Algoritmická matematika 2

část 1

LS 2011/12

Radim BĚLOHLÁVEK

Katedra informatiky
Univerzita Palackého v Olomouci

Vyhledávání

Základní otázky a problémy

- Jak efektivně vyhledávat data počítači?
- Jak data ukládat, tak aby je bylo možno ekeftivně vyhledávat?
- Otázky jak ukládat a jak vyhledávat jsou úzce spjaty.
- Problém ukládání a vyhledávání lze je chápat jako problém návrhu efektivních datových struktur.
- Při volbě metody ukládání a metody vyhledávání je třeba vycházet z toho, jak často bude probíhat ukládání (např. vkládání nových položek) a jak často vyhledávání, jak velká data budou vkládána, jaký je typ dat (číselná, textová, obrázky, zvuk, video).
- Jeden ze základních problémů informatiky.

Příklad: vyhledávání v poli

Problém: Vyhledávání v poli čísel
Vstup: pole $A[0 \ldots n-1]$, hodnota x (číslo)
Výstup: NE, pokud se x v A nenachází; ANO, pokud se x v A nachází

Search(x, A)
1 $i \leftarrow 0$
2 **while** $(A[i] \neq x$ and $i \leq n-1)$ **do** $i \leftarrow i+1$
3 **if** $i \leq n-1$ **return** YES
3 **else return** NO

Časová složitost v nejhorším případě je zřejmě $\Theta(n)$.

Lze zlepšit?

Ano, pokud je pole setříděné. Pak je totiž následujcící algoritmus správný (tj. vzhledem k problému vyhledávání v setříděném poli). Jde o algoritmus binárního vyhledávání, který má široké uplatnění.

Bin-Search(x, A, p, r)
1 **if** $p > r$ **then return** NO
2 **else** $q \leftarrow \lfloor (p + r)/2 \rfloor$
3 **if** $x = A[q]$ **then return** YES
4 **else if** $x < A[q]$ **then return** Bin-Search(x, A, p, $q - 1$)
5 **else return** Bin-Search(x, A, $q + 1$, r)

Časová složitost v nejhorším případě je $\Theta(\lg n)$.

Vidíme, že vhodné uspořádání prvků v datové struktuře může zlepšit složitost vyhledávání.

Jaká je ale režie potřebná pro "vhodné" uspořádání? Vyplatí se pole nejdříve uspořádat a pak v něm vyhledávat?

Záleží na tom, kolikrát budeme vyhledávání provádět.

Záleží na tom, kolikrát budeme vyhledávání provádět.

Pole lze setřídit v čase $\Theta(n \lg n)$, tj. pomocí řádově $c_1 n \lg n$ kroků. Vyhledávat v něm pak lze pomocí Bin-Search v čase $\Theta(\lg n)$, tj. pomocí řádově $c_2 \lg n$ kroků. Provedeme-li k vyhledání, potřebujeme řádově

$$c_1 n \lg n + k c_2 \lg n \text{ kroků.}$$

Použijeme-li pro vyhledávání Search (řádově $c_3 n$ kroků), pole nemusíme nejprve setřídit. Provedeme-li takto k vyhledání, potřebujeme řádově

$$k c_3 n \text{ kroků.}$$

Pro malé hodnoty k je výhodnější pole netřídit a použít přímo Search. Pro velké hodnoty k je výhodnější pole nejdřív setřídit a pak používat Bin-Search. (Zvolte konkrétní hodnoty c_1, c_2, c_3 a k a rozhodněte.)

Jakou datovou strukturu a jaký algoritmus vyhledávání použít, tedy záleží na dalších okolnostech (praktické podmínky použití). K nim je třeba při volbě datové struktury a algoritmů.

Pole má pro vyhledávání prvků zásadní nevýhodu. Má konstantní velikost, není to tzv. dynamická datová struktura (je to statická struktura). Vložímě-li do pole o velikosti n postupně n prvků, pole se zaplní a nelze do něj vkládat další nové prvky.

Možné řešení spočívá v tom použít pole dostatečné velikosti. Vím-li, že nikdy nebudu potřebovat mít v paměti víc než n prvků (např. $n = 1000000000$), použiji pole o velikosti n.

Nevýhody:
–Často nevíme, kolik prvků budu v paměti potřebovat, tj. n neznáme.
–Plýtvání pamětí (při volbě příliš velkého n).
–Další.

Proto používáme tzv. **dynamické datové struktury**.

Datové struktury, zejména dynamické

Co je datová struktura?

Volně řečeno, způsob, způsob uložení dat v počítači a způsob, jakým můžeme k datům přistupovat.

Základní datové struktury:

- pole (známe)
- seznam (někdy také spojový seznam; jednosměrný nebo dvousměrný)
- zásobník
- fronta
- strom (jedna z nejdůležitějších, existuje mnoho variant, uvidíme)
- graf
- další . . .

“Dynamické množiny”

“Dynamická množina” je název pro obecnou datovou strukturu, která umožňuje uchovávat prvky. Název zdůrazňuje fakt, že narozdíl od množin, jak jsou chápané v matematice, “dynamické množiny” se mohou měnit (lze do nich vkládat prvky, vybírat z nich prvky). Příklady dynamických množin: zásobník, fronta, spojový seznam (jednosměrný, obousměrný), stromy, grafy.

Jaké operace je třeba s dynamickými množinami provádět, závisí na dalších okolnostech (k čemu je používáme). Např. dynamické množiny, nad kterými lze provádět operace vložení prvku, odstranění prvku a zjištění, zda daný prvek do množiny patří, se nazývají slovníky (dictionary).

Prvky množin mohou být atomické (nestrukturované), jako např. čísla, znaky. Mohou být ale strukturované. Např. prvek může být "**záznamem**" (record, např. v jazyce C datový typ struct), který sestává ze tří položek (field): celé číslo n, racionální číslo x, znak c. Tj. prvek má tvar:

celé číslo n
racionální číslo x
znak c

, např.

15
0.123
A

nebo

-154
86.7
h

, což vhodně odráží uspořádání záznamu v paměti počítače.

Je-li *rec* proměnná typu záznam, odkazujeme se na hodnotu položky *pol* takového záznamu $rec.pol$, popř. $pol[rec]$. Tedy např. $x[rec] = 0.123$.

Některé datové struktury předpokládají, že jednou z položek záznamu je tzv. **klíč** (angl. key). Hodnota klíče je často číselná hodnota. Obecně je to hodnota typu, jehož hodnoty jsou úplně uspořádané (to znamená: na hodnotách toho typu je dána relace úplného uspořádání, tj. reflexívní, antisymetrická a tranzitivní relace $\leq$ taková, že pro každé dvě hodnoty a a b daného typu je $a \leq b$ nebo $b \leq a$).

Klíč se používá pro vyhledávání (je v možině záznam s hodnotou klíče rovnou x?, popř. najdi prvek v množině s nejmenší hodnotou klíče).

Položka záznamu může být hodnota daného typu (celé číslo, racionální číslo, znak, popř. jiného atomického typu, ale i strukturovaného typu, jako např. řetězec znaků, (jiný) záznam). Důležitým atomickým typem je tzv. **ukazatel** (angl. pointer). Ukazatel je datový typ, jehož hodnota "ukazuje na místo v paměti" (více na cvičení), např. na místo v paměti kde je uložena hodnota daného typu. Hodnota ukazatele NIL (nula) indikuje, že ukazatel na místo v paměti neukazuje.

Ukazatel se často používá jako typ položky záznamu. Ukazatel je vhodný pro vytváření datových struktur, jejichž prvky jsou zřetězené (seznamy, stromy, grafy). V tom případě jsou prvky datové struktury záznamy, jejichž jednou položkou (pomocnou položkou) je ukazatel (ukazatel na záznam) a dalšími položkami jsou položky nesoucí podstatné informace (jednou z nich je obvykle klíč). Takový záznam (řekněme typu Z) má pak např. tvar:

celé číslo key
typ2 polozka2
...
typn polozkan
ukazatelNaZ ukaz

.

První položkou je klíč, v tomro případě je typu celé číslo, následují další položky, poslední je ukazatel na Z.

Některé **operace s dynamickými množinami** (lze rozdělit na dotazy a manipulace):

Search(S,k): Pokud se v množině S vyskytuje prvek s hodnotou klíče k, vrátí ukazatel na tento prvek (tj. na prvek x, pro který $key[x] = k$) nebo vrátí NIL, pokud se tam takový prvek nevyskytuje.

Insert(S,x): Vloží do množiny S prvek x (popř. prvek, na který ukazuje x).

Delete(S,x): Odstraní z S prvek, na který ukazuje ukazatel x).

Min(S): Vrátí ukazatel na prvek S s nejmenší hodnotou klíče.

Max(S): Vrátí ukazatel na prvek S s největší hodnotou klíče.

...

Zásobník

Angl. stack (LIFO, Last In First Out).

obrázek

Push, Pop, Delete

implementace zásobníku

cvičení

Fronta

Angl. queue (FIFO, First In First Out).

obrázek

Insert, Delete

varianty fronty, implementace fronty

cvičení

(Spojový) seznam

Obsahuje zřetězené prvky (lineárně zřetězené).

varianty: jednosměrný spojový seznam (singly linked list), dvousměrný spojový seznam (doubly linked list)

obrázek

Je-li x prvek seznamu (nebo ukazatel na prvek seznamu), pak
$key[x]$ je hodnota klíče prvku x (prvku, na který ukazuje x),
$next[x]$ je ukazatel, který ukazuje na další prvek seznamu
$prev[x]$ je ukazatel, který ukazuje na předchozí prvek seznamu (v případě dvousměrného seznamu)
x může obsahovat další položky ($polozka[x]$)

Prvky jednosměrného seznamu lze chápat (a implementovat) jako záznamy (typu Z)

keyType key
pointerToZ next

a seznam pak jako zřetězení těchto prvků.

Je-li L seznam (nebo ukazatel na seznam), pak
$head[L]$ je první prvek seznamu (ukazatel na první prvek seznamu)
$tail[L]$ je poslední prvek seznamu (ukazatel na poslední prvek seznamu)

vložení prvku, odstranění prvku, vyhledání prvku

cvičení

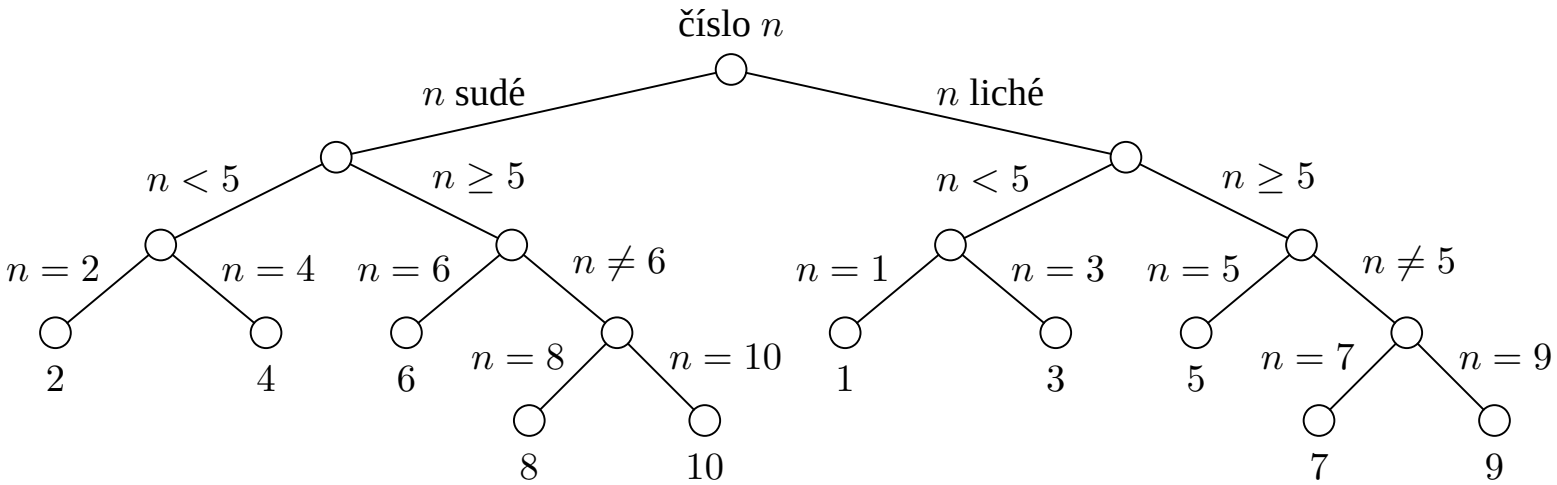
Grafy a stromy

Grafy a stromy

(Úvod např. viz Bělohlávek, Vychodil: Diskrétní matematika 1, 2, kap. 4.5, http://belohlavek.inf.upol.cz/vyuka/dm2.pdf).

Stromy: základní pojmy a vlastnosti

- Stromy jsou důležité struktury používané v informatice.
- Používají se jako datové struktury (různé druhy stromů).
- Objevují se při analýze různých problémů, např. při analýze složitosti algoritmů.
- Stromy jsou speciálním případem obecnějších struktur, zvaných grafy.
- Konkrétní strom (doplněný popisy uzlů a hran) je na následujícím obrázku. Reprezentuje algoritmus pro uhodnutí čísla zvoleného z $1, \ldots, 10$.



... začneme grafy

- Grafy slouží k popisu situací, ve kterých se vyskytují místa a spojení mezi těmito místy. Místa mohou být křižovatky, spojeními cesty mezi křižovatkami. Místa mohou být lidé, spojení mohou vyznačovat vztahy mezi lidmi.
- Místa jsou v grafu reprezentována tzv. vrcholy (uzly), spojení pak hranami.
- Pokud nezáleží na orientaci hrany, nazývá se hrana neorientovaná (cesta s obousměrným provozem), jinak orientovaná (cesta s jednosměrným provozem). Graf je neorientovaný/orientovaný, jsou-li všechny jeho hrany neorientované/orientované.
- V grafech se zavádějí různé přirozené pojmy jako cesta, vzdálenost uzlů, apod.

Grafy mají zajímavé aplikace.

Např. ve městě může být pekařská firma, která má každé ráno do prodejen pečiva rozvést zboží. Majiteli firmy přitom záleží na tom, aby se neplýtvalo pohonnými hmotami, tj. aby byl při ranním rozvozu počet ujetých kilometrů co nejmenší. Z pohledu teorie grafů jde o úlohu najít cestu, která vychází z pekařství, prochází v libovolném pořadí všemi prodejnami pečiva a končí opět v pekařství. Přitom hledáme cestu, která je ze všech takových nejkratší.

Podobný příklad: Cestující vlakem chce najít co nejrychlejší spojení ze jedné stanice do druhé. Může ji najít vyhledávacím programem na internetu. Samotný program vlastně hledá nejkratší cestu v grafu, který představuje železniční síť.

Definice Neorientovaný graf je dvojice $G = \langle V, E \rangle$, kde V je neprázdná množina tzv. **vrcholů** (někdy také **uzlů**) a $E \subseteq \{\{u, v\} \mid u, v \in V,\ u \neq v\}$ je množina dvouprvkových množin vrcholů, tzv. **(neorientovaných) hran**. **Orientovaný graf** je dvojice $G = \langle V, E \rangle$, kde V je neprázdná množina tzv. vrcholů (uzlů) a $E \subseteq V \times V$ je množina uspořádaných dvojic vrcholů, tzv. **(orientovaných) hran**.

Hrana se tedy u neorientovaných grafů chápe jako dvouprvková množina $\{u, v\}$ vrcholů $u, v \in V$. Říkáme, že hrana $\{u, v\}$ vede mezi u a v, popř. spojuje u a v. To odpovídá záměru: Graf je neorientovaný, tj. na pořadí vrcholů u hrany nezáleží. U orientovaných grafů se hrana chápe jako uspořádaná dvojice $\langle u, v \rangle$ vrcholů u a v. Pak říkáme, že hrana vede z u do v. I to odpovídá záměru: Graf je orientovaný, tj. na pořadí vrcholů u hrany záleží. Hrana $\langle u, v \rangle$ je tedy něco jiného než hrana $\langle v, u \rangle$. **Koncové vrcholy** hrany jsou u neorientované hrany $\{u, v\}$ vrcholy u a v, u orientované hrany $\langle u, v \rangle$ také vrcholy u a v.

Příklad Uvažujme množinu vrcholů $V = \{u, v, w, x, y\}$. Uvažujme množinu neorientovaných hran $E_1 = \{\{u, v\}, \{u, w\}, \{u, x\}, \{v, w\}\}$. $G_1 = \langle V, E_1 \rangle$ je neorientovaný graf a vidíme ho znázorněný na Obr. 1 vlevo. Uvažujme množinu orientovaných hran $E_2 = \{\langle u, v \rangle, \langle v, w \rangle, \langle w, u \rangle, \langle w, v \rangle, \langle x, u \rangle\}$. Pak $G_2 = \langle V, E_2 \rangle$ je orientovaný graf a vidíme ho znázorněný na Obr. 1 vpravo.

Obrázek: 1: Neorientovaný (vlevo) a orientovaný (vpravo) graf.

Definice Neorientované grafy $G_1 = \langle V_1, E_1 \rangle$ a $G_2 = \langle V_2, E_2 \rangle$ se nazývají **izomorfní**, právě když existuje bijekce $h : V_1 \to V_2$ (říká se jí izomorfismus), pro kterou

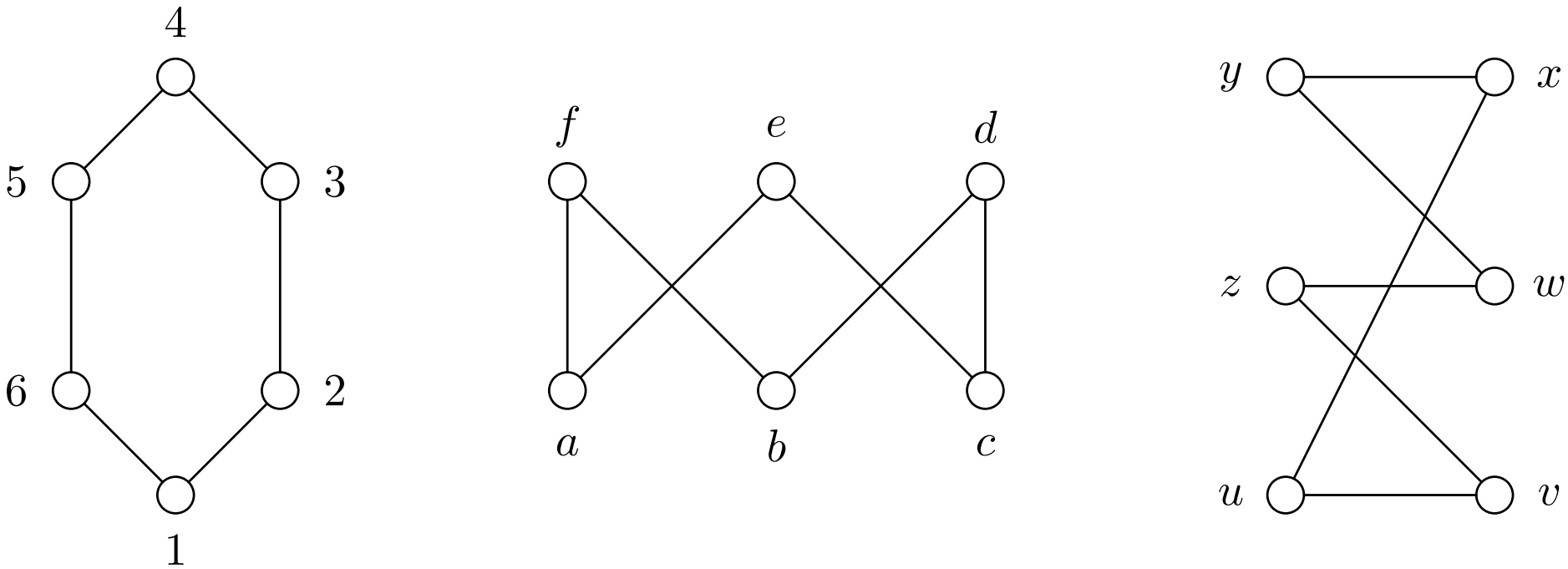
$$\{u, v\} \in E_1 \quad \text{právě když} \quad \{h(u), h(v)\} \in E_2.$$

Orientované grafy $G_1 = \langle V_1, E_1 \rangle$ a $G_2 = \langle V_2, E_2 \rangle$ se nazývají izomorfní, právě když existuje bijekce $h : V_1 \to V_2$, pro kterou

$$\langle u, v \rangle \in E_1 \quad \text{právě když} \quad \langle h(u), h(v) \rangle \in E_2.$$

Jsou-li G_1 a G_2 izomorfní, píšeme $G_1 \cong G_2$.

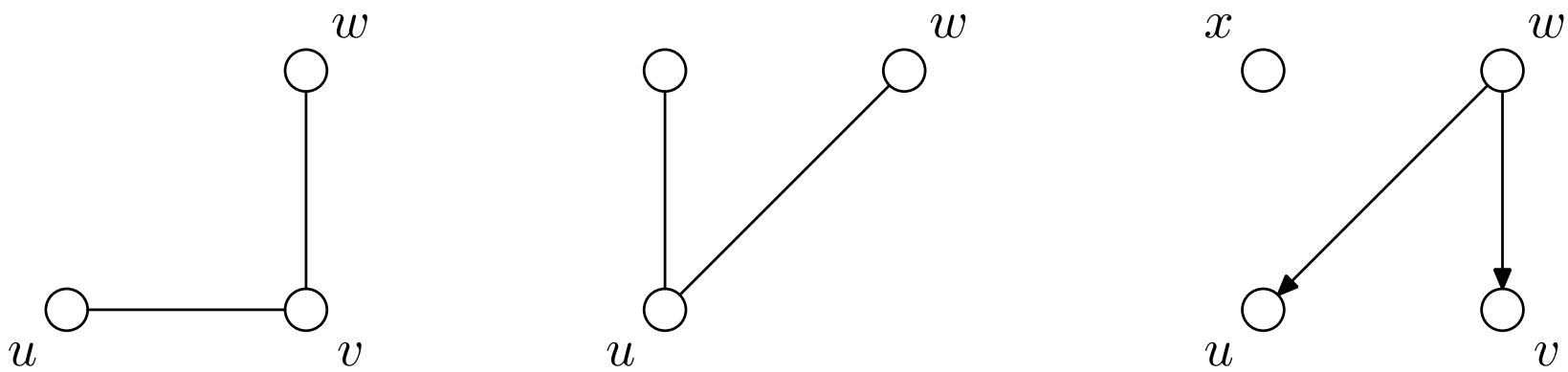
Příklad Všechny grafy z Obr. 2 jsou po dvou izomorfní. Např. bijekce h, pro kterou $h(1) = a$, $h(2) = e$, $h(3) = c$, $h(4) = d$, $h(5) = b$, $h(6) = f$, je izomorfismus mezi prvním a druhým grafem.



Obrázek: 2: Izomorfní neorientované grafy.

Definice (Orientovaný nebo neorientovaný) graf $\langle V_1, E_1 \rangle$ je **podgrafem** grafu $\langle V_2, E_2 \rangle$, právě když $V_1 \subseteq V_2$ a $E_1 \subseteq E_2$. Podgraf $\langle V_1, E_1 \rangle$ grafu $\langle V_2, E_2 \rangle$ se nazývá **indukovaný**, právě když E_1 obsahuje každou hranu z E_2, jejíž oba koncové vrcholy patří do V_1.

Příklad První dva grafy na Obr. 3 jsou podgrafy grafu z Obr. 1 vlevo, přitom první z nich není indukovaný (není v něm hrana $\{u, w\}$), druhý ano. Třetí graf na Obr. 3 je podgrafem grafu z Obr. 1 vpravo.



Obrázek: 3: Podgrafy.

Definice (cestování) Sled v (neorientovaném nebo orientovaném) grafu $G = \langle V, E \rangle$ je posloupnost

$$v_0, e_1, v_1, e_2, v_2, \ldots, e_n, v_n,$$

kde $v_i \in V$ jsou vrcholy, $e_j \in E$ jsou hrany a platí, že

- $e_i = \{v_{i-1}, v_i\}$ pro $i = 1, \ldots, n$, je-li G neorientovaný,
- $e_i = \langle v_{i-1}, v_i \rangle$ pro $i = 1, \ldots, n$, je-li G orientovaný.

číslo n se nazývá **délka sledu**. Sled $v_0, e_1, v_1, e_2, v_2, \ldots, e_n, v_n$, se nazývá

- **uzavřený**, je-li $v_0 = v_1$,
- **tah**, neopakuje-li se v něm žádná hrana (tj. pro $i \neq j$ je $e_i \neq e_j$),
- **cesta**, neopakuje-li se v něm žádný vrchol (tj. pro $i \neq j$ je $v_i \neq v_j$),
- **kružnice**, je-li $v_0 = v_n$ a s výjimkou vrcholů v_0 a v_1 jsou každé dva vrcholy různé.

Vzdálenost z vrcholu u do vrcholu v je délka cesty z u do v, která má ze všech cest z u do v délku nejmenší.

Říkáme také, že sled $v_0, e_1, \ldots, v_n$ vede z v_0 do v_n. Z definice máme, že každý tah je sledem. Každá cesta je tahem, neboť neopakují-li se ve sledu vrcholy, nemohou se opakovat ani hrany. Kružnice nemůže být cestou, protože se v ní opakují vrcholy (první a poslední).

Příklad Uvažujme graf na Obr. 4 vlevo.
$u, \{u, w\}, w, \{w, v\}, v, \{v, u\}, u, \{u, w\}, w$ je sled, který není tahem (a tedy ani cestou), protože se v něm opakuje hrana $\{u, w\}$.
$u, \{u, w\}, w, \{w, v\}, v, \{v, u\}, u, \{u, x\}, x$ je tah, který není cestou, protože se v něm opakuje vrchol u. Sled
$x, \{x, u\}, u, \{u, w\}, w, \{w, v\}, v, \{v, u\}, u, \{u, x\}, x$ je sice uzavřený, ale není to kružnice, protože se v něm opakuje vrchol u. Sled
$u, \{u, w\}, w, \{w, v\}, v, \{v, u\}, u,$ je kružnice.

Pro graf na Obr. 4 vpravo je posloupnost $u, \langle u, v\rangle, v, \langle v, w\rangle, w, \langle w, u\rangle, u$ sledem, který je kružnicí.

Obrázek: 4: Neorientovaný (vlevo) a orientovaný (vpravo) graf.

Defnice Neorientovaný graf $G = \langle V, E \rangle$ se nazývá **souvislý**, právě když pro každé dva vrcholy $u, v \in V$ existuje sled z u do v. **Komponenta** neorientovaného grafu je každý jeho maximální souvislý podgraf.

Komponenta grafu $G = \langle V, E \rangle$ je tedy podgraf indukovaný množinou vrcholů $V' \subseteq V$ takovou, že každé dva vrcholy z V' lze spojit tahem a že k V' není možné přidat další vrchol, aby to stále platilo. Např. graf na Obr. 1 (předchozí slajd) není souvislý (vrcholy x a y nejsou spojeny sledem). Jeho podgraf indukovaný vrcholy u, v a w je souvislý, ale není to komponenta, protože není maximální souvislý. Komponenty v tomto grafu jsou dvě. První je podgraf indukovaný vrcholy u, v, w, x, druhá je podgraf indukovaný vrcholem y. I obecně platí, že komponenty tvoří „rozklad grafu“.

Věta Nechť $G_1 = \langle V_1, E_1 \rangle, \ldots, G_n = \langle V_n, E_n \rangle$ jsou všechny komponenty grafu $G = \langle V, E \rangle$. Pak každý vrchol $v \in V$ patří právě do jedné V_i a každá hrana $e \in E$ patří právě do jedné E_i.

Důkaz Vezměme vrchol $v \in V$. Podgraf indukovaný $\{v\}$ je zřejmě souvislý. Proto je podgrafem nějakého maximálního souvislého podgrafu grafu G, tj. komponenty G_i. Proto $v \in V_i$, tj. v patří aspoň do jedné z množin $V_1, \ldots, V_n$. Ukažme, že v nemůže patřit do dvou různých $V_i \neq V_j$. Kdyby $v \in V_i \cap V_j$, pak uvažujme nějaký $v_i \in V_i - V_j$ (takový existuje, protože G_i a G_j jsou komponenty, a tedy $V_i \not\subseteq V_j$ a $V_j \not\subseteq V_i$). Protože G_i je souvislý, existuje sled $v_i, e_1, u_1, \ldots, v$. Uvažujme množinu vrcholů V', která vznikne z V_j přidáním všech vrcholů sledu $v_i, e_1, u_1, \ldots, v$. Pak $V_j \subseteq V'$ a podgraf indukovaný množinou V' je souvislý (vezmeme-li $v_1, v_2 \in V'$, pak existuje sled z v_1 do v i sled z v do v_2 a složením těchto sledů dostaneme sled z v_1 do v_2). Tedy G_j by nebyl maximální souvislý podgraf, tj. nebyl by komponetou, což je spor s předpokladem.
Že každá hrana $e \in E$ patří právě do jedné E_i dostaneme podobnou úvahou, když si uvědomíme, že pro $e = \{u, v\}$ je podgraf indukovaný množinou vrcholů $\{u, v\}$ je souvislý.

Definice Stupeň vrcholu $v \in V$ grafu $\langle V, E \rangle$ je počet hran, jejichž jedním z koncových vrcholů je v, a značí se $\deg(v)$.

U orientovaných grafů se někdy zavádí vstupní a výstupní stupeň vrcholu jako počet hran, které do přicházejí, a počet hran, které z něj vycházejí. Stupeň vrcholu je pak součet vstupního a výstupního stupně. Pro graf na Obr. 1 vlevo je $\deg(u) = 3$, $\deg(v) = 2$, $\deg(w) = 2$, $\deg(x) = 1$, $\deg(y) = 0$. Pro graf vpravo je $\deg(u) = 3$, $\deg(v) = 3$, $\deg(w) = 3$, $\deg(x) = 1$, $\deg(y) = 0$.

Věta V grafu $G = \langle V, E \rangle$ je $\sum_{v \in V} \deg(v) = 2|E|$.

Důkaz Máme dokázat, že součet stupňů všech vrcholů grafu je roven dvojnásobku počtu hran. Tvrzení je téměř zřejmé, uvědomíme-li si následující. Každá hrana $e \in E$ má dva vrcholy, u a v. Hrana e přispívá jedničkou do $\deg(u)$ (je jednou z hran, jejichž počet je roven $\deg(u)$), jedničkou do $\deg(v)$ a do stupně žádného jiného vrcholu nepřispívá. Hrana e tedy přispívá právě počtem 2 do $\sum_{v \in V} \deg(v)$. To platí pro každou hranu. Proto $\sum_{v \in V} \deg(v) = 2|E|$.

Důsledek Počet vrcholů lichého stupně je v libovolném grafu sudý.

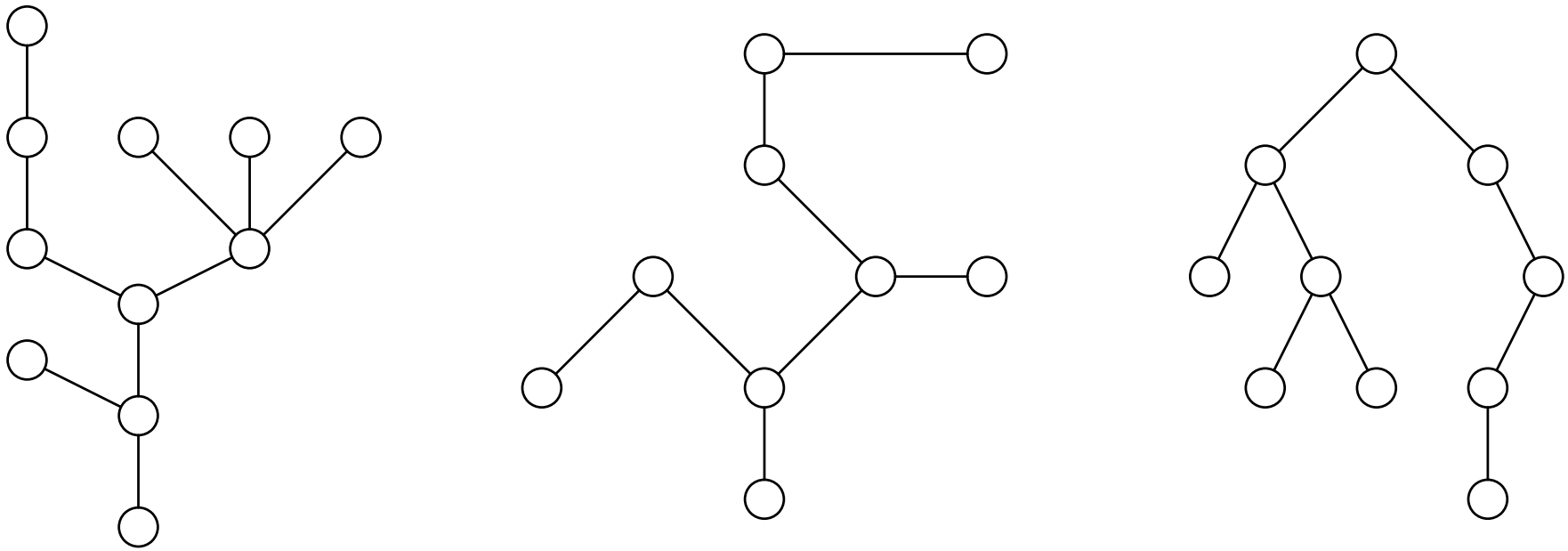
Důkaz Označme S a L množiny vrcholů, které mají sudý a lichý stupeň. Protože každý vrchol patří buď do S, nebo do L, je $\sum_{v \in V} \deg(v) = \sum_{v \in S} \deg(v) + \sum_{v \in L} \deg(v)$. Je jasné, že $\sum_{v \in S} \deg(v)$ je sudé číslo. Podle Věty je $\sum_{v \in V} \deg(v) = 2|E|$, tedy $\sum_{v \in V} \deg(v)$ je sudé číslo. Proto i $\sum_{v \in L} \deg(v)$ musí být sudé číslo. Kdyby byl počet vrcholů s lichým stupněm lichý, byl by $\sum_{v \in L} \deg(v)$ součet lichého počtu lichých čísel, a tedy by $\sum_{v \in L} \deg(v)$ bylo liché číslo, což není možné. Počet vrcholů s lichým stupněm je tedy sudý.

... a pokračujeme se stromy

Stromy jsou speciální grafy, které dostaly název podle toho, že vypadají podobně jako stromy, popř. keře v přírodě. Typický strom-graf vypadá jako strom v přírodě. Má svůj kořen (speciální vrchol), ve kterém se větví (vedou z něj hrany) do míst (vrcholů), ve kterých se opět větví atd. Stromy jsou grafy, které mají nejčastější použití. Setkáváme se s nimi v běžném životě (různá členění, např. členění knihy na kapitoly, podkapitoly atd., mají stromovou strukturu), jako uživatelé počítačů (stromová struktura adresářů) i jako informatici (rozhodovací stromy, vyhledávací stromy).

Stromy lze zavést několika ekvivalentními způsoby. Jeden zvolíme a o ostatních doážeme, že jsou s ním ekvivalentní.

Definice Strom je neorientovaný souvislý graf bez kružnic.
(Někdy se zavádí i pojem strom pro orientované grafy. My se tím ale zabývat nebudeme.) Příklady stromů vidíme na Obr. 5.



Obrázek: 5: Stromy.

Vrchol grafu se stupněm 1 se nazývá **koncový**. Koncový vrchol stromu se nazývá **list**. Následující tvrzení ukazují, že stromy vznikají přidáváním listů.

Věta V každém stromu s alespoň dvěma vrcholy existují alespoň dva listy.

Důkaz Uvažujme cestu $v_0, e_1, \ldots, v_n$, která má maximální délku. Tvrdíme, že v_0 i v_n jsou listy. Kdyby např. v_0 nebyl list, pak by existovala hrana $e = \{v, v_0\}$, která je různá od e_1. Kdyby $v = v_i$ pro nějaké $i = 2, \ldots, n$, pak by $v, e, v_0, \ldots, v_i$ byla kružnice, což je spor s tím, že G je strom. Pak ale $v, e, v_0, e_1, \ldots, v_n$ je cesta, která je delší než $v_0, e_1, \ldots, v_n$, což je opět spor. Podobně se ukáže, že v_n je list.

Věta Pro graf G a jeho koncový vrchol v jsou následující tvrzení ekvivalentní.

1. G je strom.
2. $G - v$ je strom.

Poznamenejme, že $G - v$ vznikne z G vymazáním v a hrany, která do něho vede.

Důkaz Tvrzení je zřejmé.

Věta Pro neorientovaný graf $G = \langle V, E \rangle$ jsou následující tvrzení ekvivalentní.

1. G je strom.
2. Mezi každými dvěma vrcholy existuje právě jedna cesta.
3. G je souvislý a vynecháním libovolné hrany vznikne nesouvislý graf.
4. G neobsahuje kružnice, ale přidáním jakékoli hrany vznikne graf s kružnicí.
5. G neobsahuje kružnice a $|V| = |E| + 1$.
6. G je souvislý a $|V| = |E| + 1$.

Důkaz

„1. $\Rightarrow$ 2.“: Předpokládejme, že G je strom. Protože G je podle definice souvislý, existuje mezi každými dvěma vrcholy cesta. Kdyby mezi nějakými vrcholy u a v existovaly dvě různé cesty, znamenalo by to, že v G je kružnice. Totiž, jsou-li ty cesty $u, e_1, v_1, \ldots, e_n, v$ a $u, e'_1, v'_1, \ldots, e'_m, v$, pak jejich spojením je uzavřený sled $s = u, e_1, v_1, \ldots, e_n, v, e'_m, \ldots, v'_1, e'_1, u$. Pokud ten ještě není kružnicí, opakuje se v něm nějaký vrchol $w \neq u$, tj. exsituje v něm úsek $w, \ldots, w$. Nahrazením tohoto Úseku jen uzlem w toto opakování odstraníme. Pokud se ve zbylém uzavřeném s' úseku už žádný vrchol neopakuje, je s' hledanou kružnicí. Pokud ano, můžeme v něm opět nahradit nějakou část $w', \ldots, w'$ uzlem w'. Tak postupně dostaneme kružnici. To je ale spor s tím, že G je strom.

„2. $\Rightarrow$ 3.“: Vynechme hranu $e = \{u, v\} \in E$ stromu G, dostaneme tak graf G'. Kdyby byl G' souvislý, existovala by v něm cesta $u, e_1, \ldots, v$ mezi u a v. To by ale znamenalo, že v G existují dvě cesty z u do v: jednou je $u, e_1, \ldots, v$, druhou je u, e, v. To je spor s tím, že mezi každými dvěma vrcholy je v G právě jedna cesta.

Důkaz (pokrač.)

„3. $\Rightarrow$ 4.“: Kdyby G obsahoval kružnici, pak odstraněním jedné její hrany dostaneme opět souvislý graf, což je spor s předpokladem 3. Kdyby po přidání hrany $e = \{u, v\}$ nevznikla kružnice, v G by neexistovala cesta mezi u a v (kdyby ano, přidáním e k této cestě dostaneme kružnici), a tedy G by nebyl souvislý, což je spor s 3.

„1. $\Rightarrow$ 5.“: Máme ukázat, že ve stromu je $|V| = |E| + 1$. Dokažme to indukcí podle počtu vrcholů. Pro $n = 1$ vrchol to zřejmě platí (pak je totiž $|E| = 0$). Předpokládejme, že to platí pro každý strom o n vrcholech. Má-li G $n + 1$ vrcholů, odstraňme z něj nějaký list. Výsledný graf $\langle V', E' \rangle$ je podle uvedené Věty opět strom, má n vrcholů, a tedy podle předpokladu platí $|V'| = |E'| + 1$. Protože však $|V| = |V'| + 1$ a $|E| = |E'| + 1$, platí i $|V| = |E| + 1$.

Důkaz (pokrač.)

„5. $\Rightarrow$ 6.“: Máme dokázat, že G je souvislý. K tomu zřejmě stačí dokázat, že G má právě jednu komponentu.

Nechť k je počet komponent grafu G. Vyberme z každé komponenty po jednom vrcholu a označme tyto vrcholy $v_1, \ldots, v_k$. Přidejme $r-1$ hran $\{v_1, v_2\}, \{v_2, v_3\}, \ldots, \{v_{k-1}, v_k\}$ ke grafu G. Takto vzniklý graf $G' = \langle V, E' \rangle$ je strom (je souvislý a nemá kružnice, protože G neměl kružnice). Z výše dokázaného „1. $\Rightarrow$ 5.“ plyne, že $|V| = |E'| + 1$. Podle předpokladu je ale $|V| = |E| + 1$. Tedy $|E| = |E'|$. Protože $|E'| = |E| + r - 1$, je $r = 1$, tedy G má právě jednu komponentu, tj. je souvislý.

Důkaz (pokrač.)

„6. $\Rightarrow$ 1.“: Dokážeme indukcí podle počtu vrcholů. Má-li G jeden vrchol, je tvrzení zřejmé. Předpokládejme, že tvrzení platí pro každý graf s n vrcholy a že G má $n+1$ vrcholů a splňuje $|V| = |E| + 1$. Součet stupňů jeho vrcholů je $2|E| = 2|V| - 2$. Ze souvislosti plyne, že každý vrchol má stupeň aspoň 1. Kdyby měl každý vrchol stupeň aspoň 2, byl by součet stupňů všech vrcholů aspoň $2|V|$, ale ten součet je $2|V| - 2 < 2|V|$. Tedy musí existovat vrchol v stupně právě 1, tj. list. Jeho odstraněním dostaneme graf $G' = \langle V', E' \rangle = G - v$, který je zřejmě souvislý, má n vrcholů a platí pro něj $|V'| = |V| - 1$, $|E'| = |E| - 1$. G' tedy splňuje $|V'| = |E'| + 1$ a z indukčního předpokladu plyne, že je to strom. Proto je i G strom (viz Větu uvedenou výše).

Kořenové stromy

Definice Kořenový strom je dvojice $\langle G, r \rangle$, kde $G = \langle V, E \rangle$ je strom a $r \in V$ je vrchol, tzv. **kořen**.

Kořenový strom je tedy strom, ve kterém je vybrán jeden vrchol (kořen). Může to být kterýkoliv vrchol. Bývá to ale vrchol, který je v nějakém smyslu na vrcholu hierarchie objektů, která je stromem reprezentována.

To, že je v kořenovém stromu jeden vrchol pevně zvolený a že ve stromu exsituje mezi vrcholy jediná cesta, umožňuje ve kořenovém stromu zavádět uspořádání vrcholů. Na základě tohoto uspořádání se stromy kreslí. Základem je následující definice.

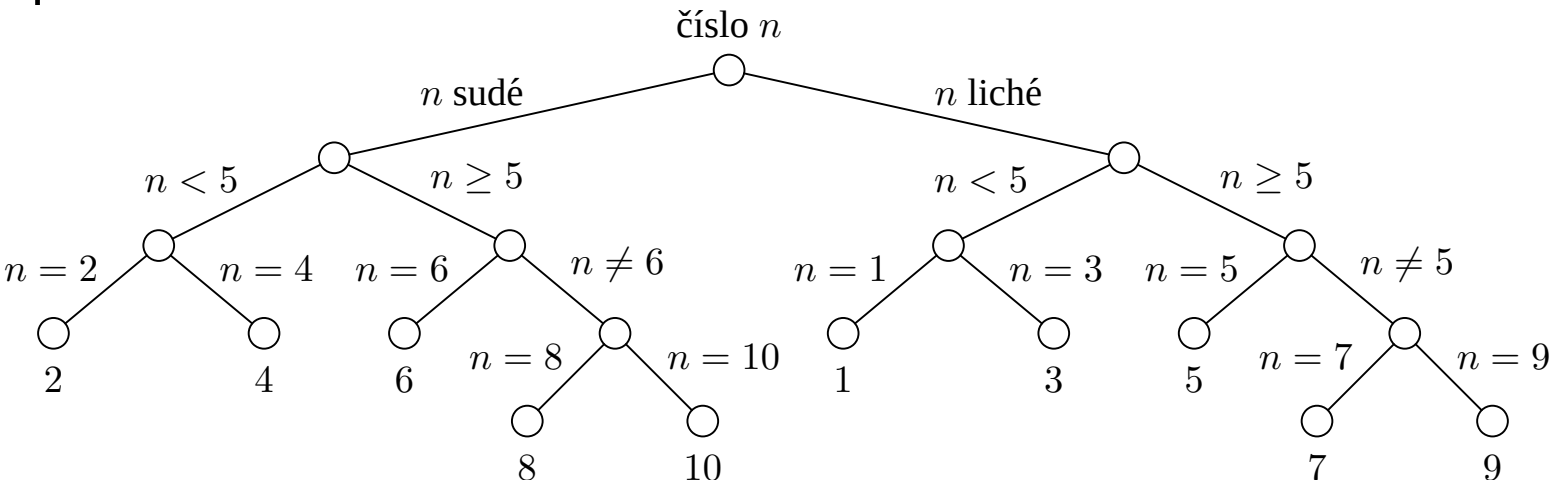
Definice Nechť $\langle G, r \rangle$ je kořenový strom.

- Vrchol v se nazývá **potomek** vrcholu u (u se nazývá **rodič** vrcholu v), právě když cesta z kořene r do v má tvar $r, \ldots, u, e, v$.
- **Úroveň** vrcholu v je délka cesty od kořene r do v.
- **Hloubka** (nebo výška) stromu $\langle G, r \rangle$ je největší z úrovní jeho listů.

Kořenový strom hloubky h se nazývá **vyvážený**, právě když každý jeho list má úroveň h nebo $h - 1$.

Vrchol může mít několik potomků, ale má právě jednoho rodiče (ukažte). Hloubku lze také definovat jako délku nejdelší cesty, která vychází z kořene (ukažte).

Na základě pojmů rodič-potomek a úroveň se kořenové stromy kreslí: Nejvýše se nakreslí kořen, pod něj se nakreslí jeho potomci, tj. vrcholy, které mají úroveň 1. Vrcholy úrovně $l + 1$ přitom kreslíme pod vrcholy úrovně l tak, aby se hrany na obrázku nekřížily. Toho lze zřejmě dosáhnout tak, že všechny potomky $v_1, \ldots, v_n$ vrcholu v nakreslíme pod vrchol v tak, že mezi libovolnými dvěma potomky v_i a v_j vrcholu v buď není žádný vrchol, nebo opět potomek vrcholu v. Příklad stromu nakresleného tímto způsobem vidíme zde:



Zvolíme-li v kořenovém stromu libovolný vrchol v, pak vrcholy, které se nacházejí „pod ním“, indukují podgraf, který se nazývá **podstrom indukovaný** vrcholem v.

Definice Kořenový strom se nazývá m-**ární**, právě když každý jeho vrchol má nejvýše m potomků. 2-ární strom se nazývá **binární**. Kořenový strom se nazývá **úplný** m-**ární**, právě když každý jeho vrchol nemá buď žádného nebo má právě m potomků.

Strom na předchozím slajdu je tedy úplný binární strom, který je vyvážený. Má hloubku 4.

Základní kombinatorická tvrzení o stromech

Věta Nechť G je m-ární strom s l listy a hloubkou h.

(1) Je-li G úplný m-ární strom, ve kterém mají všechny listy stejnou úroveň, pak $l = m^h$, neboli $h = \log_m l$, a počet vnitřních uzlů je $\frac{m^h-1}{m-1}$.

(2) $l \leq m^h$ a $h \geq \lceil \log_m l \rceil$.

(3) Je-li G vyvážený a každý uzel má 0 nebo m potomků až na nejvýše jeden uzel úrovně $h-1$, který může mít ≥ 2 ale $< m$ potomků (takovým je např. každý vyvážený úplný m-ární strom), je $h = \lceil \log_m l \rceil$.

Důkaz

(1) Na úrovni $g = 0$ je $m^g = m^0 = 1$ uzel; je-li na úrovni g počet uzlů roven m^g, pak na úrovni $g + 1$ je uzlů $m^g \cdot m = m^{g+1}$, protože každý uzel úrovně g má m potomků.

Je-li tedy h hloubka stromu, má strom $l = m^h$ listů (listy jsou právě uzly na úrovni h).

Počet vnitřních uzlů je roven součtu počtu všech uzlů v úrovních $0, \ldots, h-1$, tj. je roven

$$1 + m + m^2 + \cdots + m^{h-1} = \frac{m^h - 1}{m - 1}.$$

Důkaz (pokrač.)

(2): Ukažme nejdřív $l \leq m^h$. Máme tedy ukázat, že úplný m-ární strom hloubky h nemůže obsahovat více než m^h listů. Představme si tedy takový strom, který má listů nejvíce (je nejplnější). Je jasné, že to bude strom, který je úplný m-ární a ve kterém má každý list úroveň h, tj. strom z části (1). Dle (1) má právě m^h listů. Každý jiný m-ární strom hloubky h má nejvýše m^h listů, tedy $l \leq m^h$.

Zlogaritmujme nyní tuto nerovnost při základu m. Protože logaritmus je rostoucí funkce (tj. z $x < y$ plyne $\log_m(x) < \log_m(y)$), dostáváme $\log_m l \leq h$. Protože pro $x \leq y$ je $\lceil x \rceil \leq \lceil y \rceil$ a pro celé číslo x je $\lceil x \rceil = x$, máme $\lceil \log_m l \rceil \leq \lceil h \rceil = h$.

Důkaz (pokrač.)

(3): Protože je G vyvážený, má listy jen na úrovních h a $h-1$ (na úrovni $h-1$ nemusí být žádný). Každý list na úrovni $0, 1, \ldots, h-2$ má tedy právě k potomků. Z (1) plyne, že na úrovni $h-1$ je právě m^{h-1} uzlů. Nejmenší počet listů ve stromu je tedy $m^{h-1}+1$ (pokud má potomky pouze jeden uzel úrovně $h-1$ a pokud počet těchto potomků je 2), největší počet listů je m^h (pokud má každý uzel úrovně h právě m potomků). Platí tedy

$$m^{h-1} \leq l \leq m^h,$$

po zlogaritmování tedy

$$h-1 < \log_m l \leq \log_m m^h = h,$$

tedy

$$h-1 < \lceil \log_m l \rceil \leq h,$$

z čehož plyne

$$\lceil \log_m l \rceil = h.$$

Důkaz je hotov.

Stromy a logaritmy

Uvedený vztah $h = \lceil \log_m l \rceil$ je často používaným vztahem při analýze rekurzívních algoritmů, které pracují metodou „rozděl a panuj“.

Uvedené vztahy nabízejí následující, „geometrické“ pohledy na význam logaritmu (podle definice je $\log_m l$ číslo x, pro které $m^x = l$):

Pohled 1: Je dáno l objektů, které chceme umístit do listů m-árního stromu. $\lceil \log_m l \rceil$ je nejmenší hloubka m-árního stromu (hloubka nejnižšího m-árního stromu), pro který to lze provést (plyne z (3), uvědomíme-li si, že požadovaný strom splňuje podmínky popsané v (3)).

Pohled 2 (pohled 1 s interpretací): Máme sestavit návod (postup, algoritmus) na uhodnutí čísla x, které si myslí osoba O. x je některým z čísel $1, \ldots, l$. Osobě O můžeme klást otázky, které mají m možných odpovědí. Např. „Je $x \leq 5$?“ pro $m = 2$ (odpovědi „ano“, „ne“), „Ve kterém intervalu z $[1, 3], [4, 5], [5, l]$ je x?“ pro $m = 3$. Návod má být popisem, jak otázky klást, tj. např. jakou otázku položit jako další, pokud je odpověď na předchozí otázku „ano“. Návod musí být správný, tj. musí vést k uhodnutí každého zvoleného čísla. Uhodnutí zajistí instrukce typu „je-li odpověď „ano“, je zvoleným číslem 4.“
Označne h kvalitu návodu, tj. je nejmenší číslo takové, že pro uhodnutí libovolného x nebude podle návodu položeno víc než h otázek.

Jaký je nejlepší takový návod, tj. návod, jehož kvalita h je nejmenší možná?

Je zřejmé, že každý návod lze reprezentovat úplným m-árním stromem s l nebo více listy (více, pokud k nějakému číslu vedou různé cesty). Příkladem je výše uvedený strom s kořenem označeným „číslo n“. Kvalita návodu je pak rovna hloubce stromu. Podle Pohledu 1 je $h = \lceil \log_m l \rceil$.

Pohled 3 (dělení objektů do m skupin): Varianta 1: Objekty umístěny v listech. Jaká je nejmenší hloubka takového stromu?

Pohled 4 (dělení objektů do m skupin): Varianta 2: Objekty umístěny v listech i vnitřních uzlech. Jaká je nejmenší hloubka takového stromu?

Reprezentace kořenových stromů

Binární stromy

- Podobně jako spojové seznamy: spojovými strukturami.
- Spojované prvky jsou záznamy, které obsahují položky: key (klíč), left, right (ukazatele na levého a pravého potomka), parent (ukazatel na rodiče). Tj. záznamy jsou následujícího typu:

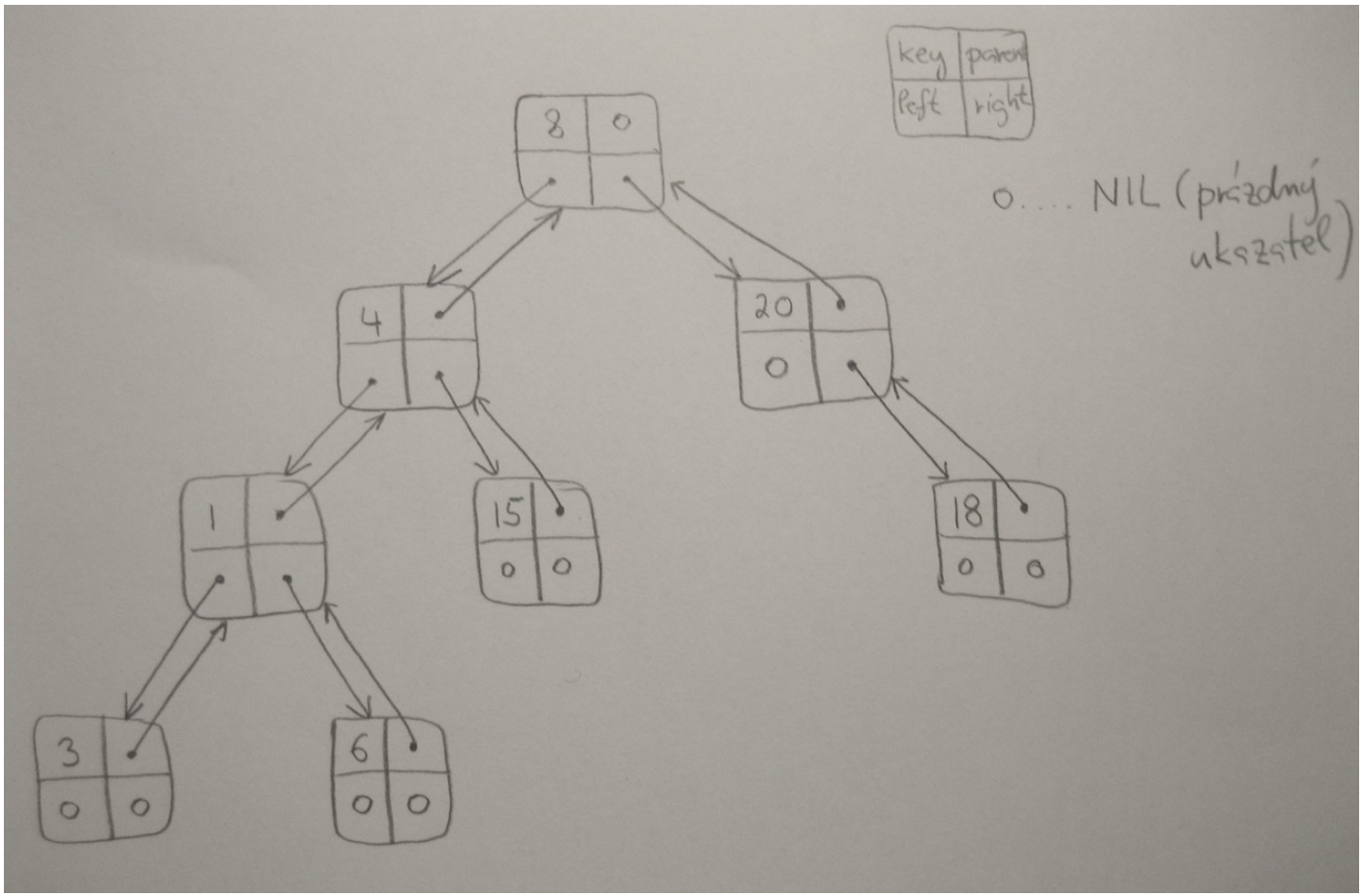
keyType key
pointerToZ left
pointerToZ right
pointerToZ parent

- Podle okolností se používají jiné typy záznamů, např. není přitomen ukazatel na rodiče:

keyType key
pointerToZ left
pointerToZ right

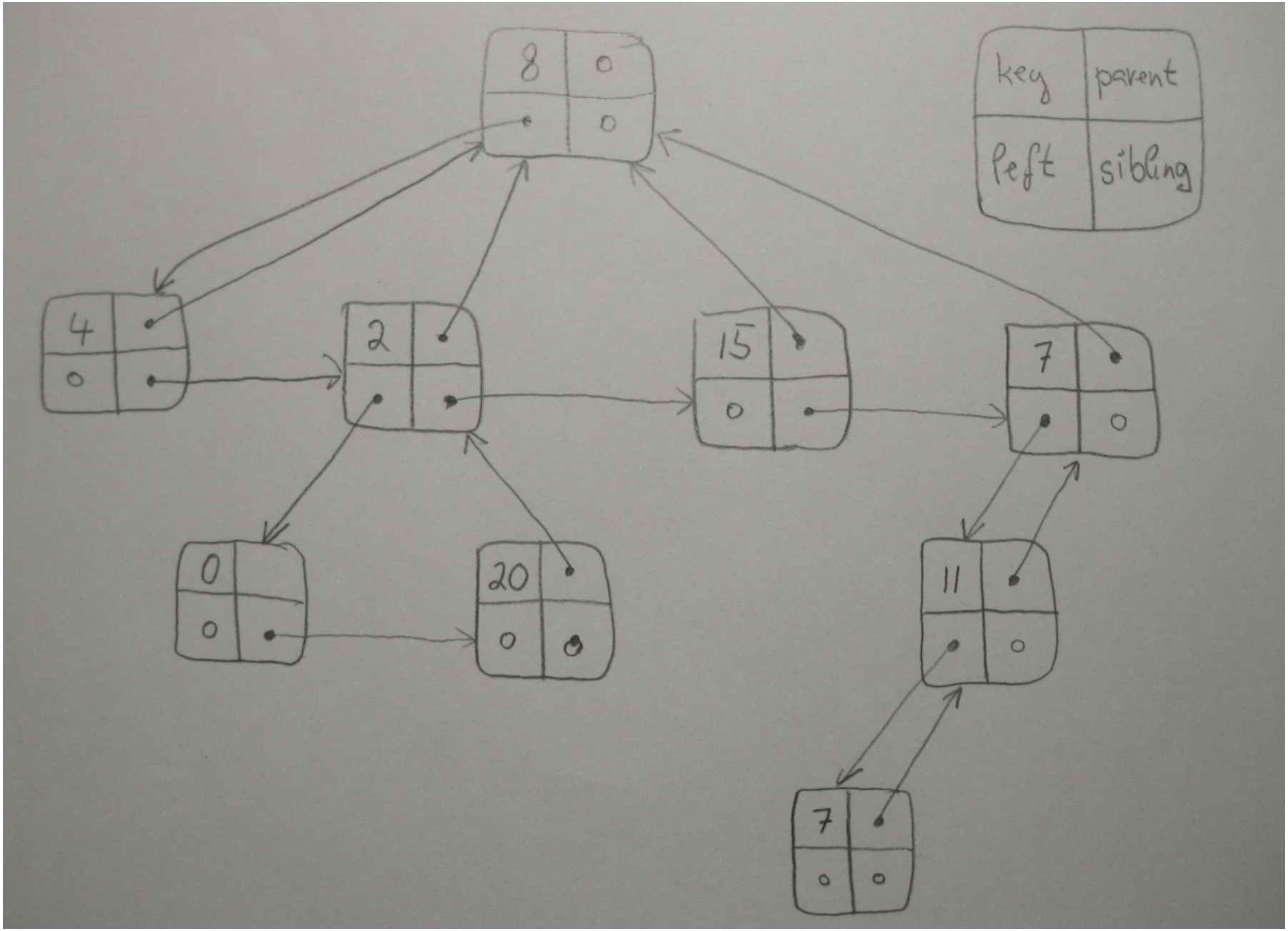
– Strom je pak reprezentován ukazatelem na kořen. Je-li T strom, je $root[T]$ (ukazatel na) jeho kořen, $left[root[T]]$ je (ukazatel na) levý potomek kořene, $key[left[root[T]]]$ je hodnota klíče v levém potomku kořene atd.

– Následující obrázek ilustruje takovou reprezentaci.



m-ární stromy

- Je možné jako binární stromy, záznam obsahuje místo 2 položek left a right m položek child1, child2, ..., childm. Nevýhoda: Pokud uzly mívají méně než m potomků, plýtvá se místem.
- Lepší je tzv. „left-child-right-sibling“ způsob („levý-potomek-pravý-sourozenec“, reprezentace pomocí sourozenců). Uzel obsahuje ukazatel na prvního potomka, místo ukazatele na druhého potomka pak ukazatel na jeho prvního sourozence (to je uzel, který má stejného rodiče). Ukazatele se jmenují left a sibling.
- Ostatní je stejné jako u výše popsané reprezentace binárních stromů.
- (Ukazatel na) druhý potomek kořene stromu T je tedy $sibling[left[root[T]]]$, hodnota klíče třetího potomka kořene stromu T je $key[sibling[sibling[left[root[X]]]]]$ apod..
- Tento způsob je možné použít i pro reprezentaci stromů s potenciálně neomezeným (dopředu neznámým) počtem potomků uzlů.
- Příklad takové reprezentace ja na dalším obrázku.


key
parent
left
sibling
8
4
2
15
7
0
20
11
7

Cvičení

(1) Napište rekurzívní funkci, která vytiskne hodnoty klíčů všech uzlů binárního stromu T. Použijte reprezentaci s left, right, parent.

(2) Napište nerekurzívní funkci, která vytiskne hodnoty klíčů všech uzlů binárního stromu T. Použijte reprezentaci s left, right, parent.

(3) Napište rekurzívní funkci, která vytiskne hodnoty klíčů všech uzlů binárního stromu T. Použijte reprezentaci s left, sibling, parent.

(4) Napište nerekurzívní funkci, která vytiskne hodnoty klíčů všech uzlů binárního stromu T. Použijte reprezentaci s left, sibling, parent.

(5) Rozeberte problém pořadí, v jakém se hodnoty budou tisknout. Jaké přirozené možnosti se nabízejí?

Binární vyhledávací stromy

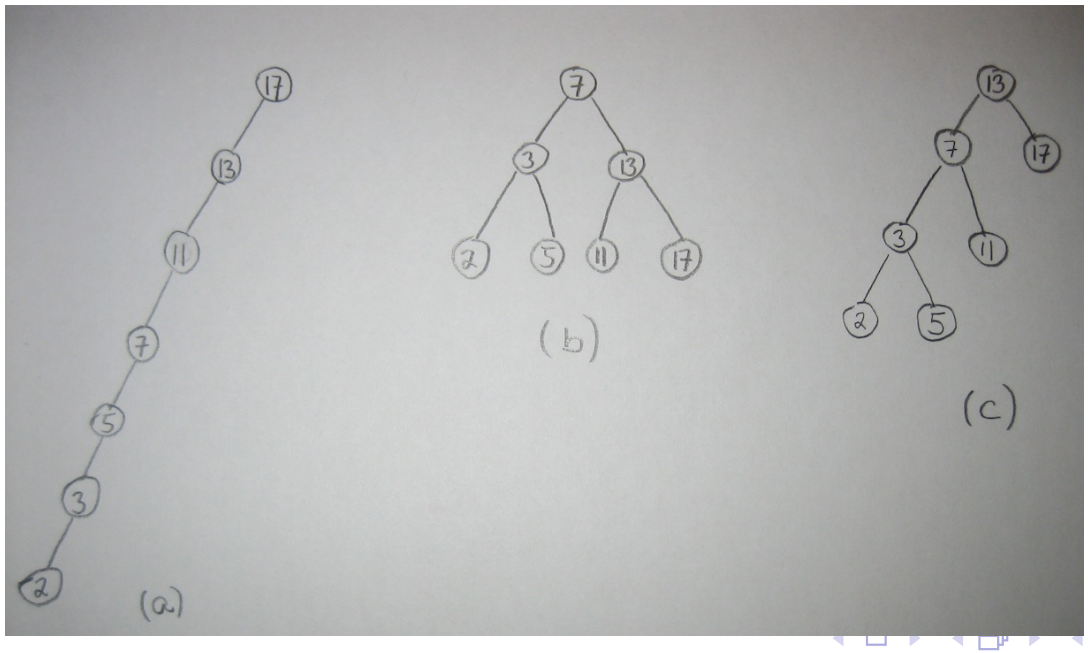
Základní vlastnosti

- Binární vyhledávací stromy jsou speciální binární stromy. Lze je proto reprezentovat výše popsanými spojovými strukturami pro repreentaci binárních stromů.
- Základní operace nad binárními vyhledávacími stromy mají časovou složitost $O(h)$, kde h je hloubka stromu. Žádoucí jsou tedy stromy s malou hloubkou.
- Z předchozího víme, že malou hloubku mají úplné binární vyvážené stromy. Takový strom s n uzly má hloubku $\Theta(\lg n)$. Binární vyhledávací stromy nemusí být vyvážené, a to je jejich nevýhoda. Jak uvidíme, v nejhorším případě má binární vyhledávací strom s n uzly hloubku $n - 1$ (řetězec).
- Binární vyhledávací stromy jsou ale základní strukturou, ze které vycházejí pokročilejší, různým způsobem vyvážené stromy. Proto se jimi budeme zabývat.

Definice Binární vyhledávací strom je takový binární strom, ve kterém pro každý uzel x platí, že hodnota (klíč) uzlu x je větší nebo rovna hodnotě každého uzlu jeho levého podstromu a menší nebo rovna hodnotě každého uzlu jeho pravého podstromu.

Levý podstrom uzlu x je strom, jehož kořenem je levý potomek uzlu x. Pokud tento potomek neexistuje, je levý podstrom prázdný. Podobně pro pravý podstrom.

Příklady binárních vyhledávacích stromů (zobrazeny jsou jen hodnoty v uzlech, nakreslete jejich reprezentaci pomocí spojových struktur).



Průchod binárním (vyhledávacím) stromem

Cílem je projít všechny uzly binárního stromu T a pro každý uzel x stromu T provést operaci *process* (např. vytisknout hodnotu $key[x]$).

Uvedeme dva základní způsoby průchodu binárním stromem (strom nemusí být vyhledávací):

- **do hloubky** a jeho tři základní varianty :
  - **inorder**,
  - **preorder**,
  - **postorder**,
- **do šířky**.

Průchody lze implementovat různými způsoby (rekurzívně, nerekurzívně). Některé ukážeme, další proberete ve cvičení.

Průchod inorder

```
Inorder-Traversal(x)
1   if x ≠ NIL
2      then Inorder-Traversal(left[x])
3           process(x)
4           Inorder-Traversal(right[x])
```

Strom T projdeme zavoláním Inorder-Traversal(root[T]).

process(x) může být např.

```
process(x)
1   print(key[x])
```

Průchod preorder

Preorder-Traversal(x)
1 **if** $x \neq NIL$
2 **then** process(x)
3 Preorder-Traversal(left[x])
4 Preorder-Traversal(right[x])

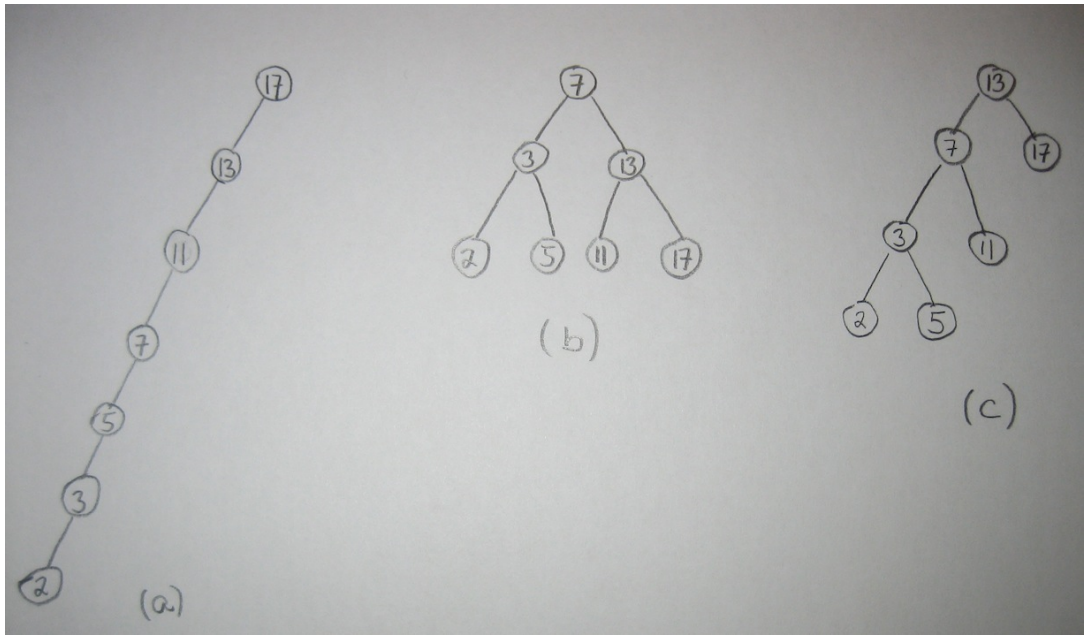
Strom T projdeme zavoláním Preorder-Traversal(root[T]).

Průchod postorder

```
Postorder-Traversal(x)
1   if x ≠ NIL
2      then Postorder-Traversal(left[x])
3           Postorder-Traversal(right[x])
4           process(x)
```

Strom T projdeme zavoláním Postorder-Traversal(root[T]).

Průchody do hloubky a do šířky lze ilustrovat následujícím obrázkem. Čísla u průchodu do šířky označují pořadí, v jakém jsou uzly zpracovávány funkcí *process* (u průchodu do hloubky jsou možnosti inorder, preorder, postorder, viz další slajd).



Strom (c):
inorder: 2, 3, 5, 7, 11, 15, 17
preorder: 15, 7, 3, 2, 5, 11, 17
postorder: 2, 5, 3, 11, 7, 17, 15
breadth-first: 15, 7, 17, 3, 11, 2, 5

Cvičení

1. Nakreslete několik binárních stromů s hodnotami v uzlech. Pro každý z nich vypište hodnoty v pořadí, v jakém budou vypisovány průchody inroder, preorder, postotder.

2. Všimněte si, že uvedené rekurzívní funkce pro průchody inorder, preorder, postorder nepoužívají ukazatel na rodiče (p). Lze je tedy použít i pro reprezentaci stromu, které tento ukazatel nepoužívá.

3. Implementujte průchody inorder, preorder, postorder nerekurzívně s využitím zásobníku (bez využití ukazatele na rodiče uzlu).

4. Implementujte průchody inorder, preorder, postorder nerekurzívně s využitím ukazatele na rodiče. Jaká je výhoda oproti nerekurzívní variantě s využitím zásobníku? (Uvažujte, kolik paměti může zásobník zabrat.)

Průchod do šířky

```
Breadth-First-Traversal(x)
1   if x ≠ NIL
2     then create empty queue q
3          enqueue(q,x)
4          while (q is not empty)
5            do node ← dequeue(q)
6               process(node)
7               if left[node] ≠ NIL then enqueue(q,left[node])
8               if right[node] ≠ NIL then enqueue(q,right[node])
```

Strom T projdeme zavoláním Breadth-First-Traversal(root[T]).
enqueue(q,x) vloží prvek x na konec fronty q.
dequeue(q) vrátí první prvek fronty q a ostraní ho z fronty.

Cvičení Nakreslete binární strom s hodnotami v uzlech a simulujte průběh algoritmu Breadth-First-Traversal.

Základní vlastnosti algoritmů průchodu

Věta Je-li T binární vyhledávací strom a funkce $process(x)$ vypíše hodnotu $key[x]$ uzlu x, vypíše Inorder-Traversal($root[T]$) hodnoty uložené v uzlech stromu T v nesestupném pořadí (od nejmenší po největší).

Důkaz Indukcí dle výšky stromu s použitím vlastnosti vyhledávacích stromů.
Dokazované tvrzení triviálně platí pro stromy výšky $h = 0$.
Nechť tvrzení platí pro stromy výšky $\leq h$. Nechť T je vyhledávací strom výšky h. Pak Inorder-Traversal($root[T]$) provede Inorder-Traversal(left[x]), pak process(x), pak Inorder-Traversal(right[x]). left[x] je ale kořen stromu výšky nejvýše h, dle předpokladu tedy Inorder-Traversal(left[x]) vypíše hodnoty jeho uzlů v nesestupném pořadí. Podobně pro right[x]. Tvrzení plyne z vlastnosti vyhledávacích stormů, totiž že všechny hodnoty ve stromu s kořenem left[x] jsou $\leq$ key[x] $\leq$ všechny hodnoty ve stromu s kořenem right[x].

Věta Předpokládejme, že časová složitost v nejhorším případě funkce *process* je $O(1)$ (tj. konstantní, tak je to např. u *print*(x)). Pak časová složitost (v nejhorším i nejlepším případě) algoritmů Inorder-, Preorder-, Postorder- a Breadth-First-Traversal je $\Theta(n)$, kde n je počet vrchodů stromu indukovaného vrcholem x (vstup).

Důkaz Pro Inorder-, Preorder-, Postorder-Traversal: Každý uzel je navštíven právě jednou. Během návštěvy je proveden konstantní počet c_1 (protože dle předpokladu je $O(1)$) instrukcí potřebných pro vykonání process(x), konstantní počet instrukcí c_2 potřebných pro zavolání Xorder-Traversal(*left*$[x]$), stejně pro Xorder-Traversal(*right*$[x]$) (zahrnuje režii toho zavolání a příp. zjištění, že levý/pravý potomek neexistuje, ne zpracování potomka, pokud existuje), tj. max. $c = c_1 + 2c_2 \geq 1$ instrukcí. Celkem tedy k ($n \leq k \leq cn$, tj. $k = \Theta(n)$ instrukcí.

Podobně pro Breadth-First-Traversal.

Vyhledávání v binárních vyhledávacích stromech

Problémy:

- Je ve stromě uzel s danou hodnotou k?
- Jaký je nejmenší/největší prvek ve stromě?
- Jaký je následovník (podle uspořádání klíčů) daného uzlu ve stromě?

Ukážeme, že tyto operace lze realizovat s časovou složitostí v nejhorším případě $O(h)$, kde h je výška stromu.

Vyhledání uzlu s danou hodnotou

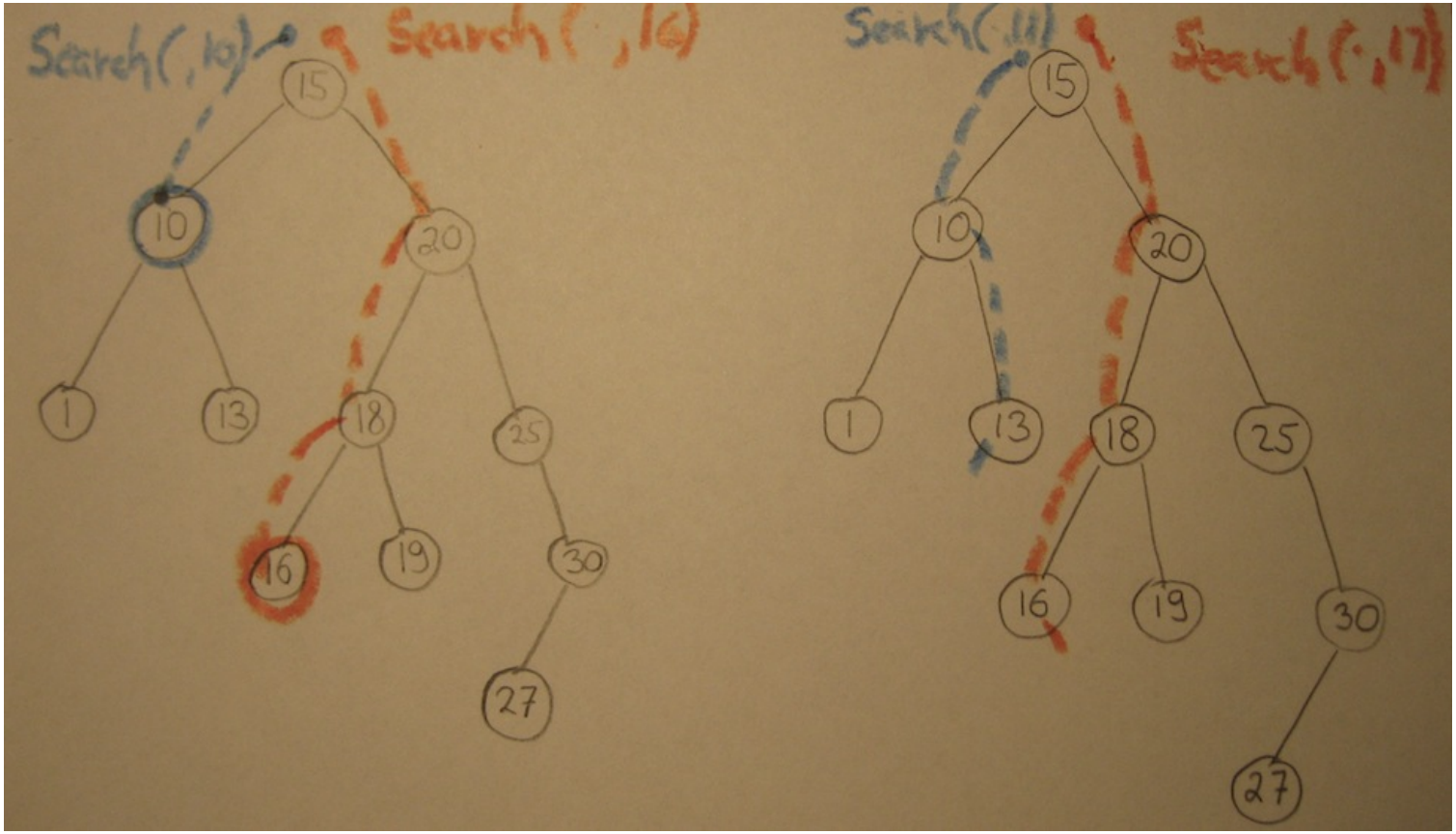
Vstup: (Ukazatel na) kořen x binárního vyhledávacího stromu a hodnota k.
Výstup: (Ukazatel na) uzel, jehož hodnota je k, pokud takový existuje; NIL, pokud neexistuje.

```
Search(x, k)
1   if k = key[x] or x = NIL
2     then return x
3   if k < key[x]
4     then return Search(left[x],k)
5     else return Search(right[x],k)
```

Popisuje přirozený postup (tak by každý postupoval).

Časová složitost v nejhorším případě $O(h)$ (v nejhorším případě dojde k listu s největší úrovní).

Příklad Vyhledávání v binárním stromu T.



Vlevo: Modře: Search($root[T]$, 10) vrátí ukazatel na uzel s hodnotou 10.
Oranžově: Průchod pro Search($root[T]$, 16) vrátí ukazatel na uzel s hodnotou 16.
Vpravo: Modře: Search($root[T]$, 11) vrátí NIL.
Oranžově:: Search($root[T]$, 17) vrátí NIL.

Implementace vyhedávání bez rekurze.

Search-Iterative(x, k)

1 **while** $k \neq key[x]$ and $x \neq NIL$
2 **do if** $k < key[x]$
3 **then** $x \leftarrow left[x]$
4 **else** $x \leftarrow right[x]$
5 **return** x

Vyhledání nejmenšího a největšího prvku

Vstup: (Ukazatel na) kořen x binárního vyhledávacího stromu.
Výstup: (Ukazatel na) uzel s nejmenší hodnotou ve stromu s kořenem x, pokud je strom neprázdný; NIL, pokud je prázdný.

Minimum(x)
1 **if** $x \neq NIL$
2 **while** $left[x] \neq NIL$
3 **do** $x \leftarrow left[x]$
4 **return** x

Správnost je zřejmá z vlastností binárních vyhledávacích stromů.

Časová složitost v nejhorším případě $O(h)$ (v nejhorším případě je nejmenší prvek listem na největší úrovni).

Vstup: (Ukazatel na) kořen x binárního vyhledávacího stromu.
Výstup: (Ukazatel na) uzel s největší hodnotou ve stromu s kořenem x, pokud je strom neprázdný; NIL, pokud je prázdný.

Maximum(x)
1 **if** $x \neq NIL$
2 **while** $right[x] \neq NIL$
3 **do** $x \leftarrow right[x]$
4 **return** x

Správnost je zřejmá z vlastností binárních vyhledávacích stromů.

Časová složitost v nejhorším případě $O(h)$ (v nejhorším případě je největší prvek listem na největší úrovni).

Následník

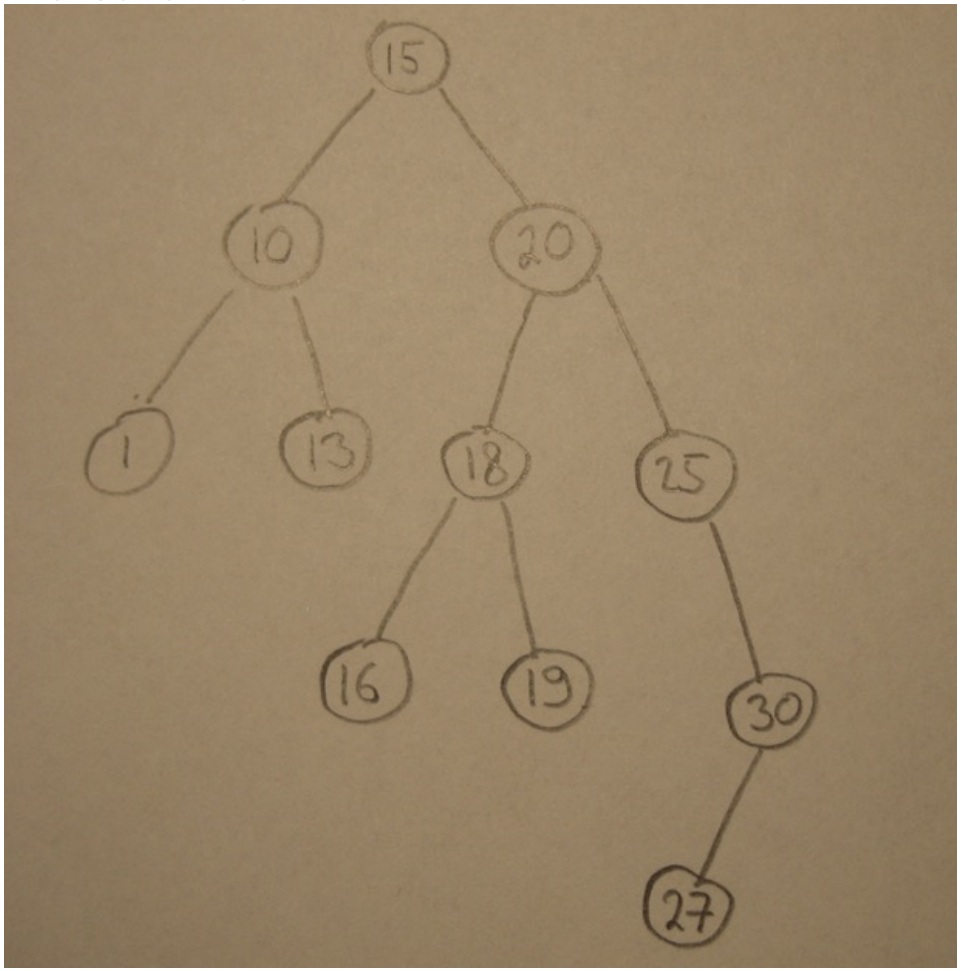
Vstup: (Ukazatel na) uzel x binárního vyhledávacího stromu.
Výstup: (Ukazatel na) uzel, který by byl zpracován při průchodu inorder hned po uzlu x (pokud jsou hodnoty uzlů různé: uzel, jehož hodnota následuje za hodnotou uzlu x), pokud takový existuje; NIL, pokud neexistuje (x je v průchodu inorder navštíven jako poslední).

Successor(x)
1 **if** $x = NIL$ **then return** NIL
2 **if** $right[x] \neq NIL$
3 **then return** Minimum($right[x]$)
4 $y \leftarrow p[x]$
5 **while** $y \neq NIL$ and $right[y] = x$
6 **do** $x \leftarrow y$
7 $y \leftarrow p[y]$
8 **return** y

Správnost je zřejmá z vlastností binárních vyhledávacích stromů (vysvětlíme na přednášce).

Časová složitost v nejhorším případě $O(h)$ (z x cestujeme buď směrem ke kořeni nebo směrem k listům, tedy nejvýše provedeme h přechodů).

Příklad Následník ve stromu T.



Simulujte průběh algoritmu pro vyhledání následníků uzlů s hodnotami 30, 27, 25, 16, 19.

Předchůdce

Vstup: (Ukazatel na) uzel x binárního vyhledávacího stromu.
Výstup: (Ukazatel na) uzel, který by byl zpracován při průchodu inorder hned před uzlem x (pokud jsou hodnoty uzlů různé: uzel, jehož hodnota předchází hodnotou uzlu x), pokud takový existuje; NIL, pokud neexistuje (x je v průchodu inorder navštíven jako první).

Predecessor(x)
1 **if** $x = NIL$ **then return** NIL
2 **if** $left[x] \neq NIL$
3 **then return** Maximum($left[x]$)
4 $y \leftarrow p[x]$
5 **while** $y \neq NIL$ and $left[y] = x$
6 **do** $x \leftarrow y$
7 $y \leftarrow p[y]$ 8 **return** y

Správnost a složitost jako u Successor.

Vložení

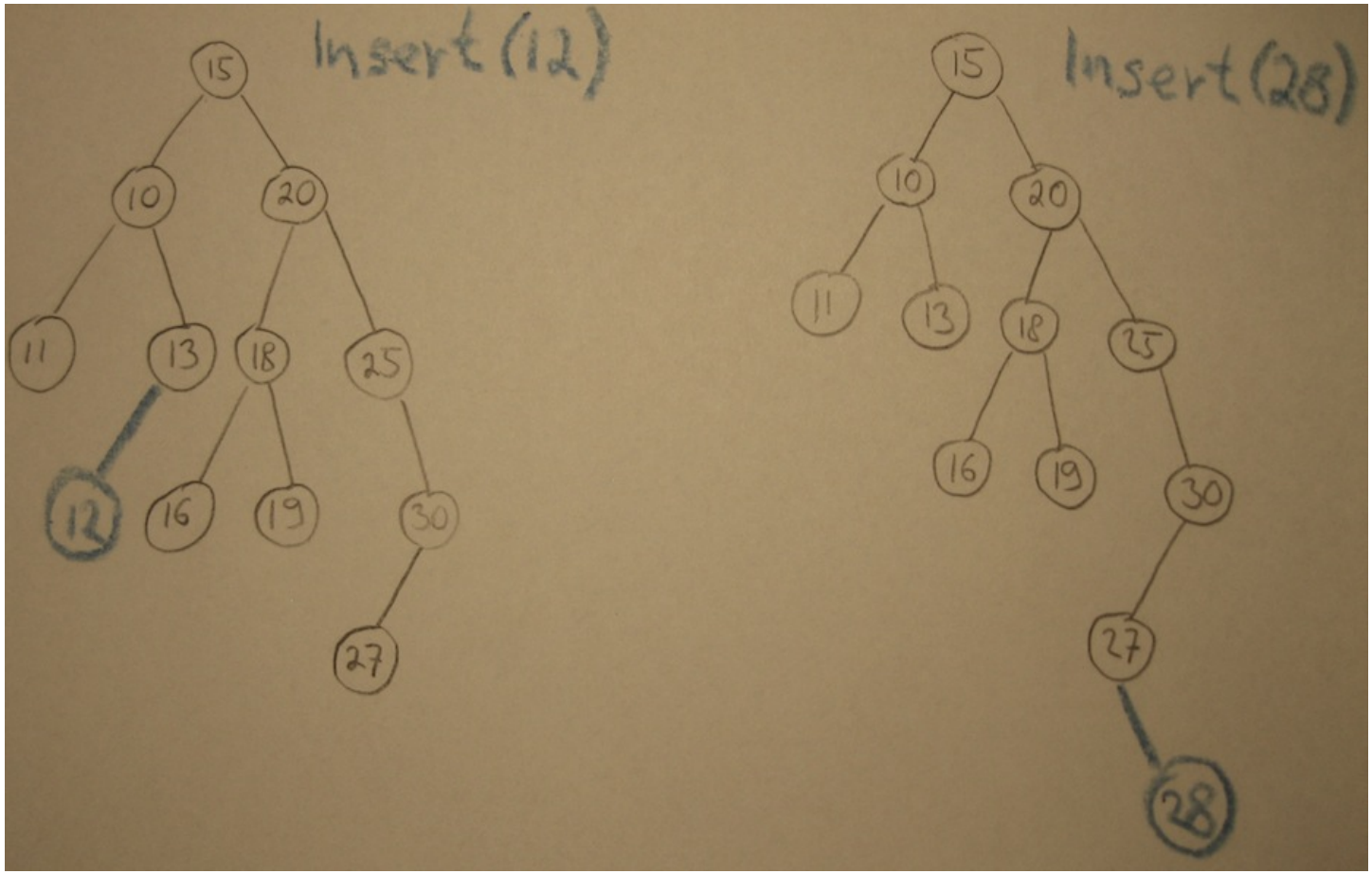
Vstup: (Ukazatel na) kořen x binárního vyhledávacího stromu a (ukazatel na) uzel nový uzel z, pro který $key[z] = v$, $left[z] = NIL$ a $right[z] = NIL$.
Výstup: Vznikne binární vyhledávací strom s kořenem x (změnený původní strom), do něhož byl správně vložen uzel s hodnotou v.

```
Insert(x, z)
1   y ← NIL
2   while x ≠ NIL
3       do y ← x
4           if key[z] < key[x] then x ← left[x] else x ← right[x]
5   p[z] ← y
6   if y = NIL
7     then x ← z
8     else if key[z] < key[y]
9             then left[y] ← z
10            else right[y] ← z
```

Správnost je zřejmá z vlastností binárních vyhledávacích stromů (vysvětlíme na přednášce).

Časová složitost v nejhorším případě $O(h)$ (z x cestujeme k listům).

Příklad Vložení do stromu T.



Vložení prvků 12 a 28.

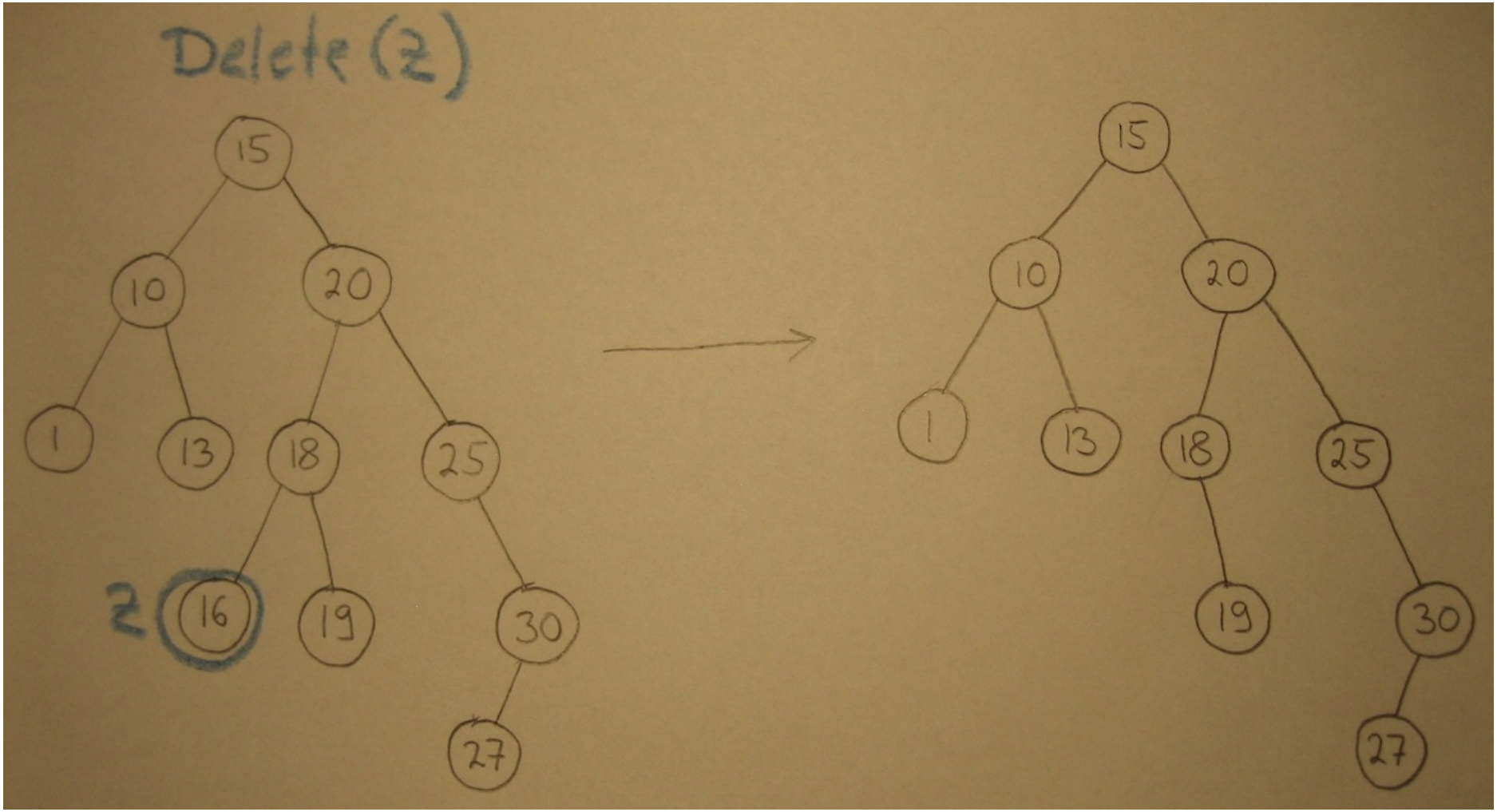
Odstranění

Vstup: (Ukazatel na) kořen x binárního vyhledávacího stromu a (ukazatel na) uzel z tohoto stromu.
Výstup: Vznikne binární vyhledávací strom (změnený původní strom), ze kterého byl odstraněn uzel z.

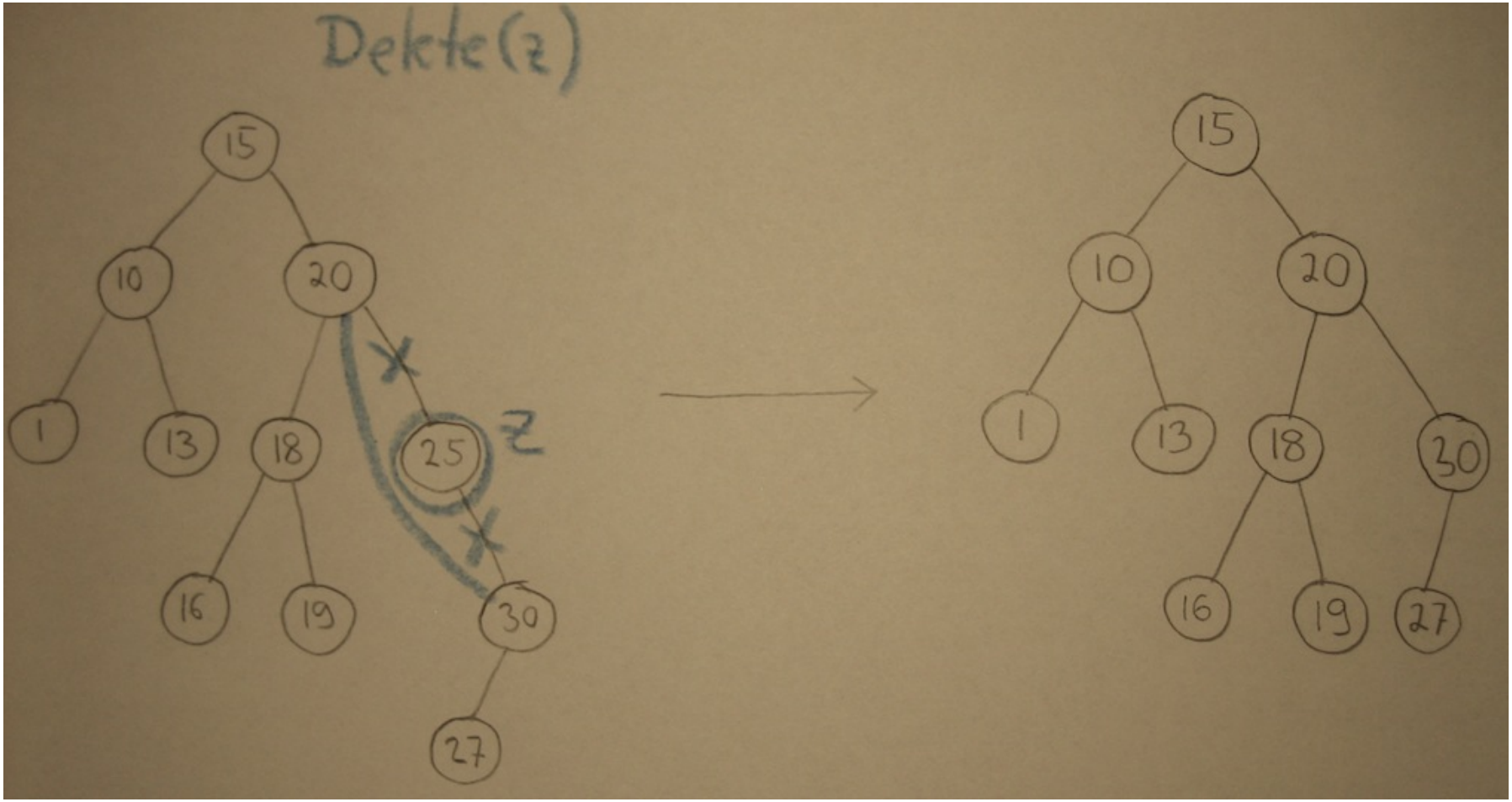
Delete(x, z)
1 **if** $left[z] = NIL$ or $right[z] = NIL$ **then** $y \leftarrow z$ **else** $y \leftarrow$Successor(z)
2 **if** $left[y] \neq NIL$ **then** $w \leftarrow left[y]$ **else** $w \leftarrow right[y]$
3 **if** $w \neq NIL$ **then** $p[w] \leftarrow p[y]$
4 **if** $p[y] = NIL$
5 **then** $x \leftarrow w$
6 **else if** $y = left[p[y]]$ **then** $left[p[y]] = w$ **else** $right[p[y]] = w$
7 **if** $y \neq z$
8 **then** $key[z] \leftarrow key[y]$
9 **return** y

Správnost z vlastností binárních vyhledávacích stromů (na přednášce).
Časová složitost v nejhorším případě $O(h)$ (Successor a konstantní počet instrukcí).

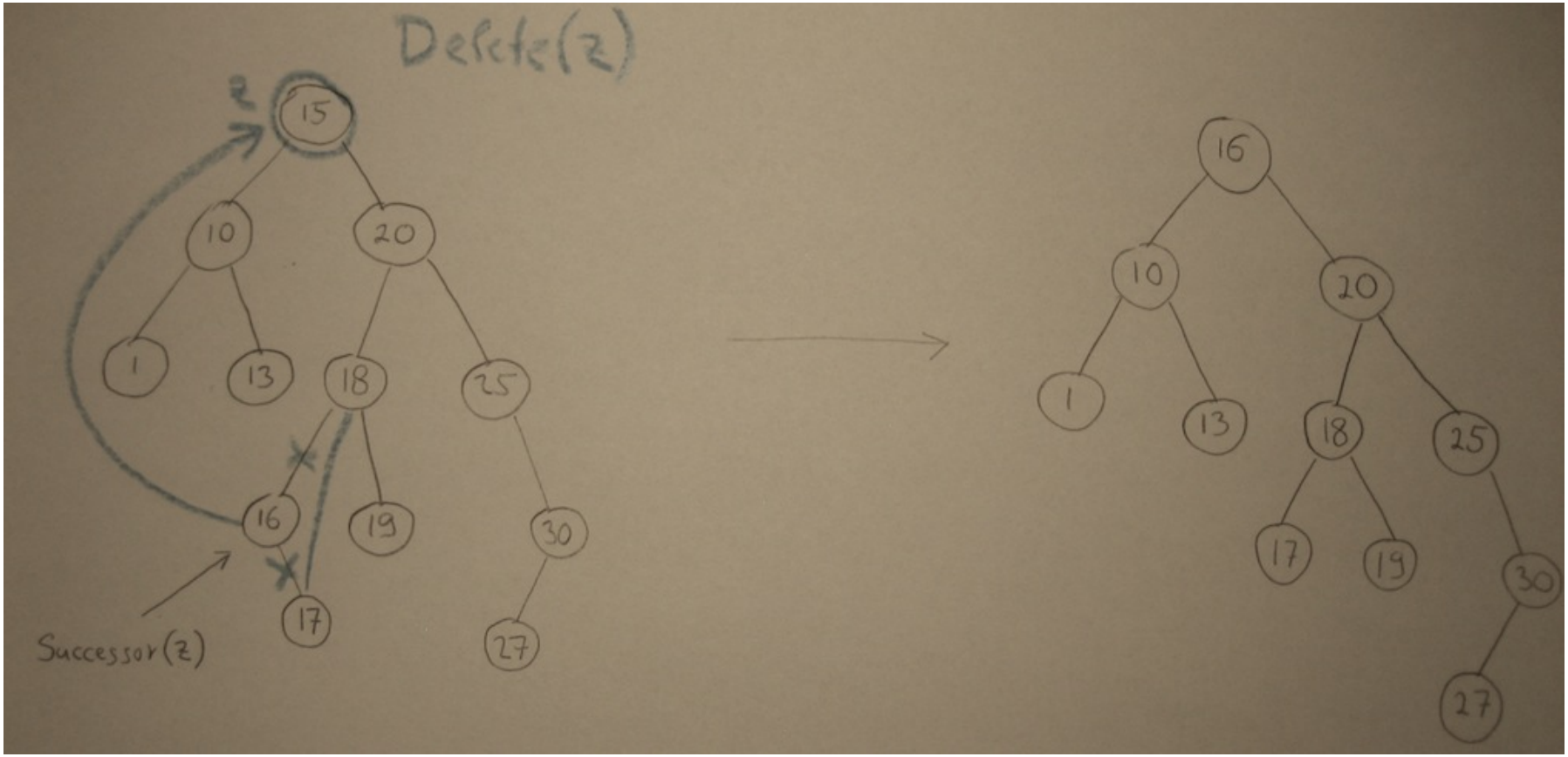
Příklad Odstranění prvku s žádným potomkem (listu).



Příklad Odstranění prvku s jedním potomkem.



Příklad Odstranění prvku se dvěma potomky.



Náhodně vytvářené binární vyhledávací stromy

Viz přednášky.

Červeno-černé stromy (red-black trees)

Informace k červeno-černým stromům jsou na stránkách předmětu (kopie z knihy "Cormen T. et al.: Introduction to Algorithms, MIT Press".)

Další informace (výklad látky a poznámky k textu) byly podány na přednáškách.

AVL stromy

Informace byly podány na přednáškách (materiály jsou dostupné na stránkách předmětu).

Hashování (hašování, hashing)

Úvod k hashování

- Hashování je jednoduchá a účinná metoda ukládání a vyhledávání dat.
- Použitá datová struktura se nazývá hashovací tabulka.
- Hashovací tabulka implementuje datovou strukturu dynamická množina, která umožňuje vkládat, vyhledávat a odstraňovat prvky (tzv. slovník, angl. dictionary). Některé operace tedy podporovány nejsou, ale v mnoha použitích tyto tři operace stačí.
- Např. vyhledávání trvá v nejhorším případě $\Theta(n)$. V průměrném případě je mnohem rychlejší: $O(1)$ za realistických předpokladů.
- Hashovací tabulku T lze nahlížet jako zobecnění pole: Prvek s klíčem *key* se nachází v $T[i]$, kde i je index, který se vypočítá z *key* pomocí tzv. hashovací funkce h, tj. $i = h(key)$. (To je základní myšlenka, je třeba ošetřit drobné komplikace, viz dále.)

Tabulky s přímým adresováním

- Tabulky s přímým adresováním jsou vhodné pro případ, kdy množina U všech hodnot klíče je dostatečně malá. Tabulka má pak $|U|$ položek (každé hodnotě klíče odpovídá jedna položka tabulky).
- Předpokládejme $U = \{0, 1, \ldots, m-1\}$.
- Tabulka T je pak pole s položkami $T[0], T[1], \ldots, T[m-1]$.
- Položkou $T[i]$ tabulky T je ukazatel na záznam s klíčem hodnoty i, který obsahuje uložená data (tj. klíč s hodnotou i, příp. další položky).
- $T[i] = NIL$ znamená, že na místě s indexem i v T není uložen žádný záznam.
- Předpokládáme, že dva záznamy s různými hodnotami mají různé hodnoty kíčů (tj. hodnota klíče jednoznačně určuje hodnoty dalších položek záznamu).
- Jiná možnost: Položkou $T[i]$ není ukazatel, ale přímo záznam. V tom případě je třeba umět rozpoznat, že v $T[i]$ není uožen záznam (např. dohodou, že v takovém případě je $key[T[i]]$ hodnota, která se v U nevyskytuje).

Vyhledání:

Search(T, k)
1 **return** $T[k]$

Vrátí ukazatel na záznam s hodnotou klíče k, popř. NIL, pokud v tabulce T takový záznam není.

Vložení:

Insert(T, x)
1 $T[key[x]] \leftarrow x$

x je (ukazatel na) vkládaný záznam. Ten se vloží na pozici $key[x]$ pole T.

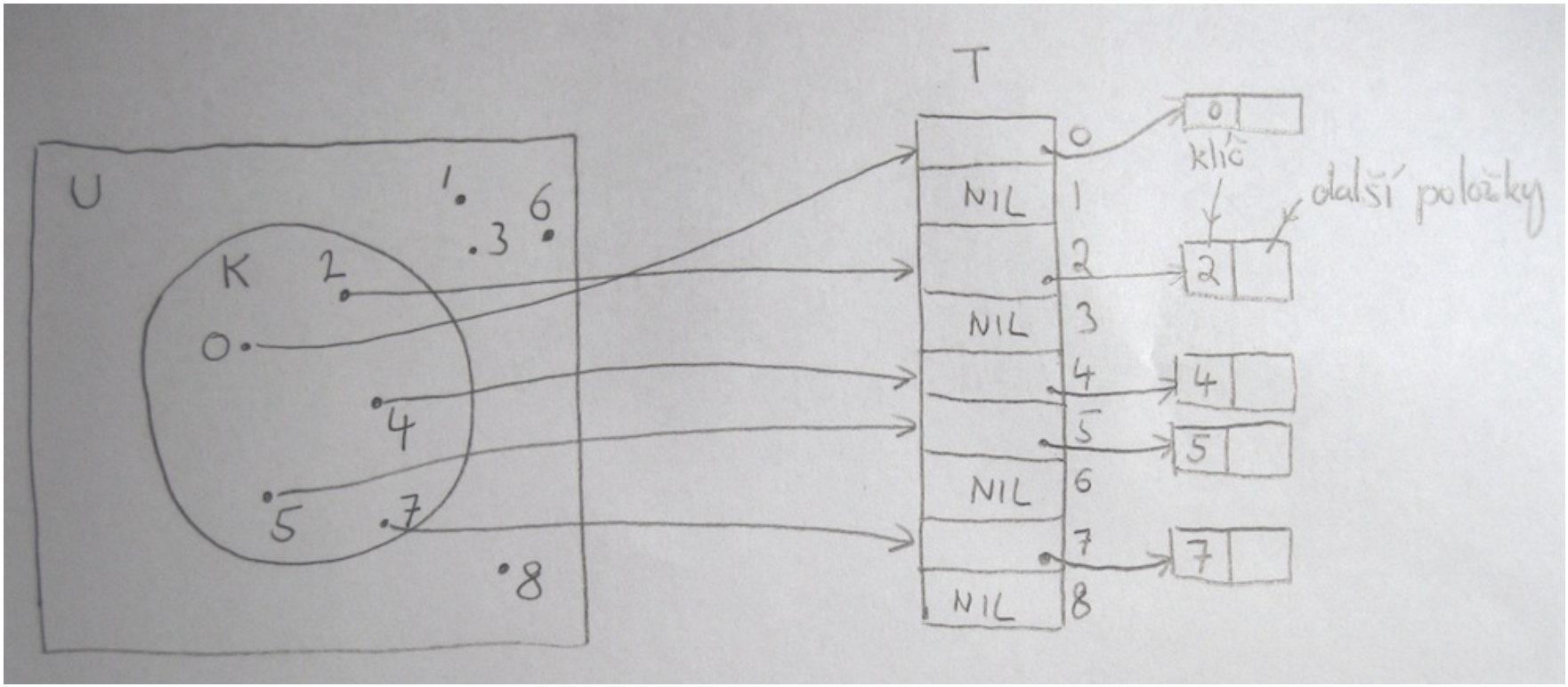
Odstranění:

Delete(T, x)
1 $T[key[x]] \leftarrow NIL$

x je (ukazatel na) vkládaný záznam. Ten se provedenou operací z tabulky odstraní.

Složitost operací Search, Insert a Delete v nejhorším případě je $\Theta(1)$ (konstantní).

Příklad tabulky s přímým adresováním.



Hashovací tabulky

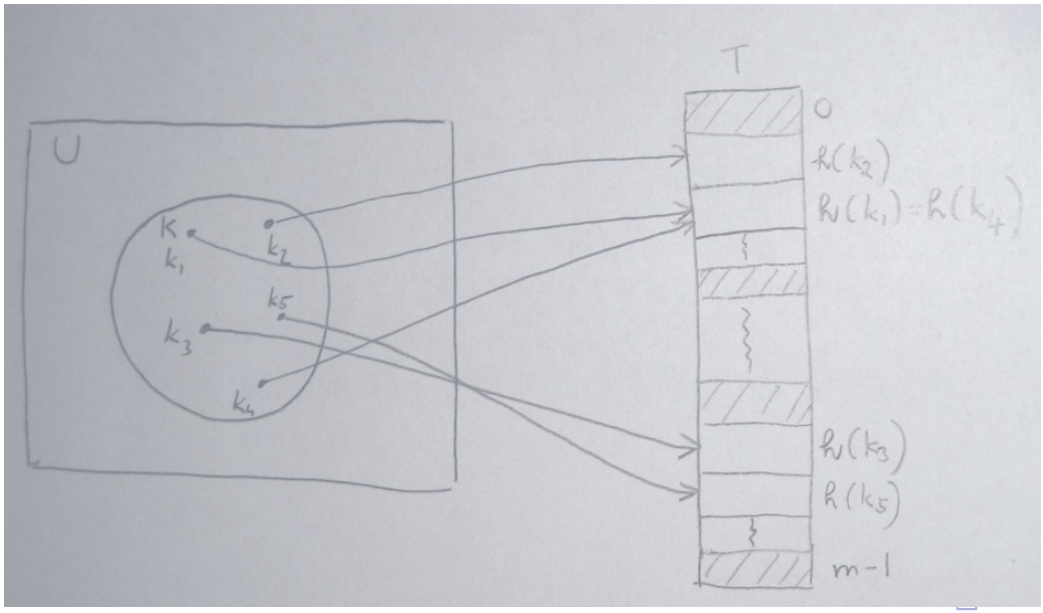
- Nevýhoda tabulek s přímým adresováním: Je-li množina U hodnot klíče velká, je tabulka T velká. Přitom množina K hodnot klíče, které se v uložených datech skutečně vyskytují, může být mnohem menší.
- Dochází tedy k plýtvání pamětí, je třeba $\Theta(|U|)$ paměťových míst. Hashovací tabulka potřebuje jen $\Theta(|K|)$ paměťových míst.
- Přitom časová složitost operací nad hashovací tabulkou je $\Theta(1)$ v průměrném případě (u tabulky s přímým adresováním je stejná, ale v nejhorším případě).
- **Hashovací tabulka** je pole T s položkami $T[0], T[1], \ldots, T[m-1]$.
- Záznam s klíčem k je uložen v položce s indexem $h(k)$, kde h je tzv. **hashovací funkce**. $h(k)$ se nazývá (hashovaná) hodnota klíče k. h je tedy funkce

$$h : U \to \{0, 1, \ldots, m-1\}.$$

  (U tabulky s přímým adresováním je tedy $h(k) = k$.)
- h jedy zobrazuje množinu s $|U|$ hodnotami klíče na množinu s m hodnotami indexů (U může obecně obsahovat i jiné hodnoty než čísla).

- Problém, který je třeba vyřešit: Dvě různé hodnoty klíče jsou zobrazeny na stejnou hashovanou hodnotu, tj. $h(k_1) = h(k_2)$ pro $k_1 \neq k_2$.
- Takový problém se nazývá **kolize**. Kolize lze efektivně řešit (viz dále, metoda řetězení a otevřeného adresování).
- Protože $|U| > m$, nelze kolizím zcela zabránit. Počet kolizí se však snažíme omezit volbou vhodné hasovací funkce (viz dále, jde o rovnoměrné přidělení hodnot indexů hodnotám klíče).

Schéma hashovací funkce.



Hashovací funkce

Základní otázky:

- Co je to dobrá hashovací funkce?
- Jak navrhovat hashovací funkce?

Dobrá hashovací funkce by měla splňovat předpoklad **jednoduchého rovnoměrného hashování** (SUH, simple uniform hashing):
Pravděpodobnost p_i, že náhodně vybrané hodnotě u klíče bude funkcí h přiřazen index i ($h(u) = i$), je stejná pro všechna i (tj. $p_i = p_j$ pro každé $i, j \in \{0, \ldots, m-1\}$). (Tj. h rozdělí hodnoty klíče mezi indexy tabulky rovnoměrně.)

V praxi zpravidla neznáme pravděpodobnosti výskytu hodnot u klíče (nemusí být stejné).
Někdy to ale znát můžeme. Např. když víme, že klíče jsou náhodná racionální čísla k z intervalu $[0, 1)$, pak funkce

$$h(k) = \lfloor km \rfloor,$$

kde m je velikost hashovací tabulky, splňuje SUH (zdůvodněte).

Hashovací funkce jsou často založeny na **heuristikách**, jejichž cílem je přiblížit se SUH. Např. pokud se některé hodnoty klíče vyskutují častěji, je třeba s tím při návrhu počítat.

Některé aplikace vyžadují **silnější předpoklady** než SUH. Např. na množině U (hodnoty klíče) může být definována vzdálenost (blízkost, podobnost) a můžeme chtít, aby blízké hodnoty klíče byly zobrazeny na vzdálené hodnoty indexů.

Metody hashování obvykle předpokládají, že **hodnoty klíče jsou čísla**. Pokud tomu tak není, je třeba hodnoty klíče převádět na čísla.
Příklad: Pokud jsou hodnoty klíče řetězce sestávající z 3 ASCII znaků (ARG, BEL, CZE, POL, USA, ...), může být převod řetězce XYZ na čísla zajištěn rovnicí:
$u(XYZ) = 128^2 * c(X) + 128 * c(Y) + c(Z)$,
kde $c(X)$ je ASCII kód znaku X. Tedy např.
$u(CZE) = 128^2 * c(C) + 128 * c(Z) + c(E) = 128^2 * 67 + 128 * 90 + 69 = 1109317$.

Hashovací funkce metodou zbytku po dělení

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

m je velikost hashovací tabulky. Např. pro $m = 75$ a $k = 1328$ je $h(k) = 53$.

Při použití této metody je třeba zvážit hodnotu m. m by např. neměla být mocnina 2. Je-li totiž $m = 2^p$ pro nějaké kladné celé p, je $k \mod m$ číslo, jehož zápis v dvojkové soustavě je tvořen posledními p číslicemi zápisu čísla k v dvojkové soustavě (zdůvodněte). Funkce h by pak pracovala jen s částí hodnoty k klíče (v krajním případě by tato část mohla být stejná pro všechny hodnoty klíče, které se vyskytnou).

Vhodné je zvolit jako m **prvočíslo, které není blízké mocnině 2**.
Příklad: Navrhujeme hashovací tabulku, do které budeme ukládat cca 3000 hodnot, tj. $n = 3000$, požadujeme faktor α zhruba 4 (viz dále, je pro nás přijatené prohledávat seznam s 4 prvky). Je $3000/4 = 768$. Nejbližší mocniny čísla 2 jsou 512 a 1024. Zvolíme tedy $m = 769$, protože je to prvočíslo, které není blízké mocnině 2 a je blízké číslu $3000/4$.

Hashovací funkce metodou násobení

$$h(k) = \lfloor m \cdot (kA \mod 1) \rfloor$$

m je velikost hashovací tabulky, A je zvolená konstanta, $0 < A < 1$.
Poznamenejme, že $(kA \mod 1)$ je desetinná část čísla kA, tj. $(kA \mod 1) = kA - \lfloor kA \rfloor$.

Výhodou oproti metodě dělení je fakt, že volba hodnoty m není tak kritická.
Zpravidla se volí $m = 2^p$ pro nějaké kladné celé p. Důvod: funkci h lze snadno implementovat.

Podrobněji: Předpokládejme, že čísla jsou reprezentována w-bitovými slovy a že těmito slovy lze reprezentovat všechny hodnoty k klíče. Zvolíme $A = s/2^w$, pro nějaké s, $0 < s < 2^w$.
Zápis čísla $h(k)$ v dvojkové soustavě má p bitů. Ty získáme následovně: k vynásobíme číslem s; výsledkem je $2w$-bitové číslo, které lze zapsat ve tvaru $r_1 2^w + r_0$ (tj. r_0 a r_1 jsou čísla, jejichž binární zápis tvoří dolních a horních w bitů čísla ks); horních p bitů čísla r_0 pak tvoří binární zápis čísla $h(k)$.

Tato metoda pracuje správně pro jakoukoli volbu čísla A (resp. čísla s, jedno číslo určuje jednoznačně druhé).
Cvičení: Ověřte (zkuste na příkladě, pak dokažte).

Některé hodnoty A jsou výhodnější.
Knuth D.: The Art of Computer Programming, Vol. 3, doporučuje volit A blízké číslu

$$(\sqrt{5}-1)/2 \approx 0.61803399.$$

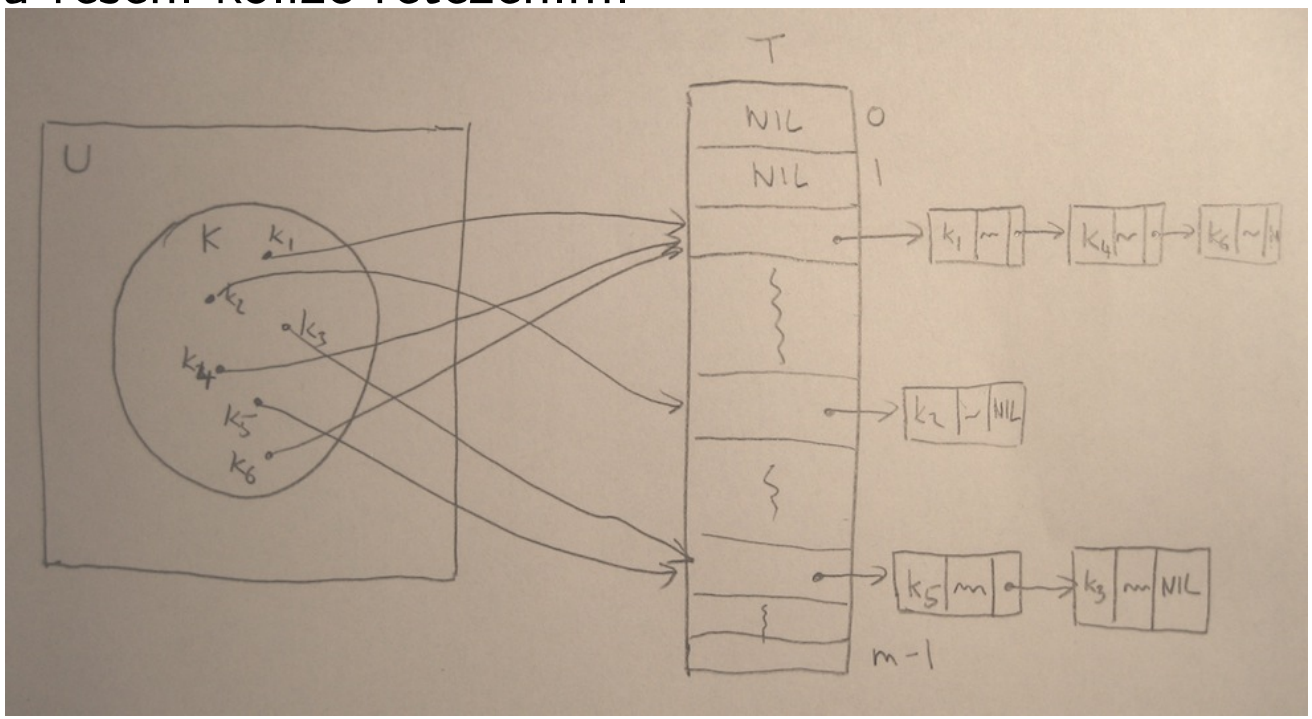
Příklad (Cormen T. et al.: Introduction to Algorithms):
$k = 123456$, $p = 14$, $m = 2^{14}$, $w = 32$.
Zvolili bychom A ve tvaru $s/2^{32}$, které je nejblíže hodnotě $(\sqrt{5}-1)/2$. To je pro $s = 2654435769$.
Pak je $ks = 327706022297664 = 76300 \cdot 2^{32} + 17612864$, tedy $r_1 = 76300$ a $r_0 = 17612864$. Prvních 14 bitů binárního zápisu čísla r_0 je binárním zápisem čísla 67, tj. $h(k) = 67$.

Univerzální hashování

Viz materiál z knihy Cormen T. et al.: Introduction to Algorithms, MIT Press (na stránkách předmětu).

Řešení kolizí metodou řetězení

- Všechny záznamy s hodnotami klíče, které se zobrazí na stejnou hodnotu indexu (řekněme hodnotu i), se ukládají do spojového seznamu. $T[i]$ obsahuje ukazatel na první prvek seznamu, popř. NIL, pokud je seznam prázdný).
- Schéma řešení kolize řetězením.



– Operace vyhledání, vložení a odstranění se snadno implementují (T je hašovací tabulka, x je (ukazatel na) záznam, x je hodnota klíče):

Chained-Hash-Search(T, k)
1 **return** Linked-List-Search($T[h(k)], k$)

Chained-Hash-Insert(T, x)
1 Linked-List-Insert($T[h(key[x])], x$)

Chained-Hash-Delete(T, x)
1 Linked-List-Delete($T[h(key[x])], x$)

Analýza hashování s metodou řetězení

- Vkládání: Vkládá-li se na začátek seznamu a je-li zajištěno, že vkládaný záznam x se v seznamu nevyskytuje, je počet instrukcí $\Theta(1)$ (zjištění, zda se x v seznamu vyskytuje, trvá $O(n)$ instrukcí, kde n je délka seznamu $T[h(key[x])]$).
- Odstranění: Pokud je použit obousměrný seznam, je počet instrukcí $\Theta(1)$ (argumentem operace je ukazatel na odstraňovaný záznam). (Je-li použit jednosměrný seznam, je třeba najít předchůdce x, k čemuž je třeba $O(n)$ instrukcí).
- Vyhledání: Počet instrukcí je $\Theta(n)$, kde n je délka seznamu $T[h(k)]$. Ale: v průměrném případě je za realistických předpokladů opět $\Theta(1)$.

Podrobněji ke složitosti vyhledání v průměrném případě v hashovací tabulce se zřetězením.

Mějme tabulku $T[0..m-1]$, ve které je uloženo n záznamů. Označme
$\alpha = \frac{n}{m}$ (faktor zaplnění/zatížení tabulky T, load factor).

Předpokládejme, že h splňuje předpoklad jednoduchého rovnoměrného hashování (SUH, simple uniform hashing): Pravděpodobnost p_i, že náhodně vybrané hodnotě u klíče bude funkcí h přiřazen index i ($h(u) = i$), je stejná pro všechna i (tj. $p_i = p_j$ pro každé $i, j \in \{0, \ldots, m-1\}$). (Tj. h rozdělí hodnoty klíče mezi indexy tabulky rovnoměrně.)

Předpokládejme, že časová složitost výpočtu $h(k)$ je $\Theta(1)$. Pak platí:

Věta Časová složitost vyhledání prvku v průměrném případě je $\Theta(1+\alpha)$.

Důkaz vynecháme. K důkazu je třeba znát teorii pravděpodobnosti.
Neformální argument na přednášce. Je třeba rozlišit případ, kdy se hledaný prvek v tabulce nevyskutuje, a případ, kdy se tam vyskytuje.
Tedy, je-li $n = O(m)$, je $\alpha = n/m = O(1)$, tedy složitost vyhledání v průměrném případě je $O(1)$.

Řešení kolizí metodou otevřeného adresování

- Při tomto řešení jsou všechny záznamy uloženy v hašovací tabulce (nedochází tedy k řetězení záznamů, jejichž klíče hashovací funkce zobrazí na stejnou hodnotu indexu).
- Do tabulky lze tedy umístit maximálně m záznamů (počet položek tabulky). To znamená, že musí být $n \leq m$, tj. $\alpha \leq 1$ (faktor zaplnění).
- Předpokládáme, že položka tabulky, která neobsahuje uložený záznam, obsahuje hodnotu NIL (prakticky to lze implementovat např. tak, že hodnota *key* takové položky je dohodnutá hodnota, která nepatří do množiny U všech hodnot klíče).
- Základní myšlenka řešení kolize: Vyhledáváme-li (nebo ukládáme-li) záznam s hodnotou k klíče, hledáme nejprve v položce $T[i_0]$, kde $i_0 = h(k)$. Pokud je tam záznam s jinou hodnotou klíče, tj. nastala kolize, hledáme v další položce, jejíž index i_1 spočítáme. Takto případně zkoušíme posloupnost indexů $i_0, i_1, i_2, \ldots$
- Metoda tedy nepotřebuje ukazatele. To šetří paměť.

– Prohledávání (angl. probing) tabulky na místech $i_0, i_1, \ldots$ je zajištěno hashovací funkcí

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

tj. $i_p = h(k, p)$. Aby bylo možné nalézt každou volnou položku tabulky T, požadujeme, aby pro každou hodnotu k tvořila prohledávaná posloupnost (probe sequence)

$$\langle h(k, 0), h(k, 1), \ldots, h(k, m-1) \rangle$$

permutaci prvků $0, 1, \ldots, m-1$.

V následujících pseudokódech předpokládáme, že do položky hashovací tabulky T obsahují přímo hodnoty klíče (tj. uložené záznamy obsahují jedinou položku, a to klíč) nebo hodnotu NIL, která indikuje, že v položce není uložen žádný záznam.

Vkládání

```
Open-Address-Hash-Insert(T, k)
1   i ← 0
2   while i < m
3       do j ← h(k, i)
4          if T[j] = NIL
5             then T[j] ← k
6                  return j
7             else i ← i + 1
8   error(hash table overflow)
```

Vyhledávání

```
Open-Address-Hash-Search(T, k)
1   i ← 0
2   j ← h(k, i)
3   while (T[j] ≠ NIL and i < m)
4       do if T[j] = k
5             then return j
6          i ← i + 1
7          j ← h(k, i)
8   return NIL
```

Odstranění

Open-Address-Hash-Delete(T, k)
Je poměrně komplikované. Pokud je třeba odstraňovat prvky, používá se proto spíše metoda řetězení.

Totiž, pokud je odstraňovaná hodnota k na pozici j, nelze pouze provést $T[j] \leftarrow NIL$, protože pozice j může být součástí prohledávané posloupnosti, která byla při vkládání záznamu s nějakým klíčem k' využita. $T[j] \leftarrow NIL$ by způsobila, že by k' při vyhledávání nebyl nalezen.

Možným řešením je použít speciální hodnotu *DELETED*, která by indikovala, že záznam na pozici byl odstraněn. Pak by bylo třeba upravit funkce pro vyhledávání (přes pozice s hodnotou *DELETED* by se prohledávalo dál, to by znamenalo nárůst počtu kroků při vyhledávání) a pro vkládání (záznam by bylo možné vložit na pozici s hodnotou *NIL* nebo *DELETED*).

Cvičení. Implementujte výše popsaný postup.

Hashovací funkce pro otevřené adresování

Jde o funkce $h(k, i)$. Podobně jako u funkcí $h(k)$, dobrá hashovací funkce by měla splňovat předpoklad **rovnoměrného hashování** (UH, uniform hashing), který je zobecněním předpokladu SUH:
Pravděpodobnost p_π, že náhodně vybrané hodnotě k klíče bude funkcí h přiřazena jako prohledávaná posloupnost permutace π prvků $0, 1 \ldots, m-1$, je stejná pro všechny π.

UH je obtížné splnit. Prakticky se používají metody, které UH splňují apoň přibližně.

Ukážeme 3 metody konstrukce hashovací funkce: lineární prohledávání, kvadratické prohledávání, dvojité hashování.

Každá z nich zaručuje, že prohledávaná posloupnost $\langle h(k, 0), h(k, 1), \ldots, h(k, m-1) \rangle$ permutaci prvků $0, 1, \ldots, m-1$.

Žádná však nesplňuje UH, protože žádná nevygeneruje více než m^2 permutací (pro splnění UH by h musela vygenerovat všech $m!$ možných permutací).

Lineární prohledávání (linear probing)

Vyjdeme z hashovací funkce $h' : U \to \{0, \ldots, m-1\}$ (tzv. pomocná hashovací funkce). Hashovací funkce h je definována předpisem

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

Snadno je vidět, že prohledávaná posloupnost je $h'(k), h'(k)+1, \ldots, m-1, 0, \ldots, h'(k)-1$.

Jednoduché na implementaci.

Nevýhoda: Dochází k primárnímu shlukování (primary clustering). Vytváření se dlouhé posloupnosti (po sobě jdoucích) obsazených položek tabulky T, což způsobuje narůst doby vyhledávání. K takovému vytváření dochází proto, že pravděpodobnost, že volná pozice, kterou předchází i zaplněných pozic, se zaplní, je $(i+1)/m$.

Prohledávaná posloupnost indexů je úplně určena svým prvním prvkem. Tedy je využito právě m permutací indexů (a ne $m!$, jak by vyžadovala UH).

Kvadratické prohledávání (quadratic probing)

Vyjdeme z hashovací funkce $h' : U \to \{0, \ldots, m-1\}$ (tzv. pomocná hashovací funkce). Hashovací funkce h je definována předpisem

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

kde c_1 a $c_2 \neq 0$ jsou konstanty. Index první prohledávané položky je $h'(k)$, další jsou dány parametry c_1 a c_2.

Stále jednoduché na implementaci. Nedochází k primárnímu shlukování. Aby došlo k využití všech položek tabulky T, volba c_1, c_2 a m musí splňovat omezení.

Nevýhoda: Dochází k sekundárnímu shlukování (secondary clustering). Pokud dvě hodnoty klíče mají stejný první index prohledávané posloupnosti (tj. $h'(k_1) = h'(k_2)$), mají stejné prohledávané posloupnosti.

Jako u lineárního prohledávání, prohledávaná posloupnost indexů je úplně určena svým prvním prvkem (tedy je využito právě m permutací indexů).

Dvojité hashování (double hashing)

Jedna z nejlepších metod. Přibližně splňuje UH. Vyjdeme ze dvou pomocných hashovacích funkcí, $h_1, h_2 : U \to \{0, \ldots, m-1\}$. Hashovací funkce h je definována předpisem

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

Index první prohledávané položky je $h_1(k)$, další jsou posunuty o $h_2(k)$ modulo m (narozdíl od předchozích dvou metod, posun nyní závisí na k).

Aby prohledávaná posloupnost tvořila permutaci hodnot $0, \ldots, m-1$, musí být hodnoty m a $h_2(k)$ nesoudělné (relatively prime, tj. mají nejvetší společný dělitel 1). Cvičení: Dokažte.

To lze zajistit např. takto: m je mocnina 2, h_2 má jako hodnoty jen lichá čísla;
nebo m je prvočíslo, h_2 má jako hodnoty čísla $< m$.
Např. m je prvočíslo, $h_1(k) = k \mod m$, $h_2(k) = 1 + (k \mod m')$, kde m' je o něco menší než m (např. $m' = m - 1$).
Cvičení. Konstruujte hashovací funkce podle výše uvedených návodů.

Metoda využije $\Theta(m^2)$ prohledávaných posloupností (předchozí jen m).

Analýza metody otevřeného hashování

Předpokládáme faktor zaplnění $\alpha < 1$ a předpokládáme UH. Pak platí.

Věta (1) Počet prohledávaných indexů při vyhledávaní hodnoty, která se v hashovací tabulce nevyskytuje, je v průměrném případě nejvýše $\frac{1}{1-\alpha}$.

(2) Počet prohledávaných indexů při vyhledávaní hodnoty, která se v hashovací tabulce vyskytuje je v průměrném případě nejvýše $\frac{1}{\alpha} \ln \frac{1}{1-\alpha}$, za předpokladu, že pravděpodobnost vyhledávání je pro všechny hodnoty klíče v tabulce stejná.

Důsledek Počet prohledávaných indexů při vložení hodnoty je v průměrném případě nejvýše $\frac{1}{1-\alpha}$.

Ilustrace k Věta (2): Pro $\alpha = 0.5$ (tabulka je zpola plná) je $\frac{1}{\alpha} \ln \frac{1}{1-\alpha} \approx 1.386$, pro $\alpha = 0.95$ (tabulka je z 95% plná) je $\frac{1}{\alpha} \ln \frac{1}{1-\alpha} \approx 3.153$.

Perfektní hashování (perfect hashing)
Hashování, při kterém je složitost vyhledávání v nejhorším případě $O(1)$.

Lze toho dosáhnout, když množina hodnot klíčů je statická: jakmile jsou hodnoty do tabulky vloženy, nikdy se nezmění (např. když klíče jsou klíčová slova programovacího jazyka, názvy souborů na ROM paměti apod.).

Viz materiál z knihy Cormen T. et al.: Introduction to Algorithms, MIT Press (na stránkách předmětu).