Cvičení DS I

Jak napsat DÚ z DS I

Krátká předmluva

Tento text je určen k tomu, aby vám napomohl s vypracováním domácích úkolů na předmět Datové struktury I. Naleznete v něm několik rad, kterými si myslíme, že je dobré se řídit, stejně jako několik varování na přístupy, které vám podle našeho názoru můžou práci ztížit. Tento text byste určitě neměli brát doslovně a rovněž určitě nepopisuje jedinou možnost, jak úkoly splnit na dostatek bodů ani jedinou správnou metodu, jak se s podobnými úkoly vypořádávat, pokud je potkáte jinde.

Letos budou také dva nové předměty implementační seminář a teoretický seminář k datovým strukturám. Viz stránky Jiřího Finka a Martina Mareše.

Proč vlastně takovéto domácí úkoly?

Dovolte nám ještě napsat pár slov, proč si myslíme, že tyhle domácí úkoly dávají smysl a co byste si z nich měli odnést:

1. Vyzkoušíte si naprogramovat něco nejspíše více abstraktního, než programy, které jste psali na bakalářských předmětech.
2. Experimentálně si ověříte věty, které se na přednášce dokazují a získáte představu o tom, jak se probírané algoritmy chovají na reálném hardwaru.
3. Získáte představu o tom, jak se popisují výsledky měření a zjistíte, na co si u popisu měření dávat pozor.
4. Možná se i trochu procvičíte v plnění úkolů s deadline.

Zároveň bychom chtěli zmínit, že i v praxi nebo ve vědě může člověk narazit na potřebu měření (nejen chování datových struktur). Například kvůli profilování času, který program tráví v jednotlivých částech výpočtu. Což umožňuje vybrat část programu, která by se měla zrychlit, aby čas strávený optimalizací nesl i kýžené ovoce. Nebo kvůli navrhování či experimentálnímu ověřování hypotéz, jež poté mohou být dokazovány.

Začínáme s DÚ – než začneme programovat

Přečtěte si pravidla

Víte, co smíte a nesmíte používat?

- Je ok použít LinkedList v Javě? (ne)
- Je ok použít std::max? (spíš ano)
- Je ok použít printf? (určitě ano)
- Je ok použít LinkedList v Javě pro čtení vstupu? (ano)

- Nevíte? (zeptejte se cvičících / přednášejících)
- Víte, jaký je bodovací systém?
- Smíte odevzdat cizí kód? (ne)
- Přečtěte si i další pravidla!

Koukněte se na vzorový úkol 0 a projděte si jeho řešení

Cvičící pro vás připravili vzorový úkol. Něco podobného se od vás čeká v dalších úkolech.

Začněte včas

Nejprve bychom rádi upozornili, abyste na domácích úkolech začali pracovat včas. Za předčasné odevzdání žádná penalizace není, avšak pokud nestihnete odevzdat před deadline, bude vás to stát cenné body. Nejlepší je začít na domácím úkolu pracovat, hned poté, co dostanete zadání. Silně nedoporučujeme nechávat vypracování na poslední víkend - nenalhávejte si, že vás o víkendu nebude od pracování na domácím úkolu nic vyrušovat a že to napíšete na první dobrou. Také naměřit data zabere nějaký ten čas a často se chybky v programu ukryté, začnou projevovat až u velkých dat. Nejasnosti zadání lze probrat na cvičení, ale ne o víkendu před termínem odevzdání!

Nejprve čtěte

Určitě nezačínejte ihned programovat. Místo toho se ujistěte, že jste si zadání přečetli pořádně a víte přesně, co se po vás chce. Nezapomeňte, že velká část bodů závisí na tom, jestli zadání splníte a i nesplnění malého detailu v zadání se může hodně projevit v bodech za tuto část. Speciálně proto, že můžete naprogramovat něco jiného, vaše naměřené hodnoty nebudou vykazovat požadované vlastnosti a vy nebudete mít co popisovat.

Nastudujte si přednášku

Nezačínejte programovat dokud si nejste jistí, že chápete jak má struktura fungovat. Přepisovat kód později stojí více sil než rovnou začít psát správnou věc.

Udělejte si návrh na papír

Zkuste si strukturu odsimulovat na malém vstupu. Rozmyslete si:

1. V jakém formátu chcete držet data?
2. Jaké metody se vám budou hodit?

Nejlépe si i vyzkoušejte práci s takto navrženými metodami. Kde mají jednotlivé přístupy své výhody a kde jsou naopak jejich slabiny? Pokud zjistíte, že jste něco navrhli nešikovně, pak je v tomto kroku jednoduché to napravit, rozhodně snáz než po několika dnech programování.

Ptejte se

Pokud zjistíte, že něčemu nerozumíte (ať už zadání, nebo látce z přednášky), nebojte se zeptat přednášejícího nebo cvičícího.

Programujeme

Napište si funkce pro validaci korektnosti a vykreslování

I pokud už máte představu, jak úkol naprogramovat, je vhodnější začít psaním testů. Napište si metodu, která ověří, že po provedení operace dostanete strukturu v konzistentním stavu - tj. jsou splněny všechny invarianty. Může se hodit i metoda, která sice testuje jen část invariantů, ale bude je schopna testovat i na větších vstupech.

Řiďte se pravidlem: "Když mám v programu chybu, tak ať na ni program shoří co nejdříve." Do těla funkce si přidejte testování splnění podmínek při vstupu do funkce (má-li mít uzel v dané chvíli syna, měl by program spadnout, pokud místo ukazatele na syna drží uzel null pointer). Nejen v C/C++ můžete na testování použít funkce assert.

Pro otestování správné funkce se hodí si strukturu vizualizovat. Pokud se vám nebude chtít programovat vlastní zobrazování struktury (což plně chápeme), můžete použít například program DOT. DOT je kompatibilní s libovolným programovacím jazykem. Jedná se o program, kterému dáte data v požadovaném formátu a on vám vykreslí jednotlivé instance včetně pointerů vedoucích mezi nimi. Více o DOT si můžete přečíst třeba na `https://en.wikipedia.org/wiki/DOT_(graph_description_language)`.

Pro testování korektnosti programu se vám dále mohou hodit:

1. váš oblíbený debugger (většinou součástí IDE). Pokud nepoužíváte IDE, možná budete chtít použít gdb (případně jeho nadstavbu DDD) – program pro debugování programů v C/C++ na linuxu, viz například `https://www.youtube.com/watch?v=PorfLSr3DDI&t=452s`. Nebo nějakou přívětivější klikací verzi jako je v QtCreator nebo `https://gdbgui.com/`.
2. valgrind – program, který kontroluje využívání paměti a upozorní vás na memory-leaky. Ale umí i spoustu dalších věcí. Podle stránek by měl být kompatibilní se spoustou běžně používaných programovacích jazyků. Viz `http://valgrind.org/` nebo `https://www.youtube.com/watch?v=bb1bTJtgXrI`.

Neztěžujte si práci, nepřehánějte optimalizace

Zkuste se držet zadání. Někdy méně je více. Za to, že si zadání ztížíte a naprogramujete více než se po vás chce vám sice nebudou body odebrány, ale nebudou vám ani přidány. Mnohdy se stalo, že rozšíření programu, studenti naprogramovali na úkor lepšího zpracování zadaného úkolu.

To samé se týká přehnaných optimalizací programu. V případě úkolu by datová struktura měla mít správnou asymptotickou složitost a neměla by být příliš neefektivní, na druhou stranu však není potřeba mít vlastní speciální alokátor paměti, který způsobí, že váš program bude 1.05-krát rychlejší. Mimochodem často nám vycházelo, že čím byl program kratší, tím byl čitelnější, rychlejší a obsahoval méně chyb. Při větší optimalizaci kódu není těžké zanést do programu více chyb – tedy programování trvá delší čas, a přitom zisk není vždy významný.

Nekopírujte svůj vlastní kód (a cizí už vůbec ne!)

Ač víme, že jste tuto radu již slyšeli mnohokrát, stejně si ji neodpustíme. Neboť ruku na srdce, jak moc jí řídíte. Přitom zrovna kopírování je jedním z častých zdrojů chyb. Ať už proto, že při okopírování nepřepíšete všechny proměnné do nového kontextu správně, nebo proto, že udržovat stejný kód na mnoha místech je mnohem těžší, než udržovat jednu správnou funkci. S nekopírováním kódu vám v C/C++ mohou pomoci například ifdefy, templaty nebo dědičnost (všimněte si, že správná a naivní varianta se často liší pouze v několika málo řádcích kódu).

Dereferencujte a nazývejte věci pravými jmény

Další již "ohranou" radou je, nepsat tak zvaný "špagetový kód". Ač je mnohdy jednodušší napsat jednu funkci, která udělá spoustu věcí, typicky tento přístup nevede k úplně hezkému kódu. Každý člověk je schopen udržet pouze určité množství informací, které často úzce koresponduje s množstvím kódu, které se vleze na jednu obrazovku. Tedy pokud se velikost vámi psané funkce blíží či snad dokonce překoná počet řádků vaší obrazovky, měli byste se zamyslet jestli ji nemůžete rozumně dělit na více částí.

Rovněž uvažujte nad tím, jestli vaše pojmenování funkcí či proměnných skutečně odpovídá jejich významu. Funkce "uklid' schody" určitě neměla schody vytřít a poté na ně položit slupku od banánu, na které se následně polámete.

V reálném životě, vás nikdo spíš nepochválí pokud sice uvaříte jídlo, ale přitom převrátíte kuchyni naruby a necháte ji neuklizenou. A vaše příprava dalšího jídla bude nejspíš v součtu náročnější, než kdybyste kuchyni uklidili hned a poté přišli znovu do připravené kuchyně. Obdobně to funguje i při programování. Při psaní programu sice může být rychlejší si funkce detailně nerozmyslet a poté napsat funkce, které nebudou dělat práci, kterou by odvést měly, nebo naopak udělají i práci, kterou by z jejich názvu nevyčetl ani věštec. Avšak poté při hledání chyb a ladění programu strávíte mnohem více času, než který jste si předtím ušetřili. Navíc věřte, že váš cvičící po vás program bude skutečně číst a čím více jeho časem budete plýtvat nevhodným pojmenováváním, tím méně bodů dostanete za kvalitu kódu.

Přemýšlejte, k čemu byly dané konstrukty vymyšleny

Spousta z vás se naučilo mnoho užitečných věcí ve svém bakalářském studiu a naučili jste se je používat automaticky. Občas je, ale dobré mít i na paměti, k čemu byly tyto konstrukty vytvořeny. Tedy například pokud používáte gettery a settery, vězte, že byly vymyšleny pro to, abyste si mohli lépe hlídat proměnné ve vaši struktuře. Není tedy nejrozumnější, aby datová struktura měla například public setter na počet jejích prvků. Obdobně pokud struktura má pointer, jehož hodnota by nikdy neměla být nullptr, pak by setter této proměnné asi měl ohlídat, že se do ní nullptr nikdy nepřiřazuje. Podobných příkladů existuje celá řada.

Rekurze vs nerekurze

Spousta lidí preferuje rekurzivní funkce. A má to své opodstatnění a my se mu nebráníme. Jako vše v životě to má svá ale.

Stromy. Pokud víte, že binární strom je vyvažovaný, tedy má logaritmickou hloubku v počtu prvků, bývá celkem bezproblémové použít rekurzi (rozmyslete si velikost zásobníku a velikost alokované paměti). Na druhou stranu pokud naše stromy můžou degenerovat až skoro ve spojový seznam, pak rekurzivní nemusí být vždy dobré řešení (to se vám v úkolech nejspíš stane – znovu porovnejte velikost zásobníku a alokovatelné paměti). Navíc se stromy se často dobře pracuje i nerekurzivně.

Rekurzivní algoritmy a obtížné datové struktury. Budete i programovat algoritmy, které mají hloubku rekurze omezenou logaritmem velikosti datové struktury. Ani zde není příliš těžké napsat nerekurzivní řešení, ale není to zas tak nezbytné. Pokud je datová struktura velmi obtížně implementovatelná, může rekurzivní implementace práci usnadnit (zvažte nakolik to je nebo není tento případ). Naopak nejspíš budete i implementovat datové struktury, kde by rekurzivní přístup znamenal příliš mnoho rekurzivních volání, tam je velice žádoucí napsat nerekurzivní řešení.

Testujte

Když už si myslíte, že máte naprogramovanou strukturu, která je správně. Otestujte ji. Dejte ji malé vstupy a zjistěte jak se chová. Ideální je testování automatizovat (vytvořit si sadu testů, které můžete opakovaně pouštět dokud nebudou procházet). Nezapomeňte vyzkoušet různé okrajové případy. Porovnejte její výstupy s fungující datovou strukturou (ideálně z vašeho jazyka), která může být i pomalejší, ale u které si jste jistí, že funguje.

Krasopis

Pište kód tak, jako by ho po vás měl někdo číst. My to totiž uděláme a dáme vám za něj body. Nezáleží nám na tom, jestli používáte ten či onen coding guide nebo mezery vs tabulátory. Vadí nám, když to mícháte. Vadí nám, když kód nejde číst. Vadí nám, když mezi dvěma funkcemi máte sto padesát prázdných řádků, atd.

Obecně platí:

- Je lepší napsat kód a odevzdat řešení než se trápit s nerekurzivním řešením a neodevzdat nic.
- Možná bude lepší nebo jednodušší začít psát rekurzivní řešení – viz bod výše.
- Pozor na to, že někdy bude nerekurzivní řešení naopak jednodušší!
- Preferujte krátký a čitelný kód, pokud to jde i efektivní (kupříkladu nerekurzivní). Tedy pokud se rekurze umíte zbavit jen přidáním padesáti řádek programu, zvažte, jestli to má cenu.
- Preferujte programy, které jsou krátké, čitelné, a evidentně správné.
- Neexistuje pravidlo, které by nemělo výjimku.

Měření

Automatizace

Ve vzorovém úkolu jsme připravili i automatizovaný způsob, jak měřit data (zvolili jsme makefile, ale můžete použít python script, ant nebo cokoliv jiného). Nejen, že potěšíte cvičící, kteří nemusí přemýšlet, jakou verzi jazyka, kompilátoru a prostředí používáte (a tím pádem stráví méně času koukáním do vašeho kódu). Ale další a možná největší výhodou je, že pokud něco upravíte v kódu, je daleko jednodušší pustit měření znova. Možná vás to překvapí, ale většina z vás nenaměří hned napoprvé. Další výhodou je, že si nebudete muset do dalšího úkolu pamatovat jak se vyrábějí grafy atd. Automatické vygenerování výstupů a možná i grafů vám tedy může ušetřit značné množství času (ale je to váš čas, takže klidně můžete grafy dělat v tabulkovém editoru). Je totiž rozdíl, pokud po změně kódu zadáte jeden příkaz a za čtyři minuty máte hotové grafy nebo po změně kódu ručně kompilujete, pouštíte pro každý parametr zvlášť a nakonec ještě děláte grafy a pak je vkládáte do textu.

Měření bez času

Zajímá nás asymptotická složitost. Pokud umíte zrychlit váš kód aby místo čtyř minut běžel tři a půl minuty, uvažte jestli je to opravdu v danou chvíli priorita (radši si dejte záležet na popisu

a tohle nechte na potom). Na druhou stranu pokud něco, co nám doběhne do minuty běží na obdobném stroji už pár hodin, je někde chyba.

Měření času

Opět se zajímáme spíš o asymptotickou složitost a platí předchozí. Na druhou stranu si dejte víc pozor na zbytečně neoptimální kód. Rady pro přesné měření:

- Opakujte měření a berte průměr nebo ještě lépe medián.
- Dejte si pozor ať neběží něco jiného (oblíbený webový prohlížeč může zabírat hodně operační paměti, nehrajte v době měření hardwarově náročné hry ani netěžte kryptoměny...).
- U notebooků vypněte power management.
- Neměřte moc krátké věci (jednou změřte opakované vykonání krátkých operací).

Tvorba grafů

Nástroje

Pro tvorbu grafů můžete použít libovolný dostatečně schopný nástroj, který znáte. Pokud umíte python, pak doporučujeme pandas. Dalším klasickým nástrojem je gnuplot. Samozřejmě lze také využít grafické nástroje jako je gnumeric nebo excel.

Barvy

Volte dostatečně snadno odlišitelné barvy (či nerozlišujte jen barvami), někteří cvičící nemají tak skvělý barvocit (Karel).

Klasické vs. logaritmické měřítko

V logaritmickém měřítku uvidíte mnohem lépe, jestli se jedná o funkci logaritmickou nebo konstantní. Také lépe zobrazíte zároveň velké a malé hodnoty. Pokud si nejste jistí, můžete vygenerovat i oba grafy (jak v logaritmickém tak v lineárním měřítku). Každopádně nezapomínejte u všech grafů v logaritmickém měřítku uvést, že jsou v logaritmickém měřítku.

Popisky u grafu

Všichni známe zadání. Stejně chceme popisky os, jednotky, atd. Věřte, že vaši spolužáci odevzdají grafy, u kterých nebudete vědět, co tam je.

- Co je na grafu?
- Co jsou osy?
- Jaké jednotky jsou na osách použity?
- Jaké je měřítko?

Vzhled

Nehodnotíme estetické cítění, ale hodně hodnotíme přehlednost. Pokud nějakou křivku nevidíte (například malé hodnoty, schovaná za jinou moc tlustou křivkou, malý graf...) je to špatně.

Popis výsledků

Rozlišujte mezi hypotézou, měřením a dokázanými tvrzeními

To, co vidím z grafu, jsou naměřené hodnoty a věty, které je popisují, by měly upozorňovat, že jde o výsledky měření. Například takto:

1. "**Z grafu vidíme**, že studenti, se snahou o dodržování těchto pravidel dostali 2-krát více bodů z domácích úkolů"
2. "**V testech nám vyšlo**, že ani dodržování těchto pravidel nemusí zajistit plný počet bodů ze všech domácích úkolů."

Na základě těchto výsledků potom můžeme vytvořit hypotézu, která by měla být uvedena tak, aby bylo jasné, že jde o domněnku. Příkladem budiž:

1. "**Na základě naměřených výsledků se domnívám**, že přečtení těchto rad dopomůže studentům ke lepším výsledkům"
2. "**Myslím si**, že lepší výsledků bylo dosaženo díky tomu, že studenti měli lepší představu, co se od nich očekává"

Může se však hodit i odkázat se na dokázanou větu, například:

1. "**Jak jsme si ukázali na přednášce**, jednička je různá do nuly"
2. "**Podle článku [citace] platí**, že ..."

A ideálně byste se měli vyhnout tvrzením, u kterých není jasné, kterého druhu (měření, domněnka, věta) jsou:

1. "Všichni koně mají stejnou barvu."

Dané příklady rozhodně nejsou jediným způsobem jak rozlišit měření od hypotézy a od dokázaných vět. Pouze na nich chceme ilustrovat rozdíly mezi danými typy vět. Všimněte si, že nedoporučujeme tvrzení, které není nijak uvedeno. To speciálně pokud váš text bude obsahovat taková tvrzení, jež budou lživá, očekávejte, že cvičícího bude zajímat důkaz.

Rozlišujte mezi asymptotickou, amortizovanou, očekávanou a průměrnou složitostí

Prvně připomeňme intuitivně co tyto pojmy znamenají:

1. **Asymptotická složitost** – odhad kolik kroků program provede (nezajímají nás tolik multiplikativní ani aditivní konstanty[1]). Nejčastěji uvádíme nejhorší případ.

[1] Pozor, u měření výsledků budete multiplikativní konstantu odhadovat!

2. **Amortizovaná složitost** – odhad kolik kroků připadne na jednu operaci (například insert) v sérii za sebou jdoucích operací.

3. **Průměrná složitost** – střední hodnota složitosti, pokud vstup je náhodný (v tomto předmětu se s ní spíš nesetkáme). Dalším typem je pokud je náhoda skrytá v samotné datové struktuře a funguje proti nezávislým datům (například hash-tabulka s náhodným výběrem hash funkce). Je nutné rozlišovat přes jakou náhodnost bereme průměr (nebo střední hodnotu)!

4. **Asymptotická prostorová složitost** – kolik paměti využije datová struktura (nebo program). Většinou bude jednoduché určit pro datové struktury, kterými se budeme zabývat.

V domácích úkolech budete měřit průměr počtu kroků potřebných na jednu operaci přes danou sadu operací, ale to neznamená, že výsledky říkají něco o průměrné složitosti. Ve skutečnosti děláte experiment, jestli pro danou posloupnost operací se datová struktura chová tak, jak očekáváme podle amortizované analýzy časové složitosti. Ale stejně jako nemůžete změřit asymptotickou složitost algoritmu, nemůžete ani změřit amortizovaný počet kroků.

Pozor na zaměňování dolního a horního odhadu

Horní odhad (intuitivně a ve většině případů):

- Máte algoritmus, znáte jeho worst-case asymptotickou složitost.
- To je to, s čím se často setkáváte – například bubble sort asymptotická složitost $\mathcal{O}(n^2)$, (ale na některých vstupech může být i lineární).

Dolní odhad pro daný problém a libovolný algoritmus:

- Buď se jedná o velice omezený model – třídění, ale umíme jen porovnávat a kopírovat čísla.
- Nebo je to opravdu zajímavý výsledek (který jste ještě nejspíš neviděli / neuvidíte). Právě o problémech na datových strukturách se nějaké netriviální věci umí dokázat. Takže rozhodně nezkoušejte v domácím úkolu tvrdit, že pro daný problém je třeba vykonat aspoň tolik kroků, pokud to neumíte dokázat nebo neumíte citovat zdroj.

Dolní odhad (pro daný algoritmus):

- Pro tento algoritmus mám tento vstup, na kterém běžel dlouho.
- Často uvidíte spíš formou měření nebo poznámek na přednášce než rigorózních vět z přednášky.

Nedělejte gramatické chyby

Pokud jich bude moc, budeme smutní.