# 7. Výpočetní modely

Když jsme v předešlých kapitolách studovali algoritmy, nezabývali jsme se tím, v jakém přesně výpočetním modelu pracujeme. Konstrukce, které jsme používali, totiž fungovaly ve všech obvyklých modelech a měly tam stejnou časovou i prostorovu složitost. Ne vždy tomu tak je, takže se výpočetním modelům podívejme na zoubek trochu blíže.

### Druhy výpočetních modelů

Obvykle se používají následující dva modely, které se liší zejména v tom, zda je možné paměť indexovat v konstantním čase či nikoliv.

**Pointer Machine (PM)** [1] pracuje se dvěma typy dat: *čísly* v pevně omezeném rozsahu a *pointery,* které slouží k odkazování na data uložená v paměti. Paměť tohoto modelu je složená z pevného počtu registrů na čísla a na pointery a z neomezeného počtu *krabiček.* Každá krabička má pevný počet položek na čísla a pointery. Na krabičku se lze odkázat pouze pointerem.

Aritmetika v tomto modelu (až na triviální případy) nefunguje v konstantním čase, datové struktury popsatelné pomocí pointerů (seznamy, stromy ...) fungují přímočaře, ovšem pole musíme reprezentovat stromem, takže indexování stojí $\Theta(\log n)$.

**Random Access Machine (RAM)** [2] je rodinka modelů, které mají společné to, že pracují výhradně s (přirozenými) čísly a ukládají je do paměti indexované opět čísly. Instrukce v programu (podobné assembleru) pracují s operandy, které jsou buď konstanty nebo buňky paměti adresované přímo (číslem buňky), případně nepřímo (index je uložen v nějaké buňce adresované přímo). Je vidět, že tento model je alespoň tak silný jako PM, protože odkazy pomocí pointerů lze vždy nahradit indexováním.

Pokud ovšem povolíme počítat s libovolně velkými čísly v konstantním čase, dostaneme velice silný paralelní počítač, na němž spočítáme téměř vše v konstantním čase (modulo kódování vstupu). Proto musíme model nějak omezit, aby byl realistický, a to lze udělat více způsoby:

- *Zavést logaritmickou cenu instrukcí* – operace trvá tak dlouho, kolik je počet bitů čísel, s nimiž pracuje, a to včetně adres v paměti. Elegantně odstraní absurdity, ale je dost těžké odhadovat časové složitosti; u většiny normálních algoritmů nakonec po dlouhém počítání vyjde, že mají složitost $\Theta(\log n)$-krát větší než v neomezeném RAMu.
- *Omezit velikost čísel* na nějaký pevný počet bitů (budeme mu říkat *šířka slova* a značit $w$) a operace ponechat v čase $\mathcal{O}(1)$. Abychom mohli alespoň adresovat vstup, musí být $w \geq \log N$, kde $N$ je celková velikost vstupu. Jelikož aritmetiku s $\mathcal{O}(1)$-násobnou přesností lze simulovat s konstantním zpomalením, můžeme předpokládat, že $w = \Omega(\log N)$, tedy že lze přímo pracovat s čísly polynomiálně velkými vzhledem k $N$. Ještě bychom si měli ujasnit, jakou množinu operací povolíme:
  - *Word-RAM* – „céčkové“ operace: $+$, $-$, $*$, /, mod (aritmetika); $<<$, $>>$ (bitové posuvy); $\wedge$, $\vee$, $\oplus$, $\neg$ (bitový and, or, xor

a negace).

- $AC^0$-*RAM* – libovolné funkce vyčíslitelné hradlovou sítí polynomiální velikosti a konstantní hloubky s hradly o libovolně mnoha vstupech.⟨1⟩ To je teoreticky čistší, patří sem vše z předchozí skupiny mimo násobení, dělení a modula, a také spousta dalších operací.
- *Kombinace předchozího* – tj. pouze operace Word-RAMu, které jsou v $AC^0$.

Ve zbytku této kapitoly ukážeme, že na RAMu lze počítat mnohé věci efektivněji než na PM. Zaměříme se na Word-RAM, ale podobné konstrukce jdou provést i na $AC^0$-RAMu. (Kombinace obou omezení vede ke slabšímu modelu.)

**Van Emde-Boas Trees**

Van Emde-Boas Trees neboli VEBT [6] jsou RAMová struktura, která si pamatuje množinu prvků $X$ z nějakého omezeného universa $X \subseteq \{0, \ldots, U-1\}$, a umí s ní provádět „stromové operace“ (vkládání, mazání, nalezení následníka apod.) v čase $\mathcal{O}(\log\log U)$. Pomocí takové struktury pak například dokážeme:

| | pomocí VEBT | nejlepší známé pro celá čísla |
|---|---|---|
| třídění | $\mathcal{O}(n \log\log U)$ | $\mathcal{O}(n \log\log n)$ [3] |
| MST | $\mathcal{O}(m \log\log U)$ | $\mathcal{O}(m)$ [viz příští kapitola] |
| Dijkstra | $\mathcal{O}(m \log\log U)$ | $\mathcal{O}(m + n \log\log n)$ [5], neorientovaně $\mathcal{O}(m)$ [4] |

My se přidržíme ekvivalentní, ale jednodušší definice podle Erika Demaine [2].

**Definice:** VEBT($U$) pro universum velikosti $U$ (BÚNO $U = 2^k = 2^{2^\ell}$) obsahuje:

- *min*, *max* reprezentované množiny (mohou být i nedefinovaná, pokud je množina moc malá),
- *přihrádky* $P_0, \ldots, P_{\sqrt{U}}$ obsahující zbývající hodnoty.⟨2⟩ Hodnota $x$ padne do $P_{\lfloor x/\sqrt{U} \rfloor}$. Každá přihrádka je uložena jako VEBT($\sqrt{U}$), který obsahuje příslušná čísla mod $\sqrt{U}$. [Bity každého čísla jsme tedy rozdělili na vyšších $k/2$, které indexují přihrádku, a nižších $k/2$ uvnitř přihrádky.]
- Navíc ještě *„sumární“* VEBT($\sqrt{U}$) obsahující čísla neprázdných přihrádek.

⟨1⟩ Pro zvídavé: $AC^k$ je třída všech funkcí spočítetelných polynomiálně velkou hradlovou sítí hloubky $\mathcal{O}(\log^k n)$ s libovolně-vstupovými hradly a $NC^k$ totéž s omezením na hradla se dvěma vstupy. Všimněte si, že $NC^0 \subseteq AC^0 \subseteq NC^1 \subseteq AC^1 \subseteq NC^2 \subseteq \ldots$.

⟨2⟩ Alespoň jedno z *min*, *max* musí být uloženo zvlášť, aby strom obsahující pouze jednu hodnotu neměl žádné podstromy. My jsme pro eleganci struktury zvolili uložit zvlášť obojí.

**Operace** se strukturou budeme provádět následovně. Budeme si přitom představovat, že v přihrádkách jsou uložena přímo čísla reprezentované množiny, nikoliv jen části jejich bitů – z čísla přihrádky a hodnoty uvnitř přihrádky ostatně dokážeme celou hodnotu rekonstruovat a naopak hodnotu kdykoliv rozložit na tyto části.

*FindMin* – minimum nalezneme v kořeni v čase $\mathcal{O}(1)$.

*Find*$(x)$ – přímočaře rekurzí přes přihrádky v čase $\mathcal{O}(k)$.

*Insert*$(x)$:

1. Ošetříme triviální stromy (prázdný a jednoprvkový)
2. Je-li třeba, prohodíme $x$ s $min$, $max$.
3. Prvek $x$ padne do přihrádky $P_i$, která je buď:
4. prázdná ⇒ *Insert* hodnoty $i$ do sumárního stromu a založení triviálního stromu pro přihrádku; nebo
5. neprázdná ⇒ převedeme na *Insert* v podstromu.

V každém případě buď rovnou skončíme nebo převedeme na *Insert* v jednom stromu nižšího řádu a k tomu vykonáme konstantní práci. Celkem tedy $\mathcal{O}(k)$.

*Delete*$(x)$ – smazání prvku bude opět pracovat v čase $\mathcal{O}(k)$.

1. Ošetříme triviální stromy (jednoprvkový a dvouprvkový).
2. Pokud mažeme $min$ (analogicky $max$), nahradíme ho minimem z první neprázdné přihrádky (tu najdeme podle sumárního stromu v konstantním čase) a převedeme na *Delete* v této přihrádce.
3. Prvek $x$ padne do přihrádky $P_i$, která je buď:
4. jednoprvková ⇒ zrušení přihrádky a *Delete* ze sumárního stromu; nebo
5. větší ⇒ *Delete* ve stromu přihrádky.

*Succ*$(x)$ – nejmenší prvek větší než $x$, opět v čase $\mathcal{O}(k)$:

1. Triviální případy: pokud $x < min$, vrátíme $min$; pokud $x \geq max$, vrátíme, že následník neexistuje.
2. Prvek $x$ padne do přihrádky $P_i$ a buďto:
3. $P_i$ je prázdná nebo $x = max(P_i)$ ⇒ pomocí *Succ* v sumárním stromu najdeme nejbližší další neprázdnou přihrádku $P_j$:
4. existuje-li ⇒ vrátíme $min(P_j)$,
5. neexistuje-li ⇒ vrátíme $max$.
6. nebo $x < max(P_i)$ ⇒ převedeme na *Succ* v $P_i$.

Složitosti operací jsou pěkné, ale nesmíme zapomenout, že strukturu je na počátku nutné inicializovat, což trvá $\Omega(\sqrt{U})$.[3] Z následujících úvah ovšem vyplyne, že si inicializaci můžeme odpustit.

[3] Svádí to k nápadu ponechat přihrádky neinicializované a nejdříve se vždy zeptat sumárního stromu, ale tím bychom si pokazili časovou složitost.

### Modely inicializace

Jak může být definován obsah paměti na počátku výpočtu:

**„Při odchodu zhasni“:** Zavedeme, že paměť RAMu je na počátku inicializována nulami a program ji po sobě musí uklidit (to je nutné, aby programy šlo iterovat). To u VEBT není problém zařídit.

**Neinicializovano:** Na žádné konkrétní hodnoty se nemůžeme spolehnout, ale je definováno, že neinicializovanou buňku můžeme přečíst a dostaneme nějakou korektní, i když libovolnou, hodnotu. Tehdy nám pomůže:

**Věta:** Buď $P$ program pro Word-RAM s nulami inicializovanou pamětí, běžící v čase $T(n)$. Pak existuje program $P'$ pro Word-RAM s neinicializovanou pamětí počítající totéž v čase $\mathcal{O}(T(n))$.

*Důkaz:* Během výpočtu si budeme pamatovat, do kterých paměťových buněk už bylo něco zapsáno, a které tedy mají definovanou hodnotu. Prokládaně uložíme do paměti dvě pole: $M$, což bude paměť původního stroje, a $L$ – seznam čísel buněk v $M$, do kterých už program zapsal. Přitom $L[0]$ bude udávat délku seznamu $L$.

Program nyní začne tím, že vynuluje $L[0]$ a bude simulovat původní program, přičemž kdykoliv ten bude chtít přečíst nějakou buňku z $M$, podíváme se do $L$, zda už je inicializovaná, a případně vrátíme nulu a buňku připíšeme do $L$.

To je funkční, ale pomalé. Redukci tedy vylepšíme tak, že založíme další proložené pole $R$, jehož hodnota $R[i]$ bude říkat, kde v $L$ se vyskytuje číslo $i$-té buňky, nebo bude neinicializována, pokud takové místo dosud neexistuje.

Před čtením $M[i]$ se teď podíváme na $R[i]$ a ověříme, zda $R[i]$ neleží mimo seznam $L$ a zda je $L[R[i]] = i$. Tím v konstantním čase zjistíme, jestli je $M[i]$ již inicializovaná, a jsme také schopni tuto informaci v témže čase udržovat. ♡

**„Minové pole“:** Neinicializované buňky není ani dovoleno číst. V tomto případě nejsme schopni deterministické redukce, ale alespoň můžeme použít randomizovanou – ukládat si obsah paměti do hashovací tabulky, což při použití universálního hashování dá složitost $\mathcal{O}(1)$ na operaci v průměrném případě.

### Technické triky

VEBT nedosahují zdaleka nejlepších možných parametrů – lze sestrojit i struktury pracující v konstantním čase. To v následující kapitole také uděláme, ale nejdříve v této poněkud technické stati vybudujeme repertoár základních operací proveditelných na Word-RAMu v konstantním čase.

Rozcvička: *nejpravější jednička* ve dvojkovém čísle (hodnota, nikoliv pozice):

$$\begin{aligned} x &= \mathbf{01}\cdots\mathbf{011000000} \\ x-1 &= \mathbf{01}\cdots\mathbf{010111111} \\ x \wedge (x-1) &= \mathbf{01}\cdots\mathbf{010000000} \\ x \oplus (x \wedge (x-1)) &= \mathbf{00}\cdots\mathbf{001000000} \end{aligned}$$

Nyní ukážeme, jak RAM používat jako vektorový počítač, který umí paralelně počítat se všemi prvky vektoru, pokud se dají zakódovat do jediného slova. Libovolný

$n$-složkový vektor, jehož složky jsou $b$-bitová čísla ($n(b+1) \leq w$), zakódujeme poskládáním jednotlivých složek vektoru za sebe, proloženě nulovými bity:

$$\mathbf{0}x_{n-1}\mathbf{0}x_{n-2}\cdots\mathbf{0}x_1\mathbf{0}x_0$$

S vektory budeme provádět následující operace: (latinkou značíme vektory, alfabetou čísla, $\mathbf{0}$ a $\mathbf{1}$ jsou jednotlivé bity, $(\ldots)^k$ je $k$-násobné opakování binárního zápisu)

- *Replicate*$(\alpha)$ – vytvoří vektor $(\alpha, \alpha, \ldots, \alpha)$:

  $$\alpha * (\mathbf{0}^b\mathbf{1})^n$$

- *Sum*$(x)$ – sečte všechny složky vektoru (předpokládáme, že se součet vejde do $b$ bitů):

  a) vymodulením číslem $\mathbf{1}^{b+1}$ (protože $\mathbf{10}^{b+1} \bmod \mathbf{1}^{b+1} = 1$), či

  b) násobením vhodnou konstantou:

$$\begin{array}{cccccccccc}
 & & & & & x_{n-1} & \cdots & x_2 & x_1 & x_0 \\
 & & & & * & \mathbf{0}^b\mathbf{1} & \cdots & \mathbf{0}^b\mathbf{1} & \mathbf{0}^b\mathbf{1} & \mathbf{0}^b\mathbf{1} \\
\hline
 & & & & & x_{n-1} & \cdots & x_2 & x_1 & x_0 \\
 & & & & x_{n-1} & x_{n-2} & \cdots & x_1 & x_0 & \\
 & & & x_{n-1} & x_{n-2} & x_{n-3} & \cdots & x_0 & & \\
 & & & \vdots & \vdots & \vdots & \vdots & & & \\
x_{n-1} & \cdots & x_2 & x_1 & x_0 & & & & & \\
\hline
r_{n-1} & \cdots & r_2 & r_1 & s_n & \cdots & s_3 & s_2 & s_1 &
\end{array}$$

  Zde je výsledkem dokonce vektor všech částečných součtů: $s_k = \sum_{i=0}^{k-1} x_i$, $r_k = \sum_{i=k}^{n-1} x_i$.

- *Cmp*$(x, y)$ – paralelní porovnání dvou vektorů: $i$-tá složka výsledku je 1, pokud $x_i < y_i$, jinak 0.

$$\begin{array}{cccccccccc}
 & \mathbf{1} & x_{n-1} & \mathbf{1} & x_{n-2} & \cdots & \mathbf{1} & x_1 & \mathbf{1} & x_0 \\
- & \mathbf{0} & y_{n-1} & \mathbf{0} & y_{n-2} & \cdots & \mathbf{0} & y_1 & \mathbf{0} & y_0
\end{array}$$

  Ve vektoru $x$ nahradíme prokládací nuly jedničkami a odečteme vektor $y$. Ve výsledku se tyto jedničky změní na nuly právě u těch složek, kde $x_i < y_i$. Pak je již stačí posunout na správné místo a okolní bity vynulovat a znegovat.

- *Rank*$(\alpha, x)$ – spočítá, kolik složek vektoru $x$ je menších než $\alpha$:

  $$\mathit{Rank}(\alpha, x) = \mathit{Sum}(\mathit{Cmp}(\mathit{Replicate}(\alpha), x)).$$

- *Insert*$(\alpha, x)$ – zatřídí hodnotu $\alpha$ do setříděného vektoru $x$:
  Zde stačí pomocí operace *Rank* zjistit, na jaké místo novou hodnotu zatřídit, a pak to pomocí bitových operací provést („rozšoupnout“ existující hodnoty).

- *Unpack*$(\alpha)$ – vytvoří vektor, jehož složky jsou bity zadaného čísla (jinými slovy proloží bity bloky $b$ nul).
  Nejdříve číslo $\alpha$ replikujeme, pak andujeme vhodnou bitovou maskou, aby v $i$-té složce zůstal pouze $i$-tý bit a ostatní se vynulovaly, a pak provedeme *Cmp* s vektorem samých nul.
- *Unpack*$_\varphi(\alpha)$ – podobně jako předchozí operace, ale bity ještě prohází podle nějaké pevné funkce $\varphi$:
  Stačí zvolit bitovou masku, která v $i$-té složce ponechá právě $\varphi(i)$-tý bit.
- *Pack*$(x)$ – dostane vektor nul a jedniček a vytvoří číslo, jehož bity jsou právě složky vektoru (jinými slovy škrtne nuly mezi bity):
  Představíme si, že složky čísla jsou o jeden bit kratší a provedeme *Sum*. Například pro $n = 4$ a $b = 4$:

$$\left|\mathbf{0}\ \mathbf{0}\ \mathbf{0}\ \mathbf{0}\ x_3\right|\mathbf{0}\ \mathbf{0}\ \mathbf{0}\ \mathbf{0}\ x_2\left|\mathbf{0}\ \mathbf{0}\ \mathbf{0}\ \mathbf{0}\ x_1\right|\mathbf{0}\ \mathbf{0}\ \mathbf{0}\ \mathbf{0}\ x_0\right|$$
$$\left|\mathbf{0}\ \mathbf{0}\ \mathbf{0}\ \mathbf{0}\right|x_3\ \mathbf{0}\ \mathbf{0}\ \mathbf{0}\left|\mathbf{0}\ x_2\ \mathbf{0}\ \mathbf{0}\right|\mathbf{0}\ \mathbf{0}\ x_1\ \mathbf{0}\left|\mathbf{0}\ \mathbf{0}\ \mathbf{0}\ x_0\right|$$

  Jen si musíme dát pozor na to, že vytvořený vektor s kratšími složkami není korektně prostrkán nulami. Konstrukce *Sum* pomocí modula proto nebude správně fungovat a místo $\mathbf{1}^b$ vygeneruje $\mathbf{0}^b$. To můžeme buď ošetřit zvlášť, nebo použít konstrukci přes násobení, které to nevadí.

Nyní ještě několik operací s normálními čísly. Chvíli předpokládejme, že pro $b$-bitová čísla na vstupu budeme mít k dispozici $b^2$-bitový pracovní prostor, takže budeme moci používat vektory s $b$ složkami po $b$ bitech.

- $\#1(\alpha)$ – spočítá jedničkové bity v zadaném čísle.
  Stačí provést *Unpack* a následně *Sum*.
- *Permute*$_\pi(\alpha)$ – přehází bity podle zadané fixní permutace.
  Provedeme *Unpack*$_\pi$ a *Pack* zpět.
- *LSB*$(\alpha)$ – Least Significant Bit (pozice nejnižší jedničky):
  Podobně jako v rozcvičce nejdříve vytvoříme číslo, které obsahuje nejnižší jedničku a vpravo od ní další jedničky, a pak tyto jedničky posčítáme pomocí #1:

$$\begin{aligned}\alpha &= \cdots\cdots\mathbf{10000}\\ \alpha - 1 &= \cdots\cdots\mathbf{01111}\\ \alpha \oplus (\alpha - 1) &= \mathbf{0}\cdots\mathbf{011111}\end{aligned}$$

- *MSB*$(\alpha)$ – Most Significant Bit (pozice nejvyšší jedničky):
  Z *LSB* pomocí zrcadlení (operací *Permute*).

Poslední dvě operace dokážeme spočítat i v lineárním prostoru, například pro *MSB* takto: Rozdělíme číslo na bloky velikosti $\lfloor\sqrt{w}\rfloor$. Pak pro každý blok zjistíme, zda v něm je aspoň jedna jednička, zavoláním *Cmp*$(0, x)$. Pomocí *Pack* z toho dostaneme slovo $y$ odmocninové délky, jehož bity indikují neprázdné bloky. Na toto číslo

zavoláme předchozí kvadratické *MSB* a zjistíme index nejvyššího neprázdného bloku. Ten pak izolujeme a druhým voláním kvadratického algoritmu najdeme nejlevější jedničku uvnitř něj.⟨4⟩

**Literatura**


[1] Ben-Amram. What is a "pointer machine"? *SIGACTN: SIGACT News (ACM Special Interest Group on Automata and Computability Theory)*, 26, 1995.

[2] E. Demaine. Advanced Data Structures. MIT Lecture Notes, 2005.

[3] Y. Han. Deterministic sorting in O (nloglogn) time and linear space. *Journal of Algorithms*, 50(1):96–105, 2004.

[4] M. Thorup. Undirected single-source shortest paths with positive integer weights in linear time. *Journal of the ACM (JACM)*, 46(3):362–394, 1999.

[5] M. Thorup. Integer priority queues with decrease key in constant time and the single source shortest paths problem. *Proceedings of the thirty-fifth ACM symposium on Theory of computing*, pages 149–158, 2003.

[6] P. van Emde Boas. Preserving order in a forest in less than logarithmic time and linear space. *Inf. Process. Lett.*, 6(3):80–82, 1977.


⟨4⟩ Dopouštíme se drobného podvůdku – vektorové operace předpokládaly prostrkané nuly a ty tu nemáme. Můžeme si je ale snadno pořídit a bity, které jsme nulami přepsali, prostě zpracovat zvlášť.