1. Úvod do amortizované složitosti

Při analýze časové složitosti algoritmů či datových struktur skládajících se z volání posloupnosti n obslužných operací se může stát, že časy jednotlivých operací se od sebe podstatně liší. Některé operace mohou například trvat velmi dlouho a na druhou stranu jiné operace mohou být zase velmi rychlé. A to třeba tak rychlé, že výsledný čas volání všech n dílčích operací vyjde lépe, než kdybychom vynásobili n nejhorším možným časem jedné operace.

Pokud se nám tuto skutečnost podaří o algoritmu či datové struktuře precizně dokázat, dává v takovém případě smysl celkový čas rozpočítat na jednotlivé operace. Takovému vyjádření se říká amortizovaná časová složitost. Než se pustíme do jejího formálního zavedení, demonstrujeme amortizovanou analýzu na následujícím jednoduchém problému.

1.1. Telegrafistův problém

Začneme následujícím nevinným problémem ze života. Na přepážku k telegrafu chodí zákazníci a předávají zde své depeše k odeslání. Nahromaděné depeše jsou průběžně odesílány, když je zrovna volná linka. To samozřejmě nějakou dobu trvá a je navíc nezbytně nutné, aby depeše postupně odcházely v pořadí, v jakém byly podány na přepážce. To by nebylo tak těžké zařídit, kdyby ovšem telegrafista neměl na stole tak velký nepořádek, že zbývá místo už jen na dvě hromádky depeší. Na hromádku lze nahoru položit depeši nebo z hromádky naopak horní depeši vzít. Z těchto hromádek navíc nejde vytahovat depeši zespodu ani zprostředka, ani jich nelze najednou uchopit do ruky víc než jednu (rozsypaly by se). Navrhněte způsob, jakým má telegrafista postupovat, aby správně obsluhoval průběžně příchozí a odchozí depeše!

Úlohu nejprve přeformulujeme. Jak jistě bystrý čtenář postřehl, dvě hromádky s depešemi se chovají jako zásobníky⟨1⟩ A a B podporující operace Push a Pop. My však potřebujeme realizovat frontu⟨2⟩, do které v nějakém pořadí přijde a odejde n depeší, neboli potřebujeme realizovat operace Enqueue a Dequeue.

Prozkoumejme nejprve naivní přístup. Budeme udržovat depeše v zásobníku A a přidávat na jeho vrch. Kdykoli přijde požadavek Dequeue, přeházíme po jedné všechny depeše na zásobník B a úplně poslední depeši odebereme, načež obsah B opět po jedné vrátíme na A. Na operaci Enqueue tedy budeme potřebovat $\mathcal{O}(1)$ přesunů a na operaci Dequeue $\Theta(n)$ přesunů, což pro celkem n depeší bude v nejhorším případě vyžadovat celkem $\Theta(n^2)$ jednotlivých přesunů. Nejhorší případ nastane, když nejprve všechny depeše přijdou operací Enqueue a teprve potom začnou odcházet. Tento odhad bude platit, i kdybychom obě operace realizovali „líně“: obsah

---

⟨1⟩ Neboli LIFO – Last In, First Out.

⟨2⟩ Čili FIFO – First In, First Out.

B by se do A přesouval až při zavolání Enqueue a stejně tak obsah A by se přesouval do B až pří volání Dequeue – představme si, že nejprve vložíme $n/2$ depeší a potom střídavě voláme Enqueue a Dequeue.

Lepší postup vypadá následovně. Zásobník A bude sloužit jako vstupní a zásobník B jako výstupní. Enqueue zařadí depeši do A. Pokud je B neprázdný, vrátí z něj Dequeue jednoduše vrchní prvek, pokud je B prázdný, nejprve přesune obsah A do B. Protože přesun obrátí pořadí prvků, jsou dříve vložené depeše k dispozici naopak na vrchu B.

Při analýze složitosti se zdánlivě od prvního přístupu nic nezměnilo. Jedna operace Dequeue vyžaduje v nejhorším případě $\Theta(n)$ přesunů a proto i celkový počet přesunů pro n depeší bude $\mathcal{O}(n^2)$. Je celkový počet přesunů i $\Theta(n^2)$?

Ukážeme, že celková složitost je, poněkud překvapivě, nižší. Konkrétně, přestože na jednu operaci Dequeue vychází složitost $\Theta(n)$ v nejhorším případě, celkový počet operací sečtený přes libovolnou posloupnost n volání Enqueue a n volání Dequeue vyjde vždy $\Theta(n)$. Jednu depeši totiž za dobu běhu algoritmu uchopíme do ruky maximálně třikrát: poprvé při Enqueue do A, podruhé při přesunu z A do B a potřetí při Dequeue z B. Celkový počet přesunů je tudíž nejvýše $3n$ a nejméně n, což dává celkovou složitost $\Theta(n)$.

Můžeme se na tuto skutečnost dívat i tak, že celkový dosažený čas $\Theta(n)$ rozpočítáme na jednotlivé operace Enqueue a Dequeue a dostaneme tak čas $\mathcal{O}(1)$ na operaci. Tomuto způsobu analýzy složitosti říkáme *amortizovaná složitost*. Čtenář nechť si povšimne, v čem spočívá hlavní rozdíl: n vyvolání těchto operací dá celkový čas $\Theta(n)$, přestože víme, že v nejhorším případě jedna operace bude trvat čas $\Theta(n)$. Tento nejhorší případ však bude nastávat poměrně zřídka.

1.2. Zavedení amortizované složitosti

Zavedeme nyní pojem amortizované časové složitosti přesněji. Přesnou a jednotnou definici amortizované složitosti je velmi obtížné vyslovit. Skutečnost, že daná operace má amortizovanou složitost, tedy vyslovíme ve znění příslušných vět a odhadů rovnou jako odhad celkové časové složitosti při provedení sady operací. Podělíme-li tedy celkový čas počtem operací, dostaneme kýženou amortizovanou složitost.

Zdůrazňujeme, že je důležité si uvědomit, v jakém kontextu studujeme amortizovanou složitost dané operace či algoritmu. Může se totiž snadno stát, že když jednotlivá volání studované operace proložíme voláními jiné nevhodně navržené operace, tak nepříznivě změní stav datové struktury či algoritmu a studovaná operace dobrou amortizovanou složitost mít přestane.

Pokud u asymptotické složitosti neuvedeme, jakého je druhu, chápeme ji jako složitost v nejhorším případě, která se označuje *worst-case*⟨3⟩. Amortizovanou

⟨3⟩ Jazykovým puristům se omlouváme, ale pojem *složitost v nejhorším případě* je natolik dlouhý a neohrabaný, že jsme se raději rozhodli dávat přednost anglickému termínu.

složitost budeme vždy důsledně u odhadů explicitně zdůrazňovat. V literatuře se pro amortizovaný odhad občas používá i notace $\mathcal{O}^*(f(n))$, případně zkratkami jako „operace je $\mathcal{O}(f(n))$ w.c. a $\mathcal{O}(g(n))$ amort.“.

V následujících oddílech ukážeme několik běžně používaných metod pro amortizovanou analýzu složitosti, které aplikujeme na několik problémů. Konkrétně předvedeme:

- Agregační metodu, která spočívá jednoduše v spočtení celkového počtu operací. Výhodou této metody je její jednoduchost, nevýhodou naopak nepoužitelnost v případě komplikovaných datových struktur a algoritmů. Již jsme ji potkali u zásobníkové úlohy v oddílu 1.1. Dále k vidění v sekci 1.3.
- Penízkovou metodu. Představme si, že operace si schovává v různých místech datové struktury čas ve formě penízků, kterými se teprve v budoucnu zaplatí časové náročnější operace. K vidění v sekci 1.4.
- Potenciálovou metodu. Nejobecnější postup, který je podobný penízkové metodě s tím rozdílem, že zavádí celkový „účet“, kam se střádá čas do zásoby a v případě potřeby opět vybírá. K vidění v sekci 1.5.

1.3. „Nafukovací“ pole a agregační metoda

Každý programátor nejspíš už ve své praxi potkal problém, kdy potřeboval načítat do paměti prvky, jejichž počet předem neznal. K tomuto účelu existuje arzenál jednoduchých datových struktur, z nichž jmenujme například různé varianty spojového seznamu. V tomto oddílu ukážeme strategii realokace pole, která pří postupném přidávání n nových prvků do pole zajistí celkovou časovou složitost $\mathcal{O}(n)$ (a tedy amortizovanou složitost $\mathcal{O}(1)$ na přidání), i když počet prvků n není předem známý.

Zadání problému zní takto: Na vstup chodí data, která je potřeba ukládat do pole, i-tý prvek na index $i-1$. Navrhněte operaci Insert, která vkládá prvky do pole, když předem neznáme jejich počet n a nelze tedy předem alokovat vhodně velké pole.

Je zjevné, že při neznámém počtu vkládaných prvků n budeme muset nějak v průběhu pole přealokovávat. Zvolíme následující strategii: Označme kapacitu pole P v prvcích jako m a počet aktuálně uložených prvku v P jako i. Pokud dojde volné místo ($i > m$), přealokujeme P na velikost $2m$ prvků, do první poloviny nakopírujeme staré pole, které následně zrušíme, a přidávat nové prvky x začneme od indexu m. Počáteční kapacitu pole zvolíme $m = 1$. V reálné aplikaci bychom samozřejmě zvolili počáteční velikost větší, ale nám to usnadní analýzu složitosti, která však i tak vyjde příznivě.

Algoritmus INSERT(P, x)

1. Pokud je $i < m$, polož $P[i] \leftarrow x$ a $i \leftarrow i + 1$ a skonči.

2. Pokud $i = m$:
3. Alokuje pole P' o velikosti $2m$.
4. Překopíruj P do první poloviny pole P'.
5. Dealokuj P a polož $P \leftarrow P'$.
6. Polož $P[i] \leftarrow x$ a $i \leftarrow i + 1$.

Časová složitost operace Insert ve worst-case je nyní zjevně $\Theta(n)$, protože na alokaci, dealokaci a kopírování pole je třeba $\Theta(n)$ elementárních operací.

Spočteme nyní, jaká bude celková časová složitost při n-násobném vyvolání Insertu.

Věta: Uvažujme na počátku prázdné nafukovací pole. Potom celková časová složitost posloupnosti n operací Insert je $\mathcal{O}(n)$, neboli amortizovaná složitost operace Insert je $\mathcal{O}(1)$.

Důkaz: Povšimněme si, že nejhorší případ pro analýzu časové složitosti vzhledem k n nastane v okamžiku, kdy $n = 2^k + 1$ pro nějaké k, neboli že poslední zavolání Insert právě provedlo realokaci pole. Analýzu provedeme „pozpátku“, od posledního Insertu k prvnímu.

Poslední Insert vyžadoval nejvýše $2n + n + n \leq c \cdot n$ elementárních operací na alokaci, kopírování, dealokaci a $\mathcal{O}(1)$ na vložení prvku. Pak se $n/2$-krát pouze za $\mathcal{O}(1)$ vkládal nový prvek do pole. Potom $(n/2 + 1)$-ní Insert od konce opět prováděl operace spjaté s realokací, tentokrát vyžadující nejvýše $n + n/2 + n/2 \leq c \cdot n/2$ elementárních operací a opět $\mathcal{O}(1)$ na vložení prvku, a tak dále. Všechny časy za vkládání se tedy nasčítají na $\mathcal{O}(n)$, pomalé realokace se však provádějí pouze tehdy, je-li počet aktuálně uložených prvků roven mocnině dvojky.

Dostáváme tedy pro celkový čas n volání funkce Insert odhad

$$cn + c\frac{n}{2} + c\frac{n}{4} + \ldots + 1 + \mathcal{O}(n) \leq 2cn + \mathcal{O}(n) = \mathcal{O}(n).$$

To však znamená, že amortizovaná časová složitost funkce Insert je $\mathcal{O}(1)$, přestože některá volání Insert jsou lineárně pomalá. □

Čtenáře odkážeme na cvičení 1.3.1 aby si rozmyslel, že konstanta 2 není jediná vhodná, se kterou příznivě vyjde časová složitost. Zmiňme ještě, že kdybychom „nafukovali“ pole nikoli k-krát, ale zvětšovali pole o pevnou konstantu, časová složitost už příznivě nevyjde, což si čtenář může rozmyslet ve cvičení 1.3.2.

Cvičení:

1. Ukažte, že pokud přealokováváme pole nikoli na 2-násobek, ale obecně na k-násobek kde $k > 1$ je fixní konstanta, bude amortizovaná složitost operace Insert $\mathcal{O}(1)$.
2. Ukažte, že pro každou pevnou konstantu $k \geq 1$ vyjde amortizovaná složitost Insertu, který pole velikosti n přealokovává na pole velikosti $n + k$, horší než $\mathcal{O}(1)$.

1.4. Binární sčítačka a penízková metoda

Binární sčítačka je obvod, který uchovává bitovou reprezentaci čísla uloženou v buňkách nastavitelných na 0 nebo 1. Pro jednoduchost předpokládejme, že počet buněk sčítačky není omezen. Po zapnutí má sčítačka všechny bity nastaveny na 0. Sčítačka podporuje dvě operace: Inc(x) a Add(x, y). Inc zvýší reprezentované číslo ve sčítačce x o 1, Add přičte k číslu ve sčítačce x číslo ze sčítačky y.

Obě operace fungují tak, jak nás učili v první třídě: zapíšeme obě čísla v dvojkové soustavě pod sebe (v případě Inc je druhé číslo 1) a po jednotlivých řádech počínaje nejméně významným bity sčítáme. Při součtu dvou bitů 1 a 1 vznikne přenos, který je třeba přičíst k dvojici bitů vyššího řádu.

Uvažme nejprve sčítačku, která podporuje pouze operaci Inc. Zanalyzujeme složitost operace Inc, kterou budeme měřit počtem změn bitů. Worst-case složitost Inc je zjevně lineární s počtem bitů reprezentujících číslo – například čísla tvaru $2^k - 1$ mají všechny bity nastavené na 1. Všimněme si však, že každé druhé volání Inc změní pouze nejnižší bit a pak se může ukončit:

$$\begin{array}{r} 0 \\ 1 \\ 10 \\ 11 \\ 100 \\ 101 \\ 110 \\ 111 \\ 1000 \\ \vdots \end{array}$$

Podobně ke změně bitu na druhém nejnižším řádu dojde jen v každém druhém inkrementu, ke změně bitu na třetím nejnižším řádu doje v každém čtvrtém inkrementu, a tak dále. To nás vede k myšlence, že Inc má ve skutečnosti amortizovanou složitost nižší, konkrétně $\mathcal{O}(1)$.

Věta: *(o amortizované složitosti Inc)* Uvažujme na počátku nulovou binární sčítačku. Potom celková složitost měřená počtem bitových změn n volání operace Inc je $\mathcal{O}(n)$, neboli amortizovaná složitost Inc je $\mathcal{O}(1)$.

Důkaz: Důkaz provedeme následovně. Představme si, že Inc bude trvat 2 časové jednotky reprezentované dvěma mincemi $\star$. Jedna mince $\star$ zaplatí změnu jednoho bitu. Pro účely amortizované analýzy si představíme, že Inc bude střádat mince do zásoby, konkrétně, bude udržovat položenou jednu minci na každém jedničkovém bitu. Tedy třeba takto:

$$\begin{array}{ccccccccccccc} \star & \star & & \star & \star & \star & & & & \star & \star & \star & \star \\ 1 & 1 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \end{array}$$

Inc funguje tak, že kráčí zprava doleva, přehazuje jedničkové bity na 0 tak dlouho, dokud nenarazí na první 0, kterou změní na 1. Protože na každé 1 leží $\star$, je pomocí ní možno zaplatit změnu tohoto bitu na 0. V okamžiku, kdy dorazíme k první 0, máme stále k dispozici dvě $\star$ – první $\star$ zaplatíme změnu bitu z 0 na 1 a druhou $\star$ položíme do zásoby na právě vzniklou jedničku. Náš příklad se tedy změní takto:

$$\begin{array}{ccccccccccccc} \star & \star & & \star & \star & \star & & & & \star & & & \\ 1 & 1 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \end{array}$$

Inc tudíž zachovává invariant s přítomností $\star$ na každém jedničkovém bitu. To znamená, že celkový čas potřebný na posloupnost n Inc-ů za sebou lze zaplatit pro každý Inc dvěma mincemi/časovými jednotkami. Z toho nutně vyplývá, že amortizovaná složitost operace Inc je $\mathcal{O}(1)$. □

Zdůrazněme ještě, že v reálném algoritmu se samozřejmě žádné mince nikam nepokládají. Mince, jejich pokládání a sbírání je pouze naše virtuální abstrakce, která slouží k pohodlnější amortizované analýze.

Nyní rozmyslíme, co se stane, pokud přidáme operaci Add(x, y). Její worst-case složitost je zjevně $\Theta(\max(b(x), b(y)))$, kde $b(z)$ udává počet bitů v reprezentaci čísla ve sčítačce z.

Věta: Uvažujme na počátku nulovou binární sčítačku x podporující operace Inc a Add, na kterou provedeme posloupnost operací Inc a Add, z nichž n operací je Inc(x) a m operací Add(x, y_i), kde y_i je opět binární sčítačka. Potom celková složitost měřená počtem bitových změn spotřebovaná operacemi Inc je $\mathcal{O}(n)$ a operacemi Aadd $\mathcal{O}(m \min(b(x), b(y)))$, neboli amortizovaná složitost Inc je $\mathcal{O}(1)$ a Add $\mathcal{O}(\min(b(x), b(y)))$.

Důkaz: Při amortizované analýze budeme opět držet invariant, že na každém jedničkovém bitu v obou sčítačkách bude položena mince $\star$. Cenu operace Add nastavíme na $2m + 2$ časové jednotky/mince, kde $m = \min(b(x), b(y))$. Nejprve pod sebou sčítáme bity obou čísel počínaje nejméně významným bitem až do okamžiku, kdy bity jednoho z čísel na m-té pozici dojdou a zbývá pouze možný přenosový bit. Tehdy se však zbývá vyřešit již prozkoumaný problém přičítání jedničky, který je zaplatitelný dvěma mincemi. Na zaplacení změn bitů až do m-té pozice jistě postačí m mincí, dalších m mincí rozmístíme na jedničkové bity, které mohly na nejnižších m řádech vzniknout. Podle věty o amortizované složitosti Inc je tedy amortizovaná časová složitost zbývajícího inkrementu o přenosový bit $\mathcal{O}(1)$, z čehož plyne amortizovaná složitost funkce Add $\mathcal{O}(\min(b(x), b(y)))$.

Protože obě operace zachovávají invariant, že na jedničkových bitech po jejich doběhnutí zbude položená mince $\star$, z Věty o složitosti Inc vyplývá i korektnost právě uvedených amortizovaných analýz pro Inc a Add dohromady. □

Čtenáři necháme ve cvičení 1.4.3 dokázat, že amortizovaně dobře se bude chovat i k-ární sčítačka, tedy sčítačka reprezentující čísla v k-ární soustavě.

Cvičení:

1. Analyzujte amortizovanou složitost binární sčítačky agregační metodou.

2. Rozmyslete, že kdyby měla binární sčítačka podporovat zároveň funkce Inc a Dec (tedy dekrement o 1), operace rozhodně nebudou mít amortizovanou složitost $\mathcal{O}(1)$.
3. Dokažte, že k-ární sčítačka, kde $k \geq 3$ je nějaká pevná konstanta, má amortizovanou časovou složitost inkrementu $\mathcal{O}(1)$.
4. Analyzujte penízkovou metodou amortizovanou složitost operace Insert v nafukovacím poli.

1.5. Potenciálová metoda

Opusťme koncept pokládání mincí na různá místa z předchozího oddílu. Představme si, že tentokrát máme k dispozici bankovní účet, kam lze ukládat čas do zásoby a v případě potřeby z něj opět čas vybírat. Zásobní čas je tedy počítaný hromadně. Na účtu je povoleno dostat se i do záporných čísel, na závěr běhu algoritmu však musí být na účtě nezáporný obsah.

Místo názvu „účet“ budeme nadále používat pojem *potenciál* značený Φ. Klíčovým problémem je určit, kdy a jak přesně potenciál nabíjet a vybíjet. Typicky se jedná o komplikovaná pravidla, která nějak reagují na změny obsahu datové struktury nebo interního stavu algoritmu. Změnu potenciálu při jedné operaci S budeme značit $\Delta\Phi$. Pokud je $\Delta\Phi > 0$, znamená to, že operace potenciál zvětšila („nabíjela účet“), pokud $\Delta\Phi < 0$, operace si potřebovala čas půjčit z potenciálu.

Dejme tomu, že chceme o jisté operaci S dokázat, že funguje v amortizovaném čase a. Označme t reálný čas spotřebovaný operací S. Potom chceme ukázat, že

$$a = t + \Delta\Phi.$$

Jinými slovy, chceme ukázat, že deficit $t - a$ při $t > a$ je zaplacený záporným potenciálem $\Delta\Phi$ a naopak přebytek $a - t$ při $t < a$ je využit k nabití potenciálu hodnotou $\Delta\Phi$.

Proveďme nyní posloupnost operací $S_1, \ldots, S_n$. Pro operaci S_i označíme

- t_i reálný čas provádění,
- a_i zamýšlený amortizovaný čas,
- $\Delta\Phi_i$ změnu potenciálu v operaci S_i,
- Φ_i hodnotu potenciálu po provedení S_i a
- Φ_0 počáteční hodnotu potenciálu.

Celkový čas provádění operací $S_1, \ldots, S_n$ je

$$\sum_{i=1}^{n} a_i = \sum_{i=1}^{n} (t_i + \Delta\Phi_i) = \sum_{i=1}^{n} t_i + \sum_{i=1}^{n} \Delta\Phi_i = \sum_{i=1}^{n} t_i + \Phi_n - \Phi_0.$$

Pokud ukážeme, že vždy je $\Phi_n \geq \Phi_0$, znamená to, že

$$\sum_{i=1}^{n} t_i \leq \sum_{i=1}^{n} a_i$$

a tedy jsme právě pomocí amortizovaných časů a správného systému změn potenciálu shora odhadli reálný čas a dokázali tak, že operace $S_1, \dots, S_n$ mají amortizované časy $a_1, \dots, a_n$.

Shrňme tedy, co je třeba udělat pro úspěšnou amortizovanou analýzu posloupnosti operací $S_1, \dots, S_n$:

1 Vhodně definovat potenciálovou funkci Φ v závislosti na konfiguraci struktury či algoritmu. To může být poměrně náročný úkol.

2 Počáteční potenciál Φ_0 bývá typicky 0, často se používá nezáporný potenciál, ale samozřejmě to není nutné.

3 Ukázat, že $\Phi_0 \leq \Phi_n$ (aneb nezůstali jsme „dlužit čas“).

4 Dokázat, že $a_i = t_i + \Delta\Phi_i$ pro $i = 1, \dots, n$, což typicky obnáší detailně rozebrat chování i-té operace a zdůvodnit, jak se prováděné kroky zaplatí z potenciálu.

Příklady na amortizaci potenciálovou metodou

Protože je nám jasné, že předchozí teoretický úvod je třeba doprovodit konkrétními příklady, provedeme amortizovanou analýzu binární sčítačky s inkrementem z oddílu 1.4, kde provedeme n inkrementů.

Jako potenciál Φ zvolíme počet jedničkových bitů v aktuálně reprezentovaném čísle, je tedy $\Phi_0 = 0$ a jistě $\Phi_n \geq \Phi_0$. Budeme dále chtít ukázat, že $a_i = 2$ pro každé $i = 1, \dots, n$. Zjevně $\Delta\Phi_i \leq 1$, protože přidáváme nejvýše 1 jedničkový bit. Čas $t_i = 1 + b_i$ kde b_i je počet změn jedničkových bitů na nulové, dostáváme tedy $a_i = 1 + b_i + \Delta\Phi_i$. Zjevně však je $\Delta\Phi_i = 1 - b_i$, protože do potenciálu musíme zaplatit 1 za nově nahozený bit a naopak z potenciálu zaplatit shozené bity. Dostáváme tedy $a_i = 2$, což jsme chtěli dokázat.

Čtenář nechť si povšimne, že potenciálová analýza binární sčítačky vlastně přesně zrcadlí analýzu penízkovou metodou, s tím rozdílem, že všechny penízky jsou sesypány na jednu hromadu.

Jako druhý jednoduchý příklad aplikujeme potenciálovou analýzu na nafukovací pole, nad kterým provede posloupnost n Insertů. Zde bude volba potenciálu Φ následující: zvolíme $\Phi_i = 8(i - \ell_i)$, kde ℓ_i značí nejbližší nižší mocninu dvojky k i. Zjevně $\Phi_n \geq \Phi_0 = 0$. Budeme chtít ukázat, že $a_i = 9$. Pokud je $i \neq 2^k + 1$, platí $t_i = 1$ a $\Delta\Phi_i = 8$, dostáváme tedy $a_i = 9$. Pro $i = 2^k + 1$ je $t_i = 4i + 1$: za $2i$ kroků alokujeme nové pole velikosti $2i$, za i kroků překopírujeme staré pole, za i kroků dealokujeme staré pole a konečně za 1 krok přidáme nový prvek. Změna potenciálu je $\Delta\Phi_i = -4i + 8$. Dostáváme $a_i = 4i + 1 - 4i + 8 = 9$.

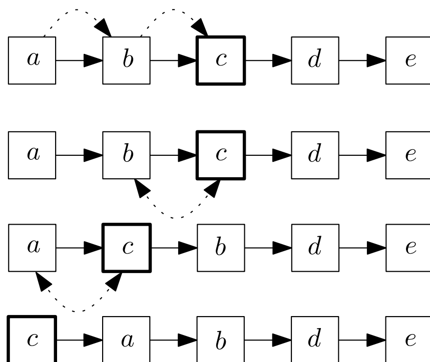
Cvičení:

1. Analyzujte potenciálovou metodou amortizovanou složitost nafukovacího pole, které místo 2-násobku používá realokaci na k-násobek pro $k > 1$.
2. Analyzujte amortizovanou složitost k-ární sčítačky potenciálovou metodou.

Analýza pravidla Move-to-front

Naše povídání o amortizované analýze potenciálovou metodou zakončíme netriviální aplikací – analýzou „move-to-front“ (MTF) heuristiky pro přistupování k datům uloženým ve spojovém seznamu.

Představme si, že máme vyrovnávací paměť (cache) a prvky v ní uložené v jednosměrném spojovém seznamu. Když nějaký prvek v seznamu vyhledáme, je typicky docela slušná naděje, že se k němu bude v budoucnu opět přistupovat. Heuristika MTF spočívá v tom, že když v seznamu vyhledáme prvek x, přesuneme ho postupnými výměnami sousedních prvků v seznamu na první pozici. Pokud k nějakým prvkům přistupujeme často, znamená to, že brzy budou všechny blízko začátku seznamu a časy na jejich vyhledání budou nízké.



Obr. 1.1: Pravidlo MTF při přístupu k prvku c.

Pravidlo MTF patří mezi tzv. *online algoritmy,* které dopředu neznají, jaká vstupní data budou následovat. Pomocí amortizované analýzy lze ukázat, že pravidlo MTF vždy dosáhne toho, že počet kroků pro sekvenci n vyhledání prvků je vždy nejhůře 4-krát horší než optimální řešení – tedy i než optimální algoritmus, který zná dopředu posloupnost dotazů.

Věta: Pro každý algoritmus A, který na základě přístupů k prvkům spojového seznamu L přeuspořádává pořadí prvků v L, platí, že pravidlo MTF potřebuje nejvýše 4-násobek počtu kroků, než kolik provede A.

Důkaz: Zvolme libovolný algoritmus A, který řeší přeuspořádávání seznamu, a uvažme posloupnost n operací přístupu k prvkům $p_1, \ldots, p_n$. Představme si, že souběžně pouštíme pravidlo MTF a udržujeme jím seznam L_M a souběžně algoritmus A, kterým udržujeme seznam L_A – obě pravidla začala s prvotním seznamem L. Pokud je prvek x v seznamu L před prvkem y, značíme tento fakt $x \prec_L y$. Definujeme potenciál po i-té operaci jako

$$\Phi_i = 2 \cdot |\{(x,y);\ (x \prec_{L_M} y \ \&\ y \prec_{L_A} x) \text{ nebo } (y \prec_{L_M} x \ \&\ x \prec_{L_A} y)\}|,$$

neboli dvakrát počet párů prvků, jejichž pořadí v L_M se liší od L_A. Například, pokud je $L_A = (a, b, c, d, e)$ a $L_M = (a, c, b, d, e)$, vyjde potenciál 2. Potenciál $\Phi_0 = 0$, protože oba algoritmy začínají s identickým seznamem. Zjevně také $\Phi_i \geq 0$ a tedy $\Phi_n \geq \Phi_0$.

Provedeme nyní analýzu přístupu k jednomu prvku p_i. Nechť p_i je na pozici k_i v L_M a na pozici ℓ_i v L_A. V MTF je cena vyhledání prvku $k_i - 1$ a cena jeho přesunu na začátek $k_i - 1$, celkem tedy $2(k_i - 1)$. V algoritmu A je cena přístupu ℓ_i. Přesun p_i na začátek L_M změní pořadí všech uspořádaných dvojic obsahujících p_i a prvků na pozicích 1 až $k - 1$, celkem tedy $k - 1$ párů. Relativní pozice ostatních párů se nezmění. V L_A je před p_i umístěno ℓ_i prvků, z nichž všechny budou v seznamu L_M po přesunu p_i na začátek položeny napravo od p_i. Z toho vyplývá, že nejvýše $\min(k_i - 1, \ell_i - 1)$ inverzních dvojic je přidáno přesunem p_i na začátek. Všechny ostatní změny pořadí (alespoň $k_i - 1 - \min(k_i - 1, \ell_i - 1)$) způsobí snížení počtu inverzních dvojic.

Změna potenciálu je tedy

$$\begin{aligned}\Delta\Phi_i \leq& 2(\min(k_i - 1, \ell_i - 1) - (k_i - 1 - \min(k_i - 1, \ell_i - 1))) \\ =& 4\min(k_i - 1, \ell_i - 1) - 2(k_i - 1).\end{aligned}$$

Amortizovanou cenu přístupu k p_i lze odhadnout

$$a_i = t_i + \Delta\Phi_i \leq k_i + 3 + 4\min(k_i - 1, \ell_i - 1) - 2(k_i - 1) \leq 4\min(k_i - 1, \ell_i - 1) \leq 4\ell_i.$$

Amortizovaná cena jednoho přístupu úpravy seznamu pravidla MTF je tedy shora omezena čtyřnásobkem ceny pravidla A.

Zatím jsme však neuvažovali, že by pravidlo A také mohlo provádět přeuspořádání seznamu L_A. Nechť A prohodí pozice dvou sousedních prvků v L_A. Toto prohození nezmění reálný čas t_i v pravidle MTF, nicméně změní (zvýší nebo sníží) nový potenciál o 2 a zvýší cenu přístupu v pravidle A o 1. Odhad na amortizovanou cenu MTF a_i stále platí, protože amortizovaná cena je sice zvýšena o 2, ale horní odhad se zvýší o 4. Tato skutečnost platí pro libovolný počet prohazovacích operací, které A vykoná. □

Protože právě dokázaná věta platí pro libovolný algoritmus A, tedy i pro optimální algoritmus, dostáváme následující důsledek.

Důsledek: Pravidlo MTF vygeneruje nejvýše 4-krát horší čas přístupů k vyhledávaným prvkům než optimální algoritmus.