1 (True or False) and Justify [20 bodov]

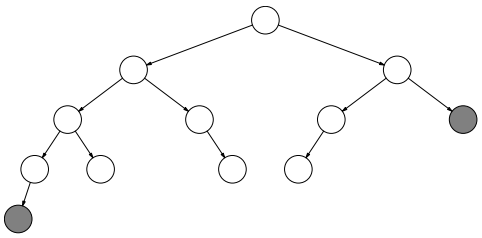
1.1 Zadanie

O každom tvrdení uveďte, či je pravdivé, a tromi vetami svoj názor zdôvodnite. Úplne správna odpoveď: 2.5 boda. Správna odpoveď bez zdôvodnenia: 0.5 boda. Nesprávna odpoveď: 0 bodov. Správna odpoveď s úplne zlým zdôvodnením: 0 bodov.

1. Ak budeme v MergeSorte pole zakaždým deliť na tretinu a dve tretiny, bude časová zložitosť naďalej $O(n \log n)$.
2. Vo vektore už máme 47 prvkov. Postupne na jeho koniec pridáme n ďalších. Toto celé má časovú zložitosť $O(n)$.
3. Usporiadať ľubovoľné pole veľkosti $4\sqrt{n}$ vieme v čase $O(n)$.
4. Hodnoty v *listoch* binárneho vyhľadávacieho stromu sú usporiadané podľa veľkosti zľava doprava.
5. Vo vyváženom binárnom strome sa hĺbky každých dvoch listov líšia nanajvýš o 1.
6. Z binárnej haldy s minimom v koreni sa dá v čase $O(n)$ spraviť binárna halda s maximom v koreni.
7. Vieme si rovnomerne náhodne vybrať z 5 možností, ak smieme počas rozhodovania max. $47\times$ hodiť klasickou kockou?
8. Pri univerzálnom hešovaní do poľa veľkosti $n > 1$ vie nepriateľ vopred pripraviť dvojicu prvkov, ktorá vyrobí kolíziu.

1.2 Riešenie

1. TRUE. Najhlbšia vetva rekurzie bude mať hĺbku približne $\log_{3/2} n < 2 \lg n$.
2. TRUE. Platí ten istý argument ako pre prázdny vektor.
3. TRUE. Zafunguje skoro ľubovoľné triedenie, dokonca aj také s časovou zložitosťou kvadratickou *od dĺžky poľa*.
4. TRUE. Vyplýva to napr. z toho, že in-order zápis listy navštívi v tomto poradí.
5. FALSE.

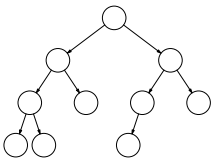


6. TRUE. Odignorujeme, že ide o haldu s minimom v koreni a jednoducho z daných údajov v poli univerzálnym algoritmom bežiacim v čase $O(n)$ vyrobíme haldu s maximom v koreni.
7. FALSE. Ak by taký algoritmus existoval, ľahko ho upravíme do podoby, kedy najskôr hodí presne $47\times$ kockou a potom porobí všetky rozhodnutia. Ale 6^{47} (počet možných postupností výsledkov 47 hodov) nie je deliteľné piatimi.
8. FALSE. Pointa univerzálneho hešovania je práve v tom, že hešovaciu funkciu si náhodne volíme my. Vďaka tomu neexistujú žiadne konkrétne zlé vstupy.

2 Pohľad zvnútra: binárny strom [3+7 bodov + 10 bonus]

2.1 Zadanie

Na obrázku je binárny vyhľadávací strom obsahujúci *párne* čísla od 2 do 20, vrátane.

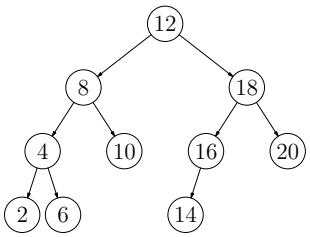


a) Ktoré hodnoty môžu byť v koreni?
b) Nájdite jedno možné poradie, v ktorom mohli byť hodnoty (bez vyvažovania stromu) vkladané, aby mal tento tvar.
c) Koľko rôznych poradí vyhovuje ako odpoveď v časti b)?

2.2 Riešenie

Podúloha a)

Každá hodnota má jednoznačne určené miesto – výpis stromu in-order musí vyrobiť usporiadanú postupnosť. Keďže v ľavom podstrome je 5 vrcholov, v koreni musí byť šieste najmenšie číslo: 12.
Nasledujúci obrázok ukazuje jediné korektné vyplnenie stromu hodnotami:



Podúloha b)

Ako prvé muselo byť zjavne vložené číslo v koreni, teda 12. Následne dostávame dve menšie samostatné podúlohy: povkladať v správnom poradí čísla z ľavého a čísla z pravého podstromu. To môžeme urobiť postupne, a to opakovaním tej istej úvahy. Napr. z ľavého podstromu koreňa sme ako prvú museli vložiť hodnotu 8. Takto dostávame jednu možnú postupnosť vkladaní: 12, 8, 4, 2, 6, 10, 18, 16, 14, 20.
(Povšimnite si, že sme vždy najskôr zostrojili ľavý podstrom, až potom pravý. V dôsledku toho sme dostali práve poradie, ktoré je pre-order zápisom nášho stromu.)

Podúloha c) – bonus

Vždy, keď máme v podúlohe b) dva nezávislé podproblémy, môžeme ich riešenia ľubovoľne preložiť cez seba. Keď napr. vieme, že postupnosť vkladaní 8, 4, 2, 6, 10 zostrojí ľavý podstrom a postupnosť 18, 16, 14, 20 zostrojí pravý, tak nutne musí celý strom správne zostrojiť napríklad aj postupnosť vkladaní: 12, 8, 4, 18, 16, 2, 14, 6, 10, 20.
Pre pravý podstrom koreňa nášho stromu takto zjavne existujú práve tri postupnosti vkladania, ktoré ho zostroja: prvou je 18, 16, 14, 20, druhou 18, 16, 20, 14 a treťou 18, 20, 16, 14.
Pre podstrom s koreňom vo vrchole s číslom 4 sú dve možnosti: 4, 2, 6 alebo 4, 6, 2.
Pre ľavý podstrom koreňa teda máme osem možností. Musíme začať vložením hodnoty 8. Následne máme na výber, či ľavý podstrom pod hodnotou 8 vyrobíme v poradí 4, 2, 6 alebo 4, 6, 2. A tiež máme na výber štyri možnosti, kedy počas toho vložíme hodnotu 10.
Už teda vieme, že existuje 8 postupností vkladania hodnôt, ktoré vyrobia ľavý podstrom koreňa a 3 postupnosti, ktoré vyrobia pravý podstrom koreňa. Existuje teda $8 \times 3 = 24$ možností, ako si vybrať dve konkrétne. A keď sme si už vybrali dve konkrétne, koľkými spôsobmi ich vieme preložiť cez seba? Presne $\binom{9}{5} = 126$, lebo si musíme spomedzi 9 vkladaní vybrať tých 5, kedy vkladáme hodnotu do ľavého podstromu.
Dokopy teda existuje $24 \times 126 = 3024$ vyhovujúcich poradí vkladania.

3 Pohľad zvnútra: nefronta nezásobník [20 bodov]

3.1 Zadanie

Detailne implementujte čo najefektívnejšiu dátovú štruktúru, podporujúcu nasledovné operácie: i) vloženie prvku, ii) výber niektorého prvku. Pri operácii typu ii) je nám skoro jedno, ktorý prvok bude vybraný, máme len jedno obmedzenie: ak sú práve v dátovej štruktúre viac ako dva prvky, tak vybraný prvok nesmie byť ani prvý vložený, ani posledný vložený prvok – teda ani ten prvok, ktorý by bol vybraný, keby šlo o frontu, ani ten, ktorý by sme vybrali, keby šlo o zásobník. Odhadnite časovú zložitosť každej z operácií pri vašej implementácii.
(Nanajvýš 14 bodov môžete dostať za riešenie ľahšej verzie úlohy. V nej môžete bez uvádzania detailov implementácie použiť dátové štruktúry fronta, zásobník, vektor a spájaný zoznam.)

3.2 Riešenie

Obe operácie vieme implementovať v konštantnom čase (resp. amortizovanom konštantnom, ak použijeme polia namiesto spájaných zoznamov).
Jedno možné riešenie je mať všetky prvky v jednom zásobníku. Vkladať budeme ako do normálneho zásobníka, ale vyberať budeme druhý prvok zvrchu (ak sú v zásobníku aspoň dva). Celé by to napr. v Pythone vyzeralo nasledovne:

```
class DivnaFronta:
    def __init__(self):
        self.stack = []

    def push(self,x):
        self.stack.append(x)

    def pop(self):
        assert len(self.stack) > 0
        if len(self.stack) == 1: return self.stack.pop()
        else: return self.stack.pop(-2) # druhy prvok od konca

Q = DivnaFronta()
Q.push(47); print( Q.pop() )
Q.push(47); Q.push(42); print( Q.pop(), Q.pop() )
Q.push(47); Q.push(42); Q.push(23); Q.push(17); print( Q.pop(), Q.pop() )
Q.push(19); print( Q.pop(), Q.pop() )

# output:
# 47
# 47 42
# 23 42
# 17 47
```

Prípadne ekvivalentná implementácia len pomocou klasického `pop()`:

```
    def pop(self):
        assert len(self.stack) > 0
        if len(self.stack) == 1: return self.stack.pop()
        else:
            first  = self.stack.pop()
            second = self.stack.pop()
            self.stack.append(first)
            return second
```

(Inou možnosťou je odložiť si prvý a posledný vložený prvok do samostatných premenných a len ostatné prvky mať všetky v jednej dátovej štruktúre, napríklad zásobníku. Táto možnosť je trochu otravnejšia na implementáciu, ale nie príliš.)

Do úplnosti už treba len doplniť implementáciu samotného zásobníka, tú vieme spraviť buď pomocou spájaného zoznamu, alebo ako vektor: zdvojnásobovaním použitej pamäte.
Kompletná implementácia pomocou spájaného zoznamu:

```
class PolozkaZoznamu:
    def __init__(self, data=None, dalej=None): self.data, self.dalej = data, dalej

class Zasobnik:
    def __init__(self): self.head = None
    def empty(self): return self.head == None
    def push(self,x): self.head = PolozkaZoznamu(data=x,dalej=self.head)

    def pop(self):
        assert self.head != None
        out = self.head.data
        self.head = self.head.dalej
        return out

class DivnaFronta:
    def __init__(self): self.stack = Zasobnik()
    def push(self,x): self.stack.push(x)

    def pop(self):
        first = self.stack.pop()
        if self.stack.empty():
            return first
        else:
            second = self.stack.pop()
            self.stack.push(first)
            return second
```

4 Pohľad zvnútra: záporné čísla [10 bodov]

4.1 Zadanie

V pamäti máme pole $A[0..n-1]$, v ktorom je *ostro rastúca* postupnosť reálnych čísel. Napíšte funkciu, ktorá (ideálne v lepšom ako lineárnom čase) zistí, koľko z týchto čísel je záporných.

4.2 Riešenie

Použijeme binárne vyhľadávanie, aby sme našli posledné záporné a prvé nezáporné číslo. Predstavíme si, že $A[-1] = -\infty$ a $A[n] = \infty$ a inicializujeme ukazovatele l, r na $-1, n$, aby nám od začiatku platil invariant, že l ukazuje na záporné a r na nezáporné číslo. A kým sa ukazovatele nedostanú k sebe, opakujeme: pozrieme sa na hodnotu uprostred medzi nimi a podľa jej znamienka na ňu posunieme jeden z ukazovateľov. Takto v každom kroku zmenšíme interval, v ktorom hľadáme, približne na polovicu, dostávame teda riešenie s časovou zložitosťou $O(\log n)$.
Implementácia v Pythone:

```
l, r = -1, len(A)
while r-l > 1:
    m = (r+l) // 2
    if A[m] < 0: l = m
    else:        r = m
print( 'zapornych cisel je', r )
```

5 Pohľad zvonka: behy [15 bodov]

5.1 Zadanie

Beh je maximálny usporiadaný úsek poľa. Napr. pole $(10, 20, 30, 5, 6, 7, 7, 8, 3, 8)$ má tri behy. Dokážte alebo vyvráťte tvrdenie: existuje algoritmus, ktorý ľubovoľné pole s n prvkami a b behmi usporiada v čase $O(n \log b)$.

5.2 Riešenie

Pole vieme v čase $O(n)$ rozdeliť na jednotlivé behy: jednoducho sekvenčne spracúvame prvky, pričom ak je prvok väčší alebo rovný ako predchádzajúci, pridáme ho do aktuálneho behu, inak ukončíme aktuálny beh a práve spracúvaný prvok použijeme ako prvý prvok nového behu.
Takto dostaneme b rôzne dlhých polí, z ktorých každé už je usporiadané. Súčet dĺžok týchto polí je presne n. Ostáva nám už len pospájať tieto polia do jedného usporiadaného poľa.
To vieme spraviť vhodnými volaniami funkcie Merge z triedenia MergeSort. Spájanie bude prebiehať vo fázach. V každej fáze popárujeme polia do dvojíc (pričom možno ostane jedno nespárované) a na každú dvojicu polí zavoláme Merge. Tým počet polí zmenšíme približne na polovicu. Každá fáza trvá $O(n)$ a celkový počet fáz je $O(\log b)$, čím dostávame požadovanú časovú zložitosť.
Všimnite si, že nestačí pôvodných b polí postupne po jednom mergovať do výsledného poľa. V najhoršom možnom prípade (ak $b = n$, teda vstupné pole bolo usporiadané zostupne) by sme takto dostali riešenie s časovou zložitosťou $\Theta(n^2)$.

Iný funkčný postup spájania polí: Polia, ktoré ešte treba pospájať, si môžeme udržiavať v halde usporiadané podľa počtu prvkov. V každom kroku vyberieme dve najkratšie polia a volaním Merge ich spojíme do jedného. Tento postup bude tiež

mať časovú zložitosť $O(n \log b)$, niekedy dokonca ešte lepšiu. Dokonca sa dá dokázať, že takto pospájame polia pomocou najmenšieho možného celkového počtu operácií. (Táto skutočnosť súvisí s optimálnosťou tzv. Huffmanovho kódu.)

A ešte jedno úplne iné riešenie: Po tom, ako pole v $O(n)$ rozdelíme na jednotlivé behy, zoberieme z každého behu prvý prvok a navkladáme do haldy dvojice (prvok, id behu z ktorého je). A teraz dokola n-krát z tejto haldy vyberieme najmenší prvok a následne do nej vložíme ďalší prvok z toho behu, do ktorého patril práve vybraný. Aj takto dosiahneme časovú zložitosť $O(n \log b)$.

6 Pohľad zvonka: zber známok [15/25 bodov]

6.1 Zadanie

Deti zbierajú známky a nosia ich mame. Navrhnite pre mamu čo najefektívnejší algoritmus, ktorý bude evidovať udalosti nižšie popísaných typov. Mená detí sú ľubovoľné (rozumne dlhé) reťazce, známky majú 9-ciferné evidenčné čísla. Môžete sa bez rozpisovania odvolávať na algoritmy a dôkazy z prednášok.

Ľahšia verzia (15 bodov): Jediný typ operácie je „dieťa s menom x prinieslo známku číslo y“. Pri jej spracovaní je potrebné upozorniť, ak dané dieťa už má rovnakú známku.

Ťažšia verzia (25 bodov): Treba efektívne implementovať nasledovné operácie:
– „dieťa s menom x prinieslo známku číslo y“,
– „dieťa s menom x_1 dalo dieťaťu s menom x_2 známku číslo y“,
– „chceme zoznam všetkých známok, ktoré má dieťa s menom x“.

6.2 Riešenie ľahšej verzie

Stačí si pamätať množinu dvojíc (dieťa,známka). Na to je najvhodnejšou dátovou štruktúrou hešovacia tabuľka. Napr. v C++ by sme mohli použiť `unordered_set< pair<string,int> >`. Ak si označíme ℓ najdlhšiu dĺžku mena dieťaťa, vieme každú operáciu spracovať v očakávanom čase $O(\ell)$.

6.3 Riešenie ťažšej verzie

Tu si stačí samostatne pre každé dieťa pamätať jeho množinu známok. Budeme mať teda jednu hešovaciu tabuľku, v ktorej bude kľúčom meno dieťaťa a hodnotou nová hešovacia tabuľka obsahujúca jeho známky.
Opäť, prvé dve operácie vieme spracovať v očakávanom čase $O(\ell)$, pre tretiu operáciu je to očakávaný čas $O(\ell)$ na nájdenie záznamu pre dané dieťa plus zaručený čas $\Theta(z)$ na výpis jeho z známok.
(Všetky hešovacie tabuľky vieme prípadne nahradiť vyvažovanými stromami, čím sa časová zložitosť vyhľadania zmení na zaručených $O(\ell \log n + \log z)$, kde n je počet rôznych detí a z maximálny počet známok jedného dieťaťa. Prípadne vieme použiť trie na dosiahnutie zaručenej časovej zložitosti $O(\ell)$. Všetky tieto riešenia mohli dostať plný počet bodov.)
Nasleduje ukážková implementácia v C++ a v Pythone.

```
#include <iostream>
#include <unordered_map>
#include <unordered_set>
#include <cassert>
using namespace std;

int main() {
    unordered_map< string, unordered_set<int> > znamky;
    while (true) {
        string cmd; cin >> cmd;
        if (cmd == "pridaj") {
            string x; int z; cin >> x >> z;
            if (znamky[x].count(z)) cout << "duplikat\n";
            znamky[x].insert(z);
        }
        if (cmd == "odovzdaj") {
            string x1, x2; int z; cin >> x1 >> x2 >> z;
            assert( znamky[x1].count(z) );
            if (znamky[x2].count(z)) cout << "duplikat\n";
            znamky[x1].erase(z);
            znamky[x2].insert(z);
        }
        if (cmd == "vymenuj") {
            string x; cin >> x;
            for (int i : znamky[x]) cout << i << "\n";
        }
    }
}
```

```
-----------------------------------------------------------------------

from collections import defaultdict
znamky = defaultdict(set)

while True:
    cmd = input().split()
    if cmd[0] == "pridaj":
        x, z = cmd[1:]
        if z in znamky[x]: print("duplikat")
        znamky[x].add(z)
    if cmd[0] == "odovzdaj":
        x1, x2, z = cmd[1:]
```

```
    assert z in znamky[x1]
    if z in znamky[x2]: print("duplikat")
    znamky[x1].remove(z)
    znamky[x2].add(z)
if cmd[0] == "vymenuj":
    x = cmd[1]
    print( [z for z in znamky[x]] )
```