a ignorovat jim asociovaná data. Tyto struktury tak vlastně implementují `set`, kde ukládáme množinu klíčů. (Existují ovšem datové struktury pro `set`, kde se ke klíčům nedají přidat asociaovaná data.)

I.1 POINTER MACHINE

K dostatečně přesnému výkladu datových struktur použijeme fiktivního modelu počítače, tj. modelu výpočtu, správy paměti a příslušného programovacího jazyka (v našem případě spíše pseudokódu), kterému budeme říkat *pointer machine*.

Základním kamenem jsou struktury, které mají *referenční sémantiku*. Struktura je pojmenovaný soubor pojmenovaných položek. V pseudokódu strukturu vytvoříme následovně.

```
struct Node
   id: Key
   next: Node
   prev: Node
```

Na prvním řádku zavádíme strukturu se jménem `Node`. Na každém dalších řádku pak vyjmenováváme jednotlivé položky této struktury, ve tvaru `jmeno: typ`. Typ má stejnou úlohu jako v programovacích jazycích, určuje množinu hodnot, které může daná položka nabýt. Mezi tyto typy patří například `Key`, který je rezervovaným typem pro klíče. Zavedením struktury vytváříme nový typ, jehož jménem je jméno dané struktury, zde `Node`. Instanci struktury vytvoříme procedurou, která má stejné jméno jako struktura.

```
x ← Node()
```

Symbol ← je operace přiřazení. Volání procedury `Node()` alokuje v paměti místo pro strukturu `Node` a vrátí na ni referenci. Tím vytvoří instanci dané struktury, která je uložena v paměti. Reference potom slouží jako jednoznačný identifikátor této instance, který umíme testovat na rovnost. Nic dalšího o referencích nepředpokládáme. V příkladu výše je přiřazena proměnné `x`. Referenci můžeme využít pro přístup k jednotlivým položkám s využitím tečky a jména položky zapsaného za referenci. (Pro konkrétní hodnoty klíčů budeme v příkladech často používat celá čísla.)

```
x.id ← 5
```

O paměti, kterou pointer machine využívá nebudeme víc vědět nepotřebujeme. Stačí nám mechanismus přístupu přes referenci a úvědomnění si, že entity (instance struktur a další) existují, když mají alokované místo v paměti.

Uvažme následující příklad.

```
1  x ← Node()
2  y ← Node()
3  z ← x
4  z.id ← 10
5  y.next ← z
```

Za předpokladu, že reference jsou celá čísla (tento předpoklad děláme jenom proto, že nějaký konkrétní objekt jako referenci v příkladu potřebujeme), můžeme situaci po provedení kroku 3 zapsat do následujících dvou tabulek. V levé tabulce je obsah paměti a v pravé tabulce hodnoty proměnných. Procedura `Node()` inicializuje položky na přirozené hodnoty: u `id` je to 0, u referencí je to `nil`. Tato speciální hodnota představuje referenci, ke které v paměti nepatří žádná struktura a je tedy neplatná.

reference	položky

1	`id = 0`
	`prev = nil`
	`next = nil`

2	`id = 0`
	`prev = nil`
	`next = nil`

Proměná	hodnota
x	1
y	2
z	1

Po provedení řádků 4 a 5 v příkladu se situace změní na následující.

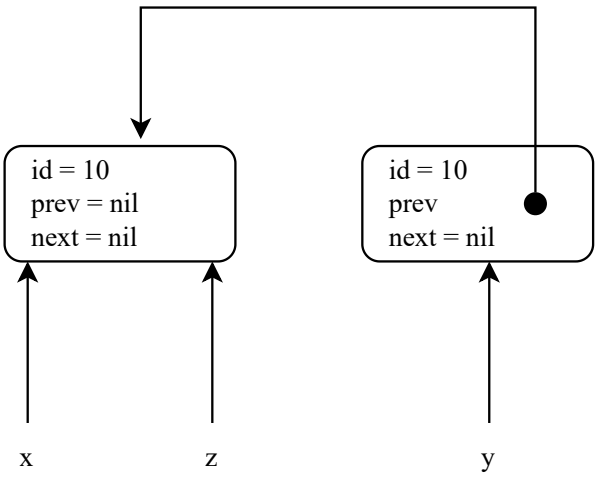
reference	položky

1	`id = 10`
	`prev = nil`
	`next = nil`

2	`id = 0`
	`prev = nil`
	`next = 1`

Proměná	hodnota
x	1
y	2
z	1

Situaci lze i namalovat. Na obrázku jednotlivé instance struktur reprezentujeme obdélníky, do nichž zapíšeme hodnoty položek. Reference potom zachytíme pomocí šipek. Z obrázku jde vidět, že se potom na konkrétní datovou strukturu můžeme dívat jako na graf, jeho vrcholy jsou jednotlivé instance struktur a hrany vyčteme z referencí (ať už je budeme brát jako orientované, nebo neorientované).



Z referenční sémantiky struktur také plyne, že pokud porovnáváme dvě proměnné, odkazující se na instanci struktury, porovnáváme reference, nikoliv hodnoty položek. Toho často využijeme pro porovnání s `nil` pro zjištění platnosti reference.

```
x ← Node()
y ← Node()
x.id ← 10
y.id ← 10
x = y     // Výsledkem porovnání je nepravda, i když hodnoty položek jsou stejné
x = nil     // Výsledkem porovnání je nepravda, x obsahuje platnou referenci
```

Pole je základní datová struktura, která slouží k uložení fixního počtu datových položek, kterým také říkáme *prvky* pole. Počet prvků, tomu říkáme *velikost pole*, je nutno zadat při vytváření pole, později jej nelze změnit.

```
— Příklad vytvoření pole velikosti 6, s inicializací jednotlivých prvků
A ← [6]Key{5, -2, 13, 3, 10, 42}

— Příklad vytvoření pole velikosti 5, inicializace na vychozí hodnoty
B ← [5]Key

— Procedura len vrací velikost pole
len(A) — Výsledek je 6
```

K prvků pole přistupujeme pomocí indexů `0, 1, ...`, největším indexem je velikost pole zmenšená o jedna. Tento přístup má konstatní složitost, nezávisí vůbec na velikosti pole.[1]

```
— Syntax přístupu k prvkům pole
A[3] ← 10
B[1] ← A[5] + 1
```

V běžných programovacích jazycích jsou prvky pole uloženy v paměti za sebou a přístup k prvku na indexu `i` je realizován výpočtem:

```
adresa-zacatku-pole + i * velikost-jednoho-prvku.
```

V pointer machine mají pole referenční sémantiku. Pro vytváření kopií pole (a struktur) budeme používat následující procedury.

```
— vytvoříme nové pole/strukturu, jejíž obsah je shodný se src a referenci uložíme do dest
dest ← clone(src)

— zkopírujeme obsah src do dest, dest už musí existovat
copy(dest,src)
```

Jsou-li argumenty `copy` pole, potom se obsah `src` kopíruje od začátku (indexu `0`) na začátek `dest`, přičemž kopírování končí indexem `min(len(src), len(dest)) - 1`.

2.1 INDEX JAKO KLÍČ

Pokud jsou klíče z množiny `{0,1,…, n - 1}`, lze pole o velikosti `n` jednoduše použít k implementaci abstraktní datové struktury `set` (pro podmnožinu množiny `{0,1,…, n - 1}`. Stačí k tomu vytvořit pole, jehož prvky mohou nabývat hodnot `0` a `1`.[2] Pokud je na indexu `k` v poli `1`, pak `k` do množiny patří, jinak je tam `0` a `k` do množiny nepatří.

[1] V literatuře nalezneme tuto vlastnost popsánu větou: *K prvkům pole máme náhodný přístup*.

[2] Lze si vybrat jinou dvojici různých hodnot, v programovacích jazycích je například často typ `bool` s hodnotami `true` a `false`.

Operace pro vložení klíče do množiny, odstranění klíče z množiny a test toho, jestli se klíč v množině nachází jsou velmi jednoduché. Navíc mají konstatní složitost.

```
A ← [10]Int — výchozí hodnoty prvků jsou 0

— vložení klíče k
A[k] ← 1

— odebrání klíče k
A[k] ← 0

— je klíč k v množině?
A[k] = 1
```

Čtenář si snadno představí, jak v naší situaci fungují další operace, například nalezení nejmenšího nebo největšího klíče v množině, sjednocení a průnik dvou množin a další.

Strukturu můžeme také snadno rozšířit tak, aby byla implementací `dictionary`. Pro udržování asociace mezi klíčem `k` a jemu odpovídajícími daty stačí v poli index `k` uložit právě tato data. Pokud s klíčem `k` žádná data asociována nejsou, dáme do pole na index `k` nějaký příznak, podle kterého to poznáme.

2.2 OBECNÉ KLÍČE

Vytvoříme datovou strukturu pro abstraktní datovou strukturu `dictionary`. (Budeme se pořád držet úmluvy z předchozího textu, datové položky v `dictionary` budou pouze klíče.) Ideou je udržovat datové položky umístěny na začátku pole: pokud je ve struktuře `n` položek, jsou na indexech `0` až `n-1`. Současně si budeme pamatovat nejmenší index neobsazeného prvku, v našem případě je roven `n`.

```
struct Array-Dict
   data: []Key
   top: Int
```

Struktura podporuje tři operace, přidání a odebrání klíče, a vyhledání indexu, na kterém se zadaný klíč nachází.

```
insert
 ⟵ ad: Array-Dict
 ⟵ k: Key
```

```
1  if (ad.top < len(ad.data))
2     ad.data[ad.top] ← k
3     ad.top ← ad.top + 1
```

Díky testu na řádku 1 se vložení provede pouze pokud je v poli `ad.data` ještě neobsazený prvek. Tento test by šel vypustit, museli bychom ovšem zajistit, že operaci provedeme pouze v situaci, kdy `ad.data` není plné. V testu využíváme toho, že díky indexování od nuly je `ad.top` roven počtu klíčů ve struktuře.

```
remove
⟵ ad: Array-Dict
⟵ i: Int – index odebíraného prvku
```

```
1  if (i ⩾ 0 and i < ad.top)
2     ad.data[i] ← ad.data[ad.top - 1]
3     ad.top ← ad.top - 1
```

Odebíraný klíč identifikujeme pomocí indexu, na kterém je v poli `ad.data`. Mazání provedeme tak, že na jeho místo vložíme klíč, který se nachází na konci obsazené části pole (na indexu `ad.top - 1`). Poté `ad.top` dekrementujeme o 1 a zmenšíme tak počet klíčů ve struktuře. Složitost odebrání je konstatní. Na řádku 1 navíc kontrolujeme platnost indexu `i`.

Poslední slíbenou operací je vyhledání klíče, věnujeme se jí v další kapitole.

2.3 VYHLEDÁVÁNÍ V POLI

Úkol nalezení prvku v poli je následující: máme pole klíčů `array` (ve kterém se žádný klíč nenachází na dvou různých indexech) a klíč `k`. Chceme nalézt index, na kterém se `k` v poli `array` nachází, nebo zjistit, že se v něm nenachází. Pokud o rozmístění klíčů v poli nemůžeme předpokládat nic dalšího, nezbude nám nic jiného, než použít následující proceduru.

```
array-search
⟵ array: []Key — prohledávané pole
⟵ k: Key — hledaný klíč
⟶ Int — index klíče k nebo -1, pokud v ar není
```

```
1  for i in 0 ..< len(array)
2     if (array[i] = k) then return i
3  return -1
```

Postupně v cyklu procházíme všechny indexy v poli (od `0` až do `len(array) - 1`). Zastavíme v momentě, kdy je na nějakém indexu klíč `k`, daný index je potom výsledkem. Pokud se na žádném zkoumaném indexu `k` nenachází, vrátíme `-1` značící, že `k` v poli není.

V nejhorším případě je složitost `array-search` lineární, například pokud hledání končí výsledkem `-1`. K analýze složitosti v průměrném případě potřebujeme znát frekvence vyhledávání klíčů na jednotlivých indexech. Označme f_i frekvenci volání `array-search`, ve kterých se hledaný klíč nachází na indexu `i`. Máme tedy

$$f_i = \frac{\text{počet volání } \texttt{array-search} \text{ s } \texttt{k} \text{ rovným array[i]}}{\text{počet všech volání } \texttt{array-search}}$$

Můžeme uvažovat pouze o frekvencích volání, které končí nalezením klíče. Můžeme předstírat, že pole je o jedna delší než ve skutečnosti, a neúspěšná volání v kratším poli jsou úspěšná volání v delším poli, která končí nalezením klíč na posledním políčku. Průměrný počet provedených porovnání, na němž je průměrná složitost linárně závislá, je potom

$$f_0 + 2 \cdot f_1 + 3 \cdot f_2 + \cdots + n \cdot f_{n-1},$$

kde n je velikost pole. Například, pokud pro všechny indexy i platí $f_i = \frac{1}{n}$, je průměrný počet porovnání roven $\frac{n+1}{2}$. V průměru pole prohledáme do poloviny.

Snadno nahlédneme, že vhodným uspořádáním klíčů v poli můžeme zmenšit průměrný počet porovnání (u úspěšných volání), které je nejmenší možné, pokud

$$f_0 \geqslant f_1 \geqslant \cdots \geqslant f_{n-1}.$$

V extrémních případech může být průměrná složitost vyhledávání konstantní. Uvažme například frekvence

$$\begin{aligned} f_0 &= \frac{1}{2}, \\ f_1 &= \frac{1}{4}, \\ &\vdots \\ f_{n-2} &= \frac{1}{2^{n-1}}, \\ f_{n-1} &= \frac{1}{2^{n-1}}. \end{aligned}$$

Označíme-li $c(n) = \sum_{i=0}^{n}(i+1) \cdot f_i$, potom snadno ověříme že pro $n \geqslant 2$ máme

$$c(n) - \frac{c(n-1)}{2} = 1.$$

Počet porovnání je tak dán rekurencí

$$\begin{aligned} c(n) &= \frac{c(n-1)}{2} + 1, \\ c(1) &= 1, \end{aligned}$$

pro níž indukcí snadno dokážeme, že $c(n) \leqslant 2$.

Pokud je pole `array`, ve kterém vyhledáváme setřízeno, tj. pro každé dva indexy `i < j` platí `array[i] < array[j]`, potom můžeme vyhledávání významně zrychlit. Pro libovolný index `s` jsou na indexech menších než `s` klíče menší než `array[s]`; na indexech větších než `s` jsou pak klíče větší než `array[s]`. Nabízí se tak následující algoritmická myšlenka hledání klíče `k`:

1. vybereme index `s`,
2. pokud je `array[s] > k`, pokračujeme s vyhledávání v části pole s indexy menšími než `s`,
3. v opačném případě pokračujeme s vyhledáváním v části pole s indexy většími než `s`.

Část pole, ve kterém vyhledáváme tak se tak zmenší. Pokud zařídíme, aby se zmenšila významně, vyhledávání zrychlíme. Pokud například zvolíme `s` tak, rozdělilo pole na dvě (přibližně) stejně velké části, máme záruku, že se prohledávaná část pole vždy zmenší na polovinu. To vede k algoritmu známému jako *binární vyhledávání* nebo *vyhledávání půlením intervalů.*

```
binary-search
⟵ array: []Key — prohledávané pole
⟵ k: Key — hledaný klíč
⟶ Int — index klíče k nebo -1, pokud v ar není
```

```
1  l ← 0
2  p ← len(array) - 1
3  while (l ⩽ p)
4     s ← floor((l + p) / 2)
5     if (ar[s] = k) then return s
6     if (ar[s] > k) then p ← s - 1
7     if (ar[s] < k) then l ← s + 1
8  return -1
```

V algoritmu prohledávanou část pole vymezujeme indexy `l` a `p`. Na začátku jsou tyto indexy nastaveny tak, aby vyhledávaná část odpovídala celému poli. Index `s` vypočítáme jako `floor((l + p) / 2)`, kde `floor` je procedura zaokrouhlující dolů. Čtenář snadno ověří, že `(s - l)` a `(p - s)` se liší nejvýše o jedna. V závislosti na výsledku porovnání hledaného klíče s `array[s]` potom upravíme indexy `p, l`. Je užitečné představit si situaci, kdy je `l = p`. V ní vyhledáváme v části pole velké jeden prvek, platí `s = l = p` a pokud `array[s]`$\neq$ `k`, dostaneme po úpravě indexů (řádky `6,7`) nerovnost `l > p`. V této situaci končí vyhledávání neúspěchem.

Mimo řádku `8` vždy platí alespoň jedna z nerovností

$$\texttt{array[l]} \leqslant \texttt{k} \text{ a } \texttt{k} \leqslant \texttt{array[p]}.$$

Je tomu tak na začátku algoritmu, a řádky `4--7` tuto nerovnost zachovávají. Každé provedení cyklu `while` zmenší `(p-l+1)` na polovinu původní hodnoty nebo ještě o jedna méně, minimálně však o `1`. Proto pokud je `k` v poli, na řádku `5` jednou nastane rovnost. Nejpozději v situaci, kdy `p=l`. Naopak, pokud `k` v poli není, jednou dojde k tomu, že `l>p`. Složitost v nejhorším případě je dána rekurencí

$$\begin{aligned} T(1) &= O(1), \\ T(n) &= T(n/2) + O(1), \end{aligned}$$

jejímž řešením je $\Theta(\log n)$.

V interpolačním vyhledávání počítáme index `s` jinou metodou. Položíme

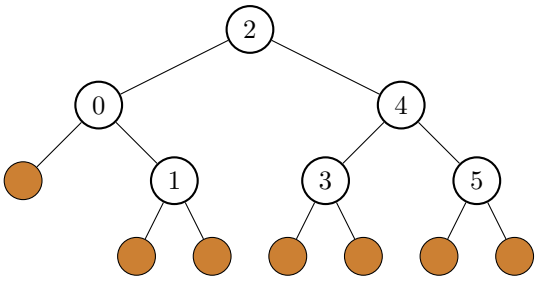
$$\begin{aligned} x &= \texttt{array[l]}, \\ y &= \texttt{array[p]}, \end{aligned}$$

(a máme tedy $x \leqslant k \leqslant y$). Potom spočítáme

$$\begin{aligned} r &= (k - x)/(y - x), \\ s &= \lfloor l + (p - l) * r \rfloor \end{aligned}$$

Interpolační vyhledávání funguje nejlépe, pokud jsou klíče čísla a všechny rozdíly mezi sousedními prvky v poli jsou přibližně stejně velké, složitost vyhledávání pak může být až $\Theta(\lg \lg n)$. Intuitivně můžeme interpolační vyhledávání přirovnat k hledání slova ve slovníku. Obvykle se podíváme na první písmeno slova, a pokud je například ve dvou třetinách abecedy, otevřeme slovník přibližně ve dvou třetinách (a nikoliv v jedné polovině).

Vyhledávací algoritmus založený na diskutovaných myšlenkách můžeme vizualizovat pomocí binárního stromu. Pro dvojici indexů `(l,p)` sestavíme binární strom, jehož kořenem jim odpovídající hodnota indexu `s`, levým podstromem je strom sestavený pro `(l, s-1)` a pravým podstromem strom sestavený pro `(s+1, l)`. Pokud je `l > p`, výsledek zakreslíme bronzovým vrcholem. Na následujícím obrázku vidíme takto sestavený strom pro binární vyhledávání v poli velikosti 6.



2.4 DYNAMICKÉ POLE

Dynamické pole je struktura, která řeší problémy s fixní velikostí pole tak, že v případě potřeby pole nahradí větším (nebo menším) polem, do nějž zkopíruje obsah toho původního. Dynamické pole zavedeme jako strukturu:

```
struct Dynamic-Array
   data: []Key — pole obsahujici data
   top: Int — velikost pole, může se lišit od len(data)
```

U dynamického pole `a` označujeme `len(a.data)` jako *kapacitu*, a `a.top` jako `velikost`. Ke změnám kapacity pole využijeme následující procedury

```
enlarge
⟵ A: []Key
⟶ []Key
```

```
1 new ← [2 * len(A.data)]Key
2 copy(new, A.data)
3 return new
```

```
shrink
⟵ A: []Key
⟶ []Key
```

```
1 new ← [len(A.data) / 2]Key
2 copy(new, A.data)
3 return new
```

Procedura `enlarge` zvětšuje kapacitu na dvojnásobek a procedura `shrink` ji zmenšuje na polovinu. Toto je dobrá volba z pohledu složitosti operací, příklad uvidíme později v

této kapitole.

Přidávat klíče do dynamického pole lze v zásadě pomocí tří operací

- `append` vkládá na konec pole,
- `assign-at` vloží klíč na konkrétní index, původní klíč je přepsán
- `inject-at` vkládá na konkrétní index, původní klíč není přepsán, ale je spolu s klíči na vyšších indexech posunut (na index o jedna vyšší).

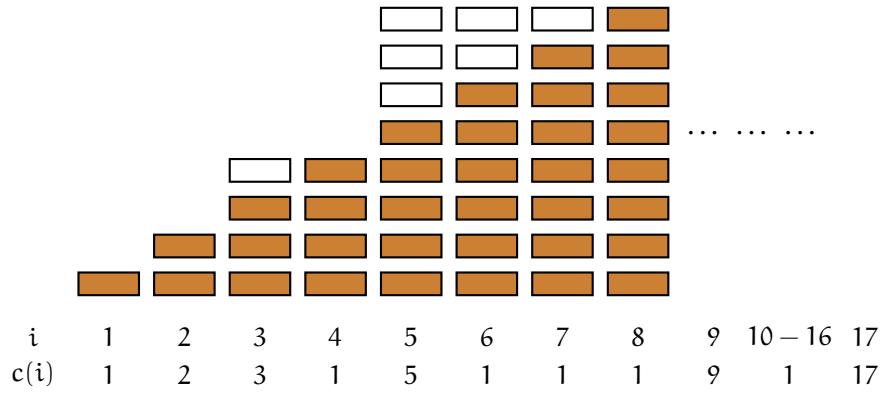
Jednotlivé operace si nyní projdeme.

```
append
⟵ A: Dynamic-Array
⟵ k: Key — vkládaný klíč
```

```
1 if (A.data = nil) then A.data ← [1]Key
2 if (A.top = len(A.data)) then A.data ← enlarge(A.data)
3 A.data[A.top] ← k
4 A.top ← A.top + 1
```

Operace `append` je velmi podobná operaci `insert` z kapitoly 2.2. Rozdíl je v tom, že nyní navíc na řádcích `1` a `2` kontrolujeme velikost `data` a v případě potřeby jej zvětšíme.

Při analýze složitosti musíme připustit, že ačkoliv se někdy může stát, že `append` má konstantní složitost, pokud dojde na řádku `2` ke zvětšení kapacity, dojde v proceduře `enlarge` ke zkopírovaní všech prvků v `data` do nového pole. Složitost je tak lineární. Podívejme se ovšem na situaci podrobněji a představme si posloupnost n zavolání `append`. Pro prvních pár zavolání můžeme situaci zobrazit na následujícím obrázku.



Řádek označený i říká o kolikáté zavolání se jedná, $c(i)$ je počet operací zápisu prvku do pole, které při i-tém zavolání `append` provedl. Sloupce potom visualizují položku `data`, bíle jsou značeny neobsazená a bronzově obsazená políčka tohoto pole. Podíváme-li se na `c(i)`, všimneme si, že pokud je `i` mocninou 2 zvětšenou o jedna, došlo při daném volání ke zvětšení kapacity. Původní kapacita byla $i - 1$, operace `enlarge` tak překopírovala $i-1$ prvků. Navíc jsme zapsali nově vkládaný prvek na konec. Celkem tedy došlo k i zápisům. Pokud i není mocninou dvou zvětšenou o jedna, dojde pouze k jednomu zápisu. Máme

tak

$$c(i) = \begin{cases} i & \text{pokud je } i-1 \text{ mocninou 2,} \\ 1 & \text{jinak.} \end{cases}$$

Dále

$$\sum_{i=1}^{n} c(i) \leqslant n + \sum_{j=0}^{\lfloor \lg n \rfloor} 2^j < n + 2n = 3n.$$

Na jednu operaci `append` tak průměrně (rozpočítáno přes n operací) vyjdou maximálně 3 zápisy do pole. Složitost jedné operace je tak z tohot pohledu $\Theta(1)$. Takovému přístupu ke složitosti, kdy rozdělíme složitost provedení celé sekvence operací na jednotlivé operace, říkáme *amortizovaná složitost*.

```
assign-at
⟵ A: Dynamic-Array
⟵ i: Int — index, kde měníme obsah
⟵ k: Key — vkládaná hodnota
```

```
1  cap ← len(A.data)
2  while (i ⩾ cap) do cap ← cap * 2
3  if (cap > len(A.data)
4     new ← [cap]Key
5     copy(new, A.data)
6     A.data ← new
7  A.data[i] ← k
8  for j in A.top ..< k do A.data[j] ← 0
9  if (A.top < k) then A.top ← k + 1
```

V operaci `assign-at` musíme řešit dvě zvláštní situace. První nastane, pokud je `i` větší než současná kapacita dynamického pole. V takové situaci musíme najít kapacitu novou, je to nejmenší mocnina dvou, která je ostře větší než `i`. V proceduře tuto kapacitu hledáme na řádku `3`. Na tuto novou kapacitu pole `A.data` na řádcích `3` až `6` zvětšíme. Druhá zvláštní situace nastane, pokud je `i > A.top`. Do dynamického pole nám totiž přibudou i prvky `A.data` na indexech `A.top` až `i-1`. Dopředu nevíme, co jaké klíče tyto prvky obsahují. Jedním řešením této situace na nastavit je na výchozí hodnotu pro typ hodnot, které v dynamickém poli uchováváme (u klíčů je to hodnota 0). Složitost operace je v nejhorším případě lineární vzhledem k `i`.

Operace `inject-at` je velmi podobná `assign-at`. Pokud je index `i`, na který vkládáme, větší roven `top`, chová se `inject-at` stejně jako `assign-at`. V opačném případě musíme po případném zvětšení kapacity posunout všechny prvky od indexu `i` až pod index `top - 1` o jednu pozici doprava a poté vložit nový klíč na index `i`. Detaily a složitost si laskavý čtenář dopracuje sám.

Pro odebírání poskytuje dynamické pole tři operace.

- `unordered-remove` odebírá prvek stejně jako procedura `remove` v kapitole 2.2.

- `ordered-remove` odebírá prvek tak, že pořadí ostatních klíčů zůstává nezměněno
- `pop` odebírá poslední prvek v dynamickém poli, a vrací jej jako návratovou hodnotu.

Všechny tři operace berou jako argument index odebíraného prvku. U všech operací dojde ke zmenšení kapacity, pokud je velikost dynamického pole menší nebo rovna čtvrtině jeho kapacitu. Po zmenšení kapacity je tedy kapacita využita z jedné poloviny. Volba jedné čtvrtiny je výhodná z pohledu amortizované analýzy, vyhneme se situaci, kdy díky střídáním operací odebírání a přidávání neustále dochází ke zvětšování a zmenšování kapacity.

```
unordered-remove
⟵ A: Dynamic-Array
⟵ i: Int — index odebíraného prvku
```

```
1 A.data[i] ← A.data[A.top - 1]
2 A.top ← A.top - 1
3 if (A.top ⩽ len(A.data) / 4) then A.data ← shrink(A.data)
```

```
pop
⟵ A: Dynamic-Array
⟶ Key — vrácený prvek
```

```
1 ret ← A.data[A.top - 1]
2 A.top ← A.top - 1
3 if (A.top ⩽ len(A.data) / 4) then A.data ← shrink(A.data)
4 return ret
```

```
ordered-remove
⟵ A: Dynamic-Array
⟵ i: Int — index odebíraného prvku
```

```
1 for j in i ..< A.top - 1
2    A.data[j] ← A.data[j + 1]
3 A.top ← A.top - 1
4 if (A.top - 1 ⩽ len(A.data) / 4) then A.data ← shrink(A.data)
```

Operace `ordered-remove` je analogií `inject-at`. Prvek na indexu `i` odebereme tak, že všechny prvky na indexech větších než `i` posunume na o jedna menší index.

Všechny tři operace mají lineární složitost v případech, kdy dojde ke zmenšení kapacity. Pokud nedojde k snížení kapacita, jsou složitosti `unordered-remove` a `pop` konstantní. U operace `ordered-remove` ovšem složitost závisí i na volbě indexu `i`, její složitost je totiž lineární vzhledem k rozdílu `top - i`.

2.5 CYKLICKÝ BUFFER

Cyklický buffer je datová struktura, která často slouží jako vyrovnávací pamět. Poskytuje dvě operace: `write` pro vkládání prvků a `read` pro odebírání prvků. Přitom jsou prvky

odebírány ve stejném pořadí, v jakém byly vloženy.

Základní ideou implementace pomocí pole mít indexy `head` a `tail` a jako obsah bufferu brát prvky na indexech `head`, až `tail-1`. Přitom zapisujeme na index `tail`, po zápisu se tento index o jedna zvětší. Prvky čteme z indexu `head`, po čtení se tento index také o jedna zvětší.

Lze si rychle všimnout, že ať už uděláme pole jakkoliv velké, po konečném počtu operací `write` dojedeme s indexem `tail` na konec pole. Do bufferu by tak už nešlo zapisovat, a to i v případě, kdy je `head` větší než 0 a na začátku pole je tak pro vložení nových prvků místo.

Řešením situace je použít pro inkrementaci indexů aritmetiky modulo velikost pole. Získáme tím iluzi kruhovosti, když s indexem dojedeme na konec pole, inkrementací přejdeme na jeho začátek.

Nakonec je nutno určit, jak poznáme, že je buffer prázný nebo plný. Prázdný buffer poznáme tak, že se `head` a `tail` rovnají. Buffer je plný, pokud je `tail` o jedna menší než `head` (modulo velikost pole).

Detaily lze nalézt v následujícím pseudokódu.

```
struct Cyclic-Buffer
   data:[] Key
   head: Int — index pro čtení
   tail: Int — index pro zápis
```

```
empty?
 ⟵ cb: Cyclic-Buffer
 ⟶ Bool
```

```
return cb.head = cb.tail
```

```
full?
 ⟵ cb: Cyclic-Buffer
 ⟶ Bool
```

```
return (cb.tail + 1) mod len(cb.data) = cb.head
```

```
write
 ⟵ cb: Cyclic-Buffer
 ⟵ k: Key
```

```
1 cb.data[cb.tail] ← k
2 cb.tail ← (cb.tail + 1) mod len(cb.data)
```

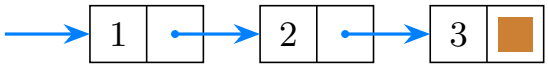
```
read
 ⟵ cb: Cyclic-Buffer
 ⟶ Key
```

```
ret ← cb.data[cb.head]
cb.head ← (cb.head + 1) mod len(cb.data)
return ret
```

Složitosti všech operací jsou konstantní. Existuje i dynamický cyklický buffer, je to varianta cyklického bufferu, ve které se pole zvětšuje podle potřeby. Detaily této možnosti si čtenář dopracuje jako cvičení.

Seznam je základní datovou strukturou, která existuje v několika variantách. V této kapitole se budeme hlavně věnovat té nejjednoduší: jednosměrnému spojovému seznamu. Další varianty krátce zmíníme na konci kapitoly.

Jednosměrný spojový seznam je konečná posloupnost různých uzlů propojených pomocí referencí. Jeden uzel je tvořen strukturou, s položkou pro data (u nás to budou klíče), která v seznamu uchováváme, položkou pro referenci na další uzel v posloupnosti. Poslední prvek v této posloupnosti má tuto referenci nastavenu na `nil`. Následující obrázek zachycuje seznam obsahující klíče `1, 2, 3`.



Pro uzel použijeme strukturu

```
struct Node
   id: Key
   next: Node — další uzel v seznamu
```

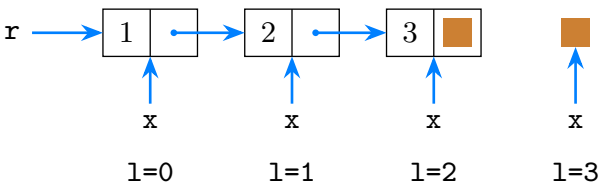
Celý seznam si budeme pamatovat pomocí reference na první uzel. Výhodou toho je, že reference na libovolný uzel v seznamu tak vlastně můžeme brát jako seznam, který začíná tímto uzlem a na seznam se tak můžeme dívat jako na rekurzivní strukturu.

Na následující proceduře, která počítá délku seznamu, si ukážeme idiom procházení seznamu: Je-li aktuální uzel `x`, je následující uzel, který navštívíme `x.next`.

```
list-length
⟵ r: Node
⟶ Int — Délka seznamu, jehož první uzel je r
```

```
1  x ← r
2  l ← 0
3  while (x ≠ nil)
4     l ← l + 1
5     x ← x.next
6  return l
```

Klíčovými částmi idiomu jsou řádky `3` a zejména `5`, ve kterém do proměnné `x` uchovávající aktuální uzel, přiřadíme uzel následující. Průběh algoritmu je zachycen na následujícím obrázku.



Pomocí idiomu procházení seznamu můžeme v seznamu i vyhledávat.

```
search
⟵ r: Node — první uzel seznamu
⟵ k: Key — hledaný klíč
⟶ Node — uzel s klíčem k nebo nil
```

```
1  x ← r
2  while (x ≠ nil and x.id ≠ k)
3     x ← x.next
4  return x
```

Složitost vyhledávání je lineární, v nejhorším případě musíme projít celý seznam.

Vyhledávání půlením intervalů má pro seznamy nikoliv logaritmickou, ale lineární složitost. Je proto, že složitost nalezení uzlu, který je seznamu na indexu `s`,[1] je lineární vzhledem k `s`. Nalézt prvek na indexu `s` nejde jinak, než navštívením všech uzlů na menších indexech, viz následující procedura.

```
element-at
⟵ r: Node — první uzel seznamu
⟵ n: Int — index hledaného uzlu
⟶ Node — nalezený uzel nebo nil
```

```
1  m ← 0
2  x ← r
3  while (x ≠ nil and m < n)
4     x ← x.next
5     m ← m + 1
6  return x
```

Složitost algoritmu půlení intervalů je tak dána rekurencí

$$T(n) = T(n/2) + O(n),$$
$$T(1) = 1,$$

jejím řešením je $\Theta(n)$. Vidíme tedy první významný rozdíl mezi seznamem a polem, u seznamu nemáme ke všem uzlům přístup v konstantním čase.

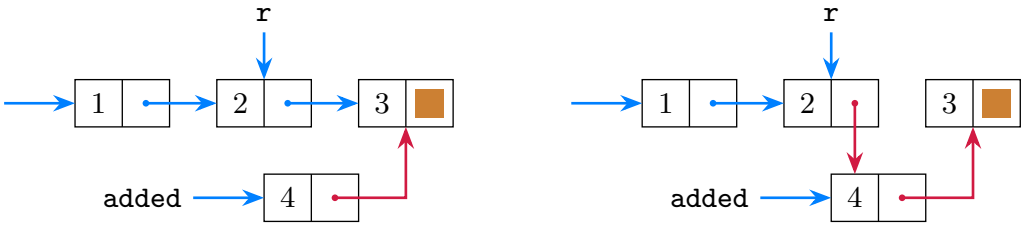
Výhodou seznamů (oproti polím) je možnost v konstatním čase vložit na nějaké místo seznamu nový uzel, pokud máme referencí na uzel předchozí.

```
insert-after
⟵ r: Node — uzel, za který vkládáme
⟵ added: Node — vkládaný uzel
```

```
1  added.next ← r.next
2  r.next ← added
```

Průběh operace lze vidět na následujícím obrázku. Pořadí přepojení referencí na řádcích `1` a `2` je důležité, kdybychom je provedli v opačném pořadí, dostali bychom cyklickou strukturu. (`Doporučuji čtenáři, abys si to na obrázku vyzkoušel.`)

[1] První uzel má index `0`, druhý uzel má index `1` atd., index `s` je v algoritmu půlení intervalů v polovině pole, zde je myšleno v polovině seznamu.

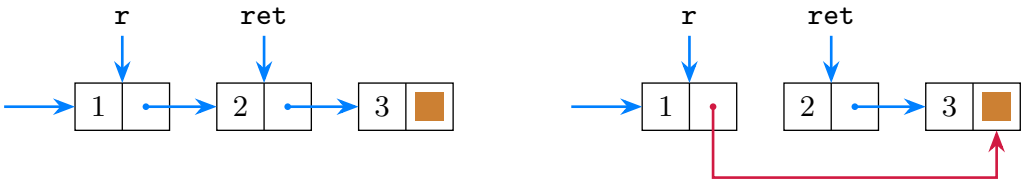


Podobně jako přidat uzel za uzel, na který máme referenci, můžeme také odebrat uzel, který se tam již nachází. Složitost této operace je také konstantní.

```
remove-after
⟵ r:node — uzel, jehož následníka odstraníme
⟶ node — odstraněný uzel
```

```
1  ret ← r.next
2  if (r.next ≠ nil) then r.next ← r.next.next
3  return ret
```

Na obrázku vypadá operace následovně



Operace vrací jako svůj výsledek odebíraný uzel. Na řádku 2 kontrolujeme, jesli náhodou není r na konci seznamu, pokud ano, odebírání uzlu za r nemá smysl a operace vrací nil.

Operace, které mohou změnit první prvek seznamu, musí jako návratovou hodnotu vracet nový začátek seznamu. Je to proto, abychom měli možnost si v případě změny tento nový začátek někam uložit (vzpomeňme si, že seznam si pamatujeme tak, že si pamatujeme jeho první uzel).

V následující pasáži čtenář nalezne operace pro vkládání a odebírání uzlů z nějakého místa seznamu. Pro tyto operace je typické, že změnu prvního uzlu provádí jako speciální případ. Změny v ostatních částech seznamu pak provádí nalezením uzlu před místem změny, a použitím operací insert-after nebo remove-after.

Operace insert-at vkládá nový uzel na specifický index v seznamu (za předpokladu, že index menší roven délce seznamu).

```
insert-at
⟵ r: Node — začátek seznamu
⟵ added: Node — přidaný uzel
⟵ n: Int — index, na kterém bude přidaný uzel
⟶ Node — začátek seznamu
```

```
if (n = 0)
   added.next ← r
   return added
x ← element-at(r, n - 1)
if (x ≠ nil) then insert-after(x,added)
return r
```

Operace `remove-at` odstraní uzel na daném indexu v seznamu, a vrátí jej.

```
remove-at
⟵ r: Node — začátek seznamu
⟵ n: Int — index odstraněného uzlu
⟶ Node — odstraněný uzel
⟶ Node — nový začátek seznamu
```

```
if (r = nil) then return nil, nil
if (n = 0) then return r, r.next
x ← element-at(r, n-1)
if (x ≠ nil) then x ← remove-after(x)
return x, r
```

Operace `remove` odstraní daný uzel ze seznamu. Předpokládá, že se uzel v seznamu nachází.

```
remove
⟵ r: Node — začátek seznamu
⟵ d: Node — odstraňovaný uzel, víme že je v seznamu
⟶ Node — nový začátek seznamu
```

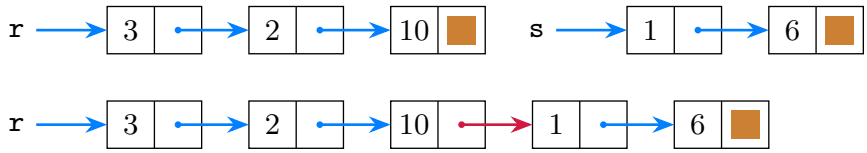
```
if (d = r) then return r.next
x ← r
while x.next ≠ d do x ← x.next
remove-after(x)
return r
```

Složitost všech operací výše je lineární vzhledem k maximu z vkládaného indexu a délky upravovaného seznamu.

Poslední operace spojí dva seznamy, její složitost je lineární vhledem k délce prvního seznamu.

```
concatenate
⟵ r: Node — první uzel prvního seznamu
⟵ s: Node — první uzel druhého seznamu
⟶ Node — první uzel seznamu vzniklého spojením r a s
```

```
1  if (r = nil) then return s
2  if (s = nil) then return r
3  x ← r
4  while (x ≠ nil) do x ← x.next
5  x.next ← s
6  return r
```



3.1 TŘÍDICÍ ALGORITMY NA SEZNAMECH

Pole i seznamy uchovávají posloupnosti prvků, algoritmické myšlenky tak lze často mezi oběma strukturami přenášet, i když jsou nutné technické adaptace. Jako ukázku tohoto faktu obsahuje tato kapitola třídicí algoritmy InsertionSort a QuickSort pro seznamy.

Není obtížné si představit, že v seznamu uchováváme uzly uspořádány vzestupně podle podle položky `id` a že máme k dispozici proceduru s lineární složitostí, která přidá uzel do uspořádaného seznamu na správné místo tak, aby zůstal uspořádaný.

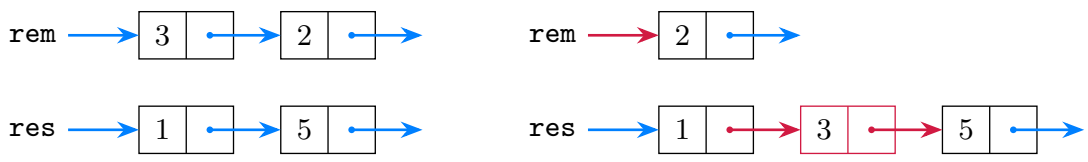
```
ordered-insert
⟵ r: Node — první uzel seznamu
⟵ added: Node — přidaný uzel
⟶ Node — první uzel seznamu po přidání
```

Algoritmus InsertionSort realizuje následující procedura.

```
insert-sort
⟵ r: Node — první uzel seznamu
⟶ Node — první uzel setříženého seznamu
```

```
1  res ← nil
2  rem ← r
3  while (rem ≠ nil)
4     x ← rem
5     rem ← rem.next
6     res ← ordered-insert(res, x)
7  return res
```

V seznamu `rem` operace udržuje doposud nesetříděnou část seznamu, na začátku je to celý seznam, setřízená část je `res`, na začátku rovna `nil`. V obecném kroku cyklu na řádcích `4`--`6` vždy odebereme první uzel `rem` a vložíme jej na správné místo do `res`. Jedna iterace cyklu je zobrazena na následujícím obrázku.



Složitost algoritmu je kvadratická, v nejhorším případě při vkládání do `res` projdeme tento seznam celý.

Klíčovou procedurou algoritmu Quicksort je `partition`. Jejím úkolem je vybrat jeden uzel, tzv. *pivot*, a rozdělit seznam na dva seznamy: první z nich s klíči menšími než pivot, a druhý z nich z klíči většími než pivot.

```
partition
 ⟵ r: Node — začátek děleného seznamu (předpokládáme, že je neprázdný)
 ⟶ Node — uzel s pivotem
 ⟶ Node — začátek seznamu s klíči menšími než pivot
 ⟶ Node — začátek seznamu s klíči většími než pivot
```

```
1  pivot ← r
2  rem ← r.next
3  left ← nil
4  right ← nil
5  while (rem ≠ nil)
6     x ← rem
7     rem ← rem.next
8  if (x.id ⩽ pivot.id) then left ← insert-at(left, x, 0)
9  else right ← insert-at(right, x, 0)
10 return pivot, left, right
```

Jako pivota volí procedura první uzel seznamu. V seznamu `rem` uchovává doposud nezpracované uzly, na začátku jsou to všechny uzly mimo pivota. V každé iteraci cyklu na řádcích `5--9` odebereme první prvek `rem`, porovnáme jeho klíč s klíčem uzlu `pivot` a podle výsledku jej vložíme do na začátek `left` (má-li menší klíč), nebo na začátek `right` (má-li větší klíč). Po skončení cyklu tak `left` obsahuje uzly s klíči menšími než `pivot` a `right` uzly s klíči většími než `pivot`. Operace má lineární složitost, musí projít celý vstupní seznam a přitom zpracování jednoho uzlu tohoto seznamu trvá konstantní čas.

Samotný algoritmus je potom realizován procedurou `quicksort`, která zavoláním procedury `partition` rozdělí vstupní seznam, poté rekurzivně setřídí vzniklé podseznamy, a získané výsledky zapojí spolu s pivotem do jednoho seznamu.

```
quicksort
 ⟵ r: Node — začátek seznamu k setřídění
 ⟶ Node — začátek setřízeného seznamu
```

```
1  if (r = nil or r.next = nil) then return r
2  pivot, left, right ← partition(r)
3  left ← quicksort(left)
4  right ← quicksort(right)
5  right ← insert-at(right, pivot, 0)
6  return concatenate(left, right)
```

Složitost `quicksort` je v nejhorším případě kvadratická, pokud je totiž `pivot` v proceduře `partition` opakovaně největším nebo nejmenším prvek rozdělovaného seznamu, je složist dána rekurencí

$$T(n) = T(n-1) + O(n),$$

jejím řešením je $\Theta(n^2)$. Pokud je `pivot` mediánem v rozdělovaném seznamu, je složitost dána rekurencí

$$T(n) = 2T(n/2) + O(n),$$

jejímž řešením je $\theta(n \log n)$.

3.2 DALŠÍ VARIANTY SEZNAMŮ

V této kapitolu naznačíme některé varianty seznamů, které nějakým způsobem zjemňují nepříjemné vlastnosti obyčejného seznamu. K jednotlivým variantám neuvádíme všechny operace, pro ilustraci principu uvedeme pouze některé. Od čtenáře se očekává, že si podle toho ostatní operace vytvoří sám jako cvičení.

Potíže, které způsobuje u některých operací změna prvního uzlu v seznamu (je to speciální případ, procedury musí vracet začátek seznamu) můžeme vyřešit použitím *zarážky*. Je to uzel, který je na technicky na začátku seznamu, ale ignorujeme jeho obsah. Za skutečný seznam, který obsahuje data, bere až seznam začínající druhým uzlem. Výhodou je, že odpadnou výše zmiňované obtíže se začátkem seznamu, nevýhodou je, že ztratíme flexibilitu při práci s podseznamy: referenci na libovolný uzel v seznamu teď nemůžeme brát jako seznam, chybí zarážka. Čtenář jistě snadno upraví existující operace tak, aby fungovali pro seznamy se zarážkou. Jako příklad uveďme proceduru pro přidání, ze které vypadl speciální případ pro přidávání na začátek seznamu, a která nemusí vracet uzel na začátku seznamu.

```
sentinel-insert-at
⟵ r: Node — zarážka
⟵ added: Node — přidaný uzel
⟵ n: Int — index přidaného uzlu
```

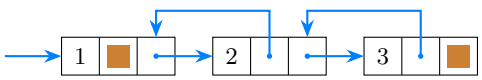
```
1  x ← element-at(r, n)
2  if (x ≠ nil) insert-after(x,added)
```

(Procedury `element-at` a `insert-after` pracují se seznamy bez zarážky.)

Druhou variantou je *obousměrný seznam*. Ke struktuře pro uzel přidáme i položku pro referenci na předchozí uzel.

```
struct Node
   id: Key
   next: Node — další uzel v seznamu
   prev: Node — předchozí uzel v seznamu
```

Příklad obousměrného seznamu je na následujícím obrázku.



Seznam teď navíc můžeme procházet pomocí `prev` i směrem od konce na začátek. To je výhodné, protože k daném uzlu snadno najdeme předchozí uzel, který potřebujeme pro `insert-after` a `delete-after`. Tyto procedury ovšem musíme upravit tak, aby pracovali i s položkou `prev`. Níže vidíme příklad úpravy pro `delete-after`.

```
doubly-linked-remove-after
⟵ r: Node — uzel, za kterým mažeme
⟶ Node — mazaný uzel
```

```
1  ret ← ret.next
2  if (r.next ≠ nil)
3     r.next ← r.next.next
4     if (r.next ≠ nil)
5        r.next.prev ← r
6  return ret
```

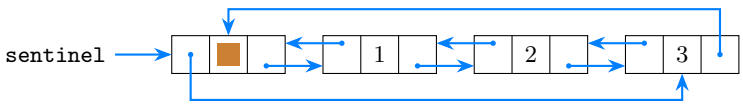
Při odstranění konkrétní uzlu nepotřebujeme nalézt uzel mu předcházející procházením od začátku seznamu, můžeme použít položku `prev`.

```
doubly-linked-remove
⟵ r: Node — první uzel seznamu
⟵ del: Node — mazaný uzel
⟶ Node — první uzel seznamu po smazání
```

```
1  p ← del.prev
2  if (p = nil)
3     ret ← del.next
4     if (ret ≠ nil)
5        ret.prev ← nil
6  else
7     doubly-linked-remove-after(p)
8     return r
```

Na řádcích `2` -- `5` ošetřujeme případ, kdy odebíráme první prvek seznamu. Povšimněme-li speciálně řádků `4` a `5`, ve kterých nastavujeme novému začátku seznamu položku `prev` na `nil`, předtím ovšem musíme zkontrolovat, jestli samotný nový začátek není `nil` (to se stane, pokud odebíráme z jednoprvkového seznamu).

Nutnosti mít speciální případ pro začátek seznamu se můžeme zbavit pomocí zarážky na konci seznamu. Pokud chceme odstranit speciální případ při odebírání prvku na konci seznamu (řádek `4` v proceduře `doubly-linked-remove-after`, řádek `4` v proceduře `doubly-linked-remove`) můžeme zarážku umístit i na konec seznamu. Nic pak dokonce nebrání tomu, abychom na začátku a konci seznamu použili jako zarážku stejný uzel. Dostaneme tak cyklický seznam: seznam, ve kterém je za poslední uzel zapojen uzel první[2], viz následující obrázek.



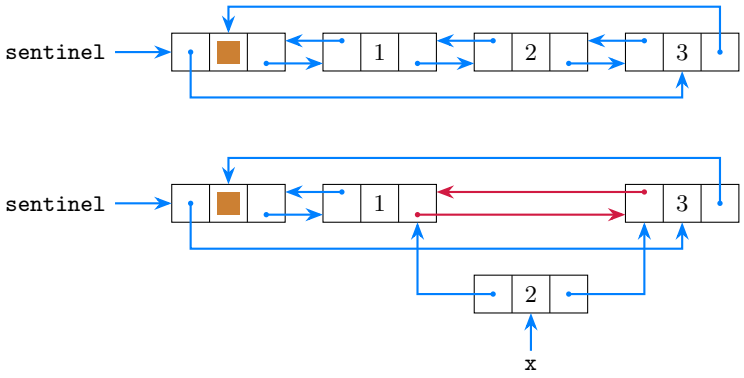
[2] V našem případě je v seznamu umístěn sentinel a seznam je obousměrný. Cyklický ovšem může být i seznam bez sentinelu nebo seznam jednosměrný.

Operace odebrání prvku je v cyklickém obousměrném seznamu se zarážkou velice jednoduchá, nemusíme kontrolovat žádné speciální případy, pro operaci odebrání vrcholu dokonce nepotřebujeme znát ani první vrchol seznamu.

```
cyclic-sentinel-double-linked-remove
⟵ r: Node ⟶ odebíraný uzel, který není sentinelem
```

```
1  r.prev.next ← r.next
2  r.next.prev ← r.prev
```

Příklad použití operace vidíme na následujícím obrázku.



ZÁSOBNÍK A FRONTA

Zásobník je abstraktní datová struktura s operacemi

- `push` pro vkládání a
- `pop` pro odebírání.

Dalším požadavkem je, že datové položky jsou ze struktury odebírány v opačném pořadí, než ve kterém byly vloženy. Zásobník lze implementovat mnoha způsoby. Lze použít dynamické pole, kde operaci `push` provádíme pomocí `append`, a operaci `pop` pomocí stejně pojmenované operace pro dynamická pole. Další možností je zásobník implementovat pomocí seznamu, kde `push` děláme vkládáním na index 0 pomocí `insert-at`, a `pop` odebíráním z indexu 0 pomocí `remove-at`.

Dalšími operacemi, které zásobník obvykle poskytuje je zjištění velikosti zásobníku, tj. počtu datových položek, které obsahuje, případně test prázdnosti.

Zásobník je v informatice používán na mnoha místech, pro ukázku si uvedeme algoritmus vyhodnocující aritmetické výrazy. Za dobře uzávorkovaný výraz považujeme `(op1 operator op2)`, kde `op1` a `op2` jsou také dobře uzávorkované výrazy a `operator` je aritmetický operátor. Správně uzávorkovaným výrazem je také číslo. Příkladem správně uzávorkovaného výrazu je `(1 + ((2 + 3) * 4))`. Algoritmus pro vyhodnocení takového výrazu používá zásobník `Q1` pro operátory a zásobník `Q2` pro čísla. Vyhodnocovaný výraz čteme zleva doprava,

- narazíme-li na číslo, vložíme jej do zásobníku `Q2`
- narazíme-li na operátor, vložíme jej do zásobníku `Q1`
- narazíme-li na `)`, odebere z `Q1` operátor a z `Q2` dva operandy, na ně operátor aplikujeme a výsledek vložíme do `Q2` (u nekomutativních operátorů si tady musíme dávat pozor na to, že jejich argumenty jsme ze zásobníku odebrali v opačném pořadí, než ve kterém jsme je vložili).

Po přečtení výrazu obsahuje `Q2` výsledek.

Fronta je abstraktní datová struktura s operacemi

- `enqueue` pro vkládání a
- `dequeue` pro odebrání.

Dalším požadavkem je, že datové položky odebíráme ve stejném pořadí, ve kterém jsme je vložili.

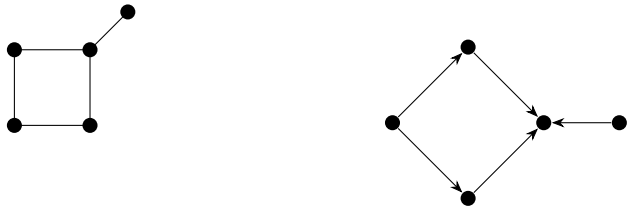
Frontu lze implementovat implementovat pomocí (dynamického) cyklického bufferu: `enqueue` děláme pomocí `write`, a `dequeue` pomocí `read`. Případně můžeme použít obousměrného seznamu se sentinelem, `enqueu` přidává prvek za sentinel, `dequeue` odebírá prvek před sentinelem. (Je samozřejmě možná i implementace pomocí jiného typu seznamu, ta je ovšem komplikovanější.)

Vyhledávací stromy jsou datové struktury, na které se můžeme dívat jako na stromy, jak je známe z diskrétní matematiky: jako na speciálný typ grafu. Tento pohled je užitečný, poskytne nám potřebné pojmy pro popis vyhledávacích struktur i nástroje pro formální analýzu operací.

5.1 STROM JAKO SPECIÁLNÍ TYP GRAFU

Připomeňme tedy, že graf G je dán konečnou množinou vrcholů V a množinou hran H. Toto značíme $G = (V, H)$. Hrana je dvojice vrcholů, přičemž u orientovaných grafů je tato dvojice uspořádaná: záleží na pořadí, ve kterém jsou vrcholy ve dvojici uvedeny. U neorientovaných grafů na jejich pořadí nezáleží. Abychom si zjednodušili život, budeme v textu v obou případech používat pro dvojici vrcholů u, v používat značení (u, v), u neorientovaných grafů si budeme pamatovat, že na pořadí nezáleží (a že (u, v) a (v, u) značí tutéž dvojici).

Graf můžeme namalovat tak, že vrcholy namalujeme jako kolečka a hrany jako čáry, které vrcholy spojují. U orientovaných grafů navíc směr (z prvního vrcholu v hraně do druhého) vyznačíme šipkou. Na následujícím obrázku je nalevo neorientovaný a napravo orientovaný graf.



Cesta z uzlu s do uzlu t je posloupnost po dvojicích různých vrcholů

$$s = v_1, v_2, \ldots, v_n = t$$

taková, že pro všechna $1 \leqslant i < n$ je dvojice (v_i, v_{i+1}) hranou v grafu. O každém vrcholu v posloupnosti a o každé hraně (v_i, v_{i+1}) řekneme, že na cestě leží. Délka cesty rovna počtu hran, které na ní leží. To je rovno počtu vrcholů, které na ní, leží zmenšenému o jedna. V grafu existuje *kružnice*, pokud v něm existuje cesta z s a do t, a přitom je v grafu hrana (t, s), která na této cestě neleží. Graf na předchozím obrázku vlevo kružnici obsahuje, vpravo kružnici neobsahuje. O vrcholech a hranách ležících na cestě z s do t a o hraně (t, s) řekneme, že leží na daném cyklu.

Řekneme, že neorientovaný graf je *souvislý*, pokud mezi libovolnými dvěma jeho vrcholy existuje cesta. *Strom* je neorientovaný graf, který je souvislý a neobsahuje kružnici.

Věta 5.1

Pro neorientovaný graf $G = (V, H)$ jsou následující tvrzení ekvivalentní

(a) G je strom.

(b) Mezi každými dvěma různými vrcholy G existuje právě jedna cesta.

(c) G neobsahuje kružnici a $|V| = |E| + 1$.

(d) G je souvislý a $|V| = |E| + 1$.

(e) G je souvislý, vynecháním jakékoliv hrany vznikne nesouvislý graf.

(f) G neobsahuje kružnice, ale přidáním jakékoliv hrany vznikne graf s kružnicí.

Důkaz. Jako příklad dokážeme ekvivalenci mezi (a) a (b). Zbytek byl probrán v kurzu diskrétních struktur.
Z toho, že G je souvislý plyne, že mezi libovolnými dvěma různými vrcholy existuje alespoň jedna cesta. Musíme tak ukázat, že G neobsahuje kružnici, právě když mezi libovolnými dvěma různými vrcholy existuje nejvýše jedna cesta.

Předpokládejme, že G obsahuje kružnici. Vybereme dva různé vrcholy x, y ležící na této kruznici hned za sebou. Mezi těmito vrcholy existují nejméně dvě cesty: jedna je tvořena hranou (x, y) a druhá vede z uzlu x přes právě přes vrcholy ležící na uvažované kružnici do uzlu y. Dokázali jsme tak, že pokud mezi libovolnými dvěma vrcholy existuje nejvýše jedna cesta, pak v grafu není kružnice.

Zbývá dokázat opačnou implikaci. Uvažme dvojici různých vrcholů u, v a dvě cesty mezi těmito uzly

$$u = x_0, x_1, \ldots, x_m = v$$
$$u = y_0, y_1, \ldots, y_n = v$$

Najdeme nejmenší index i tak, že $x_i \neq y_i$. Jistě platí $\min(m, n) > i > 0$. Položíme $s = i - 1$ a najdeme nejmenší $l > i$ tak, že platí jedna z následujících podmínek

$$x_l = y_j \text{ pro nějaké } j > s \tag{5.1}$$
$$x_j = y_l \text{ pro nějaké } j > s \tag{5.2}$$

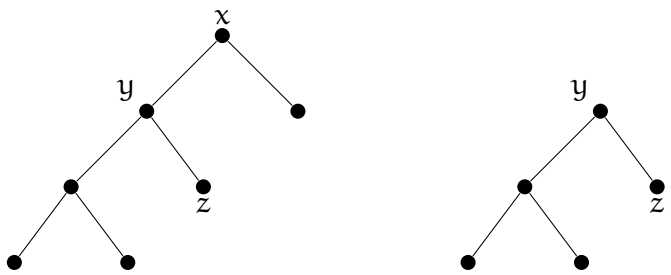
Požadované l určitě existuje, protože $x_m = y_n$ a předchozí podmínky platí pro $l = m$ a $j = n$. Pokud jsme našli l podle (9.1), pak je hledaná kružnice $x_s \ldots, x_l = y_j, \ldots, y_s = x_s$. Pro případ (5.2) je kružnicí $x\ x_s, \ldots, x_j = y_l, \ldots, y_s = x_s$. □

Podgrafem grafu $G = (V, H)$ indukovaném množinou vrcholů V' je graf $G' = (V', H')$, kde

$$H = \{(x, y) \in H \mid x, y \in V'\}$$

Graf G' tedy obsahuje všechny vrcholy z množiny V' a všechny hrany grafu G, jejichž oba vrcholy se nachází ve V'. Na následujícím obrázku je vpravo podgraf (grafu vlevo) indukovaný množinou $\{a, b, c\}$.

Kořenový strom je strom, ve kterém vybereme jeden vrchol a označíme jej za *kořen*. Ve zbytku textu budeme uvažovat pouze kořenové stromy, proto pod pojmem strom budeme myslet kořenový strom (který konkrétní vrchol je kořenem bude vždy jasné z kontextu, nebo to nebude důležité). Budeme uvažovat cesty z kořene do ostatních vrcholů a tím si zavedeme ve stromu směr dolů, který odpovídá směru, kterým vedou cesty z kořene.Vrchol x je *následníkem* vrcholu y, pokud $x \neq y$ a y leží na cestě z kořene do x. Vrcholu y pak říkáme *předchůdce* x. Pokud je délka cesty z y do x rovna 1, pak je x *potomkem* y, a y je *rodičem* x. Vrchol z je *list*, pokud nemá žádné potomky. *Podstrom s kořenem* x je podgraf indukovaný množinou vrcholů $\{x\} \cup \{y \mid y \text{ je následník } x\}$. Je nutné si uvědomit, že podgraf stromu, který je stromem a jehož kořenem je x, nemusí být nutně podstromem s kořenem x. U podstromu s kořenem x totiž vyžadujeme, aby obsahoval všechny následníky x. Podstrom s kořenem x označíme jako podstrom vrcholu y, pokud je x potomkem y. Na následujícím obrázku je ve stromu nalevo y potomkem x; z je následníkem x, ale není jeho potomkem; z je list, x je kořen, y není ani list ani kořen. Napravo je podstrom s kořenem y, který je podstromem vrcholu x.



Hloubka vrcholu x je délka cesty z kořene do x. *Výška* stromu je rovna maximu z hloubek jeho vrcholů. Množinu uzlů majících hloubku i budeme označovat za i-tou úroveň stromu. Následující tvrzení plyne přímo z definic a přechozí věty.

Věta 5.2

Pro libovolný vrchol x ve stromu platí:

(a) Počet předchůdců x je roven jeho hloubce.

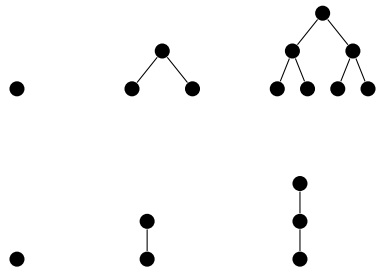
(b) Různé podstromu x obsahují disjuktní množiny vrcholů.

(c) Výška podstromu s kořenem x je rovna maximu výšek podstromů vrcholu x, ke kterému přičteme jedna.

Strom je m-*ární* (má aritu m), pokud je počet potomků libovolného vrcholu nejvýše m. Pro 2-ární stromy máme speciální jméno, říkáme jim *binární*. Arita stromu omezuje počet potomků seshora, nikoli zespodu. V m-ární stromu tak mohou existovat vrcholy s méně než m potomky. (Např. listy mají 0 potomků; pokud bychom vyžadovali, aby měl každý uzel m potomků, museli bychom připustit, že obsahuje nekonečno uzlů.)

Z pohledu složitosti algoritmů pracujících se stromy (které budeme dělat ve zbytku kapitoly) je důležitý vztah mezi počtem vrcholů stromu a jeho výškou. Pro následující úvahu se pro jednoduchost omezíme na binární stromy.

Zafixujeme hloubku h a pokusíme se najít stromy této výšky s nejméně a nejvíce vrcholy. Pro $h = 0, 1, 2$ vidíme na prvním řádku následujího obrázku stromy s maximálním počtem vrcholů, na druhém řádku stromy s minimálním počtem vrcholů.



Můžeme si všimnout vzoru. Pokud chceme nalézt strom výšky h s minimálním počtem vrcholů, každý vrcholu musí mít minimální povolený počet potomků: mimo listů mají všechny vrcholy jednoho potomka. V každé úrovni stromu je tak jeden vrchol. Výška stromu je tak rovna počtu vrcholů mínus jedna. Pokud naopak chceme dosáhnout maximálního počtu vrcholů, musí mít všechny vrcholy mimo listů dva potomky, a přitom jsou všechny listy až v úrovni h. Listu z úrovně menší než h můžeme totiž přidat dva potomky a dostaneme tak strom výšky h s více vrcholy. Ve stromu je tedy v každé úrovni maximální možný počet vrcholů: V úrovni nula 1 vrchol, v úrovni jedna 2 vrcholy (protože kořen má dva potomky), v úrovni dva 4 vrcholy (protože každý vrchol v úrovni jedna má dva potomky). Obecně je tedy v úrovni $i+1$ dvojnásobek počtu vrcholů v úrovni i. V úrovni i je tak 2^i vrcholů. Ve stromu výšky h je potom maximálně

$$\sum_{i=0}^{h} 2^i = 2^{h+1} - 1$$

vrcholů. Označíme počet vrcholů n, platí tak

$$h + 1 \leqslant n \leqslant 2^{h+1} - 1$$

a odtud dostaneme

$$h \leqslant n - 1, \tag{5.3}$$
$$\ln(n) - 1 < h, \tag{5.4}$$

Druhou nerovnost ovšem můžeme zpřesnit. Situace, kdy je výška vzhledem k počtu uzlů minimální, odpovídá situaci, kdy je počet vcholů vzhledem k výšce maximální s tím, že v poslední vrstvě nemusí být maximální počet vrcholů. Máme tedy

$$2^h \leqslant n < 2^{h+1},$$
$$h \leqslant \ln(n) < h + 1$$

v důsledku čehož máme

$$h = \lfloor \ln(n) \rfloor.$$

Předchozí úvahy shrneme do následující věty.

Věta 5.3: O výšce binárního stromu

V binárním stromu výšky h obsahujícím n vrcholů platí

$$h + 1 \leqslant n \leqslant 2^{h+1} - 1, \tag{5.5}$$
$$\lfloor \ln(n) \rfloor \leqslant h \leqslant n - 1. \tag{5.6}$$

5.2 BINÁRNÍ VYHLEDÁVACÍ STROM

S pomocí následující struktury můžeme vytvořit binární strom tak, že si pamatujeme instanci této strukturu představující kořen stromu, v položkách `left` a `right` si pamatujeme kořeny levého a pravého podstromu. Navíc, potomci si udržují v položce `parent` referenci na svého rodiče.

```
struct Node
   id: Key — klíč
   left: Node — levý potomek (nil, pokud neexistuje)
   right: Node — pravý potomek (nil, pokud neexistuje)
   parent: Node — rodič (nil, pokud neexistuje)
```

Pokud se na takto vytvořené instance struktury propojené pomocí `left` a `right` podíváme jako na graf (jako v první kapitole), je to binární strom.

V položce `id` uchováváme ve stromu klíče. Pokud jsou klíče do vrcholů rozmístěny tak, že následující procedura pro nalezení vrcholu s konkrétním klíčem funguje, budeme hovořit o `vyhledávacím stromu`.

```
tree-search-rec
⟵ k: Key — hledaný klíč
⟵ x: Node — kořen prohledávaného podstromu
⟶ Node — vrchol obsahující nalezený klíč nebo nil
```

```
1  return { x                              pokud (x == nil || x.id == k)
         { tree-search-rec(k, x.left)     pokud (x.id > k)
         { tree-search-rec(k, x.right)    pokud (x.id < k)
```

Upřesněme, co myslíme korektností algoritmu: pro libovolný klíč `k` v případě, že se vrchol s klíčem `k` ve stromu nachází, vrátí procedura právě tento vrchol. V opačném případě vrátí `nil`.

Čtenář snadno nahlédne, že výše uvedená podmínka je ekvivalentní tzv. *podmínce uspořádání:* pro každou dvojici různých vrcholů `x, y` platí, že

- pokud je vrchol `y` v levém podstromu `x`, pak `y.id < x.id`,
- pokud je `y` v pravém podstromu `x`, pak `y.id > x.id`.

Binární vyhledávací strom je přirozeně rekurzivní struktura, a je přirozené používat pro práci s nimi rekurzivní procedury. V textu budeme střídat procedury rekurzivní a nerekurzivní, podle toho, která z nich lépe ilustruje danou problematiku, případně je snažší ji analyzovat. Uveďme si tedy i imperativní verzi procedury vyhledávání ve stromu.

```
tree-search
⟵ k: Key – hledaný klíč
⟵ x: Node – kořen prohledávaného podstromu
⟶ Node – vrchol obsahující nalezený klíč
```

```
1  y ← x
2  while (y ≠ nil)
3     if (k = y.id) then return y
4     y ← { y.left    pokud (k < y.id)
          { y.right   pokud (k > y.id)
5  return nil
```

Jaká je složitost vyhledávání? Po každé iteraci cyklu na řádcích 2 -- 4 se hloubka vrcholu y zvětší o jedna. Počet provedení cyklu je tak v nehorším případě roven výšce stromu. Uvnitř cyklu provádíme konstantní počet operací, složitost vyhledávání je tedy v nejhorším případě linární vzhledem k výšce stromu. Pokud se klíč nachází v kořeni, je složitost konstantní a na výšce stromu vůbec nezáleží (to je nejlepší případ). Chceme-li složitost vyhledávání vyjádřit jako závislou na počtu vrcholů stromu, zjistíme že musíme brát v úvahu i tvar stromu (který určuje jeho výšku). Podle Věty 5.3 potom dostaneme následující tabulku složitostí.

strom / klíč	nejlepší	nejhorší
nejlepší	$\Theta(1)$	$\Theta(\lg n)$
nejhorší	$\Theta(1)$	$\Theta(n)$

Pomocí průchodu binárního stromu můžeme navštívit všechny jeho vrcholy. Akt navštívení je v konkrétním použití průchodu vždy nějakou akcí, kterou s vrcholem provedeme, můžeme například vytisknout klíč, který je ve vrcholu obsažen.

Typ průchodu, který si ukážeme, je *průchod do hloubky* (více podrobností o průchodech grafů je v kapitole o grafových algoritmech). Tento průchod je snadné popsat pomocí rekurzivní procedury, která pro vstupní vrchol x v nějakém pořadí provede následující: navštíví x, rekurzivně projde x.left, rekurzivně projde x.right. Základní tři možnosti, ve kterých je vždy projití x.left před projitím x.right, jsou *preorder*, *inorder* a *postorder* průchod, ve kterých je navštívení x první, druhou, respektive třetí. Pro ukázkou je níže kód průchodu *inorder*.

```
in-order-walk
⟵ x: Node
```

```
1  if (x == nil) then return
2  in-order-walk(x.left)
3  navštiv x
4  in-order-walk(x.right)
```

Věta 5.4

Procedura in-order-walk navštíví vrcholy ve vzestupném pořadí (podle hodnot klíčů). Složitost procedury je lineární vzhledem k počtu vrcholů.

Důkaz. Procedura navštíví každý vrchol ve stromu s kořenem x právě jednou: podstromy s kořenyx.left a x.right totiž obsahují disjunktní množiny vrcholů. (Formálně by se důkaz vedl indukcí podle výšky stromu.) Odtud plyne i lineární složitost průchodu.

Vrcholy navštívíme ve vzestupném pořadí, protože podle podmínky uspořádání jsou všechny klíče v podstromu s kořenem `x.left`, menší než `x.id` a ten je menší než klíče v podstromu s kořenem `x.right`. □

Představme si následující úkol. Dostaneme pole obsahující n vrcholů stromu (s různými klíči) a máme najít algoritmus, který z nich sestaví binární vyhledávací strom. Jaká může být složitost takového algoritmu? Existence procedury `in-order-walk` a výsledek, který znáte z minulého semestru, totiž

> *Každý algoritmus třídění porovnáváním potřebuje k setřídění n-prvkového pole nejméně $\Omega(n \log n)$ porovnání,*

nás vedou k tomu, že složitost našeho algoritmu je $\Omega(n \log n)$. Kdyby byla tato složitost $o(n \log n)$, potom byl algoritmus

```
1  vytvoř ze vstupního pole binární vyhledávací strom T
2  projdi T pomocí in-order-walk, při návštěvě vrcholy postupně vkládej do pole.
```

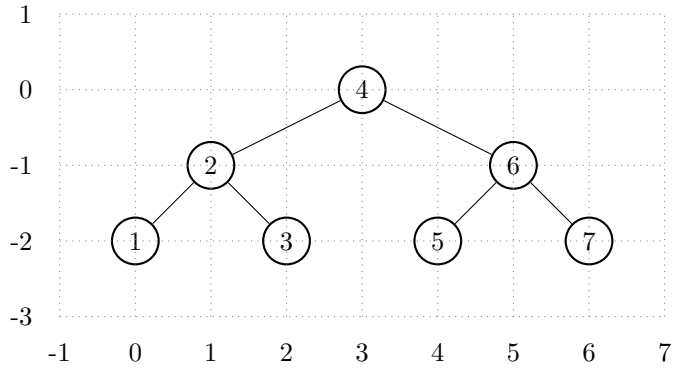
byl třídicím algoritmem, který provede $o(n \log n)$ porovnání. To je ale spor.

Jako druhou aplikaci `in-order-walk` si ukážeme její drobnou modifikaci, která vede k jednoduchému algoritmu kreslení stromu. Samotné kreslení provedeme přiřazením souřadnic na dvojrozměrné ploše každému vrcholu ve stromu. Do struktury pro vrchol přidáme položky `x,y` pro souřadnice. Následující procedura do těchto vrcholů spočítá souřadnice (a strom pak můžeme nakreslit).

```
node-coordinate
⟵ r: Node — kořen stromu (předp. r ≠ nil)
⟵ y: Int — vertikální souřadnice vrcholu
⟵ x: Int — horizontální souřadnice dosud nejpravějšího vrcholu
⟶ Int — horizontální souřadnice nejpravějšího vrcholu v podstromu s kořenem r
```

```
1  r.y ← y
2  ret ← x
3  if (r.left ≠ nil) then ret ← node-coordinates(r.left, y-1, x)
4  ret ← ret + 1
5  r.x ← ret
6  if (r.right ≠ nil) then ret ← node-coordinates(r.right, y-1, ret)
6  return ret
```

Proberme si volání procedury, kde prvním argumentem je kořen stromu a pro další argumenty máme `y=0, x=-1`. V každém dalším (rekurznivním) volání procedury je `y` hloubkou `r` vynásobenou -1, a `x` je počet vrcholů, které již byli průchodem navštíveny zmenšený o jedna. Protože se jedná o inorder průchod, můžeme `x` považovat za horizontální souřadnici doposud nejpravějšího projitého vrcholu. Procedura vrací tuto hodnotu upravenu tak, že je to horizontální souřadnice nejpravějšího vrcholu v celém stromu s kořenem `r`. Dosáhne toho tak, že projde levý podstrom, odtud získá souřadnici nejpravějšího vrcholu z levého podstromu, tu o jedna zvětší a má tak horizontální souřadnici `r`. Nakonec projde pravý podstrom a horizontální souřadnici nejpravějšího vrcholu v tomto podstromu vrátí jako výsledek. Na následujícím obrázku je příklad stromu, který byl namalován předchozím algorimem.



Z podmínky uspořádání plyne, že vrchol, který obsahuje nejmenší klíč, nemá levý podstrom a sám se nachází v levém podstromu všech svých předchůdců. Ve stromu jej tedy můžeme najít tak, že z kořene budeme opakovaně přecházet do levého podstromu, skončíme v momentě, kdy je levý podstrom `nil`.

```
tree-min
⟵ r: Node — kořen stromu (předp. r ≠ nil)
⟶ Node — vrchol s nejmenším klíčem
```

```
1  y ← r
2  while (y.left ≠ nil)
3     y ← y.left
3  return y
```

Algoritmus má složitost, která je lineární vzhledem k výšce stromu. Vrchol s minimálním klíčem může totiž mít maximální hloubku (mezi vrcholy ve stromu). Nalezení maximálního klíče se dělá analogicky.

Pořádkový následník vrcholu `x` je vrchol, který má v množině

$$\{\texttt{z} \mid \texttt{z.id} > \texttt{x.id}\}$$

nejmenší klíč. Pokud je tato množina prázdná, pořádkový následník neexistuje.

Věta 5.5: Kde je pořádkový následník?

(a) Pokud má `x` neprázdný pravý podstrom, je jeho pořádkový následník minimálním prvkem tohoto podstromu.

(b) Pokud má `x` prázdný pravý podstrom a jeho pořádkový následník `y` existuje, pak je nejhlubším předkem vrcholu `x` takovým, že `y.left` je buď také předkem `x` nebo přímo je přímo `x`.

Důkaz. Předpokládejme, že existuje vrchol `y`, který je pořádkovým následníkem `x` a nenachází se v pravém podstromu `x`. Pak existuje předek `p` vrcholu `x` tak, že `p` je přímo `y`,

nebo je `y` v pravém podstromu `p`. (Z podmínky uspořádání plyne, že `x` je v levém podstromu `p`.) Přitom musí platit první možnost: pokud by `y` byl v pravém podstromu `p`, pak bychom měli `x.id < p.id < y.id`, a `y` by nebyl pořádkovým následníkem `x`.

(a) Pokud je pravý podstrom `x` neprázdný, jsou vrcholy pravého podstromu `x` v levém podstromu `y`. Pro každý vrchol `w` v pravém podstromu `x` tak máme `x.id < w.id < y.id`. To je spor s tím, že `y` je pořádkový následník `x`.

(b) Předpokládejme, že existují dva různí předci `r` a `s` vrcholu `x` tak, že `r.left` i `s.left` jsou předci `x` nebo (jeden z nich je) přímo `x`. Pokud je vrchol `r` je ve větší hloubce než `s`, pak je `r` v levém podstromu vrcholu `s`, a tedy `x.id < r.id < s.id` a vrchol `s` není pořádkovým následníkem `x`. Odtud plyne maximální hloubka. □

Pořádkového následníka vrcholu tak můžeme najít pomocí následující procedury.

```
tree-successor
⟵ r: Node — vrchol, jehož pořádkového následníka hledáme
⟶ Node — pořádkový následník nebo nil
```

```
1  if (r.right ≠ nil) then return tree-min(r.right)
2  x ← r
3  while (x.parent ≠ nil)
4     y ← x.parent
5     if (x = y.left) then return y
6     x ← y
7  return nil
```

Složitost je lineární vzhledek k výšce stromu, protože strom projdeme nejhůře od `r` do listu, nebo od `r` do kořene. V obou případech je délka cesty omezena výškou stromu.

Duální pojem je pořádkový předchůdce, operace jeho nalezení a důkaz její správnosti jsou analogické

Protože operace, které mění strom, mohou přirozeně změnit i to, který vrchol je kořenem stromu. Protože si přístup ke stromu uchováváme tak, že si pamatujeme jeho kořen, musí buď všechny procedury měnící strom vracet kořen stromu po změně, nebo si vytvoříme analogii zarážky u seznamů. V této kapitole budeme oba přístupy střídat tak, aby vysvětlovaná látka byla co nejjasnější, a také aby čtenář viděl oba přístupy.[1]. Jako analogii zarážky budeme používat následující strukturu.

```
struct tree
   root:node — kořen stromu
```

Po (korektní) operaci vložení nového vrcholu do vyhledávací stromu musí strom zůstat vyhledávacím, tj. procedura `tree-search` musí korektní. Operaci vložení vrcholu `x` tak můžeme provést šalamounsky tak, že spustíme vyhledávání klíče `x.id`. To nutně skončí neúspěchem. Nicméně najdeme tak místo, na které můžeme `x` najít tak, aby příště

[1] Vytvořit verzi operací používající opačného přístupu, než který použijeme, je triviální cvičení

`tree-search(x.id)` vrátilo vrchol `x`. Stačí `x` připojit jako správného potomka vrcholu, který jsme při vyhledávání navštívili před tím, než jsme narazili na `nil`.

Pro proceduru vkládání vrcholu i pro další procedury v této kapitole budeme potřebovat procedury pro nastavení levého a pravého potomka vrcholu. Níže je procedura pro připojení levého potomka, připojení pravého potomka je analogické.

```
set-left-child
⟵ r: Node — rodič
⟵ c: Node — nový levý potomek
```

```
r.left ← c
if (c ≠ nil) then c.parent ← r
```

Pozornost si zaslouží řádek 2, kde nastavujeme položku `parent`. Připouštíme, že se argument `c` může rovnat `nil`.

```
tree-insert
⟵ t: Tree — strom, do kterého vkládáme
⟵ added: Node — vkládaný vrchol
```

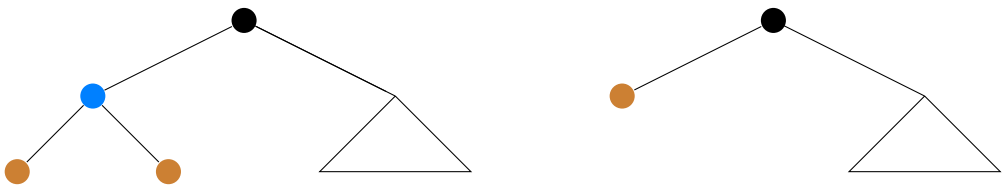
```
1  if (t.root = nil)
2     t.root ← added
3  else
4     y ← poslední navštívený vrchol při neúspěšném hledání klíče added.id
5     if (added.id < y.id)
6        set-left-child(y, added)
6     else
8        set-right-child(y, added)
```

Složitost (lineární vzhledem k výšce stromu) a korektnost procedury přímo plyne ze složitosti a korektnosti `tree-search`.

Při odebírání vrcholu `x` ze stromu s kořenem `r` rozlišujeme tří případy podle toho, kolik má `x` potomků.

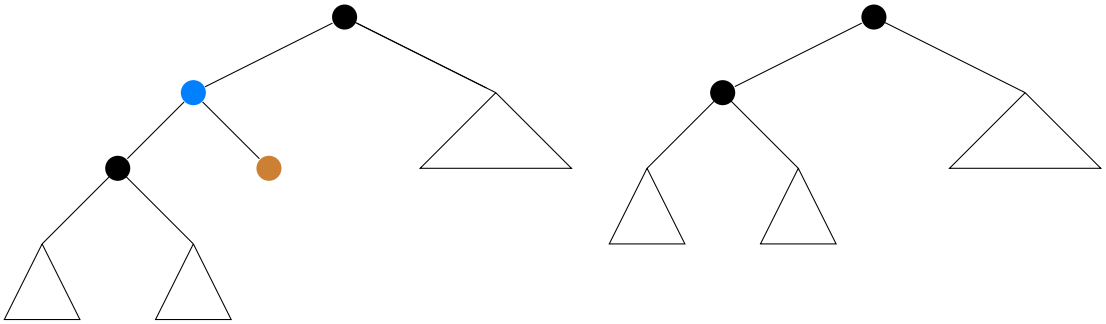
- `x` *nemá žádného potomka*

  Vrchol odebereme tak, že jeho rodiči nastavíme příslušný podstrom na nil. (Na obrázcích níže mažeme modrý vrchol, bronzové vrcholy jsou `nil`.)

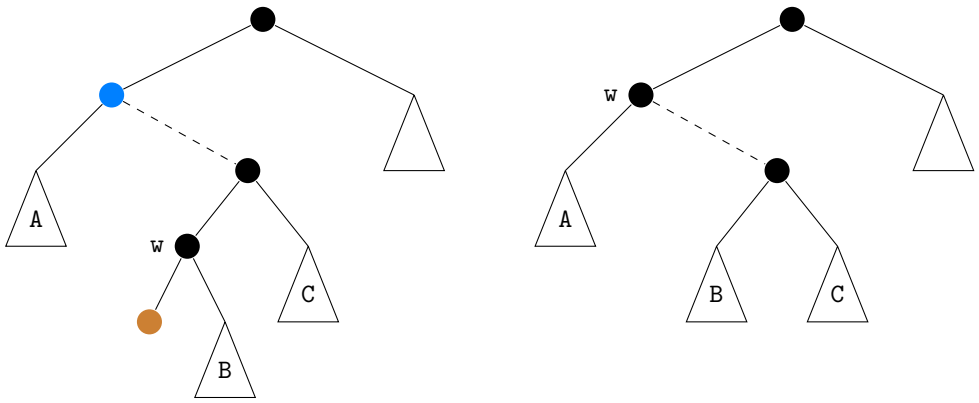


- `x` *má jednoho potomka* `y`

  Příslušný podstrom rodiče vrcholu `x` nastavíme na podstrom s kořenem `y`



- `x` *má dva potomky*
  Najdeme pořádkového následníka `w` vrcholu `x` jako vrchol s minimálním klíčem v pravém podstromu `x`. Vrchol `y` odstraníme ze stromu (nutně to bude jeden z předchozích dvou případů, `w` totiž nemá levého potomka). Vrchol `x` potom ve stromu nahradíme vrcholem `w`.



Pro proceduru mazání potřebujeme následující procedury. První nahradí jeden z podstromů jiným podstromem, druhá nahradí jeden vrchol jiným vrcholem. Fungování procedur samo o sobě nezaručuje zachování podmínky uspořádání. Obě procedury mají konstatní složitost.

```
tree-swap
⟵ t: Tree
⟵ u: Node — kořen nahrazovaného podstromu
⟵ v: Node — kořen podstromu, kterým nahrazujeme
```

```
1  if (t.root == u) then t.root ← v
2  else
3     y ← u.parent
4     if (u == y.left) then set-left-child(y, v)
5     else set-right-child(y, v)
```

Na řádku 1 ošetříme případ, kdy je nahrazovaný vrchol `u` kořenem stromu. V `else` větvi potom najdeme rodiče vrcholu `u` (který nutně existuje, `u` zde není kořen stromu), a připojíme `v` jako příslušného potomka tohoto rodiče.

```
node-swap
⟵ t: Tree
⟵ u: Node — nahrazovaný vrchol
⟵ v: Node — vrchol, kterým nahrazujeme
```

```
1 set-left-child(v, u.left
2 set-right-child(v, u.right)
3 tree-swap(t, u, v)
```

Na řádcích 1 a 2 zapojíme potomky vrcholu `u` jako potomky vrcholu `v`. Stromem s kořenem `v` potom nahradíme podstrom s kořenem `u` (tady nevadí, že `u` udržuje reference na své původní potomky, v proceduře `tree-swap` se s nimi vůbec nepracuje.

Nyní již můžeme uvést proceduru pro odebírání vrcholu.

```
tree-delete
⟵ t: Tree
⟵ z: Node — odebíraný vrchol, předp. že je ve stromu
```

```
1 if (z.left = nil)
2    tree-swap(t, z, z.right)
3 else if (z.right = nil)
4    tree-swap(t, z, z.left)
5 else
6    y ← tree-min(z.right)
7    tree-delete(t, y)
8    node-swap(t, z, y)
```

Na řádcích procedura `1` až `4` řeší první dva případy, které mohou nastat. Neporušíme podmínku uspořádání, protože nahrazujeme `z` jedním z jeho podstromů, jejichž klíče jsou ve stejném vztahu (tj. menší nebo větší) ke klíči vrcholu `z.parent` jako klíč samotného `z`. Na řádku `6` víme, že `z` má dva potomky, a proto je `y` pořádkovým následníkem `z`. Vrchol `y` je minimem v `z.right` a proto nemá levého potomka. Rekurzivní zavolání `tree-delete` na řádku `7` tak skončí na řádcích `1` až `4` (a k dalšímu rekurzivnímu zavolání již nedojde). Nahrazením vrcholu `z` vrcholem `y` neporušíme podmínku uspořádání: vrchol `y` má větší klíč než vsechny vrcholy v podstromu s kořenem `z.left` a menší klíč než všechny vrcholy v podstromu s kořenem `z.right` (mimo sebe samotného). Současně byl `y` před řádkem 7 v podstromu s kořenem `z`, a proto je uspořádání `y.id` a `z.parent.id` stejné, jako uspořádání `z.id` a `z.parent.id`.

Složitost `tree-delete` je lineární vzhledem k výšce stromu. První dva případy (řádky 1 až 4) jsou v konstantním čase. Pro třetí případ procedura musí najít pořádkového následníka pomocí `tree-min`, jehož složitost je lineární vzhledem k výšce stromu. Procedura `node-swap` má složitost konstantní.

5.3 ROTACE

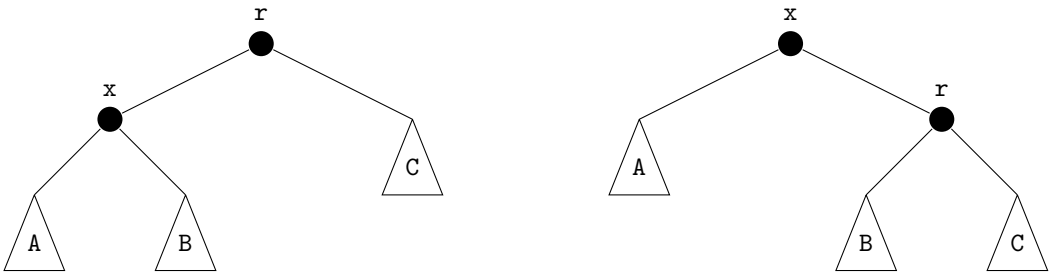
Rotace je operace, kterou aplikujeme na dva vrcholu ve stromu: na vrchol `r` a jeho potomka `x`. Po provedení rotace je jejich role prohozena: vrchol `r` je potomkem `x`. Operace je přitom navržena tak, aby i po jejím provedení platila podmínka uspořádání.

Rozlišujeme levou a pravou rotaci. Je-li `r` napravo od `x` (tedy `x` je levým potomkem `r`), potom se jedná o rotaci *levou*, je-li tomu opačně, jedná o rotaci `pravou`.

Procedura provádějící rotaci bere jako argument vrchol `r` a vrací jako svůj výsledek vrchol `x`. Ten je vlastně novým kořenem podstromu, jehož původním kořenem byl `r`. Pokud po provedení rotace vrchol `x` jako jako příslušného potomka původního rodiče vrcholu `r` (tj. rodiče před rotací), dostaneme korektně zapojený binární strom. Ten obsahuje stejné klíče jako před rotací, pouze se změnila pozice vrcholů `r` a `x`.

```
rotate-R
⟵ r: Node — kořen rotovaného stromu
⟶ Node
```

```
1  x ← r.left
2  B ← x.right
3  set-left-child(r, B)
4  set-right-child(x, r)
5  return x
```

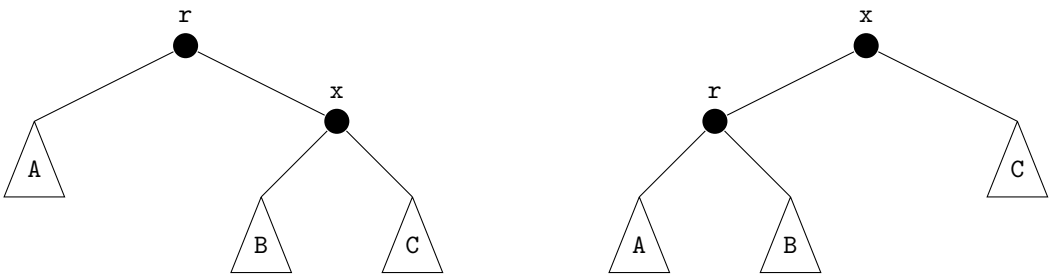


Doporučuji čtenáři si uvědomit, že podstrom `B` je na předchozím obrázku po rotaci zapojen tak, že je dodržena podmínka uspořádání.

Levá rotace je duální operací (pouze je prohozena levá a pravá strana).

```
rotate-L
⟵ r: Node — kořen rotovaného stromu
⟶ Node
```

```
1  x ← r.right
2  B ← x.left
3  set-right-child(r, B)
4  set-left-child(x, r)
5  return x
```



Složitost obou procedur je konstatní.

5.4 VELIKOST STROMU

Velikostí stromu myslíme počet jeho vrcholů. Velikosti jednotlivých podstromů ve stromu budeme udržovat tak, že do struktury pro vrchol přidáme položku `count`. Pro každý vrchol `x` ve stromu tak bude platit rovnost

$$\texttt{x.count} = \texttt{x.left.count} + \texttt{x.right.count} + 1,$$

přičemž pokud je `x.left` nebo `x.right` rovno `nil`, nahradíme v rovnosti příslušnou `count` položku nulou. Hodnota `x.count` je tak velikost podstromu s kořenem `x`.

Položka `count` je užitečná pro operace, které potřebují znát velikosti podstromů. Ty bychom mohli počítat na místě (například průchodem `in-order-walk`). Efektivnější je ovšem tuto položku mít rovnou ve struktuře pro vrchol.

Procedury, které nějakým způsobem mění strom, musí položku `count` vrcholů udržovat tak, aby její hodnoty odpovídala skutečné velikosti daného podstromu. Tato nutnost vede k drobným úpravám těchto procedur, jež krátce shrnujeme níže. Čtenář si podrobnosti dopracuje jako cvičení.

Procedura `tree-insert`. Vloženému vrcholu nastavíme `count` na jedna. Všem vrcholům navštíveným při hledání místa pro vkládání zvedneme položku `count` o jedna. Složitost vkládání zůstává $\Theta(\texttt{h})$.

Procedura `tree-delete`. Před provedením `tree-swap` zmenšíme všem vrcholům na cestě od rodiče nahrazovaného vrcholu do kořene položku `count` o jedna. Navíc v proceduře `node-swap` zkopírujeme z nahrazovaného vrcholu do vrcholu, kterým nahrazujeme, hodnotu položky `count`. Oproti původní verzi `tree-delete` potřebujeme jeden průchod od mazaného listu do kořene navíc, ovšem složitost zůstává $\Theta(\texttt{h})$.

Procedura `rotate-R`. Po provedení rotace upravíme počty následovně (značením odkazujeme do kódu procedury uvedenému výše).

$$\texttt{transfer} \leftarrow \begin{cases} \texttt{0} & \text{pokud } \texttt{B} = \texttt{nil} \\ \texttt{B.count} & \text{jinak} \end{cases}$$

```
r.count ← r.count + transfer - x.count
x.count ← x.count - transfer + r.count
```

Složitost zůstává konstantní.

Procedura `rotate-L`. Analogicky `rotate-R`.

Pro přirozené číslo $1 \leqslant \texttt{k} \leqslant \texttt{n}$ je `k`-tá pořádková statistika vrchol, který obsahuje `k`-tý nejmenší klíč ve stromu. Naivně ji můžeme najít pomocí průchodu `in-order-walk`, který případně zastavíme po navštívení `k` vrcholů. Složitost takové procedury je lineárně závislá na `k`, které ovšem můžeme zvolit lineárně závislé na počtu vrcholů stromu. S využitím položky `count` dostaneme algoritmus, jehož složitost je závislá na výšce stromu. Z podmínky uspořádání plyne, že pro strom s kořenem máme:

- Pokud platí `x.left.count == k - 1`, pak je `k`-tou pořádkovou statistikou přímo vrchol `x`.
- Pokud platí `x.left.count > k - 1`, pak `k`-tou pořádkovou statistikou ve stromu s kořenem `x` je `k`-átá pořádková statistika ve stromu s kořenem `x.left`.

- Pokud platí `x.left.count < k - 1`, pak `k`-tou pořádkovou statistikou ve stromu s kořenem `x` je `(k - 1 - x.left.count)`-tá pořádková statistika ve stromu s kořenem `x.right`.

Přepsáním do pseudokódu dostaneme

```
select
⟵ r: Node — kořen stromu, ve kterém hledáme
⟵ k: Int — parametr statistiky
⟶ Node
```

```
1  t ← 0
2  if (r.left ≠ nil)
3     t ← r.left.count
4  if (t > k - 1)
5     return select(r.left, k)
6  else if (t < k - 1)
7     return select(r.right, k - t - 1)
8  else
9     return r
```

Složitost procedury `select` je lineárně závislá na výšce stromu s kořenem `r`: v každém rekurzivním volání je výška stromu, jehož kořen předáváme jako první argument, nejméně o jedna menší než výška stromu s kořenem `r`.

Další procedurou, ve které výhodně využijeme položek `count` je `rank`. Ta je v jistém smyslu inverzní k `select`. Pro vstupní vrchol `x` hledáme číslo `k` takové, že `x` je `k`-tou pořádkovou statistikou. Z podmínky uspořádání plyne následující tvrzení:

Věta 5.6

Pro každou dvojici vrcholů `x,y` takovou, že `x.id > y.id`, platí právě jedno z následujících tvrzení:

(a) `y` je v levém podstromu `x`,

(b) `x` je v pravém podstromu `y`,

(c) existuje vrchol `z` tak, že `y` je v jeho levém podstromu a `x` je v jeho pravém podstromu.

Pro vrchol `x` proto můžeme spočítat, kolik je ve stromu vrchol s klíčem menším než `x.id`, tak, že projdeme všechny vrcholy na cestě z `x` do kořene. Pro každý z nich ověříme, jestli neplatí některá z podmínek z předchozí věty a podle toho zvětšíme počet nalezených vrcholů s klíčem menším než `x`. Podrobnosti čtenář nalezne v pseudokódu.

```
rank
⟵ r: Node — kořen stromu
⟵ x: Node — vrchol, pro nějž hledáme pořadí
⟶ Int
```

```
1  t ← 1
2  if (x.left ≠ nil)
     t ← t + x.left.count
3  y ← x
4  while (y ≠ r)
5     if (y = y.parent.right)
5        t ← t + 1
6        if (y.parent.left ≠ nil)
7           t ← t + y.parent.left.count
8     y ← y.parent
9  return t
```

Na řádku 2 připočítáme vrcholy podle bodu (a), Na řádcích 5 pak připočítáme vrcholy podle bodu (b) a na řádku 7 pak podle bodu (c) přechozí věty. Složitost procedury je lineárně závislá na výšce stromu, každá iterace cyklu na řádcích 4--8 totiž sníží hloubku vrcholu y o jedna.

5.5 KOŘENOVÉ VERZE OPERACÍ

Kořenová verze operace s binárním stromem je její modifikace, který pro danou operaci významný vrchol přesune do kořene stromu. Udělá to tak, aby pořád platila podmínka uspořádání. Pro operaci vkládání je významným právě vkládaný vrchol, pro operaci vyhledávání, hledání pořádkové statistiky či jiné dotazy je to nalezený vrchol.

Kořenových operací používáme v případech, kdy v sérii operací se stromem přistupujeme k jednomu vrcholu (nebo malé množině vrcholů) opakovaně. Přesunem tohoto vrcholu do malé hloubky snížíme reálnou složitost operací, které k nim přistupují.

Pro přesun nějakého vrcholu do kořene použijeme rotací. Rotace prohodí role dvou vrcholů: rodič se stane potomkem a potomek rodičem. Přitom se hloubka potomka zmenší o jedna. Pokud tedy pro vrchol x provedeme posloupnost rotací, ve kterých vystupuje jako potomek, musíme se eventuálně dostat do situace, kdy je x kořenem stromu. Toto „probublání“ x do kořene lze realizovat více způsoby, v operacích níže tak učiníme pomocí rekurze, kdy rotace provedeme před návratem z rekurzivního volání. To lze, protože rotace je potřeba provádět s vrcholy, které se nachází na cestě z kořene do x, a tytéž vrcholy jsou argumenty rekurzivních volání operací, jejichž kořenové verze vytváříme.

Níže si ukážeme kořenové verze procedur `insert` a `select`. Operace implementujeme rekurzivně, vrací kořen stromu po provedení operace.

```
root-insert
⟵ r: Node — kořen stromu, do kterého vkládáme
⟵ added: Node — vkládaný vrchol
⟶ Node — kořen stromu po vložení
```

```
1  if (r == nil) then return added
2  if (r.id > added.id)
3     left-child ← root-insert(r.left, added)
4     set-left-child(r, left-child
5     return rotate-R(r)
6  else
7     right-child ← root-insert(r.right, added)
6     set-right-child(r, right-child)
7     return rotate-L(r)
```

Na řádku 1 provádíme vložení do prázdného stromu. V takovém případě je jediným vrcholem a tedy i současně kořenem vrchol `added`. Na řádcích 3-5 vkládáme do levého podstromu. Nejdříve (řádek 3) provedeme rekurzivní zavolání, které vrátí kořen levého podstromu po provedeném vložení.[2] Tento vrchol na řádku 4 připojíme zpět do stromu, dostaneme tak korektní vyhledávací strom. Nakonec na řádku 5 provedeme pravou rotaci, ve které přesuneme kořen levého podstromu do kořene celého stromu a tento nový kořen vrátíme jako výsledek. Složitost operace zůstává lineární vzhledem k výšce stromu. Při každé návštěvě vrcholu na cestě z kořene do místa, kde je (před rotacemi) přidán nový vrchol, provedeme oproti obyčejné operaci `insert` pouze konstantní počet operací: připojení nového podstromu a rotaci.

Analogicky postupem můžeme upravíme i operaci `select`, pseudokód po úpravě je níže. Další operace (např. vyhledávání) si čtenář upraví jako cvičení.

```
partition
⟵ r: Node — kořen stromu, ve kterém hledáme
⟵ k: Int — parametr statistiky
⟶ Node — kořen stromu po hledání
```

```
1  t ← 0
2  if (r.left ≠ nil)
3     t ← r.left.count
4  if (t == k - 1) then return r
5  if (t > k - 1)
6     set-left-child(r, partition(r.left, k))
7     return rotate-R(r)
8  else
9     set-right-child(r, partition(r.right, k))
10    return rotate-L(r)
```

5.6 Manuální vyvážení

Manuální vyvážení je operace, která změní strukturu strom tak, aby měl logaritmickou výšku (tedy jeho výška je $\lg n$, kde n je počet vrcholů stromu).

Prvním způsobem, jak vyvážení provést, je projít strom pomocí `in-order-walk` a navštívené vrcholy si uložit do pole vzestupně podle klíčů. Poté z těchto vrcholů vybudujeme strom znovu tak, aby byl vyvážený. Označme si pole s vrcholy `A`. Potom k sestavení využijeme například následující algoritmus.

[2] To je nutně vrchol `added`.

```
balance
⟵ A: []Node — Pole vrcholů stromu uspořádané vzestupně podle hodnot klíčů.
⟵ l: Int — První index zpracovávané části pole.
⟵ p: Int — Poslední index zpracovávané části pole.
⟶ Node — Kořen vytvořeného stromu.
```

```
1  if (l > p) then return nil
2  s ← floor((l + p) / 2)
3  left ← balance(A, l, s-1)
4  right ← balance(A, s+1, p)
5  set-left-child(A[s], left)
6  set-right-child(A[s], right)
7  return A[s]
```

Při bližším prozkoumání algoritmu vidíme, že vlastně explicitně vytvoří strom algoritmu vyhledávání pomocí půlení intervalů, jak jsme si ho ukázali v kapitole 2. Složitost algoritmu je dána rekurencí

$$T(n) = 2T(n/2) + \Theta(1),$$

jejímž řešením je $\Theta(n)$. Místo explicitní konstrukce stromu bychom alternativně mohli vrcholy na indexu `s` (viz řádek 2) vkládat s použitím `tree-insert` (volaného před rekurzivním zavoláním `balance-1`). Složitost by tak byla dána

$$T(n) = 2T(n/2) + \Theta(\lg n),$$

protože výška vytvářeného stromu je v průběhu algoritmu omezena logaritmem z počtu již vložených vrcholů, a tím je omezena i složitost `tree-insert`. Řešením rekurence zůstává $\Theta(n)$.

Alternativní algoritmus pro vyvážení dostaneme s využitím operace `partition` s jejíž pomocí přesuneme medián do kořene. Medián je totiž $n/2$-tá pořádková statistika (s případným zaokrouhlením). Poté rekurzivně aplikujeme stejný postup pro levý a pravý podstrom kořene. Detaily procedury ponecháváme jako cvičení.

Složitost výše uvedeného postupu záleží na tom, jaká je výška stromu, na který jej aplikujeme. Pokud je tato výška logaritmická (např. $c \lg n$ pro nějakou konstantu c), je i složitost `partition` logaritmická a složitost je dána rekurencí $T(n) = 2T(n/2)+\Theta(\lg n)$, stejně jako v předchozím případě při použití `tree-insert`. Pokud je však výška lineární vzhledem k počtu vrcholů, je složitost `partition` také lineární vzhledem k počtu vrcholů a složitost vyvážení je dána rekurencí

$$T(n) = 2T(n/2) + \Theta(n),$$

jejímž řešení je $O(n \lg n)$.

Nekonečnou množinu stromů F považujeme za vyváženou, pokud funkce $h : \mathbb{N} \to \mathbb{N}$, přiřazující n maximální výšku stromu s n vrcholy obsaženého z F, roste nejvýše logaritmicky (tedy $f \in O(\lg n)$). Přísnější podmínkou je požadavek, aby existovalo $c > 0$ tak, že pro každé n platí $f(n) < c \lg n$.

Ukážeme si dva přístupy, jak definovat množinu binárních stromů, které splňují tuto přísnější podmínku a uvidíme, jak modifikovat operace přidání a odebrání tak, aby na ně byla tato množina uzavřená.[1] Oba přístupy se drží následujícího vzoru:

1. Definujeme podmínku pro binární vyhledávací stromy. (Pro každý strom tato podmínka buď platí, nebo neplatí.) Do naší množiny vybereme ty stromy, pro které tato podmínka platí.
2. Dokážeme, že platnost výše uvedené podmínky implikuje logaritmickou výšku stromu. (viz přísnější podmínka vyváženosti výše).
3. Upravíme operace pro vkládání a odebírání vrcholů tak, aby výsledný pro výsledný strom podmínka platila. Složitost těchto operací musí zůstat lineární vzhledem k výšce stromu.

6.1 AVL STROMY

Vyváženost vrcholu x v binárním stromu je rozdíl výšky jeho levého podstromu a výšky jeho pravého podstromu, přičemž výšku prázdného podstromu definujeme rovnu -1. Řekneme, že strom je *přípustný*, pokud je vyváženost každého vrcholu v něm z množiny $\{1, 0, -1\}$.

Věta 6.1: O výšce AVL stromu

Existuje konstanta $c > 0$ tak, že výška přípustného stromu s n vrcholy je menší nebo rovna $c \lg n$.

Důkaz. Hlavní myšlenka důkazu je následující. Pokusíme se namalovat přípustné stromy výšek $0, 1, 2, \ldots$ s co nejméně vrcholy. U těchto stromů je funkce popisující výšku stromu v závislosti na počtu vrcholů nejvíce rostoucí.

Všimneme si, že strom strom výšky m má jako levý podstrom strom výšky $m - 1$ a jako pravý podstrom strom výšky $m - 2$. Proto, pokud počet vrcholů ve stromu výšky m označíme $T(m)$, platí

$$\begin{aligned} T(m) &= T(m-1) + T(m-2) + 1, \\ T(0) &= 1, \\ T(1) &= 2. \end{aligned}$$

Řešením předchozí rekurence je exponenciální funkce (všimněme si, že rekurence je velmi podobná definici Fibonacciho posloupnosti). Funkce vyjadřující výšku v závislosti na počtu vrcholů, která je k T inverzní funkcí, tedy musí být logaritmická. □

[1] Aplikací operace na strom z naší množiny opět dostaneme strom z naší množiny.

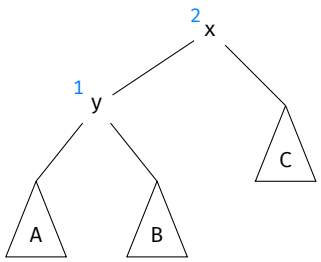
Snadno najdeme příklad toho, že operace vložení nebo operace odebrání transformují přípustný strom na strom, který není přípustný. Stačí vzít některý ze stromů z minulého důkazu a vložit vrchol jako potomka nejhlubšího vrcholu, případně odebrat list v nejmenší hloubce. Situace však není zcela beznadějná. Pokud vložení či odebrání přípustnost stromu poruší, můžeme jedním průchodem od místa změny do kořene přípustnost opravit. Ve zbytku kapitoly si ukážeme jak.

Do struktury pro vrchol přidáme položku `bf`, ve které budeme udržovat vyváženost vrcholu. Hodnoty položek `bf` budeme udržovat konzistentní. Přestavme si, že provedeme vložení/odebrání vrcholu jako v klasické binárním vyhledávacím stromu. U některých vrcholů ve stromu pak položky `bf` neodpovídají jejich skutečné vyváženosti. Protože vyváženost závisí na výškách podstromů a jediné vrcholy, kterým operace mohla změnit výšku podstromu leží na cestě z kořene do místa změny ve stromu (tj. místa kam byl vložen, nebo odkud byl odebrán vrchol), stačí opravit položky `bf` právě jim.

Jak uvidíme později, položku `bf` vrcholu `x` budeme upravovat pouze v situaci, kdy budeme vědět, že se výška právě jednoho z `x.left` nebo `x.right` změnila o jedna. K opravě tak stačí k `x.bf` přičíst, nebo od něj odečíst, hodnotu jedna, v závislosti na tom, výška kterého z podstromů se změnila, a jestli se zvýšila nebo snížila. Úpravy jsou zachyceny v následující tabulce. Čtenáři doporučuji, aby si je odvodil i sám.

podstrom	výška se zvětšila	výška se zmenšila
`x.left`	$+1$	-1
`x.right`	-1	$+1$

Hodnoty položek `bf` budeme měnit nejvýše o jedna, proto musí být nepřípustné vyváženosti, na které můžeme narazit, z množiny $\{2, -2\}$. Představme se tedy situaci, kdy je vyváženost vrcholu `x` rovna `2` a oba podstromy `x` jsou přípustné. Protože výška levého podstromu `x` je o 2 větší než výška toho pravého, má `x` levého potomka. Označme jej `y`. Vyváženost `y` je z množiny $\{-1, 0, 1\}$, pro začátek tedy předpokládejme, že je rovna 1. Situace je zachycena na následujícím obrázku. (Vyváženosti jsou zachyceny modře).



Ze vyvážeností `x` a `y` můžeme zjistit vztah mezi výškami podstromů `A, B, C`. Výšku budeme značit pomocí `h()`, kde argumentem bude buď označení podstromu nebo jeho

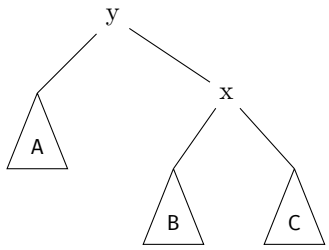
kořene. Protože vyváženost y je 1, máme

$$h(A) = h(B) + 1.$$

Podobně z vyváženosti x dostaneme

$$\begin{aligned} h(y) &= h(C) + 2 \\ h(A) + 1 &= h(C) + 2 \\ h(A) &= h(C) + 1 \end{aligned}$$

Nakonec vidíme, že $h(x) = h(A) + 2$. Důvodem, proč je vyváženost x rovna 2 je tedy to, že nejvyšší z podstromů A,B,C je v hloubce 2, zatímco nejnižší je v hloubce 1. Situaci opravíme provedením `rotate-R(x)`, výsledný strom je na následujícím obrázku.

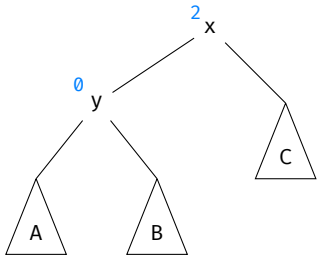


Ze vztahů mezi výškami A,B,C dopočítáme vyváženosti vrcholů:

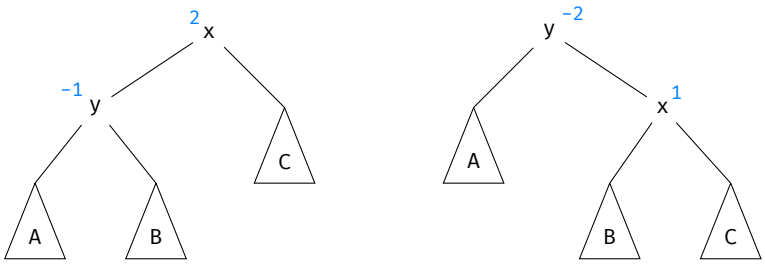
- vyváženost y je $h(B) - h(C) = 0$
- vyváženost x je $h(A) - (h(C) - 1) = 0$

Výška stromu po rotaci je $h(y) = h(A) + 1$. Rotace tedy výšku stromu zmenšila o jedna.

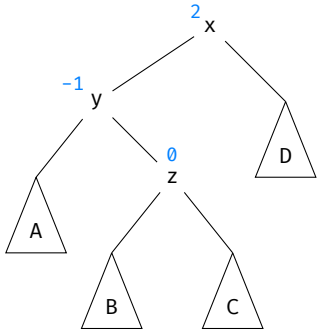
Pokud v situaci zachycené na následujícím obrázku, kde je nyní vyváženost vrcholu y rovna 0, použijeme pravou rotaci, také úspěšně opravíme strom na přípustný. Vyváženost vrcholu x bude 1 a vyváženost y bude -1. Na rozdíl od prvního případu ovšem rotace nezmění výšku stromu.



Je lákavé použít pravou rotaci i v případě, kdy se je vyváženost y rovna -1. Tam se ovšem přípustnost opravit nepovede. Problém je, že $h(B) > h(A) = h(B)$, ale po rotaci zůstane B ve stejné hloubce, zatímco stromy A a C si hloubky prohodí, viz následující obrázek.



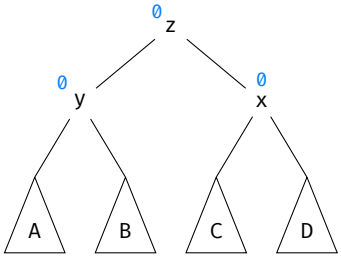
Musíme se tedy podívat na strom `B` detailněji. Uvažme tedy vrchol `z`, který je kořenem `B` a jeho dva podstromy. Situace je zachycena na následujícím obrázku, oproti obrázkům výše jsme jednotlivé stromy přeznačili, v dalším textu budeme uvažovat toto nové značení.



Spočítáme vztahy mezi výškami jednotlivých podstromů a dostaneme

$$h(A) = h(B) = h(C) = h(D)$$

Provedeme `rotate-L(y)` a poté `rotate-R(x)`, všechny čtyři podstromy budou ve stejné výšce a situace je vyřešena.

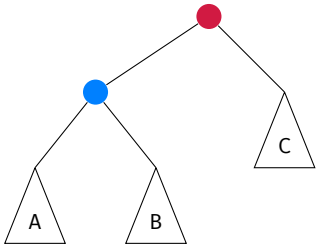


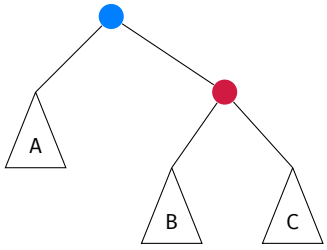
Před rotacemi byla výška stromu rovna $h(B) + 3$, po nich je rovna $h(B) + 2$. Úprava tak snížila výšku stromu o 1.

V situaci, kdy je před úpravami `z.bf` rovno `-1` nebo `1` je situace analogická. Právě jeden ze stromů `B`, `C` má o jedna menší výšku než ostatní a v důsledku toho je buď `y.bf = 1` nebo `x.bf = -1`, přičemž dva zbylé vrcholy mají položku `bf` rovnu `0`.

Výše jsme odvodili úpravy, pokud byla hodnota položky `bf` některého z vrcholů rovna `2`. Pro hodnotu `-2` jsou úpravy symetrické: místo levých rotací se použijí pravé rotace a vyváženostem se otočí znaménko. Všech 10 rotací je zachyceno na obrázcích a v tabulkách níže. Na obrázcích je vždy strom před úpravou a po úpravě, v tabulce je šipkou zachycen směr úpravy, vyváženosti vrcholů před a po rotaci a rozdíl výšky stromu před a po rotaci.

Jednoduché rotace

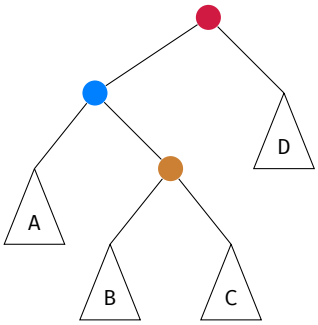


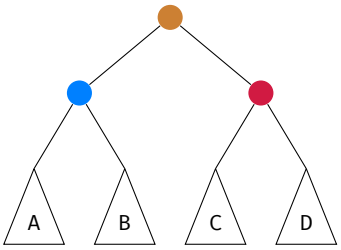


⟶	•	•	•	•	změna výšky
	2	1	0	0	-1
	2	0	1	-1	0

⟵	•	•	•	•	změna výšky
	-2	-1	0	0	-1
	-2	0	-1	1	0

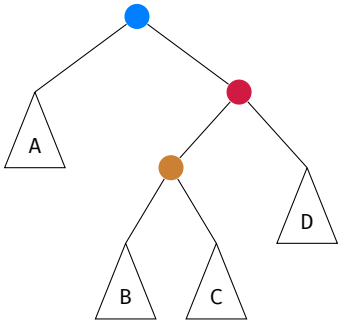
Dvojitá rotace levo-pravá

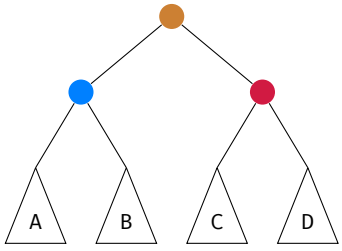




⟶	•	•	•	•	•	•	změna výšky
	2	-1	0	0	0	0	-1
	2	-1	1	0	0	-1	-1
	2	-1	-1	0	1	0	-1

Dvojitá rotace pravo-levá





⟶	●	●	●	●	●	●	změna výšky
	-2	1	0	0	0	0	-1
	-2	1	-1	0	1	0	-1
	-2	1	1	0	0	-1	-1

Nyní si uvedeme algoritmus pro přidání vrcholu do AVL stromu. Pro vrcholy ve stromu a pro strom samotná máme definovány následující struktury.

```
struct Node
   id: Key
   left: Node
   right: Node
   parent: Node
   bf : Int — vyváženost
```

```
struct Avl-Tree
   root: Node — kořen stromu
```

Algoritmus samotný je zachycen následující procedurou.

```
avl-insert
⟵ t: Avl-Tree – strom, do kterého přidáváme
⟵ added: Node – přidaný vrchol
```

```
1  vložíme added jako do binárního vyhledávacího stromu
2  added.bf ← 0
3  u ← added.parent
4  v ← added
5  while (u ≠ nil)
6     if (v = u.left)
7        u.bf ← u.bf + 1
8     else
9        u.bf ← u.bf - 1
10    if (u.bf = 0)
11       return
12    if (u.bf = 2) or (u.bf = -2)
13       p ← u.parent
14       w ← fix-tree(u)
15       if (p = nil)
16          t.root ← w
17       else
18          připojíme w jako správného potomka vrcholu p
19       return
20    v ← u
21    u ← u.parent
```

V cyklu `while` procházíme s pomocí proměnných `u,v` cestu od přidaného vrcholu do kořene, přitom `v` je vždy potomkem `u`. Na řádcích `6` až `9` opravíme `u.bf` na korektní hodnotu. Pokud je (takto upravená) `u.bf` rovna `0`, algoritmus končí. Pokud je rovna `1` nebo `-1`, pokračujeme na cestě ke kořeni. Pokud je `u.bf` rovna `-2` nebo `2`, pak přidání porušilo přípustnost stromu a pomocí procedury `fix-tree` provedeme správnou rotaci. Tato procedura vrací kořen stromu získaného rotací, tento strom opětovně zapojíme zpět do původního stromu (v proměnné `p` si pamatuje jeho rodiče). Po rotaci algoritmus končí.

Přidání vrcholu na řádku `1` má lineární složitost vzhledem k výšce stromu. Zbytek algoritmu v nejhorším případě projde strom od přidaného vrcholu do kořene a pro každý navštívený vrchol provedeme konstatní počet operací. Složitost tohoto průchodu je tak také lineárně závislá na výšce stromu. Protože přidání provádíme do přípustného stromu, je jeho složitost $\Theta(\lg n)$.

Věta 6.2: Invariant zvyšování výšky

Pokud algoritmus provádíme na přípustném stromu, pak na řádku 5 je vždy výška podstromu s kořenem `v` o jedna větší, než byla před přidáním na řádku 1.

Důkaz. Důkaz povedeme indukcí přes počet provedení cyklu `while`.

Před prvním provedením cyklu je `v` vrchol `added`, a je potomkem `u`. Před řádkem jedna byl tento potomek `u` roven `nil`, výška mu odpovídajícího podstromu tak byla `-1`.

Nyní předpokládejme, že (a) platí po provedení $i - 1$ iterací cyklu a jsme na začátku iterace `i`. Jediný způsob, jak se opět dostat na řádek `5` je po provedení řádku `21`. Po úpravě `u.bf` na řádcích `6` až `9` proto máme `u.bf = -1` nebo `u.bf = 1`. Před úpravou a i před přidáním na řádku `1` tak muselo být `u.bf = 0`, vkládáme totiž do přípustného stromu. Oba podstromy vrcholu `u` tedy měli před přidáním stejnou výšku. Z indukčního předpokladu ovšem víme, že výška jednoho z těchto podstromů (toho s kořenem `v`) se zvýšila o 1. Výška `u` se tedy také musela přidáním zvýšit o 1. Pravdivost tvrzení tak plyne z přiřazení na řádku `20`. □

Z předchozí věty hned plyne, že na řádcích `10` až `20` obsahuje je `u.bf` skutečnou vyváženost vrcholu `u`.

Věta 6.3

Pokud algoritmus provádíme na přípustném stromu a na řádku 10 je hodnota `u.bf` rovna 0, jsou výšky podstromů s kořen `u` před a po provedení přidání na řádku 1 stejné.

Důkaz. Z přechozí věty plyne, že `u.bf` na řádku `10` odpovídá skutečné vyváženosti vrcholu `u`. Na řádku `6` muselo být `u.bf = 1` nebo `u.bf = -1`, a tato hodnota odpovídá skutečné vyváženosti `u` před přidáním vrcholu na řádku `1`, vkládáme totiž do přípustného stromu. Z předchozí věty víme, že výška jednoho z podstromů `u` se přidáním zvýšila o jedna (je to právě podstrom s kořenem `v`), a protože po přidání jsou výšky obou podstromů `u` stejné, musel to být ten nižší podstrom. Výška stromu s kořenem `u` tak musela zůstat nezměněna. □

Věta 6.4

Pokud algoritmus provádíme na přípustném stromu, je po provedení `fix-tree` na řádku `14` algoritmu výška stromu s kořenem `w` stejná jako byla výška stromu s kořenem `u` před přidáním vrcholu na řádku 1.

Důkaz. Pokud dojde k rotaci, musí být `u.bf` $= \pm 2$ (skutečná vyváženost `u` po přidání). Před úpravou na řádcích `6` až `9` tak muselo být `u.bf` $= \pm 1$ (skutečná vyváženost `u` před přidáním). Výška podstromu s kořenem `u` je proto o jedna větší po přidání na řádku 1 než před ním.

Pokud ukážeme, že použijeme rotaci, která sníží výšku stromu o jedna, bude důkaz hotov. Jediná rotace, která nesnižuje výšku stromu je jednoduchá rotace v situaci, kdy `u.bf` $= \pm 2$ a `v.bf` $= 0$. To by ovšem v předchozí iteraci cyklu `while` muselo být `u.bf = 0` (v předchozí iteraci je `u` rovno současnému `v`, viz řádky `20` a `21`). To by ale algoritmus skončil na řádku `11`. □

Věta 6.5: Koreknost algoritmu `avl-insert`

Nechť T je přípustný strom. Potom je strom, který získáme pomocí přidání vrcholu do T algoritmem `avl-insert`, taky přípustný.

Důkaz. Uvědomíme si, že pokud se nějakou operací se stromem změní vyváženost vrcholu, pak se nutně musela změnit výška některého jeho podstromu. Pokud tedy výšky obou podstromů zůstanou stejné, vyváženost vrcholu se nezmění.

Po přidání vrcholu se mohou změnit pouze vyváženosti vrcholů, které jsou na cestě z přidaného vrcholu do kořene, ostatním vrcholům se totiž nezmění výšky podstromů. Algoritmus tuto cestu prochází. Po cestě mění položky `bf` tak, aby odpovídali skutečné hodnotě vyváženosti. Pokud narazíme na nepovolenou hodnotu vyváženosti, strom upravíme tak, aby byl opět přípustný pomocí rotací.

Pokud algoritmus skončí předčasně, je výška podstromu, jehož kořenem je aktuálně zpracovaný vrchol (tj. buď `u` na řádku `10`, nebo `w` na řádku `14`), stejná jako výška odpovídajícího podstromu před přidáním vrcholu na řádku 1. Vyváženosti zbývajících vrcholů na cestě do kořene tedy nemusíme upravovat. □

Algoritmus odebrání vrcholu podobně jako algoritmus přídání vrcholu prochází cestu od místa změny do kořene. Zatímco u přidání upravuje položky `bf` vrcholů na této cestě, protože výška jednoho z jejich podstromů se zvýšila o 1, v algoritmu odebírání to je proto, že výška jednoho z podstromů se snížila. U odebírání si z technických důvodů si nemůžeme pamatovat vrchol, který jsme na cestě navštívili před aktuálním vrcholem (u přidání to byl vrchol `v`), naštěstí si stačí pamatovat, jestli to byl levý nebo pravý podstrom aktuálního vrcholu. Za tímto účelem budeme používat výčtového typu.

```
enum Direction {left, right}
```

Je nutné upravit proceduru `tree-delete` aby vracela vrchol s maximální hloubkou, kterému se snížila výška jednoho podstromu, a který podstrom to byl. Procedura `tree-delete` tedy bude vracet dvojici `(Node, Direction)`, která je při odebírání vrcholu `z` určena následovně.

- `z` *nemá dva potomky:*
  Pokud `z == z.parent.left` funkce vrací `(z.parent, left)`, jinak vrací `(z.parent, right)`.

- `z` *má dva potomky:*
  Nechť `w` je pořádkový následník `z`. Pokud je `w = z.right`,[2] potom procedura vrací `(w, right)`. Jinak vrací `(w.parent,left)`.

Další procedura, která potřebuje úpravu je `node-swap`. Do vrcholu, kterým nahrazujeme, musí z vrcholu, který nahrazujeme, zkopírovat hodnotu položky `bf`.[3]

Algoritmus odebírání je zachycen následující procedurou.

```
avl-delete
⟵ t: Avl-Tree
⟵ z: Node
```

```
1  u, from ← tree-delete(t,z)
2  while (u ≠ nil)
3     if (from = left)
4        u.bf ← u.bf - 1
5     else
6        u.bf ← u.bf + 1
7     if (u.bf = 1) or (u.bf = -1)
8        return
9     p ← u.parent
10    if (p ≠ nil) and (u = p.left)
11       from ← left
12    else
13       from ← right
14    if (u.bf = 2) or (u.bf = -2)
15       w ← fix-tree(u)
16       if (p = nil)
17          t.root ← w
18       else
19          připojíme w jako správného potomka vrcholu p
20       pokud fix-tree nesnížilo výšku stromu, algoritmus končí
21    u ← p
```

V cyklu `while` procházíme cestu od místa odebrání vrcholu ke kořeni. V proměnné `from` si pamatujeme, jestli jsme do aktuálnímu vrcholu přisli z levého nebo pravého podstromu. Na řádcích `3` až `6` upravujeme hodnotu položky `bf` na korektní hodnotu. Pokud je tato upravená hodnota rovna `-1` nebo `1`, algoritmus končí. Pokud je rovna `-2` nebo `2`, provedeme pomocí `fix-tree` příslušnou rotaci a výsledný podstrom zapojíme zpět do stromu jako správného potomka. Po rotaci algoritmus končí, pokud rotace sníží výšku rotovaného podstromu.

Složitost odebrání vrcholu na řádku `1` je lineární vzhledem k výšce stromu. Ve zbytku algoritmu v v nejhorším případě projdeme strom od odebraného vrcholu do kořene. Pro každý navštívený vrchol provedeme konstatní počet operací, složitost je tak lineárně závislá na výšce stromu. Vzhledem k větě o výšce přípustného stromu tak máme složitost $\Theta(\lg n)$.

[2] Toto je speciální případ, který způsobuje výše zmíněné technické obtíže.

[3] Vyváženost vrcholu nezáleží na tom, jaký obsahuje vrchol klíč. Záleží pouze na „tvaru“ stromu. Pokud uprostřed stromu nahrazujeme vrchol, musíme vyváženost do vrcholu, kterým nahrazujeme, zkopírovat. Tvar stromu totiž se touto operací nemění.

Věta 6.6

Spustíme-li algoritmus odebrání vrcholu na přípustný strom, potom:

(a) na řádku 2 algoritmu platí, že výška příslušného podstromu vrcholu `u` (levého pokud `from = left` a pravého pokud `from = right`) je o jedna menší než před provedením řádku 1 algoritmu;

(b) před provedením `fix-tree` na řádku 15 jsou výšky stromů s kořenem `u` před a po odebrání vrcholu na řádku 1 totožné;

(c) je-li na řádku 7 hodnota `u.bf` rovna 1 nebo -1, jsou výšky podstromu s kořenem `u` před a po odebrání vrcholu na řádku 1 totožné.

Důkaz. (a) Indukcí přes počet provedení cyklu `while`. Před prvním provedením plyne tvrzení z úprav operace `tree-delete`.

Předpokládejme, že tvrzení platí na začátku i-té iterace cyklu. Na řádek 2 se opakovaně můžeme dostat, pokud `u.bf` $\neq \pm 1$ na řádku 7.

Pokud `u.bf` $= 0$ na řádku 7, potom před úpravou na na řádcích 3 až 6 muselo být `u.bf` $= \pm 1$. Proto musela být výška podstromu s kořenem `u` před provedením řádku 1 dána výškou toho podstromu `u`, jehož výšku jsme snížili podle indukčního předpokladu. Tím pádem jsme snížili i výšku podstromu s kořenem `u`.

Pokud `u.bf` $= \pm 2$ na řádku 7, potom před úpravou na řádcích 3 až 6 muselo být `u.bf` $= \pm 1$. Indukční předpoklad tedy musíme aplikovat na výšku toho podstromu vrcholu `u`, který měl menší výšku už před provedením řádku 1. Z toho plyne, že výšky postromu s kořenem `u` před a po provedení řádku 1 se rovnají. Výšku odpovídajícího podstromu (porovnáváme výšky `u` před `fix-tree` a `w` po `fix-tree`) může snížit pouze rotace, která snižuje výšku stromu. To ale předně odpovídá řádku 20.

(b) jsme dokázali už v předchozím odstavci.

(c) Pokud `u.bf` $= \pm 1$ na řádku 7, pak `u.bf` $= 0$ před úpravami na řádcích 3 až 6. Oba podstromy vrcholu `u` měli před řádkem jedna stejnou výšku. Snížením výšky jednoho z nich (podle (a)), jsme výšku podstromu s kořenem `u` nezmenšili. □

Věta 6.7: korektnost algoritmu `avl-delete`

Nechť T je přípustný strom. Strom, který získáme pomocí algoritmu odebrání vrcholu ze stromu T, je také přípustný

Důkaz. Po odebrání vrcholu se mohou změnit vyváženosti vrcholů, které jsou na cestě z rodiče skutečně odstraněného vrcholu do kořene. Algoritmus tuto cestu prochází a všem vrcholům, které navštíví, korektně upraví položky `u.bf` a v případě jejich nepřípustnosti provede rotaci. Pokud algoritmus skončí předčasně, je to v situaci, kdy se výška aktuálního podstromu oproti situaci před spuštěním algoritmu nezměnila a není tedy nutné upravovat položku `bf` zbývajících vrcholů na cestě do kořene. □

6.2 RED-BLACK STROMY

Binární vyhledávací strom upravíme tak, že vrcholy budeme uvažovat jako obarvené červenou nebo černou barvou. Technicky to uděláme tak, že do struktury pro vrchol přidáme položku `color`, která může nabývat hodnot `red` a `black`. Podle barvy pak budeme mluvit o *červených* a *černých* vrcholech.

Další změnou je, že místo hodnoty `nil` budeme používat speciální vrchol `NIL`. Pro jednoduchost si můžeme situaci představit tak, vždy, když je potomek vrcholu z v obyčejném stromě roven `nil`, je ve stromu s obarvenými kořeny nahrazen jednou kopií vrcholu `NIL`. Ta má položku `parent` nastavenu na z.[4] Procedury se stromem samozřejmě musíme upravit tak, aby pracovali s `NIL` namísto `NIL`.

Strom s obarvenými vrcholy je red-black stromem, pokud platí následující čtyři podmínky.

1. Kořen je černý.
2. `NIL` je černý.
3. Pokud je vrchol červený, oba jeho potomci jsou černí.
4. Pro každý vrchol platí, že všechny cesty z něj do listů obsahují stejný počet černých vrcholů.

Dále v textu je budeme zmiňovat pomocí čísel, tj podmínka 1, podmínka 2 atd.

Nyní dokážeme, že red-black stromy mají logaritmickou výšku. Pro vrchol x zavedeme jeho černou výšku $\textsc{bh}(x)$ jako počet černých vrcholů na cestě z x do listů, přičemž samotný x se do toho nepočítá. Speciálně pak zavedeme $\textsc{bh}\texttt{NIL} = 0$. Všimněme si, že v red-black stromu je to korektní definice, všechny cesty do listů obsahují stejný počet černých vrcholů.

Věta 6.8

Podstrom kořenem x má nejméně $2^{\textsc{bh}(x)} - 1$ vrcholů, které nejsou `NIL`.

Důkaz. Důkaz povedeme indukcí přes výšku podstromu (obyčejnou výšku, nikoliv černou výšku).

Pokud je výška podstromu rovna 0, obsahuje pouze x, který je `NIL`. Přitom máme $\textsc{bh}(x) = 0$ a $2^0 - 1 = 0$.

Předpokládejme, že podstrom s kořenem x má výšku $h > 0$ a že pomocné tvrzení platí pro všechny (pod)stromy s menší výškou.

Víme, že x není `NIL`. Potom levý a pravý podstrom x mají výšku nejvýše $h-1$. Přitom je jejich černá výška $\textsc{bh}(x) - 1$. (Pokud je potomek x červený, je jeho černá výška $\textsc{bh}(x)$, je-li černý, je to $\textsc{bh}(x)-1$.) Z indukčního předpokladu tak máme, že podstrom s kořenem x má nejméně

$$2 \cdot (2^{\textsc{bh}(x)-1} - 1) + 1 = 2^{\textsc{bh}(x)} - 1$$

vrcholů, které nejsou `NIL`. □

Věta 6.9: Výška red-black stromu

Red-black strom s n vrcholy, které nejsou `NIL`, má výšku nejvýše $2\lg(n+1)$.

Důkaz. Z podmínky 3 plyne, že černé vrcholy tvoří nejméně polovinu vrcholů na cestě z kořene stromu do listu. Černá výška kořene je tedy nejméně $h/2$, kde h je výška stromu. S použitím předchozí věty tak máme $n \geqslant 2^{h/2} - 1$. Úpravami dostaneme $h \leqslant 2\lg(n+1)$. □

[4] Při implementaci používáme pouze jednu kopii `NIL`, při práci se stromem jí musíme ve správný moment nastavit toho pravého rodiče. To je ovšem technický detail, který vynecháme.

Algoritmus vkládání vrcholu nejdříve vloží vrchol pomocí `tree-insert` (po úpravě pro `NIL` vrcholy). Poté ve stromu zkontroluje, jestli jsme porušili některou z podmínek ve definic red-black stromu a pokud ano, pak strom opraví. Připomeneme si nejdříve struktury pro vrchol, pro samotný red-black strom a zavedeme výčtový typ pro barvu vrcholu.

```
enum Color { red, black } — výčtový typ pro barvy
```

```
struct Node
   id: Key
   left: Node
   right: Node
   parent: Node
   color : Color — barva
```

```
struct RB-tree
   root : Node — kořen stromu
```

Vkládání vrcholu do red-black stromu realizuje následující procedura.

```
rb-insert
 ⟵ t: RB-tree
 ⟵ added: Node
```

```
1  added.left ← NIL
2  added.right ← NIL
3  added.color ← red
4  tree-insert(t, added)
5  rb-fixup(t, added) — procedura, která opraví strom
```

Vkládanému vrcholu nastavíme červenou barvu, oba jeho potomky nastavíme na `NIL`. Po vložení algoritme pro klasického binárního vyhledávací stromu mohou být porušeny podmínky 1 nebo 3. Toto opravíme pomocí procedury `rb-fixup`.

```
rb-fixup
 ⟵ t: RB-tree
 ⟵ z: Node — aktuální vrchol
```

```
1  z ≠ t.root
2     if (z.parent.color = black)
3        return – algoritmus končí
4     z ← local-fix(t,z) — procedura pro opravy, viz dalsi slajdy
5  t.root.c ← black
```

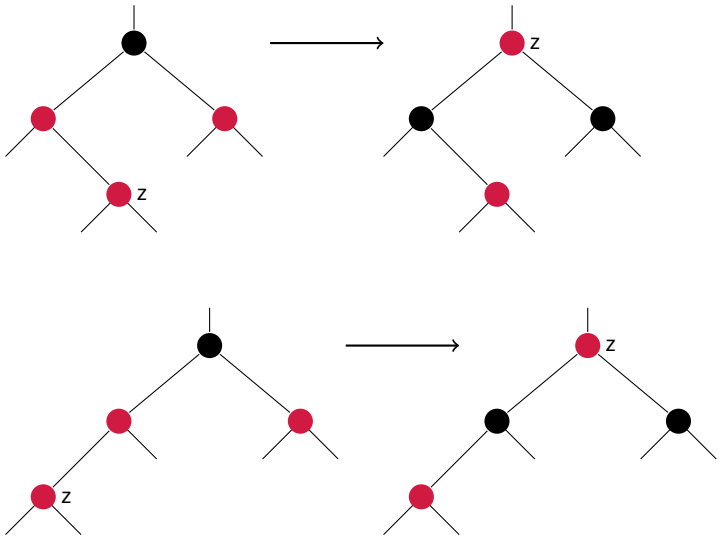
Platnost podmínky 1 lze obnovit snadno, stačí přebarvit kořen na černý (čtenář snadno ověří, že tím neporušíme platnost zbylých podmínek). Toto obnovení dělá `rb-fixup` na posledním řádku.

Podmínku 3 porušíme, pokud má přidaný vrchol červeného potomka. Toto opravíme pomocí `local-fix(t,z)`. Tato procedura bere jako argument červený vrchol, jehož ro-

dič je také červený, a situaci ve stromu opraví lokálními úpravami. Potom vrátí vrchol, který je červený a může mít červeného rodiče. Jeho podstromy už jsou ale opraveny tak, aby vyhovovali definici red-black stromu (s výjimkou podmínky 1). Pokud vrácený vrchol skutečné červeného rodiče má, v příští iteraci cyklu cyklu `while` algoritmus neskončí řádkem 3, a pro vrchol je také zavolána procedura `local-fix`. Princip fungování `local-fix` ovšem zajistí, že vrácený vrchol má menší hloubku než její argument. Cyklus `while` tedy jednou skončí. Nakonec je kořen obarven na černo (pokud tedy `local-fix` obrarví kořen na červeno, poslední řádek `rb-fixup` to opraví). Za předpokladu, že je `local-fix` korektní je strom po skončení `rb-fixup` red-black stromem. Proceduru `rb-insert` totiž provádíme na red-black stromu a mimo vrcholů, kterými tato procedura manipuluje, nemůže mít žádný červený vrchol červeného rodiče.

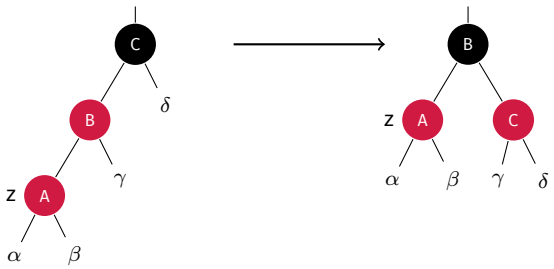
Zbývá vysvětlit proceduru `local-fix`. Jejím (druhým) argumentem je červený vrchol, který má červeného rodiče. V závislosti na barvách dalších vrcholů procedura detekuje jeden z případů, které mohou nastat, strom opraví a vrátí vrchol, se kterým má algoritmust `rb-fix` v opravách pokračovat. Proceduru vysvětlíme pomocí obrázků. Na levém obrázku je vždy zobrazena situace před opravou, na pravém obrázku potom situace po opravě. V obou obrázcích je pomocí `z` vyznačen vrchol, na levém obrázku je to argument `local-fix`, na pravém obrázku je to její návratová hodnota.

V prvním případě, když je strýc vrcholu `z` červený, stačí přebarvení. (Prarodič vrcholu `z` existuje, jeho rodič je totiž červený a nemůže být kořenem. Prarodič `z` musí být černý, protože červený vrchol nesmí mít červené potomky.)



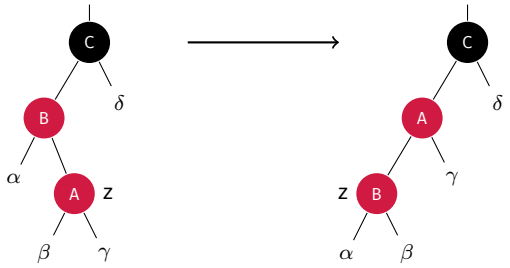
Abychom zkontrolovali korektnost úpravy, stačí si všimnout, že ve stromu napravo platí podmínka 3. Dále všem vrcholům po úpravě zůstala stejná černá výška jako před úpravou, s výjimkou kořene, kterému se černá zvětšila o 1. Černá výška rodiče kořene (pokud existuje) ale zůstává stejná.

Ve druhém případě, pokud je strýc vrcholu `z` černý (na obrázku je to kořen podstromu δ) a `z` je levý potomek, provedeme pravou rotaci na vrchol `C` a prohodíme barvy vrcholů `B` a `C`.



Černá výška žádného z podstromů α, β, γ, δ nemění. Černá výška vrcholu `B` po změně se tak rovná černé výšce vrcholu `C` před změnou. Černé výšky všech vrcholů ve stromu tak zůstavají stejné. Úprava tedy neporušila podmínku 4, podmínka 3 také platí. Stojí za povšimnutí, že vrchol `z` napravo má černého rodiče. Procedura `rb-fixup` tak v další iteraci skončí.

Ve třetím případě, pokud je strýc vrcholu `z` černý (na obrázku je to kořen podstromu δ) a `z` je pravý potomek, převedeme levou rotací vrcholu `B` situaci na předchozí případ. (Ten je vyřešen zavoláním `local-fix` v další iteraci procedury `rb-fixup`).



Úprava očividně nezmění černou výšku žádného z vrcholů v podstromu, neporušíme tedy podmínku 4.

V předchozích třech případech je `z` v levém podstromu svého prarodiče. Pokud je v jeho pravém podstromu, děláme symetrické úpravy. Mezi druhým a třetím případem tak rozlišujeme podle toho, jesli míří vrchol `z` *vně stromu*: je stejným potomkem svého rodiče jako je rodič `z` potomkem prarodiče `z`, tj. buď jsou oba levými potomky nebo oba pravými potomky. Třetí případ nastává, pokud vrchol `z` míří *dovnitř stromu*: vrchol `z` a rodič `z` jsou různými potomky.

Složitost operace `rb-insert` je lineárně závislá na výšce stromu. Po vlastním vložení do stromu, jehož složitost je lineárně závislá na výšce, provedeme buď maximálně 2 rotace, případně se při přebarvování posuneme od přidaného vrcholuu do kořene. Protože je výška red-black stromu omezena logaritmem (viz první věta v této části), je složitost operace $\Theta(\lg n)$.

Algoritmus odebírání vrcholu nejdříve odebere vrchol stejně jako z klasického binárního vyhledávacího stromu. Abychom poté mohli opravit případná porušení podmínek pro red-black strom, potřebujeme upravit procedury `node-swap` a `tree-delete` tak, abychom měli informaci o tom, na kterém místě došlo ve stromu ke změně a současně jsme zachovali barvy vrcholů tak, aby odpovídali situaci před odebrání vrcholu. Za tímto účelem provedeme upravíme procedury následovně.

Procedura `node-swap` zkopírujeme hodnotu položky `color` z vrcholu který nahrazujeme do vrcholu kterým nahrazujeme.

Procedura `tree-delete` odebírající vrchol `z` vrátí dvojici `(u,c)`, kde `u` je potomek skutečně odebraného vrcholu a `c` je barva skutečně odebraného vrcholu. Skutečně odebíraný vrchol je v situaci, kdy má `z` jednoho potomka, samotný vrchol `z`. Pokud má `z` dva potomky, je skutečně odebíraným vrcholem pořádkový následník vrcholu `z`.

Procedura odebrání nejdříve zavolá `tree-delete` a poté, pokud byl skutečně odebraný vrchol černý, opraví strom s pomocí procedury `rb-delete-fixup`. Odebrání červeneho vrcholu neporuší žádnou z podmínek z definice red-black stromu. Naopak, odebrání černého vrcholu poruší podmínku 4.

```
rb-delete
⟵ t: RB-tree
⟵ z: Node — odebíraný vrchol
```

```
1  u, col ← tree-delete(t,z)
2  if (col = red),
3     return
4  rb-delete-fixup(t,u)
```

Jako pomůcku pro pochopení `rb-delete-fixup` si můžeme představit, že černou barvu ze smazaného vrcholu uložíme do žetonu, kterým budeme pohybovat po vrcholech stromu. Žeton budeme chápat jako poukázku na přebarvení červeného vrcholu na černý, kterou využijeme, pokud se žeton dostane do červeného vrcholu, nebo ji zahodíme, pokud žeton dostane do kořene. Úpravy stromu prováděné `rb-delete-fixup(t,u)` přitom budou udržovat invariant, že pokud započítáme žeton jako černý vrchol, bude strom red-black stromem. Pokud se tedy žeton dostane do červeného vrcholu, můžeme ho bez obav přebarvit na černý. Na začátku žeton položíme do potomka skutečně odebraného vrcholu, přičemž preferujeme potomka, který není `NIL`.

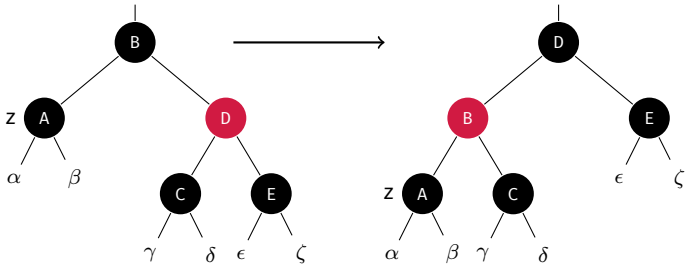
```
rb-delete-fixup
⟵ t: RB-tree
⟵ z: Node — aktuální vrchol (vlastnící žeton)
```

```
1  while (z ≠ t.root)
2     if (z.color = red)
3        z.color ← black
4        return
5     z, quit ← local-delete-fix(t,z) — procedura pro opravy, viz dalsi slajdy
6     if quit
7        return
```

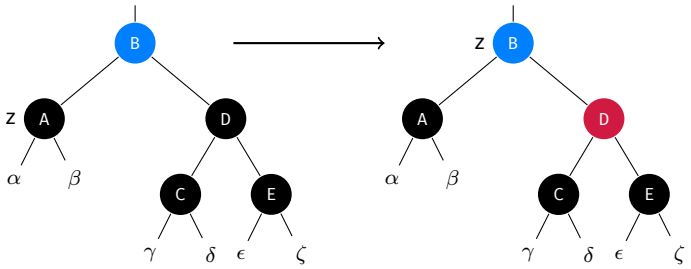
Procedura v cyklu s pomocí `local-delete-fix` opravuje strom. Žeton je vždy uložen ve vrcholu `z`. Na řádcích `2` až `4` ošetřujeme situaci, ve které je žeton na červeném vrcholu. Procedura `local-delete-fix` vrací dvě hodnoty: vrchol, na který byl přesunut žeton a příznak, jestli je možné s opravami stromu skončit. Bude přitom platit, že v obou podstromech vrcholu `z` platí podmínky 2 až 4. Toto plyne indukcí z korektnosti úprav provedených `local-delete-fix`. Čtenář si může explicitní důkaz tohoto tvrzení po prostudovaní této procedury napsat jako cvičení.

Procedura `local-delete-fix` detekuje jeden z možných případů. Poté provede úpravy a vrátí vrchol, kde mají úpravy pokračovat. Případy a jejich řešení si ukážeme na obrázcích. Pokud je na obrázku po úpravě (obrázek vpravo) označen nějaký vrchol `z` je tento vrchol vrácen spolu s hodnotou `false`, jinak je vráceno `true` (první návratová hodnota je libovolný vrchol). Jednotlivé případy rozlišujeme podle barvy sourozence vrcholu `z` a barev jeho potomků.

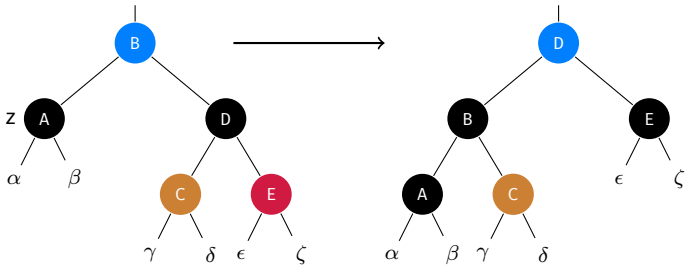
První případ nastane, pokud je sourozenec vrcholu `z` červený. S pomocí rotace s přebarvením přejdeme ke stromu, ve kterém je sourozenec `z` černý. V levém stromu musí být vrchol `B` černý, protože `D` je červený.



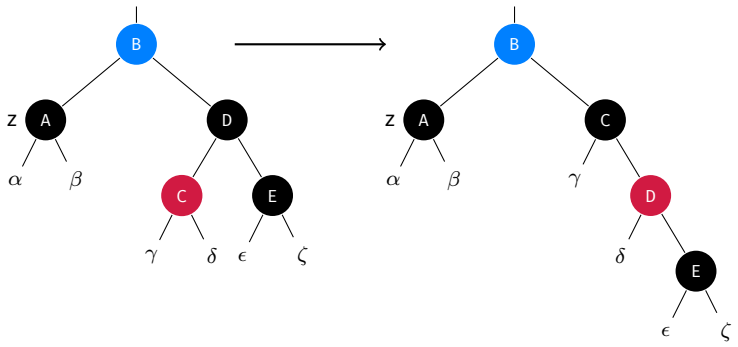
Ve druhém případě je sourozenec vrcholu `z` černý. Rodič vrcholu `z` může mít libovolnou barvu, která je na obrázcích zachycena modře. Oba potomci sourozence jsou černí. Sitaci vyřešíme přebarvením a přesunem žetonu do rodiče vrcholu `z`.



Ve třetím případě je sourozenec vrcholu `z` černý, rodič vrcholu `z` může mít libovolnou barvu (na obrázku zachycena modře) a pravý potomek sourozence je červený. Levý potomek sourozence může mít libovolnou barvu, na obrázku je zachycena bronzově. Po rotaci s prohozením barev v pravém podstromu vrcholu `D` chybí pro platnost podmínky 4 jeden černý vrchol, přebarvíme tedy `E` na černo a zbavíme se tak žetonu. Úpravy stromu mohou skončit.



Ve čtvrtém případě je sourozenec vrcholu `z` černý, rodič vrcholu `z` může mít libovolnou barvu (na obrázku zachycena modře) a pravý potomek sourozence je černý. Jeho levý potomek tak musí být červený. Jednou rotací s přebarvením situaci překlopíme na předchozí případ.



Ve všech čtyřech případech můžeme na obrázcích zkontrolovat korektnost úprav. Pokud při započítání žetonu ve stromu nalevo obsahují pro každý vrchol cesty z něj do podstromů α až ζ stejný počet černých vrcholů, platí to i pro strom napravo. Úprava tedy zachovává podmínku 4. Žádná z úprav také neporuší žádnou z ostatních podmínek z definice red-black stromu, pokud ní platila.

Procedura `local-delete-fix` pracuje v konstantním čase. Procedura `rb-delete-fixup` buď posloupností nejvýše tří iterací (první, třetí a čtvrtý případ pro `local-delete-fix`) skončí, nebo se aktuální vrchol posune po nejvýše dvou úpravách (první a druhý případ pro `local-delete-fix`) blíže ke kořeni. Celkem je tedy složitost `rb-delete-fixup` lineární vzhledem k výšce stromu. Díky větě o výšce red-black stromu pak máme, že složitost odebírání vrcholu je $\Theta(\lg n)$.

V některých aplikacích jsou vyhledávací stromy uloženy na pomalém médiu, např. na disku, a při operacích s nimi je nutné vrcholy přesouvat z disku do operační paměti. Skutečnou rychlost běhu operace na počítači pak nejvíce brzdí právě tyto přesuny. Přesun mezi pamětmi různých rychlostí se typicky dělá po blocích. Je výhodné naplnit blok co nejvíce a minimalizovat tak počet přesunů. Protože při operacích se stromy přesuváme jeden vrchol, naplnění bloku maximálním počtem klíčů dosáhneme tím, že v jednom vrcholu stromu budeme udržovat více klíčů, nikoliv právě jeden, jak tomu je u binárních vyhledávacích stromů.

B strom je vyhledávací strom, který parametrizujeme přirozeným číslem $t > 2$. (Hodnota tohoto čísla má vliv na počet klíčů v jednom vrcholu, a nastavíme ji tak podle velikostí bloku, který lze přesunou mezi pamětmi). B strom je definován následujícími podmínkami

1. *Počet klíčů ve vrcholech*
   V každém vrcholu stromu je maximálně $2t - 1$ klíčů a minimálně $t - 1$ klíčů. Výjimkou je kořen, kde může být méně než $t - 1$ klíčů.
2. *Počet potomků*
   Pokud vrchol obsahuje n klíčů, má 0 (pak je to list) nebo $n + 1$ potomků.
3. *Hloubka listů*
   Všechny listy jsou ve stejné hloubce.
4. *Podmínka uspořádání*
   Klíče jsou ve vrcholu uspořádány vzestupně. Označíme-li klíče $k_0 < k_1 < \ldots k_{n-1}$ a podstromy $\alpha_0, \alpha_1, \ldots, \alpha_n$, pak
   - pro $0 \leqslant i < n - 1$ jsou klíče v podstromu α_i menší než k_i,
   - pro $0 < i \leqslant n$ jsou klíče v podstromu α_i větší než k_{i-1}.

Podmínky 1 až 3 vynutí to, že B strom má vzhledem k počtu klíčů logaritmickou výšku (viz následující věta). Podmínka 4 umožňuje ve stromu efektivně vyhledávat.

Věta 7.1: O výšce B-stromu

B strom s parametrem t obsahující $n \geqslant 1$ klíčů má výšku nejvýše $\log_t \frac{n+1}{2}$.

Důkaz. Kořen obsahuje alespoň jeden klíč, ostatní vrcholy obsahují alespoň $t - 1$ klíčů. Strom tak má aspoň 2 vrcholy v hloubce 1, aspoň $2t$ vrcholů v hloubce 2, a obecně aspoň $2t^{i-1}$ vrcholů v hloubce i. Pro n tedy platí

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

Odtud dostaneme $t^h \leqslant (n+1)/2$ a obě strany logaritmujeme. □

Ve zbytku kapitoly si ukážeme operace vyhledávání, vkládání a odebírání klíče z B stromu. Pro vrchol použijeme následující strukturu

```
struct Node(t)
   id: [2t-1]Key — pole klíčů, uspořádány vzestupně
   children: [2t]Node — pole potomků
   parent: Node
   n: Int — počet klíčů ve vrcholu
   leaf: Bool — příznak toho, jestli je vrchol list

struct Tree
   root: Node — kořen stromu
```

Klíče jsou uloženy v poli `id` uspořádány vzestupně. Indexy v tomto poli a v poli `children` odpovídají indexům z podmínky `4` v definici B stromu. Kořen má položku `parent` nastavenu na `nil`. V prázdném B-stromu není položka `root` rovna `nil`, ale je to `Node`, jehož položka `n` je rovna 0.

Vyhledávání v B stromu je zobecněním vyhledávání v binárním vyhledávacím stromu. Nejdříve se pokusíme klíč najít v aktuálním vrcholu, za tím účelem projdeme pole `id` klíčů. Využijeme toho, že klíče jsou v tomto pole uspořádány vzestupně. Pokud klíč v tomto poli nenajdeme, index, kde prohledávání `id` skončilo neúspěšně, je indexem potomka, ve kterém máme s vyhledáváním pokračovat. Vyhledávání skončí neúspěšně, pokud je aktuální vrchol listem a přitom jsme v něm klíč nenašli.

```
search
⟵ x: Node
⟵ k: Key
⟶ Node — vrchol, ve kterém se nachází klíč nebo nil
⟶ Int — index, na kterém se klíč nachází v polozce id
```

```
1  i ← 0
2  while (i < x.n) and (k > x.id[i])
3     i ← i + 1
4  if (i < x.n) and (k == x.id[i])
5     return x, i
6  if x.leaf
7     return nil, i
8  return search(x.children[i], k)
```

Cyklus na řádku 2 skončí pokud `i = x.n` nebo `k ⩽ x.keys[i]`. Otestujeme, který z těchto důvodů to byl. Pokud to byl ten druhý a navíc `k == x.keys[i]`, nalezli jsme požadovaný klíč (viz řádek `4`). Na řádcích `7` a `8` ošetříme situaci, kdy hledaný klíč není ve vrcholu `x`. Korektnost operace plyne z podmínky uspořádání. Její složitost je $O(t \log_t n)$.

Při vkládání klíče nejdříve nalezneme vrchol, do kterého vkládaný klíč patří. K tomu použijeme následující operaci, jejíž kód si čtenář doplní jako cvičení. Je to jednoduchá úprava operace vyhledávání.

```
find-insertion-vertex
⟵ r:node
⟵ k:key
⟶ node — vrchol, do kterého vložíme, nutně list
```

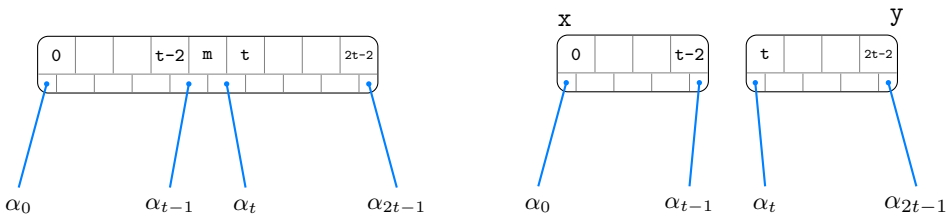
Předpokládejme, že vkládáme klíč `k` a že výsledkem volání `find-insertion-vertex` je `x`. Pokud bychom se mohli spolehnout, že `x.n < 2t-1`, pak by stačilo provést následující.

```
1  i ← 0
2  while (i < x.n) and (k > x.id[i]) do i ← i + 1
3  od indexu i posuneme obsah pole x.id o jedno políčko doprava
4  x.id[i] ← k
5  x.n ← x.n + 1
```

Vkládáme do listu a nemusíme se starat o potomky. Posun na řádku 3 můžeme bezpečně provést, protože pole `id` není obsazené a na posledním políčku není klíč.

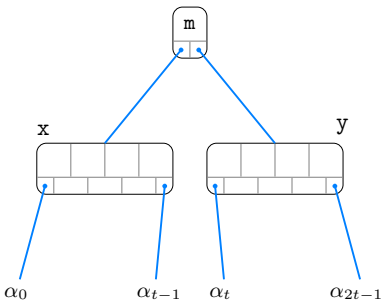
Pokud ovšem `x.n == 2t-1`, vložením klíče do `x` bychom v tomto vrcholu měli `2t` klíčů a porušili bychom podmínku 1 z definice B-stromu. (`x.id` také není dostatečně velké.) Situaci vyřešíme tak, že vrchol `x` rozdělíme.

Vrchol `x` má `2t-1` klíčů. Klíč `x.id[t-1]` je mediánem mezi klíči v poli `x.id`, označíme si jej `m`. Vytvoříme nový vrchol `y`, do kterého z vrcholu `x` přesuneme klíče na indexech `t` až `2t - 2` a potomky na indexech `t` až `2*t-1`. Rozdělení je zobrazeno na následujícím obrázku (místo klíčů jsou v polích jejich indexy).

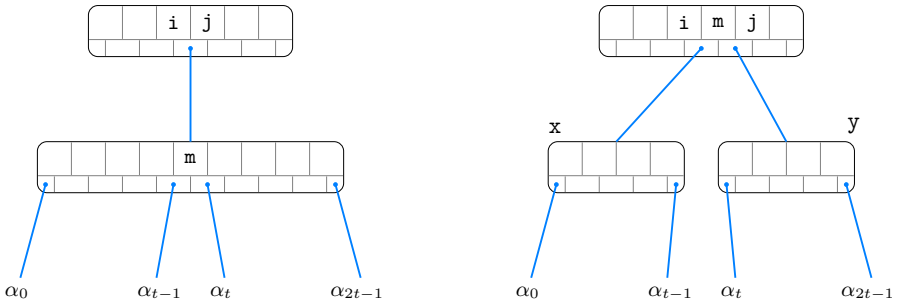


Nyní přidáme klíč `k` do správného vrcholu. Pokud `k < m` přidáme jej do `x`, jinak do `y`. Jeden z vytvořených vrcholů tak obsahuje `t-1` a druhý `t` klíčů. Klíč `m` vložíme do rodiče vrcholu `x`.

Speciální případ nastane, pokud je `x` před rozdělením kořenem. Situaci řešíme tak, že vytvoříme nový kořen a vložíme do něj `m` jako jediný klíč. Vrcholy `x` a `y` po rozdělení nastavíme jako potomky nového kořene.



Pokud `x` nebyl před rozdělením kořen, vložíme `m` do vrcholu `x.parent`, přičemž vrcholy `x` a `y` jsou potomci *okolo* klíče `m` (je-li `m` vložen na index `i`, jsou potomci *okolo* něj v poli `children` na indexech `i` a `i+1`).



Pokud je ovšem `x.parent` před vložením `m` zaplněný (tj. `x.parent.n == 2t-1`), musíme jej nejdříve právě popsanou metodou rozdělit a až poté vložit `m` (a potomky `x`, `y`) do správného z vrcholů vzniklých rozdělením. Při rozdělování `x.parent` dostaneme nový medián, který pak nutno zařadit do `x.parent.parent`. Můžeme tak provést celou kaskádu dělení vrcholů, při které rozdělíme zaplněné vrcholy na cestě do kořene. Může být nutné vytvořit i nový kořen. Toto je jediný způsob, jakým lze zvýšit výšku B stromu.

Při implementaci vkládání je klíčová procedura pro vložení klíč a potomků okolo něj do vrcholu.

```
insert-with-subtrees
⟵ tr: Tree
⟵ x: Node — vrchol, do kterého vkládáme
⟵ k: Key — vkládaný klíč
⟵ left: Node — podstrom nalevo od klíče
⟵ right: Node — podstrom napravo od klíče
```

Tuto proceduru použijeme i pro `insert` (s argumenty `tr` a `k`).

```
1 target ← find-insertion-vertex(tr.root, k)
2 insert-with-subtrees(tr, target, k, nil, nil)
```

Následuje kód procedury `insert-with-subtrees`.

```
if x == nil — Půlení kořene
   z ← Node()
   z.id[0] ← k
   z.leaf ← false
   z.n ← 1
   z.parent ← nil
   set-child(z, left, 0)
   set-child(z, right,1)
   tr.root ← z
   return

if x.n == 2*t-1 — x je zaplněný
   p ← x.parent
   rozděl vrchol x a tím získej klíč m a vrcholy l a r.
   if k < m
      target ← l
   else
      target ← r
   insert-with-subtrees(tr, target, k, left, right)
   insert-with-subtrees(tr, p, m, l, r)
   return

if x.n < 2*t-1 — x není zaplněný
   i ← 0
   while (i < x.n) and (k > x.id[i]) do i ← i + 1
   od indexu i posuneme obsah polí x.id a x.children o jedno políčko doprava
   x.id[i] ← k
   set-child(x, left, i
   set-child(x, right, i+1
   x.n ← x.n + 1
```

Procedura `set-child(x, y, i)` nastaví potomka na indexu `i` vrcholu `x` na vrchol `y`.

Procedura `find-insertion-node` má stejnou složitost jako procedura vyhledávání. Půlení vrcholu provádíme v každé úrovni stromu nejvýše jednou, přitom jedno půlení má složitost lineárně závislou na t. Proto je složitost přidávání $O(t \cdot \log_t n)$, kde n je počet klíčů ve stromu.

Výše popsaný algoritmus přidávání klíče můžeme označit jako dvouprůchodový. Nejdříve projde procedura `find-insertion-node` strom od kořene do listu, poté v nejhorším případě dojde ke kaskádě dělení vrcholů od listu až do kořene. Algoritmus lze upravit tak, aby byl jednoprůchodový a zaplněné vrcholy rozděloval už po cestě od kořene do listu. Výhodou je, že se sníží počet přesunů mezi pomalou a rychlou pamětí.

V algoritmu zajistíme, že pokud vkládáme do vrcholu `x`, tak `x` není zaplněný.

- Pokud je kořen zaplněný, rozdělíme jej a vytvoříme nový kořen. Proceduru přidávání spustíme v tomto novém koření.
- Předp. že aktuálně přidáváme do nezaplněného vrcholu `x`. Pokud je `x` list, pak můžeme klíč přidat bez nutnosti dělení.
  Pokud `x` není list a máme pokračovat do potomka `y`, nejdříve zkontrolujeme, jestli je `y` zaplněný. Pokud ano, rozdělíme jej. Toto rozdělení nemůže vést k potřebě rozdělit `x`, protože `x` není zaplněný. Algoritmus potom pokračuje jedním z vrcholů, které vznikly dělením `y`. Připomeňme, že dělením vzniklý vrchol není zaplněný.

Složitost je $O(t \cdot \log_t n)$, strom projdeme od kořene do listu, přitom může být v každé úrovni potřeba dělit vrchol. Dělení vrcholu má složitost lineárně závislou na `t`.

Odebrání klíče `k` ze stromu `tree` má 2 fáze. V první fázi klíč skutečně smažeme. Ve druhé fázi algoritmu mazání upravujeme nekorektní počty klíčů ve vrcholech, které mohli díky první fázi vzniknout.

Samotné odebírání je analogií odebírání vrcholu z binárního vyhledávacího stromu. Nejdříve pomocí operace `search` najdeme vrchol `x`, ve kterém se `k` nachází a index `i`, na kterém se `k` nachází v poli `x.id`. Pokud je `x` listem, v poli `x.keys` posuneme prvky s indexem větším než `i` o jedno políčko doleva. Položku `x.n` dekrementujeme o 1. Pokud `x` není listem, najdeme v podstromu `x.children[i+1]` minimální klíč `m`. Toto je pořádkový následník klíče `k` ve stromu. Tímto klíčem nahradíme ve vrcholu `x` klíč na indexu `i`. Potom smažeme klíč `m` z listu, ve kterém se nacházel.

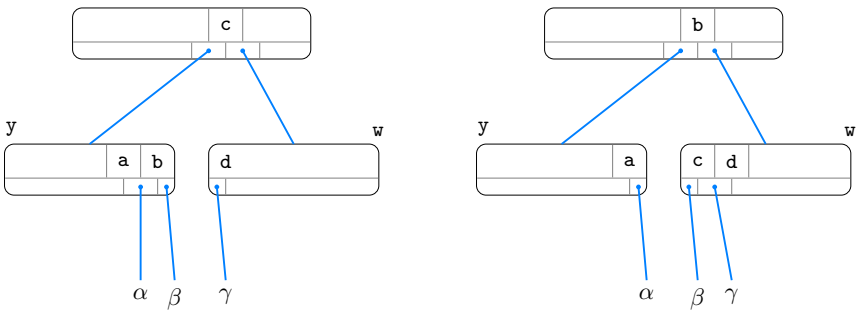
V obou případech ubude klíč v listu. Může se stát, že po smazání má list `t-2` klíčů a došlo k porušení podmínky 1 z definice B-stromu. Toto napravíme ve druhé fázi algoritmu.

Aktuálně upravovaný vrchol označíme `w`. Na začátku fáze je `w` list, ze kterého jsme odebrali klíč v první fázi. Algoritmus potom probíhá následovně.

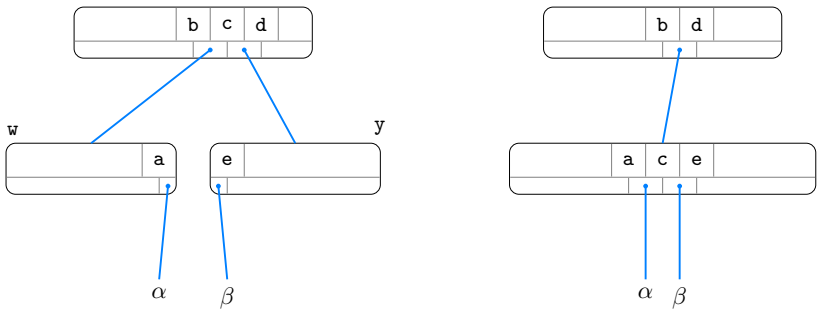
1. *Můžeme skončit?*
   Pokud je `w` kořen, nebo `(w.n ⩾ t-1)` algoritmus končí.

2. *Převod klíče ze sourozence*
   Pokud `w` má sourozence `y`, jehož index v rodičově poli `children` se liší o 1 (takovému sourozenci budeme říkat soused), takového, že `(y.n > t-1)`,

   přesuneme do `w` jeden klíč z vrcholu `w.parent`, a ten zase nahradíme klíčem z vrcholu `y`. Mezi `y` a `w` musíme také přesunout jednoho potomka (tj. provedeme analogii rotace). Vrcholu `y` ubude jeden klíč. Existují dvě symetrické varianty této úpravy (podle toho, jestli je `y` napravo nebo nalevo od `w`), níže je zobrazena jedna z nich.



3. *Sloučení se sourozencem*
   Pokud nelze provést předchozí bod, protože sousedi vrcholu `w` obsahují `t-1` klíčů, vybereme si jednoho ze sousedů, označíme ho `y`, a spojíme `w` a `y` do nového vrcholu `z`. K tomu musíme z rodiče vrcholu `w` odebrat klíč, který leží mezi `w` a `y` a vložit jej do nového vrcholu. Viz následující obrázek.



Pokud je `w.parent` kořen, ze kterého jsme odebrali poslední klíč, je novým kořenem `z`. Jinak nastavíme `w` ← `z` a pokračujeme krokem 1.

První průchod je vyhledávání klíče (a případná operace vyhledání minima). Ten je v čase $O(t \log_t n)$. Samotné smazání klíče z vrcholu je v čase $O(t)$. V druhé fázi procházíme strom směrem ke kořeni, v každém level děláme $O(t)$ práce. Celkem tedy $O(t \log_t n)$. Složitost odebírání klíče je tak $O(t \log_t n)$.

Operaci odebrání klíče popsána výše je dvouprůchodová (první průchod od kořene do listu je ve fázi 1, druhý průchod od listu do kořene může nastat ve fázi 2). Operaci lze upravit tak, aby byla jednoprůchodová. Úprava spočívá v tom, že zabráníme situaci, ve které má nějaký vrchol méně než `t-1` klíčů tak, že při prvním průchodu preventivně spojíme vrcholy, které mají `t-1` klíčů.

Algoritmus pracuje s aktuálním vrcholem `x`, maže klíč `k`. Budeme předpokládat, že `x` obsahuje alespoň `t` klíčů a po odebrání jednoho klíče jich tak bude mít pořád dostatek.

1. Pokud je `x` list obsahující `k`, klíč smažeme standardním způsobem.
2. Pokud `x` není list a obsahuje klíč `k`
   (a) *Levý potomek* `y` *klíče* `k` *má alespoň* `t` *klíčů.*
   Najdeme pořádkového předchůdce `l` klíče `k` a rekurzivně jej smažeme z vrcholu `y`. To lze provést jedním průchodem tak, že pokračujeme jakobychom mazali `l` z `y`. Klíč `l` sice v ten momen neznáme, víme ale, kde ve stromu je. Je to maximum v podstromu s kořenem `y`. V momentě, kdy `l` smažeme a známe tak jeho identitu, nahradíme jím ve vrcholu `x` klíč `k`.
   (b) *Pravý potomek* `z` *klíče* `k` *má alespoň* `t` *klíčů.*
   Analogicky předchozímu bodu, pouze hledáme pořádkového následníka.
   (c) *Oba potomci (*`y` *a* `z`*) mají* `t - 1` *klíčů.*
   Spojíme `y` a `z` do jednoho vrcholu `w`, tak jako ve dvouprůchodovém mazání. Tím z vrcholu `x` ztratíme klíč `k`, který se přesune do `w` (ale `x` má pořád dostatek klíčů). Rekurzivně mažeme klíč `k` z vrcholu `w`.
3. Pokud `x` není list a neobsahuje klíč `k`. Klíč `k` se nachází v podstromu, jehož kořenem je potomek `y` vrcholu `x`.
   (a) `y` *má* `t-1` *klíčů a má souseda* `z` *s alespoň* `t` *klíči.*
   Rotací převedeme jeden klíč z vrcholu `z`, tak jako při dvouprůchodovém mazání.
   (b) `y` *má* `t-1` *klíčů a nemá souseda s alespoň* `t` *klíči.*
   Sloučíme `y` s jedním sousedem tak jako při dvouprůchodovém mazání (a ubereme tím jeden klíč z `x`). Výsledný vrchol označíme `y`.

   Rekurzivně mažeme klíč `k` z vrcholu `y`.

Pokud se v kroku 2 (c) nebo 3 (b) stane, že (v případě, že x je kořen obsahující jeden klíč) vytvoříme kořen bez klíčů, stane se v tomto kroku nově vytvořený vrchol novým kořenem stromu. Díky tomu je předpoklad, že x má dostatečný počet klíčů, v průběhu mazání vždy zajištěn.

Hašovací tabulka je datová struktura, která je optimalizovaná pro operace vkládání a vyhledávání, některé varianty umožňují i efektivní odebírání.

Základem tabulky je pole, do kterého jsou ukládány datové položky (klíče a jim přidružená data). Transformaci z obecného klíče zajišťuje *hašovací funkce*, která obecnému klíči přiřazuje platný index v poli.

Složitosti operací s hašovací tabulkou jsou v nejhorším případě lineární, ovšem za rozumných předpokladů lze v průměrném případě dosáhnout konstatní (amortizované) složitosti.

Pro množinu klíčů U, velikost tabulky m a hašovací funci h nastane kolize, pokud existují různé klíče $x, y \in U$ tak, že $h(u) = h(v)$. Kolize je nepříjemná tím, že na jeden index v poli patří dvě různé datové položky. Než se podíváme, jak se dají kolize v hašovací tabulce řešit, ukážeme si, že kolize nastávají velmi často.

Rychle vidíme, že pokud $|U| > m$, pak určitě existuje dvojice klíčů, pro které dojde ke kolizi (to je důsledek Dirichletova principu) a hašovací funkce, pro kterou by ke kolizi nedošlo, neexistuje. Zamysleme se tedy nad tím, ke kolika kolizím dojde pro jednu hašovací funkci dojde.

Předpokládejme pro jednoduchost, že U je konečná. Pro $i = 0, \ldots, m-1$ je počet kolizí v rámci indexu i roven

$$\frac{b_i \cdot (b_i - 1)}{2},$$

kde b_i je počet klíčů, pro které h dá výsledek i. Dohromady je kolizí

$$\sum_{i=0}^{m-1} \frac{b_i \cdot (b_i - 1)}{2}.?? \tag{8.1}$$

Položme například $|U| = 300$ a $m = 3$ a spočtěmě následující počty kolizí.

1. $b_0 = b_1 = b_2 = 100$.
   Počet kolizí je $3 \cdot \frac{100 \cdot 99}{2} = 14850$
2. $b_0 = 300, b_1 = b_2 = 0$
   Počet kolizí je $\frac{300 \cdot 299}{2} = 44850$
3. $b_0 = 200, b_1 = 80, b_3 = 20$
   Počet kolizí je $\frac{200 \cdot 199}{2} + \frac{80 \cdot 79}{2} + \frac{20 \cdot 19}{2} = 23250$

Příklad naznačuje, že čím je rozhložení klíčů mezi indexy rovnoměrnější, tím je počet kolizí menší. Čtenář snadno dokáže, že tomu tak skutečně je. (Například výraz (??) je minimální, pokud pro každé $i \neq j$ je $|b_i - b_j| \leqslant 1$.)

Situaci můžeme nahlédnou také s použitím pravděpodobnosti. Řekněme, že pro $x \in U$ je dána pravděpodobnost $p(x)$, že x vybereme. Představíme si následující pokus: náhodně vybereme klíč x a vypočítáme $h(x)$. Pravděpodobnost, že výsledkem pokusu je i je rovna

$$P[h(x) = i] = \sum_{x : h(x) = i} p(x)$$

Pokud platí, že pro každé $0 \leqslant i < m$ je $P[h(x) = i] \approx 1/m$, pak h označujeme jako *uniformní hašhovací funkci*. Za předpokladu $p(x) = \frac{1}{|U|}$ hašovací funkce z případu 1 předchozího příkladu uniformní.

Předpokládejme nyní, že máme uniformní hašovací funkci h a dostatečně velkou množinu klíčů. Zamysleme se nad tím, jaká je pravděpodobnost, že dojde ke kolizi v množině $n < m$ klíčů. Pravděpodobnost, že nedojde ke kolizi je dána

$$\overline{p}(n) = 1 \cdot (1 - \frac{1}{m}) \cdot (1 - \frac{2}{m}) \cdot \cdots \cdot (1 - \frac{n-1}{m}).$$

Po úpravách dostaneme

$$\overline{p}(n) = \frac{m!}{m^n \cdot (m-n)!}$$

Ke kolizi dojde s pravděpodobností $p(n) = 1 - \overline{p}(n)$. V následující tabulce jsou pro ilustraci spočítány některé hodnoty.

m / n	2	3	5	10	20	30
3	0.33	0.77				
100	0.01	0.02	0.09	0.37	0.86	0.99
200	0.00	0.01	0.04	0.20	0.62	0.89
365	0.00	0.01	0.03	0.11	0.41	0.71

8.1 Řetězení

Datové položky neukládáme přímo do pole. V poli máme spojové seznamy (libovolného typu), a datové položky ukládáme do nich. V seznamu na indexu `i` v poli jsou tak datové položky, pro jejichž klíče hašovací funkce vrátí `i`.

Uvážíme-li pro hašovací tabulku následující strukturu

```
struct Table
   data: []Key
   hash-function: (Key) → Int
```

vypadají operace přidání, vyhledání a smazání následovně.

```
insert
 ⟵ T: Table
 ⟵ k: Key
```

```
1 i ← T.hash-function(k)
2 list-insert(T.data[i], k)
```

```
search
 ⟵ T: Table
 ⟵ k: Key
 ⟶ Node — Vracíme data odpovídající klíči, tady pro ilustraci uzel seznamu
```

```
1 i ← T.hash-function(k)
2 return list-search(T.data[i], k)
```

```
delete
⟵ T: Table
⟵ k: Key
```

```
1 i ← T.hash-function(k)
2 list-delete(T.data[i], k)
```

Operace `insert` pracuje v konstantním čase. Operace `search` a `delete` mají v nejhorším případě složitost lineární. Před odebráním v `delete` vlastně proběhne celý `search`, stačí se proto zaměřit na `search`. Nejhorší případ nastane, pokud:

- Všechny klíče v tabulce jsou v jednom seznamu.
- Klíč vyhledáváme v tomto seznamu.
- Hledaný klíč se v tabulce nenachází (nebo je v seznamu poslední).

Věta 8.1

Pokud je `T.hash-function` uniformní hašovací funkce, je průměrná složitost vyhledávání v tabulce `T` obsahující n klíčů $\Theta(1 + \frac{n}{m})$.

Důkaz. (Kostra důkazu). Délku seznamu `T.data[j]` označíme n_j. Očekávaná hodnota této proměnné je

$$E[n_j] = \sum_{i=1}^{n} 1 \cdot \frac{1}{m} = \frac{n}{m}$$

Pro všechny indexy j platí, že se do nich hledaný klíč zahašuje se stejnou pravděpodobností. Při neúspěšném vyhledávání se musí prohledat celý seznam. Při úspěšném vyhledávání musíme v průměru prohledat půlku seznamu. □

Vidíme tedy, že pokud máme $n = O(m)$ (tj. ex. konstanta c tak, že $n < cm$), pak je průměrná složitost vyhledávání konstantní. Pro návrh tabulky s řetězením to znamená následující: Vybereme konstantu c. To odpovídá délce seznamu, kterou jsme ještě ochotni snést. Tabulka pak má kapacitu cm a více prvků do ní nepřidáváme (je to podobné fixnímu poli). Alternativně při přidání klíče do tabulky, která obsahuje cm klíčů, tuto tabulku zvětšíme a všechny její klíče znovu přídáme (je to analogie dynamického pole). Amortizovaná složitost přidání bude pořád konstantní.

8.2 Otevřené adresování

Datové položky jsou umístěny přímo v poli, prázdné políčko obsahuje hodnotu `nil`.

Pro účely vyhledávání a vkládání používáme *průzkumnou funkci*

$$g : U \times \{0, 1, \ldots, m-1\} \to \{0, 1, \ldots, m-1\},$$

Pro klíč $k \in U$ je *průzkumná posloupnost* daná tuto funkcí rovna

$$g(k, 0), g(k, 1), \ldots, g(k, m-1)$$

Řekneme, že průzkumná funkce projde pro k celou tabulku, pokud je průzkumná posloupnost klíče k permutací posloupnosti $0, 1, \ldots, m-1$.

Princip otevřeného adresování můžeme ukázat na operaci vyhledávání. Při vyhledávání indexu, na kterém je klíč k, prohledáváme políčka pole v pořadí daném průzkumnou posloupnosti pro k. Pokud je na aktuálním indexu klíč různý od k, pokračujeme v průzkumné posloupnosti dál. Pokud je na aktuálním indexu `nil`, klíč se v tabulce nenachází. Nakonec, pokud projdeme celou průzkumnou posloupnost, klíč se tabulce nenachází. Analogický postup použijeme pro přidávání uzlu: uzel přidáme na první index z průzkumné posloupnosti, na kterém je `nil`. Podobnosti jsou v následujícím pseudokódu.

```
struct table
   data:[]Key
   g: (Key, Int) → Int — průzkumná funkce
```

```
search
⟵ T: Table
⟵ k: Key
⟶ Int — index, na kterém se nachází k nebo nil
```

```
1  for i in 0 ..< len(T.data)
2     j ← T.g(k,i)
3     if (T.data[j] = k) then return j
4     if (T.data[j] = nil then return nil
5  return nil
```

Přidávání klíče je analogické, pouze na řádku 4 vložíme klíč na index `j`.

Odstranění klíče z tabulky pouhým nahrazením `nil` by způsobilo přerušení průzkumné posloupnosti při vyhledávání. Políčko v tabulce proto nenastavíme na `nil`, ale dáme mu příznak, že je prázdné. Potom musíme příslušným způsobem upravit procedury pro přidávání (přidáváme i na místo, kde bylo předtím mazáno) a prohledávání (smazané prvky přeskakujeme).

Podívejme se nyní, jak lze zkonstruovat průzkumné funkce.

Lineární prozkoumávání. Předpokládejme, že máme hašovací funkci h. K ní vytvoříme průzkumnou funkci

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

Průzkumná posloupnost tedy začíná $h(k)$ a pak kontrolujeme políčkou s o 1 větším indexem (mod m). Například pro $h(k) = 3$ a $m = 7$ je průzkumná posloupnost

$$3, 4, 5, 6, 0, 1, 2$$

Může nastat problém nazývaný primární shlukování: vzniknou dlouhé úseky za sebou jdoucích obsazených políček, které tak zvyšují průměrný čas nutný k vyhledávání. Důvod jejich vzniku je následující: prázdné políčko, které je za úsekem i obsazených políček, bude jako další zaplněno s pravděpodobností $(i+1)/m$. Dlouhé obsazené úseky tak mají tendenci se prodlužovat.

Pro g existuje pouze m různých průzkumných sekvencí, nicméně pro každý klíč projde průzkumná funkce celou tabulku.

Kvadratické prozkoumávání. Předpokládejme, že máme hašovací funkci h. K ní vytvoříme průzkumnou funkci

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

kde c_1, c_2 jsou vhodně zvolené konstanty (jsou-li neceločíslné, pak před operací $\mod$ zaokrouhlíme dolů.)

Pro některou kombinaci hodnot c_1, c_2 a m nemusí g prozkoumat celou tabulku (máme $i \neq j$ takové, že $g(k, i) = g(k, j)$). Pokud je např. m prvočíslo, pak většina voleb c_1, c_2 (např. 1,1 nebo 0,1). vede k tomu, že délka průzkumné posloupnosti před tím, než narazíme na opakující se hodnotu, je přibližně $m/2$.

Dochází k sekundárnímu shlukování, pokud máme $k_1 \neq k_2$, ale $h(k_1) = h(k_2)$, mají k_1 a k_2 stejné průzkumné sekvence. Existuje m různých průzkumných sekvencí.

Dvojité hašování. Pro hašovací funkce h_1, h_2 zavedeme průzkumnou funkci

$$g(k, i) = (h_1(k) + i h_2(k)) \mod m$$

Hašovací funkce h_1 určuje počáteční pozici, hašovací funkce h_2 určuje offset, o který se posouváme.

Aby g prohledala celou tabulku, musí být $h_2(k)$ a m nesoudělné. Například zvolíme m jako prvočíslo a

$$h_1(k) = k \mod m$$
$$h_2(k) = 1 + (k \mod m - 1)$$

Pro prvočíselné m existuje $\theta(m^2)$ průzkumných posloupností, protože každá možná dvojice $h_1(k), h_2(k)$ vede k rozdílné průzkumné posloupnosti.

Věta 8.2

Pokud je každá možná průzkumná sekvence stejně pravděpodobná, je v tabulce velikosti m organizované otevřeným adresováním a obsahující n klíčů průměrná délka části průzkumné posloupnosti, kterou projdeme při neúspěšném prohledávání, nejvýše $\frac{1}{1-\frac{n}{m}}$.

Velmi silný předpoklad předchozí věty žádný z uvedených přístupů nesplňuje. Empiricky se k ideální složitosti z věty blíží dvojité hašování.

8.3 KONSTRUKCE HAŠOVACÍCH FUNKCÍ

Výpočet výsledku hašovací funkce závisí na typu klíče. Typicky obsahuje proces převodu klíče na číslo v rozsahu, který odpovídá velikosti slova na daném počítači, a poté toto číslo transformuje na index.

Klíč k často bereme jako posloupnost číslic v nějaké číselné soustavě. Při programování se hodí například posloupnost číslic v soustavě o základu 256, to odpovídá posloupnosti bajtů, které odpovídají uložení klíče v paměti. Tuto posloupnost převedeme na číslo v rozsahu, který odpovídá velikosti slova na daném počítači. Takto získané číslo potom transformujeme na index z množiny $\{0, \ldots, m-1\}$. To lze provést například následujícími metodami.

- *Dělící metoda:* $x \mapsto x \mod m$
- *Metoda násobení:* $x \mapsto \lfloor m \cdot (xA - \lfloor xA \rfloor) \rfloor$ pro $0 < A < 1$.

Uvedeme malý příklad. Předpokládejme, že čísla jsou jsou třípísmenné ASCII řetězce, které interpretujeme jako číslo o základu 128. Například řetězec `now` podle ASCII tabulky bereme jako posloupnost čísel `110 111 119`, kterou převedeme na $119 + (111 * 128) + (110 * 128^2) = 1816567$. Použijeme dělící metodu s hodnotami $m = 64$, $m = 31$, výsledné indexy vidíme pro `now` i další příklady slov v následujících tabulkách.

	64	31
`now`	55	29
`for`	50	20
`tip`	48	1
`ilk`	43	18
`dim`	45	21
`tag`	39	22

	64	31
`sob`	34	26
`nob`	34	8
`cab`	34	24
`sky`	57	2
`jay`	57	4
`joy`	57	29

V druhé tabulce si můžeme všimnout, že pokud slova končí stejným písmenem, jsou pro $m = 64$ hašována na stejný index. To není náhoda, dělící metoda pro m, které je mocninou 2 vrací jako výsledek odpovídající počet vstupních bitů vstupní hodnoty. Jako dobrá volba velikosti tabulky se tak při použití dělící metody jeví prvočísla. Typicke volíme největší prvočíslo menší než daná mocnina dvou (při zvětšování tabulky ji pak přibližně zdvojnásobujeme). Níže je tabulka s příklady takových prvočísel (a jim odpovídajících mocnin 2).

2^6	53
2^7	97
2^8	193
2^9	389
2^{11}	1543
2^{16}	49157

Převod klíče na číslo, který jsme použily v předchozím příkladu je naivní. V reálných aplikacích se používá sofistikovanějších metod. V tomto kurzu se jim ovšem věnovat nebudeme.

9.1 REPREZENTACE GRAFU V POČÍTAČI

Uvažujeme graf G s množinou vrcholů $V = \{0, 1, \ldots, n-1\}$ a množinou hran H. Graf lze uložit v paměti počítače různými způsoby, v této kapitole budeme diskutovat dva základní: seznamy sousedů a matici sousednosti.

Pro uložení pomocí *seznamů sousedů* potřebujeme n-prvkové pole `adj`. Na indexu `i` tohoto pole je seznam všech vrcholů, do nichž vede hrana z vrcholu `i`, tedy seznam obsahující všechny vrcholy z množiny $\{v \mid (u, v) \in H\}$. Součet délek všech seznamů obsažených v `adj` je u orientovaných grafů roven počtu hran, u neorientovaných grafů je to dvojnásobek počtu hran, v obou případech je tedy lineární vzhledem k počtu hran. Seznam sousedů se tedy hodí pro řídké grafy, tj. pro grafy které obsahují málo hran a pro situace, kdy potřebujeme efektivně procházet všechny sousedy nějakého vrcholu. Nevýhodou je, že nelze v konstantním čase testovat existenci nějaké konkrétní hrany; za tímto účelem musíme vyhledávat v některém ze seznamů. Složitost je tak lineární vzhledem k maximálnímu (výstupnímu) stupni vrcholu v grafu.

Matice sousednosti je dvourozměrné pole `A` velikosti $n \times n$. Hodnota `A[i][j]` se rovná `1`, pokud graf obsahuje hranu (i, j), jinak se rovná `0`. Matice vždy zabírá $O(n^2)$ paměti, a není výhodná pro řídké grafy. Projití všech sousedů vrcholu `j` má lineární složitost vzhledem k počtu vrcholů grafu: musíme totiž projít `A[j][i]` pro `i=0,1, ... ,n-1`, sousedy jsou ty vrcholy, pro které je tato hodnota rovna `1`. Na druhou stranu, existenci konkrétní hrany lze testovat v konstantním čase.

Ve zbytku kapitoly boudeme předpokládat reprezentaci grafu pomocí seznamů sousedů. Graf zachytíme následující strukturou.

```
struct Graph
   n — počet vrcholů
   adj — seznam sousedů
```

9.2 PRŮCHOD GRAFEM OBECNĚ

Uvažujeme datovou strukturu `M` pro uchování množiny vrcholů a následující operace na ní.

- `insert(M, x)`. Vloží vrchol `x` do `M`.
- `pop(M)`. Odebere jeden vrchol a vrátí jej jako výsledek. Blíže nespecifikujeme, který vrchol to je. Každá konkrétní struktura, kterou můžeme dosadit za `M`, to může mít jinak.
- `empty(M)`. Test prázdnosti `M`.

Obecný průchod od hloubky je algoritmus, který pro graf `G` a počáteční vrchol `s` navštíví každý vrchol, který je z vrcholu `s` dosažitelný, právě jednou. Přitom vrcholy navštěvuje systematicky: nově lze navštívit pouze sousedy již navštíveného vrcholu.

```
graph-search
 ⟵ g: Graph — reprezentace pomocí matice sousednosti
 ⟵ s: Int — počáteční vrchol
```

```
X ← [g.n]Bool — prvky inicializovány na false
vytvoříme prázdnou strukturu M
insert(M,s)
X[s] ← true
while (not empty(M))
   u ← pop(M)
   visit(u)
   foreach w in g.adj[u] such that X[w] = false
      X[w] ← true
      insert(M,w)
```

Ve zbytku kapitoly dokážeme několik základních vlastností obecného průchodu.

Věta 9.1

Volání `graph-search(G,s)` navštíví každý vrchol dosažitelný z vrcholu `s` právě jednou. Vrcholy, které dosažitelné nejsou, toto volání nenavštíví.

Důkaz. Vrchol je navštíven poté, co je odebrán z `M`. Vrchol je vložen do `M` pouze pokud je jeho příznak v poli `X` nastaven na `false`. Protože po vložení vrcholu do `M` nastavíme tento příznak na `true`, je vrchol vložen do `M` nejvýše jednou, a tedy navštíven nejvýše jednou.

Každý vrchol, který je vložen do `M` je eventuálně navštíven: Z předchozího víme, že každý vrchol je vložen nejvýše jednou, počet vrcholů je konečný a v každé iteraci cyklu `while` jeden vrchol z `M` odebíráme. Pokud se tedy vrchol `v` během běhu algoritmu ocitne v `M`, ocitnou se tam i jeho potomci. Buď jsou přidáni před iterací cyklu `while`, ve které je `v` navštíveným vrcholem, nebo jsou přidání v cyklu `foreach`. Vidíme tedy, že s každým navštíveným vrcholem jsou navštíveni i všechny vrcholy, které jsou z něj dosažitelné. Současně z předchozího plyne, že vrchol, který není dosažitelný z `s`, se nemůže dostat do `M` a tím pádem nebude navštíven. □

Předpokládejme, že složitost operace `insert` je dána funkcí f_i, složitost operace `pop` funkcí f_p a složitost `empty` funkcí f_e. V nejhorším případě je tedy složitost jednotlivých volání procedur omezena aplikací těchto funkcí na počet vrcholů grafu, protože v `M` může být každý vrchol nejvýše jednou.

Věta 9.2

Složitost `graph-search` pro graf $G = (V, E)$ je dána

$$O(|V| \cdot (f_i(|V|) + f_p(|V|) + f_e(|V|)) + |E|).$$

Důkaz. Každý vrchol je vložen či odebrán nejvýše jednou, cyklus `while` proběhne nejvýše $|V|$-krát (každý vrchol je do `M` vložen nejvýše jednou, viz důkaz předchozího tvrzení). Počet iterací cyklu `foreach` je dán součtem délek seznamů v poli `g.adj`, pro každý vrchol totiž procházíme seznam jeho sousedů nejvýše jednou. Tento součet je $O(|E|)$. □

Algoritmus průchodu rozšíříme o mechanismus, díky kterému uvidíme, že průchod sestavuje strom, který je podgrafem `G` a jehož kořenem je počáteční vrchol. Tento mechanismus je zajišten polem `P`, ve kterém si na indexu `i` pamatujeme rodiče vrcholu `i` v sestaveném stromu. Pokud vrchol rodiče nemá, nebo je z počátečního vrcholu nedosažitelný, je v poli `P` na jeho indexu hodnota `-1`.

```
graph-search
⟵ g: Graph — reprezentace pomocí matice sousednosti
⟵ s: Int — počáteční vrchol
```

```
1  X ← [g.n]Bool — prvky inicializovány na false
2  P ← [g.n]Int — prvky inicializovány na -1
3  vytvoříme prázdnou strukturu M
4  insert(M,s)
5  X[s] ← true
6  while (not empty(M))
7     u ← pop(M)
8     visit(u)
9     foreach w in g.adj[u] such that X[w] = false
10       X[w] ← true
11       P[w] ← u
12       insert(M,w)
```

Věta 9.3

Uvažujme běh `graph-search` pro graf G a vrchol s, obsah `P` na jeho konci a graf $T = \{V_T, E_T\}$, kde $V_T = \{v \mid v \text{ je dosažitelný z } s\}$ a $E_T = \{(u,v) \mid \texttt{P[v] = u}\}$. Potom platí následující:

(a) T je podgraf G.

(b) Považujeme-li hrany v E_T za neorientované, je T strom s kořenem s.

Důkaz. Důkaz.
(a) Z kódu algoritmu vidíme, že každý vrchol i hrana (modulo orientace), které jsou v T se museli nacházet i v G. U vrcholů je to zřejmé, u hran si stačí všimnout, že na řádku 11 musí `w` a `u` sousedit.

(b) *Absence cyklu.* Vrchol v se nemůže stát potomkem vrcholu v', který je navštíven později než v, protože v ten moment má již v nastaven příznak v poli `X`. Nelze tedy sestavit kružnici.

Souvislost. Indukcí snadno dokážeme, že v momentě, kdy je na řádku 12 vložen vrchol `w` do `M`, existuje v (tomuto kroku odpovídající části) T cesta z `s` do `w`. Platí to totiž před první iterací cyklu `while`, a cyklus tento invariant zachovává. Nastavujeme-li totiž na řádku 11 `u` jako rodiče `w`, jsou `u` a `w` sousedi. □

Různé volby `M` dají průchody s dalšími vlastnosti, to pak určuje jejich použití. Také je možné algoritmus upravit tak, že v poli `X` uchovává informaci o tom, jestli byl vrchol již navštíven, a po odebrání vrcholu z `M` v hlavním cyklu provede zbytek iterace pro daný vrchol pouze pokud byl je odebraný vrchol ještě nenavštíven. Může se tak stát, že nějaký vrchol vložíme do `M` vícekrát, nicméně navštíven bude pouze jednou. Všechna tvrzení (nebo jejich analogie) ze současné kapitoly platí, čtenář si je může dokázat jako cvičení.

9.3 PRŮCHOD DO ŠÍŘKY

Průchod do šířky dostaneme z obecného průchodu tak, že za strukturu `M` zvolíme frontu. Za účelem analýzy algoritmu jej rozšíříme o práci s polem `D`, jehož význam bude zřejmý později.

```
bfs — průchod do šířky
⟵ g: Graph — reprezentace pomocí matice sousednosti
⟵ s: Int — počáteční vrchol
```

```
1  X ← [g.n]Bool — prvky inicializovány na false
2  P ← [g.n]Int — prvky inicializovány na 0
3  D ← [g.n]Int — prvky inicializovány na ω, ω > m pro všechna reálná m
4  vytvoříme prázdnou frontu Q
5  enqueue(Q,s)
6  X[s] ← true
7  D[s] ← 0
8  while (not empty(Q))
9     u ← dequeue(Q)
10    visit(u)
11    foreach w in g.adj[u] such that X[w] = false
12       X[w] ← true
13       P[w] ← u
14       D[w] ← D[u] + 1
15       insert(M,w)
```

Nejkratší vzdálenost $\delta(u, v)$ z vrcholu u do vrcholu v je nejmenší počet hran, které má nějaká cesta z u do v. Pokud cesta z u do v neexistuje, pak $\delta(u, v) = \omega$. Cesta z u do v, která má $\delta(u, v)$ hran je *nejkratší cesta* z u do v.

Věta 9.4

Nechť $G = (V, H)$ je graf a s je jeho vrchol. Potom pro každou hranu $(u, v) \in H$ platí, $\delta(s, v) \leqslant \delta(s, u) + 1$.

Důkaz. Pokud existuje cesta z s do u, pak existuje i cesta z s do v, kterou dostaneme prodloužením nejkratší cesty z s do u o hranu (u, v). Cestu tak prodloužíme o 1 hranu. Pokud cesta z s do v neexistuje, pak neexistuje ani cesta z s do u. V obou případech dokazovaná nerovnost platí. □

Věta 9.5

Po skončení běhu `bfs(G,s)` platí pro každý vrchol `v` nerovnost `D[v]` $\geqslant \delta(s, v)$.

Důkaz. Indukcí přes počet provedení `enqueue`. Víme, že že libovolný vrchol `x` je do fronty vložen maximálně jednou a tedy `D[x]` je změněno maximální jednou.

Při prvním provedení operace tvrzení platí, protože vkládáme vrchol `s`, kterému předtím nastavíme `D[s] ← 0`.

V obecném kroku předpokládejme, že do fronty vkládáme vrchol `w`, který jsme našli při procházení seznamu sousedů vrcholu `u`. Vrchol `u` tak musel být vložen do fronty před `w` a platí pro něj indukční předpoklad, tedy `D[u]` $\geqslant \delta(s, u)$. Na řádku 12 nastavíme `D[w] ← D[u] + 1`. S použitím predchozí věty máme

$$\texttt{D[w]} = \texttt{D[u]} + 1 \;\geqslant\; \delta(s, u) + 1 \;\geqslant\; \delta(s, w).$$

Graf totiž musí obsahovat hranu `(u,w)`, protože vrchol `w` je obsažen v `G.adj[u]`. □

Věta 9.6

Je-li během běhu `bfs` obsah fronty `Q` uspořádaný od nejdříve vloženého vrcholu roven $v_1, v_2, \ldots v_r$, pak

- $\texttt{D}[v_r] \leqslant \texttt{D}[v_1] + 1$,
- pro $i = 1, 2, \ldots, r-1$ máme $\texttt{D}[v_i] \leqslant \texttt{D}[v_{i+1}]$

Důkaz. Indukcí podle počtu operací s frontou. Po první operaci tvrzení očividně platí, protože obsahuje pouze jeden prvek.

Pokud v obecném případě provedeme odebrání vrcholu z fronty, fronta nově začíná od vrcholu v_2 a tvrzení očividně platí.

Uvažme první operaci vložení vrcholu (označíme jej v_{r+1}) v rámci cyklu na řádcích 8 až 15. Předchozí operace musela být odebrání vrcholu (před kterým tvrzení podle indukčního předpokladu tvrzení platilo). Označme odebraný vrchol `u`. Máme tak $\texttt{D}[u] \leqslant \texttt{D}[v_1]$ a $\texttt{D}[v_r] \leqslant \texttt{D}[u] + 1$. Díky řádku 14 máme

$$\texttt{D}[v_r] \leqslant (\texttt{D}[v_{r+1}] = \texttt{D}[u] + 1) \leqslant \texttt{D}[v_1] + 1.$$

a tvrzení tedy platí. Další operace přidání ve stejné iteraci vyřešíme analogicky. □

Věta 9.7: `bfs` hledá nejkratší cesty

Na konci běhu `bfs(G,s)` pro každý vrchol `v` platí

(a) $\texttt{D}[v] = \delta(s, v)$.

(b) Existuje-li cesta z `s` do `v`, je cesta, kterou sestavíme tak, že vezmeme nejkratší cestu z `s` do `P[v]` a připojíme k ní hranu vrchol `v`, nejkratší cestou z `s` do `v`.

Důkaz. (a) Sporem. Předpokládejme, že existuje vrchol v, pro který tvrzení neplatí a má mezi vrcholy, pro které tvrzení neplatí, minimální délku nejkratší cesty z vrcholu s. (Očividně, $v \neq s$.) Z předchozích lemmat máme $\texttt{D}[v] \geqslant \delta(s, v)$, a tedy musí být

$$\texttt{D}[v] > \delta(s, v).$$

Platí tak $\delta(s, v) \neq \omega$ a existuje cesta z s do v. Vybereme vrchol u, který leží na nejkratší cestě z s do v těsně před vrcholem v. Pro u tvrzení musí platit (protože $\delta(s, u) < \delta(s, v)$) máme tak $\texttt{D}[u] = \delta(s, u)$. Odtud dostaneme

$$\texttt{D}[v] > \delta(s, v) = \delta(s, u) + 1 = \texttt{D}[u] + 1 \tag{9.1}$$

Zaměříme se na moment, kdy algoritmus odebral vrchol u z fronty. Pro vrchol v máme dvě možnosti:

- `X[v] = true`.
  Vrchol `v` je buď ve frontě, nebo už ve frontě v minulosti byl. Z předchozího lemmatu pak plyne $\texttt{D}[v] \leqslant \texttt{D}[u] + 1$, což je spor s (9.1).
- `X[v] = false`.
  Vrchol `v` ještě nebyl ve frontě a na řádku 12 algoritmu nastavíme `D[v] ← D[u] + 1`, což je opět spor s (9.1).

(b) je důsledkem (a) a toho, že `bfs` sestaví strom, který je podgrafem G. □

Nyní si můžeme ukázat, že algoritmus `bfs` lze snadno upravit na Dijkstrův algoritmus pro hledání nejkratších cest v grafu. Úprava spočívá v nahrazení fronty jinou datovou strukturou, tzv. prioritní frontou, a v drobné úpravě hlavního cyklu algoritmu.

Zopakujme si nejdříve, jak je definován problém nalezení nejkratší cesty v orientovaném grafu. Vstupem je graf $G = (V, H)$ s ohodnocením hraf $c : H \to \mathbb{R}^+$, a vrcholy s, t. Chceme, aby graf nalezl nejkratší cestu v G z vrcholu s do vrcholu t.

V popisu algoritmu předpokládáme, že máme speciální hodnotu ω takovou, že
- pro každé $r \in \mathbb{R}$ platí $r < \omega$,
- pro každé $r \in \mathbb{R}$ platí $\omega + r = r + \omega = \omega$,
- $\omega + \omega = \omega$

Tato hodnota pro nás hraje roli *dostatečně velkého čísla*, pokud neexistuje cesta z s do t, je délka nejkratší cesty z s do t rovna ω. Ve skutečné implementaci algoritmu lze tuto hodnotu nahradit konkrétním číslem, laskavý čtenář si toto číslo najde jako cvičení.

Připomeňme si samotný Dijkstrův algoritmus. Uvedeme si jej ve znění, ve kterém pouze počítá délky nejkratších cest z vrcholu s do ostatních vrcholů v grafu. Rozšířit jej tak, aby počítal i samotné nejkratší cesty, nebo aby zastavil v momentě, kdy najde nejkratší cestu do vrcholu t je snadné.

1. *inicializace*
   - nastavíme $d(s) \leftarrow 0$
   - nastavíme $A \leftarrow V$
   - pro $u \in V - \{s\}$ nastavíme $d(u) \leftarrow \omega$
2. *nalezneme minimální vrchol*
   - pokud $A = \emptyset$, algoritmus končí
   - nastavíme $m \leftarrow \operatorname{argmin}_{u \in A} d(u)$ (Tady už víme, že $d(m)$ je délkou nejkratší cesty.)
   - pokud $d(m) = \omega$, algoritmus končí
3. *update nalezených cest*
   - pro všechny sousedy u vrcholu m spočítáme $x \leftarrow d(m) + c(m, v)$ a nastavíme $d(u) \leftarrow \min(d(u), x)$.
   - nastavíme $A \leftarrow A - \{m\}$
   - Pokračujeme krokem 2.

Prioritní fronta je (abstraktní) datová struktura sloužící uložení dvojic tvořených prioritou a daty. Poskytuje následující operace:
- `insert`. Vložení dvojice.
- `extract-min`. Odebrání dvojice s nejmenší prioritou.
- `decrease-key` Zmenšení priority u dvojice, která již je v prioritní frontě.

Prioritní frontu lze vytvořit například pomocí vyhledávacího stromu, ve kterém považujeme priority za klíče (a musíme tedy povolit to, aby strom obsahoval nějaký klíč vícekrát, to je ovšem triviální úprava), a jednotlivé operace s prioritní frontou provedeme pomocí operací se stromem. Operaci `insert` uděláme pomocí `tree-insert`, operaci `extract-min` pomocí operací `tree-min` a `tree-remove`, a nakonec operaci `decrease-key` pomocí `tree-remove` a opětovného vložení (pomocí `tree-insert`) dané dvojice s již

upravenou prioritou. Pro prioritní fronty existují i specializované struktury, čtenáři již známým příkladem je binární halda z algoritmu heapsort.

Předpokládeme tedy, že máme k dispozici prioritní frontu a přepišme Dijkstrův algoritmus tak, aby strukturou připomínal průchod do šířky.

```
dijkstra
⟵ g: Graph — součástí je i ohodnocení hran g.c
⟵ s: Int — počáteční vrchol
⟶ []Float — pole vzdáleností od s
```

```
1  Y ← [g.n]Bool — navštíven? Inicializováno na false
2  X ← [g.n]Bool — je v prioritní frontě? Inicializováno na false
3  D ← [g.n]Float — inicializováno na ω
4  vytvoř prázdnou prioritní frontu Q
5  insert(Q, s, 0)
6  D[s] ← 0
7  X[s] ← true
8  Y[s] ← true
9  while (not empty(Q))
10    u, d ← extract-min(Q)
11    Y[u] ← true
12    D[u] ← d
13    foreach w in g.adj[u] such that Y[u] = false
14       if (not X[w])
15          insert(Q, w, d + g.c(u,w))
16          X[w] ← true
17       else
18          decrease-key(Q, w, d+g.c(u,w))
19 return D
```

Pro operaci `decrease-key` je nutný mechanismus, který identifikuje konkrétní dvojici v prioritní frontě, které chceme prioritu snížit. To závisí na konkrétní zvolené prioritní frontě a proto to vynecháme.

Vidíme tedy, že na Dijkstrův algoritmus se můžeme dívat jako na variantu průchodu do šířky, kde používáme prioritní frontu a operaci `decrease-key`. Pokud jsou ohodnocení hran v grafu rovná 1 nebo jsou ohodnocení všech hran stejná, pak Dijkstrův algoritmus je průchodem do šířky. Změna priority na ř. `18` totiž nevede ke změně pořadí v prioritní frontě. Prioritu tak vrcholu přiřazujeme nejvýše jednou, a to při vkládání do prioritní fronty, tedy přesně tak, jak se to děje v algoritmu `bfs`. Lze také ukázat, že pokud vkládáme vrchol do prioritní fronty, nemá menší prioritu, než je maximální priorita mezi vrcholy již v prioritní frontě obsaženými.

9.4 PRŮCHOD DO HLOUBKY

Průchod do hloubky dostaneme z obecného průchodu úpravou zmíněnou na konci kapitoly 9.2 a použitím zásobníku na místě `M`. Alternativně, jak tomu učiníme v této kapitole, lze práci se zásobníkem nahradit rekurzí. Pro účely analýzy algoritmu použijeme následující globální data.

- Proměnnou `time`, inicializovanou na `0`.
- Pole `D` a `F` velikosti `g.n`, prvky inicializovány na `0`.

- Pole `P` velikosti `g.n`, prvky inicializovány na `nil`.
- Pole `X` velikosti `g.n`, prvky inicializovány na `false`.

Následující procedura odpovídá jednomu průchodu spuštěnému z vrcholu `t`.

```
dfs-visit
⟵ g: Graph
⟵ t: Int
```

```
1  time ← time + 1
2  D[t] ← time
3  X[t] ← true
4  foreach u in g.adj[t] such that X[u] == false
5     P[u] ← t
6     dfs-visit(g, u)
7  time ← time + 1
8  F[t] ← time
```

Následující procedura pomocí série průchodů do hloubky spuštěných z dosud nenavštívených vrcholů, zajistí, že navštívíme všechny vrcholy grafu.

```
dfs-all
⟵ g: Graph
```

```
1  for u in 0 ..< g.n
2     if (X[u] == false) then dfs-visit(g, u)
```

Věta 9.8

Volání `dfs-all(g,t)` sestaví les.

Důkaz. Předpokládejme, že `dfs-all` volá `dfs-visit` postupně pro vrcholy `t1, t2, ...`.

Už víme, že `dfs-visit(g, t1)` sestaví strom s kořenem `t1`, jehož vrcholy jsou všechny vrcholy dosažitelné z `t1`. Díky poli `X` pak `dfs-visit(g, t2)` sestaví strom s kořenem `t2` obsahující vrcholy dosažitelné z `t2` mimo těch, které jsou dosažitelné z `t1`.

Obecně volání `dfs-visit(g, ti)` sestaví strom s kořenem `ti` obsahujícím vrcholy dosažitelné z `ti` mimo těch, které jsou dosažitelné z `tj` pro `j < i`. □

Věta 9.9: Uzávorkovací

Po provedení `dfs-all` platí pro libovolné různé vrcholy u, v a les vytvořený `dfs-all` právě jedna z následujících možností.

(a) $\langle \texttt{D}[u], \texttt{F}[u] \rangle \cap \langle \texttt{D}[v], \texttt{F}[v] \rangle = \emptyset$, vrcholy u a v nejsou ve vztahu následovník/-předek;

(b) $\langle \texttt{D}[u], \texttt{F}[u] \rangle \subset \langle \texttt{D}[v], \texttt{F}[v] \rangle$, vrchol u je následovníkem vrcholu v;

(c) $\langle \texttt{D}[v], \texttt{F}[v] \rangle \subset \langle \texttt{D}[u], \texttt{F}[u] \rangle$, vrchol v následovníkem vrcholu u.

Důkaz. Z algoritmu vidíme, že `D[u]`,`D[v]`,`F[u]`,`F[v]` jsou po dvojicích různé a $\texttt{D}[u] < \texttt{F}[u]$ a $\texttt{D}[v] < \texttt{F}[v]$. Předpokládejme $\texttt{D}[u] < \texttt{D}[v]$.

Pokud `D[v] < F[u]`, pak celé rekurzivní volání `dfs-visit` pro vrchol v proběhlo v rámci volání pro vrchol u a tedy `F[v] < D[u]`. Podle (důkazu) předchozí věty je ve vytvořeném stromu u předkem v.

Pokud naopak platí, že `D[v] > F[u]`, tak dostaneme `D[u] < F[u] < D[v] < F[v]` a intervaly jsou disjunktní. Volání `dfs-visit` pro u skončilo před tím, než začalo voláni pro v. Vrcholy tedy podle (důkazu) předchozí věty leží v různých podstromech. □

Věta 9.10: O nenavštívené cestě

Nechť `T` je les sestavený `dfs-all` pro graf G. Potom pro libovolné různé vrcholy u, v jsou následující podmínky ekvivalentní:

(a) v je následníkem u v lese `T`,

(b) v čase `D[u]` (tj. v momentě, kdy algoritmus nastavoval hodnotu `D[u]`) existovala cesta v grafu z vrcholu u do vrcholu v taková, že pro každý vrchol $w \neq u$ na této cestě platí `X[w] = false`.

Důkaz. *(a) implikuje (b):* Pokud je vrchol w následníkem vrcholu u, pak z věty o uzávorkování plyne `D[u] < D[w]`, že v čase `D[u]` tedy muselo být `X[w] = false`. Dále musí z u do w existovat v grafu cesta, jinak by se w nemohl stát následníkem u.

(b) implikuje (a): Sporem. Před. že $w \neq u$ je první vrchol na cestě z u do v, který není následníkem u. Nechť z je vrchol, který je na této cestě těsně před w. Z uzávorkovací věty plyne `F[z] ⩽ F[u]` (může být $z = u$). Vrchol w není navštíven dříve než vrchol u, proto `D[u] < D[w]`. V grafu je hrana (z, w), proto je vrchol w navštíven nejpozději během volání `dfs-visit` pro vrchol `z`, odtud `D[w] < F[z]`. Dohromady tedy `D[u] < D[w] < F[z] ⩽ F[u]`. Z uzávorkovací věty ale plyne, že `F[w] < F[u]` a w je potomek u. □ □

S pomocí lesa sestaveného pro sestavený procedurou `dfs-all` můžeme klasifikovat hrany grafu do následujících kategorií. Hrana (u, v) je

1. *stromová*, pokud u je v lese T rodičem v,
2. *zpětná*, pokud v je v lese T předchůdcem u,
3. *dopředná*, pokud není stromová a v je v lese T následníkem u,
4. *křížová*, jinak.

Pro neorientované grafy musíme doplnit podmínku určující, jestli je hrana $\{u, v\}$ zpětná nebo dopředná (podle předchozí definice může být obojí). Řekneme, že `dfs-all` zkoumá hranu $\{u, v\}$ při volání `dfs-visit` pro u, pokud v tomto zavolání při průchodu `adj[u]` narazíme na vrchol v. Pokud `dfs-all` nejdříve zkoumal hranu $\{u, v\}$ ze zavolání `dfs-visit` pro u, bereme hranu orientovanou jako (u, v). Jinak je orientace opačná.

Věta 9.11

Každá hrana v neorientovaném grafu je buď stromová nebo zpětná.

Důkaz. Uvažme libovolnou hranu $\{u, v\}$ a předpokládejme `D[u] < D[v]`. Protože v je v seznamu `adj[u]`, musí `dfs-visit` pro v skončit dříve, než to pro u. Máme tak `F[v] < F[u]` a podle uzávorkovací věty je v následníkem u.

Pokud algoritmus poprvé zkoumá hranu $\{u, v\}$ při volání `dfs-visit` pro u, musí být `X[v] = false`, vrchol v se stane potomkem u a hrana je stromová.

Pokud naopak na tuto hranu narazíme poprvé až během volání pro v, přiřadíme jí orientaci (v, u) a je tak zpětná. □

9.5 TOPOLOGICKÉ USPOŘÁDÁNÍ

Orientovaný graf bez cyklů budeme nazývat *dag*. (To je zavedená anglická zkratka pojmu *directed acyclic graph*).

Topologické uspořádání dagu $G = (V, E)$ je lineární uspořádání vrcholů grafu takové, že pokud $(u, v) \in E$, pak u je v tomto uspořádání před v.

K nalezení topologického uspořádání můžeme s výhodou využít průchodu do šířky, který jenom drobně upravíme. Výsledný algoritmus nazveme `topol`.

- Inicializujeme prázdný seznam vrcholů,
- Spustíme upravený průchod do hloubky. Úprava spočívá v tom, že vždycky, když nastavujeme `F[u]` pro vrchol u, připojíme u na začátek seznamu
- Po skončení průchodu obsahuje seznam vrcholy uspořádané sestupně podle hodnot položky `F`. Topologické uspořádání je dáno pořadím v tomto seznamu.

Složitost algoritmu `topol` je $O(|V| + |E|)$. Vkládání vrcholu na začátek seznamu je v konstantním čase a vkládáme $|V|$ vrcholů. To je jediná práce navíc přidaná k průchodu do hloubky.

Věta 9.12

Orientovaný graf G neobsahuje cyklus, právě když `dfs-all`(G) nevytvoří žádnou zpětnou hranu.

Důkaz. Ukážeme, že G obsahuje cyklus, právě když `dfs-all` vytvoří zpětnou hranu.

Pokud `dfs-all` vytvoří zpětnou hranu (u, w), pak vrchol u musí být potomkem w v příslušném stromu. Cesta z w do u v grafu G (která odpovídá cestě mezi těmito vrcholy ve stromu vytvořeném `dfs-all`) prodloužená o hranu (u, w) je kružnice.

Nechť G obsahuje cyklus c a w je první vrchol na tomto cyklu, pro který je zavoláno `dfs-visit`. Vrchol, který w na cyklu předchází označme u. V čase `D[w]` tvoří část c bez w nenavštívenou cestu. Podle věty o nenavštívené cestě pak musí být vrchol u následníkem vrcholu w ve stromu sestaveném `dfs-all`. Hrana (u, w) je proto zpětná. □

Věta 9.13

Algoritmus `topol` nalezne topologické uspořádání.

Důkaz. Ukážeme, že pro každou hranu (u, v) platí, že `F[u]` > `F[v]`.

V momentě, kdy `dfs-all` zkoumá hranu (u, v) (v rekurzivním zavolání `dfs-visit` pro `u`), máme dvě možnosti:

- `X[v]` = `true`
  Potom algoritmus již nastavil `D[v]` a tak musí platit `D[v]` < `F[u]`. Pokud bychom měli `F[u]` < `F[v]`, tak v důsledku věty o uzávorkování máme `D[v]` < `D[u]` < `F[u]` < `F[v]`, a hrana (u, v) je zpětná. To je ovšem podle předchozí věty spor, a proto musí plati `F[v]` < `F[u]`.

- `X[`v`] = false`
  Potom volání `dfs-visit` pro v proběhne v rámci volání `dfs-visit` pro u, a máme tak `F[`v`] < F[`u`]`.

□

9.6 SILNĚ SOUVISLÉ KOMPONENTY

Zavedeme značení pro existenci cest (v příslušném grafu):

- $u \rightsquigarrow v$ značí, že cesta z u do v existuje
- $u \not\rightsquigarrow v$ značí, že cesta z u do v neexistuje

Množina vrcholů C v orientovaném grafu je *silně souvislá*, pokud pro libovolné různé vrcholy $u, v \in C$ máme $u \rightsquigarrow v$ a $v \rightsquigarrow u$. Množina vrcholů C v orientovaném grafu je *silně souvislá komponenta* (dále jen komponenta), pokud je silně souvislá a neexistuje vrchol, který bychom mohli do C přidat tak, abychom po přidání dostali silně souvislou množinu.

Věta 9.14

Nechť C a C' jsou různé komponenty. Pro libovolné $u, v \in C$, $u', v' \in C'$ pak $u \rightsquigarrow u'$ implikuje $v' \not\rightsquigarrow v$.

Důkaz. $u \rightsquigarrow u'$ implikuje, že pro každé $x \in C, y \in C'$ máme $x \rightsquigarrow y$. Pokud bychom měli $v' \rightsquigarrow v$, pak pro každé $z \in C', w \in C$ máme $z \rightsquigarrow w$. Tedy C a C' by nemohli být různé komponenty. □

Důsledkem předchozí věty je, že průnik různých komponent je prázdný. Dále si všimneme, že množina tvořená jedním vrcholem je silně souvislá a proto je každý vrchol v nějaké silně souvislé komponentě. Množina komponent grafu je tedy rozkladem množiny jeho vrcholů. Můžeme tak vytvořit graf komponent $\overline{G} = (\overline{V}, \overline{H})$ grafu G, kde $\overline{V}$ je množina všech silně souvislých komponent v G a $(C, C') \in \overline{H}$ když existují $u \in C, v \in C'$ tak, že $(u, v) \in H$. Z předchozí věty navíc plyne, že $\overline{G}$ je dag.

Předpokládejme, že jsme pro G provedli `dfs-all` a máme k dispozici pole `X, D, F`. Pro množinu vrcholů W zavedeme `F[`W`]` $= \max_{v \in W}$ `F[`v`]`.

Věta 9.15

Nechť C, C' jsou různé komponenty. Potom existuje hrana (u, v) taková, že $u \in C$, $v \in C'$, pak `F[C] > F[C']`.

Důkaz. Pro $W \subseteq V$ zavedeme: `D[`W`]` $= \min_{v \in W}$ `D[`v`]`.
Případ 1: `D[C] < D[C']`
Vybereme vrchol $x \in C$, pro který `D[C] = D[x]`. Potom pro libovolný $y \in C \cup C'$ různý od x máme `D[x] < D[y]` (protože $x \rightsquigarrow y$). V čase `D[x]` (tedy na začátku volání `dfs-visit` pro vrchol x) tedy existuje nenavštívená cesta z x do y a tedy y je ve stromu sestaveném `dfs-all` potomkem vrcholu x. Z uzávorkovací věty potom plyne, že `F[x] > F[y]`.
Případ 2: `D[C] > D[C']`
Vybereme vrchol $x \in C'$, pro který `D[C] = D[x]`. Potom pro libovolný $y \in C'$ máme `D[x] < D[y]` a v čase `D[x]` existuje nenavštívená cesta do y, vrchol y je tak ve stromu sestaveném `dfs-all` potomkem x a podle uzávorkovací věty tak máme `F[x] < F[y]` a

tedy `F[x]` $=$ `F[C′]`. Protože $x \not\rightsquigarrow z$ pro libovolný vrchol $z \in C$, máme `D[C]` $>$ `F[C′]` a tedy `F[C]` $>$ `F[C′]`. □

Důsledkem předchozí věty je, že pokud bychom v grafu komponent procházeli vrcholy podle příslušných hodnot `F[]`, projdeme je v topologickém uspořádání. To vede k následujícímu nápadu na algoritmus, který vypočítá množinu komponent grafu.

1. Najdeme vrchol u s maximálním `F[]` v celém grafu.
2. Nalezneme silně souvislou komponentu, ve které se u nachází.
3. Odstraníme tuto komponentu z grafu a pokud je graf neprázdný, pokračujeme krokem 1.

Krok 2 lze realizovat pomocí `dfs-visit`, pokud dokážeme zajistit, aby `dfs-visit` nenavštívil vrcholy z jiných komponent. Toho lze dosáhnout obrácením směru hran! Navíc v kroku 3 potom nemusíme z grafu nalezenou komponentu odstraňovat, supluje práce s polem `X` v algoritmu `dfs-visit`. Detaily tohoto algoritmu si uvedeme ve zbytku kapitoly.

Transpozicí orientovaného grafu $G = (V, H)$ je orientovaný graf $G^T = (V, H^T)$, kde $H^T = \{(u, v) \mid (v, u) \in H\}$.

Věta 9.16

G a G^T mají stejné množiny komponent.

Důkaz. Pro libovolné vrcholy máme $u \rightsquigarrow v$ v grafu G, právě když $v \rightsquigarrow u$ v grafu G^T. □

Důsledkem věty je rovnost $(\overline{G})^T = \overline{(G^T)}$.

Algoritmus `strong-components`, který nalezne množinu komponent v grafu G, funguje následovně.

1. Spočítá topologické uspořádání pomocí `dfs-all` spuštěného pro graf G.
2. Zavoláme `dfs-all` pro graf G^T s tím, že v hlavní smyčce `dfs-all` procházíme vrcholy v pořadí podle uspořádání získaného v prvním kroku.
3. Silně souvislé komponenty pak odpovídají stromům v lese sestaveném `dfs-all` z kroku 2.

Složitost algoritmu je $\Theta(|V| + |E|)$. Kroky 1 a 2 mají složitost $\Theta(|V| + |E|)$, sestavení G^T lze provést se složitostí $\Theta(|E|)$.

Věta 9.17: Korektnost algoritmu `strong-components`

Algoritmus `strong-components` nalezne právě všechny silně souvislé komponenty grafu.

Důkaz. Značení: `F[]`, `D[]` patří k `dfs-all` řádku 1 algoritmu. `F[]′`, `D[]′` k průchodu na řádku 2.

Indukcí přes počet spuštění `dfs-visit` v hlavní smyčce algoritmu `dfs-all` z řádku 2 ukážeme, že doposud vytvořené stromy odpovídají právě silně souvislým komponentám.

Pro 0 spuštění je tvrzení triviálně pravdivé.

Předpokládejme, že tvrzení platí pro prvních k spuštění, a že další spuštění `dfs-visit` je pro vrchol `x` patřící do silně souvislé komponenty C.

V čase $\texttt{D[x]}'$ existuje nenavštívená cesta z vrcholu x do libovolného jiného vrcholu v komponentě C (z indukčního předpokladu plyne, že žádný vrchol v C dosud nebyl navštíven). Všechny vrcholy v C tedy patří podle věty o nenavštívené cestě do jednoho stromu.

Zbývá ukázat, že do tohoto stromu nepatří vrchol mimo C. To plyne z toho, že v grafu G^T pro všechny hrany (u, v) takové, že $u \in C, v \notin C$ platí, že v patří do komponenty C' s $\texttt{F[C']} > \texttt{F[C]}$, tzn. v čase $\texttt{D[x]}'$ už byl vrchol v navštíven. □

10.1 BINÁRNÍ VYHLEDÁVACÍ STROM

Uvažujeme množinu U vrcholů s různými klíči, $|U| = n$. Permutace množiny U je posloupnost

$$u_1, \ldots, u_n$$

prvků U taková, že každý prvek z U se v ní nachází právě jednou. Takový posloupností existuje $n!$.

O binárním vyhledávacím stromu řekneme, že byl sestaven metodou náhodných klíčů, pokud je výsledkem následující procedury.

1. Navzorkujeme permutaci vrcholů (s uniformní pravděpodobností, každá permutace má pravděpodobnost výběru rovnu $\frac{1}{n!}$).
2. Vezmeme prázdný BVS a vložíme do něj operací `insert` vrcholy z U v pořadí daném permutací z předchozího bodu.

O metodě vzorkování binárních vyhledávacích stromů, jejichž množina klíčů je U, řekneme, že splňuje podmínku kořenové uniformity, pokud

(a) pro každý vrchol $x \in U$ platí, že pravděpodobnost toho, že se stane kořenem navzorkovaného stromu, je rovna $\frac{1}{|U|}$.

(b) Je-li x kořenem stromu, potom kořenová uniformita platí i pro

- levý podstrom a množinu $L = \{y \in U \mid y \text{ má menší klíč než } x\}$
- pravý podstrom a množinu $R = \{y \in U \mid y \text{ má větší klíč než } x\}$

Jde o induktivní definici. Pro $|U| = 1$ a $|U| = 0$ je podmínka triviálně splněna.

Věta 10.1

Vzorkování metodou sestavení z náhodných klíčů je kořenově uniformní.

Důkaz. Předpokládejme $|U| = n > 1$.

Uvážíme-li posloupnost $u_1, \ldots, u_n$, pak se kořenem stane vrchol u_1, kořenem levého podstromu se stane první prvek v posloupnosti s menším klíčem než u_1, kořenem pravého podstromu pak první prvek v posloupnosti s větším klíčem než u_1.

Protože permutace vzorkujeme uniformně, je pravděpodobnost, že $x \in U$ se stane kořenem (tj. stane se u_1), rovna

$$\frac{(n-1)!}{n!} = \frac{1}{|U|}.$$

Připomeňme, že pro kořen x, máme množiny

$$L = \{y \in U \mid y \text{ má menší klíč než } x\}$$
$$R = \{y \in U \mid y \text{ má větší klíč než } x\}$$

Dále vidíme, že levý podstrom závisí pouze na podpermutaci permutace $u_2, \ldots, u_n$, ve které uvažujeme pouze prvky z L. Přitom každá taková konkrétní podpermutace má stejnou pravděpodobnost výskytu (bereme-li pst přes navzorkované permutace prvků z $U - \{x\}$).

Tady ovšem platí, že $y \in L$ je kořenem levého podstromu, pokud je v této podpermutaci na prvním místě. To je ovšem s pravděpodobností

$$\frac{(|L|-1)!}{|L|!} = \frac{1}{|L|}$$

Podobně bychom dokázali i pravděpodobnost pro pravý podstrom a R. □

Předpokládejme, že je dán binární vyhledávací strom t s množinou klíčů U. Dále předpokládejme, že pro každý klíč $x \in U$ je pravděpodobnost, že jej budeme vyhledávat rovna $1/n$. Dále nechť $h_t(x)$ je hloubka vrcholu obsahujícího klíč x ve stromu t. Průměrný počet porovnání dvou klíčů, který je potřeba v algoritmu vyhledávání, které skončí úspěchem, je potom

$$\mathrm{avg}(t) = \frac{1}{n} \sum_{x \in U} h_t(x) + 1.$$

Nechť S je množina klíčů, které jsou v t ve vrcholu s méně než 2 potomky. Předpokládejme, že pravděpodobnost toho, že při neúspěšném vyhledávání skončí algoritmus ve vrcholu s klíčem x, je rovna $1/|S|$. Průměrný počet porovnání dvou klíčů, který provede vyhledávání končící neúspěchem, je

$$\overline{\mathrm{avg}}(t) = \frac{1}{|S|} \sum_{x \in S} h_t(x) + 1.$$

Uvažujeme nyní vzorkování stromů obsahujících množinu klíčů U, kde $|U| = n$. Pravděpodobnost, s jakou je strom t navzorkován označíme $P(t)$. Průměrné počty porovnání dvou klíčů v algoritmu vyhledávání ve stromu získaném vzorkováním jsou potom

$$\mathrm{avg}(n) = \sum_t P(t) \cdot \mathrm{avg}(t),$$

$$\overline{\mathrm{avg}}(n) = \sum_t P(t) \cdot \overline{\mathrm{avg}}(t),$$

kde sumy probíhají přes všechny binární vyhledávací stromy s množinou klíčů U.

Věta 10.2

Pokud vzorkujeme metodou, která je kořenově uniformní, pak platí $\mathrm{avg}(n) \approx 2\ln(n)$ a $\overline{\mathrm{avg}}(n) \approx 2\ln(n)$.

Zdroj náhodnosti u stromu sestaveného z náhodných klíčů je ve vzorkování permutace. Přesuneme jej dovnitř algoritmu pro přidávání/mazání prvku do/z binárního vyhledávacího stromu. Algoritmus přidávání pak vypadá následovně.

```
random-insert
⟵ r: Node
⟵ x: Node
⟶ Node
```

```
1  if (r == nil) then return x
2  if (random(0, r.count) == 0) then return root-insert(r,x)
3  if (x.id < r.id)
4     set-left-child(r, random-insert(r.left, x))
5  else
6     set-right-child(r, random-insert(r.right, x))
7  return r
```

Procedura `random(a,b)` vrací náhodné číslo z množiny $\{a, a+1, \ldots, b\}$ (uniformní rozložení pravděpodobnosti).

Odebíráme-li ze stromu vrchol `x`, pak pokud má `x` 2 potomky, s pravděpodobností `x.left.count/(x.count-1)` jen nahradíme pořádkovým předchůdcem. Jinak je nahradíme pořádkovým následníkem. Zbytek algoritmu je nezměněn.

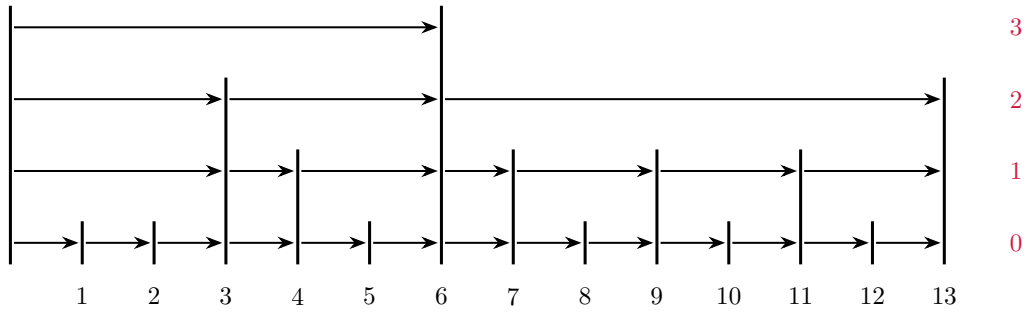
Věta 10.3

Pokud se na fixní posloupnost randomizovaných operací `insert` a `delete` díváme jako na proces vzorkování, je toto vzorkování kořenově uniformní.

10.2 SKIPLIST

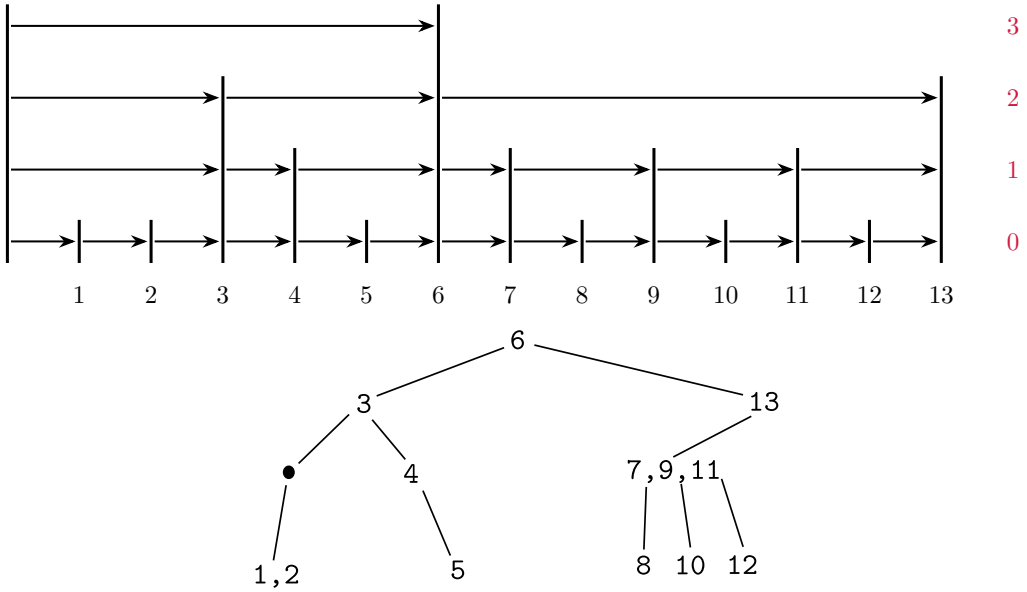
Skiplist si lze na začátku představit jako analogii spojového seznamu, ve kterém může být uzel seznamu zapojen do více spojových seznamů na jednou. Těchto seznamů je maximálně t, kde t je přirozené číslo, které parametrizuje skip-list.

V každém z t seznamů jsou uzly uspořádány vzestupně podle klíčů a navíc i seznamy jsou uspořádány hierarchicky do úrovní. V každé z t úrovní, které číslujeme od 0 a máme tak úrovně 0 až $t-1$, se nachází jeden seznam. Přitom seznam na úrovni 0 obsahuje všechny uzly, seznam na úrovni $i > 0$ je obsahuje podmnožinu uzlů seznamu na úrovni $i-1$. Na začátku seznamů je zarážka. Příklad skiplistu pro $t = 4$ je na následujícím obrázku.



Skiplist můžeme interpretovat jako vyhledávací strom výšky $t-1$, v jehož vrcholech může být libovolný počet klíčů (je to tedy zobecnění B stromu). Kořen stromu obsahuje klíče obsažené ve skiplistu v seznamu na úrovni $t-1$. Uzly seznamu na úrovni $t-1$ přirozeně rozdělují uzly, které do něj nejsou zapojeny, do intervalů, které jsou vlastně

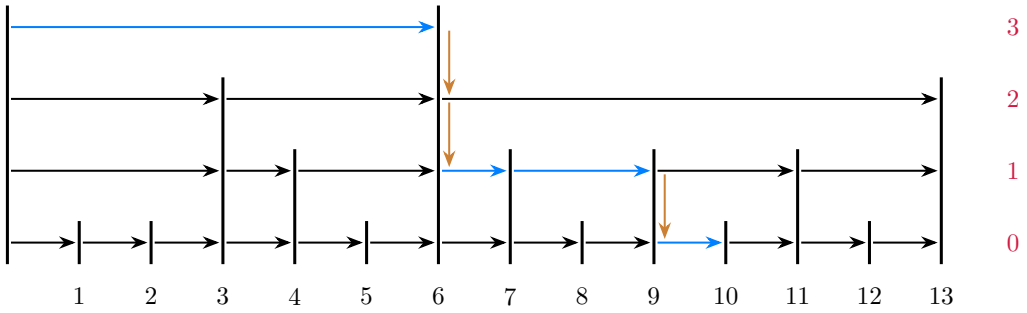
skiplisty s parametrem $t-1$. Tyto skiplisty transformujeme na strom právě popisovaným způsobem a jejich kořeny zapojíme jako potomky kořenu našeho stromu. Jedinou komplikací je to, že ve stromu mohou vzniknout prázdné vrcholy neobsahující žádný klíč. To nastane pokud je seznam v nejvyšší úrovni skiplistu prázdný. Příklad skiplistu a jemu odpovídajícího vrcholu je na následujícím obrázku.

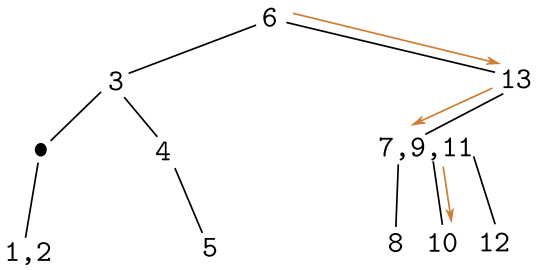


Ve skiplistu vyhledáváme uzel obsahující klíč k následovně,

1. Za aktuální seznam vybereme seznam na úrovni $t-1$, za aktuální uzel vybereme zarážku.
2. Prohledáváme od aktuálního uzlu aktuální seznam. Prohledávání končí buď nalezením uzlu s klíčem k, nebo nalezením uzlu jehož soused je buď nil, nebo obsahuje klíč větší než k.
3. Pokud je aktuální seznam na úrovni 0, vyhledávání končí neúspěšně. Jinak za aktuální seznam o úroveň jedna nižší a pokračujeme bodem 2. (Aktuálním uzlem zůstává uzel, ve kterém skončilo prohledávání v minulém bodě.)

Složitost algoritmu je dána počtem navštívených uzlů. Na následujícím obrázku je zobrazeno vyhledávání klíče 10. (Z obrázku je zřejmé, proč se struktuře říká skiplist.)





Při vkládání uzlu `x` do skiplistu se musíme rozhodnout, do kterých seznamů jej zapojíme. Stačí určit nejvyšší úroveň seznamu, do kterého bude uzel patřit, z definice skiplistu plyne, že musí patřit i do všech seznamů úrovně menší.

K určení této úrovně poslouží parametr `s`. Představuje očekávaný počet uzlů, které se nachází v seznamu na úrovni `i` mezi dvěma uzly zapojenými v seznamu na úrovni `i-1`. Pro složitost vyhledávání je optimální, pokud je úrovni `1` ja každý `s`-tý uzel, v senzamu na úrovni 2 každý s^2-tý uzel. Obecně pak v seznamu na úrovni `i` každý s^i-tý uzel.

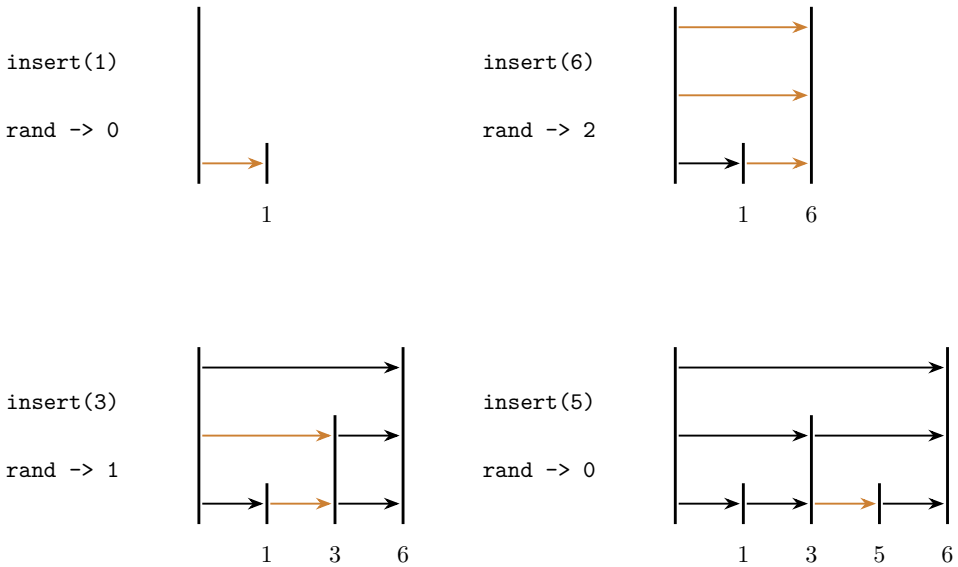
O maximální úrovni seznamu rozhodneme pravděpodobnostně pomocí procedury `rand`, takové, že

$$\texttt{rand(s,t)} = \begin{cases} \texttt{j} & \text{s pravděpodobností } 1/s^{\texttt{j}}, \text{ pokud } \texttt{t} > \texttt{j} > 0 \\ 0 & \text{jinak, tj. s pravděpodobností } 1 - \sum_{\texttt{k}=1}^{\texttt{t}-1} 1/s^{\texttt{k}} \end{cases}$$

Uzlu `x` nastavíme položku `sz` rovnu `rand(s,t)+1`. To znamená, že `x` bude zapojen do seznamů na úrovních `0` až `x.sz - 1`.

Vlastní vložení uzlu provádíme pomocí úpravené procedury vyhledávání: jsme-li na úrovni `l < x.sz` ve vyhledávání klesáme o úroveň níže, nebo jsme na úrovni 0 a došlo k neúspěšnému vyhledávání, zapojíme `x` do seznamu na úrovni `l` za aktuální uzel.

Na následujících obrázcích můžeme vidět příklad přidávání vrcholů do skiplistu s parametry `t=3, s=2`.



Věta 10.4

Vyhledávání a vkládání do skip-listu s parametrem s vyžaduje průměrně $O(s \log_s n)/2$ porovnání.

V předchozí větě se nevyskytuje parametr t. Aby věta platila, musí být tento parametru ve správném vztahu k s a n. Čtenář si tento vztah promyslí jako cvičení.