# Korespondenční Seminář z Programování

38. ročník | KSP | Duben 2026


## Vzorová řešení čtvrté série třicátého osmého ročníku KSP

### 38-4-1 Vesmírné trubky

Nejprve informatická formulace úlohy: máme zadaných $N$ bodů v rovině a postupně se nám dalších $K$ bodů objevuje. Chceme si udržovat *konvexní obal* všech bodů a po každém přidání bodu vypsat, jak se obal změnil. Konvexní obal množiny bodů $P$ definujeme jako mnohoúhelník s nejmenším obvodem, který obsahuje všechny body z $P$.

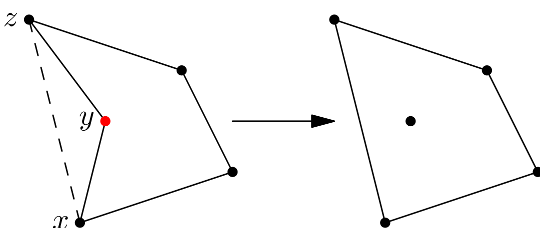
Kdybychom chtěli konvexní obal spočítat jen jednou, můžeme na to použít algoritmus z geometrické kuchařky.[1] (Pokud ho neznáte, nezoufejte, vybudujeme si od základu vhodnější algoritmus.) Jedno volání kuchařkového algoritmu trvá $\Theta(|P| \cdot \log |P|)$ času. Nejpřímočařejší řešení je jednoduše $(K+1)$-krát pustit kuchařkový algoritmus, nejdříve na prvních $N$ bodech, pak na $N+1$ bodech, …, až nakonec na $N+K$ bodech ze vstupu. To dává celkovou časovou složitost v $\Theta(K \cdot (N+K) \log(N+K))$, s čímž se nespokojíme.

Nejprve ale trocha geometrie. Jak praví kuchařka, konvexní obal $U$ se také ekvivalentně dá definovat jako mnohoúhelník splňující následující dvě vlastnosti:

- *Obal.* Vrcholy $U$ jsou body z $P$ a všechny body z $P$ leží v $U$ nebo na jeho hranici.
- *Pravotočivost.* Všechny vnitřní úhly $U$ jsou menší než $180\,°$. Jinými slovy, $U$ v sobě nemá žádné „zuby“: každá trojice jeho sousedních vrcholů, označených $x$, $y$ a $z$ v pořadí podle hodinových ručiček, „zatáčí doprava“, tedy $z$ leží napravo od polopřímky $xy$. (Jeden nepravotočivý mnohoúhelník je vyobrazen nalevo na obrázku níže.)

Můžeme si rozmyslet, že z původní definice konvexního obalu obě vlastnosti plynou: mějme mnohoúhelník $U$ s nejmenším obvodem a obsahující všechny body z $P$ a uvažujme:

- *Obal.* $U$ z definice pokrývá $P$. Pro spor předpokládejme, že $U$ má za vrchol bod mimo $P$. Ten pak můžeme mírně posunout, mírně zkrátit obvod $U$ a přitom zachovat vlastnost, že $U$ pokrývá všechny body z $P$ (rozmyslete si detaily). Tak jsme vyrobili mnohoúhelník $U'$, který je pořád obal, ale zároveň má menší obvod než $U$. To je spor s tím, že $U$ je obal s nejmenším možným obvodem.
- *Pravotočivost.* Pro spor předpokládejme, že $U$ není pravotočivý. Pak má tři sousední vrcholy $x$, $y$, $z$, kde $z$ je nalevo od polopřímky $xy$. Pak ale můžeme $y$ odstranit, hranice se jen zkrátí (neboť $|xy| + |yz| \geq |xz|$) a oblast pokrytá mnohoúhelníkem jen poroste, tedy to stále bude obal (viz obrázek). To je opět spor s tím, že $U$ je konvexní obal.



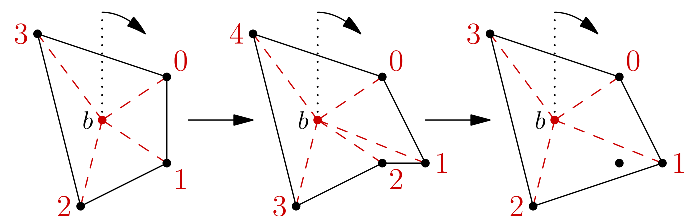

Obdobně se dá rozmyslet opak: pravotočivý obal $P$ je jednoznačně určený a je to právě konvexní obal $P$ (důkaz zde vynecháme). Odteď tedy můžeme pojmy „pravotočivý obal“ a „konvexní obal“ používat zaměnitelně.

Myšlenku z druhého bodu důkazu můžeme použít algoritmicky k tomu, abychom z nepravotočivého obalu $P$ udělali pravotočivý: Stačí nám jen opakovaně hledat trojici sousedních vrcholů $x$, $y$, $z$ zatáčející doleva a odstraňovat $y$, dokud nějaká taková trojice existuje. Mnohoúhelník nikdy nepřestal být obalem a v okamžiku, kdy žádná vhodná trojice vrcholů neexistuje, je to nutně obal pravotočivý, a tedy konvexní.

Začíná se nám rýsovat plán: budeme si udržovat hranici konvexního obalu jako posloupnost bodů (začínající libovolným bodem a pokračující po směru hodinových ručiček) a pokaždé, když se objeví nový bod, ho na vhodné místo do hranice přidáme. Tím zaručíme, že naše hranice pořád obaluje všechny body včetně toho nově přidaného, nejspíš tím ale porušíme konvexitu. Tu ale umíme obnovit pomocí předchozího algoritmu. Vrcholy obalu si budeme reprezentovat v uspořádané množině, například v binárním vyhledávacím stromě (BVS). Tak budeme v čase $\mathcal{O}(\log |P|)$ umět vrcholy přidávat, mazat a ptát se na jejich sousedy na hranici.

Zbývá si rozmyslet důležitý detail: BVS po nás požaduje nějaké uspořádání – funkci, která dostane dvojici bodů a rozhodne, který je z nich je „menší“. Ideálně by navíc měla pracovat v konstatním čase. My chceme body v BVS ukládat v pořadí, jak jdou za sebou na hranici. Jaké uspořádání zvolit, abychom toho docílili?

Jedna možnost je zafixovat si libovolný bod $b$ v konvexním obalu (třeba počkat, dokud v BVS nemáme tři body a pak zvolit jejich těžiště) a pak všechny body porovnávat podle toho, na jakou „světovou stranu“ leží od $b$. Důležité je, že obal je konvexní a $b$ po celý běh algoritmu zůstává uvnitř něj (jelikož konvexní obal jen roste). Pak uspořádaní určené bodem $b$ přesně odpovídá pořadí, jak za sebou jdou body na hranici (rozmyslete si; zároveň si rozmyslete, že obecně jsou obě podmínky nutné).




Pokud teď do našeho konvexního mnohoúhelníka přidáme nový bod a budeme se přitom řídit uspořádáním podle $b$, přestane sice mnohoúhelník dočasně být konvexní, ale pořád alespoň platí, že se jeho hranice nekříží – což je malý, ale důležitý detail, neboť všechny naše algoritmy a úvahy se rozbijí, pokud dovolíme, aby mnohoúhelník sám sebe protínal.

[1] `http://ksp.mff.cuni.cz/viz/kucharky/geometrie`

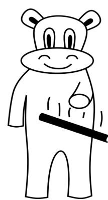
A to už je celý algoritmus: udržujeme si v BVS cyklicky uspořádaný seznam vrcholů konvexního obalu. Na začátku je konvexní obal prázdný a body do něj přidáváme po jednom. Přidaný bod vložíme na vhodné místo určené bodem $b$ a pak opakovaně hledáme trojici $x$, $y$, $z$ zatáčející doleva a mažeme (a vypisujeme) $y$. Navíc platí, že v každé trojici vedoucí k smazání bodu bude jeden z bodů ten nově přidávaný, takže nemusíme zbytečně procházet celý obal, ale stačí vždy opakovaně kontrolovat jen lokální okolí nově přidaného bodu.

Časová složitost přidání jednoho bodu je tudíž $\mathcal{O}(D \log |P|)$, kde $D$ je počet bodů, které jsme následkem přidání odstranili. Celková časová složitost je pak $\mathcal{O}((N + K) \log |P|) = \mathcal{O}((N+K) \log(N+K))$, protože přes všechny operace jsme dohromady mohli smazat nejvíce $N+K$ bodů. Můžeme též říci, že amortizovaná časová složitost jedné operace přidání bodu je $\mathcal{O}(\log |P|)$.

Poznámka na závěr: uspořádat body cyklicky podle $b$ nebyl jediný správný přístup. Stejně dobře funguje řešení, které můžete znát z kuchařkového algoritmu: místo jednoho konvexního obalu si budeme zvlášť udržovat *horní* a *dolní obálku*. Body v obálkách prostě seřadíme podle $x$-ové souřadnice – vzestupně v horní, sestupně v dolní. Obě obálky udržujeme nezávisle a každý bod přidáváme do obou z nich. Výhoda je o něco snažší třídění, nevýhoda je trochu větší množství okrajových případů. Na tomto přístupu je založeno i vzorové řešení.

Program (C++):
`http://ksp.mff.cuni.cz/viz/38-4-1.cpp`

*Úlohu připravili: Daniel Culliver, Ríša Hladík, Patrik Prítrský, Ondra Sedláček, Vladimír Sklenár*



## 38-4-2 Kdo se v tom vyzná

Ako prvé si všimnime, že systém hrochov a vier, ktorý máme simulovať, si vieme jednoducho previesť do abstraktnejšieho matematického jazyka: pozostáva z prvkov – hrochov – a disjunktných množín, ktoré spolu obsahujú všetky prvky – každá množina pritom obsahuje hrochov s rovnakou vierou. Na začiatku máme každého hrocha v samostatnej množine, množiny teda vieme prirodzene indexovať podľa hrochov, ktorí sa v nich na začiatku nachádzali.

Ďalej si všimnime, že na celom systéme vykonávame iba jednu operáciu, ktorá ho nejakým spôsobom mení, zadanú inštrukciou `J` $A$ $B$. Tá zoberie dve množiny a spojí ich do jednej, počet množín zníži o 1.

Ďalšie inštrukcie už sú iba dotazy, na ktoré dokážeme odpovedať ak zistíme:

- `S` $A$ $B$ – V ktorej množine sa nachádza prvok $A$, v ktorej prvok $B$ a či sú tieto dve množiny rovnaké.
- `V` $A$ – V ktorej množine sa nachádza prvok $A$ a koľko prvkov daná množina obsahuje.
- `P` – Koľko rôznych množín existuje.

S týmito operáciami nám prirodzene pomôže dátová štruktúra Union-Find, ktorá si udržiava systém množín, dokáže množiny spájať (operáciou *union*, ktorú použijeme pri inštrukcií `J`) a pre každý prvok určiť, v ktorej množine sa nachádza (operáciou *find*).

Každý prvok má priradený jeden pointer, ktorým ukazuje na „koreň“ svojej množiny, teda taký prvok, ktorého index je rovnaký ako index množiny samotnej. Operácia *union* jednoducho vezme jednu z množín a jej hlavnému prvku nastaví pointer na koreň druhej množiny. Tým sa nám môže stať, že všetky prvky v prvej množine ukazujú na nesprávny koreň, takže operácia *find* prestane fungovať. Aktualizovať všetky pointery by však mohlo trvať veľmi dlho.

Namiesto toho si upravíme operáciu *find* tak, aby nehľadala koreň iba tam, kam ukazuje pointer z aktuálneho prvku, ale pokračovala rovnakým spôsobom ďalej, až dokým nenájde skutočný koreň. Koreň množiny pritom poznáme podľa toho, že ukazuje sám na seba, takže stačí hľadanie zastaviť, keď sa zacyklí a prvok ukazuje sám na seba.

Mohlo by sa nám stať, že pri určitej sekvencií operácií *union* dostaneme štruktúru, v ktorej bude takéto krokované vyhľadávanie pomalé. To sa dá riešiť viacerými spôsobmi – buď si pre každú množinu budeme udržovať jej maximálnu hĺbku a spájať budeme vždy plytšiu množinu do hlbšej, alebo pri každej operácií *find* prejdeme znova všetky prvky, ktoré sme pri hľadaní koreňa navštívili, a zmeníme ich poiter priamo na koreň. Ľubovoľná z týchto modifikácií nám zabezpečí, že celková časová zložitosť práce s dátovou štruktúrou bude od počtu prvkov závisieť logaritmicky (a samozrejme od celkového počtu operácií lineárne). Pre viac detailov o štruktúre Union-Find odporúčame nahliadnuť do Průvodce labyrintem algoritmů.[2]

Zodpovedať dotaz `S` je s takto vybudovanou štruktúrou triviálne, dvakrát použijeme operáciu *find* a zistíme, či sú odpovede pre $A$ a $B$ rovnaké.

Aby sme vedeli zodpovedať dotaz `V`, budeme si pre každú množinu pamätať aj jej veľkosť (počet hrochov). Na začiatku má každá množina veľkosť 1. Pri operácií *union* potom do veľkosti množiny, ktorej koreň zachovávame, pripočítame veľkosť druhej množiny. Pri odpovedaní na dotaz nám stačí jedna operácia *find*.

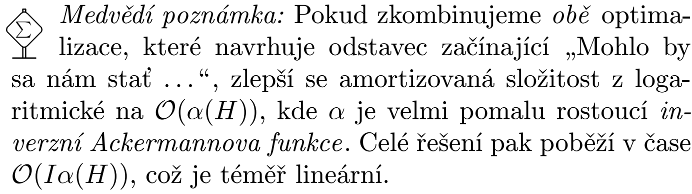
Dotazy `P` budeme zodpovedať ešte jednoduchšie – po celý čas si budeme pamätať počet množín a vždy, keď použijeme operáciu *union*, tento počet znížime o 1. Musíme si ale dať pozor (a to nielen kvôli týmto dotazom), aby sme počítadlo neznižovali pri pokuse o *union* dvoch prvkov z rovnakej množiny – teda aby sme v prípade, že žiadna množina nezanikne, pretože spájané prvky už sú spolu v jednej množine, počet množín neznižovali. To vieme samozrejme zistiť pomocou operácí *find*, ktoré musíme použiť aj na to, aby sme našli korene samotných spájaných množín.

Pre každú inštrukciu nám stačí vykonať vykonať konštantne mnoho operácií *find* a prípadne operáciu *union* – obe majú amortizovane logaritmickú časovú zložitosť voči počtu hrochov, celé riešenie má teda časovú zložitosť $\mathcal{O}(I \log H)$.

Program (Python):
`http://ksp.mff.cuni.cz/viz/38-4-2.py`

*Úlohu pripravili: David „Dejwut“ Pacák, Kristýna Petrlíková, Ján „Janči“ Plachý, Matúš Púll*

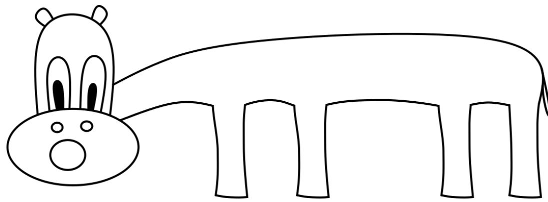
[2] `https://pruvodce.ucw.cz/static/pruvodce.pdf#s7.4`



*Medvědí poznámka:* Pokud zkombinujeme *obě* optimalizace, které navrhuje odstavec začínající „Mohlo by sa nám stať ...“, zlepší se amortizovaná složitost z logaritmické na $\mathcal{O}(\alpha(H))$, kde $\alpha$ je velmi pomalu rostoucí *inverzní Ackermannova funkce*. Celé řešení pak poběží v čase $\mathcal{O}(I\alpha(H))$, což je téměř lineární.

Existuje i čistě lineární řešení, které se dá poskládat z několika dost pokročilých triků. Předně kdybychom dopředu věděli, které *union*y se skutečně provedou (spojují množiny, které do té doby byly různé), dal by se *union* i *find* provádět v amortizovaně konstantním čase; detaily najdete v kapitole o dekompozici stromů ve skriptíčkách Krajinou grafových algoritmů.[3]

Provedené *union*y poznáme, pokud vytvoříme graf, jehož vrcholy budou hroši, hrany budou odpovídat *union*ům a každou hranu ohodnotíme pořadovým číslem instrukce *union*. Pak si všimneme, že minimální kostru tohoto grafu tvoří právě ty *union*y, které se provedly. A jako posledního králíka z klobouku vytáhneme Fredmanův-Willardův algoritmus, který dokáže najít minimální kostru v lineárním čase, pokud jsou hrany ohodnocené celými čísly – více opět najdete ve skriptíčkách o grafových algoritmech.



## 38-4-3 Lidská pyramida

### Pozorování

Nejprve pojďme ukázat, jak celkovou výšku ovlivní student v $i$-té hladině (hladiny indexujeme odshora). Označme $V$ výšku pyramidy, $V_{ij}$ výšku umístění hlavy $j$-tého studenta v $i$-té hladině a $v_{ij}$ jeho tělesnou výšku. Vyjádříme postavení studenta na dvou pod ním:

$$V_{ij} = v_{ij} + \frac{V_{i+1,2\cdot j} + V_{i+1,2\cdot j+1}}{2} = v_{ij} + \frac{V_{i+1,2\cdot j}}{2} + \frac{V_{i+1,2\cdot j+1}}{2}$$

Teď vyjádříme výšku pyramidy:

$$V = V_{00} = v_{00} + \frac{V_{10}}{2} + \frac{V_{11}}{2} =$$

$$= v_{00} + \frac{v_{10} + \frac{V_{20}}{2} + \frac{V_{21}}{2}}{2} + \frac{v_{11} + \frac{V_{22}}{2} + \frac{V_{23}}{2}}{2}$$

$$= v_{00} + \frac{v_{10}}{2} + \frac{v_{11}}{2} + \frac{V_{20}}{4} + \frac{V_{21}}{4} + \frac{V_{22}}{4} + \frac{V_{23}}{4} = \ldots$$

Z toho už by mohlo být vidět, jak student v $i$-té hladině přispěje do výšky celé pyramidy.

Zaprvé s klesnutím hladiny studenti svou výškou přispějí o polovinu méně. Zadruhé bude započítán každý student pouze jednou, jelikož na něm stojí právě jeden student, na kterém také, na kterém také ... až ke kořeni. Z toho získáváme, že každý student přispívá do výšky pyramidy $\frac{v_{ij}}{2^i}$.

[3] `https://mj.ucw.cz/vyuka/ga/`

[4] `http://ksp.mff.cuni.cz/viz/kucharky/halda-a-cesty`

### Princip řešení

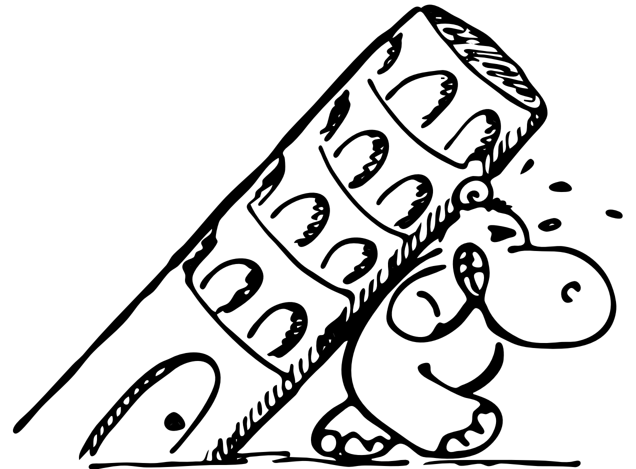
Představme si, že přišel student s výškou $v$. Kam ho umístit? Pokud na pozici $(i, j)$, pak zase odejde student s výškou $v_{ij}$, což dává rozdíl výšky na dané pozici $v - v_{ij}$. Protože se nachází v $i$-té hladině, je rozdíl výšky pyramidy $d(v, v_{ij}) := \frac{v - v_{ij}}{2^i}$. Hledáme kam umístit příchozího studenta, aby byl rozdíl co nejmenší (ideálně i záporný).

Podívejme se nyní na studenty v $i$-té hladině. Rozdíl výšky $d$ vyjádříme jako $\frac{v}{2^i} - \frac{1}{2^i} \cdot v_{ij}$ a z toho již vidíme, že aby byl rozdíl nejmenší, musí být vyřazený student nejvyšší. Tedy z každé hladiny chceme vybrat nejvyššího studenta. V zadání ještě zní, že pokud jich je víc, chceme toho nejvíce napravo – s maximálním $j$.

Pro efektivní výběr studenta k nahrazení nám najednou stačí projít všechny hladiny a vybrat tu s nejvhodnějším nejvyšším studentem podle $d(v, v_{ij})$. Jelikož se hladiny neprolínají, můžeme každou řešit zvlášť.

No a jakou kouzelnou datovou strukturu použijeme, která umí najít a odstranit maximum a přidat prvek? No přece haldu! O té si můžete přečíst v naší kuchařce o haldě a cestách.[4] Umí přidávat prvky a odebírat maximum v čase $\mathcal{O}(\log N)$, my ale máme v každé hladině hladině maximálně $\mathcal{O}(2^H)$ studentů, což dává složitost operací v $\mathcal{O}(H)$. Pro nás je ale důležité, že umí najít maximum v konstantním čase.

U studentů v haldě ale zatím nevíme na jaké pozici stojí. Budeme si kvůli tomu v haldě ukládat dvojici čísel $(v_{ij}, j)$. To nám zároveň pomůže, až bude více nejvyšších studentů v hladině. Porovnání dvojic se bude provádět lexikograficky – primárně podle $v_{ij}$, sekundárně podle $j$. Většina programovacích jazyků by se takhle ke dvojicím chovala sama.



### Shrnutí

Náš algoritmus bude vypadat následovně:

Pro každou hladinu založíme haldu a všechny její studenty do ní naházíme (opatrně).

Poté pro každého příchozího studenta s výškou $v$ projdeme všechny hladiny, kde na hladině $i$ nám halda řekne studenta na pozici $j$ s maximální výškou. Z nich vybereme hladinu, kde bude $\frac{v - v_{ij}}{2^i}$ minimální. Ve vybrané haldě odstraníme maximum a přidáme $(v, j)$. Nakonec triumfálně vracíme vybrané $(i, j)$ a pokračujeme na dalšího studenta.

Program (C++):
`http://ksp.mff.cuni.cz/viz/38-4-3.cpp`

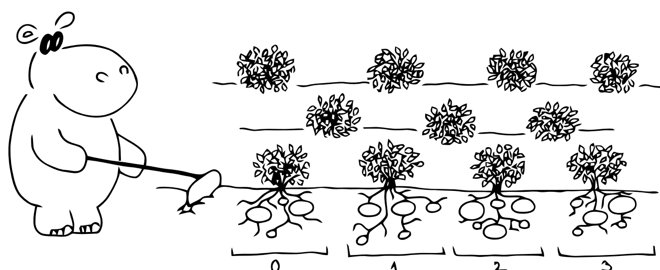
*Úlohu připravili: Adam Jahoda, Matúš Púll*

## 38-4-4 Alokátor

O každém bloku paměti si potřebujeme pamatovat tři parametry: *adresu* začátku bloku, *délku* a *stav* (obsazený nebo volný). Jelikož alokace potřebuje vyhledávat podle velikostí, zatímco ostatní operace podle pořadí nebo adres, bloky budeme ukládat do dvou datových struktur.

*Adresová* struktura si bude pamatovat bloky uspořádané podle adresy a bude umět následující operace:

- *Insert* a *Delete* – vloží/smaže blok s danými parametry.
- *SeekLeft*($x$) – najde poslední blok s adresou menší nebo rovnou $x$ (pokud hledaný blok neexistuje, je výsledkem speciální hodnota $\emptyset$; podobně pro další operace).
- *SeekRight*($x$) – najde první blok s adresou větší nebo rovnou $x$.
- *Select*($i$) – najde $i$-tý blok v pořadí.
- *Prev*($b$) – najde předchůdce bloku $b$.
- *Next*($b$) – najde následníka bloku $b$.




*Délková* struktura tytéž bloky seřadí primárně podle velikosti, sekundárně podle adresy. Bude umět tytéž operace, jen budou pracovat s délkami místo adres.

Kdykoliv blok vznikne, zanikne, nebo změní stav, musíme aktualizovat obě datové struktury. To v nejhorším případě obnáší jeden *Insert* a jeden *Delete* na každé struktuře.

Nyní si rozmyslíme, jak pomocí našich dvou struktur vytvořit alokátor:

- *Alokace* bloku velikosti $d$ nalezne pomocí *SeekRight* v délkové struktuře nejmenší blok velikosti aspoň $d$. Pokud má délku přesně $d$, stačí ho prohlásit za obsazený. Jinak ho musíme rozdělit na obsazený blok a volný zbytek. To obnáší *Delete* a dva *Inserty*.
- *Uvolnění* $i$-tého bloku naopak hledá v adresové struktuře. Nejprve pomocí *Select*u blok najde. Pak ho změní na volný. Poté se pomocí *Prev* zeptá na předchozí blok; pakliže existuje a je také volný, oba bloky spojí (2 × *Delete* a *Insert*). Nakonec udělá totéž s následníkem pomocí *Next*.
- *Vyhledání* bloku, který obsahuje zadanou adresu, provede *SeekLeft* na adresové struktuře.

Vidíme tedy, že každou operaci alokátoru umíme převést na $\mathcal{O}(1)$ operací na datových strukturách.

Zbývá zvolit konkrétní datové struktury. Použijeme staré dobré binární vyhledávací stromy. Budeme je udržovat vyvážené (třeba jako AVL strom), abychom zajistili hloubku $\mathcal{O}(\log K)$, kde $K$ je aktuální počet bloků.

Navíc si v každém vrcholu $v$ budeme pamatovat *size*($v$) udávající, kolik má tento vrchol potomků (včetně sebe sama).

Všimněte si, že při rotacích během vyvažování dokážeme *size* vždy přepočítat ze *size* v dětech.

Jednotlivé operace implementujeme takto:

- *Insert* provádíme standardně: vyhledáme správné místo, tam přidáme list a pak se vracíme do kořene, přičemž jednak vyvažujeme, ale také zvyšujeme *size* o 1 všem vrcholům, přes které jsme prošli.
- *Delete* je také standardní, jen na cestě do kořene snižujeme *size*.
- *SeekLeft*($x$) popíšeme rekurzivně: ve stromu, jehož kořen obsahuje klíč $k$, nejprve porovnáme $k$ s $x$. Pakliže $x < k$, zavoláme rekurzivně *SeekLeft* na levém podstromu. Pokud $x = k$, je hledaným vrcholem kořen. A pro $x > k$ zavoláme *SeekLeft* na pravém podstromu a pokud vrátí $\emptyset$, je řešením kořen.
- *SeekRight*($x$) je symetrický k *SeekLeft*.
- *Select*($i$) – opět rekurzivně: porovnáme $i$ se *size* levého dítěte kořene. Je-li $i \leq \mathit{size}$, budeme rekurzivně hledat $i$-tý prvek levého podstromu. Pro $i = \mathit{size} + 1$ je odpovědí kořen. A pro $i > \mathit{size}+1$ rekurzivně hledáme $(i - \mathit{size} - 1)$-tý prvek pravého podstromu.
- *Prev*($b$) – převedeme na *SeekLeft*($a - 1$), kde $a$ je adresa bloku $b$ (v délkovém stromu ani *Prev*, ani *Next* nepoužíváme).
- *Next*($b$) – podobně převedeme na *SeekRight*($a + 1$).

Každá operace tráví čas $\mathcal{O}(1)$ na jedné hladině stromu, celkem tedy $\mathcal{O}(\log K)$. To je tedy i složitost operací alokátoru.

Logaritmická složitost je příjemně nízká, ale přesto nám v mysli hryže červíček pochybnosti, zda to nejde rychleji. Inu, tak trochu .... Především se nám budou hodit van Emde Boasovy stromy.[5] Ty si pamatují podmnožinu univerza $\{0, \ldots, U-1\}$ s operacemi *Insert*, *Delete*, *SeekLeft* a *SeekRight*, tím pádem i *Prev* a *Next*, v čase $\mathcal{O}(\log\log U)$. Univerzem jsou v našem případě adresy a délky, obojí v rozsahu velikosti paměti, což podle zadání značíme $P$. To dává čas $\mathcal{O}(\log\log P)$, což je lepší než $\mathcal{O}(\log K)$ pro $K > \log P$. To je realistický předpoklad – alokovaných bloků bude nejspíš víc, než je logaritmus velikosti paměti.

Jenže ouha: autor rozhraní alokátoru byl tak trochu nešika, takže *Free* určuje uvolňovaný blok pořadovým číslem místo daleko praktičtější adresy. Proto potřebujeme i *Select*, ale ten už ve vEB stromu efektivně neumíme. Nejlepší známá struktura podporující *Insert*, *Delete* a *Select* pochází z Dietzova článku Optimal Algorithms for List Indexing and Subset Rank,[6] pracuje v čase $\mathcal{O}(\log U / \log\log U)$ a ví se, že její složitost optimální. Alokátor s touto strukturou by tedy běžel v čase $\mathcal{O}(\log P / \log\log P)$. To je lepší než $\mathcal{O}(\log K)$ pro $K > P^{1/\log\log P}$, což nejspíš někdy platit bude, a jindy ne.

Můžeme si tedy pořídit hybridní datovou strukturu, která začne s vyhledávacím stromem a při překročení nějakého hraničního počtu bloků $H$ přepne na kombinaci vEB stromů s Dietzovou strukturou (to vyžaduje přebudování celé struktury, což lze amortizovat přes všech $H$ bloků). Pak dostaneme složitost alokátoru $\mathcal{O}(\min(\log K, \log P / \log\log P))$.

*Úlohu připravili: Martin „Medvěd“ Mareš, Matúš Púll*

[5] https://mj.ucw.cz/vyuka/ga/7-ram.pdf

[6] https://www.academia.edu/download/70418564/tr_291.pdf

## 38-4-S Univerzální hešování

### Úkol 1: Univerzalita tabelace

Mějme nějaká dvě $kt$-bitová čísla $x$ a $y$, která si rozdělíme na $t$-bitové části $x_1, \ldots, x_k$ a $y_1, \ldots, y_k$. Aby došlo ke kolizi, musí platit

$$\bigoplus_{i=1}^{k} T_i[x_i] = \bigoplus_{i=1}^{k} T_i[y_i].$$

Jelikož $x$ a $y$ jsou různá, musí pro nějaké $j$ platit $x_j \neq y_j$. Podmínku pro kolizi tedy můžeme napsat jako

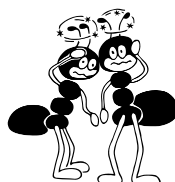
$$\left(\bigoplus_{i\neq j} T_i[x_i]\right) \oplus T_j[x_j] = \left(\bigoplus_{i\neq j} T_i[y_i]\right) \oplus T_j[y_j],$$

z čehož si vyjádříme $T_j[y_j]$ takto:

$$T_j[y_j] = \left(\bigoplus_{i\neq j} T_i[x_i]\right) \oplus T_j[x_j] \oplus \left(\bigoplus_{i\neq j} T_i[y_i]\right).$$

Zvolit náhodnou hešovací funkci do $2^b$ přihrádek znamená vyplnit tabulky náhodnými $b$-bitovými čísly. Představíme si, že tabulky vyplňujeme tak, že $T_j[y_j]$ zvolíme jako poslední. Ať už jsme ostatní položky vyplnili jakkoliv, vždy existuje právě jedna volba $T_j[y_j]$ z $2^b$ možností, pro kterou je naše rovnost splněna.

Kolize tedy nastane s pravděpodobností $1/2^b$, což je přesně to, co vyžaduje 1-univerzalita.



### Úkol 2: Hledání duplikátů

Algoritmus se bude skládat ze dvou průchodů. V prvním průchodu budeme duplikáty hledat Bloomovým filtrem. Do něj budeme vkládat všechny záznamy ze vstupu a kdykoliv filtr usoudí, že už záznam viděl, přidáme záznam mezi kandidáty na duplikát. Mezi kandidáty budou všechny skutečné duplikáty, ale kvůli nepřesnosti filtru se tam mohou dostat i další záznamy. Kandidáty si budeme udržovat v nějaké další datové struktuře (ještě upřesníme).

V druhém průchodu znovu přečteme celý vstup a spočítáme výskyty všech kandidátů. Nakonec vypíšeme ty kandidáty, kteří se vyskytli aspoň dvakrát.

Teď zvolíme parametry Bloomova filtru. Bude to vícepásmový filtr. Každé pásmo si pořídí své tabelační hešování: 32B záznamy rozdělíme na jednotlivé bajty, každým bajtem budeme indexovat samostatnou tabulku. Jedna hešovací funkce tedy použije 32 tabulek o 256 položkách po 4 B, které zabírají celkem 32 KiB.

Jelikož tabelace je 1-univerzální a jedno pásmo má mít pravděpodobnost chyby $1/2$, vychází pro $10^9$ záznamů velikost pásma $2 \cdot 10^9$ bitů. To je $1/4$ GiB, ve srovnání s tím můžeme velikost tabulek zanedbat.

Počet pásem zvolíme rovný 8. Tehdy dosáhneme pravděpodobnosti chyby $(1/2)^8 = 1/256$ a celý filtr bude měřit 2 GiB, tedy polovinu dostupné paměti.

Druhou polovinu paměti využijeme na uložení množiny kandidátů. Nejprve odhadněme, kolik jich bude. Kdyby všechny záznamy byly navzájem různé, kandidáti by vznikali pouze chybami Bloomova filtru. Těch nastane v průměru $10^9/256 \doteq 2^{32}/2^8 = 2^{24} \doteq 16\cdot 10^6$, takže zabírají v průměru $16 \cdot 10^6 \cdot 32$ B $= 512$ MiB $= 0.5$ GiB. Režie datové struktury (vyhledávacího stromu nebo hešovací tabulky) bude v porovnání s velikostí záznamu zanedbatelná.

Pokud ale budeme mít smůlu, nasbíráme kandidátů mnohem víc. Stanovíme si proto limit na velikost množiny kandidátů na dvojnásobek průměru, což pořád zaručí spotřebu paměti nejvýš 1 GiB. A pokud limit překročíme, zastavíme sběr kandidátů a rovnou začneme počítat jejich výskyty. Pokud přitom neobjevíme žádný skutečný duplikát, usoudíme, že jsme si vybrali špatnou hešovací funkci, a spustíme celý algoritmus znovu.

Jaká je pravděpodobnost, že se to stane? Podle Markovovy nerovnosti je to nejvýše $1/2$. Tím pádem podle Lemmatu o džbánu bude průměrný počet spuštění algoritmu nejvýše 2.

Celkem jsme tedy spotřebovali 3 a kousek gigabytu paměti. Zbývá odhadnout časovou složitost. Práce s Bloomovým filtrem nás stojí konstantní čas na záznam, práce s množinou kandidátů průměrně konstantní čas, pokud si ji pamatujeme v hešovací tabulce. A celý algoritmus průměrně konstanta-krát zopakujeme. Celkově je tedy průměrně lineární. (Ehm, počet záznamů byl zadán jako konstanta, takže je složitost vlastně konstantní :))

### Úkol 3: Testování perfektnosti

Máme funkci, která $n$ prvků hází do $cn^2$ přihrádek pro nějakou konstantu $c$, a chceme zjistit, jestli je prostá. Stačí čísla přihrádek zapsat v soustavě o základu $z = \lceil\sqrt{c}n\rceil$. Jelikož $z^2 \geq cn^2$, čísla budou mít 2 číslice. Tím pádem je můžeme setřídit dvouprůchodovým přihrádkovým tříděním se $z$ přihrádkami. To bude trvat čas $\mathcal{O}(2(n+z)) = \mathcal{O}(n)$.

### Úkol 4: Třídění reálných čísel

Rozhazování do přihrádek stojí $\mathcal{O}(n)$ času, bublinkové třídění $\mathcal{O}(\sum_i N_i^2)$, kde $N_i$ je počet prvků v $i$-té přihrádce.

Chceme spočítat $\mathbf{E}[\sum_i N_i^2] = \sum_i \mathbf{E}[N_i^2]$. Z rozboru perfektního hešování víme, že pro 1-univerzální systém hešovacích funkcí je tato suma v $\mathcal{O}(n)$. My ovšem do přihrádek rozdělujeme rovnoměrně náhodná čísla, takže kolize vznikají se stejnou pravděpodobností jako pro 1-univerzální systém. Musí to tedy dopadnout stejně.

*Úlohu připravil: Martin „Medvěd“ Mareš*

matfyz

KSP pro vás připravují studenti Matematicko-fyzikální fakulty Univerzity Karlovy.
Realizace projektu byla podpořena Ministerstvem školství, mládeže a tělovýchovy.

**Webové stránky:** *https://ksp.mff.cuni.cz/*
**E-mail:** *ksp@mff.cuni.cz*
**Organizátoři a kontakty:** *https://ksp.mff.cuni.cz/kontakty/*