UNIVERZA V LJUBLJANI
FAKULTETA ZA RAČUNALNIŠTVO IN INFORMATIKO

Algoritmi in podatkovne strukture 2
(zapiski vaj)

februar 2015

Asistenti 2015:
Matevž Jekovec, Bojan Klemenc, Martin Stražar, Matej Pičulin

Nosilec predmeta: dr. Andrej Brodnik

Formalnosti vaj

Vaje so konzultacijske. Študenti lahko na vaje prinesejo svoje računalnike pri posvetovanju glede domačih nalog. Lahko pa se najavite na govorilnih urah pri svojem asistentu.

0.1 Domače naloge in zapiski

Vseh domačih nalog je 6 in bodo objavljene na učilnici. Vsaka domača naloga je sestavljena iz treh ali štirih teoretičnih nalog in ene programerske. Teoretične se oddajajo na učilnici v zapisu PDF (uporabite LaTeX ali kaj podobnega!), pri praktičnem pa bo potrebno nekaj sprogramirati v Javi. Programčki bodo relativno kratki in se bodo testirali z ogrodjem JUnit v ukazni vrstici — bistvo praktičnega dela je izvedba podatkovnih struktur in algoritmov, ki jih boste jemali na predavanjih, in ne učenje obrti programiranja.

0.1.1 Marmoset

Programčke za domače naloge se oddajajo s pomočjo sistema "Marmoset". Ta omogoča samodejno ocenjevanje oddane naloge. Pri izdelavi programerskih domačih nalog sistem ocenjuje naslednje: Točno stanje vaše podatkovne strukture po vsakem koraku in kolikokrat je poklicana katera od metod podatkovne strukture (ali ta zares deluje optimalno).

Za vsako domačo nalogo najprej naložite ogrodje (*skeleton*). Nato sprogramirate manjkajoče funkcije v dotičnem razredu. V ogrodju naloge je podano nekaj javnih testov (*public tests*), ki testirajo vaš programček (JUnit). Ko nalogo oddate, bo ta na željo uporabnika testirana tudi nad končnimi testnimi primeri (*release tests*), ob poteku roka za oddajo pa bo to storjeno samodejno. Študent

bo v nekaj minutah dobil odgovor, kateri testi so bili uspešni, kateri pa ne. Glede na uspešnost vašega programa bo sledila skupna ocena domače naloge.

Program lahko oddate večkrat in preverite rezultate. Ob vsakem zagonu končnih testnih primerov porabite en žeton. Na začetku imate 3. Žetoni se nato regenerirajo vsakih 10 minut. Ko vam jih zmanjka, morate počakati. S tem želimo vzpodbutiti študente, da čim prej začnejo z izdelavo domače naloge, saj bodo le tako lahko res dodelali podatkovno strukturo, da deluje na čim več testnih primerih.

Danes se prijavimo v sistem marmoset in do konca tedna oddamo testno, 0. domačo nalogo. Prijava v sistem je na http://marmoset.fri.uni-lj.si. Up. ime in geslo sta enaka kot za učilnico. Prenesite ogrodje (*skeleton*) za nalogo 0 (Obrni niz, najdi max), sprogramirajte funkciji obrniNiz(String niz) in najdiNajvecjega(int[] polje) v `src/java/psa/naloga0/Naloga.java`, zazipajte in oddajte (*submit*). Čez nekaj minut boste dobili rezultate (*view*).

Pozor Poskrbite, da je zapis oddane datoteke res ZIP in struktura enaka skeletonu (brez odvečnih map/podmap/nadmap).

0.1.2 Prevajanje dopolnilnih spletnih strani za knjigo Algorithms

Z letošnjim letom se bomo namesto zapiskov vaj in predavanj lotili prevajanja dopolnilnih spletnih strani knjige Algorithms (in C, in Java), avtorjev Roberta Sedgewicka in Kevina Wayna [SW11]. Uradna angleška domača stran knjige je . Spletne strani so že delno prevedene v slovenščino s strani koprskih študentov v letošnjem prvem semestru. Sedaj pa smo na vrsti mi.

Na učilnici je postavljen wiki z naslovom "Slovenski prevod spletnih strani knjige Algorithms", ki vsebuje (pod)poglavja strani, njihove roke in dodeljene študente. Vsak študent bo moral sodelovati pri prevajanju, če želi opraviti predmet. Postopek prevajanja je sledeč:

1. Vsak študent si do konca tedna izbere poglavje, pri katerem bo sodeloval, in se napiše pod njega v wiki na učilnici.
2. Do roka na wikiju je skupina študentov odgovorna, da poglavje prevede in kontaktira enega od asistentov.
3. Asistent preveri prevod in ga sprejme ali zavrne. V slednjem primeru

ima skupina en teden časa, da prevod dopolni. Dokler prevod ni končan, nihče od študentov v skupini ne dobi ocene.

4. Organizacija dela znotraj skupine (prevajanje, oblikovanje, lektoriranje, usklajevanje) je na študentih. Pomemben je končni izdelek.

Knjiga se nahaja v repozitoriju SVN: . Študent si namesti ustrezen odjemalec za SVN (npr. TortoiseSVN pod Windows, `apt-get install subversion` pod Debian) in sname zadnjo različico slovenske knjige. SVN s prevodom ni javen — up. ime in geslo za dostop do repozitorija je enak tistemu za učilnico ali marmoset. Študent si poleg delovne slovenske različice v SVN sname tudi zadnjo uradno angleško verzijo strani, da uskladi morebitno novo vsebino v zadnje pol leta.

Knjiga je napisana v HTML, za katerega predvidevamo, da študenti ne bodo imeli težav z razumevanjem. Kodiranje znakov v datotekah je UTF-8.

Poglavje 1

Slovar

1.1 Dvojiška drevesa

V splošnem je drevo definirano kot **povezan usmerjen graf brez ciklov** (*angl. Directed-Acylcic Graph* ali *DAG*) + dodatek v računalništvu **z določenim korenskim vozliščem**. V drevesu imamo več vrst vozlišč:

- koren (*angl. root*) — vrhnje vozlišče,
- listi (*angl. leaf* ali *terminal*) — vozlišča brez naslednikov,
- notranje vozlišče (*angl. internal node*) — vozlišča, ki niso koren in niso listi.

Po drevesu se sprehajamo od vozlišča proti listom. Vsako vozlišče v drevesu ima natanko enega starša in $0..k$ otrok, kjer je k stopnja vozlišča (*angl. degree*). Drevo s $k = 2$ je **dvojiško drevo** (*angl. binary tree*). S tem, ko smo omejili število naslednikov, smo vpeljali **strukturno invarianto**.

Nasledniki vozlišča (*angl. descendents*) so vsa vozlišča med vključno izbranim vozliščem in vključno z dosegljivimi listi.

Pravi nasledniki vozlišča x (*angl. proper descendents*) so vsi nasledniki vozlišča x brez x (korena).

Poddrevo vozlišča (*angl. subtree*) so vsi nasledniki izbranega podvozlišča (npr. levo poddrevo ali desno poddrevo vozlišča v dvojiškem drevesu).

Višina drevesa (oz. globina, gledano s korena, *angl. tree height, depth*) je najdaljša pot od korena do listov.

Uravnoteženo ali poravnano drevo (*angl. balanced tree*) je drevo, ki ima vse liste bodisi na istem nivoju bidisi kvečjemu en nivo više.

Izrojeno drevo (*angl. degenerate tree*) sestavljajo notranja vozlišča z le enim poddrevesom. Tako drevo je identično **povezanem seznamu**.

Delno poravnano drevo (*angl. partially balanced*) ni nujno uravnoteženo, zagotovo pa ni izrojeno.

Polno drevo (angl. *full tree*) stopnje k in višine h vsebuje $1 + k + k^2 + k^3 + ... + k^{h-1} = \frac{k^h - 1}{k-1}$ vozlišč, od tega k^{h-1} listov.

Gozd (angl. *forest*) je množica dreves.

Operacije nad drevesom so identične slovarju (`Insert()`, `Find()`, `Delete()`).

Sprehodi po drevesu Poznamo 3 vrste **sprehodov (*traversal*)** po drevesu:

- premi (*angl. preorder*) — najprej vrnemo vozlišče samo, nato obiščemo poddrevesa od leve proti desni,
- obratni (*angl. postorder*) — najprej obiščemo poddrevesa od levega proti desnemu, na koncu vrnemo vozlišče,
- vmesni (*angl. inorder*) — uporabno pri dvojiških drevesih. Najprej obiščemo levo poddrevo, nato vrnemo vozlišče, nato obiščemo desno poddrevo.

1.2 Dvojiško iskalno drevo

Dvojiško iskalno drevo (*binary search tree*) je **urejeno drevo**.

Uvedemo **vsebinsko invarianto**: V levem poddrevesu vozlišča so vsi elementi manjši, v desnem poddrevesu vozlišča pa vsi elementi večji od elementa, predstavljenega v vozlišču.

1.2.1 Operacije

Iskanje Poteka rekurzivno od korena navzdol. Če je iskani element manjši od trenutnega vozlišča, zavijemo v levo poddrevo, če ni, v desnega. Če je enak, vrnemo iskano vozlišče.

Vstavljanje Invarianta je zagotovljena, saj vstavljamo v levo poddrevo, če je novi element manjši ali v desno, če je večji.

Brisanje je bolj zanimivo. Najprej poiščemo ustrezen element, če ta obstaja. Algoritem nato loči naslednje primere:

- Če element nima naslednikov, ga enostavno zbrišemo.
- Če ima določeno le eno poddrevo, element zamenjamo z njegovim edinim otrokom.
- Če pa ima vozlišče določeni obe poddrevesi, ga zamenjamo z minimalnim elementom v desnem poddrevesu (ali z maksimalnim v levem — vseeno — mi bomo uporabljali prvo varianto). Operacija $findMin$ poišče minimalni element v desnem poddrevesu in je definirana tako, da vstopi v desno poddrevo ter se spušča po vseh levih naslednikih, dokler so ti določeni. Najdeni najbližji element odstranimo na enak način, kot poteka sicer brisanje.

Pozor Podobno spuščanje, kot je pri brisanju, smo že imeli pri številskih drevesih pri `RetrieveLeft/Right` operacijah v 8.2.

Ali invarianta pri brisanju še vedno velja? Prvi dve možnosti (brisanje elementa brez naslednikov ali z enim) sta očitni. Bolj zanimiv je tretji scenarij: Vozlišče zamenjamo z najmanjšim naslednikom v desnem poddrevesu. Invarianta ne bi držala, če bi bil novi element večji od desnega otroka ali manjši od levega otroka. To pa ni mogoče, saj je najmanjši element v desnem poddrevesu zagotovo manjši ali celo isti desnemu otroku zbrisanega elementa, saj smo se spuščali le po levih naslednikih (pokaži na tabli!). Novi element je zagotovo večji od levega otroka zbrisanega elementa, saj smo najprej zavili v desno poddrevo in se šele nato spuščali po levih naslednikih.

1.2.2 Analiza

Iskanje je odvisno od višine drevesa h in potrebuje kvečjemu $O(h)$ primerjav. Kolikšna pa je višina drevesa? Če je drevo polno, imamo $h = \lceil \lg n \rceil$, kjer je n število elementov. Lahko je pa izrojeno in imamo višino drevesa ter časovno zahtevnost iskanja kar n.

Nove elemente se vstavlja vedno v liste, kar zahteva $\Theta(h)$ primerjav.

Časovna zahtevnost brisanja je sestavljena najprej iz iskanja ustreznega elementa ($\Theta(h)$), pogojno pa še iz iskanja najmanjšega elementa v desnem poddrevesu ($\Theta(h)$). Najmanjši element v desnem poddrevesu je bodisi list, bodisi ima eno poddrevo prazno, v nasprotnem primeru ne bi bil element najmanjši. To pomeni, da brisanje ne more sprožiti novih rekurzivnih brisanj in časovna zahtevnost ostane $\Theta(h)$.

Kako se boriti proti izrojenosti (višina drevesa $\to n$)? Uvedemo mehanizme, ki zagotavljajo uravnoteženost vozlišč. Če so vsa vozlišča uravnotežena, dobimo polno drevo in tako višino drevesa $\lceil \lg n \rceil$.

1.3 Drevo AVL

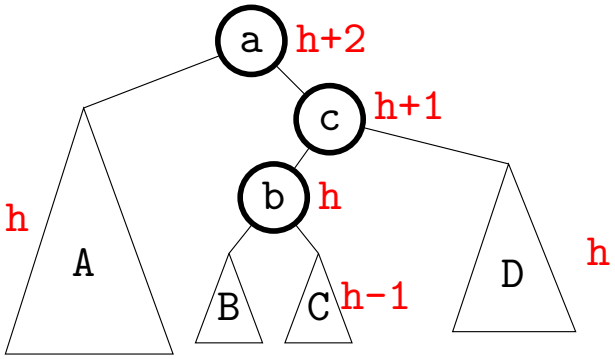
Drevo AVL avtorjev Georgija Adelson-Velskija in Evgenija Landisa je bilo predstavljeno leta 1962 v znanstveni reviji Sovjetska Matematika [AL62]. Je **delno poravnano** urejeno dvojiško drevo. Drevo ima poleg strukturne in vsebinske invariante dodano **ravnotežno invarianto**: Za vsako vozlišče se višini obeh poddreves razlikujeta največ za 1.

1.3.1 Operacije

`Find` deluje identično kot pri dvojiškem iskalnem drevesu.

Poglejmo bolj natančno operacijo `Insert`. Slika 1.1 prikazuje obliko drevesa AVL (druga oblika je simetrično obrnjena).

Ko vstavimo element, ga vstavimo v liste in so možni 4 scenariji: lahko pade v poddrevo A, B, C ali D. Če se višina omenjenih poddreves ne poveča (npr. element pade nekam v škrbino), se ne zgodi nič, saj nismo porušili ravnotežja in se ni potrebno ukvarjati z uravnoteženjem. Kaj pa, če se poddrevo poveča



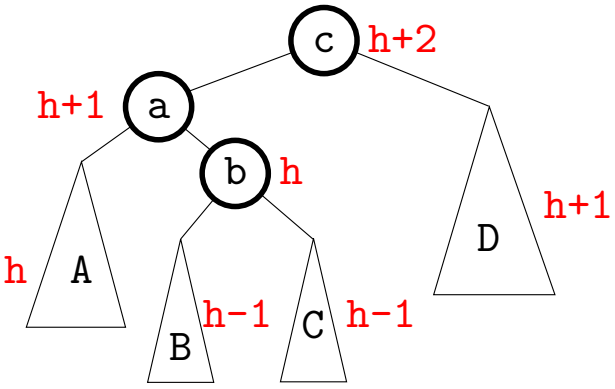
Slika 1.1: Primer drevesa AVL. Z rdečo so dopisane višine posameznih poddreves.

in invarianta ne drži več? Potrebno je popraviti drevo (uravnotežiti), tako da bo invarianta spet držala.

Iz primera razberemo naslednje relacije med elementi, ki morajo v vsakem primeru veljati:

$$A < a < B < b < C < c < D \tag{1.1}$$

Trenutni koren je a. Za koren pa bi lahko izbrali tudi druge, npr. b ali c.



Slika 1.2: Ustrezno uravnoteženo (enojna leva rotacija) drevo AVL z dodanim elementom v poddrevo D. To je edina veljavna postavitev od petih, ki ne krši invariante.

Ruska avtorja sta pokazala, da obstajajo štiri možne procedure uravnoteženja, ki pokrivajo vse primere kršenih invariant. Poimenovala sta jih **rotacije**. In

sicer levo ali desno enojno ali dvojno rotacijo. Definirane so v algoritmu 1.

V algoritmu imamo nekaj novosti:

- Funkcija `height(T)`, vrne višino poddrevesa vozlišča T.
- Funkcija `balance(T)` vrne `height(right(T))-height(left(T))`. Negativna številka pomeni, da je levo poddrevo "globlje", pozitivna pa desno.

Psevdokoda za vstavljanje in brisanje elementov v drevo AVL sta identični prvotnim za vstavljanje in brisanje elementov iz urejenega dvojiškega drevesa, le da je na koncu funkcije dodan še klic funkcije `Rebalance(T)`.

Algoritem 1: AVL-Rotations

```
1  function Rebalance(T)
2      if balance(T) < −1 then
3          if balance(left(T)) = −1 then
4              rotateR(T)
5          else
6              rotateLR(T)
7      else if balance(T) > 1 then
8          if balance(right(T)) = 1 then
9              rotateL(T)
10         else
11             rotateRL(T)

12 function RotateL(T)
13     pivot ← right(T)
14     setRight(T, left(pivot))
15     setLeft(pivot, T)
16     T ← pivot

17 function RotateR(T)
18     pivot ← left(T)
19     setLeft(T, right(pivot))
20     setRight(pivot, T)
21     T ← pivot

22 function RotateLR(T)
23     RotateL(left(T))
24     RotateR(T)

25 function RotateRL(T)
26     RotateR(right(T))
27     RotateL(T)
```

1.3.2 Analiza

Ruska avtorja sta z indukcijo Fibonaccijevega zaporedja dokazala, da je višina drevesa vedno $h \leq 1,44 \lg(n+1)$. V podrobnosti dokaza se ne bomo spuščali.

Ker je višina omejena, operacija `Find` pa je identična iskanju v dvojiškem iskalnem drevesu, je časovna zahtevnost iskanja $\Theta(\lg n)$ primerjav v najslabšem primeru.

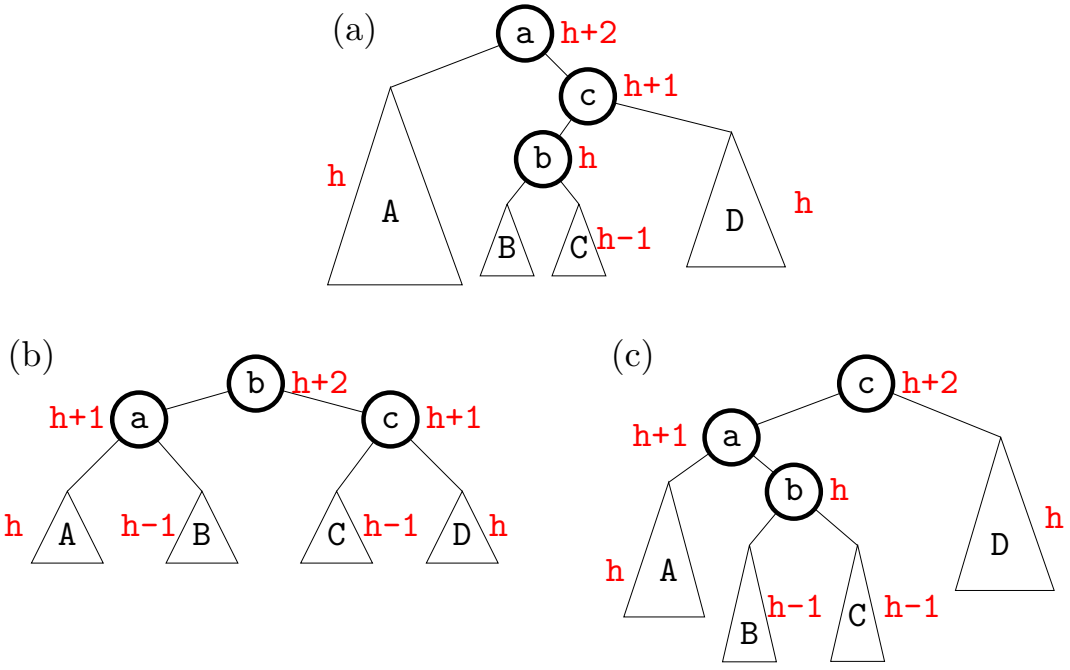
Operaciji `Insert` in `Delete` najprej potrebujeta $\Theta(\lg n)$ primerjav, da najdeta iskan element. Nato za uravnoteženje nekaj pisanj ($O(1)$). Koliko uravnoteženj pa se lahko sploh pokliče na eno vstavljanje ali brisanje elementa? Pri vstavljanju natanko 1, pri brisanju pa $\Theta(\lg n)$ (glej spodnji dokaz), tako da je časovna zahtevnost vstavljanja ali brisanja elementa še vedno $\Theta(\lg n)$ primerjav v najslabšem primeru.

Dokaz Vstavljanje — Obstajajo natanko tri veljavne postavitve vozlišč a, b in c grafa na sliki 1.1. Enkrat je a, enkrat b in enkrat c koren grafa (glej sliko 1.3). Ko vstavljamo vozlišče, lahko izberemo katero koli od treh postavitev (do njih pridemo z rotacijami, opisanimi v prejšnjem poglavju). Bistveno pa je, da za vsa štiri poddrevesa A, B, C in D vedno obstaja taka postavitev, da je ciljno poddrevo nižje od drugega poddrevesa (sorojenca). To pomeni, da se višina ciljnega poddrevesa lahko poviša, višina njegovega starša pa bo ostala nespremenjena, saj bosta sedaj poddrevesi postali šele izenačeni. Izbira prave postavitve oz. rotacije ustavi poviševanje poddreves in tako lokalizira problem. To pa pomeni, da bomo imeli kvečjemu 1 rotacijo na vstavljanje elementa.

■

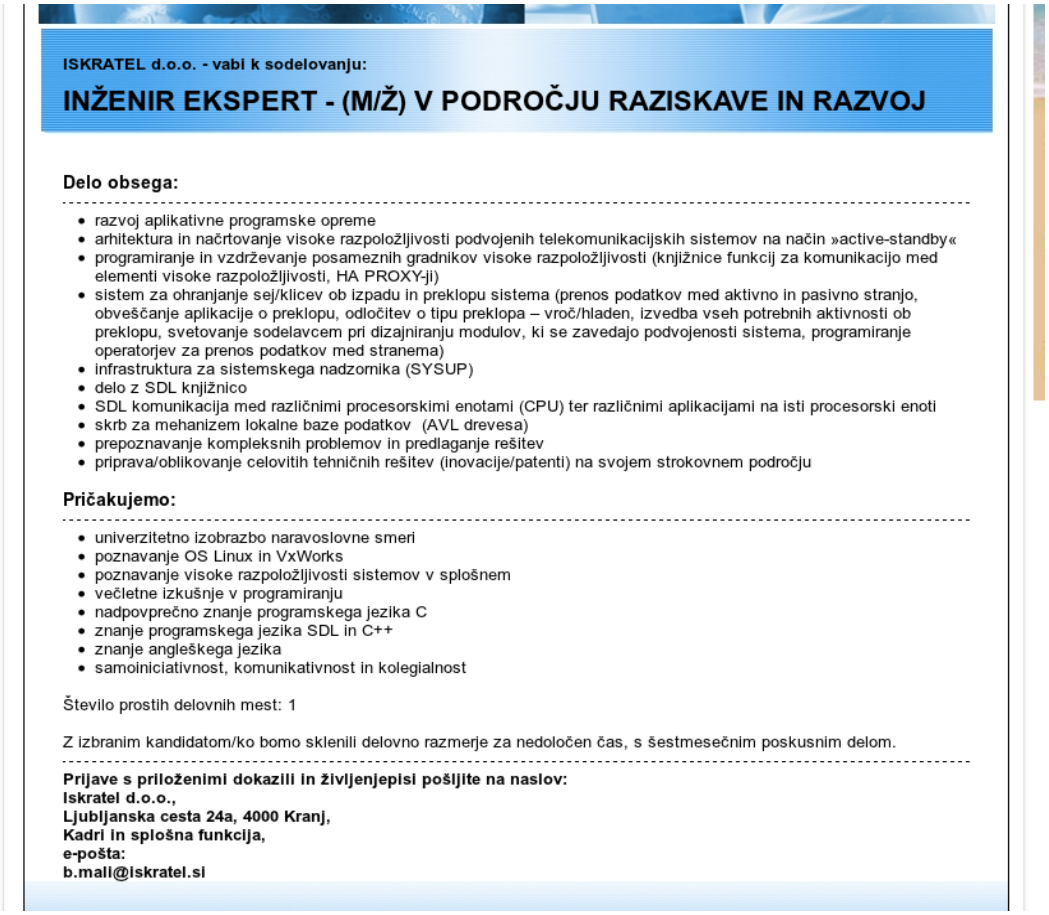
Dokaz Brisanje — Pri vstavljanju element vedno pade v poddrevesa A, B, C ali D. Zato smo lahko dokazovali v nasprotni smeri — elemente smo še pred vstavljanjem obrnili tako, da je višina ostala nespremenjena. Pri brisanju pa je težava v tem, da poleg elementov iz A, B, C ali D lahko brišemo tudi same elemente a, b in c. V tem primeru pa ne moremo vedno najti postavitve, ki bi bila veljavna, hkrati pa drevo enako visoko. Ker se višina spremeni, to lahko vpliva tudi na višja poddrevesa in sproži nova uravnoteženja. Število je omejeno z višino drevesa, torej $\Theta(\lg n)$.

■



Slika 1.3: Tri možne postavitve prvotnega drevesa 1.1. (a) prvotna postavitev, možnost vstavljanja v poddrevo A brez rotacij, (b) možnost vstavljanja v poddrevo B in C brez rotacij, (c) možnost vstavljanja v poddrevo D brez rotacij.

Znanje dreves AVL je v resnici pomembno. Na `mojedelo.si` je objavljen spodnji oglas, prikazan na sliki 1.4. Ena izmed obsegov dela pravi: skrb za mehanizem lokalne baze podatkov (AVL drevesa).



ISKRATEL d.o.o. - vabi k sodelovanju:

INŽENIR EKSPERT - (M/Ž) V PODROČJU RAZISKAVE IN RAZVOJ

Delo obsega:

- razvoj aplikativne programske opreme
- arhitektura in načrtovanje visoke razpoložljivosti podvojenih telekomunikacijskih sistemov na način »active-standby«
- programiranje in vzdrževanje posameznih gradnikov visoke razpoložljivosti (knjižnice funkcij za komunikacijo med elementi visoke razpoložljivosti, HA PROXY-ji)
- sistem za ohranjanje sej/klicev ob izpadu in preklopu sistema (prenos podatkov med aktivno in pasivno stranjo, obveščanje aplikacije o preklopu, odločitev o tipu preklopa – vroč/hladen, izvedba vseh potrebnih aktivnosti ob preklopu, svetovanje sodelavcem pri dizajniranju modulov, ki se zavedajo podvojenosti sistema, programiranje operatorjev za prenos podatkov med stranema)
- infrastruktura za sistemskega nadzornika (SYSUP)
- delo z SDL knjižnico
- SDL komunikacija med različnimi procesorskimi enotami (CPU) ter različnimi aplikacijami na isti procesorski enoti
- skrb za mehanizem lokalne baze podatkov (AVL drevesa)
- prepoznavanje kompleksnih problemov in predlaganje rešitev
- priprava/oblikovanje celovitih tehničnih rešitev (inovacije/patenti) na svojem strokovnem področju

Pričakujemo:

- univerzitetno izobrazbo naravoslovne smeri
- poznavanje OS Linux in VxWorks
- poznavanje visoke razpoložljivosti sistemov v splošnem
- večletne izkušnje v programiranju
- nadpovprečno znanje programskega jezika C
- znanje programskega jezika SDL in C++
- znanje angleškega jezika
- samoiniciativnost, komunikativnost in kolegialnost

Število prostih delovnih mest: 1

Z izbranim kandidatom/ko bomo sklenili delovno razmerje za nedoločen čas, s šestmesečnim poskusnim delom.

Prijave s priloženimi dokazili in življenjepisi pošljite na naslov:
Iskratel d.o.o.,
Ljubljanska cesta 24a, 4000 Kranj,
Kadri in splošna funkcija,
e-pošta:
b.mali@iskratel.si

Slika 1.4: Drevesa AVL so v resnici pomembna .-)

1.4 B-drevesa

B-drevesa so uravnotežena iskalna drevesa, namenjena učinkovitemu iskanju podatkov, shranjenih na sekundarnem pomnilniku (trdi disk). Utemeljila sta jih Rudolph Bayer in Edward M. McCreight leta 1972 [BM72]. Njun povod je bil vedno hitrejše delovanje CPU in pomnilnika, medtem ko se hitrost trakov in trdih diskov ni bistveno spremenila. Avtorja sta želela zmanjšati število branj

in pisanj s sekundarnega pomnilnika in sta razmišljala, kako podatke ustrezno zložiti.

Sektorji oz. bloki na disku so bili tradicionalno veliki od 512 B, danes do 4 KiB, pri SSD diskih tudi do 128 KiB. Če imamo na disku dvojiško drevo, bomo slej ko prej morali za vsak premik od vozlišča do vozlišča naložiti nov sektor (če so vozlišča zložena na disk po strategiji iskanja v širino — *Breadth First Search*).

Cilj pri B-drevesih je povečati velikost enega vozlišča (čim bolj približati velikosti sektorja na disku, vendar ne čez!), posredno znižati višino drevesa in zmanjšati število prehodov od korena do listov. Drevo zato postane sicer širše, ampak znamo vozlišča hitro obdelati in poiskati ustreznega naslednika, saj imamo vozlišče v glavnem pomnilniku.

B-drevesa se danes uporablja pri realizaciji datotečnih sistemov (*file system*), med drugim jih uporablja Applov HFS, Microsoftov NTFS in Linuxova ext4 in btrfs. B-drevesa so ravno tako uporabljena pri shranjevanju relacijskih podatkovnih baz (tam se uporablja posebna vrsta B^+ dreves).

1.4.1 Osnove

Kadar so podatki že v registrih procesorja, štejemo število primerjav, ki predstavljajo glavno ceno operacije. V primeru, da moramo do podatkov še dostopati, ozko grlo oz. glavno ceno operacije predstavljajo pogledi/dostopi (*probes*) do sekundarnega pomnilnika. V enem dostopu ne prestavimo samo enega podatka, ampak vsaj en blok podatkov (bločna shranjevalna naprava), recimo velikosti b.

B-drevesa so neke vrste razširitev dvojiških dreves. Glavne razlike so:

- Vozlišča imajo lahko tudi več kot dva naslednika (odvisno od stopnje vozlišča), recimo največ b.
- V vozliščih hranimo več ključev (odvisno od stopnje vozlišča) - natanko $b - 1$.
- Vsi listi so na isti globini h – uravnoteženost.

Izrek B-drevo reda b, $(b \geq 2)$ je drevo T s korenom $T.root$, ki zadošča naslednjim lastnostim:

1. Vsako vozlišče x ima $x.n$ ključev, kjer $\lceil b/2 \rceil - 1 \leq x.n < b$, razen korena, ki ima lahko tudi samo en ključ. Ključe označimo

z $x.key[i]$, kjer $1 \leq i \leq x.n$ in velja $x.key[i] < x.key[i+1]$ – urejenost ključev.

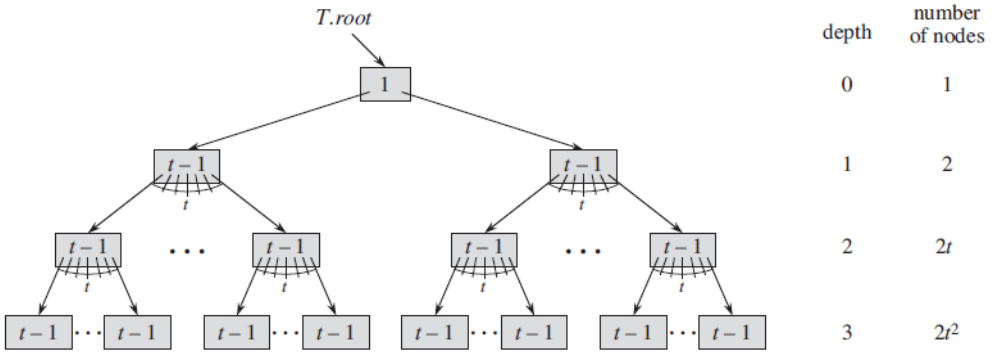
2. Vozlišče x, ki ima $x.n$ ključev, ima $x.n+1$ otrok $x.c[i]$, $(1 \leq i \leq x.n+1)$ razen listov, ki nimajo poddreves.

3. Velja, da so vsi elementi "levo" od ključa $x.key[i]$ (v poddrevesu s korenom $x.c[i]$) manjši od $x.key[i]$ in elementi "desno" od ključa $x.key[i]$ (v poddrevesu $x.c[i+1]$) večji od (ali enaki) $x.key[i]$. To velja za vse $1 \leq i \leq x.n$.

Prvi dve vrstici zagotavljata eksponentno povečevanje elementov na posamezni ravni, tretja pa omogoča iskanje po principu deli in vladaj. Vidimo, da večji kot je red b, manjša je višina h, s tem pa operacije hitrejše.

Pozor Kadar je $b = 3$ imajo vozlišča lahko 1 ali 2 ključa oz. 2 ali 3 otroke. Govorimo o 2-3 drevesih.

Naloga Koliko elementov lahko največ in najmanj vsebuje B-drevo višine h? Kolikšna je v splošnem višina B-drevesa z n elementi?



Slika 1.5: Višina B-drevesa, $t = \lceil b/2 \rceil$

Hranjenje elementov v vozliščih. Zaradi lažje razlage pri iskanju, vstavljanju in brisanju elementov v resnici mislimo na ključe teh elementov in predpostavimo, da so podatki povezani s ključem (element je par (ključ,podatek)) shranjeni v istem vozlišču kot ključ in z njim potujejo preko vozlišč oz. so lahko skupaj s ključi shranjeni kazalci na te podatke. Druga možnost je, da v vozliščih ne hranimo elementov (s podatki vred oz. kazalci na le-te), temveč zgolj ključe elementov, podatke pa hranimo v listih. V tem primeru govorimo o **B$^+$-drevesih**.

Algoritem 2: Iskanje v B-drevesu

```
1  B-Tree-Search(x, k) // vozlišče x, ključ (int) k ;
2  begin
3      i ← 1 ;
4      while (i ≤ x.n) ∧ (x.key[i] < k) do
5          i++ ;
6      if i ≤ x.n ∧ k = x.key[i] then
7          return (x, i) // vozlišče in indeks ključa k v vozlišču ;
8      else if x.leaf then
9          return ∅ ; //smo že v listu in še nismo našli ključa;
10     else
11         return B-Tree-Search(x.c[i],k)
```

1.4.2 Operacije na B-drevesih

Iskanje

Iskanje elementa po ključu k je analogno iskanju v BST - rekurzivno se sprehajamo po poti najmanjšega ključa v vozlišču, ki je večji ali enak od iskanega ključa. Ko najdemo ključ, vrnemo vozlišče in indeks iskanega ključa znotraj vozlišča. Če smo prišli do lista in še nismo našli ključa, ga očitno ni v drevesu in vrnemo `null`. Iskanje prikazuje algoritem 2.

Naloga Koliko je časovna in V/I zahtevnost iskanja po B-drevesu reda b z n elementi?

Vstavljanje

Vstavljamo element - par (ključ,podatki) - s ključem k. Enako kot prej, se ne obremenjujemo s pripadajočimi podatki. Uporabljamo naslednji postopek:

1. Nov ključ vstavljamo v list. Do tam se sprehodimo enako kot pri iskanju. Pri tem opazimo, da imamo pri iskanju vedno opravka z najmanjšim ključem `x.key[i]`, ki je večji od vstavljanega ključa k, in poddrevesom `x.c[i]`, v katerega vstavljamo. Razen, kadar vstavljamo v zadnje poddrevo, potem imamo opravka z `x.c[x.n+1]`.

2. Če je v listu še prostor (manj kot $b-1$ ključev), ključ vstavimo med ostale tako, da se ohrani naraščajoče zaporedje.
3. Če v vozlišču ni prostora, pomeni, da imamo skupaj z novim ključem b ključev, ki so urejeni po velikosti. Razdelimo jih na tri dele:
   - prvih $\lceil b/2 \rceil - 1$ ključev,
   - srednji ključ `x.key[`$\lceil b/2 \rceil$`]` (mediana) in
   - preostali ključi.

   Iz prvega in tretjega dela naredimo dve novi vozlišči x_1 in x_2, ki sta v resnici B-drevesi. Na koncu vrnemo staršu obe poddrevesi in srednji ključ `x.key[`$\lceil b/2 \rceil$`]`. Starš mora sedaj:
   - nadomestiti `x.c[i]` z x_2 in
   - pred `x.key[i]` vstaviti x_1 oz. x.
4. Pri staršu ponovimo bodisi korak 2 bodisi 3. Seveda v drugem primeru ponavljamo korak naprej proti korenu.
5. Če pa moramo korak 3 opraviti pri korenu (v bistvu razpolovimo koren drevesa), dobi celo drevo nov koren, ki bo imel samo en ključ in dve poddrevesi. S tem povečamo višino drevesa za 1.

V najslabšem primeru razpolovimo h vozlišč in zatorej potrebujemo največ $2h+1$ dostopov, kar je $2\log_b n + 1$.

Da se v primeru polnih vozliišč izognemo rekurzivnemu potovanju nazaj do korena, izvedemo vstavljanje tako, da tekom iskanja primernega vozlišča za vstavljanje **vnaprej razcepimo dovolj polna vozlišča**. Tak način uporablja algoritem 3.

Časovna zahtevnost. Pomožni algoritem 4, ki služi za razdeljevanje drevesa, teče linearno glede na red drevesa b (ki je sicer neka konstanta) in opravi konstantno dostopov do pomnilnika. Celotno vstavljanje zahteva $O(h)$ dostopov do pomnilnika, kjer je $h = O(\log_b n)$, h je višina drevesa, n število ključev (elementov) v drevesu. Število primerjav ključev je $O(b \log_b n)$, enako kot prej bi z bisekcijo lahko zmanjšali na $O(\lg n)$.

Algoritem 3: Vstavljanje ključa v B-drevo

```
1  B-Tree-Insert(T, k) // B-drevo T, ključ k ;
2  begin
3      r ← T.root ;
4      if r.n = b − 1 then
5          s = new Node() //funkcija ustvari prazno vozlišče ;
6          T.root ← s ;
7          s.leaf ← false ;
8          s.n ← 0 ;
9          s.c[1] ← r ;
10         B-Tree-Split-Child(s,1) ;
11         B-Tree-Insert-Nonfull(s,k) ;
12     else
13         B-Tree-Insert-Nonfull(r,k) ;
```

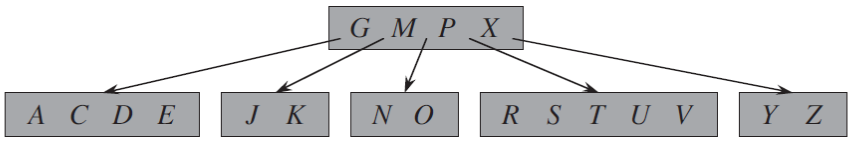
Algoritem 4: Razpolovitev vozlišča v B-drevesu

```
1  B-Tree-Split-Child(x, i) // nepolno vozlišče x, indeks i, kjer je x.c[i] polni
   otrok, ki ga razdelimo ;
2  begin
3      z ← new Node() //funkcija ustvari prazno vozlišče ;
4      y ← x.c[i] ;
5      z.leaf ← y.leaf ;
6      z.n ← ⌈b/2⌉ − 1 ;
7      for j = 1 to ⌈b/2⌉ − 1 do
8          z.key[j] ← y.key[j + ⌈b/2⌉] ;
9      if y.leaf = False then
10         for j = 1 to ⌈b/2⌉ do
11             z.c[j] ← y.c[j + ⌈b/2⌉] ;
12     y.n ← ⌈b/2⌉ − 1 ;
13     for j = x.n + 1 to i + 1 do
14         x.c[j + 1] ← x.c[j] ;
15     x.c[i + 1] ← z ;
16     for j = x.n to i do
17         x.key[j + 1] ← x.key[j] ;
18     x.key[i] ← y.key[⌈b/2⌉] ;
19     x.n + + ;
```

Algoritem 5: Vstavljanje ključa v B-drevo

```
1  B-Tree-Insert-Nonfull(x, k) // nepolno vozlišče x, ključ k ;
2  begin
3      i ← x.n ;
4      if x.leaf then
5          while i ≥ 1 ∧ k < x.key[i] do
6              x.key[i + 1] ← x.key[i] ;
7              i − − ;
8          x.key[i + 1] ← k ;
9          x.n + + ;
10     else
11         while i ≥ 1 ∧ k < x.key[i] do
12             i − − ;
13         i + + ;
14         if x.c[i].n = b − 1 then
15             B-Tree-Split-Child(x,i) ;
16             if k > x.key[i] then
17                 i + + ;
18         B-Tree-Insert-Nonfull(x.c[i],k) ;
```

Naloga Za izhodišče vzemimo drevo na sliki 1.6. Črke predstavljajo številske vrednosti od 1 do 26 in ustrezajo vrstnemu redu v angleški abecedi. V drevo po vrsti vstavite elemente B, Q, L in F. Narišite drevo po vsakem vstavljanju.



Slika 1.6: Začetno B-drevo pred vstavljanjem, $b = 6$.

Brisanje

Brisanje je analogno brisanju pri dvojiškem drevesu - izbrisani ključ nadomestimo s skrajnim levim (desnim) ključem v desnem (levem) poddrevesu.

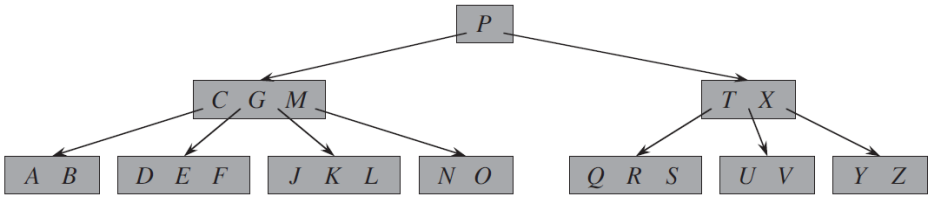
Pri brisanju ima vozlišče lahko manj kot $\lceil b/2 \rceil$ otrok in kršimo strukturno invarianto. V tem primeru moramo drevo preoblikovati.

Ločimo tri scenarije:

1. Iskan element (ključ) je najden in smo pristali v listu. Element enostavno zbrišemo. Naslednikov nima.
2. Iskan element (ključ) je najden v notranjem vozlišču. Možne variante:
   (a) Pogledamo levega otroka ključa. Če ima vsaj $\lceil b/2 \rceil$ ključev, premaknemo skrajno **desni** ključ otroka na mesto brisanega ključa. S tem rekurzivno pokličemo funkcijo brisanja desnega ključa.
   (b) Če je levi otrok ključa premajhen, analogno pogledamo desnega otroka ključa. Če ima vsaj $\lceil b/2 \rceil$ ključev, premaknemo skrajno **levi** ključ otroka na mesto brisanega ključa. S tem rekurzivno pokličemo funkcijo brisanja levega ključa.
   (c) Oba, levi in desni otrok brisanega ključa imata manj od $\lceil b/2 \rceil$ ključev. Oba otroka združimo v eno vozlišče in vmes vrinemo brisan ključ. Nato rekurzivno kličemo brisanje prvotnega ključa na novem vozlišču.
3. Iskan element (ključ) ni najden v trenutnem vozlišču, potrebno bo nadaljevati iskanje v enem izmed otrok (c_i). Zagotoviti moramo, **da ima** c_i **vsaj** $\lceil b/2 \rceil$ **ključev**. Možne variante:
   (a) c_i že ima vsaj $\lceil b/2 \rceil$ ključev. Rekurzivno nadaljujemo z brisanjem prvotnega ključa, tokrat v c_i.
   (b) c_i ima $\lceil b/2 \rceil - 1$ ključev, ampak ima sosednjega sorojenca — *sibling* (c_{i+1} ali c_{i-1}) z vsaj $\lceil b/2 \rceil$ ključi. Ta sorojenec nato donira skrajno levi oz. desni ključ vozlišču c_i. To ne gre čisto neposredno, ker lahko porušimo vsebinsko invarianto (urejenost ključev) starša. Zato vozlišče $c_{i\pm1}$ donira ključ staršu, starš pa vozlišču c_i. Naledniki premaknjenega ključa se prevežejo iz $c_{i\pm1}$ v c_i. Tako se znebimo odvečne rekurzije, ki bi jo premik ključa povzročil. Rekurzivno nadaljujemo z brisanjem prvotnega ključa, tokrat v c_i.

(c) Oba sosednja sorojenca $c_{i\pm1}$ vozlišča c_i imata $\lceil b/2 \rceil - 1$ ključev. Združimo vozliče $c_{i\pm1}$, c_i in ključ starša v eno vozlišče z $b - 1$ ključi. Rekurzivno nadaljujemo z brisanjem prvotnega ključa, tokrat v novem, združenem vozlišču.

Redka vozlišča drevesa želimo, analogno vstavljanju, **po poti navzdol združevati**. Če bodo vsa vozlišča po poti dovolj polna, se bomo tako izognili rekurzivnemu popravljanju drevesa navzgor, če bo zbrisano vozlišče potrebno preoblikovati.



Slika 1.7: Začetno B-drevo pred brisanjem.

Naloga Za izhodišče vzemimo drevo na sliki 1.7. Črke predstavljajo številske vrednosti od 1 do 26 in ustrezajo vrstnemu redu v angleški abecedi. Iz drevesa po vrsti odstranite elemente F, M, G, D in B. Narišite drevo po vsakem vstavljanju.

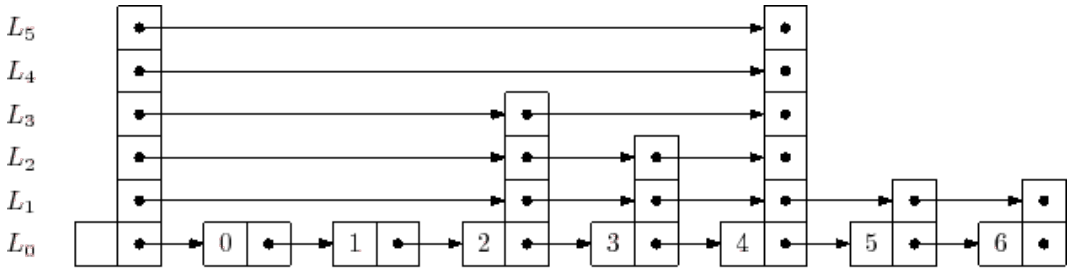
1.5 Preskočni seznam

1.5.1 Uvod

(Open Data Structures in Java, str. 77–93)

Preskočni seznam (*skiplist*) je leta 1990 predstavil William Pugh [Pug90] kot alternativo uravnoteženim drevesom. Preskočni seznami so naključnostna podatkovna struktura, saj pri delovanju **uporabljajo naključnost**, kar jih naredi **preprostejše za implementacijo kot uravnotežena drevesa**. V zameno za preprostost (ki je posledica uporabe naključnosti) je učinkovitost operacij (iskanje, vnos, brisanje) sicer zelo verjetna, vendar ne zagotovljena. Preskočni seznami se dobro obnesejo v vzporednih okoljih, saj je pri spremembah (vnos, brisanje) potrebno zakleniti manj vozlišč kot pri spremembah v uravnoteženih

drevesih.



Slika 1.8: Preskočni seznam

Preskočni seznam je hierarhična množica urejenih povezanih seznamov. Sam povezan seznam nam nudi možnost implementacije slovarja, vendar je časovna zahtevnost operacije iskanja reda $O(n)$, kjer je n število elementov (ključev) v seznamu.

Ideja preskočnega seznama je, da zgradimo hierarhijo $h + 1$ urejenih povezanih seznamov $L_0, \ldots, L_h$. Preskočni seznam z n elementi ima naslednje lastnosti:

- V L_0 je vseh n elementov, ki so v preskočnem seznamu.
- Vsak povezan seznam L_r vsebuje podmnožico elementov iz L_{r-1}, $r \geq 1$.
- Elemente za L_r dobimo tako, da "vržemo kovanec" za vsak element iz L_{r-1} in ga vključimo v L_r, če pade cifra.

Opozorimo, da eno vozlišče hrani en ključ oz. element, vendar imamo več vozlišč, saj imamo več seznamov. Kasneje bomo pri implementaciji videli, da to ne drži povsem, saj bomo zaradi učinkovitosti imeli v resnici zgolj n vozlišč, ki pa bodo realizirana tako, da bodo ustrezno hranila pripadnost posameznemu povezanemu seznamu.

Višina elementa (ključa) `x` je največja vrednost r, da je $\texttt{x} \in L_r$. Elementi, ki so zgolj v L_0 imajo višino 0. Opazimo, da je pričakovano število povezanih seznamov, v katerih se element nahaja, enako pričakovanemu številu metov kovanca preden dobimo grb. Namreč, dokler smo dobivali cifro, je bil element uvrščen v povezan seznam na višjem nivoju, potem pa ne več. Pričakovano število metov, da dobimo grb je 2, se pravi v povprečju element "pade na žrebu"

za vstop v L_2, saj v L_0 pride brez žreba. Posledično je pričakovana višina 1.

1.5.2 Implementacija in operacije

Predstavitev vozlišča. Uporabimo razred `Node`. Vozlišče tega razreda `u` sestoji iz ključa `x` (in morebitnih podatkov) in polja kazalcev na vozlišča `next`, kjer `u.next[i]` kaže na naslednika vozlišča `u` v $L_{\mathtt{i}}$, $0 \leq i \leq h$. Za današnje potrebe bo ključ tipa `double`, kot ponavadi pa bi lahko bil iz katerekoli urejene množice. Na ta način je ključ `x` (in morebitni pripadajoči podatki) shranjen zgolj enkrat (imamo zgolj eno vozlišče za en ključ), čeprav je del več povezanih seznamov. Ker enemu elementu pripada eno vozlišče, sedaj ne govorimo več o višini elementa temveč o višini vozlišča. Le-to vrne metoda `height()`, ki zgolj vrne `next.length` $-\,1$ (povezan seznam L_0 je na višini 0). Primer povezanega seznama prikazuje slika 1.8. Slika namerno prikazuje primer, kjer sta L_4 in L_5 enaka, saj je to povsem mogoče glede na naključnostno naravo strukture.

Iskanje. Na začetku vsakega povezanega seznama imamo glavo, posebno vozlišče, ki je vedno prisotno in ima višino enako največji višini izmed ostalih vozlišč v preskočnem seznamu. Bistvo preskočnih seznamov je, da obstaja hitra pot od vrha glave seznama L_h do vsakega vozlišča v L_0. Iskanje vozlišča `u` (ki mu pripada ključ `x`) poteka dokaj intuitivno. V trenutnem vozlišču (začnemo pri glavi L_h) pogledamo naslednika, kjer imamo tri možnosti:

1. Če naslednik vsebuje iskani ključ, ga (naslednika) vrnemo.
2. Če naslednik vsebuje ključ manjši od iskanega, se pomaknemo na naslednika in ponovimo postopek.
3. Sicer (naslednik je null ali vsebuje večji ključ) ostanemo na trenutnem vozlišču in se pomaknemo v seznam en nivo nižje (dokler ne pridemo do L_0) in zopet ponovimo postopek.

Tako ponavljamo dokler ne pridemo do vozlišča, katerega naslednik hrani iskani ključ, ali če smo v L_0 prišli do zadnjega vozlišča, potem vrnemo `null`, saj ključa (elementa) očitno ni v preskočnem seznamu. Primer iskanja vozlišča s ključem 4 je prikazan na sliki 1.8 – vozlišče najdemo takoj po pregledu naslednika glave. Postopek iskanja opisuje algoritem 6.

Algoritem 6: Iskanje vozlišča v preskočnem seznamu

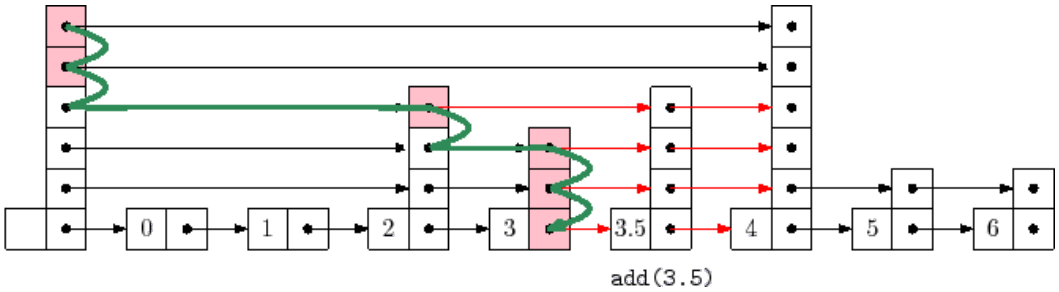
```
1  Node find(Key x) ; // ključ x
2  begin
3      Node u ← head ;
4      int r ← head.h ;
5      while r ≥ 0 do
6          while u.next[r] ≠ ∅ ∧ u.next[r].x < x do
7              u ← u.next[r] ; // gremo desno v seznamu L_r
8          if u.next[r] ≠ ∅ ∧ u.next[r].x = x then
9              return u.next[r] ; // če smo našli iskano vozlišče, ga vrnemo
10         r − − ; // gremo dol v L_{r−1}
11     return ∅ ;
12     /* očitno vozlišča z iskanim ključem ni, sicer bi ga vrnili v zunanji
       while zanki */
```

Vstavljanje. Pri vstavljanju potrebujemo metodo, ki nam naključno vrne višino, ki naj pripada novemu vozlišču. Naj ima metoda ime `pickHeight()`, s podrobnostmi implementacije se ne ukvarjamo. Pomembno je, da simulira metanje kovanca, torej mečemo kovanec in dokler pada glava, večamo višino, ko pade grb nehamo. Lahko že pri prvem metu pade grb, v tem primeru bo vozlišče višine 0, to pomeni, da bo zgolj v L_0.

Metodo `insert(x)` implementiramo tako, da v preskočnem seznamu poiščemo vozlišče, ki vsebuje x (v tem primeru ne spreminjamo preskočnega seznama in vrnemo `false`) oz. vozlišče, katerega naslednik vsebuje najmanjši ključ že večji od x. *Zato smo pri implementaciji iskanja najprej pogledali naslednika in se šele nato premaknili nanj po potrebi.* Nato x vstavimo v povezane sezname $L_0, \ldots, L_k$, pri čemer k določimo z že omenjeno `pickHeight()` metodo. Glede na našo implementacijo, vstavljanje v omenjene sezname pomeni ustvarjanje vozlišča s ključem x in poljem `next` velikosti `pickHeight`$() + 1$, ter posledično prirejanje naslednikov v polju `next` in prirejanje tega vozlišča v predhodnikovih poljih `next`.

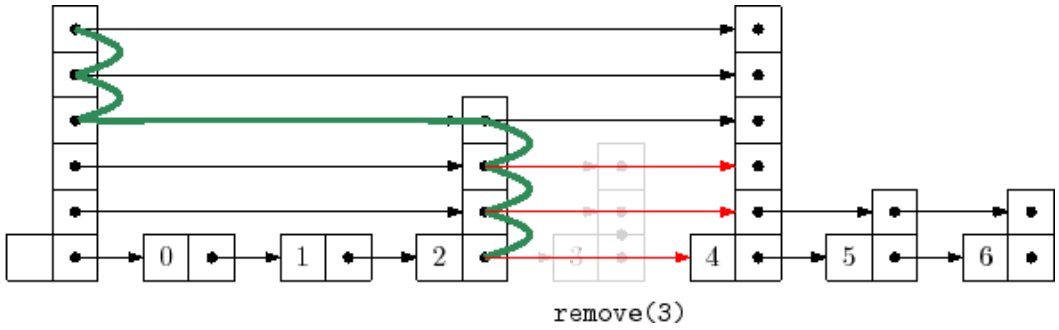
Pri vstavljanju uporabimo polje `stack`, ki beleži, pri katerem vozlišču se je pot iskanja pomaknila navzdol iz L_r na L_{r-1}. Konkretno, `stack[r]` je vozlišče v L_r, kjer se je pot iskanja premaknila navzdol v L_{r-1}. Pri vstavljanju x tako spremenimo vozlišča `stack`[0], ..., `stack`[k]. Implementacijo prikazuje

algoritem 7, primer vstavljanja ključa 3.5 pa slika 1.9.



Slika 1.9: Vstavljanje ključa 3.5 v preskočni seznam.

Brisanje. To operacijo izvedemo podobno kot vstavljanje, le da ni potrebe po skladu, ki beleži pot iskanja, saj lahko povezave preusmerjamo sproti. Ko se iskanje pomakne navzdol, preverimo če je $u.next[r].x == x$ in če je, povezavo preusmerimo na naslednika izbrisanega vozlišča. To počnemo vse do spodnjega nivoja 0, ko zbrišemo še zadnjo referenco na vozlišče, ki je vsebovalo ključ (in morebitne podatke) x ter vozlišče izbrišemo iz pomnilnika. Implementacijo prikazuje algoritem 8, primer brisanja pa slika 1.10.



Slika 1.10: Brisanje vozlišča s ključem 3 iz preskočnega seznama.

Naloga Nad preskočnim seznamom na sliki 1.8 izvedite naslednje zaporedje operacij: find(5), remove(0) insert(2.5) z višino 0, insert(5.5) z višino 3, remove(4), remove(6), insert(1.2) z višino 4.

Algoritem 7: Vstavljanje v preskočni seznam.

```
1  boolean insert(Key x); // ključ x
2  begin
3      Node u ← head ;
4      int r ← head.h ;
5      while r ≥ 0 do
6          while u.next[r] ≠ ∅ ∧ u.next[r].x < x do
7              u ← u.next[r]; // gremo desno v seznamu L_r
8          if u.next[r] ≠ ∅ ∧ u.next[r].x = x then
9              return false ; // če je ključ že v seznamu
10         stack[r − −] ← u ; // gremo dol in shranimo u
11     Node w ← new Node(x,pickHeight()) ; // ustvarimo novo vozlišče s
       ključem x in naključno višino
12     while head.h < w.h do
13         head.h + + ;
14         stack[head.h] ← head ; // povišamo glavo, če novi element višji
           od vseh dozdajšnjih
15     for i = 0 to w.h do
16         /* v tej zanki izvajamo potrebne prevezave */
17         w.next[i] ← stack[i].next[i] ;
18         stack[i].next[i] ← w ;
19     return true ;
```

Algoritem 8: Brisanje iz preskočnega seznama.

```
1  boolean remove(Key x) /* ključ x */ ;
2  begin
3      boolean removed ← false ;
4      Node u ← head ;
5      int r ← head.h ;
6      while r ≥ 0 do
7          while u.next[r] ≠ ∅ ∧ u.next[r].x < x do
8              u ← u.next[r] ; // gremo desno v seznamu L_r
9          if u.next[r] ≠ ∅ ∧ u.next[r].x = x then
10             removed ← true ;
11             u.next[r] ← u.next[r].next[r] ; // nastavimo na naslednika
               pobrisanega vozlišča
12             if u = head ∧ u.next[r] = ∅ then
13                 head.h − − ; // višina se zniža za 1
14         r − − ; // gremo dol
15     return removed ;
```

1.5.3 Zahtevnost preskočnih seznamov

Zahtevnost se pri preskočnih seznamih računa v povprečju (matematično upanje), saj se pri vnašanju elementov uporablja naključnost (višina vozlišča), ki posledično vpliva tudi na brisanje in iskanje. Višino določimo naključno s pomočjo kovanca, kot smo že opisali. Brez podrobnejše razlage imejmo v mislih, da je pri uporabi opisanega postopka povprečna (pričakovana) višina vozlišča 1 (element je v dveh seznamih, saj je višina prvega 0).

Kako je s prostorsko zahtevnostjo? Naj n označuje število elementov (ključev) v preskočnem seznamu. Kolikšno je potem število vozlišč? Teoretično bi lahko rekli, da imamo vozlišč toliko, kolikor je vsota št. elementov po vseh $h+1$ povezanih seznamih $L_0, \ldots, L_h$, torej $\sum_{i=0}^{h} |L_i|$. Povprečna višina vozlišča 1 pomeni, da je eno vozlišče v povprečju v dveh seznamih, torej je $\sum_{i=0}^{h} |L_i| = 2n$ v povprečju, kar je reda $O(n)$. Mi pa smo pri implementaciji rekli, da vsakemu ključu (elementu) pripada eno vozlišče določene višine. Nato v polju hranimo kazalce na naslednika v vsakem povezanem seznamu, katerega del je ključ. Posledično je število vozlišč enako številu elementov, torej n, kar je še vedno reda $O(n)$.

Kako pa je z časovno zahtevnostjo? Ker nimamo časa za zahtevno analizo povejmo zgolj, da je časovna zahtevnost vseh treh operacij odvisna od dolžine poti iskanja, ki je v povprečju dolga največ $2 \log n + O(1) = O(\log n)$.

Kaj pa če imamo res smolo? Npr. ob vsakem vstavljanju ključa (elementa) pade grb na kovancu že pri prvem metu. To pomeni, da so vsa vozlišča višine 0 oz. bomo imeli zgolj povezan seznam L_0. Posledično bodo operacije potrebovale $O(n)$ namesto $O(\log n)$ časa.

1.6 Razpršena tabela

Razpršeno tabelo (angl. *hash table*) je definiral Hans Peter Luhn leta 1953 znotraj tehničnega dokumenta v IBM-u [MS04, str. 9–15]. Uporabil je metodo z veriženjem. V istem obdobju je Gene Myron Amdahl s sodelavci uvedel tudi razpršeno tabelo z odprtim naslavljanjem po linearni metodi.

Pozor Po čem je še znan Amdahl? Amdahlov zakon! Največja pohitritev s povzporejanjem je $S(N) = \frac{1}{(1-P)+\frac{P}{N}}$, kjer je P delež povzporedljive kode in N število procesorjev.

1.6.1 Definicija

Razpršena tabela je podatkovna struktura, ki za poizvedbo podatkov po ključih k uporablja zgoščevalno funkcijo

$$h(k) : U \to \{0, 1, \ldots, m-1\}.$$

Rezultat zgoščevalne funkcije je indeks v našem polju velikosti m. Vedno velja, da izberemo $m \ll |U|$, kjer je U univerzalna množica vseh možnih elementov. Posledica tega pa je sovpadanje elementov ali "kolizije", se pravi, da imata dva elementa lahko enako zgoščeno vrednost. Na predavanjih ste podrobneje spoznali dve vrsti razpršenih tabel glede na način reševanja sovpadanj:

- razpršena tabela z veriženjem,
- razpršena tabela z odprtim naslavljanjem.

1.6.2 Dobra zgoščevalna funkcija

Od zgoščevalne funkcije je odvisno, kako hitro bo delovala razpršena tabela.

Metoda deljenja

$$h(k) = k \mod m$$

Vplivamo lahko na m. Kakšen mora biti? Dobre vrednosti m so praštevila, ki niso blizu potence 2.

Naloga Kaj se zgodi, če so vsi ključi oblike m^x?

Metoda množenja

$$h(k) = (k \cdot p) \mod m$$

Preden gremo modul računati, k pomnožimo z nekim p, najbolje s praštevilom.

Donald Knuth priporoča naslednjo zgoščevalno funkcijo:

$$h(k) = \lfloor m(kA \mod 1) \rfloor$$

, kjer $\mod 1$ pomeni decimalni del števila, A pa neko iracionalno ali transcendentno število (npr. $A = \hat{\Phi} = \frac{\sqrt{5}-1}{2} = 0,6180339887...$).

V praksi je računanje korena počasno. Pentium PRO je prvi, ki je dobil strojni ukaz za računanje kvadratnega korena znotraj FPU, vendar računanje ni potekalo v cevovodu. Kasneje je Intel dodal nabor MMX oz. danes SSE, kjer se tovrstne izračune dela v cevovodu. Računanje je tako hitro v primeru več izračunov, vendar imamo pri prvem vedno zamik, ki je bistveno daljši od zamika pri osnovnih aritmetičnih operacijah (vsota, razlika, množenje) in deljenja. Zato se konstanta A ponavadi kar natančno izračuna vnaprej, potem pa se le množi z njo.

Naloga Imamo velikost tabele $m = 1000$ in zgoščevalno funkcijo po Knuthu $h(k) = \lfloor m(kA \mod 1) \rfloor$ za $A = \hat{\Phi} = \frac{\sqrt{5}-1}{2}$. V katera mesta se preslikajo ključi 61, 62, 63, 64 in 65?

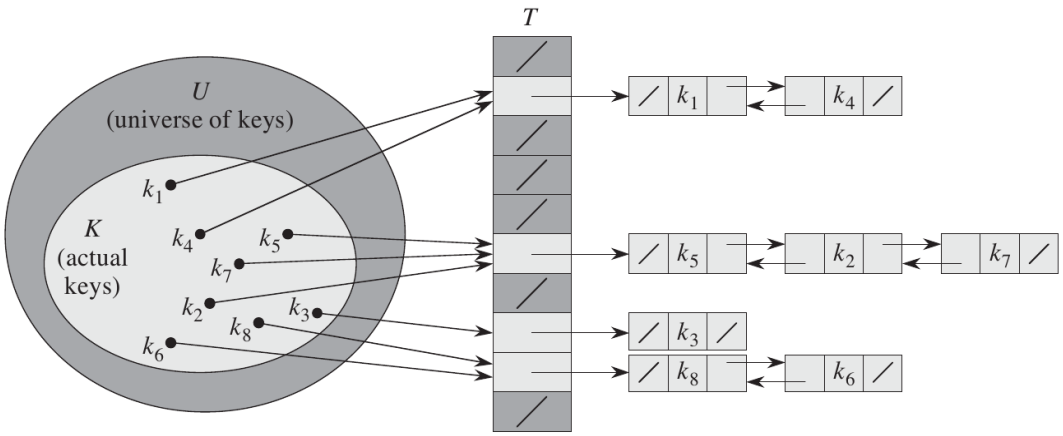
1.6.3 Izrazoslovje

Hash table prevajamo kot "razpršena tabela" zaradi njenega delovanja — ključe enakomerni porazporedi po tabeli. Drug izraz bi lahko bil "zgoščena tabela", saj s pomočjo zgoščevalne funkcije hrani zgoščene vrednosti ključev podatkov.

Še ena zmešnjava je pri izrazih za naslavljanje. *open addressing* pomeni odprto naslavljanje, sinonim za to je tudi *closed hashing* ali zaprto zgoščevanje, ker so zgoščene vrednosti "ujete" znotraj tabele. Drug način delovanja — veriženje — se lahko poimenuje tudi *open hashing* ali *closed addressing*.

1.6.4 Razpršena tabela z veriženjem

Razpršena tabela z veriženjem elemente z enako zgoščeno vrednostjo poveže v povezan seznam zunaj tabele. Slika 1.11 prikazuje delovanje razpršene tabele z veriženjem.



Slika 1.11: Primer razpršene tabele z veriženjem.

Pozor Vsako vozlišče v povezanem seznamu je sestavljeno iz ključa in vrednosti in ne samo vrednosti, saj drugače ne bi mogli primerjati elementa z iskanim ključem.

Poizvedba poteka v dveh korakih:

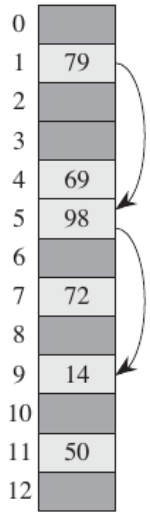
1. Izračunamo $h(k)$.
2. Sprehajamo se po povezanem seznamu z začetkom na $h(k)$, dokler ne pridemo do želenega elementa.

Naloga V razpršeno tabelo z veriženjem velikosti $m = 9$ in zgoščevalno funkcijo $h(k) = k \mod 9$ vstavi ključe 5, 28, 19, 15, 20, 33, 12, 17 in 10.

Naloga Kolikšna je hitrost poizvedbe v razpršenih tabelah v najslabšem primeru? Kako bi lahko pohitrili tako rešitev?

1.6.5 Razpršena tabela z odprtim naslavljanjem

Razpršena tabela z odprtim naslavljanjem ohranja vse elemente znotraj tabele. V primeru sovpadanja se naslov preslika na drugo mesto znotraj tabele. Vstavljanje in brisanje je zato malce zahtevnejše (psevdokoda kasneje). Slika 1.12 prikazuje delovanje razpršene tabele z odprtim naslavljanjem.



Slika 1.12: Primer razpršene tabele z odprtim naslavljanjem.

Na predavanjih ste omenili tri strategije pri odprtem naslavljanju, kjer i pomeni število prehodov in c, c_1 in c_2 vnaprej določene konstante:

- Z linearno funkcijo: $(h'(k) + ci) \mod m$.
- S kvadratično funkcijo: $(h'(k) + c_1 i + c_2 i^2) \mod m$.
- Z dvojnim zgoščevanjem: $(h'(k) + ih''(k)) \mod m$.

Pozor Funkcija $h(key, i)$ pri odprtem naslavljanju ima za razliko od zgoščevalne funkcije pri veriženju dva parametra: vhodni ključ in število poskusa.

Pri prvih dveh strategijah v primeru začetne preslikave v isto polje ohranimo nadaljnje zaporedje sovpadanj, kar je slabo. Zato je najpogostejša **tretja strategija**.

1.6.6 Operacije

Osnovno psevdokodo za vstavljanje ste napisali že na predavanjih. Vstavljanje je enostavno in skupno vsem strategijam reševanja sovpadanja pri odprtem naslavljanju, različna je le funkcija h.

Brisanje je bolj zanimivo. Če zbrišemo element, posredno zbrišemo tudi vse elemente, ki sovpadajo in so za njim. Potrebno je le **označiti, da je element zbrisan**, v primeru iskanja pa pri sovpadanju, ko obiščemo zbrisan element, nadaljujemo z iskanjem.

Iskanje je zdaj malce spremenjeno. Glej psevdokodo za vse operacije v 9.

1.6.7 Kaj narediti, ko je struktura polna?

Če imamo dobro izbrano strategijo v primeru sovpadanja (npr. dvojno zgoščevanje), imamo dovolj učinkovito strukturo tudi pri $\alpha = 90 - 95$ % zasedenosti. Kaj pa, ko nam zmanjka prostora?

Izrek Mehanizmi pri implementaciji, ki vodijo "evidenco" o številu elementov, velikosti m in stanju strukture, se imenujejo "računovodstvo" (*angl. accounting*).

Prva rešitev je, da naredimo novo razpršeno tabelo in preslikamo vse elemente iz stare v novo.

Prva rešitev je počasna, obstaja boljša varianta: Lahko naredimo novo tabelo velikosti $2m$ in od zdaj naprej vstavljamo vanjo. Staro pustimo pri miru, pri poizvedbah pa dostopamo do obeh tabel.

Naloga Kako naprej? Potem dobimo še eno velikosti $4m, 8m, 16m$ ipd. Iskanje se nam upočasni, saj ne vemo, v kateri tabeli se nahaja

Algoritem 9: Operacije v razpršeni tabeli z odprtim naslavljanjem, ki podpira brisanje.

```
1  Insert(key):
2  begin
3      foreach i ∈ {0..m − 1} do
4          if table[h(key, i)] = ∅ ∨ isDeleted(h(key, i)) then
5              table[h(key, i)] ← key ;
6              return i ;
7      return −1 ; // sproži napako!
8  Find(key):
9  begin
10     foreach i ∈ {0..m − 1} do
11         if table[h(key, i)] = key then
12             return table[h(key, i)] ;
13         else if table[h(key, i)] = ∅ then
14             break;
15     return ∅ ;
16 Delete(key):
17 begin
18     foreach i ∈ {0..m − 1} do
19         if table[h(key, i)] = key then
20             markDeleted(h(key, i)) ;
21             break;
22         else if table[h(key, i)] = ∅ then
23             break;
```

iskan element.

1.6.8 Analiza

Pričakovana časovna zahtevnost je vedno $O(1)$.

Vendar, ker imamo opravka z verjetnostjo, se da izračunati pričakovano število poskusov glede na zasedenost tabele (*load factor*) α:

$$\frac{1}{\alpha} \lg \frac{1}{1-\alpha} + \frac{1}{\alpha}$$

1.7 Bloomov filter

Bloomov filter je podatkovna struktura, ki ima definirani dve operaciji: `Insert(`x`)`, ki vstavi ključ x v Bloomov filter in `Find(`x`)`, ki poišče, ali smo vstavili ključ x v Bloomov filter. V primerjavi z večino izvedb slovarjev, ki smo jih spoznali do sedaj, Bloomov filter ne vrne vedno pravilnega odgovora. Če `Find(`x`)` vrne, da ključ ne obstaja, potem je odgovor vedno pravilen. Če pa vrne, da ključ obstaja, pa je odgovor lahko tudi napačen – napačno pritrdilen (angl. false positive) z določeno verjetnostjo. Seveda lahko Bloomov filter uporabimo le, če je v naši problemski domeni sprejemljivo, da imamo napačno pritrdilne odgovore o vsebovanosti.

Za izvedbo Bloomovega filtra ustvarimo tabelo m bitov, ki so na začetku postavljeni na 0, in definiramo k neodvisnih zgoščevalnih/razpršilnih funkcij, ki preslikajo vrednost ključa v indeks v bitnem polju. Pri vstavljanju s pomočjo k zgoščevalnih funkcij izračunamo k indeksov v tabeli, kjer postavimo bite na 1 (ne glede na prejšnje stanje bita). Pri iskanju prav tako izračunamo k indeksov. Če je vsaj na enem od indeksov bit 0, potem zagotovo ključ še ni bil vstavljen v Bloomov filter. Če so vsi 1, potem lahko po n vstavljenih ključev z verjetnostjo $f \approx 1-(1-e^{\frac{-kn}{m}})^k$ trdimo, da je bil ključ vstavljen v Bloomov filter.

Naloga Vzemimo Bloomov filter dolžine $m = 11$ in dve zgoščevalni funkciji:

1. $h(x) = (vzemi\ lihe\ bite\ dvojiškega\ zapisa\ števila\ x) \mod m$
2. $g(x) = (vzemi\ sode\ bite\ dvojiškega\ zapisa\ števila\ x) \mod m$

Narišite Bloomov filter po vstavljanju ključev 25, 159, 585. Sedaj

poiščite, če obstajajo ključi 118 in 162. Ali pride od napačnih pritrdilnih odgovorov? Kakšna je verjetnost napačnega pritrdilnega odgovora pri obeh iskanjih?

Poglavje 2

Disjunktne množice

Delo z disjunktnimi množicami danes srečamo pri analizi grafov, npr. pri gradnji najmanjšega vpetega drevesa, pri socialnih omrežjih, pri Googlovem pajku iskalnika itd. Problem se velikokrat imenuje tudi *Union-Find*.

Podatkovna struktura za delo z disjunktnimi množicami je definirana nad zbirko disjunktnih množic $S = \{S_1, S_2, ..., S_k\}$. Vsaka množica vsebuje enega ali več elementov, od katerih je eden **predstavnik množice**. Definirajmo naslednje operacije za delo z disjunktnimi množicami:

- `Make-Set(x)` — ustvari množico z enim elementom (in tudi predstavnikom) x,
- `Union(x, y)` — unija množice, ki vsebuje x in množice, ki vsebuje y v novo množico,
- `Find-Set(x)` — vrne referenco na predstavnika množice, ki vsebuje element x.

Definirajmo funkcijo `Connected-Components(G)`, ki za podan neusmerjen graf G zgradi disjunktne množice glede na povezanost elementov v grafu (isto kot

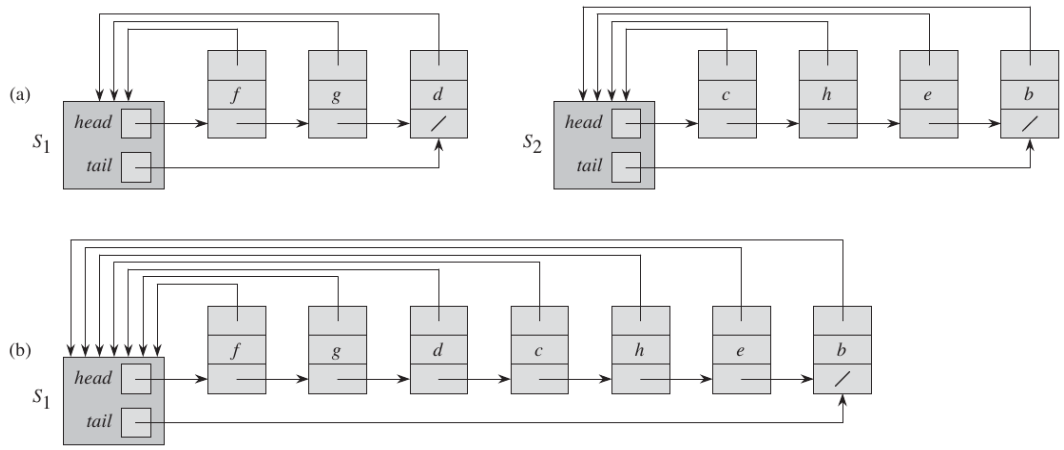
na predavanjih).

Algoritem 10: Connected-Components(G)

```
1 Vhod: neusmerjen graf G = (V, E)
2 foreach vertex v ∈ V do
3     Make-Set(v)
4 foreach edge (u, v) ∈ E do
5     if Find-Set(u) ≠ Find-Set(v) then
6         Union(u, v)
```

Naloga Pokličemo `Connected-Components(G)` nad neusmerjenim grafom $G = (V, E)$ s k **povezanimi komponentami** (to so zaprte skupine vozlišč, iz katerih lahko pridemo iz vsakega v vsakega). Kolikokrat sta poklicani operaciji `Find-Set(x)` in `Union(x,y)` v odvisnosti od $|V|, |E|, k$?

2.0.1 Izvedba s povezanim seznamom in z uteženo unijo



Slika 2.1: (a) Disjunktni množici S_1 in S_2. Vsako vozlišče poleg vrednosti sestavlja tudi referenca na glavo podatkovne strukture in referenca na naslednje vozlišče. Glava podatkovne strukture vsebuje dve referenci: na prvo in na zadnje vozlišče v strukturi. (b) Unija množic S_1 in S_2 v novo množico S_1.

Imejmo skupno m operacij v pravilnem vrstnem redu `Make-Set`, `Union` in

`Find-Set` od katerih je n operacij `Make-Set` (imamo torej n začetnih množic). `Union(`S_1`,` S_2`)` lahko izvedemo na dva načina:

1. Lahko pripnemo vse elemente druge množice k prvi. Časovna zahtevnost unije je potem linearna v odvisnosti od velikosti druge množice. Problem: Imejmo vse množice S_i enako veliko in zelo ponesrečeno kličemo unije: `Union(`S_2`,` S_1`)`, `Union(`S_3`,` S_2`)`, ..., `Union(`S_n`,` S_{n-1}`)`. Čas, potreben za unije se izrodi v $O(n^2)$, saj imamo na desni strani vedno že zgrajeno disjunktno množico z veliko elementi, pri kateri moramo vse elemente prevezati na glavo množice na levi strani.

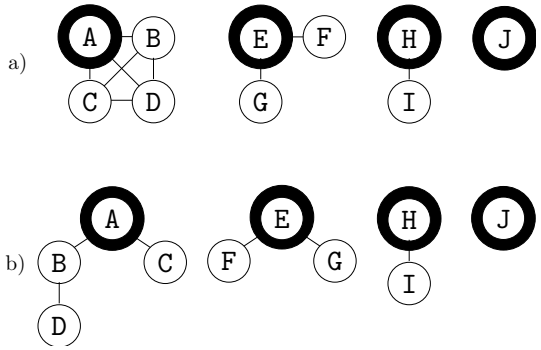
2. `Union` kličemo tako, da vedno pripnemo manjšo množico k večji. Ta pristop se imenuje **utežena unija**, kjer je prvi operand težja in drugi lažja množica ter utež pomeni število elementov v množici.

   Pozor Beseda "hevristika" pomeni priti hitreje do rešitve in jo lahko obravnavamo kot "optimizacija". Za razliko od optimizacije pa so hevristike lahko tudi suboptimalne (ne privedejo vedno do čisto eksaktne rešitve). Utežena unija je optimalna hevristika.

   Kaj smo pridobili? Poglejmo za zgornji scenarij operacij, koliko pisanj potrebujemo, če je na desni strani unije vedno manjša množica. Ker imamo n vozlišč, naredimo lahko kvečjemu $n-1$ unij oz. pisanj novih referenc v elemente na glavo strukture. Postavimo se v vlogo enega elementa in poglejmo, kolikokrat mu lahko zamenjamo referenco. Prvič ko jo, dobimo novo množico velikosti vsaj 2 (unija dveh množic s po enim elementom). Drugič ko nastavimo referenco, imamo očitno unijo množic velikosti vsaj 2 (če bi bilo manj, bi bila množica na levi strani in ne bi spreminjali referenc), torej nova množica šteje vsaj 4 elemente. Opazimo, da je rast ciljne množice eksponentna, elementu pa tako lahko spremenimo referenco kvečjemu $\lceil \lg n \rceil$-krat. Za vseh možnih $n-1$ unij potrebujemo največ $O(n \lg n)$ pisanj oz. za vse operacije skupaj $O(m + n \lg n)$. Časovna zahtevnost unij je $O(\lg n)$ *amortizirano* (kaj to pomeni, bomo videli v nadaljevanju).

2.0.2 Izvedba z drevesom in stiskanje poti

Vsaka množica je predstavljena kot drevo s korenom, ki je predstavnik množice (glej sliko 2.2). Operacija `Find-Set` pleza od vozlišča navzgor, dokler ne doseže korena drevesa. Da so drevesa uravnotežena, uvedemo **unijo glede**



Slika 2.2: (a) izvorni graf s poudarjenimi predstavniki (b) disjunktne množice za izvorni graf, implementirane s pomočjo dreves.

na globino. Ideja je v tem, da vedno pripnemo plitvejše drevo h korenu globljega, saj bi se v nasprotnem primeru drevesa lahko izrodila v povezan seznam. Uvedemo še dodatno optimizacijo: **stiskanje poti**. Tu je ideja, da vsa poddrevesa poskušamo približati korenu drevesa, tako da drevo sploščimo. Ker je pri `Find-Set` zanimiva le relacija spodnje vozlišče → starš (*parent(v)*) in ne obratno (*children(v)*), nimamo težav z implementacijo vozlišča.

Naloga Primerjajmo izvedbo s povezanim seznamom (s počasno unijo) in z drevesom z uteženo unijo glede na naslednje zaporedje operacij:

Algoritem 11: Zaporedje operacij make-set, union, find-set

```
1  for i ← 1 to 16 do
2      Make-Set(x_i)
3  for i ← 1 to 15 by 2 do
4      Union(x_i,x_{i+1})
5  for i ← 1 to 13 by 4 do
6      Union(x_i, x_{i+2})
7  Union(x_1,x_5)
8  Union(x_11, x_13)
9  Union(x_1, x_10)
10 Find-Set(x_2)
11 Find-Set(x_9)
```

Poglavje 3

Ponovitev za zamudnike APS1: Asimptotična zahtevnost in dokazovanje pravilnosti

3.1 Ocena velikosti funkcij

Zahtevnost algoritmov (**časovno** ali **prostorsko**) ponavadi izrazimo kot funkcijo velikosti vhoda, pri čemer velikost vhoda ponavadi označimo z n. Pogosto se ne izplača vlagati truda v izračun funkcije, ki izraža natančno zahtevnost algoritma, saj z večanjem n (ko gre $n \to \infty$) členi manjšega reda in konstanta ob členu največjega reda izgubljajo pomen. Zato pogosto, raje kot o natančni zahtevnosti algoritma, govorimo o **asimptotični zahtevnosti**. To pomeni, da ocenjujemo red velikosti funkcij, torej kako je zahtevnost algoritma odvisna od velikosti vhoda v limiti.

Recimo, da imamo časovno zahtevnost enako

$$T(n) = 13n^3 + 3n^2 + 5n.$$

Ko gre $n \to \infty$, členi manjšega reda v $T(n)$ izgubljajo pomen, prav tako pa konstanta 13 ob členu največjega reda. V tem primeru govorimo o "kubični" asimptotični zahtevnosti algoritma.

3.1.1 Asimptotični zapisi

Definirajmo vrste asimptotičnega zapisa funkcij. Opozorimo, da asimptotični zapis velja za matematične funkcije in je tako neodvisen od tega, kaj funkcija predstavlja. V našem primeru, bodo funkcije predstavljale časovno ali prostorsko zahtevnost algoritmov. Te funkcije so navadno definirane nad nenegativnimi celimi števili, a bomo asimptotično notacijo razširili in obravnavali kar funkcije, ki slikajo iz realnih števil v realna števila, prav tako so realna števila konstante.

Formalni zapis	Limita
$O(g(n)) = \{f(n) : \exists c, n_0, \text{ da je } 0 \leq f(n) \leq cg(n), \ \forall n \geq n_0\}$	$\lim_{n\to\infty} \frac{f(n)}{g(n)} < \infty$
$\Omega(g(n)) = \{f(n) : \exists c, n_0, \text{ da je } 0 \leq cg(n) \leq f(n), \ \forall n \geq n_0\}$	$\lim_{n\to\infty} \frac{f(n)}{g(n)} > 0$
$\Theta(g(n)) = \{f(n) : \exists c_1, c_2, n_0, \text{ da je } 0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n), \ \forall n \geq n_0\}$	$0 < \lim_{n\to\infty} \frac{f(n)}{g(n)} < \infty$
$o(g(n)) = \{f(n) : \forall c > 0 \ \exists n_0, \text{ da je } 0 \leq f(n) < cg(n), \ \forall n \geq n_0\}$	$\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$
$\omega(g(n)) = \{f(n) : \forall c > 0 \ \exists n_0, \text{ da je } 0 \leq cg(n) < f(n), \ \forall n \geq n_0\}$	$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty$
$f(n) \sim g(n) = \{f(n) : \forall \epsilon > 0 \exists n_0 \forall n > n_0 \left\lvert \frac{f(n)}{g(n)} - 1 \right\rvert < \epsilon\}$	$\lim_{n\to\infty} \frac{f(n)}{g(n)} = 1$

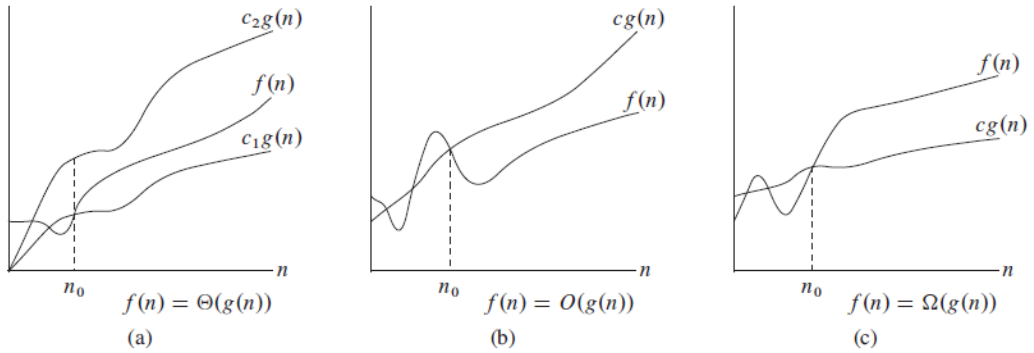
Θ-zapis

Za dano funkcijo $g(n)$ pomeni zapis $\Theta(g(n))$ *množico funkcij* definirano takole:

$$\Theta(g(n)) = \{f(n) : \exists c_1, c_2, n_0, \text{ da je } 0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n), \ \forall n \geq n_0\},$$

pri čemer so c_1, c_2 in n_0 pozitivne konstante. Zgornja definicija pomeni, da funkcija $f(n)$ pripada množici $\Theta(g(n))$, če obstajata taki pozitivni konstanti, c_1 in c_2, da je $f(n)$ "ujeta" med $c_1 g(n)$ in $c_2 g(n)$, za dovolj velike n.

Ker je $\Theta(g(n))$ množica, bi morali pisati $f(n) \in \Theta(g(n))$, vendar v praksi pišemo enačaj namesto $\in$. S takim zapisom, želimo povedati, da je za $n \geq n_0$ funkcija $f(n)$ do konstante enaka funkciji $g(n)$ oz. da je $g(n)$ **asimptotično tesna meja** za $f(n)$. Iz definicije tudi izhaja, da morajo biti $f(n)$ in $g(n)$ asimptotično nenegativne – večje ali enake nič, za $n \geq n_0$.



Zapis s tildo $\sim$

Skupina inženirjev in teoretikov, zbranih okoli Roberta Sedgewicka, velikokrat uporablja zapis s tildo $f(n) \sim g(n)$.

Tilda se bere $f(n)$ **je reda** $g(n)$. Formalno je definirana takole:

$$f(n) \sim g(n) = \{f(n) : \forall \epsilon > 0 \exists n_0 \forall n > n_0 \left| \frac{f(n)}{g(n)} - 1 \right| < \epsilon\}$$

Razlika med zapisoma Θ in $\sim$ je, da $\sim$ zahteva poleg enakega reda velikosti vodilnih členov funkcij $f(n)$ in $g(n)$ tudi enak konstantni faktor. To pomeni, da:

$$1,5n \log n \in \Theta(n \log n)$$

medtem ko

$$1,5n \log n \nsim n \log n$$

Če npr. ocenjujemo hitrost delovanja algoritma s funkcijo Θ ali $\sim$, nam $\sim$ poda tudi informacijo o konstantnem faktorju pred vodilnim členom. Pri algoritmih je velikokrat problem ravno ta faktor — teoretično zanimiv algoritem, analiziran s pomočjo funkcije Θ, da odlično časovno zahtevnost, vendar v praksi ne deluje dovolj hitro ravno zaradi tega faktorja.

Naloga Imejmo algoritem z natančno časovno zahtevnostjo

$$T(n) = \frac{1}{2}n^2 - 3n.$$

Pokaži, da velja $\frac{1}{2}n^2 - 3n = \Theta(n^2)$.

O-zapis

je zelo podoben Θ-zapisu, le da namesto asimptotično tesne meje, pomeni **asimptotično zgornjo mejo**. Za dano funkcijo $g(n)$ pomeni zapis $O(g(n))$ *množico funkcij* definirano takole:

$$O(g(n)) = \{f(n) : \exists c, n_0, \text{ da je } 0 \leq f(n) \leq cg(n), \ \forall n \geq n_0\},$$

pri čemer sta c in n_0 pozitivni konstanti. Kot vidimo, za razliko od Θ-zapisa, ki funkcijo omeji od zgoraj in spodaj, jo O-zapis omeji samo od zgoraj. Večkrat bomo uporabljali ta zapis, saj nas bo zanimala predvsem zgornja meja (asimptotične) zahtevnosti algoritma. Razlog je v tem, da kadar obravnavamo zahtevnost algoritma, nas predvsem zanima zahtevnost v najslabšem primeru (*angl. worst-case*). Če natančno zahtevnost v najslabšem primeru izrazimo z zgornjo asimptotično mejo, potem ta meja velja tudi za vse ostale (ne najslabše) primere vhodov. Slednje ne drži za Θ-zapis, saj je spodnja asimptotična meja pri najslabšem vhodu lahko različna od spodnje asimptotične meje pri ostalih vhodih.

Ko v praksi rečemo, da je zahtevnost algoritma npr. $O(n^2)$, v resnici mislimo, da je to zgornja asimptotična meja zahtevnosti v najslabšem primeru. Se pravi je za kake druge vhode, zgornja asimptotična meja lahko tudi nižja.

Vzemimo primer urejanja z vstavljanjem. Za ta algoritem rečemo, da je časovna zahtevnost $O(n^2)$, pri čemer bi morali reči, da je časovna zahtevnost algoritma v najslabšem primeru $O(n^2)$. V primeru, da na vhod dobimo že urejen seznam, je zgornja asimptotična meja $O(n)$.

Ω-zapis

je analogen O-zapisu, saj pomeni **asimptotično spodnjo mejo**. Množica $\Omega(g(n))$ je definirana takole:

$$\Omega(g(n)) = \{f(n) : \exists c, n_0, \text{ da je } 0 \leq cg(n) \leq f(n), \ \forall n \geq n_0\},$$

pri čemer sta c in n_0 pozitivni konstanti.

Izrek Za poljubni dve funkciji $f(n)$ in $g(n)$ je

$$f(n) = \Theta(g(n)) \iff f(n) = \Omega(g(n)) \wedge f(n) = O(g(n))$$

Naloga Lastnosti asimptotičnih zapisov (del naloge 3-4 iz [CLRS09] na strani 62)

Naj bosta $f(n)$ in $g(n)$ pozitivni funkciji. Dokažimo ali ovrzimo (utemeljeno) naslednje trditve o asimptotičnih zapisih.

1. $f(n) = O(g(n)) \implies g(n) = O(f(n))$
2. $f(n) + g(n) = \Omega(\min(f(n), g(n)))$

   $f(n) + g(n) = O(\min(f(n), g(n)))$

   $f(n) + g(n) = \Theta(\min(f(n), g(n)))$
3. $f(n) = O(g(n)) \implies 2^{f(n)} = O(2^{g(n)})$
4. $f(n) = O(f(n)^2)$
5. $f(n) = O(g(n)) \implies g(n) = \Omega(f(n))$
6. $f(n) = \Theta(f(\frac{n}{2}))$

o- in ω-zapis

Asimptotična zgornja meja definirana z O-zapisom je lahko tesna ali ne. Npr. $2n^2 = O(n^2)$ je tesna meja, medtem ko $2n = O(n^2)$ ni tesna. Za zgornjo mejo, ki ni tesna, uporabimo o- (mali o, angl. *little oh*) ali ω-zapis (mala omega), ki ju definiramo kot:

$$o(g(n)) = \{f(n) : \forall c > 0 \ \exists n_0, \text{ da je } 0 \leq f(n) < cg(n), \ \forall n \geq n_0\}$$

$$\omega(g(n)) = \{f(n) : \forall c > 0\ \exists n_0,\ \text{da je } 0 \leq cg(n) < f(n),\ \forall n \geq n_0\}$$

Naloga Dokaži, da za asimptotično pozitivni funkciji $f(n)$ in $g(n)$ velja:

$$f(n) \in o(g(n)) \iff \lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$$

Podobno velja tudi za ω-zapis:

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty$$

Zgornja relacija nam v povezavi z l'Hôpitalovim pravilom za računanje limit pogosto omogoča pokazati, katere družine funkcij so asimptotično manjše od drugih.

Izrek l'Hôpitalovo pravilo:

$$\lim_{x\to c} \frac{f(x)}{g(x)} = \lim_{x\to c} \frac{f'(x)}{g'(x)}$$

Če za funkciji f in g, odvedljivi na $I \setminus c$ velja

$$\lim_{x\to c} f(x) = \lim_{x\to c} g(x) = 0 \text{ or } \pm\infty$$

$$\lim_{x\to c} \frac{f'(x)}{g'(x)} \text{ obstaja}$$

$$g'(x) \neq 0\ \forall x \in I \setminus c$$

Naloga Pokaži, da so polinomske funkcije asimptotično manjše od eksponentnih.

$$n^b = o(a^n),\ \text{za } a > 1, b \geq 0$$

Naloga Pokaži, da je logaritem asimptotično manjši od poljubne polinomske funkcije.

Naloga Doma za vajo še: $\log^b(n) = o(n^a)$.
(Podobno kot pri prejšnji nalogi lahko začnemo s celoštevilskimi potencami).

Naloga Uredi funkcije

$$2^{\lg n}, \ln \ln n, \left(\frac{3}{2}\right)^n, \ln n, 4^{\lg n}, 1$$

v zaporedje $f_1, f_2, \ldots, f_6$, tako da bo veljalo

$$f_1 = \Omega(f_2), f_2 = \Omega(f_3), \ldots, f_5 = \Omega(f_6)$$

(del naloge 3-3 iz [CLRS09] stran 61.)

Naloga [1. domača naloga 2013/14]

Imamo naslednje ocene velikosti funkcij:

$$O(1), O(2n), O(n \log n), O(e^n), O(n^3), O(n^{1/3}) \text{ in } O(\log \log n)$$

za naslednje funkcije pokažite katerim vse ocenam velikostnih funkcij pripadajo:

$$n^{11}, 1/n + 23, n2^{\sqrt{n}} + n \text{ in } n^{2,45}/\log n$$

To pomeni, da morate pokazati katerim vse razredom funkcij pripada vsaka od navedenih funkcij.

Namig: pomagajte si s pojmom tranzitivnosti, da ne bo potrebno preveč dokazovati.

Naloga [1. kolokvij 2012/2013]

Imamo funkciji $f_1(n) = 12n + 11 \lg n$ in $f_2(n) = 11/n + 12 \lg n$. Katerim izmed naslednjih družin pripada vsaka od funkcij:

$$O(n), \Omega(n), \Theta(n), O(\log n), \Omega(\log n) \text{ in } \Theta(\log n).$$

Utemeljite svoj odgovor.

Naloga [2. pisni izpit 2012, 1. pisni izpit 2013]

Imamo naslednje razrede funkcij:

$$O(n \log n), O(n \log \log n) \text{ in } O(n/\log n).$$

Poleg tega imamo še funkcije $7n - 1$, $\sqrt{n} + n$ in $n^\pi / \sqrt{n}$. Katerim razredom pripada posamezna funkcija. Utemeljite odgovor.

Naloga [2. pisni izpit 2013]

Imamo naslednje funkcije ($\epsilon < 1$)

$$n, \frac{n}{\log n}, n^{2-\epsilon}, n^{1-\epsilon} \text{ in } n \log n$$

ter pripadajoče družine funkcij

$$O(n), O\left(\frac{n}{\log n}\right), O(n^{2-\epsilon}), O(n^{1-\epsilon}) \text{ in } O(n \log n).$$

Za vsako družino funkcij zapišite, katere funkcije ji pripadajo. Utemeljite odgovor.

Namig: Pomagajte si s tranzitivnostjo.

3.2 Dokazovanje pravilnosti algoritmov z indukcijo in analiza

Na predavanjih ste vzeli algoritem za urejanje z vstavljanjem in pokazali njegovo pravilnost. Tu zapišimo še algoritem za *urejanje z izbiranjem*:

Algoritem 12: Urejanje z izbiranjem

```
1 Vhod: polje celih števil A velikosti n
2 Izhod: naraščajoče urejeno polje A — rešitev problema!
3 for i ← 1 to n − 1 do
4     minIndex ← i
5     for j ← i + 1 to n do
6         if A[j] < A[minIndex] then
7             minIndex ← j
8     Swap(A[i], A[minIndex])
```

Pozor Indeksi polj se bodo začeli z 1 (isto kot v [CLRS09]) in ne z 0, kot

je navada v večini programskih jezikov.

Naloga Demonstriraj, kako poteka urejanje.

3.2.1 Dokaz pravilnosti algoritma z indukcijo

Dokaz pravilnosti algoritma z indukcijo najprej zahteva *indukcijsko predpostavko* ali *hipotezo*. Ker naš algoritem izračuna rešitev problema z zanko, bomo to poimenovali *zančna invarianta*: To je lastnost podatkovne strukture, ki mora veljati vsako iteracijo po zanki in bo služila kot temelj za dokazovanje pravilnosti.

V našem primeru bomo definirali naslednjo zančno invarianto, sestavljeno iz dveh delov:

1. Na začetku vsake iteracije i zanke **for** (3-8) je del polja $A[1..i-1]$ urejen v naraščajočem vrstnem redu in
2. elementi v preostanku polja $A[i..n]$ so večji ali enaki elementom na levi strani polja.

Ideja je naslednja: Če omenjena zančna invarianta velja za vse iteracije, potem na koncu dobimo celotno polje urejeno.

Da dokažemo pravilnost algoritma z indukcijo, moramo dokazati obnašanje indukcijske predpostavke oz. zančne invariante:

- Osnova (*angl. initialization*) — invarianta velja pred vstopom v zanko.
- Induktivni korak (*angl. maintenance*) — ob prehodu iz ene iteracije na drugo mora invarianta še vedno veljati.
- Zaključek (*angl. termination*) — najpomembnejši del — Ali nas je ob izteku zanke invarianta pripeljala do rešitve problema? Ali se zanka izteče pravi čas? Ali se sploh kdaj izteče?

Na našem konkretnem primeru gre stvar takole:

Osnova Ob vstopu v zanko je $i = 1$. V delu polja $A[1..0]$ ni elementov. Prazno polje je vedno urejeno (dokaz za prvi del invariante). Vsi elementi v preostanku polja so večji od praznega polja (dokaz za drugi del invariante).

Induktivni korak Telo druge zanke **for** (5-7) poišče najmanjši element v $A[i..n]$ z indeksom $minIndex$. Ker v (8) zamenja $A[i]$ z $A[minIndex]$,

se ob koncu iteracije na i-tem mestu sedaj nahaja najmanjši element iz $A[i..n]$.

Ob prehodu na naslednjo iteracijo $i \leftarrow i+1$ imamo po novem invarianto, ki pokriva prvotno območje prejšnje iteracije $A[1..i-2]$, in nazadnje dodan element $A[i-1]$. Ker smo dodali največji ali kvečjemu enak element prejšnjim, smo ohranili urejenost v naraščajočem vrstnem redu (dokaz za prvi del invariante). V preostanku polja $A[i..n]$ ni manjših elementov od levega dela, saj smo v območju $A[i-1..n]$ izbrali najmanjšega in ga dali na mesto $A[i-1]$ (dokaz drugega dela invariante).

Zaključek Zanka **for** (3) se zaključi natanko tedaj, ko $i = n$. Glede na prvi del invariante imamo ob izstopu iz zanke urejeno polje v območju $A[1..n-1]$, zaradi drugega dela invariante, pa je preostali element $A[n]$ večji ali enak tistim na levi strani. Celotno polje $A[1..n]$ je urejeno.

3.2.2 Časovna in prostorska analiza

Pri časovni analizi algoritma bomo merili število primerjav elementov. Začnimo z notranjo zanko (3-8). Prvič $j \leftarrow 2..n$ bo izvedla $n-1$ primerjav elementov, potem $n-2$, $n-3$. Notranja zanka bo poklicana glede na zunanjo zanko (3-11). Ta se pokliče $n-1$ krat. Dobimo torej:

$$n-1+n-2+n-3+...+1 = \sum_{i=1}^{n-1} n-i = \frac{n(n-1)}{2}$$

Opazimo, da je število primerjav enako ne glede na vhodne podatke — najslabši možni primer bo enako hitro rešen (urejen) kot najboljši. Število primerjav je ves čas $\frac{(n-1)^2}{2} = \Theta(n^2)$.

Pri prostorski zahtevnosti bomo merili število zasedenih celic, ki jih zavzemajo elementi med izvajanjem programa. Poleg n celic za vhodne podatke potrebujemo še 3 celice za spremenljivke i, j in $minIndex$. Prostorska zahtevnost je torej $n+3$ ne glede na vhodne podatke. Asimptotično je prostor omejen z $\Theta(n)$.

Naloga Kaj počne spodnji algoritem? Z indukcijo dokaži pravilnost delovanja!

Algoritem 13: (Urejanje z vstavljanjem)

```
1 for j ← 2 to A.length do
2     key ← A[j]
3     i ← j − 1
4     while i > 0 and A[i] > key do
5         A[i + 1] ← A[i]
6         i ← i − 1
7     A[i + 1] ← key
```

Poglavje 4

Vrsta s prednostjo

Vrsta s prednostjo se v praksi uporablja v naslednjih aplikacijah:

- razvrščanje procesov v operacijskem sistemu,
- zagotavljanje kakovosti storitve na usmerjevalnikih (QoS),
- iskanje najkrajše poti (Dijkstra),
- gradnja najmanjšega vpetega drevesa (Kruskal).

Pozor Vrste s prednostjo **ne smemo mešati s slovarjem in ne omogoča učinkovitega iskanja poljubnega elementa po ključu** — `Find()`! V ta namen vodimo zraven ločen slovar elementov.

Imamo naslednje operacije. Povsod bomo privzeli, da imamo opravka z minimalno vrsto s prednostjo, če ni drugače napisano. V oklepaju so napisane funkcije v primeru maksimalne vrste s prednostjo.

- `Insert(`P`,` v`)` — v kopico P dodamo nov element z vrednostjo v.
- `FindMin(`P`)` — vrnemo najmanjši element v P. (največji, `FindMax`)
- `DeleteMin(`P`)` — iz P zbrišemo najmanjši element in ga vrnemo. (največjega, `DeleteMax`)

Poleg naštetih lahko vrsto s prednostjo razširimo tudi z naslednjimi operacijami:

- `DecreaseKey(`P`,` x`,` v`)` — v P zmanjšamo vrednost elementa x na v. (povečamo, `IncreaseKey`)
- `Delete(`P`,` x`)` — iz P odstranimo element x (podamo referenco nanj).

- `Union(`P`, `Q`)` — unija dveh vrst s prednostjo P in Q. Rezultat je nova vrsta s prednostjo. **To bomo jemali drugič!**

Pozor Operacija `Union()` se včasih imenuje tudi `Merge()` ali `Meld()`. Operacija `DeleteMin()` se včasih imenuje tudi `ExtractMin()`.

4.1 Dvojiška kopica

Vrsto s prednostjo se lahko izvede na več načinov, npr. s povezanim seznamom ali z drevesom. Za nas bo za začetek zanimiva **izvedba s kopico**.

Dvojiška kopica kot podatkovna struktura je bila posredno definirana v začetku 60. let z namenom urejanja števil (J. W. Williams: Heapsort [Wil64] in Robert W. Floyd: Treesort [Flo64]). Kopica je dvojiško, uravnoteženo, levo poravnano drevo. Vsebinska invarianta se imenuje **kopičasta ureditev** (*heap order*) in loči dve izključujoči vrsti kopic: minimalna (*MinHeap*) ali maksimalna (*MaxHeap*) kopica. Pri prvi morajo biti **vsi nasledniki poljubnega vozlišča manjši ali enaki vrednosti v vozlišču**, pri drugi pa obratno — vsi nasledniki morajo biti večji od vrednosti v vozlišču.

Naloga Koliko je koren pri minimalni in kakšen pri maksimalni kopici?

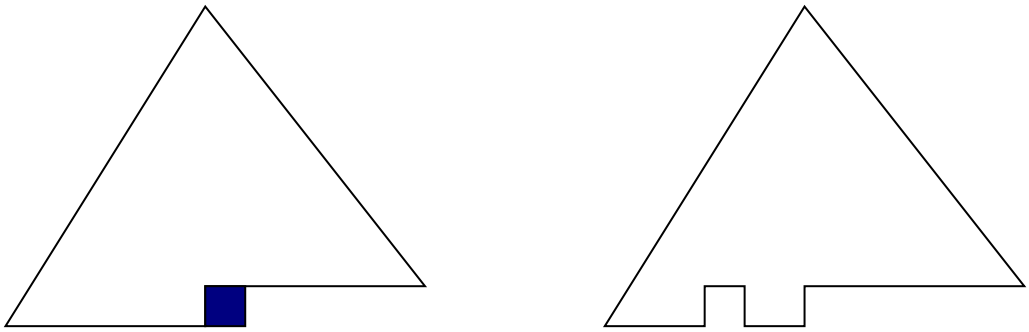
4.1.1 Operacije

Dvojiška kopica iz leta 1964 je polno in **levo poravnano** dvojiško drevo. Najprej poglejmo, kako **zmanjšati ključ elementa** (`DecreaseKey()`) v minimalni kopici. Kopičasto ureditev lahko kršimo, če element postane manjši od svojega starša. Element je potrebno dvigovati (*SiftUp*), dokler ni kopičasta ureditev spet veljavna. Druge možnosti kršenja kopice ni.

Pozor Funkcijo `DecreaseKey()` bomo velikokrat uporabljali tudi z obstoječo vrednostjo elementa in uporabili le njeno nalogo dvigovanja elementa na ustrezno mesto.

Pozor V originalnem članku je za dvigovanje in spuščanje elementa uporabljen izraz *to sift*, ki pomeni presejati npr. moko, sladkor.

Vstavljanje poteka tako, da nov element dodamo desno od zadnjega lista (glej sliko 4.1.(a)) oz. če je ta skrajni, uvedemo nov nivo in ga vstavimo čisto



Slika 4.1: (a) Oblika splošne kopice. Modro obarvano je začetno mesto novega elementa. (b) Oblika kopice ob brisanju korenskega elementa. Škrbina se je pomaknila od korena k dnu. Škrbino je potrebno zapolniti.

levo. Vsebinsko invarianto smo sedaj verjetno kršili. To popravimo tako, da element dvigamo, dokler kopica ni popravljena. V psevdokodi bomo uporabljali **implicitni zapis kopice v polju od indeksa** 1 **dalje**. Vstavljanje poteka v dveh korakih. Najprej na zadnje mesto **vstavimo element z vrednostjo** ∞, nato mu **zmanjšamo vrednost** na pravo z operacijo `DecreaseKey()`.

Vračanje najmanjšega elementa je izvedeno z vrnitvijo korena kopice oz. prvega elementa v polju, če uporabljamo implicitni zapis.

Algoritem 14 vsebuje psevdokodo za zgornje tri operacije.

Brisanje najmanjšega elementa poteka tako, da zbrišemo koren. Sedaj imamo dve možnosti (ste imeli na predavanjih):

- Škrbino korena nadomestimo z zadnjim listom in ga pomikamo navzdol, dokler ni kopica spet pravilna (po Williamsu [Wil64]).
- Škrbino korena pomikamo navzdol in jo nadomeščamo s trenutno najmanjšim otrokom. Ko pridemo do dna, nam lahko nastane škrbina (glej sliko 4.1.(b)). Škrbino zapolnemo z zadnjim listom in ga pomikamo gor, dokler ni kopica spet pravilna (po Floydu [Flo64]).

Obe varianti sta predstavljeni v algoritmu 15.

Algoritem 14: Iskanje najmanjšega elementa, vstavljanje in zmanjševanje ključa elementa v minimalni kopici.

```
1  DecreaseKey(P, i, v):
2  begin
3      P[i] ← v ;
4      while i > 1 ∧ P[parent(i)] > P[i] do
5          swap(P[parent(i)], P[i]) ;
6          i ← parent(i) ;
7  Insert(P, v):
8  begin
9      P.length++ ;
10     P[P.length] ← ∞ ;
11     DecreaseKey(P, P.length, v) ;
12 FindMin(P):
13 begin
14     if P.length > 0 then
15         return P[1] ;
16     return ∅ ;
```

Brisanje poljubnega elementa x poteka tako, da pokličemo `DecreaseKey(`P`,` x`,` $-\infty$`)` in nato `DeleteMin(`P`,` x`)`.

Izvedb unij dveh kopic je več in si jih bomo ogledali drugič.

4.1.2 Analiza

Prostorska

Podatkovna struktura v implicitni obliki potrebuje kvečjemu $n + 1$ veliko polje.

Časovna

Zmanjšanje vrednosti ključa pomika element navzgor in potrebuje kvečjemu $\lceil \lg n \rceil$ primerjav (eno primerjavo na nivo).

Vstavljanje v najslabšem primeru premakne element od zadnjega lista v koren in imamo natanko $\lceil \lg n \rceil$ primerjav.

Iskanje poteka v konstantnem času $O(1)$ in ne potrebujemo primerjav.

Poglejmo razliko obeh variant **brisanja najmanjšega elementa**. Pri prvi smo koren nadomestili z zadnjim listom. Korenski element sproti menjamo z manjšimi otroki. Za to potrebujemo 2 primerjavi na vsakem nivoju, kar znese $2\lceil \lg n \rceil$ v najslabšem primeru oz. $2(\lceil \lg n \rceil - k)$, če k plasti pred koncem, če se element umesti prej. Statistično gledamo je k relativno majhen (spomnimo: v polnem drevesu se kar polovica elementov v drevesu vedno nahaja v listih, druga polovica pa v notranjih vozliščih). Pri drugi varianti pomikamo škrbino korena do dna. Tu je potrebna **le ena primerjava na nivo**, kar znese natanko $\lceil \lg n \rceil$ primerjav. Nato škrbino nadomestimo z zadnjim listom in ga pomaknemo na ustrezno mesto, torej še dodatnih k' primerjav. Druga varianta tako potrebuje natanko $\lceil \lg n \rceil + k'$ primerjav in je običajno vedno hitrejša od prve.

Časovna zahtevnost brisanja poljubnega ključa je sestavljena iz vsote zahtevnosti za zmanjšanje vrednosti ključa in brisanje najmanjšega elementa.

Algoritem 15: Obe varianti brisanja najmanjšega elementa v minimalni kopici.

```
1  DeleteMin1(P):
2  begin
3      if P.length = 0 then
4          return ∅ ;
5      delElt ← P[1] ;
6      P[1] ← P[P.length] ;
7      P.length − − ;
8      i ← 1 ;
9      while ∃minChild(i) c : P[c] < P[i] do
10         swap(P[c], P[i]) ;
11         i ← c ;
12     return delElt ;
13 DeleteMin2(P):
14 begin
15     if P.length = 0 then
16         return ∅ ;
17     delElt ← P[1] ;
18     i ← 1 ;
19     while minChild(i) c ≠ ∅ do
20         swap(P[i], P[c]) ;
21         i ← c ;
22     P[i] ← P[P.length] ;
23     P.length − − ;
24     DecreaseKey(P, i, P[i]) ;
25     return delElt ;
```

4.2 Binomska kopica

Leta 1978 je Jean Vuillemin predstavil binomsko kopico [Vui78]. Najslabše čase razen zlivanja ima tako dobre kot dvojiška kopica. Zlivanje se zgodi v najslabšem primeru v času $O(\lg n)$ (kjer je n število elementov v večji kopici).

Kako je struktura definirana? Namesto ene kopice je uporabil več t. i. **binomskih dreves**. Se pravi, **binomsko kopico sestavlja več binomskih dreves**.

4.2.1 Binomsko drevo

Je **delno uravnoteženo drevo**, kjer velja **minimalna kopičasta ureditev**. Binomsko drevo B_k je visoko k plasti in ima natanko 2^k elementov, vendar **ni dvojiško drevo!** Na i-ti plasti se namreč nahaja vedno $\binom{k}{i}$ elementov (se pravi koren je le eden in na zadnji plasti je le en element, vmes pa najprej raste število elementov do polovice števila plasti, nato pa nazaj pada). Zaradi binoma v formuli izvira tudi ime podatkovne strukture.

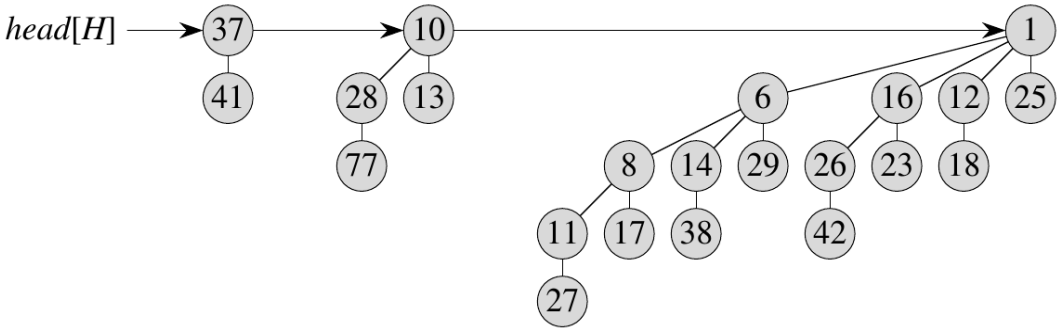
Zanimiva je gradnja binomskega drevesa. B_1 zgradimo z zlivanjem dveh B_0 dreves. To pa tako, da drevo z manjšim elementom postane novi koren. Na enak način lahko zgradimo binomsko drevo poljubne stopnje. Vedno zlivamo binomski drevesi **istih stopenj v binomsko drevo ene stopnje više**. Drevo z večjim korenom pripnemo h korenu drugega drevesa (z manjšim korenom). Opisan postopek se imenuje **zlivanje binomskih dreves**.

4.2.2 Zgradba binomske kopice

Združuje več binomskih dreves. Posamezno drevo res lahko vsebuje le 2^k elementov, vendar njihova kombinacija lahko pokrije vsako možno število elementov. Vodenje evidence, katera binomska drevesa vsebujemo, poteka s pomočjo **bitnega vektorja** (1, če je drevo vsebovano, 0, če ni).

Pozor Če imamo bitni vektor 10110, vsebujemo (beremo z desne) binomska drevesa B_1, B_2 ter B_4. Podatkovna struktura hrani torej $2^1 + 2^2 + 2^4 = 22$ elementov. Slika 4.2 prikazuje binomsko kopico s tremi binomskimi drevesi.

Binomska kopica prvotno ni izvedena kot implicitna podatkovna struktura, ampak uporabljamo reference.



Slika 4.2: Binomska kopica s tremi binomskimi drevesi.

4.2.3 Operacije in analiza

Pogledali si bomo delovanje in analizo istih operacij kot pri dvojiški kopici:

- `Insert(`P`, `v`)` — v kopico P dodamo nov element z vrednostjo v.
- `FindMin(`P`)` — vrnemo najmanjši element v P.
- `DeleteMin(`P`)` — iz P zbrišemo najmanjši element in ga vrnemo.
- `DecreaseKey(`P`, `x`, `v`)` — v P zmanjšamo vrednost elementa x na v.
- `Delete(`P`, `x`)` — iz P odstranimo element x (podamo referenco nanj).
- `Union(`P`, `Q`)` — unija (zlivanje) P in Q.

Zlivanje

Najprej bomo pogledali zlivanje oz. unijo dveh binomskih kopic. Unija binomske kopice poteka po istoležnih binomskih drevesih (tj. z istimi stopnjami). Preverjanje, katera drevesa bo potrebno združiti si najlažje predstavljamo s seštevanjem obeh bitnih vektorjev kopic. Če istoležnega drevesa v eni od kopic ni, se drevo enostavno doda k ciljni kopici. Če pa drevo z isto stopnjo že obstaja, se izvede **zlivanje dveh binomskih dreves**, opisanem v drugem odstavku poglavja 4.2.1. Posledično dobimo novo binomsko drevo višje stopnje, kar je ekvivalent **prenosu** enice naprej pri seštevanju bitnih vektorjev.

Časovna zahtevnost zlivanja dveh binomskih dreves je $O(1)$ (potrebujemo le eno primerjavo, da izvemo, kateri koren je manjši in eno prevezavo korena). Koliko pa imamo lahko največ zlivanj? V najslabšem primeru zlivamo dve binomski kopici, ki imata v bitnih vektorjih same enice. Ker bitni vektor vedno sestoji iz $\lceil \lg n \rceil$ bitov, imamo torej $O(\lg n)$ največ zlivanj, kar je tudi celotna časovna zahtevnost zlivanja.

Vstavljanje, iskanje

Operacijo vstavljanja izvedemo kot zlivanje s kopico z enim samim elementom. Časovna zahtevnost je torej enaka zlivanju ($O(\lg n)$). Iskanje poteka po sprehodu po vseh korenih binomskih dreves ($O(\lg n)$), lahko pa si sproti zapomnimo najmanjši element do sedaj (bolje — $O(1)$).

Brisanje najmanjšega elementa

Brisanje najmanjšega elementa poteka v naslednjih korakih:

1. odstranimo koren binomskega drevesa z najmanjšim elementom,
2. njegove otroke zlijemo s preostankom binomske kopice.

Analiza: Koliko ima koren binomskega drevesa B_k naslednikov? $\binom{k}{1} = k$, kar je $O(\lg n)$, se pravi, da bo potrebno imeti toliko zlivanj. Koliko bodo pa zahtevna ta zlivanja? Poglejmo, kakšna so sploh poddrevesa: natanko od stopnje 0 do $k - 1$. Česa se bojimo? Da bi vsako poddrevo potrebovalo veliko zlivanj zaradi prenosa. Ampak to pa ni možno — v najslabšem primeru že prvo poddrevo sproži verigo $\lg n$ zlivanj, nato pa imajo preostala poddrevesa same ničle v ciljnem bitnem vektorju, tako da bo minimalno število zlivanj. Časovna zahtevnost celotne operacije tako ostane $O(\lg n)$.

Zmanjševanje ključa, brisanje poljubnega elementa

Zmanjševanje ključa poteka identično kot v dvojiški kopici — izbran element v binomskem drevesu dvigamo, dokler ne priplava do ustreznega mesta. Če zamenja celo koren, po potrebi obnovimo tudi najmanjši element celotne kopice.

Analiza zmanjševanja ključa: Primerjav bo višina najvišjega binomskega drevesa. To je pa kvečjemu $\lg n$.

Brisanje poljubnega elementa poteka identično kot v dvojiški kopici — izbranemu elementu zmanjšamo ključ na $-\infty$, nato izvedemo brisanje najmanjšega elementa v kopici.

Analiza: $\lg n$ za dviganje elementa, $O(\lg n)$ za brisanje najmanjšega elementa.

Naloga [1. naloga kolokvij 120605]

Peter Zmeda je slišal, da obstajajo različne vrste kopic kot izvedbe vrst s prednostjo. Tako je slišal, da obstajata binarna (dvojiška) kopica in binomska kopica.

1. Nad binarno kopico po vrsti naredite naslednje operacije in sproti izrisujte podatkovno struktur (I pomeni vstavi, Min pomeni poišči minimum in DMin zbriši najmanjši element): `I(17), I(3), I(5), I(1), Min(), I(10), DMin(), I(4), DMin(), DMin()`
2. Iste operacije izvedite še nad binomsko kopico ter ponovno sproti izrisujte izgled strukture.

4.3 Lena binomska kopica

Operacija združevanja binomskih dreves iste stopnje je draga — $O(\lg n)$. Ta operacija se v knjigi imenuje **konsolidacija**, v izvornem članku o Fibonaccijevi kopici pa **povezovalni korak** (*linking step*). Ali jo moramo resnično pri vsaki operaciji izvesti? Lena binomska kopica ima večino operacij **lenih**:

4.3.1 Operacije

Fibonaccijeva kopica uporablja idejo podobno Binomski: vrsta s prednostjo je še vedno gozd binomskih dreves, le da so ti lahko malce pokvarjeni.

Unija Pripnemo drevesa nove Fibonaccijeve kopice k stari. Dreves z isto stopnjo ne zlivamo (**leno zlivanje**).

Vstavljanje Isto kot unija z drevesom z enim elementom, uporabimo leno zlivanje.

FindMin Deluje enako kot pri Binomski kopici. Sproti si zapomnimo najmanjši element in ga po potrebi vrnemo.

Delete Če je element koren drevesa, uporabimo `DeleteMin` nad dotičnim elementom. Sicer pa se za razliko od `DeleteMin` tu poslužujemo **lenega brisanja**. Element odstranimo iz drevesa, njegova poddrevesa pa **leno** pripnemo v podatkovno strukturo in jih ne zlivamo z drevesi enakih stopenj. Prvotno drevo je sedaj malce pokvarjeno.

DecreaseKey Elementu spremenimo ključ in ga skupaj z njegovimi poddrevesi **leno** premaknemo iz prvotnega drevesa. Prvotno drevo je malce pokvarjeno.

DeleteMin **Je edina operacija, ki ni lena!** Zbrišemo najmanjši element in njegove otroke zlijemo z drevesi enake stopnje (konsolidacija). Po operaciji imamo spet urejeno podatkovno strukturo (oblika Binomske kopice). Med koreni dreves poiščemo nov najmanjši element.

Dobra stran: Lene operacije potrebujejo le še $O(1)$ časa namesto $O(\lg n)$. Rahlo izboljšanje povprečnega časa dobimo s kombinacijo lenih zlivanj in `DeleteMin`. To pa zato, ker privarčujemo eno unijo, saj najprej zbrišemo element in nato izvedemo konsolidacijo.

Slaba stran: Stari grehi se nam kopičijo. Vsako drevo iste stopnje, ki ga takoj ne združimo ali pa ga kvarimo z `Delete`/`DecreaseKey`, nam bo ob klicu `DeleteMin` to zaračunalo! Amortizacijska analiza (v nadaljevanju) nam da točnejši odgovor, koliko. V najslabšem primeru pa bo konsolidacija zagotovo zahtevala $n - 1$ korakov.

Za domačo nalogo boste morali implementirati leno binomsko kopico in operacije z operacijami `Insert`, `FindMin` in `DeleteMin`.

Naloga Nad leno binomsko kopico po vrsti naredite naslednje operacije in sproti izrisujte podatkovno struktur (I pomeni vstavi, Min pomeni poišči minimum in DMin zbriši najmanjši element):
`I(17), I(3), I(5), I(1), Min(), I(10), DMin(), I(4), DMin(),`

```
        DMin()
```

4.3.2 Amortizacijska analiza [Tar85]

Ko so časi operacij odvisni od njihovega zaporedja in števila ponovitev, je najslabši primer za posamezno operacijo običajno preveč pesimističen in ne odraža realnega stanja. Za izračun povprečnega časa pa ne moremo dovolj poenostaviti problema. Z amortizacijsko analizo dokažemo, da je *povprečni čas izvajanja operacij majhen, kljub temu, da je posamezna operacija lahko draga.* Razlika z običajnim povprečnim časom je, da amortizacijska analiza garantira za *povprečni čas operacij v najslabšem primeru*, medtem ko je pri običajni analizi s povprečnim časom prisotna neka verjetnost (npr. operacija traja v povprečju $O(1)$ z veliko verjetnostjo, če se želimo poglobiti v konkreten primer, da izračunamo to verjetnost, pa se stvari močno zakomplicirajo, običajno moramo nekaj predpostaviti o podatkih, nad katerimi operiramo itd.).

Dvojiški števec Za začetek si bomo ogledali problem dvojiškega števca z le eno operacijo — povečavo števca za 1.

Imejmo k-bitni števec, ki se povečuje za 1 od 0 dalje. Uporabimo polje $A[0..k-1]$, tako da je predstavljeno število enako $x = \sum_{i=0}^{k-1} A[i]2^i$. Povečevanje števila, zapisanega z biti, prikazuje algoritem 16, ilustrira pa slika 8.

Algoritem 16: Povečevanje števca bitov za 1

```
1 Vhod: število, zapisano s poljem bitov A
2 Izhod: število, zapisano s poljem bitov A, povečano za 1
3 i ← 0
4 while i < A.length ∧ A[i] = 1 do
5 |   A[i] ← 0
6 |   i ← i + 1
7 if i < A.length then
8 |   A[i] ← 1
```

Na prvi pogled bi rekli, da je časovna zahtevnost algoritma za n povečav enaka $O(nk)$ oz. $O(k)$ na posamezno operacijo. Ali je v resnici tako slaba — ali lahko dokažemo, da je boljša?

Counter value	A[7]	A[6]	A[5]	A[4]	A[3]	A[2]	A[1]	A[0]	Total cost
0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	1	1
2	0	0	0	0	0	0	1	0	3
3	0	0	0	0	0	0	1	1	4
4	0	0	0	0	0	1	0	0	7
5	0	0	0	0	0	1	0	1	8
6	0	0	0	0	0	1	1	0	10
7	0	0	0	0	0	1	1	1	11
8	0	0	0	0	1	0	0	0	15
9	0	0	0	0	1	0	0	1	16
10	0	0	0	0	1	0	1	0	18
11	0	0	0	0	1	0	1	1	19
12	0	0	0	0	1	1	0	0	22
13	0	0	0	0	1	1	0	1	23
14	0	0	0	0	1	1	1	0	25
15	0	0	0	0	1	1	1	1	26
16	0	0	0	1	0	0	0	0	31

Opazimo, da enkrat obrnemo kar veliko število bitov, drugič le enega. Uporabimo eno izmed metod amortizacijske analize — *računovodsko metodo* (včasih imenovano tudi *metodo bankrija*), ki pravi, da za vsako operacijo vložimo nekaj kovancev. Nekaj jih porabimo za izvedbo, nekaj jih shranimo za kasneje. Najmanjše število potrebnih kovancev, da še lahko izvedemo vse operacije, imenujemo amortiziran povprečen čas na operacijo.

Koliko kovancev potrebujemo pri nas? Poskusimo in dokažimo, da bosta za eno povečanje 2 kovanca dovolj. Ideja je naslednja: Vsakemu bitu dodelimo ob povečavi 2 kovanca. Enega porabimo, ko bit nastavimo na 1 in drugega, ko ga damo nazaj na 0. V zanki while (4) dajemo bite le na 0, torej porabimo že akumulirane kovance nad biti. Bit nastavimo na 1 le na enem mestu (9), za kar zahtevamo 2 kovanca: enega, da bit nastavimo na 1 in drugega, ki ga shranimo za v bodoče, ko bomo dali bit nazaj na 0. Amortizacijski čas na operacijo je torej $2 = O(1)$.

4.4 Fibonaccijeva kopica

Tisto 4. temeljno vprašanje pri gradnji podatkovnih struktur — ali se da narediti stvar še hitrejšo — je vedno zanimivo. Leta 1987 sta Michael L. Fredman in Robert Endre Tarjan predstavila Fibonaccijevo kopico [FT87]. Z njo sta uspela izboljšati amortizirane čase binomske kopice kot tudi povprečne čase operacij.

Spomnimo, amortiziran čas neke operacije je vedno čas v nekem določenem scenariju (recimo nizu k vstavljanj, l iskanj, m brisanj). Je bolj pesimističen od povprečnega, saj imamo še vedno lahko nekaj sredstev vloženih na *banki*.

Fibonaccijeva kopica je lena binomska kopica z naslednjo zahtevo: Dreves z operacijama `Delete` in `DecreaseKey` ne moremo kar kvariti v nedogled. Vsako vozlišče lahko izgubi kvečjemu enega otroka. Pri brisanju drugega otroka pa je pravilo, da se po brisanju tudi starš odstrani iz drevesa. S tem lahko rekurzivno povzroči razpad celotnega drevesa (če so njegovi predhodniki že izgubili po enega otroka). To se imenuje **kaskadni rez** (angl. *cascading cut*). To pravilo nam omogoča, da omejimo največjo globino drevesa: od $\lfloor \lg n \rfloor$ do $\lfloor \log_\Phi n \rfloor$. Konstanta Fidias $\Phi = \frac{1+\sqrt{5}}{2} = 1{,}6180$ se med drugim uporablja za izračun poljubnega člena Fibonaccijevega zaporedja v zaprti obliki, od tu tudi ime podatkovne strukture.

4.4.1 Analiza

Največji dosežek so zelo dobri amortizirani časi. Pri analizi se osredotočimo na število binomskih dreves v strukturi, saj na to število pade glavnina dela ob konsolidaciji (tej meri rečemo tudi *potencial*). Z zahtevo, da nobenemu vozlišču ne odstranimo več kot 2 otrok, zagotovimo, da ima drevo stopnje k še vedno 2^k, če je celo in vsaj 2^{k-1} vozlišč, če je najbolj skvarjeno (vsa vozlišča označena).

Za operacije `Union, Insert, FindMin, DecreaseKey` dobimo $O(1)$ amortizirano, saj kvečjemu toliko novih dreves ustvarijo, za vsako drevo pa založijo po en kovanec za bodočo konsolidacijo. Za `Delete, DeleteMin` pa $O(\lg n)$ amortizirano, saj ob vsakem brisanju vseh svojih k otrok priklopita na korenski povezan seznam in založita po en kovanec na otroka za bodočo konsolidacijo.

4.4.2 Naloge

Naloga Nad fibonaccijevo kopico po vrsti naredite naslednje operacije in sproti izrisujte podatkovno struktur (I pomeni vstavi, Min pomeni poišči minimum in DMin zbriši najmanjši element):
`I(17), I(3), I(5), I(1), Min(), I(10), DMin(), I(4), DMin(), DMin()`

Naloga Nad Fibonaccijevo kopico na sliki **??**a izvedite naslednje zaporedje operacij:

```
 decreaseKey(H, h, 14)
decreaseKey(H, k, 18)
decreaseKey(H, e, 1)
decreaseKey(H, j, 4)
decreaseKey(H, c, 3)
deleteMin(H)
```

Naloga Peter Zmeda želi izvesti vrsto s prednostjo. Katero izvedbo mu priporočate?

Naloga Recimo, da imamo v vrsti s prednostjo 2012 elementov. Odgovorite na naslednja vprašanja za različne implementacije vrste s prednostjo in vsakega od odgovorov utemeljite:

1. Kako visoka je dvojiška kopica?
2. Kako visoko je najvišje drevo binomske kopice?
3. Kako visoko je največ najvišje drevo Fibonaccijeve kopice?
4. Ali je možno, da je najvišje drevo Fibonaccijeve kopice visoko 1 povezavo?

Poglavje 5

Razširjene podatkovne strukture

5.1 Rang, izbira

Imamo niz n števil. Želimo učinkovito implementirati naslednji operaciji:

- `Rank(`S`,` x`)` $\to i$ — kateri element v S po vrsti je x.
- `Select(`S`,` i`)` $\to x$ — v S poiščemo i-ti element po vrsti.

Pozor Za operacijo `Rank` bomo privzeli, da že imamo referenco na element in ga ni potrebno iskati v strukturi.

5.1.1 Rešitev s poljem

Očitna rešitev je, da števila hranimo v urejenem polju. Operacija `Rank` vrne indeks elementa v polju (element se mora zavedati, kje v polju je), `Select` pa vrednost elementa na i-tem mestu v polju. Obe zahtevata konstantni čas poizvedbe $O(1)$.

Ponovimo ostale operacije nad poljem: `Find` išče

- `Find(`S`,` v`)` išče element s podano vrednostjo z razpolavljanjem. Potrebujemo $O(\lg n)$ primerjav.

- `Insert(`S`,` v`)` in `Delete(`S`,` x`)` pa zahtevata zamike ali premike vseh elementov v polju, kar je počasi — $O(n)$.

Pozor V resnici se da vstavljati v $O(\lg^2 n)$ časa in $O\left(\frac{\lg^2 n}{B}\right)$ prenosov blokov amortizirano [BDFC00] ali $O(1)$ časa [BCD+99]. Problem se imenuje tudi *"ordered file maintenance"*.

5.1.2 Rešitev z drevesom

Elemente shranjujemo v dvojiško iskalno drevo (glej poglavje 1.2). Dvojiško drevo razširimo (*augmented tree*) tako, da k vsakemu elementu pripišemo še število njegovih naslednikov +1 (vključno z njim samim). Glej sliko 5.1a. Število vseh naslednikov bomo poimenovali c (kot *count*).

Rang, Izbira

Rang poišče element in se sprehodi do korena ter iz vrednosti c na poti izračuna rang. To zahteva $\Theta(h)$ primerjav.

Izbira se glede na vrednosti števca c odloča, ali se spusti v levo ali desno poddrevo vozlišča. Časovna zahtevnost je $\Theta(h)$ primerjav.

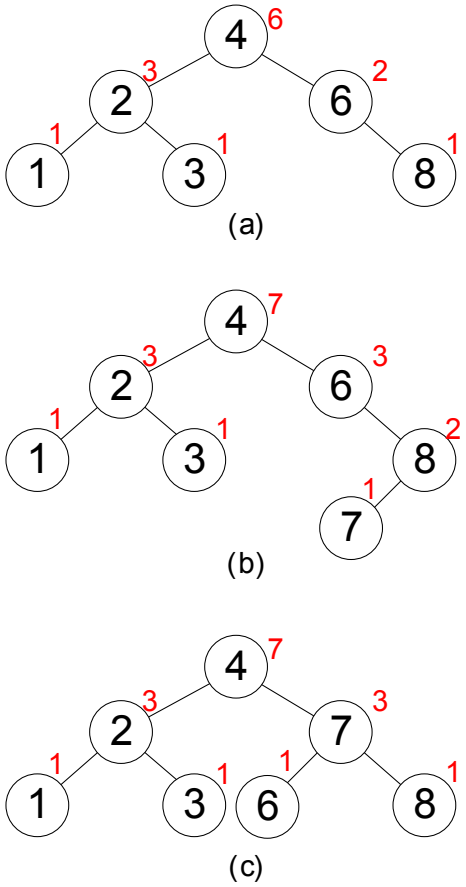
Pozor Psevdokodo za obe operaciji ste že napisali na predavanjih in je tu ne bomo.

Vstavljanje

Ponovimo vstavljanje v dvojiško drevo. Začnemo pri korenu, sledimo ustreznim naslednikom (če je novi element manjši od vozlišča, zavijemo v levo poddrevo, sicer v desno). Ko prispemo do praznega mesta, ga vstavimo.

Operacija nad drevesom, ki vsebuje števce naslednikov, je prikazana na sliki 5.1b. Vsem obiskanim vozliščem po poti povečamo c za 1.

Časovna zahtevnost vstavljanja ostane nespremenjena — $\Theta(h)$.



Slika 5.1: (a) Dvojiško iskalno drevo s števili naslednikov. (b) Vstavljanje števila 7. (c) Dvojna rotacija Desno-Levo za uravnoteženje drevesa (AVL).

Brisanje

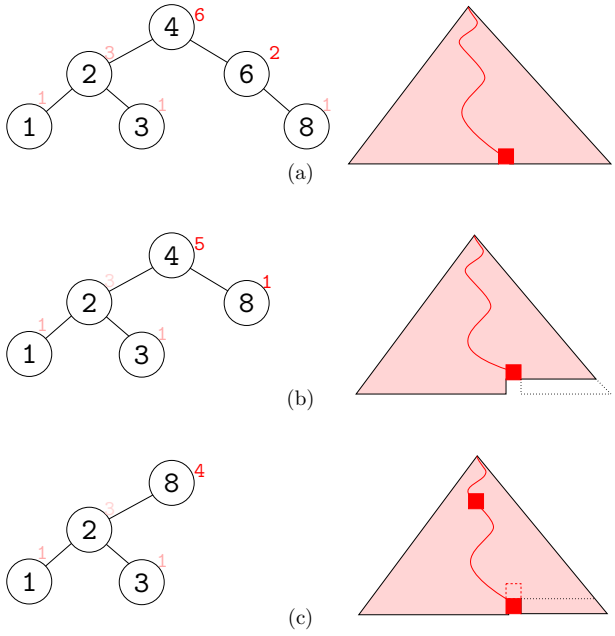
Spomnimo, pri brisanju elementa lahko nastanejo tri situacije (glej poglavje 1.2.1). Pri prvi je zbrisan element list in nima naslednikov — v tem primeru ga enostavno odstranimo. Pri drugi situaciji ima brisan element eno od poddreves prazno — škrbino zamenjamo s poddrevesom, ki poddrevo ima. Pri tretji situaciji ima brisan element obe poddrevesi — element zamenjamo z najmanjšim v desnem poddrevesu ali z največjim v levem.

Ideja pri brisanju je, da zmanjšamo števec za 1 od korena do brisanega elementa. Pri prvi in drugi situaciji deluje algoritem natanko tako.

Tretja situacija je malce bolj zapletena: najprej pomanjšamo števce po poti

od korena do brisanega elementa. Nato pomanjšamo števce od brisanega elementa do najmanjšega v desnem poddrevesu (oz. največjega v levem), saj ga pravzaprav zbrišemo iz desnega poddrevesa (oz. levega). Zamenjan element prevzame prejšnjo vrednost števca zbrisanega elementa -1.

Skica brisanje v vseh treh situacijah je na sliki 5.2.



Slika 5.2: (a) Brisanje elementa brez otrok. (b) Brisanje elementa z enim otrokom. (c) Brisanje elementa z dvema otrokoma.

5.1.3 AVL

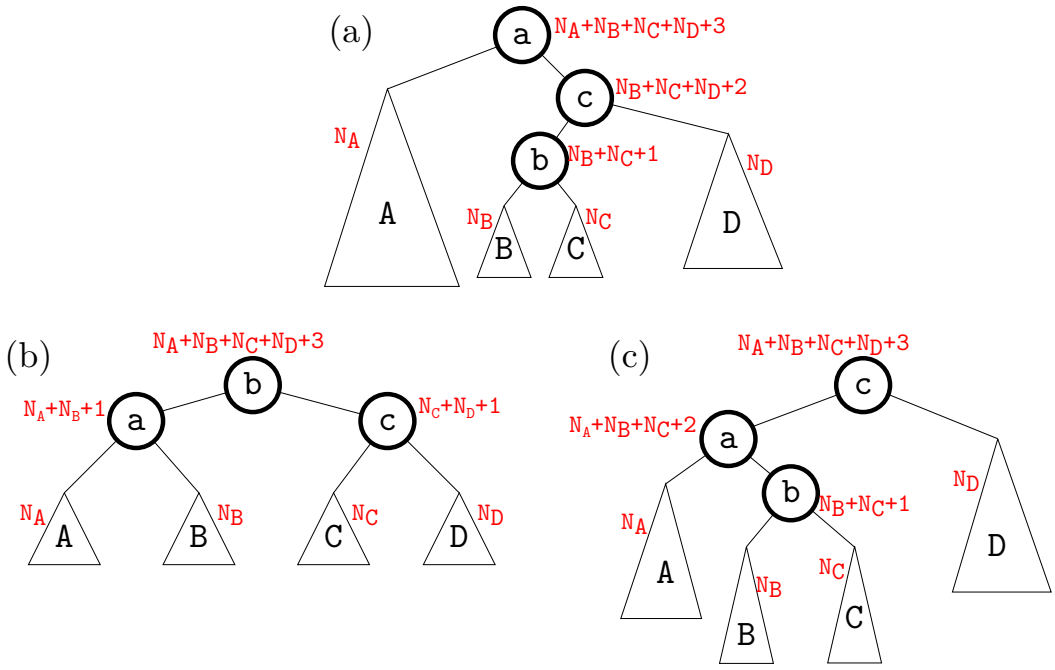
Do sedaj smo imeli le operacije nad dvojiškim drevesom. Kaj pa, če imamo drevo AVL?

Ponovimo, drevo AVL je delno uravnoteženo drevo. V vsakem vozlišču je lahko razlika med višinama njegovih poddreves kvečjemu 1. Operacije za vstavljanje in brisanje v AVL so enake tistim v dvojiškem, le da je na koncu še klic funkcije `Rebalance` nad staršem vstavljenega ali brisanega vozlišča.

Ideja je naslednja: Ker vstavljanje in brisanje lahko ostane enako, želimo spremeniti le funkcijo `Rebalance` tako, da ohrani invarianto števca c.

Recept za obe vrsti rotacij je prikazan na sliki 5.3. Pri enojni rotaciji je potrebno popraviti novi in stari koren (vozlišči a in c). Pri dvojni pa vrednosti vseh treh vozlišč (a, b in c). Število naslednikov v poddrevesih A, B, C in D ostane nespremenjeno.

Psevdokode ne bomo pisali, potrebno pa je k funkcijam `RotateL, RotateRL, RotateR, RotateLR` dodati popravljanje števca zgoraj omenjenih elementov.



Slika 5.3: Število naslednikov glede na rotacije AVL. (a) Prvotno drevo AVL. (b) Dvojna desna-leva rotacija (RL). (c) Enojna leva rotacija (L).

5.2 Kombiniranje podatkovnih struktur

Naloga Disjunktno množico smo implementirali z drevesom. Sedaj želimo imeti operacijo `PrintSet(x)`, ki bo izpisala vse elemente množice, kateri pripada `x`, pri čemer vrstni red izpisa ni pomemben. Zasnujmo kombinacijo podatkovnih struktur, ki bo omogočala, da operacijo `PrintSet(x)` izvedemo v času $O(n)$, pri čemer je n število elementov v množici, asimptotični časi ostalih operacij pa ostanejo enaki.

Poglavje 6

Dinamično programiranje

6.1 Uvod

Dinamično programiranje je način reševanja problemov. Sodobni pojem je vpeljal ameriški matematik Richard Bellman (Bellman-Fordov algoritem za iskanje najkrajših poti v grafu, tudi ko so uteži negativne) leta 1953. Beseda *programiranje* se ne nanaša na računalniško programiranje (npr. zasedanje in sproščanje pomnilnika pri objektnem programiranju), ampak na iskanje optimalne rešitve optimizacijskega problema, ki ga zapišemo kot matematični program.

Problem rešujemo z dinamičnim programiranjem, kadar izkazuje dve lastnosti:

1. **Optimalna podstruktura** (*optimal substructure*). Rečemo, da problem izkazuje optimalno podstrukturo, kadar lahko (optimalno) rešitev izračunamo z uporabo (optimalnih) rešitev podproblemov. To dejstvo se pri problemih ponavadi izrazi z rekurzivnimi enačbami.
2. **Prekrivajoči se podproblemi** (*overlapping subproblems*). Rečemo, da ima problem prekrivajoče se podprobleme, kadar pri reševanju različnih podproblemov naletimo na enake podpodprobleme.

Torej, če ima problem zgolj prvo lastnost, ga rešimo preprosto z metodo deli in vladaj. Taka problema sta npr. Quicksort ali Mergesort. Če pa problem izkazuje tudi drugo lastnost, potem bi program po metodi deli in vladaj opravljal odvečno delo, saj bi enake (pod)probleme reševal večkrat.

Preprost primer je izračun Fibonaccijevega števila ($fib(n) = fib(n-1)+fib(n-2)$, $fib(0) = 0, fib(1) = 1$). Le-tega lahko sprogramiramo tako, da funkcija rekurzivno računa "levi" podproblem ($fib(n-1)$) in "desni" podproblem ($fib(n-2)$). To je preprost primer metode deli in vladaj. Vendar opazimo, da izračun "levega" podproblema vsebuje tudi izračun "desnega" in tako po rekurziji naprej s pod-pod-...podproblemi, vse do najmanjšega (glej sliko **??** za primer iskanja zaporedja množenja matrik, podobno se lahko nariše na tablo za Fibonaccijevo zaporedje).

Z uporabo dinamičnega programiranja poskrbimo, da se enaki podproblemi rešujejo zgolj enkrat. Poznamo dva pristopa:

1. **Od zgoraj navzdol** (*Top-down*). Algoritem za problem z optimalno podstrukturo ponavadi zapišemo v rekurzivni obliki. V naivnem zapisu v rekurzivni obliki je algoritem neučinkovit (velikokrat ima eksponentno časovno zahtevnost). Razširimo ga tako, da ko prvič reši nek podproblem, rešitev shrani v tabelo. Pred vsakim rekurzivnim klicem tako preverimo, če smo podproblem že rešili, in če smo ga, vrnemo že izračunano shranjeno rešitev. To tehniko (shranjevanje že izračunanih rešitev) imenujemo **memoizacija** (*memoization*). Pristop se imenuje od zgoraj navzdol, ker je algoritem še vedno zapisan rekurzivno in tako najprej izvede klic na glavnem problemu, nato pa rekurzivno na podproblemih.

2. **Od spodaj navzgor** (*Bottom-up*). Pri tem pristopu rekurzivni zapis problema reformuliramo tako, da iterativno rešujemo podprobleme različnih velikosti. Začnemo z reševanjem najmanjših podproblemov, iz rešitev teh zgradimo rešitve večjih (pod)problemov in s tem nadaljujemo, dokler ne zgradimo rešitve problema želene velikosti.

Red časovne zahtevnosti algoritmov je pri obeh pristopih enak. Pristop od zgoraj navzdol z memoizacijo je morda bolj naraven in zahteva manj spreminjanja naivnega (počasnega) rekurzivnega algoritma (dodati je potrebno memoizacijo), vendar so algoritmi, ki gradijo rešitve od spodaj navzgor, pogosto hitrejši, ker ni režije rekurzivnih klicev. Je pa res, da če nek problem zahteva rešitev zgolj nekaterih podproblemov, in pri pristopu od spodaj navzgor ne delamo dodatnih optimizacij, potem je pristop od zgoraj navzdol boljši, saj izračuna zgolj tiste podprobleme, ki so potrebni za rešitev osnovnega problema.

Vsak problem, ki ima obe lastnosti, da ga rešujemo z dinamičnim programiranjem, lahko rešimo z obema pristopoma (od zgoraj navzdol in od spodaj navzgor). Prikazali bomo primere obeh vrst dinamičnih algoritmov.

Na tem mestu še poudarimo, da kadar rešujemo optimizacijske probleme, algoritem lahko vrne zgolj vrednost (ceno) najbolj optimalne rešitve, lahko pa vrne tudi rešitev samo. Slednje od algoritma zahteva, da pri iskanju optimalne rešitve shrani podatke o razdelitvi na podprobleme, če želimo, da rekonstrukcija rešitve na posameznem koraku zahteva le konstantnem dodatnega časa (kasneje vidimo primer tabele $s[i, j]$).

6.2 Rezanje palice

Glej [CLRS09], poglavje 15.1.

Poglavje 7

Rešitev 1. kolokvija

1. naloga

1.1 (6 točk)

Če so narisali in dobili odgovor 14 je pravilno. Odgovor 32 in manjša utemeljitev (samo črne) je bilo vredno točko. Tisti, ki so šli računat, kot da je AVL, so dobili 2-3 točke od 6, odvisno od utemeljitve.

1.2 (6 točk)

Obarvamo vozlišča 6 ali (4,7) ali (4,7,2,12). Če so napisali, da to ni RB drevo, ker nima barv in ne ustreza pravilom RB drevesa so dobili delne točke.

1.3 (8 točk)

i) slika(dve možnosti), Vedno dodajamo v desni list + rotacije.

II) je uravznoteženo, $\log_{b/2}((n-1)/2)+1$, nekaj točk za samo $\lg(n)$.

2. naloga

2. naloga je bila pretežno opisne narave.

2.1 (4 točke)

Ugotoviti je bilo potrebno, da je najboljši postopek za dan problem gozd disjunktnih množic. Dobili so tudi kakšno točke za morebitne druge rešitve (graf ipd., če so zadevo smiselno argumentirali).

2.2 (5 točk)

Dodajanje poteka enostavno preko operacij UNION in MAKE SET.

2.3 (6 točk)

Je malenkostno odvisna od tega, kar so predlagali v 2.1. (spet so dobili kakšno točko za argumentiranje svoje strukture, četudi ni bila optimalna). Za gozd dij. množic: predprocesiranje $O(nlogn)$ amortizirano poizvedba $O(1)$ vstavljanje $O(1)$

3. naloga

3.1 (5 točk)

```
(98, 130)
(98, 102)
(114, 103)
(116, 102)
(117, 116)
(120, 112)
(128, 78)
(129, 113)
(149, 122)
```

`(153, 108)`

Teza(108, 141) = 624

Teza(121, 151) = 313

Teza(97, 114) = 335 ali 232 (če velja stroga neenakost)

Teza(149, 124) = Teza(124, 149) = 313 ali 191

- če so okoli obrnili (večji od 149, manjši od 124), pol točk

Teza(124, 123) = Teza(123, 124) = 0

- če so okoli obrnili (dobili vsoto vseh), pol točk

=> ANDY je pravilno interpretacijo obrnjenega intervala NAPISAL NA TABLO

3.2 (3 točke)

V osnovi gre za rang in izbiro. Učinkovita rešitev je: - shrani predponske vsote za vse elemente v P. Potrebujemo $O(n)$ prostora. - poizvedba $Teza(x, y)$: poišči levega soseda od x v P in desnega soseda od y. $Teza(x, y) = P(y) - P(x)$

2 točki se dobi za manj učinkovito rešitev (npr. računanje vseh parov, $O(n^2)$ prostor in čas, tudi dinamično programiranje).

3.3 (6 točk)

Razširjeno dvojiško iskalno drevo za Rang in izbiro. Ključi vozlišč so višine otrok, dodana vrednost pa vsota tež npr. v levem poddrevesu. Tako je postopek iskanja podoben kot v 3.2 (s predponsko vsoto). Čas. zahtevnost O(log n), prostorska O(n).

4. naloga

4.1 (4 točke)

Od vseh tež odštejemo 111.

```
<višina, nova teža>
98 102
98 130
114 103
116 102
117 116
120 112
128 78
129 113
149 122
153 108
```

Največja vsota je med višinama 129 do vključno 149, to je 13.

Največ težav so imeli, ker so vzeli tistih 5 tež iz tretje naloge in niso izračunali največje teže med vsemi (urejenimi) pari. Za to so dobili 2 točki.

4.2 (6 točk)

Urejanje po višini: $O(n \log n)$ Časovna zahtevnost: $O(n^2)$ (iskanje vseh možnih parov) Prostorska: $O(n)$ (vhodni podatki+nekaj začasnih spremenljivki)

Če so ponovili napako in napisali, da gredo le skozi teže, podane iz prejšnje naloge, so dobili $O(n)$ čas in 3 pike.

4.3 (8 točk)

i) Uporabili bi dinamično programiranje. Gre za problem iskanja najdaljšega podzaporedja (*longest common subsequence*).

ii) Imejmo dva niza X in Y. $LCS(X_0, Y_0) = 0$

$$LCS(X_i, Y_j) = \max \begin{cases} LCS(X_{i-1}, Y_j), \\ LCS(X_i, Y_{j-1}), \\ LCS(X_{i-1}, Y_{j-1}) + 1, \text{ če } x_i = y_j \text{ ali } +0, \text{ če } x_i \neq y_j \end{cases}$$

Poglavje 8

Delo z nizi

8.1 Iskanje podnizov v besedilu

8.1.1 Naivna metoda

Glej [SW11], str. 760.

8.1.2 KMP

Glej [SW11], str. 762.

8.1.3 Boyer-Moore

Glej [SW11], str. 770.

8.2 *Trie* — Številsko drevo [De 59, Fre60]

Imejmo množico besed (nizov). Kako ugotovimo, ali je v množici prisoten nek niz ali ne? Lahko jih uredimo in uporabimo bisekcijo. Kaj pa, če so nizi dolgi — ali bomo pri primerjavah primerjali vedno celoten niz oz. do tja, kjer se

niza razlikujeta? V takih primerih uporabimo številsko drevo — je prostorsko učinkovitejše in vsaj tako hitro kot omenjena bisekcija.

Naše vhodne nize bomo poimenovali kar **ključi**, saj bomo po njih iskali podatke.

8.2.1 Operacije

Številsko drevo je iskalno drevo z naslednjimi operacijami:

- `Insert(String key)` — vstavi ključ `key` v drevo,
- `Retrieve(String key)` — poišče ključ `key` in vrne vrednost,
- `Remove(String key)` — odstrani ključ `key` v drevesu,
- `Left(String key)` — poišče prvi manjši element od `key`,
- `Right(String key)` — poišče prvi večji element od `key`.

Dodamo še dve uporabni operaciji (niste jemali na predavanjih):

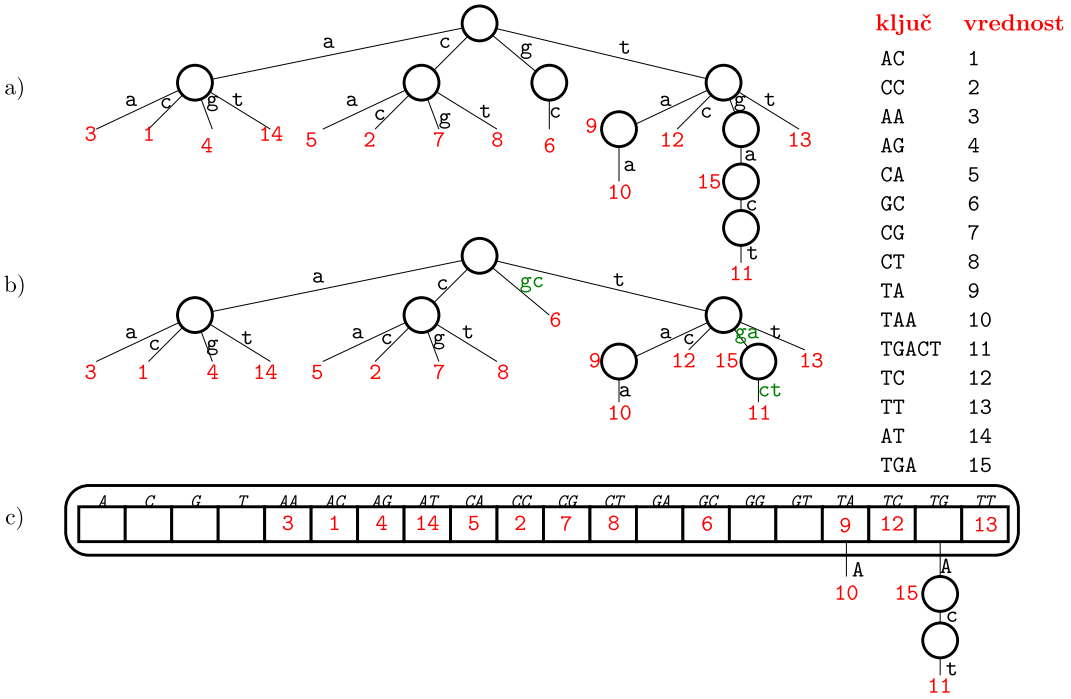
- `LongestPrefixOf(String s)` — poišče in vrne najdaljši ključ v drevesu, ki je predpona podanega niza s. Uporabno pri usmerjevalnikih!
- `KeysWithPrefix(String s)` — poišče in vrne vse ključe, ki se začnejo na s. Uporabno pri iskalnikih (npr. autocomplete)!

V drevesu so shranjeni ključi tako, da vsako vozlišče predstavlja en znak od korena proti listom. Vozlišče z zadnjim znakom ključa vsebuje tudi vrednost (npr. referenco na iskan objekt). Znaki so del abecede Σ, dolžine ključev so lahko poljubne.

Pozor Na predavanjih ste imeli n nizov, dolgih fiksno m bitov, se pravi $\Sigma = \{0, 1\}$. Pri nas bomo imeli $\Sigma = \{A, C, G, T\}$ in različno dolge nize.

Ali lahko pohitrimo keysWithPrefix? Da. V notranja vozlišča vstavimo kazalce na najbolj levi in najbolj desni liste poddrevesa. Nato liste povežemo v povezan seznam in se le sprehodimo skozi njih — ni se več potrebno sprehajati gor in dol. Pohitritev iz $O(mk) \to O(k)$, kjer je k število zadetkov in m dolžine nizov.

Pozor Listi v številskem drevesu so leksikografsko urejeni!



Slika 8.1: a) številsko drevo (*trie*), b) stiskanje poti (*PATRICIA*), c) stiskanje plasti (*LC trie*)

8.2.2 Analiza

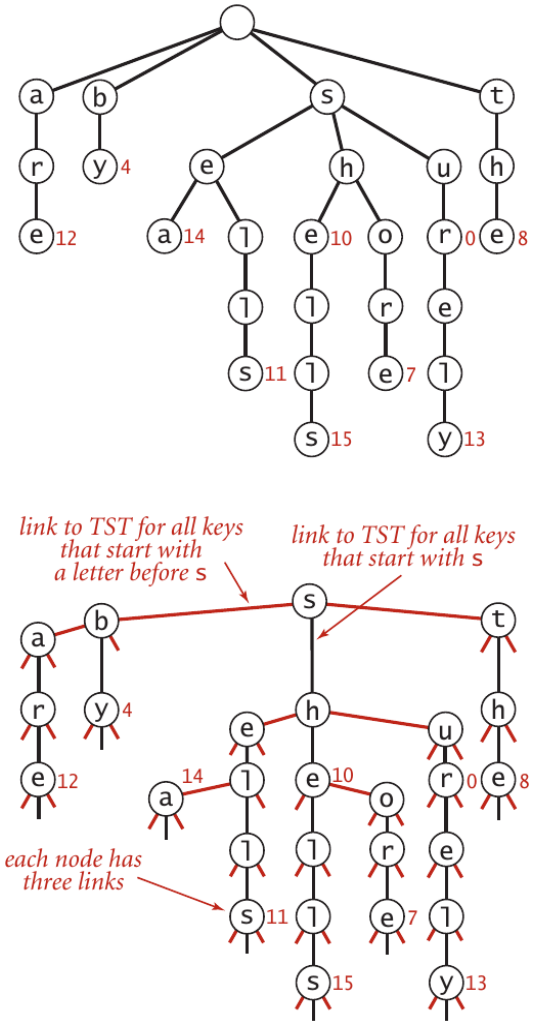
Časovna zahtevnost vstavljanja, brisanja in poizvedbe je omejena z dolžino podanega niza, se pravi $O(m)$.

Prostorska zahtevnost je odvisna od števila ključev n, dolžine ključev m in podobnosti med seboj. Če je drevo polno, je skupnih veliko predpon in imamo $O(|\Sigma|^m) = O(n)$ vozlišč — vsi ključi so dolgi m in se končajo na zadnjem nivoju, imamo torej $n = |\Sigma|^m$ listov in $O(n)$ notranjih vozlišč (za dvojiško drevo $n - 1$ notranjih, v splošnem $\frac{|\Sigma|^m - 1}{|\Sigma| - 1}$). Lahko pa je drevo zelo redko. Npr. dolžine ključev so $m >> n$. V tem primeru imamo nm vozlišč v najslabšem primeru, če se vsaka beseda začne na drug znak. Prostorska zahtevnost v najslabšem primeru je $O(nm)$.

8.2.3 Zapis vozlišča

Kako zapisati kazalce na naslednike v vozlišče? S slovarjem. Slovar običajno izvedemo z enim od treh načinov (glej [SW11]):

- s povezanim seznamom. Prostor $O(nm)$, čas $O(m|\Sigma|)$.
- s poljem. Prostor $O(nm|\Sigma|)$ — če ne štejemo le vozlišč, ampak tudi njihovo velikost, čas $O(m)$.
- trojiško številsko drevo (*Ternary Search Trie*): Dvojiško iskalno drevo znotraj vozlišča. Prostor $O(nm)$, čas $O(m \lg |\Sigma|)$. Priporoča Sedgewick — glej sliko 8.2.



Slika 8.2: Trojiško iskalno drevo.

8.3 *PATRICIA* — Stiskanje poti [Mor68]

Ali res potrebujemo nm vozlišč? Ne. Lahko uporabimo stiskanje poti (*path compression*, poimenovano PATRICIA). Vozlišča z le enim naslednikom združimo v eno vozlišče. Vozlišče sedaj ne predstavlja enega ampak niz znakov (implementiran npr. s povezanim seznamom). Sedaj imamo prostorsko zahtevnost reda $O(n)$ vozlišč. Časovna zahtevnost ostane enaka, saj moramo še zmeraj primerjati vse znake znotraj vozlišča z iskanim nizom.

Pozor Ko združimo vozlišča, s tem res zmanjšamo njihovo število, povečamo pa velikost posameznega vozlišča! Zakaj je to pomembno? Vsako vozlišče v drevesu potrebuje vsaj dva kazalca: na svojega otroka in naslednjega sorojenca. Kazalci so dragi (v praksi 32− ali 64−bitov) in se jih poskušamo izogibati. Z združitvijo dveh vozlišč tako prihranimo dva kazalca, vozlišče pa povečamo zgolj za 1 znak. Na koncu se združevanje še vedno splača.

Dodatna prednost je tudi boljša predpomnilniška učinkovitost pri iskanju.

PATRICIA je bila prvotno uporabljena kot izboljšava nad *priponskimi drevesi*.

8.4 *LC Trie* — Stiskanje plasti [AN93]

Če je drevo zelo gosto ($n \to |\Sigma|^m$), lahko iskanje pohitrimo z združevanjem več plasti drevesa v polje. Če vozlišča predstavimo v polju, je dostop do elementa direkten in s tem pohitrimo iskanje (nimamo več primerjav znakov, ampak v $O(1)$ izračunamo mesto vozlišča v polju). Ta rešitev se imenuje *Level-Compressed Trie*.

Koliko hitreje deluje LC-številsko drevo od običajnega številskega drevesa? Če združimo vseh m plasti, dobimo polje veliko $|\Sigma|^m$, kjer je m dolžina vstavljenih ključev (recimo, da je dolžina fiksna). Vse možne nize, ki jih lahko zapišemo v naši podatkovni strukturi imenujemo **univerzalna množica** $U = \Sigma^m$. Ker je celotna struktura eno veliko polje, je iskanje po njem $O(1)$, saj moramo le izračunati, na katerem mestu se nahaja iskani element, do njega pa dostopamo direktno. V praksi je univerzalna množica običajno prevelika ali celo neskončna (bodisi m, bodisi Σ nista znana vnaprej), tako da združujemo le prvih nekaj nivojev od korena proti listom in mogoče še v katerem od poddreves. Zgornji

del drevesa je namreč statistično gostejši od spodnjega (stopnje vozlišč — koliko neposrednih naslednikov ima vozlišče — blizu korena so večje od tistih spodaj). Časovna zahtevnost iskanja elementa se zmanjša na $O(m - L)$, kjer je L število stopenj, ki smo jih združili. Če smo združili le del poddreves, vzamemo za L sorazmerni delež združenih vozlišč na plasti. Prostorska zahtevnost ostane $O(nm)$.

Naloga Koliko vozlišč obiščemo za iskanje elementa TAA v 8.1.a)?

Sedaj pride na vrsto postopek stiskanja plasti. Izberemo si prag (gostoto vozlišč) α, ki jo želimo imeti, da plasti še združimo. Recimo $\alpha = 0{,}8$. Začnemo pri korenu in obdelamo prvi nivo. Prisotna so vozlišča vseh znakov abecede (A,C,T,G). Imamo torej gostoto 1. Obdelamo drugi nivo. Vozlišče A ima vse štiri znake abecede, vozlišče C prav tako. Vozlišče T ima na drugem nivoju tudi še vse štiri naslednike. Vozlišče G pa le enega (C). Dobimo gostoto $17/20 = 85\%$, kar je še nad našim pragom. Obdelamo tretji nivo. Prisotna sta le dva naslednika: vozlišče A ima naslednika A in vozlišče G naslednika A. Gostota je torej $19/84 = 22{,}62\%$. Ker smo padli pod prag, po pravilu vse prejšnje plasti združimo, torej 1. in 2. nivo združimo v eno vozlišče.

Po končanem postopku stiskanja dobimo drevo na sliki 8.1.c). Na prvem nivoju imamo zdaj vozlišče s poljem, velikim 20 referenc (4 potencialna vozlišča na prvem nivoju in 16 na drugem). 12 je zasedenih, 8 je prostih. Iz polja peljeta še dva ključa, ki sta dolga 3 znake (`TAA` in `TGA`) in sta na naslednjem nivoju. Koliko primerjav znakov potrebujemo za iskanje elementa `TAA` sedaj? 2, ker za iskanje `TA` nimamo primerjav (izračunamo, da je to 13. element v polju po vrsti in ga enostavno pogledamo), za iskanje `A` pa potrebujemo eno običajno primerjavo.

Pozor Običajno nas $O(m-L)$ ne zanima, če je L majhen, ker lahko zapišemo tudi, da smo izboljšali $O(m)$ za nek faktor in ostane red velikosti enak. L postane pomemben takrat, ko je $L \approx m$. Takrat dobimo povprečno časovno zahtevnost iskanja v našem številskem drevesu $O(1)$.

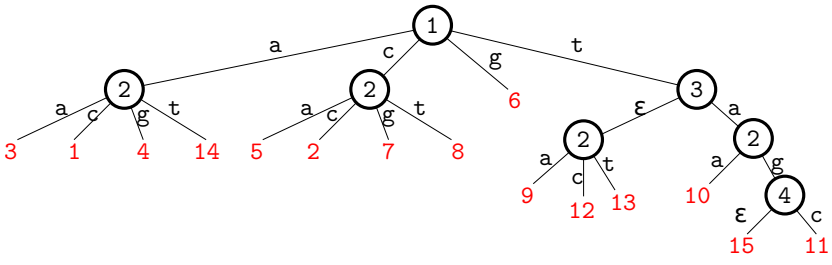
8.5 Številsko drevo z nezaporednimi indeksi

Na predavanjih ste omenili, da namesto primerjav znakov od korena do listov v naraščajočem redu lahko primerjamo tudi poljuben znak. Pogoj je, da imajo

notranja vozlišča > 1 otrok.

Drevo z nezaporednimi indeksi izhaja iz stiskanja poti (hranimo le "pomembna" vozlišča). V praksi je to uporabno, ker pri dodajanju ne spreminjamo naslovov obstoječih vozlišč (ni cepljenja vozlišč, ampak le dodajamo).

Poglejmo si številsko drevo z nezaporednimi indeksi na ključih iz prejšnjega poglavja:



ključ	vrednost
AC	1
CC	2
AA	3
AG	4
CA	5
GC	6
CG	7
CT	8
TA	9
TAA	10
TGACT	11
TC	12
TT	13
AT	14
TGA	15

Slika 8.3: Številsko drevo z nezaporednimi indeksi.

8.6 Priponsko drevo (*Suffix tree*)

Priponsko drevo je PATRICIA, ki vsebuje vse pripone podanega besedila dolžine n. Namesto ločenih besed, dolgih po $O(n)$ znakov si lahko zapomnimo le začetek pripone in dolžino, če gre za notranje vozlišče, kar zmanjša velikost posamezne povezave na $O(1)$. Celotno priponsko drevo tako veliko zavzame le $O(n)$ prostora.

Iskanje poteka na enak način kot pri PATRICIJI.

8.7 Priponsko polje (*Suffix array*)

Priponsko polje SA je polje, ki hrani leksikografski vrstni red pripon [MM90]. $SA[1]$ vsebuje mesto začetka pripone v besedilu, ki je leksikografsko prva. Priponsko polje je veliko natanko n besed, zato je prostorsko učinkovitejše od priponskega drevesa.

Iskanje po priponskem polju poteka podobno kot v običajnem urejenem polju števil — z dvojiškim iskanjem. Le da namesto vrednosti elementov i in j primerjamo leksikografsko urejenost pripon, ki se začnejo na mestu $SA[i]$ in $SA[j]$ v besedilu. Še en detajl — z dvojiškim iskanjem moramo najti leksikografsko prvo pripono, k. Nato izpisujemo po vrsti pripone $SA[k]$, $SA[k+1]$..., dokler se te še ujemajo z vzorcem. Časovna zahtevnost iskanja je $O(\lg n)$ primerjav pripon oz. $O(m \lg n)$ primerjav znakov.

Poleg SA omenimo še LCP, to je *longest common prefix*. Gre za dolžino najdaljše skupne predpone v SA. Če izračunamo obe polji — LCP in SA, lahko s pomočjo Range-Minimum-Query iščemo po priponskem polju v $O(m + \lg n)$ primerjavah besed. Algoritem za RMQ nam poišče najmanjšo vrednost elementa v podanem intervalu v $O(1)$ času. S pomočjo tega zagotovimo, da ne primerjamo večkrat iskani niz s priponami, ampak pregledamo vsak znak kvečjemu enkrat. Preostanek se sprehajamo po LCP in z bisekcijo zmanjšujemo interval za RMQ.

Poglavje 9

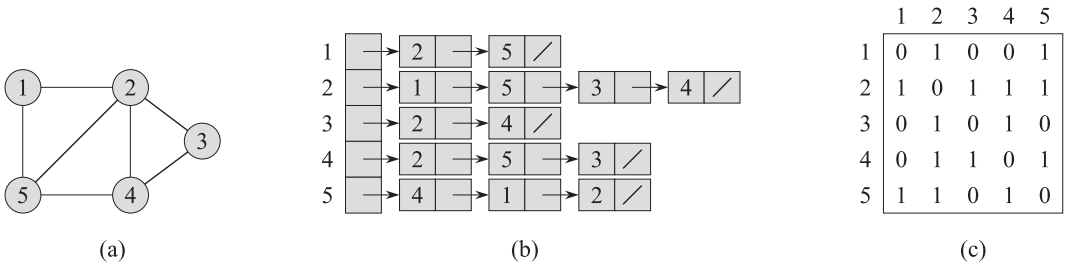
Algoritmi na grafih

Veliko problemov v računalništvu modeliramo s pomočjo grafov, kar daje pomembnost algoritmom na grafih.

Graf je v računalništvu abstraktna podatkovna struktura, s katero predstavimo matematični pojem grafa. Le-ta sestoji iz množice vozlišč (node) oz. **točk** (*vertex*) in množice **povezav** (*edge*) med točkami. Prvo množico ponavadi označimo z V, slednjo pa z E, graf je potem $G = (V, E)$. V splošnem so povezave urejeni pari točk, govorimo torej o **usmerjenem** (*directed*) grafu, kjer je povezava $(a, b),\ a, b \in V$ različna od povezave (b, a). Kadar sta ti dve povezavi enaki, govorimo o **neusmerjenem** (*undirected*) grafu. Omenimo še, da je graf lahko **utežen** (*weighted*). V tem primeru vsaki povezavi pripada še neka teža (cena, kapaciteta, ...). Omenimo, da je pomen točk, povezav, teže itd. splošen in določen s problemom, ne pa s samo teorijo grafov.

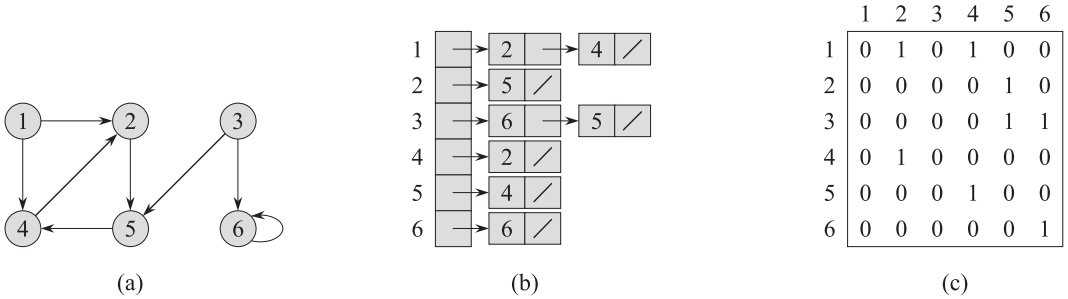
Graf lahko predstavimo s **seznamom sosednosti** (*adjacency list*) ali z **matriko sosednosti** (*adjacency matrix*). Oba načina prikazujeta sliki 9.1 (neusmerjen graf) in 9.2 (usmerjen graf). Način s seznamom je bolj primeren za grafe z manj povezavami, saj mora biti matrika sosednosti velika $|V|^2$, kot je največje možno število povezav. V načinu s seznamom imamo polje seznamov Adj dolžine $|V|$, kjer za vsako točko $u \in V$, seznam $Adj[u]$ vsebuje vse točke $v \in V$, za katere je $(u, v) \in E$. Matrika sosednosti pa je matrika $|V| \times |V|$, kjer je $a_{ij} = 1$, če je $(i, j) \in E$, in 0 sicer. Pri tem je $1 \leq i, j \leq |V|$, točke torej oštevilčimo po nekem poljubnem vrstnem redu.

Obstaja tudi incidenčni zapis. V primeru seznama vsako vozlišče hrani seznam sosednjih povezav (povezav, ki se začnejo ali končajo v tem vozlišču),



	1	2	3	4	5
1	0	1	0	0	1
2	1	0	1	1	1
3	0	1	0	1	0
4	0	1	1	0	1
5	1	1	0	1	0

Slika 9.1: [CLRS09] str. 590: Predstavitev (a) neusmerjenega grafa in zapis s seznamom sosedov (b) in matriko sosedov (c).



	1	2	3	4	5	6
1	0	1	0	1	0	0
2	0	0	0	0	1	0
3	0	0	0	0	1	1
4	0	1	0	0	0	0
5	0	0	0	1	0	0
6	0	0	0	0	0	1

Slika 9.2: [CLRS09] str. 590: Predstavitev (a) usmerjenega grafa in zapis s seznamom sosedov (b) in matriko sosedov (c).

ter vsaka povezava hrani seznam sosednjih vozlišč (vozlišč, ki so prva ali druga komponenta povezave). V primeru matrike imamo $|V| \times |E|$ matriko, ki hrani enice za vozlišča (v vrsticah), ki so del povezave (v stolpcih). Vendar nas to tu ne zanima. Od tu dalje delamo z seznamov sosednosti kot v [CLRS09].

9.1 Preiskovanje grafov — ponovitev APS1

9.1.1 Preiskovanje grafov (*graph traversal*)

Preiskovanje (obhod, sprehod) grafa je problem, kako obiskati vse točke (vozlišča) grafa. Gre za generalizacijo preiskovanja drevesa, z razliko, da v primeru grafov posamezno vozlišče lahko obiščemo večkrat, prav tako pa morda ne obstaja korensko vozlišče, t.j. vozlišče, od koder lahko pridemo do vseh ostalih vozlišč (npr. večdelni grafi, ali pa določeni primer usmerjenih grafov). Pri tem

poznamo dva načina, in sicer **preiskovanje (iskanje) v širino** (*breadth-first search*) ter **preiskovanje v globino** (*depth-first search*).

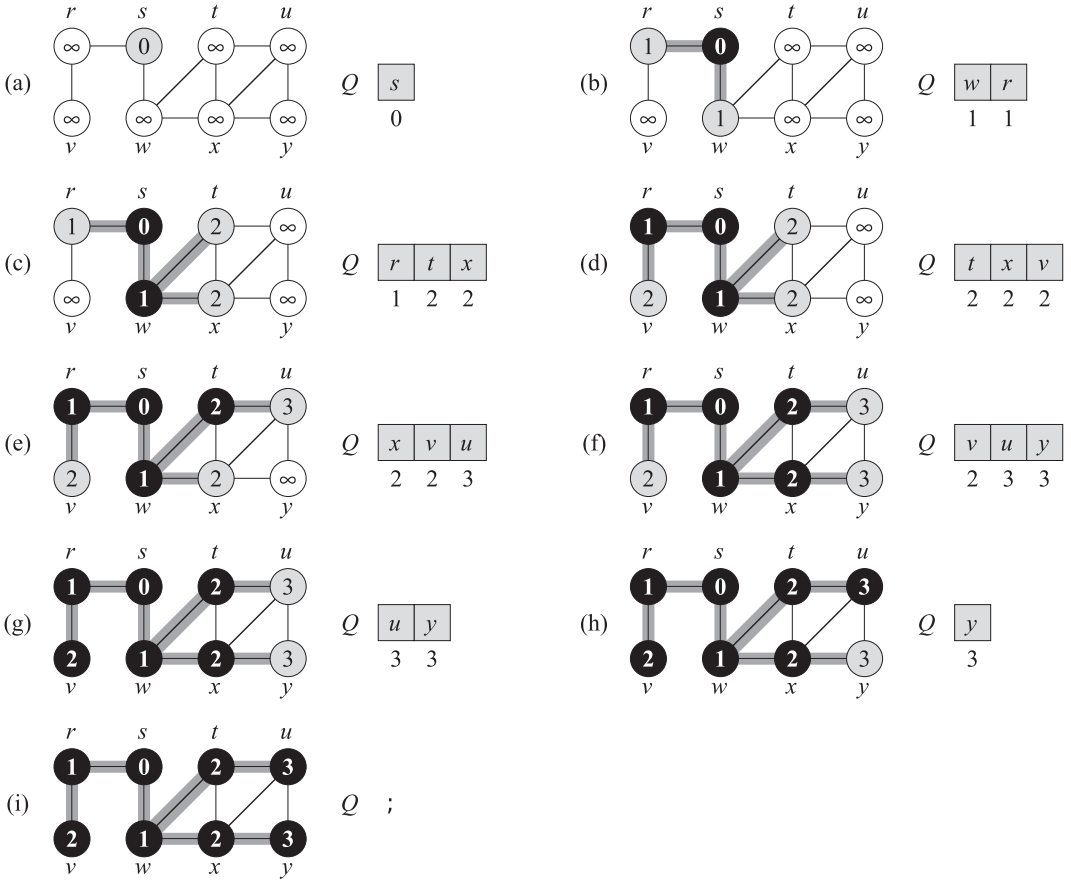
Preiskovanje v širino (BFS)

Imejmo graf $G = (V, E)$ ter **izvorno točko** (*source vertex*) s. BFS (glej algoritem 17) preišče povezave G, tako da najde vse točke, ki so preko povezav dosegljive iz s. Pri tem shrani razdaljo (število povezav) do vsake dosegljive točke. Ker algoritem deluje tako, da najprej poišče vse točke na razdalji k, nato na $k+1$ itd., rečemo da najprej preišče širino na neki razdalji, potem pa poveča globino. Algoritem s tem tudi zgradi drevo najkrajših poti do točk dosegljivih iz s, pri čemer je s koren. BFS deluje tako na usmerjenih kot neusmerjenih grafih. Primer delovanja algoritma prikazuje slika 9.3.

Algoritem 17: BFS(G, s)

```
1  foreach vertex u ∈ G.V − {s} do
2      u.color ←WHITE
3      u.d ← ∞
4      u.π ← ∅
5  s.color ←GRAY
6  s.d ← 0
7  s.π ← ∅
8  Q ← empty FIFO priority queue
9  ENQUEUE(Q,s)
10 while Q ≠ ∅ do
11     u = DEQUEUE(Q)
12     foreach v ∈ G.Adj[u] do
13         if v.color = WHITE then
14             v.color ← GRAY
15             v.d ← u.d + 1
16             v.π ← u
17             ENQUEUE(Q,v)
18     u.color ←BLACK
```

Algoritem si pomaga z barvo točk, ki je lahko bela, siva ali črna. Poleg tega uporablja FIFO vrsto Q za hranjenje sivih točk. $u.\pi$ hrani prednika (starša)



Slika 9.3: [CLRS09] str. 596: Primer izvajanja algoritma BFS.

točke u na poti iz s do u. V $u.d$ je shranjena razdalja (število povezav) od s do u. Bele točke so še neobiskane. Sive in črne so že odkrite, pri čemer velja, da če je $(u, v) \in E$ in je u črna, potem je v črna ali siva, torej vse točke sosednje črnim so bile že odkrite. Točke sosednje sivim, pa imajo lahko bele sosede. Sive točke so torej meja med dolžinama iskanja k in $k + 1$. Kadar algoritem tekom preiskovanja seznama sosednosti točke u naleti na belo točko v, doda v in povezavo (u, v) v drevo. V tem primeru je torej $v.\pi = u$.

Časovna zahtevnost algoritma je reda $O(V + E)$, torej linearno glede na velikost predstavitve s seznami sosednosti. Prva zanka (inicializacija) traja $O(V)$, forzanka znotraj while pa se izvede po enkrat za vsako povezavo znotraj vsakega seznama sosednosti. Ker je vsota povezav čez vse sezname reda $O(E)$, ta korak tudi traja toliko časa.

Naloga [CLRS09] 22.2-3: Ali algoritem BFS deluje tudi, če uporabimo le dve barvi namesto treh — brez sive? Tako bi imeli v vozlišču že dovolj le en bit za barvo.

Naloga Premer drevesa je definiran kot največja izmed vseh najkrajših poti med vsemi vozlišči. Podajte učinkovit algoritem za izračun premera grafa in njegovo časovno zahtevnost.

Preiskovanje v globino (DFS)

BFS bi lahko razširil tako, da ko najde vse točke dosegljive iz s, nadaljuje s poljubnim novim izvorom izmed preostalih točk. Vendar se BFS ponavadi ne uporablja za ta namen, po drugi strani pa DFS se, saj je večkrat podprogram drugih algoritmov, kot bomo videli pri topološkem urejanju.

Primer izvajanja algoritma DFS (18) prikazuje slika 9.4. Algoritem je malo bolj zapleten kot tisti na Wikipediji, zaradi nekaterih podrobnosti. Razlaga je spodaj.

Kot že ime pove, pri DFS iščemo po povezavah v globino naprej od neke točke v, in šele ko ne preostane nobena povezava več, se vrnemo nazaj (*backtrack*) in nadaljujemo z nepreiskanimi povezavami, ki gredo ven iz točke, iz katere smo prišli do v. Enako kot pri BFS se ta postopek nadaljuje, dokler imamo nepreiskane povezave. Če je po koncu tega postopka še kakšna točka neobiskana, kot smo že omenili, izberemo nov izvor.

Enako kot pri BFS, kadar algoritem pri preiskovanju seznama sosednosti točke u odkrije novo točko v, nastavi $v.\pi = u$. Za razliko od BFS, kjer je podgraf prednikov drevo, je pri DFS (zaradi več možnih izvorov) podgraf prednikov gozd dreves. Enako kot pri BFS, so neobiskana vozlišča bela, obiskana vendar ne zaključena (seznam sosednosti ni preiskan v celoti) siva, ter zaključena črna. DFS si tudi shrani vrednost d in f za vsako točko, nekakšni časovni oznaki, kdaj je bila točka odkrita in kdaj zaključena (iskanje je preiskalo celoten seznam sosednosti). Ker ima vsaka točka dve oznaki, so te oznake med 1 in $2|V|$. S pomočjo teh oznak se lažje analizira izvajanje algoritma, prav tako pa bodo uporabljene pri topološkem urejanju, konkretno oznaka f.

Algoritem deluje v času $O(V + E)$.

Algoritem 18: Algoritem preiskovanja v globino.

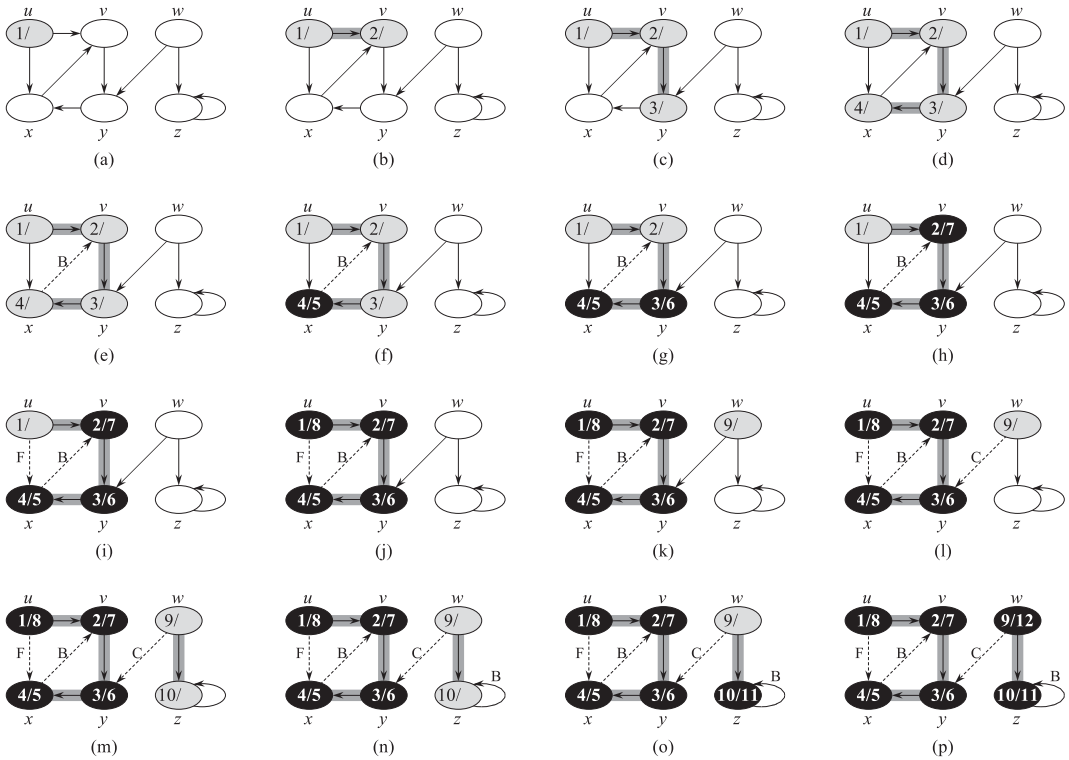
```
1  function DFS(G)
2      foreach vertex u ∈ G.V do
3          u.color ← WHITE
4          u.π = ∅
5      time = 0
6      foreach vertex u ∈ G.V do
7          if u.color = WHITE then
8              DFS-VISIT(G,u)

9  function DFS-VISIT(G, u)
10     time ← time + 1
11     u.d ← time
12     u.color ← GRAY
13     foreach vertex v ∈ G.Adj[u] do
14         if v.color = WHITE then
15             v.π ← u
16             DFS-VISIT(G,v)
17     u.color ← BLACK
18     time ← time + 1
19     u.f ← time
```

9.2 Topološko urejanje — ponovitev APS1

Tu si ogledamo algoritem za topološko urejanje **usmerjenega acikličnega grafa** (*directed acyclic graph*) oz. DAG-a. Topološko urejanje DAG-a $G = (V, E)$ je tako urejanje točk, da če je v grafu povezava (u, v), potem je u pred v v urejanju. Lahko si predstavljamo, da gre za urejenost točk, če so usmerjene povezave razporejene vodoravno tako, da kažejo v desno.

Usmerjeni aciklični grafi se velikokrat uporabljajo za predstavitev precedenčne odvisnosti med dogodki. Slika 9.5 prikazuje primer grafa precedenc za oblačenje profesorja :) ter pripadajoče urejanje. Ob točkah sta števili d in f, ki smo jih opisali pri DFS. Samo urejanje ob prej zapisanem DFS algoritmu poteka v eni izvedbi treh korakov:

1. Izvedi $DFS(G)$, da dobiš vrednosti f za vse točke.



Slika 9.4: [CLRS09] str. 605: Primer izvajanja algoritma DFS.

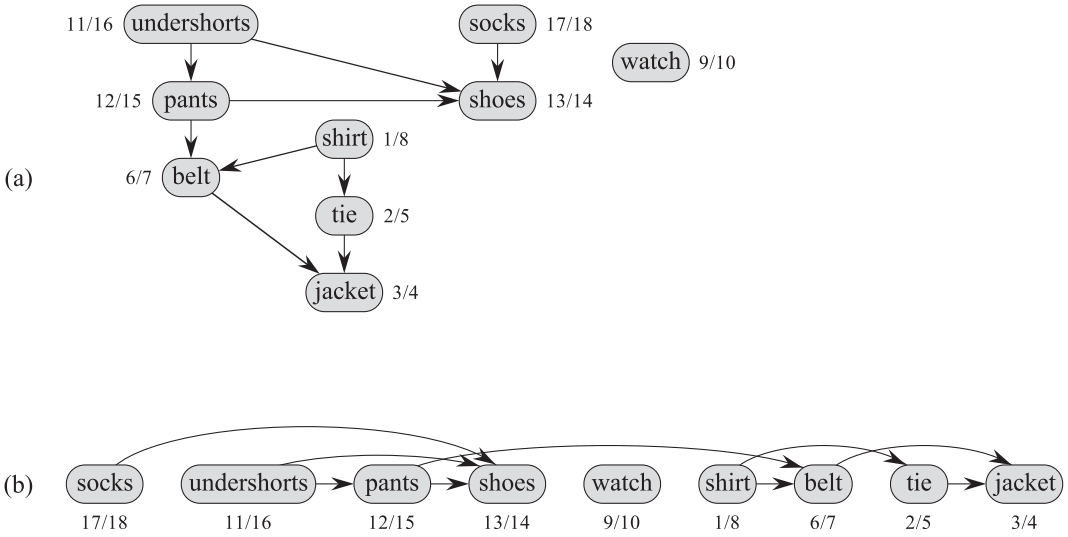
2. Točke po padajočih f vstavi na *začetek* povezanega seznama (glej sliko 9.5, kako vrednosti f padajo po vrsti).

3. Vrni povezan seznam.

Z drugo alinejo dosežemo, da se točka v doda na seznam tako, da so levo vse tiste točke, od katerih je v odvisna, torej tiste, ki imajo večji f. Algoritem potrebuje $O(V + E)$ časa za izvedbo DFS ter dodatnih $O(V)$ za vnos vseh točk na začetek povezanega seznama, skupno torej $O(V + E)$.

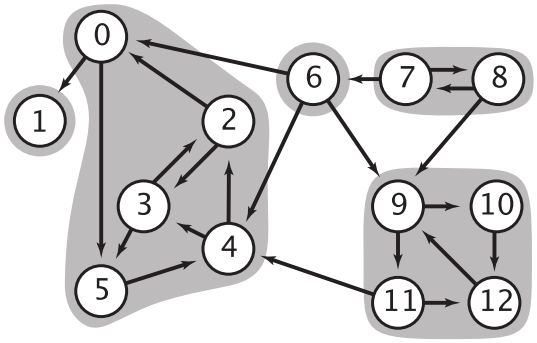
9.3 Krepko povezane komponente v usmerjenem grafu

Naloga Imamo neusmerjen graf. Zanima nas, v kateri povezani komponenti je posamezno vozlišče. Katero podatkovno strukturo že uporabimo?



Slika 9.5: Primer topološkega urejanja (b) za dani DAG (a).

Pri usmerjenem grafu nam disjunktne množice pomagajo bolj malo. Na sliki 9.6 so s sivo označene krepko povezane komponente v usmerjenem grafu.



Slika 9.6: [SW11], str. 584: Krepko povezane komponente.

Če želimo poiskati povezane komponente v usmerjenem grafu (imenovane *krepko povezane komponente*), lahko uporabimo Kosarajujev algoritem (objavljen l. 1981):

1. Poleg vhodnega grafa G uporabimo še sklad. Izberemo naključno vozlišče v in naredimo sprehod v globino.
2. Ob sestopanju dodamo vozlišče, ki ga zapuščamo na sklad. (na koncu je na skladu naše izvorno vozlišče za trenutno rundo)

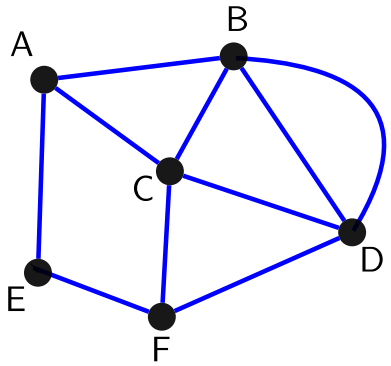
3. Postopek ponavljamo (skočimo na korak 1), dokler sklad ne vsebuje vseh vozlišč grafa.
4. Obrnemo smeri povezav v G.
5. Vzamemo zadnje vozlišče na skladu v in naredimo sprehod v globino.
6. Vsa obiskana vozlišča so v isti krepko povezani komponenti.
7. Vozlišča odstranimo s sklada in skočimo na korak 5.

Intuicija algoritma: Povezane komponente so cikli. Težavo povzročajo tuja vozlišča, ki imajo enosmerno povezavo v cikel, niso pa dostopna s cikla navzven. Zato povezave obrnemo in ga poskušamo doseči, kar ni mogoče. Vozlišča znotraj cikla pa še vedno lahko dosežemo. Primer: $A \leftrightarrow B \rightarrow C$. npr. začnemo v B, dobimo sklad: C, A, B. Sklad nam pa zagotovi, da se najprej lotimo vozlišč znotraj cikla (B ali A bosta na koncu sklada, nikakor pa ne C). V nasprotnem primeru bi dobili, da je zunanje vozlišče C tudi del krepko povezane komponente, kar pa ni.

Naloga S Kosarajujevim algoritmom poišči vse krepko povezane komponente grafa na sliki 9.6.

9.4 Eulerjev obhod, sprehod

Pri Eulerjevem obhodu po grafu $G = (V, E)$ moramo prehoditi vsako povezavo natančno enkrat, lahko pa večkrat obiščemo vozlišča.



Slika 9.7: Vhodni graf.

Naloga
1. Pokažite, da obstaja Eulerjev obhod samo, če za vsa vozlišča velja, da je število vstopajočih povezav enako številu izstopajočih povezav.
2. Zapišite psevdokodo algoritma.

Namig: Če najdemo v grafu dva cikla, ki se stikata v vsaj eni točki, ju lahko združimo v daljši Eulerjev cikel (še ne obhod).

Naloga Naredi Eulerjev obhod, če je mogoč, na grafu s slike 9.7.

9.5 Minimalno vpeto drevo

Minimalno vpeto drevo (*minimal spanning tree* ali *minimal-weight spanning tree*) nad grafom G je drevo, ki pokriva vsa vozlišča grafa in minimizira skupno vsoto uteži povezav.

Naloga Koliko je povezav v minimalnem vpetem drevesu?

9.5.1 Primov algoritem

[Vojtéch Jarník 1930, Robert Clay Prim 1957] Začne v izbrani točki (lahko naključni) in gradi MVD v širino tako, da izbere najcenejšo povezavo do še ne vključenega vozlišča. Potrebuje $O(|E| + |V| \log |V|)$ w.c. z uporabo Fibonaccijeve kopice.

Naloga Za podan graf s Primovim algoritmom zgradi minimalno vpeto drevo.

Algoritem 19: MST-Prim(G, w, r)

```
1  begin
2      foreach u ∈ G.V do
3          u.key ← ∞
4          u.π ← ∅
5      r.key ← 0
6      Q ← G.V
7      while Q ≠ ∅ do
8          u ← Q.DeleteMin()
9          foreach v ∈ G.Adj[u] do
10             if v ∈ Q ∧ w(u, v) < v.key then
11                 v.π ← u
12                 v.key ← w(u, v)
```

9.5.2 Kruskalov algoritem

[Kru56] Izbira med vsemi najcenejšimi povezavami v grafu in jih dodaja k MVD, če vozlišča še niso vključena. Potrebuje $O(|E| \log |V|)$ časa w.c.

Algoritem 20: MST-Kruskal(G, w)

```
1  begin
2      A ← ∅
3      foreach v ∈ G.V do
4          MakeSet(v)
5      sort the edges of G.E into nondecreasing order by weight w
6      foreach edge (u, v) ∈ G.E do
7          if FindSet(u) ≠ FindSet(v) then
8              A ← A ⋃ {(u, v)}
9              Union(u, v)
10     return A
```

Naloga Za podan graf s Kruskalovim algoritmom zgradi minimalno vpeto drevo.

Naloga Podana je psevdokoda treh algoritmov: MAYBE-MST-A, MAYBE-MST-B, MAYBE-MST-C (21). Algoritmom podamo povezan graf G ter seznam uteži w, kot rezultat pa dobimo seznam povezav T. Za vsak algoritem bodisi dokažite, da je T minimalno vpeto drevo, bodisi dokažite da ni. Kako bi najbolj učinkovito implementirali podane algoritme (ne glede na to, ali vrnejo MST ali ne)?

Algoritem 21: MAYBE-MST(G,w)

```
1  MAYBE-MST-A(G, w) begin
2      sort the edges into nonincreasing order of edge weights w
3      T ← E
4      foreach edge e, taken in nonincreasing order by weight do
5          if T − {e} is a connected graph then
6              T ← T − {e}
7      return T
8  MAYBE-MST-B(G, w) begin
9      T ← ∅
10     foreach edge e, taken in arbitrary order do
11         if T ∪ {e} has no cycles then
12             T ← T ∪ {e}
13     return T
14 MAYBE-MST-C(G, w) begin
15     T ← ∅
16     foreach edge e, taken in arbitrary order do
17         T ← T ∪ {e}
18         if T has a cycle c then
19             let e′ be a maximum-weight edge on c
20             T ← T − {e′}
21     return T
```

9.6 Iskanje najkrajših poti

PONOVIMO: Na predavanjih ste omenili, da so **požrešni algoritmi** nadaljevanje dinamičnega programiranja in pri reševanju prekrivajočih podproblemov uporabijo tako hevristiko, da vedno najprej izberejo najbolj "smiselnega". Seveda moramo zdaj znati podprobleme nekako oceniti, da sploh vemo, kaj je smiselno. Na koncu se zato lahko zgodi, da dobimo suboptimalno rešitev, ampak pridobimo pa na času.

Prejšnjič smo vzeli gradnjo MVD. Oba, Primov in Kruskalov algoritem spadata med požrešne algoritme, ker predpostavljata, da lahko izkoristita izbiro najcenejše povezave za gradnjo končne rešitve. Tako se izogneta številnim kombinacijam podproblemov pri gradnji končne rešitve, npr. da bi gradili MVD kar po vrsti z vsemi povezavami in se potem vračali (*angl. backtracking*) in dopolnjevali našo rešitev.

9.6.1 Najkrajša pot od izvora do vseh točk brez negativnih razdalj (Dijkstra)

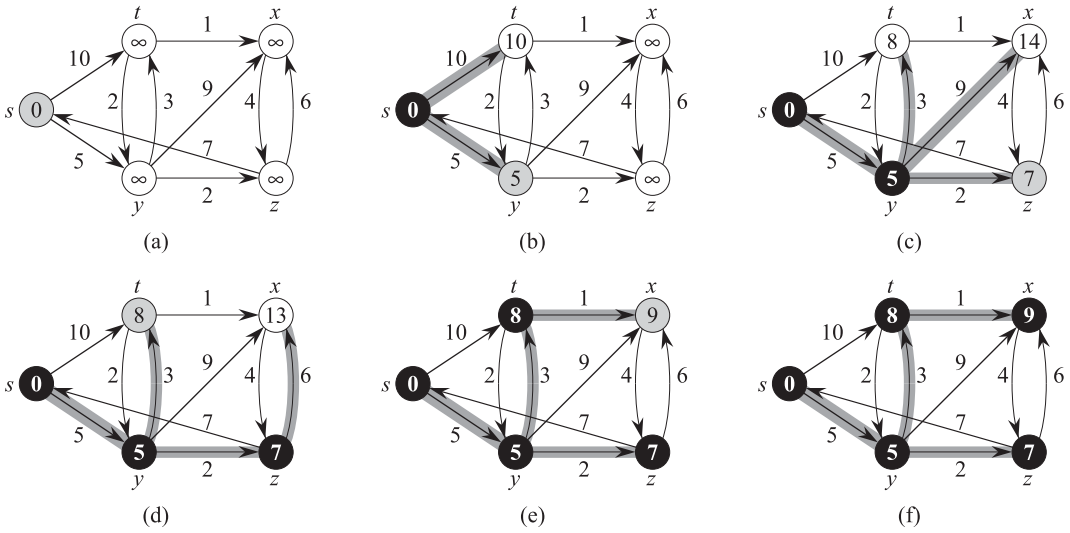
Na predavanjih ste si ogledali algoritem za iskanje najkrajših poti od enega izvora do vseh ostalih točk v grafu z nenegativnimi utežmi na povezavah (Edsger Wybe Dijkstra [Dij59], mimogrede, l. 1972 je prejel Turingovo nagrado). Algoritem spada med optimalne požrešne algoritme. Delovanje:

1. Ceno najkrajših poti vseh vozlišč `u.d` inicializiraj na ∞. Ceno izvornega vozlišča na 0.
2. Vsa vozlišča dodaj v vrsto s prednostjo, kjer je prioriteta njihova cena `u.d`.
3. Dokler vrsta s prednostjo ni prazna, ponavljaj:
   (a) Pokliči `DeleteMin`, dobimo trenutno vozlišče u.
   (b) Vsakemu sosednjemu vozlišču v trenutnega vozlišča u popravi ceno najkrajše poti, če je ta boljša od obstoječe. Nova cena je $u.d+w(u,v)$. Prioriteto popravi tudi v vrsti s prednostjo (`DecreaseKey`).

Časovna zahtevnost algoritma je naslednja: najprej vstavimo $|V|$ vozlišč, nato na vsako vozlišče izmenično kličemo nekajkrat `DecreaseKey` (skupno pokrijemo vse povezave) in enkrat `DeleteMin`. Dvojiška kopica potrebuje

$O(|V| + |E| \lg |V| + |V| \lg |V|) = O((|E| + |V|) \log |V|)$. Če uporabimo Fibonaccijevo kopico, pohitrimo vsako operacijo `DecreaseKey` na $O(1)$ in dobimo $O(|E| + |V| \log |V|)$ w.c.. Dijkstra je bila tudi sicer motivacija za zasnovo Fibonaccijeve kopice, ker imamo običajno veliko več klicev `DecreaseKey` kot `DeleteMin` in sta avtorja kopice še posebej želela pohitriti `DecreaseKey`.

Naloga Poišči vse najkrajše poti iz točke s na grafu 9.8 s pomočjo algoritma Dijkstre.



Slika 9.8: Iskanje najkrajših poti z Dijkstro.

Naloga Zakaj Dijkstrin algoritem ne deluje z negativnimi povezavami?

9.6.2 Najkrajša pot od izvora do vseh točk z negativnimi razdaljami (Bellman-Ford)

DOBRA STRAN Dijkstre: Algoritem je odporen na cikle, saj hrani že obiskana vozlišča.

Pozor Algoritem predvideva, da cene poti (prioritete v vrsti s prednostjo) naraščajo od izvornega vozlišča navzven. Kaj pa, če imamo negativne povezave? Kako je s cikli?

Negativne povezave so za prvotni algoritem problematične, saj nam omogočajo, da pridemo do že obiskanega vozlišča tudi kasneje po hitrejši poti. V nadaljevanju bomo videli, da uporabimo pojem **sproščanja** (*angl. relax*), ki preveri,

če smo našli optimalnejšo pot do poljubnega vozlišča in če ja, ceno najkrajše poti do tega vozlišča zmanjša na novo vrednost.

V čem je problem pri ciklih? Če so v ciklu vsote vseh cen poti pozitivne (pozitivni cikel), potem vračanje v novi krog po ciklu nima smisla, saj bo pot kvečjemu daljša od obstoječe. Če so v ciklu vsote vseh cen negativne (negativni cikel), potem imamo resno težavo, saj se bomo pri preiskovanju vrteli v krogu in zmanjševali ceno proti $-\infty$ (*napihovanje*).

Uporabimo Bellman-Ford-ov algoritem (glej 22) avtorjev Richarda Bellmana [Bel58] in Lesterja Forda [For56] — oba sta odkrila isto stvar — ki zna reševati tudi problem iskanja najkrajših poti z negativnimi cenami povezav, kot tudi zaznati negativne cikle.

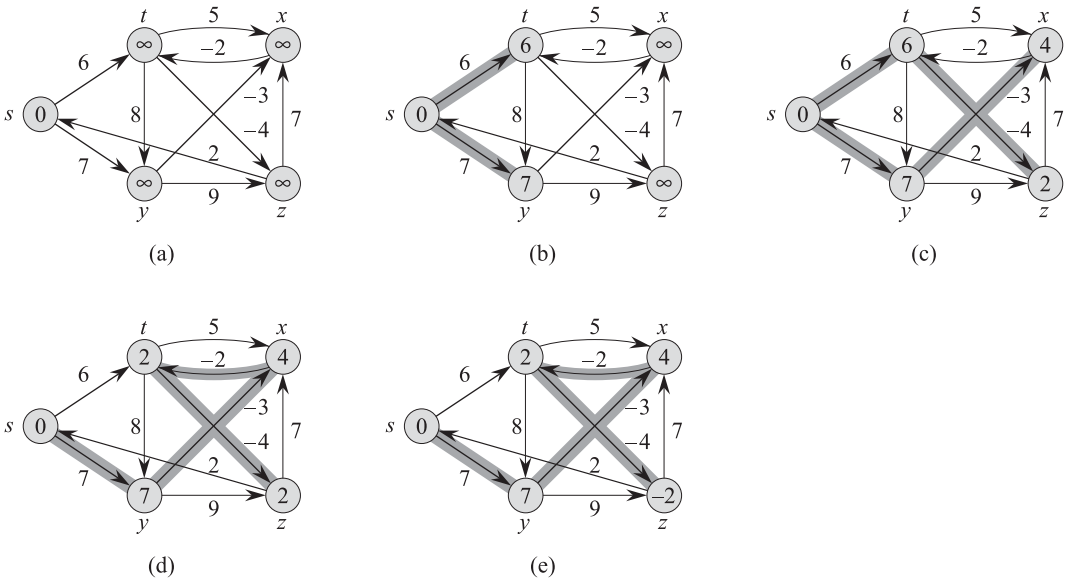
Algoritem inicializira graf na enak način, kot pri Dijkstri (cene na ∞, izvorno na 0). Nato gre v prvi fazi najprej $|G.V|$-krat skozi vse povezave in osvežuje (sprošča) cene poti do vozlišč glede na cene sosednjih vozlišč + ceno poti do vozlišča. V drugi fazi preveri, če vsebuje graf negativne cikle. Če jih vsebuje, bodo nekatera vozlišča ostala neobiskana in imela razdaljo še vedno ∞. Če jih ni, morajo biti vse poti optimalne: $v.d \leq u.d + w(u, v)$ pri usmerjeni povezavi $u \to v$.

Časovna zahtevnost algoritma je $O(|V| \cdot |E|)$ zaradi dvojne zanke v prvi fazi. Delovanje nam pove še eno stvar: Algoritem temelji le na **dinamičnem programiranju** in ne predpostavlja nobenih bližnjic pri preiskovanju prostora tako kot Dijkstra.

Naloga Poišči vse najkrajše poti iz točke s na grafu 9.9 s pomočjo algoritma Bellman-Ford.

Algoritem 22: Bellman-Fordov algoritem za iskanje najkrajših poti od podanega izvornega vozlišča v grafu z negativnimi povezavami in cikli.

```
1  BellmanFord(G, w, s):
2  begin
3      /* Inicializacija */ ;
4      foreach v ∈ G.V do
5          v.d ← ∞ ;
6          v.π ← ∅ ;
7      s.d ← 0 ;
8      for i ← 1...|G.V| − 1 do
9          foreach edge(u, v) ∈ G.E do
10             /* Sproščanje */ ;
11             if v.d > u.d + w(u, v) then
12                 v.d ← u.d + w(u, v) ;
13                 v.π ← u ;
14     foreach edge(u, v) ∈ G.E do
15         if v.d > u.d + w(u, v) then
16             return False ;
17     return True ;
```



Slika 9.9: Primer delovanja Bellman-Fordovega algoritma.

9.6.3 Najkrajša pot med vsemi pari (Floyd-Warshall)

Želimo ustvariti tabelo vseh najkrajših razdalj med mesti. Ena varianta je, da uporabimo Dijkstro oz. Bellman-Fordov algoritem nad vsemi V izvornimi vozlišči. Časovna zahtevnost postane pomnožena z $|V|$. Dobimo $O(|V|^2 \log |V| + |V||E|)$ za Dijkstro in $O(|V|^2 \cdot |E|)$ za Bellman-Forda (če upoštevamo $|E| = O(|V|^2)$ pri gostih grafih, dobimo celo $O(V^4)$).

Ali si lahko zapomnimo že izračunane rešitve in rešimo problem hitreje? Da. Uporabimo Floyd-Warshall-ov algoritem 23. Zasnovali so ga kar trije v istem času [Roy59, War62, Flo62].

Tokrat bomo naračunali matriko sosednosti vozlišč D, ki bo hranila najcenejše cene poti med vsakima vozliščema. Poleg tega bomo uporabili še eno matriko Π, ki bo za (i, j)-to povezavo hranila predhodnika vozlišča j.

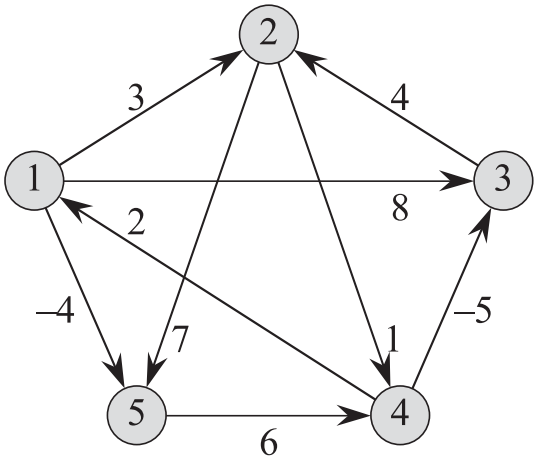
Algoritem 23: Floyd-Warshallov algoritem za iskanje najkrajših poti od vseh vozlišč do vseh ostalih v grafu z negativnimi povezavami in cikli.

```
1  FloydWarshall(G, w):
2  D matrika velikosti |G.V| × |G.V|, inicializirana na ∞ ;
3  begin
4      foreach v ∈ G.V do
5          D[v][v] ← 0
6      foreach edge(u, v) ∈ G.E do
7          D[u][v] ← w(u, v)
8      for k ← 1..|G.V| do
9          for i ← 1..|G.V| do
10             for j ← 1..|G.V| do
11                 if D[i][k] + D[k][j] < D[i][j] then
12                     D[i][j] ← D[i][k] + D[k][j] ;
13                     Π[i][j] ← k ;
14     return D, Π ;
```

Časovna zahtevnost algoritma je $\Theta(|V|^3)$. V primerjavi z Bellman-Fordom nad vsemi vozlišči smo tako hitrejši za red velikosti. V primeru, da imamo redkejšo matriko (veliko manj povezav od vozlišč) pa je še vedno najbolje izbrati Dijkstro s Fibonaccijevo vrsto s prednostjo. Floyd-Warshall temelji na **dinamičnem**

programiranju.

Naloga Poišči vse najkrajše poti od vsakega vozlišča do vseh možnih ponorov na grafu 9.10 s pomočjo algoritma Floyd-Warshall.



Slika 9.10: Primer grafa za prikaz delovanja Floyd-Warshallovega algoritma.

$$D^{(0)} = \begin{pmatrix} 0 & 3 & 8 & \infty & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & \infty & \infty \\ 2 & \infty & -5 & 0 & \infty \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \quad \Pi^{(0)} = \begin{pmatrix} \text{NIL} & 1 & 1 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 2 & 2 \\ \text{NIL} & 3 & \text{NIL} & \text{NIL} & \text{NIL} \\ 4 & \text{NIL} & 4 & \text{NIL} & \text{NIL} \\ \text{NIL} & \text{NIL} & \text{NIL} & 5 & \text{NIL} \end{pmatrix}$$

$$D^{(1)} = \begin{pmatrix} 0 & 3 & 8 & \infty & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & \infty & \infty \\ 2 & 5 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \quad \Pi^{(1)} = \begin{pmatrix} \text{NIL} & 1 & 1 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 2 & 2 \\ \text{NIL} & 3 & \text{NIL} & \text{NIL} & \text{NIL} \\ 4 & 1 & 4 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 5 & \text{NIL} \end{pmatrix}$$

$$D^{(2)} = \begin{pmatrix} 0 & 3 & 8 & 4 & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & 5 & 11 \\ 2 & 5 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \quad \Pi^{(2)} = \begin{pmatrix} \text{NIL} & 1 & 1 & 2 & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 2 & 2 \\ \text{NIL} & 3 & \text{NIL} & 2 & 2 \\ 4 & 1 & 4 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 5 & \text{NIL} \end{pmatrix}$$

$$D^{(3)} = \begin{pmatrix} 0 & 3 & 8 & 4 & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & 5 & 11 \\ 2 & -1 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \quad \Pi^{(3)} = \begin{pmatrix} \text{NIL} & 1 & 1 & 2 & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 2 & 2 \\ \text{NIL} & 3 & \text{NIL} & 2 & 2 \\ 4 & 3 & 4 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 5 & \text{NIL} \end{pmatrix}$$

$$D^{(4)} = \begin{pmatrix} 0 & 3 & -1 & 4 & -4 \\ 3 & 0 & -4 & 1 & -1 \\ 7 & 4 & 0 & 5 & 3 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & 5 & 1 & 6 & 0 \end{pmatrix} \quad \Pi^{(4)} = \begin{pmatrix} \text{NIL} & 1 & 4 & 2 & 1 \\ 4 & \text{NIL} & 4 & 2 & 1 \\ 4 & 3 & \text{NIL} & 2 & 1 \\ 4 & 3 & 4 & \text{NIL} & 1 \\ 4 & 3 & 4 & 5 & \text{NIL} \end{pmatrix}$$

$$D^{(5)} = \begin{pmatrix} 0 & 1 & -3 & 2 & -4 \\ 3 & 0 & -4 & 1 & -1 \\ 7 & 4 & 0 & 5 & 3 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & 5 & 1 & 6 & 0 \end{pmatrix} \quad \Pi^{(5)} = \begin{pmatrix} \text{NIL} & 3 & 4 & 5 & 1 \\ 4 & \text{NIL} & 4 & 2 & 1 \\ 4 & 3 & \text{NIL} & 2 & 1 \\ 4 & 3 & 4 & \text{NIL} & 1 \\ 4 & 3 & 4 & 5 & \text{NIL} \end{pmatrix}$$

Slika 9.11: Delovanje Floyd-Warshallovega algoritma nad grafom na sliki 9.10.

9.7 Največji pretok (Ford-Fulkerson, Edmonds-Karp)

Namesto dolžine ali cene lahko uteži povezav grafa $G = (V, E)$ interpretiramo kot kapacitete povezav. Zanima nas, kakšna je največja količina materiala (ali česa drugega) na časovno enoto, ki ga lahko prenašamo po takšnem omrežju od nekega izvora do ponora.

Graf ima usmerjene povezave, pri čemer ne dovolimo nasprotnih tokov med poljubnim parom vozlišč – torej ne dovolimo povezav (v_1, v_2) in (v_2, v_1). Če imamo par nasprotnih tokov, na enem od njiju ustvarimo novo vozlišče v', tako da imamo potem namesto povezave (v_1, v_2) povezavi (v_1, v') in (v', v_2) z isto kapaciteto kot (v_1, v_2).

Vzemimo tok f v grafu $G = (V, E)$ in poglejmo povezavo (u, v), preko katere teče tok f. Koliko lahko še povečamo tok preko povezave (u, v)? Povečamo ga lahko za največ za preostanek kapacitete $c_f(u, v) = c(u, v) - f(u, v)$. Preostanke kapacitet predstavimo v grafu preostankov $G_f = (V, E_f)$, $E_f = \{(u, v) \in V \times V : c_f(u, v) > 0\}$. Če je $c_f(u, v) > 0$ (povezav s kapaciteto 0 ni v grafu) dodamo še povezavo med vozliščema u in v v obratni smeri z vrednostjo $c_f(v, u) = f(u, v)$ (tako lahko pošiljamo tok v obratni smeri, oziroma izničimo tok preko povezave (u, v)).

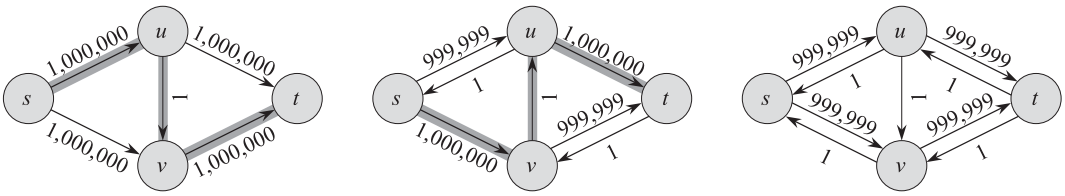
Največji pretok lahko izračunamo s Ford-Fulkersonovim algoritmom [For56] (glej algoritem 24).

Če so kapacitete povezav cela števila, je vsako povečanje pretoka $|f| \geq 1$. Če je največji pretok $|f^*|$, potem potrebujemo $\leq |f^*|$ iteracij povečave pretoka. V vsaki iteraciji iščemo pot od izvora do ponora npr. z iskanjem v globino ali širino po omrežju preostankov tokov, kar je $O(|V| + |E'|) = O(|E|)$. Časovna zahtevnost algoritma je tako $O(|E| \cdot |f^*|)$. V najslabšem primeru je največji pretok precej velik in vsaka iteracija poveča pretok za zgolj 1 enoto (slika 9.12).

V primeru, da pretok povečujemo le za 1 enoto, se nam splača, da pot povečanja pretoka od izvora do ponora poiščemo kot najkrajšo pot z iskanjem v širino po še prostih povezavah, pri čemer ima vsaka povezava utež 1; dobimo Edmonds-Karpov algoritem [EK72]. Časovna zahtevnost algoritma je $O(|V| \cdot |E|^2)$.

Algoritem 24: Ford-Fulkersonov algoritem za iskanje največjega pretoka.

```
1  Ford-Fulkerson(G, s, t):
2  begin
3      foreach edge (u, v) ∈ G.E do
4          (u, v).f ← 0 ;
5      while there exists a path p from s to t in the residual network G_f do
6          c_f(p) = min{c_f(u, v) : (u, v) ∈ p} ;
7          foreach edge (u, v) ∈ p do
8              if (u, v) ∈ G.E then
9                  (u, v).f ← (u, v).f + c_f(p) ;
10             else
11                 (v, u).f ← (v, u).f − c_f(p) ;
```



Slika 9.12: [CLRS09] str. 728: V najslabšem primeru je največji pretok precej velik (v konkretnem primeru je $|f^*| = 2.000.000$) in vsaka iteracija Ford-Fulkersonovega algoritma poveča pretok za zgolj 1 enoto.

Naloga [CLRS09] 26.2-3: Za podan graf s pomočjo Edmonds-Karpovega algoritma poiščite največji tok po omrežju od izvora s do ponora t.

Poglavje 10

Aproksimacijski algoritmi

10.1 Iskanje najdaljših poti

Glej [SW11], stran 661-662. Nato od strani 910 dalje za obrnjen Bellman-Fordov algoritem za reševanje istega problema (če ni ciklov), končno algoritem na strani 911, ki je NP-poln.

Definicija P, NP (najkrajša pot = P, najdaljša pot = NP), razlika med odločitvenimi in optimizacijskimi problemi ter pretvorba optimizacijskega v odločitvenega.

Ali je P = NP? Še vedno odprto vprašanje.

Aproksimacijski algoritmi za razliko od deterministčnih namesto optimalne rešitve, izračunajo približno rešitev, ki je v praksi "dovolj dobra". Genetski algoritmi so algoritmi, ki uporabijo tri metode: selekcijo, križanje in mutacijo.

10.2 0/1-nahrbtnik

Naloga Imamo 0/1-nahrbtnik s štirimi predmeti, podana je cena in teža posameznega predmeta.

$c = (20, 10, 20, 35)$, $w = (15, 10, 20, 30)$, $W_{max} = 40$.

Poiščite kombinacijo predmetov, ki jih še lahko spravimo v nahrb-

tnik (brez rezanja predmetov).

Požrešen pristop vzame razmerje c/w, uredi padajoče in vstavlja v nahrbtnik, dokler lahko. Zagotovi 2-kratno ceno optimalne.

Druga možnost: genetski algoritem za 0/1-nahrbtnik, npr. začnemo z geni 1100, 1010, 1111, kjer 0 pomeni, da predmeta ni v nahrbtniku, 1 pa, da je. Zamislimo si ustrezno ocenitveno funkcijo (*fitness function*) za selekcijo, delamo križanje, mutacijo, čez nekaj iteracij se približamo optimalni rešitvi.

Poglavje 11

Priprava na 2. kolokvij in izpit

11.0.1 Trie

Ponovi LC-trie, poglavje 8.4.

11.0.2 Grafi - sprehod v širino

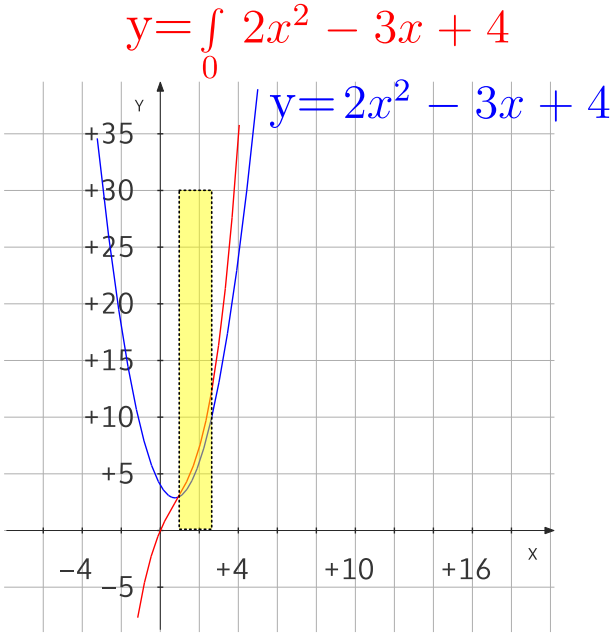
2. kolokvij 2013/2014, naloga 3.

11.0.3 Grafi - SSSP, APSP

2. kolokvij 2013/2014, naloga 4.

11.0.4 Integriranje s pomočjo metode Monte Carlo

Podano imamo funkcijo $y = 2x^2 - 3x + 4$. Empirično izračunajte $\int_1^{2,5} 2x^2 - 3x + 4 dx$ določen integral na območju $a = 1$ in $b = 2{,}5$ s pomočjo metode Monte-Carlo.


y= ∫ 2x² − 3x + 4
0
y= 2x² − 3x + 4
Y
+35
+30
+25
+20
+15
+10
+5
−4
−5
+4
+10
+16
X

Literatura

[AL62] Georgy Adelson-Velskii and Evgenii Landis. An algorithm for the organization of information. In *Proceedings of the USSR Academy of Sciences*, volume 146, pages 263–266, Moscow, USSR, 1962. Russian Academy of Sciences.

[AN93] Arne Andersson and Stefan Nilsson. Improved behaviour of tries by adaptive branching. *Information Processing Letters*, 46(6):295–300, July 1993.

[BCD+99] Andrej Brodnik, Svante Carlsson, Erik D. Demaine, J. Ian Munro, and Robert Sedgewick. Resizable Arrays in Optimal Time and Space. In *WADS '99 Proceedings of the 6th International Workshop on Algorithms and Data Structures*, pages 37–48. Springer-Verlag, August 1999.

[BDFC00] M. A. Bender, E. D. Demaine, and M. Farach-Colton. Cache-oblivious B-trees. In *FOCS '00: Proceedings of the 41st Annual Symposium on Foundations of Computer Science*, page 399. IEEE Computer Society, November 2000.

[Bel58] Richard Bellman. On a Routing Problem. *Quarterly of Applied Mathematics*, 16:87–90, 1958.

[BM72] R. Bayer and E.M. McCreight. Organization and maintenance of large ordered indexes. *Acta informatica*, 1(3):173–189, 1972.

[CLRS09] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Third Edition*. The MIT Press, 3rd edition, 2009.

[De 59] Rene De La Briandais. File searching using variable length keys. In *ACM Western joint computer conference*, pages 295–298, New York, USA, 1959. ACM Press.

[Dij59] E W Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1(1):269–271, 1959.

[EK72] Jack Edmonds and Richard M. Karp. Theoretical Improvements in Algorithmic Efficiency for Network Flow Problems. *Journal of the ACM*, 19(2):248–264, April 1972.

[Flo62] Robert W. Floyd. Algorithm 97: Shortest path. *Communications of the ACM*, 5(6):345, June 1962.

[Flo64] R.W. Floyd. Algorithm 245: Treesort. *Communications of the ACM*, 7(12):701, 1964.

[For56] Lester R Ford. Network Flow Theory. Report P-923, The Rand Corporation, 1956.

[Fre60] E. Fredkin. Trie memory. *Communications of the ACM*, 3(9):490–499, 1960.

[FT87] Michael L. Fredman and Robert Endre Tarjan. Fibonacci heaps and their uses in improved network optimization algorithms. *Journal of the ACM*, 34(3):596–615, July 1987.

[Kru56] Joseph B Kruskal. On the shortest spanning subtree of a graph and the traveling salesman problem. *Proceedings of the American Mathematical Society*, 7(1):48–50, 1956.

[MM90] Udi Manber and Gene Myers. Suffix arrays: a new method for on-line string searches. In *Proceedings of the first annual ACM-SIAM symposium on Discrete algorithms*, pages 319–327. Society for Industrial and Applied Mathematics, January 1990.

[Mor68] Donald R. Morrison. PATRICIA - Practical Algorithm To Retrieve Information Coded in Alphanumeric. *Journal of the ACM*, 15(4):514–534, 1968.

[MS04] Dinesh P. Mehta and Sartaj Sahni. *Handbook Of Data Structures And Applications (Chapman & Hall/Crc Computer and Information Science Series.)*. Chapman & Hall/CRC, October 2004.

[Pug90] William Pugh. Skip lists: a probabilistic alternative to balanced trees. *Communications of the ACM*, 33(6):668–676, June 1990.

[Roy59] Bernard Roy. Transitivité et connexité. *C. R. Acad. Sci. Paris*, 249:216–218, 1959.

[SW11] Robert Sedgewick and Kevin Wayne. *Algorithms*. Addison-Wesley Professional, 4th ed. edition, 2011.

[Tar85] R.E. Tarjan. Amortized computational complexity. *SIAM Journal on Algebraic and Discrete Methods*, 6(2):306–318, 1985.

[Vui78] Jean Vuillemin. A data structure for manipulating priority queues. *Communications of the ACM*, 21(4):309–315, 1978.

[War62] Stephen Warshall. A Theorem on Boolean Matrices. *Journal of the ACM*, 9(1):11–12, January 1962.

[Wil64] J.W.J. Williams. Algorithm 232: heapsort. *Communications of the ACM*, 7(6):347–348, 1964.