1. Amortizace

Při analýze datových struktur nás zatím zajímala složitost operací v nejhorším případě. Překvapivě často se ale stává, že operace s tou nejhorší časovou složitostí se vyskytují jen zřídka a většina je mnohem rychlejší. Například, dejme tomu, že jedna operace trvá $\Theta(n)$ v nejhorším případě, ale provedení libovolných m po sobě jdoucích operací se stihne za $\mathcal{O}(n+m)$. Pro m větší než n se tak operace v dlouhodobém měřítku chová, jako by měla konstantní složitost.

Tyto úvahy vedou k pojmu *amortizované časové složitosti,* který nejprve přiblížíme na několika příkladech a poté precizně nadefinujeme.

1.1. Nafukovací pole

Představme si, že nám přicházejí nějaké prvky a my je chceme postupně ukládat na konec pole: i-tý prvek na pozici i, a nevíme předem, kolik prvků má přijít. V okamžiku vytváření pole ovšem musíme vyhradit (alokovat) nějaký počet po sobě jdoucích paměťových buněk. A ať už pole vytvoříme jakkoliv velké, časem se může stát, že se do něj další prvky nevejdou.

Tehdy nám nezbude než pole zvětšit. Jenže paměťové buňky těsně za polem mohou obsahovat jiná data, takže musíme pořídit nový blok paměti, data do něj zkopírovat a starý blok paměti uvolnit. To se snadno provede,[1] ale není to zadarmo: kopírování musí sáhnout na každý prvek, tedy celkově potřebuje lineární čas. Pole proto nesmíme zvětšovat příliš často.

Osvědčený způsob je začít s jednoprvkovým polem (nebo o nějaké jiné konstantní velikosti) a kdykoliv dojde místo, zdvojnásobit velikost. Tím vznikne takzvané *nafukovací pole.*

Pseudokód pro přidání prvku bude vypadat následovně. Proměnná P bude ukazovat na adresu pole v paměti, m bude značit velikost pole (tomu budeme říkat *kapacita*), i aktuální počet prvků a x nově vkládaný prvek.

Procedura ArrayAppend(x) (přidání do nafukovacího pole)

1. Pokud $i = m$: ◁ *už na nový prvek nemáme místo*
2. $\quad m \leftarrow 2m$
3. $\quad$ Alokujeme paměť na pole P' o velikosti m.
4. $\quad$ Pro $j = 0, \ldots, i-1$: $P'[j] \leftarrow P[j]$
5. $\quad$ Dealokujeme paměť pole P.
6. $\quad P \leftarrow P'$
7. $P[i] \leftarrow x$ ◁ *uložíme nový prvek*
8. $i \leftarrow i + 1$

[1] V některých programovacích jazycích nicméně musíme dávat pozor, aby v jiných proměnných nezůstaly ukazatele na původní pozice prvků.

Věta: Přidání n prvků do zpočátku prázdného nafukovacího pole trvá $\Theta(n)$.

Důkaz: Práce se strukturou sestává z vkládání jednotlivých prvků (každý v konstantním čase) proložených zvětšováním pole. Jedno zvětšování trvá čas $\Theta(i)$, celkový čas vkládání potom $\Theta(n)$. Ke zvětšování dochází právě tehdy, když je aktuální počet prvků mocnina dvojky. Všechna zvětšení dohromady tedy stojí $\Theta(2^0 + 2^1 + \ldots + 2^k)$, kde 2^k je nejvyšší mocnina dvojky menší než n. To je geometrická řada se součtem $2^{k+1} - 1 < 2n$. Celková časová složitost proto činí $\Theta(n)$. □

Ačkoliv je tedy složitost přidání jednoho prvku v nejhorším případě $\Theta(n)$, v posloupnosti operací se chová, jako kdyby byla konstantní. Budeme proto říkat, že je *amortizovaně konstantní.* Způsob, jakým jsme to spočítali, se nazývá *agregační metoda* – operace slučujeme neboli agregujeme do větších celků a pak zkoumáme chování těchto celků.

Zmenšování pole

Uvažujme nyní, že bychom mohli chtít prvky také odebírat. To se hodí třeba při implementaci zásobníku v poli. Tehdy se může stát, že nejprve přidáme spoustu prvků (čímž se pole nafoukne), načež většinu z nich zase smažeme a skončíme s obřím polem, v němž nejsou skoro žádné prvky. Mohlo by se proto hodit umět pole zase „vyfouknout“, abychom neplýtvali pamětí.

Nabízí se hlídat zaplnění pole a kdykoliv klesne pod polovinu, realokovat na poloviční velikost. To ale bude pomalé: představme si, že pole obsahovalo n prvků a mělo kapacitu $2n$. Pak jsme jeden prvek smazali, došlo k realokaci a pole nyní obsahuje $n - 1$ prvků a má kapacitu n. Následně přidáme 2 prvky, ty se ale nevejdou, takže pole opět realokujeme, a teď má $n + 1$ prvků a kapacitu $2n$. Nyní 1 prvek smažeme a jsme tam, kde jsme byli. Můžeme tedy pořád dokola opakovat posloupnost 4 operací, která pokaždé vynucuje pomalou realokaci.

Problém nastal proto, že „skoro prázdné“ pole se po zmenšení okamžitě stalo „skoro plným“. Pomůže tedy oddálit od sebe meze pro zvětšování a zmenšování. Obvyklé pravidlo je *zvětšovat při přeplnění, zmenšovat při poklesu zaplnění pod čtvrtinu.* Počáteční kapacitu struktury nastavíme na 1 prvek (či jinou konstantu) a pod to ji nikdy nesnížíme.

Věta: Provedení libovolné posloupnosti n operací s nafukovacím polem, v níž se libovolně střídá přidávání a odebírání prvků, trvá $\mathcal{O}(n)$.

Důkaz: Posloupnost operací rozdělíme na bloky. Blok končí okamžikem realokace nebo koncem celé posloupnosti operací. Realokaci ještě k bloku počítáme, ale přidání prvku, které realokaci způsobilo, už patří do následujícího bloku.

V každém bloku tedy přidáváme a mažeme prvky, což stojí konstantní čas na prvek, a pak nejvýše jednou realokujeme. Dokážeme, že čas strávený realokací lze „rozúčtovat“ mezi operace zadané během bloku tak, aby na každou operaci připadl konstantní čas.

Některé bloky se chovají speciálně: V posledním bloku se vůbec realokuje. V prvním bloku a možná i některých dalších obsahuje pole nejvýše 1 prvek. Čas

na realokaci v každém tomto bloku tedy můžeme omezit konstantou, což je též nanejvýš konstanta na operaci.

Zaměřme se nyní na některý ze zbývajících bloků. Označme p počet prvků v poli na začátku bloku. Předchozí blok skončil realokací a jelikož jak zmenšení, tak zvětšení pole ponechává přesně polovinu pole volnou, musí být aktuální kapacita pole přesně $2p$. K příští realokaci nás tedy donutí buďto nárůst počtu prvků na $2p$, anebo pokles na $p/2$. Aby se to stalo, musíme přidat alespoň p nebo ubrat alespoň $p/2$ prvků. Cenu $\Theta(p)$ za realokaci tedy můžeme rozpočítat mezi tyto operace tak, že každá přispěje konstantou. □

Tentokrát jsme použili takzvanou *účetní metodu*. Obecně spočívá v tom, že čas „přeúčtujeme" mezi operacemi tak, aby celkový čas zůstal zachován a nová složitost každé operace vyšla nízká.

Předchozí důkaz lze také převyprávět v řeči *hustoty datové struktury*. Tak se říká podílu počtu prvků a kapacity struktury. Naše nafukovací a vyfukovací pole udržuje hustotu v intervalu $[1/4, 1]$.

Kdykoliv hustota klesne pod $1/4$, pole zmenšujeme; při nárůstu nad 1 zvětšujeme. Po každé realokaci přitom vychází hustota přesně $1/2$, takže mezi každými dvěma realokacemi se hustota změní alespoň o $1/4$. Konstantní změna hustoty přitom odpovídá lineární změně počtu prvků, takže lineární složitost realokace rozpočítáme mezi lineárně mnoho operací a amortizovaná složitost vyjde konstantní.

Zbývá drobný detail: kdykoliv je pole prázdné, třeba na začátku výpočtu, je hustota nulová, což leží mimo povolený interval. Při prázdné struktuře je ale kapacita nanejvýš 4 (maximální kapacita při 1 prvku) a tehdy mají operace konstantní složitost i v nejhorším případě. Hustotu prázdného pole tedy nemusíme uvažovat. To odpovídá výjimce pro prázdný blok v předchozím rozboru.

Další nafukovací datové struktury

Přístup zvětšování a zmenšování kapacity podle potřeby funguje i pro jiné datové struktury, jejichž kapacita musí být v okamžiku vytváření známá. Vyzkoušejme to třeba pro haldu z oddílu **??**.

Když nám dojde kapacita haldy, pořídíme si novou, dvakrát větší haldu. Přesuneme do ní všechny prvky staré haldy a všimneme si, že jsme tím nepokazili haldové uspořádání. Stále tedy stačí, aby každý prvek na zvětšení struktury přispěl konstantním časem.

Podobně můžeme vytvářet nafukovací intervalové stromy (oddíl **??**) nebo hešovací tabulky (**??** a **??**). Tam ale nestačí data zkopírovat, strukturu je potřeba znovu vybudovat. Jelikož však budování stihneme provést v lineárním čase, zamortizuje se úplně stejně jako pouhé kopírování.

Cvičení

1. Uvažujme, že bychom pole namísto zdvojnásobování zvětšovali o konstantní počet prvků. Dokažte, se tím pokazí časová složitost.

2. Jak by to dopadlo, kdybychom m-prvkové pole rovnou zvětšovali na m^2-prvkové? Počáteční velikost musíme samozřejmě zvýšit na konstantu větší než 1.
3. K dispozici jsou dva zásobníky, které podporují pouze operace Push (přidej na vrchol zásobníku) a Pop (odeber z vrcholu zásobníku). Navrhněte algoritmus, který bude pomocí těchto dvou zásobníků simulovat frontu s operacemi Enqueue (přidej na konec fronty) a Dequeue (odeber z počátku fronty). Kromě zásobníků máte k dispozici pouze konstantní množství paměti. Ukažte, že operace s frontou budou mít amortizovaně konstantní časovou složitost.

1.2. Binární počítadlo

Další příklad, na kterém vyzkoušíme amortizovanou analýzu, je *binární počítadlo.* To si pamatuje číslo zapsané ve dvojkové soustavě a umí s ním provádět jednu jedinou operaci: Inc – zvýšení o jedničku.

0
1
10
1**1**
100
10**1**
1**10**
11**1**
1000

Obr. 1.1: Prvních 8 kroků dvojkového počítadla (změněné bity tučně)

Průběh počítání můžeme sledovat na obrázku 1.1. Pokud číslo končí nulou, Inc ji přepíše na jedničku a skončí. Končí-li jedničkami, dochází k přenosu přes tyto jedničky, takže jedničky se změní na nuly a nejbližší nula vlevo od nich na jedničku.

V pseudokódu to můžeme zapsat následovně. Číslice počítadla si budeme pamatovat v poli P, nejnižší řád bude uložený v $P[0]$, druhý nejnižší v $P[1]$, atd. Za nejvyšší jedničkou bude následovat dostatečně mnoho nul.

Algoritmus Inc

1. $i \leftarrow 0$
2. Dokud $P[i] = 1$:
3. $\quad P[i] \leftarrow 0$
4. $\quad i \leftarrow i + 1$
5. $P[i] \leftarrow 1$

Po provedení n operací bude mít počítadlo $\ell = \lfloor \log n \rfloor$ bitů. Složitost operace Inc je lineární v počtu změněných bitů. Může tedy dosáhnout $\Theta(\ell)$, třeba pokud

přecházíme z `0111...1` na `1000...0`. Překvapivě ale vyjde, že v amortizovaném smyslu je tato složitost konstantní.

Můžeme to nahlédnout agregací: nejnižší řád se mění pokaždé, druhý nejmenší v každé druhé operaci, další v každé čtvrté operaci, atd., takže celá posloupnost n operací trvá

$$\sum_{i=0}^{\ell} \left\lfloor \frac{n}{2^i} \right\rfloor \le \sum_{i=0}^{\ell} \frac{n}{2^i} \le n \cdot \sum_{i=0}^{\ell} \frac{1}{2^i} \le n \cdot \sum_{i=0}^{\infty} \frac{1}{2^i} = 2n.$$

Existuje ale elegantnější a „ekonomičtější“ způsob analýzy.

Věta: Provedení n operací Inc na zpočátku nulovém počítadle trvá $\mathcal{O}(n)$.

Důkaz: Pro potřeby analýzy si představíme, že za zavolání jedné operace Inc zaplatíme dvě mince a každá z nich reprezentuje jednotkové množství času. Některé mince spotřebujeme ihned, jiné uložíme do zásoby a použijeme později.

Konkrétně budeme udržovat invariant, že ke každému jedničkovému bitu počítadla patří jeden penízek v zásobě. Třeba takto ($\star$ symbolizuje jeden penízek):

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

Nyní provedeme Inc. Dokud přepisuje jedničky na nuly, platí za to penízky uloženými u těchto jedniček. Nakonec přepíše nulu na jedničku, za což jeden ze svých penízků rovnou utratí a druhý uloží k této jedničce. Tím jakoby předplatí její budoucí vynulování. Výsledek vypadá takto:

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

Provedeme-li tedy n operací Inc, zaplatíme $2n$ penízků. Pomocí nich zaplatíme všechny provedené operace, ty tedy trvají $\mathcal{O}(n)$. Na konci nám nějaké penízky zbudou v zásobě, ale to jistě nevadí. □

Tomuto typu úvah se říká *penízková metoda*. Obecně ji můžeme popsat takto: Slíbíme nějakou amortizovanou časovou složitost operace vyjádřenou určitým počtem penízků. Některé penízky utratíme rovnou, jiné uložíme „na horší časy“. Pozdější operace mohou tento naspořený čas využít, aby mohly běžet déle, než jsme slíbili.

Cvičení

1. Spočítejte, jak dlouho bude trvat posloupnosti n operací Inc, která začne s nenulovým stavem počítadla.
2. Rozmyslete, že kdyby mělo binární počítadlo podporovat zároveň operace Inc a Dec (snížení o 1), operace rozhodně nebudou mít konstantní amortizovanou složitost.

3.* Navrhněte jinou reprezentaci čísel, v níž půjdou provádět operace Inc, Dec a TestZero (zjisti, zda číslo je nulové) v amortizovaně konstantním čase.

4. Dokažte, že počítadlo v soustavě o základu k, kde $k \geq 3$ je nějaká pevná konstanta, také provádí Inc v amortizovaně konstantním čase.

5. Uvažujme místo Inc operaci Add(k), která k počítadlu přičte číslo k. Dokažte, že amortizovaná složitost této operace je $\mathcal{O}(\log k)$.

6. Použijte penízkovou metodu k analýze nafukovacího pole z minulého oddílu.

1.3. Potenciálová metoda

Potkali jsme několik příkladů, kdy amortizovaná složitost datové struktury byla mnohem lepší než její složitost v nejhorším případě. Důkazy těchto tvrzení byly založené na nějakém přerozdělování času mezi operacemi: někdy explicitně (účetní metoda), jindy tak, že jsme čas odkládali a využili později (penízková metoda). Pojďme se nyní na tento princip podívat obecněji.

Mějme nějakou datovou strukturu podporující různé operace. Uvažme libovolnou posloupnost operací $O_1, \ldots, O_m$ na této datové struktuře. *Skutečnou cenou* C_i operace O_i nazveme, jak dlouho tato operace doopravdy trvala. Cena závisí na aktuálním stavu struktury, ale často ji postačí odhadnout shora časovou složitostí operace v nejhorším případě. Jednotky, ve kterých cenu počítáme, si přitom můžeme zvolit tak, aby nám výpočty vyšly hezky. Jen musíme zachovat, že časová složitost operace je lineární v její ceně.

Dále každé operaci stanovíme její *amortizovanou cenu* A_i. Ta vyjadřuje naše mínění o tom, jak bude tato operace přispívat k celkovému času všech operací. Smíme ji zvolit libovolně, ale součet všech amortizovaných cen musí být větší nebo roven součtu cen skutečných. Vnějšímu pozorovateli, který nevidí dovnitř výpočtu a sleduje pouze celkový čas, můžeme tvrdit, že i-tá operace trvá A_i, a on to nemůže jakkoliv vyvrátit.

Na provedení prvních i operací jsme tedy potřebovali čas $C_1+\ldots+C_i$, ale tvrdili jsme, že to bude trvat $A_1+\ldots+A_i$. Rozdíl těchto dvou hodnot si můžeme představit jako stav jakéhosi „bankovního účtu“ na spoření času. Obvykle se mu říká *potenciál* datové struktury a značí se $\Phi_i := (\sum_{j=1}^{i} A_j) - (\sum_{j=1}^{i} C_j)$. Před provedením první operace je přirozeně $\Phi_0 = 0$.

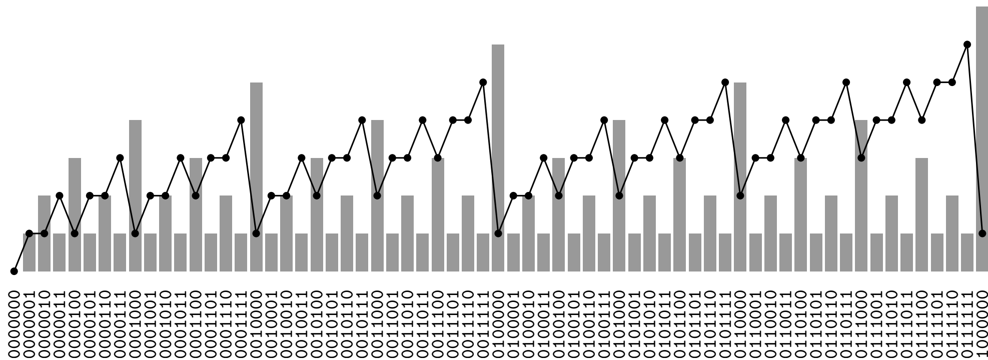
Potenciálový rozdíl $\Delta\Phi_i := \Phi_i - \Phi_{i-1}$ je pak roven $A_i - C_i$ a poznáme z něj, jestli i-tá operace čas ukládá ($\Delta\Phi_i > 0$), nebo naopak spotřebovává dříve naspořený čas ($\Delta\Phi_i < 0$).

Příklady:

- V úloze s binárním počítadlem odpovídá potenciál celkovému počtu naspořených penízků. Amortizovaná cena každé operace činí vždy 2 penízky, skutečnou cenu stanovíme jako $1 + j$, kde j je počet jedniček na konci čísla. Potenciálový rozdíl vyjde $2 - (1 + j) = 1 - j$,

což odpovídá tomu, že jedna jednička přibyla a j jich zmizelo. Sledujme obrázek 1.2: černá křivka zobrazuje vývoj potenciálu, šedivé sloupce skutečnou cenu jednotlivých operací.

- Pří analýze nafukovacího pole se jako potenciál chová počet operací s polem od poslední realokace. Všem operacím opět přiřadíme amortizovanou cenu 2, z čehož jedničku spotřebujeme a jedničku uložíme do potenciálu. Provádíme-li realokaci, tak „rozbijeme prasátko“, vybereme z potenciálu všechen naspořený čas a nahlédneme, že postačí na provedení realokace.



Obr. 1.2: Potenciál a skutečná cena operací u binárního počítadla

V obou případech existuje přímá souvislost mezi hodnotou potenciálu a stavem či historií datové struktury. Nabízí se tedy postupovat opačně: nejprve zavést nějaký potenciál podle stavu struktury a pak podle něj vypočíst amortizovanou složitost operací. Vztah $A_i - C_i = \Delta\Phi_i = \Phi_i - \Phi_{i-1}$ totiž můžeme „obrátit naruby“: $A_i = C_i + \Delta\Phi_i$. To říká, že *amortizovaná cena je rovna součtu skutečné ceny a rozdílu potenciálů.*

Pro součet všech amortizovaných cen pak platí:

$$\sum_{i=1}^{m} A_i = \sum_{i=1}^{m} (C_i + \Phi_i - \Phi_{i-1}) = \left(\sum_{i=1}^{m} C_i \right) + \Phi_m - \Phi_0.$$

Druhé sumě se říká *teleskopická:* každé Φ_i kromě prvního a posledního se jednou přičte a jednou odečte.⟨2⟩

Kdykoliv je tedy $\Phi_m \geq \Phi_0$, součet skutečných cen je shora omezen součtem amortizovaných cen a amortizace funguje.

⟨2⟩ Říká se jí tak podle starých dalekohledů, které se skládaly z do sebe zasunutých kovových válců. Má-li i-tý válec vnitřní poloměr r_i a vnější R_i, můžeme celkovou tloušťku válců spočítat jako $\sum_i (R_i - r_i)$, ale to je evidentně také rovno $R_n - r_1$.

Shrneme, jak se potenciálová metoda používá:

- Definujeme vhodnou potenciálovou funkci Φ v závislosti na stavu datové struktury. Na to neexistuje žádný univerzální návod, ale obvykle chceme, aby potenciál byl tím větší, čím víc se blížíme k operaci, která bude trvat dlouho.
- Ukážeme, že $\Phi_0 \leq \Phi_m$ (aneb nezůstali jsme „dlužit čas“). Často náš potenciál vyjadřuje počet něčeho ve struktuře, takže je přirozeně na počátku nulový a pak vždy nezáporný. Tehdy požadovaná nerovnost triviálně platí.
- Vypočteme amortizovanou cenu operací ze skutečné ceny a potenciálového rozdílu: $A_i = C_i + \Phi_i - \Phi_{i-1}$. Často ji neumíme vyjádřit přesně (třeba proto, že neznáme přesné C_i), a tak se spokojíme s horním odhadem.
- Někdy ještě upravíme multiplikativní konstantu u skutečné ceny (tedy zvolíme vhodnou jednotku času), aby cena lépe odpovídala zvolenému potenciálu.

Amortizovaná analýza je užitečná v případech, kdy datovou strukturu používáme uvnitř nějakého algoritmu. Tehdy nás nezajímají konkrétní časy operací, nýbrž to, jak datová struktura ovlivňuje časovou složitost celého algoritmu. Mohou se tudíž hodit i struktury, které mají špatnou časovou složitost v nejhorším případě (říká se také *worst-case složitost*), ale dobrou amortizovanou.

Jsou ovšem případy, kdy to nestačí: programujeme-li automatické řízení letadla, musíme na události reagovat okamžitě. Kdybychom reakci odložili, protože zrovna uklízíme datové struktury, program by mohl doslova spadnout.

Dodejme ještě, že bychom si neměli plést amortizovanou a průměrnou složitost. Průměr počítaný přes všechny možné vstupy nebo přes všechny možné průběhy randomizovaného algoritmu (blíže viz kapitola **??**) obvykle nic neslibuje o konkrétním vstupu. Naproti tomu amortizovaná složitost nám dá spolehlivý horní odhad času pro libovolnou posloupnost operací, jen neprozradí, jak bude tento čas rozdělen mezi jednotlivé operace.

Cvičení

1. Analyzujte potenciálovou metodou amortizovanou složitost nafukovacího pole, které místo na dvojnásobek realokuje na k-násobek pro nějaké pevné $k > 1$.
2. *Okénková minima:* Na vstupu postupně přicházejí čísla. Kdykoliv přijde další, vypište minimum z posledních k čísel. Na rozdíl od cvičení **??** existuje i řešení pracující v amortizovaně konstantním čase na operaci.
3. ** Vyřešte předchozí cvičení ve worst-case konstantním čase.
4. *Minimový strom* pro posloupnost $x_1, \ldots, x_n$ navzájem různých prvků je definován takto: v kořeni leží prvek x_j s nejmenší hodnotou, levý podstrom je minimovým stromem pro $x_1, \ldots, x_{j-1}$, pravý podstrom pro $x_{j+1}, \ldots, x_n$. Navrhněte algoritmus, který sestrojí minimový strom v čase $\mathcal{O}(n)$.

5.* V binárním vyhledávacím stromu budeme provádět operace FIND (nalezení prvku se zadaným klíčem) a SUCC (nalezení následníka prvku, který nám vrátila předchozí operace FIND nebo SUCC). Najděte potenciál, vůči kterému vyjde amortizovaná složitost FIND $\mathcal{O}(\log n)$ a SUCC $\mathcal{O}(1)$.

6. $(2,3)$-*stromy* z oddílu **??**: Operaci INSERT rozdělíme na hledání ve stromu a *strukturální změny* stromu: to jsou úpravy klíčů a ukazatelů uložených ve vrcholech. Ukažte, že pokud na původně prázdný $(2,3)$-strom aplikujeme jakoukoliv posloupnost INSERTů, každý z nich provede amortizovaně konstantní počet strukturálních změn. Zobecněte pro libovolné (a,b)-stromy.

7.* Podobně jako v předchozím cvičení budeme počítat strukturální změny v (a,b)-stromu, ale tentokrát pro libovolnou kombinaci operací INSERT a DELETE. Dokažte, že ve $(2,4)$-stromu každá taková operace provede amortizovaně $\mathcal{O}(1)$ změn, zatímco ve $(2,3)$-stromech je jich lineárně mnoho. Zobecněte na $(a,2a)$-stromy a $(a,2a-1)$-stromy.

1.4. Líné vyvažování stromů

Při ukládání dat do binárního vyhledávacího stromu potřebujeme strom vyvažovat, abychom udrželi logaritmickou hloubku. Potkali jsme několik vyvažovacích technik pracujících v logaritmickém čase, nicméně všechny byly dost pracné. Nyní předvedeme mnohem jednodušší variantu stromů. Složitost vyvažování sice budeme mít v nejhorším případě lineární, ale amortizovanou stále logaritmickou.

Vzpomeňme na definici dokonalé vyváženosti: ta požaduje, aby pro každý vrchol platilo, že velikosti jeho podstromů se liší nejvýše o 1. Tedy že poměr těchto velikostí je skoro přesně 1 : 1. Ukázalo se, že je to příliš přísná podmínka, takže ji nelze efektivně udržovat. Tedy ji trochu uvolníme: poměr velikostí podstromů bude ležet někde mezi 1 : 2 a 2 : 1. Totéž můžeme formulovat pomocí poměru mezi velikostí podstromů otce a synů:[3]

Definice: V binárním stromu zavedeme *mohutnost vrcholu* $m(v)$ jako počet vrcholů v podstromu zakořeněném pod v (list má tedy mohutnost 1). Strom je *v rovnováze,* pokud pro každý vrchol v a jeho syna s platí $m(s) \leq 2/3 \cdot m(v)$.

Lemma: Strom o n vrcholech, který je v rovnováze, má hloubku $\mathcal{O}(\log n)$.

Důkaz: Sledujme, jak se mění mohutnosti vrcholů na libovolné cestě z kořene do listu. Kořen má mohutnost n, každý další vrchol má mohutnost nejvýše $2/3$ předchozího, až dojdeme do listu s mohutností 1. Cesta tedy může být dlouhá nejvýše $\log_{2/3}(1/n) = \log_{3/2} n = \mathcal{O}(\log n)$. □

Nyní rozmyslíme, jak strom udržovat v rovnováze. Abychom mohli rovnováhu kontrolovat, zapamatujeme si v každém vrcholu jeho mohutnost.

[3] Proč jsme nezůstali u původní definice pomocí poměru velikostí podstromů? To proto, že je-li některý z podstromů prázdný, dělili bychom nulou.

Budeme předpokládat, že do stromu pouze vkládáme nové prvky; mazání ponecháme jako cvičení. Vyjdeme z algoritmu BvsInsert pro obyčejný vyhledávací strom. Ten se pokusí nový prvek najít a když se mu to nepovede, přidá nový list. Navíc se pak budeme vracet z přidaného listu zpět do kořene a všem vrcholům po cestě zvyšovat mohutnost o 1 (ostatním vrcholům se mohutnost evidentně nezmění).

Kdekoliv změníme mohutnost, zkontrolujeme, zda je stále splněna podmínka rovnováhy. Pokud všude je, jsme hotovi. V opačném případě nalezneme nejvyšší vrchol, v němž je porušena, a celý podstrom pod tímto vrcholem rozebereme a přebudujeme na dokonale vyvážený strom. Rozebrání a přebudování stihneme v lineárním čase (blíže viz cvičení **??**).

Insert do našeho stromu se tedy nezatěžuje lokálními nepravidelnostmi, pouze udržuje rovnováhu. To je takový „líný" přístup – dokud situace není opravdu vážná, tváříme se, jako by nic. Rovnováha nám zaručuje logaritmickou hloubku, tím pádem i logaritmickou složitost vyhledávání. A jakmile nevyváženost překročí kritickou mez, uklidíme ve stromu pořádně, což sice potrvá dlouho, ale pak zase dlouho nebudeme muset nic dělat.

Věta: Amortizovaná složitost operace Insert s líným vyvažováním je $\mathcal{O}(\log n)$.

Důkaz: Zavedeme šikovný potenciál. Měl by vyjadřovat, jak daleko jsme od dokonale vyváženého stromu: vkládáním by měl postupně růst a jakmile nějaký podstrom vyvedeme z rovnováhy, potenciál by měl být dostatečně vysoký na to, abychom z něj zaplatili přebudování podstromu.

Potenciál proto definujeme jako součet příspěvků jednotlivých vrcholů, přičemž každý vrchol přispěje rozdílem mohutností svého levého a pravého syna (chybějícím synům přisoudíme nulovou mohutnost). Budeme ovšem potřebovat, aby dokonale vyvážený podstrom měl potenciál nulový, takže přidáme výjimku: pokud se mohutnosti liší přesně o 1, příspěvek bude nula.

$$\Psi := \sum_v \psi(v), \quad \text{kde}$$
$$\psi(v) := \begin{cases} |m(\ell(v)) - m(r(v))| & \text{pokud je to alespoň 2,} \\ 0 & \text{jinak.} \end{cases}$$

Přidáme-li nový list, vrcholům na cestě mezi ním a kořenem se zvýší mohutnost o 1, tím pádem příspěvky těchto vrcholů k potenciálu se změní nejvýše o 2 (obvykle o 1, ale pokud je zrovna rozdíl vah jedničkový, příspěvek skočí z 0 rovnou na 2 či opačně).

Pokud nedošlo k žádnému přebudování, strávili jsme $\mathcal{O}(\log n)$ času průchodem cesty tam a zpět a změnili potenciál taktéž o $\mathcal{O}(\log n)$. To dává amortizovanou složitost $\mathcal{O}(\log n)$.

Nechť tedy nastane přebudování. V nějakém vrcholu v platí, že jeden ze synů, bez újmy na obecnosti $\ell(v)$, má příliš velkou mohutnost relativně k otci: $m(\ell(v)) > 2/3 \cdot m(v)$. Opačný podstrom tedy musí mít naopak malou mohutnost, $m(r(v)) <$

$1/3 \cdot m(v)$. Příspěvek $\psi(v)$ proto činí aspoň $1/3 \cdot m(v)$. Tento příspěvek se přebudováním vynuluje, stejně tak příspěvky všech vrcholů ležících pod v; ostatním vrcholům se příspěvky nezmění. Potenciál tedy celkově klesne alespoň o $1/3 \cdot m(v)$.

Skutečná cena přebudování činí $\Theta(m(v))$, takže pokles potenciálu o řádově $m(v)$ ji vyrovná a amortizovaná cena přebudování vyjde nulová. (Kdybychom chtěli být přesní, vynásobili bychom potenciál vhodnou konstantou, aby pokles potenciálu přebil i konstantu z Θ.) □

Na závěr zmíníme, že myšlenku vyvažování pomocí mohutností podstromů popsal již v roce 1972 Edward Reingold. Jeho BB-α stromy byly ovšem o něco složitější a vyvažovaly se pomocí rotací.

Cvičení

1. Doplňte operaci Delete. Pro analýzu použijte tentýž potenciál.
2.* Co by se pokazilo, kdybychom v definici $\psi(v)$ neudělali výjimku pro rozdíl 1?

1.5.* Splay stromy

Ukážeme ještě jeden pozoruhodný přístup k vyhledávacím stromům, který vede na logaritmickou amortizovanou složitost. Objevili ho v roce 1983 Daniel Sleator a Robert Tarjan. Je založený na prosté myšlence: kdykoliv chceme pracovat s nějakým vrcholem, postupnými rotacemi ho „vytáhneme“ až do kořene stromu. Této operaci budeme říkat Splay a budeme mluvit o *splayování*⟨4⟩ vrcholu.

Rotace na cestě od vybraného prvku do kořene ovšem můžeme volit více způsoby a většina z nich nevede k dobré složitosti (viz cvičení 1). Kouzlo spočívá v tom, že budeme preferovat dvojité rotace. Možné situace při splayování vrcholu x vidíme na obrázku 1.3: Je-li x levým synem levého syna, provedeme krok typu LL. Podobně krok LP pro pravého syna levého syna. Kroky PP a PL jsou zrcadlovými variantami LL a LP. Konečně pokud už x je synem kořene, provedeme jednoduchou rotaci, tedy krok L nebo jeho zrcadlovou variantu P.

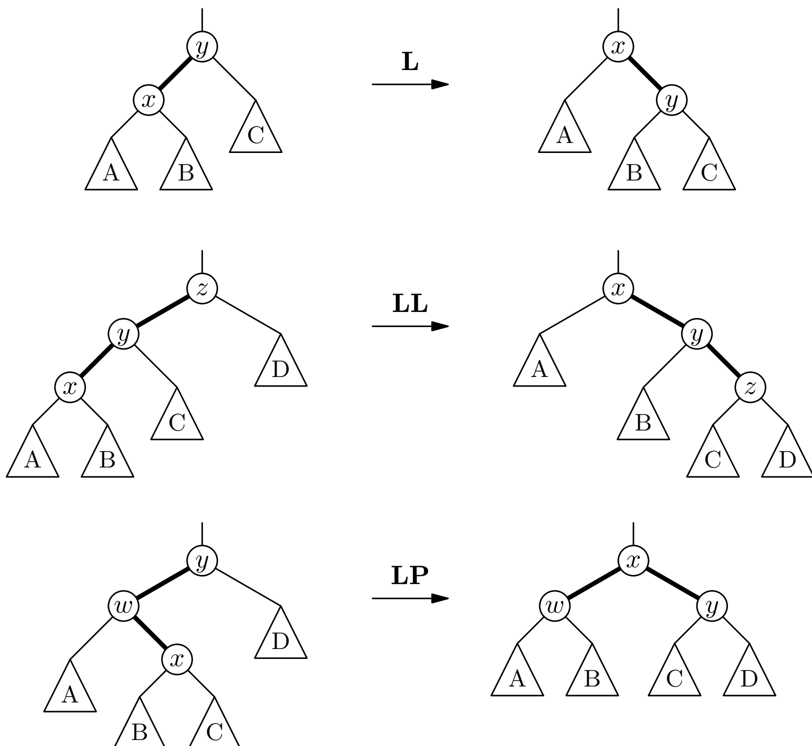
Jak se z těchto kroků složí celé splayování, můžeme pozorovat na obrázku 1.4: Nejprve provedeme krok PP, pak opět PP, a nakonec P. Všimněte si, že splayování má tendenci přetvářet dlouhé cesty na rozvětvenější stromy. To nám dává naději, že nahodile vzniklé degenerované části nás nebudou dlouho brzdit.

Pojďme se pustit do amortizované analýzy. Základem je následující potenciál. Vypadá poněkud magicky – Sleator s Tarjanem ho vytáhli jako králíka z kouzelnického klobouku, aniž by za ním byla vidět jasná intuice. Jakmile známe potenciál, zbytek už bude snadný.

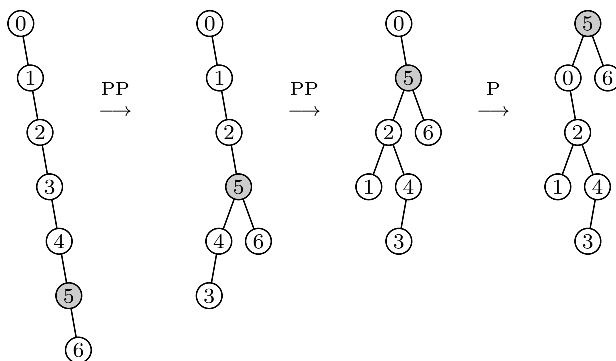
Definice:

- $T(v)$ označíme podstrom zakořeněný ve vrcholu v,

⟨4⟩ Anglické *splay* znamená zešikmení či zkosení. Nám však průběh operace nic takového nepřipomíná, takže raději strpíme neelegantní anglicismus.



Obr. 1.3: Splayovací kroky L, LL a LP

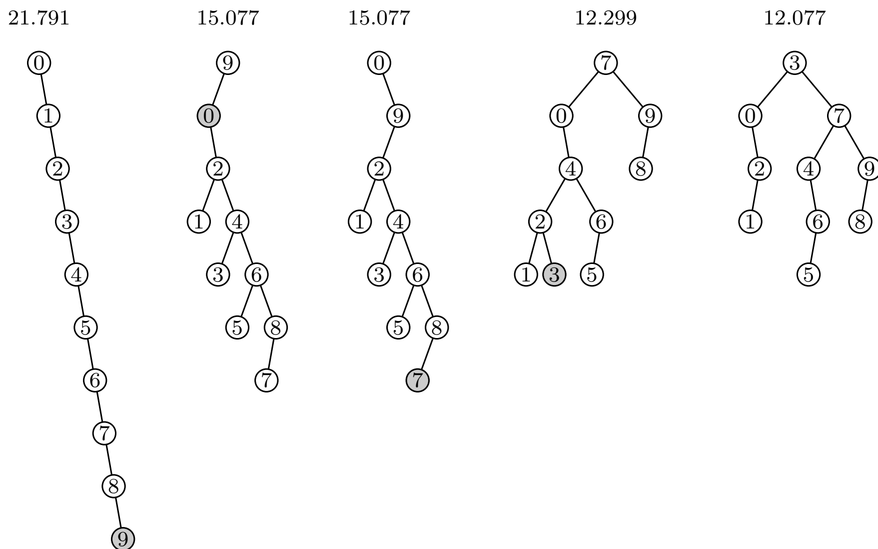


Obr. 1.4: Postup splayování vrcholu 5

- *mohutnost vrcholu* $m(v)$ je počet vrcholů v podstromu $T(v)$,
- *rank vrcholu* $r(v)$ je dvojkový logaritmus mohutnosti $m(v)$,

- *potenciál* splay stromu je součet ranků všech vrcholů.

Obrázek 1.5 naznačuje, že vyšší potenciály opravdu odpovídají méně vyváženým stromům – cesta se postupným splayováním proměňuje na košatý strom a potenciál při tom vytrvale klesá, tím víc, čím víc práce nám splayování dalo.



Obr. 1.5: Vývoj potenciálu během splayování hodnot 9, 0, 7, 3

Dokážeme, že tomu tak je obecně. Cenu operace Splay budeme měřit počtem provedených rotací (takže dvojitá rotace se počítá za dvě); skutečná časová složitost je zjevně lineární v této ceně. Platí následující věta:

Věta: Amortizovaná cena operace Splay(x) je nejvýše $3\cdot(r'(x)-r(x))+1$, kde $r(x)$ je rank vrcholu x před provedením operace a $r'(x)$ po něm.

Nyní větu dokážeme. Čtenáři, kteří se zajímají především o další operace se splay stromy, mohou přeskočit na stranu 16 a pak se případně k důkazu vrátit.

Důkaz: Amortizovaná cena operace Splay je součtem amortizovaných cen jednotlivých kroků. Označme $r_1(x),\ldots,r_t(x)$ ranky vrcholu x po jednotlivých krocích splayování a $r_0(x)$ rank před prvním krokem.

V následujících lemmatech dokážeme, že cena každého kroku je shora omezena $3r_i(x) - 3r_{i-1}(x)$. Jedinou výjimku tvoří kroky L a P, které mohou být o jedničku dražší. Jelikož preferujeme dvojrotace, nastane takový krok nejvýše jednou. Pro celkovou cenu tedy dostáváme:

$$A \le \sum_{i=1}^{t}\bigl(3r_i(x) - 3r_{i-1}(x)\bigr) + 1.$$

To je teleskopická suma: kromě r_0 a r_t se každý rank jednou přičte a jednou odečte, takže pravá strana je rovna $3r_t(x) - 3r_0(x) + 1$. To dává tvrzení věty. □

Nyní doplníme výpočty ceny jednotlivých typů kroků. Vždy budeme splayovat vrchol x, nečárkované proměnné budou odpovídat stavu před provedením kroku a čárkované stavu po něm. Nejprve ovšem dokážeme obecnou nerovnost o logaritmech.

Lemma (o průměru logaritmů): Pro každá dvě kladná reálná čísla α, β platí

$$\log \frac{\alpha + \beta}{2} \geq \frac{\log \alpha + \log \beta}{2}.$$

Důkaz: Požadovaná nerovnost platí kromě logaritmu pro libovolnou *konkávní* funkci f. Tak se říká funkcím, pro jejichž graf platí, že úsečka spojující libovolné dva body na grafu leží celá pod grafem (případně se může grafu dotýkat). Konkávnost se pozná podle záporné druhé derivace.

Pro logaritmus to jde snadno: první derivace přirozeného logaritmu $\ln x$ je $1/x$, druhá $-1/x^2$, což je záporné pro každé $x > 0$. Dvojkový logaritmus je $(\ln 2)$-násobkem přirozeného logaritmu, takže je také konkávní.

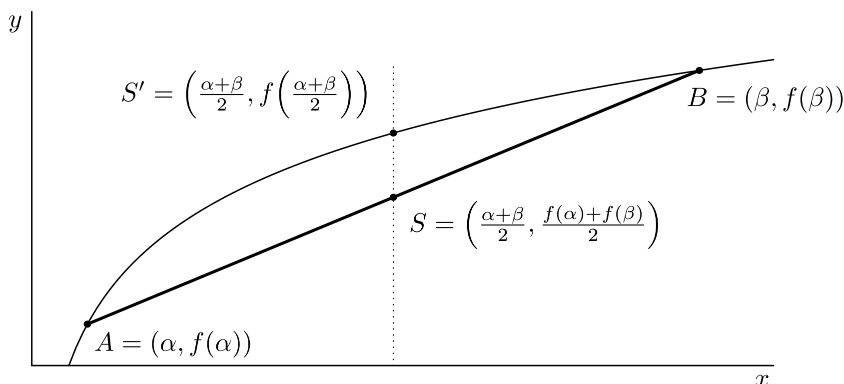
Uvažujme nyní graf nějaké konkávní funkce f na obrázku 1.6. Vyznačíme v něm body $A = (\alpha, f(\alpha))$ a $B = (\beta, f(\beta))$. Najdeme střed S úsečky AB. Jeho souřadnice jsou průměrem souřadnic krajních bodů, tedy

$$S = \left(\frac{\alpha + \beta}{2}, \frac{f(\alpha) + f(\beta)}{2} \right).$$

Díky konkávnosti musí bod S ležet pod grafem funkce, tedy speciálně pod bodem

$$S' = \left(\frac{\alpha + \beta}{2}, f\left(\frac{\alpha + \beta}{2} \right) \right).$$

Porovnáním y-ových souřadnic bodů S a S' získáme požadovanou nerovnost. □



Obr. 1.6: Průměrová nerovnost pro konkávní funkci f

Důsledek: Jelikož $\log\frac{\alpha+\beta}{2} = \log(\alpha+\beta)-1$, můžeme také psát $\log\alpha+\log\beta \le 2\log(\alpha+\beta)-2$.

Lemma LP: Amortizovaná cena kroku typu LP je nejvýše $3r'(x)-3r(x)$.

Důkaz: Sledujme obrázek 1.3 a uvažujme, jak se změní potenciál. Jediné vrcholy, jejichž rank se může změnit, jsou w, x a y. Potenciál tedy vzroste o $(r'(w)-r(w)) + (r'(x)-r(x)) + (r'(y)-r(y))$. Skutečná cena operace činí 2 jednotky, takže pro amortizovanou cenu A platí:

$$A = 2 + r'(w) + r'(x) + r'(y) - r(w) - r(x) - r(y).$$

Chceme ukázat, že $A \le 3r'(x) - 3r(x)$. Potřebujeme proto ranky ostatních vrcholů nějak odhadnout pomocí $r(x)$ a $r'(x)$.

Na součet $r'(w)+r'(y)$ využijeme lemma o průměru logaritmů:

$$\begin{aligned} r'(w)+r'(y) &= \log m'(w) + \log m'(y) \\ &\le 2\log(m'(w)+m'(y)) - 2. \end{aligned}$$

Protože podstromy $T'(w)$ a $T'(y)$ jsou disjunktní a oba leží pod x, musí platit $\log(m'(w)+m'(y)) \le \log m'(x) = r'(x)$. Celkem tedy dostáváme:

$$r'(w)+r'(y) \le 2r'(x) - 2.$$

To dosadíme do nerovnosti pro A a získáme:

$$A \le 3r'(x) - r(w) - r(x) - r(y).$$

Zbývající ranky můžeme odhadnout triviálně:

$$\begin{aligned} r(w) &\ge r(x) \quad \text{protože } T(w) \supseteq T(x), \\ r(y) &\ge r(x) \quad \text{protože } T(y) \supseteq T(x). \end{aligned}$$

Dostáváme tvrzení lemmatu. □

Lemma LL: Amortizovaná cena kroku typu LL je nejvýše $3r'(x)-3r(x)$.

Důkaz: Budeme postupovat podobně jako u kroku LP. Skutečná cena je opět 2, ranky se mohou změnit pouze vrcholům x, y a z, takže amortizovaná cena činí

$$A = 2 + r'(x) + r'(y) + r'(z) - r(x) - r(y) - r(z).$$

Chceme se zbavit všech členů kromě $r(x)$ a $r'(x)$.

Zase by se nám hodilo použít průměrové lemma na nějaké dva podstromy, které jsou disjunktní a dohromady obsahují skoro všechny vrcholy. Tentokrát se nabízí $T(x)$ a $T'(z)$:

$$\begin{aligned} r(x)+r'(z) &= \log m(x) + \log m'(z) \\ &\le 2\log(m(x)+m'(z)) - 2 \\ &\le 2\log m'(x) - 2 = 2r'(x) - 2. \end{aligned}$$

To je ekvivalentní s nerovností $r'(z) \leq 2r'(x) - r(x) - 2$. Tím pádem:

$$A \leq 3r'(x) + r'(y) - 2r(x) - r(y) - r(z).$$

Zbylé nežádoucí členy odhadneme elementárně:

$$\begin{aligned} r(z) = r'(x) &\quad \text{protože } T(z) = T'(x), \\ r(y) \geq r(x) &\quad \text{protože } T(y) \supseteq T(x), \\ r'(y) \leq r'(x) &\quad \text{protože } T'(y) \subseteq T'(x). \end{aligned}$$

Z toho plyne požadovaná nerovnost $A \leq 3r'(x) - 3r(x)$. □

Lemma L: Amortizovaná cena kroku typu L je nejvýše $3r'(x) - 3r(x) + 1$.

Důkaz: Skutečna cena je 1, ranky se mohou měnit jen vrcholům x a y, takže amortizovaná cena vyjde:

$$A = 1 + r'(x) + r'(y) - r(x) - r(y).$$

Z inkluze podstromů plyne, že $r'(y) \leq r'(x)$ a $r(y) \geq r(x)$, takže:

$$A \leq 1 + 2r'(x) - 2r(x).$$

Z inkluze ale také víme, že $r'(x) - r(x)$ nemůže být záporné, takže tím spíš platí i $A \leq 1 + 3r'(x) - 3r(x)$, což jsme chtěli. □

Důsledek: Jelikož definice ranku je symetrická vzhledem k prohození stran, kroky typů PP, PL a P mají stejné amortizované ceny jako LL, LP a L.

Hledání podle klíče

Dokázali jsme, že amortizovaná složitost operace SPLAY(x) je $\mathcal{O}(r'(x) - r(x) + 1)$, kde $r(x)$ a $r'(x)$ jsou ranky vrcholu x před operací a po ní. Ranky jakožto logaritmy mohutností nikdy nepřekročí $\log n$, takže složitost evidentně leží v $\mathcal{O}(\log n)$. Nyní ukážeme, jak pomocí splayování provádět běžné operace s vyhledávacími stromy.

Operaci FIND, tedy vyhledání prvku podle klíče, provedeme stejně jako v obyčejném vyhledávacím stromu a nakonec nalezený prvek vysplayujeme do kořene. Kdybychom klíč nenalezli, vysplayujeme poslední navštívený vrchol. Samotné hledání trvá lineárně s hloubkou posledního navštíveného vrcholu. Práci lineární s hloubkou ovšem vykoná i splayování, takže vychází-li amortizovaná složitost splayování $\mathcal{O}(\log n)$, musí totéž vyjít i pro hledání.

Můžeme si to představit také tak, že operaci SPLAY naúčtujeme i čas spotřebovaný hledáním. Tím se splayování zpomalí nejvýše konstanta-krát, takže stále bude platit amortizovaný odhad $\mathcal{O}(\log n)$.

Vkládání prvků

Operaci INSERT implementujeme také obvyklým způsobem, takže nový prvek se stane listem stromu. Ten posléze vysplayujeme do kořene. Opět můžeme složitost hledání správného místa pro nový list naúčtovat provedenému splayi. Je tu ovšem drobný háček: připojení listu zvýší ranky vrcholů ležících mezi ním a kořenem, takže musíme do složitosti INSERTu započítat i zvýšení potenciálu. Naštěstí je pouze logaritmické.

Lemma: Přidání listu zvýší potenciál o $\mathcal{O}(\log n)$.

Důkaz: Označíme $v_1, \ldots, v_t$ vrcholy na cestě z kořene do nového listu ℓ. Nečárkované proměnné budou jako obvykle popisovat stav před připojením listu, čárkované stav po něm. Potenciálový rozdíl činí:

$$\Delta\Phi = \sum_{i=1}^{t} \bigl(r'(v_i) - r(v_i)\bigr) + r'(\ell).$$

Jelikož ℓ je list, má jednotkovou mohutnost a nulový rank. Nový rank v_i je $r'(v_i) = \log m'(v_i) = \log(m(v_i) + 1)$, jelikož v podstromu $T(v_i)$ přibyl právě list ℓ.

Logaritmus výrazu $m(v_i)+1$ vzdoruje pokusům o úpravu, ale můžeme ho trochu nečekaně odhadnout shora pomocí $m(v_{i-1})$. Vskutku, podstrom $T(v_{i-1})$ obsahuje vše, co leží v $T(v_i)$, a ještě navíc vrchol v_{i-1}. Proto musí platit $m(v_{i-1}) \geq m(v_i)+1 = m'(v_i)$, a tím pádem také $r(v_{i-1}) \geq r'(v_i)$.

Tuto nerovnost dosadíme do vztahu pro potenciálový rozdíl (případ $i = 1$ jsme museli ponechat zvlášť):

$$\Delta\Phi \leq \bigl(r'(v_1) - r(v_1)\bigr) + \sum_{i=2}^{t} \bigl(r(v_{i-1}) - r(v_i)\bigr).$$

Tato suma je teleskopická – většina ranků se jednou přičte a jednou odečte. Získáme

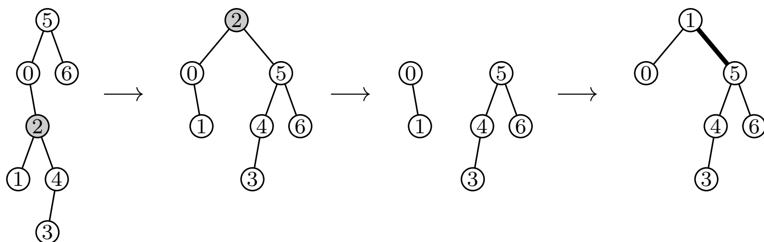
$$\Delta\Phi \leq r'(v_1) - r(v_t),$$

a to je jistě nejvýše logaritmické. □

Mazání prvků

Z klasických operací zbývá DELETE. Ten provedeme netradičně: nalezneme zadaný vrchol a vysplayujeme ho do kořene. Pak ho odebereme, čímž se strom rozpadne na levý a pravý podstrom. Nyní nalezneme maximum levého podstromu a opět ho vysplayujeme. Tím jsme levý strom dostali do stavu, kdy jeho maximum leží v kořeni a nemá pravého syna. Jako pravého syna mu tedy můžeme připojit kořen pravého podstromu, čímž podstromy opět spojíme.

Amortizovaná analýza se provede podobně, jen musíme pracovat s více stromy najednou. To je snadné: potenciály stromů sečteme do jednoho celkového potenciálu.



Obr. 1.7: Postup mazání ze splay-stromu

Hledání mazaného prvku a minima v podstromu naúčtujeme následujícímu splayování. Odebrání kořene potenciál sníží o rank kořene, ostatní vrcholy přispívají stále stejně. Nakonec spojíme dva stromy do jednoho, čímž vzroste pouze rank nového kořene, nejvýše o $\log n$.

Dokázali jsme tedy, že amortizovaná časová složitost operací Find, Insert a Delete ve splay stromu je $\mathcal{O}(\log n)$. Algoritmus mazání navíc můžeme rozdělit na operace Split (rozdělení stromu okolo daného prvku na dva) a Join (slepení dvou stromů, kde všechny prvky jednoho jsou menší než všechny prvky druhého, do jednoho stromu), obě rovněž amortizovaně logaritmické.

Zmíníme ještě jednu pozoruhodnou vlastnost splay stromů: často používané prvky mají tendenci hromadit se blízko kořene, takže přístup k nim je rychlejší než k prvkům, na které saháme zřídka. Blíže o tom ve cvičení 5.

Cvičení

1. *Naivní splayování:* Ukažte, že kdybychom splayovali pouze jednoduchými rotacemi, tedy kroky typu L a P, amortizovaná složitost operace Splay by byla lineární. Dobře je to vidět na stromech ve tvaru cesty.
2. *Splayování shora dolů:* Nevýhodou splay stromů je, že se po nalezení prvku musíme po stejné cestě ještě vrátit zpět, abychom prvek vysplayovali. Ukažte, jak splayovat už během hledání prvku, tedy shora dolů.

3.* *Sekvenční průchod:* Dokažte, že pokud budeme postupně splayovat vrcholy stromu od nejmenšího po největší, potrvá to celkem $\mathcal{O}(n)$.

4.* *Vážená analýza:* Vrcholům splay stromu přiřadíme *váhy*, což budou nějaká kladná reálná čísla. Algoritmus samotný o vahách nic neví, ale použijeme je při analýze. Mohutnost vrcholu předefinujeme na součet vah všech vrcholů v podstromu. Rank a potenciál ponecháme definovaný stejně. Dokažte, že věta o složitosti splayování stále platí. Rozeberte složitost operací Find, Insert a Delete. Pozor na to, že ranky už nemusí být logaritmické, takže složitosti budou záviset na vahách. A také pozor, že pro váhy v intervalu $(0,1)$ mohou ranky vycházet záporné.

5.* *Náhodné přístupy:* Uvažujme splay strom, který si pamatuje prvky $x_1, \ldots, x_n$ a přicházejí na ně dotazy náhodně s pravděpodobnostmi $p_1, \ldots, p_n$, přičemž

$\sum_i p_i = 1$. Využijte výsledku předchozího cvičení a dokažte, že amortizovaná složitost přístupu k x_i je $\mathcal{O}(\log(1/p_i))$. Pokuste se o srovnání se statickými optimálními vyhledávacími stromy z oddílu **??**.