UNIVERZA V LJUBLJANI
FAKULTETA ZA RAČUNALNIŠTVO IN INFORMATIKO

Algoritmi in podatkovne strukture 2
Visokošolski strokovni program
(zapiski vaj)

februar 2018

Asistent: Matevž Jekovec

Nosilec predmeta: dr. Andrej Brodnik

Poglavje 1

Formalnosti vaj

Vaje so z letošnjim letom prvič avditorne. Potekajo v avditornih učilnicah, saj se večino težav mogoče rešiti brez računalnikov oz. s prenosnikom na licu mesta. Računalnikov na vajah, razen prvi teden, ki je namenjen spoznavanju okolja za programerske domače naloge, ne bomo potrebovali.

1.1 Domače naloge in zapiski

Vseh domačih nalog je 6 in bodo objavljene na učilnici. Prispevajo 30% h končni oceni predmeta. Vsaka domača naloga je sestavljena iz treh teoretičnih nalog in ene programerske.

Teoretične naloge se ocenjujejo na zagovoru in sicer mora vsak študent zagovoriti eno teoretično nalogo v semestru. Termin in nalogo si sami izberete ter se napišite na wiki na učilnico. Zagovori bodo potekali med vajami in sicer dva študenta na nalogo, tj. 6 zagovorov na vaje. Teoretično nalogo oddajate tudi na učilnico v zapisu PDF (uporabite LaTeX ali kaj podobnega!). To ni obvezno (razen za nalogo, ki jo zagovarjate), vam pa močno priporočamo, da sproti rešujete teoretične naloge z namenom utrjevanja snovi, v primeru, da ste med oceno, pa lahko s profesorjem za nazaj oceniva vašo oddajo, ki vam pomaga pri zviševanju ocene.

Programerske naloge se ocenjujejo samodejno. Pri vsaki morate doseči vsaj 20%. 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. Pričakuje se, da osnove programiranja že imate.

1.2 Marmoset

Programčke za domače naloge se oddaja s pomočjo kanadskega 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 prenesite ogrodje (*skeleton*). Nato sprogramirate manjkajoče funkcije v razredu, ki je del ogrodja. V ogrodju naloge je podano nekaj javnih testov (*public tests*), ki testirajo vaš programček (JUnit). Ko nalogo oddate, se bodo pognali tudi končni testni primeri (*release tests*). Študent bo v nekaj sekundah 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 0. programersko domačo nalogo. Prijava v sistem je na `https://marmoset.fri.uni-lj.si`. Up. ime in geslo sta enaka kot za učilnico. Prenesite ogrodje (*skeleton*) za nalogo 0 (Reverse string, find maximum), sprogramirajte funkciji reverseString(String str) in findMax(int[] arr) v `src/java/aps2/reversestringfindmax/ReverseStringFindMax.java`, zazipajte in oddajte (*submit*). Kmalu boste videli rezultate testov (*view*).

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

Poglavje 2

Slovar

2.1 Dvojiška drevesa

V splošnem je drevo definirano kot **povezan neusmerjen graf brez ciklov** + 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 iščemo elemente od vozlišča proti listom.

Definicija **Strukturna invarianta dreves**: 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*).

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. Ali štejemo število vozlišč (vključno s korenom) ali število povezav zavisi od definicije.

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

Levo/Desno poravnano drevo (*angl. left/right balanced tree*) je drevo, ki ima na vsaki plasti vsa vozlišča na levi/desni strani drevesa.

Celovito drevo (*angl. complete tree*) je uravnoteženo, levo poravnano drevo.

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

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

Polno drevo (angl. *full tree* ali *strictly binary tree*) je drevo, kjer imajo vsa vozlišča natanko 0 ali k vozlišč. Ni pa nujno, da je to drevo uravnoteženo.

Popolno drevo (angl. *perfect tree*) stopnje k in višine h je polno drevo, ki ima vse liste na natanko istem nivoju. 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.

Drevo bomo uporabili bodisi kot množico ali pa kot slovar. Operacije so torej `Insert()`, `Find()` in `Delete()`.

2.2 Dvojiško iskalno drevo

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

Definicija **Vsebinska invarianta dvojiškega iskalnega drevesa**: V levem poddrevesu vozlišča so vsi elementi manjši, v desnem poddrevesu vozlišča pa vsi elementi večji od ključa, predstavljenega v vozlišču.

2.2.1 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.

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.

2.2.2 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 $\lceil \lg n \rceil$.

2.3 Sprehodi po dvojiškem iskalnem drevesu

Sprehodi po drevesu Poznamo 3 vrste **sprehodov v globino (*depth-first traversals*)** 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.

Pri izvedbi zgornjih treh sprehodov potrebujemo sklad.

Poleg tega imamo še **sprehod v širino (*breadth-first traversal*)**. Velikokrat ga imenujemo tudi **po plasteh (*lavel traversal*)**. Za izvedbo sprehoda v širino potrebujemo vrsto FIFO.

Pozor Sprehode v globino in širino bomo srečali tudi v poglavju z grafi. Princip bo identičen.

Izrek Pri neurejenih drevesih potrebujemo kombinacijo katerih koli dveh sprehodov v globino, da rekonstruiramo drevo. Pri urejenih pa le

pre-order ali post-order. In-order premalo pove.

2.4 Drevo AVL [AVL62]

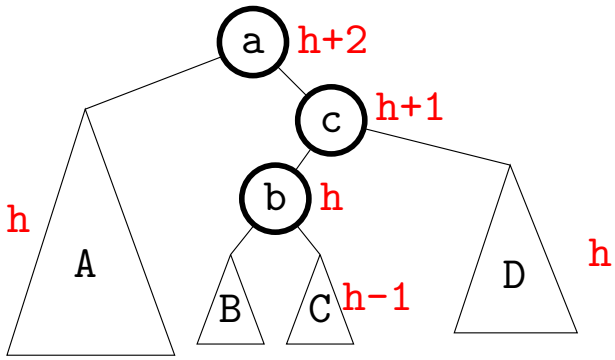
Drevo AVL avtorjev Georgija Adelson-Velskija in Evgenija Landisa je bilo predstavljeno leta 1962 v znanstveni reviji Sovjetska Matematika [AVL62]. Je **delno poravnano** urejeno dvojiško drevo.

Definicija **Ravnotežna invarianta drevesa AVL**: Za vsako vozlišče se višini obeh poddreves razlikujeta največ za 1.

2.4.1 Operacije

`Find` deluje identično kot pri dvojiškem iskalnem drevesu.

Poglejmo bolj natančno operacijo `Insert`. Slika 2.1 prikazuje obliko drevesa AVL (druga oblika je simetrično obrnjena).



Slika 2.1: Primer drevesa AVL. Z rdečo so dopisane višine posameznih poddreves.

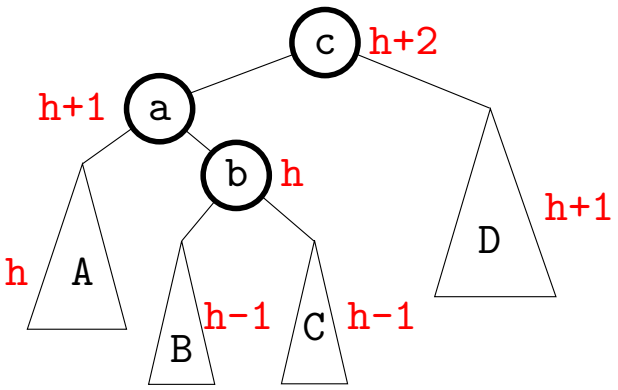
Ker gre za iskalno drevo, zaradi vsebinske invariante velja:

$$A < a < B < b < C < c < D \tag{2.1}$$

Element, podobno kot pri dvojiškem iskalnem drevesu vstavimo v liste. 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 in invarianta ne drži več? Potrebno je popraviti drevo (uravnotežiti), tako da bo invarianta spet držala.

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



Slika 2.2: 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 je identični tisti iz dvojiških iskalnih dreves, le da na koncu funkcije pokličemo še `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)
```

2.4.2 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.

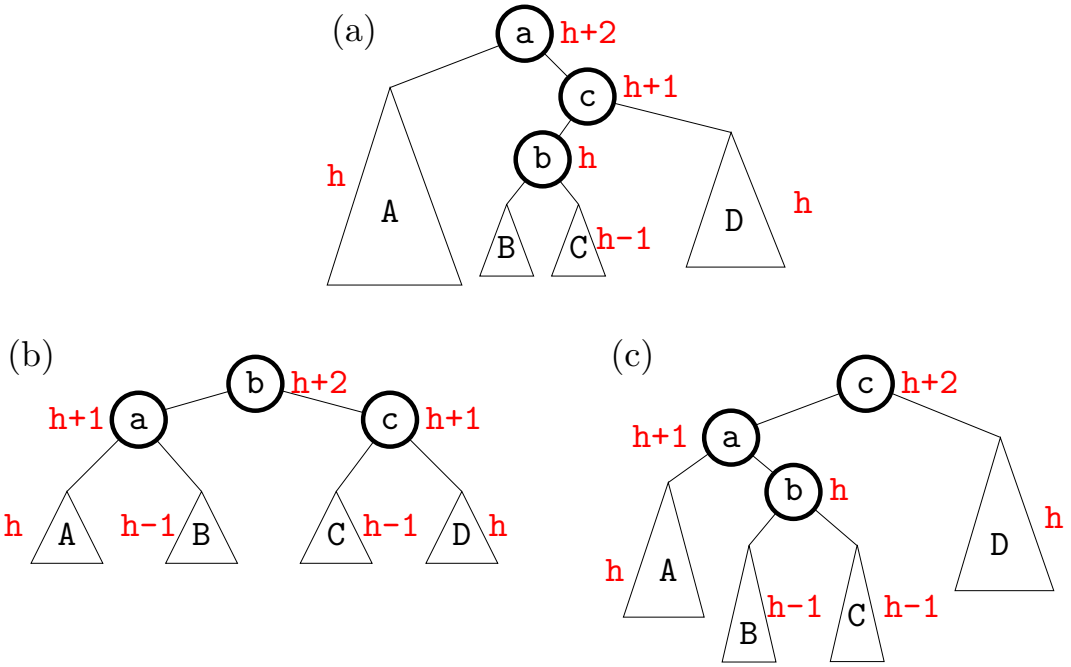
Operaciji `Insert` in `Delete` najprej potrebujeta $\Theta(\lg n)$ primerjav, da najdeta iskan element. Nato za uravnoteženje nekaj pisanj ($O(1)$). Koliko uravnoteženj pa se lahko sploh pokliče na eno vstavljanje ali brisanje elementa? Pri vstavljanju natanko 1, pri brisanju pa $\Theta(\lg n)$ (glej 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 2.1. Enkrat je a, enkrat b in enkrat c koren drevesa (glej sliko 2.3). 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 2.3: Tri možne postavitve prvotnega drevesa 2.1. (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.

2.5 B-drevo [BM72]

B-drevesa so uravnotežena iskalna drevesa, namenjena učinkovitemu iskanju podatkov, shranjenih na sekundarnem pomnilniku (trdi disk). Definirala 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 indeksov podatkovnih baz (tam se uporablja posebna vrsta B^+ dreves, kjer je ključ primarni ključ, vrednost pa vsebina vrstice).

2.5.1 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.

B-drevo ima drugačno strukturno invarianto kot dvojiška drevesa:

Definicija **Strukturna invarianta B-drevesa**: Vsako notranje vozlišče ima vsaj $\lceil b/2 \rceil$ otrok in največ b otrok razen korena, ki ima lahko le 2 otroka.

Definicija **Vsebinska invarianta B-drevesa**: Vsako notranje vozlišče ima vsaj $\lceil b/2 \rceil - 1$ in največ $b - 1$ ključev razen korena, ki ima lahko le 1 ključ. Ključi so urejeni po velikosti. Nasledniki vozlišča levo so manjši, desno pa večji od ključa.

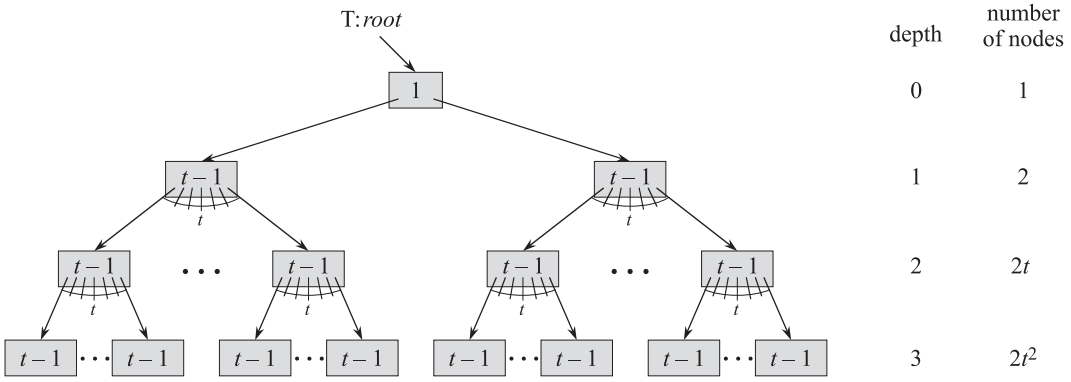
Definicija **Ravnotežna invarianta B-drevesa**: Vsi listi so na istem nivoju.

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)
```

Vidimo, da večji kot je red b, manjša je višina h, s tem pa operacije hitrejše.

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



Slika 2.4: Višina B-drevesa, $t = \lceil b/2 \rceil$

2.5.2 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

Element k vstavimo v list podobno kot pri dvojiškem iskalnem drevesu. Ko je list poln, ga razdelimo na dve vozlišči, mediano pa porinemo v starša. Da se izognemo rekurzivnemu cepljenju vozlišč in dvojnemu sprehodu (najprej dol, da poiščemo ustrezen list, potem pa v najslabšem primeru do korena, da razcepimo vozlišča), uporabimo optimizacijo: **predčasno cepimo vozlišča z b otroki**. Tak način uporablja algoritem 3.

Formalno:

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.

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) ;
```

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 $2 \log_b n + 1$.

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 $O(\lg n)$.

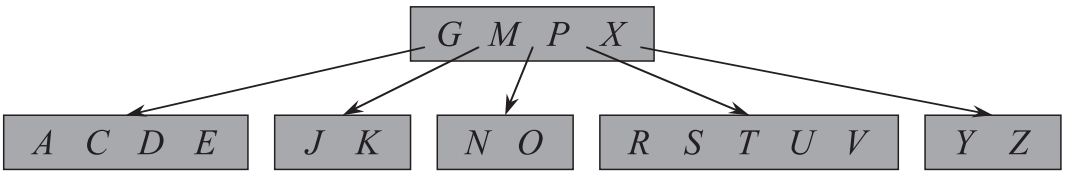
Naloga Za izhodišče vzemimo drevo na sliki 2.5. Črke predstavljajo številske vrednosti od 1 do 26 in ustrezajo vrstnemu redu v angleški abecedi. V drevo po vrsti vstavite elemente B, Q, L in F. Narišite drevo po vsakem vstavljanju.

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) ;
```



Slika 2.5: 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.

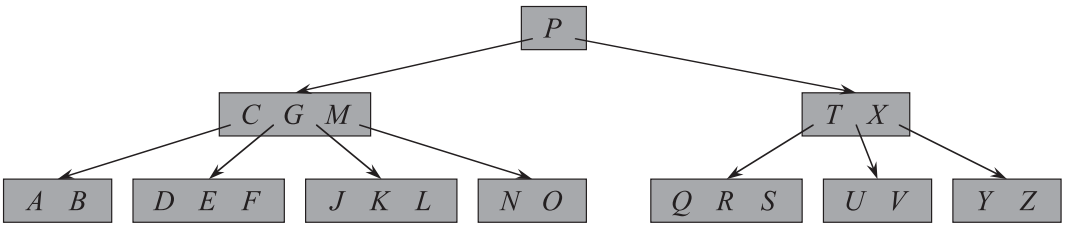
Pri brisanju se lahko zgodi, da ima vozlišče manj kot $\lceil b/2 \rceil$ otrok in kršimo strukturno invarianto. V tem primeru moramo drevo preoblikovati.

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 2.6: Začetno B-drevo pred brisanjem.

Naloga Za izhodišče vzemimo drevo na sliki 2.6. 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.

2.5.3 B-drevo kot slovar

Zaradi lažje razlage pri iskanju, vstavljanju in brisanju elementov smo uporabili B-drevo kot množico. Če ga želimo uporabiti kot slovar, imamo dve možnosti:

1. Lahko hranimo par (ključ,vrednost) v istem vozlišču,
2. lahko pa v vozliščih hranimo le ključe, vrednosti pa v listih.

V drugem primeru govorimo o **B$^+$-drevesih**.

2.5.4 2-3-4 drevesa

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

Kadar je $b = 4$, ima vsako notranje vozlišče 2, 3 ali 4 otroke oz. 1, 2 ali 3 ključe. Govorimo o 2-3-4 drevesih (angl. *Two-Three-Four tree*).

2.5.5 Rdeče-Črno drevo

Ima naslednjo ravnotežno invarianto:

- Vozlišče je bodisi rdeče, bodisi črno.
- Vsi listi morajo biti črni.
- Vsako rdeče vozlišče pa mora imeti le črne otroke.
- Na kateri koli poti od korena do katera koli lista mora biti enako število črnih vozlišč.

Vsako 2-3-4 drevo enostavno pretvorimo v dvojiško iskalno drevo. Vozlišče s 3 otroki spremenimo v dve notranji vozlišči in tri liste. Vozlišče s 4 otroki spremenimo v tri notranja vozlišča in 4 liste. Novo nastala vozlišča obarvamo rdeče.

2.5.6 Pomnilniško nezavedno dvojiško iskalno drevo

Z idejo čim manjšega števila zgrešitev v predpomnilniku so se veliko ukvarjali. Leta 1999 so Frigo, Leiserson, Prokop, Ramachandran definirali *predpomnilniško nezavedno dvojiško iskalno drevo* [FLPR99] (angl. *cache-oblivious binary search tree*). Razmišljali so v smeri, ali sta sprehod v globino in sprehod po plasteh res edina načina, kako implicitno zapisati podatkovno strukturo v polje. Prišli so do naslednjega zapisa: najprej zgornja polovica drevesa, potem pa njegova spodnja poddrevesa od leve proti desni. Zgornje poddrevo in njegova poddrevesa rekurzivno zapisujemo, dokler ne pridemo do praznih poddreves. Posledica takega zapisa je učinkovita V/I zahtevnost pri iskanju elementa — enaka kot pri B-drevesu. Le da pri Frigo et al rešitvi ne potrebujemo parametra b.

2.6 Implicitni zapis drevesa

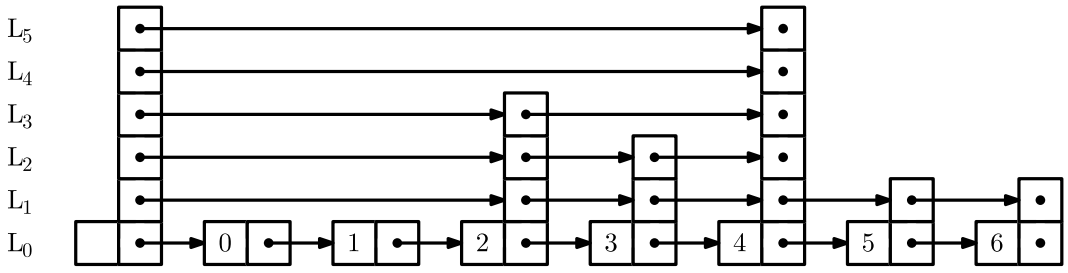
Za domačo nalogo morate zapisati Huffmanovo drevo v polje. Gre za levo-poravnano dvojiško drevo. Za vsako plast imate podano število vozlišč brez otrok in, ali obstaja mejno vozlišče z enim otrokom. Definirati morate funkciji child(i, j) in parent(i). Prva vrne levega oz. desnega otroka vozlišča i, druga pa starša vozlišča i. Pri APS1 ste delali že nekaj podobnega, le da ste imeli tam popolno drevo. Premislite, zakaj točno ste pri APS1 zapisali childL $= 2 * i$ oz. childR $= 2 * i + 1$. Zakaj ravno $*2$? Kaj pa, če je na koncu plasti luknja in vozlišča manjkajo?

2.7 Preskočni seznam [Pug90]

2.7.1 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 2.7: 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 $\mathtt{x} \in L_r$. 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.

2.7.2 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 2.7. 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 2.7 – vozlišče najdemo takoj po pregledu naslednika glave. Postopek iskanja opisuje algoritem 6.

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.

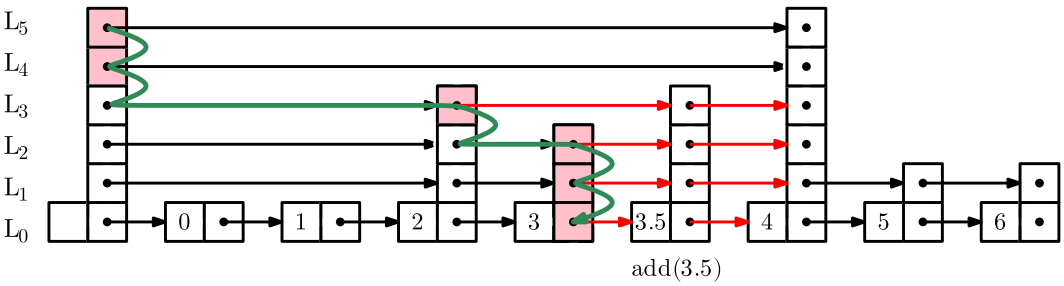
Metodo `insert(x)` implementiramo tako, da v preskočnem seznamu poiščemo vozlišče, ki vsebuje x (v tem primeru ne spreminjamo preskočnega seznama in vrnemo `false`) oz. vozlišče, katerega naslednik vsebuje najmanjši ključ že večji od x. *Zato smo pri implementaciji iskanja najprej pogledali naslednika 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

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 */
```

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 2.8.



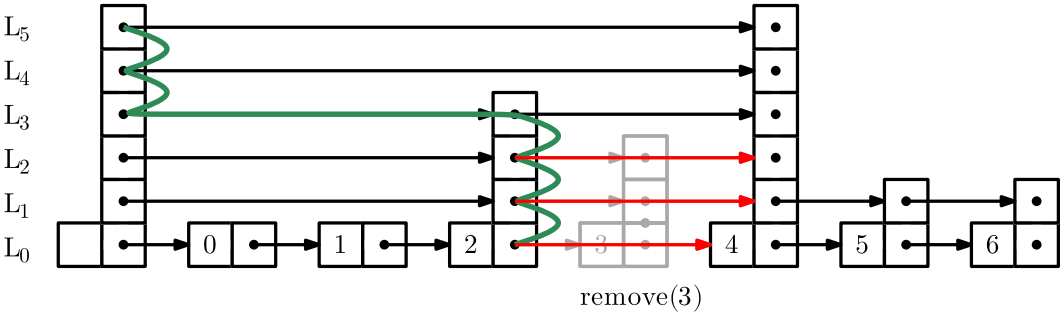
Slika 2.8: Vstavljanje ključa 3.5 v preskočni seznam.

Brisanje. To operacijo izvedemo podobno kot vstavljanje, le da ni potrebe po skladu, ki beleži pot iskanja, saj lahko povezave preusmerjamo sproti. Ko se

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 ;
```

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 2.9.



Slika 2.9: Brisanje vozlišča s ključem 3 iz preskočnega seznama.

Naloga Nad preskočnim seznamom na sliki 2.7 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.

2.7.3 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.

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 ;
```

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.

2.8 Razpršena tabela [MS04, str. 9–15]

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.

2.8.1 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.

2.8.2 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?

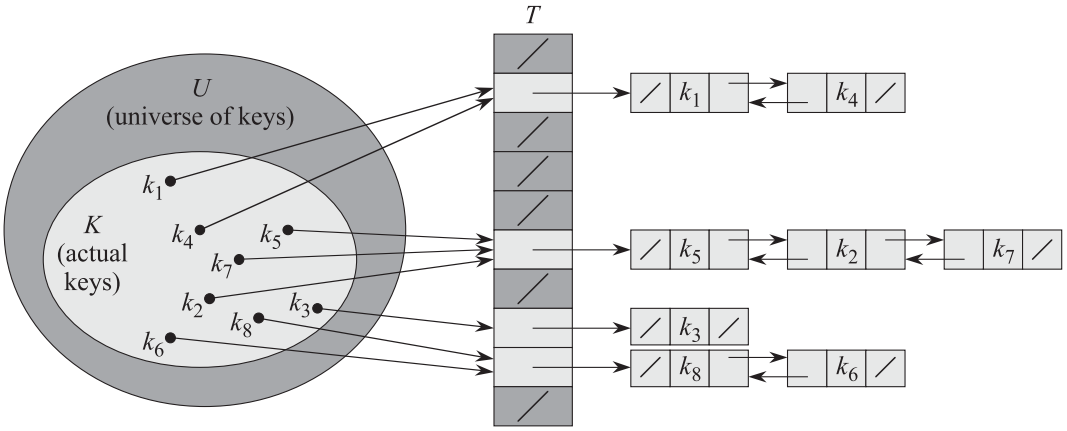
2.8.3 Izrazoslovje

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*.

2.8.4 Razpršena tabela z veriženjem

Razpršena tabela z veriženjem elemente z enako zgoščeno vrednostjo poveže v povezan seznam zunaj tabele. Slika 2.10 prikazuje delovanje razpršene tabele z veriženjem.



Slika 2.10: 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.

Operacije

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.

Analiza

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

Časovna zahtevnost v najslabšem primeru je $O(n)$ — vsi elementi se preslikajo v isto vrednost. Namesto povezanega seznama uporabimo kaj hitrejšega. npr. eno izmed iskalnih dreves. Dobimo **kombinirano podatkovno strukturo**.

2.8.5 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).

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**.

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.

Naloga V razpršeno tabelo z odprtim naslavljanjem velikosti $m = 7$ in zgoščevalno funkcijo $h(k) = (k \mod m) + 3i \mod m$ izvedi:

- Insert(12)
- Insert(14)
- Insert(16)
- Insert(18)
- Insert(19)
- Insert(26)
- Delete(12)
- Find(18)
- Find(26)

Analiza

Pričakovana časovna zahtevnost je vedno $O(1)$. V najslabšem primeru je časovna zahtevnost m.

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

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

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?

Definicija 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 (*rehashing*).

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 iskan element.

2.9 Bloomov filter

Bloomov filter je podatkovna struktura, ki ima definirani dve operaciji: `Insert(`x`)`, ki vstavi ključ x v Bloomov filter in `MaybeContains(`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 `MaybeContains(`x`)` vrne False, ključ zagotovo ne obstaja v tabeli.
- Če `MaybeContains(`x`)` vrne True, ključ *lahko* obstaja v tabeli. Odgovor je torej 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. $h(x) = (\textit{vzemi lihe bite dvojiškega zapisa števila } x) \mod m$
2. $g(x) = (\textit{vzemi sode bite dvojiškega zapisa števila } x) \mod m$

Narišite Bloomov filter po vstavljanju ključev 38, 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?

Pozor Bloomov filter zaradi optimizacije prostora nikoli ne hrani ključev elementov. Za razliko od razpršenih tabel, kjer so vedno shranjeni tudi ključi in zato vedno lahko ugotovimo, ali smo našli pravi ključ, četudi smo imeli kolizije na poti.

Poglavje 3

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

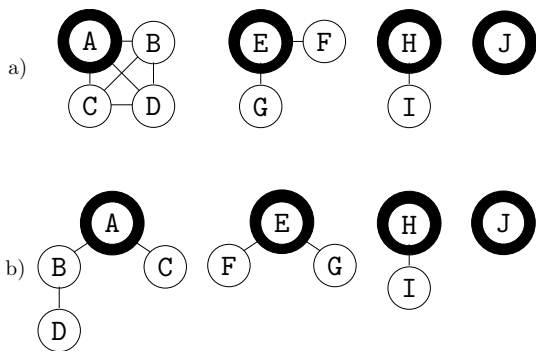
na predavanjih).

Algoritem 9: 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 $|V|, |E|, k$?

3.1 Izvedba z drevesom in stiskanje poti



Slika 3.1: (a) izvorni graf s poudarjenimi predstavniki (b) disjunktne množice za izvorni graf, implementirane s pomočjo dreves.

Vsaka množica je predstavljena kot drevo s korenom, ki je predstavnik množice (glej sliko 3.1). Operacija `Find-Set` pleza od vozlišča navzgor, dokler ne doseže korena drevesa. Da so drevesa uravnotežena, uvedemo **unijo glede 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 Z uteženo unijo in s stiskanjem poti izvedi naslednje operacije:

Algoritem 10: 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_4)
8  Union(x_3, x_6)
9  Union(x_3, x_13)
10 Find-Set(x_16)
```

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?

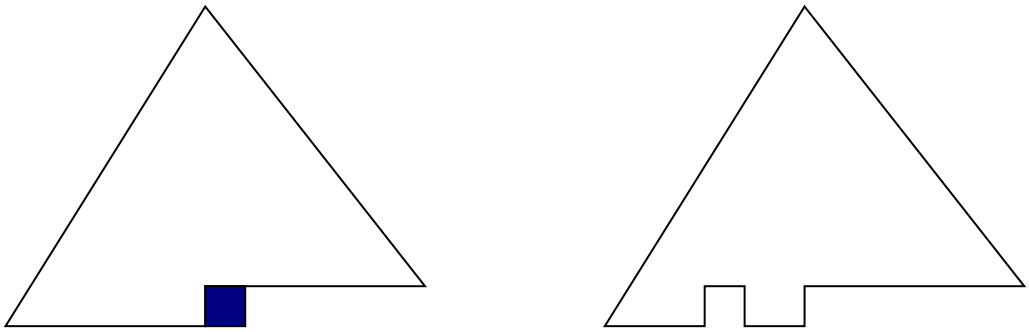
4.1.1 Operacije

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 11 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 12.

Algoritem 11: 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 12: 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 [Vui78]

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.

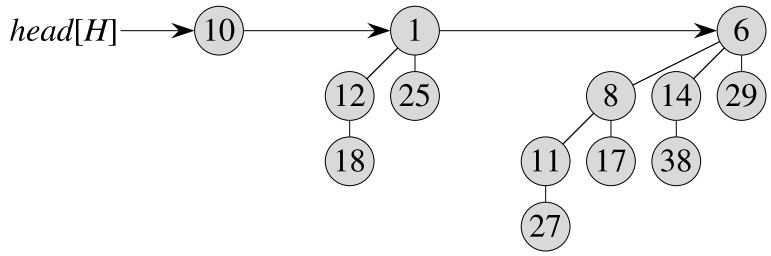
Zanimiva je gradnja binomskega drevesa. B_1 zgradimo z zlivanjem dveh B_0 dreves. To pa tako, da drevo z manjšim elementom postane novi koren. Na enak način lahko zgradimo binomsko drevo poljubne stopnje. Vedno zlivamo binomski drevesi **istih stopenj v binomsko drevo ene stopnje više**. Drevo z večjim 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.

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.

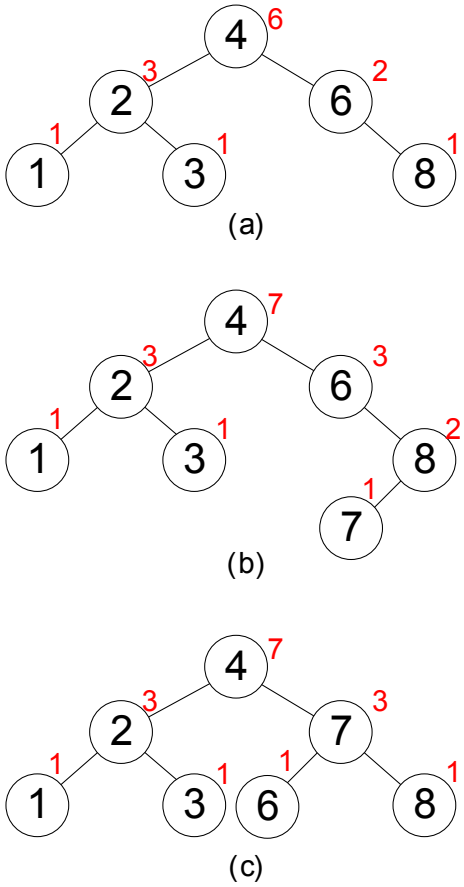
Pozor Za operacijo `Rank` bomo privzeli, da že imamo referenco na element in ga ni potrebno iskati v strukturi.

5.1.1 Rešitev z drevesom

Elemente shranjujemo v dvojiško iskalno drevo (glej poglavje 2.2). 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.



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

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 — $\Theta(h)$.

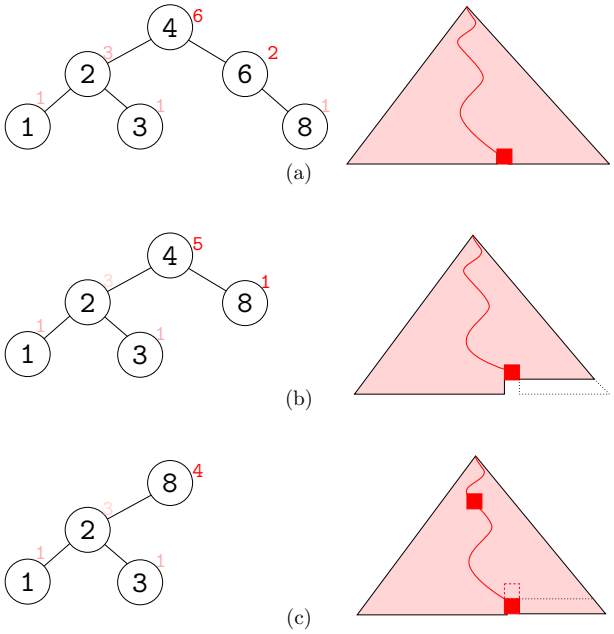
Brisanje

Spomnimo, pri brisanju elementa lahko nastanejo tri situacije (glej poglavje 2.2.1). 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.

5.1.2 AVL

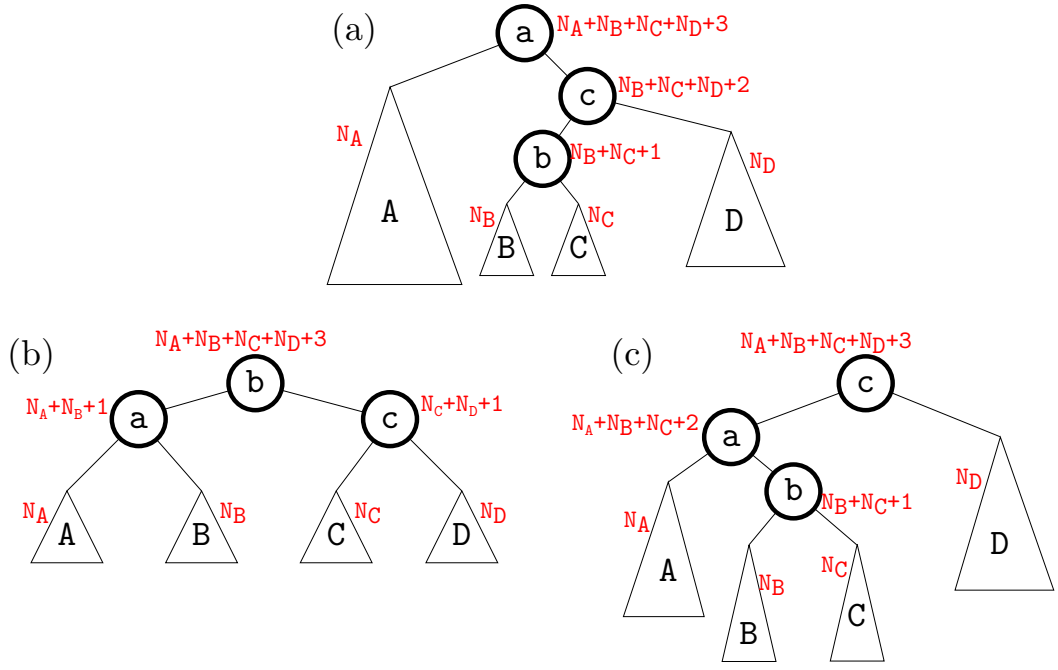
Do sedaj smo imeli le operacije nad dvojiškim drevesom. Kaj pa, če imamo drevo AVL?

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).

Poglavje 6

Dinamično programiranje

6.1 Uvod

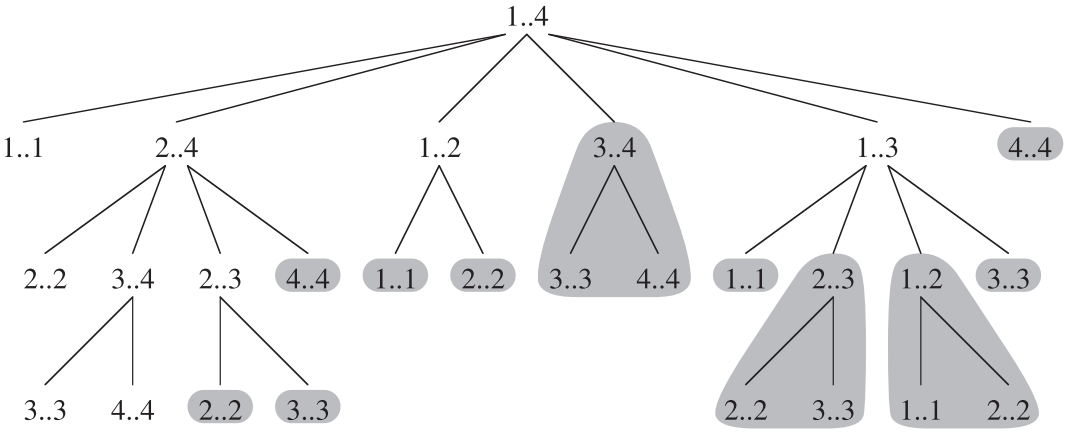
Dinamično programiranje je način reševanja problemov. Sodobni pojem je vpeljal ameriški matematik Richard Bellman (Bellman-Fordov algoritem za iskanje najkrajših poti v grafu, tudi ko so uteži negativne) leta 1953. Beseda *programiranje* se ne nanaša na računalniško programiranje (npr. zasedanje in sproščanje 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 6.1 za primer iskanja zaporedja množenja matrik).



Slika 6.1: Primer več enakih rekurzivnih klicev za enake podprobleme. Z memoizacijo sive klice nadomestimo s konstantnim vpogledom v tabelo.

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

[CLRS09, Poglavje 15.1]

Na voljo imamo jeklene palice, ki jih želimo razrezati tako, da bomo maksimizirali dobiček. Prodajne cene posameznih palic so v spodnji tabeli.

dolžina i	1	2	3	4	5	6	7	8	9	10
cena p_i	1	5	8	9	10	17	17	20	24	30
dobiček r_i										
razrez s_i										

Dobiček je rekurzivno definiran kot $r_i = max(p_i, r_1 + r_{i-1}, r_2 + r_{i-2}, ..., r_{i-1} + r_i)$.

6.3 Točkovna matrika

Imamo dva niza, za katere želimo izračunati podobnosti. Za osnovo bomo uporabili točkovno matriko (angl. *dot matrix*). Na levo stran napišemo prvi niz, na vrh napišemo drugi niz. Točkovna matrika vsebuje točke tam, kjer se znaka ujemata oz. prazno polje, če se ne.

6.3.1 Najdaljši skupni podniz (LCSubstr)

Rekurzivna formula za dolžino najdaljšega skupnega podniza x in y je:

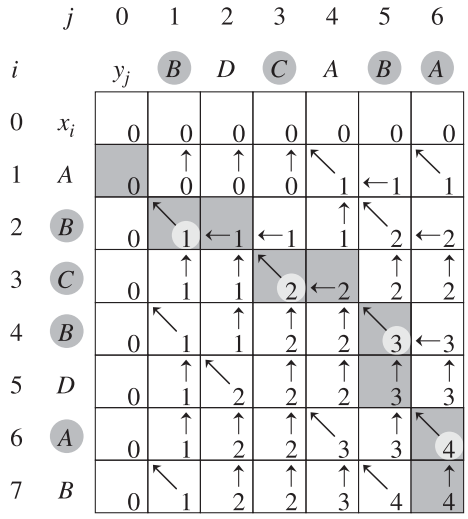
- $LCSuff(x_{1..p}, y_{1..q}) = LCSuff(x_{1..p-1}, y_{1..q-1}) + 1$, če $x_p = y_q$
- 0 sicer

6.3.2 Najdaljše skupno podzaporedje (LCSubseq)

Glej [CLRS09, poglavje 15.4].

Rekurzivna formula za dolžino najdaljšega skupnega podzaporedja nizov x in y je:

- $c[i, j] = 0$ za $i = 0, j = 0$
- $c[i, j] = c[i - i, j - 1] + 1$, če $i, j > 0$ in $x_i = y_j$
- $c[i, j] = max(c[i, j - 1], c[i - 1, j])$, če $i, j > 0$ in $x_i \neq y_j$



Slika 6.2: Najdaljše skupno podzaporedje besed ABCBDAB in BDCABA.

Poglavje 7

Delo z nizi

7.1 *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. Podobno jih lahko urejeno vstavimo v dvojiško iskalno drevo, kjer so ključi nizi. 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.

7.1.1 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!

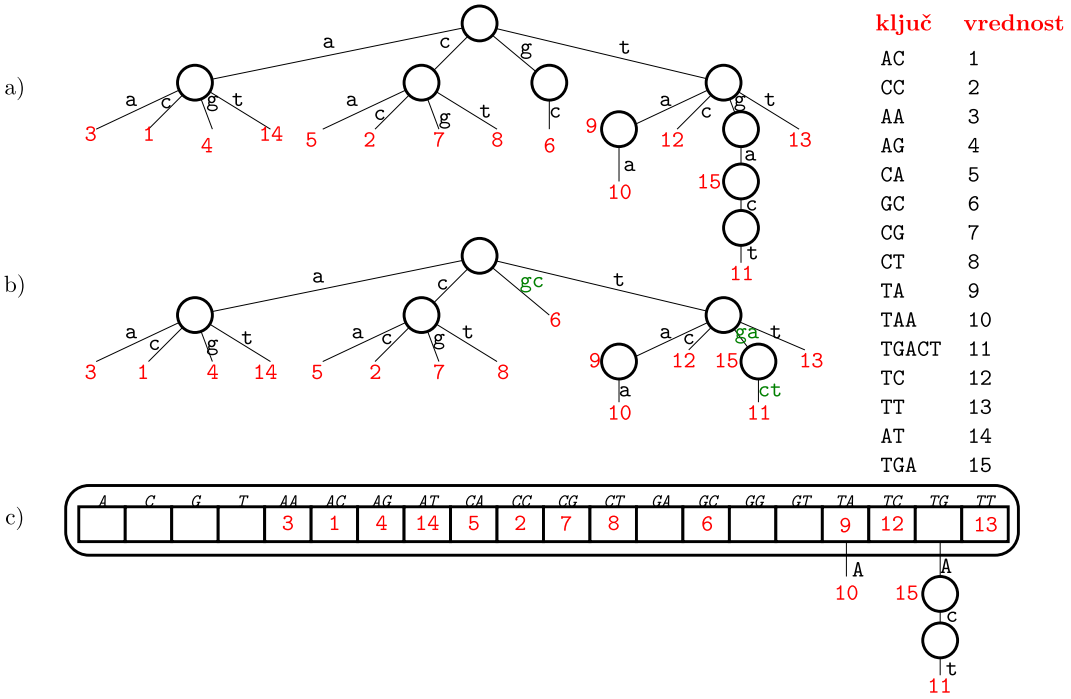
7.1.2 Analiza

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 $O(nm)$.

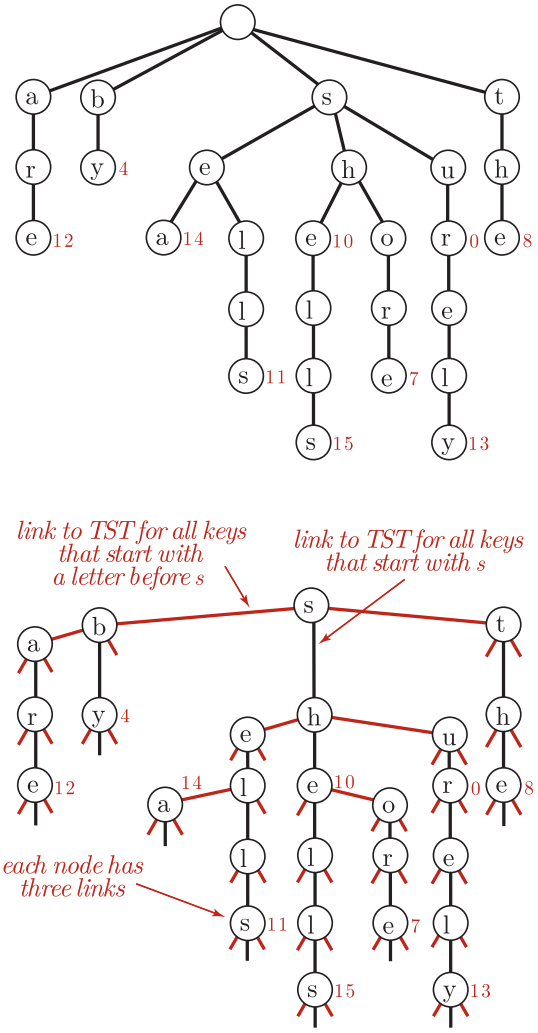
7.1.3 Zapis vozlišča

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



Slika 7.1: a) številsko drevo (*trie*), b) stiskanje poti (*PATRICIA*), c) stiskanje plasti (*LC trie*)

- 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 7.2.



TST representation of a trie

Slika 7.2: Trojiško iskalno drevo.

7.2 *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*.

7.3 *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 7.1.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 7.1.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 $O(1)$.

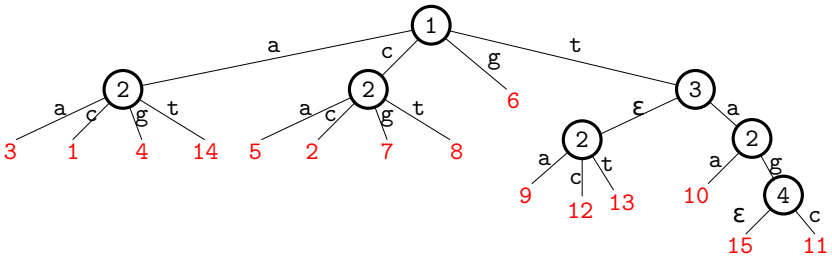
7.4 Številsko drevo z nezaporednimi indeksi

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 7.3: Številsko drevo z nezaporednimi indeksi.

7.5 Iskanje podnizov v besedilu

Problem: Imamo podano besedilo dolžine n in vzorec dolžine m. Želimo najti mesta pojavitev vzorcev po celotnem besedilu.

Naivna metoda z oknom je trivialna. Preverjamo znak po znak in premikamo okno po eno polje skozi celotno besedilo. Časovna zahtevnost je $O(nm)$.

Dva učinkovitejša pristopa sta:

- Predprocesiramo besedilo in zgradimo kazalo, po katerem iščemo (npr. priponsko drevo, priponsko polje)
- Predprocesiramo vzorec in zgradimo končni avtomat (npr. Knuth-Morris-Pratt, Boyer-Moore).

7.5.1 Priponsko drevo (*Suffix tree*) [McC76]

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.

7.5.2 Priponsko polje (*Suffix array*) [MM90]

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 sosednjih pripon v SA. Če izračunamo obe polji — LCP in SA, lahko iščemo po priponskem polju v $O(m + \lg n)$ primerjavah besed. Uporabimo algoritem za iskanje najmanjšega elementa v podanem intervalu Range-Minimum-Query (RMQ), ki potrebuje $O(1)$ časa. S pomočjo tega zagotovimo, da ne primerjamo večkrat iskani niz s predponami, ampak pregledamo vsak znak kvečjemu enkrat. Preostanek se sprehajamo po LCP in z bisekcijo zmanjšujemo interval za RMQ.

Dodatno lahko zmanjšamo čas iskanja na $O(m)$ [AKO04] in dosežemo isto asimptotično hitrost kot pri priponskem polju.

7.5.3 Knuth-Morris-Pratt [KMP77]

Povzeto po [SW11], str. 762.

Primer besedila: `ABAABABABAC`

Vzorec: `ABABAC`

Položimo okno na prvi znak besedila. Ugotovimo, da je v besedilu na poziciji 4 znak A, v oknu pa B. Ideja: Namesto, da premaknemo okno za 1 polje naprej, ga premaknemo za 3 polja naprej. Ideja je, da ne pregledujemo istih znakov večkrat. Pozor: Ne smemo pa premakniti okna dlje kot za 3 znake, ker bi v tem primeru eno možno pojavitev lahko izpustili!

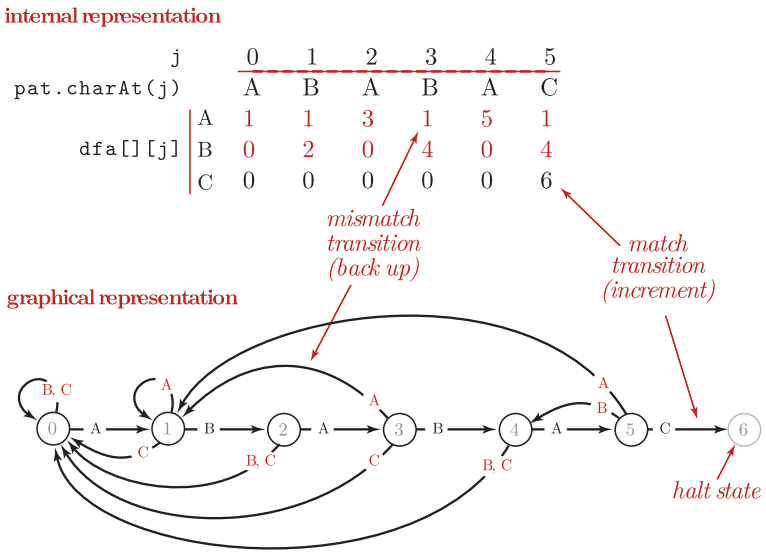
Postopek

Da vemo, koliko moramo preskočiti besedilo, vnaprej najprej obdelamo vzorec. Zgradimo *deterministični končni avtomat* in ga shranimo v 2-dimenzionalno polje dfa[][]. Vrednost dfa[c][i] bo pomenila, da če naletimo na i-tem mestu vzorca na znak c v besedilu, v katero novo stanje preidemo (oz. se vrnemo nazaj).

Primer končnega avtomata in njegove predstavitve v polju je na sliki 7.4. Številke v polju + 1 pomenijo premik okna v desno.

Gradnja

Pri gradnji KMP-DKA moramo imeti ves čas v mislih že prebran (pod)niz, v primeru, da back-trackamo. Zato pri gradnji hranimo kazalec X. Ta kaže na



Slika 7.4: Deterministični končni avtomat za vzorec ABABAC.

stanje, na katerega backtrackamo, če pride popolnoma tuj znak.

Algoritem 13: Algoritem KMP

```
1 Vhod: vzorec P dolžine m, abeceda Σ
2 Izhod: deterministični končni avtomat v 2-d polju dfa
3 dfa[P.charAt(0)][0] ← 1
4 for X ← 0, i ← 1; i<m; i++ do
5     for c ← 0; c<|Σ|; c++ do
6         dfa[c][i] ← dfa[c][X]
7     dfa[P.charAt(i)][i] ← i+1
8     X ← dfa[P.charAt(i)][X]
```

Časovna zahtevnost gradnje je $O(m|\Sigma|)$, časovna zahtevnost iskanja vzorca v besedilu pa $O(n)$ v najslabšem primeru.

Nedeterministični končni avtomat

Poleg DKA, ki smo ga ustvarili s pomočjo algoritma KMP, obstaja tudi NKA. V splošnem je razlika med deterističnim in nedeterističnim končnim avtomatom ta, da ima nedeterministični več izhodov iz stanja za isti znak. Tako ne vemo točno, v katero novo stanje bo avtomat šel. Ko ga simuliramo,

gremo zato v vsa možna stanja. Hramba aktivnih stanj in simulacija običajno potrebuje eksponenten čas in prostor! Na osebnih računalnikih (Turingovih strojih) so za večje avtomate posledično uporabni le DKA, medtem ko za simulacijo NKA potrebujemo npr. kvantne računalnike.

Trivialen NKA za poljuben vzorec je za vsako črko vzorca po ena povezava naprej (isto kot DKA), iz vsakega stanja pa dodamo povezavo po vseh znakih v začetno stanje. Na ta način dovolimo, da se NKA kadar koli vmes resetira in začne preiskovati okno od začetka.

7.5.4 Boyer-Moore [BM77]

Povzeto po [SW11], str. 770.

Če se lahko po besedilu in vzorcu sprehajamo tudi nazaj, je enostavnejši Boyer-Moorov algoritem. Ta še vedno premika okno z leve proti desni, vendar znotraj okna primerja znake z desne proti levi.

Slika 7.5: Boyer-Moore: Skoki ob neujemanju znakov.

Iskanje

Položimo okno na besedilo in preverjamo znake vzorca z desne proti levi. Če se znaka besedila in vzorca ujemata, nadaljujemo s pregledovanjem okna. Če pridemo vse do začetka okna, je bilo iskanje uspešno in vzorec se v besedilu nahaja na mestu, kjer se trenutno nahaja okno. Če se znaka v vzorcu in besedilu na katerem koli mestu ne ujemata:

- Če je znak v besedilu tuj kateremu koli znaku vzorca, premakni začetek okna na mesto po koncu trenutnega okna.
- Če znak v besedilu ni tuj vzorcu, premakni okno desno za toliko, da bo znak besedila poravnan s prvo pojavitvijo znaka v vzorcu z desne. Prve

pojavitve znakov predhodno naračunamo in jih stranimo v preskočno tabelo `right`.

Ko okno premaknemo, začnemo ponovno pregledovati celotno okno od desne proti levi.

Pričakovana časovna zahtevnost iskanja vzorca je $O(n/m)$. V najslabšem primeru $O(nm)$. Zvi Galil je pokazal, da z majhno izboljšavo najslabši primer iskanja lahko omejimo na $O(n)$.

Gradnja

Podobno kot pri KMP tudi tu zgradimo preskočno tabelo (*skip table*). Ta je 1-d, za vsak znak abecede po ena vrednost. Glej algoritem 7.5.4.

```
1 public class BoyerMoore
2 {
3 private int[] right;
4 private String pat;
5 BoyerMoore(String pat)
6 { // Compute skip table.
7 this.pat = pat;
8 int M = pat.length();
9 int R = 256;
10 right = new int[R];
11 for (int c = 0; c < R; c++)
12 right[c] = -1;             // -1 for chars not in pattern
13 for (int j = 0; j < M; j++)       // rightmost position for
14 right[pat.charAt(j)] = j; // chars in pattern
15 }
16
17 public int search(String txt)
18 { // Search for pattern in txt.
19 int N = txt.length();
20 int M = pat.length();
21 int skip;
22 for (int i = 0; i <= N-M; i += skip)
23 { // Does the pattern match the text at position i ?
24 skip = 0;
25 for (int j = M-1; j >= 0; j--)
26 if (pat.charAt(j) != txt.charAt(i+j))
27 {
28 skip = j - right[txt.charAt(i+j)];
29 if (skip < 1) skip = 1;
30 break;
31 }
32 if (skip == 0) return i; // found.
33 }
34 return N; // not found.
35 }
36
37 public static void main(String[] args) {
38 String pat = args[0];
39 String txt = args[1];
40 KMP bm = new BoyerMoore(pat);
41 StdOut.println("text:    " + txt);
42 int offset = bm.search(txt);
43 StdOut.print("pattern: ");
44 for (int i=0; i<offset; i++)
45 StdOut.print(" ");
46 StdOut.println(pat);
47 }
```

```
48 }
```

Preskočno tabelo zgradimo v času $O(m + |\Sigma|)$.

Naloga Zgradi preskočno tabelo za vzorec `NEEDLE`. Poišči jo v besedilu `FINDINAHAYSTACKNEEDLEINA`.

7.6 Burrows-Wheelerjeva preslikava [BW94]

Burrows-Wheelerjeva preslikava je obojesmerna preslikava. Uporablja se kot predkorak pri kompresiji besedil. Intuitivno BWT izpostavi kombinacije parov znakov, ki se ponavljajo. Če se par večkrat ponovi, bo desni znak para sosed v preslikanem nizu BWT, kar bo tvorilo verigo istih znakov. Ti se potem učinkovito zakompresirajo (npr. kodiranje z dolžino niza ali angl. *run-length encoding, RLE*). Ko želimo besedilo odkompresirati, dodatno izvedemo še inverzno preslikavo BWT, da dobimo izvorni niz.

7.6.1 Preslikava

Preslikavo naredimo z naslednjimi koraki:

1. Leksikografsko uredimo vse pripone besedila.
2. Pripone zapišemo v tabelo, vsaka pripona svoja vrstica, vsak stolpec en znak pripone. Ko se pripona zaključi, jo ciklično nadaljujemo od začetka, tako da so vse vrstice dolge n znakov.
3. Vzamemo zadnje stolpce tabele od zgoraj navzdol. Dobimo niz, preslikan z BWT.

Naloga Naredite preslikavo BWT besedila `T=ABRAKADABRA$`.

7.6.2 Inverzna preslikava

Za inverzno preslikavo uporabimo podobno tabelo kot pri preslikavi. n-krat ponovimo naslednje korake:

1. Kot prvi stolpec vrinemo rezultat preslikave BWT.
2. Leksikografsko uredimo nize v tabeli.

Izvoren niz lahko poiščemo kot vrstico, ki se konča z $.

Naloga Naredite inverzno preslikavo BWT za `ARD$KRAAAABB`.

7.7 Gradnja priponskega drevesa v linearnem času [Ukk95]

Ukkonenov algoritem v lineranem času zgradi priponsko drevo glede na podano besedilo. Še več, algoritem v enem samem prehodu od leve proti desni zgradi priponsko drevo, zato je algoritem primeren za besedila, ki še niso dokončana (angl. *online algorithm*).

Deluje v dveh stanjih. V prvem sledi obstoječim povezavam v drevesu glede na prebran znak besedila in polne medpomnilnik B. V vsakem trenutku se algoritem nahaja nekje v drevesu, ki sovpada s trenutno vsebino medpomnilnika. Temu vozlišču pravimo *aktivno vozlišče*, če pa je nekje na sredini povezave, pa povezavi *aktivna povezava*. Ko se pojavi v besedilu znak, ki še ni vsebovan, algoritem preklopi v drugo stanje, tj. praznjenje medpomnilnika:

1. Trenutno aktivno povezavo zlomimo in vstavimo novo notranje vozlišče.
2. Pripnemo obstoječega otroka.
3. Pripnemo novo vozlišče — list – ki sovpada z zadnjim neujemoajočim znakom.
4. Iz medpomnilnika B odstranimo prvi (najstarejši) znak.
5. Skočimo na ustrezno novo aktivno vozlišče oz. aktivno povezavo glede na nov B.

6. Preverimo, če se na novi poziciji v drevesu tuj znak sedaj nahaja. Če se, preklopimo algoritem v prvo stanje. Če se ne, ponovimo rekurzivno ponovimo korak 1 tega postopka.

Ko pridemo do konca besedila, vstavimo še zaključni \$, ki poskrbi, da se medpomnilnik popolnoma sprazne in se drevo dokonča.

Da algoritem zares teče v lineranem času, sta pomembna dva detajla. Vsakič ko preberemo nov znak (ne glede na stanje algoritma), pripnemo nov znak vsem $O(n)$ listom. Ker znaki niso zares shranjeni eksplicitno v povezavah, interpretiramo pripone na listih, da vedno trajajo do konca besedila — kjer koli to že je v danem trenutku. Ko premaknemo povečamo kazalec na konec besedila za en znak, se tako vsem listom podaljša pripona za en znak.

Drug detajl je v 5. koraku zgornjega postopka. V splošnem ta zahteva $O(n)$ časa, da poišče ustrezno novo vozlišče. Notranja vozlišča zato opremimo s *priponskimi povezavami* (angl. *suffix link*). Te povezujejo vozlišča, ki predstavljajo niz cX, z vozliščem X. Se pravi, ko odstranimo najstarejši znak (c) iz medpomnilnika, v času $O(1)$ sledimo priponski povezavi in pristanemo v pravem novem vozlišču. Priponske povezave vozliščem nastavljamo v konstantnem času v 1. koraku postopka tekom gradnje.

Naloga Za besedilo `T=ABRAKADABRA$` zgradite priponsko drevo z Ukkonenovim algoritmom.

Poglavje 8

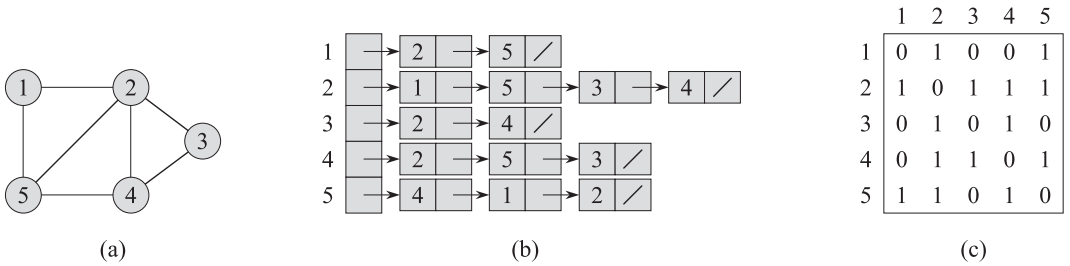
Algoritmi na grafih

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

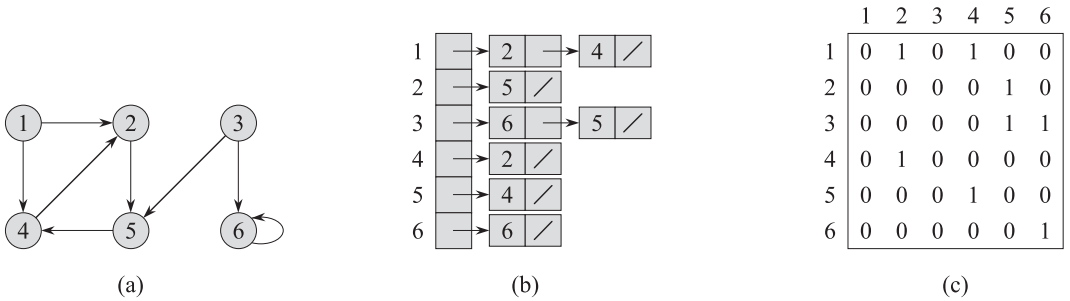
Graf je v računalništvu abstraktna podatkovna struktura, s katero predstavimo matematični pojem grafa. Le-ta sestoji iz množice vozlišč (node) oz. **točk** (*vertex*) in množice **povezav** (*edge*) med točkami. Prvo množico ponavadi označimo z V, slednjo pa z E, graf je potem $G = (V, E)$. V splošnem 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 8.1 (neusmerjen graf) in 8.2 (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),



Slika 8.1: [CLRS09, str. 590]: Predstavitev (a) neusmerjenega grafa in zapis s seznamom sosedov (b) in matriko sosedov (c).



Slika 8.2: [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].

8.1 Preiskovanje grafov — ponovitev APS1

8.1.1 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*).

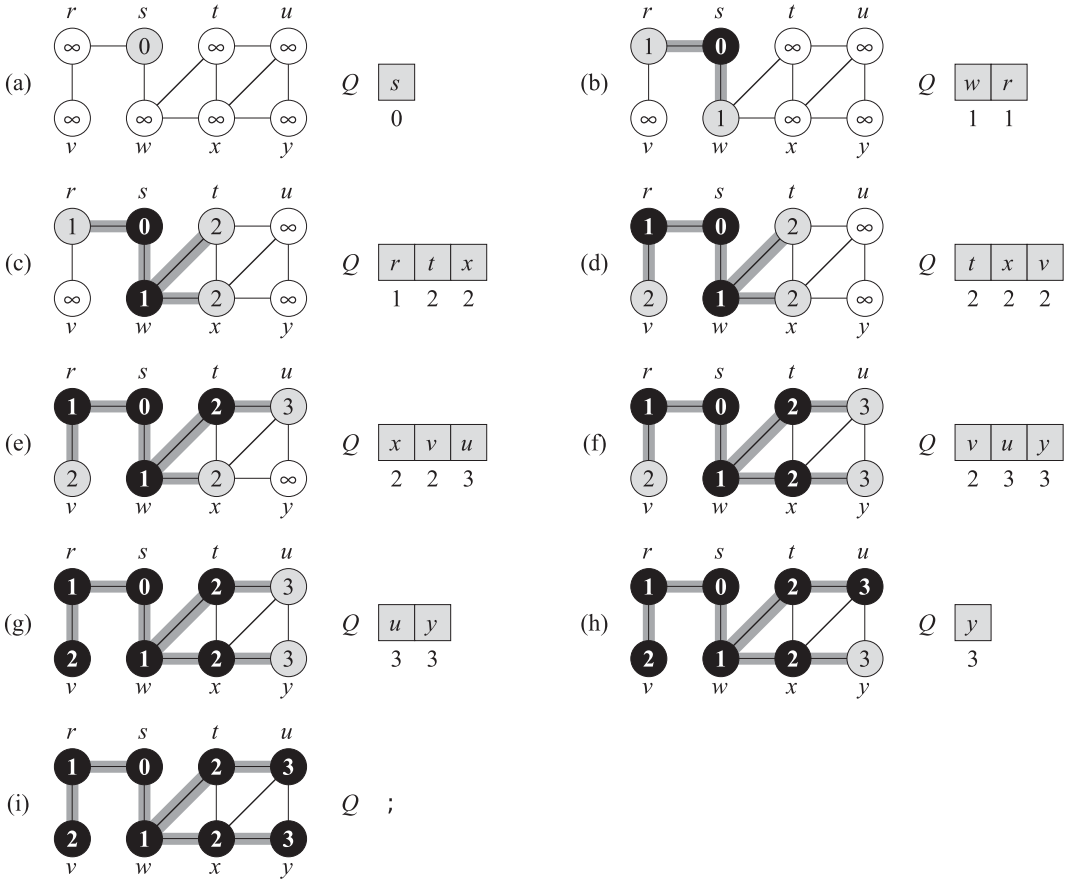
Preiskovanje v širino (BFS)

Imejmo graf $G = (V, E)$ ter **izvorno točko** (*source vertex*) s. BFS (glej algoritem 14) 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 8.3.

Algoritem 14: 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 8.3: [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 $O(V)$, for-zanka 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 (15) prikazuje slika 8.4. 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 15: Algoritem preiskovanja v globino.

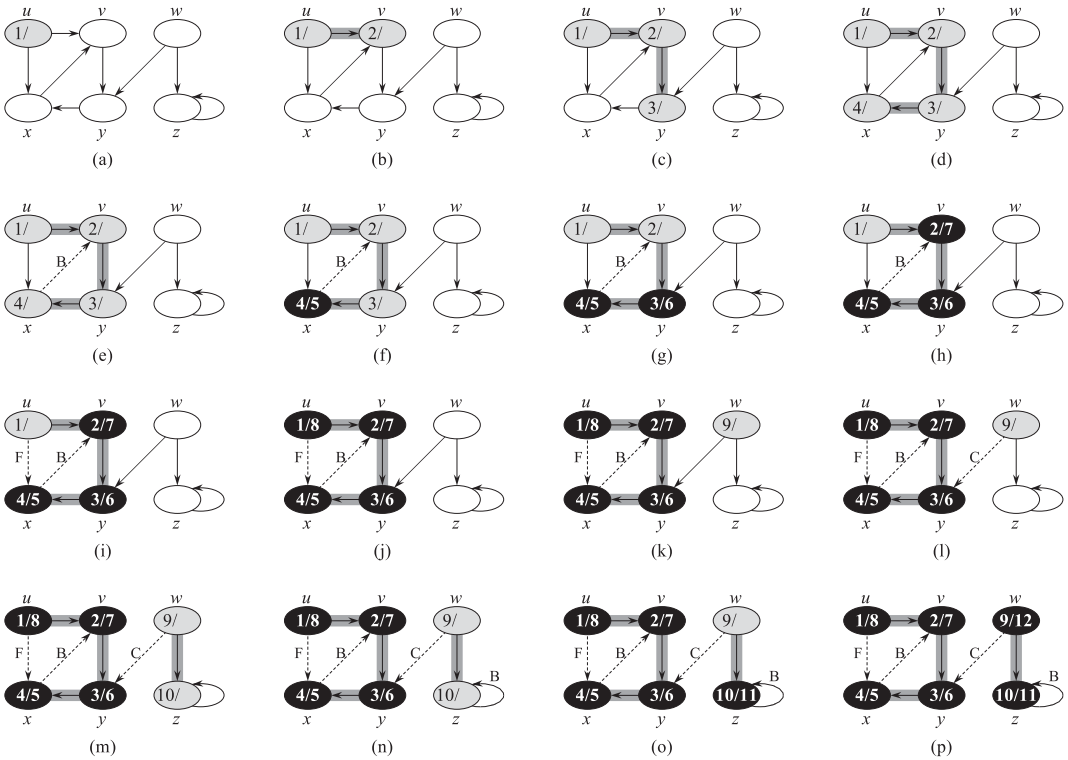
```
1  function DFS(G)
2      foreach vertex u ∈ G.V do
3          u.color ← WHITE
4          u.π = ∅
5      time = 0
6      foreach vertex u ∈ G.V do
7          if u.color = WHITE then
8              DFS-VISIT(G,u)

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
```

8.2 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 8.5 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 8.4: [CLRS09, str. 605]: Primer izvajanja algoritma DFS.

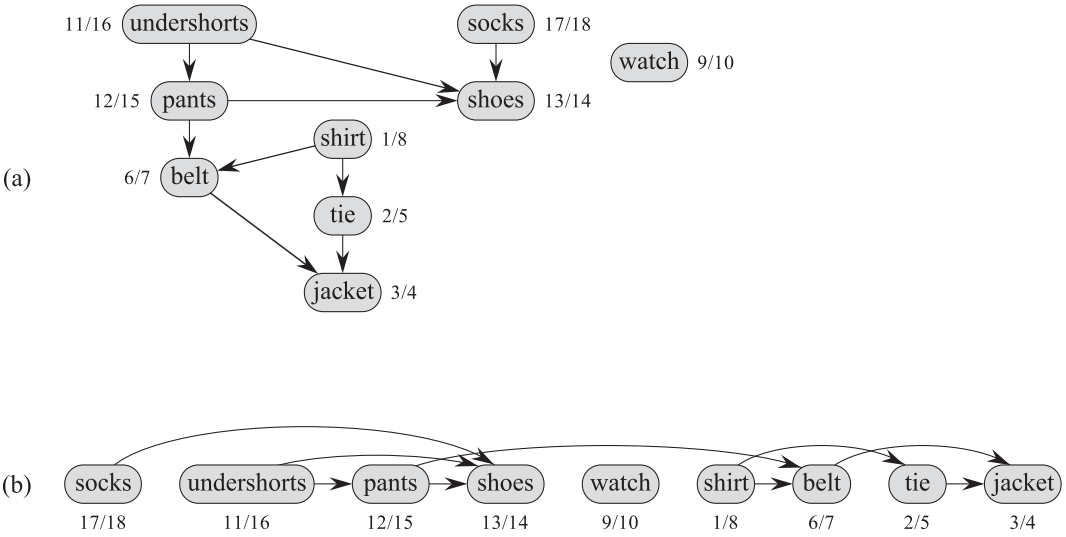
2. Točke po padajočih f vstavi na *začetek* povezanega seznama (glej sliko 8.5, kako vrednosti f padajo po vrsti).

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 $O(V + E)$.

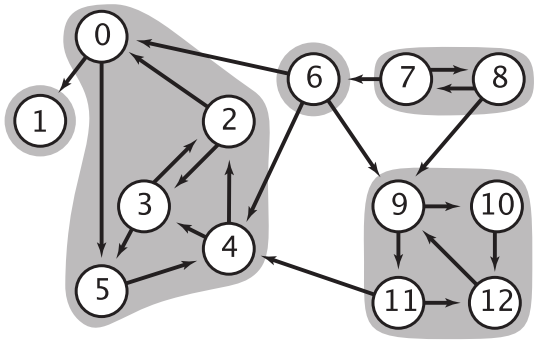
8.3 Krepko povezane komponente v usmerjenem grafu

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



Slika 8.5: Primer topološkega urejanja (b) za dani DAG (a).

Pri usmerjenem grafu nam disjunktne množice pomagajo bolj malo. Na sliki 8.6 so s sivo označene krepko povezane komponente v usmerjenem grafu.



Slika 8.6: [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)

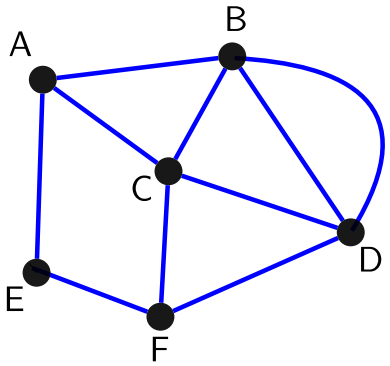
3. Postopek ponavljamo (skočimo na korak 1), dokler sklad ne vsebuje vseh vozlišč grafa.
4. Obrnemo smeri povezav v G.
5. Vzamemo zadnje vozlišče na skladu v in naredimo sprehod v globino.
6. Vsa obiskana vozlišča so v isti krepko povezani komponenti.
7. Vozlišča odstranimo s sklada in skočimo na korak 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 8.6.

8.4 Eulerjev obhod, sprehod

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



Slika 8.7: 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 8.7.

8.5 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?

8.5.1 Primov algoritem [Jar30, Pri57]

Začne v izbrani točki (lahko naključni) in gradi MST 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 16: 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)
```

8.5.2 Kruskalov algoritem [Kru56]

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

Algoritem 17: 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 (18). 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)?

8.6 Iskanje najcenejših poti

Prejšnjič smo si pogledali algoritma za gradnjo MST. Oba, Primov in Kruskalov algoritem, spadata med požrešne algoritme, ker predpostavljata, da v vsakem trenutku lahko izkoristita izbiro najcenejše povezave za gradnjo končne rešitve. Tako se izogneta številnim kombinacijam podproblemov pri gradnji končne rešitve; npr. da bi gradili MST kar po vrsti z vsemi povezavami in potem sestopali (*angl. backtracking*) ter dopolnjevali našo rešitev.

Algoritem 18: 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
```

Pozor Pogostokrat se uporablja poleg izraza najcenejša pot tudi najkrajša pot (angl. *shortest path*). Prvotno so bili algoritmi v tem poglavju zares namenjeni iskanju najkrajših poti na zemljevidu, saj je bila cena=razdalja. V splošnem pa je cena lahko marsikaj (npr. čas, stroški) in bomo zato uporabljali izraz iskanje najcenejših poti.

8.6.1 Najcenejša pot od izvora do vseh točk brez negativnih uteži (Dijkstra) [Dij59]

Na predavanjih ste si ogledali algoritem za iskanje najcenejših 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.

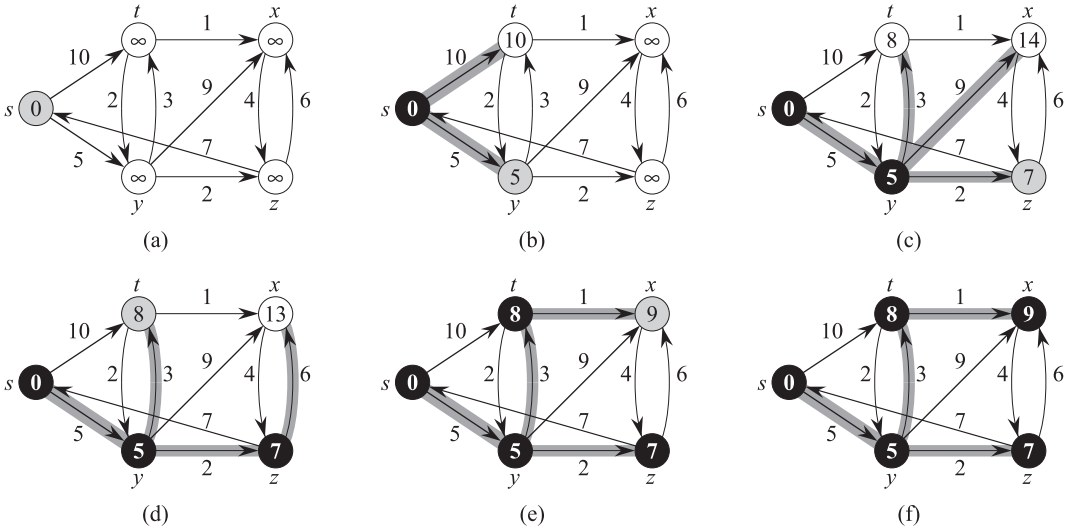
Delovanje:

1. Ceno najcenejše 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 najcenejše poti, če je ta boljša od obstoječe. Operaciji se reče **sproščanje** (*angl. relaxation*). 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`. Z uporabo Fibonaccijeve kopice 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 najcenejše poti iz točke s na grafu 8.8 s pomočjo algoritma Dijkstre.

Naloga Zakaj Dijkstrin algoritem ne deluje z negativnimi povezavami?



Slika 8.8: Iskanje najcenejše poti z Dijkstro.

8.6.2 Najecenejša pot od izvora do vseh točk z negativnimi utežmi (Bellman-Ford) [Bel58, For56]

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

Pozor Dijkstra 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 pri Dijkstri problematične, saj nam omogočajo, da pridemo do že obiskanega vozlišča tudi kasneje po hitrejši poti.

Kaj pa cilki? 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 algoritem 19) avtorjev Richarda Bellmana [Bel58] in Lesterja Forda [For56] — oba sta odkrila isto stvar — ki zna reševati dvoje: problem iskanja najcenejših poti z negativnimi cenami in zaznati negativne cikle.

Algoritem inicializira graf na enak način kot pri Dijkstri (cene na ∞, izvorno na

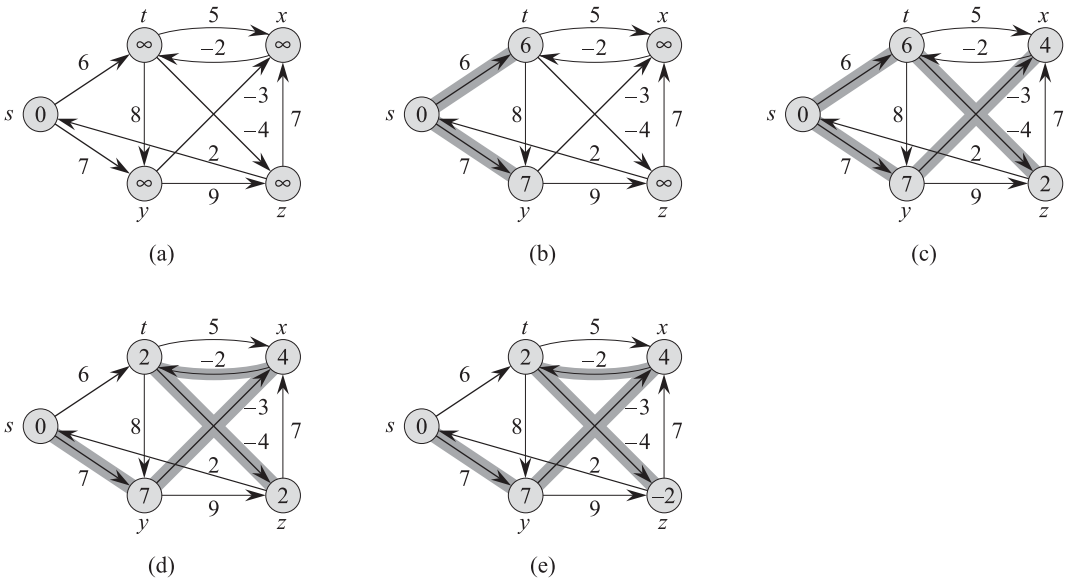
0). Nato gre v prvi fazi najprej n-krat (t.j. število vozlišč) skozi vse povezave in iterativno osvežuje (sprošča) cene poti do vozlišč glede na cene sosednjih vozlišč + ceno poti do vozlišča. V drugi fazi preveri, če vsebuje graf negativne cikle. Če jih ni, morajo biti vse poti optimalne: $v.d \leq u.d + w(u, v)$ pri usmerjeni povezavi $u \to v$. Če so, se bodo razdalje tudi po n iteracijah še naprej krajšale.

Algoritem 19: Bellman-Fordov algoritem za iskanje najcenejših 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 ;
```

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 najcenejše poti iz točke s na grafu 8.9 s pomočjo algoritma Bellman-Ford.



Slika 8.9: Primer delovanja Bellman-Fordovega algoritma.

8.6.3 Najcenejša pot med vsemi pari (Floyd-Warshall) [Roy59, War62, Flo62]

Želimo ustvariti tabelo vseh najcenejših 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 20. 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.

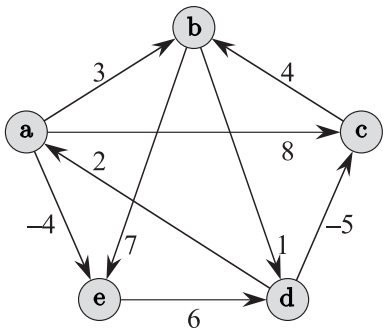
Algoritem 20: Floyd-Warshallov algoritem za iskanje najcenejših poti od vseh vozlišč do vseh ostalih v grafu z negativnimi povezavami in cikli.

```
1  FloydWarshall(G, w):
2  D matrika velikosti |G.V| × |G.V|, inicializirana na ∞
3  Π matrika velikosti |G.V| × |G.V|, inicializirana na ∅
4  begin
5      foreach v ∈ G.V do
6          D[v][v] ← 0
7      foreach edge(u, v) ∈ G.E do
8          D[u][v] ← w(u, v)
9      for k ← 1..|G.V| do
10         for i ← 1..|G.V| do
11             for j ← 1..|G.V| do
12                 if D[i][k] + D[k][j] < D[i][j] then
13                     D[i][j] ← D[i][k] + D[k][j]
14                     Π[i][j] ← k
15     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 8.10 s pomočjo algoritma Floyd-Warshall.



Slika 8.10: Primer grafa za prikaz delovanja Floyd-Warshallovega algoritma.

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

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

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

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

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

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

Slika 8.11: Delovanje Floyd-Warshallovega algoritma nad grafom na sliki 8.10.

8.7 Največji pretok (Ford-Fulkerson, Edmonds-Karp) [FF56, EK72]

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)).

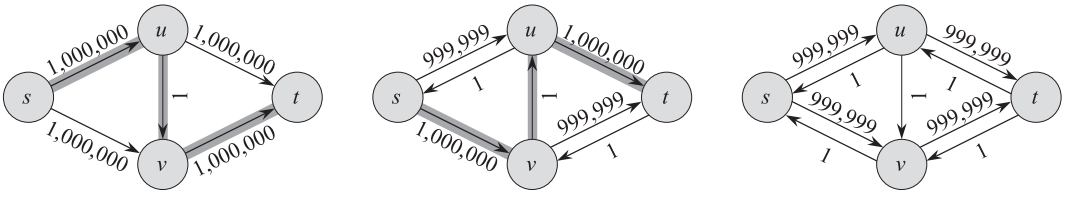
Največji pretok lahko izračunamo s Ford-Fulkersonovim algoritmom [FF56] (glej algoritem 21).

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 8.12).

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 21: 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) ;
```



(a) (b) (c)

Slika 8.12: [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 9

Računska zahtevnost

9.1 Razredi zahtevnosti P in NP

Nek problem je v razredu problemov NP, če obstaja deterministični turingov stroj (to vključuje tudi današnje računalnike — slednji imajo dostop do poljubne besede v konstantnem času ter $O(1)$ število procesnih enot, z vidika izračunljivosti pa so enako močni), ki v polinomskem času $O(n^k)$ **zna preveriti**, ali je podana rešitev problema (*certifikat*) res pravilna rešitev.

Nek problem je v razredu problemov P, če obstaja deterministični turingov stroj, ki v polinomskem času $O(n^k)$ **zna poiskati rešitev problema**.

Naloga Pokaži, da je problem urejanja števil v razredu NP.

Naloga Pokaži, da je problem urejanja števil v razredu P.

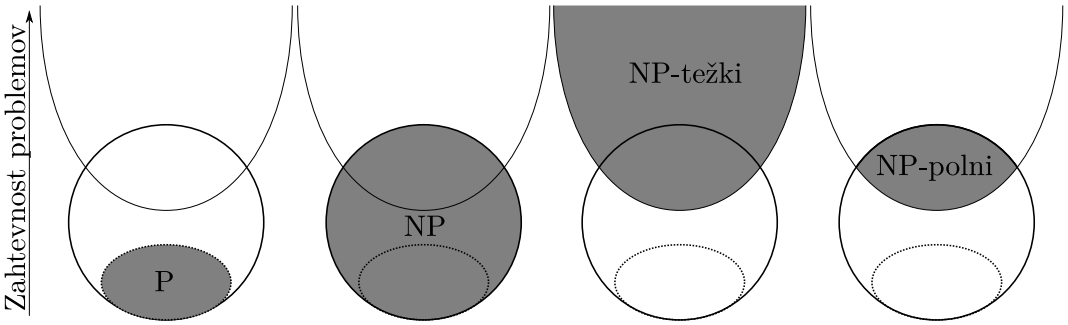
9.2 Razredi težavnosti

9.2.1 Prevedba problema

Problem A je prevedljiv (*reducible*) na problem B ($A \leq B$), če lahko uporabimo algoritem, ki reši problem B, za reševanje problema A. Pri prevedbi zahtevamo, da je polinomsko prevedljiva. To pomeni, da mora biti narejena v polinomskem času — če bi si privoščili eksponenten čas, potem že s prevedbo rešimo problem npr. tako, da naračunamo vse možne rešitve.

S prevedbo pokažemo, da je problem A kvečjem tako težek kot problem B. To pa ne pomeni, da je A resnično tako zelo težek kot B! Lahko je mnogo lažji, pa smo le izbrali premočen algoritem za reševanje problema B.

9.2.2 Hierarhije razredov težavnosti



Slika 9.1: Hierarhija razredov zahtevnosti ob predpostavki $P \neq NP$.

NP-polni problemi so najtežji problemi v NP. "Najtežji" pomeni, da nanje lahko prevedemo vse probleme v NP.

NP-težki problemi so vsi NP-polni problemi in težji (tudi izven NP).

9.2.3 Dilema $P \subset NP$ ali $P = NP$

Če najdemo algoritem, ki na deterističnem Turingovem stroju v polinomskem času reši vsaj en NP-poln problem, smo dokazali, da je $P = NP$ (vsi problemi v NP so enako težko kot v P). Zakaj? Ker so NP-polni problemi prevedeni drug na drugega, znamo posledično rešiti z istim algoritmom vse ostale NP-polne probleme, pa tudi sicer vse probleme v NP.

Če želimo dokazati, da je $P \subset NP$, moramo dokazati, da ne obstaja algoritem, ki bi v polinomskem času rešil kateri koli NP-poln problem. Čeprav to zveni dokaj "naravno", ko od daleč gledamo na NP-polne probleme in jih primerjamo s problemi iz P, do sedaj tega še nismo dokazali in tega ne moremo trditi.

Pozor Da je $P \subset NP$ še ni dokazano, kljub temu, da hierarhije razredov običajno tako rišemo!

9.3 Vrste problemov

Poznamo več vrst problemov:

- Pri **problemih iskanja** (*search problems*) znamo v polinomskem času ugotoviti, ali je podana rešitev optimalna (npr. pogoj, da moramo obiskati vsa vozlišča). Ukvarjamo se s tem, kako jo poiskati oz. skonstruirati (npr. v katerem zaporedju obiskati vsa vozlišča). Vsi problemi iskanja so v razredu NP.
- Pri **optimizacijskih problemih** (*optimization problems*) moramo poiskati najboljšo rešitev za podan problem (npr. na najhitrejši način obiskati vsa vozlišča). V polinomskem času ne znamo preveriti, ali je podana rešitev najboljša.
- Pri **odločitvenih problemih** (*decision problems*) je rešitev problema odgovor da ali ne. Odločitveni problemi so težki glede na vprašanje, ki ga postavimo.

Naloga Kakšne vrste je problem urejanja števil? Problem iskanja, problem optimizacije ali odločitveni problem.

Nekaj primerov:

- Iskanje Hamiltonovega cikla: Problem iskanja, NP-poln.
- Iskanje Eulerjevega cikla: Problem iskanja, P.
- 3-SAT: Problem iskanja, NP-poln.
- 2-SAT: Problem iskanja, P.
- Problem trgovskega potnika:
  - Optimizacijski problem, NP-težek: Za podana mesta in povezave poiščite najkrajši cikel, ki obišče vsa mesta natanko enkrat.
  - Odločitveni problem, NP-poln: Ali je podana pot krajša od podane dolžine L?
- Barvanje grafa:
  - Optimizacijski problem, NP-težek: Iskanje kromatičnega števila (najmanjše število barv) grafa, da je ima vsako vozlišče za soseda drugo barvo.

  - Odločitveni problem, NP-poln: Ali se da graf pobarvati z vsaj k barvami?

- Pokritje grafa:
  - Optimizacijski problem, NP-težek: Minimalna množica vozlišč, da se dotaknemo vseh povezav.
  - Odločitveni problem, NP-poln: Ali ima podan G najmanjšo množico vozlišč veliko največ k?
- Polnjenje košev in 0-1 nahrbtnik:
  - Optimizacijski problem NP-težek: Najmanjše število košev, da spakiramo vse elemente oz. najvišja cena reči v nahrbtniku velikosti w?
  - Odločitveni problem, NP-poln: Ali je podano število košev dovolj za shranjevanje elementov? Oz. za nahrbtnik, ali je lahko skupna cena reči v nahrbtniku velikosti w vsaj v?
- Iskanje najcenejše poti: Optimizacijski problem, P (poženemo Dijkstro ali Bellman-Forda).
- Iskanje najdražje poti (Glej [SW11, str. 911]):
  - Optimizacijski problem, NP-težek. Od podane točke s do t poišči najdaljšo pot, ne da bi obiskali katero vozlišče dvakrat.
  - Odločitveni problem, NP-poln. Za podano pot povejte, ali je krajša od k.
- Celoštevilsko programiranje (*integer linear programming*) — iskanje rešitev sistema neenačb za necela števila: Optimizacijski problem, NP-težek. V praksi se zato poslužimo ene izmed numeričnih metod. Poseben primer celoštevilskega programiranja je, ko spremenljivke zavzemajo vrednost 0 ali 1 — v tem primeru gre za problem iskanja. Podobno, če so spremenljivke cela števila, uporabimo simplex. V obeh primerih gre za NP-poln problem.

Poglavje 10

Približnostni algoritmi

Približnostni algoritmi za razliko od eksaktnih namesto optimalne rešitve, izračunajo približno rešitev. Ta je v praksi "dovolj dobra".

Poznamo nekaj vrst približnostnih algoritmov:

- *Monte Carlo* in *Las Vegas* naključno izbirajo primere rešitev. Več primerov (točk) si izmislimo, bolj natančna bo rešitev. Monte Carlo se zaključi, ko preteče določeno število iteracij. Las Vegas se zaključi, ko dosežemo želeno natančnost. Primer: merjenje površine ali volumna predmetov glede na obliko, izračun konstante π, numerično integriranje.
- *Hevristike* so deterministični algoritmi, ki načeloma izračunajo suboptimalno rešitev, pri določenih pogojih pa celo optimalno. Formalno lahko izračunamo, kako natančna bo rešitev. Ker je algoritem deterministične, se brez izboljšanja algoritma natančnosti ne da izboljšati (za razliko od Monte Carlo ali Las Vegas, kjer dlje časa kot računamo, boljšo rešitev dobimo). Pri hevristikah je običajna tehnika dinamičnega programiranja. Primer: iskanje najcenejših poti z A*, First-Fit in Best-Fit hevristiki za polnjenje košev in 0-1 nahrbtnik.
- *Genetski algoritmi* uporabijo tri metode: selekcijo, križanje in mutacijo. Vsak osebek v generaciji predstavlja eno potencialno rešitev. Ocenitvena funkcija (*fitness function*) oceni vsak osebek. V naslednji iteraciji slabši umrejo (selekcija), s križanjem dveh osebkov pridemo do novih osebkov, poleg tega pa pri križanju mutacija naključno zamenja poljuben bit ali več bitov osebka in poskrbi za raznolikost. Primer: gradnja fakultetnega urnika. Sorodni algoritmi genetskim so še algoritmi za simulirano ohla-

janje (angl. *simulated annealing*) ter sistem kolonije mravelj (angl. *ant colony optimization*).

10.1 Problem Trgovskega Potnika (TSP)

Hamiltonov cikel: Za podan graf želimo obiskati vsa vozlišča natanko 1x in se vrniti v izvorno.

Problem trgovskega potnika: Želimo najti najkrajši Hamiltonov cikel.

Za domačo nalogo imate poln graf, kjer med vozlišči velja evklidska razdalja. Sprogramirati morate dve rešitvi za TSP:

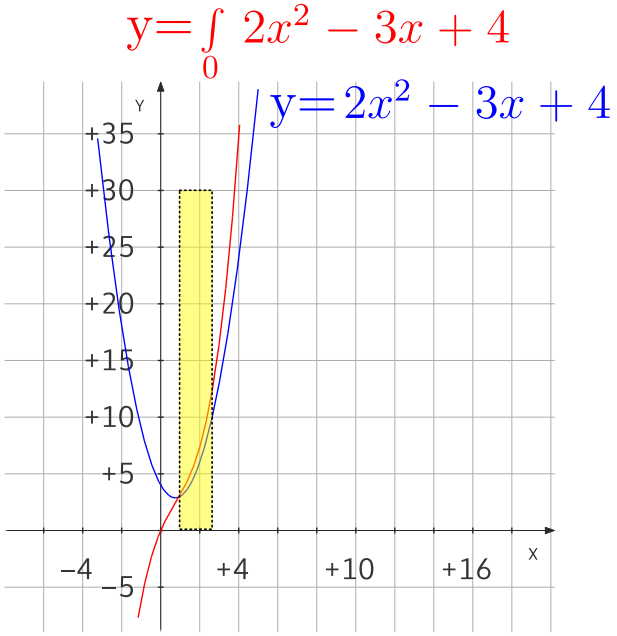
1. eksaktna rešitev z izčrpnim preiskovanjem in dodano hevristiko, naj ne preiskuje že predolgih poti
2. približna rešitev: začnemo s požrešno metodo najbližjih sosedov, rešitev izboljšamo s paroma menjajočimi vozlišči.

10.2 Integriranje s pomočjo metode Monte Carlo

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

Oris algoritma:

```
1 float integralMonteCarlo( float *f ) {
2   int MAX_IT=1000;
3   int pointSum = 0; // number of points below the function
4   for (int i=0; i<MAX_IT; i++) {
5     float px = Random.rand()*1.5 + 1;
6     float py = Random.rand()*30;
7
8     if (f(px) > py) {
9       pointSum++;
10    }
11  }
12  return ((float)pointSum/MAX_IT) * 30 * 1.5;
13 }
```


$y = \int_0 2x^2 - 3x + 4$
$y = 2x^2 - 3x + 4$
Y
X
+35
+30
+25
+20
+15
+10
+5
−4
−5
+4
+10
+16

10.3 Polnjenje košev

Naloga Podane elemente velikosti $x_1, x_2, \ldots$ želimo spraviti v čim manjše število košev velikosti V.

Obstajata pa dve najbolj znani hevristiki:

- First-Fit gre po vrsti po elementih in spravi element v prvi koš, ki je dovolj prost.
- Best-Fit gre po vrsti po elementih in po vseh še ne-polnih koših in spravi element v koš, ki bo najbolje izkoriščen.

Vsaka ima še podvarianto *ordered* First-Fit oz. Best-Fit, ki najprej uredi elemente po velikosti padajoče. Natančnost, ki jo dobimo brez urejanja je 2-kratnik optimalnega števila košev v najslabšem primeru. Natančnost z urejanjem je okoli 1.2-kratnik optimalnega števila košev.

10.4 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 nahrbtnik (brez rezanja predmetov).

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

Genetski algoritem za 0/1-nahrbtnik: 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 1. kolokvij

11.1 Slovar — drevesa in razpršene tabele

Naloga

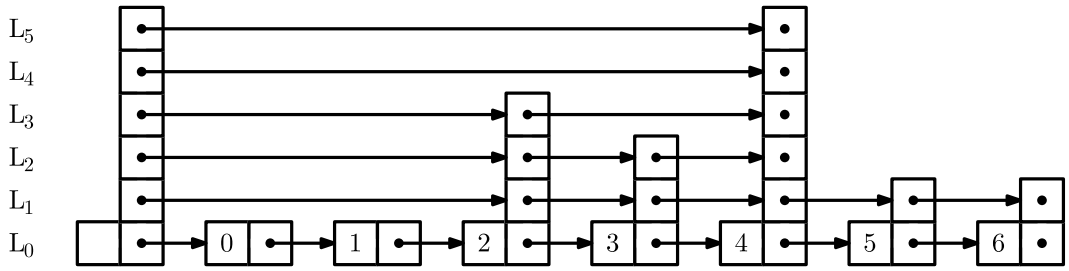
1. Naštej obhode po drevesu. Kako delujejo?
2. Ali znamo zgraditi dvojiško iskalno drevo iz že urejenih števil? Ali je uravnoteženo? Kolikšna je časovna zahtevnost?
3. Koliko časa potrebujemo za iskanje v dvojiškem iskalnem drevesu? Ali znamo hitreje s katero drugo podatkovno strukturo? Čemu potem sploh uporabljati drevesa, če se da hitreje?

Naloga Imamo n vstavljanj in n^2 poizvedb.

11.2 Slovar — preskočni seznami

Naloga Nad preskočnim seznamom na zgornji sliki 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.

Želimo poiskati k-ti najmanjši v skip listu. Kako ga boste poiskali?



Slika 11.1: Preskočni seznam

11.3 Priprava - Disjunktne množice

Peter Zmeda in njegova prijateljica Špela se igrata naslednjo igrico:

1. na začetku imamo n lučk, ki so označene od 1 do n;
2. Peter k krat poveže po dve lučki;
3. Špela se z žico dotakne ene od lučk in potem vse lučke, ki jih je Peter povezal z dotaknjeno lučko, zasvetijo.

Vprašanja:

1. Naj bo n = 10 in Peter nato poveže pare: (5, 7), (1, 8), (2, 9), (3, 8), (4, 7) in (7, 1). Špela se z žico dotakne lučke 4. Katere od naslednjih lučk svetijo: 2, 5, 8, 9 in 10?
2. Pri velikih n postane igrica precej težko izvedljiva, zaradi česar sta se Špela in Peter odločila napisati računalniški program, ki bo simuliral igrico. Program in posledično podatkovna struktura mora nudite naslednje ukaze:
   - `Povezi(x, y)`, ki poveže lučki x in y;
   - `Prizgi(x)`, s katerim se dotaknemo lučke x in
   - `Sveti(x)`, ki vrne DA ali NE odvisno od tega, ali lučka x sveti.

   Opišite kako naj izgleda podatkovna struktura, ki podpira opisano igirco ter kako so implementirani posamezni ukazi.

   Namig: Morda lahko uporabite kakšno znano podatkovno strukturo.

11.4 Vrste s prednostjo - dvojiška kopica, binomska kopica, Fibonaccijeva kopica

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?
5. Ali obstaja primer, ko je Fibonaccijeva kopica primernejša od binomske kopice in ali obstaja primer, kjer je binomska kopica primernejša od Fibonaccijeve.

11.5 Priprava - Razširjene podatkovne strukture

3. Kako težki so?

Naloga Otroci so se igrali na dvorišču naslednjo igro ugibanja. Ugibali so skupno težo določene podmnožice otrok in sicer je ta podmnožica definirana na podlagi višine otrok. Tako je bila poizvedba lahko: "Kolikšna je skupna teža vseh otrok, ki so višji od 112 cm in nižji od 127 cm?" Recimo, da to poizvedbo poimenujemo `Teza(112, 127)`.

Recimo, da imamo naslednjo množico otrok s težami (prva številka predstavlja višino in druga težo): (120, 112), (128, 78), (153, 108), (98, 130), (117, 116), (98, 102), (149, 122), (129, 113), (116, 102) in (114, 103). Vprašanja:

1. Pri opisani množici otrok odgovorite na naslednje poizvedbe:

   `Teza(108, 141)`, `Teza(121, 151)`, `Teza(97, 114)`,

`Teza(149, 124)` in `Teza(124, 123)`.

2. Recimo, da je v množici n otrok. Opišite podatkovno strukturo, ki časovno kar se da učinkovito odgovarja na poizvebo `Teza()`. Utemeljite njeno pravilnost ter njeno prostorsko in časovno učinkovitost.

3. Množica otrok na dvorišču seveda ni stalna, ker jih mame kličejo na kosilo ali večerjo ali pa se pridružujejo skupini. Dodelajte podatkovno strukturo, da bo podpirala še operaciji

   `Odsel(visina, teza)` in `Prisel(visina, teza)`.

   Kakšna je časovna zahtevnost vseh treh vaših operacij? Kakšna je prostorska zahtevnost podatkovne strukture? Utemeljite odgovor.

4. Kateri so najtežji?

Naloga Ponovno smo pri igri iz prejšnje naloge, le da tokrat težo pri vsakem otroku zmanjšamo za 111 enot in imamo sedaj pare: (120, 1), (128, -33), (153, -3), (98, 19), ..., (114, -8).

Vprašanja:

1. Vsota tako spremenjenih tež med katerima višinama je največja? Opišite postopek, kako ste to izračunali.

2. Recimo, da imamo n otrok in da so vse njihove višine različne. Kakšna je časovna in prostorska zahtevnost vašega postopka iz prvega vprašanja? Ocenite zahtevnosti za posamezne korake (faze), če se seveda vaš postopek sestoji iz njih.

11.6 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.

11.7 Dinamično programiranje

11.7.1 Longest Common Subsequence, Substring, Edit distance

[2. kolokvij 2015/2016, naloga 1]

Imejmo dva niza: $X = PREDMETE$, $Y = REDNE$.

Izračunajmo najdaljše skupno podzaporedje.

Formula za izračun LCSubSeq je:

- 0, če $i = 0$, $j = 0$,
- $LCSubSeq[i-1][j-1] + 1$, če $i, j > 0$ in $X_i = Y_j$,
- $max(LCSubSeq[i][j-1], LCSubSeq[i-1][j])$, če $i, j > 0$ in $X_i \neq Y_j$

Formula za izračun LCSubStr je:

- 0, če $X_i \neq Y_j$,
- $LCSubStr[i-1][j-1] + 1$, če $i, j > 0$ in $X_i = Y_j$,

Formula za izračun urejevalne (*Levenshteinove*) razdalje je:

- $ED[i-1][j-1]$, če $Y_i = X_i$,
- če $X_1 \neq Y_1$, potem **minimum** od:
  - $ED[i-1][j] + w_{del}(Y_i)$
  - $ED[i][j-1] + w_{ins}(X_j)$
  - $ED[i-1][j-1] + w_{rep}(X_i, Y_j)$

Primer na spodnji sliki:

		k	i	t	t	e	n
	0	1	2	3	4	5	6
s	1	1	2	3	4	5	6
i	2	2	1	2	3	4	5
t	3	3	2	1	2	3	4
t	4	4	3	2	1	2	3
i	5	5	4	3	2	2	3
n	6	6	5	4	3	3	2
g	7	7	6	5	4	4	3

		S	a	t	u	r	d	a	y
	0	1	2	3	4	5	6	7	8
S	1	0	1	2	3	4	5	6	7
u	2	1	1	2	2	3	4	5	6
n	3	2	2	2	3	3	4	5	6
d	4	3	3	3	3	4	3	4	5
a	5	4	3	4	4	4	4	3	4
y	6	5	4	4	5	5	5	4	3

Slika 11.2: Urejevalna razdalja za besedi `sitting` in `kitten` ter `Sunday` in `Saturday`.

11.7.2 0-1 nahrbtnik in podobni

[3. pisni izpit 2015/2016, 2. naloga]

Naloga No, tudi to se je zgodilo. Peter Zmeda se je brezmejno zaljubil v Alenčico, ki pa mu njegove pozornosti ne vrača. Zato se je odločil, da bo prite- gnil njeno pozornost s tem, da se nauči nekaj novih spretnosti, s katerimi jo bo očaral. Kot dober informatik, se je lotil zadeve silno sistematično. Najprej je naredil seznam spretnosti, za katere misli, da se jih lahko nauči in ocenil za za vsako spretnost koliko časa bi mu vzela, da se je nauči. Poleg tega je poizvedel pri Alenčičini sestri Cvetoslavi, koliko Alenčica ceni posamezno spretnost. Vse skupaj je spravil v spodnjo preglednico:

spretnost čas učenja vrednost

igranje orglic 12 11

igranje kitare 20 15

ples 8 15

mešanje koktejlov 6 9

astronomija 15 10

igranje košarke 5 7

kuhanje hrenovk 2 6

dokaz Einstein-Pitagorovega izreka 5 6

Stolpec vrednost podaja koliko Alenčica ceni posamezno spretnost, medtem ko stolpec čas učenja podaja čas v dnevih, ki ga Peter potrebuje, da se nauči določene spretnosti.

VPRAŠANJA :

A) Najprej predpostavimo, da se Peter določene spretnosti lahko nauči tudi delno. Z drugimi besedami, če se bo učil igranje košarke samo tri dni, se je bo naučil zgolj 60% in bo Alenčica to cenila ne z vrednostjo 7, ampak samo z vrednostjo $4,2$. Peter ima do novega snidenja z Alenčico natančno 24 dni časa. Katere spretnosti in koliko se jih naj nauči, da bo naredil na Alenčico najboljši možen vtis. Utemeljite pravilnost svojega odgovora.

B) Jojmene! Ko je Peter malce premislil vse skupaj, je ugotovil, da tole delno učenje spretnosti ne deluje: spretnosti se mora naučiti povsem ali pa ne bo učinka. Koliko največ vtisa lahko Peter naredi na Alenčico tokrat, če je še vedno do njunega naslednjega snidenja natančno 24 dni? Utemeljite odgovor.

C) V splošnem imamo n spretnosti, kjer spretnost i vzame ti časa, da se je Pe- ter nauči in naredi vtis vi na Alenčico. Zapišite algoritem, ki poišče nabor spretnosti, ki se jih Peter lahko nauči v času T, da bo naredil najboljši vtis na Alenčico.

Priprava na 2. kolokvij

11.8 Delo z nizi

11.8.1 Številsko drevo, stiskanje poti, stiskanje plasti

Priponsko drevo je številsko drevo + stiskanje poti.

a) Vstavimo elemente $(0100, A)$, $(0001, G)$, $(1000, R)$, $(1100, N)$, $(0101, P)$, $(1111, X)$ v številsko drevo, stisnjeno po poteh (PATRICIA).

b) Sedaj uporabimo stiskanje plasti. Izračunejte zasedenost polja α za $L = 2$ in $L = 3$.

11.8.2 Priponsko drevo, priponsko polje

[2. kolokvij 2015/2016, naloga 2]

Vstavimo vse pripone $X = PREDMETE$ in $Y = REDNE$ v priponsko drevo.

Nariši še priponsko polje.

Zgradi DKA za vzorec X.

11.9 Grafi

11.9.1 Zapis grafa

Ponovimo: Z matriko sosednosti, s seznamom sosedov, na predavanjih pa ste imeli tudi incidenčno matriko (vozlišče -> povezave).

Ali znate sprogramirati pretvorbo iz matrike v seznam sosedov?

11.9.2 Premer grafa

[2. kolokvij 2015/2016, naloga 3]

Naloga Naj bo premer grafa definiran kot najkrajša pot med najbolj oddaljenima vozliščema. Kako poiskati premer grafa?

11.9.3 Sprehod v globino

Uporabno za iskanje ciklov v grafu.

11.9.4 Sprehod v širino

[2. kolokvij 2013/2014, naloga 3]

11.9.5 SSSP, APSP

[2. kolokvij 2013/2014, naloga 4]

11.10 P, NP, približnostni algoritmi

Naloga Imejmo naslednji problem MAX 3-SAT:

$$(\neg x_1 \vee \neg x_2 \vee \neg x_3) \wedge (x_1 \vee x_1 \vee x_1) \wedge (\neg x_1 \vee \neg x_2 \vee x_3) \wedge (\neg x_1 \vee x_2 \vee \neg x_1)$$

a) Rešite izraz z deterministočnim algoritmom. Kakšna je časovna zahtevnost?

b) Rešite izraz z enim izmed približnostnih algoritmov. Kakšna je časovna zahtevnost sedaj?

[2. kolokvij 2014/2015, naloga 4]

Naloga Na predavanjih smo spoznali problem iskanja najcenejšega vpetega drevesa v grafu G(V, E). To je optimizacijski problem. (i.) Preoblikujte ga v odločitveni problem. (ii.) Ali je vaš odločitveni problem v NP? Utemeljite odgovor.

Naloga Eden od zanimivih neuporabnih algoritmov je Bogosort, katerega psevdokoda je:

```
WHILE NOT urejeno(polje):
    NakljucnoPermutiraj(polje)
```

(i.) Ali gre za Monte Carlo ali za Las Vegas algoritem? Utemeljite odgovor. (ii.) Kakšna je: (a) najboljša, (b) najslabša in kakšna (c) pričakovana časovna zahtevnost algoritma? Utemeljite odgovor.

Literatura

[AKO04] Mohamed Ibrahim Abouelhoda, Stefan Kurtz, and Enno Ohlebusch. Replacing suffix trees with enhanced suffix arrays. *Journal of Discrete Algorithms*, 2(1):53–86, 2004.

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

[AVL62] Georgii Maksimovich Adelson-Velskii and Evgenii Mikhailovich Landis. An algorithm for organization of information. *Dokl. Akad. Nauk SSSR*, 146:263–266, 1962.

[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.

[BM77] Robert S. Boyer and J. Strother Moore. A fast string searching algorithm. *Communications of the ACM*, 20(10):762–772, oct 1977.

[BW94] Michael Burrows and David J Wheeler. A block-sorting lossless data compression algorithm. Technical report, 1994.

[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, apr 1972.

[FF56] L R Ford and D R Fulkerson. Maximal Flow through a Network. *Canadian Journal of Mathematics*, 8:399–404, 1956.

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

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

[FLPR99] Matteo Frigo, Charles E. Leiserson, Harald Prokop, and Sridhar Ramachandran. Cache-Oblivious Algorithms. In *Foundations of Computer Science, 1999. 40th Annual Symposium on*, page 285. IEEE Computer Society, 1999.

[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.

[Jar30] Vojtěch Jarník. O jistém problému minimálním: (Z dopisu panu O. Borůskovi). *Práce moravské přírodovědecké společnosti*, 6(4):57–63, 1930.

[KMP77] Donald Knuth, J Morris Jr., and V Pratt. Fast Pattern Matching in Strings. *SIAM Journal on Computing*, 6(2):323–350, 1977.

[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.

[McC76] Edward M McCreight. A Space-Economical Suffix Tree Construction Algorithm. *Journal of the ACM*, 23(2):262–272, 1976.

[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, jan 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, oct 2004.

[Pri57] Robert Clay Prim. Shortest connection networks and some generalizations. *The Bell System Technical Journal*, 36(6):1389–1401, nov 1957.

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

[Ukk95] Esko Ukkonen. On-Line Construction of Suffix Trees. *Algorithmica*, 14(3):249–260, 1995.

[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, jan 1962.

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