Státnicový opakovací výcuc I1

Zapsal Michal Tuláček. O mnoho vysvětlivek se zasloužil Martin Dekar.

Kapitola 1
Složitost

1.1 Věty o hierarchii tříd složitosti, konstruovatelné funkce, vztahy mezi časovými a prostorovými mírami a determinismem a nedeterminismem, Savitchova věta

1.1.1 Věty o hierarchii tříd složitosti

V **Věta 1.1** (***O deterministické prostorové hierarchii***): *$S_1 : \mathbb{N} \to \mathbb{N}$ a $S_2 : \mathbb{N} \to \mathbb{N}$ jsou funkce takové, že $S_1 \in o(S_2)$, S_2 je prostorově konstruovatelná a $S_1(n) \geq \log_2(n)$. Potom existuje jazyk L, pro který platí $L \in \mathbf{DSPACE}(S_2(n)) \setminus \mathbf{DSPACE}(S_1(n))$.*

Důkaz: Snažíme se zkonstruovat spor vyčíslitelnického typu, kdy dojdeme k tomu, že nějaký jazyk do stejné třídy patří a nepatří zároveň.

Mějme univerzální DTS M, který pracuje v prostoru $S_2(n)$ a s každým DTS pracujícím v prostoru $S_1(n)$ se neshoduje alespoň na 1 vstupu (diagonalizace).

Vyrobíme prefixová očíslování w všech jednopáskových strojů nad abecedou $\{0, 1\}$ (prefixové očíslování gödelova čísla x vypadá tak, že před něj nalepíme libovolně dlouhý řetězec jedniček, oddělený od x nulou, takže např. $111111111110x$. Výhodou je, že ho můžeme libovolně natahovat a přitom kóduje stejný stroj).

UDTS M pracuje na vstupu $w \in \{0, 1\}, n = |w|$ takto:

1) Označí $S_2(n)$ buněk na pracovní pásce ($S_2(n)$ je prostorově konstruovatelná). Pokud v další práci M šlápneme mimo, odmítneme w.
2) UDTS M simuluje M_w (tedy DTS kódované gödelovým číslem skrytým v w) nad vstupem w *(známe z vyčíslitelnosti, stroj puštěný nad svým zdrojákem)* a přijme jen tehdy když:

   (i) Simulace proběhne v prostoru $S_2(n)$ a
   (ii) M_w **zastaví** a odmítne w.

Z toho nám pak pro jazyk $L, L = L(M)$ plyne:

A) $L \in \mathbf{DSPACE}(S_2(n))$, což plyne z toho, že M nám běželo dle bodu 1 v prostoru $S_2(n)$, jinak odmítlo vstup.
B) $L \notin \mathbf{DSPACE}(S_1(n))$, což lze ukázat sporem:

Pro spor, nechť $L \in \mathbf{DSPACE}(S_1(n))$. Pak existuje DTS N s t páskovými symboly, pracující v prostoru $S_1(n)$: $L(N) = L = L(M)$. N zastaví pro každý vstup a je jednopáskový. To že zastaví uděláme tak, že si odpočítáme maximální počet kroků, odpovídající počtu možných konfigurací a zastavíme když kroky došly (počet konfigurací k odpočítání lze zapsat v $S_1(n)$, tedy nám počitadlo nezvedne prostorovou složitost). Toho že je jednopáskový dosáhneme pomocí věty o redukci pásek.

Máme tedy jednopáskový DTS N s pracovní abecedou t, který vždy zastaví. Protože N chceme simulovat v M, které má abecedu $\{0, 1\}$, změníme N na N' s abecedou $\{0, 1\}$, což nám prodlouží prostor na $\lceil \log_2 t \rceil \cdot S_1(n)$.

Mějme prefixový kód x takový, že $N' = M_x$. Teď potřebujeme, aby se to celé vešlo do $S_2(n)$, takže budeme protahovat prefix x tak, aby platilo ($n = |x|$): $\lceil \log_2 t \rceil S_1(n) \leq S_2(n)$ *(musíme to natahovat, protože pro malá n může být $\lceil \log_2 t \rceil S_1(n)$ větší než $S_2(n)$)*. Takové n existuje, protože $S_1 \in o(S_2)$.

A nyní:

- Pokud N' přijme x, tedy $x \in L(N') = L(M)$. Dle definice M pak ale $x \notin L(M)$.
- Pokud N' odmítne x, tedy $x \notin L(N') = L(M)$. Dle definice M pak ale $x \in L(M)$.

Oba případy vedou ke sporu, a tedy platí, že $L \notin \mathbf{DSPACE}(S_1(n))$. ♡

Věta 1.2 (***O deterministické časové hierarchii***): *$T_1 : \mathbb{N} \to \mathbb{N}$ a $T_2 : \mathbb{N} \to \mathbb{N}$ jsou funkce takové, že $T_1 \log_2 T_1 \in o(T_2)$ a T_2 je časově konstruovatelná. Potom existuje jazyk L, pro který platí $L \in$ **DTIME**$(T_2(n)) \setminus$ **DTIME**$(T_1(n))$.* **V**

Důkaz: Důkaz je podobný důkazu věty o deterministické prostorové hierarchii 1.1. Také si stvoříme nějaký DTS, na kterém ukážeme vyčíslitelnický spor.

Cílem důkazu je zkonstruovat DTS pracující v čase $T_2(n)$, který se od každého stroje pracujícího v čase $T_1(n)$ liší alesoň na jednom vstupu.

Mějme tedy univerzální DTS M a prefixovým kódem označené všechny dvoupáskové stroje. Pro prefixový kód w, kde $|w| = n$, vypadá práce takto:

1) Na dvou páskách stroj M simuluje práci stroje M_w (tedy stroje, s prefixovým kódem w) se vstupem w (tedy opět sám na sobě).
2) Protože je T_2 časově konstruovatelná, tak existuje nějaký DTS, který ji počítá. Jeho simulaci pustíme na dalších pracovních páskách a bude trvat přesně $T_2(n)$.
3) Pokud stále simulace běží po $T_2(n)$ krocích, vstup odmítneme.
4) Pokud simulace skončí odmítnutím dříve než po $T_2(n)$ krocích tím, že M_w odmítne vstup w, pak M vstup w přijme.

Označme $L = L(M)$. Platí, že $L \in$ **DTIME**$(T_2(n))$, protože M zastaví v čase $T_2(n)$ pro všechna $w \in L$. Zbývá ukázat, že $L \notin$ **DTIME**$(T_1(n))$.

Pro spor, nechť existuje DTS N, který přijímá L v čase $T_1(n)$. Potom dle věty o redukci počtu pásek (logaritmická verze) existuje stroj N', který má dvě pásky a přijímá L v čase $T_1(n) \log_2(T_1(n))$. Navíc, nechť N' má v páskové abecedě t symbolů. Pak bude M potřebovat na simulaci N' čas $c(t)T_1(n) \log_2(T_1(n))$, kde $c(t)$ je konstanta závislá na t.

Mějme prefixový kód x pro stroj N'. Potřebujeme, aby x bylo tak dlouhé, aby platilo $c \cdot T_1(|x|) \cdot \log(T_1(|x|)) \leq T_2(|x|)$. Z předpokladů věty platí, že taková délka prefixu existuje.

Toto jsme dělali proto, abychom M zajistili dost času na simulaci N'. Mohou nastat dvě situace:

- Pokud N', tedy M_x, přijme x, pak $x \in L = L(N') = L(M)$. Ale v takovém případě M odmítne x, tedy $x \notin L(M)$.
- Pokud N' odmítne x, pak $x \notin L = L(N') = L(M)$. V takovém případě ale M přijme x, tedy $x \in L(M)$.

Oba případy vedou na spor, a tedy $L \notin$ **DTIME**$(T_1(n))$, čímž je věta dokázaná. ♡

1.1.2 Konstruovatelné funkce

Definice 1.3 (***Rekurzivní funkce***): *$f : \mathbb{N} \to \mathbb{N}$ je rekurzivní $\Leftrightarrow \exists$ DTS tranducer s jednoprvkovou vstupní abecedou takový, že pro vstup 1^n dává výstup $1^{f(n)}$ znaků na pásce.* □ **D**

Definice 1.4 (***Funkce vyčíslitelná v lineárním čase***): *$f : \mathbb{N} \to \mathbb{N}$ je vyčíslitelná v lineárním čase $\Leftrightarrow$ f je rekurzivní a $\exists$ konstanta c a DTS transducer s jednoprvkovou vstupní abecedou takový, že na vstupu 1^n udělá nejvýše $c \cdot f(n)$ kroků, než vydá výstup $1^{f(n)}$.* □ **D**

Definice 1.5 (***Funkce vyčíslitelná v lineárním prostoru***): *$f : \mathbb{N} \to \mathbb{N}$ je vyčíslitelná v lineárním prostoru $\Leftrightarrow$ f je rekurzivní a $\exists$ konstanta c a DTS transducer s jednoprvkovou vstupní abecedou takový, že na vstupu 1^n použije nejvýše $c \cdot f(n)$ buněk na pracovních páskách, než vydá výstup $1^{f(n)}$.* □ **D**

Definice 1.6 (***Časově konstruovatelná funkce***): *$f : \mathbb{N} \to \mathbb{N}$ je časově konstruovatelná funkce $\Leftrightarrow \exists$ DTS takový, že pro každý vstup délky n zastaví po právě $f(n)$ krocích.* □ **D**

Definice 1.7 (***Prostorově konstruovatelná funkce***): *$f : \mathbb{N} \to \mathbb{N}$ je prostorově konstruovatelná funkce $\Leftrightarrow \exists$ DTS takový, že pro každý vstup délky n zastaví po použití právě $f(n)$ buněk.* □ **D**

Lemma 1.8: *$f_1 + f_2$ a f_2 časově konstruovatelné a nechť $\exists \varepsilon > 0 \exists n_0 \forall n > n_0 : f_1(n) \geq \varepsilon f_2(n) + (1 + \varepsilon)n$. Pak f_1 je časově konstruovatelná.*

V **Věta 1.9**: $f:\mathbb{N} \to \mathbb{N}, \exists \varepsilon > 0 \exists n_n \forall n \geq n_0 : f(n) \geq (1+\varepsilon)n$. *Pak platí: f časově konstruovatelná $\Leftrightarrow$ f je vyčíslitelná v lineárním čase.*

Důkaz: "$\Rightarrow$": triviální

"$\Leftarrow$": DTS M vyčísluje f v $O(n)$. Označme $g(n)$ počet kroků stroje M nad vstupem 1^n. Tedy $\exists c > 0 : g(n) \leq c \cdot f(n)$.

1) g je časově konstruovatelná (máme příslušný DTS M).
2) $g + f$ je časově konstruovatelná (simuluji M a pak odjedu na začátek výstupu).
3) Tedy z lemmatu 1.8 plyne, že f je časově konstruovatelná, pokud platí předpoklady.

Definujme:

$$\begin{array}{lcl} \varepsilon_1 & = & \text{z předpokladu} \\ \varepsilon_2 & & \text{tak, aby } (1+\varepsilon_1)(1-\varepsilon_2) > 1 \\ \varepsilon_3 & = & (1+\varepsilon_1)(1-\varepsilon_2) - 1 \\ \varepsilon_4 & = & \min\left\{\frac{\varepsilon_2}{c}, \varepsilon_3\right\} \end{array}$$

$f(n) = (\varepsilon_2 - \varepsilon_2)f(n) + f(n) = \varepsilon_2 f(n) + (1-\varepsilon_2)f(n) \geq \frac{\varepsilon_2}{c} g(n) + (1-\varepsilon_2)(1+\varepsilon_1)n = \frac{\varepsilon_2}{c} g(n) + (1-1+(1+\varepsilon_1)(1-\varepsilon_2))n = \frac{\varepsilon_2}{c} g(n) + (1+\varepsilon_3)n \geq \varepsilon_4 g(n) + (1+\varepsilon_4)n$.

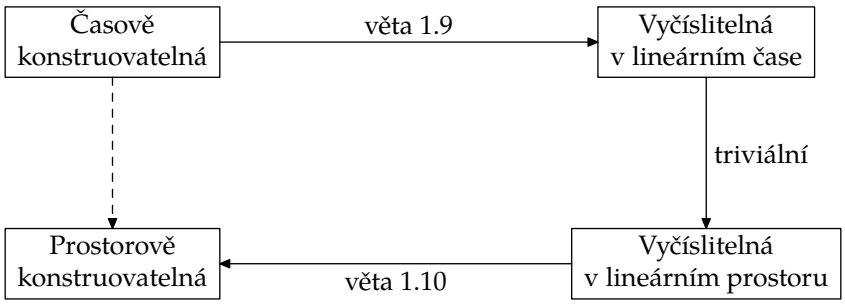
Tedy předpoklady věty platí. ♡

V **Věta 1.10**: $f:\mathbb{N} \to \mathbb{N}$. *Pak platí: f prostorově konstruovatelná $\Leftrightarrow$ f je vyčíslitelná v lineárním prostoru.*

Důkaz: "$\Rightarrow$": triviální

"$\Leftarrow$": M je DTS s k páskami, vyčíslující $f(n)$ v prostoru $c \cdot f(n)$. Dle věty o linerální prostorové kompresi zkonstruuji k-páskový stroj M', vyčíslující $f(n)$ v prostoru $f(n)$. M' dále převedu podle věty o redukci pásek na jednopáskový stroj M'', který dokazuje prostorovou konstruovatelnost $f(n)$. ♡

Důsledek 1.11: *Každá časově konstruovatelná funkce, splňující podmínku $\exists \varepsilon > 0 \exists n_n \forall n \geq n_0 : f(n) \geq (1+\varepsilon)n$, je také prostorově konstruovatelná.*



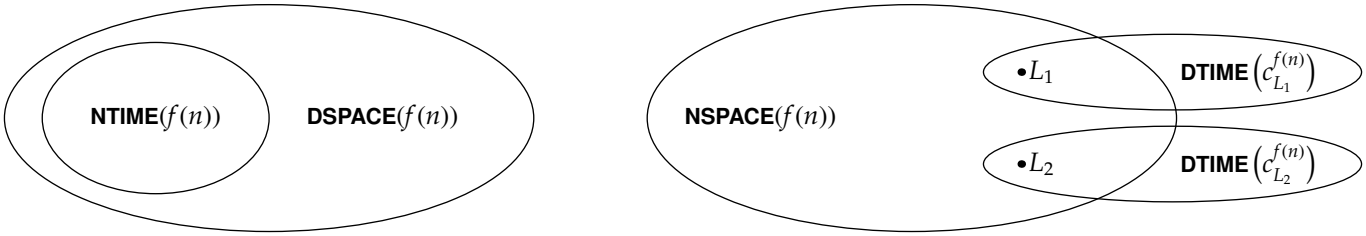
Obrázek 1.1.1. Vztah časové a prostorové konstruovatelnosti.

1.1.3 Vztahy mezi časovými a prostorovými mírami a determinismem a nedeterminismem

V **Věta 1.12** (***Věta o vztazích***): $f:\mathbb{N} \to \mathbb{N}$, *je funkce. Potom platí:*

a) $\mathbf{NTIME}(f(n)) \subseteq \mathbf{DSPACE}(f(n))$,

b) $L \in \mathbf{NSPACE}(f(n)), f(n) \geq \log_2(n) \Rightarrow \exists c_L : L \in \mathbf{DTIME}\left(c_L^{f(n)}\right)$.



Obrázek 1.1.2. Věta o vztazích.

Důkaz: a) Idea: brute-force vyrobím hádací pásku.

Mějme NTS s k páskami. Sestrojíme DTS s $k+1$ páskami, který na přebytečnou pásku lexikograficky generuje všechny možné kódy výpočtů (např. od $111\ldots111$ po $rrr\ldots rrr$, kde r je maximální arita větvení) a simuluje příslušný běh NTS. Pokud přijme náš DTS, odpovídá to přijetí původního NTS. Pokud nepřijme, počítáme dál.

b) Idea důkazu: uděláme si graf konfigurací a spustíme nad ním DFS, které zjistí, zda graf obsahuje cestu k přijímacímu stavu a celé to stihneme v $T(n) = c_L^{f(n)}$.

$L \in$ **NSPACE**$(f(n))$, tedy máme NTS M rozhodující L v prostoru $f(n)$. Nechť je búno jednopáskový. Pokud existuje přijímající výpočet, existuje přijímající výpočet bez cyklů. Acyklický výpočet trvá nejvýše „počet konfigurací“ kroků, $\leq d_L^{f(n)}$, pro nějakou konstantu d (zde vyvstává potřeba pro $f(n) \geq \log_2(n)$ z předpokladů).

Uvažujme graf G všech konfigurací. Hrana (a, b) znamená, že konfigurace b je z konfigurace a dosažitelná v jednom kroku. Počet vrcholů a konfigurací je omezený:

$$|V(G)| \leq d^{f(n)}$$

$$|E(G)| \leq c \cdot d^{f(n)}$$

Omezení počtu hran plyne z faktu, že změny v hlavní konfiguraci po jednom kroku jsou pouze lokální, tedy každá konfigurace má pouze konstantní počet sousedů.

Na pracovní pásku vygenerujeme seznam všech konfigurací, což zabere prostor $O(d^{f(n)} \cdot f(n))$, (jedna konfigurace má délku $f(n)$). Konfigurace lze zapsat v lineárním čase vzhledem k jejich délce.

Deterministický stroj pak prochází G do hloubky a na pomocnou pásku poznamenává, co navštívil. Na zjištění, zda už vrchol navštívil potřebuje čas úměrný dosavadnímu počtu navštívených vrcholů (tedy tomu, co má na té pomocné pásce, kterou tvoří). Stroj tedy pracuje v čase

$$T(n) = |V(G)| \cdot O(f(n)) + \left|E(G) O\left(d^{f(n)} \cdot f(n)\right)\right| \leq d^{f(n)} \cdot O(f(n)) + k \cdot d^{f(n)} \cdot O\left(d^{f(n)} \cdot f(n)\right) \leq O\left(\left(d^{f(n)}\right)^2 \cdot f(n)\right) \leq O\left(c^{f(n)}\right)$$

Najde-li stroj při průchodu grafem přijímající konfiguraci, je tato dosažitelná z výchozího stavu a tedy vstup přijmeme v čase $T(n) \leq O\left(c^{f(n)}\right)$, a tedy $L \in$ **DTIME** $\left(c_L^{f(n)}\right)$. ♡

1.1.4 Savitchova věta

Věta 1.13 (***Savitchova věta***): *$S : \mathbb{N} \to \mathbb{N}$ je prostorově konstruovatelná funkce, taková, že $S(n) \geq \log_2(n)$.* V
Potom **NSPACE**$(S(n))$ = **DSPACE** $\left((S(n))^2\right)$

Důsledek 1.14: **NPSPACE** = **PSPACE**.

1.2 Úplné problémy pro různé třídy (NP, PSPACE, P, #P)

D **Definice 1.15** (***C-těžkost***): *Jazyk L je* **C***-těžký (***C***–hard), pokud je každý jazyk $L' \in$* **C** *převoditelný na L.* □

D **Definice 1.16** (***C-úplnost***): *Jazyk L je* **C***-úplný (***C***–c), pokud je L* **C***-těžký a $L \in$* **C**. □

D **Definice 1.17** (***Převoditelnost v polynomiálním čase***): *Jazyk A je převoditelný na jazyk B v polynomiálním čase, $A \propto B$, pokud existuje funkce vyčíslitelná v polynomiálním čase $f\colon \{0,1\}^* \to \{0,1\}^*$ a $x \in A$ právě když $f(x) \in B$ pro každé $x \in \{0,1\}^*$.* □

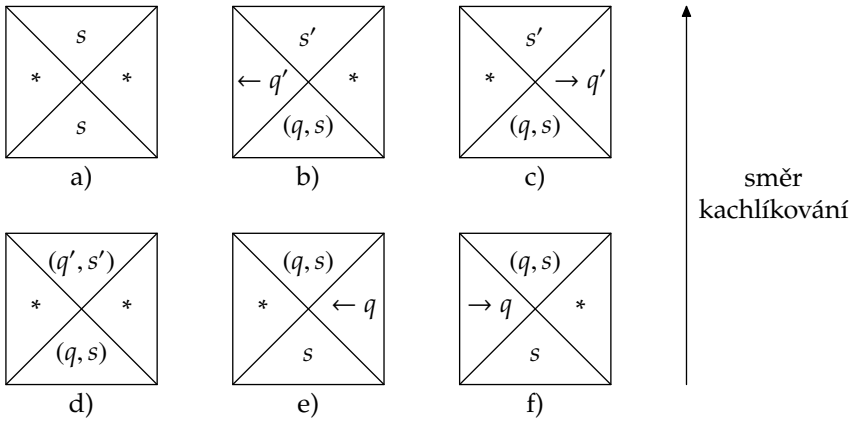
V **Věta 1.18** (***Cook-Lewinova***): *Existuje NP-úplný problém.*

Důkaz: Idea: Větu dokážeme tak, že na jazyk $\mathcal{KACHL}$ převedeme všechny jazyky z **NP**, čímž dokážeme, že $\mathcal{KACHL}$ je **NP**-těžký, a následně dokážeme, že $\mathcal{KACHL} \in$ **NP**, tedy $\mathcal{KACHL} \in$ **NP**–c.

Problém $\mathcal{KACHL}$ je definován takto: Mějme čtvercové kachlíky, které nelze otáčet, a které mohou mít každou stranu obarvenou jinou barvou, a dále mějme stěnu, kterou tvoří čtvercová síť, jejíž buňky jsou stejně velké jako kachlíky. Krajní buňky sítě jsou na vnější straně také obarvené různými barvami. Lze vymezený prostor vykachlíkovat tak, aby se kachlíky dotýkaly jiných kachlíků, nebo hranice stěny, hranami se stejnou barvou?

Mějme jazykk $L \in$ **NP**. Pak existuje turingův stroj $T = (Q, \Sigma, \delta, q_0, F)$, který L přijímá v polynomiálním čase $T(n) \leq p(n)$. Bez újmy na obecnosti, nechť má stroj pouze jednostrannou pásku (tedy má pásku nekonečnou doprava, ale konečnou doleva), obsahuje prázdný symbol Λ, který je na doposud nepoužitých buňkách pásky a přijímá jediným stavem q_F, poté co smaže pásku. Toho není problém dosáhnout, stačí pásku „přehnout" a v jedné buňce držet dvojici symbolů, jeden z levé a druhý z pravé části pásky. Navíc můžeme k libovolnému turingovu stroji přidat krátký kód, který v přijímacím stavu smaže pásku a přepne se do stavu q_F. Stroj T nyní převedeme na zadání problému $\mathcal{KACHL}$.

Každý řádek kachlíků představuje krok výpočtu stroje T a každý kachlík představuje políčko pásky. K tomu si připravíme sadu kachlíků šesti typů (viz obrázek, trojúhelníky na kachličkách představují příslušné barvy). Pro každý symbol $s \in \Sigma$ máme kachlík typu a), který představuje situaci, kdy hlava není nad symbolem s, a tak se symbol s propíše beze změny do dalšího kroku výpočtu. Kachlíky typu b), c) a d) odpovídají situaci, kdy je hlava stroje T nad příslušnou buňkou ve stavu q a na pásce je symbol s a přechodová funkce $\delta(q, s) = (q', s', \leftarrow)$ nebo $(q', s', \rightarrow)$ nebo $(q', s', \bullet)$. Kachlíky typu e) a f) pak představují situaci, kdy vpravo či vlevo od nich došlo k pohybu hlavy, a tedy v příštím kroku výpočtu bude hlava nad touto buňkou.



Obrázek 1.2.1. Kachlíky pro převod Turingova stroje na problém $\mathcal{KACHL}$.

Protože T běží v čase omezeném polynomem $p(n)$, je i počet buněk pracovní pásky omezitelný $p(n)$ (kdyby T nedělalo nic, než že by jelo doprava, tak by dojelo nejdál $p(n)$ daleko a pak by skončilo). Stačí nám tedy koupelna velikosti $p(n) \times p(n)$. Pro všechny kombinace stavů, symbolů a pohybů stroje T si připravíme kachlíky. Vstup stroje T nakreslíme na spodní stranu koupelny tak, že spodní hrana koupelny obsahuje výchozí stav pásky (barva reprezentující pouze písmeno s a nebo znak Λ). Hlava T je na začátku výpočtu na pásce vlevo, takže levou dolní buňku obarvíme barvou (s, q_0). Levá a pravá hrana koupelny

je obarvena symboly *, a konečně horní hrana má vlevo symbol (q_F, Λ) a zbytek horní hrany je pokryt symboly Λ.

Kachlíkováním koupelny není možné dosáhnout toho, že by na jednom řádku bylo více hlav, a nebo že by činnost hlavy neodpovídala přechodové funkci. Námi zadaná úloha tedy správně simuluje stroj T.

Takto zadaný problém má řešení, právě když stroj T přijímá L, tedy jsme převedli jazyk L na jazyk problému $\mathcal{KACHL}$, a tedy $\mathcal{KACHL} \in$ **NP**$-hard$. Certifikát lze ověřit v polynomiálním čase, a tedy $\mathcal{KACHL} \in$ **NP**. Dokázali jsme tedy, že $\mathcal{KACHL} \in$ **NP**$-c$. ♡

Definice 1.19 (L): **L** = **LOG** = **DSPACE**$(log(n))$ □ D

Definice 1.20 (***Vyčíslitelnost v logaritmickém prostoru***): *Funkce* $f: \{0,1\}^* \to \{0,1\}^*$ *je vyčíslitelná v logaritmickém prostoru, pokud její výstup je polynomiálně omezený (tedy* $\exists c: |f(x)| \leq |x|^c$ *pro každé* $x \in \{0,1\}^*$*) a jazyky* $L_f = \{\langle x, i\rangle | f(x)_i = 1\}$ *a* $L'_f = \{\langle x, i\rangle | i \leq |f(x)|\}$ *jsou v* **L**. □ D

Poznámka: *To znamená, že jsem schopen spočítat i-tý bit výstupu, a jeho velikost, v logaritmickém prostoru.*

Definice 1.21 (***Převoditelnost v logaritmickém prostoru***): *Jazyk A je převoditelný na jazyk B v logaritmickém prostoru,* $A \leq_l B$*, pokud existuje funkce vyčíslitelná v logaritmickém prostoru* $f: \{0,1\}^* \to \{0,1\}^*$ *a* $x \in A$ *právě když* $f(x) \in B$ *pro každé* $x \in \{0,1\}^*$. □ D

Poznámka (***Tranzitivita logspace převoditelnosti v logaritmickém čase***): *Operace* $\leq_l$ *je tranzitivní. Pokud počítáme* $a \leq_l c$*, což je* $a \leq_l b, b \leq_l c$*, tak se počítá nejprve to* $b \leq_l c$*, a když potřebujeme bit z b, tak se to dopočítá on-demand z a.*

Definice 1.22 (***Problém CIRCUIT-EVAL***): *Mějme jazyk obsahující všechny páry* $\langle C, x\rangle$*, kde C je obvod s n vstupy a jedním výstupem a* $x \in \{0,1\}^n$ *takové, že* $C(x) = 1$. □ D

Poznámka: *Obvod je síť hradel, které jsou navzájem nějak propojené, s tím že celá síť má na vstupu jeden vektor (n drátů) a jeden výstup (jeden drát). Vícevrstvá neuronová síť je jistě obvodem. Problém* $\mathcal{CIRCUIT\text{-}EVAL}$ *je pak jen o tom, pro daný vstup spočítat výstup sítě.*

Věta 1.23 (***P-úplný problém***): *Problém* $\mathcal{CIRCUIT\text{-}EVAL}$ *je* **P***-úplný problém.* V

Poznámka: *V* **P** *se mezi problémy převádí pomocí převoditelnosti v logaritmickém prostoru.*

1.3 Polynomiální hierarchie, pseudopolynomiální algoritmy, silná NP-úplnost, třída #P a #P-úplnost

1.3.1 Polynomiální hierarchie

Definice 1.24 (***DTS s orákulem***): *DTS M s orákulem A, kde A je jazyk, se liší od „obyčejného" DTS takto:* D

- *M má navíc tzv. dotazovací pracovní pásku (kde používá abecedu jazyka A). Prostor zabraný během výpočtu na této pásce se počítá do prostorové složitosti M.*
- *M má navíc tři stavy řídící jednotky:* $\mathcal{DOTAZ}$, $\mathcal{ANO}$ *a* $\mathcal{NE}$.
- *Pokud během práce M dojde k přechodu do stavu* $\mathcal{DOTAZ}$*, tak v následujícím kroku M buď přejde do stavu* $\mathcal{ANO}$*, pokud aktuální obsah dotazovací pásky je slovo z jazyka A, nebo přejde do stavu* $\mathcal{NE}$*, pokud aktuální obsah dotazovací pásky není slovo z jazyka A, a navíc v obou případech následně vymaže (naráz) celý obsah dotazovací pásky. Jazyk slov přijímaných DTS M s orákulem A značíme* $L(M, A)$.

□

Poznámka: *DTS bez orákula lze chápat jako DTS s prázdným orákulem (tj. s $A = \emptyset$).*

D **Definice 1.25** (***Turingovská převoditelnost***): *Nechť A a B jsou jazyky. Řekneme, že jazyk A je (deterministicky) Turingovsky převoditelný na jazyk B v polynomiálním čase, pokud existuje DTS M s orákulem pracující v polynomiálním čase takový, že $A = L(M, B)$. Toto značíme $A \leq_T B$.* □

Příklad 1.26: $A \in \mathbf{P} \Rightarrow A \leq_T \emptyset$.

D **Definice 1.27**: *Nechť B je jazyk.* $\mathbf{P}(B) = \{A | A \leq_T B\}$. *Nechť* $\mathbf{C}$ *je třída jazyků.* $\mathbf{P}(\mathbf{C}) = \{A | \exists B \in \mathbf{C}: A \leq_T B\}$. □

D **Definice 1.28** (***Nedeterministicky Turigovská převoditelnost***): *Nechť A a B jsou jazyky. Řekneme, že A je nedeterministicky Turingovsky převoditelný na B v polynomiálním čase, pokud existuje NTS M s orákulem pracující v polynomiálním čase takový, že $A = L(M, B)$. Tento fakt značíme $A \leq_{NP} B$.* □

D **Definice 1.29**: *Nechť B je jazyk.* $\mathbf{NP}(B) = \{A | A \leq_{NP} B\}$. *Nechť* $\mathbf{C}$ *je třída jazyků.* $\mathbf{NP}(\mathbf{C}) = \{A | \exists B \in \mathbf{C}: A \leq_{NP} B\}$. □

Poznámka: *A tak dále pro* **PSPACE**, **LOG**, *..*

D **Definice 1.30** (***Polynomiální hierarchie - zjednodušená verze***): *Definujme třídy jazyků $\Sigma_0, \Sigma_1, \Sigma_2, \ldots$ se startovní podmínkou $\Sigma_0 = \mathbf{P}$ a rekurzivním předpisem $\forall k \geq 0: \Sigma_{k+1} = \mathbf{NP}(\Sigma_k)$. Potom polynomiální hierarchií rozumíme třídu $\mathbf{PH} = \bigcup_{k \geq 0} \Sigma_k$.* □

D **Definice 1.31** (***co-A, co-C***): *Nechť A je jazyk nad abecedou Σ. Pak co-$A = \{x \in \Sigma^* | x \notin A\}$ je doplněk A. Nechť třída* $\mathbf{C}$ *je třída jazyků. Pak co-*$\mathbf{C} = \{co\text{-}A | A \in \mathbf{C}\}$ *je doplněk* $\mathbf{C}$. □

V **Věta 1.32** (***Vztah PH a PSPACE***): $\mathbf{PH} \subseteq \mathbf{PSPACE}$.

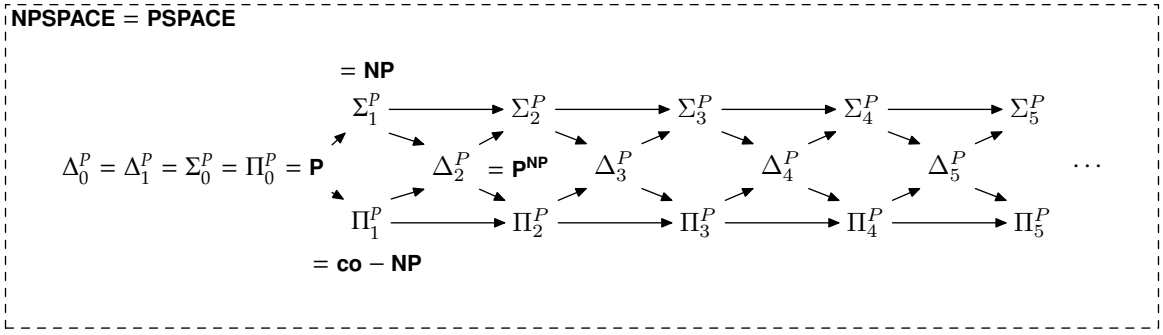
D **Definice 1.33** (***Polynomiální hierarchie***): *Polynomiální hierarchie sestává z Σ_k, Π_k a Δ_k pro $k \geq 0$, kde:*

1) $\Sigma_0 = \Pi_0 = \Delta_0 = \mathbf{P}$.
2) $\forall k \geq 0: \Sigma_{k+1} = \mathbf{NP}(\Sigma_k)$,
3) $\forall k \geq 0: \Pi k + 1 = co\text{-}\mathbf{NP}(\Sigma_k)$,
4) $\forall k \geq 0: \Delta k + 1 = \mathbf{P}(\Sigma_k)$.

□

V **Věta 1.34** (***Vztahy v polynomiální hierarchii***): *Následující vztahy platí pro $k \geq 0$:*

1) $\Sigma_1 = \mathbf{NP}$, $\Pi_1 = co\text{-}\mathbf{NP}$, $\Delta_1 = \mathbf{P}$.
2) $\Pi_k = co\text{-}\Sigma_k$ (a tedy $\Sigma_k = co\text{-}\Pi_k$).
3) $\Sigma_{k+1} = \mathbf{NP}(\Pi_k)$.
4) $\Delta_{k+1} = \mathbf{P}(\Pi_k)$.
5) $\Pi_{k+1} = co\text{-}\mathbf{NP}(\Pi_k)$.
6) $\Sigma_{k+1} = \mathbf{NP}(\Delta_{k+1})$.
7) $\Pi_{k+1} = co\text{-}\mathbf{NP}(\Delta_{k+1})$.
8) $\Delta_k = co\text{-}\Delta_k$.
9) $\Delta_k = \mathbf{P}(\Delta_k)$.
10) $\Sigma_k \cup \Pi_k \subseteq \Delta_{k+1}$.
11) $\Delta_k \subseteq \Sigma_k \cap \Pi_k$.
12) Pokud $\Sigma_k \subseteq \Pi_k$ (nebo $\Pi_k \subseteq \Sigma_k$), tak $\Sigma_k = \Pi_k$.



Obrázek 1.3.1. Polynomiální hierarchie. Šipky znázorňují inkluze.

Definice 1.35 (***Existenční kvantifikátor***): *Symbol* $\exists^{p(n)}$: $R(x)$ *znamená, že* $\exists x$: $(|x| \leq p(n)) \wedge (x$ *má vlastnost* $R)$. **D** □

Definice 1.36 (***Obecný kvantifikátor***): *Symbol* $\forall^{p(n)}$: $R(x)$ *znamená, že* $\forall x$: $(|x| \leq p(n)) \Rightarrow (x$ *má vlastnost* $R)$. **D** □

Definice 1.37 (***Třída jazyků existitko C***): *Nechť* **C** *je třída jazyků. Pak* $\exists$**C** *je třída jazyků, kde* $A \in \exists$**C**, *pokud platí* $\exists B \in$ **C**, $\exists$ *polynom* p: $x \in A \Leftrightarrow \exists^{p(|x|)} y$: $\langle x, y \rangle \in B$. **D** □

Poznámka: *Když se ta definice rozzipuje, tak říká:* $A \in \exists$**C**, *když* $\exists B \in$ **C**, $\exists$ *polynom* p: $x \in A \Leftrightarrow \exists y$: $(|y| \leq p(|x|)) \wedge \langle x, y \rangle \in B$

Definice 1.38 (***Třída jazyků všechnítko C***): *Nechť* **C** *je třída jazyků. Pak* $\forall$**C** *je třída jazyků, kde* $A \in \exists$**C**, *pokud platí* $\exists B \in$ **C**, $\exists$ *polynom* p: $x \in A \Leftrightarrow \forall^{p(|x|)} y$: $\langle x, y \rangle \in B$. **D** □

Poznámka: *Opět, pokud to rozzipujeme, tak dostaneme:* $A \in \forall$**C**, *když* $\exists B \in$ **C**, $\exists$ *polynom* p: $x \in A \Leftrightarrow \forall y$: $(|y| \leq p(|x|)) \Rightarrow \langle x, y \rangle \in B$

Definice 1.39 (***Třída uzavřená na zdvojování***): *Třída* **C** *je uzavřená na zdvojování, pokud* $\forall A \in$ **C** *platí, že* $B = \{\langle x, y \rangle | x \in A\} \in$ **C**. **D** □

Věta 1.40: **V**

1) $\exists$**P** = **NP**.
2) $\forall$**P** = *co*-**NP**.
3) $\forall k > 0$: $\exists \Sigma_k = \Sigma_k$.
4) $\forall k > 0$: $\forall \Pi_k = \Pi_k$.
5) $\forall k \geq 0$: $\exists \Pi_k = \Sigma_{k+1}$.
6) $\forall k \geq 0$: $\forall \Sigma_k = \Pi_{k+1}$.

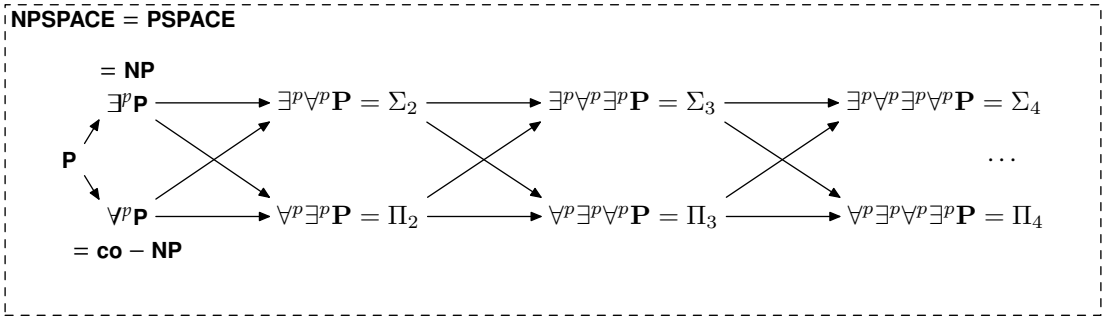
Důsledek 1.41 (***PH pomocí alternujících kvantifikátorů***):

a) $A \in \Sigma_k$ tehdy a jen tehdy, když existují $B \in$ **P** a polynom p takové, že

$$x \in A \Leftrightarrow \exists^{p(|x|)} y_1 \forall^{p(|x|)} y_2 \ldots : (x, y_1, y_2, \ldots, y_k) \in B$$

b) $A \in \Pi_k$ tehdy a jen tehdy, když existují $B \in$ **P** a polynom p takové, že

$$x \in A \Leftrightarrow \forall^{p(|x|)} y_1 \exists^{p(|x|)} y_2 \ldots : (x, y_1, y_2, \ldots, y_k) \in B$$



Obrázek 1.3.2. Polynomiální hierarchie. Šipky znázorňují inkluze.

Důsledek 1.42 (***Kolaps PH na k-té úrovni***): *Pokud pro nějaké $k > 0$ platí $\Sigma_k = \Pi_k$, pak $\forall j \geq 0: \Sigma_{k+j} = \Pi_{k+j} = \Sigma_k$.*

Důsledek 1.43 (***Ostré inkluze anebo konečnost PH***): *Buď platí $\forall k \geq 0: \Sigma_k \subset \Sigma_{k+1}$ nebo se polynomiální hierarchie skládá z konečně mnoha různých tříd Σ_k.*

Důsledek 1.44 (***Neostrá inkluze v hierarchii***): *Pokud existuje k takové, že $\mathbf{P} = \Sigma_0 \subset \Sigma_k$, pak platí $\mathbf{P} \subset \mathbf{NP}$.*

1.3.2 Pseudopolynomiální algoritmy

Motto: Čísla jsou struktury, jejichž hodnota roste exponenciálně s délkou jejich zápisu (pokud použijeme jinou než unární soustavu). Pokud doba běhu závisí na hodnotě čísla, musíme s tím počítat, a a a vo tom to je...

D **Definice 1.45** (***kód(X)***): *Mějme dán rozhodovací problém Q a jeho instanci X. Kód(X) je délka (počet bitů) instance X v binárním či vyšším kódování.* □

D **Definice 1.46** (***max(X)***): *Mějme dán rozhodovací problém Q a jeho instanci X.* max(X) *je největší číslo v X. (hodnota, nikoliv délka binárního zápisu).* □

Příklad 1.47: *Např. kód*$(\langle 10000, 3, 2\rangle) = 9$, *ale max*$(\langle 10000, 3, 2\rangle) = 10000$.

D **Definice 1.48** (***Pseudopolynomiální algoritmus***): *Algoritmus řešící Q se nazývá pseudopolynomiální, pokud je jeho časová složitost při spuštění na vstupu X omezena polynomem v proměnných kód(X) a* max(X). □

Poznámka: *Každý polynomiální algoritmus je i pseudopolynomiální.*

D **Definice 1.49** (***Číselný problém***): *Rozhodovací problém Q se nazývá číselný, pokud neexistuje polynom p takový, že pro každou instanci X problému Q platí* max$(X) \leq p($*kód*$(X))$. □

Poznámka: *Pokud v problému nejsou čísla (např. SAT), tak si* max(X) *můžeme dodefinovat dle své libosti jako nějakou konstantu.*

V **Věta 1.50**: *Nechť Q je* **NP***-úplný problém, který není číselný. Potom pokud* $\mathbf{P} \neq \mathbf{NP}$*, tak Q nemůže být řešen pseudopolynomiálním algoritmem.*

Důkaz: Idea důkazu: Sporem ukážeme, že kdyby existoval pseudopolynomiální algoritmus, tak bychom **NP**-úplné problémy uměli řešit v polynomiálním čase a tedy by platilo $\mathbf{P} = \mathbf{NP}$.

Mějme Q nečíselný **NP**-úplný problém (tedy $\exists p: \forall X$ instance Q: max$(X) \leq p($kód$(X))$).

Chceme, aby $\mathbf{P} \neq \mathbf{NP} \Rightarrow \nexists$ pseudopolynomiální algoritmus řešící Q.

Sporem: Nechť takový algoritmus A existuje. Pak pro něj existuje polynom q takový, že počet kroků A na instanci $X \leq q($max$(X),$ kód$(X)) \leq q(p($kód$(X)),$ kód$(X)) = \hat{q}($kód$(X))$, pro nějaký polynom $\hat{q}$. Polynom p je polynom, zmíněný výše u připomenutí, co je to nečíselný problém.

Z toho ale plyne, že máme polynomiální algoritmus pro **NP**-úplný problém, tedy **P** = **NP**. SPOR. ♡

$$T(n) \text{ algoritmu omezené pomocí } p(max(X)) = \begin{cases} \text{ano} & \text{pseudopolynomiální} \\ \text{ne} & \text{není pseudopolynomiální} \end{cases}$$

$$\text{Rozhodovací problém } Q = \begin{cases} \nexists p\text{: } \forall X, X \text{ je instance } Q : \max(X) \leq p(\text{kód}(X)) & \text{číselný} \\ \text{jinak} & \text{není číselný} \end{cases}$$

Příklad 1.51 (***Pseudopolynomiální algoritmus pro součet podmnožiny***): *Předpokládejme, že platí* $a_1 \geq a_2 \geq \ldots \geq a_n$*, a že* A *je pole délky* b*. Existuje množina indexů* $S \subseteq \{1, \ldots, n\}$*, taková, že* $\sum_{i \in S} a_i = b$*?*

1. **for** $j \leftarrow 1$ to b **do**:
2. $A[j] \leftarrow 0$
3. **end for**
4. $a_0 \leftarrow b + 1$
5. **for** $i \leftarrow 1$ to n **do:**
6. $A[a_i] \leftarrow 1$
7. **for** $j \leftarrow b$ downto a_{i-1} **do:**
8. **if** $(A[j] = 1)\&\&(j + a_i \leq b)$ **then**
9. $A[j + a_i] \leftarrow 1$
10. **end if**
11. **end for**
12. **end for**
13. **return** $A[b] = 1$

Algoritmus běží v čase $O(n \cdot b)$, což je exponenciální, pokud je vstup kódovaný v soustavě $r \geq 2$ ($O(r^{|n|} \cdot r^{|b|})$), ale polynomiální, pokud je vstup kódovaný unárně. Buď tedy mám exponenciálně dlouhý vstup, a pak už nám na tom běží algoritmus v polynomiálním čase, a nebo mám polynomiálně dlouhý vstup a pak algoritmus poběží v exponenciálně dlouhém čase.

1.3.3 Silná NP-úplnost

Motto: I když nějaké **NP**-úplné problémy zbavíme číselnosti, například si v TSP povolíme maximální váhu 3, tak nám stále zůstanou **NP**-úplné.

Definice 1.52 (***Množina instancí problému***): *Nechť je* Q *rozhodovací problém a* p *polynom. Symbolem* Q_p *označíme množinu instancí problému* Q *(tj. podproblém problému* Q*), pro které platí* $\max(X) \leq p(\text{kód}(X))$*, tj.* $Q_p = \{X \in Q | max(X) \leq p(\text{kód}(X))\}$. □ **D**

Věta 1.53: *Nechť* Q *je silně* **NP***-úplný problém. Potom pokud* **P** ≠ **NP***, tak* Q *nemůže být řešen pseudopolynomiálním algoritmem.* **V**

Důkaz: Dle věty 1.50 nelze nečíselný **NP**-úplný problém řešit pseudopolynomiálním algoritmem, za předpokladu, že **P** ≠ **NP**. Silně **NP**-úplný problém je **NP**-úplný problém a není číselný. Tedy ani ten nelze řešit pseudopolynomiálním algoritmem za předpokladu, že **P** ≠ **NP**. ♡

1.3.4 Třída #P a #P-úplnost

Definice 1.54 (***Certifikační funkce***): $w : \Sigma^* \to \mathcal{P}(\Gamma^*)$ □ **D**

Definice 1.55 (***Certifikát***): $y \in w(x) \subseteq \Gamma^*$ □ **D**

Definice 1.56 (***Rozhodovací problém asociovaný s certifikační funkcí***): $A_w = \{x \in \Sigma^* | w(x) \neq \emptyset\}$ □ **D**

Definice 1.57 (***Sharp-P***): *Množina certifikačních funkcí* w*:* **D**

(i) ∃ *deterministkický algoritmus:* $\forall x \in \Sigma^*, \forall y \in \Gamma^*$ *ověří v polynomiálním čase* $T = p(|x| + |y|)$, *zda* $y \in w(x)$.
(ii) ∃p, *polynom:* $\forall y \in w(x) : |y| \leq p(|x|)$.

□

V **Věta 1.58**: $w \in \#\mathbf{P} \Rightarrow A_w \in \mathbf{NP}$.

Důkaz: Mám $x \in \Sigma^*, x \in A_w$. Pomocí NTS uhodnu řešení y.

- Díky (ii) vím, že na pásce stačí $p(|x|)$ symbolů (certifikát polynomiální délky).
- Díky (i) mám polynomiální algoritmus na ověření.

♡

V **Věta 1.59**: $A \in \mathbf{NP} \Rightarrow \exists w \in \#\mathbf{P} : A = A_w$.

Důkaz: $A \in \mathbf{NP} \Rightarrow \exists$NTS M: $\forall x \in A \exists$ pijmajc vpoet dlky $\leq |p(x)|$. Definuji pomocnou certifikační funkci

$$w(x) = \{y | y \in \Gamma^* : M \text{ pijme } x \text{ po cest s kdem } y \text{ dlky } \leq p(x)\}$$

(Můžeme si ji představovat jako množinu logů TS, polynomiálně dlouhých.)
$A = A_w$ přímo z definice. Platí $w \in \#\mathbf{P}$?
(i) $\begin{cases} y \in w(x) & \text{NTS toto ověří} \\ y \notin w(x) & \begin{cases} |y| > p(x) & \text{pak ne, protože je to moc dlouhé.} \\ \text{jinak} & \text{trasujeme NTS dle } y \text{ a pokud neskončí v přijímacím stavu, pak ne.} \end{cases} \end{cases}$
(ii) ano, máme jen kódy délky $\leq p(x)$. ♡

Pozorování 1.60: $|w(x)|$ lze spočítat v polynomiálním čase. Tedy lze v polynomiálním čase rozhodnout zda $x \in A_w$ i zda $x \in (\mathcal{P}(\Sigma^*) \setminus A_w)$. Z tohoto důvodu jsou #SAT, #KLIKA, #SP, ... alespoň tak těžké jako SAT, KLIKA, SP, ...

D **Definice 1.61** (***Polynomiální redukce***): *Certifikační funkce* $w: \Sigma^* \to \mathcal{P}(\Gamma^*), v: \Pi^* \to \mathcal{P}(\Delta^*), w, v \in \#\mathbf{P}$. *Polynomiální redukce je dvojice funkcí vyčíslitelných v polynomiálním čase* $\sigma: \Sigma^* \to \Pi^*$ *a* $\varphi: \mathbb{N} \to \mathbb{N}$ *takových, že* $\forall x \in \Sigma^*: |w(x)| = \varphi(|v(\sigma(x))|)$. □

D **Definice 1.62** (***Šetrná polynomiální redukce***): *Polynomiální redukce, kde* φ *je identita, tedy* $\forall n \in \mathbb{N}: \varphi(n) = n$. □

V **Věta 1.63**: *#KACHL je* #**P***-úplný problém.*

Důkaz: ♡

1.4 Aproximační algoritmy a schémata

1.4.1 Aproximační algoritmy

Motto: NP-úplné optimalizační problémy se řeší špatně, ale občas se spokojíme s tím, když se řešení alespoň blíží.

D **Definice 1.64** (***Poměrová chyba***): *Aproximační algoritmus* A *řeší optimalizační problém* X *s poměrovou chybou* $r(n)$, *pokud pro všechna zadání problému* X *velikost* n *platí* $\max\left\{\frac{f(APR)}{f(OPT)}, \frac{f(OPT)}{f(APR)}\right\} \leq r(n)$. □

D **Definice 1.65** (***Relativní chyba***): *Aproximační algoritmus* A *řeší optimalizační problém* X *s relativní chybou* $e(n)$, *pokud pro všechna zadání problému* X *velikosti* n *platí* $\frac{|f(APR)-f(OPT)|}{f(OPT)} \leq e(n)$. □

Poznámka: *Při zlepšení aproximace obě chyby klesnou. Poměrová chyba vyjadřuje kolikrát je aproximované řešení horší než optimální, relativní chyba vyjadřuje, o kolik (normovaně) je aproximované řešení horší než optimální.*

Příklad 1.66 (***Maximalizační problém***): *Optimalizační verze problému $\mathcal{KLIKA}$. Cílem je nalézt největší kliku v grafu. Umí se $f(APR) \geq \frac{3}{4}f(OPT)$.*

Příklad 1.67 (***Minimalizační problém***): *Optimalizační verze problému $\mathcal{TSP}$. Cílem je nalézt nejkratší hamiltonovskou kružnici v daném grafu. Umí se $f(APR) \leq 2f(OPT)$.*

Příklad 1.68 (***Aproximační algoritmus pro TSP s trojúhelníkovou nerovností na úplném grafu***): *Mějme na vstupu Neorientovaný graf G, kde je každá hrana ohodnocena funkcí $w\colon E(G) \to \mathbb{R}_0^+$. Chceme najít nejkratší hamiltonovskou kružnici. Existuje algoritmus, který najde cestu s chybu nejvýše 2.*

Důkaz: Nejprve najdeme minimální kostru grafu, která má váhu T. Minimální kostru zakořeníme (jeden z vrcholů zvolíme jako kořen) a pomocí DFS ji projdeme. Vrcholy budeme postupně dávat do nalezené hamiltonovské cesty. Pokud však narazíme na vrchol, který již v cestě je, tak ho budeme ignorovat a přesuneme se na další nenavštívený. To si můžeme dovolit, protože graf je úplný, a tedy obsahuje hrany mezi všemi vrcholy. Cestu to neprodlouží, protože váhy splňují trojúhelníkovou nerovnost.

Celkem tedy naše nalezená cesta má váhu $T' \leq 2T$ (konstru váhy T jsme prošli dvakrát, některé hrany jsme zkrátili). Pokud bychom našli optimální kružnici C, a odebrali z ní jednu hranu, dostali bychom také kostru, která by měla váhu T_C. Protože T byla minimální, tak $T \leq T_C$. Tedy i platí, že $2T \leq 2T_C$. Hodnota C je navíc vyšší než T_C (T_C jsme dostali vyhozením hrany z C). Když to vše poskládáme, tak dostaneme $T' \leq 2T \leq 2T_C \leq 2C$. ♡

Věta 1.69 (***O neexistenci aproximačního algoritmu pro TSP***): *Nechť $r \geq 1$ je libovolná konstanta. Potom pokud* **P** $\neq$ **NP**, *tak neexistuje polynomiální aproximační algoritmus řešící obecný případ obchodního cestujícího s poměrovou chybou nejvýše r.* **V**

Důsledek 1.70 (***O existenci neaproximovatelných úloh***): *Existují* **NP***-těžké optimalizační úlohy, pro které neexistují polynomiální aproximační algoritmy s konstantní poměrovou chybou (pokud* **P** $\neq$ **NP***).*

1.4.2 Aproximační schémata

Definice 1.71 (***Aproximační schéma***): *Aproximační schéma (AS) pro optimalizační úlohu X je algoritmus, jehož vstupem je zadání Y úlohy X a (racionální) číslo $e > 0$, který pro libovolné pevné e pracuje jako aproximační algoritmus pro úlohu X s relativní chybou e.* □ **D**

Poznámka: *Doba běhu může být exponenciální (!) a to jak ve velikosti zadání Y, tak v $\frac{1}{e}$.*

Definice 1.72 (***Polynomiální aproximační schéma***): *Polynomiální aproximační schéma (PAS) pro optimalizační úlohu X je AS, jehož časová složitost je polynomiální vzhledem k velikosti zadání Y úlohy X.* □ **D**

Poznámka: *Doba běhu ale může být exponenciální vzhledem k $\frac{1}{e}$.*

Definice 1.73 (***Úplně polynomiální aproximační schéma***): *Úplně polynomiální aproximační schéma (ÚPAS) pro optimalizační úlohu X je PAS, jehož časová složitost je polynomiální také vzhledem k $\frac{1}{e}$.* □ **D**

Příklad 1.74 (***Algoritmus pro součet podmnožiny***): *Součet podmnožiny můžeme vyřešit v exponenciálním čase algoritmem 1.75. Exponenciela se schovává v řádku 4, protože možných podmnožin je až exponenciálně mnoho (2^i). Toto vyřešíme tím, že do algoritmu vpašujeme proceduru TRIM (algoritmus 1.76, běžící v $\Theta(m)$), která seznam prořeže. Zápisem $L_1 \otimes L_2|_t$ se rozumí slití seznamů L_1 a L_2, přičemž vyloučíme hodnoty, které jsou větší než t. Zápisem $L+c$ rozumíme přičtení konstanty c ke každému prvku L. Nakonec tedy dostaneme ÚPAS pro součet podmnožiny (algoritmus 1.77), parametrizovaný relativní*

chybou ε. ÚPAS volá TRIM s parametrem $\frac{\varepsilon}{n}$ – každým prořezáním se chyba může zvětšit, a my chceme, aby při n iteracích nepřesáhla ε.

A **Algoritmus 1.75**: Exponenciální algoritmus pro součet podmnožiny.

1. **procedure** SP-OPTIMAL(S, t)
2. $n \leftarrow |S|, L_0 \leftarrow \langle 0 \rangle$
3. **for** $i \leftarrow 1, \ldots, n$ **do:**
4. $L_i \leftarrow L_{i-1} \otimes (L_{i-1} + c_i)\,|_t$
5. **end for**
6. **return** největší prvek z L_n.

A **Algoritmus 1.76**: Procedura, která prořežává seznamy. Nechá tam jen prvky, které se liší alespoň „o δ procent“. Seznam L je vzestupně setříděný.

1. **procedure** TRIM(L, δ)
2. $m \leftarrow |L|, L' = \emptyset, first \leftarrow= -\infty$
3. **for** $i \leftarrow 1, \ldots, m$ **do:**
4. **if** $first < (1 - \delta)y_i$ **then**
5. $first \leftarrow y_i$
6. $L' \leftarrow L' \cup \{first\}$
7. **end if**
8. **end for**
9. **return** L'.

A **Algoritmus 1.77**: ÚPAS pro součet podmnožiny.

1. **procedure** SP-OPTIMAL-UPAS(S, t, ε)
2. $n \leftarrow |S|, L_0 \leftarrow \langle 0 \rangle$
3. **for** $i \leftarrow 1, \ldots, n$ **do:**
4. $L_i \leftarrow L_{i-1} \otimes (L_{i-1} + c_i)\,|_t$
5. $L_i \leftarrow TRIM(L_i, \frac{\varepsilon}{n})$
6. **end for**
7. **return** největší prvek z L_n.

V **Věta 1.78**: *Algoritmus popsaný v příkladu 1.74 je ÚPAS.*

1.5 Metody tvorby algoritmů: dynamické programování, hladový algoritmus na matroidu

1.5.1 Dynamické programování

Motto: Dynamické programování řeší problémy tak, že kombinuje řešení překrývajících se podproblémů.

Poznámka (***Kroky při návrhu algoritmu využívajícího technik dynamického programování***):

1) Charakterizace struktury optimálního řešení.
2) Rekurzivně urči hodnotu optimálního řešení.
3) Spočtení hodnoty optimálního řešení postupem *bottom up*.
4) Konstrukce optimálního řešení ze spočtené informace.

D **Definice 1.79** (***Top-down přístup***): *Rekurzivní volání podproblémů a pamatování si jejich výsledků.* □

D **Definice 1.80** (***Bottom-up přístup***): *Výpočet od nejmenších podproblémů až ke konečnému.* □

Příklad 1.81 (***Levenshteinova editační vzdálenost***): *Na vstupu dostaneme dvě slova a naším cílem je najít nejmenší počet editačních operací (přidání znaku, smazání znaku a změna znaku) tak, abychom*

z jednoho slova dostali druhé. Používá se to například při detekci překlepů („nechtěli jste spíše hledat ... ?“). Podle následujícího předpisu postupně vyplníme tabulku a zároveň si v ní pamatujeme, odkud jsme se do políčka dostali. Odpověď nalezneme v pravém dolním rohu. Když pak půjdeme zpět po šipkách, nalezneme i příslušnou editační posloupnost. Slovo SUNDAY se změní na slovo SATURDAY pomocí tří operací S, přidat A, přidat T, U, změnit N na R, D, A, Y. Tabulka obsahuje změny z libovolného podřetězce těchto slov. Sami si tento příklad zkuste v připravené prázdné tabulce. Uvedený postup je ukázkou bottom-up přístupu.

$$
\begin{aligned}
A[0][j] &= j \\
A[i][0] &= i \\
A[i][j]^{\downarrow} &= A[i-1][j]+1 \\
A[i][j]^{\searrow} &= A[i-1][j-1] + \begin{cases} 0 & A[i-1][j-1] = A[i][j] \\ 1 & A[i-1][j-1] \neq A[i][j] \end{cases} \\
A[i][j]^{\rightarrow} &= A[i][j-1]+1 \\
A[i][j] &= \min\left\{A[i][j]^{\downarrow}, A[i][j]^{\searrow}, A[i][j]^{\rightarrow}\right\}
\end{aligned}
$$

	ε	S	U	N	D	A	Y
ε	**0**$^{\downarrow}$	1$^{\rightarrow}$	2$^{\rightarrow}$	3$^{\rightarrow}$	4$^{\rightarrow}$	5$^{\rightarrow}$	6$^{\rightarrow}$
S	1$^{\downarrow}$	**0**$^{\searrow}$	1$^{\rightarrow}$	2$^{\rightarrow}$	3$^{\rightarrow}$	4$^{\rightarrow}$	5$^{\rightarrow}$
A	2$^{\downarrow}$	**1**$^{\downarrow}$	1$^{\searrow}$	2$^{\searrow}$	3$^{\searrow}$	3$^{\searrow}$	4$^{\rightarrow}$
T	3$^{\downarrow}$	**2**$^{\downarrow}$	2$^{\searrow}$	2$^{\searrow}$	3$^{\searrow}$	4$^{\searrow}$	4$^{\searrow}$
U	4$^{\downarrow}$	3$^{\searrow}$	**2**$^{\searrow}$	3$^{\searrow}$	3$^{\searrow}$	4$^{\searrow}$	5$^{\searrow}$
R	5$^{\downarrow}$	4$^{\downarrow}$	3$^{\downarrow}$	**3**$^{\searrow}$	4$^{\searrow}$	4$^{\searrow}$	5$^{\searrow}$
D	6$^{\downarrow}$	5$^{\downarrow}$	4$^{\downarrow}$	4$^{\searrow}$	**3**$^{\searrow}$	4$^{\rightarrow}$	5$^{\searrow}$
A	7$^{\downarrow}$	6$^{\downarrow}$	5$^{\downarrow}$	5$^{\searrow}$	4$^{\downarrow}$	**3**$^{\searrow}$	4$^{\rightarrow}$
Y	8$^{\downarrow}$	7$^{\downarrow}$	6$^{\downarrow}$	6$^{\searrow}$	5$^{\downarrow}$	4$^{\downarrow}$	**3**$^{\searrow}$

	ε	P	I	V	O
ε	0$^{\downarrow}$	1$^{\rightarrow}$	2$^{\rightarrow}$	3$^{\rightarrow}$	4$^{\rightarrow}$
L	1$^{\downarrow}$				
Á	2$^{\downarrow}$				
S	3$^{\downarrow}$				
K	4$^{\downarrow}$				
A	5$^{\downarrow}$				

1.5.2 Hladový algoritmus na matroidu

Motto: Hladové algoritmy udělají lokálně optimální rozhodnutí a nikdy ho nemění.

Definice 1.82 (***Matroid***): *Matroid je uspořádaná dvojice $M = (S, I)$ splňující následující tři podmínky:* D

1) S je konečná a neprázdná množina prvků matroidu M.
2) (Dědičná vlastnost) I je neprázdná množina podmnožin množiny S (nezávislých podmnožin množiny S), která má následující dědičnou vlastnost:

$$(B \in I) \wedge (A \subseteq B) \Rightarrow (A \in I) \tag{1}$$

3) (Výměnná vlastnost) I má následující výměnnou vlastnost:

$$A, B \in I, |A| < |B| \Rightarrow \exists x \in B \setminus A : A \cup \{x\} \in I \tag{2}$$

□

Příklad 1.83 (***Grafový matroid***): *$G = (V, E)$ je neorientovaný graf. Pak G definuje matroid $M_G = (S_G, I_G)$, kde $S_G = E$ a $A \subseteq S_G$ je nezávislá (tzn. A neobsahuje cyklus). Podmnožinou nezávislé množiny je nezávislá množina, takže dědičná vlastnost je zachovaná. Navíc, z každé větší nezávislé množiny mohu přehodit nějakou hranu do menší nezávislé množiny a nerozbít ji (výměnná vlastnost).*

Věta 1.84: *Všechny maximální nezávislé množiny v matroidu mají stejnou velikost.* V

Důkaz: Plyne z výměnné vlastnosti. ♡

Definice 1.85 (***Vážený matroid***): *Matroid $M = (S, I)$ je vážený, pokud je dána váhová funkce $w : S \to \mathbb{R}^+$* D
□

D **Definice 1.86** (***Matroidový problém***): *Pro daný vážený matroid M najděte nezávislou množinu v M s co největší váhou (taková množina se nazývá optimální množina matroidu M).* □

A **Algoritmus 1.87**: Hladový algoritmus pro matroidový problém. Nechť je dán matroid $M = (S, I)$, kde $S = \{x_1, x_2, \ldots, x_n\}$ a váhová funkce $w: S \to \mathbb{R}^+$. Algoritmus běží v $T(n) = \Theta(n \log n + n(f))$, kde $f(n)$ je složitost testu na nezávislost.

```
1. procedure GREEDY(M, w):
2.   A ← ∅
3.   setřiď a přečísluj S sestupně podle vah (w(x_1) = max{w(x_i)})
4.   for i ← 1, ..., n do:
5.     if (A ∪ {x}) ∈ I then:
6.       A ← A ∪ {x}
7.     end if
8.   end for
9.   return A
```

V **Věta 1.88**: *Algoritmus 1.87 vrací optimální množinu matroidu M.*

Důkaz: Prvky přeskočené na počátku nemohou být v žádné optimální množině. Po výběru prvního prvku je pak zaručena existence optimální množiny. Zbývající problém je pak převeden na problém stejného typu, ale na menší množině prvků (opakujeme dokud existují neprázdné nezávislé množiny). ♡

1.6 Základy pravděpodobnostních algoritmů.

Motto: Pravděpodobnostní algoritmus dělá narozdíl od deterministického náhodné kroky.

D **Definice 1.89** (***Pravděpodobnostní turingův stroj***): *Pravděpodobnostní turingův stroj (PTS) M je turingův stroj, jehož přechodová funkce je modifikovaná tak, že pro $\forall q \in Q$ a $\forall a \in \Sigma$ platí $\delta(q, a) = \{\mu_0, \mu_1\}$, kde $\mu_0, \mu_1 \in Q \times \Sigma \times \{\leftarrow, \bullet, \rightarrow\}$. Navíc požadujeme, aby pravděpodobnost $P[\delta(q, a) = \mu_0] = P[\delta(q, a) = \mu_1] = \frac{1}{2}$.* □

D **Definice 1.90** (***Jazyk přijímaný PTS***): *Jazyk přijímaný PTS M definujeme jako množinu $L(M) = \left\{w \in \Sigma^* | P[T \text{ přijímá}] > \frac{1}{2}\right\}$.* □

D **Definice 1.91** (***NP (pravděpodobnostní varianta)***): *Třída jazyků, pro které existuje polynomiální algoritmus $A(x)$ takový, že $\forall x \notin L: P[A(x) \text{ přijme}] = 0$ a $\forall x \in L: P[A(x) \text{ přijme}] > 0$ se nazývá* **NP** *(nedeterministicky polynomiální).* □

D **Definice 1.92** (***RP***): *Třída jazyků, pro které existuje polynomiální algoritmus $A(x)$ takový, že $\forall x \notin L: P[A(x) \text{ přijme}] = 0$ a $\forall x \in L: P[A(x) \text{ přijme}] \geq \frac{1}{2}$ se nazývá* **RP** *(randomized polynomial time).* □

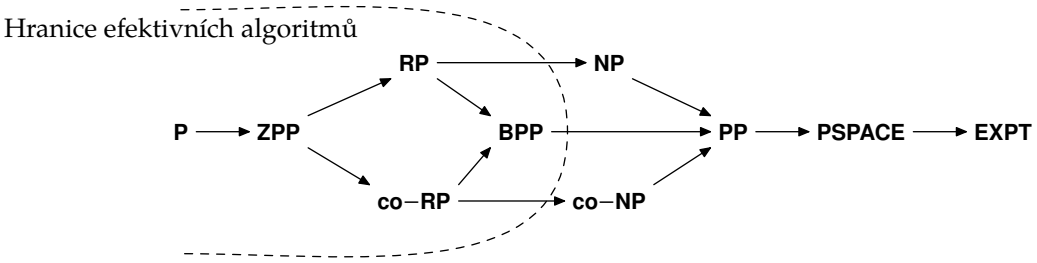
D **Definice 1.93** (***BPP***): *Třída jazyků, pro které existuje polynomiální algoritmus $A(x)$ takový, že $\forall x \notin L: P[A(x) \text{ přijme}] \leq \frac{1}{3}$ a $\forall x \in L: P[A(x) \text{ přijme}] \geq \frac{2}{3}$ se nazývá* **BPP** *(bounded-error probabilistic polynomial time).* □

D **Definice 1.94** (***PP***): *Třída jazyků, pro které existuje polynomiální algoritmus $A(x)$ takový, že $\forall x \notin L: P[A(x) \text{ přijme}] < \frac{1}{2}$ a $\forall x \in L: P[A(x) \text{ přijme}] \geq \frac{1}{2}$ se nazývá* **PP** *(probabilistic polynomial time).* □

D **Definice 1.95** (***ZPP***): *Třída jazyků, pro které existuje polynomiální algoritmus $A(x)$ takový, že $\forall x \notin L: P[A(x) \text{ přijme}] = 0$ a $\forall x \in L: P[A(x) \text{ přijme}] = 1$ se nazývá* **ZPP** *(zero-error probabilistic polynomial time).* □

Třída	$P\left[A(x) \text{ přijme} \,\middle|\, x \notin L\right]$	$P\left[A(x) \text{ přijme} \,\middle|\, x \in L\right]$
NP	0	> 0
RP	0	$\geq \frac{1}{2}$
BPP	$\leq \frac{1}{3}$	$\geq \frac{2}{3}$
PP	$< \frac{1}{2}$	$\geq \frac{1}{2}$
ZPP	0	1

Tabulka 1.6.1. Pravděpodobnostní třídy.



Obrázek 1.6.1. Známé inkluze pravděpodobnostních tříd.

Poznámka: *Neznáme* **BPP***-úplný problém, ale pokud se ukáže správným předpoklad, že* **BPP** = **P**, *pak by jimi byly* **P***-úplné problémy. Dá se zavést pravděpodobnostní polynomiální redukce, ale není tranzitivní. Platí, že* $\mathbf{BPP} \subseteq \Sigma_2 \cap \Pi_2$.

Definice 1.96 (***Algoritmus typu Las Vegas***): *Výsledek algoritmu typu Las Vegas je vždy správný, ale náhodnost ovlivňuje dobu běhu algoritmu (tzn. po jaké cestě se algoritmus ke správnému výsledku dobere).* □ **D**

Poznámka: *Problémy řešitelné polynomiálními algoritmy typu Las Vegas jsou v třídě* **ZPP**.

Příklad 1.97 (***Randomizovaný Quick-Sort***): *Pokud povita vybírám náhodně, pak je očekávaný průměrný čas algoritmu* $O(n \log n)$, *a to i v případě, kdy data na vstupu nejsou náhodné permutace (žádný vstup není špatný). Mimoto se pak dá Quick-Sort paralelizovat (vracíme výsledek té instance, která doběhne nejdřív).*

Definice 1.98 (***Algoritmus typu Monte Carlo***): *Náhodnost ovlivňuje v algoritmu typu Monte Carlo jak dobu běhu algoritmu, tak i správnost jeho výsledku.* □ **D**

Poznámka: *Problémy řešitelné polynomiálními algoritmy typu Monte Carlo jsou v třídě* **BPP**. *Pokud lze navíc problém řešit polynomiálním algoritmem typu Monte Carlo takovým, že pokud v případě* $x \notin L$ *vrátí vždy správný výsledek, pak je takový problém ve třídě* **RP**.

Definice 1.99 (***Algoritmus typu Atlanta***): *Algoritmus typu Atlanta vrátí správný výsledek v 75 % případů.* □ **D**

Definice 1.100 (***Polynomiálně ověřitelný důkaz (PCP)***): *Polynomiálně ověřitelný důkaz (PCP) je typ důkazu, který může být ověřen náhodným algoritmem s omezeným vstupem, který použije omezené množství náhodných bitů.* □ **D**

Definice 1.101 (***PCP(r,q)***): $\mathbf{PCP}(r, q)$ *je třída jazyků, která je dokazatelná pomocí polynomiálně ověřitelného důkazu, který položí nejvýše* q *dotazů do tabulky o velikosti* 2^r. □ **D**

Věta 1.102 (*PCP*): $\mathbf{NP} = \mathbf{PCP}(O(\log n), O(1))$ **V**

Seznam definic

Seznam vět

Kapitola 2
Vyčíslitelnost

2.1 Algoritmicky vyčíslitelné funkce, jejich vlastnosti, ekvivalence jejich různých matematických definic

D **Definice 2.1** (***Turingův stroj***): *Deterministický Turingův stroj M s k páskami (kde k je konstanta), je pětice $M = (Q, \Sigma, \delta, q_0, F)$, kde Q je konečná množina stavů jednotky, Σ je konečná pásková abeceda, $\delta: Q \times \Sigma^k \to Q \times \Sigma^k \times \{\leftarrow, \bullet, \rightarrow\}^k$ je (částečná) přechodová funkce, $q_0 \in Q$ je počáteční stav a $F \subseteq Q$ je množina koncových stavů.* □

Poznámka: *Existuje mnoho různých ekvivalentních definic TS, např. naposedný TS (v každém kroku se musí posunout doleva nebo doprava), TS s velmi malou abecedou (kompenzuje si to mnoha stavy), TS s velmi málo stavy (kompenzuje si to velkou abecedou), apod. Jedna z mála operací, která TS rozbije, je odebrání možnosti posuvu v jednom směru, to z TS udělá konečný automat.*

D **Definice 2.2** (***Podmíněná rovnost***): *$A \simeq B$. Pokud jeden výraz má smysl, pak má smysl i druhý výraz a platí rovnost.* □

D **Definice 2.3** (***Univerzální Turingův stroj***): *Univerzální turingův stroj $\mathcal{U}$ je Turingův stroj, který pro libovolný turingův stroj T a jeho vstup S, oba zakódované v pracovní abecedě $\mathcal{U}$, vyčísluje $T(S)$, tedy $\mathcal{U}(kód(T), kód(S)) \simeq T(S)$.* □

V **Věta 2.4** (***Kleenova o normální formě***): *Pro každé $k \leq 1$ existují*

- ČRF Ψ_k k+1 proměnných
- PRP T_k $k + 2$ proměnných
- PRF U jedné proměnné
- PRF s_k $k + 1$ proměnných

takové, že

1) Ψ_k je univerzální funkcí pro třídu všech ČRF k proměnných.
   $\Psi_k(e, x_1, \ldots, x_k)$ vyčísluje e-tou ČRF k proměnných. Navíc z odvození ČRF lze efektivně získat e a naopak z e lze efektivně získat odvození příslušné ČRF.
2) $\Psi_k(e, x_1, \ldots, x_k) \simeq U(\mu_y T_k(e, x_1, \ldots, x_k, y))$
3) s_k jsou prosté funkce rostoucí ve všech proměnných.
4) $\Psi_{m+n}(e, z_1, \ldots, z_m, x_1, \ldots, x_n) \simeq \Psi_n(s_m(z_1, \ldots, z_m), x_1, \ldots, x_n)$ (s-m-n věta).
   $T_{m+n}(e, z_1, \ldots, z_m, x_1, \ldots, x_n, y) \equiv T_n(s_m(z_1, \ldots, z_m), x_1, \ldots, x_n, y)$
5) $T_k(e, x_1, \ldots, x_k, y) \wedge T_k(e, x_1, \ldots, x_k, z) \Rightarrow y = z$

Důkaz: Technický důkaz přes turingovy stroje ♡

Poznámka: *Predikát $T_k(e, x_1, \ldots, x_k, y)$ vyjadřuje, že Turingův stroj, vyčíslující program e na vstupu $(x_1, \ldots, x_k)$ zastaví přesně po y krocích.*

Důsledek 2.5: *Libovolný program se dá upravit tak, že se z něj dá vytknout while cyklus.*

2.2 Primitivně a částečně rekurzivní funkce

Motto: Turingův stroj je pěkný výpočetní model, ale jednodušeji se pracuje s funkcemi. Zavedeme proto formalismus, který je s TS ekvivalentní, ale více odpovídá matematice.

D **Definice 2.6** (***Základní funkce***): *Základními funkcemi jsou následující totální (všude vyčíslitelné) efektivně vyčíslitelné funkce:*

- ***Nulová funkce**: $o(x) \simeq 0$.*

- ***Následník***: $s(x) \simeq x + 1$.
- ***Projekce***: $I_n^j(x_1, \ldots, x_n) \simeq x_j, 1 \leq j \leq n$.

□

Definice 2.7 (***Odvozovací pravidla (operátory)***): *Definujeme následující operátory na funkcích, tedy jejichž vstupem jsou funkce a výstupem jiné funkce:* **D**

- ***Operátor substituce***: $S_n^m(f, g_1, \ldots, g_m) = h$, *kde* $h(x_1, \ldots x_n) \simeq f(g_1(x_1, \ldots, x_n), \ldots, g_m(x_1, \ldots, x_n))$.
- ***Operátor primitivní rekurze***: $R_n(f, g) = h$, *kde:*

  - $h(0, x_2, \ldots, x_n) \simeq f(x_2, \ldots, x_n)$
  - $h(y + 1, x_2, \ldots, x_n) \simeq g(y, h(y, x_2, \ldots, x_n), x_2, \ldots, x_n)$

- ***Operátor minimalizace***: $\mu_n(f) = h$, *kde:*

$$h(x_1, \ldots, x_n) \simeq y \Leftrightarrow (\forall z < y)(f(x_1, \ldots, x_n, z) \downarrow \neq 0) \wedge (f(x_1, \ldots, x_n, y) \downarrow = 0)$$

□

Poznámka: *Operátor substituce odpovídá volání podprogramu. Operátor primitivní rekurze odpovídá for cyklu. Operátor minimalizace odpovídá while cyklu.*

Definice 2.8 (***Primitivně rekurzivní funkce***): *Třída primitivně rekurzivních funkcí (PRF) je nejmenší třídou funkcí, obsahující základní funkce, která je uzavřená na operátory substituce a primitivní rekurze.* **D**
□

Definice 2.9 (***Částečně rekurzivní funkce***): *Třída částečně rekurzivních funkcí (ČRF) je nejmenší třídou funkcí, obsahující základní funkce, která je uzavřená na operátory substituce, primitivní rekurze a minimalizace.* **D**
□

Definice 2.10 (***Obecně rekurzivní funkce***): *Třída obecně rekurzivních funkcí (ORF) obsahuje ty funkce z ČRF, které jsou totální (tedy všude definované).* **D**
□

Věta 2.11: *PRF jsou totální.* **V**

Důkaz: Indukcí ♡

Věta 2.12: $PRF \subset ORF \subset ČRF$. **V**

Důkaz: Vztah PRF ⊆ ČRF plyne z definice, stejně tak ORF ⊆ ČRF. Základní funkce jsou totální a operátory totalit zachovávají. Tedy platí i PRF ⊆ ORF.

„ORF ≠ ČRF“: Definujeme $g(x, y) = y + 1$, což je PRF. $\mu_y(g(x, y) \simeq 0)$ je ČRF, která však není nikde definovaná, tedy není ORF.

„PRF ≠ ORF“: Náznak — Pomocí Ackermanovy funkce $A(x, y)$ definované takto:

$$A(0, y) = \begin{cases} 1 & y = 0 \\ 2 & y = 1 \\ y + 2 & \text{jinak} \end{cases}$$

$$A(x, 0) = 1$$

$$A(x + 1, y + 1) = A(x, A(x + 1, y))$$

ukážeme, že takovouto funkci nelze zkonstruovat jako PRF (pro $x = 0$ vyčísluje funkci $y + 2$, pro $x = 1$ vyčísluje funkci $2y$, pro $x = 2$ vyčísluje 2^y, pro $y = 3$ věžové číslo $2^{2^{\cdot^{\cdot^{2}}}}$ výšky y a tak dále), protože to se for cyklem udělat nedá. ♡

D **Definice 2.13** (***Univerzální funkce***): *Mějme spočetnou třídu funkcí jedné proměnné $\mathcal{F}$. $\mathcal{U}$ je univerzální funkce pro $\mathcal{F}$ pokud platí:*

- $(\forall i)(\lambda_x \mathcal{U}(i,x) \in \mathcal{F})$
- $(\forall g \in \mathcal{F})(\exists i)(g = \lambda_x \mathcal{U}(i,x))$

□

2.3 Rekurzivní a rekurzivně spočetné množiny a jejich vlastnosti

Motto: Množiny, u kterých si můžu postavit turingův stroj, který mi říká, zda tam prvek je a nebo ne. Link do složitosti - může to být i množina jazyků...

D **Definice 2.14** (***Charakteristická funkce***): *Mějme množinu M. Její charakteristická funkce $C_M(x)$ je definovaná takto:*

$$C_M(x) = \begin{cases} 1 & x \in M \\ 0 & x \notin M \end{cases}$$

□

D **Definice 2.15** (***Rekurzivní množina***): *Množina M je rekurzivní, jestliže C_M je ORF.* □

Poznámka: *Množina je tedy rekurzivní, pokud existuje program, který vždy zastaví a rozhodne, zda prvek do množiny patří a nebo ne.*

D **Definice 2.16** (***Rekurzivně spočetná množina***): *Množina M je rekurzivně spočetná, jestliže existuje ČRF $\alpha(x)$ taková, že $\alpha(x) \downarrow \Leftrightarrow x \in M$.* □

Poznámka: *Množina je tedy rekurzivně spočetná, pokud je definičním oborem nějakého programu.*

Poznámka: *Duální k množinám jsou predikáty (vlastnosti, podmínky). Např. $Barva(x, Indigo)$ odpovídá tomu, že x je v množině $\{x | x$ má barvu indigo$\}$.*

D **Definice 2.17** (***Obecně rekurzivní predikát***): *Predikát P je obecně rekurzivní, jestliže jeho C_P je ORF.* □

D **Definice 2.18** (***Rekurzivně spočetný predikát***): *Predikát P je rekurzivně spočetný, jestliže P je definičním oborem nějaké ČRF.* □

D **Definice 2.19** (***Primitivně rekurzivní predikát***): *Predikát P je primitivně rekurzivní, jestliže C_P je PRF.* □

D **Definice 2.20** (***W***): *x-tá rekurzivně spočetná množina $W_x = \mathrm{dom}(\varphi_x) = \{y | \varphi_x(y) \downarrow\}$.* □

Poznámka: *W_x je definiční obor (množina možných vstupů) pro funkci s gödelovým číslem x.*

D **Definice 2.21** (***K***): *$K = \{x | x \in W_x\} = \{x | \varphi_x(x) \downarrow\} = \left\{x | \Psi_1(x,x) \downarrow\right\}$* □

Poznámka: *K je množina gödelových čísel funkcí, které samy na sobě zastaví.*

D **Definice 2.22** (***1-převeditelnost***): *Množina A je 1-převeditelná na B (značíme $A \leq_1 B$), jestliže existuje prostá ORF f taková, že $x \in A \Leftrightarrow f(x) \in B$.* □

D **Definice 2.23** (***m-převeditelnost***): *Množina A je m-převeditelná na B (značíme $A \leq_m B$), jestliže existuje ORF f taková, že $x \in A \Leftrightarrow f(x) \in B$.* □

Poznámka: *Vypadá to úplně stejně jako převeditelnosti ve složitosti.*

Definice 2.24 (***1-úplnost***): *Množina M je 1-úplná, jestliže je rekurzivně spočetná a každá rekurzivně spočetná množina je na ni 1-převeditelná.* □ **D**

Definice 2.25 (***m-úplnost***): *Množina M je m-úplná, jestliže je rekurzivně spočetná a každá rekurzivně spočetná množina je na ni m-převeditelná.* □ **D**

Poznámka: *Opět stejné jako C-úplnost ze složitosti.*

Věta 2.26: *K je 1-úplná.* **V**

Důkaz: K je rekuzrivně spočetná – triviální.

Mějme W_x libovolnou rekurzivně spočetnou funkci. Ta má charakteristickou funkci $\beta(y, x)$. My ji však upravíme a definujeme $\alpha(y, x, w) = \beta(y, x)$. Funkce α, s gödelovým číslem a, na hodnotě w nezávisí, jde o klasický trik ve vyčíslitelnosti.

$$\alpha(y, x, w) \simeq \Psi_3(a, y, x, w) \overset{s-m-n}{\simeq} \Psi_1(s_1^2(a, y, x), w) \simeq \varphi_{s_1^2(a,y,x)}(w)$$

Označme $h(y, x) = s_1^2(a, y, x)$. Ze s-m-n věty plyne, že s_1^2 je prostá PRF, tedy i ORF.

$$y \in W_x \Leftrightarrow a(y, x, w) \downarrow \Leftrightarrow \varphi_{h(y,x)}(w) \downarrow \Leftrightarrow \varphi_{h(y,x)}(h(y, x)) \downarrow \Leftrightarrow h(y, x) \in K$$

Protože na hodnotě w nezáleží, mohli jsme za něj dosadit cokoliv, třeba $h(y, x)$. Platí tedy, že $\forall x : y \in W_x \Leftrightarrow h(y, x) \in K$ pro nějakou prostou ORF, tedy $\forall x : W_x \leq_1 K$. Protože W_x jsou všechny rekurzivně spočetné množiny, je K 1-úplná. ♡

Definice 2.27 (***K0***): $K_0 = \{\langle y, x\rangle | y \in W_x\}$. □ **D**

Poznámka: *K_0 je množina všech dvojic vstupu programu a kódů programů, který na daných vstupech zastaví.*

Věta 2.28 (***Postova věta***): *Množina M je rekurzivní, právě když M i $\overline{M}$ jsou rekurzivně spočetné.* **V**

Důkaz: „⇒“ triviální, pokud je C_M ORF, pak C_M je i ČRF (tedy je M r.s.) a $C_{\overline{M}}$ můžeme definovat jako $C_{\overline{M}}(x) = 1 \Leftrightarrow C_M(x) = 0$ a $C_{\overline{M}}(x) = 0 \Leftrightarrow C_M(x) = 1$.

„⇒“ Pustíme jak C_M a tak $C_{\overline{M}}$. Obě jsou ČRF, takže obě doběhnou, když prvek patří do jejich množiny. Vrátím výsledek té, která zastaví s hodnotou 1. ♡

Definice 2.29 (***Úseková funkce***): *Funkce $f : \mathbb{N} \to \mathbb{N}$ je úseková, když jejím oborem hodnot je počáteční úsek $\mathbb{N}$ (nebo celé $\mathbb{N}$).* □ **D**

Poznámka: *Všechny PRF a ORF jsou úsekové, protože jsou totální.*

Věta 2.30 (***O selektoru***): *Nechť Q je RSP na $k + 1$ proměnných. Potom existuje ČRF φ s k proměnnými taková, že* **V**

- $\varphi(x_1, \ldots, x_k) \downarrow \Leftrightarrow \exists y Q(x_1, \ldots, x_k, y)$
- $\varphi(x_1, \ldots, x_k) \downarrow \Rightarrow Q(x_1, \ldots, x_k, \varphi(x_1, \ldots, x_k))$

Důkaz: Ukážeme pro případ $k = 1$. Mějme x, hledáme nejmenší dvojici $\langle y, s\rangle$, kde s je počet kroků turingova stroje.

$$\varphi(x) \simeq (\mu_w T_2(e, x, (w)_{2,1}, (w)_{2,2}))_{2,1}$$

♡

Poznámka: *V důkazu jsme vlastně odtrasovali výpočet TS, přičemž jsme mu zvětšovali počet možných kroků, než najde požadované y (pokud takové y nemá, tak z definice ani φ nezastaví). Využili jsme triku, kdy dvojici jde zakódovat do jednoho čísla, a z tohoto jde opět rozkódovat do dvojice.*

V **Věta 2.31**: *Každá rekurzivně spočetná množina je oborem hodnot nějaké ČRF.*

Důkaz: Mějme nějakou rekurzivně spočetnou množinu W_a. Definujme predikát $P(x, y)$ tak, že

$$P(x, y) \equiv (x \in W_a \wedge x = y)$$

Z věty o selektoru plyne, že P má selektor φ, což je funkce jedné proměnné, pro kterou platí $\varphi(x) \simeq y$. Platí, že $\forall x \in W_a : \varphi(x) \downarrow$, tedy $W_a = \mathrm{dom}(\varphi)$. Predikát P je definovaný tak, že platí $\varphi(x) \simeq x$, a tedy $\mathrm{dom}(\varphi) = \mathrm{rng}(\varphi)$. Z toho tedy plyne, že $W_a = \mathrm{rng}(\varphi)$, a tedy W_a je oborem hodnot nějaké ČRF. ♡

V **Věta 2.32**: *Každý obor hodnot ČRF je rekurzivně spočetná množina.*

Důkaz: Pro danou funkci α hledám nějakou rekurzivně spočetnou množinu W_b. Protože platí, že $W_b = \mathrm{dom}(\beta)$ pro nějaké β, stačí nám najít takové β, že $\mathrm{rng}(\alpha) = \mathrm{dom}(\beta)$.

Mějme tedy y, pro které platí, že $\beta(y) \downarrow$, a budeme hledat x takové, že $\alpha(x) \simeq y$. Funkci α si můžeme přepsat takto:

$$\alpha(x) \simeq U(\mu_z T_1(e, x, z)), \tag{1}$$

tedy hledáme hodnotu $\alpha(x)$ tak, že zkoušíme $\alpha(x) = 0?$, $\alpha(x) = 1?$, a tak dále, až hodnotu najdeme, pokud existuje. Funkci $\beta(y)$ si pak můžeme zkonstruovat takto, přičemž si v w držíme dvojici $\langle x, s \rangle$:

$$\beta(y) \simeq \mu_w \left(\alpha(x) \downarrow \text{ za } s \text{ kroků} \wedge \alpha(x) = y\right)$$

Funkci α si v definici funkce β simulujeme pomocí definice popsané v rovnici (1). Nemá cenu tu technicky rozepisovat β až na elementární predikáty, idea je stejná jako u věty o selektoru. Našli jsme tedy funkci, jejímž definičním oborem je obor hodnot funkce α. A protože $\mathrm{rng}(\alpha) = \mathrm{dom}(\beta) = W_b$, dokázali jsme, že oborem hodnot α je nějaká rekurzivně spočetná množina. ♡

V **Věta 2.33**: *Rekurzivní množiny jsou právě obory hodnot rostoucích úsekových ČRF.*

Důkaz: „⇒“: Máme tedy rekurzivní množinu M a chceme k ní najít rostoucí úsekovou funkci f. Zkonstruujeme funkci f tak, že očísluje od nuly prvky M v rostoucím pořadí:

- $f(0) \simeq \mu_x(x \in M)$
- $f(n + 1) \simeq \mu_x(x > f(n) \wedge x \in M)$

„⇐“: Máme rostoucí úsekovou funkci f a hledáme rekurzivní množinu M, pro kterou platí $\mathrm{rng}(f) = M$. Pokud je $\mathrm{dom}(f)$ konečná množina (což neumíme efektivně dokázat...), tak by $\mathrm{rng}(f)$ bylo také konečné (tzn. existuje $n = \max\{dom(f)\}$). Pak stačí forcyklem projet hodnoty od $f(0)$ až $f(n)$ a dostaneme M. Protože nám na nagenerování M stačí PRF, je M rekurzivní.

Nechť je f totální. Protože je f rostoucí a úseková, tak pro každé $f(x) = y$ platí, že $x \leq y$ (kdybychom nějaké x přeskočili, nebyla by úseková, kdybychom nějaké prohodili, nebyla by rostoucí). Pro zjištění, zda dané y je v množině M (tedy $\exists x : f(x) = y$ a $y \in M$) tedy stačí primitivní rekurze, která projde všechna možná x od nuly až po y (protože víme, že $0 \leq x \leq y$ anebo $y = f(x) \notin M$). Nepotřebujeme tedy operátor minimalizace (jde to for cyklem, nepotřebujeme while). Tedy je $M = \{f(0), f(1), \ldots\}$ rekurzivní. ♡

V **Věta 2.34**: *Nechť množina M je nekonečná, je rekurzivní právě tehdy, když je oborem hodnot rostoucí ORF.*

Důkaz: Dle věty 2.33 k M existuje rostoucí úseková ČRF φ. Protože je M nekonečná, je φ totální, tedy je i ORF. ♡

V **Věta 2.35**: *Rekurzivně spočetné množiny jsou právě obory hodnot prostých úsekových ČRF.*

Důkaz: „⇐“: Plyne z věty 2.31.

„⇒“: Máme rekurzivně spočetnou množinu M. K ní máme nějakou ČRF α, pro kterou $\mathrm{dom}(\alpha) = M$. Definujme si množinu B takto:

$$B = \{\langle x,s\rangle | \alpha(x)\downarrow \text{ za přesně } s \text{ kroků}\} = \{\langle x,s\rangle | T_1(a,x,s) \wedge (\forall j < s)(\neg T_1(a,x,j))\}$$

B je jistě rekurzivní (pro danou dvojici $\langle x,s\rangle$ přesně za s kroků zjistíme, zda $\langle x,s\rangle \in B$ a nebo ne). Dle věty 2.33 k ní máme rostoucí úsekovou ČRF β, pro kterou platí $\text{rng}(\beta) = B$. V množině B je každé x zastoupeno právě jednou. Pokud tedy definujeme $f = (\beta)_{2,1}$ (vydělení první složky dvojice), tak f je prostá úseková ČRF, kterou jsme hledali. ♡

Věta 2.36: *Každá rekurzivně spočetná množina obsahuje nekonečnou rekurzivní podmnožinu.* V

Důkaz: Mějme nekonečnou rekurzivně spočetnou množinu M. Dle věty 2.35 k ní máme prostou úsekovou ORF f takovou, že $\text{rng}(f) = M$. Definujme funkci g předpisem

- $g(0) \simeq f(0)$
- $g(n+1) \simeq f(\mu_j(f(j) > g(n)))$

Funkce g je rostoucí a úseková. Dle věty 2.33 je tedy množina $N = \text{rng}(g)$ rekurzivní množinou. Navíc platí, že N je nekonečná a $N \subseteq M$ (protože funkce g odpovídá některým hodnotám funkce f, která generuje M). Množina N je tedy hledanou nekonečnou rekurzivní množinou. ♡

2.4 Algoritmicky nerozhodnutelné problémy (halting problém)

Motto: Lze algoritmicky rozhodnout, zda se turingův stroj na daném vstupu zastaví?

Definice 2.37 (***Halting problém***): *Mějme univerzální turingův stroj $\mathcal{U}$. Halting problém je rozhodovacím problémem, který pro daný turingův stroj T a vstupní data S rozhodne, zda $\mathcal{U}(\text{code}(T), \text{code}(S))$ zastaví (tedy zda $\mathcal{U}(\text{code}(T), \text{code}(S))\downarrow$) a nebo ne.* D □

Věta 2.38 (***Nerozhodnutelnost halting problému***): *Halting problém je nerozhodnutelný.* V

Důkaz: Dokazujeme sporem. Pro spor předpokládejme, že halting problém je rozhodnutelný, tedy že příslušný univerzální turingův stroj vždy zastaví a vydá odpověď, zda turingův stroj nad daným vstupem zastaví a nebo ne.

To tedy znamená, že máme UTS $\mathcal{HALT}$, pro který platí:

- $\mathcal{HALT}(\text{code}(T), \text{code}(S)) = \text{ANO} \Leftrightarrow T(S)\downarrow$
- $\mathcal{HALT}(\text{code}(T), \text{code}(S)) = \text{NE} \Leftrightarrow T(S)\uparrow$

Sestavme turingův stroj *PODRAZ*, pro který bude platit:

$$PODRAZ(K)\downarrow \Leftrightarrow \mathcal{HALT}(\text{code}(K), \text{code}(K)) = \text{NE}$$

Stroj *PODRAZ* vložíme jako vstup stroji $\mathcal{HALT}$:

$$\mathcal{HALT}(\text{code}(PODRAZ), \text{code}(PODRAZ)) = \text{ANO} \Leftrightarrow PODRAZ(PODRAZ)\downarrow$$

Pokud stroj *PODRAZ* zastaví, tak stroj $\mathcal{HALT}$ odpoví ANO.

$$PODRAZ(PODRAZ)\downarrow \Leftrightarrow \mathcal{HALT}(\text{code}(PODRAZ), \text{code}(PODRAZ)) = \text{NE},$$

Pokud stroj $\mathcal{HALT}$ odpoví pro daný vstup ANO, tak se stroj *PODRAZ* na tomto vstupu zacyklí zacyklí (viz definice stroje *PODRAZ*), takže stroj $\mathcal{HALT}$ vrátí NE, což vede ke sporu. ♡

Poznámka: *Možná zmínit i Riceovu větu (věta 2.44 na straně 26), ze které plyne, že nelze programem rozpoznat, zda dva programy dělají to samé.*

2.5 Věty o rekurzi a jejich aplikace, Riceova věta

V **Věta 2.39** (***s-m-n***): *Pro každá dvě přirozená čísla $m, n \geq 1$ existuje prostá PRF s_m, jež je funkcí $m+1$ proměnných, a pro všechna $x, y_1, y_2, \ldots, y_m$ platí: $\varphi_{s_m(x,y_1,y_2,\ldots,y_m)}(z_1, z_2, \ldots, z_n) \simeq \varphi_x(y_1, y_2, \ldots, y_m, z_1, z_2, \ldots, z_n)$.*

Poznámka: *Věta nám říká, že z každého programu, který má na vstupu $m+n$ proměnných si můžeme vyrobit program, který má na vstupu jen n proměnných, a zbylých m proměnných zadrátujeme přímo do programu.*

V **Věta 2.40** (***O rekurzi (pevný bod)***): *Jestliže f je ORF jedné proměnné, potom existuje a takové, že $\varphi_{f(a)}(x) \simeq \varphi_a(x)$ pro všechna x.*

Důkaz: Idea důkazu: postavím $a = s_1(z, z)$ pro nějaké vhodné z a pomocí s-m-n věty najdu vhodné rovnosti.

Mějme funkci $\varphi_{f(s_1(z,z))}$ s gödelovým číslem e. Pak platí:

$$\varphi_{f(s_1(z,z))}(x) \simeq \Psi_2(e, z, x) \overset{s-m-n}{\simeq} \Psi_1(s_1(e, z), x) \simeq \varphi_{s_1(e,z)}(x)$$

Pokud za z dosadíme e, pak dostaneme

$$\varphi_{f(s_1(e,e))}(x) \simeq \Psi_2(e, e, x) \overset{s-m-n}{\simeq} \Psi_1(s_1(e, e), x) \simeq \varphi_{s_1(e,e)}(x)$$

Platí tedy, že $\varphi_{f(s_1(e,e))}(x) \simeq \varphi_{s_1(e,e)}(x)$, což pro $a = s_1(e, e)$ dává $\varphi_{f(a)}(x) \simeq \varphi_a(x)$. ♡

V **Věta 2.41** (***O rekurzi (závislost na parametrech)***): *Pro každou ČRF h s $m+1$ proměnnými existuje rostoucí PRF g s m proměnnými taková, že $\varphi_{h(g(y_1,\ldots,y_m),y_1,\ldots,y_m)} \simeq \varphi_{g(y_1,\ldots,y_m)}(x)$.*

Důkaz: Vezměme

$$\varphi_{h(s_{m+1}(z,z,y_1,\ldots,y_m),y_1,\ldots,y_m)}(x) \simeq \Psi_{m+2}(e^*, z, y_1, \ldots, y_m, x) \simeq \varphi_{s_{m+1}(e^*,z,y_1,\ldots,y_m)}(x)$$

a položme $g(y_1, \ldots, y_m) = s_{m+1}(e^*, e^*, y_1, \ldots, y_m)$. ♡

V **Věta 2.42** (***O rekurzi (mnoho pevných bodů)***): *Pro každou ČRF f existuje prostá PRF α taková, že $\varphi_{f(\alpha(j))}(x) \simeq \varphi_{\alpha(j)}(x)$.*

Poznámka: *Tato věta nám říká, že pevný bod není jen jeden, ale že jich je nekonečně mnoho.*

Důkaz: Idea důkazu: Jako u důkazu věty 2.40 budeme konstruovat $g(j)$ ve tvaru $g(j) \simeq s_2(z, z, j)$.

$$\varphi_{f(s_2(z,z,j))}(x) \simeq \Psi_3(e, z, j, x) \simeq \Psi_1(s_2(e, z, j), x) \simeq \psi_{s_2(e,z,j)} \simeq \varphi_{s_2(e,z,j)}(x)$$

Pro $z = e$ dostaneme $g(j) = s_2(e, e, j)$, a tedy

$$\varphi_{f(s_2(e,e,j))}(x) \simeq \Psi_3(e, e, j, x) \simeq \Psi_1(s_2(e, e, j), x) \simeq \psi_{s_2(e,ej)} \simeq \varphi_{s_2(e,e,j)}(x)$$

Platí tedy, že $\varphi_{f(s_2(e,e,j))}(x) \simeq \varphi_{s_2(e,e,j)}(x)$, což pro $g(j) = s_2(e, e, j)$ dává $\varphi_{f(g(j))}(x) \simeq \varphi_{g(j)}(x)$. ♡

V **Věta 2.43** (***O rekurzi (dvojná forma)***): *Pro každou ČRF h s $m+1$ proměnnými existuje a, které je indexem (gödelovým číslem) funkce $h(a, x_1, \ldots, x_m)$, tak, že $h(a, x_1, \ldots, x_m) \simeq \varphi_a(x_1, \ldots, x_m)$.*

Důkaz:

$$h(z, x_1, \ldots, x_n) \simeq \Psi_{n+1}(e, z, x_1, \ldots, x_n) \simeq \Psi_n(s_1(e, z), x_1, \ldots, x_n) \simeq \varphi_{s_1(e,z)}(x_1, \ldots, x_n)$$

Nyní již stačí použít větu 2.40 na $s_1(e, z)$ v proměnné z a dostáváme hledané a. ♡

V **Věta 2.44** (***Riceova***): *Nechť $\mathcal{A}$ je třída ČRF jedné proměnné, která je netriviální, potom $A_{\mathcal{A}} = \{x | \varphi_x \in \mathcal{A}\}$ není rekurzivní. Taková množina se nazývá indexová množina. Speciálním případem je třída jediné ČRF φ_{x_0}, kde $A = \{x | \varphi_x = \varphi_{x_0}\}$.*

Důkaz: Sporem. Nechť $A_{\mathcal{A}}$ je rekurzivní množina. Protože je $A_{\mathcal{A}}$ netriviální, tak existuje $a \in A_{\mathcal{A}}$ a $b \notin A_{\mathcal{A}}$. Definujme funkci f takto:

$$f(x) = \begin{cases} a & x \notin A_{\mathcal{A}} \\ b & x \in A_{\mathcal{A}} \end{cases}$$

Protože je množina $A_{\mathcal{A}}$ rekurzivní, je f ORF (f odpovídá charakteristické funkci množiny $A_{\mathcal{A}}$, $C_{A_{\mathcal{A}}}$, jen místo 0 a 1 vrací a a b).

Nyní budeme aplikovat větu 2.40. Funkce f je ČRF. Pak existuje nějaké e takové, že $\forall x: \varphi_{f(e)}(x) \simeq \varphi_e(x)$. Nechť platí, že $e \in A_{\mathcal{A}}$, tedy $\varphi_e \in \mathcal{A}$. Pak platí, že $f(e) = b$. Protože $b \notin A_{\mathcal{A}}$, tak $\varphi_{f(e)} \notin \mathcal{A}$. Jenže dle věty o rekurzi jsou funkce $\varphi_{f(e)}$ a φ_e ekvivalentní, a tedy nemůže nastat, že by jedna z nich v $\mathcal{A}$ byla a druhá nebyla, protože obě vyčíslují stejnou funkci a ta v $\mathcal{A}$ buď je a nebo není. SPOR. ♡

Důsledek 2.45: *Z Riceovy věty plyne, že nemůže existovat program, který o dvou libovolných programech efektivně zjistí, zda dělají to samé.*

2.6 Gödelovy věty

Definice 2.46 (***Rekurzivní neoddělitelnost***): *Dvojice množin A a B, $A \cap B = \emptyset$, je rekurzivně neoddělitelná, pokud neexistuje žádná rekurzivní množina M, $A \subseteq M$, pro kterou platí $M \cap B = \emptyset$.* □ **D**

Poznámka: *Alternativně se to dá vyjádřit tak, že $A \subseteq M$, $B \subseteq \overline{M}$.*

Definice 2.47 (***Efektivní neoddělitelnost***): *Dvojice množin A a B, $A \cap B = \emptyset$, je efektivně neoddělitelná, pokud existuje ČRF φ taková, že pokud $A \subseteq W_x$, $B \subseteq W_y$, $W_x \cap W_y = \emptyset$, tak platí $\varphi(x, y) \downarrow \notin W_x \cup W_y$.* □ **D**

Věta 2.48 (***Existence efektivně neoddělitelné dvojice***): *Existují disjunktní rekurzivně spočetné množiny A a B, které jsou efektivně neoddělitelné.* **V**

Důkaz: Idea: uděláme funkci, která pro vstup z jedné množiny vrací druhou a naopak, a provedeme diagonalizaci.

Definujme: $A = \{x | \varphi_x(x) \simeq 0\}$, $B = \{x | \varphi_x(x) \simeq 1\}$. A a B jsou disjunktní a rekurzivně spočetné. Lze sestrojit funkci $\varphi_a(x, y, w)$:

$$\varphi_a(x, y, w) = \begin{cases} 1 & w \text{ padne dříve do } W_x \text{ než do } W_y \\ 0 & w \text{ padne dříve do } W_y \text{ než do } W_x \\ \uparrow & \text{jinak} \end{cases}$$

Pomocí s-m-n věty si můžu první dva parametry pro danou dvojici fixovat (x je kód pro A, y je kód pro B) a získám tak $\varphi_{\alpha(x,y)}(w)$ pro nějakou funkci α. Nyní stačí nalézt bod, kde $\varphi_{\alpha(x,y)}$ bude divergovat. Její divergence totiž bude znamenat, že existuje bod mimo $W_x \cup W_y$.

Spustíme tedy $\varphi_{\alpha(x,y)}$ samu na sobě. Pokud by $\alpha(x, y) \in W_x$, pak by jistě padlo dříve do W_x než do W_y, tedy by $\varphi_{\alpha(x,y)}(\alpha(x, y)) \downarrow \simeq 1$. V takovém případě ale $\alpha(x, y)$ patří do B (kam patří vstupy, pro které $\varphi_x(x) \simeq 1$). Symetricky se ukáže, že kdyby $\alpha(x, y) \in W_y$, tak by nejprve padlo do W_y a přitom by mělo patřit do A. $\varphi_{\alpha(x,y)}(\alpha(x, y))$ tedy spadne mimo A i B. ♡

Věta 2.49 (***O vztahu efektivní a rekurzivní neoddělitelnosti***): *Nechť A a B jsou efektivně neoddělitelné množiny. Pak jsou i rekurzivně neoddělitelné.* **V**

Důkaz: Pro spor předpokládejme, že existuje rekurzivní množina M, která odděluje dvě efektivně neoddělitelné množiny. Protože je M rekurzivní, jsou dle Postovy věty 2.28 množiny M a $\overline{M}$ rekurzivně spočetné, tedy platí, že $M = W_x$ a $\overline{M} = W_y$ pro nějaké x a y. Protože jsou A i B efektivně neoddělitelné, tak existuje $\varphi(x, y) \notin W_x \cup W_y$. To ale znamená, že jsme našli bod, který není ani v M, ani v jejím doplňku $\overline{M}$, což je spor. ♡

Definice 2.50 (***Základní aritmetická síla***): *Jazyk obsahující numerály $\overline{0}$, $\overline{1}$, funkční symboly $+$, $\times$ a konečně mnoho axiomů (komutativita, asociativita, distributivita) označujeme jako základní aritmetickou sílu (Robinsonova aritmetika).* □ **D**

D **Definice 2.51** (***Reprezentovatelnost ČRF***): *ČRF f je reprezentovatelná v teorii T, která má ZAS, jestliže existuje formule $\mathcal{F}$ taková, že*

$$\begin{aligned} f(x_1,\ldots,x_n) \simeq y \quad &\Rightarrow \quad \vdash_T \mathcal{F}(\overline{x_1},\ldots,\overline{x_n},\overline{y}) \\ \vdash_T \mathcal{F}(\alpha_1,\ldots,\alpha_n,\beta) \wedge \vdash_T \mathcal{F}(\alpha_1,\ldots,\alpha_n,\gamma) \quad &\Rightarrow \quad \beta = \gamma \end{aligned}$$

□

Poznámka: *Značení $\overline{x}$ zde neznamená negaci x!, ale to, že x a $\overline{x}$ se každé vyskytují v jiném světě (svět čísel, svět logiky), byť se snaží vyjadřovat to samé.*

V **Věta 2.52** (***O reprezentovatelnosti***): *Každá ČRF f je reprezentovatelná v libovolné teorii T, která má ZAS. Dokonce existuje pro každou ČRF formule, která ji reprezentuje ve všech teoriích ZAS současně.*

Důsledek 2.53: *Jsou-li A a B disjunktní, rekurzivně spočetné množiny, potom existuje formule $\mathcal{G}$ taková, že $x \in A \Rightarrow \vdash_T \mathcal{G}(\overline{x})$ a $x \in B \Rightarrow \vdash_T \neg\mathcal{G}(\overline{x})$.*

D **Definice 2.54** (***Axiomatizovatelná teorie***): *Teorie T je axiomatizovatelná právě tehdy, když množina dokazatelných formulí je rekurzivně spočetná.* □

V **Věta 2.55** (***Gödelova věta o neúplnosti***): *Pro libovolnou bezespornou teorii T se základní aritmetikou sílou platí:*

1) množina dokazatelných funkcí není rekurzivní
2) je-li navíc T axiomatizovatelná, pak existují uzavřené formule $\mathcal{F}$, pro které $\nvdash_T \mathcal{F}$, $\nvdash_T \neg\mathcal{F}$.
3) bezespornost T nelze v T dokázat.

Důkaz: Bod 3 nebudeme dokazovat, to necháme logikům.

1) Tvrdíme, že množina dokazatelných formulí není rekurzivní. Vezměme dvě rekurzivně spočetné disjunkní množiny A a B, které jsou efektivně neoddělitelné. Dle důsledku 2.53 existuje formule $\mathcal{G}$ taková, že $x \in A \Rightarrow \vdash_T \mathcal{G}(\overline{x})$ a $x \in B \Rightarrow \vdash_T \neg\mathcal{G}(\overline{x})$. Definujme si množiny A_1 a B_1 takto:

- $A_1 = \{x | \vdash_T \mathcal{G}(\overline{x})\}$
- $B_1 = \{x | \vdash_T \neg\mathcal{G}(\overline{x})\}$

Na první pohled se zdá, že se A a A_1, resp. B a B_1 neliší, ale není tomu tak. My víme, že pro každé $x \in A$ se dá v teorii T dokázat, že platí $\mathcal{G}(\overline{x})$, a že pro každé $y \in B$ se v teorii T dá dokázat, že platí $\neg\mathcal{G}(\overline{y})$. Ale mějme nějaké $z \in C$, $C \cap (A \cup B) = \emptyset$. Zda platí $\mathcal{G}(\overline{z})$ nebo $\neg\mathcal{G}(\overline{z})$ nevíme. Abychom se nemuseli podobnými otázkami znervózňovat, sestavili jsme si množiny A_1 a B_1, které obsahují všechny validní vstupy pro formuli $\mathcal{G}$.

Zjevně platí, že $A \subseteq A_1$ a $B \subseteq B_1$. Teorie T je bezrozporná (tedy se v ní nedá dokázat jak výrok p tak výrok $\neg p$), takže $A_1 \cap B_1 = \emptyset$. Zároveň ani jedna z nich není rekurzivní, protože by separovala A a B (dle předpokladu jsou efektivně neoddělitelné, což dle věty 2.49 implikuje rekurzivní neoddělitelnost). Ani množina formulí dokazatelných v T tedy není rekuzivní (když není rekurzivní její podmnožina, tedy ani A_1, ani B_1, nemůže být rekurzivní sama).

2) Nechť je navíc teorie T axiomatizovatelná, tedy množina jejích dokazatelných formulí je rekurzivně spočetná. Pak i A_1 a B_1 budou rekurzivně spočetné. Protože jsou A i B efektivně neoddělitelné, tak mohu efektivně nalézt bod $k \notin A_1 \cup B_1$ (stejným postupem, jako bych hledal $l \notin A \cup B$, viz důkaz věty 2.48). Číslo k je kódem formule, která není dokazatelná v T, ani v ní není dokazatelná její negace. ♡

Seznam definic

Seznam vět

Kapitola 3
Datové struktury

3.1 Stromové vyhledávací struktury: binární stromy a jejich vyvažování, haldy, trie, B-stromy a jejich varianty

3.1.1 Binární stromy a jejich vyvažování

3.1.2 Haldy

Motto: Potřebujeme znát extrém množiny, a zbytek nás zatím nezajímá.

D **Definice 3.1** (***Halda***): *Halda je stormová sturktura, kde vrcholy reprezentují prvky z S a splňují lokální podmínku na váhovou funkci $w: S \to \mathbb{R}$: Pro každý vrchol v a jeho otce t platí, že $w(t) < w(v)$.* □

Poznámka: *U všech hald požadujeme, aby v kořeni byl minimální prvek. Pokud cheme, aby v kořeni byl maximální prvek, není problém algoritmy upravit (a nebo zvolit váhovou funkci $w'(x) = -w(x)$ a haldu neměnit).*

d-regulární haldy

D **Definice 3.2** (***d-regulární strom***): *Nechť $d > 1$ je přirozené číslo. Pak d-regulární strom je kořenový strom (T, r), pro který existuje pořadí synů jednotlivých vnitřních vrcholů takové, že očíslování vrcholů prohledáváním do šířky, v němž kořen r má číslo 1, splňuje následující podmínky:*

1) Každý vrchol má nejvýše d synů.
2) Když vrchol není list, pak všechny vrcholy s menším číslem mají právě d synů.
3) Když vrchol má méně než d synů, pak všechny vrcholy s většími čísly jsou listy.

□

D **Definice 3.3** (***Přirozené očíslování d-regulárního stromu***): *Očíslování v d-regulárním stromu se nazývá přirozené očíslování d-regulárního stromu.* □

D **Definice 3.4** (***d-regulární halda***): *d-regulární halda je d-regulární strom ve kterém platí, že pro každé $u, v \in T$, $u = otec(v)$, $w(u) < w(v)$.* □

Poznámka: *d-regulární halda je ta halda, o které jsme se učili na Programování I a II.*

Operace Up(H,x)	$T(n) = O(\log_d(n))$
Opraví porušenou podmínku uspořádání tak, že prvek prohodí s jeho otcem. Volá se rekurzivně směrem nahoru.	

Operace Down(H,x)	$T(n) = O(d \log_d(n))$
Opraví porušenou podmínku uspořádání tak, že prvek prohodí s nějakým jeho synem. Volá se rekurzivně směrem dolu.	

Operace Insert(H,x)	$T(n) = O(\log_d(n))$
Vloží nový prvek na konec haldy (za vrchol s nejvyšším číslem) a zavolá na něm Up(x).	

Operace Minimum(H)	$T(n) = O(1)$
Vrátí hodnotu vrcholu haldy.	

Operace Delete-Minimum(H)	$T(n) = O(d \log_d(n))$
Zahodí kořen stromu a místo něj umístí vrchol y z konce haldy (vrchol s nejvyšším číslem). Na novém kořeni haldy zavolá Down(y).	

Operace Delete(H,x)	$T(n) = O(d \log_d(n))$
Zahodí vrchol x a místo něj umístí vrchol y z konce haldy (vrchol s nejvyšším číslem). Pokud je $w(x) < w(y)$, pak zavolá Down(y), jinak zavolá Up(y).	

Operace Decrease(H,x,nová hodnota)	$T(n) = O(\log_d(n))$
Nastaví x novou hodnotu a zavolá Up(x).	

Operace Increase(H,x,nová hodnota)	$T(n) = O(d \log_d(n))$
Nastaví x novou hodnotu a zavolá Down(x).	

Leftist haldy

Definice 3.5 (***Null path length (npl)***): *Mějme binární kořenoý strom (T, r). Pro vrchol označme $npl(v)$ délku nejkratší cesty z vrcholu v do vrcholu, který má nejvýše jednoho syna. Vrchol, který má nejvýše jednoho syna má npl rovno 0.* □ D

Definice 3.6 (***Leftist halda***): *Mějme podmnožinu S univerza U a funkci $w : S \to \mathbb{R}$. Pak binární strom (T, r) takový, že* D

1) Má-li vrchol v jen jednoho syna, pak je to levý syn.
2) Má-li vrchol v dva syny, pak platí $npl(\text{pravý}) \leq npl(\text{levý})$.
3) Platí podmínka uspořádání v haldě.

□

Poznámka: *Leftist haldy jsou založené na operaci MERGE, kterou zvládají provést v čase $O(1)$. Oproti d-regulárním haldám navíc potřebují metodu OPRAV, která opraví porušenou podmínku 2) z definice lefist haldy.*

Operace Merge(H_1, H_2) $T(n) = O(1)$

Tato metoda sloučí dvě leftist haldy do jedné. Jednu haldu zvolí jako hlavní, a druhou postupně zmenšuje a rekurzivně zavěšuje doprava. Halda tedy roste směrem doleva.

1. **procedure** MERGE(H_1, H_2):
2. **if** $H_1 = \emptyset$ **then return** H_2 **end if**
3. **if** $H_2 = \emptyset$ **then return** H_1 **end if**
4. **if** $w(H_1.root) > w(H_2.root)$ **then**
5. prohoď H_1 a H_2
6. **end if**
7. $t \leftarrow H_1.root.right$
8. $H' \leftarrow MERGE(\text{subtree}(t), H_2)$
9. $H_1.root.right \leftarrow H'.root$
10. **if** $npl(H_1.root.right) > npl(H_1.root.left)$ **then**
11. prohoď levého a pravého syna $H_1.root$
12. **end if**
13. $npl(H_1.root) \leftarrow npl(H_1.root) + 1$
14. **return** H_1

Operace Insert(H, x, a) $T(n) = O(1)$

Vytvoří jednoprvkovou leftist haldu H', která obsahuje prvek x s váhou a, a zavolá Merge(H, H').

Operace Mimum(H) $T(n) = O(1)$

Vrátí klíč v kořeni H.

Operace Delete-Minimum(H) $T(n) = O(1)$

Označí za H_1 levý podstrom kořene H a za H_2 pravý podstrom kořene H. Následně zavolá Merge(H_1, H_2).

Operace Oprav(H, v) $T(n) = O(\log(n))$

Operace opraví haldu v případě, že po operacích Increase, Decrease a nebo Delete bude porušena podmínka npl(pravý) $\leq$ npl(levý). Zároveň dojde k utrhnutí vrcholu v.

1. **procedure** Oprav(H,v):
2. $t \leftarrow otec(v)$, $npl(t) \leftarrow 0$
3. **if** $v \neq t.right$ **then**
4. $t.left \leftarrow t.right$
5. **end if**
6. **while** $npl(t)$ se zmenšilo a t není kořen ▷ Včetně zmenšení npl na řádku 2
7. $t \leftarrow otec(t)$
8. **if** $npl(t.right) > npl(t.left)$ **then**
9. vyměň $t.left$ a $t.right$
10. **end if**
11. $npl(t) \leftarrow npl(t.pravy) + 1$
12. **end while**

Operace Decrease(H,x,a)	$T(n) = O(\log(n))$
Operace sníží hodnotu vrcholu x a zavolá metodu Oprav(H, x). Nakonec spojí podstrom x se zbytkem haldy. 1. **procedure** Decrease(H,x,a): 2. H_1 = podstrom H určený vrcholem x 3. $w(x) \leftarrow w(x) - a$ 4. $H_2 \leftarrow Oprav(H, x)$ 5. $H \leftarrow Merge(H_1, H_2)$	

Operace Increase(H,x,a)	$T(n) = O(\log(n))$

Operace Delete(H, x)	$T(n) = O(\log(n))$
Operace nejprve provede merge levého a pravého podstromu vrcholu x. Potom zavolá metodu Oprav(H, x), čímž ze stromu odstraní podstrom určený vrcholem x. Nakonec spojí zbytek haldy H s haldou, která vznikla spojením levého a pravého podstromu x. 1. **procedure** Delete(H,x): 2. $H_1 \leftarrow$ podstrom H určený levým synem vrcholu x 3. $H_2 \leftarrow$ podstrom H určený pravým synem vrcholu x 4. $H_3 \leftarrow \text{Merge}(H_1, H_2)$ 5. $H_4 \leftarrow \text{Oprav}(H, x)$ 6. $H \leftarrow \text{Merge}(H_3, H_4)$	

Binomiální haldy

Definice 3.7 (***Binomiální strom***): *Binomiální strom B_i je kořenový strom rekurentně definovaný jako strom sestávající z podstromů B_0, B_1,... atd. kde strom B_0 je jednoprvkový strom a $\forall i > 0$ platí, že strom H_i vznikne připojením stromu $H_i - 1$ ke kořeni jiného disjunktního stromu $H_i - 1$. Pro každý binomiální strom B_k platí:* **D**

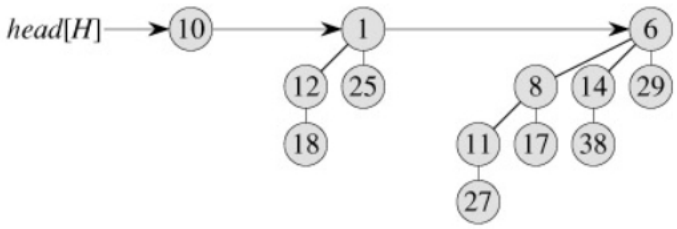
1) strom má 2^k uzlů
2) výška stromu je k
3) na i-té hladině je právě $\binom{k}{i}$ uzlů (odtud název struktury)
4) vrchol s nejvyšším stupněm ve stromu, což je k, je kořen

□

Definice 3.8 (***Binomiální halda***): *Binomiální halda H je množina binomiálních stromů, které splňují následující vlastnosti:* **D**

1) klíč uzlu je menší než klíč jeho rodiče (haldová podmínka)
2) pro každé nezáporné k, $\exists$ v H nejvýše jeden binomiální strom jehož kořen má stupeň k (počet synů k).

□



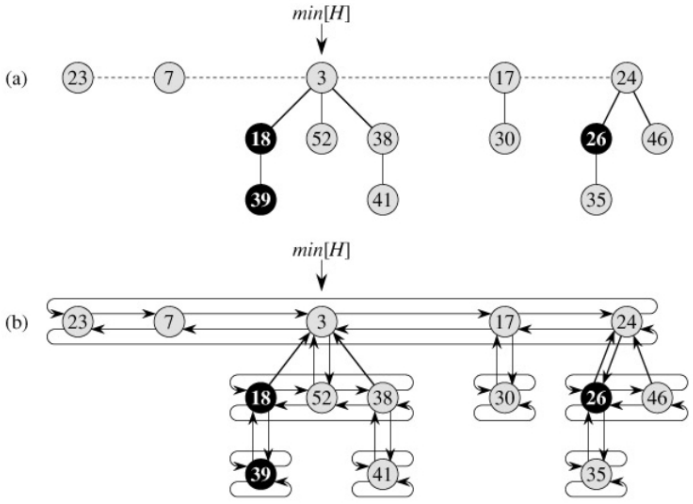
Obrázek 3.1.1. Příklad haldy H se 13 vrcholy. Binární reprezentace čísla 13 je ¡1101¿. H tedy obsahuje binomiální stromy B_3, B_2 a B_0 s počty uzlů 8,4 a 1.

Operace Minimum(H)	$T(n) = O(1)$
Vrátí hodnotu proměnné, na kterou ukazuje ukazatel aktuálního minima.	

Fibonacciho haldy

D **Definice 3.9** (***Fibonacciho halda***): *Fibonacciho haldu tvoří les, reprezentovaný spojovým seznamem částečně uspořádaných stromů, který vznikl aplikací haldových operací na prázdném lese. Některé vrcholy mohou být poznačeny (viz operace DECREASE). Držíme si pointer na vrchol stromu, který obsahuje globální minimum. (neumíme to popsat lépe)* □

Fibonacciho haldu reprezentujeme pomocí kruhových obousměrných spojáků. Každý vrchol má odkaz na svého souseda vpravo a vlevo, na jedno ze svých dětí a na svého rodiče. Sousedem nejlevějšího vrcholu je nejpravější vrchol (kruhový spoják), vrchol může být sám sobě sousedem. Vrcholy si navíc pamatují svůj stupeň (tzn. počet přímých dětí) a to, zda byly označkované. Kořeny jednotlivých stromů v lese jsou spojeny do spojáku a navíc si držíme ukazatel na globální minimum (které je vždy v kořeni nějakého stromu).



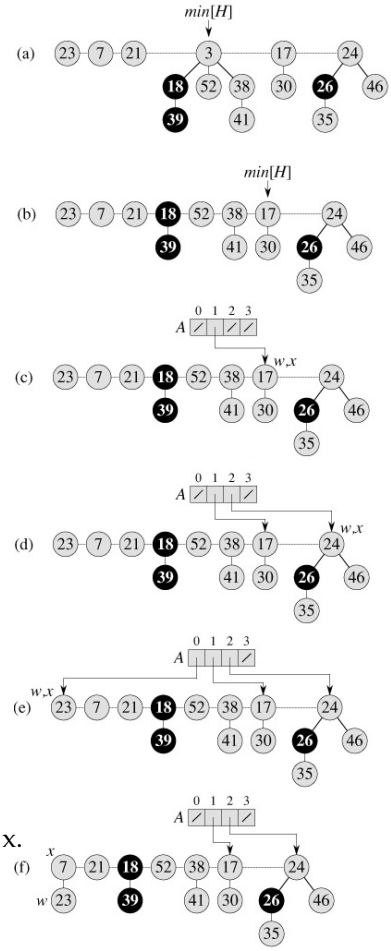
Operace Insert(H,x)	$T(n) = O(1)$
Přidá jednoprvkovou haldu do lesa a upraví ukazatel na aktuální minimum.	

Operace Minimum(H)	$T(n) = O(1)$
Vrátí hodnotu proměnné, na kterou ukazuje ukazatel aktuálního minima.	

Operace Merge(H_1, H_2)	$T(n) = O(1)$
Spojí kruhové spojáky lesů H_1 a H_2. Poté porovná Min(H_1) a Min(H_2), a nastaví podle toho nový pointer na minimum.	

Operace Delete-Minimum(H) $T(n) = O(\log(n))$

Vrchol obsahující minimum je ze stromu odstraněn a jeho děti jsou zařazeny jako nové stromy do lesa tvořícího haldu a pointer na minimum je pověšen na náhodný vrchol. Následně je halda konsolidována. Při konsolidaci slijeme postupně stromy tak, aby ve výsledném lese byl nejvýše jeden strom s daným stupněm (to si hlídám v pomocném poli A). $D(n) = \lfloor \log(n) \rfloor$.




```
1. procedure FIB-HEAP-EXTRACT-MIN(H):
2.    z ← min[H]
3.    if z ≠ NIL then
4.       for each child x of z do:
5.          add x to the root list of H
6.          p[x] ← NIL
7.       end for
8.       remove z from the root list of H
9.       if z = right[z] then
10.         min[H] ← NIL
11.      else
12.         min[H] ← right[z]
13.         CONSOLIDATE(H)
14.      end for
15.      n[H] ← n[H] - 1
16.   end if
17.   return z
```

```
1. procedure CONSOLIDATE(H):
2.    for i ← 0 to D(n[H]) do:
3.       A[i] ← NIL
4.    end for
5.    for each node w in the root list of H do:
6.       x ← w
7.       d ← degree[x]
8.       while A[d] ≠ NIL do:
9.          y ← A[d]    ▷ y má stejný stupeň jako x.
10.         if key[x] > key[y] then
11.            exchange x ↔ y
12.         end if
13.         FIB-HEAP-LINK(H, y, x)
14.         A[d] ← NIL
15.         d ← d + 1
16.      end while
17.      A[d] ← x
18.   end for
19.   min[H] ← NIL
20.   for i ← 0 to D(n[H]) do
21.      if A[i] ≠ NIL then
22.         add A[i] to the root list of H
23.         if min[H] = NIL or (key[A[i]] < key[min[H]]) then
24.            min[H] ← A[i]
25.         end if
26.      end if
27.   end for
```

```
1. FIB-HEAP-LINK(H, y, x)
2.    remove y from the root list of H
3.    make y a child of x, incrementing degree[x]
4.    mark[y] ← FALSE
```

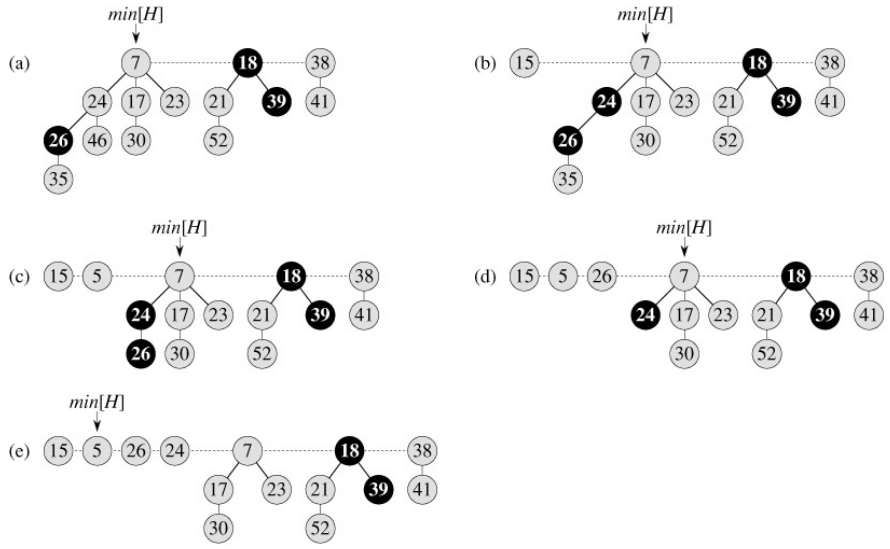
Operace Decrease(H,x, nová hodnota) $T(n) = O(1)$

Sníží hodnotu vrcholu x. Pokud se tím poruší podmínka uspořádání, tak procedura utrhne podstrom s kořenem x a celý ho vloží jako nový strom do lesa. Zároveň označkuje rodiče x, a pokud jsme mu právě utrhli už druhé dítě (což poznáme tak, že už byl označkovaný předtím), tak s ním provedeme rekurzivně to samé co s x.

```
1. procedure FIB-HEAP-DECREASE-KEY(H, x, k):
2.    key[x] ← k
3.    y ← p[x]
4.    if y ≠ NIL and key[x] ¡ key[y] then
5.       CUT(H, x, y)
6.       CASCADING-CUT(H, y)
7.    end if
8.    if key[x] < key[min[H]] then
9.       min[H] ← x
10.   end if
```

```
1. procedure CUT(H, x, y):
2.    remove x from the child list of y, decrementing degree[y]
3.    add x to the root list of H
4.    p[x] ← NIL
5.    mark[x] ← FALSE
```

```
1. procedure CASCADING-CUT(H, y):
2.    z ← p[y]
3.    if z ≠ NIL then
4.       if mark[y] = FALSE then
5.          mark[y] ← TRUE
6.       else
7.          CUT(H, y, z)
8.          CASCADING-CUT(H, z)
9.       end if
10.   end if
```



Operace Delete(H,x)	$T(n) = O(\log n)$
Protože neumíme x efektivně najít, musíme na něj dostat pointer. Provedeme Decrease(H,x,$-\infty$) a následně Delete-Minimum(H).	

3.1.3 Trie

3.1.4 B-stromy a jejich varianty

B-stromy

D **Definice 3.10** (***B strom***): □

D **Definice 3.11** (***B+ strom***): □

D **Definice 3.12** (***B* strom***): □

Operace Search(T, k)	$T(n) = O(\log_t(n))$

Operace Insert(T, k)	$T(n) = O(t \log_t(n))$

3.2 Hašování (řešení kolizí), univerzální hašování, perfektní hašování

3.2.1 Hašování (řešení kolizí)

D **Definice 3.13** (***Základní předpoklady hašování***):

1) Hašovací funkce h je rychle spočitatelná – budeme předpokládat, že hodnota $h(x)$ se spočítá v $O(1)$.
2) Hašovací funkce h rozděluje universum U rovnoměrně, tj. $-1 \leq \left|h^{-1}(i)\right| - \left|h^{-}1(j)\right| \leq 1$ pro všechna $i, j \in \{0, 1, \ldots, m-1\}$.
3) Funkce S je náhodně vybraná z univerza U podle rovnoměrného rozdělení, tj. pro každé n platí, že každá množina S, $|S| = n$, je vybrána s pravděpodobností $\frac{1}{\binom{|U|}{n}}$.
4) Každý prvek z U má stejnou pravděpodobnost být argumentem operace.

□

Hašování se separovanými řetězci

D **Definice 3.14** (***Hašování se separovanými řetězci***): *Datová struktura pro hašování se separovanými řetězci sestává z pole A délky m, kde každý prvek pole i představuje ukazatel na spojový seznam klíčů x, pro které platí, že $h(x) = i$.* □

Operace MEMBER(x)	$T(n) = O\left(\frac{n}{m}\right)$
Nejprve spočteme $i = h(x)$ a následně zjistíme, zda se x vyskytuje ve spojovém seznamu i.	

Operace INSERT(x)	$T(n) = O\left(\frac{n}{m}\right)$
Nejprve spočteme $i = h(x)$ a následně přidáme x do spojového seznamu i.	

Operace DELETE(x)	$T(n) = O\left(\frac{n}{m}\right)$
Nejprve spočteme $i = h(x)$ a následně odstraníme x ze spojového seznamu i.	

Definice 3.15 (***Hašování s uspořádanými separovanými řetězci***): *Jde o modifikaci hašování se separovanými řetězci, kdy jsou spojové seznamy uspořádané.* □ **D**

Poznámka: *Tato datová struktura nezlepší počet testů v případě, že prvek $x \in S$, ale zlepší počet testů v případě, že $x \notin S$, protože můžeme vyhledávání ukončit dříve.*

Hašování v tabulce

Poznámka: *Hašování se separovanými řetězci je neefektivní při práci s pamětí, protože musíme alokovat jak pole A, tak prvky spojových seznamů. Spojový seznam jde reprezentovat přímo v tabulce A.*

Poznámka: *Všechny metody pro hašování v tabulce potřebují proceduru, která efektivně vrací volné řádky v tabulce.*

Definice 3.16 (***Hašování s přemisťováním***): *Datová struktura pro hašování s přemisťováním sestává z tabulky A, která má m řádků a sloupce klíč, next a prev. Pomocí hodnot ve sloupcích next a prev jsou prvky v jednom řetězci uspořádány do dvojsměrného seznamu řádků. V případě, že pro dané i řetězec již existuje, je nový prvek přidán na jeho konec.* □ **D**

Operace MEMBER(x)	$T(n) =$
Spočteme $i = h(x)$. Buď na daném řádku nic není, pak $x \notin S$, nebo $prev \neq NIL$, pak v S doposud nebyl hašovaný řetězec i (a tedy $x \notin S$). Pokud $prev = NIL$, pak na řádku i začíná řetězec, který projdeme.	

```
1.  procedure MEMBER(X):
2.    i ← h(x)
3.    if A[i][prev] ≠ NIL then return x ∉ S end if
4.    while i ≠ NIL do:
5.      if A[i][key] = x then
6.        return x ∈ S
7.      end if
8.      i ← A[i][next]
9.    end while
10.   return x ∉ S
```

Operace INSERT(x)	$T(n) =$
Při vkládání může dojít ke třem situacím. Buď řetězec i ještě nezačal. Pak je buď řádek i prázdný, a můžeme tam vložit x, nebo je řádek obsazen prvkem y z jiného řetězce (což poznáme podle neprázdného sloupce prev). V takovém případě y přesuneme na volný řádek a opravíme ukazatele na něj. Ve třetí situaci řetězec i již v tabulce je. V takovém případě vložíme x na volný řádek, který napojíme na konec řetězce i.	

Operace DELETE(x)	$T(n) =$
Při mazání nejprve najdeme x v tabulce. Pokud x není prvním prvkem řetězce i, pak ho smažeme a přepojíme pointery. Pokud je prvním prvkem řetězce i, pak na jeho místo přesuneme prvek $y = A[i][next]$.	

řádek	klíč	next	prev
0			
1	1	9	
2			
3	73	6	
4			
5	161		8
6	53		3
7	7		
8	11	5	9
9	141	8	1

řádek	klíč	next	prev
0			
1	1	9	
2			
3	73	6	
4	11	5	9
5	161		4
6	53		3
7	7		
8	28		
9	141	4	1

Tabulka 3.2.1. Tabulka pro hašování s přemisťováním, která vznikla posloupností operace INSERT pro prvky 1, 141, 11, 73, 53, 7 a 161. Byla použita hašovací funkce $h(x) = x \bmod 10$. Tabulka vpravo zachycuje stav po vložení prvku 28.[1]

D **Definice 3.17** (***Hašování se dvěma ukazateli***): *Metoda hašování se dvěma ukazateli je variantou na hašování s přemisťováním. Rozdílem je to, že při hašování se dvěma ukazateli jsou prvky řetězce i pouze v jednosměrném spojovém seznamu, tedy nepotřebujeme sloupeček prev. Řetězec i začíná na řádku* $A[i][begin]$. □

Operace MEMBER(x)	$T(n) =$
Spočteme $i = h(x)$. Pokud $A[i][begin] = NIL$, pak $x \notin S$. V opačném případě prohledáme seznam začínající na řádku $A[i][begin]$	

Operace INSERT(x)	$T(n) =$
Spočteme $i = h(x)$. Pokud $A[i][begin] = NIL$, pak doposud neexistuje řetězec i. Pokud je řádek i volný, tak umístíme x na řádek i a nastavíme $A[i][begin] = i$. V opačném případě vložíme x na volný řádek j a nastavíme $A[i][begin] = j$. Pokud řetězec i již existoval, pak x umístíme na volný řádek, který připojíme na konec řetězce i.	

Operace DELETE(x)	$T(n) =$
Spočteme $i = h(x)$. Pokud $A[i][begin] = NIL$, pak skončíme. V opačném případě prohledáme seznam začínající na řádku $A[i][begin]$ a najdeme x. Pokud je x prvním prvkem řetězce, tedy je na řádku $A[i][begin]$, tak nastavíme $A[i][begin] = A[i][next]$. V opačném případě pouze odstraníme x ze spojového seznamu tím, že jemu předcházející prvek na řádku j přepojíme na následující prvek ($A[j][next] = A[i][next]$).	

[1] KOUBEK, Václav a Alena KOUBKOVÁ. Datové struktury I. Praha: MATFYZPRESS, vydavatelství Matematicko-fyzikální fakulty Univerzity Karlovy v Praze, 2011. ISBN 978-80-7378-166-8.

řádek	key	next	begin
0			
1	1	9	1
2			
3	73	7	3
4			
5	161		
6	7		
7	53		6
8	11	5	
9	141	8	

řádek	key	next	begin
0			
1	1	9	1
2			
3	73	7	3
4	28		
5	161		
6	7		
7	53		6
8	11	5	4
9	141	8	

Tabulka 3.2.2. Tabulka pro hašování se dvěma ukazateli, která vznikla posloupností operace INSERT pro prvky 1, 141, 11, 73, 53, 7 a 161. Byla použita hašovací funkce $h(x) = x \bmod 10$. Tabulka vpravo zachycuje stav po vložení prvku 28.[2]

Srůstající hašování

Poznámka: *V předchozích metodách se řetězce nemíchaly, tedy pro každý řetězec i a j, $i \neq j$ platilo, že $\{x \mid x \text{ je v řetězci } i\} \cap \{x \mid x \text{ je v řetězci } j\} = \emptyset$. Tato vlastnost ve srůstajícím hašování není zachovaná.*

Definice 3.18 (***Standardní srůstající hašování (LISCH a EISCH)***): *Podobně jako u předchozích metod obsahuje tabulka spojový seznam hašovaných prvků, realizovaný pomocí ukazatele next. Řetězce však nejsou separované, prvky se přidávají do řetězce, který prochází řádkem i. V případě metody LISCH (late-insertion standard coalesced hashing) se přidává za poslední prvek řetězce, zatímco v pčípadě metody EISCH (early-insertion standard coalesced hashing) se přidává buď přímo na řádek i (pokud je volný) nebo na volný řádek, který je do řetězce napojen hned za i.* □ **D**

Operace MEMBER(x)	$T(n) =$
Spočteme $i = h(x)$. Ve spojovém seznamu začínajícím na řádku i nalezneme x.	

Operace INSERT-LISCH(x)	$T(n) =$
Spočteme $i = h(x)$. Pokud je řádek i prázdný, vložím na něj x. V opačném případě vložím x na volný řádek a napojím na něj konec řetězce, který prrochází řádkem i	

Operace INSERT-EISCH(x)	$T(n) =$
Spočteme $i = h(x)$. Pokud je řádek i prázdný, vložím na něj x. V opačném případě vložím x na volný řádek j a zařadím ho do řetězce, který prochází řádkem i, hned za řádek i ($A[j][next] = A[i][next], A[i][next] = j$)	

Poznámka: *Efektivní algoritmus pro operaci DELETE není znám.*

[2] Koubek, Koubková: op.cit.

řádek	**key**	**next**
0		
1	1	9
2		
3	73	6
4		
5	7	
6	53	
7	161	5
8	11	7
9	141	8

LISH		
řádek	**key**	**next**
0		
1	1	9
2		
3	73	6
4	28	
5	7	4
6	53	
7	161	5
8	11	7
9	141	8

EISCH		
řádek	**key**	**next**
0		
1	1	9
2		
3	73	6
4	28	7
5	7	
6	53	
7	161	5
8	11	4
9	141	8

Tabulka 3.2.3. Tabulka pro standardní srůstající hašování, levá tabulka vznikla posloupností operací INSERT pro prvky 1, 141, 11, 73, 53, 161 a 7, pokud byla užita metoda LISCH, nebo 1, 161, 11, 73, 53, 7 a 141, pokud byla užita medota EISCH. Byla použita hašovací funkce $h(x) = x \bmod 10$. Zbývající tabulky zachycují situaci po vložení klíče 28 pro metody LISCH a EISCH.[3]

D **Definice 3.19** (***Srůstající hašování (LICH, VICH a EICH)***): *Metody LICH (late-insertion coalesced hashing) a EICH (early-insertion coalesced hashing) jsou v zásadě totožné, jako metody LISCH a EISCH. Rozdílem je to, že vedle hašovací tabulky je k dispozici prostor mimo ni, tzv. „sklep". Metoda, která vrací nový volný řádek, vrací primárně řádky ze sklepa. Metoda VICH (varied-insertion coalesced hashing) je kombinací metod LICH a EICH. Pokud existují prvky řetězce, které jsou ve sklepě, pak nové řádky v řetězci zařadí za poslední prvek ve sklepě. Jinak zařadí prvek x na konec řetězce.* □

Poznámka: *Operace MEMBER je pro metody LICH, EICH a VICH totožná s operací MEMBER pro LISCH a EISCH. Operace INSERT pro metody LISCH a LICH (resp. pro EISCH a EICH) jsou totožné. Efektivní algoritmus pro operaci DELETE není znám ani pro metody LICH, EICH a VICH.*

Operace INSERT-VICH(x)	$T(n) =$
Vkládání probíbá podobně jako u INSERT-LICH resp. INSERT-EICH. Odlišná je pozice v řetězci, kam se zařadí vkládaný prvek. Pokud je prvek vkládán na řádek $i = h(x)$, a nebo pokud je vkládán do sklepa, tak se vkládá, stejně jako u metody LICH, na konec řetězce i. V opačném případě (který nastane, když už ve sklepě není místo) je x zařazeno jako následník posledního prvku řetězce, který je ve sklepě.	

řádek	**key**	**next**
0		
1	1	10
2		
3	73	6
4		
5	7	
6		
7	161	5
8	11	7
9		
10	141	8
11	53	

LICH		
řádek	**key**	**next**
0		
1	1	10
2		
3	73	11
4	28	9
5	7	4
6		
7	161	5
8	11	7
9	31	
10	141	8
11	53	

VICH		
řádek	**key**	**next**
0		
1	1	10
2		
3	73	11
4	28	7
5	7	
6		
7	161	5
8	11	4
9	31	8
10	141	9
11	53	

EICH		
řádek	**key**	**next**
0		
1	1	9
2		
3	73	11
4	28	7
5	7	
6		
7	161	5
8	11	4
9	31	10
10	141	8
11	53	

[3] Koubek, Koubková: op.cit.

Tabulka 3.2.4. Tabulka pro srůstající hašování, levá tabulka vznikla posloupností operací INSERT pro prvky 1, 73, 141, 53, 11, 161 a 7, pokud byla užita metoda LICH, nebo 1, 73, 161, 53, 11, 141 a 7, pokud byla užita medota EISCH, a konečně 1, 73, 141, 53, 161, 11 a 7, pokud byla použita metoda VICH. Byla použita hašovací funkce $h(x) = x \bmod 10$. Zbývající tabulky zachycují situaci po vložení klíčů 28 a 31 pro metody LICH, VICH a EICH.[4]

Ostatní metody

Poznámka: *Zbývající dvě metody zcela opouští řetězce prvků se stejnou hodnotou hašovací funkce. Namísto toho nejprve najdou řádek $i = h(x)$ a v případě, že dojde ke kolizi, tak zařadí prvek na konec shluku plných řádků, ve kterém se nachází řádek i.*

Definice 3.20 (***Hašování s lineárním přidáváním***): *Při hašování s lineárním přidáváním jsou klíče vkládány do tabulky A s m řádky, a to buď na řádek $i = h(x)$, pokud je volný, a nebo na nejbližší následující volný řádek (následníkem posledního řádku je první řádek).* □ **D**

Operace MEMBER(x)	$T(n) =$
Nejprve nalezneme $i = h(x)$. Pokud se x nachází na řádku i, pak $x \in S$. V opačném případě projdeme postupně řádky pole A číslo $(i + 1) \bmod m$ až $(i + m) \bmod m$. Průchod polem zastavíme, pokud narazíme na x (pak $x \in S$), pokud narazíme na prázdný řádek (pak $x \notin S$), a konečně pokud projdeme všechny řádky (pak také $x \notin S$).	

Operace INSERT(x)	$T(n) =$
Nejprve nalezneme $i = h(x)$. Pokud je řádek i prázdný, tak do něj vložíme x. V opačném případě projdeme postupně řádky číslo $(i + 1) \bmod m$ až $(i + m) \bmod m$. Pokud narazíme na volný řádek, dřív než nám řádky dojdou, tak na tento řádek vložíme x.	

Poznámka: *Efektivní algoritmus pro operaci DELETE není znám.*

řádek	key
0	
1	1
2	11
3	73
4	141
5	161
6	53
7	7
8	
9	

řádek	key
0	
1	1
2	11
3	73
4	141
5	161
6	53
7	7
8	35
9	

Tabulka 3.2.5. Tabulka pro hašování s lineárním přidáváním, vzniklá posloupností operací INSERT pro prvky 1, 11, 73, 141, 161, 53 a 7. Byla použita hašovací funkce $h(x) = x \bmod 10$. Tabulka vpravo vznikla přidáním prvku 35.[5]

Definice 3.21 (***Dvojité hašování***): *Dvojité hašování je modifikací hašování s lineárním přidáváním. Při hledání volného řádku však algoritmus neprochází tabulku s krokem o velikosti 1 (jako hašování s lineárním přidáváním), ale velikost kroku určuje pomocná hašovací funkce $h_2(x)$.* □ **D**

[4] Koubek, Koubková: op.cit.
[5] Koubek, Koubková: op.cit.

Operace MEMBER(x)	$T(n) =$
Nejprve nalezneme řádek $i = h_1(x)$. Pokud řádek i obsahuje x, pak $x \in S$. V opačném případě projdeme postupně řádky pole A číslo $(i+1) \bmod m$ až $(i+m) \bmod m$, s krokem $h = h_2(x)$. Průchod polem zastavíme, pokud narazíme na x (pak $x \in S$), pokud narazíme na prázdný řádek (pak $x \notin S$), a konečně pokud projdeme všechny řádky (pak také $x \notin S$).	

Operace INSERT(x)	$T(n) =$
Nejprve nalezneme $i = h_1(x)$. Pokud je řádek i prázdný, tak do něj vložíme x. V opačném případě projdeme postupně řádky číslo $(i+1) \bmod m$ až $(i+m) \bmod m$, s krokem $h = h_2(x)$. Pokud narazíme na volný řádek, dřív než nám řádky dojdou, tak na tento řádek vložíme x.	

Poznámka: *Efektivní algoritmus pro operaci DELETE opět neznáme.*

řádek	key
0	11
1	1
2	
3	73
4	141
5	7
6	53
7	161
8	
9	

řádek	key
0	11
1	1
2	35
3	73
4	141
5	7
6	53
7	161
8	
9	

Tabulka 3.2.6. Tabulka pro dvojité hašování, vzniklá posloupností operací INSERT pro prvky 1, 73, 53, 141, 161, 11 a 7. Byly použita hašovací funkce $h_1(x) = x \bmod 10$ a $h_2(x) = 1 + 2(x \bmod 4)$ (když $x \bmod 4 \in \{0,1\}$), resp. $h_2(x) = 3 + 2(x \bmod 4)$, když $x \bmod 4 \in \{2,3\}$. Tabulka vpravo vznikla přidáním prvku 35.[6]

3.2.2 Univerzální hašování

Motto: Nikdo nám negarantuje pěkné vstupní hodnoty, které budou mít alespoň trochu nádhodné rozdělení. Tak si náhodnost vyrobíme sami tím, že si náhodně zvolíme hašovací funkci z katalogu, a až pak začneme hašovat.

D **Definice 3.22** (***C-univerzální systém funkcí***): *Něchť U je univerzum. Systém funkcí $\mathcal{H} = \{h_i \mid i \in I\}$, $h_i \colon U \to \{0, \ldots, m-1\}$ se nazývá c-univerzální (kde $c \in \mathbb{R}^+$), pokud*

$$\forall x, y \in U, x \neq y \colon \left|\{i \in I \mid h_i(x) = h_i(y)\}\right| \leq \frac{c\,|I|}{m}$$

□

V **Věta 3.23**: *Mějme c-univerzální systém funkcí $\mathcal{H} = \{h_i \mid i \in I\}$, $h_i \colon U \to \{0, \ldots, m-1\}$. Pak pro každé dva různé prvky $x, y \in U$ je pravděpodobnost, že pro náhodně zvolené $i \in I$ s rovnoměrným rozdělením platí $h_i(x) = h_i(y)$, je nejvýše rovna $\frac{c}{m}$.*

Důkaz: Máme dvojici klíčů x a y, chceme ukázat, že náhodně zvolená funkce i je hašuje na stejnou hodnotu s pravděpodobností menší než $\frac{c}{m}$. Množina všech funkcí, které x hašují na stejnou hodnotu

[6] Koubek, Koubková: op.cit.

jako y je $I' = \{i \in I \mid h_i(x) = h_i(y)\}$. Pravděpodobnost, že ze všech funkcí z I zvolíme funkci, která se trefí právě do I' je $\frac{|I'|}{|I|}$:

$$\frac{\{i \in I \mid h_i(x) = h_i(y)\}}{|I|} \overset{\text{z definice } \frac{c}{m}|I|}{\leq} \frac{\frac{c}{m}|I|}{|I|} = \frac{c}{m}.$$

♡

Věta 3.24 (***Existence c-univerzálního systému funkcí***): *Existuje c-univerzální systém funkcí.* **V**

Důkaz: Idea důkazu: nejprve si takový systém navrhneme a pak o něm dokážeme, že je c-univerzání.

Definujme c-univerzální systém funkcí $\mathcal{H} = \{h_{a,b} \mid (a,b) \in U \times U\}$, definovaných předpisem

$$h_{a,b}(x) = ((ax + b) \bmod N) \bmod m, \qquad (c\textbf{-univerzální systém})$$

kde $N = |U|$. $\mathcal{H}$ obsahuje pouze lineární funkce, tedy splňuje požadavek, aby funkce šlo rychle vyčíslit. Nyní dokážeme c-univerzálnost. Mějme $x, y \in U$, $x \neq y$ a hledáme $a, b \in U$ pro která $h_{a,b}(x) = h_{a,b}(y)$. Pokud taková dvojice a, b existuje, tedy platí $h_{a,b}(x) = h_{a,b}(y)$, pak musí existovat nějaké $i \in \{0, \ldots, m-1\}$ a $r, s \in \left\{0, \ldots, \left\lceil \frac{N}{m} \right\rceil - 1\right\}$ tak, že platí:

$$\begin{aligned} (ax + b \equiv i + rm) \mod N \\ (ay + b \equiv i + sm) \mod N \end{aligned}$$

Vztahy plynou z toho, že $h_{a,b}(x) = h_{a,b}(y) = i$. Protože je $h_{a,b} = ((ax + b) \bmod N) \bmod m$, tak nám aplikace $\bmod m$ zlikvidovala členy rm a sm a zůstalo jen i. Protože jsme $\bmod m$ odstranili, musíme si tam tyto členy vrátit. Když budeme x, y, i, r a s považovat za konstanty a a, b za neznámé, tak nalezení všech $(a,b) \in U \times U$, takových, že $h_{a,b}(x) = h_{a,b}(y)$, odpovídá řešení systému lineárních rovnic v tělese $\mathbb{Z}_N$. Této soustavě rovnic odpovídá matice

$$\begin{pmatrix} x & 1 \\ y & 1 \end{pmatrix},$$

která je regulární, protože $x \neq y$, a tedy má jediné řešení. Pro pevně daná x, y, i, r a s tedy existuje právě jedno řešení, čili i dvojice (a,b). Pro daná x a y může i nabývat až m hodnot a r, s nabývají až $\left\lceil \frac{N}{m} \right\rceil$ hodnot.

Celkem tedy platí, že pro každé dva prvky $x, y \in U$, takové že $x \neq y$ existuje $m\left(\left\lceil \frac{N}{m} \right\rceil\right)^2$ dvojic $(a,b) \in U \times U$. Protože velikost $\mathcal{H}$ je dána počtem všech dvojic (a,b), je rovna $|U \times U| = N^2$. Z toho plyne:

$$m\left(\left\lceil \frac{N}{m} \right\rceil\right)^2 = m\left(\left\lceil \frac{N}{m} \right\rceil\right)^2 \frac{\left(\frac{N}{m}\right)^2}{\left(\frac{N}{m}\right)^2} = \left(\left\lceil \frac{N}{m} \right\rceil\right)^2 \frac{mN^2}{m^2\left(\frac{N}{m}\right)^2} = \frac{\left(\left\lceil \frac{N}{m} \right\rceil\right)^2}{\left(\frac{N}{m}\right)^2} \frac{N^2}{m} \overset{*}{=} \frac{\left(\left\lceil \frac{N}{m} \right\rceil\right)^2}{\left(\frac{N}{m}\right)^2} \frac{|I|}{m}$$

Rovnost označená hvězdičkou plyne z toho, že $|\mathcal{H}| = |I| = \{(a,b) \mid (a,b) \in U \times U\} = N^2$. Poslední člen však odpovídá definici c-univerzálního systému, kde požadujeme, aby počet funkcí, které pro dané x a y vytváří kolize (tedy $h_{a,b}(x) = h_{a,b}(y)$) byl nejvýše $c\frac{|I|}{m}$. Protože používáme kongruence ($\equiv$), potřebujeme, aby N bylo prvočíslo.

Lze tedy shrnout, že pokud je N prvočíslo, tak systém funkcí $\mathcal{H} = \{h_{a,b} \mid (a,b) \in U \times U\}$ je c-univerzálním systémem funkcí pro $c = \frac{\left(\left\lceil \frac{N}{m} \right\rceil\right)^2}{\left(\frac{N}{m}\right)^2}$. ♡

Definice 3.25 (***Indikátor kolize***): *Mějme c-univerzální systém funkcí $\mathcal{H} = \{h_i \mid i \in I\}$. Pro každé $i \in I$ a pro každé dva prvky $x, y \in U$ definujeme indikátor kolize při použití hašovací funkce h_i jako* **D**

$$\delta_i(x,y) = \begin{cases} 1 & \text{když } x \neq y \text{ a } h_i(x) = h_i(y) \\ 0 & \text{jinak} \end{cases}$$

a dále definujeme pro danou množinu $S \subseteq U$, $x \in U$ a $i \in I$

$$\delta_i(x,S) = \sum_{y \in S} \delta_i(x,y).$$

□

Poznámka: *Proměnná $\delta_i(x, S)$ představuje počet kolizí mezi x a prvky množiny S při použití h_i, neboli délka řetězce na adrese $h_i(x)$ bez prvku x.*

V **Věta 3.26** (***Očekávané časy operací pro univerzální hašování***): *Očekávaný čas operací **MEMBER**, **INSERT** a **DELETE** v c-univerzálním hašování je $O(1 + c\alpha)$, kde $\alpha = \frac{|S|}{m}$ je faktor naplnění. Očekávaný čas provedení posloupnosti n operací **MEMBER**, **INSERT** a **DELETE** aplikované na prázdnou tabulku je $O\left(n\left(1 + \frac{c}{2}\alpha\right)\right)$*

Důkaz: Pro pevně danou množinu $S \subseteq U$ a pro pevné $x \in U$ sečteme $\delta_i(x, S)$ přes všechna $i \in I$:

$$\sum_{i\in I} \delta_i(x,S) = \sum_{i\in I}\sum_{y\in S} \delta_i(x,y) = \sum_{y\in S}\sum_{i\in I} \delta_i(x,y) = \sum_{y\in S, y\neq x} \left|\{i \in I \mid h_i(x) = h_i(y)\}\right| \leq$$

$$\leq \sum_{y\in S, y\neq x} c\frac{|I|}{m} = \begin{cases} (|S|-1)c\frac{|I|}{m} & \text{když } x \in S \\ |S|\, c\frac{|I|}{m} & \text{když } x \notin S \end{cases}$$

Z toho plyne, že horní odhad očekávané hodnoty $\delta_i(x, S)$ počítaný přes všechny indexy funkcí $i \in I$ je

$$\frac{1}{|I|}\sum_{i\in I} \delta_i(x,S) = \begin{cases} c\frac{|S|-1}{m} & \text{když } x \in S \\ c\frac{|S|}{m} & \text{když } x \notin S \end{cases}$$

V čase $O(1)$ najdu přihrádku. Pro dané x můžeme očekávat nejvýše $c\frac{|S|}{m}$ kolizí, to znamená, že příslušný řetízek pro něj bude nejvýše $c\frac{|S|}{m}$ dlouhý. To znamená, že operace ***MEMBER***, ***INSERT*** a ***DELETE*** běží v čase $O(1)$ a $O\left(c\frac{|S|}{m}\right)$. ♡

V **Věta 3.27** (***Dolní odhad c***): *Mějme univerzum U velikosti N a nechť $\mathcal{H}$ je c-univerzální systém hašovacích funkcí do tabulky T velikosti m. Pak $c \geq \frac{N-m}{N-1} > 1 - \frac{m}{N}$.*

Důkaz: Idea: Nejprve si pro nějakou obecnou funkci spočteme velikost množiny všech kolizí, a pak to zobecníme.

Mějme funkci $f: U \to T$, kde U má velikost N a T má velikost m. Definujeme $A = \{(u, v) \mid (u, v) \in U \times U, u \neq v, f(u) = f(v)\}$. Pro $t \in T$ označme $k_t = \left|f^{-1}(t)\right|$, tedy počet všech klíčů, které se hašují do přihrádky t. Velikost A se spočte následujícím postupem (za využití Cauchy-Schwarzovy nerovnosti, tedy pokud máme $b = \sum_{i=0}^{m-1} b_i$, tak platí $\sum_{i=0}^{m-1} b_i(b_i - 1) \geq b\left(\frac{b}{m} - 1\right)$):

$$|A| = \sum_{t\in T} k_t(k_t - 1) \overset{\text{C.S.}}{\geq} N\left(\frac{N}{m} - 1\right) = N\left(\frac{N-m}{m}\right),$$

protože $\sum_{t\in T} k_t = N$. Tato nerovnost platí pro jednu hašovací funkci. Pokud vezmeme v úvahu všechny hašovací funkce z $\mathcal{H}$, dostaneme vztah

$$|I|\, N\left(\frac{N-m}{m}\right) \leq \sum_{i\in I} \left|\{(x,y) \in U \times U \mid h_i(x) - h_i(y), x \neq y\}\right| \overset{*}{=} \sum_{\substack{(x,y)\in U\times U \\ x\neq y}} \left|\{i \in I \mid h_i(x) = h_i(y)\}\right| \leq$$

$$\leq \sum_{\substack{(x,y)\in U\times U \\ x\neq y}} c\frac{|I|}{m} \overset{\dagger}{=} N(N-1)c\frac{|I|}{m}.$$

V rovnosti označené hvězdičkou jsme jen prohodili objekt, přes který sčítáme, je jedno zda sčítáme dvojice přes funkce nebo funkce přes dvojice. V rovnosti označené křížkem jsme využili toho, že se ve výrazu přes dvojice nepočítá a všech dvojic s danou vlastností je $N(N-1)$. Tedy

$$|I|\, N\left(\frac{N-m}{m}\right) \leq N(N-1)c\frac{|I|}{m}$$

$$\left(\frac{N-m}{m}\right) \leq c(N-1)$$

$$c \geq \frac{N-m}{N-1} > \frac{N-m}{N} = 1 - \frac{m}{N}.$$

♡

3.2.3 Perfektní hašování

Motto: Když známe klíče dopředu, můžeme jim udělat hašovací funkci na míru.

Definice 3.28 (***Perfektní funkce na množině***): *Když U je univerzum a $S \subseteq U$ jeho podmnožina, pak řekneme, že funkce $f: U \to \{0, 1, \ldots, m-1\}$ je perfektní na množině S, když $f(x) \neq f(y)$ pro každé dva různé prvky $x, y \in S$.* □ **D**

Definice 3.29 (***Formální zadání úlohy perfektního hašování***): *Pro danou množinu $S \subseteq U$ hledáme hašovací funkci h takovou, že* **D**

1) *Pro každé dva prvky $s, t \in S$ takové, že $s \neq t$, platí $h(x) \neq h(y)$ (tedy h je perfektní hašovací funkce na množině S).*
2) *Funkce h hašuje do tabulky s m řádky, kde m je přibližně stejně velké jako $|S|$ (není praktické hašovat do příliš velkých tabulek).*
3) *Funkce h musí být rychle spočitatelná (jinak hašování není rychlé).*
4) *Uložení h nesmí vyžadovat příliš mnoho paměti (proto je nejvýhodnější analytické zadání).*

□

Definice 3.30 (***(N,m,n)-perfektní soubor funkcí***): *Mějme univerzum $U = \{0, 1, \ldots, N-1\}$. Soubor funkcí $\mathcal{H}$ z U do množiny $\{0, 1, \ldots, m-1\}$ se nazývá (N, m, n)-perfektní, když pro každou množinu $S \subseteq U$ takovou, že $|S| = n$, existuje $h \in \mathcal{H}$, perfektní na množině S.* □ **D**

Algoritmus 3.31: Perfektní hašování. Mějme univerzum U, množinu klíčů $S \subseteq U$ a označme $N = |U|$, $n = |S|$. **A**

1) *[První pomocná hašovací funkce]* Pravděpodobnostním algoritmem typu Las Vegas nalezneme prvočíslo q_0, pro které platí, že $q_0 = O(n^2 \log N)$ a hašovací funkce $\phi_{q_0} = x \bmod q_0$ je perfektní funkce na S.
2) *[Pomocná tabulka kvadratické velikosti]* Položíme $S_1 = \left\{\phi_{q_0}(s) \,\middle|\, s \in S\right\}$.
3) *[Druhá pomocná hašovací funkce]* Pravděpodobnostním algoritmem typu Las Vegas nalezneme prvočíslo q_1, pro které platí, že $n(n-1) < q_1 \leq 2n(n-1) + 2$, a dále nalezneme číslo $l \in \{1, \ldots, q_0 - 1\}$ takové, že $h_{l,q_0,q_1}(x) = ((lx) \bmod q_0) \bmod q_1$ je perfektní na $S_1 \subseteq \{0, \ldots, q_0 - 1\}$.
4) *[Druhá pomocná tabulka]*Položíme $S_2 = \{h_{l,q_0,q_1}(s) \mid s \in S_1\}$.
5) *[Třetí pomocná hašovací funkce]* Zkonstruujeme perfektní hašovací funkci $g(x)$:
   (i) Vstupem $g(x)$ jsou klíče z upraveného univerza $U' = \{0, \ldots, q_0 - 1\}$, pro které platí $|U'| = q_0$. Vyjdeme z hašovací funkce $j_k(x) = (kx \bmod q_0) \bmod m$. Pro ni si zadefinujeme proměnnou b_i^k předpisem
   $$b_i^k = \left|\{x \in S \mid j_k(x) = i\}\right|,$$
   neboli kolik prvků při použití hašovací funkce j_k padne do přihrádky i. Pro perfektní funkci očekáváme, že $b_i^k \in \{0, 1\}$.
   (ii) Nalezneme k takové, že pro $m = n$ platí $\sum_{i=0}^{m-1} \left(b_i^k\right)^2 < 3n$.
   (iii) Pro $i = 0, \ldots, m-1$ nalezneme množiny $T_i = \{s \in S_2 \mid j_k(s) = i\}$
   (iv) Pro každé $i = 0, \ldots, m-1$ takové, že $S_i \neq \emptyset$, nalezneme pro $c_i = |T_i|(|T_i| - 1) + 1$ takové k_i, že j_{k_i} je perfektní funkce na T_i, hašující do tabulky velikosti c_i. Pokud je $T_i = \emptyset$, pak položíme $c_i = 0$.
   (v) Pro $i = 0, \ldots, m-1$ definujeme $d_i = \sum_{j=0}^{i-1} c_j$ a pro $x \in U$ označíme $j_k(x) = l$.
   (vi) Položíme $g(x) = d_l + j_{k_l}(x) = d_{j_k(x)} + j_{j_k(x)}(x)$
6) *[Hledaná hašovací funkce]* Definujeme perfektní hašovací funkci $f(x) = g(h_{l,q_0,q_1}(\phi_{q_0}(x)))$, tedy

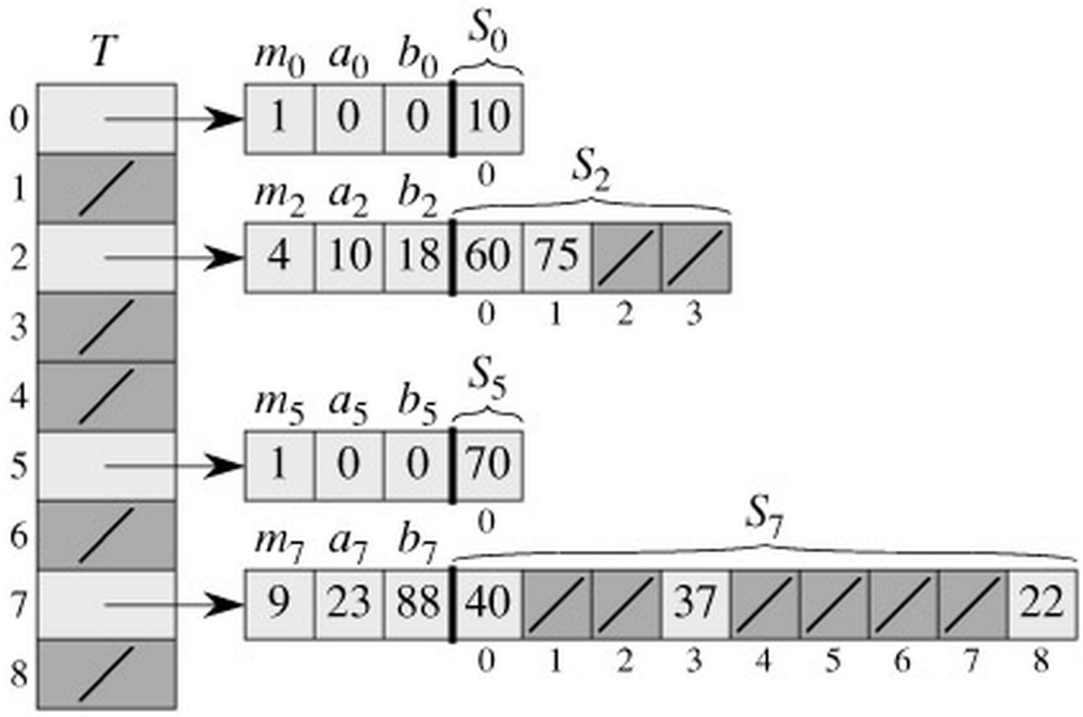
Pro zvolené konstanty q_0, q_1, l, m a tabulku c_i tedy funkce pro perfektní hašování vypadají takto:

$$\begin{aligned} \phi(x) &= x \bmod q_0 \\ h(x) &= (lx \bmod q_0) \bmod q_1 \\ j_k(x) &= (kx \bmod q_0) \bmod m \\ d_i &= \textstyle\sum_{j=0}^{i-1} c_j \\ g(x) &= d_{j_k(x)} + j_{j_k(x)}(x) \\ f(x) &= g(h(\phi(x))) \end{aligned}$$

(perfektní hašování)

Poznámka: *Celý algoritmus stojí na tom, že jsme schopni najít perfektní hašovací funkci, pokud bychom chtěli hašovat do tabulky, jejíž velikost by byla kvadratická k počtu hašovaných klíčů (tedy $O\left(n^2\right)$). To je však poměrně dost, hledáme způsob, jak to hašovat do tabulky velikosti $O(n)$. Toho dosáhneme tak, že hašujeme dvakrát. Nejprve všechna čísla přehašujeme společnou funkcí (kterou náhodně najdeme, ve skriptech se poměrně obšírně dokazuje, že to jde najít poměrně rychle) a v tabulce délky $O(m)$ najdeme odpovídající druhou hašovací funkci, kterou daný klíč přehašujeme do menší pomocné tabulky T_i. Toto dělá popsaná funkce $g(x)$. Hašujeme do prostoru $O(n)$ (což se opět ve skriptech pěkně ukáže), ale potřebujeme na to hašovací funkci zadanou tabulkou o velikosti $O(n)$, což je v rozporu se čtvrtým požadavkem. Proto nejprve univerzum pomocí funkcí h_{l,q_1} a ϕ_{q_0} sešlapeme tak, že tabulky pro funkci g mají velikost jen $O(n \log n + \log \log N)$.*

Zatímco na MIT, kde nemají problém s definicí $g(x)$ tabulkou o velikosti $O(m)$, dělají dvě volání hašovací funkce a hledají v dvojrozměrné tabulce, my děláme jeden dotaz a tabulku máme serializovanou do dlouhého pole (což vyjadřuje člen d_i).



3.3 Třídění ve vnitřní a vnější paměti

Heapsort

Mergesort

Quicksort

Bucketsort

3.4 Dolní odhady pro uspořádání (rozhodovací stromy)

Definice 3.32 (***Rozhodovací strom***): *Mějme třídící algoritmus A, který jako jedinou primitivní operaci s prvky vstupní posloupnosti používá jejich porovnání. Řekneme, že binární strom T, jehož vnitřní vrcholy jsou ohodnoceny porovnáními $a_i \leq a_j$ pro $i, j = 1, \ldots, n, i \neq j$, je rozhodovacím stromem algoritmu A pro n-prvkové posloupnosti, když pro každou permutaci π z n-prvkové množiny platí, že posloupnost porovnávání při třídění permutace π algoritmem A je stejná, jako posloupnost porovnání při průchodu permutace π stromem T.* □ D

Věta 3.33 (***Dolní odhad pro uspořádání***): *Třídící algoritmus, jehož jedinou primitivní operací s prvky vstupní posloupnosti je porovnání, běží nejméně v čase $T(n) = O(n \log(n))$.* V

Důkaz: Třídící algoritmus A je korektní právě když pro každé dvě různé permutace množiny $\{1, \ldots, n\}$ skončí v navzájem různých listech rozhodovacího stromu T. Dolní odhad pro čas běhu algoritmu A v nejhorším případě odpovídá délce nejdelší cesty z kořene do listu v rozhodovacím stromě. Při rovnoměrném rozdělení vstupních posloupností odpovídá dolní odhad pro čas běhu algoritmu A průměrná délka cesty z kořene do listu.

Rozhodovací strom je binárním stromem s $n!$ listy (počet všech možných permutací n-prvkové množiny). Zároveň platí, že pokud má nejdelší cesta z kořene do listů v binárním stromě délku h, tak má strom nejvýše 2^h listů. Tedy:

$$n! \leq 2^h$$

Použijeme odhad faktoriálu z diskrétní matematiky, a sice $n^{\frac{n}{2}} \leq n!$. Z toho vyplývá:

$$2^h \geq n! \geq n^{\frac{n}{2}}$$

$$2^h \geq n^{\frac{n}{2}}$$

$$h \log_2(2) \geq \frac{n}{2} \log_2(n)$$

$$h \geq \frac{1}{2} n \log_2(n)$$

$$h \geq cn \log(n)$$

Tedy platí, že výška rozhodovacího stromu je vyšší než $cn \log(n)$, a tedy třídící algoritmus A, jehož jedinou primitivní operací s prvky vstupní posloupnosti je porovnání, provede nejméně $O(n \log(n))$ operací. ♡

3.5 Dynamizace datových struktur

Definice 3.34 (***Zobecněný vyhledávací problém***): *Mějme univerza U_1, U_2 a U_3. Pak se funkce $f : U_1 \times \mathcal{P}(U_2) \to U_3$ nazývá zobecněným vyhledávacím problémem.* □ D

Příklad 3.35 (***Vyhledávací problémy***): *Následující funkce jsou vyhledávacími problémy:*

- Mějme $U_1 = U_2 = U$, $U_3 = \{0, 1\}$. Operace MEMBER(x, A), která pro množinu $A \subseteq U$ odpoví, zda $x \in A$ (pak MEMBER(x,A) = 1), nebo $x \notin A$ (pak MEMBER(x,A) = 0), je vyhledávací problém.
- Mějme $U_1 = U_2 = E^n$ (Eukleidovský prostor), $U_3 = \mathbb{R}^+$. Operace DISTANCE(x, A), která pro množinu $A \subseteq E^n$ a bod x spočítá vzdálenost bodu x od množiny A je vyhledávací problém.
- Mějme $U_1 = U_2 = E^2$ (Eukleidovská rovina), $U_3 = \{0, 1\}$. Operace IN-CONVEX-HULL(x, A), která pro dané $A \in E^2$ rozhodne, zda je bod x je v konvexním obalu množiny A, je vyhledávacím problémem.

Definice 3.36 (***Statická datová struktura řešící vyhledávací problém***): *Řekneme, že $\mathcal{S}$ je statická datová struktura řešící vyhledávací problém f, když je dán algoritmus, který pro množinu $A \subseteq U_2$ zkonstruuje datovou strukturu $\mathcal{S}(A)$, a algoritmus závislý na A, který pro $x \in U_1$ a $\mathcal{S}(A)$ vyčíslí $f(x, A)$.* □ D

Příklad 3.37: *Příkladem statické datové struktury je například setřízené pole.*

D **Definice 3.38** (***Semidynamická datová struktura řešící vyhledávací problém***): *Statická datová struktura řešící vyhledávací problém f se nazývá semidynamická, když navíc existují dva algoritmy, první pro $x \in U_2$ a pro $\mathcal{S}(A)$ zkonstruuje $\mathcal{S}$-strukturu $\mathcal{S}(A \cup \{x\})$, a druhý vyčíslující $f(y, A \cup \{x\})$ pro každé $y \in U_1$. Tato operace se nazývá INSERT.* □

D **Definice 3.39** (***Dynamická datová struktura řešící vyhledávací problém***): *Semidynamická datová struktura řešící vyhledávací problém f se nazývá dynamická, když navíc existuje existují dva algoritmy, první pro $x \in U_2$ a $\mathcal{S}(A)$ zkonstruuje $\mathcal{S}(A \setminus \{x\})$, a druhý vyčíslující $f(y, A \setminus \{x\})$ pro každé $y \in U_1$. Tato operace se nazývá DELETE.* □

Poznámka: *Cílem je nalézt efektivní metody, jak převést statické datové struktury na semidynamické a dynamické datové struktury.*

D **Definice 3.40** (***Rozložitelný vyhledávací problém***): *Vyhledávací problém je rozložitelný, pokud existuje binární operace $\odot$ na univerzu U_3 taková, že pro $A, B \subseteq U_2, A \cap B = \emptyset$, a pro $x \in U_1$ platí, že*

$$f(x, A) \odot f(x, B) = f(x, A \cup B).$$

□

Příklad 3.41: *Operace MEMBER je rozložitelná, protože platí, že* $\text{MEMBER}(x, A) \vee \text{MEMBER}(x, B) = \text{MEMBER}(x, A \cup B)$. *Stejně tak operace DISTANCE je rozložitelná, protože vzdálenost bodu x od množiny $A \cup B$ je minimem vzdáleností od množin A a B. U konvexního obalu to však neplatí. Pokud $x \in A$, pak bude v konvexním obalu A i $A \cup B$. Pokud je však x mimo A i B, pak není v konvexním obalu ani jedné množiny, ale může být v konvexním obalu $A \cup B$ (pokud je x mezi nimi). Operace IN-CONVEX-HULL tedy není rozložitelná.*

Konstrukce semidynamické struktury

Mějme množinu $A \subseteq U_2$. Budeme pro ni budovat strukturu $\mathcal{R}(A)$ (která bude simulovat $\mathcal{S}(A)$) tak, abychom do ní byli schopni efektivně přidávat nové prvky, tedy abychom byli schopni efektivně vytvořit strukturu $\mathcal{R}(A \cup \{x\})$. Vyjdeme z toho, že $A = \bigcup_i A_i$, a že vyhledávací problém f je rozložitelný, tedy že výsledky volání $f(x, A_i)$, které ke své činnosti využívá předpočítanou strukturu $\mathcal{S}(A_i)$ lze pro různé A_i skládat pomocí operace $\odot$. $\mathcal{R}(A)$ tedy sestavíme tak, že A rozložíme na vhodné množiny A_i, které budou obsahovat zatím vložené prvky x, a pro každou A_i si necháme vytvořit odpovídající $\mathcal{S}(A_i)$. Pokud pak dojde k volání $f(x, A)$ nad strukturou $\mathcal{R}(A)$, tak ve skutečnosti zavoláme $f(x, A_i)$ nad všemi množinami $A_i \subseteq A$ (funkce f ke své činnosti využije uložené struktury $\mathcal{S}(A_i)$) a jejich výsledek složíme.

Množiny A_i musí pro dané A_i být buď prázdné, tedy $A_i = \emptyset$, nebo musí platit $|A_i| = 2^i$. Z toho plyne, že pokud $|A| = n$, pak stačí vzít binární zápis čísla n, a pokud i-tý bit bude 1, pak A_i bude neprázdná.

Abychom efektivně zjistili, zda již x je v A, tedy zda existuje nějaké A_i, pro které platí $x \in A_i$, a tedy existovala struktura $\mathcal{S}(A_i)$ spočtená s ohledem na x, budeme si držet vhodnou vyhledávací strukturu T (např. B-strom, binární vyhledávací strom apod.), ve které si budeme držet dvojice (klíč, pointer na příslušnou množinu A_i).

Operace $\odot$ musí být spočítatelná rychle, tedy vyžadujeme, aby měla čas $O(1)$.

Složitost jednotlivých operací:

- $Q_S(n)$ – čas na vyčíslení $f(x, A)$, pokud $|A| = n$.
- $S_S(n)$ – paměťový prostor potřebný k reprezentaci n-prvkové množiny.
- $P_S(n)$ – čas na vytvoření struktury $\mathcal{S}(A)$ pro $|A| = n$.

Navíc předpokládáme, že $Q_S(n)$, $\frac{S_S(n)}{n}$ a $\frac{P_S(n)}{n}$ jsou neklesající.

Příklad 3.42: *Mějme vyhledávací problém „nalezení nejbližšího bodu v rovině pro zadaný bod“. Pěknou datovou strukturou, která nám toto umožní spočítat je Voronoi diagram, který jsme schopni pro danou*

množinu bodů velmi rychle spočítat ($O(n \log(n))$. Na druhou stranu, když už je spočtený, tak do něj neumíme efektivně přidávat nové body. Bylo by možné pro každý nový bod starý diagram zahodit a spočítat diagram nový. To však není třeba, protože hledání nejbližšího bodu je rozložitelný vyhledávací problém (když si vezmu dvě množiny bodů A, B a spočtu pro každou z nich ten nejbližší bod k danému x, pak pro $A \cup B$ stačí vzít minimum ze spočtených vzdáleností). Body v rovině mi tedy tvoří dohromady množinu A, kterou si můžu rozdělit na vhodné množiny A_i, podle toho, jak mi body přicházely. Pro každou množinu A_i si spočtu její Voronoi diagram (tedy $\mathcal{S}(A_i)$). Po přidání bodu tedy nemusím zahodit vše, co se doposud počítalo, ale přepočítat musím jen malou část prostoru.

Operace Vyčísli(x) $T(n) = O(Q_S(n) \log(n))$

Postupně projde všechny A_i a zavolá na nich $f(x, A_i)$. Následně výsledky spojí pomocí operace $\odot$.

1. **procedure** Vyčísli(x):
2. $i_0 \leftarrow$ libovolné číslo z $\left\{0, \ldots, 1 + \lfloor \log(|A|) \rfloor\right\}$ takové, že $A_{i_0} \neq \emptyset$
3. $res \leftarrow f(x, A_{i_0})$
4. **for each** $i \in \left\{0, \ldots, 1 + \lfloor \log(|A|) \rfloor\right\}, A_i \neq \emptyset, i \neq i_0$ **do:**
5. $res \leftarrow res \odot f(x, A_i)$
6. **end for each**
7. **return** *res*

Operace Insert(x) $T(n) = O\left(\frac{P_S(n)}{n} \log(n)\right)$

Nejprve nalezneme i takové, že $A_i = \emptyset$, a poté do A_i vložíme x a prvky ze všech $A_j, j < i$. Struktury $\mathcal{S}(A_j)$ zahodíme. Operace odpovídá přičtení jedničky k n v binární soustavě.

1. **procedure** Insert(x):
2. **if** $x \in T$ **then return end if**
3. INSERT(T,x)
4. $A_i = \{x\}$
5. $i \leftarrow$ nejmenší i takové, že $A_i = \emptyset$
6. **for each** $j < i$ **do:**
7. $A_i = A_i \cup A_j$
8. $A_j = \emptyset$
9. **delete** $\mathcal{S}(A_j)$
10. **end for**
11. vytvoř $\mathcal{S}(A_i)$

Konstrukce semidynamické struktury se zaručeným nejhorším časem

Pro každé i si povolíme až čtyři různé množiny velikosti 2^i. Protože u předchozí konstrukce mohlo dojít k tomu, že uživatel čekal neúměrně dlouho na to, až se mu vytvoří velká $\mathcal{S}(A_i)$, tak si nyní budeme větší $\mathcal{S}(A_i)$ připravovat, hned jak pro něj získáme data (tzn. do A bude vloženo dost prvků). Ve chvíli, kdy budeme mít plné množiny $A_{i,0}$ a $A_{i,1}$, tak máme dost prvků na to, abychom zaplnili nějakou množinu $A_{i+1,*}$ (protože $\left|A_{i+1,*}\right| = 2\left|A_{i,*}\right|$), a tedy spustíme výrobu $\mathcal{S}(A_{i+1})$. Protože nemáme k dispozici paralelizaci, a nechceme, aby uživatel dlouho čekal, tak při výrobě $\mathcal{S}(A_{i,*})$ ($\left|A_{i,*}\right| = 2^i$) necháme v jednom kroku algoritmu počítat pouze $\frac{1}{2^i}$ kroků algoritmu, který nám $S(A_i)$ vyrábí. To vede k tomu, že nové struktury máme připravené ve chvíli, kdy je potřebujeme. Zároveň po dobu výroby $\mathcal{S}(A_{i+1,*})$ můžeme používat $\mathcal{S}(A_{i-1,0})$ a $\mathcal{S}(A_{i-1,1})$.

Operace Vyčísli(x) $T(n) =$

Operace Insert(x)	$T(n) =$
Vždy vkládám do první	

Konstrukce dynamické struktury

3.6 Samoupravující datové struktury, relaxované vyhledávací stromy

3.6.1 Samoupravující datové struktury

Samoupravující seznamy

D **Definice 3.43** (***MFR strategie***): *MFR (move front rule) strategie při každé operaci Member(x) nebo Insert(x) přesune (či vloží) prvek x na začátek seznamu.* □

D **Definice 3.44** (***TR strategie***): *TR (transition rule) strategie přesune prvek x při každé operaci Member(x) nebo Insert(x) o jeden prvek blíž začátku seznamu. Pokud prvek x přes zavoláním operace Insert(x) v seznamu nebyl, vloží ho na předposlední místo.* □

D **Definice 3.45** (***TIMESTAMP strategie***): *Pro každý prvek seznamu si pamatujeme čas, kdy byl naposledy argumentem nějaké operace Member(x) nebo Insert(x). V případě, že prvek doposud nebyl argumentem žádné operace, tak nebude přesunut. V opačném případě ho přesuneme před první prvek y, který je v seznamu před x a nebyla na něm provedena žádná operace od chvíle, kdy byla naposledy provedena operace na x (tzn. je v něm uložen nižší timestamp).* □

Poznámka: *TR strategie má lepší očekávaný čas než MFR strategie, ale jsou případy, kdy je MFR strategie podstatně rychlejší než TR strategie. Žádná strategie nemůže být více než dvakrát rychlejší než MFR strategie (lze dokázat).*

Splay stromy

3.6.2 Relaxované vyhledávací stromy

Seznam definic

Seznam vět

Kapitola
Rejstřík