∃KSI

Korespondenční Seminář z Informatiky

Milé řešitelky, milí řešitelé,
děkujeme vám za vaši píli a ještě před uzavřením sady první vám nabídneme druhou sadu, která samozřejmě přináší další porci příkladů.

V úvodníku ke druhé sadě se seznámíme s několika novými konstrukcemi, které v Pythonu budeme používat. Potom si povíme něco o objektově orientovaném programování a ukážeme si, jak se s nim pracuje v Pythonu. Nakonec si představíme některé základní datové struktury, které se při programování používají.

Mapa (slovník)

Mapa je složený datový typ, který slouží pro ukládání dvojic hodnot – klíč a jeho hodnotu $(key, value)$. Ke každému klíči k máme jednoznačně přiřazenou nějakou hodnotu v. Obráceně to ale platit nemusí. Mapě se také říká slovník a není to jenom náhoda – fungují totiž úplně stejně. Jako příklad si tedy zavedeme anglicko-český slovník, se kterým budeme nadále pracovat. Mapa se označuje složenými závorkami, klíč a hodnota se oddělují dvojtečkou, mezi jednotlivými dvojicemi píšeme čárku.

```
>>> enToCz = { 'one': 'jedna', 'two': 'dva' }
>>> emptyMap = {}
>>> print enToCz
{ 'one': 'jedna', 'two': 'dva' }
>>> print emptyMap
{}
```

Vložení nové hodnoty do mapy se provádí stejně jako u seznamu, tedy přes hranaté závorky.

```
>>> enToCz['three'] = 'tri'
>>> enToCz['four'] = 'ctyry'
>>> enToCz['fajf'] = 'pet'
>>> print enToCz
```

Odstranění určité hodnoty se provádí klíčovým slovem `del`. Pro nalezení hodnoty, která je spojená s daným klíčem, použijeme hranaté závorky a do nich napsaný klíč, jehož hodnotu hledáme. Protože se hodnoty vyhledávají pouze pomocí klíče, nezáleží na pořadí, ve kterém dvojice do mapy vkládáme.

```
>>> enToCz['four'] = 'ctyri' # Oprava preklepu
>>> del enToCz['fajf'] # Smazani spatneho klice
>>> print enToCz
>>> print enToCz['four']
```

Řekli jsme si, že každý klíč může ukazovat jenom na jednu hodnotu. Přepsáním klíče `'four'` na hodnotu `'ctyri'` jsme tak zároveň z mapy odstranili dvojici `'four' : 'ctyry'`. Dva různé klíče ale mohou ukazovat na stejnou hodnotu:

```
>>> enToCz['language'] = 'jazyk'
>>> enToCz['tongue'] = 'jazyk'
>>> print enToCz
>>> print enToCz['tongue']
>>> print enToCz['language']
```

Počet dvojic uložených v mapě zjistíme použitím funkce `len`

```
>>> len({'yes':'ano', 'no':'ne'})
>>> len({})
>>> len(enToCz)
```

Často budeme potřebovat také získat všechno, co už je v mapě uloženo. K tomu nám slouží metoda `keys()`, kterou aplikujeme přímo na danou mapu. Ta nám vrátí seznam všech klíčů, pro které má tato mapa uloženou nějakou hodnotu.

```
>>> {'yes':'ano', 'no':'ne'}.keys()
>>> {}.keys()
>>> enToCz.keys()
```

Třídy

Python je objektově orientovaný jazyk. Základním stavebním kamenem programu jsou tedy objekty. Objekty mohou obsahovat nějaké proměnné a funkce, které s těmito proměnnými pracují. Pro vytváření objektů používáme *třídy* – něco jako formičky/šablony pro vykrajování perníčků (objektů). Třídy se zapisují klíčovým slovem `class`.

```
class point():
    def __init__(self, x, y):
        self.coordinate_x = x
        self.coordinate_y = y

    def print_point(self):
        print ('Bod na pozici [', self.coordinate_x, ', ',
                                    self.coordinate_y, ']')

    def distance_to(self, that_point):
        dx = self.coordinate_x - that_point.coordinate_x
        dy = self.coordinate_y - that_point.coordinate_y
        from math import sqrt        # Nauc se odmocnovat
        z~= sqrt(dx*dx + dy*dy)      # Pythagorova veta
        return
```

Nyní jsme nadefinovali třídu, která má dvě metody (tj. funkce, které lze volat přímo nad objektem). První je metoda `__init__`. Toto je speciální metoda, která se zavolá pouze jednou, a to při vytváření objektu podle šablony. Tato metoda přijímá tři argumenty: `(self, x, y)`. První z nich je odkaz objektu sama na sebe. Při volání metod se naplní automaticky a nemusíme se o něj starat. Hodnoty z proměnných x a y se při vytváření nového objektu podle šablony `point` ukládají do proměnných `self.coordinate_x, self.coordinate_y`. Všimněte si zajímavého názvu těchto proměnných, který přímo odkazuje na objekt, kterému patří. Po tomto odkazu následuje tečka a

pak název proměnné. Tento zápis znamená, že nově vytvořený objekt bude mít definovány ještě dvě proměnné `coordinate_x, coordinate_y`. Pokud v programu budeme mít takovýto objekt uložený v nějaké proměnné `bod`, tak X-ová souřadnice tohoto bodu bude uložená v proměnné `bod.coordinate_x`.

```
>>> bod1 = point(0, 0)
>>> bod2 = point(5, 0)
>>> bod3 = point(3, 4)
>>> print 'X-ova souradnice bodu 2 je: ', bod2.coordinate_x
>>> print 'Y-ova souradnice bodu 3 je: ', bod3.coordinate_y
```

Druhá metoda třídy `point` se jmenuje `print_point()`. Tato metoda vypíše souřadnice bodu na standardní výstup. Všimněte si, že i když metoda nepřijímá žádné argumenty od programátora, tak má stále jeden argument s odkazem sama na sebe.

```
>>> bod1.print_point()
>>> bod2.print_point()
>>> bod3.print_point()
```

Poslední metoda `point.distance_to(that_point)` přijímá od programátora jeden argument. Od něj navíc předpokládáme, že to bude taky objekt typu `point`. Tato metoda spočítá vzdálenost zadaného bodu od bodu, na kterém tuto metodu zavoláme.

```
>>> print 'Bod 2 je od bodu 1 vzdalen ', bod1.distance_to(bod2)
>>> print 'Bod 3 je od bodu 1 vzdalen ', bod1.distance_to(bod3)
```

Poznámka: Říkali jsme si, že Python je objektově orientovaný jazyk. I když jste si to neuvědomovali, tak s objekty jste pracovali už od začátku i v první sadě – vzpomenete si například, jak se přidává nový prvek do seznamu?

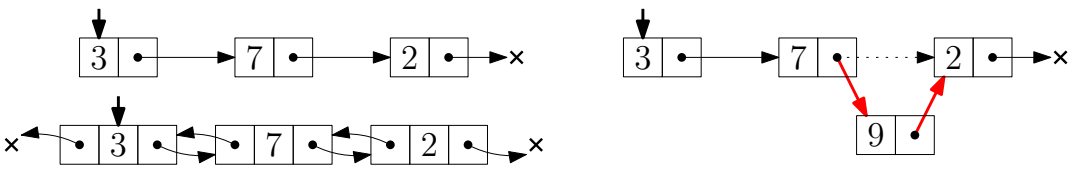
Datové struktury

Lineární spojový seznam

Základním stavebním prvkem seznamu je *uzel*. Každý uzel obsahuje hodnotu daného typu a ukazatel na další uzel. Seznam je v programu reprezentován ukazatelem na první z nich.

Seznam také může být tzv. *obousměrný*, pokud je v uzlu uložen ukazatel jak na další, tak i na předchozí uzel.

Pokud poslední uzel má za následující uzel uzel první (v obousměrném seznamu i první uzel obsahuje ukazatel na poslední jako předcházející) tak řekneme, že seznam je cyklický.



Obrázek 1: Jednosměrný a obousměrný seznam; proces přidání prvku

Přístup k prvkům se zde výrazně komplikuje. Pokud chceme zjistit/změnit hodnotu na k-té pozici (a nemáme přímý ukazatel na daný uzel), musíme ho nejdříve najít. Začneme od prvního uzlu seznamu a skočíme do následujícího uzlu. Takto musíme skočit celkem k-krát, abychom získali k-tý uzel. Přístup k prvku má tedy lineární časovou složitost vzhledem k délce seznamu.

Výhodou je, že pokud máme ukazatel na uzel uprostřed seznamu, můžeme hned za něj vložit další prvek v konstantním čase. Stačí přehodit ukazatele.

Obdobně lze provést smazání uzlu následujícího za uzlem u. Smazat přímo uzel u lze pouze v obousměrném seznamu; v jednosměrném nelze přepojit ukazatel v předcházejícím uzlu.

Zásobník a fronta

Zásobník a fronta jsou struktury, které se nejčastěji používají k dočasnému ukládání dat. Nabízí pouze dvě operace: přidat prvek(*push*) a odebrat prvek(*pop*).

Fronta (angl. *queue*) slouží pro ukládání dat, které mají postupovat jako lidé v běžné frontě – kdo první přijde, ten první odejde (také zkráceně *FIFO – First In First Out*). Její aplikace je přímočará – např. vyřizování příchozích požadavků.

Zásobník (angl. *stack*) funguje přesně obráceně – vždy nejprve odchází ten, kdo vstoupil jako poslední (*LIFO – Last In First Out*). Typické využití je ukládání rekurzivních volání v programech.

Obrázek 2: Princip fronty a zásobníku

Obě struktury lze implementovat tak, aby přidání i odebrání mělo konstantní časovou složitost.

U zásobníku se nabízí implementace pomocí spojového seznamu. Poslední přidaný prvek je vždy na první pozici (prvky jsou tedy v obráceném pořadí, než v jakém byly přidány). Přidat na začátek i odebrat první prvek umíme na seznamu v konstantním čase.

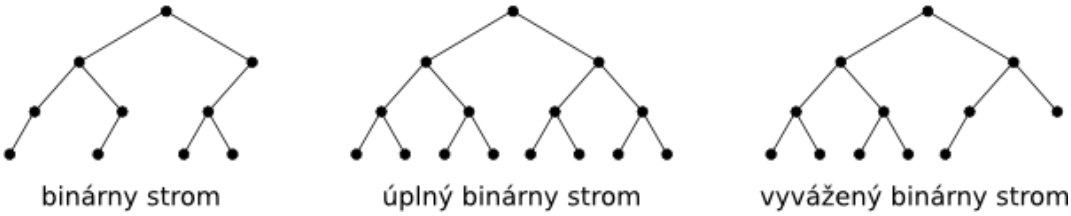
Fronta je jen o trochu komplikovanější. Vytvoříme standardní spojový seznam, ale kromě ukazatele na první uzel si budeme uchovávat i ukazatel na poslední. Odebrání ze začátku i přidání na konec pak opět zvládneme v několika operacích.

Binární strom

Graf je složený z množiny vrcholů (tzv. uzlů) a hran, kde každá hrana vede mezi dvěma vrcholy. Pokud se použitím různých hran umíme z některého vrcholu dostat zpět do tohoto vrcholu, pak řekneme, že graf obsahuje kružnici. Pokud graf žádnou kružnici neobsahuje a všechny vrcholy jsou spojené (pro každou dvojici existuje nějaká cesta z hran, která tyto dva vrcholy spojuje), pak tento graf nazýváme stromem.

Strom se skládá z kořene (libovolný vrchol, typicky ho kreslíme nahoře) a jeho potomků (kreslíme je o úroveň níž než původní vrchol). Tito potomci mohou mít své vlastní potomky, ti mohou mít další potomky, atd. Vrchol, který nemá žádné potomky, nazýváme list.

Strom, ve kterém má každý vrchol nanejvýš 2 následníky, nazýváme *binární strom*. V případě, že každý vrchol je buď listem, nebo má právě dva potomky, pak strom nazýváme úplným binárním stromem. Pokud strom nemá dostatek vrcholů, aby byl úplný, pak vrcholy zarovnáváme doleva (viz. obrázek 3).



Obrázek 3: Binární stromy

Procházení stromu

Na procházení stromů se nejčastěji používají tyto dva algoritmy: *BFS* (*Breadth First Search* – procházení do šířky) a *DFS* (*Depth First Search* – procházení do hloubky).

Myšlenka obou algoritmů je prostá – máme nějakou datovou strukturu, do které si ukládáme vrcholy, které budeme v budoucnu muset zpracovat. Na začátku si sem uložíme počáteční vrchol (kořen) a postupně přidáváme další vrcholy, které postupně potkáváme. Algoritmus si vždy načte jeden vrchol, který jsme ještě nezpracovali, a přidá do seznamu ke zpracování všechny svoje potomky. Rozdíl mezi těmito dvěma algoritmy je pak pouze v tom, jestli použijeme na ukládání potomků frontu (BFS) nebo zásobník (DFS).

Procházení do šířky zpracovává strom po vrstvách. Nejprve zpracuje všechny uzly, které jsou od kořene vzdálené o 1 hranu. Potom uzly ve vzdálenosti 2 od kořene a tak dále.

```
function BFS(vrchol (kořen) stromu v)
    vlož v do fronty
    while fronta není prázdná do
        x ← vezmi následující vrchol z fronty
        vlož všechny potomky x do fronty
    end while
end function
```

Při procházení do hloubky nahradíme frontu zásobníkem, jinak algoritmus vypadá úplně stejně. Pokud si budeme postup algoritmu zakreslovat do stromu, tak prohledávání do hloubky vybere vždy nějakého potomka právě zpracovávaného vrcholu a začne zpracovávat jeho. Pak vybere potomka tohoto potomka, apod. Výsledek tedy bude takový, že v grafu vytvoří cestu vedoucí hluboko a postupně se z ní bude vracet.

```
function DFS(vrchol (kořen) stromu v)
    for all p ← CHILDREN(v) do DFS(p)
    end for
end function
```

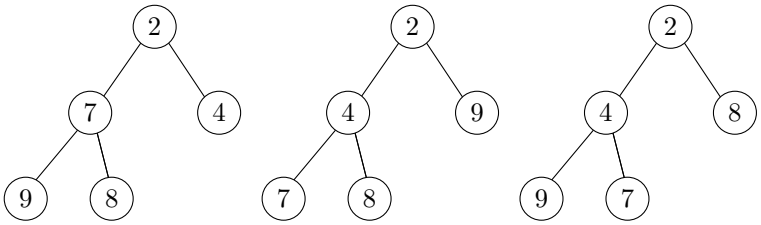
Halda

V haldě (angl. *heap*) ukládáme hodnoty, které mezi sebou umíme porovnávat. Přidání a odebrání prvku lze provést v čase $\mathcal{O}(\log N)$ a nalezení nejmenšího prvku v *konstantním čase*. Rozlišujeme minimovou a maximovou haldu. V úvodníku použijeme pro příklady minimovou. Maximová halda funguje přesně naopak.

Haldu je tedy vhodné použít, pokud máme množinu hodnot, která se průběžně mění, a často nás zajímá nejmenší prvek v daný moment.

Halda má strukturu binárního stromu s dvěma důležitými vlastnostmi:

1. Strom je *vyvážený*: je-li to možné, všechny cesty od kořene k listům mají stejnou délku. Není-li to kvůli počtu prvků možné, poslední úroveň je zarovnaná doleva (obr. 3).
2. Hodnota v každém vrcholu je *menší*, než hodnoty v jeho potomcích. Cesta z kořene do listů tedy vždy tvoří rostoucí posloupnost.

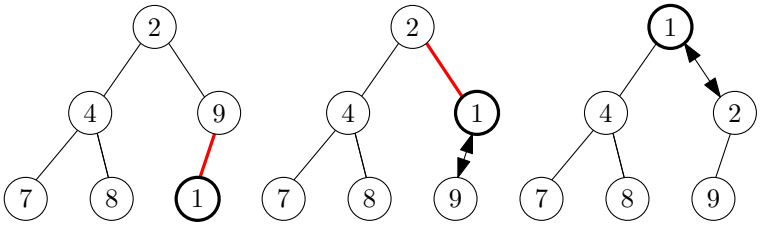


Obrázek 4: Různé haldy pro hodnoty $\{2, 4, 7, 8, 9\}$

Pro každou množinu hodnot přirozeně existuje více možností, jak může halda vypadat. Vždy ale platí, že nejmenší prvek se nachází v kořeni. Operace na haldě provádíme následovně:

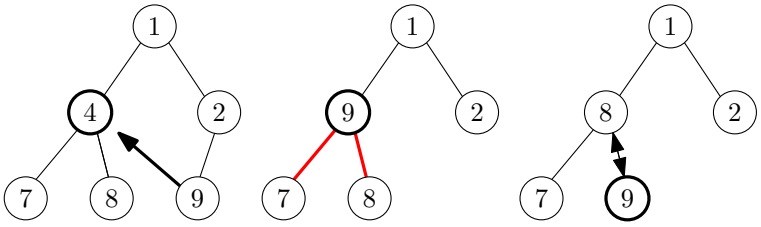
Nalezení nejmenšího prvku. Nejmenší prvek se nachází v kořeni, nemusíme tedy nic hledat. Operace má konstantní časovou složitost.

Přidání prvku. Prvek nejprve přidáme do nejnižší úrovně tak, aby byla splněna první podmínka. Nyní se může stát, že jeho rodič je větší a je tedy porušena druhá podmínka. Provedeme proto *probublání nahoru* (dokud je prvek větší než jeho rodič, prohodíme je spolu a pokračujeme dál). Takto se může nový prvek dostat až do kořene haldy.



Obrázek 5: Proces přidání čísla 1

Odebrání prvku. Nejprve se opět postaráme o platnost první podmínky. Vybereme vrchol z nejnižší vrstvy nejvíce vpravo a dáme ho na místo prvku, který chceme smazat. Mohou nastat dva problémy. Prvek může být menší, než jeho rodič. V tom případě provedeme probublání nahoru, stejně jako při přidávání. Prvek ale může také být větší, než některý jeho potomek. Potom provedeme tzv. *probublání dolů* (prvek prohodíme s menším z jeho synů a pokračujeme dále, dokud druhá podmínka neplatí). Prohození s větším z jeho synů by nefungovalo – rozmyslete si proč!



Obrázek 6: Proces odebrání čísla 4

Při přidávání a odebírání provedeme nejvýše tolik prohození, jaká je hloubka stromu. Je-li počet prvků N, výška haldy je $\lceil \log_2(N+1) \rceil$, a proto je složitost popsaných operací logaritmická.

Dodejme, že porovnávání čísel můžeme provádět přesně obráceně – v kořeni pak bude maximální prvek. Za zmínku také stojí, že haldu lze výhodně reprezentovat v poli. Kořen se nachází na pozici 1 a dále platí, že prvek na pozici k má potomky na pozicích $2k$ a $2k+1$. Pole je potom souvisle zaplněno.

Heapsort

Na haldě je založen známý algoritmus pro třídění čísel zvaný *heapsort*. Pole čísel velikosti N lze tímto algoritmem setřídit s časovou složitostí $\mathcal{O}(N \log N)$, což je *optimální* složitost – lze dokázat, že neexistuje porovnávací algoritmus, který by to zvládl s lepší asymptotickou složitostí. Pro porovnání: známý *quicksort* má časovou složitost v nejhorším případě až $O(N^2)$, ale průměrně třídí také v čase $O(N \log N)$. Reálně je trochu rychlejší a jednodušší na implementaci, takže je také používanější.

Algoritmus nejprve ze všech čísel vybuduje haldu. To lze provést např. postupným přidáváním prvků. Časová složitost tohoto vybudování je $\mathcal{O}(N \log N)$, neboť N-krát provedeme operaci přidání se složitostí $\mathcal{O}(\log N)$. Lze to provést i lépe v lineárním čase, nám však tato složitost postačí.

Potom z haldy postupně odebíráme nejmenší prvek, dokud není prázdná. Tím logicky dostaneme čísla ve vzestupném pořadí. Složitost tohoto kroku je také $\mathcal{O}(N \log N)$, stejně tak tedy složitost celého třídění.

Shrnutí

Dnes jsme si dozvěděli něco o různých datových strukturách a jejich složitosti. Ukázali jsme si, jak je v Pythonu implementovaná mapa (slovník). Probrali jsme také základní použití tříd v Pythonu jako objektově-orientovaném jazyku.

V případě zvídavých dotazů k tématu se nestyďte napsat na diskuzi na našich stránkách. Hodně štěstí v druhé sadě!

Zadání 2. sady úloh KSI (termín odevzdání: 15. 12. 2014)

Řešení zasílejte pomocí internetového systému na adrese `http://ksi.fi.muni.cz`.

Příklad 1: Vianočná akcia (10 bodů)

Blížia sa vianoce a Karík ide spraviť veľký nákup do miestneho supermarketu. Aby nič nezabudol, tak všetky veci čo potrebuje kúpiť, si napísal na jeden veľký list papiera, každú vec do nového riadku. Ak chce niečoho kúpiť viac kusov, tak sa tieto veci budú nachádzať na viac riadkoch. Aby to mal prehľadnejšie, párne riadky napísal červenou fixkou a nepárne zelenou. Keď prišiel do supermarketu, tak zistil, že práve prebieha veľký predvianočný výpredaj a na všetok tovar v supermarkete je akcia 1+1 zdarma. Pozrel sa na svoj zoznam a zistil, že posledný riadok je napísaný zelenou fixkou - počet vecí v zozname je nepárny. To zrejme značí, že niektorá vec sa tam určite nachádza nepárny počet krát - takže určite nevyužije akciu naplno. Preto sa rozhodol, že kúpi ešte jednu vec, tak aby cena nákupu zostala rovnaká. Viac by si totiž netrúfal odniesť - veď tašky majú tiež obmedzenú nosnosť :).

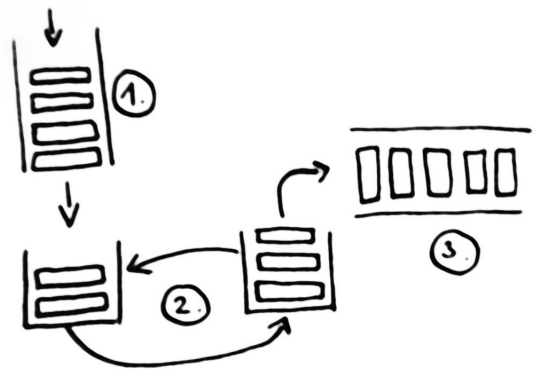
Predstavte si veci ako celé čísla(môžu byť aj záporné alebo väčšie ako miliarda). Potom je vašou úlohou pomôcť Karlíkovi a naimplementovať funkciu `FindOdd`, ktorá dostane ako vstupný parameter zoznam celých čísel. Táto funkcia musí nájsť a vrátiť jedno číslo, ktoré sa tam nachádza nepárny počet krát. Nezabúdajte, že Karlík má dosť veľký nákup a Vianoce sú už za dverami, takže vaša funkcia by to mala spraviť čo najrýchlejšie.

Hodnotiť budeme funkčnosť naimplementovanej funkcie a samozrejme stručný popis vášho riešenia so zdôvodnením ako aj správnosti tak aj časovej a pamäťovej zložitosti.

Příklad 2: Baliaci prepravník (10 bodů)

Pri nakupovaní v nákupnom centre sa Karlík zatúlal do skladových priestorov, kde už v plnom prúde prebiehalo balenie darčekov na baliacej linke.

Baliaca linka fungovala nasledovne. Na začiatku stroja bol pás, na ktorom sa balili darčeky, na konci pásu sa darčeky zhromažďovali do prepraviek, v ktorých putovali po skladovej hale, kde ich na konci čakal ďalší prepravný pás, ktorý postupne odoberal darčeky a posielal ich na odvoz do sveta(pre lepšiu predstavu viď nákres), odkiaľ sa prepravky vracali späť ku baliacemu pásu.

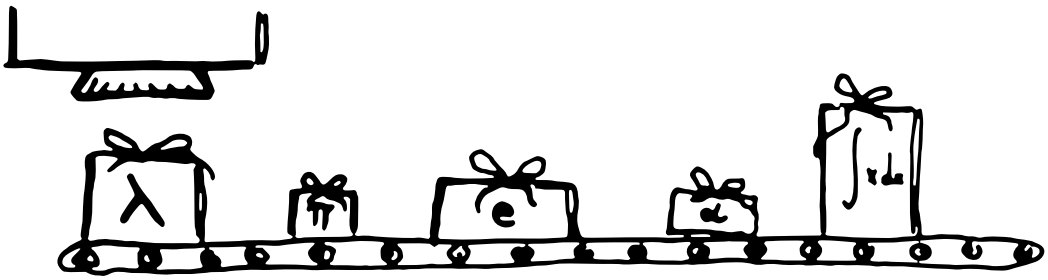


Obrázek 7: 1. Baliaci pás, 2. Prepravky, 3. Transport

Karlík si všimol, že síce prepravník funguje spoľahlivo, ale nie najefektívnejšie. Preto sa rozhodol, že navrhne lepší systém a vy mu s tým pomôžete. Karlík si všimol nasledujúce problémy:

- V prípade, že je darček zle zabalený aj tak putuje cez prepravník, preto by bolo fajn pridať do systému kontrolu kvality zabalenia.
- Ak vykladanie balíkov v časti 3. nestíha musí časť 1. čakať na volné prepravky. Nešlo by to zefektívniť pridávaním prázdnych prepravok do systému vždy, keď su potrebné? Zároveň nechceme, aby sme systém zahltili prepravkami, takže v prípade, že je ich prebytok ich môžeme zo systému odobrať, ale keďže odoberanie je zložitá operácia chceme ju vykonávať, čo najmenej krát, teda len v prípadoch, že má systém prebytok prepraviek.
- V momente, keď sa zastaví baliaca linka, existujú v systéme balíky, ktoré neboli odoslané na prepravu ďalej, preto by bolo fajn v prípade, že systém dostane požiadavok o vypnutie nechať transport darčekov dobehnuť až do konca.

Ako ste už asi pochopili vašou úlohou bude modifikovať, už funkčný systém, ktorý je vám k dispozícii.

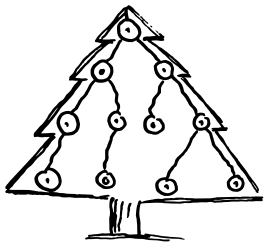


Příklad 3: Umělecká chvilka (10 bodů)

S blížícími se Vánoci Karlík přemýšlí, jak bude vypadat jeho vánoční stromeček. Své vysněné vánoční stromečky si kreslí ve formě binárních stromů (ne nutně úplných).

Protože správný vánoční stromeček musí být plný světýlek, vybarvuje Karlík postupně jednotlivé vrcholy stromu. Má však po ruce jen 3 pastelky – červenou, modrou a zelenou. Po chvilce Karlík zjišťuje, že mu často některá z barev výrazněji převažuje nad ostatními. Aby byl stromeček co nejvíce barevně vyvážený, rozhodne se zavést následující pravidlo: Žádné 2 sousední vrcholy (myšleno vrchol a jeho potomek) nesmí mít stejnou barvu a současně každí 2 potomci téhož vrcholu budou taktéž různobarevní (jinak řečeno, každý vrchol a jeho levý a pravý přímý potomek budou mít jinou barvu).

Avšak Karlík je chvílemi nepozorný a občas se splete. Pomozte Karlíkovi rozhodnout, zda je daný strom správně obarvený dle výše uvedených pravidel. Máte zaručeno že se bude jednat o korektní binární strom.



Konkrétně: Naimplementujte dodanou funkci `is_christmas_tree`, která přijímá 1 parametr, a to vstupní strom. Současně dostanete předpřipravený minimálně 1 korektní a 1 nesprávný příklad stromu pro základní otestování, další případy si již musíte vytvořit sami. Při řešení si dejte pozor

na fakt, že strom nemusí být úplný, tj. některé vrcholy můžou obsahovat např. jen 1 libovolného syna. Avšak musí být souvislý. Můžete také předpokládat, že každý vrchol bude mít přiřazenu právě 1 z výše uvedených barev.

Jako řešení dodejte implementovanou funkci (5b) a stručný slovní popis včetně zdůvodnění časové a paměťové složitosti ověřování barevné korektnosti stromu (5b).

Hint: Při použití vhodné programátorské techniky k procházení zadaného stromu se vyhnete použití pomocných datových struktur pro zapamatování doposud nezpracovaných vrcholů. Navíc bude vaše funkce krátká a přehledná.

Příklad 4: Bláznivé řazení (10 bodů)

Pro tuto úlohu si Karlík vymyslel vlastní datovou strukturu, které říká rekurzivní cyklický seznam seznamů (dále RCSS). Základem RCSS je cyklický seznam délky $k \geq 2$. Každý prvek tohoto cyklického seznamu (kromě vstupního prvku) je vstupním prvkem do cyklického seznamu délky $k-1$. Každý prvek tohoto menšího seznamu (kromě vstupního prvku) je vstupním prvkem do cyklického seznamu délky $k-2$. Takto se napojené cyklické seznamy zmenšují, dokud nemají velikost 2. Například RCSS čítající 16 prvků by obsahoval jeden cyklický seznam délky 4, tři cyklické seznamy délky 3 a 6 cyklických seznamů délky 2. Pro řešení podúloh můžete počítat s tím, že každý prvek RCSS je naplněn jedním celým číslem, přičemž žádné číslo se v RCSS nevyskytuje víckrát než jednou. Prvky jsou indexované tak, že vnitřní cyklický seznam má indexy od 0 do $(k-1)$, indexování potom pokračuje na seznamech ve druhé úrovni tak, že nejnižší indexy mají prvky seznamu vycházejícího z prvku s indexem 1 atd.

a) Vaším prvním úkolem bude seřadit data v RCSS od nejmenšího po největší co nejefektivněji tak, aby v prvku s indexem 0 bylo uloženo nejnižší číslo a v prvku s nejvyšším indexem nejvyšší číslo. Výstupem by měl být dobře popsaný algoritmus v pseudokódu, případně řádně okomentovaný spustitelný kód v Pythonu.
b) Druhým úkolem je vyjádřit rekurentním vztahem, kolik maximálně může být v RCSS prvků za předpokladu, že vnitřní cyklus bude mít k prvků. Pokud si například budete myslet, že při každém zvýšení k o jedničku se počet prvků x zdvojnásobí, vyjádřili byste to takto:
   $x_2 = 2$ (hraniční podmínka – RCSS nemůže mít méně než 2 prvky a při dvou prvcích obsahuje vnitřní cyklus oba tyto prvky)
   $x_k = 2(x_{k-1})$ (tedy například $x_4 = 2x_3 = 2(2x_2)$ a protože víme, že $x_2 = 2$, můžeme postupně dopočítat $x_4 = 8$)

Pro jednotnost vašich řešení prosíme používejte označení, kde každý vrchol RCSS *node* má tyto atributy:

- *small* je počet prvků cyklického seznamu, jehož je *node* vstupním prvkem, má hodnotu 0, pokud se jedná pouze o prvek cyklického seznamu délky 2
- *big* je počet prvků cyklického seznamu, jehož není *node* vstupním prvkem, má hodnotu 0, pokud se jedná o vstupní prvek celého RCSS
- *nextSmall* je ukazatel na následující prvek cyklického seznamu, jehož je *node* vstupním prvkem, má hodnotu 0, pokud se jedná o prvek pouze cyklického seznamu délky 2 a žádného jiného
- *nextBig* je ukazatel na následující prvek cyklického seznamu, jehož není *node* vstupním prvkem, má hodnotu 0, pokud se jedná o vstupní prvek celého RCSS
- *value* je hodnota uložená v *node*.

Příklad 5: Sobí stáj (10 bodů)

Karlík dostal od Ksanty nabídku pomáhat s řízením stájí sobů. Jeho úkolem je zaznamenávat pohyb sobů ve výpravné stáji. Výpravná stáj funguje tak, že soby postupně odlétají roznášet vánoční dárky ale také přicházejí nový. Každý sob má svoji prioritu (kde za prioritu považujeme celé číslo) a když Ksanta potřebuje nějaké soby pro roznášení dárků, odebere ze stáje několik sobů s nejnižší prioritou. Pokud do stáje nějací soby přicházejí, mají také nějaké priority - je potřeba je tedy sloučit tak, aby se žádnému sobovi nezměnila priorita a záznam o sobech ve stáji obsahoval správně všechny soby.

Vašim úkolem je vymyslet způsob, jakým by se dal pohyb ve výpravné stáji zaznamenávat. Jsou vám k dispozici některé datové struktury (halda, zásobník s destruktivním čtením, fronta a mapa), přičemž klidně můžete využívat (nebo tvořit) libovolné další struktury. Vystačit byste si ovšem měli s připravenými strukturami.

Hlavním místem, kde budete své řešení tvořit, je metoda `proved()`, která je umístěna ve třídě `KsantuvStajovyZaznamenavac`. Metoda jako parametr přijímá slovník obsahující klíče akce a data. Akce může být `pridej`, `odeber` a `vypis_epochu`, formát dat je popsaný v souboru *task2_5.py*.

Akce `pridej` znamená, že do stáje potřebujeme přidat nové soby, naopak `odeber` znamená odebrání sobů s nejnižší prioritou. Nejzajímavější je akce `vypis_epochu`, která vypíše stav stáje po určitém počtu přidání a odebrání sobů. K dispozici máte ještě funkci `udelej_obraz(struktura)`, která vrací tzv. hlubokou kopii dané struktury - taková kopie existuje nezávisle na první struktuře a jakékoliv modifikace originální struktury kopii neovlivní a naopak.

Vysvětlete, jaké struktury jste vybrali a proč, a zkuste se zamyslet, zdali by nešel problém řešit jinak. Pro jednotlivé operace (přidání, odebrání a vypsáni stavu stáje) určete asymptotickou časovou a prostorovou složitost a jednoduše popište, proč vaše řešení funguje.

A to je z této sady KSI vše. Přejeme ti hodně úspěchů při řešení, a když budeš mít jakékoliv otázky, neváhej se na nás obrátit e-mailem na adresu `ksi@fi.muni.cz` nebo v diskuzním fóru na našich webových stránkách.

Termín odevzdání 2. sady úloh KSI: 15. 12. 2014

`http://ksi.fi.muni.cz`