04 razmetani podatki

January 28, 2024

Še preden smo "odkrili" drevesa in preden smo poskušali uporabljati sezname, smo imeli idejo, ki bi jo le po krivici prezrli: problem vstavljanja bi se dalo rešiti tako, da med osebami pustimo prazne stole.

Pa poskusimo na ta način.

Najprej ne spreglejmo, da nič od tega, kar smo počeli doslej, ni vezano na nize. Namesto oseb, urejenih po imenih, bi lahko v drevesa spravljali tudi številke ali karkoli drugega, za kar imamo relacijo urejenosti, tako da lahko gre "manjši" na levo in večji na desno. Tudi urejene tabele in bisekcija delujejo za kakršnekoli objekte, za katere je definirana neka relacija urejenosti.

Recimo torej, da bomo v tabelo (ali, v naši metafori, na stole) spravljali številke. Najprej, recimo 5617. Recimo, da imamo 20 stolov. Na katerega posedemo 5617? Precej odvisno od tega, kakšne bodo druge številke, ne? Če gre za številke do 10000, najbrž nekam na sredo, če za številke do milijon, nekam na začetek, če za številke do 5620, pa kam blizu konca, drži?

Za 5617 pride 5650. Jo damo par stolov na desno ali čisto na desno? Spet odvisno od obsega.

Kaj pa imena? Ali, recimo, priimki? Koliko prostora je potrebno pustiti med L in O? Ogromno, ker se neverjetno velik del slovenskih priimkov začne s črko M.

Ideja s praznimi stoli ni slaba. Vendar deluje le, če vemo, kakšen je obseg vrednosti in, še več, kakšna je porazdelitev, se pravi, katere reči so pogostejše.

0.1 Kaotično shranjevanje

Amazon ima poseben način shranjevanja reči. Imenujejo ga Chaotic storage. Uporablja ga tudi moja žena. Ideja sistema je, da vsako stvar odložiš, kjer ti pade iz rok. Razlika med Amazonovim in ženinim sistemom je v tem, da Amazon ve, kaj so odložili kam. Če veš, kam si odložil, potem je pravzaprav vseeno, kam odložiš; pomembno je, da veš, kje reč najti.

Amazon uporablja podatkovno bazo, mi bomo naredili drugačno finto. Napišimo funkcijo, ki iz podanega niza izračuna neko številko. Za zdaj je vseeno, kako (in tudi ostalo bo dokaj vseeno, le nekaj opazk bomo naredili). Naredimo, recimo, takole.

[23]:
```
def razprsi(s):
    v = 0
    for i, c in enumerate(s):
        v += i * ord(c)
    return v
```

[24]:
```
razprsi("Ana")
```

[24]: 304

[25]:
```
razprsi("Berta")
```

[25]: 1065

[26]:
```
razprsi("Cilka")
```

[26]: 1030

Vsaka znak ima (upam, da to vemo?) zaporedno številko; presledek ima številko 32, 0 je 48, 1 je 49 (in tako naprej), A je 65, B je 66 (in tako naprej), a je 97, b je 98 (in tako naprej). Funkcija `ord(c)` vrne številko, ki pripada znaku c. To številko pomnožimo z mestom znaka v nizu in vse skupaj seštejemo.

Funkcijo bi se dalo napisati krajše, vendar je ne zapletajmo brez potrebe. (Tule je to, da vsi razumemo, kaj počnemo, pomembnejše od tega, da to počnemo na najboljši možen način.)

Zdaj recimo, da imamo tabelo z desetimi polji. Ano bomo dali na četrto polje, Berto na peto in Cilko na ničto; vzamemo, preprosto, številko, ki jo naračuna funkcija `razprsi` in izračunamo ostanek po deljenju z 10. Ker imamo deset polj.

[27]:
```
class RazprsenaTabela:
    def __init__(self):
        self.velikost = 50
        self.tabela = [None] * self.velikost

    def dodaj(self, oseba):
        self.tabela[razprsi(oseba.ime) % self.velikost] = oseba

    def poisci(self, ime):
        return self.tabela[razprsi(ime) % self.velikost]

    def izpisi(self):
        for oseba in self.tabela:
            if oseba:
                print(oseba.ime)
```

Naš razred bo imel atributa `velikost` in `tabela`. Velikost bo velikost tabele; torej `self.velikost == len(self.tabela)`. Če kdo meni, da atributa `velikost` potemtakem sploh ne bi potrebovali, ima sicer prav, vendar bomo velikost potrebovali na toliko mestih, da se jo splača imeti zapisano ločeno, da ne bo potrebno vedno pisali `len(self.tabela)`. Tabela pa so naši "stoli", na katere bomo posedali osebe. V začetku pripravimo tabelo, v kateri so sami `None`, kar pomeni, da je stol prazen.

Tule smo se odločili, da bo polj 50, saj 15 oseb očitno ne bi šlo na deset stolov.

Metoda `dodaj` doda osebo. Najprej izračuna, kam sodi: pokliče funkcijo `razprsi`, izračuna ostanek po deljenju s `self.velikost` in shrani osebo na mesto s tako izračunanim indeksom.

Metoda `poisci` je podobno preprosta. Kot argument dobi ime osebe; iz imena izračuna, v katerem polju je shranjena oseba s tem imenom in pač vrne vsebino tega polja.

Metoda `izpisi` izpiše vse osebe v tabeli: gre čez polja in izpiše vsa tista, v katerih je shranjena kakšna oseba (`None` je neresničen, osebe pa so resnične).

Pripravimo si navaden seznam oseb.

[28]:
```
class Oseba:
    def __init__(self, ime, stevilka):
        self.ime = ime
        self.stevilka = stevilka

osebe = [("Ana", 356), ("Berta", 374), ("Cilka", 698), ("Dani", 781),
         ("Ema", 972), ("Fanči", 941), ("Greta", 613), ("Helga", 197),
         ("Iva", 919), ("Jana", 591), ("Klara", 62), ("Liza", 196),
         ("Micka", 718), ("Nina", 417), ("Olga", 682)]

osebe = [Oseba(ime, stevilka) for ime, stevilka in osebe]
```

Zložimo jih v tabelo.

[29]:
```
t = RazprsenaTabela()
for oseba in osebe:
    t.dodaj(oseba)
t.izpisi()
```

```
Greta
Ema
Ana
Olga
Jana
Micka
Helga
Berta
Nina
Cilka
Klara
Liza
Fanči
```

[30]:
```
t.poisci("Jana").stevilka
```

[30]:
```
591
```

Kar smo sestavili, je imenitno, saj imamo vstavljanje in iskanje v, huh, konstantnem času! To je še boljše kot logaritemski čas. Pri logaritemskem času dvakrat večje število oseb pomeni eno operacijo več, ko dodajamo oziroma shranjujemo. To smo dosegli z drevesi. Tule pa je čas vstavljanja in iskanja neodvisen od števila oseb.

Zanimivo je, da smo pri tabelah dosegli odlične čase iskanja s tem, da smo se potrudili, da so bile urejene. Drevesa so morala biti urejena in uravnotežena. Tule pa smo dosegli odlične čase tako, da ignoriramo vrstni red.

Točneje: kar smo sestavili, bi bilo imenitno, če bi delovalo. Vendar ne deluje. Kaj se zgodi, če funkcija pri dveh nizih vrne številko, ki ima enak ostanek po deljenju s 50?

Kam je izginila Dani?! V izpisu je ni. Poskusimo jo poiskati.

[31]:
```
t.poisci("Dani").ime
```

[31]:
```
'Klara'
```

Klara je povozila Dani. Zakaj?

[32]:
```
razprsi("Klara")
```

[32]:
```
1032
```

[33]:
```
razprsi("Dani")
```

[33]:
```
632
```

Kako je tule z ostankom po deljenju s 50?

Temu rečemo trk ali "kolizija" (angl. *collision*). Funkcijam, kakršna je gornja pravimo *razpršitvena* ali *zgoščevalna* funkcija, v angleščini pa tudi in predvsem (hash function). *Razpršitvena* zato, ker jo uporabljamo, da *razprši* elemente po tabeli. Zgoščevalna pa zato, ker je številk, ki jih lahko vrne, bistveno manj kot je različnih nizov, torej množico nizov *zgosti* v manjšo množico števil.

Od dobre razprševalne funkcije zahtevamo, da čimbolj razpršuje. Zelo nerodno bi bilo, če bi napisali takšno funkcijo, ki bi vračala le soda števila, saj bi to pomenilo, da bo polovica polj vedno praznih, verjetnost trkov v lihih pa bo dvakrat večja, kot bi lahko bila. Želimo si torej, da bodo vsi ostanki po deljenju s 50 enako verjetni. Ali po deljenju z 10. Ali s 100. Ali s čimerkoli že bomo delili - pri čemer pri načrtovanju funkcije ne vemo vnaprej, s čim nameravamo deliti. Če koga zanima, si lahko ogleda teorijo o tem, kako izgledajo dobre funkcije. Veliko učenega si lahko preberete tudi na strani o razpršenih tabelah. Kot si lahko predstavljate, je to pomembna reč, o kateri so veliko razmišljali.

A pri še tako dobri funkciji bomo naleteli na trke. Različnih ostankov po deljenju s 50 je točno 50, različnih nizov pa je krvavo očitno bistveno več kot toliko.

Spomnimo se seznamov. Kaj če bi osebo, ki trči v drugo osebo, pripeli pod to osebo? Ali - kot ste, upam, odkrili na vajah - nad to osebo, saj je lažje pripenjati na začetek kot na konec.

[36]:
```
class RazprsenaTabela:
    def __init__(self):
        self.velikost = 50
        self.tabela = [None] * self.velikost

    def dodaj(self, oseba):
```

```
        kam = razprsi(oseba.ime) % self.velikost
        oseba.naslednji = self.tabela[kam]
        self.tabela[kam] = oseba

    def poisci(self, ime):
        kje = razprsi(ime) % self.velikost
        v = self.tabela[kje]
        while v is not None and v.ime != ime:
            v = v.naslednji
        return v

    def izpisi(self):
        for i, oseba in enumerate(self.tabela):
            if not oseba:
                continue
            print("\n", i, end=" ")
            while oseba:
                print(oseba.ime, end=" ")
                oseba = oseba.naslednji
```

Metoda `dodaj` zdaj doda novo osebo na začetek seznama. Polje, kamor dodaja osebo, ima enako vlogo kot `prvi` v seznamih. Kot smo delali pred dvema tednoma oz. pred dvema domačima nalogama: če hočemo dodajati na začetek, rečemo `oseba.naslednji = prvi` in `prvi = oseba`. Le da imamo tule namesto `prvi` pač `self.tabela[kam]`.

Iskanje smo napisali tako, da `if` zamenjamo z `while`.

Nekoliko smo dopolnili izpis, tako da gre čez tabelo in v vsaki celici čez njen pripadajoči seznam.

[37]:
```
t = RazprsenaTabela()
for oseba in osebe:
    t.dodaj(oseba)
```

[38]:
```
dani = t.poisci("Dani")
print(dani.ime, dani.stevilka)
```

```
Dani 781
```

[39]:
```
klara = t.poisci("Klara")
print(klara.ime, klara.stevilka)
```

```
Klara 62
```

[40]:
```
t.izpisi()
```

```

 2 Greta
 3 Ema
 4 Ana
```

```
5 Olga
8 Jana
12 Micka Iva
14 Helga
15 Berta
16 Nina
30 Cilka
32 Klara Dani
40 Liza
44 Fanči
```

Odlično. Spet smo zmagali.

0.2 Kje smo s časom?

Ali pa tudi ne.

Zgoraj smo se hvalili, da je čas iskanja v tejle odlični razpršeni tabeli neodvisen od tega, koliko elementov je shranjenih v njej. Je to še res?

Najočitneje ne. Kar se tiče vstavljanja, da, tu smo hitri. Čas vstavljanja je neodvisen od tega, kako polna je tabela - izračunamo mesto in porinemo na začetek seznama.

Iskanje pa je druga pesem. Če imamo v tabeli 1000 oseb, bo od vsake od 50 celic visel seznam 20 oseb. Če imamo 2000 oseb, bo od vsake od 50 celic visel seznam 40 oseb … in tako naprej. Dvakrat več oseb, dvakrat daljši seznami.

Tole je lepa vaja za gradnjo intuicije o časovni zahtevnosti. Kar smo naredili, je petdesetkrat hitrejše od seznamov. To pomeni, da je hitrejše od seznamov? Ne. Petdesetkrat hitrejše … je za nas enako hitro. Zanima nas relativna hitrost. Dvakrat več oseb pomeni dvakrat daljše sezname in dvakrat daljši čas iskanja. Naša razpršena tabela je enako dobra kot neurejeni seznami (ker vedno dodajamo na začetek).

Vendar ne vrzimo puške v koruzo. Rešili se bomo z goljufijo.

Predpostavimo, da imamo tabelo s 50 prostori in v njej 15 elementov, tako kot zgoraj. Kot kaže izpis, to ni tako slabo: vsaka oseba je na svojem stolu, le dva para si ga delita. Čas iskanja je malenkost več kot 1. Kaj pa, če bi imeli 500 prostorov in 150 oseb. Ista reč. No, seveda imamo v obeh primerih lahko tudi smolo, recimo vseh 15 ali vseh 150 oseb na istem stolu. A slabi razporedi bodo redki in bolj, ko bodo slabi, bolj redki bodo. Pa 5000 prostorov in 1500 oseb?

Vidite, kam molim taco? Le dovolj veliko tabelo potrebujemo. Nekoliko razsipni moramo biti in vedno vzeti dovolj preveliko tabelo.

Koliko? Kako naj vnaprej vemo, koliko prostora bomo potrebovali, koliko oseb bomo dodali?

Na srečo nam tega ni potrebno vedeti. Ko dodajamo osebe, štejemo, koliko jih imamo. Ko je polne, recimo, več kot dve tretjini tabele (ko imamo v tabeli velikost 50 že 34 elementov), povečamo tabelo, recimo za faktor 2. Ob tem moramo na novo razmetati obstoječe elemente.

[41]:
```
class RazprsenaTabela:
    def __init__(self):
        self.velikost = 10
```

```
        self.elementov = 0
        self.tabela = [None] * self.velikost

    def dodaj_t(self, oseba, tabela):
        kam = razprsi(oseba.ime) % len(tabela)
        oseba.naslednji = tabela[kam]
        tabela[kam] = oseba

    def dodaj(self, oseba):
        self.dodaj_t(oseba, self.tabela)
        self.elementov += 1
        if self.elementov > 2 / 3 * self.velikost:
            self.spremeni_velikost(2 * self.velikost)

    def spremeni_velikost(self, velikost):
        nova_tabela = [None] * velikost
        for oseba in self.tabela:
            while oseba:
                naslednja = oseba.naslednji
                oseba.naslednji = None
                self.dodaj_t(oseba, nova_tabela)
                oseba = naslednja
        self.tabela = nova_tabela
        self.velikost = velikost

    def poisci(self, ime):
        kje = razprsi(ime)
        v = self.tabela[kje % self.velikost]
        while v is not None and v.ime != ime:
            v = v.naslednji
        return v

    def izpisi(self):
        for i, oseba in enumerate(self.tabela):
            if not oseba:
                continue
            print("\n", i, end=" ")
            while oseba:
                print(oseba.ime, end=" ")
                oseba = oseba.naslednji
```

Konstruktor, `__init__`, bo zdaj zapisal tudi število elementov, ki so v tabeli. V začetku ga seveda postavi na 0. Začetna velikost tabele bo borih 10 elementov.

Metoda `dodaj_t` je nekakšna pomožna metoda za dodajanje. Podamo ji osebo in tabelo (ki ni nujno `self.tabela`, pa bo dodala osebo na ustrezno mesto v podano tabelo.

Prava metoda `dodaj` pokliče `dodaj_t`, s tem da kot tabelo poda `self.tabela` - tako bo `dodaj_t`

dodala v "pravo" tabelo. Poleg tega poveča števec shranjenih elementov. Če število elementov večje od dveh tretjin velikosti tabele, pokliče `spremeni_velikost`.

`spremeni_velikost` kot argument prejme novo želeno velikost. Sestavi nova tabela (kot bi rekel moj ded, ki je bil z Dolenjske) in vanjo zloži vse osebe iz trenutne tabele, tako da za vsako pokliče `dodaj_t`. Ker mora predtem vsaki osebi nastaviti `oseba.naslednja` na `None` (ker smo že delali s seznami, si lahko predstavljate, kaj bi se zgodilo sicer), je potrebno nekaj akrobacij, da pravilno prehodimo seznam, ki ga sproti razdiramo.

Ko je premetavanje v novo tabelo opravljeno, zamenjamo "pravo" tabelo z novo (`self.tabela = nova_tabela`) in velikost z novo velikostjo (`self.velikost = velikost`).

Iskanje in izpis ostaneta takšna kot prej.

Kje smo zdaj s časom?

Poprečnih časov ne bomo računali. Tega ne znam razložiti tako preprosto, kot pri tabelah. Zadovoljili pa se bomo z intucijo: če imamo n oseb na k stolih, bo poprečno število operacij za iskanje sorazmerno $1 + n/k$. Če je velikost tabele narašča sorazmerno s številom elementov, imamo torej $n = ck$ (z neko "sorazmernostno" konstanto c, v našem primeru do $2/3$) in čas iskanja je sorazmeren $1 + (ck)/k = 1 + c$. Ker je c konstanta, je čas iskanja neodvisen od velikosti tabele.

0.3 Popolna zmaga

Kar smo uspeli narediti, je veliko boljše od dreves. Kot prvo, čas iskanja in vstavljanja je neodvisen od števila oseb. Oh, da, zamolčal sem, da je občasno potrebno premetati celo tabelo. Vendar se to zgodi dovolj redko (velikost tabele vsakič podvojimo, torej bo pogostost premetavanja padala s številom elementov. V poprečju bo vsak element premetan natančno enkrat, torej je čas premetavanja neodvisen od števila elementov), tako da lahko premetavanje zanemarimo.

Kot drugo: vse skupaj je bistveno bistveno preprostejše od dreves. Ob drevesih smo preživeli dvoje predavanj, izumili smo dva leva in dva desna obrata (še dunajski valček jih nima toliko), napisali dva zaslona nedelujoče kode za obračanje. Brisanje iz drevesa sem napisal iz gole trme, predaval pa ga nisem iz gole milosti do študentov. (Tole velja za prvo leto izvajanja predmeta. Drugo leto sem brisanje tiščal v podzavest.)

Brisanje iz razpršenih tabel? Trivialno. Pogledaš kje je in pobrišeš. Poiskati ga znamo, brisanja iz seznamov pa se tudi ne bojimo.

Še nečesa ne prezrimo: v razpršene tabele lahko shranjujemo tudi stvari, ki jih ne moremo urediti po velikosti. Če kdo misli, da bi lahko v drevesa shranjevali kompleksna števila, naj pove, da ga zatožim matematikom.

Seveda ni nič zastonj.

Urejenosti sicer res ne potrebujemo, kar je koristno. Istočasno pa smo izgubili vrstni red. V tabelah (oziroma seznamih v Pythonovskem pomenu besede) imamo, če uporabljamo le `append`, ohranjen vrstni red dodajanja. Drevesa urejajo po velikosti (ali kakršnikoli že relaciji urejenosti). Razpršene tabele nimajo nobenega reda. Vendar - komu mar!

Drugo, kar izgubljamo - in za kar nam je mar - je prostor. Razpršene tabele so potratne. Če hočemo, da se res vedejo, kakor obljubljamo zgoraj, morajo biti za določen faktor večje od števila shranjenih elementov in več ko je shranjenih elementov, več prostora vržemo stran. Stran vrženi

prostor je seveda linearno sorazmeren številu shranjenih elementov, tako da to lahko preživimo. Če bi bil večji, ne bi šlo.

V resnici so razpršene tabele najboljše, kar imamo. Posebej danes, ko pomnilnik ni več tako dragocena dobrina kot leta sedemdeset.

0.3.1 Python

Pythonovi slovarji so v resnici razpršene tabele. Razpršitveno funkcijo izračunajo, očitno, iz ključa. V celico shranijo ključ, vrednost, poleg tega pa tudi tisto, kar je vrnila razpršitvena funkcija. Na ta način je ni potrebno vedno znova klicati. Tudi inspiracija za faktor 2/3 prihaja iz Pythona. Določili so ga, tako kot druge podobne parametre, tako da so poskušali, kaj se najboljše vede.

Kadar pride do trka, Python ne sestavlja seznamov, temveč shrani element na neko drugo, "rezervno" mesto. Če je zasedeno tudi to, uporabi drugo rezervo. Prav tako pri iskanju preišče še rezerve. Reč je podobna seznamom in tudi časovna zahtevnost je (teoretično) enaka. Takšen način uporablja, ker je iz tehničnih razlogov bolj praktičen. Tule pa smo uporabili sezname, ker je lažje in ker so nam bližji.

V ozadju slovarjev seveda niso Pythonovi seznami. "Stoli" so narejeni bolj "direktno": vrednost, ki jo vrne razpršitvena funkcija, se preslika v lokacijo v računalnikovem pomnilniku. A bistvo zdaj razumete.

Razpršitveno funkcijo lahko pokličemo tudi sami. Imenuje se `hash`.

[44]:
```
hash("Ana")
```

[44]:
```
2575919564229461234
```

[45]:
```
hash("Cilka")
```

[45]:
```
-5694556544823303295
```

[46]:
```
hash(123)
```

[46]:
```
123
```

[47]:
```
hash(12345)
```

[47]:
```
12345
```

[48]:
```
hash((1, 9, 3))
```

[48]:
```
2528494145600664752
```

[49]:
```
hash([1, 2])
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-49-0e7c08465b16> in <module>()
```

```
----> 1 hash([1, 2])

TypeError: unhashable type: 'list'
```

Iz seznama (`list`) je noče izračunati, ker bi se seznam lahko spremenil. Če spremenimo vrednost ključa, pa le-ta ni več na pravem mestu v tabeli in slovar ne bo več pravilno deloval.

Pred leti sem na lastni koži izkusil, koliko so razpršitvene tabele hitrejše od dreves. Delal sem nek tečaj iz kriptografije. Med drugim smo razbijali neko zaščito, za kar si je bilo potrebno zapomniti precej stvari. Shranjevali smo jih v slovarjih. Nekateri so programirali v C++, drugi v Pythonu. C++ je načelno, uh, stokrat ali stopetdesetkrat hitrejši od Pythona. Vendar smo tisti, ki smo delali v Pythonu, dobili rešitev v nekaj minutah, tisti, ki so delali v C++ pa so čakali ure. Zakaj? Slovarji v C++ so navadno narejeni z drevesi (tam sem jim v resnici ne reče slovar, a to ni pomembno), v Pythonu pa z razpršitvenimi tabelami. Razpršitvene tabele so toliko hitrejše od dreves, da je bil stokrat počasnejši Python zgolj zaradi boljše podatkovne strukture stokrat hitrejši od C++.

Python uporablja slovarje povsod. Spremenljivke shranjuje v slovarju, katerega ključi so imena spremenljivk, vrednosti pač vrednosti. Vsebina modula je shranjena v slovarju. Atributi in metode razredov in objektov so shranjene v slovarjih. Skoraj vse v Pythonu je slovar, ta in ona reč je terka. Argumenti funkcije, recimo, so shranjene v terkah in slovarjih.

Kako so narejene množice v Pythonu? Kot slovar, ki shranjuje le ključe. Zakaj so tako hitre? Pri Programiranju 1 sem rekel, da Python nekako "ve" ali množica vsebuje določen element, ne da bi jo moral vso prebrskati. No, zdaj veste: Python za vsak element izračuna vrednost razpršitvene funkcije in pogleda, ali je tam, kjer razpršitvena funkcija pravi, da bi moral biti.