1. Úvodní příklady, definice RAM

Příklad: Reportáž

Novinář má za úkol za rok napsat reportáž o pracovních podmínkách v jedné nejmenované firmě. Musí tedy vyzkoušet co nejvíce pracovních pozic. Chce ale, aby se mu neustále zvyšoval plat. Firma v různých časech vypisuje pracovní místa.

Řečeno matematicky, máme zadánu posloupnost $p_1, \ldots, p_n$ reálných čísel a hledáme v ní nejdelší ostře rostoucí vybranou podposloupnost.

Jak můžeme takový problém řešit?

Podle definice: budeme generovat všechny podposloupnosti a testovat, jestli jsou rostoucí. Podposloupnost můžeme popsat charakteristickým vektorem, což je posloupnost nul a jedniček, kde na i-té pozici je 1, právě když podposloupnost obsahuje i-tý člen původní posloupnosti. Charakteristické vektory odpovídají binárním zápisům čísel 1 až 2^n kde n je počet vypsaných prací.

Charakteristické vektory můžeme generovat například tak, že si cifry binárního čísla budeme udržovat v poli a budeme přičítat 1. Po hlubších úvahách (zvídaví hledejte pojem amortizovaná časová složitost) zjistíme, že na jedno přičtení jedničky potřebujeme průměrně konstantně mnoho operací.

Nyní nás bude zajímat kolik řádově provedeme kroků. Všech charakteristických vektorů je 2^n. Pro každý zkontrolujeme, jestli je podposloupnost rostoucí, což zabere n kroků. Celkem tedy provedeme řádově $2^n \cdot n$ kroků.

Rekurzívně: vytvořme funkci, která dostane začátek posloupnosti a najde všechna rozšíření na rostoucí podposloupnost. Zajímá nás ale jen nejdelší podposloupnost, položme tedy $f(i_1, \ldots, i_k) :=$ maximání délka rostoucí podposloupnosti navazující na $x_{i_1}, \ldots, x_{i_k}$.

Probereme všechna j od $i_k + 1$ do n a pro každé j takové, že $x_j > x_{i_k}$, nastavíme maximum $m \leftarrow \max(m, f(i_1, \ldots, i_k, j) + 1)$. Jako výsledek funkce vrátí m. Na začátek posloupnosti přidáme $-\infty$ a zavoláme $f(0)$.

1. Pro $j = i_k + 1$ to n
2. Když $x_j > x_{i_k}$
3. $m \leftarrow \max(m, f(i_1, \ldots, i_k, j) + 1)$
4. Vrať m

Nejhorším případem je rostoucí posloupnost, na které naše funkce vykoná řádově 2^n kroků.

Zamysleme se, jestli potřebujeme prvních $k-1$ parametrů. Pokračování podposloupnosti může ovlivnit poze poslední parametr funkce f. Zjednodušíme tedy volání funkce a místo $f(i_1, \ldots, i_k)$ budeme volat $f(i_k)$.

Rekurze s blbenkou: $f(i)$ bude volána mnohokrát pro stejné i. Nejlépe je to vidět na příkladu rostoucí posloupnosti, kde je $f(i)$ volána po každém zavolání $f(j)$ kde $j < i$.

V poli X si pamatujeme výsledky funkce f pro jednotlivá i, tedy pole X obsahuje na pozici i hodnotu $f(i)$.

Cvičení: ukažte, že algoritmus vykoná řádově n^2 operací a spotřebuje n buněk paměti.

Bez rekurze: všimněme si, že spočítat $f(n)$ je velmi snadné ($f(n) = 0$).

1. $f(n) = 0$
2. $k = n - 1 \ldots 0$
3. $\quad f(k) = 0$
4. $\quad j = k + 1 \ldots n$
5. $\quad\quad$ Když $x_j > x_k$
6. $\quad\quad\quad f(k) = \max(f(k), f(j) + 1)$

Rychlost jsme nezvýšili, dokonce ani paměť jsme neušetřili, ale zbavili jsme se rekurze.

Převést úlohu na grafovou je standardní informatický trik. Vrcholy jsou čísla $V := \{1, \ldots, n\}$, hrana $(i, j) \in E \equiv i < j \ \& \ x_i < x_j$. Cesty v tomto grafu odpovídají vybraným rostoucím posloupnostem a my hledáme nejdelší cestu v acyklickém grafu, což umíme (budeme umět) lineárně s velikostí grafu. Hran může být až $|E| = \binom{n}{2} \approx n^2$. Čímž jsme dostali další kvadratický algoritmus.

Datová struktura: během semestru poznáme šikovnou datovou strukturu, která obsahuje uspořádané dvojice reálných čísel (x, y), kde x je klíč a y hodnota. Po této struktuře budeme chtít aby uměla vložit dvojici $Insert(x, y)$ a dotaz $Query$: $Query(t) := \max\{y \mid \exists x \geq t : (x, y)$ je ve struktuře$\}$.

Postupujeme podobně jako v algoritmu *Bez rekurze* s tím rozdílem, že kroky 4 až 6 za nás udělá datová struktura. Pro každé k zavoláme $Insert(x_j, f(j))$ a $Query(x_{k+1})$. Obě trvají řádově $\log n$, struktura nám vrátí největší hodnotu $f(j)$ pro dané x_{k+1}.

Provedeme tedy řádově $n \cdot \log n$ kroků, což je nejlepší známé řešení.

Algoritmus

Na příštích přednáškách budeme studovat algoritmy a jejich vlastnosti. Co ale algoritmus doopravdy je? Jak ho definovat? Žádná pořádná definice algoritmu neexistuje. Pro nás bude algoritmus program v nějakém jazyce na nějakém výpočetním stroji (viz definice RAM).

Churchova teze: všechny definice algoritmů jsou ekvivalentní. Toto není opravdová věta, spíš vyjadřuje, že všechny rozumné definice algoritmu definují v podstatě to samé.

Model RAM

V předchozí části jsme mluvili o výpočetním modelu, pojďme tedy nějaký nadefinovat. Výpočetních modelů je více, my vybereme jeden poměrně blízký skutečným počítačům.

Definice: Random Access Machine (RAM)

RAM počítá jen s celými čísly (dále jen *čísla*). Znaky, stringy a podobně reprezentujeme čísly, jejich posloupnostmi atd. Paměť je tvořena buňkami, které obsahují čísla. Paměťové buňky jsou adresované taktéž čísly. Program samotný je konečná posloupnost instrukcí (také opatřených adresami) následujících druhů:

(kde X, Y jsou nějaké operandy)

- Datové přesuny X `<-` Y
- Aritmetické, logické a bitové: X `<-` $Y \oplus Z$

  $\oplus \in \{+, -, *, \texttt{div}, \texttt{mod}, \&, |, <<, >>\}$ kde &, | znamenají logické and a or, <<, >> znamenají bitový posun vlevo a vpravo.
- Řídící: skok `goto` Z, podmíněný skok `Když` X `<` Y `goto` Z, zastavení programu `halt`.

Operandy:

- Konstanty $(1, 2, \ldots)$
- Adresované přímo – `[`*konst.*`]` – budeme používat písmena `A-Z` jako aliasy pro buňky paměti -1 až -26 (tedy A=`[-1]`), které nazýváme *registry* a budou nám sloužit jako proměnné (samozřejmně nejen ony).
- Adresované nepřímo – `[[`*konst.*`]]`

  Můžeme se chtít podívat na adresu, kterou máme uloženou v nějaké buňce, podobně jako pointery v C.

Samotný výpočet probíhá takto:

1. Do smluvených buněk umístíme vstup, obsah zbylých paměťových buněk není definován.
2. Provádíme program po instrukcích, dokud nedojdeme k `haltu` nebo konci programu.
3. Pokud se program nezacyklil, tedy pokud skončil, ze smluvených buněk přečteme výstup.

Míry složitosti

1. *RAM s jednotkovou cenou*: čas = # instrukcí při daném vstupu, prostor = # buněk do kterých algoritmus aspoň jednou zapsal během výpočtu.

   Toto není moc dobrý nápad, protože není nijak penalizována například práce s velmi dlouhými čísly – pořád je to jedna instrukce, takže cena je stejná, ale počítače se tak přece nechovají. Velikost čísel ale konstantou (třeba 32 bitů) omezit nesmíme, protože bychom omezili paměť (čísly ji adresujeme) a co hůř i možnou velikost vstupu.
2. *RAM s logaritmickou cenou*: cena instrukce = # bitů zpracovávaných čísel, prostor = # bitů všech použitých buněk. To je teoreticky přesné, ale dost nepraktické (ve všech složitostech by byly spousty logaritmů).
3. *RAM s omezenými čísly*: jednotková cena instrukcí, ale čísla omezíme nějakým polynomem $P(n)$, kde n je velikost vstupu. Tím zmizí

paradoxy prvního modelu, ale můžeme adresovat jen polynomiální prostor (to nám ovšem obvykle nevadí).

Nadále budeme předpokládat třetí zmíněný model.

Definice:

- *Čas běhu algoritmu* $t(x)$ pro vstup x měříme jako sumu časů instrukcí, které program provedl při zpracování vstupu x. Pokud se pro daný vstup program nezastaví berme $t(x) = +\infty$.
- *Prostor běhu algoritmu* $s(x)$ je analogicky počet paměťových buněk použitých při výpočtu se vstupem x.

Chceme zavést míru časové a prostorové náročnosti programů zvanou složitost. Složitost je maximum délky běhu přes všechny vstupy určité délky.

- *Množina možných vstupů* X
- *Délka vstupu* je funkce $l : X \to \mathbb{N}$
- *Časová složitost* (v nejhorším případě) je:

$$T(n) := \max\{t(x) \mid x \text{ je vstup délky } n\}.$$

- *Prostorová složitost* (v nejhorším případě) je:

$$S(n) := \max\{s(x) \mid x \text{ je vstup délky } n\}.$$

Podobně můžeme zavést i složitost v nejlepším a průměrném případě, ale ty budeme používat jen zřídka.

2. Složitost, grafové algoritmy

(zapsal Martin Koutecký)

Model RAM

Při analýze algoritmu bychom chtěli nějak popsat jeho složitost. Abychom mohli udělat toto, potřebujeme nejprve definovat výpočetní model. Výpočetních modelů je více, my vybereme jeden poměrně blízký skutečným počítačům:

Definice: Random Access Machine (RAM)

RAM počítá jen s celými čísly – znaky, stringy a podobně reprezentujeme čísly, jejich posloupnostmi atd. Paměť je tvořena buňkami, které obsahují čísla. Paměťové buňky jsou adresované taktéž čísly. A program samotný je konečná posloupnost instrukcí následujících druhů:

- Aritmetické a logické: X `<-` $Y \oplus Z, \oplus \in \{+, -, *, \texttt{div}, \texttt{mod}, \&, |, <<, >>\}$
- Řídící: `goto` *label*, `halt`
- Podmínky: pro libovolnou nepodmíněnou instrukci můžu použít if X `<` Y `==>` instrukce

Poznámka (operandy):

- Konstanty (1, 2, ...)
- Adresované přímo – M[konst.] – budeme používat písmena `A-Z` jako aliasy pro buňky paměti -1 až -26, které nazýváme registry. (tedy A=`M[-1]`)
- Adresované nepřímo – `M[M[konst.]]` – budeme používat zkratku [[konst.]]

Samotný výpočet probíhá takto:

1. Do smluvených buněk umístíme vstup, obsah zbylých paměťových buněk není definován.
2. Provádíme program postupně po instrukcích, dokud nedojdeme k haltu nebo konci programu.
3. Pokud se program nezacyklil, tedy pokud skončil, ze smluvených buněk přečteme výstup.

Složitost

Jak dobře popsat složitost?

1. RAM *s jednotkovou cenou*: čas $\approx$ #instrukcí, prostor $\approx$ maximální číslo buňky minus minimální číslo buňky použité při výpočtu. Toto není moc dobrý nápad, protože není nijak penalizována například práce s velmi dlouhými čísly – pořád je to jedna instrukce, takže cena je stejná, ale počítače se tak přece nechovají. Velikost čísel ale omezit nesmíme, protože bychom omezili paměť (čísly ji adresujeme).
2. RAM *s logaritmickou cenou*: cena instrukce $\approx$ #bitů zpracovávaných čísel, prostor $\approx$ # bitů všech použitých buněk. To je teoreticky přesné, ale dost nepraktické (ve všech složitostech by byly logaritmy).
3. RAM *s omezenými čísly*: jednotková cena instrukcí, ale čísla omezíme nějakým polynomem $P(n)$. Tím zmizí paradoxy prvního modelu, ale můžeme adresovat jen polynomiální prostor (to nám ovšem obvykle nevadí).

Nadále tedy budeme předpokládat třetí zmíněný model.

Definice:

- *Čas běhu algoritmu* $t(x)$ pro vstup x měříme jako sumu časů provedených operací, které program provedl při zpracování vstupu x.
- *Prostor běhu algoritmu* $s(x)$ je analogicky počet paměťových buněk spotřebovaných při výpočtu se vstupem x.
- *Časová složitost* (v nejhorším případě) je:

$$T(n) := \max\{t(x); x \text{ je vstup délky } n\}.$$

- *Prostorová složitost* (v nejhorším případě) je:

$$S(n) := \max\{s(x); x \text{ je vstup délky } n\}.$$

Nyní zkusíme zanalyzovat nějaký konkrétní algoritmus. Vezměme například řazení pomocí přímeho výběru (selection sort). Na vstupu dostaneme počet čísel n (v registru `N`), v buňkách $1, \ldots, n$ je nesetříděná posloupnost čísel. Ta pak třídíme následujícím algoritmem zapsaným v pseudokódu:

1. Pro $i = 1$ do n:
2. $\quad j \leftarrow i$
3. $\quad$ Pro $k = i$ do n:
4. $\quad\quad$ Je-li $[k] < [j] \Rightarrow j \leftarrow k$
5. $\quad [i]$ prohodíme s $[j]$.

Jak by takový algoritmus vypadal zapsaný v instrukcích RAM? Budeme muset použít návěští a goto místo cyklů, jména registrů místo proměnných a třeba prohození musíme provést přes třetí proměnnou. Nějak takto:

```
        I <- 1
LOOP:   J <- I
        M <- I
MIN:    IF [J]>=[M] ==> GOTO NEXT
        M <- J
NEXT:   J <- J+1
        IF J<=N ==> GOTO MIN
        X <- [I]
        [I] <- [M]
        [M] <- X
        I <- I+1
        IF I<=N ==> GOTO LOOP
```

Pojďme se podívat, jaká je časová složitost jednotlivých částí algoritmu. Cyklus `MIN` provede za průchod 3 nebo 4 instrukce, ale zajímá nás nejhorší případ, takže 4. Zavolá se $(N - I + 1)$-krát, tedy celkem provede $4 \cdot (N - I + 1)$ instrukcí.

Mimo cyklu `MIN` je v `LOOP` ještě 7 instrukcí, tedy celý `LOOP` provede $4 \cdot (N - I + 1) + 7 = 4(N - 1) + 11$ instrukcí.

Celkově se dostáváme k součtu

$$1 + (\sum_{I=1}^{N} 4(N - I) + 11) = 1 + 11N + 4 \cdot \frac{N(N-1)}{2} = 2N^2 + 9N + 1.$$

Na multiplikativních konstantách ale nezáleží – Na reálných strojích se ceny jednotlivých (pro nás jednotkových) instrukcí stejně liší, takže nemá cenu se multiplikativními konstantami zabývat (alespoň při prvním přiblížení k problému).

$$2N^2 + 9N + 1 \approx N^2 + N$$

Navíc asymptoticky pomalejší funkce nakonec pro velké N vždy prohraje. Tím pádem nezáleží ani na členech nižších řádů:

$$N^2 + N \approx N^2$$

Když už toto víme, můžeme zanedbávat konstanty průběžně: N cyklů po $\approx N$ krocích $\Rightarrow \approx N^2$ kroků. To nás vede k zavedení tzv. *asymptotické notace:*

Asymptotická notace

Definice: Pro funkce $f, g : \mathbb{N} \to \mathbb{R}^+$ řekneme, že f je $\mathcal{O}(g)$ právě tehdy, když $\exists c > 0 : \forall^* n \in \mathbb{N} : f(n) \leq c \cdot g(n)$. Zde $\forall^* n \in \mathbb{N}$ je zkratka za „$\exists n_0 \in \mathbb{N} : \forall n \geq n_0$“, tedy „pro všechna n až na konečně mnoho výjimek.“
Poznámka: $\mathcal{O}$-notace tedy vyjadřuje, že funkce f je skoro všude menší nebo nejvýše rovná nějakému reálnému násobku funkce g. Ačkoliv zápis vypadá jako rovnost, rozhodně není symetrický: například platí $\log n = \mathcal{O}(n)$, ale neplatí $n = \mathcal{O}(\log n)$. Formálně by bylo lepší považovat $\mathcal{O}(g)$ za třídu funkcí, pro které platí, že se dají shora odhadnout kladným násobkem funkce g, a psát tedy $f \in \mathcal{O}(g)$, ale zvyk je bohužel železná košile.
Příklady: $2{,}5n^2 = \mathcal{O}(n^2)$, $2{,}5n^2 + 30n = \mathcal{O}(n^2)$.
Také platí:

$$\mathcal{O}(f) + \mathcal{O}(g) \in \mathcal{O}(f + g),$$

čímž myslíme, že pokud vezmeme libovolnou $f' = O(f)$ a $g' = O(g)$, bude $f' + g' = O(f + g)$. To platí, jelikož skoro všude je $f' \leq cf$, $g' \leq dg$, a tedy $f' + g' \leq cf + dg \leq (c + d)(f + g)$.
Cvičení: Ukažte, že:

- $\mathcal{O}(f) \cdot \mathcal{O}(g) = \mathcal{O}(f \cdot g)$,
- $\mathcal{O}(f + g) = \mathcal{O}(\max(f, g))$,
- $\mathcal{O}(n^2) + \mathcal{O}(n) = \mathcal{O}(n^2 + n) = \mathcal{O}(n^2)$.

$\mathcal{O}$-notace popisuje horní odhad asymptotického chování funkce. Mnohdy však potřebujeme také odhadnout funkci zespodu (chceme-li říci, že algoritmus potřebuje *alespoň* nějaké množství času nebo paměti), případně z obou stran:
Definice:

- $f = \Omega(g) \equiv \exists c > 0 : \forall^* n \in \mathbb{N} : f(n) \geq c \cdot g(n)$.
  Ω-notace tedy říká, že hodnota funkce f je vždy stejná nebo vyšší než nějaký c-násobek funkce g, a tedy $g = \mathcal{O}(f)$.
- $f = \Theta(g) \equiv f = \mathcal{O}(g) \wedge f = \Omega(g)$
  nebo výřečněji:
  $f = \Theta(g) \equiv \exists\ c_1, c_2 > 0 : \forall^* n \in \mathbb{N} : c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n)$ To znamená, že existují nezáporné reálné konstanty c_1, c_2 takové, že se funkce $f(n)$ dá ohraničit c_1- a c_2-násobkem funkce $g(n)$.

Poznámka: Ona rovnost je dost zavádějící. Není to totiž rovnost, není to symetrické. Jiný, možná rozumnější pohled, je zavedení $\mathcal{O}(g)$ jako množiny. Pak by se dalo psát $f \in \mathcal{O}(g)$, nebo třeba $\mathcal{O}(n) \subseteq \mathcal{O}(n^2)$.

Porovnání růstu funkcí: (aneb jak moc máme algoritmy rádi podle jejich chování od nejlepších k nejhorším)

- $\Theta(1)\ldots$ funkce zespoda i shora ohraničené konstantami
- $\Theta(\log(\log n))$
- $\Theta(\log n)$ … logaritmická
- $\Theta(n^\varepsilon), \varepsilon \in (0,1)$ … sublineární
- $\Theta(n)$ … lineární
- $\Theta(n^2)$ … kvadratická
- $\Theta(n^k)$ … polynomiální
- $\Theta(2^n)$ … exponenciální při základu 2
- $\Theta(3^n)$ … exponenciální při základu 3
- $\Theta(k^n)$ … exponenciální při základu $k > 1$
- $\Theta(n!)$ … faktoriálová
- $\Theta(n^n)$
- … nekonečně mnoho dalších tříd (i mezi těmi výše uvedenými)

Poznámka: Pokud se v odhadu složitosti vyskytne logaritmus (jinde než v exponentu), nezáleží na tom, jaký má základ, protože platí:

$$\log_k n = \frac{\log_c n}{\log_c k} = \frac{1}{\log_c k} \cdot \log_c n,$$

kde $1/\log_c k$ je jen konstanta, takže ji můžeme „schovat do $\mathcal{O}$“.
Příklad: Select sort (rozebraný výše): Když jej pustíme na n čísel, pak časová složitost je $T(n) = \Theta(n^2)$ a prostorová $S(n) = \Theta(n)$.

Úvod do grafových algoritmů

Další důležitou a zajímavou kapitolou jsou grafové algoritmy. Například následující příklady lze (i když to tak občas na první pohled nevypadá) řešit nějakým grafovým algoritmem:

- Mám mapku silniční sítě, v ní místa (vrcholy) označená „Doma“ a „Škola“. Dostanu se do školy (leží ve stejné komponentě souvislosti)? Dostanu se do školy, když v zimě napadne hodně sněhu a některé cesty budou neprůjezdné? A jaký nejkratší úsek cest musí silničáři prohrnout, aby byla všechna místa na mapě dostupná?
- Mějme hlavolam „Lloydova devítka“ – krabičku 3×3 se čtverečky označenými čísly od jedné do osmi a jednou mezerou, čtverečky jsou zamíchané a naším úkolem je správně je seřadit pomocí přesouvání čtverečků sousedících s mezerou do této mezery. Jak to udělat? Kolik nejméně kroků nám na to stačí? Jde to vůbec se zadáním, které jsme dostali?
- Jaké je nejkratší (kladné, celé) číslo v desítkové soustavě zapsané jen číslicemi 1, 0, které je dělitelné třinácti? Nakreslíme orientovaný

graf s vrcholy 0 až 12 a hranami (x, y), $y = 10 \cdot x \bmod 13$ a $y = (10 \cdot x + 1) \bmod 13$ (z každého vrcholu vychází jedna hrana za přidání číslice 1 a další za číslici 0). Hledané číslo existuje právě tehdy, když graf obsahuje orientovaný sled z 1 do 0. Jakým algoritmem takový sled najdeme?

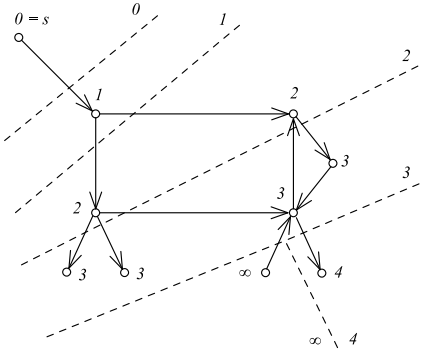
Podobné a další úlohy budeme řešit v následujících kapitolách.

3. Prohledávání grafů

Prohledání do šířky *Breadth-First Search – BFS*

Jde o grafový algoritmus, který postupně prochází všechny vrcholy v dané komponentě souvislosti. Algoritmus nejprve projde všechny sousedy počátečního vrcholu, poté sousedy sousedů, atd... Díky tomuto způsobu procházení se někdy též nazývá „*algoritmus vlny*", neboť se z počátečního vrcholu šíří pomyslná vlna, která v každém kroku nalezne všechny uzly, které mají od počátečního vrcholu stejnou vzdálenost. Algoritmus se tedy skvěle hodí například pro hledání nejkratší cesty mezi dvěma vrcholy v grafu.

Zatím předpokládejme, že graf, se kterým pracujeme, je orientovaný. Orientovanou hranu (u, v) z u do v budeme obvykle zkracovat jako uv. Pro neorientované grafy bude vše obdobné.



Prasečí graf a průchod vlny skrz něj

Popis algoritmu: Na začátku vložíme do fronty Q počáteční vrchol v_0. Dále si v poli Z budeme pro každý vrchol pamatovat značku, zda jsme ho již navštívili ($Z[v] = 1$), či nikoli ($Z[v] = 0$). Na počátku jsou všechny značky nulové, jen vrchol v_0, který je označen a vložen do fronty.

V každém dalším kroku pak zkoumáme frontu Q: pokud není prázdná, vezmeme z ní první vrchol u a podíváme se na všechny vrcholy v, do nichž z u vede hrana. Pokud sousedi ještě nejsou označení, tak je označíme a přidáme je do fronty k následnému zpracování. Toto opakujeme, dokud není fronta prázdná.

Algoritmus:

1. $Q \leftarrow \{v_0\}$.

2. $Z[*] \leftarrow 0, Z[v_0] \leftarrow 1$.
3. Dokud $Q \neq \emptyset$ opakujeme:
4. Vyzvedneme vrchol u z Q.
5. $\forall v : uv \in E$:
6. Je-li $Z[v] = 0 \Rightarrow Z[v] \leftarrow 1$, přidáme v do Q.

Pozorování: *BFS* se zastaví.
Důkaz: Zpracováváme jen vrcholy, které byly ve frontě. Každý vrchol se dostane do fronty maximálně jednou. (Každý je označen max. jednou, značky neodstraňujeme.)
Lemma: BFS(v_0) označí v právě tehdy, když existuje cesta z v_0 do v.
Důkaz: „$\Longrightarrow$“: Platí jako invariant po celou dobu běhu algoritmu. To dokážeme indukcí dle doby běhu algoritmu:

První krok indukce je triviální, neboť cesta z v_0 do v_0 existuje vždy. Nyní si představme, že označujeme vrchol v přes hranu uv. To znamená, že vrchol u již musel být označený. Dle indukčního předpokladu tedy existuje cesta z v_0 do u, a tudíž pokud k této cestě „přilepíme“ hranu uv, dostáváme sled z v_0 do v, který lze vždy zredukovat na cestu.

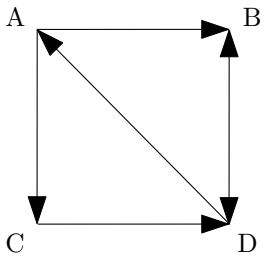
„$\Longleftarrow$“ Sporem: Nechť existuje neoznačený vrchol v dosažitelný po nějaké cestě z v_0. Uvažme nejkratší cestu (v_0, v): $v_0, v_1 \ldots, v_k = v$ a vezměme minimální i takové, že v_i není označený. Víme, že $i > 0$ (neboť v_0 je určitě označen už na začátku algoritmu). Tudíž v_{i-1} je označený. Při označení jsme ho ovšem přidali do fronty, takže jsme ho z fronty museli později zase vyjmout. Při tom jsme ovšem museli objevit hranu $v_{i-1}v_i$ a označit vrchol v_i, což je spor. ♡

Nyní tedy víme, že algoritmus je správný, a máme představu o tom, jak funguje. Podíváme-li se na něj podrobněji, zjistíme, že je hodně závislý na tom, jak si budeme graf pamatovat. Zanedlouho zároveň zjistíme, že nám reprezentace grafu v paměti znatelně ovlivní časovou (i paměťovou) složitost celého algoritmu.

Reprezentace grafu v paměti

Mějme nějaký orientovaný graf G s n vrcholy a m hranami. Jak ho reprezentovat?

Vrcholy můžeme očíslovat od 1 do n. Pro uložení hran máme na výběr hned několik způsobů. Předvedeme si je na grafu z následujícího obrázku:



Ukázkový graf

1. matice sousednosti

Matice sousednosti pro graf G na n vrcholech je čtvercová matice A o rozměrech $n \times n$, taková, že $A_{i,j}$ popisuje, jestli z vrcholu i do vrcholu j vede hrana ($A_{i,j} = 1$) nebo nikoliv ($A_{i,j} = 0$).

Náš graf z obrázku výše by tedy v maticové reprezentaci vypadal takto:

$$\begin{array}{c} \\ A \\ B \\ C \\ D \end{array} \begin{array}{c} \begin{array}{cccc} A & B & C & D \end{array} \\ \begin{pmatrix} 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 \\ 1 & 1 & 0 & 0 \end{pmatrix} \end{array}$$

S touto maticí se pracuje velmi snadno, např. všechny sousedy i-tého vrcholu zjistíme jednoduše tak, že projdeme i-tý řádek matice a najdeme všechny jedničky. Má ovšem dvě zřejmé nevýhody: časovou a paměťovou složitost. Projití sousedů jednoho vrcholu trvá vždy $\Theta(n)$, projití sousedů pro všechny vrcholy (což potřebujeme v BFS) pak trvá $\Theta(n^2)$. Velikost matice je vždy $n \times n$, bez ohledu na to, jak „řídký“ je graf. U grafu s mnoha vrcholy, ale s malým počtem hran, tedy budeme zbytečně plýtvat místem v paměti. Tato reprezentace je tedy nevýhodná především pro třídy grafů, jako jsou stromy, které mají $n-1$ hran, nebo rovinné grafy, které mají nejvýše $3n - 6$ hran.

Pozorování: BFS s reprezentací maticí sousednosti běží v čase: $\Theta(n^2)$.

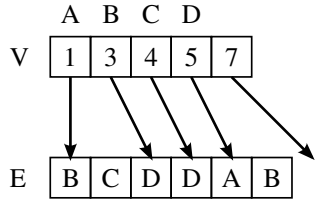
Důkaz: Už jsme si uvědomili, že každý vrchol se dostane do fronty Q nejvýše jednou. Pro každý vrchol ve frontě potřebujeme projít jeho sousedy, což nám trvá s reprezentací maticí sousednosti $\Theta(n)$. Vrcholů je celkem n, tedy časová složitost je $\Theta(n^2)$. ♡

2. seznam sousedů

V matici sousednosti jsme museli procházet jak hrany, tak nehrany, což bylo zbytečné. Bylo by tedy výhodnější pamatovat si pro každý vrchol pouze jeho sousedy. To můžeme zařídit například jedním ze dvou následujících způsobů:

Uchovávejme pole indexované vrcholy tak, že v každém prvku pole je ukazatel na spojový seznam sousedů tohoto vrcholu. Tedy $L(v) = \{w : vw \in E(G)\}$.

Pokud se nám nebude chtít pracovat se spojovými seznamy, můžeme využít reprezentaci pomocí dvou polí. První pole V bude opět indexované vrcholy. V druhém poli E budou pro každý vrchol uloženi jeho sousedé. V poli V si pamatujeme pro každý vrchol i index do pole E, kde začínají jeho sousedé, a navíc dodefinujeme, že $V[n+1] = m+1$. K sousedům vrcholu i se pak již dostaneme snadno – nalezneme je na pozicích $V[i], \ldots, V[i+1] - 1$.



Znázornění polí seznamu sousedů

Na tuto reprezentaci už stačí prostor $\Theta(n+m)$, což už je, na rozdíl od předchozího kvadratického prostoru, docela příjemné.

Pozorování: BFS s reprezentací seznamem sousedů běží v čase $\Theta(n+m)$.

Důkaz: Algoritmus zpracuje každý vrchol nejvýše jednou a stráví jím čas lineární v počtu odchozích hran, tedy $\Theta(\deg^-(v))$. Časová složitost celého algoritmu tedy činí:

$$\Theta\left(n + \sum_{v \in V(G)} \deg^-(v)\right) = \Theta(n+m).$$

♡

3. orákulum

Další možností reprezentace je jakési orákulum, které nám na požádání řekne (spočítá), kam vedou hrany z daného vrcholu. To se hodí například tehdy, pokud graf vznikl výpočtem a nechceme plýtvat pamětí na jeho uložení. To se hodí například u už zmíněného hlavolamu „Lloydova osmička“.

Neorientované grafy

Chceme-li reprezentovat neorientovaný graf, uložíme každou hranu v obou orientacích.

Výpočet vzdáleností

Abychom mohli využít toho, že algoritmus prochází vrcholy grafu ve vlně, k výpočtu vzdáleností, doplníme do něj dvě pomocná pole: $D[v]$ bude říkat, na kolik kroků jsme se do v dostali, $P[v]$ bude obsahovat *předchůdce* vrcholu v, totiž vrchol u, ze kterého jsme se do v dostali po hraně a jehož $D[u] = D[v] - 1$.

Rozšířený algoritmus:

1. $Q \leftarrow \{v_0\}$.
2. $Z[*] \leftarrow 0, Z[v_0] \leftarrow 1$.
3. $D[*] \leftarrow \infty, D[v_0] \leftarrow 0$.
4. Dokud $Q \neq \emptyset$ opakujeme:
5. Vyzvedneme vrchol u z Q.
6. Pro každý vrchol v, do kterého vede hrana z vrcholu u:
7. Je-li $Z[v] = 0$:
8. $Z[v] \leftarrow 1, D[v] \leftarrow D[u] + 1, P[v] \leftarrow u$
9. Přidáme v do Q.

Definice: *Fáze běhu algoritmu*: Ve fázi F_0 je zpracováván vrchol v_0. Ve fázi F_{i+1} jsou zpracovávány vrcholy uložené do fronty Q během fáze F_i.
Pozorování: Každý vrchol v dosažitelný z v_0 se účastní právě jedné fáze, a to té s číslem $D[v]$.
Lemma: Po zastavení BFS pro všechny vrcholy dosažitelné z v_0 platí, že $D[v]$ je rovno $d(v_0, v)$, totiž vzdálenosti (délce nejkratší cesty) z v_0 do v.
Důkaz: Nejprve si uvědomíme, že kdykoliv je v označen, vede do něj z v_0 nějaký sled délky v (indukcí, stejně jako jsme před chvílí dokazovali, že BFS projde všechny dosažitelné vrcholy). Proto nemůže být $D[v]$ menší než $d(v_0, v)$.

Sporem dokážeme, že nemůže být ani větší. Předpokládejme, že existuje nějaký „špatný" vrchol v, pro který je $D[v] > d(v_0, v)$. Nechť P je některá z nejkratších cest z v_0 do v. Z možných špatných vrcholů si vyberme takový, jehož P je nejkratší možná. Jelikož pro vrchol v_0 je zajisté $D[v_0] = d(v_0, v_0) = 0$, musí být v různý od v_0, takže má na P nějakého předchůdce u. Pro toho ovšem je vzdálenost spočítána správně: $D[u] = d(v_0, u) = d(v_0, v) - 1$.

Uvažujme nyní, co se stalo v okamžiku, kdy jsme $D[u]$ nastavili. Tehdy jsme u uložili do fronty, po čase jsme ho z fronty zase vytáhli a prozkoumali jsme všechny vrcholy, do níž vede z u hrana. Tedy i vrchol v, takže $D[v]$ v tomto okamžiku nemůže být větší než $D[u] + 1 = d(v_0, v)$, a to je spor. ♡

Víme tedy, že BFS správně spočítá délky nejkratších cest do všech vrcholů grafu. Pomocí předchůdců v poli P můžeme tyto cesty dokonce snadno rekonstruovat: předposledním vrcholem na nejkratší cestě do vrcholu v musí být vrchol $P[v]$, jeho předchůdcem $P[P[v]]$, ..., až do vrcholu v_0.

Předchůdci nám tedy kódují strukturu nejkratších cest do všech vrcholů. Můžeme se na ně dívat také následovně:
Definice: Strom nejkratších cest je orientovaný strom (W, F) s množinou vrcholů $W = \{v \in V(G) \mid v \text{ dosažitelný z } v_0\}$ a hranami $F = \{(P(v), v) \mid v \in W, v \neq v_0\}$.
Pozorování: Kořenem stromu nejkratších cest je vrchol v_0, cesta v tomto stromu z v_0 do v (jednoznačně určená, je to strom) je pak jednou z nejkratších cest z v_0 do v v původním grafu.

Komponenty souvislosti

V neorientovaných grafech můžeme BFS jednoduše použít na nalezení komponent souvislosti. Již víme, že BFS spuštěné z vrcholu v_0 projde právě ty vrcholy, které jsou z v_0 dosažitelné, což jsou v neorientovaném grafu přesně ty, které leží v téže komponentě.

Stačí opakovaně spouštět BFS z dosud neoznačených vrcholů, pokaždé nám označí jednu komponentu.
Algoritmus:

1. Pro všechny vrcholy $v \in V(G)$ opakujeme:
2. Pokud je vrchol v neoznačený:
3. Spustíme $BFS(v)$ a přiřadíme objevené vrcholy nové komponentě.

To, co jsme o BFS zjistili, můžeme shrnout do následující věty:

Věta: $BFS(v_0)$ v čase $\Theta(n+m)$ spočte:

- vrcholy dosažitelné z v_0
- vzdálenosti těchto vrcholů od v_0
- strom nejkratších cest z v_0

Prohledávání do šířky ale není jediný algoritmus, který nějak systematicky prochází graf. Jak už název kapitoly napovídá, budeme se zabývat ještě druhým algoritmem, prohledáváním do hloubky. Podívejme se, jak bude vypadat ...

Prohledávání do hloubky *Depth-First Search – DFS*

Tento algoritmus neprochází graf ve vlně jako BFS, nýbrž rekurzivně. Vždy se zanoří co nejhlouběji až do listu a pak se o kus vrátí a opět se snaží zanořit. Vrcholy, ve kterých už byl, ignoruje.

Opět uvažme nejdříve graf orientovaný. Následně si ukážeme, že v neorientovaném grafu budou pouze malé změny.

Budeme používat podobné značení jako u BFS. V poli Z si budeme pamatovat, zda jsme vrchol již navštívili (hodnota 1), nebo ne (hodnota 0). Aby se nám algoritmus lépe analyzoval, zavedeme proměnnou T, která bude fungovat jako hodiny – v každém kroku algoritmu se zvětší o jedničku. Do polí *in* a *out* si budeme ukládat čas prvního a posledního průchodu vrcholem.

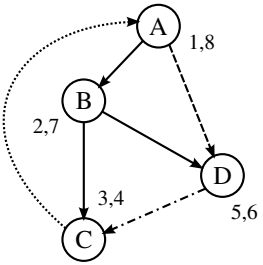
Algoritmus:

1. Inicializace: $Z[*] \leftarrow 0, T \leftarrow 1, in[*] \leftarrow ?, out[*] \leftarrow ?$
2. $DFS(v)$:
3. $\quad Z[v] \leftarrow 1$, $in[v] \leftarrow T$, $T \leftarrow T+1$
4. $\quad$ Pro w: $vw \in E(G)$:
5. $\quad\quad$ Pokud $Z[w] = 0$, zavoláme $DFS(w)$
6. $\quad out[v] \leftarrow T$, $T \leftarrow T+1$

Věta: DFS(v_0) v čase $\Theta(m+n)$ označí právě všechny vrcholy dosažitelné z v_0.
Důkaz: Korektnost dokážeme stejným argumentem, jako u BFS.

V analýze časové složitosti si pak opět uvědomíme, že algoritmus zavoláme na každý vrchol nejvýše jednou (pak už je označený) a zpracováním vrcholu strávíme čas lineární v počtu hran, které z něj vedou. Celkem tedy prozkoumáme každou hranu nejvýše jednou a strávíme tím konstantní čas. ♡

Vyzkoušejme si DFS na grafu z následujícího obrázku:



Znázornění průběhu DFS a typů hran

Graf je reprezentován tak, že v každém vrcholu jsou hrany uspořádány zleva doprava. Čísla u vrcholů ukazují jejich *in* a *out*.

Můžeme si všimnout, že k různým hranám se DFS chová různě. Po některých projde, jiné vedou do už prozkoumaných vrcholů, ale někdy do takových, ze kterých jsme se ještě nevrátili, jindy do už zpracovaných. Za chvíli uvidíme, že mohou nastat čtyři různé situace. Nejsnáze se poznávají podle hodnot *in* a *out*.

Pozorování: Chod algoritmu můžeme popsat posloupností závorek: $(_v$ bude značit „vstoupili jsme do vrcholu v“ (a nastavili $in(v)$), $)_v$ budiž opuštění vrcholu (nastavení $out(v)$). Tato posloupnost bude správně uzávorkovaná, čili páry závorek se nebudou křížit.

Klasifikace hran: DFS dělí hrany na následující čtyři druhy. Hrana uv je:

- *Stromová* (na našem obrázku plná) pokud po ní DFS prošlo do neoznačeného vrcholu. Všimněte si, že stromové hrany společně tvoří strom orientovaný od kořene v_0. (Je to strom, jelikož vzniká postupným přidáváním listů.) Tomuto stromu se říká DFS strom.
- *Zpětná* (tečkovaná) – taková hrana vede do vrcholu, do kterého jsme vstoupili, ale ještě jsme ho neopustili (to je vrchol, který máme při rekurzi na zásobníku). Odpovídá uzávorkování $(_v \ldots (_u \ldots)_u \ldots)_v$.
- *Dopředná* (čárkovaná) – vede do vrcholu, který jsme už opustili, ale který je potomkem aktuálního vrcholu: $(_u \ldots (_v \ldots)_v \ldots)_u$.
- *Příčná* (čerchovaná) – vede do vrcholu, který jsme už opustili, ale který ve stromu neleží pod aktuálním vrcholem. Musí tedy vést do vrcholů „nalevo“ od stromové cesty z v_0 do u. Napravo totiž leží vrcholy, které v okamžiku opouštění u ještě nebyly prozkoumané, takže hrana uv by se stala stromovou. Příčné hrany poznáme podle uzávorkování $(_v \ldots)_v \ldots (_u \ldots)_u$.
- Jiné druhy hran nemohou existovat, probrali jsme totiž všechny možnosti, v jakém vztahu mohou být páry závorek $(_u)_u$ a $(_v)_v$.

Kterého typu hrana je, můžeme tedy poznat podle značky $Z[v]$ a podle hodnot *in* a *out* vrcholů u a v.

Neorientované grafy: V neorientovaných grafech (každou hranu vidíme jako dvě orientované hrany) je situace daleko jednodušší. Buďto hranu objevíme jako stromovou (a v opačném směru ji vidíme jako zpětnou), nebo ji objevíme jako zpětnou (a v opačném směru se jeví dopřednou). Příčné hrany nemohou existovat, protože k nim opačná hrana by byla příčná vedoucí zleva doprava, což víme, že nenastane.

5. Nejkratší cesty

Na této přednášce budeme studovat problém hledání nejkratších cest v orientovaných grafech ohodnocených reálnými čísly.

Situace: Máme orientovaný graf G a funkci $\ell : E(G) \to \mathbb{R}$ přiřazující hranám jejich ohodnocení (délky). Pro vrcholy $u, v \in V(G)$ budeme chtít spočítat jejich vzdálenost $d(u, v)$, což bude délka nejkratší cesty z u do v nebo ∞, pokud žádná cesta neexistuje.

Chceme, aby se vzdálenosti chovaly „rozumně“, tedy co nejvíce jako metrika. Orientovanost grafů nám kazí symetričnost – nemusí nutně platit $d(x,y) = d(y,x)$. Budeme aspoň chtít, aby platily následující vlastnosti:

- $d(u,u) = 0$,
- $d(u,v) \leq d(u,w) + d(w,v)$ (trojúhelníková nerovnost).

To nemusí obecně platit (kazí nám to záporné cykly), proto budeme studovat pouze grafy, v nichž záporné cykly neexistují. Pak už budou obě vlastnosti splněny, jak plyne například z následujícího lemmatu:

Lemma: V grafu bez záporných cyklů existuje ke každému nejkratšímu sledu z u do v stejně dlouhá uv-cesta.

Důkaz: Máme-li nejkratší uv-sled, který není cestou, opakuje se v něm nějaký vrchol $w \in V(G)$ tedy uv-sled je $(u \ldots w \ldots w \ldots v)$. Délka cyklu $c = (w \ldots w)$ je $\ell(c) \geq 0$ tedy platí $\ell(u \ldots w \ldots v) \leq \ell(\text{původní sled})$. Tento postup můžeme opakovat a po konečném počtu kroků dostaneme cestu. Z toho plyne trojúhelníková nerovnost. ♡

Jednoduché případy

- Pokud ℓ je konstantní funkce, použijeme BFS. Časová složitost bude $\Theta(m+n)$.
- Délky hran jsou malá přirozená čísla – $\ell(x,y) \in \{1, \ldots, L\}$: Podrozdělíme hrany a použijeme BFS. Časová složitost $\Theta(Lm+n)$.
- V DAG (orientovaném acyklickém grafu) najdeme nejkratší cestu indukcí přes topologické uspořádání v čase $\Theta(m+n)$.

Obecný algoritmus

Definice: $D_k(v) :=$ minimální délka ze všech sledů z v_0 do v o právě k hranách. $D_0(v) = 0$ pokud $v = v_0$, jinak $D_0(v) = \infty$.

$d(v_0, v) = \min D_k(v)$ kde $1 \leq k \leq n-1$.

Jak spočítat D_k když už známe $D_0 \ldots D_{k-1}$? Zřejmně

$$D_k = \min\{D_{k-1}(u) + \ell(u,v)\}$$

pro taková u že $(u,v) \in E(G)$. Z tohoto můžeme udělat jednoduchý algoritmus: postupně pro všechna $k = 0 \ldots n-1$ zjistíme všechna $D_k(z)$ pro všechny vrcholy $z \in V(G)$. Délka nejkratší cesty z v_0 do v je $\min D_k(v)$ kde $1 \leq k \leq n-1$.

Naivní implementace poběží $n^2(\sum_v \deg^+(v)) + n$ (musíme přičíst na konec n za izolované vrcholy), upravíme $\sum_v \deg^+(v) = m$. Bohužel spotřebujeme $\Theta(n^2)$ paměti.

Nevýhodou této implementace je přistupování k hranám pozpátku. Pojďme zkusit nepatrně odlišný přístup.

Bellmanův-Fordův algoritmus

1. $D(*) \leftarrow \infty, D(v_0) \leftarrow 0$
2. Pro $k = 1, \ldots, n-1$:
3. Pro $\forall v \in V(G)$:
4. Pro $\forall w$ takové že $(v,w) \in E(G)$:
5. $D(w) \leftarrow \min(D(w), D(v) + \ell(v,w))$

Pokud by algoritmus i v n-tém kroku něco změnil, graf obsahuje záporný cyklus.
Věta: Bellmanův – Fordův algoritmus najde v čase $\Theta(nm)$ vzdálenosti $d(v_0, v)$ z v_0 do všech $v \in V(G)$.
Důkaz: Invarianty:

- Konečné $D(v)$ vždy odpovídá délce nějakého sledu z $v_0 \to v$.

Pro vrchol v_0 určitě existuje sled nulové délky z v_0 do v_0.

Jak se z původního nekonečného $D(v)$ mohlo stát konečné? Našli jsme takovou hranu, která vedla z vrcholu w s konečným $D(w)$ do vrcholu v. Tedy do v existuje sled.

Pokud snížím $D(v)$, znamená to, že jsem do vrcholu v našel kratší sled.

- Na konci k-tého průchodu vnějším cyklem platí $D(w) \leq$ min délka sledu z v_0 do v o nejvýše k hranách. Z čehož plyne že na konci je $D(w) \leq d(v_0, v)$.

Nechť nejkratší sled $v_0 \to w$ o nejvýše k hranách končí hranou (v, w). Zastavme algoritmus v okamžiku, kdy v k-tém průchodu zpracovává hranu (v, w) tehdy $D(w) \leq D(v) + \ell(v, w)$ a $D(v)$ je dle indukčního předpokladu menší nebo rovno minimální délce sledu z v_0 do v o nejvýše k hránách. ♡

V Bellman – Fordově algoritmu jsme v podstatě zlepšovali odhady na nejkratší cestu. Pojďme vymyslet algoritmus založený na podobné myšlence.
„Průzkumnický algoritmus“ Pro každý vrchol budeme udržovat jeho ohodnocení (dočasnou vzdálenost) $D(v)$ a stav vrcholu $S(v)$. Stav může být buď N neviděn – vrchol jsme ještě nepotkali, O otevřen – od posledního prozkoumání se $D(v)$ změnilo nebo Z zavřen – není potřeba zkoumat znovu, nic by se nezměnilo. Abychom mohli nejkratší cestu na konci běhu také zrekonstruovat, budeme si ještě udržovat $P(v)$ předchůdce vrcholu v.

1. $D(*) \leftarrow \infty, D(v_0) = 0, S(*) \leftarrow N, S(v_0) \leftarrow O, P(*) \leftarrow ?$
2. Dokud $\exists u$: $S(u) = O$ opakuj:
3. $\quad S(u) \leftarrow Z$
4. $\quad$ Pro $\forall v : (u, v) \in E(G)$:
5. $\quad\quad$ Je-li $D(u) + \ell(u, v) < D(v)$
6. $\quad\quad\quad D(v) \leftarrow D(u) + \ell(u, v)$
7. $\quad\quad\quad S(v) \leftarrow O$

Invariant: $D(v)$ neroste a odpovídá délce nějakého sledu z v_0 do v.

$D(v)$ volíme jako minimum z $D(v)$ a $D(u)+\ell(u, v)$, proto nikdy nemůže vzrůst.

Důkaz toho, že $D(v)$ odpovídá délce nějakého sledu z v_0 do v je stejný jako u Bellman – Fordova algoritmu.
Lemma: Algoritmus se zastaví při jakémkoliv pořadí zavíraných vrcholů. Čas při nevhodném zavírání může být až exponenciální.
Toto lemma bude zanecháno bez důkazu, neboť ho nebudeme potřebovat pro důkaz správnosti algoritmů, které od „průzkumnického algoritmu“ odvodíme.

Lemma: Pokud se algoritmus zastaví, pak dosažitelné vrcholy jsou zavřené a kdykoliv $S(v) = Z$, platí $D(v) = d(v_0, v)$.
Důkaz: Nechť cesta z v_0 do v je co do počtu hran nejmenší protipříklad, pak musí existovat vrchol u, pro který neplatí $S(u) = Z$. Což je spor, protože takový vrchol musel náš algoritmus projít.

Vezměme minimání protipříklad co do počtu hran. Nechť v je nejbližší vrchol od u takový, že $D(v) \neq d(v_0, v)$, tudíž musí být větší, než by měl být, protože odpovídá délce nějakého sledu. Označme u předchůdce vrcholu v na nejkratší cestě. Víme, že $D(u)$ je správně. Algoritmus zkoumal u nastavil finální $D(u)$, tedy přitom zpracoval hranu (u, v) a $D(v) \leq D(u) + \ell(u, v)$, což je opravdová vzdálenost. Dostali jsme spor s tím, že $D(x)$ neroste. ♡

6. Dijkstrův algoritmus a haldy

FIXME: Nerevidováno!

Na této přednášce budeme pokračovat v problému hledání nejkratších cest v grafech ohodnocených reálnými čísly. Již jsme potkali Bellmanův-Fordův algoritmus a jeho zobecnění v podobě průzkumnického algoritmu.
Situace: Máme orientovany graf G a funkce $l : E(G) \to \mathbb{R}$ přiřazující hranám jejich ohodnocení (délky). Pro vrcholy $u, v \in V(G)$ budeme chtít spočítat jejich vzdálenost $d(u, v)$, což bude délka nejkratší cesty z u do v nebo ∞, pokud žádná cesta neexistuje.

Aby se vzdálenosti chovaly „rozumně“ (tj. jako metrika), budeme chtít, aby platily následující vlastnosti:

- $d(u, u) = 0$,
- $d(u, v) \leq d(u, w) + d(w, v)$ (trojúhelníková nerovnost).

To nemusí obecně platit (kazí nám to záporné cykly), proto budeme studovat pouze grafy, v nichž záporné cykly neexistují. Pak už budou obě vlastnosti splněny, jak plyne například z následujícího lemmatu:
Lemma: V grafu bez záporných cyklů existuje ke každému nejkratšímu sledu z u do v stejně dlouhá uv-cesta.
Důkaz: Máme-li nejkratší uv-sled, který není cestou, opakuje se v něm nějaký vrchol $w \in V(G)$. Délka cyklu $l(c) \geq 0 \Rightarrow l(u \ldots v \ldots w) \leq l(\textit{původnísled})$. Tento postup můžeme opakovat a po konečném počtu kroků dostaneme cestu, tedy platí trojúhelníková nerovnost. ♡

Opakování: Dijkstrův algoritmus

- $D(v)$ ... dočasná vzdálenost z s do v
- Značky $Z(v)$
  - :Neviděn
  - :Viděn
  - :Hotov

1. $D(*) \leftarrow +\infty, D(s) \leftarrow 0$
2. $Z(*) \leftarrow Neviden, Z(s) \leftarrow V$

3. while $\exists v : Z(v) = V$
4. vybereme $v : Z(v) = V, D(v) = min$
5. $Z(v) = H$
6. for $\forall w : (v, w) \in E(G)$
7. $D(w) \leftarrow min(D(w), D(w) + l(v, w))$
8. if $Z(w) = N \Rightarrow Z(w) \leftarrow V$

Věta: Pokud G je nezáporně ohodnocený graf, pak se Dijkstrův algoritmus zastaví a vydá $\forall v : D(v) = d(s, v)$ (tedy správné hodnoty).
Důkaz: Následující posloupností lemmat. ♡
Lemma 1: Pokud $v_0, \ldots, v_k$ je nejkratší v_0v_k-cesta, pak $v_0, \ldots, v_{k-1}$ je nejkratší v_0v_{k-1}-cesta.
Důkaz: Pokud by tomu tak nebylo, můžeme $v_0, \ldots, v_{k-1}$ vyměnit za kratší cestu, a tím získat kratší v_0v_k-cestu. ♡
Lemma 2: Algoritmus se zastaví po $\leq n$ průchodech cyklem.
Důkaz: Zřejmé z toho, že každý vrchol uzavřeme nejvýše jednou. ♡
Lemma 3: Po zastavení jsou hotovy právě vrcholy dosažitelné z s.
Důkaz: Viz minulá přednáška. ♡
Lemma 4: $D(v)$ uzavíraných vrcholů tvoří neklesající posloupnost.
Důkaz: V okamžiku, kdy uzavíráme v platí $\forall w \notin H : D(w) \geq D(v)$, případně přepočítáme $D(w)$ na $D(v) - l(v, w) \geq D(v)$. ♡
Lemma 5: Pokud $v \in H$, pak $D(v)$ se už nezmění.
Důkaz: Indukcí podle běhu algoritmu. ♡
Lemma 6: Pro $\forall v D(v)$ je délka nejkratší sv-cesty, jejíž vnitřní vrcholy leží všechny v H.
Důkaz:

- po 1. průchodu OK
- uzavíráme-li další vrchol v:

  a) $D(w)$ pro $w \in H$ Podle Lemma 5 se $D(w)$ nemění. Musíme nahlédnout, že se opravdu změnit nemá.

  b) $D(w)$ pro $w \notin H$ $D(w) = minD(v) + l(v, w)$

♡

Existuje pomalejší algoritmus pro grafy se zápornými hranami bez záporných cyklů. **Bellman-Fordův algoritmus**

1. $D(*) \leftarrow \infty, D(s) \leftarrow 0$
2. Opakuji
3. Pro $\forall v \in V$
4. $prozkoumej(v)$
5. (koukne se, jestli nejde vylepšit cestu
6. do sousedů v)
7. dokud se nějaké $D(\ldots)$ mění

Lemma 1: Pokud $D(v) < \infty$, pak existuje sled z s do v délky $D(v)$. (speciálně z toho plyne $\forall v D(v) \geq d(s,v)$)
Lemma 2: $\forall v D(v)$ nikdy neroste. **Lemma 3:** Po k fázích je $\forall v D(v) \leq$ délka nejkratšího sv-sledu o $\leq k$ hranách. *Důkaz:* Indukcí... pro $k \geq 0$ OK Indukční krok: Doběhla $k-1$ fáze, pouštíme k-tou

♡

Věta: Pro graf bez záporných cyklů se Bellman-Fordův algoritmus zastaví po nejvýše m fázích a vydá $D(v) = d(s,v)$ pro všechna v. (zřejmé z lemmat)
Lemma 4: Pokud v grafu existuje záporný cyklus dosažitelný z s, algoritmus se nezastaví. (Zajímavý test na to, zda graf obsahuje záporný cyklus.)

Časová složitost Bellman-Fordova algoritmu : $O(n * m)$

Floyd-Warshallův algoritmus G je graf se záporným ohodnocením hran, bez záporných cyklů $D^k_{i,j}$ = délka nejkratší cesty z v_i do v_j přes $v_1 \dots v_k$ $D^0_{ij} = l(v_i, v_j)$, $D^0_{ii} = 0$ $D^n_{ij} = d(v_i, v_j)$ - skutečná vzdálenost v grafu

1. $D_{i,j} \to l(v_i, v_j), D_{i,i} \to 0 \forall i, j$
2. for $k = 1$ to n
3. for $i = 1$ to n
4. for $j = 1$ to n
5. $D_{i,j} = min(D_{i,j}, D_{i,k} + D_{k,j})$

Věta: Floyd-Warshallův algoritmus spočítá $D_{i,j} = d(v_i, v_j)$ v čase $O(n^3)$.
Shrnutí:
$s \to *$ Dijkstrův algoritmus $O(n^2)$ ... s haldou $O((n+m) \log m)$ ... s regulární haldou $O(n + m * logn)$ ... s Fibbonaciho haldou $O(n * logn + m)$ Bellman-Ford $O(n * m)$
$* \to *$ Floyd-Warshall $O(n^3)$

7. Problém minimální kostry

Zadání úlohy: Pro neorientovaný graf G s ohodnocením hran *váhami* $w : E(G) \to \mathbb{R}$, chceme najít kostru T s minimálním ohodnocením $w(T) := \sum_{e \in E(T)} w(e)$.
Navíc předpokládáme: (bez újmy na obecnosti)

- Graf G je souvislý (jinak ho nejprve rozložíme na komponenty).
- Váhy hran jsou navzájem různé

Nyní si ukážeme tři algoritmy pro hledání minimální kostry, konkrétně se jedná o Jarníkův, Borůvkův a Kruskalův algoritmus.

Jarníkův algoritmus

Algoritmus:

1. *Vstup:* Graf G s ohodnocením w.
2. Zvolíme libovolný vrchol $v_0 \in V(G)$.
3. $T \leftarrow (\{v_0\}, \emptyset)$ (zatím jednovrcholový strom)
4. Dokud existuji vrcholy mimo T:

5. Vybereme hranu $uv \in E(G) : u \in V(T), v \notin V(T)$ tak, aby $w(uv)$ byla minimální.
6. $T \leftarrow T + uv$.
7. *Výstup:* Minimální kostra T.

Věta: Jarníkův algoritmus se zastaví po maximálně n iteracích a vydá minimální kostru grafu G.
Důkaz: Při každé iteraci algoritmus přidá jeden vrchol do T, a proto se po maximálně n iteracích zastaví. Vydaný graf je strom, protože se stále přidává list k již existujícímu stromu, a jelikož má n vrcholů, je to kostra. Zbývá nám už jen dokázat, že nalezená kostra je minimální. K tomu pomuže následující lemma:

> **Definice:** *Řez* v grafu $G = (V, E)$ je množina hran $F \subseteq E$ taková, že $\exists A \subset V : F = \{\{u, v\} \in E : u \in A, v \notin A\}$.
> **Lemma (řezové):** Pokud G je graf, w jeho prosté ohodnocení, F je řez v grafu G a f je nejlehčí hrana v řezu F, pak pro každou minimální kostru T grafu G je $f \in E(T)$.
> *Důkaz:* Sporem: Buď T kostra a $f = uv \notin E(T)$. Pak existuje cesta $P \subseteq T$ spojující u a v. Cesta musí řez alespoň jednou překročit. Proto existuje $e \in P \cap F$ a navíc víme, že $w(e) > w(f)$. Uvažme $T' = T - e + f$. Tento graf je rovněž kostra grafu G, protože odebraním hrany e se graf rozpadne na dvě komponenty a přidáním hrany f se tyto komponenty opět spojí. Navíc $w(T') = w(T) - w(e) + w(f) < w(T)$, což je spor s minimalitou F. ♡

V důkazu korektnosti Jarníkova algoritmu toto lemma využijeme tak, že si všimneme, že hrany mezi vrcholy stromu T a zbytkem grafu tvoří řez a algoritmus nejlehčí hranu tohoto řezu přidá do T. Podle lemmatu tedy všechny hrany T musí být součástí každé minimální kostry a jelikož T je strom, musí být minimální kostrou.

♡

Důsledky: Graf G s prostým ohodnocením má pravě jednu minimální kostru. Minimální kostra je jednoznačně určená lineárním uspořádáním hran podle vah (na konkrétních hodnotách vah nezáleží).
Implementace:

- Přímočará: pamatujeme si, které vrcholy a hrany jsou v kostře T a které ne. Časová složitost je $\mathcal{O}(nm)$.
- Chytřejší: Pro každý vrchol $v \notin V(T)$ si pamatujeme

$$D(v) = \min \{w(uv) : u \in T\},$$

  tedy nejlehčí hranu, která vede mezi T a v. Při každém průchodu hlavním cyklem pak procházíme všechna $D(v)$ (to vždy trvá $\mathcal{O}(n)$) a při přidání vrcholu do T kontrolujeme okolní $D(w)$ pro $vw \in E$ a případně je snižujeme (za každou hranu $\mathcal{O}(1)$). Časovou složitost tím celkově zlepšíme na $\mathcal{O}(n^2 + m) = \mathcal{O}(n^2)$.
- S použitím haldy: $D(v)$ ukládáme do haldy. Potom provedeme nanejvýš n-krát *ExtractMin*, nanejvýš n-krát *Insert* a nanejvýš m-krát

Decrease. Pro binární haldu to má časovou složitost $\mathcal{O}(m \log n)$. Všimněte si, že Jarníkův algoritmus s $D(v)$ je velmi podobný Dijkstrovu algoritmu pro nejkratší cesty. Rozbor složitosti pro různé typy hald proto také dopadne stejně.

Borůvkův algoritmus

Algoritmus:

1. *Vstup:* Graf G s ohodnocením w.
2. $F \leftarrow (V(G), \emptyset)$
3. Dokud F má alespoň dvě komponenty (F není souvislý):
4. Pro každou komponentu F_i lesa F vybereme nejlehčí incidentní hranu e_i.
5. Všechny hrany e_i přidáme do F.
6. *Výstup:* Minimální kostra F.

Věta: Borůvkův algoritmus se zastaví po $\lfloor \log_2 n \rfloor$ iteracích a vydá minimální kostru grafu G.

Důkaz: Všimněme si nejprve, že po k iteracích mají všechny komponenty grafu F minimálně 2^k vrcholů.

To nahlédneme indukcí – na počátku jsou všechny komponenty jednovrcholové, v každé další iteraci se komponenty slučují do větších, každá s alespoň jednou sousední, takže se velikosti komponent minimálně zdvojnásobí.

Proto nejpozději po $\lfloor \log_2 n \rfloor$ iteracích už velikost každé komponenty dosáhne počtu všech vrcholů a algoritmus se zastaví, takže komponenta může být jen jedna.

Hrany mezi každou komponentou a zbytkem grafu tvoří řez, takže podle řezového lemmatu všechny hrany přidané do F musí být součástí (jednoznačně určené) minimální kostry. Graf $F \subseteq G$ je tedy vždy les a až se algoritmus zastaví, bude tento les roven minimální kostře. ♡

Implementace:

- Inicializace přímočará.
- Pomocí DFS rozložíme les na komponenty. Pro každý vrchol si pamatujeme číslo komponenty.
- Pro každou hranu zjistíme, do které komponenty patří, a pro každou komponentu si uchováváme nejlehčí hranu.

Takto dokážeme každou iteraci provést v čase $\mathcal{O}(m)$ a celý algoritmus doběhne v $\mathcal{O}(m \log n)$.

Kruskalův neboli hladový algoritmus

Algoritmus:

1. *Vstup:* Graf G s ohodnocením w.
2. Setřídíme všechny hrany z $E(G)$ tak, aby platilo $w(e_1) < ... < w(e_m)$.
3. $F \leftarrow (V(G), \emptyset)$.
4. Pro $i = 1$ do m:

5. Pokud $F + e_i$ je acyklický, provedeme $F \leftarrow F + e_i$.
6. *Výstup:* Minimální kostra F.

Věta: Kruskalův algoritmus se zastaví po m iteracích a vydá minimální kostru.
Důkaz: Každá iterace algoritmu zpracovává jednu hranu, takže iterací je m. Indukcí dokážeme, že F je vždy podgrafem minimální kostry: prázdné počáteční F je podgrafem čehokoliv, každá hrana, kterou pak přidáme, je minimální v řezu oddělujícím nějakou komponentu F od zbytku grafu (ostatní hrany tohoto řezu ještě nebyly zpracovány, a tudíž jsou těžší). Naopak žádná hrana, kterou jsme se rozhodli do F nepřidat, nemůže být součástí minimální kostry, jelikož s hranami, o kterých již víme, že v minimální kostře leží, tvoří kružnici. ♡
Implementace:

- Setřídění v čase $\mathcal{O}(m \log m) = \mathcal{O}(m \log n)$.
- Potřebujeme udržovat komponenty souvislosti grafu F, abychom uměli rychle určit, jestli právě zpracovávaná hrana vytvoří kružnici. Potřebujeme tedy strukturu pro udržování komponent souvislosti, které se m-krát zeptáme, zda dva vrcholy leží v téže komponentě (tomu budeme říkat operace *Find*), a $(n-1)$-krát spojíme dvě komponenty do jedné (*Union*).

Kruskalův algoritmus tedy poběží v čase $\mathcal{O}(m \log n + mT_f + nT_u)$, kde T_u je čas na provedení jedné operace *Union* a T_f na operaci *Find*.
Jednoduchá struktura pro komponenty: Budeme si pamatovat v poli čísla komponent, ve kterých leží jednotlivé vrcholy. *Find* zvládneme v čase $\mathcal{O}(1)$, ale *Union* bude stát $\mathcal{O}(n)$. Celý algoritmus pak poběží v čase $\mathcal{O}(m \log n + m + n^2) = \mathcal{O}(m \log n + n^2)$.
Chytřejší struktura: Každou komponentu si uložíme jako strom orientovaný směrem ke kořeni – každý vrchol si pamatuje svého otce, navíc každý kořen si pamatuje hloubku stromu. Operace *Find* vystoupá z obou vrcholů ke kořeni a kořeny porovná. *Union* rovněž najde kořeny a připojí kořen mělčí komponenty pod kořen té hlubší (pokud jsou obě stejně hluboké, vybere si libovolně). Obojí zvládneme v čase lineárním v hloubce stromu a jak si ukážeme, tato hloubka je vždy nejvýše logaritmická, a proto celý Kruskalův algoritmus poběží v čase $\mathcal{O}(m \log n + m \log n + n \log n) = \mathcal{O}(m \log n)$.[⟨1⟩]
Lemma: *Union-Find* strom hloubky h má alespoň 2^h prvků.
Důkaz: Indukcí: Pokud *Union* spojí strom s hloubkou h s jiným s hloubkou menší než h, pak hloubka výsledného stromu zůstává h. Pokud spojuje dva stromy stejné hloubky h, pak má výsledný strom hloubku $h+1$. Z indukčního předpokladu víme, že strom hloubky h má minimálně 2^h vrcholů, a tedy výsledný strom hloubky $h+1$ má alespoň 2^{h+1} vrcholů. ♡

[⟨1⟩] Drobnou úpravou bychom mohli dosáhnout daleko efektivnější struktury, ale tu bychom neupotřebili, jelikož by nás stejně brzdilo třídění, a analýza složitosti by byla ... inu, složitější.

8. Datové struktury

Co si můžeme představit pod pojmem datová struktura? V našich programech často chceme některé věci abstrahovat (použitím funkcí, objektů…). Proč tedy nezkusit abstrahovat i operace s daty? Napřed si určíme, co s našimi daty budeme provádět a pak vymyslíme jejich co nejrychlejší reprezentaci.

Můžeme třeba chtít udržovat konečnou *množinu* X prvků z nějakého universa $X \subseteq U$. Kde universum mohou být například přirozená čísla, tedy universum může být nekonečné na rozdíl od X.

Na našich datech budeme chtít provádět následující operace:

- *Insert* – vložit novou položku
- *Delete* – smazat položku
- *Find* – najít položku

Jak měřit časovou složitost? Určitě ji nechceme měřit vůči délce vstupu, protože ho nemusíme mít celý najednou ve struktuře. Časovou složitost jednotlivých operací tedy počítejme vzhledem k počtu prvků obsažených v datové struktuře v daný okamžik.

Také můžeme chtít udržovat *slovník*, tedy množinu dvojic (k, v) kde $k \in U$ se nazývá klíč a v hodnota. Dále předpokládejme, že U je lineárně uspořádaná a s klíči i hodnotami pracujeme v konstantním čase.

Dále budeme zkoumat jen slovník, protože množina je jen jeho speciálním případem.

Po slovníku budeme chtít:

- *Insert(k, v)* – vložit novou hodnotu spolu s klíčem (předpokládajíce, že ve struktuře dosud tento klíč není přítomen)
- *Delete(k)* – smazat položku podle klíče
- *Find(k)* – najít položku podle klíče

V následující tabulce jsou některé možné způsoby reprezentace naší datové struktury:

	Insert	*Delete*	*Find*
pole	$\Theta(1)$	$\Theta(n)$	$\Theta(n)$
setříděné pole	$\Theta(n)$	$\Theta(n)$	$\Theta(\log n)$
spojový seznam	$\Theta(1)$	$\Theta(n)$	$\Theta(n)$
setříděný seznam	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$
vyhledávací stromy	$\Theta(\log n)$	$\Theta(\log n)$	$\Theta(\log n)$

Pozorování: Proces binárního vyhledávání v setříděném poli se dá reprezentovat binárním vyhledávacím stromem.

Definice: *Binární strom:* Strom je binární, pokud je zakořeněný a každý vrchol má nejvýše dva syny, u nichž rozlišujeme, který je levý a který pravý.

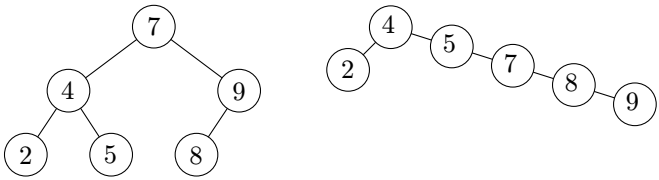
Definice: Pro vrchol v značíme:

- $l(v)$ a $p(v)$ – levý a pravý syn vrcholu v
- $L(v)$ a $P(v)$ – levý a pravý podstrom vrcholu v

- $S(v)$ – příslušný podstrom s kořenem v
- $h(v)$ – hloubka stromu $S(v)$ – délka nejdelší cesty z kořene do listu

Definice: *Binární vyhledávací strom* (BVS): Binární strom je vyhledávací, pokud v každém vrcholu je uložena dvojice (klíč, hodnota) [ztotožníme vrchol s klíčem] a pro všechny vrcholy platí: $(\forall u \in L(v) : u < v)$ & $(\forall u \in P(v) : u > v)$.

Příklady binárních vyhledávacích stromů:



Jak budou tedy vypadat operace *Find*, *Insert* a *Delete* na binárním vyhledávacím stromu?

Find(v, x)**:**

1. Pokud $v = \emptyset \Rightarrow$ vrátíme $\emptyset$.
2. Pokud $v = x \Rightarrow$ vrátíme v.
3. Pokud $v < x \Rightarrow$ vrátíme $Find(p(v), x)$.
4. Pokud $v > x \Rightarrow$ vrátíme $Find(l(v), x)$.

Insert(v, x)**:**

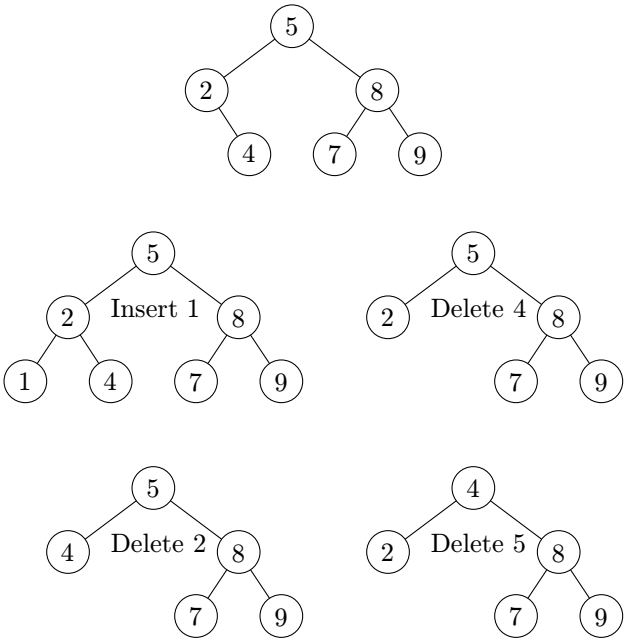
1. Jako *Find* a na konci přidáme list.

Delete(v)**:**

1. Pokud v je list $\Rightarrow$ jednoduše list utrhneme.
2. Pokud v má jednoho syna $\Rightarrow$ vrchol „vyřízneme“.
3. Jinak má v oba syny $\Rightarrow$ do vrcholu vložíme minimum z $P(v)$, minimum už umíme smazat protože je to buď list nebo má jen pravého syna.

Poznámka: Pokud má vrchol v při operaci *Delete* oba syny, je vložení minima z $P(v)$ ekvivalentní s vložením maxima z $L(v)$.

Příklady operací *Insert* a *Delete* na BVS:



Časová složitost všech tří operací je $\Theta(\textit{hloubka stromu})$, což může být $\Theta(n)$, když budeme mít smůlu a strom bude (téměř) lineární spojový seznam. Takovéto degenerované stromy vzniknou snadno, například přidáváním setříděné posloupnosti. Naopak když bude strom pěkně vyváženě vystavěný, dostaneme složitost $\Theta(\log n)$. Vidíme tedy, že složitost operací stojí a padá s hloubkou stromu. Proto by se nám líbilo, aby měl náš strom vždy hloubku $\Theta(\log n)$. Podívejme se tedy, jak se dá navrhnout binární vyhledávací strom, aby tuto podmínku splňoval ...

Vyvážené binární vyhledávací stromy

Definice: *Dokonalá vyváženost:* Strom je dokonale vyvážený, pokud pro všechny jeho vrcholy platí: $||L(v)| - |P(v)|| \leq 1$.

Takto definovaný binární strom bude mít určitě logaritmickou hloubku. Jak takový strom ale konstruovat? To se nám podaří buď staticky, nebo na něm budou operace dražší než $\Theta(\log n)$.

Statická konstrukce dokonale vyváženého BVS: Vybereme prostřední prvek ze setříděného pole (tedy medián posloupnosti) a dáme ho do kořene stromu. Jeho syny pak vystavíme rekurzivně z levé a pravé půlky pole. Celá konstrukce tedy trvá $\mathcal{O}(n)$.

Lemma: Buď *Insert* nebo *Delete* v dokonale vyváženém BVS trvá $\Omega(n)$ (ve skutečnosti oba, ale důkaz je o trochu obtížnější).

Důkaz: Nechť $n = 2^k - 1$. Pak má dokonale vyvážený BVS určený tvar a je jednoznačné, která hodnota je ve kterém vrcholu. Nechť nejmenší číslo ve stromě je x a největší je $x + n - 1$.

Proveďme posloupnost operací

$$Insert(x+n), Delete(x), Insert(x+n+1), Delete(x+1)\ldots$$

za každou dvojici operací se aspoň polovina listů posune o patro výš. Víme že listů v našem stromě je $(n+1)/2$. Tedy aspoň jedna z operací *Insert* nebo *Delete* trvá $\Omega(n)$.

Vidíme tedy, že to náš problém příliš neřeší. Potřebovali bychom, aby se strom dal také efektivně udržovat. Zkusíme proto slabší podmínku:

Definice: *Hloubková vyváženost:* Strom je hloubkově vyvážený, pokud pro všechny jeho vrcholy platí: $|h(L(v)) - h(P(v))| \leq 1$.

Stromům s hloubkovým vyvážením se říká *AVL stromy* (objeviteli je ruští matematikové G. M. Aděľson-Veľskij a E. M. Landis, podle nich jsou také pojmenovány) a platí o nich následující lemma:

Lemma: AVL strom na n vrcholech má hloubku $\Theta(\log n)$.

Důkaz: Uvažme posloupnost A_k = minimální počet vrcholů AVL stromu hloubky k. Stačí ukázat, že A_k roste exponenciálně.

Jak bude vypadat minimální AVL strom o k hladinách? S počtem hladin určitě poroste počet vrcholů, proto pro každý vrchol budeme chtít, aby se hloubky jeho synů lišily. Tedy kořen bude mít syna hloubky $k-1$ a syna hloubky $k-2$, takové, že jeho synové jsou minimální AVL stromy o daném počtu hladin.

Podívejme se na minimální AVL stromy:

$$\begin{aligned} A_0 &= 1 \\ A_1 &= 2 \\ A_2 &= 4 \\ A_3 &= 7 \\ &\vdots \\ A_k &= 1 + A_{k-1} + A_{k-2}. \end{aligned}$$

Rekurentní vzorec jsme dostali rekurzivním stavěním stromu hloubky k: nový kořen a 2 podstromy o hloubkách $k-1$ a $k-2$.

Indukcí bychom snadno dokázali, že $A_n = F_{n+2} - 1$ (kde F_n je n-té Fibonacciho číslo). My se však bez analýzy Fibonacciho posloupnosti obejdeme, protože nám stačí dokázat, že A_k rostou exponenciálně a nepotřebujeme přesný vzorec pro hodnotu A_k.

Indukcí dokážeme, že $A_k \geq 2^{\frac{k}{2}}$. První indukční krok jsme si už ukázali, teď pro $k \geq 2$ platí: $A_k = 1 + A_{k-1} + A_{k-2} > 2^{\frac{k-1}{2}} + 2^{\frac{k-2}{2}} = 2^{\frac{k}{2}} \cdot (2^{-\frac{1}{2}} + 2^{-1}) \cong 2^{\frac{k}{2}} \cdot 1.21 > 2^{\frac{k}{2}}$.

Tímto jsme dokázali, že na každé hladině je minimálně exponenciálně vrcholů, což nám zaručuje hloubku $\mathcal{O}(\log n)$. Druhou nerovnost nahlédneme zkoumáním B_k maximálního počtu vrcholů AVL stromu hloubky k. ♡

7. Vyhledávací stromy *zapsali M. Řezáč, Š. Masojídek, B. Urbancová*

Vyvážené binární vyhledávací stromy

V minulé kapitole jsme se zabývali problematikou přidávání a ubírání prvků binárního vyhledávacího stromu a jeho složitostí a zjistili, že vše záleží na hloubce stromu. Víme, že chceme hloubku logaritmickou, ale jak ji můžeme udržet při operacích? Řešením je následující definice:

Definice: *Dokonale vyvážení* je takové vyvážení, ve kterém pro všechny vrcholy v platí $||L(v)| - |P(v)|| \leq 1$.

Toto nám jistě zajišťuje logaritmickou hloubku, ale je velmi pracné na udržování. Zkusíme proto slabší podmínku:

Definice: *Hloubkové vyvážení* je takové vyvážení, ve kterém pro všechny vrcholy v platí $|h(L(v)) - h(P(v))| \leq 1$. Stromům s hloubkovým vyvážením se říká *AVL stromy* a platí o nich následující lemma.

Lemma: AVL strom o n vrcholech má hloubku $\mathcal{O}(\log n)$.

Důkaz: Uvažme a_k = minimální počet vrcholů AVL stromu o hloubce k. Lehce spočteme:

$$\begin{aligned} a_0 &= 0 \\ a_1 &= 1 \\ a_2 &= 2 \\ &\vdots \\ a_k &= 1 + a_{k-1} + a_{k-2}. \end{aligned}$$

Rekurentní vzorec jsme dostali rekurzivním stavěním stromu hloubky k: nový kořen a 2 podstromy o hloubce $k-1$ a $k-2$.

Indukcí dokážeme, že $a_k \geq 2^{\frac{k}{2}}$. První indukční krok jsme si už ukázali, teď pro $k \geq 2$ platí: $a_k = 1 + a_{k-1} + a_{k-2} > 2^{\frac{k-1}{2}} + 2^{\frac{k-2}{2}} = 2^{\frac{k}{2}} \cdot (2^{-\frac{1}{2}} + 2^{-1}) \cong 2^{\frac{k}{2}} \cdot 1.21 > 2^{\frac{k}{2}}$

Tímto jsme dokázali, že na každé hladině je minimálně exponenciálně vrcholů, což nám zaručuje hloubku $\mathcal{O}(\log n)$ ♡

Operace s AVL stromy

Find se neliší od operace find v binárních stromech.

Důraz klademe na operace *Insert* a *Delete*, protože při nich musíme ošetřit udržení struktury AVL stromů.

První nutnou podmínkou je, že si musíme *pamatovat stav* v každém vrcholu tohoto stromu. A to *vyvážení* hloubky jeho podstromů.

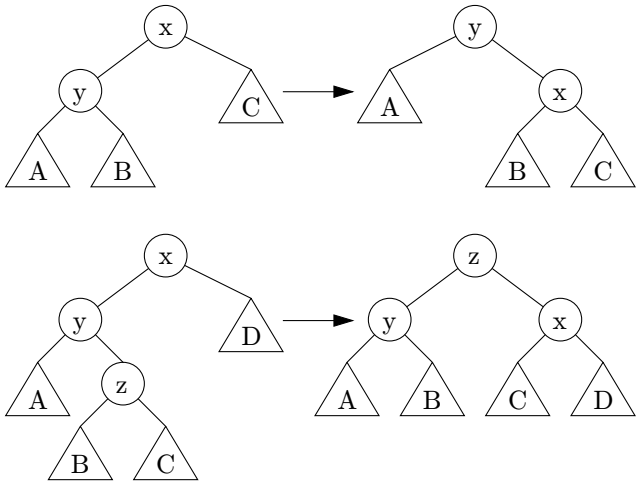
Umluvíme se např. na tomto označení:
Dostaneme tři typy vrcholů, které se mohou v AVL stromu vyskytnout:

- *Vrchol typu* $\oplus$, pokud je pravý podstrom hlubší
- *Vrchol typu* $\ominus$, pokud je levý podstrom hlubší a
- *Vrchol typu* $\odot$ *(nulou)*, který má oba syny shodné hloubky.

Sestavení AVL stromu:

Postupujeme po struktuře binárního stromu od listů ke kořeni a kontrolujeme, zda jsou vrcholy v jednom ze tří uvedených stavů. Pokud ne, opravíme strom operací jménem rotace.

Rotace



Jde o převrácení hrany mezi původním otcem (kořenem podstromu) a nevyváženým vrcholem tak, aby byli i po přeskupení synové vzhledem k otcům správně uspořádáni.

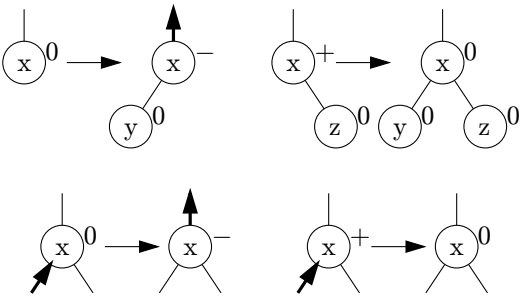
Insert - vložení vrcholu do AVL stromu.

Vložíme jej jako list. Nový list má vždy „znaménko“ nula $\odot$. Předpokládáme, že patří nalevo od posledního otce. Podíváme se na znaménko jeho otce:

- *měl* $\odot$ *(neměl syna)* $\rightarrow$ *teď má* $\ominus$, po struktuře stromu nahoru posíláme informaci, že se podstrom prohloubil o 1, což může mít samozřejmě vliv na znaménka vrcholů na cestě ke kořeni.
- *měl* $\oplus$ *(měl pravého syna, který je listem)* $\rightarrow$ *teď má* $\odot$, hloubka podstromu se nemění

- *měl* $\ominus$ – nenastane, protože v binární struktuře nemohou být dva leví synové

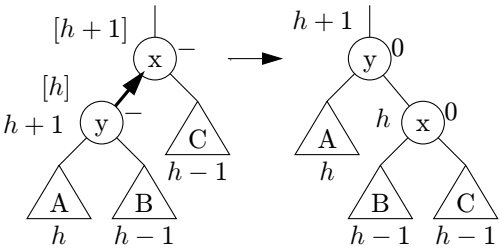
Připadne-li přidaný list napravo, řešíme zrcadlově.



Prohloubil-li se strom vložením nového listu, musíme pracovat s vyvážením:

- Informace o prohloubení přišla zleva *do vrcholu typu* $\odot \rightarrow$ mění jej na vrchol se znaménkem $\ominus$ a informace o prohloubení je třeba poslat o úroveň výš.
- Informace o prohloubení přišla zleva *do vrcholu typu* $\oplus \rightarrow$ mění jej na vrchol se znaménkem $\odot$, hloubka je vyrovnána, dál nic neposíláme.
- Informace o prohloubení přišla zleva *do vrcholu s* $\ominus \rightarrow$

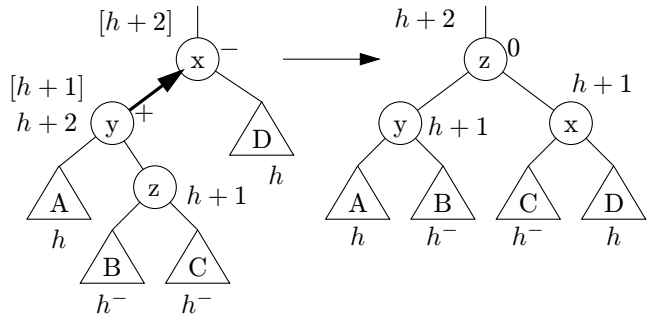
  rozebereme na tři případy podle znaménka vrcholu, ze kterého přišla informace o prohloubení:
  - Informace přišla *z vrcholu typu* $\ominus \rightarrow$ provedeme rotaci doprava tak, že novým kořenem se stane vrchol y, ze kterého přišla informace o prohloubení.



    Pozorování 1: znaménko vrcholů y a x je $\odot$

    Pozorování 2: hloubka před vkládáním byla $h + 1$ a nyní je také $h + 1$, tedy nemusíme dále posílat informaci o prohloubení a můžeme skončit
  - Informace přišla *z vrcholu typu* $\oplus$
    - uvažme ještě vrchol z jako pravého syna vrcholu y, ze kterého prišla informace o prohloubení, a jeho podstromy B a C

- vrcholy B a C mají hloubku h nebo $h-1 \rightarrow$ označme ji tedy $h-$ (to zřejmě protože vrchol y má znaménko $\oplus$, tedy jeho pravý podstrom s kořenem z má hloubku $h+1$)
- provedeme dvojrotaci tak, že novým kořenem se stane vrchol z



Pozorování 1: znaménko vrcholu z bude $\odot$

Pozorování 2: znaménka vrcholu x a y se dopočítají v závislosti na hloubce B a C

Pozorování 3: rozdíl hloubky pravého a levého podstromu bude u těchto vrcholů 0 nebo 1

Pozorování 4: hloubka před vkládáním byla $h+2$ a nyní je také $h+2$, tedy nemusíme dále posílat informaci o prohloubení a můžeme skončit

- informace přišla *z vrcholu typu* $\odot$ – to nemůže nastat, protože v tom případě by nešlo o prohloubení

Delete – odebrání vrcholu z AVL stromu Buď mažeme list nebo mažeme vrchol, který měl nějaké syny.

- pokud mažeme list, podíváme se na typ otce. Předpokládáme mazání levého syna.
  - byl typu $\ominus$ (neměl pravého syna) $\rightarrow$ změní se na $\odot$ (vrchol teď nemá žádné syny)
  - byl typu $\odot$ (měl oba syny) $\rightarrow$ změní se na $\oplus$

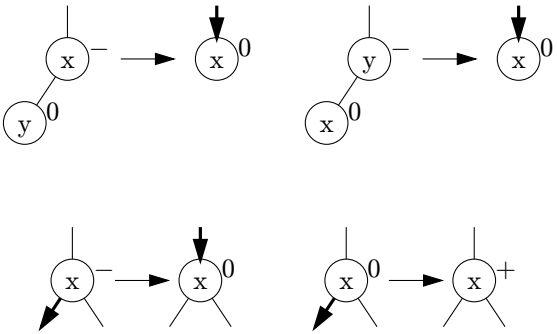
  (mažeme-li pravý list, řešíme zrcadlově)
- mažeme vrchol s jedním (levým nebo pravým) synem $\rightarrow$ syn nastupuje na místo otce a získává typ $\odot$

  V obou případech posílame informaci o změně hloubky stromu…
- mazaný vrchol měl oba syny (listy) $\rightarrow$ vybereme jednoho ze synů na místo smazaného otce. Hloubka se nemění.

- mazaný vrchol měl syny podstromy → na jeho místo vezmeme největší prvek levého podstromu (nebo nejmenší prvek pravého podstromu) a od odebraného (nahrazujícího) listu kontrolujeme vyvážení podstromu.

Úprava vyvážení stromu po odebrání listu z podstromu

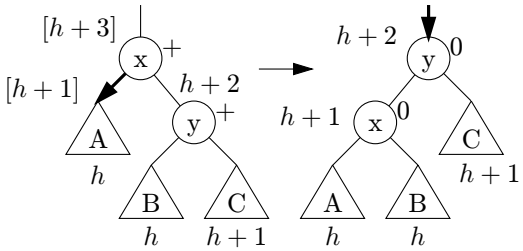
- informace o změně hloubky přišla z levého podstromu do vrcholu typu $\odot$ → vrchol se změní na $\oplus$ a dál se hloubka nemění
- informace přišla zleva do vrcholu s $\ominus$ → mění se na $\odot$ a posíláme informaci o změně hloubky.



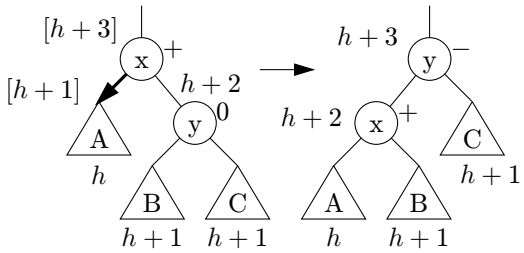
- problémová situace nastává, když informace o změně přišla zleva do vrcholu se znaménkem $\oplus$

Rozebereme na tři případy podle znaménka pravého syna nevyváženého vrcholu

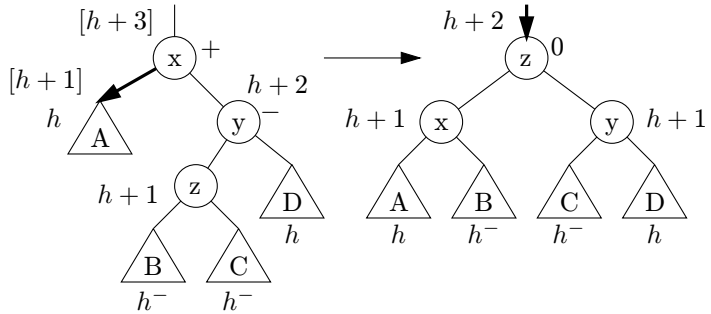
- *pravý syn je typu* $\oplus$ → provedeme rotaci vlevo, novým kořenem se stává y (pravý syn), oba vrcholy změní typ na $\odot$ a posíláme informaci o změně hloubky.



- *pravý syn je typu* $\odot$ → provedeme opět rotaci vlevo, kořenem se stává y, následně se u y změní typ na $\ominus$, u vrcholu x se typ nemění. Hloubka stromu se nemění, tudíž není třeba posílat informaci.



- *pravý syn je typu* $\ominus \rightarrow$ v tomto případě uvažujeme ještě vrchol z jako levého syna vrcholu y, s podstromy B a C, podstromy B a C mají hloubku h nebo $h-1$. Provedeme dvojrotaci, napřed vpravo rotujeme vrcholy z a y, potom vlevo vrcholy x a z tak, že se z stane novým kořenem, typ vecholu x bude potom $\ominus$ nebo $\odot$, typ y $\oplus$ nebo $\odot$ (podle toho, jaké znaménko měl původně vrchol z), typ z bude $\odot$ a opět posíláme informaci o změně hloubky stromu.



Obecné vyhledávací stromy

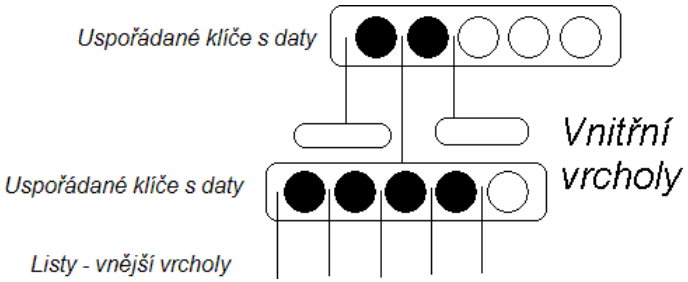
Při uložení dat na disku se snažíme, aby se čtení z disku provádělo pokud možno co nejméněkrát a nezáleží nám tolik na tom, kolik operací se vykoná v jednom uzlu. (Časově je operace porovnávání zanedbatelná oproti čtení z disku.)

Hodilo by se nám tedy mít uzly velké třeba tak, jako je velká jedna stránka cache ...

Definice: (a,b)-*strom* pro parametry a, b, $a \geq 2$, $b \geq 2a-1$ je zakořeněný strom s uspořádanými syny a vnějšími vrcholy, pro který platí následující axiomy:

1) Data jsou uložena ve vnitřních vrcholech a každý vrchol obsahuje o 1 méně klíčů než má synů.
2) Platí stromové uspořádání, tedy že $A < x_1 < B < x_2 < C < x_3 < D$.
3) Kořen má 2 až b synů, ostatní vnitřní vrcholy a až b synů.
4) Všechny vnější vrcholy jsou ve stejné hloubce (vnější vrchol=list).

Poznámka: kdekoli by mohl být syn a není, připojíme vrchol, kterému říkáme vnější vrchol. Můžeme si to představit třeba jako NULL-pointer.



Lemma: (a, b)-strom na n vrcholech má hloubku $O(\log_a n)$.
Důkaz: Zjistíme jeho minimální počet listů (označme jej m): každý vrchol až na kořen má alespoň a synů, hloubku si označíme $d \rightarrow$

$$m \geq a^{(d-1)}$$

$$\log_a m \geq d - 1$$

$$d \leq 1 + \log_a m$$

což je řádově $O(\log_a n)$, kde n je počet vrcholů.

Operace s (a,b)-stromy:

Find

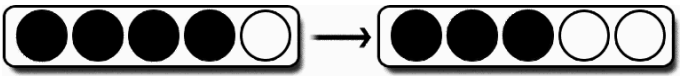
- Vždy zjistíme, mezi které 2 klíče hledaný vrchol patří, a potom se zanoříme hlouběji.

Časová složitost nalezení prvku v (a, b)-stromu je $O(\log b \cdot \log_a n)$, kde $\log b$ je čas strávený na jednom vrcholu pro zjištení, mezi které 2 vrcholy hledaný patří, $\log_a n$ je hloubka stromu.

Insert

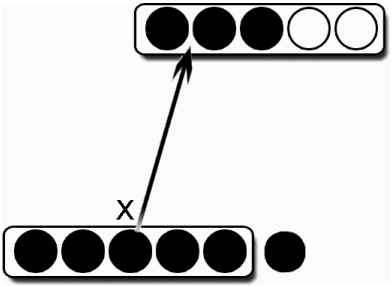
Jako Find, přičemž jestliže nenašel, skončí na posledním patře a přidáme klíč

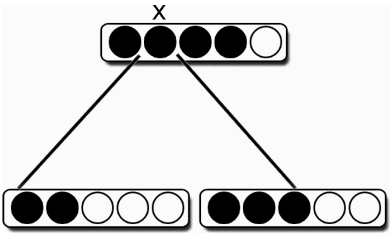
- pokud přidáním nepřesáhneme maximální počet klíčů, můžeme skončit



- pokud přidáním přesáhneme maximální počet klíčů

1. rozdělíme vrchol na 3 části: L,x,P
2. L a P jsou nové vrcholy
3. x je hodnota mezi L a P, kterou vložíme o patro výš jako klíč oddělující nově vzniklé vrcholy L a P
4. tím jsme převedli problém o patro výš a opakujeme algoritmus





Poznámka: Jestliže se dostaneme až do kořene, rozdělí se kořen na dvě části, vznikne nám nový kořen se dvěma syny (což je povoleno právě kvůli tomuto případu) a celému stromu vzroste hloubka o jedna.

Korektnost: Potřebujeme, aby

$$|L| \geq a - 1$$
$$|P| \geq a - 1$$

po sečtení obou nerovností a pričtení 1 na obě strany rovnice:

$$|L| + |P| + 1 \geq 2a - 2 + 1 = 2a - 1$$

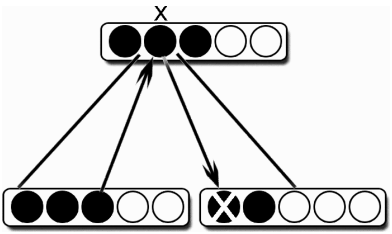
pravá strana je rovna b a to podle definice $\geq 2a - 1$.

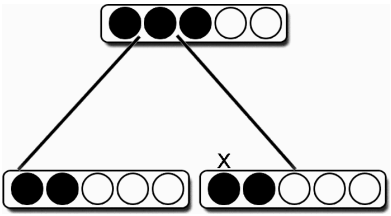
Časová složitost: vkládání prvku do (a, b)-stromu je $O(b \cdot \log_a n)$.

Delete

- převedeme na delete z listu (stejný postup jako u binárního stromu: jestliže to není list, prohodíme tuto hodnotu s nejmenší hodnotou podstromu jeho pravého syna) – v tomto případě na klíč posledního vnitřního vrcholu, protože listy jsou vnější vrcholy bez dat.
  - pokud má vrchol, ze kterého odebíráme, stále $a - 1$ klíčů, můžeme skončit
  - pokud má vrchol V, ze kterého odebíráme, $a - 2$ klíčů a jeho levý sousední vrchol L alespoň a klíčů (klíč otce oddělující tyto vrcholy označme x):

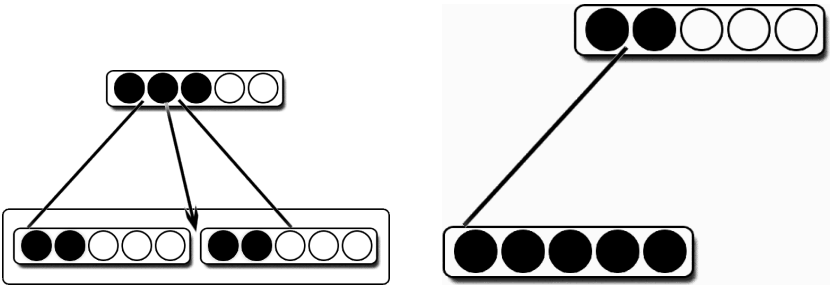
  1. smažeme největší klíč levého sousedního vrcholu L a nahradíme tím klíč otce obou vrcholů (nahradíme x za tuto hodnotu)
  2. původní klíč otce (x) přidáme jako nejmenší klíč odebíranému vrcholu V
  3. tím mají oba tyto vrcholy $a - 1$ klíčů a můžeme skončit





- pokud má vrchol V, z kterého odebíráme, $a-2$ klíčů a jeho levý sousední vrchol L má $a-1$ klíčů (klíč otce oddělující tyto vrcholy označme x):

1. sloučíme V,x,L do jednoho vrcholu
2. tím jsme problém převedli o patro výš a opakujeme algoritmus



Poznámka: Dojdeme-li takto až do kořene, na místo klíče odebraného z kořene lze použít nejmenší nebo největší klíč nově sloučeného podstromu. Ten odebrat lze, protože po sloučení (které bylo příčinou této situace), je v nejnižším vrcholu $2a-2$ klíčů.
Časová složitost:

$$O(b \cdot \log_a n)$$

11. Vyhledávací stromy a hashování

zapsal P. Taufer

FIXME: Zatím chybí červeno-černé stromy a trie.

Hashovaní

Mějme universum U (jeho velikost označíme u), množinu P přihrádek ($p = |P|$) a nějakou funkci $h : U \to P$, které budeme říkat *hashovací funkce.*

Datová struktura bude fungovat takto: když prvek vkládáme, spočteme hashovací funkci a vložíme prvek do příslušné přihrádky (přihrádky budeme reprezentovat jako pole seznamů). Pokud chceme prvek vyhledat nebo smazat, opět vyhodnotíme hashovací funkci a dozvíme se, ve které jediné přihrádce ho dává smysl hledat.

Budeme-li předpokládat, že výpočet funkce h trvá $\mathcal{O}(1)$, bude vkládání pracovat v konstantním čase a ostatní operace v čase lineárním s počtem prvků v dané přihrádce. Pokud se hashovací funkce bude chovat „rozumně náhodně“, můžeme očekávat, že po vložení n prvků jich bude v každé přihrádce přibližně n/p, takže při volbě $p \approx n$ můžeme získat konstantní časovou složitost operací. (Volit $p \gg n$ nemá smysl, protože pak bychom inicializací pole trávili příliš mnoho času.)

Tento přístup má ale samozřejmě své zádrhele: potřebujeme s prvky universa umět počítat (už si nevystačíme s porovnáváním), ale hlavně potřebujeme sehnat hashovací funkci, která se chová dostatečně rovnoměrně. Často se používají funkce, které se pro obvyklé vstupy chovají „pseudonáhodně“, třeba:

- $x \mapsto ax \bmod p$, pokud je universum číselné (pro nějakou konstantu a; nejlepší je, když a i p jsou prvočísla);
- $x_1, \ldots, x_n \mapsto \left(\sum_i C^i x_i\right) \bmod p$, pokud hashujeme řetězce (C a p opět nejlépe prvočíselná, navíc je-li ℓ obvyklá délka řetězce, mělo by být $C^\ell \gg p$).

Nicméně, ať už zvolíme jakoukoliv deterministickou funkci, vždy budou existovat nepříjemné vstupy, pro které skončí všechny prvky v téže přihrádce a operace budou mít lineární složitost namísto konstantní. Pomůžeme si snadno: vybereme hashovací funkci náhodně. Ne ze všech funkcí (ty bychom neuměli reprezentovat), nýbrž z vhodně zvolené třídy funkcí, které umíme snadno popisovat pomocí parametrů.

Definice: Systém funkcí S z U do P nazveme c-universální (pro nějaké $c \geq 1$), pokud pro všechny dvojice x, y navzájem různých prvků z U platí

$$\Pr_{h \in S}[h(x) = h(y)] \leq c/p.$$

(Kdybychom volili náhodně z úplně všech funkcí, vyšla by tato pravděpodobnost právě $1/p$ – c-universální systém je tedy nejvýše c-krát horší.)

Lemma: Buď h funkce náhodně vybraná z nějakého c-universálního systému. Nechť $x_1, \ldots, x_n$ jsou navzájem různé prvky universa vložené do struktury a x je nějaký prvek universa. Potom pro očekavaný počet prvků v téže přihrádce jako x platí:

$$\mathbb{E}[\#x : h(x) = h(x_i)] \leq cn/p.$$

Důkaz: Pro dané x definujeme indikátorové náhodné proměnné:

$$Z_i = \begin{cases} 1 & \text{když } h(x) = h(x_i) \\ 0 & \text{jinak} \end{cases}$$

Jinými slovy, Z_i říká, kolikrát padl prvek x_i do přihrádky $h(x)$, což je buď 0 nebo 1. Proto $Z = \sum_i Z_i$ a díky linearitě střední hodnoty je hledaná hodnota $\mathbb{E}[Z]$ rovna $\sum_i \mathbb{E}[Z_i]$. Přitom $\mathbb{E}[Z_i] = \Pr[Z_i = 1] \leq c/p$ podle definice c-universálního systému. Takže $\mathbb{E}[Z] \leq cn/p$. ♡

FIXME: Doplnit přehashovávání.

Zbývá dořešit, kde nějaký c-universální systém sehnat. Známých konstrukcí je vícero, zde si předvedeme jednu lineárně algebraickou.

Lemma: Předpokládejme, že p je prvočíslo, přihrádky jsou identifikované prvky konečného tělesa $\mathbb{Z}_p$ a universum U je vektorový prostor dimense d nad tělesem $\mathbb{Z}_p$, tedy $\mathbb{Z}_p^d$. Uvažujme systém funkcí $S = \{h_t \mid t \in \mathbb{Z}_p^d\}$, kde $h_t(x) := t \cdot x$ (skalární součin s vektorem s). Pak tento systém je 1-universální.

Důkaz: Nechť $x, y \in \mathbb{Z}_p^d$, $x \neq y$. Potom jistě existuje i, pro nějž $x_i \neq y_i$; bez újmy na obecnosti předpokládajme, že $i = d$. Nyní volíme t náhodně po složkách a počítáme pravděpodobnost kolize (rovnost modulo p značíme $\equiv$):

$$\Pr_{t \in \mathbb{Z}_p^d}[h_t(x) \equiv h_t(y)] = \Pr[x \cdot t \equiv y \cdot t] = \Pr[(x - y)t \equiv 0] =$$

$$= \Pr\left[\sum_{i=1}^{d} (x_i - y_i)t_i \equiv 0\right] = \Pr\left[(x_d - y_d)t_d \equiv -\sum_{i=1}^{d-1} (x_i - y_i)t_i\right].$$

Pokud už jsme $t_1, \ldots, t_{d-1}$ zvolili a nyní náhodně volíme t_d, nastane kolize pro právě jednu volbu (poslední výraz je linearní rovnice tvaru $ax = b$ pro nenulové a a ta má v libovolném tělese právě jedno řešení). Pravděpodobnost kolize je tedy nejvýše $1/p$, jak požaduje 1-universalita. ♡

Věta (Bertandův postulát): Pro libovolné $n \geq 1$ existuje prvočíslo p, které splňuje nerovnost $n < p \leq 2n$.