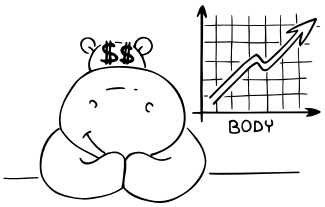
Korespondenční Seminář z Programování

31. ročník **KSP** **Květen 2019**

Milí řešitelé, milé řešitelky!



Poslední letošní série tohoto zpožděného ročníku se vám právě dostává do rukou. A i když léto již klepe na dveře, tak doufáme, že si najdete pár chvil na naše úlohy. Také podle výsledků celého ročníku budeme vybírat účastníky **podzimního soustředění** – pokud se tedy chcete účastnit, tak máte poslední šanci získat ještě nějaké body.

Připomínáme, že do celkového hodnocení se vám z každé série započte **5 nejlépe vyřešených úloh**. Každému řešiteli, který získá v tomto ročníku z každé série alespoň 5 bodů, pošleme KSP propisku, blok, placku a třeba i něco navíc.

Díky řešení KSP se také můžete vyhnout přijímacím zkouškám na MFF UK! Stačí, když získáte alespoň polovinu bodů z ročníku (tedy 150 bodů) a my vám vystavíme osvědčení, díky kterému vás přijmou na MFF bez zkoušek.

Termín série: pondělí **24. června v 8:00 ráno**

Odevzdávání: Přes web na adrese *https://ksp.mff.cuni.cz/submit/*

Odměna série: **Sladkou odměnu** si vysloužíí každý, kdo **získá z této série alespoň 42 bodů**.

Pátá série třicátého prvního ročníku KSP

Žili byli jednou jeden muž s ženou. Měli se rádi a hospodařili spolu v malém domečku už několik let. Ačkoliv byli velice chudí a jen tak tak si vydělávali na své vlastní živobytí, tuze si přáli mít děťátko.

Jednou z rána, když šel muž do lesa na dříví, uviděl malý pařízek, který vypadal docela jako malé děťátko – měl hlavičku, tělíčko, ručičky i nožičky. Potřeboval jen temínko malinko sekerou otesat, aby bylo pěkně kulaté a hladké. I přinesl muž pařízek domů a povídá ženě: „Tady máš, cos chtěla. Dítě Otesánka.“ Žena byla radostí bez sebe, zavinula děťátko do peřinky, hýčkala jej a k tomu vesele zpívala: „Hajej, dadej, hajej, Otesánku malej. Až se vzbudíš, hošíčku, uvařím ti polívčičku. Hajej, dadej, hajej.“

Najednou se začalo dítě v peřince hýbat a dalo se do křiku: „Mámo, já bych jed!“ Žena nevěděla, kam dřív skočit. Položila Otesánka do postele a jala se pro děťátko písmenkovou polívku vařit.

31-5-1 Písmenková polévka — 15 bodů

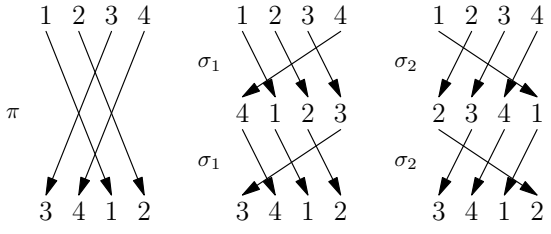
Písmenková polévka se musí při vaření správně zamíchat, aby dobře chutnala. Každé takové zamíchnutí nějak změní pořadí písmenek v polévce (matematicky můžeme říci, že každé zamíchnutí provede na písmenkách v polévce nějakou permutaci). Jak žena míchala, tu máchla vařečkou dvakrát za sebou úplně stejně. Vtom se na polévce z písmenek utvořilo nějaké zajímavé slovo. Žena si ho chvíli prohlížela a pak míchala dál. Po chvíli opět vařečkou zakroužila dvakrát stejně (ale jinak, než předtím). A co se nestalo! Na polévce stojí stejné slovo jako předtím! Tak ženu napadla otázka, kolik je takových způsobů zamíchání, aby dvě taková stejná zamíchání hned po sobě vytvořila právě toto slovo?

Protože písmenek v polévce je hodně, označíme si je raději čísly od 1 do n (tedy všechna písmenka jsou různá). Předpokládejme, že na začátku jsou písmenka v polévce uspořádána právě v pořadí od 1 do n.

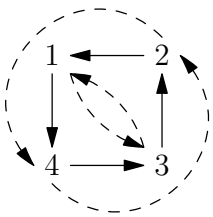
Poté dostaneme nějakou permutaci těchto čísel π (tedy nějaké pořadí čísel 1 až n, které určuje, kam se které číslo touto permutací posune) a nás zajímá, kolik existuje permutací σ, které provedené dvakrát po sobě dají to samé pořadí jako π.

Lehčí varianta (za 10 bodů): Najděte libovolnou takovou permutaci σ, nebo řekněte, že to nejde.

Příklady: Permutaci $\pi = 3, 4, 1, 2$ čtyř čísel můžeme rozložit třeba na permutaci $\sigma_1 = 4, 1, 2, 3$ provedenou dvakrát po sobě nebo na permutaci $\sigma_2 = 2, 3, 4, 1$ složenou dvakrát po sobě. Naopak permutaci tří čísel $\pi = 1, 3, 2$ na žádné dvě stejné permutace nerozložíme.



Když si permutaci σ_1 vyjádříme jako graf, tak může vypadat třeba takto. Všimněte si, že když navíc si do tohoto grafu nakreslíme čárkované šipky přeskakující dvě původní šipky, dostaneme tak permutaci π.

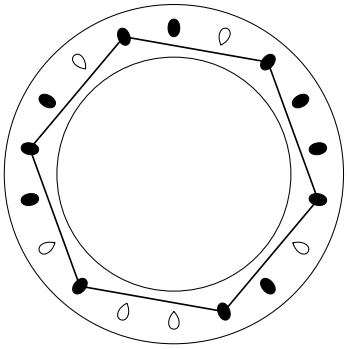


Když byla polévka hotová, Otesánek ji všechnu v mžiku zhltnul a dal se opět do křiku: „Mámo, já bych jed!“ „Počkej, Otesánku, hned ti něco donesu.“ I odběhla žena k sousedce odnaproti a po chvíli se vrátila s obrovským tvarohovým koláčem. Jen co jej doma na stůl položila, vymotal se Otesánek z peřiny a už seděl na lavici u stolu připraven na další chod.

31-5-2 Rozinky a mandle — 9 bodů

Otesánek dostal od sousedky nádherný velikánský kulatý koláč. Dokonce zdobený rozinkami a mandlemi, mňam! Po obvodu koláče bylo pravidelně rozmístěno N důlků a v každém z nich se nacházela jedna rozinka, nebo mandle. Otesánek si jako každé malé dítě rád hraje s jídlem, a tak ho zajímá, jestli by mohl z koláče vykousat pravidelný K-úhelník, v jehož vrcholech by byly pouze rozinky.

Otázka tedy zní: je možné si zvolit $K > 2$ a vybrat po obvodu koláče nějaké rozinky tak, aby tvořily pravidelný K-úhelník?



Pro koláč na obrázku lze z rozinek vykousat například vyznačený pravidelný šestiúhelník.

Otesánek snědl celý koláč raz dva a okamžitě se začal dožadovat dalšího jídla. „Dítě nešťastné, copak ještě nemáš dost?!“ Žena ani na odpověď nečekala a vydala se do vesnice sehnat pecen chleba. Protože však moc peněz neměla, musela chleba koupit na dluh. Když se pak pekař dozvěděl, že má žena doma hladového syna, smiloval se nad ní. „Co kdybych Vás naučil péct chleba? Když Vám dám trochu svého kvásku, budete si moct doma udělat vlastní.“

31-5-3 Kváskový chléb — 13 bodů

Chléb se peče z kvásku. Čím je kvásek starší a silnější, tím lépe těsto vykyne. Proto si pekař při každém pečení schová část kvásku na příště. Vždy, když pak peče chleba, smíchá kvásek z minulého a z předminulého pečení. Jeho chleby pak chutnají naprosto výtečně a jsou proslavené po celém okolí!

Pekař si dělá záznamy o tom, jaký kvásek na který chleba použil. Na svůj historicky první chléb použil žitný kvásek a do svých záznamů si zapsal $Ch_1 = Z$. Aby byl druhý chléb jiný, použil na něj kvásek pšeničný – tedy $Ch_2 = P$. Na každý další chléb pak umíchá kvásek z dvou předchozích. Jeho strukturu pak popíše tak, že napíše struktury dvou předchozích kvásků za sebe: $Ch_n = Ch_{n-2}\,Ch_{n-1}$ pro $n > 2$.

Kvásky takhle pekař skládá již dlouhou dobu a teď by chtěl ženě dát nějaký zajímavý kousek. Zajímalo by ho proto, jak vypadá nějaká část jeho n-tého kvásku (tedy jak vypadá nějaký podřetězec Ch_n). Varujeme, že u některých vstupů bude n tak velké, že se vám celý řetězec Ch_n nevejde do paměti.

Toto je praktická open-data úloha. V odevzdávacím systému si necháte vygenerovat vstupy a odevzdáte příslušné výstupy. Záleží jen na vás, jak výstupy vyrobíte.

Formát vstupu: Na vstupu dostanete na jednom řádku oddělená mezerami čísla n, a k a l udávající číslo kvásku, počáteční pozici podřetězce a délku podřetězce. Slibujeme, že délka podřetězce l bude maximálně 1 000 a že vstup bude vždy korektní (že $k + l$ se vždy vejde do Ch_n).

Formát výstupu: Na výstup vypište na jediný řádek část struktury n-tého kvásku, tedy podřetězec Ch_n, který začíná na k-té pozici a má délku l znaků (znaky indexujeme od nuly).

Ukázkový vstup:	*Ukázkový výstup:*
`5 1 3`	`PPZ`
Ukázkový vstup:	*Ukázkový výstup:*
`8 10 8`	`PZPPZPZP`
Ukázkový vstup:	*Ukázkový výstup:*
`50 150 20`	`ZPZPPZPPZPZPPZPZPPZP`

V prvním případě má celý kvásek podobu `ZPPZP`.

Jen co žena dorazila domů, položila pekařův chleba na stůl a šla zadělat svůj vlastní. Když se pak po chvíli vrátila zpátky do světnice, z pekařova chleba nezbyl ani drobeček. „Pánbůh s námi, Otesánku, snad jsi ten pecen nesnědl celý sám? Vždyť my už doma žádné další jídlo nemáme. . . “ „Snědl, mámo, a jestli už nic dalšího nemáš, budu muset sníst tebe!“ V tu ránu Otesánek otevřel pusu a než se žena nadála, byla v něm.

V poledne pak domů přišel i Otesánkův otec. Jakmile vkročil do dveří, Otesánek mu povídá: „Táto, já bych jed. Jedl jsem, snědl jsem: písmenkovou polívku, tvarohový koláč, pecen chleba, mámu a tebe taky sním!“ A chramst! Snědl i tátu.

Čím víc Otesánek jedl, tím větší měl hlad. A protože už v chaloupce snědl vše, co mohl, vypravil se do vsi po něčem se poohlédnout. Zanedlouho potkal na cestě děvečku s trakařem plným jetele. „Tys toho ale musel sníst, když máš tak velký břich!“ řekla děvečka s podivením. Na to Otesánek odpověděl: „Jedl jsem, snědl jsem: písmenkovou polívku, tvarohový koláč, pecen chleba, mámu, tátu a tebe taky sním!“ Přiskočil a děvečka i s trakařem zmizela v jeho břiše. Otesánek pak pokračoval vesele po cestě, až nakonec došel k místní vývařovně.

31-5-4 Otesánek ve výdejně — 12 bodů

Když Otesánek vkročil do výdejny, nestačil se divit. Tolik lidí pohromadě ještě neviděl. Pořád se tu něco děje. Lidi přicházejí a zase odcházejí… Na výdejnu se můžeme dívat jako na úsečku. Na jejím levém konci se nachází okýnko, kam se odkládá špinavé nádobí, a na jejím pravém konci je samotná výdejna jídel. Zpočátku se ve výdejně nachází pouze jedna paní u výdeje jídel a druhá paní u okýnka na špinavé nádobí. Postupně pak do výdejny přicházejí jednotliví vesničané. V jednom okamžiku nastane právě jeden ze dvou druhů událostí:

a) Do výdejny přijde vesničan. A jelikož v této vesnici žijí lidé povahově velice podobní matfyzákům (kteří se, jak je známo, ostatních lidí bojí), sedne si tak, aby byl co nejdále od všech ostatních lidí (včetně obou paní u okének). Speciálně, je-li tedy výdejna prázdná, sedne si doprostřed.
b) Vesničan číslo v dojí a odejde.

Vaším úkolem je odsimulovat průběh jednoho oběda ve výdejně tak, abyste pro každý výskyt události typu a) dokázali co nejrychleji vypsat pozici, kam si daný vesničan sedne. Pokud má vesničan několik stejně dobrých možností, kam se posadit, můžete si pro zjednodušení vybrat libovolnou.

Otesánek dostal nápad – schoval se za roh výdejny a každého, kdo z ní odcházel, snědl. Když ve výdejně skončila výdejní doba oběda a nikdo v ní už nebyl, pokračoval Otesánek na své cestě za jídlem.

Kolébal se zrovna kolem malé chaloupky na kraji vesnice, když tu najednou ucítil čerstvě upečenou pizzu. I vydal se za touto vůní. Dorazil k chaloupce, před kterou nějaká babička okopávala zelí.

31-5-5 Přísady na pizze — 11 bodů

„Babičko, dejte mi pizzu, já mám hlad," povídá Otesánek. Babička se chvíli zamyslí a pak říká: „Dobrá, ale jen když uhodneš, kolik je na ni potřeba různých druhů přísad. Prozradím ti jen, že spolu žádné dvě stejné přísady nesousedí a že druhů přísad není víc, než je nutné."

Pizza představuje graf (ne nutně rovinný) a na každý jeho vrchol chce babička umístit nějakou přísadu (kus sýra, zrnko kukuřice, olivu, nebo něco jiného). Dostanete tento graf (bez přísad) a vaším úkolem bude zjistit, kolik nejméně druhů přísad je nutné použít, aby spolu žádné dvě stejné přísady nesousedily.

To je docela těžká úloha, a proto máte k dispozici i kouzelnou krabičku, které můžete dát libovolný graf a ona vám řekne, jestli je nejmenší počet přísad v tomto grafu sudý, nebo lichý. Využijte ji k tomu, abyste zjistili počet přísad na pizze, jejíž graf vám babička dá.

Vaše řešení bude hodnoceno primárně podle počtu volání krabičky, sekundárně podle časové složitosti. Ta musí být polynomiální, nemůžete tedy například vyzkoušet všechna možná rozmístění přísad (a krabičku vůbec nepoužít).

Otesánek žádnou takovou kouzelnou krabičku neměl, a tak ať se snažil, jak se snažil, na správný počet nemohl přijít. Netušil totiž, že bez kouzelné krabičky se jedná o NP-těžký problém. Tu se Otesánek dopálil, zašklebil se na babičku a povídá: „Jedl jsem, snědl jsem: písmenkovou polívku, tvarohový koláč, pecen chleba, mámu, tátu, výdejnu plnou lidí a tebe, babičko, tebe taky sním!" A otevřel svou velikánskou pusu. Babička však byla rychlá a sekla Otesánka motyčkou do břicha. Otesánek spadl na zem a z břicha mu vyskočili lidé z výdejny, děvečka s trakařem, táta a máma, která si nesla pod paží pecen chleba. Všichni moc poděkovali babičce za záchranu a byli rádi, že všechno nakonec dobře dopadlo.

Zuzka Urbanová & Klárka Tauchmanová

31-5-6 Se-ri-ál — 0 bodů

Milí účastníci, v době, kdy jsem pro KSP slíbila napsat seriál, jsem měla nerealistická očekávání o svém volném čase a především o časové náročnosti opravování úloh. (Nestalo se mi to poprvé, což mi je obzvláště trapné.) Vzhledem k tomu vyjde pátý díl seriálu již jen jako výukový text bez bodovaných úloh. Vaše řešení třetího a čtvrtého dílu opravím v nejbližší možné době. Omlouvám se vám nejen za průtahy, ale také za přehnaný optimismus v průběhu roku, který mě vedl k pocitu, že »už to brzy udělám« a velmi nefunkční komunikaci. Díky za pochopení. Maria

Maria Matějka

Recepty z programátorské kuchařky: Vyhledávací stromy

V kuchařce první série jsme probrali základní způsoby ukládání dat v počítači, tzv. datové struktury, a také často používané programátorské techniky. Konkrétně jsme se naučili udržovat čísla nebo jiné objekty v poli, ve spojovém seznamu, v grafu nebo ve stromu. Ukázali jsme si rekurzi a její využití v backtrackingu (prostém zkoušení všech možností). Dále jsme již nakoukli pod pokličku dalším technikám: rozděl a panuj, dynamickému programování, hladovým algoritmům a pár dalším.

Nyní se podíváme podrobněji na binární vyhledávání, které bylo rovněž minule představeno, a pokusíme se ho vylepšit, abychom mohli průběžně měnit data, v nichž vyhledáváme. Zdá-li se vám to na jednu kuchařku málo, vězte, že problém není jednoduchý, ale zajímavý a řešení jsou navíc v praxi často používána.

Nejprve však zopakujme binární vyhledávání.

Binární vyhledávání

Stejně jako minule máme obrovské pole setříděných záznamů, třeba identifikačních čísel studentů nejmenované univerzity (záznamy však nemusí být čísla, stačí, když jsou navzájem porovnatelné). Naším úkolem je najít záznam z v poli s N záznamy $x_1 < x_2 < \ldots < x_N$.

Při použití binárního vyhledávání neboli půlení intervalu se podíváme na prostřední záznam x_m a porovnáme s ním naše z. Pokud $z < x_m$, víme, že se z nemůže vyskytovat „napravo“ od x_m, protože tam jsou všechny záznamy větší než x_m, a tím spíše než z. Analogicky pokud $z > x_m$, nemůže se z vyskytovat v první polovině pole. V obou případech nám zbude jedna polovina a v ní budeme pokračovat stejným způsobem. Tak budeme postupně půlit interval, ve kterém se z může nacházet, až buďto z najdeme, nebo vyloučíme všechny prvky, kde by mohlo být.

Tento algoritmus můžeme naprogramovat buďto rekurzivně, nebo pomocí cyklu, v němž si budeme udržovat interval $\langle l, r\rangle$, ve kterém se hledaný prvek ještě může nacházet. My si ukážeme přístup s cyklem:

```
def bin_najdi(z):
  levy = 0
  pravy = N
  while levy <= pravy:
    median = (levy+pravy)/2
    # hledaná hodnota je vlevo
    if z < x[median]:
      pravy = median - 1
    # je vpravo
    elif z > x[median]:
      levy = median+1
      # našli jsme přímo hodnotu
    else:
      return median
    # hledaná hodnota nebyla nikde
    return -1
```

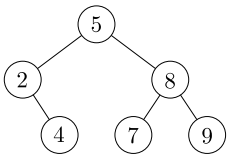
Samozřejmě bychom při vyhledávání záznamu mohli být ještě chytřejší. Víme-li třeba, že čísla jsou z rozsahu 1 až 1000 a dostaneme číslo 900, můžeme se napřed podívat do devíti desetin pole místo do poloviny. Obecně se tedy snažíme odhadovat, kde bude záznam v rámci pole podle jeho hodnoty. Tomuto přístupu se říká *interpolační vyhledávání* a v průměru je lepší než binární (průměrná časová složitost je $\mathcal{O}(\log\log N)$), byť v nejhorším případě je lineární.

Binární vyhledávání je velmi rychlé, pokud máme možnost si data předem setřídit. Jakmile ale potřebujeme za běhu programu přidávat a odebírat záznamy, se zlou se potážeme. Buďto budeme mít záznamy uložené v poli a pak nezbývá než při zatřiďování nového prvku ostatní „rozhrnout“, což může trvat až N kroků, anebo si je budeme udržovat v nějakém seznamu, do kterého dokážeme přidávat v konstantním čase, jenže pak pro změnu nebudeme při vyhledávání schopni najít tolik potřebnou polovinu.

Zkusme ale provést jednoduchý myšlenkový pokus:

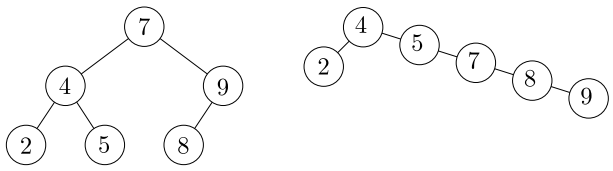
Vyhledávací stromy

Představme si, jakými všemi možnými cestami se může v našem poli binární vyhledávání ubírat. Na začátku porovnáváme s prostředním prvkem a podle výsledku se vydáme jednou ze dvou možných cest (nebo rovnou zjistíme, že se jedná o hledaný prvek, ale to není moc zajímavý případ). Na každé cestě nás zase čeká porovnání se středem příslušného intervalu a to nás opět pošle jednou ze dvou dalších cest atd. To si můžeme přehledně popsat pomocí stromu:



Jeden vrchol stromu prohlásíme za kořen a ten bude odpovídat celému poli (a jeho prostřednímu prvku). K němu budou připojené vrcholy obou polovin pole (opět obsahující příslušné prostřední prvky) a tak dále. Ovšem jakmile známe tento strom, můžeme náš půlící algoritmus provádět přímo podle stromu (ani k tomu nepotřebujeme vidět původní pole a umět v něm hledat poloviny): začneme v kořeni, porovnáme a podle výsledku se buďto přesuneme do levého, nebo pravého podstromu, a tak dále. Každý průběh algoritmu bude tedy odpovídat nějaké cestě z kořene stromu do hledaného vrcholu.

Teď si ale všimněte, že aby hledání hodnoty podle stromu fungovalo, strom vůbec nemusel vzniknout půlením intervalu – stačilo, aby v každém vrcholu platilo, že všechny hodnoty v levém podstromu jsou menší než tento vrchol a naopak hodnoty v pravém podstromu větší. Hledání v témže poli by také popisovaly následující stromy (např.):



Hledací algoritmus podle jiných stromů samozřejmě už nemusí mít pěknou logaritmickou složitost (kdybychom hledali podle „degenerovaného“ stromu z pravého obrázku, trvalo by to dokonce lineárně). Důležité ale je, že takovéto stromy se dají poměrně snadno modifikovat a že je při troše šikovnosti můžeme udržet dostatečně podobné ideálnímu půlení intervalu. Pak bude hloubka stromu stále $\mathcal{O}(\log N)$, tím pádem i časová složitost hledání, a jak za chvilku uvidíme, i mnohých dalších operací.

Definice

Zkusme si tedy pořádně nadefinovat to, co jsme právě vymysleli:

Binární vyhledávací strom (podomácku BVS) je buď prázdná množina, nebo *kořen* obsahující jednu hodnotu a mající dva *podstromy* (levý a pravý). Tyto podstromy jsou opět BVS, ovšem takové, že všechny hodnoty uložené v levém podstromu jsou menší než hodnota v kořeni, a ta je naopak menší než všechny hodnoty uložené v pravém podstromu.

Úmluva: Pokud x je kořen a L_x a R_x jeho levý a pravý podstrom, pak kořenům těchto podstromů (pokud nejsou prázdné) budeme říkat *levý a pravý syn* vrcholu x a naopak vrcholu x budeme říkat *otec* těchto synů. Pokud je některý z podstromů prázdný, pak vrchol x příslušného syna nemá. Vrcholu, který nemá žádné syny, budeme říkat *list* vyhledávacího stromu. Všimněte si, že pokud x má jen jediného syna, musíme stále rozlišovat, je-li to syn levý, nebo pravý, protože potřebujeme udržet správné uspořádání hodnot. Také si všimněte, že pokud známe syny každého vrcholu, můžeme již rekonstruovat všechny podstromy.

Každý BVS také můžeme popsat velmi jednoduchou strukturou v paměti:

```
struct vrchol {
    struct vrchol *levy, *pravy; // synové
    int x;                 // hodnota ve vrcholu
};
```

Pokud některý ze synů neexistuje, zapíšeme do příslušné položky hodnotu `NULL`.

Hledání

V řeči BVS můžeme přeformulovat náš vyhledávací algoritmus takto:

```
# Dostane kořen stromu a hodnotu. Vrátí vrchol,
# ve kterém se hodnota nachází, nebo None, když
# ve stromu není.
def strom_najdi(vrchol, x):
  while (v != None) and (vrchol.x != x):
    if x < vrchol.x:
      vrchol = vrchol.levy
    else:
      vrchol = vrchol.pravy
  return vrchol
```

Funkce `strom_najdi` bude pracovat v čase $\mathcal{O}(h)$, kde h je hloubka stromu, protože začíná v kořeni a v každém průchodu cyklem postoupí o jednu hladinu níže.

Vkládání

Co kdybychom chtěli do stromu vložit novou hodnotu (aniž bychom se teď starali o to, zda tím strom nemůže degenerovat)? Stačí zkusit hodnotu najít, a pokud tam ještě nebyla, určitě při hledání narazíme na odbočku, která je `NULL`. A přesně na toto místo připojíme nově vytvořený vrchol, aby byl správně uspořádán vzhledem k ostatním vrcholům (že tomu tak je, plyne z toho, že při hledání jsme postupně vyloučili všechna ostatní místa, kde nová hodnota být nemohla). Naprogramujeme opět snadno, tentokráte si ukážeme rekurzivní zacházení se stromy:

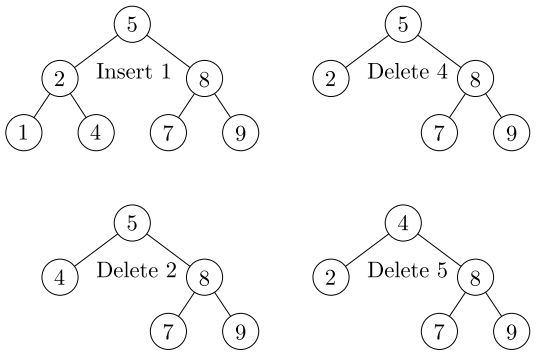
```
# Dostane kořen stromu a hodnotu ke vložení,
# vrátí nový kořen
def strom_vloz(vrchol, x):
  # prázdný strom
  if vrchol is None:
    # založíme nový kořen
    vrchol = Vrchol()
    vrchol.levy = None
    vrchol.pravy = None
    vrchol.x = x
  elif x < vrchol.x
    # vkládáme vlevo
    vrchol.levy = strom_vloz(vrchol.levy, x)
  elif x > vrchol.x:
    # vkládáme vpravo
    vrchol.pravy = strom_vloz(vrchol.pravy, x)
  return vrchol
```

Mazání

Mazání bude o kousíček pracnější, musíme totiž rozlišit tři případy: Pokud je mazaný vrchol list, stačí ho vyměnit za `NULL`. Pokud má právě jednoho syna, stačí náš vrchol v ze stromu odstranit a syna přepojit k otci v. Ale pokud má syny dva, najdeme největší hodnotu v levém podstromu (tu najdeme tak, že půjdeme jednou doleva a pak pořád doprava), umístíme ji do stromu namísto mazaného vrcholu a v levém podstromu ji pak smažeme (což už umíme, protože má 1 nebo 0 synů). Program následuje:

```
def strom_vymaz(vrchol, x):
  if vrchol is None:
    # prázdný strom
    return vrchol
  elif x < vrchol.x:
    # hledáme x
    vrchol.levy = strom_vymaz(vrchol.levy, x)
  elif x > vrchol.x:
    vrchol.pravy = strom_vymaz(vrchol.pravy, x)
  else:
    # našli jsme x, jaké má syny?
    if (vrchol.levy is None) and (vrchol.pravy is None)
      return None
    elif vrchol.levy is None:
      # má jen pravého syna
      return vrchol.pravy
    elif vrchol.pravy is None:
      # má jen levého syna
      return vrchol.levy
    else:
      # má oba syny
      w = vrchol.levy
      while not w.pravy is None:
        w = w.pravy
      vrchol.x = w.x    # prohazujeme
      # mažeme původní max(L)
      vrchol.levy=strom_vymaz(vrchol.levy, w.x)
  return v
```

Když do stromu z našeho prvního obrázku zkusíme přidávat nebo z něj odebírat prvky, dopadne to takto:



Jak vkládání, tak mazání opět budou trvat $\mathcal{O}(h)$, kde h je hloubka stromu. Ale pozor, jejich používáním může h nekontrolovatelně růst (v závislosti na počtu prvků ve stromu).

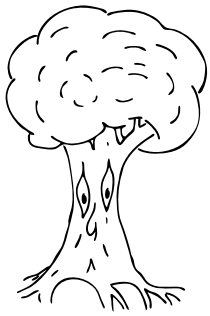
Cvičení

- Zkuste najít nějaký příklad, kdy h dosáhne až N – při postupném budování stromu operacemi vkládání i při mazání ze stromu hloubky $\mathcal{O}(\log N)$.

Procházení stromu

Pokud bychom chtěli všechny hodnoty ve stromu vypsat, stačí strom rekurzivně projít a sama definice uspořádání hodnot ve stromu nám zajistí, že hodnoty vypíšeme ve vzestupném pořadí: nejdříve levý podstrom, pak kořen a pak podstrom pravý. Časová složitost je, jak se snadno nahlédne, lineární, protože strávíme konstantní čas vypisováním každého prvku a prvků je právě N. Program bude opět přímočarý:

```
def strom_ukaz(vrchol):
  if vrchol is None:
    return
  print("({})({})({})".format(
    strom_ukaz(vrchol.levy),
    vrchol.x,
    strom_ukaz(vrchol.pravy)
  ))
```



Vyvážené stromy

S binárními stromy lze dělat všelijaká kouzla a prakticky všechny stromové algoritmy mají společné to, že jejich časová složitost je lineární v hloubce stromu. (Pravda, právě ten poslední je výjimka, leč všechny prvky rychleji než lineárně s N opravdu nevypíšeme.)

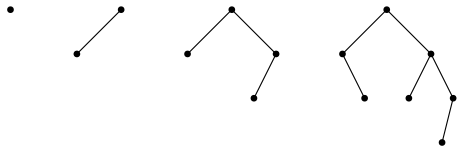
Jenže jak jsme viděli, neopatrným insertováním a deletováním prvků mohou snadno vznikat všelijaké degenerované stromy, které mají lineární hloubku. Abychom tomu zabránili, musíme stromy *vyvažovat.* To znamená definovat si nějaké šikovné omezení na tvar stromu, aby hloubka byla vždy $\mathcal{O}(\log N)$. Možností je mnoho, my uvedeme jen ty nejdůležitější:

Dokonale vyvážený budeme říkat takovému stromu, ve kterém pro každý vrchol platí, že počet vrcholů v jeho levém a pravém podstromu se liší nejvýše o jedničku. Takové stromy kopírují dělení na poloviny při binárním vyhledávání, a proto mají vždy logaritmickou hloubku. Jediné, čím se liší, je, že mohou zaokrouhlovat na obě strany, zatímco náš půlící algoritmus zaokrouhloval polovinu vždy dolů, takže levý podstrom nemohl být nikdy větší než pravý.

Z toho také plyne, že se snadnou modifikací půlícího algoritmu dá dokonale vyvážený BVS v lineárním čase vytvořit ze setříděného pole. Bohužel se ale při Insertu a Deletu nedá v logaritmickém čase strom znovu vyvážit.

AVL stromy

Zkusíme tedy vyvažovací podmínku trochu uvolnit a vyžadovat, aby se u každého vrcholu lišily o jedničku nikoliv velikosti podstromů, nýbrž pouze jejich hloubky. Takovým stromům se říká *AVL stromy* a mohou vypadat třeba takto:



Každý dokonale vyvážený strom je také AVL stromem, ale jak je vidět na předchozím obrázku, opačně to platit nemusí. To, že hloubka AVL stromů je také logaritmická, proto není úplně zřejmé a zaslouží si to trochu dokazování:

Věta: AVL strom o N vrcholech má hloubku $\mathcal{O}(\log N)$.

Důkaz: Označme A_d nejmenší možný počet vrcholů, jaký může být v AVL stromu hloubky d. Snadno zjistíme, že $A_1 = 1$, $A_2 = 2$, $A_3 = 4$ a $A_4 = 7$ (příslušné minimální stromy najdete na předchozím obrázku). Navíc platí, že $A_d = 1 + A_{d-1} + A_{d-2}$, protože každý minimální strom hloubky d musí mít kořen a 2 podstromy, které budou opět minimální, protože jinak bychom je mohli vyměnit za minimální a tím snížit počet vrcholů celého stromu. Navíc alespoň jeden z podstromů musí mít hloubku $d-1$ (protože jinak by hloubka celého stromu nebyla d) a druhý hloubku $d-2$ (podle definice AVL stromu může mít $d-1$ nebo $d-2$, ale s menší hloubkou bude mít evidentně méně vrcholů).

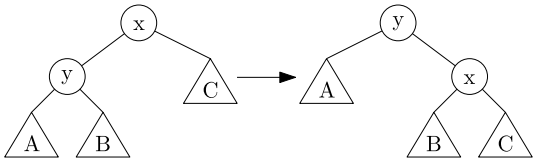
Spočítat, kolik přesně je A_d, není úplně snadné. Nám však postačí dokázat, že $A_d \geq 2^{d/2}$. To provedeme indukcí: Pro $d < 4$ to plyne z ručně spočítaných hodnot. Pro $d \geq 4$ je $A_d = 1 + A_{d-1} + A_{d-2} > 2^{(d-1)/2} + 2^{(d-2)/2} = 2^{d/2} \cdot (2^{-1/2} + 2^{-1}) > 2^{d/2}$ (součet čísel v závorce je ≈ 1.207).

Jakmile už víme, že A_d roste s d alespoň exponenciálně, tedy že $\exists c : A_d \geq c^d$, důkaz je u konce: Máme-li AVL strom T na N vrcholech, najdeme si nejmenší d takové, že $A_d \leq N$. Hloubka stromu T může být maximálně d, protože jinak by T musel mít alespoň A_{d+1} vrcholů, ale to je více než N. A jelikož A_d rostou exponenciálně, je $d \leq \log_c N$, čili $d = \mathcal{O}(\log N)$. *Q.E.D.*

AVL stromy tedy vypadají nadějně, jen stále nevíme, jak provádět Insert a Delete tak, aby strom zůstal vyvážený. Nemůžeme si totiž dovolit strukturu stromu měnit libovolně – stále musíme dodržovat správné uspořádání hodnot. K tomu se nám bude hodit zavést si nějakou množinu operací, o kterých dokážeme, že jsou korektní, a pak strukturu stromu měnit vždy jen pomocí těchto operací. Budou to rotace a dvojrotace.

Rotace

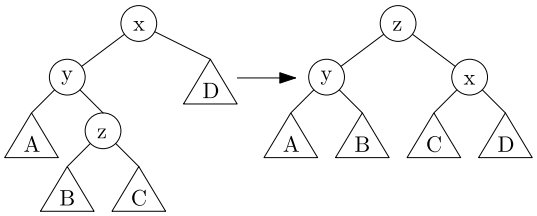
Rotací binárního stromu (respektive nějakého podstromu) nazveme jeho „překořenění“ za některého ze synů kořene. Místo formální definice ukažme raději obrázek (trojúhelník zastupuje podstrom, který může být v některých případech i prázdný):



Strom jsme překořenili za vrchol y a přepojili jednotlivé podstromy tak, aby byly vzhledem k x a y opět uspořádané správně (všimněte si, že je jen jediný způsob, jak to udělat). Jelikož se tím okolí vrcholu y „otočilo“ po směru hodinových ručiček, říká se takové operaci *rotace doprava.* Inverzní operaci (tj. překořenění za pravého syna kořene) se říká *rotace doleva* a na našem obrázku odpovídá přechodu zprava doleva.

Dvojrotace

Také si nakreslíme, jak to dopadne, když provedeme dvě rotace nad sebou lišící se směrem (tj. jednu levou a jednu pravou nebo opačně). Tomu se říká *dvojrotace* a jejím výsledkem je překořenění podstromu za vnuka kořene připojeného „cikcak“. Raději opět předvedeme na obrázku:



Znaménka

Při vyvažování se nám bude hodit pamatovat si u každého vrcholu, v jakém vztahu jsou hloubky jeho podstromů. Tomu budeme říkat *znaménko* vrcholu a bude buďto 0, jsou-li oba stejně hluboké, − pro levý podstrom hlubší, nebo + pro pravý hlubší. V textu budeme znaménka, respektive vrcholy se znaménky značit $\odot$, $\ominus$ a $\oplus$.

Pokud celý strom zrcadlově obrátíme (prohodíme levou a pravou stranu), znaménka se změní na opačná ($\oplus$ a $\ominus$ se prohodí, $\odot$ zůstane). Toho budeme často využívat a ze dvou zrcadlově symetrických situací popíšeme jenom jednu s tím, že druhá se v algoritmu zpracuje symetricky.

Často také budeme potřebovat nalézt otce nějakého vrcholu. To můžeme zařídit buďto tak, že si do záznamů popisujících vrcholy stromu přidáme ještě ukazatele na otce a budeme ho ve všech operacích poctivě aktualizovat, anebo využijeme toho, že jsme do daného vrcholu museli někudy přijít z kořene, a celou cestu z kořene si zapamatujeme v nějakém zásobníku a postupně se budeme vracet.

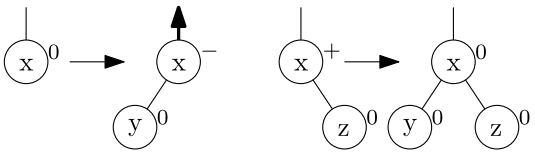
Tím jsme si připravili všechny potřebné ingredience, tož s chutí do toho.

Vyvažování po Insertu

Když provedeme Insert tak, jak jsme ho popisovali u obecných vyhledávacích stromů, přibyde nám ve stromu list. Pokud se tím AVL vyváženost neporušila, stačí pouze opravit znaménka na cestě z nového listu do kořene (všude jinde zůstala zachována). Pakliže porušila, musíme s tím něco provést, konkrétně ji šikovně zvolenými rotacemi opravit.

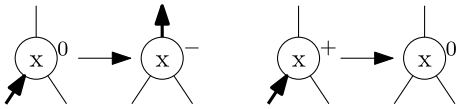
Popíšeme algoritmus, který bude postupovat od listu ke kořeni a vše potřebné zařídí.

Nejprve přidání listu samotné:

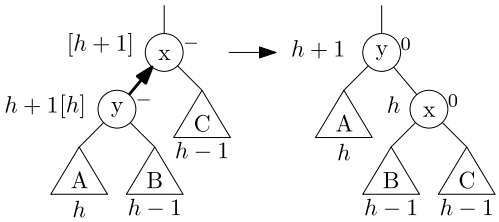


Pokud jsme přidali list (bez újmy na obecnosti levý, jinak vyřešíme zrcadlově) vrcholu se znaménkem $\odot$, změníme znaménko na $\ominus$ a pošleme o patro výš informaci o tom, že hloubka podstromu se zvýšila (to budeme značit šipkou). Přidali-li jsme list k $\oplus$, změní se na $\odot$ a hloubka podstromu se nemění, takže můžeme skončit.

Nyní rozebereme případy, které mohou nastat na vyšších hladinách, když nám z nějakého podstromu přijde šipka. Opět budeme předpokládat, že přišla zleva; pokud zprava, vyřešíme zrcadlově. Pokud přišla do $\odot$ nebo $\oplus$, ošetříme to stejně jako při přidání listu:

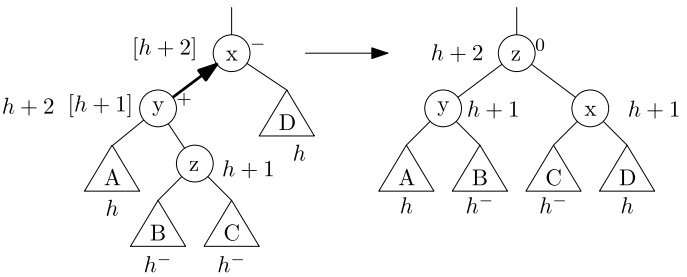


Pokud ale vrchol x má znaménko $\ominus$, nastanou potíže: levý podstrom má teď hloubku o 2 vyšší než pravý, takže musíme rotovat. Proto se podíváme o patro níž, jaké je znaménko vrcholu y pod šipkou, abychom věděli, jakou rotaci provést. Jedna možnost je tato (y je $\ominus$):



Tehdy provedeme jednoduchou rotaci vpravo. Jak to dopadne s hloubkami, jsme přikreslili do obrázku – pokud si hloubku podstromu A označíme jako h, B musí mít hloubku $h-1$, protože y je $\ominus$, atd. Jen nesmíme zapomenout, že v x jsme ještě $\ominus$ nepřepočítali (vede tam přeci šipka), takže ve skutečnosti je jeho levý podstrom o 2 hladiny hlubší než pravý (původní hloubky jsme na obrázku naznačili [v závorkách]). Po zrotování vyjdou u x i y znaménka $\odot$ a celková hloubka se nezmění, takže jsme hotovi.

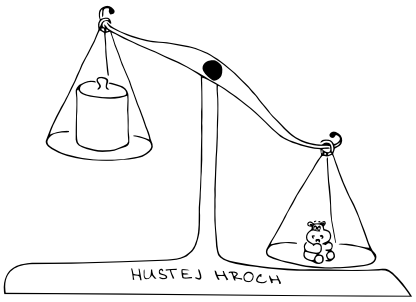
Další možnost je y jako $\oplus$:



Tehdy se podíváme ještě o hladinu níž a provedeme dvojrotaci. (Nemůže se nám stát, že by z neexistovalo, protože jinak by v y nebylo $\oplus$.) Hloubky opět najdete na obrázku. Jelikož z může mít libovolné znaménko, jsou hloubky podstromů B a C buďto h, nebo $h-1$, což značíme h^-. Podle toho pak vyjdou znaménka vrcholů x a y po rotaci.

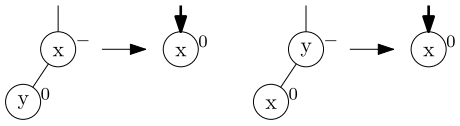
Každopádně vrchol z vždy obdrží $\odot$ a celková hloubka se nemění, takže končíme.

Poslední možnost je, že by y byl $\odot$, ale tu vyřešíme velmi snadno: všimneme si, že nemůže nastat. Kdykoliv totiž posíláme šipku nahoru, není pod ní $\odot$. (Kontrolní otázka: jak to, že $\oplus$ může nastat?)

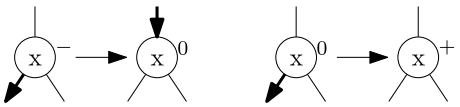


Vyvažování po Deletu

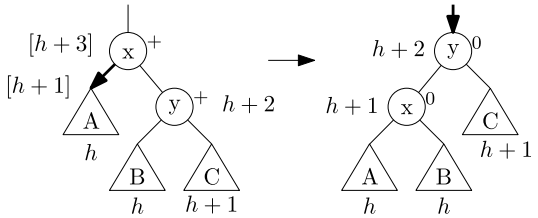
Vyvažování po Deletu je trochu obtížnější, ale také se dá popsat pár obrázky. Nejdříve opět rozebereme základní situace: odebíráme list (bez újmy na obecnosti (BÚNO) levý) nebo vnitřní vrchol s jediným synem (tehdy ale musí být jeho jediný syn listem, jinak by to nebyl AVL strom):



Šipkou dolů značíme, že o patro výš posíláme informaci o tom, že se hloubka podstromu snížila o 1. Pokud šipku dostane vrchol typu $\ominus$ nebo $\odot$, vyřešíme to snadno:

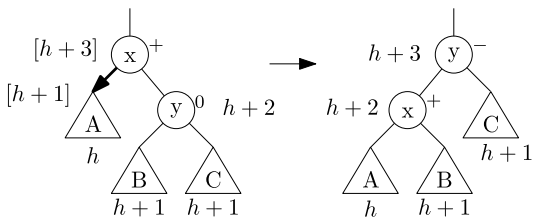


Problematické jsou tentokráte ty případy, kdy šipku dostane $\oplus$. Tehdy se musíme podívat na znaménko *opačného* syna a podle toho rotovat. První možnost je, že opačný syn má $\oplus$:



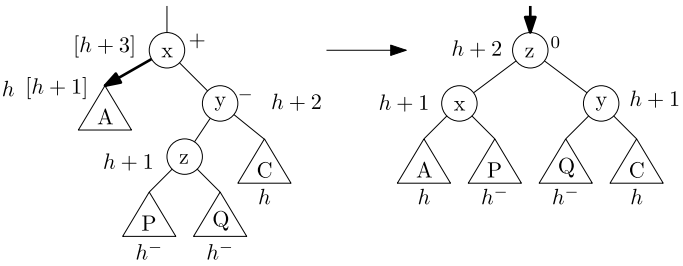
Tehdy provedeme rotaci vlevo, x i y získají nuly, ale celková hloubka stromu se sníží o hladinu, takže nezbývá, než poslat šipku o patro výš.

Pokud by y byl $\odot$:



Opět rotace vlevo, ale tentokráte se zastavíme, protože celková hloubka se nezměnila.

Poslední, nejkomplikovanější možnost je, že by y byl $\ominus$:



V tomto případě provedeme dvojrotaci (z určitě existuje, jelikož y je typu $\ominus$), vrcholy x a y obdrží znaménka v závislosti na původním znaménku vrcholu z a celý strom se snížil, takže pokračujeme o patro výš.

Happy end

Jak při Insertu, tak při Deletu se nám podařilo strom upravit tak, aby byl opět AVL stromem, a trvalo nám to lineárně s hloubkou stromu (konáme konstantní práci na každé hladině), čili stejně jako trvá Insert a Delete samotný. Jenže o AVL stromech jsme již dokázali, že mají hloubku vždy logaritmickou, takže jak hledání, tak Insert a Delete zvládneme v logaritmickém čase (vzhledem k aktuálnímu počtu prvků ve stromu).

Další typy stromů

AVL stromy samozřejmě nejsou jediný způsob, jak zavést stromovou datovou strukturu s logaritmicky rychlými operacemi. Jaké jsou další?

2-3-stromy nemají v jednom vrcholu uloženu jednu hodnotu, nýbrž jednu nebo dvě (a synové jsou pak 2 nebo 3, odtud název). Přidáme navíc pravidlo, že všechny listy jsou na téže hladině. Hloubka vyjde logaritmická, vyvažování řešíme pomocí spojování a rozdělování vrcholů.

Červeno-černé stromy si místo znamének vrcholy barví. Každý je buďto červený nebo černý a platí, že nikdy nejsou dva červené vrcholy pod sebou a že na každé cestě z kořene do listu je stejný počet černých vrcholů. Hloubka je pak znovu logaritmická.

Po Insertu a Deletu barvy opravujeme rotováním a přebarvováním na cestě do kořene, jen je potřeba rozebrat podstatně více případů než u AVL stromů. (Za to jsme ale odměněni tím, že nikdy neděláme více než 2 rotace.) Počet případů k rozebrání lze omezit zpřísněním podmínek na umístění červených vrcholů – dvěma různým takovým zpřísněním se říká *AA-stromy* a *left-leaning červeno-černé stromy*.

Interpretujeme-li červené vrcholy jako rozšíření otcovského vrcholu o další hodnoty, pochopíme, že jsou červeno-černé stromy jen jiným způsobem záznamu 2-4-stromů. Proč se takový kryptický překlad dělá? S třemi potomky vrcholu a dvěma hodnotami se pracuje nešikovně.

V případě *splay stromů* nezavádíme žádnou vyvažovací podmínku, nýbrž definujeme, že kdykoliv pracujeme s nějakým vrcholem, vždy si jej vyrotujeme do kořene, a pokud to jde, preferujeme dvojrotace. Takové operaci se říká Splay a dají se pomocí ní definovat operace ostatní: Find hodnotu najde a poté na ni zavolá Splay. Insert si nechá vysplayovat předchůdce nové hodnoty a vloží nový vrchol mezi předchůdce a jeho pravého syna. Delete vysplayuje mazaný prvek, pak uvnitř pravého podstromu vysplayuje minimum, takže bude mít jen jednoho syna a můžeme jím tedy nahradit mazaný prvek v kořeni.

Jednotlivé operace samozřejmě mohou trvat až lineárně dlouho, ale dá se o nich dokázat, že jejich *amortizovaná* složitost je vždy $\mathcal{O}(\log N)$. Tím chceme říci, že provést t po sobě jdoucích operací začínajících prázdným stromem trvá $\mathcal{O}(t \cdot \log N)$ (některé operace mohou být pomalejší, ale to je vykoupeno větší rychlostí jiných).

To u většiny použití stačí – datovou strukturu obvykle používáme uvnitř nějakého algoritmu a zajímá nás, jak dlouho běží všechny operace dohromady – a navíc je Splay stromy daleko snazší naprogramovat než nějaké vyvažované stromy. Mimo to mají Splay stromy i jiné krásné vlastnosti: přizpůsobují svůj tvar četnostem hledání, takže často hledané prvky jsou pak blíž ke kořeni, snadno se dají rozdělovat a spojovat, atd.

Treapy jsou randomizovaně vyvažované stromy: něco mezi stromem (tree) a haldou (heap). Každému prvku přiřadíme *váhu,* což je náhodné číslo z intervalu $\langle 0, 1\rangle$. Strom pak udržujeme uspořádaný stromově podle hodnot a haldově podle vah (všimněte si, že tím je jeho tvar určen jednoznačně, pokud tedy jsou všechny váhy navzájem různé, což skoro jistě jsou). Insert a Delete opravují haldové uspořádání velmi jednoduše pomocí rotací. Časová složitost v průměrném případě je $\mathcal{O}(\log N)$.

BB-α stromy nabízí zobecnění dokonalé vyváženosti jiným směrem: zvolíme si vhodné číslo α a vyžadujeme, aby se velikost podstromů každého vrcholu lišila maximálně α-krát (prázdné podstromy nějak ošetříme, abychom nedělili nulou; dokonalá vyváženost odpovídá $\alpha = 1$ (až na zaokrouhlování)). V každém vrcholu si budeme pamatovat, kolik vrcholů obsahuje podstrom, jehož je kořenem, a po Insertu a Deletu přepočítáme tyto hodnoty na cestě zpět do kořene a zkontrolujeme, jestli je strom ještě stále α-vyvážený.

Pokud ne, najdeme nejvyšší místo, ve kterém se velikosti podstromů příliš liší, a vše od tohoto místa dolů znovu vytvoříme algoritmem na výrobu dokonale vyvážených stromů. Ten, pravda, běží v lineárním čase, ale čím větší podstrom přebudováváme, tím to děláme méně často, takže vyjde opět amortizovaně $\mathcal{O}(\log N)$ na operaci.

Cvičení

- Jak konstruovat dokonale vyvážené stromy?
- Jak pomocí toho naprogramovat BB-α stromy?
- Najděte algoritmus, který k prvku v obecném vyhledávacím stromu najde jeho následníka, což je prvek s nejbližší vyšší hodnotou (zde předpokládejte, že ke každému prvku máte uložený ukazatel na jeho otce).
- Jak vypsat celý strom tak, že začnete v minimu a budete postupně hledat následníky? (I když nalezení následníka může trvat až $\mathcal{O}(h)$, všimněte si, že projití celého stromu přes následníky bude lineární.)
- Jak do vrcholů stromu ukládat různé pomocné informace, jako třeba počet vrcholů v podstromu každého vrcholu, a jak tyto informace při operacích se stromem (při Insertu, Deletu, rotaci) udržovat?
- Ukažte, že lze libovolný interval $\langle a, b\rangle$ rozložit na logaritmicky mnoho intervalů odpovídajících podstromům vyváženého stromu.
- Ukažte, že zkombinováním předchozích dvou cvičení lze odpovídat i na otázky typu „kolik si strom pamatuje hodnot ze zadaného intervalu“ v logaritmickém čase ...

Poznámky

- Představte si, že budujete binární vyhledávací strom pomocí vkládání prvků v náhodném pořadí. Obecně nemusí být vyvážený, ale v průměru v něm bude možné vyhledávat v čase $\mathcal{O}(\log N)$. Žádný div: Stromy, které nám vzniknou, odpovídají přesně možným průběhům QuickSortu, který má průměrnou časovou složitost $\mathcal{O}(N \log N)$.
- Pokud bychom připustili, že se mohou vyskytnout dva stejné záznamy, budou stromy stále fungovat, jen si musíme dát o něco větší pozor na to, co všechno při operacích se stromem může nastat.
- Jakpak přišly AVL stromy ke svému jménu? Podle Adeľsona-Veľského a Landise, kteří je vynalezli.
- Rekurenci $A_d = 1 + A_{d-1} + A_{d-2}$, $A_1 = 1$, $A_2 = 2$ pro velikosti minimálních AVL stromů je samozřejmě možné vyřešit i přesně. Žádné překvapení se nekoná, objeví se totiž stará známá Fibonacciho čísla: $A_n = F_{n+2} - 1$.

Martin „Medvěd“ Mareš & Tomáš Valla

matfyz

KSP pro vás připravují studenti Matematicko-fyzikální fakulty Univerzity Karlovy.

Webové stránky: *https://ksp.mff.cuni.cz/*
E-mail: *ksp@mff.cuni.cz*
Diskusní fórum: *https://ksp.mff.cuni.cz/forum/*

Chcete-li s námi komunikovat bezpečně, můžete si ověřit náš HTTPS certifikát – jeho SHA1 fingerprint je: `E9:DB:EE:C6:62:BC:14:DE:09:E4:E8:97:DC:36:0E:87:B3:50:B0:01`.