

1. OBLAST-Analiza algoritama

*Algoritam je neki standardizovani postupak, niz koraka ali da bude i struktuiran

*Cilj algoritma je da se neki zadatak obavi

? Prednosti koriscenja gotovog algoritma u odnosu na izmisljanje naseg algoritma?

Vreme se stedi, postojacem algoritmima je ispravnost matematicki dokazana, ne moramo da se brinemo da li cemo doci do resenja ili nećemo, optimalni su, postoji granica u kom vremenu sa kojom memorijskom zavisnoscu ce doci do rezultata, smanjujemo mentalno opterecenje, garancija kvaliteta rezultata

*algoritam će od nekog ulaza proizvesti neki izlaz ULAZ –ALGORITAM – IZLAZ

? Da li je algoritam funkcija, kako bi uporedili?

U zavisnosti od osobe do osobe ona funkciju drugacije dozivljava (matemacka, programeri kao funkcija u programu..). Kod algoritama je fokus samo na postupku, dok je kod funkcije akcenat na samoj transformaciji ulaz,izlaz, ali kada gledamo da teorijske strane nema tu mnogo razlike drastične

VREME IZVRŠAVANJA

*Većina algoritama transformiše objekte na ulazu u objekte na izlazu. Vreme izvršavanja algoritma obično raste sa veličinom ulaza. Teško je izračunati prosečno vreme izvršavanja, posmatračemo najgori slučaj - jednostavnije za analizu

-ključno za primene kao što su igre, finansije ili robotika

*Jedan nacin da procenimo sta se desava sa nasim algoritmom jeste da ga pokrenemo vise puta, dobicemo razlicita vremena izvršavanja, za razlicite ulaze, razlicito vreme izvršavanja. Upitno pitanje je srednja vrednost izvršavanja.

EKSPERIMENTALNO PROUČAVANJE ALGORITAMA

1. napisati program koji implementira posmatrani algoritam

2. Pokrenuti program za različite veličine i strukturu ulaznih podataka

3. Meriti vreme izvršavanja (modul time())

4. Analizirati rezultate (grafik, da izvučemo neku f-ju koja ce biti približna, greške prisutne, postavlja se pitanje koju mi funkciju možemo da analiziramo (da li je to linearna,kvadratna..)

OGRANIČENJA EKSP. PRISTUPA

*Potrebno je implementirati algoritam — može biti teško

* teško je predvideti rezultate za ulaze koji nisu obuhvaćeni eksperimentom

*za poređenje dva algoritma mora se koristiti identično hardversko i softversko okruženje

TEORIJSKI PRISTUP

*Koristi se opis algoritma visokog nivoa umesto implementacije

*opisuje vreme izvršavanja kao funkciju veličine ulaza — n

*uzima u obzir sve moguće ulaze

*omogućava procenu brzine algoritma nezavisno od korišćenog hardvera ili softvera

PSEUDOKOD

*opis algoritma visokog nivoa

* bolje strukturiran od prirodnog jezika

*manje detalja nego u stvarnom programu

*sakriva detalje vezane za dizajn programa

* poželjna notacija za opisivanje algoritama

7 VAŽNIH F-JA

1. Konstantna f-ja

-osnovna funkcija $(n) = 1$

-svaka druga može se prikazati kao $(n) = c \cdot (n)$

-može da opiše broj koraka potrebnih za neku od osnovnih operacija npr. sabiranje, dodela vrednosti, poređenje

2. Logaritamska f-ja

-za $b > 1$ za osnovu logaritma se najčešće koristi 2

-2 se podrazumeva, tj. $\log n = \log_2 n$

-podela problema na dva dela: čest princip koji se koristi u algoritmima

3. Linearna f-ja

-kada treba obaviti prostu operaciju nad svakim od n elemenata ulaza , npr. poređenje broja sa svim elementima niza

4. N-tog n funkcija

-raste nešto brže od linearne funkcije i znatno sporije od kvadratne funkcije npr. najbrže sortiranje n brojeva zahteva $n \log n$ vreme

5. Kvadratna funkcija

-npr. dve ugnježdene petlje gde unutrašnja obavlja linearan broj operacija nad elementima ulaza a spoljna se izvršava linearan broj puta su proporcionalne sa n na 2

6. Kubna funkcija

-ređe se javlja od kvadratne, ali predstavlja jednu klasu polinomijalnih funkcija

7. Eksponencijalna funkcija $(n) = b^n$ na n

- b je baza , n je eksponent, često je $b = 2$, najsporija

PRIMITIVNE OPERACIJA

- *osnovne operacije koje izvršava algoritam
- *prikazane u pseudokodu
- *nezavisne od programskog jezika
- *troše konstantnu količinu vremena
- *na primer: izračunavanje izraza, dodela vrednosti promenljivoj, pristup elementu niza preko indeksa, poziv funkcije, vraćanje rezultata
- *analizom pseudokoda možemo odrediti maksimalan broj primitivnih operacija koje izvršava algoritam kao funkciju veličine ulaza

PROCENA VREMENA IZVRŠAVANJA

-npr algoritam se izvršava $8(n-1)$

-neka je

a : vreme izvršavanja najbrže primitivne operacije

b : vreme izvršavanja najsporije primitivne operacije

$T(n)$ vreme u najgorem slučaju

- tada je

$$(8n - 2) \leq T(n) \leq (8n - 2)$$

$\Rightarrow T(n)$ je ograničena sa dve linearne funkcije!

PORAST VREMENA IZVRŠAVANJA

- izmena u hardverskom ili softverskom okruženju menja $T(n)$ za konstantan faktor ali ne menja brzinu rasta $T(n)$
- linearni porast vremena izvršavanja $T(n)$ je suštinska osobina algoritma arrayMax $(8(n-1))$

KONSTANTNI ČINIOCI

-porast vremena izvršavanja ne zavisi od konstantnih činilaca i izraza nižeg reda

VELIKO O NOTACIJA (Big-Oh)

-Opisuje granično ponašanje funkcije kada argument raste

-za date (n) i (n) kažemo da $f(n)$ je $O(g(n))$ ako postoje pozitivne konstante c i n_0 takve da

$$f(n) \leq c \cdot g(n) \text{ za } n \geq n_0$$

-primer: $2n + 10$ je (n)

$$2n + 10 \leq cn$$

$$(c - 2)n \geq 10$$

$$n \geq 10/(c - 2)$$

izaberemo $c = 3$ i $n_0 = 10$

*primer: n na 2 nije (n)

$$n \text{ na } 2 \leq cn$$

$$n \leq c$$

ova nejednakost ne može biti zadovoljena jer je c konstanta

*primer

$$3 \log n + 5 \text{ je } (\log n)$$

tražimo $c > 0$ i $n_0 \geq 1$ takve da $3 \log n + 5 \leq c \log n$ za $n \geq n_0$

ovo je zadovoljeno za $c = 8$ i $n_0 = 2$

~Veliko O I porast vremena

*veliko O definiše gornju granicu na rast funkcije

*tvrdnja (n) je ($g(n)$) znači da $f(n)$ ne raste brže od $g(n)$

*možemo da koristimo veliko O da rangiramo funkcije po brzini rasta

~Još neka pravila.. sve niže operacije zanemarujemo!

*ako je (n) polinom stepena d tada (n) je $O(nd)$, tj.

-možemo zanemariti niže stepene polinoma

-možemo zanemariti konstantne koeficijente

*koristimo najsporiju moguću klasu funkcija

- kažemo „ $2n$ je $O(n)$ “ umesto „ $2n$ je $O(n^2)$ “

*koristimo najjednostavniji izraz koji predstavlja klasu

-kažemo „ $3n + 5$ je (n)“ umesto „ $3n + 5$ je $O(3n)$ “ Analiza algoritama

ASIMPTOTSKA ANALIZA ALGORITAMA

*asimptotska analiza algoritama određuje vreme izvršavanja u „veliko O“ notaciji

*kako obaviti asimptotsku analizu

-odredimo broj primitivnih operacija u najgorem slučaju kao funkciju veličine ulaza

-izrazimo funkciju u „veliko O“ notaciji

*primer:

-ustanovimo da arrayMax izvršava najviše $8n - 2$ primitivnih operacija

-kažemo da je složenost arrayMax algoritma (n)

*konstantne činioce i izraze nižeg stepena svakako ne iskazujemo na kraju, pa ih možemo zanemariti kada brojimo primitivne operacije

RAČUNANJE PROSEKA PREFIKSA

* i -ti prosek prefiksa niza X je prosek vrednosti prvih $i + 1$ elemenata

$X[i] = ([0] + X[1] + \dots + X[i]) / (i + 1)$, niz A se koristi u finansijskoj analizi

*obratiti pažnju da različite implementacije sličnog problema mogu biti različite složenosti

ROĐACI VELIKOG O –retko u praksi

veliko O :

(n) je ($g(n)$) ako je $f(n)$ asimptotski manje ili jednako $g(n)$ -ograničava sa gornje strane

veliko Ω :

(n) je $\Omega(g(n))$ ako je $f(n)$ asimptotski veće ili jednako $g(n)$ -ograničava sa donje strane

veliko Θ :

(n) je $\Theta(g(n))$ ako je $f(n)$ asimptotski jednako $g(n)$

?Ako se bubble sort za 1000el izvršio za 0.3s, za 10 000 el za 22s, koliko će se izvršiti za 100 000 s obzirom da ima kvadratnu složenost?

Još 100 puta više od 30s, sa porastom broja el, potrebno je više vremena da se algoritam izvrši.

2.OBLAST – Rekurzija

*rekurzija: kada funkcija poziva samu sebe

*klasičan primer: faktorijel

```
def fact(n):
```

```
    if n == 0:
```

```
        return 1
```

```
    else:
```

```
        return n * fact(n-1)
```

SADRŽAJ REKURZIVNE FUNKCIJE

1.osnovni slučajevi

- vrednosti ulaznih promenljivih za koje ne pravimo rekurzivne pozive

-mora postojati bar jedan

2. rekurzivni pozivi

-poziv iste funkcije

- svaki rekurzivni poziv bi trebalo definisati tako da predstavlja napredovanje prema osnovnom slučaju (da ne bi došlo do beskonačne rekurzije)

VIZUELIZACIJA REKURZIJE

*trag rekurzije (slika slajd 6)

- pravougaonik za svaki rekurzivni poziv
- strelica od pozivača ka pozvanom
- strelica od pozvanog ka pozivaču sa rezultatom koji se vraća

*primer rekurzije: engleski lenjir (donji deo lenjira u inch, ideja da krenemo od najduže linije, pa malo kraće itd..)

*crtanje engleskog lenjira –

-ulaz: dužina crtice

-izlaz: lenjir sa crticom date dužine u sredini i manji lenjiri sa leve i desne strane

-Fokusira se samo na jedan podeok između dva cela broja npr 0 i 1, i neće se eksplicitno baviti iscrtavanjem 0 i 1 nego samo segmenta između dva podeoka, ulaz -dužina srednje crtice, ako je dužina veća od 0, iscrtaj levu polovinu lenjira, i onda ponavljamo i za desni segment, levi i desni podeok isto izgledaju pa ih iscrtavamo na isti način

-drawTicks(length)

-BINARNA REKURZIJA – rekurzija u kojoj imamo dva rekurzivna poziva

-BINARNA PRETRAGA- efikasniji način traženja elemenata u kolekciji, kolekcija mora biti sortirana, ona traži element tako što pristupi srednjem elementu kolekcije, npr ako je srednji element 11, a mi tražimo 7, odbacujemo polovinu koja je veća od 11, onda ostatak polovine opet polovimo, onda da li se element nalazi sa leve ili desne strane tog el itd..

*radi u $(\log n)$ vremenu

LINEARNA REKURZIJA

*jedan rekurzivni poziv

*primer- suma prvih n prirodnih br u A

*testiranje baznih slučajeva

-početi sa testiranjem baznih slučajeva (mora biti bar jedan)

- obrada baznog slučaja ne sme koristiti rekurziju

- svaki mogući lanac rekurzivnih poziva mora se završiti dolaskom do baznog slučaja

*rekurzivno jednom

-napraviti jedan rekurzivni poziv

-možemo napraviti grananje sa odlukom da se izabere jedan od mogućih rekurzivnih poziva --svaki mogući rekurzivni poziv treba da se približi baznom slučaju

*problematočno kada predjemo na implementaciju – python ima ograničenje pozivanja rekurzivnih poziva (1000)

DEFINISANJE ELEMENATA ZA REKURZIJU

*prilikom dizajniranja rekurzivnih funkcija važno je definisati ih tako da je rekurzija jednostavna
*ponekad to znači da treba definisati dodatne parametre funkcije

*na primer, definisali smo $\text{ReverseArray}(A, i, j)$ umesto $\text{ReverseArray}(A)$ Rekurzija

REPNA REKURZIJA

-kada je rekurzivni poziv poslednji korak u funkciji

-lako se preradi u iterativni postupak

-rekurziju mozemo da ne volimo zato sto svaki poziv zauzima mnogo memorije, ogranicenje broja poziva i zbog toga nam je često drago da rekurziju konvertujemo u iterativni postupak

-O notacija ne vidi razliku izmedju ove dve iteracije, a u realnosti ova razlika itekako postoji

*Postoje algoritmi koji se mogu implementirati i pomoću binarne i linearne rekurzije

*Primer- fibonačijevi brojevi:

1. rekurzivno

-tako što kažemo da je n-ti fibonačijev broj zapravo zbir n-1 i n-2 fibonacijevog broja

-nije baš najsajnije jer imamo čitava podstabla koja se ponavljaju

-broj poziva raste eksponencijalno

2. linearne rekurzije

-evidentiramo dva uzastopna fibonačijeva broja, ima samo k-1 rekurzivnih poziva

VIŠESTRUKA REKURZIJA

*primer problema: zagonetke sabiranja

$pot + pan = bib$

$dog + cat = pig$

$boy + girl = baby$

*višestruka rekurzija potencijalno pravi puno rekurzivnih poziva , ne samo jedan ili dva

3. OBLAST - Objektno-orijentisano programiranje i Python

*svaki objekat koji se kreira u programu je instanca nečega što zovemo klasa

* klasa spoljašnjem svetu predstavlja pogled na objekte koji su njene instance

* bez nepotrebnih detalja ili davanja pristupa unutrašnjosti

*klasa sadrži attribute (instance variables, data members) i metode (member functions) koje objekat može da izvrši

CILJEVI

*robustnost -želimo da softer može da prihvati neočekivane ulazne podatke koji nisu ranije bili predviđeni

*adaptivnost- želimo da softer može da evoluira tokom vremena kao odgovor na promene u zahtevima ili okruženju

* ponovna iskoristivost (reusability) -želimo da omogućimo da se isti programski kôd koristi kao komponenta u različitim sistemima ili primenama (funkcije, moduli)

APSTRAKтни TIPOVI PODATAKA

*apstrakcija predstavlja izdvajanje najvažnijih osobina nekog sistema (odstraniti nebitne osobine)

*primena apstrakcije na dizajn struktura podataka dovodi do apstraktnih tipova podataka (ATP)

*ATP je model strukture podataka koji definiše tip podataka, operacije nad njima, i tipove parametara tih operacija

* ATP definiše šta operacija radi, ali ne i kako to radi

* skup operacija koje definiše ATP je interfejs (public interface)

*primer : lista -znamo da lista čuva neke podatke, pristup po indeksu, dodavanje elementa, brisanje itd, to su operacije koje mi znamo, međutim kako je pristup elementu interno implementiran u okviru liste koje smo koristili

PRINCIPI OO DIZAJNA

1.modularnost-podela na logičke celine

2.apstrakcija-ocenjivanje nepotrebnog, fokus na važne stvari

3.enkapsulacija-skrivanje nepotrebnih detalja

DUCK TYPING

*program tretira objekte kao da imaju određenu funkcionalnost ako se ponašaju ispravno i ispunjavaju traženo

*Python je interpretirani jezik sa dinamičkim tipovima

-nema compile-time provere tipova podataka, provere se izvršavaju tek u toku izvršavanja

- nema posebnih formalnih zahteva kod definisanja novih tipova podataka

APSTRAKTNE BAZNE KLASE

*Python radi sa ATP pomoću mehanizma apstraktnih baznih klasa (abstract base classes, ABC)

* ABC se ne može instancirati, ali definiše zajedničke metode koje sve implementacije te apstrakcije moraju imati

- *ABC se realizuje pomoću jedne ili više konkretnih klasa koje nasleđuju ABC i implementiraju metode koje propisuje ABC

- *možemo koristiti postojeće ABC i postojeće konkretne klase iz Python-ove biblioteke

ENKAPSULACIJA

- *komponente softverskog sistema ne bi trebalo da otkrivaju detalje svog unutrašnjeg funkcionisanja
 - *neki aspekti strukture podataka su javni a neki predstavljaju interne detalje i privatni su

- * Python delimično podržava enkapsulaciju konvencija: atributi i metode koje počinju donjom crtom (npr. `_secret`) su privatni i ne treba ih koristiti izvan klase

OO DIZAJN SOFTVERA

- *odgovornost: podeliti posao različitim učesnicima, svako sa različitim odgovornostima

- *nezavisnost: definisati namenu svake klase što je moguće više nezavisno od drugih klasa

- * ponašanje: definisati ponašanje svake klase pažljivo i precizno, tako da posledice svake akcije budu dobro shvaćene od strane drugih klasa

OBJEDINJENI JEZIK ZA MODELOVANJE (UML)

- *Unified Modeling Language (UML): (grafički) jezik za opis softverskih sistema

- * prikaz klase na UML dijagramu ima tri celine: ime klase, attribute, metode

DEFINICIJE KLASA

- *klasa je osnovno sredstvo apstrakcije u OOP

- *u Pythonu je svaki podatak predstavljen instancom neke klase

- * klasa definiše ponašanje pomoću metoda; sve instance imaju iste metode

- *klasa definiše stanje pomoću atributa; svaka instanca ima svoju kopiju atributa

IDENTIFIKATOR SELF

- *svaka klasa može imati više svojih instanci

- *svaka instanca ima svoj primerak atributa

- *stanje svake instance predstavljeno je vrednošću njenih atributa

- * `self` predstavlja instancu za koju je metoda pozvana

KONSTRUKTORI

- *kreiranje instanci klase `CreditCard`:

```
cc = CreditCard('Žika Žikić', 'ABC Banka', '5931 0375 9837 5309', 1000)
```

interno će se ovo prevesti na poziv metode `__init__`

- *njen zadatak je da novokreirani objekat dovede u korektno početno stanje postavljanjem odgovarajućih vrednosti atributa

POKLAPANJE OPERATORA

- *isti operator može da se koristi za različite tipove podataka, a da mu se u tom kontekstu dodeljuju različita značenja
- *na primer, izraz $a + b$ predstavlja sabiranje kod brojčanih podataka, a konkatenciju kod stringova i lista
- *kada pišemo svoju klasu možemo da definišemo operator $+$ za instance naše klase

ITERATORI

- *iterator za bilo kakvu kolekciju podataka omogućava da se svaki element kolekcije dobije tačno jednom
- *potrebno je napisati metodu `__next__` koja vraća sledeći element kolekcije
- *ili izaziva izuzetak `StopIteration` ako nema više elemenata
- *umesto `__next__` mogu se napraviti `__len__` i `__getitem__`

NASLEDJIVANJE

- ***nasleđivanje** je mehanizam za modularnu i hijerarhijsku organizaciju
- * omogućava da se nova klasa definiše pomoću postojeće kao početne tačke
- * postojeća klasa se obično zove **bazna, roditeljska** ili **superklasa**
- * nova klasa se obično zove **potklasa, dete**-klasa ili **naslednik**
- * postoji dva načina da se potklasa učini različitom od roditelja
 - potklasa može da promeni ponašanje tako što će imati novu implementaciju neke nasleđene (postojeće) metode
 - potklasa može da proširi roditelja dodavanjem novih metoda ili atributa
- *primer - numerička progresija je niz brojeva kod koga vrednost svakog elementa zavisi od vrednosti jednog ili više prethodnih elemenata
 - aritmetička progresija određuje sledeći broj dodavanjem fiksne konstante na prethodni broj - geometrijska progresija određuje sledeći broj množenjem prethodnog broja fiksnom konstantom
 - Fibonačijeva progresija koristi formulu $F_{i+1} = F_i + F_{i-1}$

4. OBLAST – Nizovi

- *Python ima ugrađene tipove list, tuple i str (sekvence)
- *svaki od ovih tipova omogućava pristup elementima po indeksu, npr. $A[i]$
- *svaki od ovih tipova interno koristi niz za skladištenje podataka
- *niz je skup susednih memorijskih lokacija koje mogu biti adresirane pomoću sukcesivnih indeksa koji počinju od 0

*niz može da čuva primitivne elemente, na primer karaktere, predstavljajući **kompaktni niz**

*niz može čuvati i reference na objekte

? Ako bismo imali niz brojeva, da li bi to u Pythonu bio kompaktni niz?

Ne, izgledao bi kao niz koji može čuvati reference na objekte

KOMPAKTNI NIZOVI

*podrška za rad sa kompaktnim nizovima nalazi se u modulu array

*ovaj modul definiše klasu array koja predstavlja kompaktni niz za primitivne tipove podataka

*konstruktor za array kao prvi parametar očekuje slovo koje označava tip elemenata

```
primes = array('i', [2, 3, 5, 7, 11, 13, 17, 19])
```

UBACIVANJE ELEMENATA

*u operaciji `add(i, o)` treba napraviti mesta za novi element pomeranjem $n - i$ elemenata $A[i], \dots, A[n - 1]$ u desno za jedno mesto (treba dodati novi element o na poziciji i)

*mi unapred zauzimamo odredjen broj memorijskih lokacija, koje postepeno popunjavamo, na početku su prazne

* u najgorem slučaju ($i = 0$) za ovo je potrebno $O(n)$ vreme

UKLANJANJE ELEMENATA

*u operaciji `remove(i)` treba popuniti rupu na mestu elementa koji se uklanja pomeranjem $n - i - 1$ elemenata $A[i + 1], \dots, A[n - 1]$ u levo za jedno mesto

*u najgorem slučaju ($i = 0$) za ovo je potrebno $O(n)$ vreme

PERFORMANSE NIZA

*za implementaciju liste pomoću niza

- prostor koji zauzima struktura u memoriji je $O(n)$

- pristup i -tom elementu je u $O(1)$ vremenu

- ubacivanje i uklanjanje su u $O(n)$ vremenu u najgorem slučaju

?šta je zaista najgori slučaj kod dodavanja?

- niz popunjen do kraja

- zauzmemo novi (veći) niz u memoriji

- prepíšemo sve podatke iz starog niza

- odbacimo stari niz

*moramo unapred znati veličinu niza!

STRATEGIJE ZA PROŠIRENJE NIZA

? koliko velik treba da bude novi niz prilikom proširenja?

-inkrementalna strategija: novi niz će biti duži za neko konstantno c

-strategija dupliranja: novi niz će biti duplo duži od prethodnog

POREĐENJE STRATEGIJA

*poredimo strategije analizirajući ukupno vreme $T(n)$ potrebno za obavljanje n operacija ubacivanja
*krećemo od niza dužine 1

* amortizovano vreme add operacije: prosečno vreme potrebno za operaciju za niz od n operacija,
 $T(n)/n$

Poređenje strategija: inkrementalna

*pravimo novi niz $k = n/c$ puta

* ukupno vreme $T(n)$ za seriju od n operacija ubacivanja je proporcionalno sa:

$$n + c + 2c + 3c + 4c + \dots + kc =$$

$$n + c(1 + 2 + 3 + \dots + k) =$$

$$n + ck(k + 1)/2$$

*c je konstanta, sledi da $T(n)$ je $O(n + k^2)$ odnosno $O(n^2)$

* \Rightarrow amortizovano vreme operacije ubacivanja je $O(n)$

Poređenje strategija: dupliranje

*pravimo novi niz $k = \log_2 n$ puta

* ukupno vreme $T(n)$ za seriju od n operacija ubacivanja je proporcionalno sa:

$$n + 1 + 2 + 4 + 8 + \dots + 2^k =$$

$$n + 2^{k+1} - 1 = n + 2 * 2^k - 1 =$$

$$n + 2 * 2^{\log_2 n} - 1 = n + 2n - 1 =$$

$$3n - 1$$

* $T(n)$ je $O(n)$

* \Rightarrow amortizovano vreme operacije ubacivanja je $O(1)$

5. OBLAST – Stekovi

APSTRAKTNI TIPOVI PODATAKA

_ATP je apstrakcija strukture podataka

*ATP definiše

- podatke koji se čuvaju
- operacije nad podacima
- uslovi kada dolazi do greške

* primer: ADT koji modeluje sistem za trgovinu akcijama

- podaci: kupi/prodaj narudžbe
- operacije:
 - order buy(stock, shares, price)
 - order sell(stock, shares, price)
 - void cancel(order)
- greške:
 - nepostojeće akcije
 - otkazivanje narudžbe

STEK ATP

*stek (stack) ATP čuva proizvoljne objekte

*ubacivanje i uklanjanje poštuje LIFO (last-in-first-out) princip (poslednji element je prvi element koji skidamo)

*primer: PEZ bombone :)

*glavne operacije: push(object): dodaje element na vrh steka

object pop(): skida element sa vrha steka

* dodatne operacije: object top(): vraća najviši element bez skidanja

integer len(): vraća broj elemenata na steku

boolean is_empty(): vraća True ako je stek prazan

PRIMENE STEKA

*neposredne primene:

- istorija poseta stranicama u web čitaču
- undo sekvenca u tekst editor
- lanac rekurzivnih poziva u programu

*indirektne primene:

- pomoćna struktura podataka za mnoge algoritme

- komponenta u okviru drugih struktura podataka

STEK POMOĆU NIZA

*dodajemo elemente s leva u desno

*posebna promenljiva čuva indeks poslednjeg elementa(zbog dodavanja novog elementa, uklanjanja poslednjeg, pristupanju poslednjeg elementa)

*ako se niz popuni... nova operacija push mora da proširi niz i iskopira sve elemente

PERFORMANSE STEKA

*neka je n broj elemenata na steku

*prostor u memoriji je (n)

*svaka operacija je (1) (amortizovano za push)

*primeri – uparivanje zagrada, uparivanje html tagova, izračunavanje aritmetičkih izraza(prioritet operatora), izračunavanje raspona

*LINEARNI ALGORITAM - svaki indeks u nizu... ..je stavljen na stek tačno jednom ...je skinut sa steka najviše jednom naredbe u while petlji su izvršene najviše n puta algoritam radi u (n) vremenu

6. OBLAST – Redovi

RED ATP

*red (**queue**) ATP čuva proizvoljne objekte

*ubacivanje i uklanjanje poštuje FIFO (first-in-first-out) princip (prvi element koji smo stavili je prvi element I koji uklanjamo)

*ubacivanje se vrši na kraju reda, a uklanjanje na početku reda

*glavne operacije:

-**enqueue(object)**: dodaje element na kraj reda

- object **dequeue()**: uklanja element sa početka reda

*dodatne operacije:

-object **first()**: vraća element sa početka bez uklanjanja

-integer **len()**: vraća broj elemenata u redu

-boolean **is_empty()**: vraća True ako je red prazan

* greške:

-poziv **dequeue** ili **first** za prazan red

*svi elementi imaju isti prioritet

PRIMENE REDOVA

*neposredne primene

- liste čekanja, birokratija

- pristup deljenim resursima (npr. štampač)

-deljenje procesora među paralelnim procesima

* indirektne primene:

-pomoćna struktura podataka za mnoge algoritme

- komponenta u okviru drugih struktura podataka

RED POMOĆU NIZA

*implementiraćemo stek pomoću niza dužine N

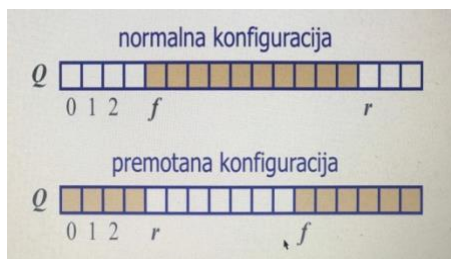
* koristimo ga cirkularno

*posebne promenljive čuvaju indekse prvog i poslednjeg elementa

f – indeks prvog elementa

r – indeks prvog elementa iza poslednjeg

* element niza sa indeksom r je uvek prazan!



Treba da primetimo da problem kod redova nastaje kad mi dodajemo neke elemente pa onda uklonimo, i tu gde smo uklonili elemente ostaju nam nepopunjena mesta, i ako bismo nastavili da dodajemo elemente na kraj, i uklanjamo s početka, doći ćemo u situaciju da mi zapravo imamo prazne memorijske lokacije ali pošto dodajemo samo na kraj ne možemo da ih iskoristimo. Da bi se ovaj problem rešio, počinjemo cirkularno da smeštamo elemente, dodajemo elemente na kraj, ako ne stane memorijskih lokacija mi ćemo smatrati da kao 0 sledi posle poslednje memorijske lokacije i novi element će se na početku ubaciti itd

*koristićemo moduo operator (ostatak pri deljenju) –za uvećavanje da ne bi izašao iz opsega

*formula za dužinu niza $r-f$ za normalnu konf

*formula za dužinu niza $N-f+r$ za premotanu konf

*zajednička formula $(r-f)*N\%N$

*operacija enqueue izaziva izuzetak ako je niz pun

*operacija dequeue izaziva izuzetak ako je red prazan

IMPLEMENTACIJA REDA U PYTHONU

*imaćemo tri atributa:

_data: lista sa fiksnim kapacitetom

_size: broj elemenata u redu

_front: indeks prvog elementa u redu

*prilikom skidanja elementa sa reda, imamo par korekcija, dužina da se smanji za 1, indeks prvog elementa uveća po modulu N zauzetih memorijskih lokacija, ta pozicija se označi kao prazna tako što stavimo None

*prilikom dodavanja elementa na red, imamo par korekcija, proveravamo da li ima mesta:

- ako nema, pozivamo funkciju resize, gde će se zauzeti duplo više memorijskih lokacija, stari elementi će se prepisati

- ako ima, pronalazimo indeks onaj gde bi trebao da bude taj dodati element (na indeks prvog elementa, dodamo dužinu i pazimo da ne izađemo iz opsega), ako smo utvrdili, novi element se dodaju na to mesto, dužina se uvećava za 1

*resize – zauzima se više memorijskih lokacija, stavljaju se na None, stare se prepisuju u nove memorijske lokacije

ROUND ROBIN RASPOREĐIVANJE

- obrada zahteva u krug, element se skida sa reda, obrađuju se, vraća se na red, obavlja se sa jednakim prioritetom

7. OBLAST – Liste

*ponavljanje: Dobre strane nizova - pristup direktan

Loše strane nizova - premeštanje indeksa, dupliranje sadržaja i zauzimanje više memorijskih lokacija

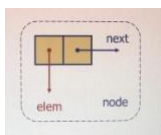
JEDNOSTRUKO SPREGNUTA LISTA

*predstavlja sekvencu elemenata

*elementi su sadržani u „čvorovima“ liste (nodes)

*susedstvo između elemenata se opisuje vezama/referencama/pokazivačima

*svaki čvor sadrži: podatak koji se čuva i link prema sledećem čvoru



*čvorovi ne zauzimaju susedne memorijske lokacije – mogu biti „razbacani“ po memoriji

*redosled se održava pomoću veza između čvorova

*svaki čvor ima vezu prema sledećem

*koji je prvi? potrebna nam je posebna referenca na prvi element liste („glava“)

*na koga pokazuje poslednji element? njegova referenca na sledećeg je None

*Iterator: obilazak svih elemenata liste

*Šta znači iterirati kroz listu? Pa trebalo bi da vratimo te čvorove na koje nailazimo.

POSLEDNJI ELEMENT LISTE U JSL

*kako doći do poslednjeg elementa liste?

krenemo od glave dok ne dođemo do elementa čiji `_next` je None

ovaj postupak je $O(n)$

* bilo bi zgodno čuvati referencu na poslednji element liste analogno glavi, referenca se zove „rep“ (tail)

GRANIČNI SLUČAJEVI

*kako predstaviti praznu listu? `head = tail = None`

*kako predstaviti punu listu? lista nema ograničenje na maksimalan broj elemenata :)

* ako lista ima jedan element? `head == tail`

DODAVANJE ELEMENTA NA POČETAK LISTE U JSL

1. kreiraj novi čvor
2. upiši podatak u čvor
3. link na sledeći novog čvora pokazuje na glavu
4. glava pokazuje na novi čvor

DODAVANJE ELEMENTA NA KRAJ LISTE U JSL

1. kreiraj novi čvor
2. upiši podatak u čvor
3. link na sledeći novog čvora je None
4. poslednji→sledeći pokazuje na novi čvor
5. tail pokazuje na novi čvor

Dodavanje iza u jednostruko spregnutoj listi:

1. Napravimo novi element
 2. Next od prethodnog stavimo na novi element
 3. Next od novog elementa povežemo sa sledećim
- ako znamo koji je element, ako imamo referencu na taj čvor, složenost dodavanju el je $O(1)$
- a ako ne znamo gde je čvor, nego na osnovnu vrednosti ili pozicije ubacujemo element onda je složenost $O(n)$

Dodavanje ispred u jednostruko spregnutoj listi:

1. Krenemo od heada i gledamo da li je sledeci element taj što nam treba, itd..
2. Kada nadjemo traženi element, ponovimo postupak kao dodavanje iza

BRISANJE SA KRAJA LISTE U JSL

1. Moramo da idemo ispočetka kroz sve elemente dok ne nadjemo na pretposlednji
2. Kada nadjemo on postaje novi tail, a njegov next postaje ništa i onda čekamo da se oslobodi iz memorije

- $O(n)$

BRISANJE SA POČETKA LISTE U JSL

1. head treba da pokazuje na drugi element liste $head = head_next$

- $O(1)$

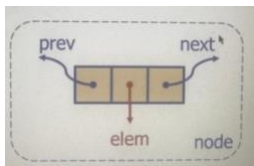
BRISANJE PROIZVOLJNOG ELEMENTA U JSL

1. Moramo da nadjemo element koji se nalazi pre njega
2. Njegovu vezu ka nextu preusmerimo na next od ovog našeg elementa

?Kako izračunati dužinu liste? Moramo da pristupimo svakom element $O(n)$

DVOSTRUKO SPREGNUTA LISTA

- * Problem kod jednostruko spregnute liste, treba da pronadjemo element koji se nalazi pre, nekako da evindetiramo dva elementa pa da stanemo pre nego što dodjemo do elementa
- * Dvostruko spregnuta lista taj problem nema, jer pamti oba elementa (prev i next), olakšava
- * kretanje „unazad“ (od repa prema glavi) u jednostruko spregnutoj listi je nemoguće
- * rešenje: čvorovi treba da sadrže referencu i na prethodni i na sledeći element liste



*GLAVA I REP:

- prvi i poslednji element imaju poseban status
- ne koriste se za čuvanje podataka
- prazna lista: $head.next == tail$ and $tail.prev == head$

UBACIVANJE ELEMENTA U LISTU U DSL

1. Formiramo novi node, upišemo podatak

-ako dodajemo na početak liste

2.next koji je bio od našeg headera će postati next od novog čvora

3. Prošli od novog je header

4. Sledeći od headera će postati novi

5. Prošli od datog sledećeg stavimo da je taj novi

-ako dodajemo iza ili ispred elementa, upitno je, zavisi da li imamo referencu na taj element imamo 4 slučajeve

1. Imamo referencu i iza dodajemo $O(1)$

2. Nemamo referencu i iza dodajemo $O(n)$

3.imamo referencu i ispred dodajemo $O(1)$

4. Nemamo referencu i ispred dodajemo $O(n)$

-jedina razlika je u dodavanju ispred sa referencom izmedju jsl i dsl, kod dsl se može obaviti bez iteriranja bez pronalaženja te pozicije, a kod jsl ta operacija nije moguća

BRISANJE SA KRAJA U DSL

1. Pronadjemo element koji se nalazi pre prethodnjeg elementa

2. Njegov sledeći je trailer

3. Njegov next se upise u next od trailera

-nismo iterirali $O(1)$

-ako brisemo sa pocetka isto kao sa kraja je $O(1)$

BRISANJE SA POČETKA U DSL

1. Brišemo element posle header

1. Nadjemo koji je njegov sledeci, i njegov prethodni podesimo na head

2. Od head sledeći je od trenutnog elementa koji brišemo next

*problem kod dsl je pristup po poziciji, indeksu – $O(n)$

8. OBLAST – Stabla

*stablo je apstraktni model hijerarhijske structure

* sastoji se od čvorova koji su u vezi roditelj/dete

*svaki čvor ima najviše jednog roditelja; tačno jedan čvor nema roditelja

* čvor ima nula ili više dece

TERMINOLOGIJA

*koren (root): jedini čvor bez roditelja

*unutrašnji čvor: čvor sa bar jednim detetom spoljašnji

*čvor/list (leaf): čvor bez dece

*predak: roditelj, deda, praded, ...do korena

* dubina čvora: broj predaka

*visina stabla: najveća dubina

* potomak: dete, unuč, prauuč, ...

* podstablo: čvor stabla i njegovi potomci

-Stablo je rekursivno definisana struktura podataka - kad posmatramo cvor sa decom ono opet čini stablo

STABLO ATP

*opšte metode:

int len(), boolean is_empty(), iterator nodes()

*metode za pristup podacima:

node root(), node parent(n), iterator children(n), int num_children(n)

*metode za ispitivanje čvorova:

boolean is_leaf(n), boolean is_root(n)

* ažuriranje sadržaja:

element replace(n, o)

-svaki čvor čuva neke podatke zato nam i služi kao struktura podataka, npr. Folder

STABLO U MEMORIJI

-svaki čvor stabla se može predstaviti kao trojka

-prvi element trojke je sam podatak

- drugi element je veza ka roditelju

- treci element je veza ka deci

*Equals - ako želimo da poredimo po recerenci onda nam ne treba, a ako želimo da poredimo podatak sa podatkom moramo da refinišemo metodu __eq__

*Kod uporedjivanja fajlova po folderima, ne možemo tek tako da uporedimo, nego moramo malo dublje

OBILAZAK STABLA

*obilazak po dubini (depth-first): obiđi čvor i njegove potomke pre braće

preorder: prvo čvor pa deca

postorder: prvo deca pa čvor

*obilazak po širini (breadth-first): obiđi čvor i njegovu braću pre potomaka

obilazak „po generacijama“ u stablu

OBILAZAK STABLA PO DUBINI

*Krenemo ka levoj strani i idemo do najdubljeg čvora, zatim se vraćamo na roditelja i opet odlazimo najdublje što može

*Razlikujemo dve varijante:

1. Preorder- prvo obiđemo čvor roditelja pa onda čvor dece (to može da bude ispis podataka u čvoru, vraćanje čvora kao nekakav iterator, obradu čvora- možemo metodi preorder da pošaljemo referencu ka nekoj funkciji koja će se baviti obradom pojedinačnih čvorova na najraznovrsniji mogući način), u pitanju je rekurzivno

2. Postorder- prvo obilazimo decu čvora pa onda čvor

OBILAZAK STABLA PO ŠIRINI

*treba obići sve čvorove dubine d pre nego što se pređe na čvorove dubine $d + 1$

*Ideja je da se iskoristi struktura podataka red (queue), u njega prvo ispisujemo korenski čvor, zatim skidamo jedan element sa reda, taj čvor obradjujemo, a zatim na kraj queue dodajemo svu njegovu decu, i tako u krug

*primer: stablo igre – svi mogući ishodi igre koju igra čovek ili računar; koren je početno stanje igre za igru „iks-oks“ (tic-tac-toe)

* $Q.enqueue(child)$ – za dodavanje elementa na kraj reda

* $Q.dequeue()$ – za uklanjanje elementa sa početka reda

BINARNO STABLO

*stablo za koje važi: svaki čvor ima najviše dvoje dece, svako dete je označeno kao levo dete ili desno dete, levo dete po redosledu prethodi desnom detetu

*levo podstablo – levo dete kao koren

*desno podstablo – desno dete kao koren

*pravilno binarno stablo: svaki čvor ima 0 ili 2 deteta

OSOBINE BINARNOG STABLA

*nivo stabla d ima najviše 2^d čvorova

* broj čvorova po nivou raste eksponencijalno

* n – broj čvorova

* e – broj listova

* i – broj internih čvorova

* h – visina

BINARNO STABLO U MEMORIJI

* Čvor se sastoji od 4 sributa: referenca ka roditelju, podatak, referenca ka levom detetu i referenca ka desnom detetu

BINARNO STABLO U MEMORIJI POMOĆU NIZA

* rang čvora:

za levo dete: $\text{rang}(\text{node}) = 2 \cdot n + 1$

za desno dete: $\text{rang}(\text{node}) = 2 \cdot n + 2$

za čvor: $(n-1)/2$

* Obilasku po širini popunjavamo niz, za cvorove koji nemaju decu ostavljamo prazno polje u nizu

* Inorder obilazak - ako im levo dete obidji celo levo podstablo, zatim obradu, a onda ka desnom detetu

STABLA ODLUČIVANJA

* binarno stablo strukturirano prema procesu odlučivanja

* unutrašnji čvorovi – pitanja sa da/ne odgovorima

* listovi – odluke

* primer: gde za večeru?

* broj pitanja odgovara visini stabla, a visina stable odgovori algoritmu broja elemenata

STABLO ARITMETIČKIH NIZOVA

* binarno stablo kreirano na osnovu aritmetičkog izraza

* unutrašnji čvorovi – operatori

* listovi – operandi

* primer: $2 * (a - 1) + 3 * b$

* specijalni slučaj inorder obilaska- svaki operator treba staviti u zagrade –samo da ispišemo izraz

* specijalni slučaj postorder obilaska- prvo dobavimo jedan operand pa onda dobavimo vrednost drugog operanda I tek onda na ta dva operanda možemo da primenimo operaciju-samo da izračunamo izraz

OJLEROV OBILAZAK STABLA

- * opšti postupak za obilazak stable
- * preorder, inorder, postorder su specijalni slučajevi
- * posmatramo grane stabla kao zidove koji uvek moraju da nam budu sa leve strane prilikom kretanja
 - * svaki čvor se poseti tri puta:
- sa leve strane (preorder)
- sa donje strane (inorder)
- sa desne strane (postorder)

CRTANJE STABLA

- * treba odrediti (x, y) koordinate čvorova stabla
- * (n) : broj čvorova posećenih pre čvora n u inorder obilasku
- * (n) : dubina čvora n

9. OBLAST - Red sa prioritetom, heap, adaptivni RSP

RED SA PRIORITETOM

- * red sa prioritetom čuva kolekciju elemenata
- * svaki element je par (ključ, vrednost)-drugačije nego rečnik
- * osnovne operacije:
 - $\text{add}(k, x)$: dodaje element sa ključem k i vrednošću x
 - $\text{remove_min}()$: uklanja element sa najmanjim ključem
- * dodatne operacije:
 - $\text{min}()$: vraća, ali ne uklanja, element sa najmanjim ključem
 - $\text{len}()$, $\text{is_empty}()$
- * primer: red u ambulanti, prioritet koliko je njihovo stanje hitno,

KLJUČEVI I RELACIJA PORETKA

- * ključevi mogu biti bilo kog tipa za koga je definisana relacija poretka
- * elementi u redu mogu imati jednake ključeve – u tom slučaju se primenjuje FIFO princip
- * relacija poretka
 - refleksivna: $x \leq x$
 - antisimetrična: $x \leq y \wedge y \leq x \Rightarrow x = y$
 - tranzitivna: $x \leq y \wedge y \leq z \Rightarrow x \leq z$

IMPLEMENTACIJA RSP

* implementacija sa nesortiranom listom:

- add je $O(1)$ jer dodavanje možemo raditi na bilo kom kraju liste
- remove_min i min su $O(n)$ jer moramo tražiti najmanji ključ u listi

* implementacija sa sortiranom listom

- add je $O(n)$ jer moramo da nađemo pravo mesto za ubacivanje novog elementa
- remove_min i min su $O(1)$ jer je najmanji ključ uvek na početku

OSOBI NE STABALA

* **Balansirano** stablo je binarno stablo za koje važi da je:

Razlika broja čvorova levog i desnog podstabla najviše 1.

Levo podstablo balansirano.

Desno podstablo balansirano.

* Održavanje stabla balansiranim minnimizuje visinu stabla, čime utiče na efikasnost operacija poput dodavanja i uklanjanja čvora.

* Popunjeno stablo (full, proper binary tree) je binarno stablo kod kog svi čvorovi osim lisnih imaju tačno 2 potomka.

* Kompletno stablo (complete binary tree) je binarno stablo kod kog su svi nivoi osim eventualno poslednjeg popunjeni, a svi čvorovi su pomereni ulevo što je više moguće.

HEAP

* heap je binarno stablo čiji elementi su uređeni parovi (ključ, vrednost) i koje zadovoljava još 2 uslova:

redosled: za svaki čvor n osim korena ključ od n je veći ili jednak ključu roditelja od n **kompletnost:** heap visine h ima nivoe $0, 1, 2, \dots, h-1$ sa maksimalnim brojem čvorova (i -ti nivo ima 2^i čvorova za $0 \leq i \leq h-1$)

* malo prostije rečeno, roditelj ima manji ključ od dece, svi čvorovi imaju po dvoje dece, ukoliko nivo nije popunjen nalaze se elementi redom sa leve strane, mnogo je bolje koristiti nego sort i nesort

* poslednji čvor je poslednji čvor sa desne strane na najnižem nivou stable (to je čvor koji možemo najlakše ukloniti)

DUBINA HEAP-A

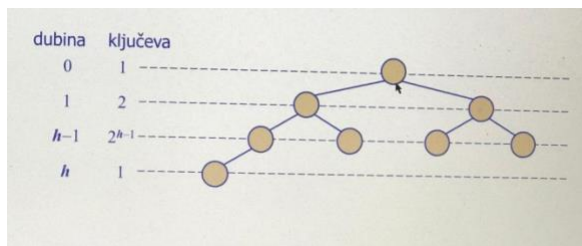
* teorema: heap koji čuva n ključeva ima dubinu $O(\log n)$

h : visina heapa sa n ključeva

ima 2^i ključeva na dubini $i = 0, \dots, h-1$ i bar jedan ključ na dubini h

prema tome, $n \geq 1 + 2 + 4 + \dots + 2^{h-1} + 1$

$$n \geq 2 \text{ na } h, h \leq \log$$



HEAP I RED SA PRIORITETOM

- * red sa prioritetom možemo implementirati pomoću heapa
- * u svakom čvoru stabla čuvamo par (ključ, vrednost)
- * pamtimo položaj poslednjeg čvora
- ? Gde ćemo naći kod heapa minimalni element, element sa najmanjim prioritetom? Uvek se nalazi u korenu.

DODAVANJE U HEAP

- * novi element se uvek dodaje na kraj, prioritet će možda biti manji od roditelja
- * add u redu sa prioritetom se implementira kao dodavanje u heap
- * dodavanje se vrši u tri koraka
- 1. nađi novi poslednji čvor z
- 2. sačuvaj (k, v) u z
- 3. restauriraj pravilan redosled – ukoliko je dete manje od roditelja, menjamo dete i roditelja dok ne dobijemo pravilnu implementaciju heapa – u najgorem slučaju menjamo do korena

DODAVANJE U HEAP: RESTAURACIJA REDOSLEDA

- * nakon dodavanja novog ključa k redosled čvorova može biti narušen
- * algoritam **upheap** uspostavlja korektan redosled zamenom k duž putanje od novog čvora prema korenu
- * upheap se završava kada k dođe u koren ili njegov roditelj ima ključ manji ili jednak k
- * pošto heap ima visinu $(\log n)$, upheap radi u $(\log n)$ vremenu

UKLANJANJE IZ HEAPA

- * `remove_min` se implementira kao uklanjanje korena iz heapa
- * uklanjanje se vrši u tri koraka
- 1 na mesto korena stavi poslednji čvor w – na poziciju gde je jednostavno ukloniti
- 2 ukloni w
- 3 restauriraj pravilan redosled

UKLANJANJE IZ HEAP-a: RESTAURACIJA REDOSLEDA

- * nakon smeštanja ključa k poslednjeg čvora u koren redosled čvorova može biti narušen
- * algoritam **downheap** uspostavlja korektan redosled zamenom k duž putanje od korena
- * downheap se završava kada k dođe u list ili njegova deca imaju ključeve veće ili jednake k
- * pošto heap ima visinu $O(\log n)$, downheap radi u $O(\log n)$ vremenu

Nađi mesto za novi poslednji prilikom dodavanja

- * mesto za novi poslednji čvor se može naći prolaskom kroz putanju od $O(\log n)$ čvorova
- idi prema gore dok ne dođeš do korena ili nečijeg levog deteta
- ako si došao do nečijeg levog deteta, idi na desno dete
- idi prema dole levo dok ne dođeš do lista
- * sličan je i algoritam prilikom uklanjanja

IMPLEMENTACIJA HEAP-a POMOĆU NIZA

- * olakšavajuća okolnost je što je kompletno stablo, tako da nećemo imati problem sa praznim mestima
- * popunjava se obilaskom po širini
- * heap sa n ključeva se može smestiti u niz dužine n
- * za čvor ranga i
 - levo dete ima rang $2i + 1$
 - desno dete ima rang $2i + 2$
- * veze između čvorova se ne čuvaju
- * dodavanje se svodi na upis čvora ranga $n + 1$
- * uklanjanje se svodi na uklanjanje čvora ranga n

SPAJANJE DVA HEAP-A

- * imamo dva heap-a i ključ k
- * kreiramo novi heap sa korenom k i dva heap-a kao podstabla
- * pokrenemo downheap da restauriramo redosled

Konstrukcija heap-a od dole (bottom-up)

- * možemo da napravimo heap sa n ključeva pomoću bottom-up spajanja u $O(\log n)$ koraka
- * u i -tom koraku, par heapova sa $2i - 1$ ključeva se spajaju u heap sa $2i + 1 - 1$ ključeva

?Kako mi znamo da ćemo kreirati heap od n elemenata, kako da znamo koliko ćemo listova da uzmemo? Najveći stepen dvojke koji je umanjen za 1 da je manji od n

Primer: ako je broj čvorova 15, 8 lisnih čvorova

Bottom-up primer - uzimamo nasumično čvorove, dodajemo novi čvor i spajamo ga sa listovima, vrši se restauracija, kad smo sredili elemente dodajemo nove čvorove i povezujemo i vršimo restauraciju, formiramo heap koji je za 1 nivo viši od prethodnog i tako dok ne dodjemo do čvora

ANALIZA KONSTRUKCIJE HEAPA

- * najgori slučaj za downheap: prvo krene desno pa onda stalno levo do dna heapa
- * svaki čvor se obiđe u najviše dve putanje
- * ukupan broj čvorova u putanjama je (n)
- * bottom-up konstrukcija heapa radi u (n) vremenu
- * bottom-up konstrukcija heapa je brža nego n dodavanja u heap sa upheap korekcijom

ADAPTIVNI RED SA PRIORITETOM

- * to je red sa prioritetom koji se može menjati u nekim trenucima
- * primer: sistem za kupoprodaju akcija koristi dva reda sa prioritetom, jedan za prodaju i drugi za kupovinu sa elementima (p, s)

ključ p je cena

vrednost s je broj akcija

nalog za kupovinu (p, s) se izvršava kada se pojavi nalog za prodaju (p', s') sa cenom $p' \leq p$ (postupak je završen ako $s' \geq s$)

nalog za prodaju (p, s) se izvršava kada se pojavi nalog za kupovinu (p', s') sa cenom $p' \leq p$ (postupak je završen ako $s' \geq s$)

- * šta ako neko hoće da otkaže nalog pre nego što se izvrši?
- * šta ako neko hoće da izmeni cenu ili broj akcija? Ideja je da se uvede locator
- * **remove**(loc): ukloni i vrati element e iz reda za lokator loc
- * **update**(loc, k, v): zameni ključ/vrednost par (k, v) za lokator loc

LOKATORI

- * element sa lokatorom identifikuje i prati poziciju (k, v) unutar strukture podataka
- * primeri: broj kaputa u garderobi, broj rezervacije
- * osnovna ideja: pošto elemente kreira i vraća sama struktura podataka, oni mogu biti takvi da pamte svoju lokaciju, što pojednostavljuje kasnije ažuriranje

LOKATORI I LISTE

- * element liste čuva: ključ, vrednost, poziciju
- * reference se ažuriraju u swap operaciji
- * ako se promeni prioritet nekog elementa, to može da utiče na raspored elemenata

LOKATORI I HEAP

- * element heapa čuva: ključ, vrednost, poziciju
- * reference se ažuriraju u swap operaciji
- * Ako se prioritet čvora promeni, kako restaurirati redosled? Treba da odredimo downheap ili upheap u zavisnosti od vrednosti koja se ovde promeni

metoda	nesortirana lista	sortirana lista	heap
len, is_empty	$O(1)$	$O(1)$	$O(1)$
add	$O(1)$	$O(n)$	$O(\log n)$
min	$O(n)$	$O(1)$	$O(1)$
remove_min	$O(n)$	$O(1)$	$O(\log n)$
remove	$O(1)$	$O(1)$	$O(\log n)$
update	$O(1)$	$O(n)$	$O(\log n)$

RED SA PRIORITETOM I SORTIRANJE

- * možemo upotrebiti red sa prioritetom za sortiranje niza elemenata
- dodamo elemente jedan po jedan putem **add** operacije
- uklonimo elemente jedan po jedan putem **remove_min** operacije
- *vreme izvršavanja zavisi od načina implementacije
- *uopšten šablon,možemo I na heap, I nesortirane I sortirane sa prioritetom da primenimo

SELECTION SORT

- * selection sort je varijanta PQ-sorta gde je RSP implementiran pomoću **nesortirane liste**
- * vreme izvršavanja selection sorta:
- dodavanje n elemenata u RSP traje $O(n)$
- uklanjanje n elemenata u sortiranom redosledu traje
- $1 + 2 + \dots + n$
- * selection sort radi u $O(n^2)$ vremenu

INSERTION SORT

- * insertion sort je varijanta PQ-sorta gde je RSP implementiran pomoću **sortirane liste**
- * vreme izvršavanja insertion sorta:
- dodavanje n elemenata u RSP traje $1 + 2 + \dots + n$
- uklanjanje n elemenata traje $O(n)$

*insertion sort radi u (n^2) vremenu

HEAP SORT

* posmatramo RSP sa n elemenata, implementiran pomoću **heapa**

potreban prostor je $O(n)$

add i remove_min traju $O(\log n)$

len, is_empty, min traju $O(1)$

* ovakav RSP možemo koristiti za sortiranje n elemenata za $O(n \log n)$ vreme

*rezultujući algoritam se zove **heap sort**

*znatno brži od kvadratnih algoritama kao što su selection i insertion sort

$$(n \log n) < (n^2)$$

Sortiranje unutar iste strukture podataka (in-place)

* U svim prethodnim primerima smo imali po dve sekvence, iz jedne sekvence prebacivali u drugu, iz druge u prvu i samim tim zauzimali duplo više memorijskih lokacija nego što nam je bilo neophodno

* umesto korišćenja 2 strukture možemo implementirati selection i insertion sort u okviru jedne structure – u mestu

* Na početku su svi elementi sortirani, onda polako sa leve strane grupišemo elemente u sortiranom redosledu, i tako dobijamo sortiranu kolekciju

10. OBLAST - Mape, heš tabele, skip liste, skupovi

MAPA

* Pythonov rečnik (klasa dict) preslikava ključeve na vrednosti

*drugo ime: **asocijativni niz** ili **mapa**

*ključevi su jedinstveni (nema ponavljanja), vrednosti ne moraju biti jedinstvene

MAPA ATP: OSNOVNE OPERACIJE

* $M[k]$ vraća vrednost v vezanu za ključ k u mapi M ; ako ne postoji, izaziva KeyError; implementira je `__getitem__`

* $M[k] = v$ dodeljuje vrednost v ključu k u mapi M ; ako ključ već postoji, zamenjuje staru vrednost; implementira je `__setitem__`

* `del M[k]` uklanja element sa ključem k iz mape M ; ako ne postoji, izaziva KeyError; implementira je `__delitem__`

* `len(M)` vraća broj elemenata u mapi M ; implementira je `__len__`

* `iter(M)` generiše listu ključeva iz mape M ; implementira je `__iter__`

MAPA ATP: DODATNE OPERACIJE

- * k in M vraća True ako mapa M sadrži ključ k ; implementira je `__contains__`
- * $M.get(k, d=None)$ vraća $M[k]$ ako ključ k postoji u M ; inače vraća default vrednost d ; ne izaziva `KeyError`
- * $M.setdefault(k, d)$ ako k postoji u mapi, vraća $M[k]$; ako ne postoji, postavlja $M[k] = d$ i vraća d
 - * $M.pop(k, d=None)$ uklanja element sa ključem k i vraća vezanu vrednost v ; ako ključ k nije u mapi M , vraća d ili izaziva `KeyError` ako je d jednako `None`

MAPA ATP: JOŠ NEKE OPERACIJE

- * $M.popitem()$ uklanja neki element mape i vraća (k, v) ; ako je mapa prazna izaziva `KeyError`
 - * $M.clear()$ uklanja sve elemente iz mape
- * $M.keys()$ vraća skup svih ključeva iz M
- * $M.values()$ vraća skup svih vrednosti iz M
- * $M.items()$ vraća skup svih parova (k, v) iz M
- * $M.update(M2)$ dodeljuje
- * $M[k]=v$ za svaki (k, v) iz $M2$
- * $M == M2$ vraća True ako mape sadrže iste parove (k, v)
- * $M != M2$ vraća True ako mape ne sadrže iste parove (k, v)

MAPA POMOĆU LISTE

- * Jedna moguća implementacija mape je pomoću dvostruko spregnute liste
- * elemente čuvamo u proizvoljnom redosledu
- * moguće je da element ne pronadjemo, složenost $O(n)$

MAPA POMOĆU LISTE: Performanse

- * **dodavanje** traje $O(1)$ – novi element možemo dodati na početak ili na kraj
- * **traženje** ili **uklanjanje** traje $O(n)$ – u najgorem slučaju (nije pronađen element) mora se proći kroz celu listu
- * ovakva implementacija je korisna samo za mape sa malim brojem elemenata
- * ili ako je dodavanje najčešća operacija, dok se traženje i uklanjanje retko obavljaju
- * Ovakve strukture se uglavnom ne koriste ili se vrlo ograničeno koriste, performanse su loše, ovo ima smisla za strukture sa malim brojem elemenata ako uopšte i ima smisla, ako samo dodajemo a retko uklanjamo i tražimo, onda možda ova implementacija i nije tako loša

HASH TABELA

- * mapa omogućava pristup korišćenjem ključeva kao indeksa – $M[k]$
- * zamislimo mapu koja kao ključeve korisiti cele brojeve iz intervala $[0, N - 1]$ za neko $N > n$

* za čuvanje elemenata možemo koristiti lookup niz dužine N

*npr. mapa sa elementima $(1, D), (3, Z), (6, C), (7, Q)$

*ZNAČI: Ideja je da se unapred zauzme odredjen broj memorijskih lokacija, i ubacujemo elemente i svaki od ovih parova $(1, D)$.. ključ, vrednost, ubacujemo na poziciju koja odgovara ključu, daklr na indeks ovog našeg lookup niza na indeks koji odgovara ključu, na ovaj način dobijamo jednostavan pristup

* operacije su $O(1)$

?Koja su ograničenja ovog pristupa?

-zameramo zato što ključ mora biti integer, mora biti samo jedan od intidžera iz ovog interval, resize je potreban, zauzeli smo više memorija nego što nam je potrebno, nastaje problem I ako se ova struktura popuni

*šta ako je $N \gg ?$ šta ako ključevi nisu celi brojevi? pretvorićemo ključeve u cele brojeve pomoću **hash funkcije**

* dobra hash funkcija će ravnomerno distribuirati ključeve u $[0, N - 1]$ ali može biti duplikata

*duplikate ćemo čuvati u „kantama“ – tzv. **bucket array**

*npr hash funkcija po modulu 9, desice se da vise različitih funkcija mapirati u istu memorijsku lokaciju

* hash funkcija mapira ključeve na indekse u hash tabeli

*npr. poslednje četiri cifre broja telefona

* hash funkcija mapira ključ k na ceo broj u intervalu $[0, N - 1]$ gde je N kapacitet niza kanti A

*element (k, v) čuvamo u nizu kao $[h(k)]$

***kolizija**: dve vrednosti ključa koje daju isti hash

* dobre hash funkcije imaju **vrlo malo** kolizija

?Sa ovakvom implementacijom kako izgleda pristup vrednosti po ključu? Npr ako tražimo vrednost koja je pridružena ključu 17 mi prvo moramo da izracunamo hash vrednost(ovde je primer ostatak pri deljenju sa 11, pa je hadh vrednost 6), mi pristupano direktno indeksu 6, ali s obzirom da se tu nalazi više elemenata mi moramo da poredimo naš traženi ključ

* Najgori slučaj da su u jednoj kanti svi elementi

* U praksi se koriste dobre hash funkcije koje će moći ravnomerno da rasporede naše ključeve po baketima, šanse su male da se ponove

* često se hash funkcija može posmatrati kao kompozicija dve funkcije:

-proizvoljan objekat pretvoriti u neki integer-**HASH CODE**

- **compression function**: mapira hash kôd na broj u intervalu $[0, N - 1]$

- tada hash code ne zavisi od veličine niza kanti

* vrednosti koje su „blizu“ u skupu ključeva ne moraju imati hasheve koji su „blizu“-u prevodu: da li ako heširamo 5 i 6, da li ce one zauzeti uzastopne memorijskr lokacije, sve zavisi, ako

uzmemo hash funkciju deljenje po modulu, onda hoće, a za neke druge se to neće desiti, npr. lozinke na serveru se čuvaju u heširanom obliku, ako neko otkrije da je heš naše lozinke jako blizak hešu neke slične lozinke, on lako može da otkrije koja je naša lozinka, zbog toga je ova osobina poprilično nepoželjna

HASH CODE

*Možemo da iskoristimo hash code za memorijsku adresu:

- adresa Python objekta u memoriji kao hash code
- dobro osim za numeričke tipove(zauzima se jedan objekat za sve moguće jedinice) i stringove(nepromenljivi-ako imamo dva stringa i zauzeti su na različitim memorijskim lokacijama, njihova memorijska lokacija će biti različita iako su stringovi isti)

* integer cast:

- za svaki tip podataka koji se predstavlja sa najviše onoliko bita koliko i int možemo uzeti int interpretaciju njegovih bita
- za tipove koji zauzimaju više memorije moramo nekako „sažeti“ njegove bite
- npr. float broj u Pythonu zauzima 64 bita a hash kod 32; možemo izabrati

gornjih 32 bita

donjih 32 bita

neku kombinaciju sva 64 bita: XOR ili zbir gornje i donje polovine, itd.

* suma komponenti

podelimo bitove ključa na delove po 32 bita

saberemo delove (ignorišemo overflow)

zgodno za numeričke ključeve duže od int-a

* polinomska akumulacija(ona najviše dolazi do izražaja kod tipova koji imaju neke pozicije, npr kod stringova

-podelimo bitove ključa na delove fiksne dužine (npr. 8, 16, 32 bita) $a_0 a_1 a_2 \dots a_{n-1}$

-izračunamo polinom (ignorišući overflow) za fiksno z :

$$(z) = a_0 + a_1 z + a_2 z^2 + \dots + a_{n-1} z^{n-1}$$

- posebno zgodno za stringove ($z = 33$ daje samo 6 kolizija za 50.000 engleskih reči)

- polinom $p(z)$ se može izračunati u $O(n)$ vremenu Hornerovom metodom

KOMPRESUJUĆA FUNKCIJA

* celobrojno deljenje

$$h(y) = y \bmod N$$

veličina hash tabele N je obično prost broj

*multiply, add and divide (MAD)

$h(y) = (ay + b) \bmod N$ –množimo prostim brojem a , i najčešće dodajemo neki veliki broj b

a i b su nenegativni celi brojevi takvi da je $a \bmod N \neq 0$

RUKOVANJE KOLIZIJAMA

* kolizije nastaju kada se različiti elementi mapiraju na istu ćeliju

* ulančavanje duplikata: svaki element heš tabele je glava liste koja čuva elemente

* traži dodatnu memoriju pored same heš tabele

LINEARNO TRAŽENJE

* linear probing: smešta element u koliziji u prvu sledeću slobodnu ćeliju (cirkularno)

* elementi u koliziji se nagomilavaju izazivajući dalje kolizije

ČITANJE SA LINEARNIM TRAŽENJEM

* Heširamo taj broj, i pristupimo indeksu, ako se tu ne nalazi proveravamo na sledećim memorijskim lokacijama, ukoliko dodjemo do prazne memorijske lokacije, moramo da uvedemo razliku da li je nekada bio tu traženi broj pa se izbrisao ili se nikada nije ni nalazio broj, ukoliko se nikada nije nalazio možemo da tvrdimo da se traženi broj ne nalazi u hash tabeli

IZMENE SA LINEARNIM TRAŽENJEM

* uvodimo poseban objekat AVAILABLE koji zamenjuje uklonjene elemente

* $remove(k)$: tražimo element sa ključem k ako smo ga našli, vraćamo ga i na njegovo mesto upisujemo AVAILABLE inače vratimo None

* $put(k, o)$:

-izuzetak ako je tabela puna

-počinjemo od ćelije $h(k)$

-ispitujemo naredne ćelije sve dok se ne dogodi nešto od:

-našli smo ćeliju koja je prazna ili sadrži AVAILABLE

-isprobali smo svih N ćelija

-Upišemo (k, o) u ćeliju i

DUPLO HEŠIRANJE

* koristi se sekundarna heš funkcija $d(k)$

* prilikom kolizije element se smešta u prvu slobodnu ćeliju iz niza

$(i + (k)) \bmod N$ za $j = 0, 1, \dots, N - 1$

* sekundarna heš funkcija ne sme vratiti 0

*veličina tabele N mora biti prost broj da bi se mogle probati sve ćelije

* čest izbor za $d(k)$: $d(k) = q - (k \bmod q)$, $q < N$, q je prost broj, rezultat $d(k)$ je u intervalu $[1, q]$

DUPLO HEŠIRANJE:PRIMER

* heš tabela sa duplim heširanjem

$$N = 13$$

$$h(k) = k \bmod 13$$

$$g(k) = 7 - (k \bmod 7)$$

*duplo heširanje se vrši ako je prilikom prvog heširanja memorijska lokacija već zauzeta

PERFORMANSE HEŠIRANJA

* u najgorem slučaju pretraga, dodavanje, uklanjanje traju $O(n)$

*najgori slučaj: kada su svi ključevi u koliziji

* faktor popune $\alpha = n/N$ utiče na performanse

*ako heševi liče na slučajne brojeve može se pokazati da je očekivani broj probanja prilikom dodavanja $1/(1 - \alpha)$

* očekivano vreme izvršavanja svih operacija je $O(1)$

*u praksi heširanje je vrlo brzo ako faktor popune nije blizu 100%

SKIP LISTA

*u hash tabeli se zauzima više memorijskih lokacija nego što je potrebno, pa se pokušava skip listama da se to popravi

* binarna pretraga sa sortiranim nizom: omogućava pronalaženje elemenata u $O(\log n)$ vremenu

*ali su operacije dodavanja i uklanjanja (n) u najgorem slučaju

*skip lista omogućava sve u $(\log n)$ vremenu u **prosečnom** slučaju

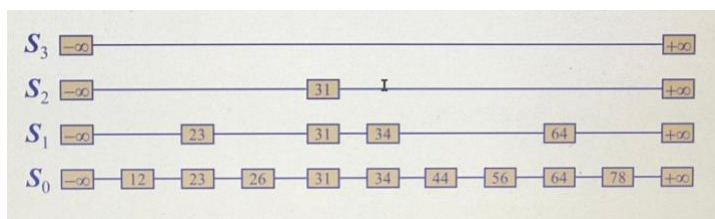
* skip lista za skup S elemenata (k, v) je serija lista S_0, S_1, \dots, S_h takvih da

1 svaka lista S_i sadrži posebne ključeve $-\infty$ i ∞

2 lista S_0 sadrži ključeve iz S u neopadajućem redosledu

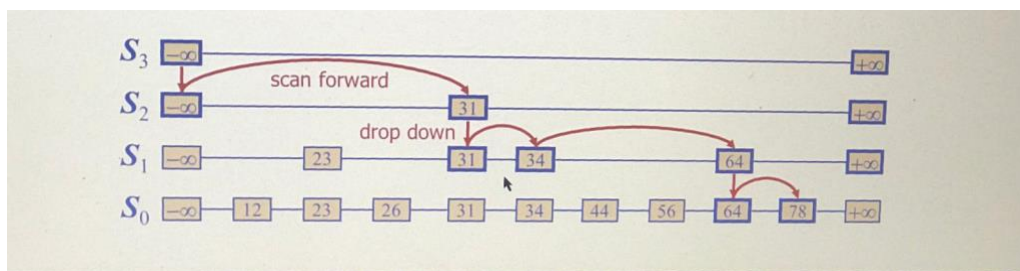
3 svaka lista je podskup prethodne $S_0 \supseteq S_1 \supseteq \dots \supseteq S_h$

4 lista S_h sadrži samo posebne ključeve



SKIP LISTA PRETRAGA

- * tražimo ključ x u skip listi na sledeći način:
- * počnemo od prvog elementa liste na vrhu
- * poredimo traženi ključ sa sledećim, na tekućoj poziciji p_i poredimo x sa $y = k(next(p))$
 - $x =$: pronašli smo traženi element
 - $x >$: idemo napred (**scan forward**)
 - $x <$: idemo dole (**drop down**)
- * ako smo na nivou S_0 i treba da idemo dole, nema traženog elementa
- * primer 78



ALGORITMI SA UVEDENOM SLUČAJNOŠĆU

- * koriste (pseudo)slučajne vrednosti da upravlja svojim izvršavanjem, nasumično odabrane vrednosti
- * vreme izvršavanja zavisi od ishoda „bacanja novčića“ (gore smo uzeli 31 zbog bacanja novčića)
- * možemo statistički pristupiti proceni kvaliteta ovog algoritma što nismo imali do sada
- * analiza vremena izvršavanja ovakvih algoritama podrazumeva: svi ishodi „bacanja novčića“ su jednako verovatni, bacanja su međusobno nezavisna
- * vreme izvršavanja u **najgorem slučaju** za ovakve algoritme je često veliko ali je vrlo malo verovatno (npr. sva bacanja novčića imaju isti ishod)
- * koristićemo ovakav algoritam za dodavanje elemenata u skip listu

DODAVANJE U SKIP LISTU

- * dodavanje novog elementa (x, o) u skip listu:
 - Bojana: Npr imamo $-\infty$ i $+\infty$, sad treba da ubacimo broj 18 koji se nalazi između. Svaki put kada ubacimo element u praznu listu, moramo da dodamo neki nivo S_0 , moramo da dodamo još jedan nivo S_1 , posto na gornjem nivou uvek mora da je prazna lista. Sad nam preostaje da izračunamo na koliko nivoa iznad će se ova 18 propagirati, i to odredjujemo nasumično..bacanje novčića :
 - ponavljamo bacanje novčića sve dok ne dobijemo „pismo“
 - sa i označimo broj puta koliko se pojavila „glava“
 - ako je $i \geq h$ dodaćemo nove liste S_{h+1}, \dots, S_{i+1} , svaku samo sa $-\infty + \infty$

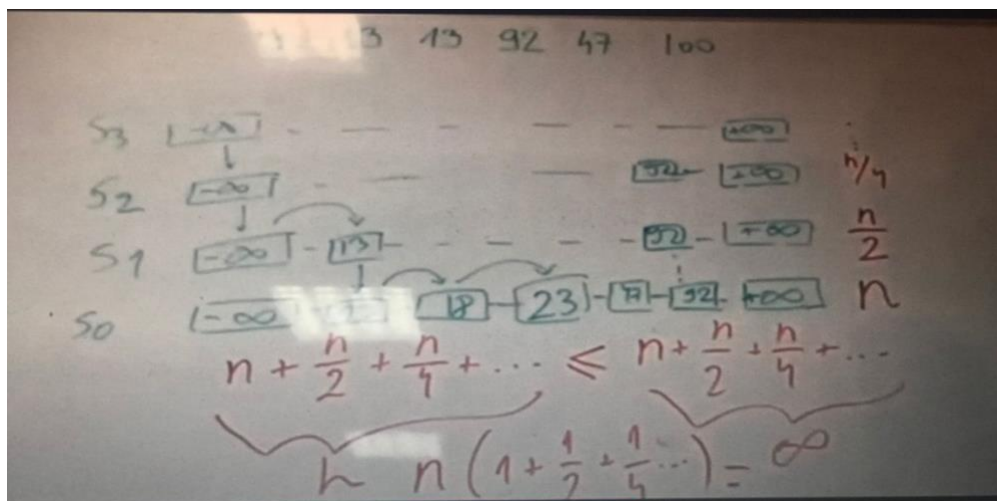
-tražimo x u skip listi i nađemo pozicije p_0, p_1, \dots, p_i elemenata sa najvećim ključem manjim od x u svakoj od lista S_0, S_1, \dots, S_i

-za $j \leftarrow 0, \dots, i$ dodaćemo (x, o) u listu S_j nakon pozicije p_j

-Nastavak: Sad ubacujemo broj 23, pronalazimo mesto gde se nalazi, kretanjem kroz više nivoa nam omogućuje da brže nađemo poziciju, ovde imam složenost dodavanja operacije što nam ne dopada, zbog toga krećemo od najvišeg nivoa, posto je broj 23 manji od $+\infty$, spuštamo se jedan nivo niže, poredimo sa elementom 18, prelazi 23 na mesto pored 18, poredimo onda sa $+\infty$, posto je manji ostaje, onda moramo da utvrdimo na koliko nivoa se broj 23 propagira, čekamo glavu, bacamo novčić, ispada odmah glava, što znači da ni 23 ne propagiramo nigde gore, dolazi broj 13... opet isto.. dolazi na mesto pre 18, bacamo novčić, npr je ispalo jednom novčić, jednom glava, znači propagira jedan nivo iznad i onda mora da se doda još jedna lista na nivou iznad itd...

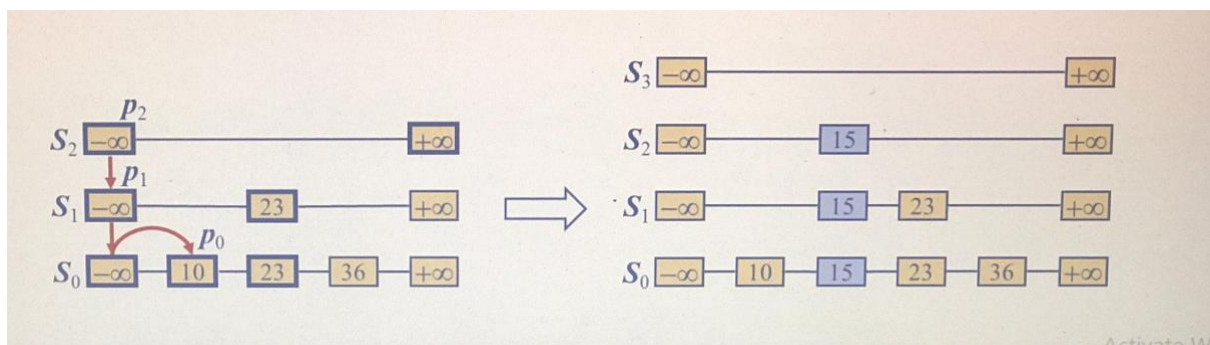
?Koliko ključeva će biti na nivou S_1 , i čemu taj broj odgovara? Broj ključeva odgovara verovatnoći da u jednom bacanju novčića dobijemo prvi put pismo. Verovatnoca je 50%, što znači da je broj ključeva na nivou S_1 , 50 % od ključeva na nivou S_0

? Koja verovatnoća je da dva puta bacimo novčić, i dva puta da dobijemo pismo? $\frac{1}{4}$



? Koliko nam memorijskih lokacija ovde treba? Manje od $2n$, što znači da imamo linearno zauzeće memorije.

primer: dodajemo ključ 15, za $i = 2$



UKLANJANJE IZ SKIP LISTE

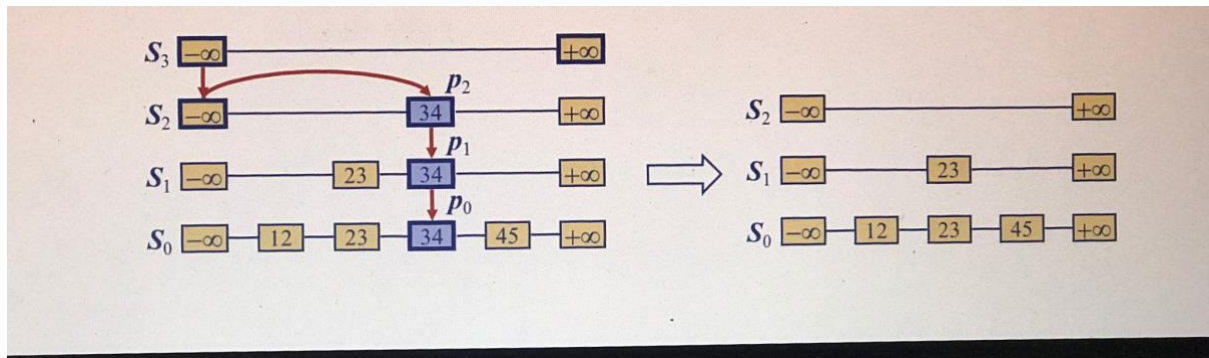
-uklanjanje elementa sa ključem x iz skip liste:

-tražimo x u skip listi i nađemo pozicije p_0, p_1, \dots, p_i elemenata sa najvećim ključem manjim od x u svakoj od lista S_0, S_1, \dots, S_i

-uklonimo pozicije p_0, p_1, \dots, p_i iz listi

- uklonimo sve prazne liste osim jedne

primer: uklanjamo ključ 34



-Nikad se ne vraćamo gore i uvek idemo sa leva ka desnoj strani

IMPLEMENTACIJA SKIP LISTE

-možemo da koristimo quad-nodes (element, link na prethodni, link na sledeći, link na čvor ispod, link na čvor iznad)

SKIP LISTE I ZAUZEĆE PROSTORA

-količina zauzete memorije zavisi od bacanja novčića

- iz teorije verovatnoće:

(a) verovatnoća da se dobije i uzastopnih glava je $1/2^i$ (Verovatnoća da se jedan čvor propagira na i nivoa visine)

(b) ako je svaki od n elemenata prisutan u listi sa verovatnoćom p , veličina skupa je np

(c) ako svaki od n događaja ima verovatnoću p , verovatnoća da će se desiti bar jedan nije veća od np

(d) očekivani broj bacanja novčića da se dobije „pismo“ je 2

-posmatramo skip listu sa n elemenata:

prema (a), dodaćemo čvor u listu S_i sa verovatnoćom $1/2^i$

-prema (b), očekivana veličina liste S_i je $n/2^i$

-očekivani broj čvorova u skip listi sa n elemenata je (n)

VISINA SKIP LISTE

-Vreme izvršavanja pretrage i dodavanja u skip listu zavisi od njene visine

- Prvi pitanje koje može da nam postavi je koje su onda preformanse operacija dodavanja, brisanja i pristupa? Šta je najgori slučaj? Performanse ove 3 operacije su sve iste. Pretežno najgori slučaj je koji smo imali sa 92 je npr ako hoćemo da umetnemo 90.
- Čemu odgovara kompletnost ove operacije u najgorem slučaju? Jedan deo su svi -besk(visina skip liste $O(h + n)$ - visina plus broj elemenata na najnižem nivou-male su šanse da se ovo desi statistički gledano), a drugi deo su svi elementi S_0 nivoa sve do 92 ($O(n)$)
- ?Koje su šanse da se pogodi najgori slučaj? Verovatnoća da nivo i postoji jednaka je verovatnoći da smo i puta bacili novčić, i da smo i puta dobili pismo, ova verovatnoća da nivo i postoji mora biti manja ili jednaka $1/2^i$
- Verovatnoća da bilo koji od ovih n elemenata je dostigao visinu h , odnosno verovatnoća da skip lista ima visinu h odgovara zapravo $n/2^i$ jer za svaki element imamo jednaku verovatnoću
- posmatramo skip listu sa n elemenata:
- prema (a), dodaćemo čvor u listu S_i sa verovatnoćom $1/2^i$
- prema (c), verovatnoća da S_i ima bar jedan čvor je najviše $n/2^i$
- ako izaberemo $i = 3 \log n$, verovatnoća da $S_{3 \log n}$ ima bar jedan čvor je najviše

$$\frac{n}{2^{3 \log n}} = \frac{n}{n^3} = \frac{1}{n^2}$$

prema tome, skip lista sa n elemenata je visoka najviše $3 \log n$ sa verovatnoćom najmanje $1 - 1/n^2$

- Verovatnoća da je h manje od \log je 1
- sa vrlo velikom verovatnoćom visina skip liste sa n elemenata je $(\log n)$
- Svaki niži nivo ima otprilike duplo više elemenata, statistika kaže da postoji 50% šanse da smo mi polovinu elemenata nivoa ispod preskočili, pa dobijamo efekte koji odgovaraju binarnoj pretrazi sa polovljenjem
- I na kraju zapravo imamo neku vrstu binarnog stabla, broj operacija odgovara prolasku kroz binarno stablo od korena do lista, jednom putanju što zapravo odgovara \log broja elemenata

PRETRAGA I AŽURIRANJE SKIP LISTE

- vreme pretrage je proporcionalno:
- broju drop down koraka, plus
- broju scan forward koraka (Broj koraka koji treba da obavimo ovim prelaskom na sledeći desni element)
- drop down koraci su ograničeni, odgovaraju visini skip liste, dakle $(\log n)$ sa velikom verovatnoćom
- kada radimo scan forward korak, ključ se ne nalazi u listi iznad scan forward, postoji zato što je ranije novčić dao „pismo“
- prema (d), u svakoj listi očekivani broj scan forward koraka je 2 prema tome, ukupan broj scan forward koraka je $O(\log n)$, očekivano vreme za pretragu u skip listi je $O(\log n)$ (slično tome dodavanje i uklanjanje)

-Zaključak: upotreba memorije je značajno manja u odnosu na hash tabelu

-brzina pretrage, dodavanja i uklanjanja je $(\log n)$ sa velikom verovatnoćom

SKUP, MULTISKUP, MULTIMAPA

-**skup** (set) je kolekcija elemenata koja ne poznaje redosled i ne sadrži duplikate

-elementi skupa su nalik ključevima koji nemaju sebi asocirane vrednosti

-**multiskup** (multiset, bag) je skup koji dopušta duplikate

-**multimapa** je mapa koja dopušta da za jedan ključ bude vezano više vrednosti

-indeks u knjizi preslikava pojam na jednu ili više stranica na kojima se on pominje

<code>S.add(e)</code>	dodaje element e u S ; nema efekta ako je e već prisutan u S
<code>S.discard(e)</code>	uklanja e iz S ; nema efekta ako e nije prisutan u S
<code>e in S</code>	vraća True ako je e prisutan u S ; implementira je <code>__contains__</code>
<code>len(S)</code>	vraća broj elemenata u S ; implementira je <code>__len__</code>
<code>iter(S)</code>	iterira kroz elemente iz S ; implementira je <code>__iter__</code>

-pomoću osnovnih operacija implementiraju se sve ostale

-dodatne operacije:

<code>S.remove(e)</code>	uklanja e iz S ; ako e nije prisutan u S izaziva <code>KeyError</code>
<code>S.pop()</code>	vraća i uklanja proizvoljan element iz S ; ako je S prazan izaziva <code>KeyError</code>
<code>S.clear()</code>	uklanja sve elemente iz S

-operacije nad skupovima:

<code>S == T</code>	vraća True ako skupovi imaju jednak sadržaj
<code>S != T</code>	vraća True ako skupovi nemaju jednak sadržaj
<code>S <= T</code>	vraća True ako je S podskup od T
<code>S < T</code>	vraća True ako je S pravi podskup od T
<code>S >= T</code>	vraća True ako je S nadskup od T
<code>S > T</code>	vraća True ako je S pravi nadskup od T
<code>S.isdisjoint(T)</code>	vraća True ako su S i T disjunktni

IMPLEMENTACIJA SKUPA: STRUKTURA PODATAKA

-pomoću liste, upotreba memorije je (n) , `__contains__` je $O(n)$

-pomoću hash tabele, hash tabela čuva samo ključeve tj. Elemente, `__contains__` je $O(1)$

?Za koje vreme rade presek i unija?

-Performanse preseka dva skupa ako koristimo hash tabelu za predstavljanje skupa (Kako izgleda postupak spajanja dve hash tabele?), npr imamo skup S koji ima n elemenata, skup T koji ima m elemenata, koliko operacija nam treba da pronadjemo uniju ova dva skupa? Imamo dve varijante, jedna varijanta je da pravimo novi skup R koji će predstavljati uniju(iz S i T prebacimo sve elemente u R, u tom slučaju složenost operacije će biti $O(n+m)$, a imamo drugu mogućnost da skup S proširimo skupom T ili čak da uporedimo n i m , i da nadjemo koji od ta dva je manji, i da onda taj skup, njegov pridruženi, sve elemente prebacimo u drugi skup, npr ako je $n < m$, onda možemo u skup T, prebaciti sve elemente iz skupa S, potpuno je svejedno(složenost operacije odgovaraće manjem elementu izmedju m i n)

-Što se tiče preseka, slično je

11. OBLAST- Stabla pretrage

MAPE SA PORETKOM

-postoji relacija poretka nad ključevima

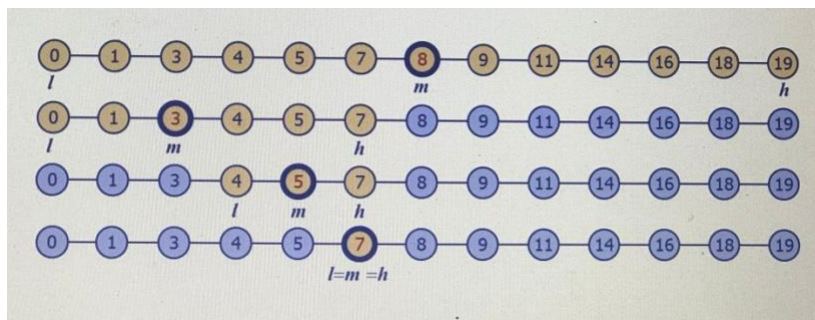
-elementi se skladište prema vrednosti ključa

-pretrage „najbliži sused“ (nearest neighbor): nađi element sa najvećim ključem manjim ili jednakim k , nađi element sa najmanjim ključem većim ili jednakim k

BINARNA PRETRAGA

-binarna pretraga može da pronade „najbližeg suseda“ za mapu sa poretkom implementiranu pomoću niza koji je sortiran po ključu , u svakom koraku prepolovi se broj kandidata, radi u $O(\log n)$ vremenu

-Primer nadji 7, krećemo od stedišnjeg elementa, pristupano, zakljucujemo da je 8, i 7 je sigurno sa leve strane, onda polovimo tu prvu polovinu kolekcije, onda pronalazimo novog kandidata to je 3, 7 je veće od 3, pa onda zanemarujemo elemente sa leve strane i samu trojku, onda posmatrano isečak izmedju 3 i 8, polovimo, broj je 5, 7 je veći od 5, pa ne gledamo sa leve strane, itd... i tako završavamo dok ne pronadjemo element koji tražimo, ili nam se isečak svede na veličinu nula ili manju, tako zakljucujemo da trazenog elementa nema



-Ovaj algoritam nije previše vezan za stabla ni za mape, nego se može generalno koristiti samo za pronalaženje elemenata u sortiranoj kolekciji, složenost $O(\log n)$

-Što se tiče implementacije, beleže se indeksi početka i kraja, sredina-saberu se i podele sa 2, i onda ako utvrdimo da je element levo, onda gornja granica se smanjuje za 1 itd..

TABELA PRETRAGE

- tabela pretrage je mapa sa poretком implementirana pomoću sortiranog niza, eksterni komparator za ključeve
- performanse: binarna pretraga je $O(\log n)$, dodavanje je $O(n)$, uklanjanje je $O(n)$
- radi efikasno samo za mali broj elemenata ili tamo gde je pretraga česta a izmene retke (npr. provera kreditne kartice)
- Ovo ima smisla ako imamo mali broj elemenata i ne želimo da implementiramo kompleksne strukture podataka, i ako retko dodajemo, uklanjamo..

SORTIRANA MAPA ATP

-standardne operacije:

-M[k] vraća vrednost v za ključ k u mapi M ; implementira je `__getitem__`

-M[k]=v dodaje novi element (k, v) u M ili menja postojeći; implementira je `__setitem__`

-del M[k] uklanja element sa ključem k iz M ; implementira je `__delitem__`

*dodatne funkcionalnosti: sortiran redosled prilikom iteracije, nađi veće: `find_gt(k)`, nađi u opsegu: `find_range(start, stop)`

BINARNO STABLO PRETRAGE

-binarno stablo pretrage je binarno stablo koje čuva (k, v) parove u čvorovima p tako da važi: - ključevi koji se nalaze u levom podstablu od p su manji od k

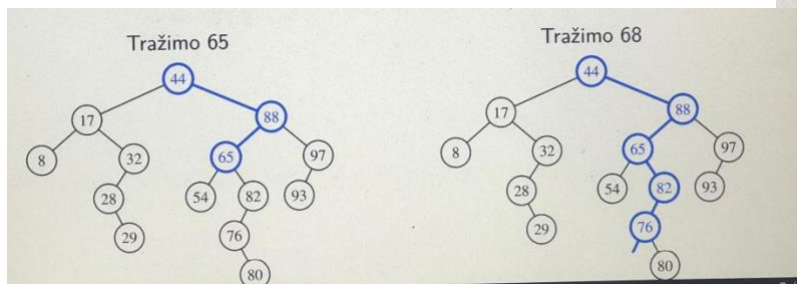
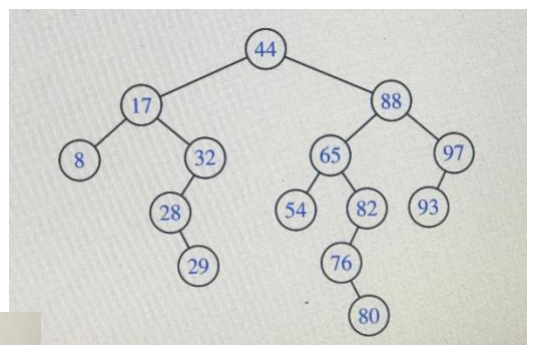
-ključevi koji se nalaze u desnom podstablu od p su veći od k

- listovi ne čuvaju elemente, reference na listove mogu biti None

- inorder obilazak: ključevi u rastućem redosledu

PRETRAGA U BINARNOM STABLU

-tražimo ključ k polazeći od korena, idemo levo ako je k manji od tekućeg čvora, idemo desno ako je k veći od tekućeg čvora, ako dođemo do lista, k nije nađen



PERFORMANSE PRETRAGE U BINARNOM STABLU

-u svakom rekurzivnom pozivu spuštamo se za jedan nivo, u stablu testiranje u okviru jednog nivoa je (1), ukupan broj testova je $O(h)$, gde je h visina stable

DODAVANJE U STABLO

-dodajemo element (k, v) , prvo tražimo k , ako k nije u stablu, došli smo do lista gde treba dodati čvor

UKLANJANJE IZ STABLA

- pronadjemo element gde se nalazi i onda kreće uklanjanje

- Imamo 3 slučaja: kada uklanjamo list stabla, kada uklanjamo čvor koji ima samo jedno dete, i kad uklanjamo čvor koji ima dvoje dece

- kada ima najviše jedno dete- njegovo dete vežemo na mesto čvora koji se uklanja

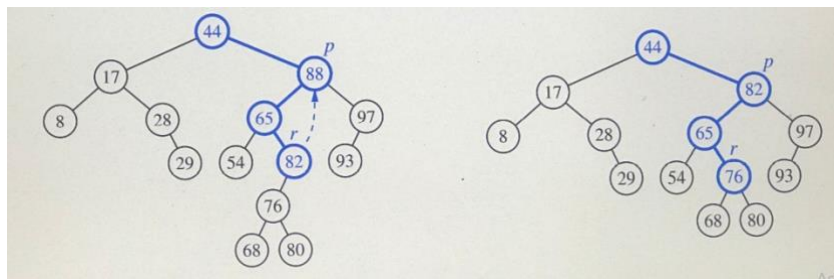
-ako p ima dva deteta:

- nađemo čvor r čiji ključ neposredno prethodi p – to je „najdesniji“ čvor u njegovom levom podstablu

-vežemo r na mesto p ; pošto r neposredno prethodi p po vrednosti ključa, svi elementi u desnom podstablu od p su veći od r i svi elementi u levom podstablu od p su manji od r

-treba još obrisati stari r – pošto je to „najdesniji“ element, on nema desno dete, pa se može obrisati po prethodnom algoritmu

-primer: uklanjamo 88



*Performanse: zauzeće memorije je (n) pretraga, dodavanje i uklanjanje su $O(h)$, visina stabla h je $O(\log n) \leq h \leq O(n)$

*balansirano stablo ima bolje performance

BALANSIRANJE BINARNOG STABLA

*osnovna operacija za balansiranje je **rotacija**

- „rotiramo“ dete i njegovog roditelja

-tom prilikom i podstabla menjaju mesta

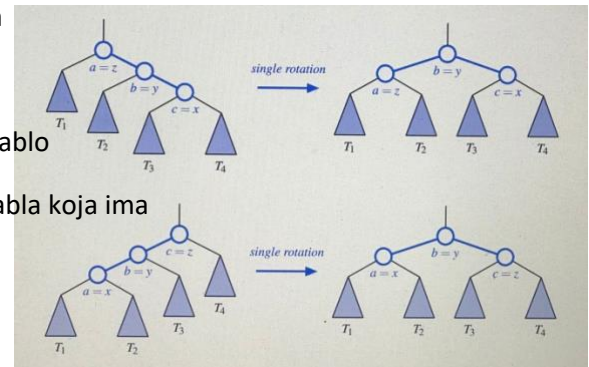
- jedna rotacija traje $O(1)$

* postoji i kompozitna operacija „**restrukturiranje tri čvora**“ (tri-node restructuring)

- posmatraju se čvor, njegovo dete i unuča
- cilj je da se skрати putanja od čvora do unučeta
- četiri moguća rasporeda čvorova: prva dva traže jednu rotaciju, druga dva traže dve rotacije

RESTRUKTURIRANJE SA JEDNOM ROTACIJOM

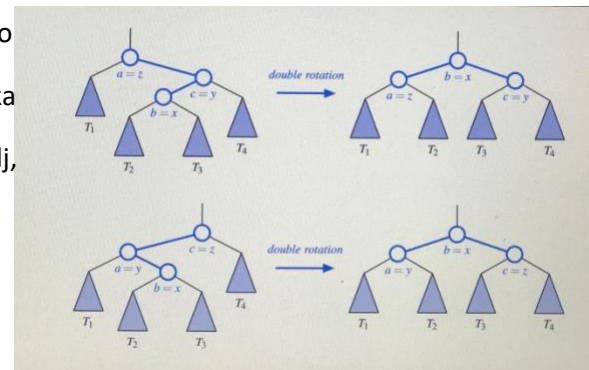
-Prvi slučaj: dedino desno dete je roditelj, pa desno dete roditelja je naš čvor c, ovu situaciju rešavamo tako što roditelj postaje novi koren stabla, dedi ostaje podstablo T1 i dodaje mu se podstablo T2, roditelj postaje roditelj dede i unuka, a unuk zadržava dva stabla koja ima



-Drugi slučaj – slično..

RESTRUKTURIRANJE SA DVE ROTACIJE

- Prvi slučaj: u krajnjoj liniji unuk postaje koren podstabla, njegovo levo dete postaje deda, dedi ostaje podstablo T1 i dobija od unuka levo podstablo T2, a u desnom podstablu unuka se dodaje roditelj, ostaje mu T4 i pridodaje mu se desno podstablo unuka.



-Drugi slučaj: slično...

AVL STABLO

- u pitanju je balansirano binarno stablo, autori: G.M. Adelson-Velskii i E. Landis
- stablo je balansirano ako za svaki čvor važi da je razlika u visini njegove dece najviše jedan
- visina podstabla: broj čvorova na najdužoj putanji od korena do lista
- visina čvora = visina podstabla sa njim kao korenom
- AVL** stablo je binarno stablo koje ima dodatnu osobinu:
 - za svaki čvor u stablu, visine njegove dece razlikuju se najviše za 1

VISINA AVL STABLA

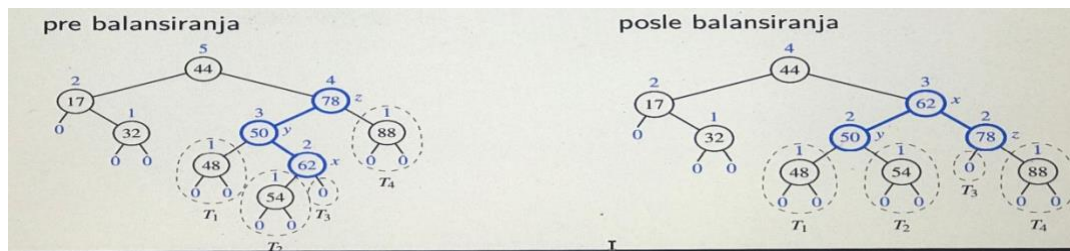
- teorema: visina AVL stabla sa n čvorova je $O(\log n)$
- dokaz: $n(h)$ – najmanji broj unutrašnjih čvorova u AVL stablu visine h

AVL STABLO –DODAVANJE

- stablo u koje dodajemo novi čvor je AVL stablo
- dodavanje se vrši isto kao kod binarnog stabla – u list

- dodavanje može da naruši balansiranost čvorovi koji mogu biti disbalansirani su samo preci novog čvora

- primer: dodajemo čvor 54



- za svaki čvor do kojeg dodjemo, moramo da proverimo da li važi pravilo visine – 1, i onda primenimo tehniku balansiranja

- „search-and-repair“ strategija, radimo trinode restructuring

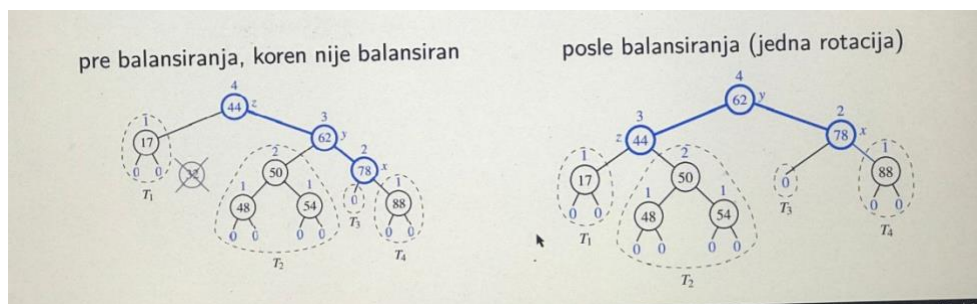
- dodavanje utiče na promenu visine, remeti balans

AVL STABLO – UKLANJANJE

- uklanjanjem se može narušiti balans AVL stable

- i ovde radimo restrukturiranje posle uklanjanja

- primer: uklanjam 32



- Iako deluju komplikovano, veoma su jednostavne, jedno restrukturiranje je (1)

- pretraga je $O(\log n)$ – visina stabla je $O(\log n)$

- dodavanje je $O(\log n)$:

- pronalaženje mesta je $O(\log n)$

- restrukturiranje uz stablo je $(\log n)$

- uklanjanje je $O(\log n)$:

- pronalaženje mesta je $O(\log n)$

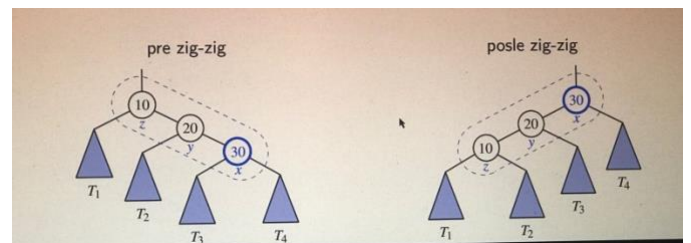
- restrukturiranje uz stablo je $(\log n)$

SPLAY STABLO

- splay: „rašireno“, ne nameće logaritamsko ograničenje na visinu
- splaying: „širenje“ stabla prilikom dodavanja, uklanjanja i pretrage
- ideja: da češće korišćeni elementi budu bliže korenu
- čvor x se premešta u koren nizom restrukturiranja
- operacije restrukturiranja zavise od položaja x , y (roditelja) i z (dede, ako postoji)
- postoje tri slučaja:
 1. zig-zig
 2. zig-zag
 3. zig

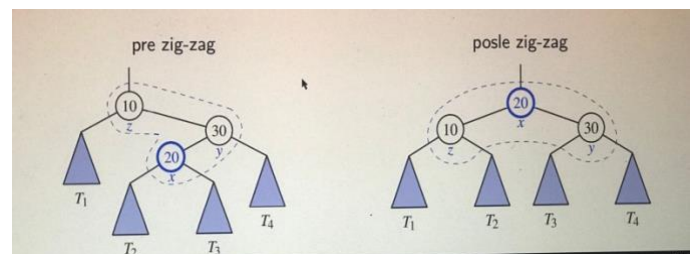
ZIG-ZIG

- x i y su: obojica levo dete svog roditelja ili obojica desno dete svog roditelja
- x postaje koren, y njegovo dete, z njegovo unuče



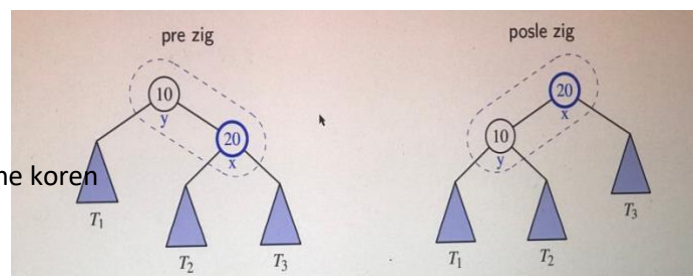
ZIG-ZAG

- x i y prvi je levo dete a drugi je desno dete, ili prvi je desno dete a drugi je levo dete
- x postaje koren, y i z njegova deca



ZIG

- x ima roditelja y ali nema dedu z
- x postaje koren, y njegovo dete
- * zig-zig, zig-zag i zig primenjujemo sve dok x ne postane koren



SPLAY STABLO: PREFORMANSE

- zig-zig, zig-zag i zig su $O(1)$
- splaying čvora p je (d) gde je d dubina čvora p
- tj. isto koliko je potrebno i za navigaciju od korena do p
- u najgorem slučaju, pretraga, dodavanje i uklanjanje su $O(h)$ gde je h visina stable
- stablo nije balansirano \Rightarrow može biti $h = n \Rightarrow$ slabe performanse u najgorem slučaju
- za **amortizovane operacije** vreme je $O(\log n)$

- a za često tražene podatke pretraga je i **brža** od $O(\log n)$

N-ARNO STABLO

- neka je w čvor stabla; ako w ima d dece zovemo ga d -čvor

- n-arno stablo pretrage ima sledeće osobine:

-svaki unutrašnji čvor ima bar dva deteta, tj. svaki je d -čvor za $d \geq 2$

-svaki unutrašnji d -čvor sa decom c_1, c_2, \dots, c_d čuva $d - 1$ parova $(k_1, v_1), \dots, (k_{d-1}, v_{d-1})$

-za $k_0 = -\infty, k_d = +\infty$ važi: za svaki element (k, v) iz podstabla od w kome je koren c_i važi da je $k_{i-1} \leq k \leq k_i$

- npr imamo ovaj čvor 5 i 10, i ima troje dece, kad dodjemo do tog čvora

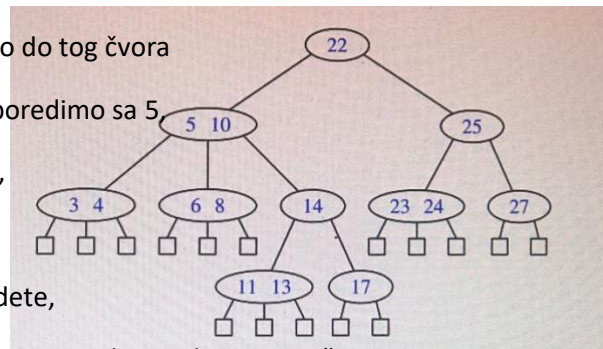
poredimo ključ 5 sa ključevima sa kojima se nalaze, prvo poredimo sa 5,

ako je traženi ključ manji od ključa 5, idemo na prvo dete,

ako je vrednost ključa veća od 5, poredimo je sa ključem

10 koji je u čvoru, a ako je manje od 10, idemo na drugo dete,

ako je veće od 10 idemo na treće dete. Dakle gde ćemo ići, zavisi u kom odnosu je naš traženi ključ sa ključevima koji se nalaze u jednom čvoru



- pretraga unutar čvora?

-treba nam **sekundarna struktura podataka**-

-binarna pretraga po nizu je $(\log d)$

-sortirana mapa

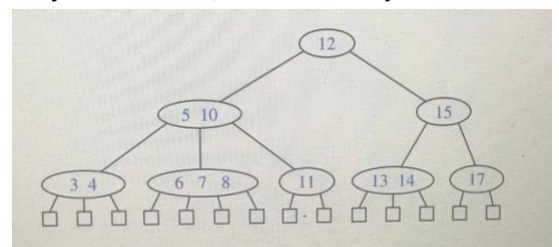
- pretraga u stablu je $O(h \log d_{\max})$ (d_{\max} element sa najvećom vrednošću)

(2,4) STABLO

- (2,4) stablo je n-arno stablo sa dve osobine: unutrašnji čvor ima najviše 4 deteta, svi listovi imaju istu dubinu

- svaki čvor ima 2, 3 ili 4 deteta

- visina stabla od n elemenata je $O(\log n)$



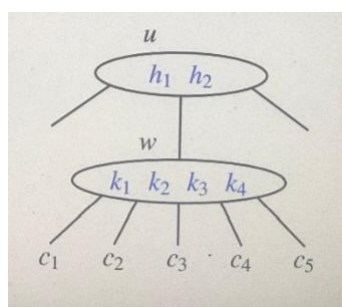
(2,4) STABLO – DODAVANJE

-Npr ako želimo da dodamo element 9, idemo do ključa 6,7,8 i vidimo da ako dodamo ključ 9, onda će taj čvor imati 5 dece, što znači da mora doći do restrukturiranja

-prvo tražimo ključ k

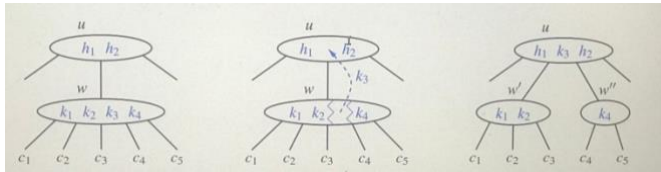
-neuspešna pretraga se završava u listu

- dodamo k u roditelja w tog lista



- (prelivanje, overflow): ako je taj roditelj bio 4-čvor, sada je 5-čvor; moramo ga podeliti (split):

*Deljenje se obavlja tako što se prva dva ključa (k_1, k_2) prebacuju u novi čvor, levi, ključ k_3 ide u roditelja, a k_4 ide u novi desni čvor



-medjutim šta ako je roditelj imao već troje dece, onda bi četvrto dodavanje izazvalo i cepanje roditelja-podela može biti kaskadna!

(2,4) STABLO – UKLANJANJE

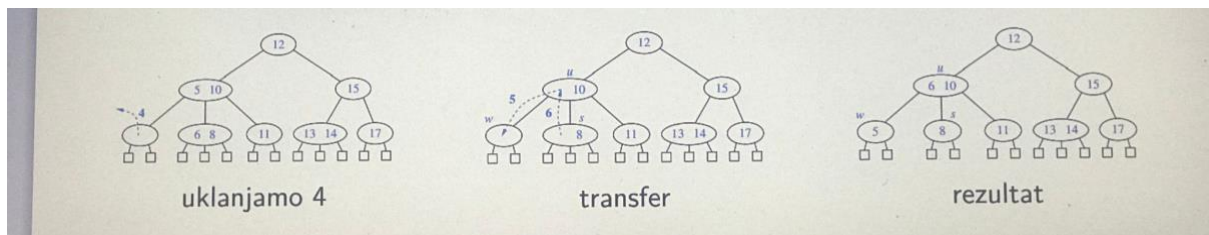
*prvi slučaj: uklanjanjem ključa ne narušavaju se osobine (2,4) stable – samo se ukloni

*drugi slučaj: uklanjanje iz w izaziva **underflow**

-da li je jedan od najbliže braće 3-čvor ili 4-čvor?

-radimo **transfer**: premeštamo ključ iz brata u roditelja, ključ iz roditelja u w

-primer: uklanjanje 4

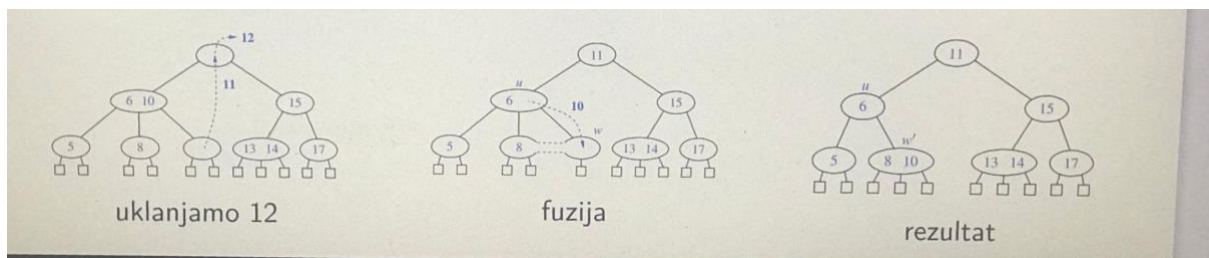


*treći slučaj: uklanjanje izaziva **underflow**

-nijedan od najbliže braće nije 3-čvor ili 4-čvor

radimo fuziju: spajamo w sa bratom, ključ iz roditelja spuštamo u spojeni čvor

-primer: uklanjanje 12



-Na njegovo mesto dodajemo najbliži manji ključ, kako pronalazimo najbliži manji čvor? Idemo u levo podstablo, pa onda desno desno, dokle god možemo. Čvor 11 prebacujemo u čvor 12, a onda čvor koji je ostao prazan gledamo da li možemo da pozajmimo od brata, u ovom slučaju brat je sam, što znači da prazan čvor spajamo sa čvorom brata- fuzija

? Kako se spajanje izvrši? Pozajmljujemo jedan ključ od roditelja, iz roditelja se najveći ključ spušta u novi spojeni ključ (jer nam ostaje jedna veza praznog čvora i onda spajanjem sa bratom, ima 3 veze, i onda zato mora roditelj da se spusti)

-Šta se dešava ukoliko se uspostavi da roditelj ne može da nam pozajmi jedan ključ, onda nastavljamo dalje sa fuzijom, odnosno fuzija se propagira ka korenu

-fuzija može da propagira underflow

-ako se koren isprazni fuzijom, prosto se obriše

-smanjuje se visina, stalno upoređujemo da li možemo da pozajmimo ključ od brata

*primer dodavanja: 53, 97, 50, 48, 83, 69, 64, 80, 73, 87, 71, 84, 65, 44, 18, 88

1. stablo je inicijalno prazno, kreiramo koren i dodajemo prvi ključ u njega – 53

2. 97 dodajemo u koren, ima mesta 53 97

3. 50 dodajemo u koren, ima mesta 50 53 97

4. 48 dodajemo u koren, imamo prelivanje 48 50 53 97 (Treći ključ postaje roditelj, prva dva prelaze u levo podstablo, a četvrti u desno podstablo)

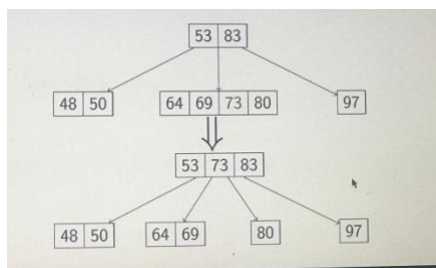
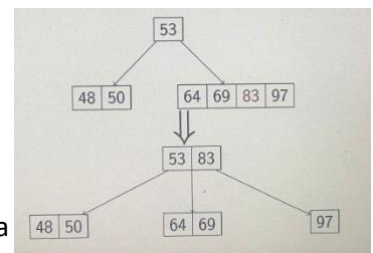
5. nakon prelivanja dodajemo 83, ima mesta (veće od 54, u desnom podstablu pored 97)

6. dodajemo 69, ima mesta (veće od 53, u desnom podstablu pored 83)

7. dodajemo 64, imamo prelivanje, 83 se penje u roditelja

8. dodajemo 80, ima mesta (80 je između 53 i 83, dodajemo pored 69)

9. dodajemo 73, imamo prelivanje, 73 se penje u roditelja, tamo ima mesta



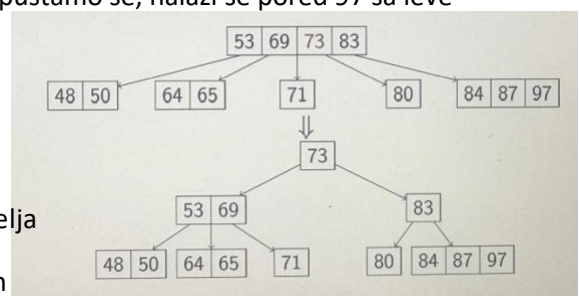
10. dodajemo 87, ima mesta (nalazi se između 73 i 83, spuštamo se, nalazi se pored 97 sa leve strane)

11. dodajemo 71, ima mesta (isto.. pored 69)

12. dodajemo 84, ima mesta (isto..pored 87)

13. dodajemo 65, imamo prelivanje, 69 penjemo u roditelja

14. sada u korenu imamo prelivanje i pravimo novi koren



-(2,4) stablo povećava broj nivoa kada se desi prelivanje u korenu

- raste „iz korena“ umesto „kroz listove“

- u prethodnom koraku je 73 postao novi koren nakon prelivanja

15. dodajemo 44, ima mesta(ima mesta, pored 48)

16. dodajemo 18, imamo prelivanje, 48 ide u roditelja (čvor 18 44 48 50).... Itddd

(2,4) STABLO: PERFORMANSE

-dodavanje u (2,4) stablu sa n elemenata:

- visina stabla je $O(\log n)$
- traženje ključa je $O(\log n)$
- dodavanje novog ključa u čvor je (1)
- svaka podela čvora je (1)
- ukupan broj podela je $(\log n)$

⇒ dodavanje je $O(\log n)$

-uklanjanje u (2,4) stablu sa n elemenata:

- visina stabla je $O(\log n)$
- traženje ključa je $(\log n)$
- uklanjanje ključa je (1)
- može da usledi $(\log n)$ iza kojih je max 1
- transfer fuzija i transfer su (1)

⇒ uklanjanje je $O(\log n)$

	search	insert	delete	napomene
hash tabela	1 očekivano	1 očekivano	1 očekivano	nema metode za sortiranu mapu; jednostavna implementacija
skip lista	$\log n$ verovatno	$\log n$ verovatno	$\log n$ verovatno	randomized insert; jednostavna implementacija
AVL i (2,4)	$\log n$ najgore	$\log n$ najgore	$\log n$ najgore	komplikovana implementacija

CRVENO-CRNO STABLO

-red-black stablo je reprezentacija (2,4) stabla pomoću binarnog stabla čiji čvorovi su obojeni crveno ili crno

-u poređenju sa (2,4) stablom, RB stablo ima jednake $O(\log n)$ performance, jednostavniju implementaciju

-crveno-crno stablo je binarno stablo pretrage sa osobinama:

- koren je **crn**
- deca **crvenog** čvora su **crna**
 - deca **crnog** čvora ne moraju biti **crvena**!
- sve putanje od korena do lista sadrže isti broj crnih čvorova
 - **crna dubina čvora**: broj crnih predaka (računajući i dati čvor)

*posledice:

- putanja od korena do najdaljeg lista nije više od duplo duža od putanje do najbližeg lista
- RB-stablo je prilično dobro balansirano
- visina RB stabla sa n elemenata je $O(\log n)$
- pretraga na isti način kao za binarno stablo
- pretraga je takođe $O(\log n)$

CRVENO-CRNO STABLO: DODAVANJE

- dodavanje na isti način kao za binarno stablo
- novi čvor se boji u crveno ako nije koren; koren je uvek crn
- ako je roditelj novog čvora crven tada imamo duplo crveno – potrebna je reorganizacija stable

[add] Korekcija duplog crvenog: stric je crn

-Situacija u kom dodajemo novi čvor koji je crven na već postojaći čvor koji je crven, to su sledeća 4 slučaja.

*slučaj 1: brat roditelja novog čvora je crn ili ne postoji

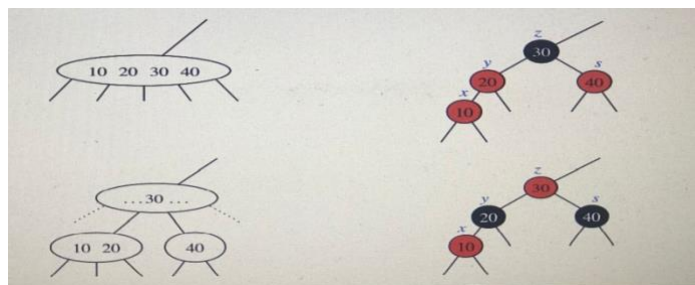
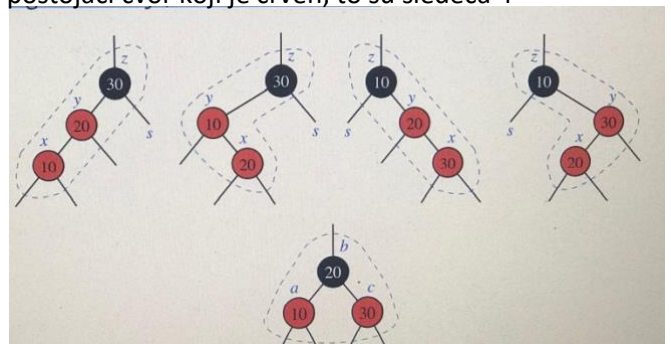
- radimo tri-node restructuring
- četiri moguće situacije, sa istim rezultatom:
- a,b,c konačan oblik sve 4 situacije

*slučaj 2: brat roditelja novog čvora je crven

-radimo **recoloring**:

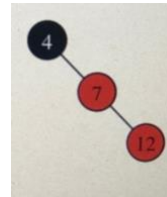
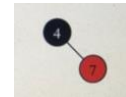
- obojimo roditelja i strica u crno a dedu u crveno (ako je deda koren, ostaje crn)

-moguća propagacija uz stablo



PRIMER:

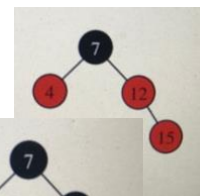
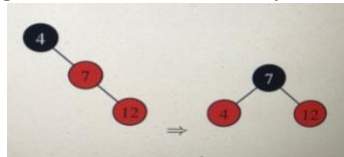
1. stablo je na početku prazno, dodajemo 4, on će biti koren bojimo ga u crno
2. dodajemo 7 kao desno dete od 4, bojimo ga u crveno, nije potrebno restrukturiranje



3. dodajemo 12 kao desno dete od 7, bojimo ga u crveno, imamo duplo crveno

a) 12 nema strica: slučaj 1

-radimo tri-node restructuring

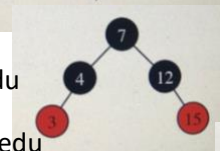
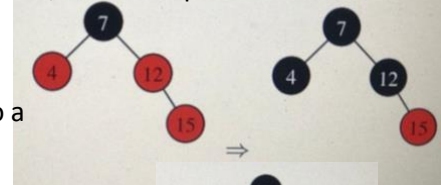


4. dodajemo 15 kao desno dete od 12, bojimo ga u crveno, imamo duplo crveno

a) 15 ima crvenog strica: slučaj 2

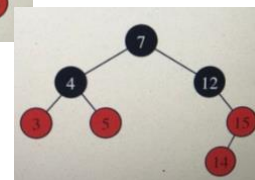
-radimo recoloring: obojimo oca i strica u crno a

dedu u crveno osim ako je koren



5. dodajemo 3 kao levo dete od 4, bojimo ga u crveno, sve je u redu

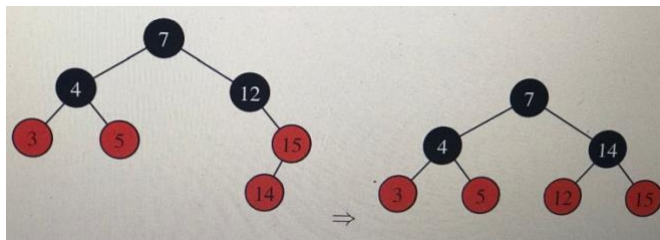
6. dodajemo 5 kao desno dete od 4, bojimo ga u crveno, sve je u redu



7. dodajemo 14 kao levo dete od 15, bojimo ga u crveno, imamo duplo crveno

a) 14 nema strica: slučaj 1

-radimo tri-node restructuring za 12-15-14

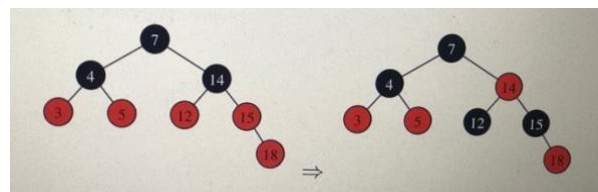


8. dodajemo 18 kao desno dete od 15, bojimo ga u crveno, imamo duplo crveno

a) 18 ima crvenog strica: slučaj 2

- radimo recoloring: oca i strica u crno

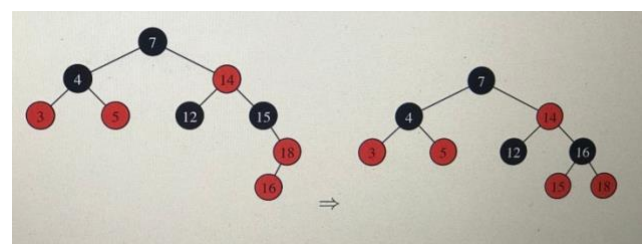
a dedu u crveno



9. dodajemo 16 kao levo dete od 18, bojimo ga u crveno, imamo duplo crveno

a) 16 nema strica: slučaj 1

-radimo tri-node restructuring za 15-18-16

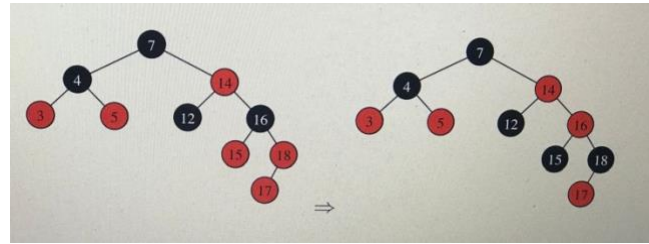


10. dodajemo 17 kao levo dete od 18, bojimo ga u crveno, imamo duplo crveno

a) 17 ima crvenog strica: slučaj 2

-radimo recoloring:

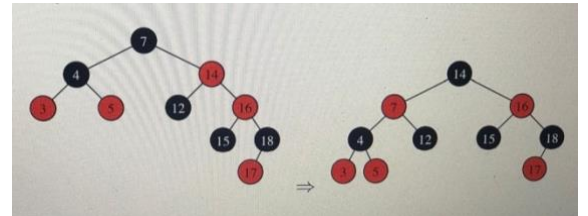
oca i strica u crno a dedu u crveno



b) sada je deda (16) u problemu - imamo duplo crveno

-16 ima crnog strica: slučaj 1

radimo tri-node restructuring za 7-14-16



CRVENO-CRNO STABLO : UKLANJANJE

-najkompleksnije

-uklanjanje na isti način kao za binarno stablo \Rightarrow uklanja se čvor koji ima najviše jedno dete

-ako je uklonjeni čvor bio crven sve je OK (ne menja se crna dubina)

-ako je uklonjeni čvor bio crn:

- ako je uklonjeni čvor list - njegov brat je koren podstabla sa crnom dubinom 1

-ako mu je dete crveno: pomeramo dete na mesto uklonjenog roditelja i bojimo crno

-ako mu je dete crno: gubimo jedan crni čvor na putanji, narušava se crna dubina u tom podstablu

KOREKCIJA CRNE DUBINE

-uklonjeni crni čvor je bio koren podstabla T_{light}

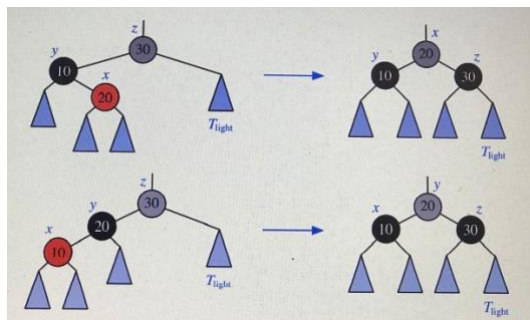
- gledamo šta se nalazi u podstablu njegovog brata y

(brat mora da postoji jer bi inače bila narušena crna dubina)

*Korekcija crne dubine 1: brat je crn i ima crveno dete

-slučaj 1: brat y je crn i ima crveno dete x

-radimo **tri-node restructuring**

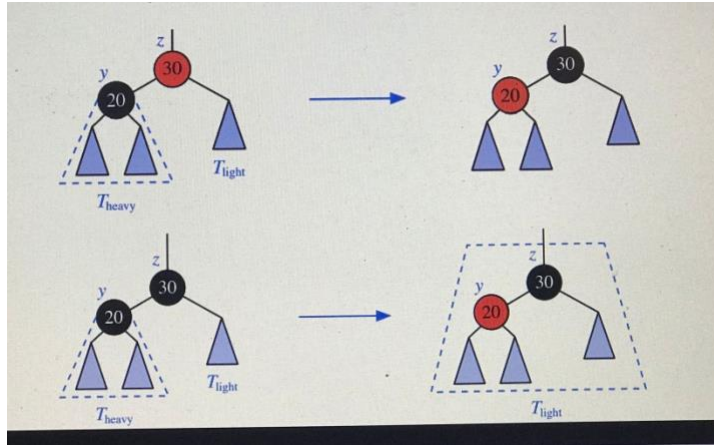


*Korekcija crne dubine 2: brat je crn i ima dva crna deteta

-slučaj 2: brat y je crn i oba deteta su mu crna ili ih nema

-radimo **recoloring**

- ako je z bio crven, tu je kraj; ako je bio crn, recoloring propagira prema gore

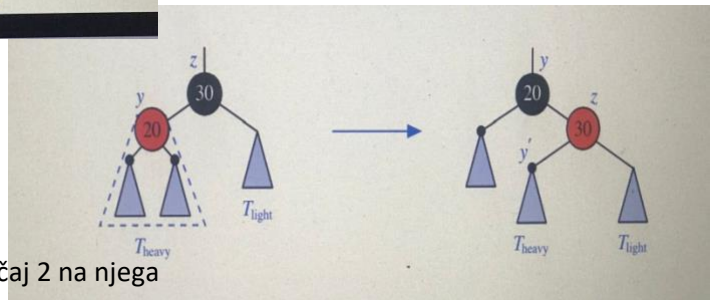


*Korekcija crne dubine 3: brat je crven

-slučaj 3: brat y je crven

-radimo **rotaciju** pa **recoloring**

-čvor y' je crn pa treba primeniti slučaj 1 ili slučaj 2 na njega



PRIMER UKLANJANJE

1. uklanjamo 3,3 je crven: ne menja se crna dubina

2. uklanjamo 12 - to je crni list

-njegov brat je crn i ima crveno dete

- slučaj 1: radimo tri-node restructuring

a) slučaj 1: radimo tri-node restructuring

za 7-4-5

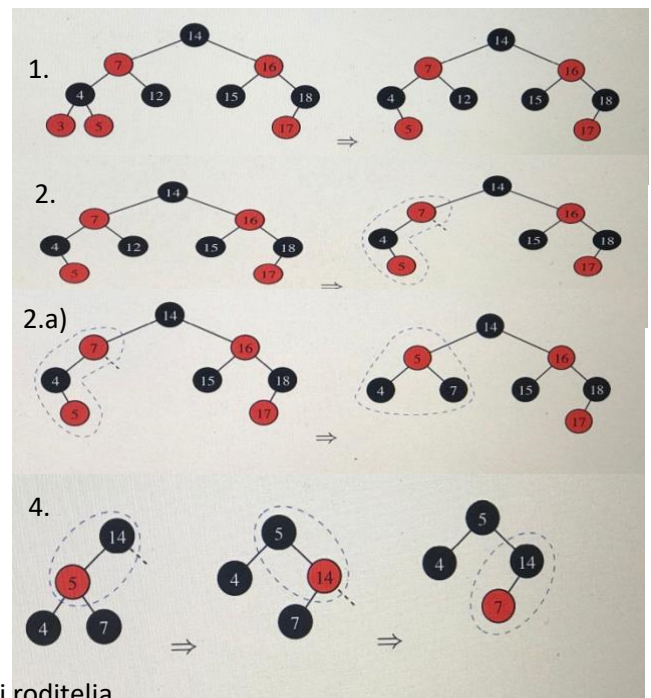
3. uklanjamo 17 - to je crveni list (samo uklonimo)

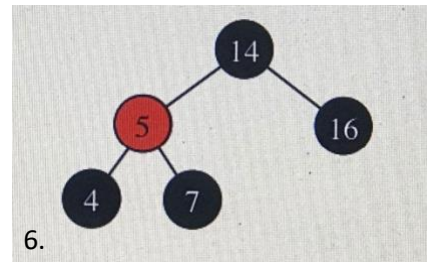
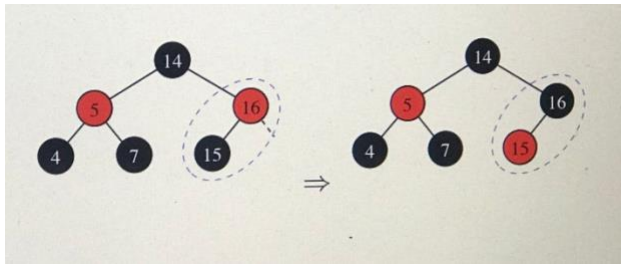
4. uklanjamo 18 - crni list

-njegov crni brat nema dece

-slučaj 2: recoloring - menjamo boju brata i roditelja

a) slučaj 2: recoloring - menjamo boju brata i roditelja





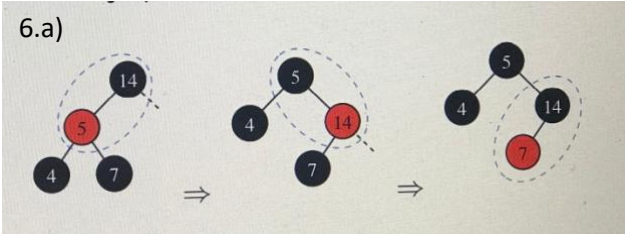
5. uklanjamo 15 - crveni list (samo uklonimo)

6. uklanjamo 16 - crni list

- njegov brat 5 je crven: slučaj 3

a) slučaj 3: radimo rotaciju pa recoloring

6.a)



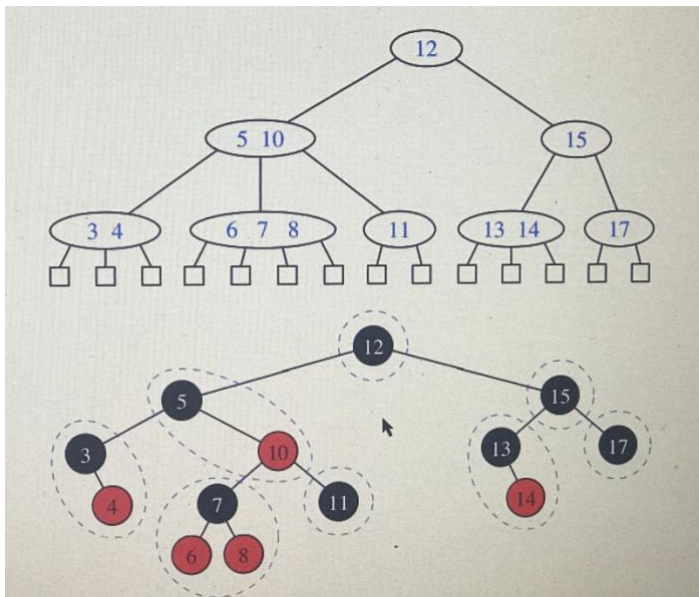
CRVENO-CRNO STABLO: PERFORMANCE

-jednake kao i za (2,4) stablo

-sve operacije su ($\log n$)

- dodavanje i uklanjanje zahtevaju konstantan broj operacija restrukturiranja

-bojanje je zapravo cepanje čvora u (2,4) stablu



12. OBLAST – SELEKCIJA I SORTIRANJE

SORTIRANJE

-**sortiranje**: izmena redosleda elemenata u kolekciji tako da budu poređani od najmanjeg ka najvećem

- videli smo da red sa prioritetom može da posluži za sortiranje **selection sort** je $O(n^2)$, **insertion sort** je $O(n^2)$, **heap sort** je $O(n \log n)$

MERGE SORT

- merge sort je primer **divide-and-conquer** šablona
- divide**: podeli S na dva disjunktne podskupa S_1 i S_2
- recur**: reši potproblem za S_1 i S_2
- conquer**: kombinuj rešenja za S_1 i S_2 u rešenje za S
- bazni slučaj za rekurziju je skup veličine 0 ili 1
- merge sort je rekurzivna
- merge sort je $(n \log n)$, kao i heap sort
- za razliku od heap sorta ne koristi pomoćnu strukturu podataka (RSP), pristupa podacima sekvencijalno – pogodno za sortiranje podataka na disku

MERGE SORT-ALGORITAM

- sortira sekvencu S dužine n u tri koraka

1 divide: podeli S na S_1 i S_2 , svaki dužine $n/2$

2 recur: rekurzivno sortiraj S_1 i S_2

3 conquer: spoj sortirane S_1 i S_2 u sortiranu sekvencu – kreće se od indeksa 0, pa upoređujemo koji je manji i njega stavljamo u novu sekvencu, onda poredimo sledeći indeks i tako dok ne stignemo do kraja, ako poredimo dva broja koja su na različitim indeksima, u novu sekvencu stavićemo manji element na indeks koji je u zbiru indeksa ta dva broja koja poredimo. Ponavljamo postupak dok se prva kolekcija ne isprazni, ili druga.

?Šta se dešava kada dođemo do kraja neke kolekcije? Samo nadovežemo sve što je ostalo od druge.

STABLO SORTIRANJA

- izvršavanje merge sorta može se prikazati binarnim stablom

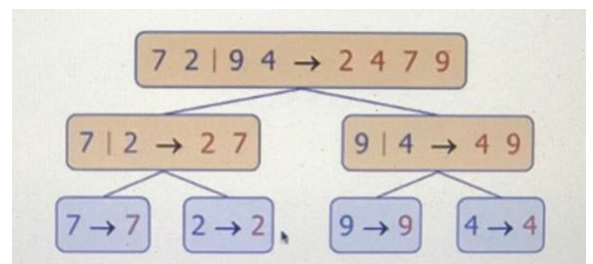
- čvor stabla predstavlja jedan rekurzivni poziv i čuva

- nesortiranu sekvencu pre podele
- sortiranu sekvencu nakon završetka

- koren je početni poziv funkcije

- listovi su pozivi sa podsekvence dužine 0 ili 1

-Pozivano nad 7 2 9 4 polovljenje, pozivamo merge sort algoritam nad 7 i 2 i jos jedan nad 9 i 4, onda delimo 7 i 2 i pozivamo merge sort nad 7 i nad 2, bazni slučajevi su plavi, trivijalno su sortirani 7 sa 7, 2 sa 2, dolazi do faze spajanja merge faze, gde spajano 7 i 2, i dobijamo 2 i 7



MERGE SORT- PERFORMANSE

- visina h stabla za merge sort je $(\log n)$

- delimo sekvencu na pola za svaku rekurziju

- ukupan broj operacija na nivou i je (n)
- delimo i spajamo 2^i sekvenci dužine $n/2^i$
- pravimo 2^{i+1} rekurzivnih poziva
- ukupno vreme izvršavanja je $O(n \log n)$

algoritam	vreme	napomene
selection	$O(n^2)$	<ul style="list-style-type: none"> ▷ spor ▷ in-place ▷ za male sekvence ($< 1K$)
insertion	$O(n^2)$	<ul style="list-style-type: none"> ▷ spor ▷ in-place ▷ za male sekvence ($< 1K$)
heap	$O(n \log n)$	<ul style="list-style-type: none"> ▷ brz ▷ in-place ▷ za velike sekvence (1K-1M)
merge	$O(n \log n)$	<ul style="list-style-type: none"> ▷ brz ▷ sekvencijalan ▷ za ogromne sekvence ($> 1M$)

*nerekutivna varijanta -Izdelimo našu kolekciju na pojedinačne elemente, a onda primenjujemo merge na prva dva, pa sledeće dva itd, onda ćemo ona prva dva da spojimo sa sledeća dva, da dobijemo dva elementa itd... za nijansu je brža od rekurzivne

QUICK SORT

-To znači da se proizvoljno odabere jedan element (nemamo pojma koji je to element i nemamo pojma u kakvom je on odnosu sa ostalim elementima naše kolekcije, takav element zovemo pivot i cilj nam je da elemente raspodelimo tako što se sa leve strane nalaze manji elementi od pivotu, a sa desne veći, na levu i desnu kolekciju primenjujemo quick sort i tako naša kolekcija dolazi u sortirani oblik

-quick sort je randomized podeli-pa-vladaj algoritam

1. divide: izaberi slučajni element x (pivot) i podeli S na

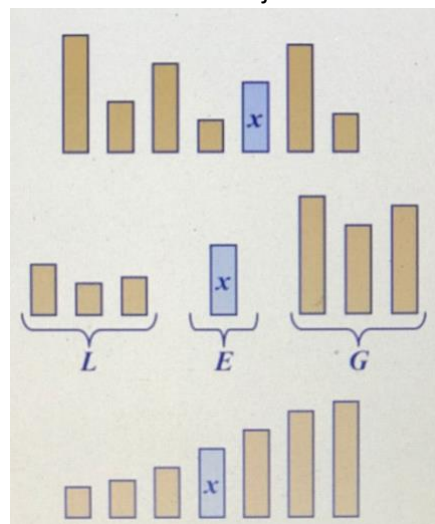
L : elementi manji od x

E : elementi jednaki x

G : elementi veći od x

2. recur: sortiraj L i G

3. conquer: spoj sortirane L , E i G

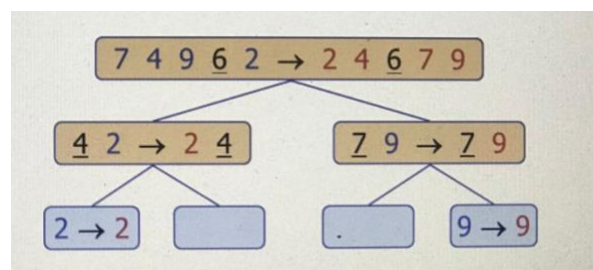


STABLO SORTIRANJA

-zvršavanje quick sorta može se prikazati binarnim stablom

- čvor stabla predstavlja jedan rekurzivni poziv i čuva

- nesortiranu sekvencu pre podele oko pivotu



- sortiranu sekvencu nakon završetka

-koren je početni poziv funkcije

-listovi su pozivi sa podsekvence dužine 0 ili 1

-6 ne prebacujemo nigde, stavljamo u kolekciju jednakih elemenata, 4 i 2 u kolekciju manjih elemenata, 7 i 9 u kolekciju većih elemenata, onda pozivamo quick sort za obe kolekcije, u kolekciji manjih elemenata bira se pivot, to je 4, i ponavljamo istu stvar, manjih elemenata od pivota ima to je 2, većih nema tkd ovde imamo poziv quick sort algoritma za kolekciju dužine nula, što predstavlja bazni slučaj i posle toga imamo spajanje- treba samo da sve elemente iz manje kolekcije izlistamo i zatim dodajemo elemente iz jednake kolekcije i slepimo sve veće elemente

-Ovde je spajanje jednostavno, ali je problem u particionisanju ko ima $O(n)$ performance

PERFORMANSE U NAJGOREM SLUČAJU

-najgori slučaj: kada je pivot najveći ili najmanji element

-jedan od L ili G ima dužinu 0 a drugi dužinu $n - 1$

-vreme izvršavanja je tada proporcionalno sumi

$$n + (n - 1) + \dots + 2 + 1$$

\Rightarrow najgori slučaj je $O(n^2)$

-posmatrajmo rekurzivni poziv za sekvencu dužine s

* dobar izbor: dužine L i G su obe manje od $s \cdot \frac{3}{4}$

* loš izbor: L ili G ima dužinu veću od $s \cdot \frac{3}{4}$

-slučajan izbor pivota je dobar sa verovatnoćom $\frac{1}{2}$

* $\frac{1}{2}$ mogućih pivota su dobar izbor

*OČEKIVANE PERFORMANSE

-iz verovatnoće: očekivani broj bacanja novčića da bismo dobili k glava je $2k$

- za čvor dubine i očekujemo

- $i/2$ predaka su dobri izbori

-veličina ulazne sekvence za tekući poziv je najviše $n \cdot (3/4)^2$

-za čvor dubine $2 \log_{4/3} n$ očekivana veličina ulaza je 1

- očekivana visina stabla je $O(\log n)$

-broj operacija za čvorove iste dubine je (n)

\Rightarrow ukupno očekivano vreme quick sorta je $O(n \log n)$

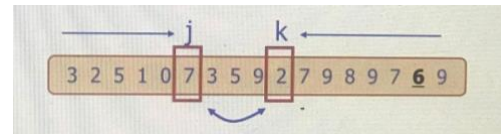
IN-PLACE PARTICIJA

- Na početku treba da znamo koji deo kolekcije uopšte razmatramo, od starta razmatramo kolekciju od indeksa 0, biramo jedan proizvoljan indeks, npr 6 je pilot
- Definišemo dva indeksa j i k , j ide od 0 pa naviše, k ide on $n-1$ pozicije o a naniže, onda poredino j i k ,
- ponavljaj dok se j i k ne mimoiđu ($k < j$)

pomeraj j u desno dok ne naiđemo na element $\geq x$

pomeraj k u levo dok ne naiđemo na element $< x$

zameni elemente na pozicijama i i k



algoritam	vreme	napomene
selection	$O(n^2)$	▷ in-place ▷ spor (dobar za male ulaze)
insertion	$O(n^2)$	▷ in-place ▷ spor (dobar za male ulaze)
heap	$O(n \log n)$	▷ in-place ▷ brz (dobar za velike ulaze)
merge	$O(n \log n)$	▷ sekvencijalan ▷ brz (dobar za ogromne ulaze)
quick	$O(n \log n)$ očekivano	▷ in-place, randomized ▷ najbrži (dobar za velike ulaze)

SORTIRANJE ZASNOVANO NA POREĐENJU

- mnogi algoritmi se zasnivaju na poređenju elemenata, porede se parovi elemenata (bubble, selection, insertion, heap, merge, quick...)

BROJANJE OPERACIJA POREĐENJA

- svako pokretanje algoritma odgovara jednoj putanji koren→list u **stablu odlučivanja**

STABLO ODLUČIVANJA

- visina stabla odlučivanja je donja granica za vreme sortiranja
- da bi odredili koliko parova ima, moramo da nađemo koliko permutacija od n elmenata postoji
- svaka permutacija ulaza vodi do posebnog lista
- stablo ima $n!$ listova \Rightarrow visina stabla je barem $O(\log(n!))$
- Ideja onda do koje dolazimo je da pogledamo odnosno da pokušamo da sa donje strane ograničimo ovu funkciju $\log(n!)$
- svaki algoritam koji se zasniva na poređenju je barem $(\log(n!))$
- prema tome, vreme izvršavanja svakog algoritma je najmanje
- tj. svaki ovakav algoritam je $\Omega(n \log n)$ – nije ova tvrdnja tačna u svakom slučaju, moguće je da uvedemo neka ograničenja kako bi dobili bolje performance

$$\log n! \geq \log \left(\frac{n}{2} \right)^{\frac{n}{2}} = \frac{n}{2} \log \frac{n}{2}$$

BUCKET SORT

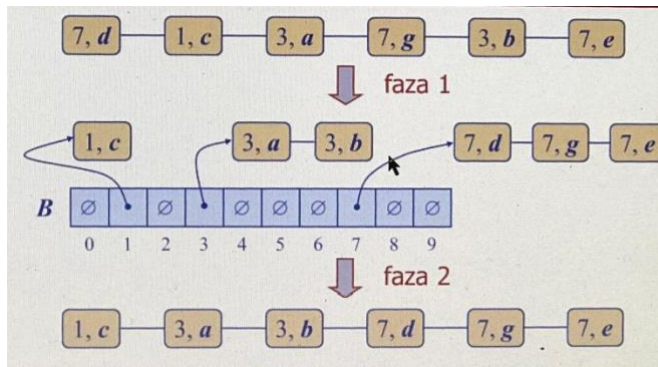
-ograničava vrednosti koje poredimo

- S je sekvenca n parova (ključ, element) sa ključevima u opsegu $[0, N - 1]$

- **bucket sort** koristi ključeve kao indekse u pomoćnom nizu sekvenci (kanti) B

faza 1: isprazni S premeštanjem svakog (k, o) u svoju kantu $B[k]$

faza 2: za $i = 0, \dots, N - 1$ premesti elemente kante $B[i]$ na kraj S



-Želimo da sortiramo elemente iz kolekcije po ključu, zauzimo 10 memorijskih lokacija ($[0, 9]$) i u svakoj memorijskoj lokaciji za početak upisujemo prazan element, kada 7 ubacujemo, ubacujemo na poziciju 7, čim element prebacimo na poziciju koja prethodno nije bila zauzeta, kreiramo bucket i ubacujemo naš element 7d, tako ubacimo sve.. Ono što možemo da primetimo je zapravo direktno mapiranje ključeva na memorijskim lokacijama

-analiza: faza 1 je (n)

faza 2 je $O(n + N)$

\Rightarrow bucket sort je $O(n + N)$

OSOBBINE BUCKET SORTA

-**tip ključa**: ključevi su isključivo integeri

-**stabilno sortiranje**: čuva se poredak među elementima sa istom vrednošću ključa proširenja

ključevi u opsegu $[a, b]$: element sa ključem k se smešta u $B[k - a]$

string ključevi iz konačnog skupa D mogućih vrednosti – sortiramo stringove, pa njihove indekse koristimo kao ključeve

?Da li moramo da se držimo int u tom opsegu? Ne moramo, možemo opseg da pomerimo $[a, b]$, onda imamo neku funkciju koja će pretvarati naše pozicije na taj opseg

LEKSIKOGRAFSKI POREDAK

- d -torka je sekvenca od d ključeva (k_1, \dots, k_d)

- ključ k_i je i -ta dimenzija torke -primer: Dekartove koordinate 3D tačke su 3-torke leksikografski poredak dve d -torke se definiše rekursivno:

$$(x_1, \dots, x_d) < (y_1, \dots, y_d)$$

$$\Leftrightarrow 1 < y_1 \vee x_1 = y_1 \wedge (x_2, \dots, x_d) < (y_2, \dots, y_d)$$

-torke se porede po prvoj dimenziji, pa onda po drugoj, itd.

-neka je C_i komparator koji poredi dve torke po njihovoj i -toj dimenziji

-neka je $\text{stableSort}(S, C)$ algoritam koji koristi komparator C

- **lexicographic sort** sortira sekvencu d -torki u leksikografskom redosledu izvršavajući d puta algoritam stableSort , jednom za svaku dimenziju

- lexicographic sort je $O(dT(n))$ gde je $T(n)$ vreme stableSort -a

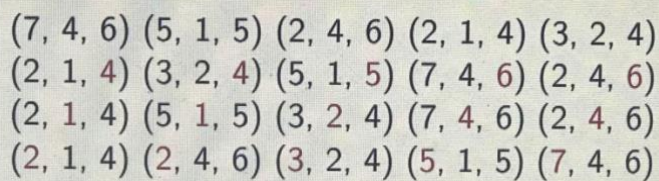
$\text{lexicographicSort}(S)$

Input: sekvenca S sa d -torkama

Output: sortirana S

for $i \leftarrow d$ to 1 do

$\text{stableSort}(S, C_i)$



-Sortira po poslednjoj dimenziji to je 6,5,6,4,4, dobicemo onda drugi red, prvo idu oni koji imaju 4 na kraju, pa onda oni koji imaju 5 na kraju, pa 6. Zatim se d smanjuje, ne idemo više po trećoj dimenziji nego po drugoj, 1,2,1,4,4. Isti postupak sa 1 idu na pocetak sa 4 na kraju. I na kraju ih sortiramo po prvoj dimenziji i to je to.

RADIX SORT

-radix sort je specijalizacija leksikografskog sortiranja koje koristi bucket sort kao stabilni sort algoritam za svaku dimenziju

-radix sort je primenljiv na torke gde su ključevi u svakoj dimenziji integeri iz $[0, N - 1]$

- radix sort je $O(d \cdot (n + N))$ – zavisi od broja dimenzija

-Ovaj algoritam možemo da koristimo i za sortiranje integera, ali u binarnom zapisu

-posmatramo sekvencu od n b -bitnih integer

$$x = x_{b-1} \dots x_1 x_0$$

-ove elemente tretiramo kao b -torku sa integerima u opsegu $[0, 1]$

-primenimo radix sort za $N = 2$

-ova varijanta radix sorta je (bn)

⇒ možemo sortirati niz 32-bitnih integera u linearnom vremenu!

-sortira se isto po dimenzijama

PROBLEM SELEKCIJE

-za dati integer k i n elemenata x_1, \dots, x_n treba naći k -ti najmanji element

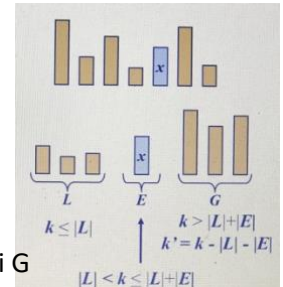
-možemo sortirati niz za $(n \log n)$ vreme i pristupiti k -tom element, ovde možemo koristiti algoritam quick select, koristi istu ideju kao quick sort, ali ipak se malo razlikuju

QUICK SELECT

-quick select je randomized algoritam zasnovan na **prune-and-search** principu

prune: izaberi slučajan element x (pivot) i podeli S na L , E i G

search: zavisno od k , ili je element u E ili moramo da tražimo rekursivno u L ili G

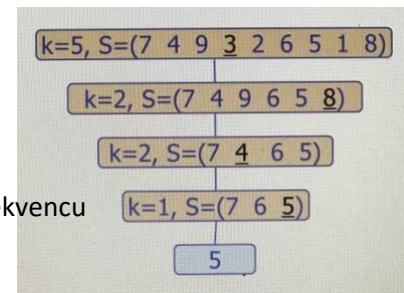


-u zavisnosti od toga koliko elemenata ima svaka kolekcija, mi tačno znamo u kom delu se nalazi naš k -ti element, kad odredimo u kom delu se nalazi naš k -ti element, ponavljamo ovaj postupak(rekursivno)

VIZUELIZACIJA QUICK SELECTA

-quick select možemo prikazati kao putanju rekurzije

- svaki čvor je jedan rekursivni poziv quickSelecta, čuva k i preostalu sekvencu



-Tražimo element na indeksu 5, birano nasumično pivot, i ispada da je to 3, vršimo particiju ove kolekcije, manji elementi od 3 su 2 i 1, veći elementi su 7,4,9,6,5,8. S obzirom da kolekcija manjih elemenata ima samo dva elementa, pivot se dodaje na to jos jedan, možem da tvrdimo da se element na poziciji 5 nalazi sigurno u većoj kolekciji, što znači da pozivamo isti ovaj algoritam nad kolekcijom većih elemenata. Moramo i da korigujemo indeks k , indeks umanjujemo za broj elemenata u manjoj kolekciji i u jednakoj kolekciji, tkd 5 umanjujemo za 3, sad je $k=2$, onda ponovo nasumično biramo pivot i ponavljamo postupak... dodjemo da je prvi element taj koji smo tražili

OČEKIVANE PERFORMANSE

-posmatrajmo rekursivni poziv quick selecta za sekvencu dužine s

*dobar izbor: dužine L i G su obe manje od $s \cdot 3/4$

*loš izbor: L ili G ima dužinu veću od $s \cdot 3/4$

-slučajan izbor pivota je dobar sa verovatnoćom $1/2$

* $1/2$ mogućih pivota su dobar izbor

-(1) iz verovatnoće: očekivani broj bacanja novčića da bismo dobili k glava je $2k$

-(2) iz verovatnoće: očekivanje je linearna funkcija

$$E(X + Y) = E(X) + E(Y)$$

$$E(cX) = cE(x)$$

- neka je $T(n)$ vreme quick selecta
- prema (2): $T(n) = T(3n/4) + n \cdot (\text{očekivani broj izbora do dobrog})$
- prema (1): $T(n) = T(3n/4) + bn$
- tj. $T(n)$ je geometrijska progresija $T(n) \leq 2bn + 2(3/4)n + 2b(3/4)2n + 2b(3/4)3n \dots$ dakle
- $T(n)$ je (n) , možemo rešiti problem selekcije u linearnom vremenu

Deterministička selekcija

- možemo uraditi selekciju i u najgorem slučaju za (n)
- osnovna ideja: rekurzivno koristimo quick select da bismo našli dobrog pivota za quick select
 - podeli S na 5 delova dužine $n/5$
 - nađi median u svakom podskupu
 - nađi median „malih“ mediana rekurzivno

13.OBLAST – OBRADA TEKSTA

STRING

- **string** je niz karaktera
- primeri stringova: Python program, HTML document, DNK sekvenca, digitalna slika
- **alfabet Σ** je skup mogućih karaktera za familiju stringova
- primeri alfabeta: ASCII, Unicode, $\{0, 1\}$, $\{A, C, G, T\}$
- neka je P string dužine m
 - **podstring** $P[i..j]$ od P je podsekvenca od P koja sadrži karaktere sa rangom između i i j
 - **prefiks** od P je podstring tipa $[0..i]$
 - **sufiks** od P je podstring tipa $[i..m - 1]$
- za date stringove T (tekst) i P (šablon, pattern) pattern matching problem je pronalaženje podstringa od T koji je jednak P
- primene: editori teksta, mašine za pretragu (search engines), bioinformatika

NALAŽENJE PODSTRINGA GRUBOM SILOM

- nalaženje grubom silom (brute force) poredi šablon P sa tekstom T za svaki mogući položaj P u odnosu na T sve dok se
 - * ne pronađe poklapanje
 - * ne testiraju sve
- gruba sila radi u (nm) vremenu, vreme zavisi od dužine teksta i od dužine šablona
- primer najgoreg slučaja:

$T = aaa \dots ah,$

$P = aaah,$

može da se pojavi u slikama i DNK sekvencama

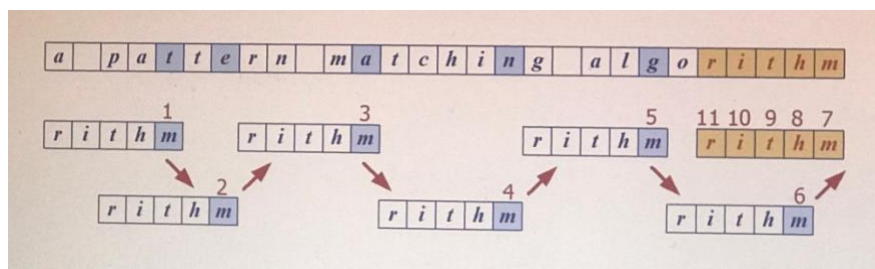
retko u tekstovima

BOYER-MOORE

-Boyer-Moore algoritam se zasniva na dva principa:

*ogledalo: poredi P sa podsekvencom u T idući unazad

*skok: ako se razlika otkrije u $T[i] = c$, ubrzamo poredjenje, kad utvrdimo da nema šanse da dodje do poklapanja, šablon može da skoči odnosno da se pomeri i više



-Na početku poravnamo poslednji karakter teksta sa poslednjim karakterom šablona. Ako se ne poklapaju, proveravamo da li karakter teksta na kom je došlo do razlike, da li uopšte postoji u našem šablonu, ako postoji, šablon ćemo pomeriti tako da se sada poklopi karakter teksta t sa poslednjim pojavljivanjem tog karaktera u šablonu, vraćamo se na kraj i nastavljamo poredjenje, onda poredimo e i m , međjutim pošto se razlikuju tražimo da li uopšte e postoji u šablonu, pošto e uopšte ne postoji u šablonu, možemo šablon da pomerimo za čitavu dužinu, zatim poredimo a i m , zaključujemo da se razlikuju, gledamo da li a postoji u šablonu, ne postoji, pomeramo šablon za čitavu dužinu, n i m .. isto opet pomeramo nema g u šablonu, g takodje, i onda dolazimo do h , h i m se razlikuju, tražimo da li h postoji u šablonu, postoji, pomeramo šablon tako da se oba h poklope, krećemo od kraja, poredimo sve, i na kraju nalazimo pogodak

-Da bismo znali koje je poslednje pojavljivanje, svakog od karaktera, moramo da formiramo last occurrence funkciju, funkciju poslednjeg pojavljivanja, nije dovoljno da uzmemo sve karaktere iz šablona nego ceo alphabet

-Boyer-Moore algoritam formira last occurrence funkciju L koja mapira alfabet Σ na cele brojeve gde je (c) definisano kao

- najveći indeks i takav da $P[i] = c$

- -1 ako takvog indeksa nema

-primer: $\Sigma = \{a, b, c, d\} = abacab$

$c \quad a \ b \ c \ d$

$(c) \quad 4 \ 5 \ 3 \ -1$

-zahteva jedan prolazak kroz šablon i jedan prolazak kroz alphabet

-može se predstaviti kao niz indeksiran numeričkim kodovima karaktera

- može se izračunati za $O(m + s)$ vreme gde je m dužina P a s je veličina Σ

- **bad character shift**: preskoči sigurno neuspešna poređenja

0: ako P ne sadrži c : pomeri P tako da se poklope $P[0]$ i $T[i + 1]$

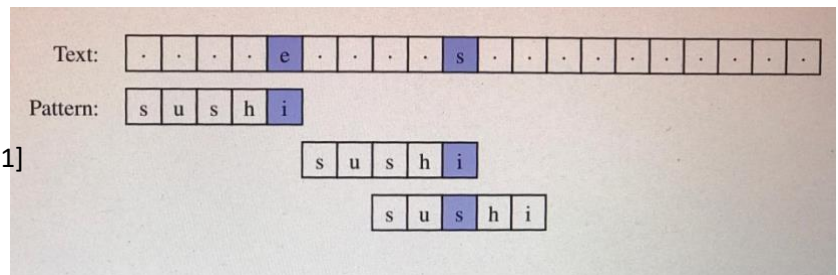
1: ako P sadrži c i poslednja pojava c je levo od pozicije i : pomeri P tako da se $T[i]$ poklopi sa poslednjom pojavom c u P

2: ako P sadrži c i poslednja pojava c je desno od pozicije i : pomeri P za jedno mesto

Boyer-Moore skok, slučaj 0

-slučaj 0 – ako P ne sadrži c :

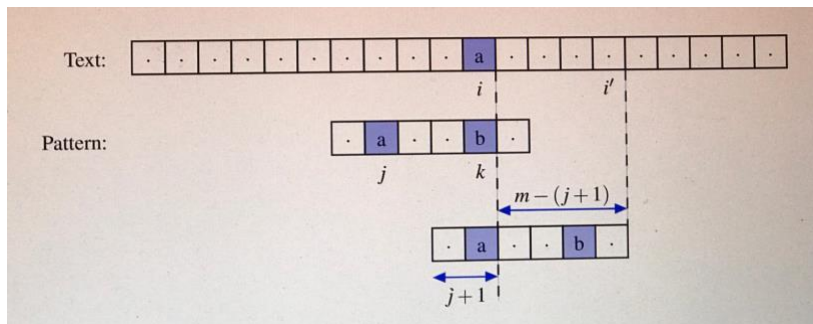
- pomeri P tako da se poklope $P[0]$ i $T[i + 1]$



Boyer-Moore skok, slučaj 1

-slučaj 1 – ako P sadrži c i poslednja pojava c je levo od pozicije i :

-pomeri P tako da se $T[i]$ poklopi sa poslednjom pojavom c u P

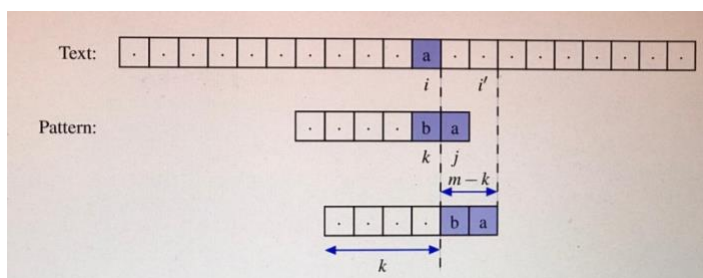


Boyer-Moore skok, slučaj 2

-slučaj 2 – ako P sadrži c i poslednja pojava c je desno od pozicije i :

-rešenje A: pomeri P za jedno mesto

- rešenje B: pomeri P tako da se $T[i]$ poklopi sa sledećom pojavom c u P – last occurrence funkcija više nije dovoljna!



BOYER-MOORE-ANALIZA

-Boyer-Moore je ($nm + s$)

-primer najgoreg slučaja: $T = aaa \dots a$

$$P = baaa$$

-najgori slučaj nije verovatan u tekstovima

- znatno brži od grube sile za tekstove na prirodnom jeziku

-nekad možemo da zanemarimo dužinu šablona, jer je znatno manja od dužine teksta

-najgori slučaj za BM je (nm), isto kao i gruba sila, za realne tekstove malo verovatan postoji i drugo pravilo za skok, **good suffix shift**, koje se zasniva na ideji koju koristi KMP algoritam (sledeći)

Knuth-Morris-Pratt

-**Knuth-Morris-Pratt** poredi tekst sa šablonom sleva u desno ali pomera šablon pametnije od grube sile

-kada se nađe razlika, koliko najviše možemo pomeriti šablon da izbegnemo suvišna poređenja?

-odgovor: najveći prefiks $P[0..j]$ koji je sufiks $P[1..j]$

-Ideja je sledeća, ako poredimo svaki element prva dva reda, i dodjemo do prve razlike x i a . Možemo da uočimo da su prva dva karaktera šablona ista kao poslednja dva karaktera prefiksa paterna. Gledamo da li postoji neki prefiks koji je isti sufiksu ovog dela šablona, imamo na početku a i b i na kraju a i b , i nailazimo na prvu razliku x i a , mi možemo sigurno da tvrdimo da kada pomerimo šablon da nemamo potrebe opet da proveravamo a sa a , b sa b , pošto će se sigurno poklopiti, nego da možemo da nastavimo poredjenje prve sledeće simetrije (j sa a) pozicije gde ta simetrija nije utvrđena

-Postavlja se pitanje na koji način mi da izanaliziramo šablon da utvrdimo koja je dužina istog prefiksa i sufiksa.

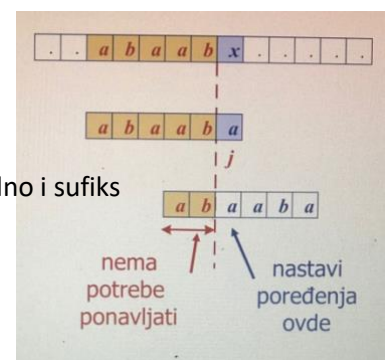
FUNKCIJA NEUSPEHA

-KMP analizira šablon da pronade njegove prefikse unutar samog šablona

-**funkcija neuspeha** $F(j)$ je veličina najvećeg prefiksa $P[0..j]$ takvog da je ujedno i sufiks $P[1..j]$

-ako nema poklapanja za $P[j] \neq T[i]$ pomeramo $j \leftarrow F(j - 1)$

-Prvo dodajemo sve karaktere našeg šablona, ova tabela bi bila kao neki niz koji ima onoliko pozicija koliko naš šablon ima karaktera, i svuda na početku upisujemo 0. Zatim krećemo od a (P_j-0), pošto je prvi karakter sigurno ne postoji sufiks i prefiks, prelazimo zatim na b (P_j-1). Da su se a i b preklopili onda bi dužina sufiksa bila jednaka dužini prefiksa i u F_j-1 bi pisali 1, al pošto se ne poklapaju pišemo 0. P_j-2 je a i poredimo sa prethodnim, pošto već ima a , poklapa se, i pišemo F_j-2 da je 1. Zatim prelazimo na sledeće a , gledama kolika je prethodna bila dužina sekvence (1), što znači da mi karakter a treba da poredimo sa karakterom na poziciji 1, a to je b . Postoji se a i b ne poklapaju, prelazimo na a (P_j-0), a i a se poklapaju, pišemo dužina da je 1. Prelazimo na sledeću poziciju P_j-4 i tu je karakter b , poredimo prethodnu dužinu sekvence to je 1, na poziciju 1 se nalazi karakter b , b i b se



poklapaju, dužina je 2. Sledeći karakter je a (Pj-5), gledamo kolika je sekvenca prethodne dužine (2), na poziciji 2 se nalazi a, a i a se poklapaju, pa je dužina sekvence za 1 duža i pišemo 3

Knuth-Morris-Pratt algoritam

-funkcija neuspeha se može prikazati nizom koji se izračuna za $O(m)$

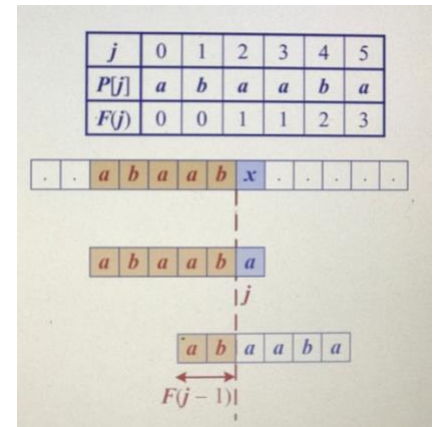
- u svakoj iteraciji petlje, ili

-i se poveća za 1, ili

- pomeraj $i - j$ se poveća za najmanje 1 (primeti da $F(j - 1) < j$)

⇒ nema više od $2n$ iteracija u petlji

⇒ KMP je $O(m + n)$



KMP: izračunavanje funkcije neuspeha

-funkcija neuspeha se može prikazati nizom koji se izračuna za $O(m)$

-slično kao i sam KMP algoritam u svakoj iteraciji petlje, ili

-i se poveća za 1, ili

pomeraj $i - j$ se poveća za najmanje 1 (primeti da $F(j - 1) < j$)

⇒ nema više od $2n$ iteracija u petlji

DINAMIČKO PROGRAMIRANJE

-**dinamičko programiranje** je pristup dizajnu algoritama

-Želimo da pomnožimo matrice A i B i da dobijemo matricu C

-Vreme je srazmerno dimenzijama matrica A i B, tako da ukupno vreme zavisi $O(d \cdot e \cdot f)$

MNOŽENJE MATRICA

-računamo $A = A_0 \cdot A_1 \cdot \dots \cdot A_{n-1}$

- A_i ima dimenzije $d_i \times d_{i+1}$

-koji redosled množenja izabrati? Važno nam je zato što dobrim odabirom redosleda množenja dobijamo značajno manje računanje, manji broj operacija koje treba da izvršimo, možemo:

1. Raspoređivanje zagrada / gruba sila

-traženje rešenja grubom silom: isprobati sve moguće kombinacije zagrada za $A = A_0 \cdot A_1 \cdot \dots \cdot A_{n-1}$, izračunati broj operacija za svaku i izabrati najbolju

-vreme izvršavanja: broj mogućih rasporeda zagrada je jednak broju različitih binarnih stabala sa n čvorova, **eksponencijalna** zavisnost! tzv. Katalanov broj, iznosi skoro 4^n

2. Pohlepni pristup

-ideja #1: ponavljaj izbor onog proizvoda koji će imati najviše operacija

-ideja #2: ponavljaj izbor onog proizvoda koji će imati **najmanje** operacija

-pohlepni pristup ne donosi optimalan izbor, ne dovodi do tačnih rešenja

-Ova 2 primera nisu najbolja..

3. „Rekurzivni“ pristup

-definišemo potprobleme

nađi najbolji raspored zagrada za podniz $A_i \cdot A_{i+1} \cdot \dots \cdot A_j$ neka je

$N_{i,j}$ broj operacija za ovaj potproblem

optimalno rešenje za ceo problem je $N_{0,n-1}$

-**optimalnost potproblema**: optimalno rešenje se može dobiti pomoću optimalnih potproblema

mora postojati poslednje množenje (koren stabla izraza) za optimalno rešenje

neka je to na i -tom indeksu: $(A_0 \cdot \dots \cdot A_i) \cdot (A_{i+1} \cdot \dots \cdot A_{n-1})$

optimalno rešenje za ceo problem $N_{0,n-1}$ je suma dva optimalna potproblema plus poslednje množenje

ako bi optimalno rešenje imalo bolje potprobleme, ne bi bilo optimalno

KARAKTERISTIČNA JEDNAČINA

-Bitno je da primetimo da ova optimalna rešenja nisu nezavisna, ne možemo da primenimo onaj pristup, podeli pa vladaj, pa da potpuno rekurzivno posmatramo, ovi potproblemi se medjusobno preklapaju tako da ne možemo da pristupimo uz upotrebu DC taktike

-globalni optimum se definiše pomoću optimalnih potproblema u zavisnosti od indeksa poslednjeg množenja

-razmotrimo sve moguće vrednosti tog indeksa

- A_i je dimenzije $d_i \times d_{i+1}$

-karakteristična jednačina za $N_{i,j}$ je: Razmotrimo sve mogućnosti proizvoljnih k , tako što se izračuna broj operacija za množenje matrice od i -te do k -te, od k -te do prve j -te, na to dodajemo još broj operacija koje posmatrano za poslednje množenje i razmotrimo sve ovakve zbireve i nadjemo najmanji- osnovna ideja

-konstruisaćemo optimalne potprobleme od dole na gore (bottom-up)

- $N_{i,i}$ je lako - nema množenja, tj. 0 operacija, počnemo od njih

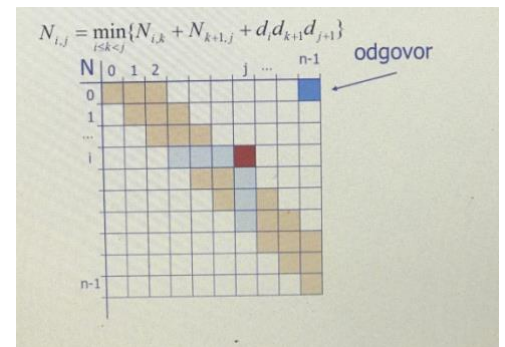
-onda pređemo na potprobleme dužine 2, 3, ... vreme izvršavanja je $O(n^3)$

VIZUELIZACIJA ALGORITMA

-Formiramo matricu mogućih rešenja, elementi ove matrice su minimalni broj operacija koji je potreban da se pomnože matrice. Kako čitamo ovu matricu? Ako gledamo 0-0 na ovom mestu

upisujemo koliko je operacija potrebno da se pomnože matrice od nulte do nulte, pošto je u pitanju samo jedna matrica broj operacija je 0, itd..

- bottom-up prvo popuni dijagonalu (sve 0 na dijagonali)
- $N_{i,j}$ se dobija na osnovu vrednosti iz i -tog reda i j -te kolone
- popunjavanje svake ćelije u tabeli je $O(n)$
- ukupno vreme je $O(n^3)$
- raspored zagrada dobijamo pamćenjem k u ćelijama table



-Broj operacija koji je potreban da se pomnoži od nulte i prve, ne znamo napamet, on zavisi od dimenzija ove matrice, poenta priče je da razmatramo koliko je prethodno bilo potrebno operacija i onda taj broj uvećava za broj operacija potrebnih da se pomnože matrice nulta i prva, svako sledeće rešenje baziramo na prethodnim

OPŠTI POSTUPAK DINAMIČKOG PROGRAMIRANJA

-primenljivo na probleme čije rešavanje traži puno vremena (moguće eksponencijalni) ukoliko postoje: jednostavni potproblemi:

potproblemi se mogu definisati pomoću promenljivih j, k, l, m itd.

optimalni potproblemi: globalni optimum se može definisati pomoću optimalnih potproblema

preklapanje potproblema: potproblemi nisu nezavisni i treba ih konstruisati bottom-up

PODSEKVENCA

-podsekvenca stringa $x_0x_1x_2 \dots x_{n-1}$ je string $x_{i_1}x_{i_2} \dots x_{i_k}$ gde je $i_j < i_{j+1}$

-nije isto što i podstring!

-primer stringa: ABCDEFGHIJK

jeste podsekvenca: ACEGIJK

jeste podsekvenca: DFGHK

nije podsekvenca: DAGH

*Problem najduže zajedničke podsekvence

-**longest common subsequence (LCS)**

- za stringove X i Y , LCS je najduža podsekvenca od X i Y

- primena: ispitivanje sličnosti DNK (alfabet je $\{A,C,G,T\}$)

-primer: ABCDEFG i XZACKDFWGH imaju LCS: ACDFG

-primena grube sile na LCS: pronađi sve podsekvence od X izdvoj one koje su i podsekvence od Y izaberi najdužu

-analiza: ako je X dužine n , ima 2^n podsekvenci, ovo je eksponencijalno vreme!

-tkd ovaj pristup ne možemo da primenimo već $\rightarrow \gg$

LCS dinamičkim programiranjem

-ovaj problem moramo da razbijemo na potprobleme, koji bi bili potproblemi? Pa da ne gledamo ceno string X, nego da gledamo neki njegov delić, takodje i za Y, I koristimo L, oznaku za najdužu zajedničku sekvencu, takvu da se razmatraju svu karakteri stringa X do pozicije i, I stringa Y do pozicije j

-Vrednosti koje u L možemo da beležimo mogu biti INDEKSI iz stringa od 0,i, a sa druge strane mogu biti i dužine stringova(ako napišemo i to znači da uzimamo prvih i karaktera X)

-neka postoji indeks -1, tako da je $L[-1, k] = 0$ i $L[k, -1] = 0$; to znači da null deo X ili Y nema poklapanja sa drugim

- sada definišemo $L[i, j]$ u opštem slučaju:

- ako je $x_i = y_j$ onda $L[i, j] = L[i - 1, j - 1] + 1$ (imamo poklapanje), onda je sigurno ta sekvencu za 1 duža od sekvence kada bismo taj i-ti i j-oti karakter zanemarili iz prvog odnosno drugog stringa

- ako je $x_i \neq y_j$ onda $L[i, j] = \max\{L[i - 1, j], L[i, j - 1]\}$ (nemamo poklapanje), onda možemo da kažemo da je dužina zajedničke podsekvence jednaka ili kada zanemarimo poslednji karakter prvog stringa ili najduže sekvence kada zanemarimo karakter drugog stringa, uzimamo veći broj

*primer slučaja $x_i = y_j$, primer: $L_{10,12} = 1 + L_{9,11}$

-kod prve sekvence uzimamo prvih 10 karaktera, kod druge prvih 12 karaktera, pošto du 10. i 12. karakter jednaki, onda je L sekvenca za 1 duža od sekvence L9,11, da su se razlikovali onda bismo uzimali dužu od sekvenci ili 9,12 ili 10,11

?Kako dolazimo do ovog rešenja? Koristimo matricu- dimenzije u zavisnosti od toga koliku dužinu stringa uzimamo, vrednosti su broj zajedničkih podstringova

	0	1	2	3	4	5	6	7	8	9	10	11	12
0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	1	1	1	1	1	1	1	1	1	1
2	0	0	1	1	2	2	2	2	2	2	2	2	2
3	0	0	1	1	2	2	2	3	3	3	3	3	3
4	0	1	1	1	2	2	2	3	3	3	3	3	3
5	0	1	1	1	2	2	2	3	3	3	3	3	3
6	0	1	1	1	2	2	2	3	4	4	4	4	4
7	0	1	1	2	2	3	3	3	4	4	5	5	5
8	0	1	1	2	2	3	4	4	4	4	5	5	6
9	0	1	1	2	3	3	4	5	5	5	5	5	6
10	0	1	1	2	3	4	4	5	5	5	6	6	6

X = G T T C C T A A T A
Y = C G A T A A T T G A G A

0 1 2 3 4 5 6 7 8 9
0 1 2 3 4 5 6 7 8 9 10 11

LCS algoritam: analiza

* algoritam ima dve ugnježdene petlje

spoljna ima n ciklusa

unutrašnja ima m ciklusa

telo unutrašnje petlje ima konstantno vreme

⇒ ukupno vreme je $O(nm)$

- odgovor je sačuvan u $L[n, m]$

POHLEPNA METODA

- **pohlepna metoda** je pristup dizajnu algoritama zasnovan na:

konfiguracije: različiti izbori, kolekcije ili vrednosti koje treba pronaći

funkcija cilja: vrednost (score) dodeljena konfiguracijama koju želimo da minimizujemo ili maksimizujemo

- najbolje funkcionise za probleme koji imaju osobinu pohlepnog izbora:

- globalno optimalno rešenje se može pronaći serijom lokalnih unapređenja polazeći od početne konfiguracije, dakle biramo uvek najbolju opciju i tako gradimo naše najbolje rešenje

KOMPRESIJA TEKSTA

- dati string X zapiši/kodiraj kao Y tako da Y zauzima manje memorije, štedi memoriju i/ili propusni opseg mreže

- odličan primer: **Huffman-ovo kodiranje** – ideja je da manje znakova koristimo za zapisivanje karaktera koji se češće koriste, njega možemo zapisati sa 0, a za one koji se ređe pojavljuju za njih možemo zauzeti i veće pozicije

- izračunaj frekvenciju pojavljivanja svakog znaka

- najčešće znakove kodiraj najkraćim kodovima

- nijedan kôd nije prefiks nekog drugog

- koristi optimalno stablo kodiranja za određivanje kodova

STABLO KODIRANJA

- **kôd** je preslikavanje karaktera iz alfabeta na binarni reprezent – kodnu reč

- **prefiksni kôd** je takav binarni kôd da nijedna kodna reč nije prefiks druge kodne reči

- **stablo kodiranja** predstavlja prefiksni kôd

- listovi čuvaju karaktere iz alfabeta

- kodna reč dobija se obilaskom putanje od korena do lista

- 0 za levo dete i 1 za desno dete

OPTIMIZACIJA STABLA KODIRANJA

- za dati string X tražimo prefiksni kod takav da rezultat kompresije bude što kraći

česti karakteri treba da imaju kratke kodne reči

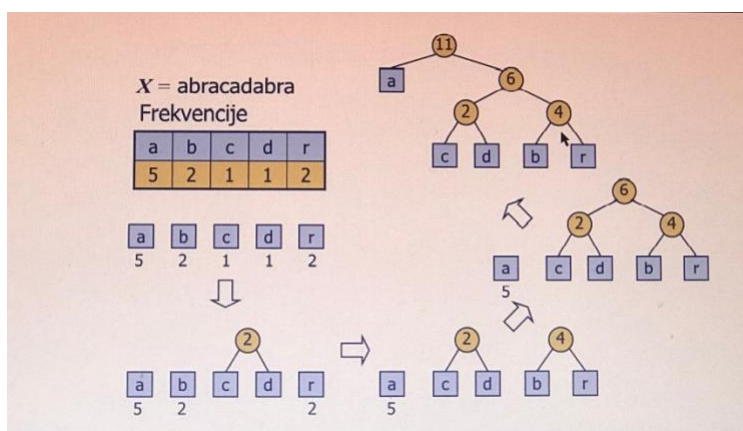
retki karakteri mogu da imaju duže kodne reči

HUFFMAN-OVO KODIRANJE

- za dati string X Huffman-ovo kodiranje konstruiše prefiksni kod koji minimizuje dužinu kôda od X
- radi u $O(n + d \log d)$ vremenu, n je dužina X , d je veličina alfabeta
- pomoćna struktura podataka: red sa prioritetom implementiran pomoću heapa

Primer:

- Počinjemo tako što prvo izračunamo tablicu frekvencije za svaki od karaktera koji se nalazi u našem stringu X , izračunamo koliko puta se karakter pojavljuje
- ako uzmemo da je naš string dužine n , koliko operacija nam treba da formiramo tablicu frekvencije? Zavisí koju strukturu koristimo. Najprimitivnije je da koristimo nekakav niz, onda bismo za svaki karakter da tražimo gde se nalazi u tom nizu, pa bi složenost bila n^2 , međutim ako je ovo neka hashmapa, mi ćemo pronalaziti poziciju našeg karaktera u $O(1)$ vremenu i moći ćemo da korigujemo tu vrednost, zapravo imaćemo linearnu zavisnost
- Ovaj algoritam predlaže da prvo ubacimo sve ove elemente u heap, prioritet kojim dodeljujemo odgovara njihovoj frekvenciji, a vrednost odgovara karakteru, u pitanju je heap i skidamo 2 elementa sa najmanjim prioritetom, to su c i d , sa ovim elementima formiramo stablo, tako što dodajemo korenski elemente i na taj način formiramo jedno malo podstablo, to podstablo vraćamo u heap, ali dodeljujemo mu prioritet koji odgovara zbiru prioriteta elemenata c i d . Zatim ponavljamo postupak sve dok u heapu ne ostane samo 1 element, taj 1 element koji je u heapu ostao, zapravo odgovara našem stablu



PROBLEM RANCA

- dat je skup S od n elemenata, svaki element i ima

b_i cenu (benefit)

w_i težinu

- cilj: izaberi elemente sa maksimalnom ukupnom vrednošću ali ukupnom težinom ne većom od W
- ovaj postupak se može zakomplikovati ako dozvolimo da se elementi mogu delimično uzeti
- Prva stvar je da za svaki element izračunamo vrednost, koliki je benefit po jedinici težine i da onda biramo elemente koji nam se najviše isplate, pa onda sve lošije i lošije

- ako je moguće uzeti razlomljene količine elemenata: x_i je količina elementa i cilj: maksimizovati, kao deo nekog elementa, po količini i masi, Uz ograničenje da je suma manja od kapaciteta ranca
- pohlepni izbor: izaberi element sa najvećom vrednošću (cena/težina)

radi u $O(n \log n)$ vremenu

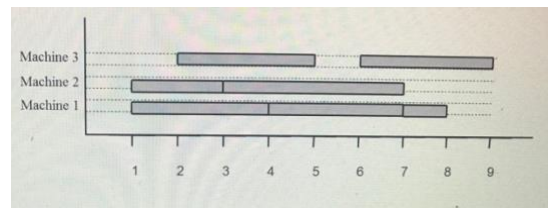
- **korektnost:** pretpostavimo da postoji bolje rešenje, bazira se na kontradiktornosti
- sličan problem imamo i kod raspoređivanja zadataka

RASPOREĐIVANJE ZADATAKA

- za dati skup T od n zadataka svaki zadatak ima

vreme početka s_i

vreme završetka f_i (gde je $s_i < f_i$)



- cilj: obaviti sve zadatke sa minimalnim brojem mašina
- Na početku imamo 0 mašina, pošto nemamo slobodnu mašinu, dodajemo mašinu 1, i raspoređujemo prvi zadatak na tu mašinu, zatim sledeći zadatak počinje u istom trenutku 1, pošto je prva mašina zauzeta, dodajemo još jednu mašinu i drugi zadatak započinjemo na njoj, zatim treći zadatak dolazi i on počinje u trenutku 2, međutim mašine 1 i 2 su zauzete, tako da moramo da dodamo i treću mašinu na kojoj će se izvršavati zadatak 3. U trenutku 3, pojavljuje se novi zadatak, proveravamo mašinu 1, ona je zauzeta, proveravamo mašinu 2 ona je slobodna, tako da ovaj zadatak počinjemo da izvršavamo na mašini 2, itd...
- pohlepni izbor: razmatraćemo zadatke po vremenu početka i koristiti što manje mašina za ovaj redosled vreme izvršavanja $O(n \log n)$
- korektnost: pretpostavimo da postoji bolji raspored

PREDPROCESIRANJE STRINGOVA

- predprocesiranje šablona ubrzava pattern matching
- vreme za KMP je proporcionalno dužini teksta nakon predprocesiranja
- ako je tekst dugačak, ne menja se i često se pretražuje mogli bismo da predprocesiramo tekst umesto šablona
- trie (čita se kao „try“) je struktura podataka za čuvanje stringova, npr. svih reči u tekstu

vreme pretrage je proporcionalno dužini šablona

STANDARDNI TRIE

- standardni trie za skup stringova S je stablo:
 - svaki čvor osim korena čuva jedan karakter
 - rodeca čvora su u alfabetskom redosledu
 - putanja od korena do lista daje čuvani string
- na početku razdvojimo tekst na reči i onda ubacujemo jednu po jednu reč

- standardni trie troši $O(n)$ prostora
- dodavanje, uklanjanje i pretraga su $O(dm)$

n ukupna dužina stringova u S

m dužina stringa u operaciji

d veličina alfabeta

-poslednji listovi sadrže podatke o tome gde se ta reč nalazi u tekstu, na kojim pozicijama, koliko puta se nalazi...

Traženje reči u trie:

dodaj reči iz teksta u trie

svaki list je jedna reč

list čuva indekse gde počinje reč

KOMPRESOVANI TRIE

- kompresovani trie ima unutrašnje čvorove sa bar 2 deteta, dobija se od standardnog kompresovanjem lanaca „redundantnih“ čvorova

KOMPAKтна REPRESENTACIJA

- kompaktna reprezentacija kompresovanog trie-a za niz stringova

čvorovi čuvaju opsege indeksa umesto podstringove

troši $O(s)$ prostora, gde je s broj stringova u nizu

služi kao pomoćna indeksna struktura

- **sufiksni trie** stringa X je kompresovani trie svih sufiksa od X , ovo može da nam pomogne ako pokušavamo da mećujemo samo neki deo stringa

Sufiksni trie: analiza

- string X dužine n , alfabet veličine d

troši $O(n)$ prostora

pretraga za $O(dm)$ vreme; m dužina traženog šablona

konstruiše se za $O(n)$ vreme

KODNI TRIE

- kodni trie predstavlja prefiksni kod

svaki list čuva karakter

kodna reč predstavlja putanju od korena do lista

0 za levo dete, 1 za desno dete

14. OBLAST – Grafovi

- graf je par (V, E) V je set čvorova (vertices) E je skup grana (edges) čvorovi i grane čuvaju elemente
- primer: čvorovi predstavljaju aerodrome i čuvaju šifre aerodrome, grane predstavljaju letove između aerodroma i čuvaju dužinu puta

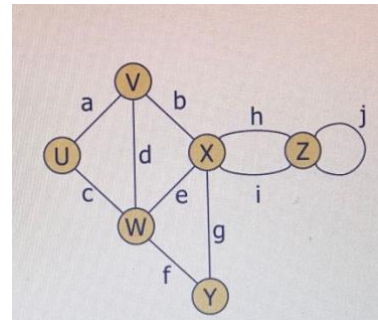
VRSTE GRANA

- **usmerena grana** uređeni par čvorova (u, v) , prvi čvor u je polazište, drugi čvor v je odredište npr. konkretan let aviona
- **neusmerena grana** neuređeni par čvorova (u, v) npr. putanja leta
- **usmereni graf** sve grane su usmerene npr. mreža letova
- **neusmereni graf** sve grane su neusmerene npr. mreža putanja

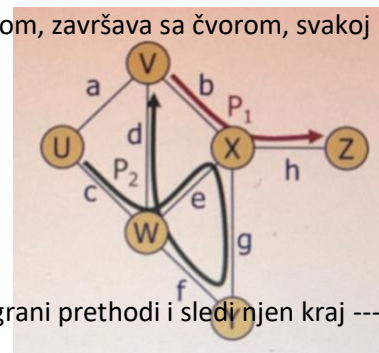
PRIMENA - elektronska kola, štampane ploče, integrisana kola, transportne mreže(putevi, avionski letovi), računarske mreže, lokalne mreže, Internet ,baze podataka, ER dijagrami

TERMINOLOGIJA

- krajevi grane - U i V su krajevi a
- grane incidentne na čvoru - a, d i b su incidentni na V
- susedni čvorovi - povezani granom, U i V su susedni
- stepen čvora - broj grana kojima je on kraj, stepen of X je 5
- paralelne grane: h i i
- petlja: j



- putanja - sekvenca naizmenično čvorova i grana , počinje sa čvorom, završava sa čvorom, svakoj grani prethodi i sledi njen kraj
- prosta putanja - svi čvorovi i grane su različiti
- primeri $P_1 = (V, b, X, h, Z)$ je prosta



$P_2 = (U, c, W, e, X, g, Y, f, W, d, V)$ nije prosta

- petlja - cirkularna sekvenca naizmenično čvorova i grana svakoj grani prethodi i sledi njen kraj -----
- prosta petlja

s vi čvorovi i grane su različiti

- primeri

$C_1 = (V, b, X, g, Y, f, W, c, U, a, V)$ je prosta

$C_2 = (U, c, W, e, X, g, Y, f, W, d, V, a, U)$ nije prosta

OSObine

- n – broj čvorova

m – broj grana

$deg(v)$ – stepen čvora

- suma stepena čvorova $\sum_v deg(v) = 2m$ (svaka grana se broji dva puta)

- u neusmerenom grafu bez petlji i višestrukih grana $m < n(n - 1)/2$ (svaki čvor ima stepen najviše $n - 1$)

ČVOROV I GRANE

- graf je kolekcija čvorova i grana

- prikazaćemo ga pomoću tri tipa: Vertex, Edge i Graph

- Vertex je „lagani“ objekat koji čuva sadržaj (npr. šifru aerodroma) ima metodu `element()` kojom se može dobiti taj sadržaj

- Edge čuva sadržaj (npr. broj leta, rastojanje)

`element()` vraća taj sadržaj

`endpoints()` vraća par (u, v) polazište i odredište

`opposite(v)` vraća suprotni kraj grane

IMPLEMENTACIJA 1 : Lista čvorova i lista grana

- Vertex čuva sadržaj element liste

- Edge čuva sadržaj, referenca na polazište, referenca na odredište, element liste

- lista čvorova

- lista grana

- nije loša za dodavanje čvorova $O(1)$ vremenu, ali je za brisanje loša, moramo prvo sve grane da bi stigli do čvorova

IMPLEMENTACIJA 2: Lista suseda

- lista grana za svaki čvor

- grane mogu imati reference na drugu pojavu iste grane (za drugi krajnji čvor)

- lako je da se pristupi element, ali što se tiče brisanja, opet za svaki čvor moramo da prodjemo i kroz grane, a i da znamo gde se svaka ta grana nalazi, pa treba obrisati i tu granu ako je i u drugom čvoru

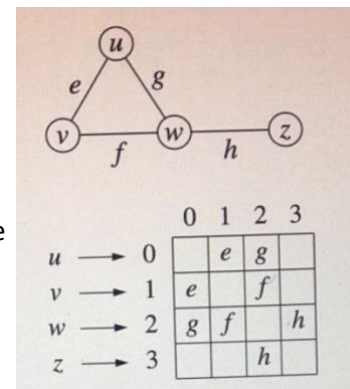
- složena je, ali okej ako nema puno izemene na grafu

IMPLEMENTACIJA 3 : Matrica incidencije

- svakom čvoru dodeljen int ključ

- matrica sadrži referencu na granu ukoliko ona povezuje dva čvora, ili None

- ili samo 0/1 u matrici



n čvorova, m grana, bez paralelnih grana, bez petlji

	lista grana	lista suseda	matrica incidencije
<i>prostor</i>	$n + m$	$n + m$	n^2
<code>incident_edges(v)</code>	m	$deg(v)$	n
<code>are_adjacent(v,w)</code>	m	$\min\{deg(v), deg(w)\}$	1
<code>insert_vertex(v)</code>	1	1	n^2
<code>insert_edge(v,w,e)</code>	1	1	1
<code>remove_vertex(v)</code>	m	$deg(v)$	n^2
<code>remove_edge(e)</code>	1	1	1

PYTHON implementacija

-koristićemo mapu susedstva

- za čvor v čuvamo Python rečnik sa susedima $I(v)$

-listu čvorova zamenićemo rečnikom D koji mapira svaki čvor na njegovu mapu suseda $I(v)$, možemo proći kroz sve čvorove sa $D.keys()$

-čvor ne mora da čuva svoj položaj u D jer se to izračuna za $O(1)$

-performanse iste kao za listu suseda ali u **očekivanom** slučaju

PODGRAFI

-**podgraf** S grafa G je graf takav da čvorovi S su podskup čvorova, G grane S su podskup grana G

-**pokrivajući podgraf** sadrži sve čvorove G

POVEZANOST

-graf je **povezan** ako postoji putanja između svaka dva čvora

-**povezana komponenta** je maksimalni povezani podgraf

STABLO I ŠUMA

-ako pričamo o teorijskom delu ne mora da ima koren

-**stablo** je neusmereni graf T takav da

T je povezan

T nema petlje

Ovde može doći do problema navigacije, ako imamo neusmereni graf mi možemo iz jednog čvora doći u drugi, uglavnom u praktičnom delu se ne koristi, koristi se usmereni

-**šuma** je neusmereni graf takav da

nema petlje

povezane komponente su stable

POKRIVAJUĆE STABLO I ŠUMA

- pokrivajuće stablo** je podgraf koji je stablo i pokriva graf
- pokrivajuće stablo nije jedinstveno ako graf nije stablo
- pokrivajuća šuma** je podgraf koji je šuma i pokriva graf

OBILAZAK GRAFA PO DUBINI

- depth-first-search (DFS) je opšti metod za obilazak grafa
 - obilazi sve čvorove i grane
 - određuje da li je graf povezan
 - određuje povezane komponente grafa
 - određuje pokrivajuću šumu grafa
- DFS na grafu sa n čvorova i m grana traje $O(n + m)$
- može se proširiti za rešavanje drugih problema
 - naći putanju između dva čvora
 - naći petlju u grafu
- DFS je za graf isto što i Ojlerov obilazak za binarna stable

DFS ALGORITAM

- DFS algoritam dodeljuje oznake (labele) čvorovima i granama, vrši laboliranje, beležimo na neki način grane i čvorove koje smo prošli, kako ne bismo više puta obilazili isto
- Grane obeležavamo sa dve oznake, unexplored i visited, na početku su unexplored, a kada ih prvi put obidjemo labela postaje visited
- Kada nadjemo neki čvor koji nismo obišli na njega započinjemo obilazak po dubini, čvor označavo kao posećen i onda gledamo sve grane koje izlaze iz tog čvora, ako naidjemo na granu koju nismo prošlo, idemo na čvor koji se nalazi sa druge strane, ako na tom čvoru nismo bili, granu označavamo sa discovery i nastavljamo dalje obilazak po dubini, u pitanju je rekurzija, a ako tom granom nismo prošli, a ipak smo već posetili čvor sa druge strane, onda ćemo tu granu označavati sa back

DFS I PROLAZAK LAVIRINTA

- svaku raskrnicu, ugao (skretanje) i kraj puta označimo kao posećen čvor
- svaki hodnik (granu) kao posećenu
- pamtimo odakle smo počeli pomoću steka rekurzije
- uglavnom se ide uvek pravo dok se ne dodje do zida...

OSOBI NE DSF

1. DFS(G, v) obilazi sve čvorove i grane u povezanoj komponenti od v

2. grane označene kao DISCOVERY čine pokrivaјуće stablo povezane komponente od v

PERFORMANSE DNS

- stavljanje labele na čvor/granu traje $O(1)$
- svaki čvor se označi dva puta, jednom kao UNEXPLORED, drugi put kao VISITED
- svaka grana se označi dva puta, jednom kao UNEXPLORED, drugi put kao DISCOVERY ili BACK
- metoda incident_edges se poziva jednom za svaki čvor DFS traje $O(n + m)$ ako je graf predstavljen listom suseda $\sum_v deg(v) = 2m$

DFS za pronalaženje putanje

- DFS se može upotrebiti za pronalaženje putanje između dva čvora u i z
- Znamo od kog čvora krećemo i u kom čvoru želimo da završimo. Ako smo završili u tom čvoru onda ćemo izlistati putanju, ideja je da imamo stek i svaki put kada pristupimo nekom čvoru, odnosno pozovemo dfs za čvor, dodajemo taj čvor na stek. Ako dodje do toga da se mi vratimo do originalne putanje, dakle vraćamo se unazad do čvora do kog smo stigli i nismo našli čvor koji smo tražili onda taj put kojim smo krenuli treba da uklonimo sa steka, to nije dobar put i neće nas dovesti do rešenja
- Kada nadjemo čvor koji smo tražili, sve elemente skidamo sa steka, odnosno svi elementi na steku reprezentuju putanju kojom smo se kretali da bismo došli do traženog čvora
- Drugi način za upotrebu algoritma za obilaženje po dubini je zapravo nalaženje petlje. Kako da utvrdimo da li uopšte postoji petlja u okviru našeg grafa? Mi zapravo tražimo BACK granu i čim je nadjemo zaključujemo da smo pronašli petlju. Dakle krećemo od nekog čvora, označavamo ga kao posećenog i onda za sve njegove susedne ivice ako ih nismo već istražili, idemo na drugu stranu i tu ivicu dodajemo na stek, ako je čvor koji se nalazi sa druge strane unexplored onda označavamo ivicu sa discovery i pozivamo algoritam za pronalazak putanje i skidamo našu ivicu sa steka. Ono što je ovde zapravo ključno je da kada dodjemo do back grane to znači da treba da ispraznimo stek i to je to.

OBILAZAK GRAFA PO ŠIRINI

- breadth-first-search (BFS) je opšti metod za obilazak grafa
 - obilazi sve čvorove i grane
 - određuje da li je graf povezan
 - određuje povezane komponente grafa
 - određuje pokrivaјuću šumu grafa
- BFS na grafu sa n čvorova i m grana traje $O(n + m)$
- može se proširiti za rešavanje drugih problema
 - nađi najkraću putanju između dva čvora
 - nađi prostu petlju

BFS ALGORITAM

- Na početku označavamo sve grane i čvorove kao unexplored, i onda krećemo od neke prve grane koja je unexplored i taj čvor uzimamo za početak obilaska. Prvo kreiramo liste L_1, L_2, \dots i to su nam zapravo ti nivoi, na prvom nivou dodajemo samo taj početni čvor i označavamo ga kao visited, a zatim počinjemo formiranje sledećeg nivoa, pravimo novu listu i onda analiziramo sve čvorove koji postoje u staroj listi i njima susedne čvorove dodajemo u novu listu L_1 , ako je naša grana unexplored, možemo je označiti ili sa discovery oznakom ili sa cross oznakom (kao back za dubinu). Idemo kroz sve incidentne ivice čvora, ako nismo tom ivicom prošli, dobavljamo čvor koji se nalazi sa druge strane, ako taj čvor takodje nismo nikada posetili, granu označavamo sa discovery, čvor sa visited i taj čvor dodajemo na kraj našeg novog reda. Ukoliko je čvor sa druge strane već posećen onda ćemo granu označiti sa cross.

- Možemo da primetimo da cross grane povezuju čvorove na istom nivou ili povezuju čvorove sa višeg na niži nivo

OSOBINE BFS

G_s : povezana komponenta od s

1 $BFS(G, v)$ obilazi sve čvorove i grane od G_s

2 grane označene kao DISCOVERY čine pokrivajuće stablo T_s za G_s

3 za svaki čvor u L_i

putanja iz T_s od s do v ima i čvorova

svaka putanja od s do v u G_s ima bar i čvorova

PERFORMANSE BFS

- stavljanje labele na čvor/granu traje $O(1)$

-svaki čvor se označi dva puta, jednom kao UNEXPLORED, drugi put kao VISITED

-svaka grana se označi dva puta, jednom kao UNEXPLORED, drugi put kao DISCOVERY ili CROSS

-svaki čvor se jednom dodaje u sekvencu L_i

-metoda incident_edges se poziva jednom za svaki čvor

-BFS traje $O(n + m)$ ako je graf predstavljen listom suseda $\sum_v \deg(v) = 2m$

PRIMENA BFS

- možemo prilagoditi BFS za rešavanje sledećih problema u trajanju $O(n + m)$:

određivanje povezanih komponenti

određivanje pokrivajuće šume

pronalaženje proste petlje ili je graf šuma

pronalaženje najkraće putanje između dva čvora, ili ne postoji

DFS vs BFS

primena	DFS	BFS
pokrivajuća šuma, povezane komponente, putanje, petlje	✓	✓
najkraći put		✓
bipovezane komponente	✓	

BACK grana (v, w)

- w je predak od v u stablu koje čine DISCOVERY grane

CROSS grana (v, w)

- w je na istom nivou ili na sledećem nivou u odnosu na v

* Kod dfs back grane pokazuju na predak našeg čvora, dok kod bfs ili na čvorove istog nivoa ili na čvorove narednog nivoa

USMERENI GRAF

- usmereni graf (directed graph, „digraph“) je graf čije su sve grane usmerene

*OSOBI NE

- graf $G = (V, E)$ takav da svaka grana je usmerena grana (a, b) , ide od a ka b , i ne ide od b ka a

- ako je G prost (nema petlji i višestrukih grana):

$$m \leq n(n - 1)$$

- ako čuvamo ulazne i izlazne grane u posebnim listama, njihovo listanje traje proporcionalno njihovom broju

*PRIMENE

- **raspoređivanje zadataka** (scheduling) grana (a, b) označava da se zadatak a mora uraditi pre nego što počne zadatak b

DFS ZA USMERENI GRAF

- biće četiri vrste grana:

Discovery

Back

forward: vodi prema potomku u pokrivaćem stablu

cross: vodi prema čvoru u pokrivaćem stablu koji nije ni predak ni potomak

- DFS za usmereni graf polazeći od čvora s određuje čvorove dostupne iz s

*DOSTUPNOST

-DFS stablo sa korenom u v : čvorovi dostupni iz v duž usmerenih grana

*JAKA POVEZANOST

- iz svakog čvora se može stići do svakog drugog čvora

?Kako se proverava da li je ova osobina prisutna?

1. izaberi čvor v iz G
2. uradi DFS iz v , ako postoji w koji nije obišćen, ispiši „ne“
3. formiramo novi graf - neka je G' dobijen okretanjem grana u G
4. uradi DFS iz v u G' ako postoji w koji nije obišćen, ispiši „ne“, inače ispiši „da“
5. vreme izvršavanja: $O(n + m)$

JAKO POVEZANA KOMPONENTA

- maksimalni podgraf takav da su iz svakog čvora dostupni svi drugi čvorovi

- takođe traje $O(n + m)$ pomoću DFS ali je složenije

TRANZITIVNO ZATVORENJE

- Tranzitivno usmerenje nekog grafa npr G je proširenje grafa G , tako da ako postoji putanja između dva čvora, dodaje se i direktna veza između ta dva čvora

- Predstavlja proširenje nekog usmerenog grafa takav da između svaka dva proširena čvora postoji i direktna grana

IZRAČUNAVANJE TRANZITIVNOG ZATVORENJA

- možemo raditi DFS iz svakog čvora – traje $O(n(n + m))$

- možemo koristiti dinamičko programiranje – Floyd-Warshall algoritam

FLOYD-WARSHALL ALGORITAM

ideja #1: označi čvorove brojevima $1, 2, \dots, n$

ideja #2: razmatramo putanje koje imaju samo čvorove $1, 2, \dots, k$ kao međučvorove

- vreme je $O(n^3)$ ako je proverava da li su neka dva čvora susedna $O(1)$ (koristi se matrica incidencije)

- Svaki čvor numeriši na početku, kreni od nekog početnog grafa G_0 , uzmi iz prvog samo prvi element, pa prvi drugi, prvi treći.. na početku uzimamo element samo koji je označen sa 1, i proveravamo za svaka dva čvora našeg grafa takva da su oni različiti od k , proveravamo da li je k susedni čvor čvoru i , da li je k susedni čvor čvoru j , ako je susedan i jednom i drugom, proveravamo da li su naša dva čvora susedna, ako nisu, dodajemo vezu koja ih spaja

TOPOLOŠKO UREDJENJE USMERENOG GRAFA

- **usmereni aciklični graf** (DAG) je aciklični graf koji nema petlje

- topološko uređenje je numerisanje $v_1 \dots v_n$ takvo da za svaku granu (v_i, v_j) važi $i < j$ (uvek se ide od manjeg čvora ka većem)

- primer: u grafu raspodele zadataka topološko uređenje je sekvenca zadataka koji ispunjavaju uslove prethođenja

-usmereni graf ima topološko uređenje akko je acikličan

TOPOLOŠKO SORTIRANJE

- Kada uvodimo topološko sortiranje, svakom čvoru ovog grafa se dodaje brojna vrednost, i na taj način je jasno odredjen tok izvršavanja ovih zadataka.

- Algoritam izgleda tako što idemo do čvora koji nema izlazne grane, i taj čvor označavamo najvećim brojem, onda se vraćamo jedan nivo nazad, ako nema izlaznih grana, njega označavamo jednim brojem manje itd...

Topološko sortiranje pomoću DFS

- Sve čvorove i sve grane označimo sa unexplored i onda idemo u topološko sortiranje, kada posetimo neki čvor označimo ga sa visited, gledamo da li ima izlazne grane, ako ima, idemo na drugu stranu grane, nalazimo čvor koji se nalazi sa druge strane grane, ako u tom čvoru nismo bili, granu označavamo kao discovery i pozivamo topolosku pretragu za taj čvor sa druge strane, ako smo u tom čvoru već bili, granu označavamo ili forward ili cross, i naš čvor označavamo brojem n

TEŽINSKI GRAF

- svaka grana ima dodeljenu težinu

-može da predstavlja rastojanje, trošak, itd.

-primer: rastojanje između aerodrome

- trudimo se da nekako tu težinu smanjimo, znači to je nešto što nam pravi problem, za rastojanje-da što pre stignemo do nekog grada, za trošak - što jeftinije, što znači da trudimo se da nadjemo neku putanju kroz ovaj graf, tako da ta težina bude najmanja

- Uvodimo termin najkraći put

NAJKRAĆI PUT

- za dati težinski graf i čvorove u i v želimo da nađemo putanju sa najmanjom ukupnom težinom između u i v

-težina putanje je suma težina grana u putanji

- primer: rutiranje paketa u mreži

OSOBINE NAJKRAĆEG PUTA

1 podput najkraćeg puta je takođe najkraći put

2 postoji stablo najkraćih puteva od početnog čvora do svih ostalih čvorova

-za ovo koristimo Dijkstra algoritam

DIJKSTRA ALGORITAM

- rastojanje od čvora v do čvora s je dužina najkraćeg puta $v \rightarrow s$

- Dajkstrin algoritam računa rastojanja do svih čvorova od datog čvora s
- pretpostavke: graf je povezan, grane nisu usmerene, težine nisu negativne
- stvaramo „oblak“ čvorova počevši od s dok ne uključimo sve čvorove
- za svaki čvor v čuvamo labelu $d(v)$ koja predstavlja rastojanje od v do s u podgrafu sastavljenom od čvorova iz oblaka
- u svakom koraku:

u oblak dodamo čvor u sa najmanjim rastojanjem $d(u)$

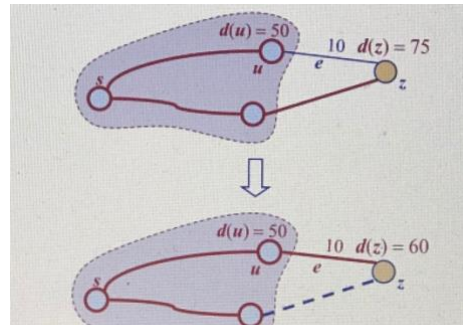
ažuriramo labele čvorova susednih sa u

RELAKSIRANJE GRANA

- posmatramo granu $e = (u, z)$ takvu da

u je čvor poslednji dodat u oblak

z nije u oblaku



- do sada, do jednog trenutka, smatramo da je najkraći put ili najmanja težina za čvor 75, međutim ispostavlja se da postoji neki čvor u , koji ima težinu 50 i moguće je da uz još +10 po ovoj grani dodjemo do ovog čvora, to znači da je pogrešno korigovati težinu za čvor z i promeniti na $50 + 10 = 60$, našli smo bolju putanju za čvor z , ovaj postupak se zove relaksiranje grana

- Dakle poredimo prethodnu težinu, i težinu čvora plus grane i onda uporedimo težine, i ako je bolja težina tj manja, izmenićemo težinu čvora z

- U startu svi imaju težinu beskonačno osim prvog čvora A koji kreće od 0. Idemo kroz grane do svih susednih čvorova čvora A , u čvoru B je pisalo beskonačno, međutim grana je težine 8, i menjamo težinu čvora B na 8, zatim idemo na čvor C , grana 2, $0 + 2$ težina čvora C je 2, onda za D isto je 4, i sad kad smo obišli sve susedne čvorove čvora A , određujemo koji čvor ćemo ubaciti u oblak, to je čvor sa najmanjom težinom, u ovom slučaju je to C , pa C ubacujemo u oblak. Pristupamo svim susednim čvorovima čvora C , idemo do B i vidimo da je to $2 + 7$ što je ukupno 9, ali taj broj nije bolji od broja 8 i onda ništa ne menjamo, zatim idemo na E čvor, beskonačan je pa onda dodajemo $2 + 3 = 5$, isto tako i čvor F je 11, idemo do čvora D , $2 + 1$ je 3 i ta vrednost je bolja od prethodne vrednosti 4, što znači da će se vrednost 4 korigovati na vrednost 3 jer je bolja putanja, opet biramo čvor koji ima najmanju težinu i dodajemo u oblak.. i ponavljamo postupak..

- Pitanje je kako mi da znamo koji čvor ima trenutno

najmanju težinu, pa tako što ćemo koristiti red sa

prioritetom, heap, kada korigujemo težinu nekog čvora,

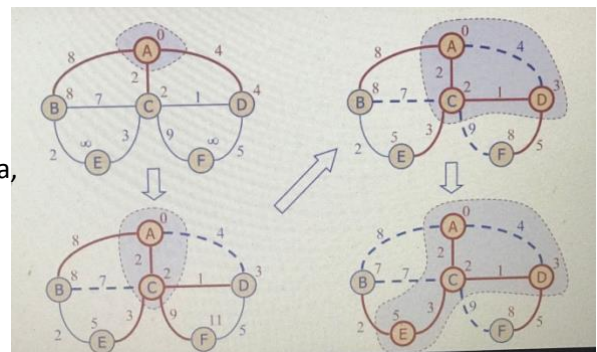
moramo da promenimo ključ u heapu

DIJKSTRA ALGORITAM – ANALIZA

- * operacije nad grafom

nalazimo susedne grane po jednom za svaki čvor

- * operacije sa labelama



za čvor z postavljamo rastojanje i labelu $O(deg(z))$ puta

postavljanje labelu traje $O(1)$

*operacije nad redom sa prioriteto

svaki čvor se dodaje jednom i uklanja jednom iz RSP, svaka operacija traje $O(\log n)$

ključ čvora u RSP se menja najviše $deg(w)$ puta, svaka zamena ključa traje $O(\log n)$
postavljanje labelu traje $O(1)$

* Dajkstrin algoritam traje $O((n + m)\log n)$ ako je u implementaciji korišćena lista suseda

DIJKSTRA I POHLEPA

- Dijkstra koristi pohlepni metod – dodaje čvorove po rastućem rastojanju
- pretpostavimo da nije našao najkraća rastojanja; neka je F prvi pogrešan čvor
- najkraći put je bio OK za prethodni čvor D ali grana (D, F) je tada relaksirana
- prema tome, sve dok je $d(F) \geq d(D)$, $d(F)$ nije pogrešno, tj. nismo pogrešili čvor

DIJKSTRA NE RADI ZA NEGATIVNE TEŽINE

- Dijkstra koristi pohlepni metod – dodaje čvorove po rastućem rastojanju
- ako bi čvor sa negativnom granom bio dodat u oblak, pokvario bi rastojanja čvorova koji su već tamo

BELLMAN-FORD ALGORITAM

- radi i za negativne težine
- grane moraju biti usmerene – inače bismo imali negativne petlje!
- i -ta iteracija pronalazi najkraće puteve sa i grana
- vreme: $O(nm)$
- Ovaj algoritam predlaže da se na početku za naš početni čvor postavi 0, a za sve ostale beskonačno, zatim idemo redom, za sve ivice obavi relaksaciju grana, zatim dobavi dva čvora koja se nalaze sa suprotnih strana ivice, na osnovu njih izračunaj težinu i gledaj da li je ta težina bolja ili lošija od prethodne
- koliko ima čvorova, toliko puta predji kroz sve ivice to je neka osnovna ideja

ALGORITAM ZA USMERENI ACIKLIČNI GRAF

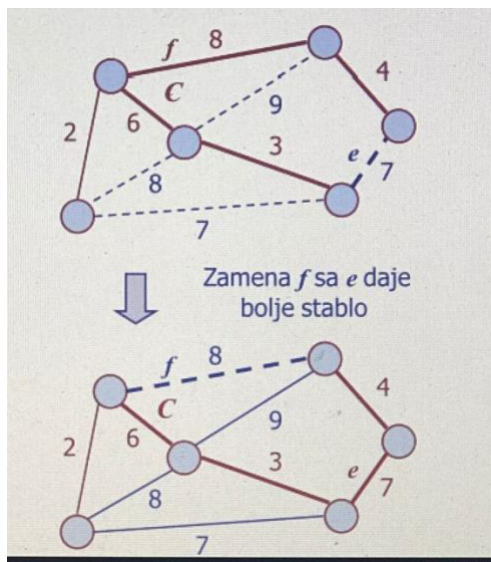
- radi i za negativne težine, koristi topološko uređenje, ne koristi posebnu strukturu podataka, znatno brži od Dijkstre: $O(n + m)$
- Početni čvor postavimo sa 0, ostale sa beskonačno, obavljamo topološko sortiranje čvorova i onda obilazimo čvorove u topološkom redosledu, gledamo izlazne grane i pokušavamo za sve izlazne grane našeg čvora relaksiramo granu ako je to moguće

MINIMALNO POKRIVAJUĆE STABLO

- **pokrivajući podgraf**: podgraf od G koji sadrži sve čvorove iz G
- **pokrivajuće stablo**: pokrivajući podgraf koji je stablo
- **minimalno pokrivajuće stablo**: pokrivajuće stablo sa najmanjom sumom težina grana
- „minimum spanning tree“ (MST)
- koristiti se u internet mrežama

MST I PETLJE

- Postoji ovde diskusija zašto je to tako, u suštini ideja kao dokaza je da analiziramo minimalno pokrivajuće stablo (crvenim) i diskutujemo šta je sa granom e , šta ako bismo u minimalno pokrivajuće stablo dodali još i granu e , koja formira petlju
- Teza je da je težina ove grane e manja od težine svih ostalih grana koje pripadaju minimalnom pokrivajućem stablu, dokaz je da ova težina nije manja od svih ostalih težina koje kreiraju minimalno pokrivajuće stablo, ta grana bi bila uključena u mst, neku drugu bismo izbacili
- Pošto ovde ne važi da je težina grane e manja od svih ostalih grana u mst, e ne bi činilo mst, već bi u mst zamenili granu f granom e , na ovaj način dobijamo bolje stablo
- Tako da zaista možemo da tvrdimo da važi teza da težina neke proizvoljne grane koja bi činila petlju uvek uvek veća od težine svih grana koje čine mst
- Da nije ovo slučaj mogli bi neku postojeću da zamenimo novom



MST I PARTICIJE

- particije - neke logične celine
- mi imamo dva mst, i mogućnost da te dve celine, particije spojimo, spojimo ih vezom sa najmanjom težinom, u ovom slučaju je to veza 7, pošto imamo dve grane sa težinom 7, pa možemo odabrati oba i na taj način dobijamo dva različita mst

ALGORITMI ZA DOBIJANJE DVA MST

1. PRIM-JARNIK ALGORITAM

- sličan Dijkstra algoritmu
- izaberemo čvor s i pravimo MST kao oblak čvorova počevši od s
- čvor v čuva labelu $d(v)$ – najmanja težina grane koja povezuje v sa oblakom
- u svakom koraku:
 - dodamo u oblak čvor u sa najmanjim $d(u)$
 - ažuriramo labele čvorova susednih sa u

-ovde mi ne dodajemo čvoru težinu koja odgovara sumi grana, nego dodajemo čvoru težinu koja odgovara najmanjoj težini grane do tog čvora

- Na početku početnom čvoru dodajemo težinu 0, svim ostalim čvorovima dodajemo težinu beskonačno, treba nam red sa prioritetom u koji ubacujemo težine naših čvorova, skidamo čvor sa najmanjim prioritetom i granu koja vodi do njega, i povezujemo taj čvor sa našim oblakom, izanaliziramo sve grane, dakle proveravamo da li postoji neka druga grana koja povezuje naš čvor sa ostatkom, upoređujemo ih, ako treba korigujemo težinu, ubacujemo najmanji čvor u oblak itd...

*ANALIZA

- operacije nad grafom

prolazimo kroz susedne grane po jednom za svaki čvor

- operacije sa labelama

za čvor z postavljamo rastojanje i labele $O(deg(z))$ puta

postavljanje labele traje $O(1)$

-operacije nad redom sa prioritetom

svaki čvor se dodaje jednom i uklanja jednom iz RSP, svaka operacija traje $O(\log n)$

ključ čvora u RSP se menja najviše $deg(w)$ puta, svaka zamena ključa traje $O(\log n)$
postavljanje labele traje $O(1)$

-Prim-Jarnik algoritam traje $O((n + m)\log n)$ ako je u implementaciji korišćena lista suseda

2. KRUSKAL ALGORITAM

- particija čvorova u klastere (grozdove)

inicijalno klasteri sa po jednim čvorom

održava se MST za svaki klaster

spajanje „najbližih“ klastera i njihovih MST

-red sa prioritetom čuva grane izvan klastera

ključ: težina

vrednost: grana

-na kraju rada algoritma: jedan klaster i jedno MST

- Ovaj algoritam predlaže da dodamo sve grane koje imamo u red sa prioritetom, skidamo najprioritetniju granu, posmatramo jedan čvor u kom se klasteru nalazi, posmatramo drugi čvor kom se klasteru nalazi, ako se nalaze u istom klasteru, onda ne radimo ništa, a ako se ne nalaze u istom, spajamo ta dva klastera u jedan

- Na početku ovaj algoritam predviđa da svaki čvor čini klaster i onda dodajemo grane i radimo sve gore objašnjeno

- Kako mi da znamo da li naš algoritam pripada jednoj particiji ili drugoj? Treba nam neka posebna struktura koja će reprezentovati tu particiju

Struktura podataka za Kruskal algoritam: particija

- algoritam rukuje šumom stabala

- RSP daje grane u redosledu rastućih težina

- grana se prihvata ako spaja različita stable

- potrebna je struktura podataka koja čuva particiju, tj. kolekciju disjunktnih skupova, sa operacijama

makeSet(u): kreiraj skup koji sadrži u

find(u): nađi skup koji sadrži u

union(A, B): zameni skupove A i B njihovom unijom

-Možemo particiju da formiramo pomoću:

1. LISTE

- imamo listu elemenata i imamo jedan početni element koji reprezentuje zapravo ime tog klastera

- svaki skup se čuva u sekvenci svaki element ima referencu na skup

find(u) traje $O(1)$ i vraća skup koji sadrži u

union(A, B): premeštamo elemente iz manjeg skupa u veći i ažuriramo reference na skup; traje $O(\min\{|A|, |B|\})$

-kada se element obradi, prelazi u skup koji je bar duplo veći, dakle svaki element se obrađuje najviše $\log n$ puta

- spajanje klastera pomoću union

- pronalaženje klastera pomoću find

- brzina je $O((n + m)\log n)$

operacije sa RSP: $O(m \log n)$

union-find operacije: $O(n \log n)$

2.STABLA

- čvor stabla čuva element i referencu na skup

- čvor v čija referenca pokazuje na v je istovremeno i skup

- svaki skup je stablo, njegov koren čuva referencu na sebe
- **union**: koren jednog stabla treba da pokazuje na koren drugog
- **find**: prati reference do korena koji pokazuje na samog sebe
- kada se radi union, neka koren manjeg stabla pokaže na koren većeg
- $O(n \log n)$ vreme za izvođenje n union-find operacija
 - svaki put kada pratimo referencu, idemo prema stablu koje je bar duplo veće od trenutnog
 - prema tome, pratimo najviše $O(\log n)$ referenci za svaki find
- Savet je da kada već pratimo putanju do korena, nema razloga da se ove putanje ne ažuriraju i da svaki čvor ne prevežemo na koren -> **kompresija putanje**: nakon find, neka sve reference čvorova koje smo upravo obišli pokazuju na koren

15. OBLAST – UPRAVLJANJE MEMORIJOM I B-STABLA

MEMORIJA RAČUNARA

- memorija je potrebna za implementaciju svake strukture podataka
- memorija je organizovana kao sekvenca **reči** gde se svaka reč sastoji od 4, 8 ili 16 bajtova (zavisno od računara)
- ove reči su numerisane od 0 do $N - 1$, gde je N broj reči dostupnih računaru
- broj povezan sa svakom od reči zove se **adresa**

KREIRANJE OBJEKATA

- u Python programu svi objekti se čuvaju u delu memorije koji se zove memory heap ili Python heap – ne treba mešati sa strukturom podataka koja se zove heap
- šta se dešava kada izvršimo nešto kao: `w = Widget()`, kreira se nova instanca klase i skladišti se negde na heapu
- memorijski heap je podeljen u **blokove** – kontinualne „parčiće“ memorije koji mogu biti fiksne ili promenljive veličine
- mora biti moguće brzo zauzimanje memorije za nove objekte
- jedno popularno rešenje – čuvanje slobodnih „rupa“ u heapu u povezanoj listi, zvanj **lista slobodnih blokova**
- odlučivanje kako dodeljivati blokove iz liste slobodnih prilikom zauzimanja (alokacije) memorije je deo **upravljanja memorijom**

UPRAVLJANJE MEMORIJOM

- postoji više načina za alokaciju memorije na heapu koji minimizuju fragmentaciju (kada računar nema uzastopnih lokacija, pa se podaci ne čuvaju na uzastopnim lokacijama što bi ubrzalo ustvari pristup, nego se fregmentiraju, odnosno dele na parčiće koji se na različitim memorijskim lokacijama onda stavljaju, fragmentacija ustvari usporava računar, pojava fragmentacije povećava verovatnoću da opet se i sledeći objekti opet smetaju tako fregmentirano)

best fit: pronađi u celoj listi onaj blok čija veličina je najbliža traženoj veličini, problem je što moramo da pristupimo svim elementima liste

first fit: kreni od početka liste i pronađi prvi blok koji je dovoljno velik, dosta brzo

next fit: traži se prvi sledeći dovoljno veliki blok počevši od prethodne pozicije; lista je cirkularna

worst fit: pronađi najveći slobodan blok u koji ubacujemo sadržaj

SKUPLJANJE ĐUBRETA

- **garbage collection:** proces otkrivanja „ustajalih“ objekata, oslobađanje memorije koju ti objekti zauzimaju, i vraćanje toga u listu slobodnih blokova

- da bi program mogao da pristupi objektu, mora imati referencu na njega (direktnu ili indirektnu)

takvi objekti su **živi** objekti

- živi objekti koji su direktno dostupni (postoji promenljiva koja sadrži referencu na njih) su **korenski objekti**

- **indirektna referenca** na živi objekat je referenca koja se nalazi u nekom drugom živom objektu

- najjednostavnija tehnika za implementiranje garbage collection-a je da se broji za svaki od objekata se skladišti i brojač koji zapravo broji koliko referenci postoji na taj objekat

BROJANJE REFERENCI

- **reference counting:** svaki objekat ima uz sebe i brojač referenci na sebe; brojač se ažurira prilikom operacija dodele vrednosti

- kada brojač padne na nulu, objekat može da se ukloni jer je nedostupan

- ali šta kada dva objekta imaju reference jedan na drugog, a nisu dostupni spolja (**cirkularne reference**)? - problem se rešava uvodjenjem generacija garbage collectiona - a

import sys

```
>>> a = 'test'
```

```
>>> b = [a]
```

```
>>> c = {'key': a}
```

```
>>> sys.getrefcount(a)
```

Generational garbage collection

- GC prati sve objekte u memoriji

- svaki novi objekat počinje život u „prvoj generaciji“

- kada se pokrene GC proces i objekat preživi, seli se u narednu (drugu) generaciju

- svaka generacija ima limit na broj objekata koji može da primi

- Python ima 3 generacije za GC

- statistika kaže: većina objekata u programu su kratkog veka

Java: mark-and-sweep algoritam

- svakom objektu dodeljena je oznaka (mark) da li je objekat živ
- kada odlučimo da je potrebno skupljati đubre, **zaustavimo sve druge aktivnosti** i
 - ukinemo mark za sve objekte na heapu
 - prođemo kroz sve module i sve korenske objekte označimo kao žive
 - odredimo da li su ostali objekti dostupni preko korenskih objekata – pretragom grafa po dubini

HIJERARHIJA MEMORIJE

- računari imaju hijerarhiju sa različitim vrstama memorije
- nivoi hijerarhije se razlikuju po veličini i udaljenosti od procesora
 - najbliži su interni **registri** procesora; pristup je vrlo brz ali ih ima vrlo malo
 - drugi nivo: **cache** memorija
 - treći nivo: **operativna** memorija (RAM)
 - četvrti nivo: **spoljašnja** memorija (diskovi)
- problem nastaje ako dodje do situacije da jednostavno nema više toliko radne memorije, našoj strukturi treba više, ideja je da se uvede virtualna memorija

VIRTUELNA MEMORIJA

- virtuelna memorija: adresni prostor velik kao kapacitet spoljne memorije
- stranice(delovi memorije) se premeštaju iz spoljne u operativnu memoriju kada su potrebne
 - virtuelna memorija ukida ograničenje veličine operativne memorije
- pošto ne možemo sav sadržaj spoljne memorije da učitamo, moramo da se odlučimo, ovde je važan princip vremenske lokalnosti - ako smo sad koristili jednu stranicu, šanse da će nam ona uskoro trebati je dosta veća

Strategije zamene blokova u operativnoj memoriji

- kada se traži nova stranica a operativna memorija je popunjena moramo izbaciti neku postojeću stranicu
- strategije za page replacement
 - LIFO
 - FIFO
 - Random

RANDOM STRATEGIJA

- zaboravi stranicu koju ćeš izbaciti slučajnim putem
- traje $O(1)$ ali ne pokušava da iskoristi vremensku lokalnost

FIFO STRATEGIJA

- jednostavna za implementaciju – potreban je red koji čuva reference na stranice u kešu
 - stranice se dodaju u red prilikom učitavanja
 - kada treba izbaciti stranicu, uklanja se prva stranica iz reda – $O(1)$
 - pokušava da iskoristi vremensku lokalnost

LRU STRATEGIJA

- least recently used – najdavnije korišćena stranica
- odlična politika ali implementacija može biti komplikovana
- potreban je adaptivni red sa prioritetom
- ako se implementira kao sortirana sekvenca pomoću povezane liste, uklanjanje je $O(1)$

BLOKOVI NA DISKU

- čuvamo veliku kolekciju elemenata koja ne može stati u operativnu memoriju
- spoljnu memoriju smo podelili na **disk blokove** – red veličine 8KB
- prenos bloka između spoljne i operativne memorije je **disk transfer** ili **I/O**
- velika razlika između vremena pristupa operativnoj i spoljnoj memoriji
- ⇒ želimo da minimizujemo broj disk transfera da bismo izvršili pretragu ili ažuriranje
- ovaj broj zovemo **I/O kompleksnost** algoritma (a , o)

(a,b) STABLO

- možemo predstaviti mapu za pretragu pomoću n -arnog stabla (a, b)
- stablo predstavlja uopštenje (2,4) stabla (a, b)
- stablo je n -arno stablo u kome svaki čvor ima između $a - 1$ i $b - 1$ dece
- ova struktura služi da vrlo lako možemo proći od korena do lista i list zapravo referencira neki sadržaj u sekundarnoj spoljnoj memoriji i onda nam je pristup lakši
- podešavanjem parametara a i b u odnosu na veličinu disk bloka možemo postići dobre I/O performance
- (a, b) stablo gde su a i b celobrojni parametri takvi da je $2 \leq a \leq (b + 1)/2$ je n -arno stablo pretrage sa sledećim osobinama:

veličina: svaki interni čvor osim korena ima najmanje a dece; koren ima najviše b dece
dubina: svi listovi imaju istu dubinu

* visina (a,b) stabla sa n elemenata je

$$\Omega(\log n / \log b)$$

$$O(\log n / \log a)$$

PRETRAGA I AŽURIRANJE U (a,b) STABLU

- pretraga se odvija kao u n -arnom stablu

- dodavanje slično $(2,4)$ stablu

- overflow nastupa kada se dodaje element u b -čvor

- čvor se deli pomeranjem median vrednosti u roditelja i zamenom čvora sa dva nova $(b + 1)/2$ -čvora

- uklanjanje slično $(2,4)$ stablu

underflow nastupa kada se ukloni element iz a -čvora

ako je brat a -čvor radi se fuzija

ako brat nije a -čvor radi se transfer

B-STABLO

- najpoznatija struktura za čuvanje mape u spoljnoj memoriji

- B-stablo reda d je (a,b) stablo za $a = d/2$ i $b = d$

- B-stablo sa n čvorova ima I/O složenost $O(\log_B n)$ za pretragu i ažuriranje, i troši $O(n/B)$ blokova, gde je B veličina bloka

