TARTU ÜLIKOOL
Arvutiteaduse instituut

Jüri Kiho

ALGORITMID JA ANDMESTRUKTUURID

TARTU 2003

T A R T U Ü L I K O O L
Arvutiteaduse instituut

Jüri Kiho

ALGORITMID JA ANDMESTRUKTUURID

Kolmas, parandatud ja täiendatud trükk

TARTU 2003

Toimetaja: Peeter Laud

©Jüri Kiho 1994, 1997, 2003

ISBN 9985-56-767-6

Tartu Ülikooli Kirjastus
Tiigi 78, Tartu 50410
Tellimus nr. 409

Sisukord

Sissejuhatus **5**

1 Algoritmi ajaline keerukus **11**

2 Andmestruktuurid **23**
2.1 Tüübi mõiste 23
2.2 Magasin ja järjekord 26
2.3 Puud 27
2.4 Graafi mõiste 40

3 Andmestruktuuride realiseerimine **41**
3.1 Järjestikpaigutus ja seotud paigutus 41
3.2 Paisksalvestus 45
3.3 Klasside kujutamine 49
3.4 Kuhjad 52

4 Sorteerimine **59**
4.1 Pistemeetod 60
4.2 Kiirmeetod 62
4.3 Ühildusmeetod 66
4.4 Sorteerimise ajalise keerukuse alampiir 70
4.5 Sorteerimise erimeetodeid 72

5 Sõnetöötlus **75**
5.1 Alamsõne otsimine 75
5.2 Teksti pakkimine 83
5.3 Pikima ühise osasõne otsimine 87

6 Graafitöötlus **93**
6.1 Graafi tippude topoloogiline sorteerimine 93
6.2 Lühimad teed graafis 98
6.3 Graafi läbimine 106
6.4 Minimaalne toes 109

7 Algoritme planimeetriast **113**
7.1 Lihtsamaid erivõtteid 113
7.2 Punktihulga töötlemine 118

8 Algoritmi korrektsus **125**
8.1 Algoritmi korrektsuse mõiste 125
8.2 Algoritmi korrektsuse tõestamine 129
8.2.1 Tühi algoritm 129
8.2.2 Järjestikalgoritm 130
8.2.3 Omistamisdirektiiv 130
8.2.4 Tingimuslik direktiiv 132
8.2.5 Tsüklidirektiiv 134
8.3 Näide: kiire astendamise algoritmi korrektsus 140

Kirjandus **143**

Aineregister **144**

Sissejuhatus

Käesolev õppevahend on alusmaterjaliks loengutele ja praktikumidele aines *algoritmid ja andmestruktuurid*. Selles antakse konspektiivses vormis põhiosa vajalikust faktilisest materjalist. Esitatakse näiteid juba klassikaliseks kujunenud algoritmidest ja andmestruktuuridest ning tutvustatakse efektiivsete algoritmide koostamise põhimõtteid.

Õppevahendi esimeses peatükis esitatakse algoritmi töökiiruse hindamisega seotud põhimõisted: algoritmi ajaline keerukus, O-relatsioon, funktsiooni asümptootiline hinnang, polünomiaalne keerukus, algoritmianalüüsi elemendid. Teises peatükis defineeritakse peamised struktuursed andmetüübid ehk andmestruktuurid, kusjuures üksikasjalikumalt kirjeldatakse mitmesuguseid puustruktuure ja nendega seotud operatsioone. Peale lihtsa kahendotsimise puu vaadeldakse veel AVL- ja B-puud ning esitatakse ka binomiaalpuu mõiste. Kolmas peatükk on pühendatud andmestruktuuride realiseerimise küsimustele. Kõrvuti selliste üldiste realiseerimismeetoditega, nagu järjestikpaigutus, seotud paigutus ja paisksalvestus, kirjeldatakse veel klasside (ühisosata hulkade) töötlemist ning kuhja mõistel põhinevaid realisatsioone. Sorteerimismeetodeid vaadeldakse neljandas peatükis, kus esitatakse üksikasjalikud algoritmid massiivi sorteerimiseks kiirmeetodil ja ahela sorteerimiseks ühildusmeetodil. Nimetatud meetodid annavad lugejale ettekujutuse „jaga-ja-valitse"-tüüpi algoritmidest. Selles peatükis tuletatakse ka võrdlemisel põhinevate sorteerimisalgoritmide ajalise keerukuse alampiir ja tutvustatakse mõningaid kiiremaid erimeetodeid. Viiendas peatükis kirjeldatakse tuntumaid sõnetöötlusalgoritme: Knuth-Morris-Pratti, Boyer-Moore'i ja Rabin-Karpi algoritmid alamsõne otsimiseks, teksti pakkimine Huffmani koodide alusel, pikima ühise osasõne otsimine. Viimane on ühtlasi ka dünaamilise

kavandamise meetodi näiteks. Ahne algoritmi mõistele pööratakse lugeja tähelepanu aga seoses Huffmani algoritmiga. Graafitöötluse põhivõtteid valgustatakse kuuendas peatükis. Käsitletakse tsükliteta graafi tippude topoloogilist sorteerimist ja selle rakendusi võrkgraafiku analüüsimisel, graafi läbimise meetodeid, lühimate ja otseteede leidmist ning graafi minimaalse toese konstrueerimist nii Kruskali kui ka Primi meetodil. Seitsmendas peatükis selgitatakse võimalusi efektiivsete planimeetriaalgoritmide koostamiseks. Suuremate näidetena tuuakse algoritmid punktihulga kumera katte konstrueerimiseks (Grahami seiremeetodil) ja vähima vahemaaga punktipaari leidmiseks. Viimases, kaheksandas peatükis esitatakse algoritmi korrektsuse ja selle tõestamise mõiste. Õppevahendi lõpus paikneb indeks, milles tähestiku järjekorras loetletakse kasutatud terminid koos nende ingliskeelsete vastete ja leheküljeviitadega.

Esitatud algoritmide jaoks on reeglina antud ka nende ajalise keerukuse hinnangud. Õppevahendi piiratud mahu tõttu pole aga ära toodud vastavaid tuletuskäike, samuti puuduvad algoritmide korrektsuse matemaatilised tõestused. Aine süvendatud omandamiseks võib soovitada sellealase kirjanduse tähtteoseid [1] ja [2]. Üheks kõige täielikumaks algoritmide koguks on samuti [3]. Andmestruktuuride praktilist realiseerimist objekt-orienteeritud stiilis tutvustavad raamatud [4] ja [5]. Veebimaterjalidest väärib esilet ostmist algoritmide ja andmestruktuuride leksikon [6]. Algoritmide esitamist skeemkeeles selgitatakse põhjalikumalt õppevahendites [7] ning [8]. Viimases kirjeldatakse ka programmeerimiskeele C algmõisteid.

Kasutatav algoritmikeel on üldjoontes lähedane programmeerimiskeelele C. Viimasest on „laenatud“ nii algoritmi üldine struktuur kui ka omistamiskäsu ning tingimuse mõisted. Tõsi küll, omistamise märgina kasutatakse sümbolit „:=“, mis omakorda võimaldab võrdumise kontrollimist väljendada ikkagi tavalise võrdusmärgi abil. Üldse püütakse võimalikult palju kasutada tavapärast matemaatilist sümboolikat. Algoritmide koostamisel peetakse esmatähtsaks arusaadavust, mistõttu mitmelgi juhul on loobutud võimalikust lakoonilisemast esitusvõimalusest. Juhtimiskonstruktsioonid kujutatakse graafiliste skeemidena.

Lihtskeemi tähistab

$\lfloor D$

kus D on nn. skeemielementide vertikaalselt paigutatud järjend. **Skeemielemendiks** selles järjendis võib olla kas käsurühm, tingimus, (väljund)nool või (alam)skeem. Käsurühm on rida, mis sisaldab ühte või mitut (semikooloniga eraldatud) käsku.

Tingimuseks nimetatakse kirjutist, mis koosneb avaldisest ja sellele järgnevast küsimärgist. Tingimus paigutatakse alati omaette ritta.

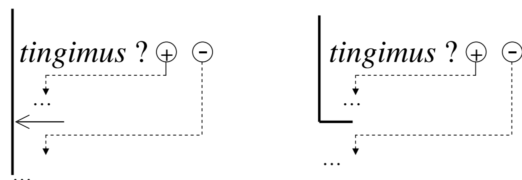
Nool

$\longleftarrow$

asub samuti omaette reas ja võib olla ka pikendatud, ulatudes mõne teise, hõlmava skeemi jooneni. Nool ise aga loetakse alati selle skeemi elemendiks, millest ta algab.

Lihtskeemi täitmine toimub ülalt-alla suunas skeemielementide kaupa. Seejuures noole tähenduseks on tema poolt osutatud skeemi täitmise lõpetamine. Lihtskeemi abil kirjeldatakse hargnevaid lahenduskäike.

Tingimuse täitmine tähendab valikut kahe võimaluse vahel: kas jätkata tingimusele järgneva skeemielemendi täitmisega või jätta osa järgnevaid skeemielemente vahele:



Valik sõltub sellest, kas tingimus on rahuldatud (st. avaldise väärtus on tõene ehk mittenull) või mitte (avaldise väärtus on väär ehk null). Nimelt jätkub tingimuse rahuldatuse korral skeemi täitmine loomulikus järjekorras, vastasel korral aga jäetakse vahele kõik tingimusele järgnevad skeemielemendid kuni järgmise nooleni (kaasa arvatud) ning jätkatakse täitmist sellele noolele järgnevast skeemielemendist. Kui mitterahuldatud tingimuse järel noolt skeemis ei esine, siis lõpetatakse üldse selle skeemi täitmine, st. väljutakse skeemist.

Funktsioonikirjeldus esitatakse päisega **moodulskeemina**, kusjuures funktsiooni väärtuse tagastamist tähistatakse noolega, millele järgneb tagastatavat väärtust määrav avaldis sulgudes:

$f(x)$

...

$\longleftarrow (y)$

...

Funktsioonikirjelduse üldkuju:

funktsioonikirjelduse päis - - - *funktsiooni välimine spetsifikatsioon*

D - - - *funktsiooni sisemine spetsifikatsioon*

Funktsioonikirjelduse sisu (D) täitmine on analoogiline lihtskeemi täitmisega.

Tsükliskeemi

D

ehk **piiramata tsükli** peamiseks iseärasuseks on see, et pärast järjendis D viimase käsurühma täitmist jätkub tsükliskeemi töö jälle algusest, st. D esimesest elemendist. Sellise skeemi täitmine saab lõppeda vaid „sunnitult“: kas noole kaudu või siis sellise mitterahuldatud tingimuse avastamisel, millele järgnevate skeemielementide seas ei leidu noolt. Tsükliskeemis võib kasutada ka **nõrka noolt**. See kujutatakse punktiiris ja tähendab mitte skeemi täitmise katkestamist, vaid selle taasalustamist algusest (nagu oleks täitmisjärjega juba jõutud tsükliskeemi lõppu).

Päisega tsükli üldkujuks on

$\star\ p$

D

kus p on **tsükli päis** ja D **tsükli sisu**.

Päisega tsükli eriliigid erinevad päise kuju poolest:

lihtkorduse puhul näidatakse päises sisu kordamiste arv;

jadatsükli päises $i = v_1, v_2, \ldots, v_n$ esitatakse tsükliloendaja (i) koos selle väärtusvaruga;

üldtsükli ehk **C-tsükli** päiseks on kolm semikoolonitega eraldatud avaldist; esimene neist arvutatakse enne esimest tsükli sammu, kolmas aga pärast iga sammu täitmist, keskmine avaldis on tsükli jätkamistingimuseks, mida kontrollitakse enne järjekordsele sammule asumist.

Üldiselt korratakse päisega tsükli sisu ettenähtud arv kordi (lihtkorduse ja jadatsükli puhul) või seni, kuni jätkamistingimus veel kehtib (üldtsükli puhul). „Enneaegselt“ saab päisega tsükli täitmine lõppeda ainult tugeva noole toimel. Erinevalt piiramata tsüklist ei põhjusta sellise tingimuse, millele ei järgne noolt, mitterahuldatus siin tsükli katkestamist, vaid viib täitmisjärje uuesti tsükli sisu kordamisele (üldtsükli puhul – jätkamistingimuse kontrollimisele)

Lihtsamad erikujulised skeemid võib esitada ka üherealistena:

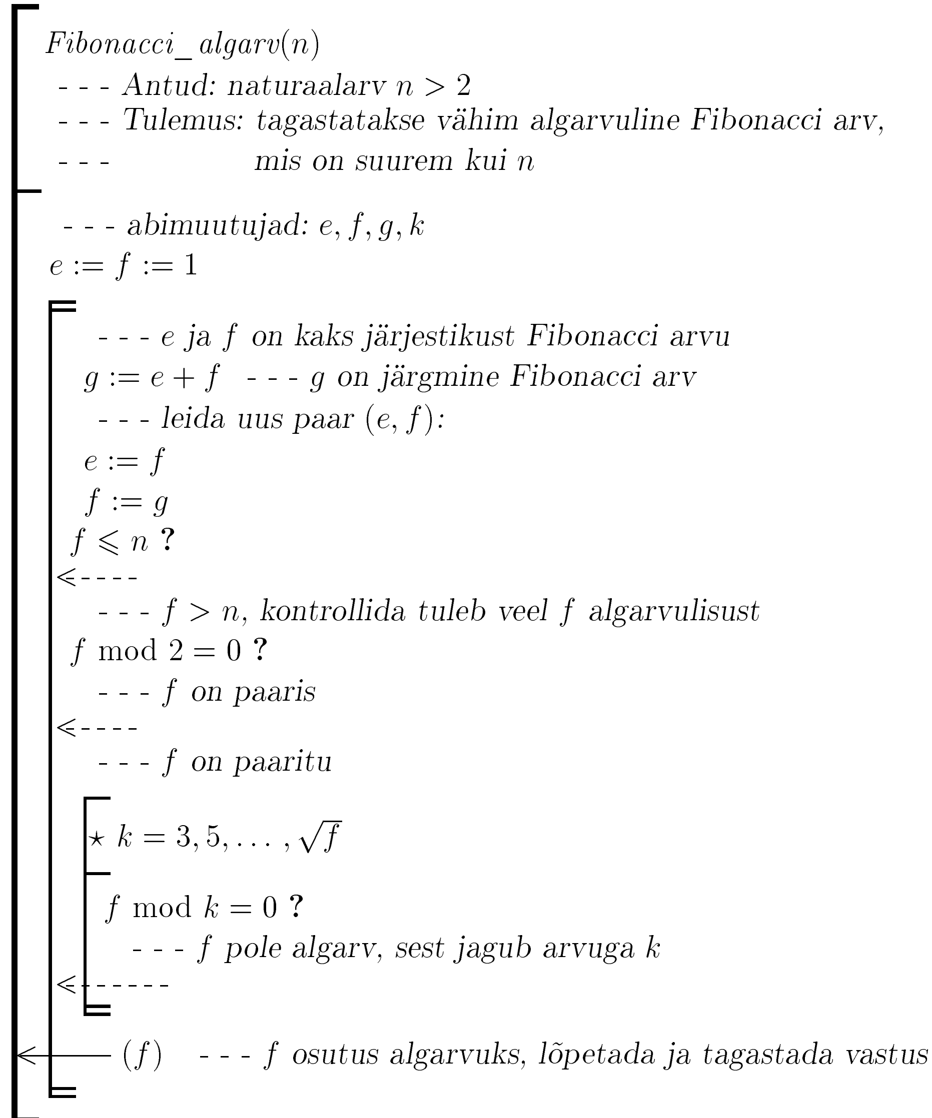
[t **?** S	$\Rightarrow$	**[?** (t) S
[t **?** S ← S’	$\Rightarrow$	**[?** (t) S : S’
[$\star$ p S	$\Rightarrow$	**[**$\star$ (p) S

kus S ja S' on käsurühmad.

Kommentaariks loetakse rea lõpuosa, mis algab kolme miinusmärgiga. Skeemi algus- ja lõpureas paikneva, aga ka noolele vahetult järgneva kommentaari võib kirjutada ilma algusmiinusteta.

Algoritmis kasutatavaid tähtsamaid abimuutujaid iseloomustatakse üksnes kommentaarides, muutujate rangeid tüübikirjeldusi ei esitata.

Skeemkeelse algoritmi näide:



Peatükk 1

Algoritmi ajaline keerukus

Algoritmi hindamise üheks olulisemaks kriteeriumiks on algoritmi töökiirus, mis avaldub konkreetse ülesande lahendamiseks vajalike sammude (algoritmis ettenähtud elementaaroperatsioonide) arvu kaudu. Viimane sõltub antud algandmete omadustest, muuhulgas ka nende mahust. Algoritmi ajalise tõhususe määramiseks uuritaksegi eeskätt just sammude arvu sõltuvust algandmete mahust, sest reeglina õnnestub ülesande algandmete mahtu väljendada lihtsalt naturaalarvuna. Näiteks järjenditöötlusülesannete mahtu iseloomustab antud järjendi pikkus, graafitöötluse puhul – vaadeldava graafi tippude (ja/või) kaarte arv jne. Vaatleme esialgu algoritme, mille korral sooritatavate sammude arv sõltub ainult lahendatava ülesande andmemahust.

Algoritmi **ajalist keerukust** väljendab funktsioon f, mis igale selle algoritmi järgi lahendatavale konkreetsele ülesandele andmetemahuga n seab vastavusse selle lahendamisel sooritatavate algoritmi sammude arvu $f(n)$.

Algoritmi ajalise keerukuse hinnang ongi aluseks algoritmi kiiruslike omaduste kirjeldamisel.

Algoritmide realiseerimisel saadavate arvutiprogrammide töökiiruse vahetu teoreetiline hindamine on väga keeruline, sõltudes suurest hulgast praktilistest detailidest. Vastavate algoritmide ajalise keerukuse teadmine aga annab lihtsa võimaluse ka nende alusel koostatud programmide võrdlemiseks. Reeglina võib eeldada, et programmi töö aeg on vastava algoritmi ajalise keerukuse kordne $cf(n)$, kus c on konstant. Näiteks saab võrrelda erinevate algoritmide realisatsioone ühes ja samas arvutikeskkonnas, sealhulgas määra-

Tabel 1.1: Lahendamisaja suhteline kasvamine, kui algandmete maht suureneb 5-lt 25-le.

Programmi töö aeg $c\,f(n)$	Lahendamisaja suhteline suurenemine $f(25)/f(5)$
c_1	1
$c_2 \log n$	2
$c_3\, n$	5
$c_4\, n \log n$	10
$c_5\, n^2$	25
$c_6\, n^3$	125
$c_7\, 2^n$	1 048 576

ta ülesande lahendusaja kasvu sõltuvalt algandmete mahu kasvust (vt. tabel 1.1). Seejuures pole konkreetset realisatsiooni iseloomustavat kordajat c_i vaja teadagi.

Algoritmi ajalise keerukuse hinnang määrab ka algoritmi praktilise rakendatavuse piiri: väga kiiresti kasvava keerukusfunktsiooniga algoritmi tasub realiseerida vaid väikesemahuliste ülesannete ringi jaoks, kuna suuremate ülesannete lahendusaeg osutub praktiliselt lõpmatuks (vt. tabel 1.2).

Algoritmide ajalise keerukuse juures pakub suuremat huvi just vastavate funktsioonide käitumine algandmete mahu piiramatul kasvamisel, st. argumendi n küllalt suurte väärtuste korral – nn. **asümptootiline hinnang**. Nagu ka tabelist 1.1 näha, kasvavad funktsioonid kujul $c\,n \log n$, mis erinevad ainult kordaja c poolest, suhtelises mõttes kõik ühtemoodi (seejuures sõltumata kordajast c). Analoogiliselt, kõik funktsioonid kujul $c\,2^n$ kasvavad omakorda ühetaoliselt, kuid oluliselt kiiremini kui eespoolnimetatud. Seega funktsioonid kujul $c\,n \log n$ on asümptootiliselt ühe ja sama hinnanguga, öeldakse, et nad on $O(n \log n)$; teise nimetatud klassi kuuluvad aga funktsioonid kujul $c\,2^n$, nende kohta öeldakse, et nad on $O(2^n)$. Üldiselt on (positiivne) funktsioon $f(n)$ asümptootilise hinnanguga $O(g(n))$, kui küllalt suurte argumendi väärtuste korral $f(n) < cg(n)$, kus $c > 0$ on konstant. Näiteks mitte ainult $126n \log n$ on $O(n \log n)$, vaid ka $5n \log n + 100 \log n$

Tabel 1.2: Erineva keerukusega programmide ajalised piirid (eeldusel, et ühele operatsioonile kulub 1 mikrosekund).

Keerukus	Suurim ülesanne, mille lahendamise aeg < 1 sek.	Suurim ülesanne, mille lahendamise aeg < 1 päev	Suurim ülesanne, mille lahendamise aeg < 1 aasta
n	$n = 1\,000\,000$	$n = 86\,400\,000\,000$	$n = 31\,530\,000\,000\,000$
$n \log_2 n$	$n = 62\,746$	$n = 2\,755\,147\,514$	$n = 798\,160\,978\,500$
n^2	$n = 1000$	$n = 293\,938$	$n = 5\,615\,692$
n^3	$n = 100$	$n = 4\,421$	$n = 31\,593$
2^n	$n = 19$	$n = 36$	$n = 44$
$n!$	$n = 9$	$n = 14$	$n = 16$

on $O(n \log n)$. Tõepoolest, leidub konstant c, nii et teatud väärtusest $n = N$ alates $5n \log n + 100 \log n < cn \log n$:

$$(5n + 100) \log n < cn \log n\,,$$

$$5n + 100 < cn\,,$$

$$100 < n(c - 5)\,.$$

Seega võib võtta $c = 6$ ja $N = 101$. Järgnevas esitatakse O-relatsiooni mõiste rangem määratlus.

Definitsioon. Olgu f ja g naturaalarvuliste argumentidega funktsioonid. Siis f on $O(g)$ parajasti siis, kui leiduvad $c > 0$ ja $N > 0$ nii, et

$|f(n)| \leqslant c|g(n)|$ iga $n \geqslant N$ korral.

Teoreem. *Olgu f ja g naturaalarvuliste argumentidega ja positiivsete väärtustega funktsioonid. Siis f on $O(g)$ parajasti siis, kui leidub $c > 0$ nii, et $f(n) \leqslant c\,g(n)$ iga naturaalarvu n korral.*

Tõestus. Piisavus on ilmne.

Tarvilikkus. Kui f on $O(g)$, siis leiduvad c ja N vastava definitsiooni kohaselt, st.

$$f(n) \leqslant c\,g(n) \quad \forall n,\ n > N$$

Olgu $d = \max_{0<n\leqslant N}\{f(n)/g(n)\}$, siis

$$f(n) \leqslant d\,g(n) \quad \forall n,\ 0 < n \leqslant N\,.$$

Kokkuvõttes,

$$f(n) \leqslant c'\,g(n) \quad \forall n,\ n > 0\,,$$

kus $c' = max\{c, d\}$. □

Edasises vaadeldavad algoritmide ajalised keerukused ja nende hinnangud vastavad kõik ülaltoodud teoreemi eeldustele, seega võime piirduda määratlusega: f on $O(g)$ parajasti siis, kui leidub $c > 0$ nii, et $f(n) \leqslant cg(n)$ iga naturaalarvu n korral.

Paneme tähele, et O-relatsioon ei tähenda mitte ühe funktsiooni väärtuste tõkestatust teise funktsiooni väärtustega, vaid nende funktsioonide väärtuste suhte (funktsioonide suhtelise kasvu) tõkestatust. Kui f on $O(g)$, siis järelikult leidub $c > 0$, nii et $f(n)/g(n) < c$, st. funktsiooni f suhteline kasv funktsiooni g suhtes on tõkestatud; teiste sõnadega, argumendi n piiramatul kasvamisel $f(n)/g(n)$ ei kasva piiramatult. Kuna on tegemist tõkestatusega „ülalt“, siis saab taolisi funktsioonide asümptootilisi võrdlemisi teha suvalise „liiaga“. Näiteks funktsioon $3n^2 + 1$ on nii $O(n^2)$ kui ka $O(n^3)$ ja $O(n^4)$ ning isegi $O(2^n)$. On ju suhted $(3n^2+1)/n^2$, $(3n^2+1)/n^3$, $(3n^2+1)/n^4$ ja $(3n^2+1)/2^n$ kõik tõkestatud ülalt (näiteks arvuga 5), kui $n \geqslant 1$.

Loomulikult pakuvad rohkem huvi võimalikult täpsed (ja samas ka võimalikult lihtsad) hinnangud. Eriti täpselt iseloomustab mingi funktsiooni f kasvamist selline funktsioon g, et f on $O(g)$ ja g on $O(f)$ (sellisel puhul öeldakse, et f on $\Theta(g)$). Nimelt osutub siis funktsiooni f suhteline kasv tõkestatuks ka altpoolt: kui f on $\Theta(g)$, siis

$$f \ on\ O(g) \ \Rightarrow\ f/g < c\ ,$$

$$g \ on\ O(f) \ \Rightarrow\ g/f < c'\ ,$$

kus c ja c' on positiivsed konstandid. Järelikult

$$(1/c') < f/g < c\ .$$

Teiste sõnadega, kui f on $\Theta(g)$, siis funktsiooni f kasvamine funktsiooni g suhtes on küll piiratud, kuid samal ajal ei jää f väärtused g väärtustest

ka lõpmatult palju maha. Nii on ülal näitena vaadeldud funktsioonide suhete korral vaid esimene neist alt tõkestatud: $3 < (3n^2+1)/n^2 < 5$. Seega $3n^2+1$ on $\Theta(n^2)$, kuid $3n^2+1$ ei ole $\Theta(n^3)$.

Seos „g on $O(f)$“ tähendab ühtlasi seda, et funktsiooni f kasvamine on tõkestatud altpoolt funktsiooni g kasvuga. Sel puhul öeldakse, et f on $\Omega(g)$.

Algoritmi ajalise keerukuse asümptootilisel hindamisel on kasulik teada O ja Θ relatsioonide järgmisi omadusi.

1. Iga $k > 0$ korral, kf on $O(f)$. Konstantsed kordajad ei mängi seega mingit rolli. Erijuhtudel, f on $O(f)$ ja k (st. konstantne funktsioon) on $O(1)$.

2. Kui f on $O(g)$ ja h on $O(g)$, siis $(f + h)$ on $O(g)$. Erijuhul, kui f on $O(g)$, siis $(f + g)$ on $O(g)$.

3. Kui f on $O(g)$ ja g on $O(h)$, siis f on $O(h)$.

4. n^r on $O(n^s)$, kui $0 \leqslant r \leqslant s$.

5. Kui p on d-astme polünoom, siis p on $\Theta(n^d)$.

Näiteks $7n^5 + 2n^2 - 1$ on $\Theta(n^5)$.

6. Kui f on $O(g)$ ja h on $O(r)$, siis $(f \cdot h)$ on $O(g \cdot r)$.

7. n^k on $O(b^n)$, kui $b > 1$, $k \geqslant 0$. Näiteks, n^5 on $O(2^n)$.

8. $\log_b n$ on $O(n^k)$, kui $b > 1$, $k > 0$. Erijuhul, $\log n$ on $O(n)$.

9. $\log_b n$ on $\Theta(\log_d n)$ iga $b, d > 1$ korral.

10. $\sum_{k=1}^{n} k^r$ on $\Theta(n^{r+1})$.

Omadus 5 võimaldab anda lihtsaid ja täpseid hinnanguid kõigile polünoomina avalduvatele keerukusfunktsioonidele. Algoritmi, mille ajaline keerukus on $O(n^d)$, kus d on täisarv, nimetatakse **polünomiaalse keerukusega algoritmiks**. Sellised algoritmid moodustavad väga tähtsa klassi, kuna ülejäänud (nendest ajaliselt keerukamad) algoritmid osutuvad vähegi mahukamate algandmete puhul juba lootusetult aeglasteks. Ülesandeid, mille jaoks ei ole teada polünomiaalse keerukusega algoritmi, nimetataksegi **raskelt lahenduvateks ülesanneteks**.

Üllataval kombel leidub hulganisti selliseid raskelt lahenduvaid ülesandeid, mille kohta ei ole õnnestunud veel tõestada, et nad oleksid mittepolünomiaalsed. Teiste sõnadega, on olemas palju ülesandeid, mille jaoks ei ole leitud polünomiaalse keerukusega algoritme ega ole suudetud ka tõestada, et neid ei leidugi. Raskelt lahenduva ülesande näitena võiks nimetada nn. **seljakoti-**

ülesannet: antud n eseme hulgast valida välja selline esemete komplekt, mille koguhind oleks võimalikult suur, kuid mille kogukaal ei ületaks etteantud konstanti (seljakoti lubatud kaalu). Seljakotiülesande näol on tegemist üsnagi tüüpilise raskelt lahenduva ülesandega, kus lahenduse saab leida algandmete hulga kõigi alamhulkade läbivaatamise teel. Vastavas algoritmis joonisel 1.1 on alamhulkade süstemaatiliseks läbivaatamiseks rakendatud nn. maske.

Olgu n-elemendiline hulk X esitatud järjendina $x_1, x_2, \ldots, x_n$ ja tähistagu $B^{(n)}$ kõigi n-kohaliste kahendsüsteemi arvude hulka, st.

$$B^{(n)} = \{\underbrace{00\ldots000}_{n}, \underbrace{00\ldots001}_{n}, \underbrace{00\ldots010}_{n}, \ldots, \underbrace{11\ldots111}_{n}\}.$$

Hulga X alamhulga Y **maskiks** nimetatakse sellist n-bitilist kahendarvu $b_1 b_2 \ldots b_n \in B^{(n)}$, mille korral

$$b_i = \begin{cases} 1, & \text{kui } x_i \in Y \\ 0, & \text{vastasel korral,} \end{cases}$$

kus $i = 1, 2, \ldots, n$. Kuna igale alamhulgale vastab parajasti üks mask ja maskide arv $|B^{(n)}| = 2^n$, siis on see ka ühtlasi alamhulkade läbivaatamise tsükli (joonisel 1.1) kordamiste arvuks. Arvestades veel, et igal tsükli sammul tehtavate operatsioonide arv on $O(n)$, osutub seljakotiülesande sellise algoritmi ajaliseks keerukuseks $O(n2^n)$.

Algoritmi ajalise keerukuse asümptootilist hinnangut mõjutavad algoritmis (ja selle alamalgoritmides) leiduvad tsüklid, kusjuures üksnes need, mille kordamiste arv sõltub algandmete mahust n. Algoritmi ülejäänud osades esinevate elementaaroperatsioonide arv on n suhtes konstantne, seega $O(1)$.

Näiteks kahekordses tsüklis

```
⋆ i = 2, 3, ..., n
    ⋆ j = i − 1, i − 2, ..., 1
        S
```

täidetakse tsükli sisu S

$$1 + 2 + \ldots + (n-1) = n(n-1)/2 = 0.5n^2 - 0.5n$$

```
seljakott(n, k, h, m, a)
--- Antud: n eseme kaalud k = (k_1, k_2, ..., k_n)
---        ja hinnad  h = (h_1, h_2, ..., h_n)
---        ning seljakoti (sisu) lubatav piirkaal m;
---        järjend a = (a_1, a_2, ..., a_n) tulemuse  salvestamiseks
--- Tulemus: a_1, a_2, ..., a_n on esemete sellise valiku  (alamhulga) mask,
---          mille korral kaal on lubatud piires (⩽ m)  ja koguhind on
---          suurim

 --- Abimuutujad: järjend b = (b_1, b_2, ..., b_n) – jooksva valiku mask,
 ---              r – jooksva valiku koguhind,
 ---              s – senini leitud parima valiku koguhind
s := 0
⋆ ∀b, b ∈ B^(n)
  Σ_{i=1}^{n} b_i k_i ⩽ m  ?
   r := Σ_{i=1}^{n} b_i h_i;  --- valiku  koguhind
    --- kui valik on senisest hinnalisem, siis jätta meelde:
   [? (r > s)  s := r;  a := b
```

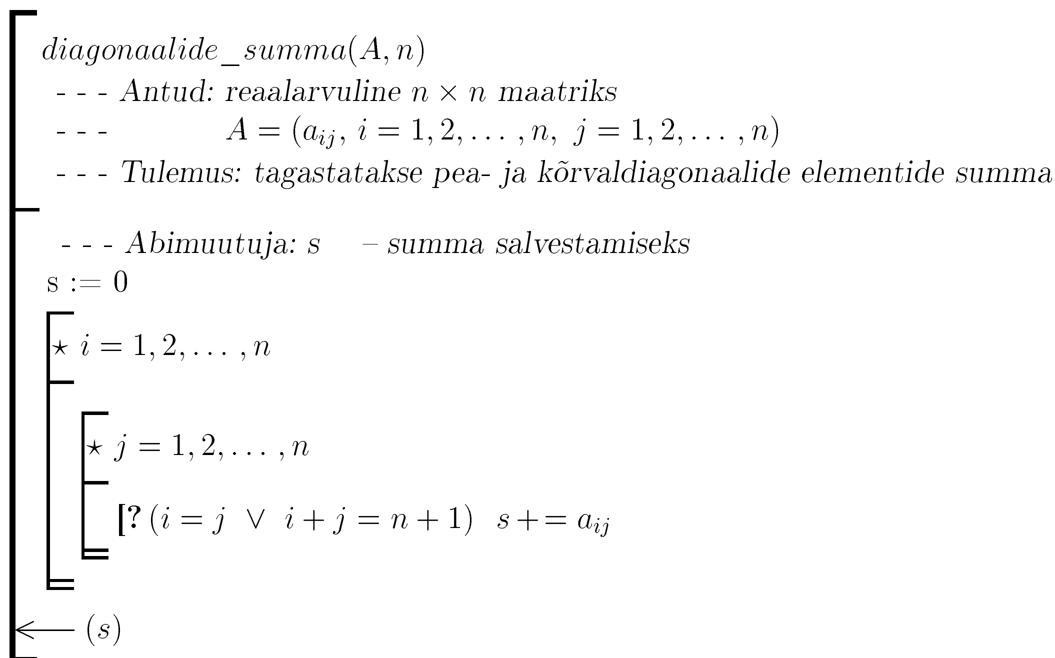
Joonis 1.1: Seljakotiülesande „jõumeetodil“ lahendamise algoritm.

korda. Kui sisus tehakse iga kord k_S ja ühe tsüklisammu organiseerimiseks veel k_0 elementaaroperatsiooni, siis selle kahekordse tsükli kogu ajaline keerukus on

$$f(n) = k_0(n-1) + (k_0 + k_S)(0.5n^2 - 0.5n)\,,$$

mis on $\Theta(n^2)$.

Tihtipeale võetakse keerukuse määramisel aluseks vaid teatavate **põhioperatsioonide** arv algoritmis. Näiteks joonisel 1.2 esitatud algoritmis (reaalarvulise ruutmaatriksi diagonaalidel paiknevate elementide summa leidmiseks) kõiki tehteid loendades saaksime n.ö. **kogukeerukuse** hinnanguks $\Theta(n^2)$.



Joonis 1.2: Maatriksi diagonaalide summa arvutamise algoritm kogukeerukusega $\Theta(n^2)$.

Kui aga arvestada põhioperatsioonidena ainult reaalarvude liitmisi ja lahutamisi, siis nende arv $f(n) = 2n$, mis on $\Theta(n)$. Antud ülesandele leidub muidugi ka algoritm lineaarse kogukeerukuse hinnanguga (vt. joonis 1.3).

Antud konkreetse ülesande lahendamiseks vajalike algoritmisammude arv

```
maatriksi_diagonaalide_summa(A, n)
 - - - Antud: reaalarvuline n × n maatriks
 - - -          A = (a_ij, i = 1, 2, ..., n, j = 1, 2, ..., n)
 - - - Tulemus: tagastatakse pea- ja kõrvaldiagonaalide elementide summa

 - - - Abimuutuja: s    – summa salvestamiseks
 s := 0
 [⋆ (i = 1, 2, ..., n)  s += a_ii + a_i,n−i+1
 - - - paaritu n korral on "keskmine" element nüüd summas topelt
 [? (n  on paaritu) s −= a_(n+1)/2,(n+1)/2
 ←— (s)
```

Joonis 1.3: Maatriksi diagonaalide summa arvutamise algoritm kogukeerukusega $\Theta(n)$.

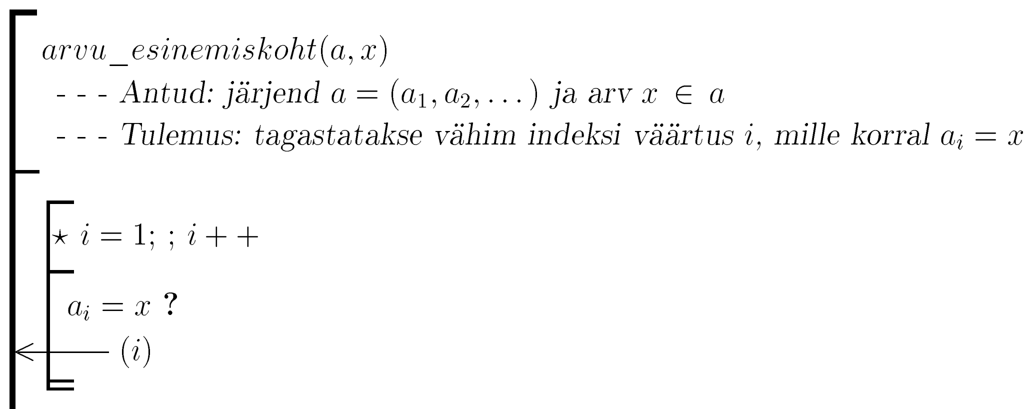
võib paljudel juhtudel sõltuda mitte ainult andmemahust n, vaid ka etteantava andmekomplekti teatavatest iseärasustest. Näiteks mingi arvu x otsimisel (joonisel 1.4 toodud algoritmi järgi) sorteerimata järjendist pikkusega $n = 100$ saame heal juhul otsitava kätte esimesel sammul (kui $a_1 = x$), halvimal juhul aga tuleb teha 100 sammu (kui $a_{100} = x$). Keskmiselt (üle kõigi 100-elemendiliste kordusteta järjendite) kulub otsimiseks 50 sammu iga järjendi korral.

Algoritmi **keskmiseks ajaliseks keerukuseks** $A(n)$ nimetatakse operatsioonide arvu, mida tuleb sooritada keskmiselt ühe konkreetse (andmemahuga n) ülesande lahendamiseks.

Algoritmi **ajaliseks keerukuseks halvimal juhul** $W(n)$ nimetatakse operatsioonide suurimat arvu, mida tuleb sooritada konkreetse (andmemahuga n) ülesande lahendamiseks.

Seega sorteerimata järjendist otsimise algoritmil (vt. joonis 1.4) $A(n) = n/2$ ja $W(n) = n$. Antud juhul nende funktsioonide asümptootilised hinnangud langevad kokku: mõlemad on $\Theta(n)$.

Kui iteratiivse algoritmi ajalise keerukuse hindamiseks leitakse tsükli(te) kordamise käigus sooritatavate põhioperatsioonide arv, siis rekursiivse algo-



Joonis 1.4: Lineaarse otsimise algoritm.

ritmi täitmiseks kuluv aeg avaldatakse rekurrentse võrrandina. Näiteks algoritmi puhul, kus lähteülesanne (mahuga n) jaotatakse kaheks alamülesandeks (mahtudega $n/2$), mõlemad lahendatakse samal meetodil ja seejärel kogu ülesande lahenduse saamiseks kulub veel n sammu (alamülesannete lahenduste töötlemiseks), avaldub lahendusaeg $T(n)$ järgmise rekurrentse võrrandina:

$T(n) = 2T([n/2]) + n$ (iga $n \geqslant 2$ korral) ja $T(1) = 0$.

Rekurrentsete võrrandite lahendamise erivõtete kõrval saab paljudel juhtudel rekursiivse algoritmi rekurrentselt avaldatud lahendusaega hinnata järgmist **põhiteoreemi** kasutades.

Teoreem. *Olgu* $a \geqslant 1$ *ja* $b > 1$ *konstandid ja* $f(n)$ *funktsioon ning olgu* $T(n)$ *mittenegatiivsete* n *väärtuste korral defineeritud valemiga*

$$T(n) = aT([n/b]) + f(n)$$

(kus nurksulud tähistavad täisosa). Siis

1. $T(n)$ *on* $\Theta(n^{\log_b a})$, *kui* $f(n)$ *on* $O(n^{\log_b a - \epsilon})$ *mingi positiivse konstandi* ϵ *korral;*

2. $T(n)$ *on* $\Theta(n^{\log_b a} \log n)$, *kui* $f(n)$ *on* $\Theta(n^{\log_b a})$,

3. $T(n)$ *on* $\Theta(f(n))$, *kui* $f(n)$ *on* $\Omega(n^{\log_b a + \epsilon})$ *mingi positiivse konstandi* ϵ *korral ning* $af(n/b) \leqslant cf(n)$ *mingi konstandi* $c < 1$ *ja kõigi küllalt suurte* n *väärtuste korral.*

(Käesolevas me selle teoreemi tõestust ei esita.)

Kuna ülal näitena toodud rekurrentse seose puhul $a = b = 2$, $\log_b a = 1$ ja kuna funktsioon $f(n) = n$ on $\Theta(n^1)$, siis teoreemi teise väite kohaselt $T(n)$ on $\Theta(n \log n)$. Üldiselt on hinnanguks „suurem“ funktsioonidest $f(n)$ ja $n^{\log_b a}$. Tõsi küll, juhul 1 nõutakse, et $f(n)$ oleks ülalt tõkestatud koguni funktsiooniga $n^{\log_b a}/n^{\epsilon}$, ja juhul 3, et $f(n)$ oleks alt tõkestatud funktsiooniga $n^{\log_b a} n^{\epsilon}$.

Põhiteoreem ei pruugi anda vastust iga $f(n)$ korral. Näiteks, kui

$$T(n) = 2T([n/2]) + n \log n$$

(st. $a = b = 2$), siis kuigi $f(n) = n \log n$ on alt tõkestatud funktsiooniga $n^{\log_b a} = n$, ei ole ta seda ühegagi funktsioonidest $n^{\log_b a} n^{\epsilon} = n^{1+\epsilon}$.

Küll aga saab hõlpsasti leida näiteks joonisel 1.5 kujutatud rekursiivse kahendotsimise algoritmi ajalise keerukuse hinnangu. Nimelt on lahendatava ülesande maht $n = j - i + 1$ ning rekursiivne rakendamine toimub üks kord, kusjuures alamülesanne on poole väiksema mahuga. Seega saame lahendusaja $T(n)$ jaoks antud juhul rekurrentse seose

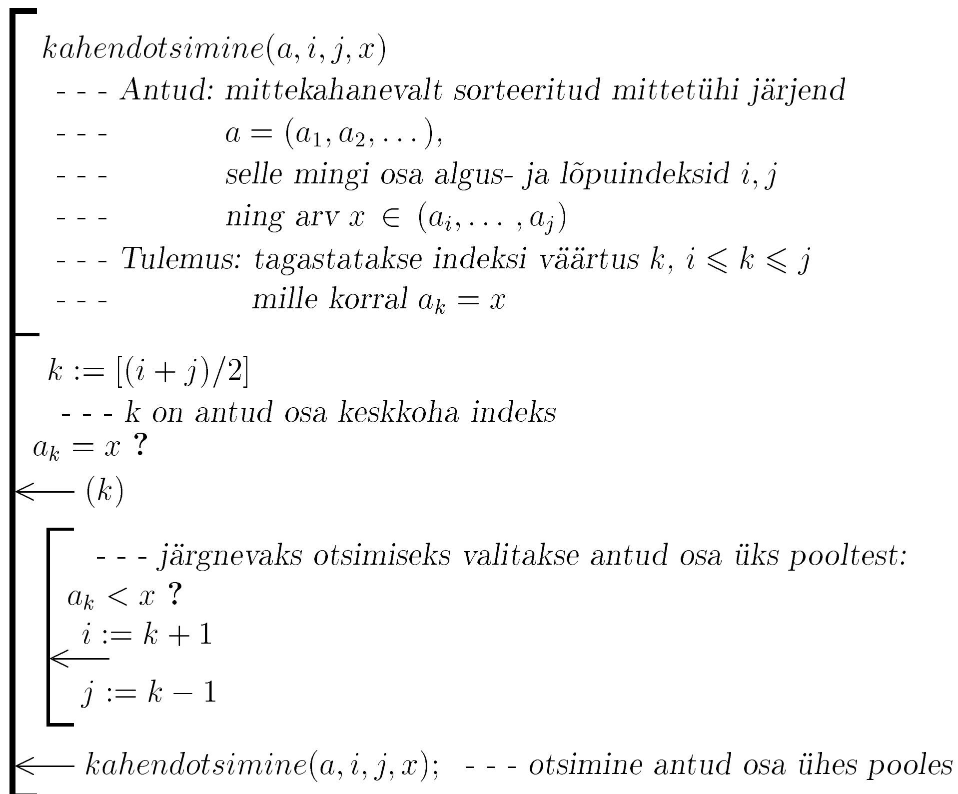
$$T(n) = T([n/2]) + f_0,$$

kus f_0 on rekursiivsele rakendamisele (algoritmi viimases reas) eelnevate operatsioonide arv. Viimane ei sõltu ülesande mahust n, seega f_0 on $\Theta(1)$. Kuna kordajad rekurrentses seoses $a = 1$ ja $b = 2$, siis $\log_b a = 0$ ja $n^{\log_b a} = 1$. Kuna f_0 on $\Theta(n^{\log_b a})$, siis saab rakendada põhiteoreemi teist väidet, mille kohaselt $T(n)$ on $\Theta(n^{\log_b a} \log n)$ ehk $T(n)$ on $\Theta(\log n)$.

Joonistel 1.4 ja 1.5 esitatud algoritmide ajalise keerukuse hinnangud, vastavalt $\Theta(n)$ ja $\Theta(\log n)$, näitavad ka seda, kuivõrd oluliselt aeglasem on lineaarne otsimine (sorteerimata järjendis) kahendotsimisest (sorteeritud järjendis). Kasvab ju funktsioon $f(n) = n$ märksa kiiremini, kui näiteks funktsioon $g(n) = \log_2 n$ (vt. ka tabel 1.3).

Tabel 1.3: Ajalise keerukuse hinnangufunktsioonide väärtusi.

n	1	16	128	512	1024	2048	4096	65536
$\log_2 n$	0	4	7	9	10	11	12	16



Joonis 1.5: Kahendotsimise algoritm.

Peatükk 2

Andmestruktuurid

2.1 Tüübi mõiste

Teatud mõttes ühesugused andmed moodustavad andmetüübi. Ühte tüüpi andmeid iseloomustab eeskätt see, et kõigile seda tüüpi väärtustele saab rakendada ühtesid ja samu operatsioone. Seega võib **andmetüübiks** nimetada paari (V, O), kus V on tüübi väärtuste hulk ehk **väärtusvaru** ja O on neile väärtustele rakendatavate operatsioonide hulk ehk **operatsioonivaru**. Näiteks loetakse mõnedes arvutiprogrammides täisarvutüübi väärtusvaruks kõigi 32-bitiliste kahendsõnede hulk, kusjuures operatsioonideks on siis aritmeetikakäsud niisuguste sõnede kui kahendsüsteemi täisarvudega manipuleerimiseks. Taolistes programmides võib reaalarvutüübi väärtusvaru isegi sama olla, kuid operatsioonideks selles on ujupunktaritmeetika käsud, milles 32-bitilisi kahendsõnesid interpreteeritakse sootuks teisiti.

Kõik operatsioonid tüübis ei pruugi olla kinnised väärtuste hulgal. Näiteks võrdlemise operatsiooni argumentideks on tavaliselt väärtused antud tüübi väärtusvarust, kuid tulemus kuulub hoopis loogilisse tüüpi. Viimane on kaheelemendilise väärtusvaruga $\{\mathit{false}, \mathit{true}\}$ (mõnedes programmeerimiskeeltes ka $\{0, 1\}$) ning reeglina kinnine oma (loogiliste) operatsioonide suhtes.

Tüübid jagunevad liht- ja struktuurseteks tüüpideks. Erinevalt lihttüübist kuuluvad **struktuurse tüübi** operatsioonivarusse veel operatsioonid väärtuse **komponentidega**. Struktuurset tüüpi nimetatakse ka **andmestruktuuriks**. Hästi tuntud struktuurseks tüübiks on massiiv programmeerimiskeeles.

Näiteks reaalarvulise kahedimensionaalse 4×5 massiivi kui andmetüübi väärtusteks on kõikvõimalikud 4×5 maatriksid. Väärtuse komponendiks on maatriksi element – väärtus teisest (reaalarvu) tüübist. Peamisteks operatsioonideks sellises massiivitüübis ongi komponendi lugemine ja salvestamine. Kui a on 4×5 massiiv, siis a_{ij} (avaldisena) on operatsioon, mis kolmikule (a, i, j) seab vastavusse väärtuse reaalarvutüübist. Omistamine $a_{ij} := b$ on operatsioon, mis argumentide komplektile (a, i, j, b) seab vastavusse väärtuse (4×5 maatriksi) selle tüübi väärtusvarust (ja omistab selle ka a uueks väärtuseks). Veelgi lihtsamaks struktuurseks tüübiks osutub kompleksarvutüüp (näiteks keeles Fortran), mille igal väärtusel on kaks reaalarvulist komponenti. Kirjetüüp aga on sellise andmestruktuuri näiteks, kus väärtuse komponendid ei pruugi olla ühte ja sama tüüpi.

Andmestruktuuri (andmetüüpi) nimetatakse **dünaamiliseks**, kui tema väärtusvarusse kuulub erineva komponentide arvuga väärtusi; vastasel korral on andmestruktuur **staatiline**. Kõik ülaltoodud näitestruktuurid – massiiv, kompleksarv, kirje – kujutavad endast staatilisi andmetüüpe. Käesolevas huvitavad meid peaasjalikult just dünaamilised andmestruktuurid.

Operatsiooni, mille tulemusena argumendiks olevast (mittetühjast) väärtusest saadakse ühe võrra väiksema komponentide arvuga väärtus, nimetatakse eemaldamiseks. Praktiliselt on aga tähtsamaks nn. **võtmise** operatsioon, kus koos eemaldamisega toimub ka eemaldatava komponendi väärtuse salvestamine. Algoritmides esitame võtmise operatsiooni kujul

$x \Leftarrow Q,$

kus Q on struktuurset tüüpi objekt ja x eemaldatava komponendi tüüpi muutuja. Selle operatsiooni käigus (1) andmestruktuurist Q eemaldatakse üks komponent (milline, see sõltub Q definitsioonist) ja (2) eemaldatud komponendi väärtus omistatakse muutujale x.

Lisamise operatsiooni tähistame

$Q \Leftarrow x,$

kus Q on struktuurset tüüpi objekt ja x lisatava komponendi tüüpi avaldis. Selle operatsiooni käigus andmestruktuurile Q lisatakse üks komponent (kuidas täpselt, see sõltub Q definitsioonist), mille väärtuseks saab avaldise x väärtus. Võtmine ja lisamine on analoogilised omistamistehtega: vasakul pool tehtemärki seisab muudetava objekti tähis, paremal aga väärtust mää-

rav avaldis.

Edasises nimetame andmestruktuuri komponente **tippudeks**. Reeglina eeldame, et tipu väärtuse tüübiks on kirjetüüp. Kui A on tipp, siis $A.x$ tähistab selles tipus oleva kirje välja nimega (identifikaatoriga) x. Kirjutist $.x$ kasutame veel rõhutamaks, et x näol on tegemist mingi kirje väljaga.

Kaasaegsetes programmeerimiskeeltes kasutatav klassi mõiste on kirjetüübi mõiste laienduseks. Nimelt võimaldab klass kui ühtne keelekonstruktsioon esitada kompaktselt vastava tüübi (klasstüübi) nii väärtusvaru kui ka operatsioonivaru. Oma liigilt esindab klass aga ikkagi staatilist andmestruktuuri. Dünaamilise andmestruktuuri realiseerimine mingis programmeerimiskeeles tähendab eeskätt otsuse tegemist, kuidas vajalikku komponentide töötlemist kirjeldada antud keele staatilist laadi struktuuride abil.

Igasuguses tarkvaraarendamise protsessis on otstarbekohane hoiduda liiga varajasest realisatsiooniprobleemidega tegelemisest. Ka kõik vaadeldavad andmestruktuurid on soovitav esialgu määratleda **abstraktsete andmestruktuuridena**, spetsifitseerides nende väärtus- ja operatsioonivarud sõltumatult mistahes võimalikust realisatsioonist.

2.2 Magasin ja järjekord

Dünaamiliseks järjendiks nimetatakse andmetüüpi, mille väärtusvaruks on kirjete suvalise pikkusega järjendid. Tähtsaimad erijuhud on magasin ja järjekord.

Magasin M on dünaamiline järjend, mille korral

$M \Leftarrow r$

tähendab kirje r lisamist järjendi M lõppu viimaseks ja

$r \Leftarrow M$

tähendab järjendi M viimase kirje eemaldamist ja selle omistamist muutujale r.

Järjekord J on dünaamiline järjend, mille korral

$J \Leftarrow r$

tähendab kirje r lisamist järjendi J lõppu viimaseks, ja

$r \Leftarrow J$

tähendab järjendi J esimese kirje eemaldamist ja selle omistamist muutujale r.

Nii magasin kui ka järjekord on realiseeritavad nii, et lisamine ning võtmine toimuksid $O(1)$ ajaga. Kõrvuti ülaldefineeritud andmetüüpidega on algoritmides väga oluliseks abistruktuuriks eelistusjärjekord.

Eelistusjärjekord Q (võtme $.x$ järgi) on dünaamiline hulk, mille korral

$Q \Leftarrow r$

tähendab operatsiooni $Q := Q \cup \{r\}$, kuid

$r \Leftarrow Q$

tähendab, et võtta (eemaldada Q-st ja salvestada muutujasse r) tuleb selline Q tipp, mille korral $.x$ on minimaalne üle kõigi kirjete hulgas Q.

Eelistusjärjekorra realiseerimine on suhteliselt keerukas, eriti veel siis, kui tegemist on nn. **muutuvate eelistustega**. Viimane tähendab, et töötlemise käigus võidakse hulka Q teisendada ka temas olevate kirjete (sh. võtme $.x$) muutmise teel.

2.3 Puud

Puustruktuur on andmete organiseerimise üks tähtsamaid vorme, esindades kirjete hulga hierarhilist ülesehitust. Viimasest tulenevalt osutub tihtipeale otstarbekohaseks defineerida puudega seotud mõisted induktiivselt, puude töötlusalgoritmid aga esitada rekursiivsetena.

Puuks nimetatakse lõplikku tippude hulka, mis on kas tühi või milles üks tipp – **juur** ehk **juurtipp** – on välja eraldatud ning ülejäänud tipud on jaotatud $m \geqslant 0$ mittelõikuvaks alamhulgaks $T_1, T_2, \ldots, T_m$, millest igaüks on omakorda puu; alamhulkasid $T_1, T_2, \ldots, T_m$ nimetatakse puu juure **alampuudeks**. **Järjestatud puu** korral nõutakse, et juure alampuude hulk oleks lineaarselt järjestatud. Puude hulka nimetatakse **metsaks**. Andmete kujutamisel osutub aga veelgi olulisemaks kahendpuu mõiste.

Kahendpuuks nimetatakse lõplikku tippude hulka, mis on kas tühi või milles üks tipp – **juur** ehk **juurtipp** – on välja eraldatud ning ülejäänud tipud on jaotatud ülimalt kaheks mittelõikuvaks alamhulgaks T_v ja/või T_p, millest igaüks on omakorda kahendpuu; alamhulkasid T_v ja T_p nimetatakse vastavalt juure **vasakuks ja paremaks alampuuks**. Paneme tähele, et kahendpuu mõiste ei ole puu ega ka järjestatud puu erijuhuks. Nimelt, kui juurel on vaid üks alampuu, siis puu korral oleks selleks T_1, kahendpuu korral aga kas T_v või T_p. Sellele erinevusele vaatamata kehtivad järgmises lõigus määratletud mõisted nii puu kui ka kahendpuu korral.

Alampuu juurt nimetatakse puu juure **alluvaks** (ka vahetuks järglaseks). Tippu nimetatakse oma alluva **ülemuseks** (ka vahetuks eellaseks). **Leht** ehk **lehttipp** on alluvateta tipp. Tippu, mis pole leht, nimetatakse **vahetipuks** ehk **sõlmeks** (ka sisemiseks tipuks). Ühise ülemusega tippe nimetatakse **kolleegideks**. Tippu y nimetatakse tipu x **järeltulijaks**, kui leidub tee $x = t_0, t_1, \ldots, t_k = y$, kus iga i korral $(0 \leqslant i < k)$ t_{i+1} on t_i alluv. Tipp x on niisugusel juhul tipu y **eelkäija** ja tipp y asub tipust x **kaugusel** k. Puu tipu **astmeks** nimetatakse selle tipu alluvate arvu.

Ei puu ega kahendpuu tipud ole lineaarselt järjestatud. Seetõttu ülesannetes, kus iga tipu korral tuleb mingi protseduur sooritada parajasti üks kord, kerkib küsimus puu läbimise järjekorrast. Kolm võimalikku sellekohast algoritmi leiduvad joonisel 2.1, kus P on protseduur, mida tuleb iga tipu korral

rakendada. Eeldatakse, et P täitmine ei muuda läbitava puu struktuuri.

Kahendpuu tippude **eesjärjestuseks** (**keskjärjestuseks**, **lõppjärjestuseks**) nimetatakse järjendit, milles tipud esinevad algoritmiga *läbida_ eesjärjestuses* (*läbida_keskjärjestuses*, *läbida_lõppjärjestuses*) määratud töötlemisjärjekorras. Analoogiliselt määratletakse ka järjestatud puu tippude ees- ja lõppjärjestus.

Puu tippude i-nda **taseme** moodustavad tipud, mille kaugus puu juurest on i. Ühel ja samal tasemel asuvate tippude järjestuseks loetakse tavaliselt puu tippude (ees)järjestusega määratud osajärjestus. Kahendpuu on **täielik**, kui tema kõik lehed asuvad ühel ja samal tasemel ning kõikide vahetippude aste on 2. **Kompaktne** kahendpuu saadakse täielikust kahendpuust null või enama lehe eemaldamisega viimase taseme lõpust.

Kahendpuu ees- ja lõppjärjestuses läbimise algoritmid on kergesti teisendatavad ka järjestatud puu vastavas järjestuses läbimise algoritmideks.

Üldiselt on kahendpuu töötlemise algoritmid lihtsamad ja ka hõlpsamini realiseeritavad, võrreldes suvalise puu töötlemise meetoditega. Paljudel juhtudel osutub võimalikuks ülesande püstituses antud lähtepuu asemel piirduda talle vastava kahendpuu töötlemisega, esitades (kodeerides) antud puu kahendpuuna.

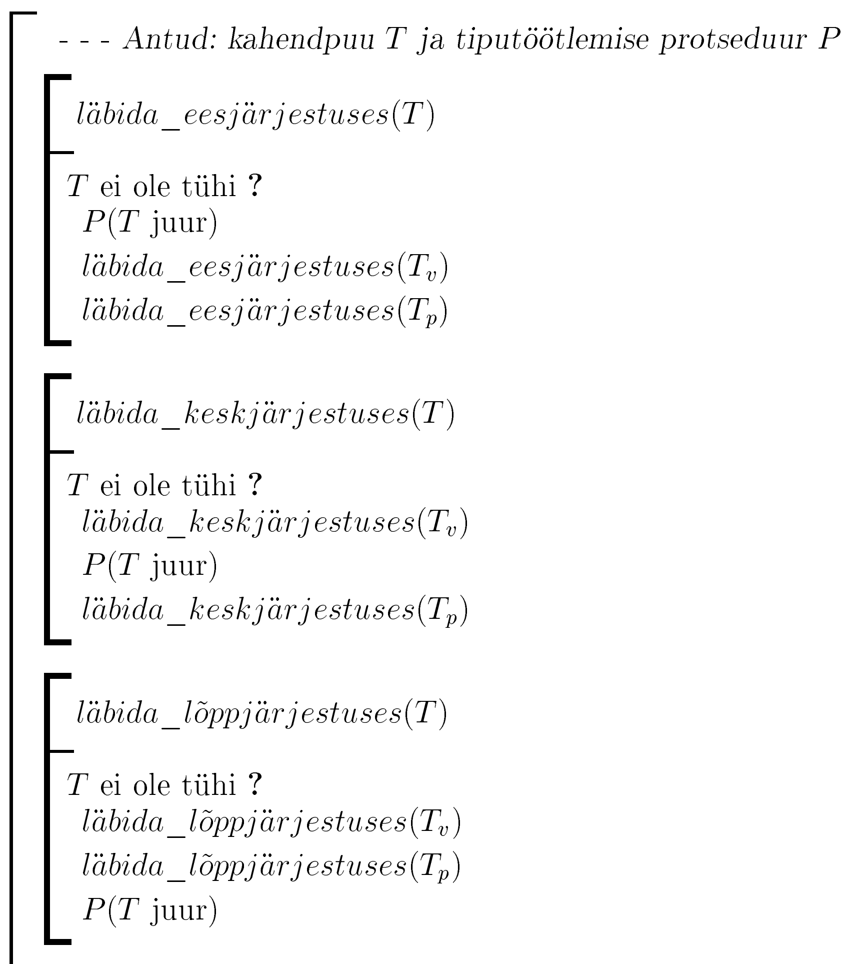
Järjestatud puule P **vastav kahendpuu** T koosneb puu P tippudest, tippude alluvad on aga määratud järgmiselt. Iga tipu $p \in T$ korral on p vasakuks alluvaks tipu p esimene alluv puus P, paremaks alluvaks aga tipule p järgnev kolleeg puus P; kui p on juur või viimane oma kolleegide seas, siis kahendpuus T tipul p parem alluv puudub.

Vaatleme nüüd puid, mille tippudeks on kirjed võtmeväljaga $.x$. Lihtsuse mõttes tähistame puu T juurtipus oleva kirje võtit $T.x$.

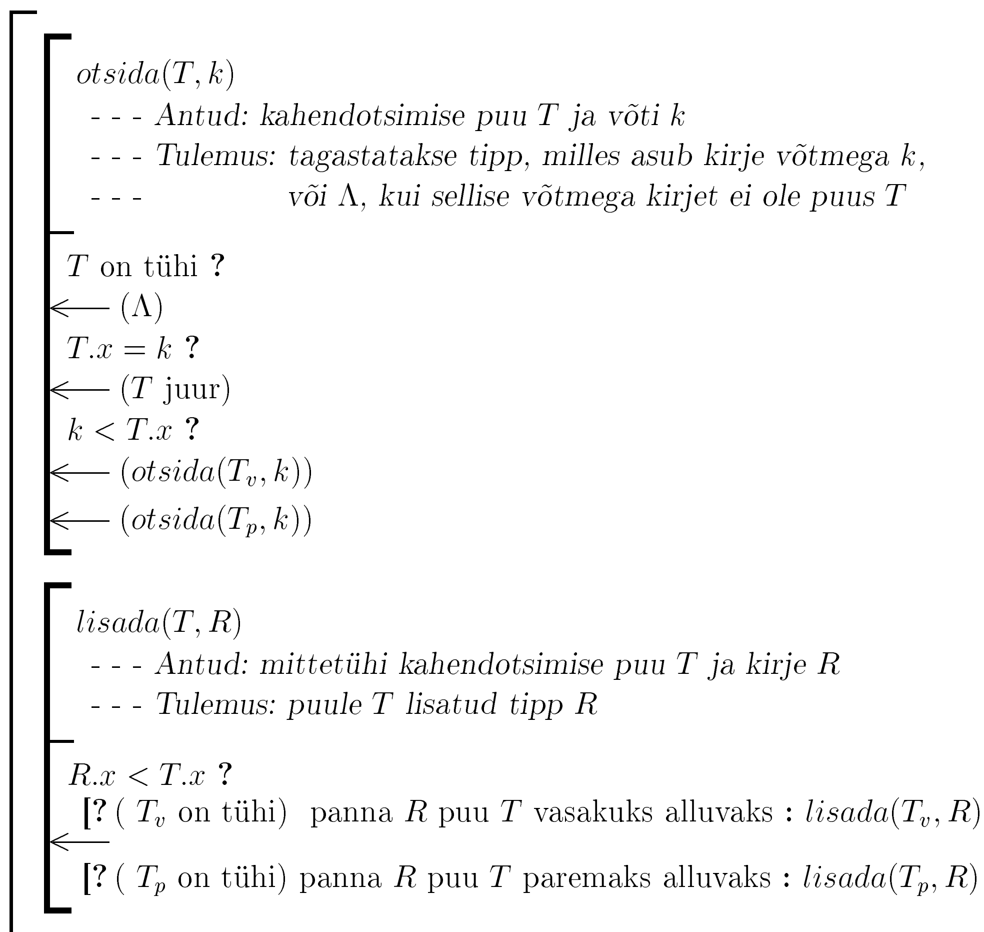
Kahendotsimise puu on kahendpuu T, mis

1. on tühi, või
2. a) T_v iga tipu t korral $t.x \leqslant T.x$;
   b) T_p iga tipu t korral $t.x \geqslant T.x$;
   c) T_v ja T_p on kahendotsimise puud.

Kahendotsimise puust kirje otsimise ja sinna uue kirje lisamise algoritmid on toodud joonisel 2.2. Kirjete hulga esitamine kahendotsimise puuna on sisuliselt sorteerimine.



Joonis 2.1: Kahendpuu läbimise algoritmid.



Joonis 2.2: Otsimise ja lisamise operatsioonid kahendotsimise puus.

Teoreem. *Kahendotsimise puu tipud keskjärjestuses on sorteeritud võtme x järgi mittekahanevalt.*

Tõestus: induktsiooniga tippude arvu järgi. Kui puus on üksainus tipp, siis väide kehtib triviaalselt. Eeldame, et i-tipulises puus kehtib: kui tipp s eelneb tipule t keskjärjestuses, siis $s.x \leqslant t.x$. Olgu T $(i+1)$-tipuline puu, milles tipp s eelneb tipule t keskjärjestuse mõttes. Võimalikud juhud:

1) kui s on juurtipp, siis t peab keskjärjestuse põhjal asuma paremas alampuus; kahendotsimise puu definitsiooni kohaselt siis $s.x \leqslant t.x$;

2) kui t on juurtipp, siis s peab keskjärjestuse põhjal asuma vasakus alampuus; kahendotsimise puu definitsiooni kohaselt siis $s.x \leqslant t.x$;

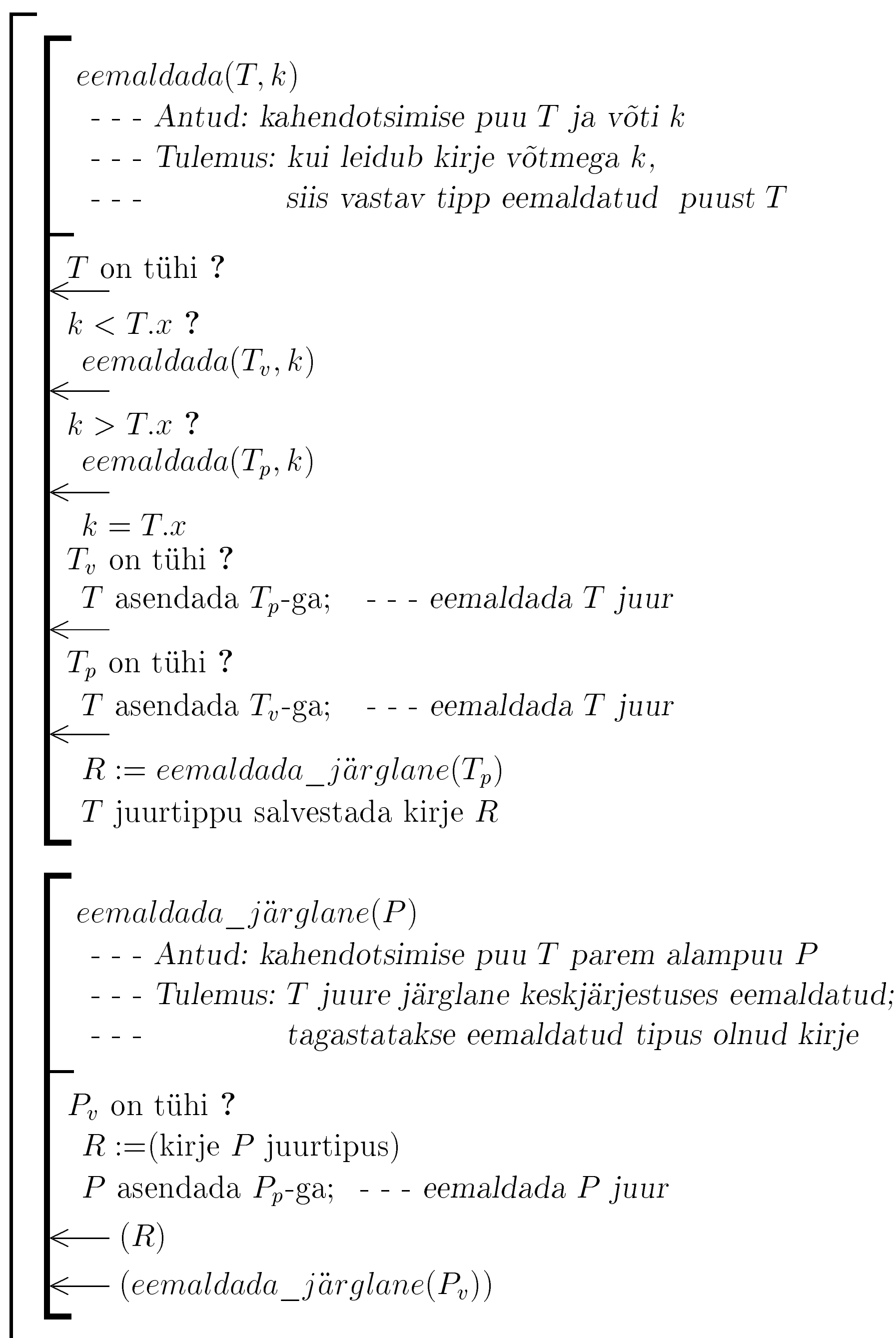
3) kui s ja t on samas alampuus, siis $s.x \leqslant t.x$ induktsiooni eelduse põhjal;

4) kui s ja t on erinevates alampuudes, siis keskjärjestuse põhjal s on vasakus ning t paremas alampuus; kahendotsimise puu definitsiooni kohaselt on $s.x \leqslant T.x$ ja $T.x \leqslant t.x$, seega $s.x \leqslant t.x$. □

Mõnevõrra keerulisem on aga tipu eemaldamine kahendotsimise puust. Olgu tarvis eemaldada tipp p. Juhul, kui tipul p ei ole alluvaid, võib selle tipu puust lihtsalt välja jätta. Kui tipul p on ainult üks alluv, siis saab tipu kõrvaldada, asendades alampuu, mille juureks on p, tipu p ainukese alampuuga. Rohkem aga tuleb vaeva näha siis, kui eemaldataval tipul on kaks alluvat. Niisugusel juhul leitakse kõigepealt tipu p järglane r keskjärjestuses, eemaldatakse hoopis see ning temas olnud kirje salvestatakse tippu p. Kahendpuu tippude vajalik järjestus, st. keskjärjestus, taolise operatsiooni käigus säilib. Antud võtmeväärtusega k kirje eemaldamise algoritm on täpsemalt kirjeldatud joonisel 2.3. Kahendpuu töötlemise operatsioonid on ajalise keerukusega $O(h)$, kus h on töödeldava puu kõrgus. Eespool kirjeldatud lisamis- ja eemaldamisoperatsioonide käigus puu kõrgus muutub kontrollimatult, st. sõltub vaid nende operatsioonide konkreetsest rakendamise järjekorrast. Tulemusena võib kahendotsimise puu omandada väga ebasoodsa, „väljavenitatud“ (tarbetult kõrge) kuju. Niisuguste juhtumite vältimiseks on välja töötatud mitmesuguseid erimeetodeid, kus lisamisel ja eemaldamisel sooritatakse lisateisendusi otsimispuu „tasakaalus“ hoidmiseks.

Öeldakse, et kahendpuu T on **tasakaalustatud**, kui

1. T on tühi või
2. T_v ja T_p on enam-vähem võrdse suurusega.



Joonis 2.3: Antud võtmega kirje eemaldamine kahendotsimise puust.

Tasakaalustatud puu mõistel põhinevad andmestruktuurid erinevad üksteisest peaasjalikult selle poolest, kuidas mõistetakse hinnangut „enam-vähem võrdse suurusega“. Tihtipeale nõutakse, et parem ja vasak alampuu oleksid ligikaudu sama kõrged. Kuna kahendpuu j-ndal tasemel asub ülimalt 2^j tippu (mida on kerge induktsiooni teel tõestada), siis kõikidel tasemetel kokku on ülimalt $n = \sum_{j=0}^{h} 2^j = 2^{h+1} - 1$ tippu, kus h on kahendpuu kõrgus. Järelikult kõrguste mõttes tasakaalus olevas kahendotsimise puus sooritatakse operatsioonid (lisada, eemaldada, otsida) ajaga $O(\log n)$.

AVL-puuks nimetatakse kahendotsimise puud T, mis

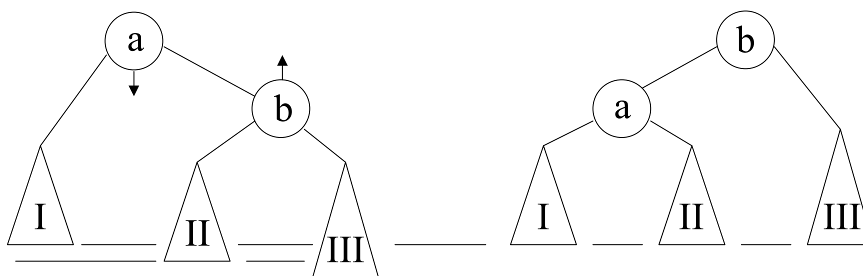
1. kas on tühi või

2. vasaku ja parema alampuu kõrgused erinevad ülimalt 1 võrra ja mõlemad on AVL-puud.

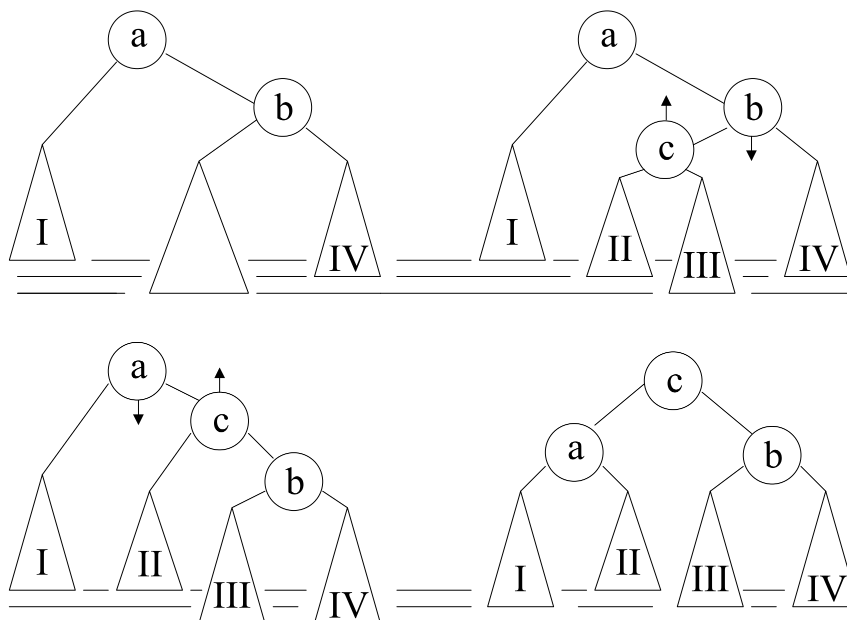
(Lühend AVL tuleneb autorite Adelson-Velski ja Landise nimedest.)

Paneme tähele, et vasaku ja parema alampuu kõrguste võrdsuse nõudmine oleks aga liig. Nimelt on see nõue iga tipu korral täidetud vaid täielikus kahendpuus. Kuid viimase tippude arv on kahe aste, mistõttu kaugeltki mitte iga võimsusega kirjete hulka ei saaks sellise „ülitasakaalus“ kahendotsimispuuna kujutada.

Otsimine AVL-puus toimub tavapärasel viisil, kuid lisamise ja eemaldamise operatsioonid on täiendatud puu tasakaalustamise operatsioonidega (vt. joonis 2.4 ja 2.5).



Joonis 2.4: AVL-puu tasakaalustamise võte.



Joonis 2.5: AVL-puu tasakaalustamise topeltvõte.

Otsimispuu kõrguse vähendamise teine tee on lubada paigutada igasse tippu rohkem kui üks kirje.

Öeldakse, et T on m-**rajaline otsimispuu**, kui

1. T on tühi, või

2. T koosneb juurest kirjete võtmete komplektiga $k_1 \leqslant k_2 \leqslant \ldots \leqslant k_j$ $(0 \leqslant j < m)$ ja alampuude järjendist $T_0, T_1, \ldots, T_j$ nii, et

a) kui k on mingi võti alampuus T_0, siis $k \leqslant k_1$;

b) kui k on mingi võti alampuus T_i, $0 < i < j$, siis $k_i \leqslant k \leqslant k_{i+1}$;

c) kui k on mingi võti alampuus T_j, siis $k \geqslant k_j$;

d) $T_0, T_1, \ldots, T_j$ on kõik kas mittetühjad m-rajalised otsimispuud, või kõik nad on tühjad.

Paneme tähele, et m-rajalise otsimispuu tipu suurim aste on m ning iga vahetipu aste on ühe võrra suurem selles tipus asuvate kirjete arvust. Mitmerajalised otsimispuud sobivad eriti hästi suuremate andmekogumite salvestamiseks arvuti välismälus, sest igale tipule saab vastavusse seada suurema,

korraga loetava/kirjutatava andmebloki. Kirjete töötlemisele võtmisel saab siis läbi väiksema arvu (suhteliselt aeglaste) välismäluoperatsioonidega.

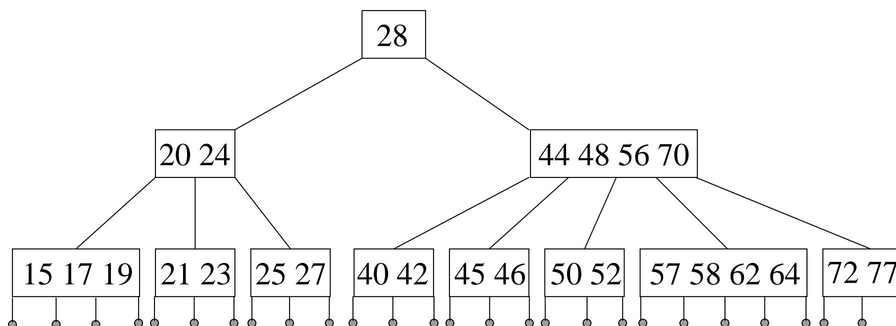
Enamkasutatavaks mitmerajalise tasakaalustatud otsimispuu vormiks on B-puu (autorid Bayer ja McGreigh):

m**-järku B-puuks** nimetatakse m-rajalist otsimispuud, mille korral

1. kõik lehed on samal tasemel ega sisalda kirjeid;

2. igal vahetipul peale juure on vähemalt $\lceil m/2 \rceil$ ja ülimalt m alluvat ning juure aste on vähemalt 2 ja ülimalt m.

Näide viiendat järku B-puu kohta on estitatud joonisel 2.6. B-puu lehed moodustavad fiktiivse taseme, edaspidi neid tühje lehti joonisel ei kujutata.



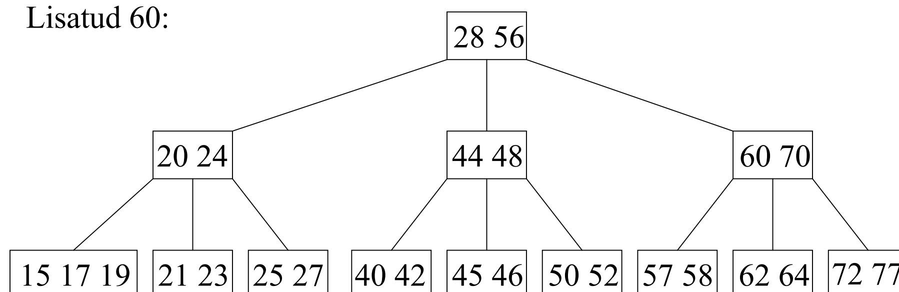
Joonis 2.6: Viiendat järku B-puu.

Kolmandat järku B-puud nimetatakse ka 2-3 puuks. B-puu on ideaalselt tasakaalustatud ja üsnagi väikese kõrgusega. Vaatleme m-järku B-puud kõrgusega h. Tasemel 0 paikneb 1 tipp, tasemel 1 vähemalt 2 tippu, edasi aga suureneb tippude arv igal tasemel vähemalt $m/2$ korda, kuna iga vahetipu aste on vähemalt $m/2$. Kui juurt mitte arvestada, siis tippude arv on vähemalt

$2 + \sum_{j=1}^{h-1} 2(m/2)^j = 2\sum_{j=0}^{h-1}(m/2)^j = 2((m/2)^h - 1)/((m/2) - 1)$.

Igas tipus on vähemalt $(m/2) - 1$ võtit, seega vaadeldava B-puu „mahutavus“ (kirjete arv puus) koos juurtipuga on vähemalt $1 + 2((m/2)^h - 1) = 2(m/2)^h - 1$. Näiteks neljatasemelise ja sajarajalise B-puu mahutavus on vähemalt $2 \times 50^{4-1} = 250000$ kirjet.

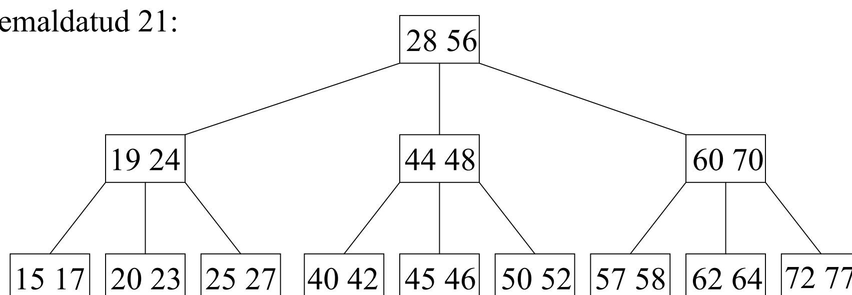
Otsimise protseduur B-puus on sirgjooneline: (alates juurest) otsitakse vaadeldavasse tippu kuuluvate kirjete seast, kui otsitavat selles tipus pole,



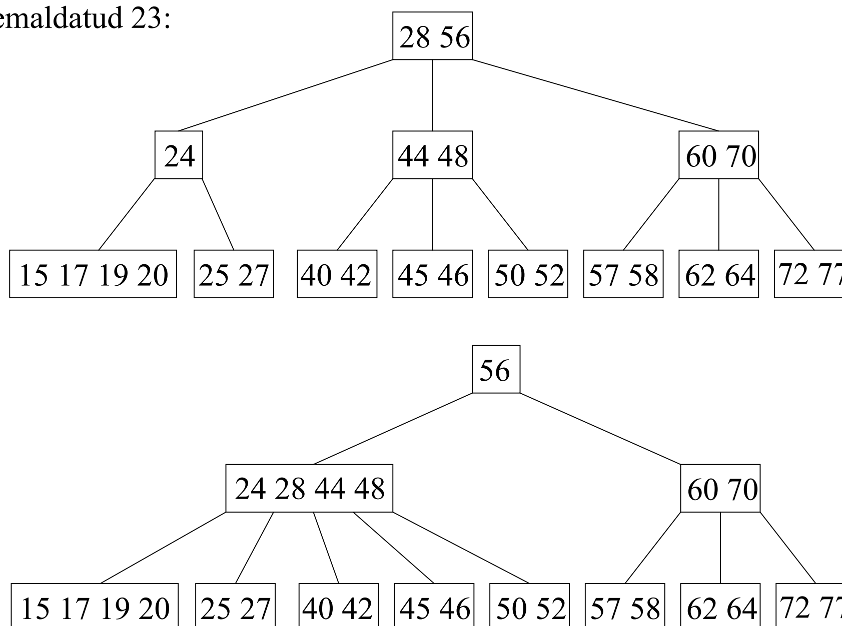
Joonis 2.7: B-puusse lisamisel poolitatakse ületäitunud tipp.

siis jätkatakse otsimist ühes alampuudest, mis leitakse vastavalt omadustele 2a, 2b, 2c m-rajalise otsimispuu definitsioonis. Leiduvad ka suhteliselt lihtsad lisamise ja eemaldamise operatsioonid, mis säilitavad B-puu struktuuriomadused. Lisatakse alati viimasele mittefiktiivsele tasemele; vajadusel „poolitatakse" ületäitunud tipp, viies sellest keskmise võtme ülemusse (mis omakorda võib kaasa tuua ülemuse poolitamise). Näide lisamisest joonisel 2.6 toodud B-puusse on esitatud joonisel 2.7.

Ka eemaldamist alustatakse alati viimaselt mittefiktiivselt tasemelt, sest vahetipust eemaldatava võtme saab eelnevalt asendada kas suurima võtmega lähimast vasakust alampuust või siis vähima võtmega lähimast paremast alampuust (mis kumbki asuvad kindlasti viimasel tasemel). Eemaldamisel võib tekkida tipu alatäitumus, mille kõrvaldamiseks tuleb „laenata" kolleegilt ülemuse kaudu või ka kolleege ühendada (vt. joonised 2.8, 2.9, 2.10).



Joonis 2.8: B-puust eemaldamisel laenatakse kolleegilt ülemuse kaudu.



Joonis 2.9: B-puust eemaldamisel ühendatakse kolleege.

Tutvume lõpuks veel sellise järjestatud puu liigiga, kus igas tipus võib olla vaid üks kirje ja mille tippude maksimaalne aste on piiratud, kuid mille kõrgus on sellegipoolest suhteliselt väike. Märgime, et piiramata tipuastmega järjestatud puu ei paku huvi, sest selle vähim kõrgus on triviaalselt 1: juurtipu alluvaks võib võtta kõik ülejäänud kirjed.

Binomiaalpuu B_k (kõrgusega k) defineeritakse järgmiselt:

1. B_0 on ühetipuline puu;

2. $B_k (k > 0)$ koosneb kahest binomiaalpuust B_{k-1}, millest üks on teise juurtipu esimene alluv. (Vt. ka joonis 2.11.)

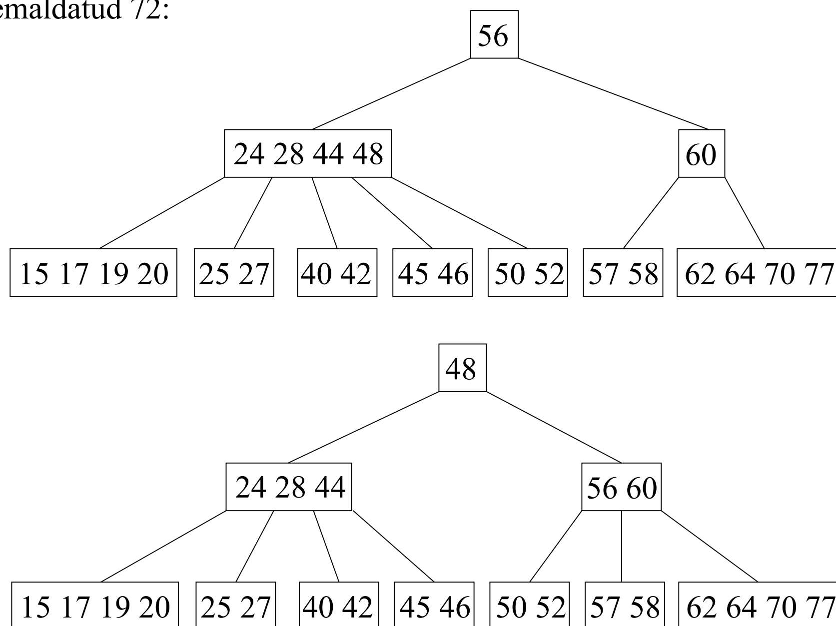
Lemma. *Binomiaalpuu* B_k

1. tippude arv on 2^k*;*

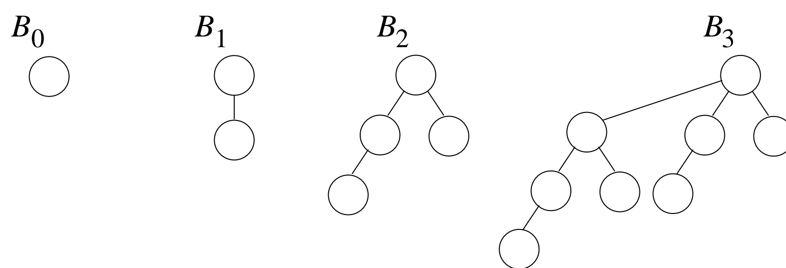
2. kõrgus on k*;*

3. tasemel i $(i = 0, 1, \ldots, k)$ *on parajasti* $\binom{k}{i}$ *tippu;*

4. juure aste on k *ning see on suurem kõigi juure järeltulijate astmetest; juure alampuudeks on binomiaalpuud* $B_{k-1}, B_{k-2}, \ldots, B_0$.



Joonis 2.10: B-puust eemaldamisel laenatakse kolleegilt ja ühendatakse kolleege.



Joonis 2.11: Binomiaalpuud.

Tõestus: induktsiooniga k järgi. Omadused 1-4 kehtivad B_0 korral triviaalselt. Eeldame, et lemma kehtib B_{k-1} korral. Siis

1. B_k koosneb definitsiooni kohaselt kahest binomiaalpuust B_{k-1}, seega B_k tippude arv on $2^{k-1} + 2^{k-1} = 2^k$.

2. B_k definitsiooni kohaselt üks tema osa B_{k-1} on teise osa B_{k-1} alampuuks, seega B_k kogukõrguseks on $(k-1)+1 = k$.

3. Olgu $D(k,i)$ tippude arv binomiaalpuu B_k i-ndal tasemel. B_k konstruktsiooni kohaselt on tema tasemel i niipalju tippe, kui palju on kokku tippe puus B_{k-1} tasemel i ja puus B_{k-1} tasemel $i-1$. Seega

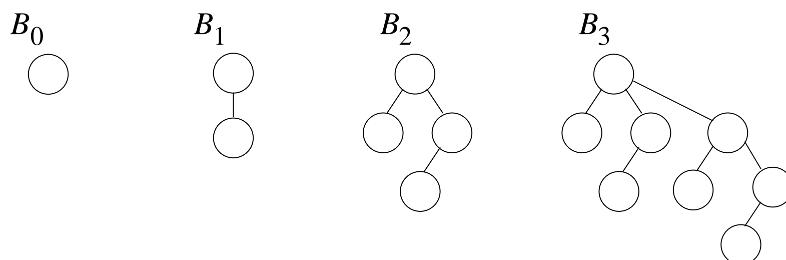
$$D(k,i) = D(k-1,i) + D(k-1,i-1) =$$

$$= \binom{k-1}{i} + \binom{k-1}{i-1} = \binom{k}{i}.$$

4. Ainus tipp, mille aste puus B_k on suurem puu B_{k-1} tippude astmetest, on B_k juur, millel on üks alluv rohkem kui B_{k-1} juurel. Viimasel on aga $k-1$ alluvat, seega B_k aste on k. Ja lõpuks, B_{k-1} juure alluvad on puude $B_{k-2}, B_{k-3}, \ldots, B_0$ juured (induktsiooni eelduse järgi). Puu B_k konstrueerimisel ühendatakse B_{k-1} ja B_{k-1}, seega B_k juure alluvateks on $B_{k-1}, B_{k-2}, \ldots, B_0$.

Järeldus. *n-tipulise binomiaalpuu tipu maksimaalne aste on* $\log n$. (Vahetult lemmast, omaduste 1 ja 4 põhjal.)

Märkus. Nn. peegeldatud binomiaalpuu $B_k (k > 0)$ koosneb kahest binomiaalpuust B_{k-1}, millest üks on teise juurtipu v i i m a n e alluv, vt. joonis 2.12. Ülaltoodud lemma kehtib ka peegeldatud binomialpuude korral, ainult B_k juure alluvate järjekord on nüüd vastupidine: $B_0, B_1, \ldots, B_{k-1}$.



Joonis 2.12: Peegeldatud binomiaalpuud.

2.4 Graafi mõiste

Erinevalt tavapärasest graafiteoreetilisest lähenemisviisist, kus aluseks võetakse orienteerimata graaf, lähtume siin just orienteeritud graafi mõistest. Põhjuseks on eeskätt asjaolu, et orienteeritud graafi jaoks leidub küllaltki loomulik ning arvutitöötluseks hästi sobiv ahelstruktuurne kujutusviis. Defineerime alljärgnevas mõned graafidega seotud lihtsamad mõisted, eeskätt terminoloogia sätestamiseks.

Öeldakse, et on antud **orienteeritud graaf** ehk lihtsalt **graaf** $G = (V, E)$, kui on antud mittetühi **tippude** hulk V ja **kaarte** hulk $E \subseteq V \times V$. Joonisel kujutatakse tippe punktidena ja kaart $(v, w) \in E$ noolena tipust v tippu w. Kaare (v, w) **algustipuks** on tipp v ja **lõpptipuks** tipp w; öeldakse ka, et tipp w on tipu v **naaber** ehk (vahetu) **järglane** ning tipp v on tipu w (vahetu) **eellane**. Graafi G mingi tipu v kõigi naabrite hulka tähistame Gv; selliste tippude hulka, millele v on naabriks, tähistame aga $G^{-1}v$. Antud tipust v väljuvate noolte arvu $|Gv|$ nimetatakse tipu v **väljundastmeks** ja sisenevate noolte arvu $|G^{-1}v|$ selle tipu **sisendastmeks**.

Graafi $G = (V, E)$ korral defineerime järgmised mõisted.

Paarikaupa erinevate tippude järjendit $(v_1, v_2, \ldots, v_k)$, kus $k \geqslant 1$ ja $(v_i, v_{i+1}) \in E$, $i = 1, 2, \ldots, k-1$, nimetatakse k-tipuliseks **elementaarteeks**. Erijuhuna loetakse järjend (v_1) ühetipuliseks teeks. Kui leidub tee tipust v tippu w, siis tippu w nimetatakse tipu v **järeltulijaks**, tippu v aga tipu w **eelkäijaks**; sel puhul öeldakse ka, et tipp w on **saavutatav** tipust v.

Tippude järjendit $(v_1, v_2, \ldots v_k, v_1)$, kus $(v_1, v_2, \ldots, v_k)$ on elementaartee ja $(v_k, v_1) \in E$, nimetatakse k-tipuliseks **elementaartsükliks**.

Kuna edasises piirdume graafides ainult selliste elementaarsete "marsruutide" vaatlemisega, siis jätame enamasti täiendi "elementaar-" ära, kasutades vastavalt termineid **tee** ja **tsükkel**.

Päristeeks nimetatakse teed, mis on vähemalt 2-tipuline, ja **päristsükliks** vähemalt 3-tipulist tsüklit.

Graafi $G = (V, U)$ nimetatakse **orienteerimata graafiks**, kui seos U on sümmeetriline, st. iga kaare $(v, w) \in U$ korral $(w, v) \in U$. Sümmeetrilist kaartepaari $\{(v, w), (w, v)\}$ nimetatakse **servaks** tippude v ja w vahel ning joonisel kujutatakse seda vastavaid tippe ühendava joonena (mitte noolena).

Peatükk 3

Andmestruktuuride realiseerimine

Andmestruktuuri realiseerimisel täpsustatakse/valitakse väärtusvaru elementide kujutusviis. Viimase alusel täpsustatakse ja realiseeritakse operatsioonivaru elementidele vastavad algoritmid. Kujutusviisi valikul arvestatakse eelkõige seda, et ajakriitilisi operatsioone realiseerivad algoritmid osutuksid võimalikult efektiivseteks. Tavaliselt on ajakriitilisteks just need operatsioonid, mida vastava andmehulga töötlemisel eriti sagedasti rakendatakse.

3.1 Järjestikpaigutus ja seotud paigutus

Kõige üldisemas plaanis võib andmestruktuuride kujutusviisid jaotada kaheks mõnevõrra vastandlikuks liigiks. Ühel juhul esitatakse andmed komponentide massiivina, milles juurdepääs üksikutele komponentidele toimub indeksi(te) järgi (reeglina $O(1)$ ajaga). Massiive kujutatakse arvutis kompaktsel mälualal, selle tõttu on taoline **järjestikpaigutus** mälu säästev: andmete organisatsiooni kirjeldamiseks kulub lisamälu vaid mahus $O(1)$. Teiselt poolt osutub aga järjestikpaigutuse rakendamine dünaamiliste andmestruktuuride korral üsnagi tülikaks, sest komponentide lisamise-eemaldamise operatsioonidega kaasneb peaaegu alati vajadus objekte mälus (massiivis) nihutada. Nihutamiseks kuluv aeg on aga $O(n)$, kus n on struktuuri komponentide arv.

Järjestikpaigutusele alternatiivseks strateegiaks on **seotud paigutus**, mille korral loobutakse andmete kompaktsest (massiivina) kujutamisest. Igale komponendile lisatakse vähemalt üks lisaväli – viit mingile teisele kompo-

nendile. Näiteks võib kirjete järjendi kujutada **lihtahelana**, kus ainuke lisaväli igas kirjes sisaldab viita järjendi järgmisele elemendile ja viimase kirje lisavälja väärtuseks on tühiviit Λ. Seotud paigutusviis sobib hästi just dünaamiliste andmestruktuuride kujutamiseks seetõttu, et komponentide lisamine ja eemaldamine on reeglina $O(1)$ ajalise keerukusega. Samas on aga ka ilmne, et selle eest tuleb maksta $O(n)$ lisamäluga viidaväljade jaoks. Võrreldes järjestikpaigutusega ilmneb veel teinegi miinus: puudub otsene juurdepääs üksikutele andmekomponentidele. Näiteks lihtahelas peab tegema $O(n)$ sammu leidmaks etteantud järjenumbriga ahela lüli.

Andmestruktuuride tegeliku realisatsiooni kavandamisel tuleb hoolikalt kaaluda paigutusviisi valikut, eeskätt just vastava andmetüübi operatsioonivaru silmas pidades ning arvestades ka üksikute operatsioonide kasutussagedust. Tihtipeale osutub sobivaks hoopis segapaigutusviis, kus ühed struktuuriosad kujutatakse massiivina, teised aga ahelatena.

Kahendpuu **naturaalesituseks** nimetame sellist kujutusviisi, mille korral igal tipul on kaks lisavälja alluvatele viitamiseks: ühel neist asub viit vasakule alluvale, teisel viit paremale alluvale; alluva puudumisel on välja väärtuseks tühiviit Λ. Naturaalesitus on üsnagi mäluraiskav kujutusviis, sest n-tipulise kahendpuu korral on tühiviitade arv $n+1$: ühetipulises kahendpuus „puudub“ kaks alluvat, iga uue lehe lisamisel suureneb „puudujate“ arv parajasti ühe võrra. Mälu kokkuhoiu huvides kasutatakse ka keerulisemaid esitusi, näiteks võetakse need viidaväljad, mis alluva puudumisel oleksid tühjad, tarvitusele mõneks muuks otstarbeks. Viimasel juhul läheb küll vaja veel ühebitilisi tunnuseid, mis iga välja korral näitaksid, mis otstarbel vastavat välja parajasti kasutatakse.

Olgu kahendpuu iga tipuga seotud väljad $.vasak$, $.vt$, $.parem$, $.pt$, kus $.vt$ on vasaku alluva ja $.pt$ parema alluva olemasolu tunnus. Kahendpuu **tagasisidestatud** ehk **traageldatud** esitus on määratud järgmiselt. Iga tipu t korral:

kui $t.vt = 1$, siis $t.vasak$ on viit tipu t vasakule alluvale;

kui $t.vt = 0$, siis $t.vasak$ on tipu t eellane kahendpuu tippude keskjärjestuse mõttes;

kui $t.pt = 1$, siis $t.parem$ on viit tipu t paremale alluvale;

kui $t.pt = 0$, siis $t.parem$ on tipu t järglane kahendpuu tippude keskjär-

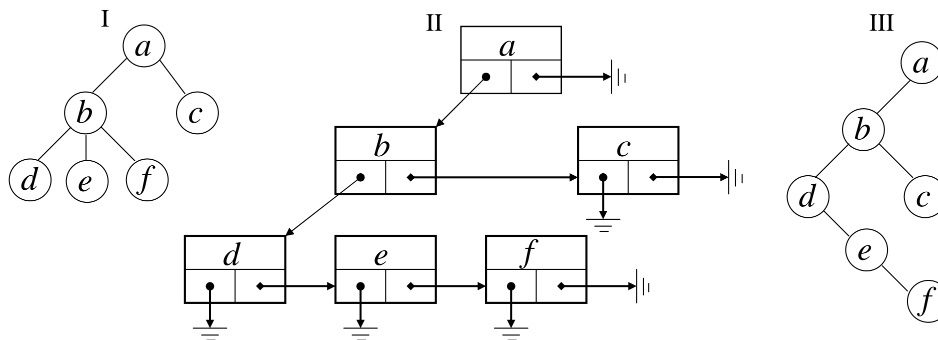
jestuse mõttes.

Taoliselt kujutatud kahendpuus saab muuseas hõlpsasti leida antud tipu t järglast keskjärjestuses:

kui $t.pt = 0$, siis järglaseks on $t.parem$;

kui $t.pt = 1$, siis järglaseks on viimane tipp tipust $t.parem$ algavas vasakute alluvate ahelas.

Järjestatud puu naturaalesituseks nimetame sellist kujutusviisi, mille korral igal tipul on samuti kaks lisavälja alluvatele viitamiseks: ühel neist asub viit esimesele alluvale (kui alluvaid ei ole, siis Λ), teisel viit järgmisele kolleegile (Λ viimase kolleegi ning juure korral). Sisuliselt on siin tegemist antud järjestatud puule vastava kahendpuu naturaalesitusega (vt. joonis 3.1).



Joonis 3.1: Puu (I) naturaalesitus (II) ja vastav kahendpuu (III).

Puu (metsa) esitamiseks tippude massiivina, st. järjestikpaigutuses, tuleb appi võtta komponentide eraldajad – lisasümbolid, mida tipukirjetes ei esine. Kui kasutada eraldajatena ümarsulge ja komasid, siis **metsa vasak suluesitus** on

tühi, kui mets on tühi, vastasel korral aga kirjutis kujul

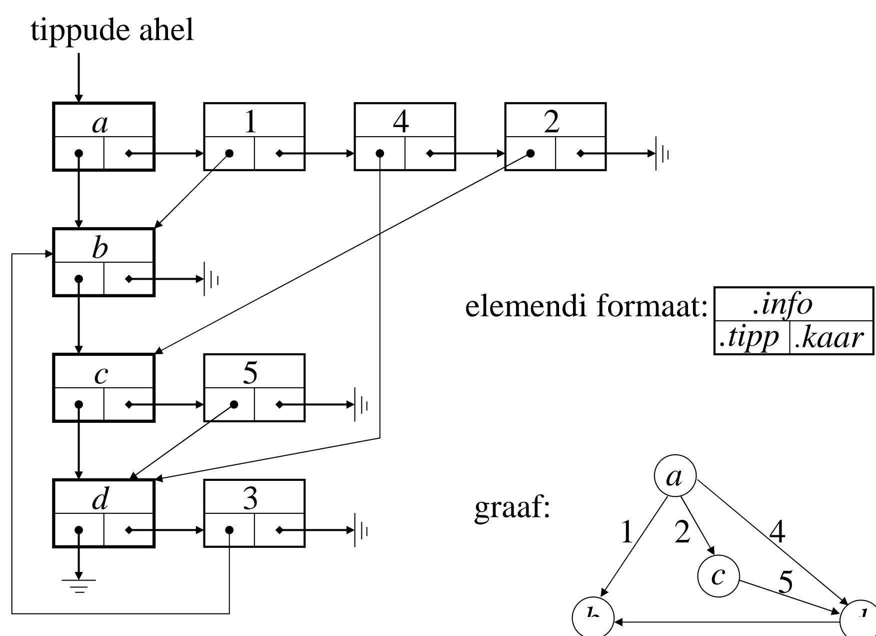
(metsa puude vasakud suluesitused komadega eraldatult).

Mittetühja puu vasak suluesitus koosneb juurest ja sellele järgnevast juure alampuude metsa vasakust suluesitusest.

Parema suluesituse korral paikneb juur oma alampuude metsa parema suluesituse järel.

Vasakus suluesituses paiknevad metsa tipud järjestuses, mis ühtib sellele metsale vastava kahendpuu tippude eesjärjestusega. Paremas suluesituses paiknevad metsa tipud järjestuses, mis ühtib sellele metsale vastava kahendpuu tippude keskjärjestusega.

Orienteeritud graafi seotud paigutuse puhul esitatakse tippude hulk lihtahelana, mis on seotud teatava viidavälja (*.tipp*) kaudu (vt. joonis 3.2). Igast tipuahela lülist algab vastavast tipust väljuvate kaarte ahel, mis on seotud mingi teise viidavälja (*.kaar*) kaudu. Iga kaarega on seotud veel viidaväli, millel asub viit vastava kaare lõpptipule, st. vastavale tipuahela lülile.



Joonis 3.2: Orienteeritud graaf ahelstruktuurina.

Järjestikpaigutuses esitatakse graaf nn. naabrusmaatriksina a, milles element a_{ij} kirjeldab kaart graafi i-ndast tipust j-ndasse tippu (või näitab selle puudumist).

Eeldades, et lugeja on juba tuttav massiivi ja ahela mõistetega ning vastavate töötlusvõtetega, vaatleme edasises mõningaid dünaamiliste andmestruktuuride kujutamise erivõtteid.

3.2 Paisksalvestus

Käesolevas jaotises kirjeldatav paisksalvestusmeetod sobib sellise dünaamilise hulga realiseerimiseks, kus põhioperatsioonideks on kirje lisamine ja (antud võtmega) kirje lugemine. Nende ajaline keerukus on siin keskmiselt $O(1)$. Seevastu kirje eemaldamine osutub raskendatuks, olles mõnel juhul $O(n)$. Põhiliselt ainult kasvava hulga näitena võib tuua programmi kompileerimise käigus koostatava ja kasutatava nimede (identifikaatorite) ning nende atribuutide tabeli.

Olgu tegemist kirjete hulgaga R, kus võtmete $.k$ väärtused kuuluvad hulka U ning $|R| \ll |U|$. Viimane tähendab, et kõikvõimalikke võtmeid on märksa rohkem, kui hulgas R üldse kirjeid saab olla. Näiteks võib R sisaldada mingi teaduskonna üliõpilaste andmekirjeid, kus võtmeks on matrikli number hulgast $U = \{10000, 10001, \ldots, 999999\}$.

Paisktabeliks P nimetatakse massiivi $p_0, p_1, \ldots, p_{m-1}$, milles iga element on ette nähtud ühe kirje (ja veel mõne lisavälja) salvestamiseks. Paisktabeli pikkus m valitakse suure liiaga, nii et kirjete hulk R sinna alati lahedasti ära mahuks. Paisktabeli elemente nimetame ka tema ridadeks.

Paiskfunktsiooniks nimetatakse funktsiooni h, mis igale võtmele hulgast U seab vastavusse paisktabeli mingi rea numbri:

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

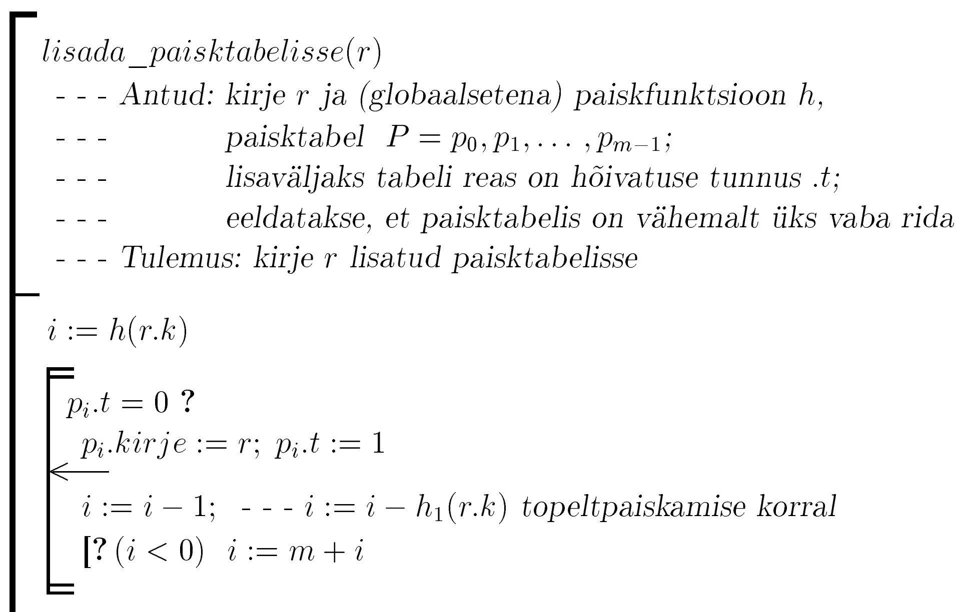
Paiskfunktsioon peab olema lihtsalt arvutatav (keerukusega $O(1)$). Tihtipeale võetakse paiskfunktsiooniks $h(k) = k \bmod m$. Lihtsustatult võib paisksalvestuse põhimõtet kirjeldada järgmiselt: kirje r lisamiseks arvutatakse $i = h(r.k)$ ja salvestatakse kirje r tabeli i-ndasse ritta; etteantud võtmega k_0 kirje lugemiseks arvutatakse $i = h(k_0)$ ja loetakse andmed tabeli i-ndast reast. Paraku tumestab seda ilusat pilti **kollisiooni** ehk **põrke** oht, kus kahe erineva võtme $k' \neq k''$ korral paiskfunktsiooni väärtused langevad kokku: $h(k') = h(k'')$. Siis lisamise katse võib ebaõnnestuda seetõttu, et tabeli rida nr. $h(r.k)$ ei ole enam vaba, ning ka lugemisel ei saaks kindel olla, et reast nr. $h(k_0)$ just selle kirje andmed saadakse, mille võti on k_0. Konkreetsed paisksalvestusmeetodi variandid erinevadki üksteisest peaasjalikult just põrkeolukordade lahendamisviisi poolest. Vaatleme alljärgnevas neist kahte.

Välisahelate meetodi puhul asuvad tabeli reas peale välja ($.kirje$) kirje

salvestamiseks veel kaks lisavälja, $.t$ ja $.viit$. Esimene neist on rea hõivatuse tunnuseks: $p_i.t = 1$, kui tabeli ritta nr. i on juba salvestatud mingi kirje, vastasel korral on selle välja väärtuseks arv 0. Põrkeolukorrad lahendatakse lihtahelate abil: need kirjed, mille võtmele vastab paiskfunktsiooni väärtus i, kuid mida enam ei saa paisktabeli ritta p_i salvestada, lisatakse uute liikmetena realt p_i algavasse lihtahelasse. Lihtahela lülisse kuulub üks kirje koos viidaga järgmisele lülile, viit ahela esimesele lülile asub väljal $p_i.viit$. Rea p_i hõivamisel salvestatakse sinna nii kirje kui ka lisaväljade väärtused: $p_i.t := 1$ ja $p_i.viit := \Lambda$. Lugemisel võtme k järgi leitakse rea number $i = h(k)$. Kui $p_i.t = 0$, siis võtmega k kirjet tabelis ei ole. Vastasel korral otsitakse selle võtmega kirjet kogu realt p_i algavas ahelas (tabeli rida kaasa arvatud), ahelat järjestikuselt läbi vaadates. Kirje ongi leitud, kui jõutakse mingi lülini, mis sisaldab otsitava võtmega kirjet. Kui otsitavat kirjet selles ahelas pole, siis pole teda veel hulka R lisatud.

Välisahelate meetodi plussiks on see, et ka kirjete eemaldamist saab korraldada keskmise ajalise keerukusega $O(1)$, miinuseks aga ebamugavused arvutiprogrammina realiseerimisel. Nimelt osutub keeruliseks hinnata ahelatele tarviliku mälu mahtu, et vajalikku mäluosa eelnevalt reserveerida. Samuti on teatavaid probleeme kogu sel moel salvestatud kirjete hulga kompaktse säilitamisega.

Järgnevas vaadeldaval **lahtise adresseerimise meetodil** eelpool nimetatud puudusi ei ole, kuid see-eest jälle ei saa selle meetodi kohaselt korraldatud paisktabelist kirjeid mõistliku ajaga eemaldada. Lisamise ja lugemise algoritmid on toodud joonistel 3.3 ja 3.4. Eeldatakse, et paisktabelis vähemalt üks rida on kindlasti hõivamata. See eeldus ei ole sugugi kitsendav, sest igasuguse paisksalvestusmeetodi korral on normaalseks tööks vajalik paisktabeli teatav alatäitumus – kui tabel oleks peaaegu täis, siis muutuks põrgete arv juba ülemäära suureks. Lisamine lahtise adresseerimise korral toimub järgmiselt: kirje võtmega k lisatakse tabeli esimesse vabasse ritta, mis leitakse, alustades otsimist reast number $h(k)$. Vaba rida otsitakse reanumbrite kahanemise järjekorras, kuid tsükliliselt üle kogu tabeli. Kirje lugemiseks võtme k_0 järgi vaadatakse järjestikuselt (ja tsükliliselt) läbi kirjed alates reast number $h(k_0)$, kuni leitakse otsitava võtmega kirje või jõutakse vaba reani (mis annab tunnistust sellise kirje puudumisest paisktabelis). Ka loetava kirje otsimine

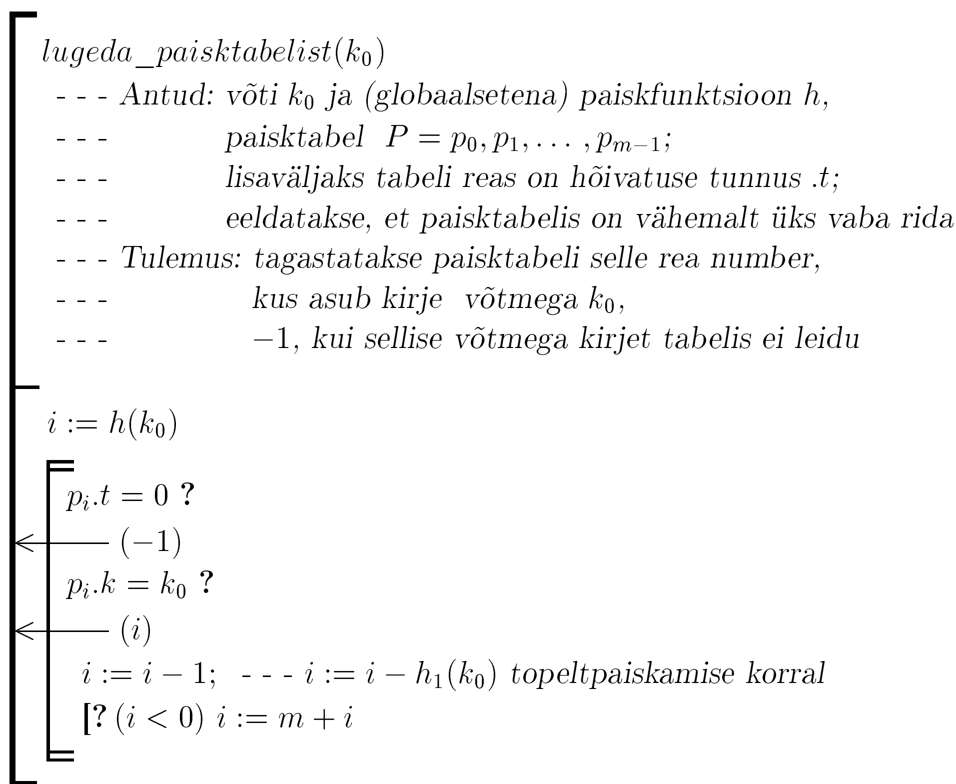


Joonis 3.3: Lisamine lahtise adresseerimisega paisktabeli korral.

toimub indeksite vähenemise suunas ning tsükliliselt.

Kirjeldatud paisksalvestusmeetodi oluliseks täiustuseks osutub **topeltpaiskamisega** lahtine adresseerimine. Selle kohaselt toimub tabelis liikumine mitte sammuga -1, vaid sõltuvalt antud võtmest (millega tabelisse siseneti). Sammu suurus arvutatakse nn. teisese paiskfunktsiooni h_1 abil. Üldiselt saadakse siis erinevad sammud nende erinevate võtmete jaoks, millega esialgse paiskfunktsiooni väärtuste kokkulangemise tõttu alustatakse samast reast. Olgu näiteks $k' \neq k''$ ja $h(k') = h(k'') = i$. Seega nii k' kui ka k'' korral alustatakse otsimist paisktabeli reast number i. Kuid võtme k' korral otsitakse sammuga $h_1(k')$ ja võtme k'' korral sammuga $h_1(k'')$ ja (loodetavasti) $h_1(k') \neq h_1(k'')$. Niisugusel moel paigutatud kirjete hilisemal otsimisel tuleb vähem ridu järjestikuselt läbi vaadata: näiteks võtme k' järgi lugemisel ei "jää ette" kirje võtmega k''. Teisese paiskfunktsiooni väärtuste hulka ei kuulu arv 0, sest niisuguse sammuga ei saa tabelit läbi käia. Seega

$$h_1 : U \to \{1, 2, \ldots, m-1\}.$$



Joonis 3.4: Lugemine lahtise adresseerimisega paisktabeli korral.

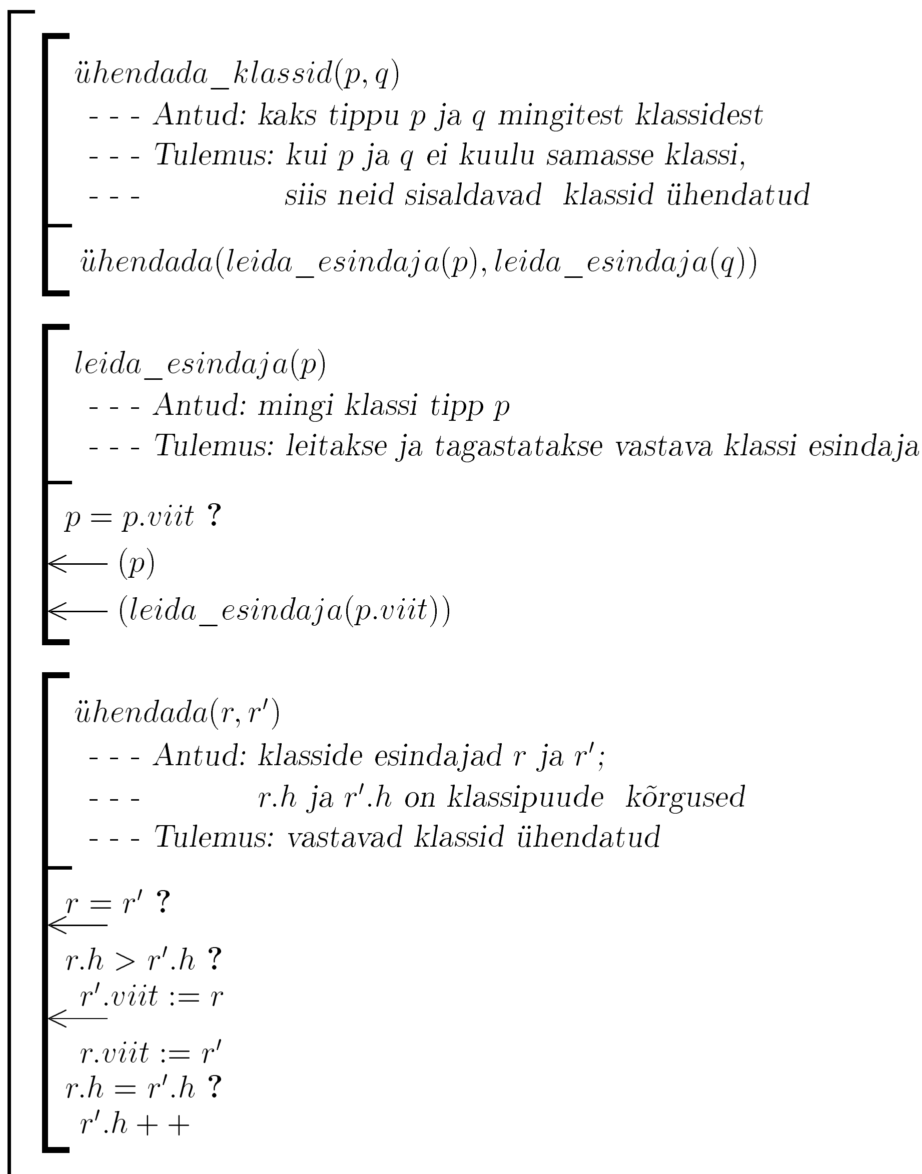
Selleks, et tabeli tsüklilisel läbivaatamisel jõuda igale reale, peavad sammu pikkus ning tabeli ridade arv olema ühistegurita. Tavaliselt valitaksegi tabeli ridade arvuks m algarv.

3.3 Klasside kujutamine

Olgu antud ühisosata hulkade ehk klasside komplekt $\{S_1, S_2, \ldots, S_n\}$, kus $S_i \cap S_j = \emptyset$, kui $i \neq j$. Meid huvitab selline dünaamiliste klasside kujutusviis, mille korral oleks lihtne klasse ühendada. Vajadus klasside ühendamiseks tekib näiteks programmi kompileerimise käigus, kus teataval etapil on leitud juba esinenud programmiobjektide ekvivalentsiklassid ning järjekordne vaadeldav direktiiv programmis kuulutab kaks objekti x ja y ekvivalentseteks. Kui x ja y kuuluvad hetkel erinevatesse klassidesse S_i ja S_j, siis tuleb nüüd vaadeldavat direktiivi arvestades moodustada üks ühendatud klass $S_i \cup S_j$.

Klasside kujutamiseks sobib nn. **Galler-Fischeri meetod**, mille kohaselt klasse säilitatakse puudena ja klasside komplekti metsana. Erinevalt tavapärasest alluvale-viidaga puudest on aga klassipuu iga tipp varustatud viidaga ülemusele; juure korral on selleks viit juurele endale. Lisaks eeldatakse veel, et leidub ka eraldi juurdepääs metsa igale tipule. Näiteks võib kogu tippude hulk olla organiseeritud kahendotsimise puuna, paisktabelina vms. Tänu üles-viitadele osutub eriti lihtsaks kahe klassi ühendamine. Kui tuleb ühendada klassid, millest üks sisaldab antud tippu x ja teine antud tippu y, siis toimitakse järgmiselt. Alustades tipust x liigutakse üles-viitade järgi puu juureni r_1, samuti leitakse tipust y lähtudes vastava puu juur r_2. Kui osutub, et $r_1 = r_2$, siis tipud x ja y kuuluvad juba samasse klassi ja ühendada polegi tarvis. Vastasel juhul, kui $r_1 \neq r_2$, asendatakse ühes leitud juurtippudest viit iseendale viidaga teise puu juurele. Sellega ongi kaks erinevat klassi ühendatud.

Klassile vastava puu juurt nimetatakse **klassi esindajaks**. Antud klassi esindaja r otsimine mingist selle klassi tipust x lähtudes on seda kiirem, mida lühem on tee tipust x üles juurtippu r. Pikkade otsimisteede vältimiseks tuleb püüda klassipuud hoida võimalikult madalad. Sel eesmärgil osutub otstarbekohaseks kahe klassi ühendamisel lisada just madalam puu kõrgema puu juure alampuuks. Selleks on omakorda vajalik teada klassipuude kõrgusi. Loomulik on säilitada iga puu kõrgus tema juurtipus. Ühendamisel saadava puu kõrgust tuleb muuta (ühe võrra suurendada) vaid siis, kui ühendati kaks ühekõrgust puud. Klasside ühendamiseks vajalike protseduuride kirjeldused on toodud joonisel 3.5.



Joonis 3.5: Klasside ühendamine.

Oluliseks klasside ühendamise tõhustamise (puude kõrguste vähendamise) võtteks on nn. **teede õgvendamine**. Viimane seisneb selles, et mingi tipu x jaoks klassi esindaja r otsimise käigus kõik tipud teel $(x, \dots, r)$ ühendatakse üles-viida kaudu otse juurega r.

Vastav nn. **kahekäiguline rekursiivne algoritm** $leida_esindaja'$ on toodud joonisel 3.6. Selles ühel käigul minnakse üles, leides tee tipust x juurtippu, teisel tullakse leitud teed mööda alla, muutes üles-viidad viitadeks otse puu juurele. Iga $leida_esindaja$ rekursiivne rakendus tagastab väärtuse $x.viit$. Kui x on juur, siis tagastatakse $x.viit = x$ ja uut rekursiooni sammu enam ei alustata. Vastasel korral järgneb uus rakendus parameetriga $x.viit$. Selle tulemusena saab $x.viit$ viidaks juurele, mis ka tagastatakse.

Paraku osutub üsna keeruliseks tuvastada teede õgvendamisega kaasnevat klassipuude kõrguste võimalikku vähenemist. Õgvendamise rakendamisel on otstarbekohasem üldse loobuda klassipuude kõrguste arvestamisest (ühendamise protseduuris).

```
leida_esindaja'(p)
  - - - Antud: mingi klassi tipp p
  - - - Tulemus: leitakse ja tagastatakse vastava klassi esindaja r,
  - - -              ühtlasi õgvendatakse tee tipust p tippu r

  [? (p ≠ p.viit)) p.viit := leida_esindaja'(p.viit)
←— (p.viit)
```

Joonis 3.6: Klasside ühendamine.

Lisaks ülalkirjeldatud Galler-Fischeri meetodile tasub hulga ja tema alamhulkade käsitlemist nõudvate ülesannete korral alati kaaluda ka võimalust alamhulkade esitamiseks maskide abil (vt. peatükk 1). Näiteks kahe alamhulga ühendi mask saadakse lihtsalt ühendatavate alamhulkade maskide bitikaupa loogilisel liitmisel; kahe alamhulga ühisosa mask saadakse nende alamhulkade maskide bitikaupa loogilisel korrutamisel jne.

3.4 Kuhjad

Üldiselt öeldakse, et puu (või mets) on **kuhi**, kui kehtib nn. **kuhja-omadus**: ühegi tipu võti pole suurem (väiksem) kui tema ülemuse võti. Konkreetsete rakenduste korral muidugi fikseeritakse, kumb tingimustest (suurem/väiksem) kuhjas peab kehtima.

Kahendkuhi ehk lihtsalt **kuhi** on kompaktne kahendpuu, milles ühegi tipu võti pole suurem kui tema ülemuse võti. On ilmne, et suurima võtmega kirje paikneb kuhja juurtipus. Mõnedes rakendustes osutub otstarbekohaseks esitada andmed nn. **pöördkuhjana**, kus ühegi alluva võti ei ole väiksem tema ülemuse võtmest. Alljärgnevas kirjeldatavad töötlusvõtted on kõik kergesti teisendatavad ka pöördkuhja juhule.

Kuhja sobivaimaks kujutusviisiks on järjestikpaigutus kahendpuu tasemete kaupa: n-tipulise kuhja tipud moodustavad massiivi $a_1, a_2, \ldots, a_n$, kus a_1 on juur, a_2 ja a_3 on juure alluvad (st. esimese taseme tipud), seejärel tulevad kõik teise taseme tipud jne.; element a_n vastab viimase taseme kõige parempoolsemale tipule (vt. joonis 3.7).

Selline kujutusviis on äärmiselt ökonoomne ning võimaldab hõlpsasti sooritada ka kõiki järgnevas vaadeldavaid kuhjatöötlemise operatsioone. Ka kuhja kui kahendpuu tippude alluvussuhted esituvad triviaalselt: vaadeldava tipu a_k korral on alluvate ja ülemuse indeksiteks

$vasak(k) = 2k;$

$parem(k) = 2k + 1;$

$ülemus(k) = \lfloor k/2 \rfloor.$

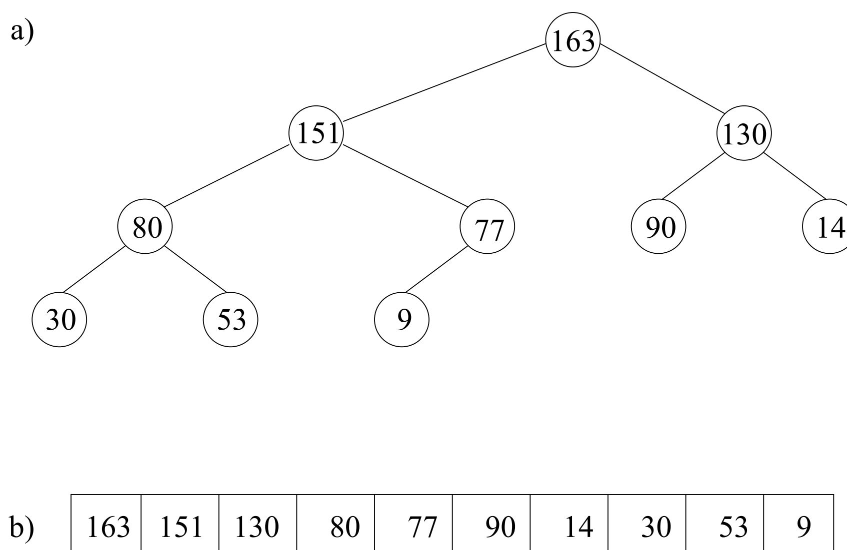
Kui mõne ülaltoodud valemi rakendamisel saame arvu väljaspool lõiku $[1, n]$, siis annab see tunnistust vastava alluva või ülemuse puudumisest.

Järgnevates kuhjatöötlemise algoritmides eeldame, et (globaalseina) on antud tippude arv n ja massiiv $a_1, a_2, \ldots, a_n$. Lihtsuse mõttes samastame kirje vastava elemendiga, mõistes omistamise all kirje salvestamist, võrdlemise all aga võtmete võrdlemist:

$a_i := a_j$; - - - elemendis a_j paiknev kirje salvestada elementi a_i

$a_i < b$? - - - kas elemendis a_i asuva kirje võti on väiksem kui b võti?

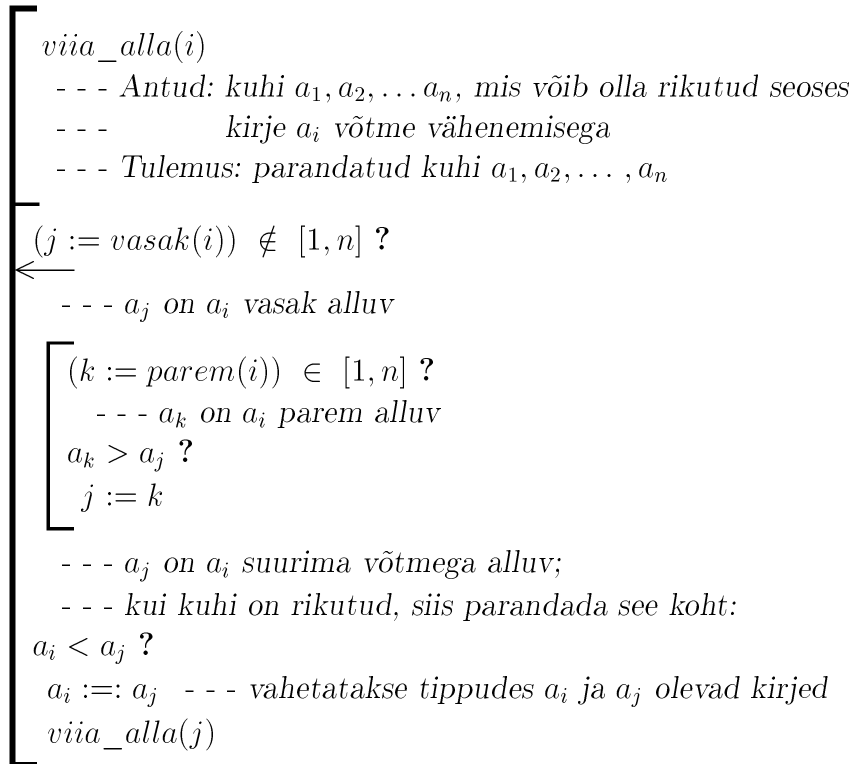
Massiivina esitatud kuhjast eemaldatakse tippe ainult massiivi lõpust ja uusi tippe lisatakse ainult massiivi lõppu. Ülejäänud töötlus seisneb kuhja



Joonis 3.7: Kahendkuhi (a) ja selle esitus massiivina (b).

tippudes asuvate kirjete muutmises, enamasti ülemuse ja alluva kirjete vahetamises. Vastavate muudatustega kaasneb reeglina ajutine kuhja rikkumine: pärast kirje (ja võtme) muutmist ei pruugi enam kehtida kuhja põhiomadus, mille kohaselt alluva võti ei tohi olla ülemuse võtmest suurem. Seetõttu tuleb pärast iga võtmemuutust sooritada teatav kuhjaparanduse operatsioon. Kui tippu salvestatud uue kirje võti on väiksem kui seal varem olnud võti, siis viiakse uus kirje järjestikuste vahetuste teel kuhjas allapoole (vt. joonis 3.8), vastasel korral aga ülespoole (vt. joonis 3.9). Mõlema kuhjaparanduse algoritmi ajaline keerukus halvimal juhul on $O(\log n)$, sest nende rakendamisel parameetri väärtus väheneb (suureneb) vähemalt 2 korda, jäädes aga ikka vahemikku $(0, n)$.

Joonisel 3.10 leidub lihtne algoritm kuhjast suurima võtmega kirje võtmiseks. Pärast juurtipust kirje lugemist ei eemaldata mitte juurtippu, vaid sinna salvestatakse kirje kuhja viimasest tipust (massiivi lõpust) ning siis eemaldatakse hoopis see "tühjaks jäänud" viimane tipp. Kuna aga nüüd juurtippu viidud kirje võti võib osutuda väiksemaks seal enne olnud kirje võtmest,



Joonis 3.8: Kuhjaparandus liiga väikese võtme allapoole viimise teel.

siis tuleb veel rakendada kuhjaparandust *viia_alla* (vt. joonis 3.8). Viimane määrab ühtlasi ka suurima võtmega kirje võtmise operatsiooni ajalise keerukuse halvimal juhul: $O(\log n)$.

Kirje lisamine kuhja toimub samuti $O(\log n)$ ajaga: uus kirje lisatakse massiivi lõppu, suurendatakse elementide arvu n ühe võrra ning rakendatakse uuele tipule kuhjaparandust *viia_üles*.

Suvalise kirjetemassiivi $a_1, a_2, \ldots, a_n$ kuhjaks muutmiseks ehk **kuhjastamiseks** saab kasutada nii üles- kui ka allaviimise operatsiooni. Esimesel juhul seisneks kuhjaks muutmine iga antud kirje korral lisamise algoritmi rakendamises. Mõnevõrra eelistatum on aga allaviimise teel kuhjastamine, mille ajaliseks keerukuseks tuleb $O(n)$. Vastav rekursiivne algoritm leidub joonisel 3.11.

```
viia_üles(i)
 --- Antud: kuhi a_1, a_2, ... a_n, mis võib olla rikutud seoses
 ---        kirje a_i võtme suurenemisega
 --- Tulemus: parandatud kuhi a_1, a_2, ... , a_n

(j := ülemus(i)) ∈ [1, n] ?
 --- a_i ei ole juur
a_i > a_j ?
 a_i :=: a_j; viia_üles(j)
```

Joonis 3.9: Kuhjaparandus liiga suure võtme ülespoole viimise teel.

```
võtta_suurim()
 --- Antud: kuhi a_1, a_2, ... , a_n; olgu juurtipus a_1 kirje R
 --- Tulemus: kirje R eemaldatud antud kuhjast; tagastatakse kirje R

R := a_1; a_1 := a_n; n −−; viia_alla(1)
←— (R)
```

Joonis 3.10: Kuhjast suurima võtmega kirje võtmine.

Kuhjastamise ning suurima võtmise algoritmi kombineerimisel saame elegantse meetodi kirjetemassiivi sorteerimiseks (vt. joonis 3.12), mida nimetatakse **kuhjameetodiks**. Kuigi viimase ajaline keerukus on soodne, olles halvimal juhul $O(n \log n)$, ei suuda see praktikas siiski võistelda (edasises vaadeldavate) kiir- ja ühildamismeetodiga.

Kirjete hulga kuhjastamine on aga asendamatuks võtteks eelistusjärjekordade realiseerimisel. On ju kuhja korral suhteliselt efektiivsed nii kirje lisamise kui ka suurima võtmega kirje võtmise operatsioonid. Ka muutuvate eelistuste käsitlemine ei valmista raskusi: kui tipus a_i olev võti k asendatakse võtmega k', siis juhul $k' < k$ tuleb lihtsalt rakendada protseduuri $viia_alla(i)$, vastasel korral aga protseduuri $viia_üles(i)$.

$kuhjastada(i)$
--- *Antud: kirjete massiiv* $a_1, a_2, \ldots, a_n$ *ja* $i \geqslant 1$
--- *Tulemus: tipus* a_i *ja tema järeltulijates asuvad kirjed ümber paigu-*
--- *tatud nii, et nende tippude jaoks kehtib kuhjaomadus*

$i \leqslant n$ **?**
$kuhjastada(vasak(i))$
$kuhjastada(parem(i))$
$viia_alla(i)$

Joonis 3.11: Kuhjastamise algoritm.

$kuhi_sorteerida()$
--- *Antud: massiiv* $a_1, a_2, \ldots, a_n$
--- *Tulemus: massiiv sorteeritud mittekahanevalt*

$kuhjastada(1)$
$k := n$
[$\star$ (n) $a_{k--} := võtta_suurim()$

Joonis 3.12: Kuhjameetodi algoritm.

Kahendkuhjade miinuseks on see, et nende ühendamise operatsioon, st. kahest või enamast kuhjast ühe ühise kuhja tegemine, on suhteliselt aeglane, toimudes lineaarse ajaga. Järgnevas vaadeldav binomiaalkuhi on selles suhtes märksa efektiivsem vahend andmestruktuuride realiseerimiseks.

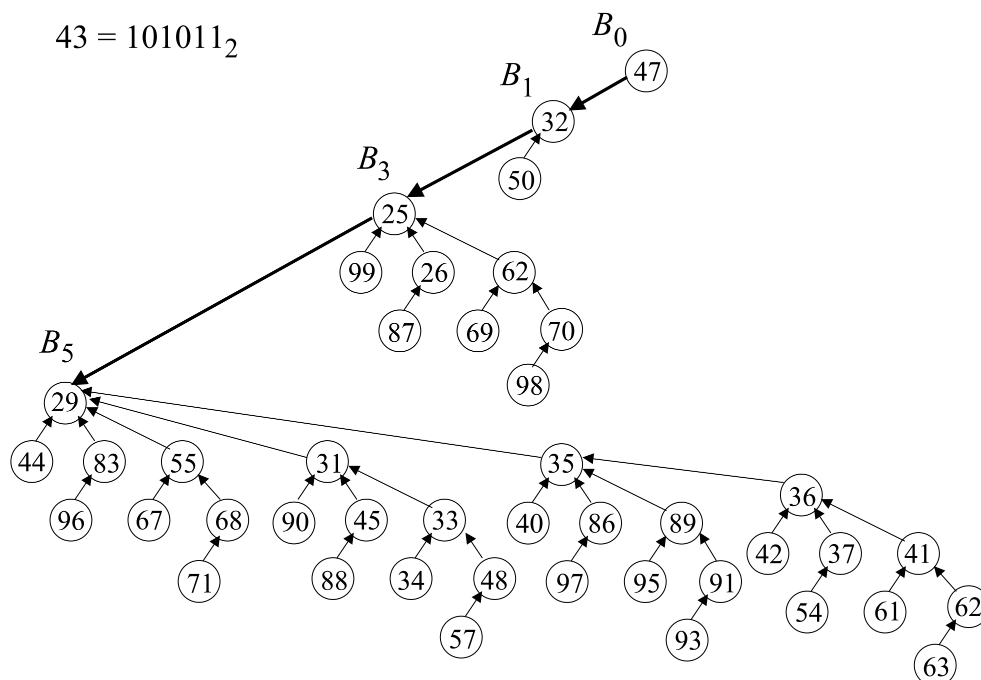
Binomiaalkuhi on binomiaalpuude mets, milles

1) ühegi tipu võti ei ole väiksem kui tema ülemuse võti;

2) puude juurte astmed on paarikaupa erinevad.

Esimene omadus tagab selle, et vähim võti igas puus asub juurtipus.

Teisest omadusest järeldub, et n-tipulises binomiaalkuhjas leidub ülimalt $\lfloor \log n \rfloor + 1$ puud. Teatavasti on binomiaalpuu tippude arv kahe aste, st. puus B_i on 2^i tippu. Järelikult on tippude koguarvuks arvu 2 erinevate astmete



Joonis 3.13: 43-tipuline binomiaalkuhi.

summa. See aga tähendab, et binomiaalpuu B_i esineb kuhjas parajasti siis, kui arvu n kahendesituses $\sum_{i=0}^{\lfloor \log n \rfloor} b_i 2^i$ vastav bitt b_i on 1. Arvu n kahendesituses on kuni $\lfloor \log n \rfloor + 1$ nullist erinevat bitti, seega ka binomiaalkuhjas saab olla ülimalt $\lfloor \log n \rfloor + 1$ puud.

Nagu kahendkuhjagi korral, nii on ka binomiaalkuhja töötlusalgoritmid tihedalt seotud konkreetse kujutusviisiga. Antud juhul valitakse selleks aga seotud paigutus. Lisaks tavapärastele viitadele esimesele alluvale ja paremale kolleegile kasutatakse veel üles-viita tipu ülemusele. Kõik metsa puude juured seotakse astmete kasvamise järjekorras lihtahelasse, mida nimetame ka binomiaalkuhja **juurahelaks**. Käesolevas me sellisel esitusel baseeruvaid üksikasjalikke algoritme ei esita, piirdudes vaid üldiste põhimõtete selgitamisega. Binomiaalkuhja näide leidub joonisel 3.13.

Vähima võtmega kirje **otsimiseks** vaadatakse läbi kuhja juurahel.

Kahe binomiaalkuhja **ühendamine** toimub järgmiselt. Ühildatakse mõle-

ma antud kuhja juurahelad, nii et saadud ahel oleks juurte astmete järgi mittekahanevalt sorteeritud. Arvestades, et binomiaalkuhjas on juurte astmed erinevad, saab moodustatud ahelas olla (kõrvuti) kuni kaks sama astmega juurt. Iga sellise paari korral lülitatakse suurema juure-võtmega puu ahelast välja ning lisatakse teise juurele esimeseks alampuuks. Edasises tuleb arvestada, et niisuguse operatsiooni tagajärjel võis tekkida ka võrdsete astmetega juurte kolmik. Sellisel juhul asutakse vaatlema kolmiku kahte viimast lüli, jättes esimese muutumatuks. Binomiaalkuhjade ühendamist rakendatakse muuhulgas ka mitmetes teistes kuhjatöötlusoperatsioonides alamprotseduurina.

Kirje **lisamiseks** binomiaalkuhja tehakse algul lisatavast tipust omaette binomiaalkuhi ja siis ühendatakse see antud kuhjaga.

Vähima võtmega kirje **võtmiseks** sooritatakse järgmised tegevused:

– otsitakse vähima võtmega kirjet sisaldav juur p, loetakse selles olev kirje ja lülitatakse tipp p juurahelast välja (saadakse kuhi H ilma puuta p);

– p alampuudest tehakse teine kuhi H', ühendades tipu p alluvad vastupidises järjekorras (paremalt vasakule) uueks juurahelaks;

– ühendatakse kuhjad H ja H'.

Võtme vähendamise puhul toimitakse nagu kahendkuhjas (võtme suurenemisel): vastavas puus viiakse muutunud võtmega kirje ülespoole oma õigele kohale, tehes järjestikuseid kirjete vahetusi ülemustega.

Antud tipu **eemaldamiseks** asendatakse eemaldatavas tipus oleva kirje võti väikese väärtusega ($-\infty$), seejärel toimitakse nagu ikka võtme vähenemise puhul ning rakendatakse siis vähima kirje võtmise protseduuri.

Kõik ülalkirjeldatud binomiaalkuhja operatsioonid on ajalise keerukusega $O(\log n)$.

Peatükk 4

Sorteerimine

Käesolevas vaatleme arvujärjendi sorteerimise ülesannet, mille kohaselt tuleb antud n-komponendilise andmestruktuuri komponentide $a_1, a_2, \ldots, a_n$ väärtused ümber paigutada selliselt, et kehtiks $a_1 \leqslant a_2 \leqslant \ldots \leqslant a_n$.

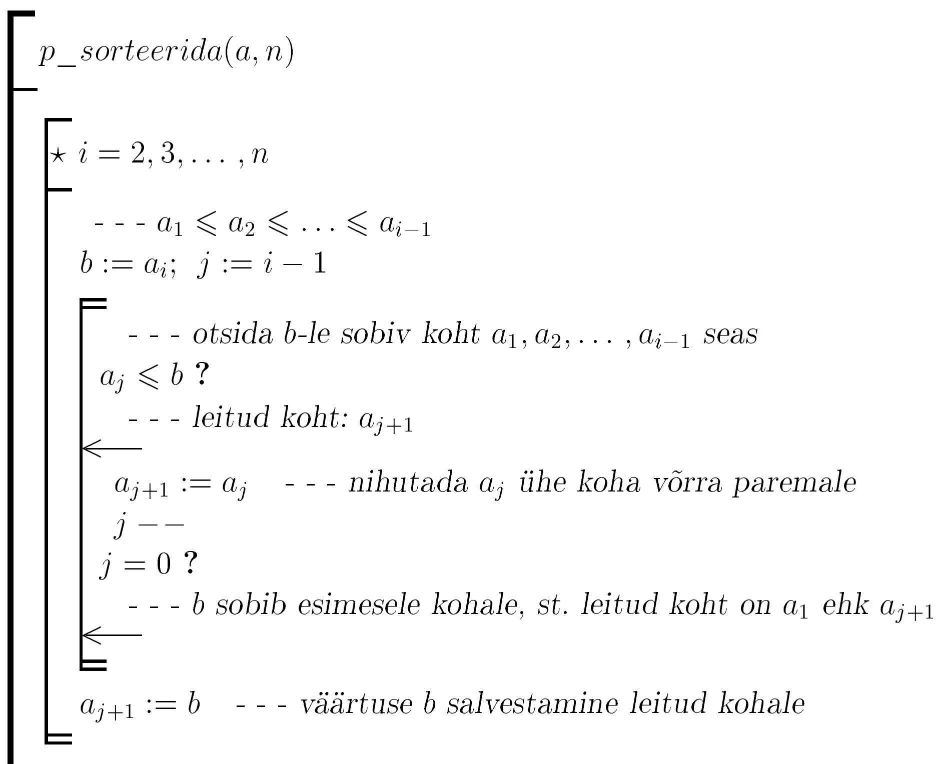
Esitatavate algoritmide eeskujul saab vajaduse korral hõlpsasti konstrueerida ka programmid keerukamate objektide järjestamiseks. Kirjete hulga sorteerimine seisneb tavaliselt nende ümberpaigutamises võtmete teatavasse, näiteks mittekahanevasse järjekorda. Sorteerimisprogrammides tuleb vältida igasuguseid liigseid operatsioone, sest tegemist on niigi üsna aeganõudva tööga: nagu allpool selgub, on parimate sorteerimisalgoritmide ajaline keerukus $O(n \log n)$ ning üldjuhul polegi lootust leida paremat meetodit. Pikkade kirjete järjendi korral loobutakse harilikult nende endi ümberpaigutamisest ja järjestuse leidmiseks sorteeritakse hoopis paarid <*võti, viit vastavale kirjele*>.

Kirjete sorteerimisel tuleb vahel silmas pidada ka võrdsete võtmetega kirjete omavahelist järjekorda. Sorteerimismeetodit või -algoritmi nimetatakse **stabiilseks**, kui selle rakendamisel võrdsete võtmetega kirjete omavaheline järjestus ei muutu.

Paremate ja ühtlasi keerukamate sorteerimismeetodite eelised ilmnevad muidugi alles suuremate (sadadesse ulatuvate) n väärtuste puhul. Lühikeste järjendite sorteerimiseks võib edukalt rakendada lihtsamaid algoritme, mille ajaline keerukus on $O(n^2)$. Järgmises jaotises tutvumegi ühe $O(n^2)$ meetodiga, millel on mitmeid eeliseid teiste omataoliste ees ning mis leiab kasutamist ka mõnedes keerukamates algoritmides alamprotseduurina lühemate või siis „peaaegu“ sorteeritud osajärjendite korrastamiseks.

4.1 Pistemeetod

Pistemeetodi idee põhineb eeskätt välimise tsükli invariandil (vt. joonis 4.1): kui järjendist on läbi vaadatud $i-1$ esimest elementi, siis on need ka juba seatud mittekahanevasse järjestusse.



Joonis 4.1: Pistemeetodi algoritm.

Tsükli samm seisneb parajasti vaatlusel oleva elemendi a_i "pistmises" sobivasse kohta talle eelnevate elementide seas. Vaadeldava elemendi väärtus salvestatakse muutujasse b, seejärel otsitakse (sisemises tsüklis) koht kuhu see võiks sobida, vaadates järestikuselt läbi elemendid $a_{i-1}, a_{i-2}, \ldots$. Väärtused, mis osutuvad suuremaks kui b, nihutatakse ühe koha võrra paremale, et teha ruumi piste sooritamiseks. Niipea, kui jõutakse elemendini $a_j \leqslant b$ või juba järjendi algusesse ($j = 0$), ongi vahelelisamiseks sobiv koht käes ja saab salvestada „meelespeetud" väärtuse: $a_{j+1} := b$. Sellega on tagatud

invariandi kehtivus ühe võrra pikema järjendi $a_1, a_2, \ldots, a_i$ korral ning võib alustada järgmist tsükli sammu. On selge, et pärast viimast sammu, kus a_n pistetakse sobivasse kohta $a_1 \leqslant a_2 \leqslant \ldots \leqslant a_{n-1}$ seas, saadaksegi otsitav mittekahanevalt sorteeritud järjend.

Pistemeetod on stabiilne.

Ebasoodsaimaks juhuks pistemeetodi jaoks on kahanev lähtejärjend, sest siis iga vaadeldav element sobib just esimeseks talle eelnevas järjendiosas ning sisemises tsüklis tehakse iga i korral $i-1$ võrdlemist. Järelikult pistemeetodi ajaline keerukus halvimal juhul on $\Theta(n^2)$. Sama hinnang kehtib paraku ka keskmisel juhul, kuna iga a_i $(i = 2, 3, \ldots, n)$ korral tuleb läbi vaadata keskmiselt $(i-1)/2$ talle eelnevat elementi.

Pistemeetodi idee on rakendatav ka lihtahela sorteerimiseks (vt. joonis 4.2). Tuleb ainult arvestada, et lihtahelas saab liikuda vaid ühes suunas. Seetõttu peab sisemises tsüklis pistekoha otsimist alustama ahela algusest.

```
p_sorteerida_ahel(p)
 - - - Antud: mittetühi päisega lihtahel p
 - - - Tulemus: ahel p sorteeritud võtmete .a järgi
 - - -             viitade ümberkorraldamise teel

 - - - Abimuutujad: qq, q – tandem välimises tsüklis
 - - -                rr, r – tandem sisemises tsüklis

 ⋆ qq := p.viit;  (q := qq.viit) ≠ Λ;  qq := q

   ⋆ rr := p;  (r := rr.viit) ≠ q;  rr := r
    pistekoha otsimine osas p . . . q
   r.a > q.a ?
     - - - koht leitud, pista liige q liikmete rr ja r vahele:
    qq.viit := q.viit;  q.viit := r;  rr.viit := q
```

Joonis 4.2: Lihtahela sorteerimine pistemeetodil.

4.2 Kiirmeetod

Kiirematest meetoditest oleme juba tutvunud kuhjameetodiga. Praktikas on vaieldamatult soosituimaks nn. kiirjärjestusmeetod ehk **kiirmeetod** keskmise ajalise keerukusega $O(n \log n)$. Asjaolu, et halvimal juhul osutub keerukuse hinnanguks $O(n^2)$, selle meetodi tegelikku praktilist tähtsust oluliselt ei vähenda. Kiirmeetodil põhinevad sorteerimisprogrammid töötavad reeglina kõige kiiremini.

Kiirmeetod on tüüpiline jaga-ja-valitse laadi algoritm (vt. joonis 4.4). Esialgne ülesanne jagatakse kaheks väiksema mahuga ülesandeks, mis siis omakorda lahendatakse samal meetodil rekursiivselt; alamülesannete lahendustulemuste põhjal leitakse esialgse ülesande lahend. Antud juhul osutub keerukamaks just „jagamise“ etapp, kus antud väärtused sorteeritakse selliselt, et järjendi esimesse ossa jäävad väiksemad, teise ossa aga suuremad elemendid. Oluline on asjaolu, et ükski element esimeses osas ei ole suurem ühestki teise osa elemendist. Tänu sellele saabki järjendi kummagi osa nüüd omaette sorteerida. Pärast osade sorteerimist osutub sorteerituks ka kogu järjend. Seega „valitsemise“ etapp siin tegelikult puudub: alamülesannete lahendustulemustest esialgse ülesande lahenduse saamiseks pole tarvis midagi teha.

Jaotamise põhitsüklis funktsioonis *jaotada* (vt. joonis 4.4) võrreldakse järjendi elemente arvuga b, mida võib vaadelda kui „veelahet“ kahe otsitava osa vahel. Nimelt need väärtused, mis ei ületa suurust b, tuuakse algusossa ja need, mis pole b-st väiksemad, viiakse järjendi lõpuossa. Joonisel 4.3 on kujutatud 9-elemendiline lähtemassiiv ja selle seis pärast jaotamist, kui veelahkmeks on 31.

Vaadeldaval juhul kasutatakse jaotamise funktsioonis *jaotada* veelahkmena seda väärtust, mis asus parameetrina antud järjendis esikohal. Tegelikult on aga b vabalt valitav. Kõige soodsam on muidugi selline veelahe, millest mõlemale poole jääb enam-vähem ühepalju väärtusi, halvim aga selline, mille korral ühte poolde satub vaid üks element (päris tühjaks ei saa kumbki osa jääda). Väga ebavõrdsete poolte puhul suureneb oluliselt rekursiivsete rakenduste arv ja seega ka sorteerimiseks kuluv aeg. Viimasest asjaolust tulenebki õigupoolest kiirmeetodi $O(n^2)$ hinnang halvimal juhul. Meetodi paljud modi-

31	69	12	16	70	31	16	69	50

16	31	12	16	70	69	**31**	69	50

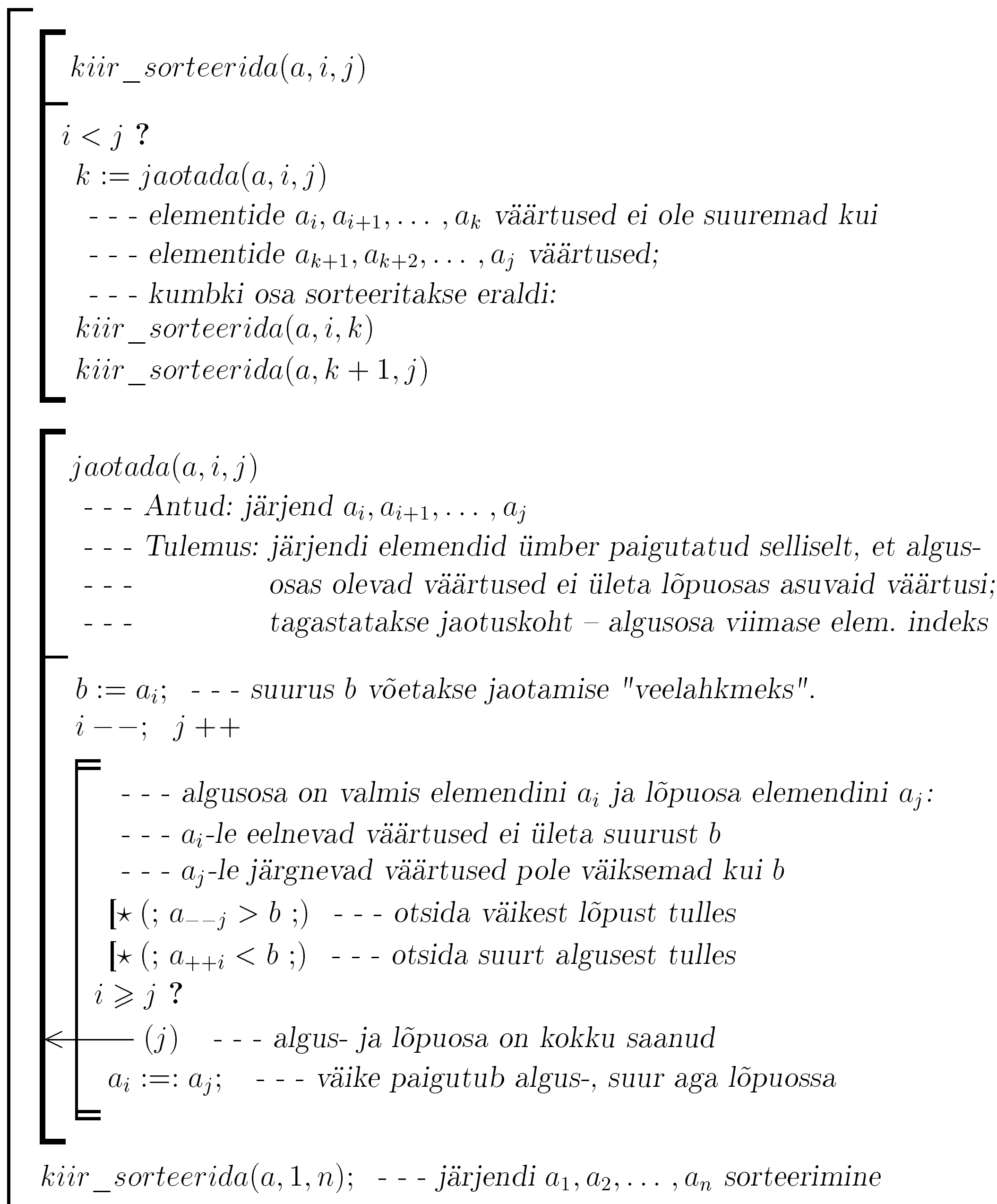
Joonis 4.3: Jaotamise näide.

fikatsioonid sätestavad mitmesuguseid erilisi veelahkme määramise võtteid.

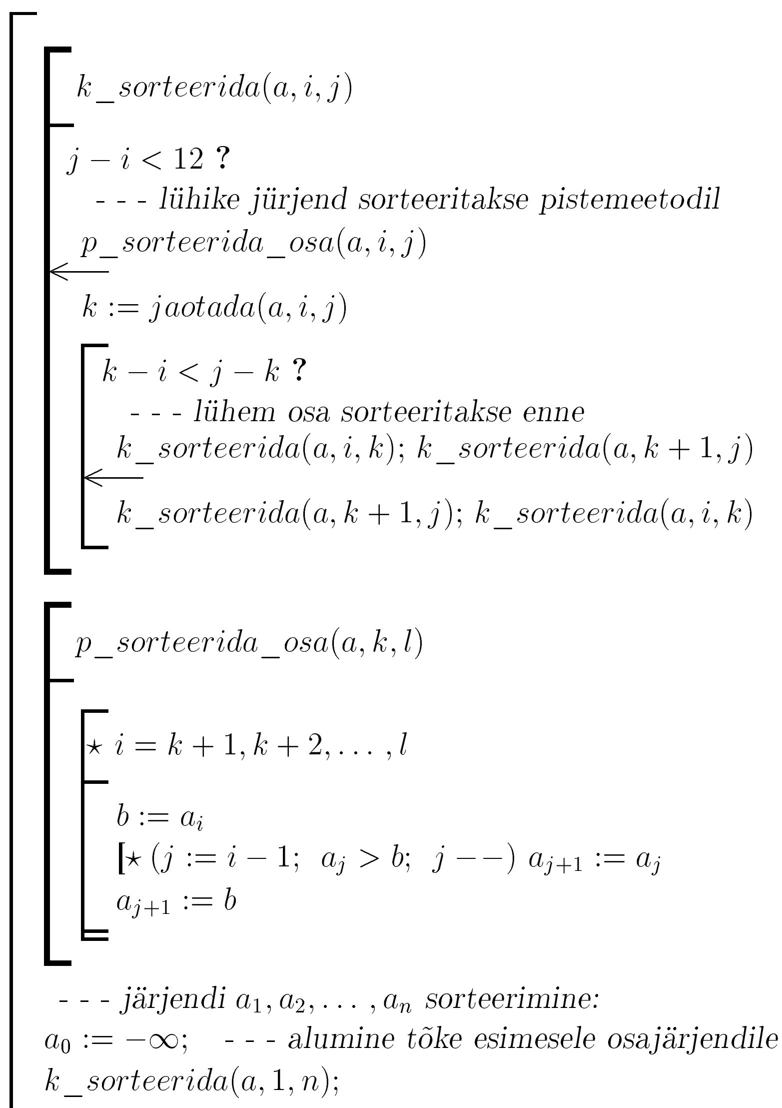
Kiirmeetodi praktilisel realiseerimisel on otstarbekohane kavandada lühikeste osajärjendite sorteerimine pistemeetodi abil (vt. joonis 4.5). Pärast järjekordse jaotuse leidmist on aga mõistlik esmalt jätkata just lühema osaga, et vähendada rekursiooni sügavust ja seega ka vajaliku lisamälu mahtu.

Paneme tähele, et kiirsorteerimisel kasutatakse pistemeetodi lihtsustatud varianti (vt. funktsioon $p_sorteerida_osa$ joonisel 4.5), kus sisemises tsüklis puudub tsükliloendaja j vähenemispiiri kontroll ($j \geqslant k$). Üldjuhul eelneb ju vaadeldavale osale selline järjendi osa, kus ükski element ei ole suurem ühestki vaadeldavast elemendist $a_k, a_{k+1}, \ldots, a_l$. Järelikult, kui pistekoha otsimisel indeks j omandab juba väärtuse $k-1$, siis lõpeb sisemine tsükkel $a_{k-1} \leqslant b$ tõttu. Selline loomulik tõke puudub vaid kõige esimesel osajärjendil, mistõttu tulebki enne protseduuri $k_sorteerida$ rakendamist massiivi esimese elemendi ette lisada veel ekstra väike väärtus ($-\infty$).

Peale tundlikkuse „ebasoodsate“ lähtejärjendite suhtes osutuvad kiirmeetodi puudusteks veel ebastabiilsus ja mitterakendatavus lihtahela sorteerimiseks. Järgnevas vaadeldav ühildamismeetod on nendest puudustest vaba, olles seega heaks alternatiiviks kiirmeetodile.



Joonis 4.4: Sorteerimine kiirmeetodil.



Joonis 4.5: Kiirmeetodi täiustatud variant.

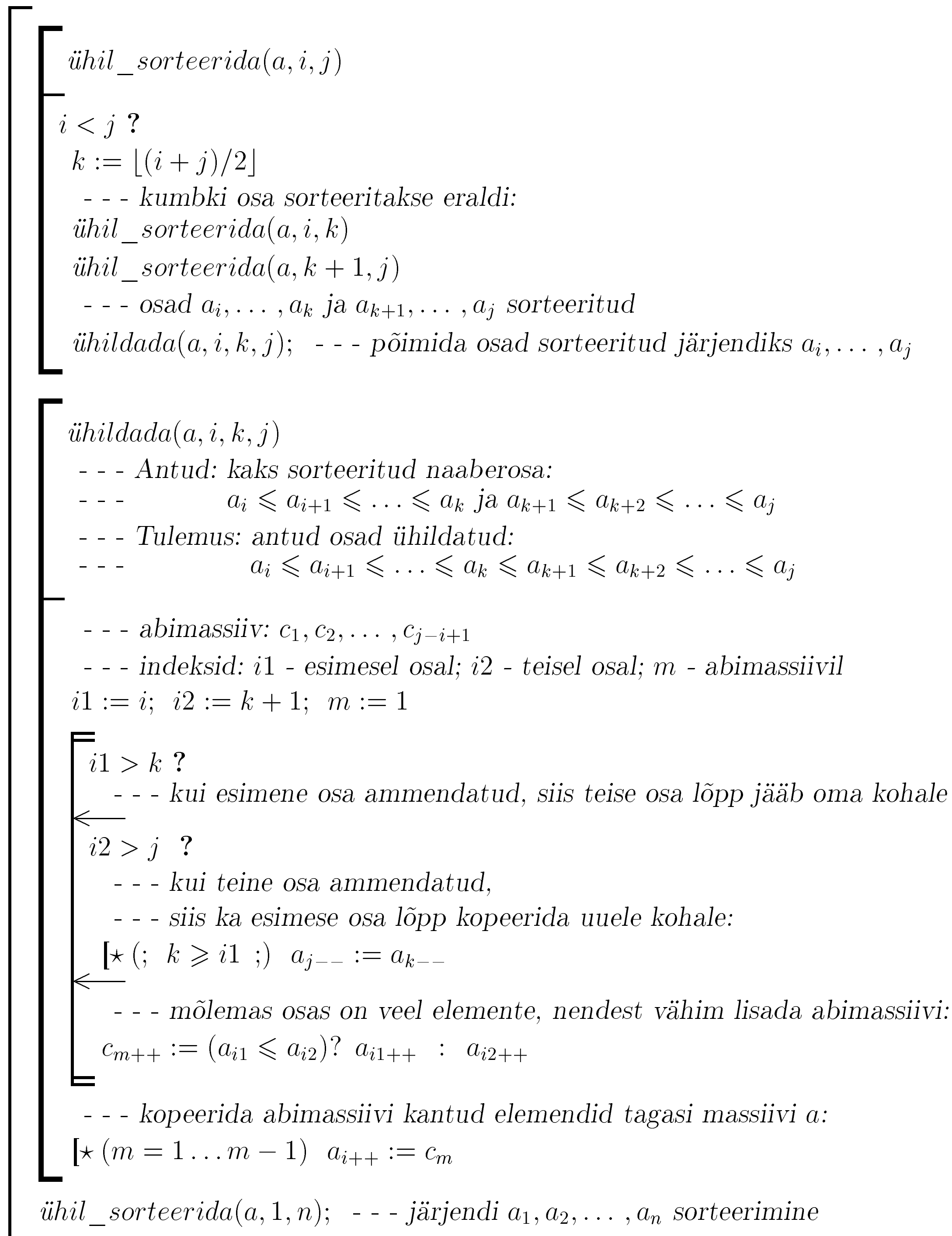
4.3 Ühildusmeetod

Ka **ühildusmeetod** (vt. joonis 4.6) ehk **põimemeetod** põhineb jaga-ja-valitse ideel. Kuid erinevalt kiirmeetodist toimub siin jagamine triviaalselt: igal sammul jaotatakse sorteeritav (osa)järjend lihtsalt kaheks võrdse (või siis ühe võrra erineva) pikkusega pooleks. Mõlemale poolele rakendatakse rekursiivselt sedasama protseduuri. Töömahukamaks osutub aga just "valitsemine": sorteeritud poolte ühendamine ehk **ühildamine**, **põimimine** üheks sorteeritud järjendiks.

Kuna poolte määramine toimub täiesti sõltumatult n-elemendilise lähtejärjendi elementide väärtustest, siis on selge, et algoritmi rekursiivse põhiosa rakenduste arv on igal juhul $O(\log n)$ ning mingit eriti halba algandmete juhtu ei saagi olla. Ühe n-elemendilise sorteeritud järjendi kokkuseadmiseks kahe lühema sorteeritud järjendi elementidest (vt. protseduur *ühildada* joonisel 4.6) kulub aega $O(n)$. Järelikult ühildusmeetodi ajaline keerukus on $O(n \log n)$. Ühildamise etapil pole põhjust võrdsete võtmetega elementide omavahelist järjestust muuta ning ühildusmeetod realiseeritakse alati stabiilsena.

Ühildusmeetodi praktilise kasutamise teeb ebamugavamaks asjaolu, et massiivina esitatud järjendi sorteerimisel tuleb (ühildamise alamprotseduuris) kasutada lisamälu mahus $\Omega(n)$. Seevastu aga lihtahela saab sellel meetodil sorteerida ilma lisamälu appi võtmata.

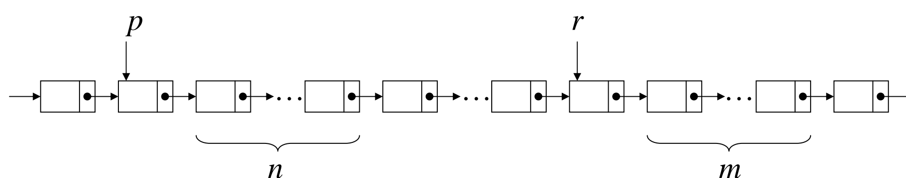
Joonistel 4.7 ja 4.9 ongi esitatud üks moodus lihtahela sorteerimiseks ühildusmeetodil. Algoritmi *ühildada_ahelad* (joonis 4.9) tsükli invarianti iseloomustab joonis 4.8.



Joonis 4.6: Sorteerimine ühildusmeetodil.

```
ü_sorteerida(p, k)
 - - - Antud: sorteeritav (päisega) lihtahel p,
 - - -        milles on k liiget (päist arvestamata)
 - - - Tulemus: antud lihtahel sorteeritud viitade ümberkorraldamise teel;
 - - -          tagastatakse viit ahela viimasele liikmele
k ⩽ 1 ?
←— (p.viit)
 r :=ü_sorteerida(p, ⌊k/2⌋)
 ü_sorteerida(r, ⌈k/2⌉)
←— (ühildada_ahelad(p, ⌊k/2⌋, r, ⌈k/2⌉))
```

Joonis 4.7: Lihtahela sorteerimine ühildusmeetodil (*ühildada_ahelad* – vt. joonis 4.9).



Joonis 4.8: Invariantne olukord lihtahelate ühildamise tsüklis joonisel 4.9.

```
ühildada_ahelad(p, n, r, m)
 - - - Antud: kaks sorteeritud ahelat (alamahelad mingis hõlmava ahelas);
 - - -        ahelate esimesed liikmed on vastavalt p.viit ja r.viit;
 - - -        ahelas p on n liiget, ahelas r on m liiget
 - - - Tulemus: antud ahelad ühildatud viitade ümberkorraldamise teel
 - - -          järgmiselt:
 - - -          kui p0 on parameetrina antud viida p väärtus, siis uus ahel,
 - - -          mis algab liikmest p0.viit, ühendab mõlemate lähteahelate
 - - -          kõik liikmed välja .a mittekahanemise järjekorras;
 - - -          tagastatakse viit ühendahela viimasele liikmele

 - - - p on ühtlasi ka viit viimasena ühildatud liikmele
   n = 0 ?
     - - - esimene ahel on ammendatud, minna teise ahela lõppu:
    [⋆ (m) r := r.viit
<------ (r)
   m = 0 ?
     - - - teine ahel on ammendatud, minna esimese ahela lõppu:
    [⋆ (n) p := p.viit
<------ (p)
     - - - valida järjekordne liige ahelast p või r
   p.viit.a ⩽ r.viit.a ?
     - - - p-liikme ühildamine (viitasid pole tarvis muuta)
    n −−; p := p.viit
  <- - - -
     - - - r-liikme ühildamine
    m −−
     - - - liige r lisada ühendahela lõppu:
    s := r.viit;  - - - s on lisatav liige
    r.viit := s.viit;  s.viit := p.viit;  p.viit := s
    p := s;  - - - ühendahela viimase liikme viit
```

Joonis 4.9: Lihtahelate ühildamine.

4.4 Sorteerimise ajalise keerukuse alampiir

Kõik eespool vaadeldud sorteerimismeetodid põhinevad antud järjendi elementide võrdlemisel, kas paarikaupa omavahel või siis ka mõne väljavalitud väärtusega. Nende meetodite ajalist keerukust hinnataksegi kui funktsiooni, mis väljendab sorteerimise käigus sooritatavate võrdlemiste arvu sõltuvust lähtejärjendi elementide arvust.

Olgu A suvaline seda tüüpi algoritm. Näitame, et n-elemendilise järjendi sorteerimisel on võrdlemiste arv $\Omega(n \log n)$.

Suvalise fikseeritud $n > 1$ korral saame konstrueerida võrdlemiste puu ehk nn. **otsustuste puu** järgmisel viisil. Puu lehtedeks võtame kõik võimalikud n elemendi permutatsioonid. Seega iga leht vastab parajasti ühele võimalikule tulemusele, mis saadakse algoritmi A rakendamisel n-elemendilise järjendi sorteerimiseks. Sõlmedeks selles puus on võrdlemistehted, kusjuures võrreldavad elemendid on antud oma konkreetsete indeksitega. Sõlme v vasakuks alluvaks on see võrdlemistehe, mis algoritmi A kohaselt tehakse siis, kui võrdlemine v andis negatiivse tulemuse, ja paremaks alluvaks võrdlemistehe, mis järgmisena täidetakse v positiivse tulemuse puhul. Lehe l ülemuseks on võrdlemistehe, mis sooritatakse viimasena enne järjendi lõplikku seadmist (sorteerimist) permutatsioonile l vastavasse järjestusse. Juureks on algoritmis A kõige esimesena teostatav võrdlemine. Igale konkreetsele n-elemendilisele järjendile vastab parajasti üks tee selles otsustuste puus, mis kujutab algoritmi täitmise kulgu läbi võrdlemiskäskude just selle järjendi sorteerimise ajal. Lühim tee juurtipust lehte vastab soodsaimale juhule, suurim teepikkus (ehk puu kõrgus) aga näitab halvimal juhul sooritatavate võrdlemiste arvu. Kuna selline puu on konstrueeritav iga vaadeldavat tüüpi algoritmi (ja iga n) korral, siis on sorteerimisalgoritmi ajalise keerukuse alumiseks piiriks otsustuste puu minimaalne kõrgus.

Arvutame nüüd m-lehelise kahendpuu vähima kõrguse. Kui ühetipulise puu kõrguseks lugeda null, siis täielikus kahendpuus kõrgusega $h \geqslant 0$ on ilmselt 2^h lehte. Täielik kahendpuu on ka kõige leherikkam, sest igale mittetäielikule kahendpuule saab lisada vähemalt ühe lehe ilma kõrgust suurendamata. Järelikult suvalise m-lehelise kahendpuu korral

$m \leqslant 2^h$ ehk $h \geqslant \log_2 m$.

st. m-lehelise kahendpuu kõrgus ei ole väiksem kui $\log_2 m$.

Otsustuste puus on $m = n!$ (s.o. n elemendi permutatsioonide arv) lehte, seega saame kõrguse hinnanguks $h \geqslant \log_2(n!)$. Stirlingi valemi

$$\sqrt{2\pi n}\left(\frac{n}{e}\right)^n \leqslant n! \leqslant \sqrt{2\pi n}\left(\frac{n}{e}\right)^{n+(1/12n)}$$

põhjal $n! > (\frac{n}{e})^n$. Niisiis

$$h \geqslant n \log_2 n - n \log_2 e.$$

Järelikult otsustuste puu minimalne kõrgus on $O(n \log n)$. Seega ka võrdlemistel põhineva suvalise sorteerimisalgoritmi ajaline keerukus halvimal juhul on $\Omega(n \log n)$.

4.5 Sorteerimise erimeetodeid

Eelmises jaotises leitud keerukuse alampiir on üldise iseloomuga ning kehtib juhul, kui ülesandeks on suvalise etteantud arvujärjendi sorteerimine. Tegelikult saab teatavaid mõnevõrra kitsendatud sorteerimisülesandeid lahendada ka $O(n)$ ajaga. Lihtsaks näiteks on selline juht, kus piirdutakse järjenditega, milles vaid vähima väärtusega element on "rivist väljas". Siis pole mõtet rakendada $O(n \log n)$ meetodit, vaid võib otsida vähima väärtuse järjendis ja pista selle siis esimesele kohale kokku $\Theta(n)$ ajaga. Nagu alljärgnevas näeme, saab lineaarse ajaga lahendada ka märksa vähem kitsendatud sorteerimisülesandeid. Paraku kasutavad kõik vaadeldavad erimeetodid abimälu mahus $\Omega(n)$.

Loendamismeetodi korral eeldatakse, et $1 \leqslant a_i \leqslant k$ iga $i = 1, 2, \dots, n$ korral, ning k on $O(n)$. Järjendi iga elemendi jaoks leitakse temast väiksemate elementide arv l_i. Juhul, kui järjendis ei ole korduvaid väärtusi, näitabki suurus $l_i + 1$ kohta, kus tulemusjärjendis peab paiknema a_i esialgne väärtus. Mõnevõrra komplitseerib loendamismeetodi algoritmi (vt. joonis 4.10) vajadus arvestada ka korduvate väärtustega. Loendamismeetod on stabiilne ja ajalise keerukusega $O(n)$.

Positsioonimeetod eeldab, et sorteeritavateks väärtusteks on nö. liitvõtmed. Iga väärtus koosneb d positsioonist, positsioonis aga paikneb arv hulgast $K = \{0, 1, \dots k\}$, kus k on $O(n)$. Seega sorteeritakse d-numbrilisi arve, mis on kujutatud positsioonilises arvusüsteemis alusel $k + 1$. Vastava algoritmi (vt. joonis 4.11) põhitsüklis vaadatakse antud järjend läbi d korda, i-ndal tsükli sammul sorteeritakse väärtused lõpust i-ndal kohal oleva positsiooni järgi (mittekahanevalt). Alamalgoritmina saab rakendada loendamismeetodit. Peale kiiruse on siin tähtis ka viimase stabiilsus. Algoritmi töö tulemuseks on positsioonide kaupa leksikograafiliselt sorteeritud järjend. Näiteks $n = 7,\ d = 3,\ K = \{0, 1, \dots, 9\}$ ja lähtejärjendi

620 457 657 839 436 720 355

puhul toimub sorteerimine järgmiselt:

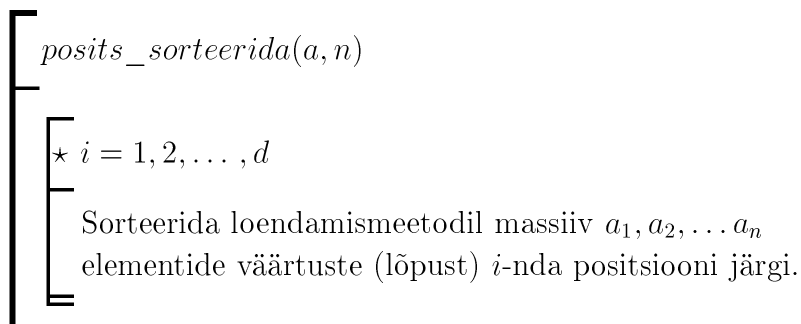
$i = 1:$ 620 720 355 436 457 657 839
$i = 2:$ 620 720 436 839 355 457 657
$i = 3:$ 355 436 457 620 657 720 839

```
loend_sorteerida(a, n)
  --- abimassiivid: c_1, c_2, ..., c_k;  b_1, b_2, ..., b_n
  [⋆ (j = 1, 2, ..., k) c_j := 0; --- loendurite nullimine
  [⋆ (i = 1, 2, ..., n) c_{a_i} ++
  --- c_j (1 ⩽ j ⩽ k) on nende elementide arv, mille väärtuseks on j
  [⋆ (i = 2, 3 ..., k) c_i += c_{i-1}
  --- c_j (1 ⩽ j ⩽ k) on nende elementide arv, mille väärtus ei ületa j
  --- sorteerida massiivist a massiivi b:
  ⋆ i = n, n - 1, ..., 1
     j := c_{a_i} --
      --- j on koht, kuhu paigutada a_i
     b_j := a_i
  --- tulemus on massiivis b
  [⋆ (i = 1, 2, ..., n) a_i := b_i;  --- tulemus salvestada massiivi a
```

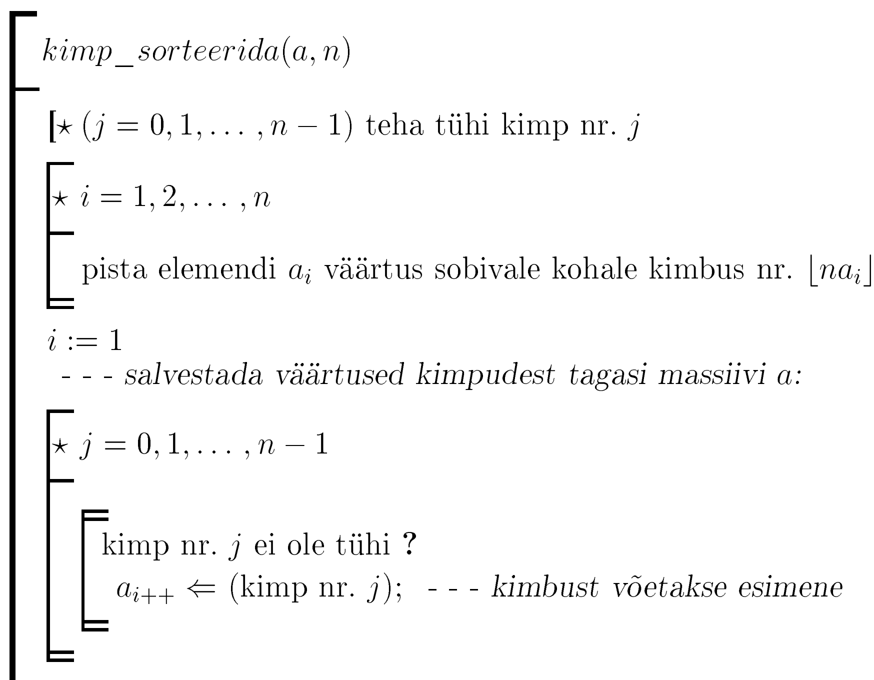
Joonis 4.10: Loendamismeetodi algoritm.

Positsioonimeetodi ajaline keerukus on $O(n)$, täpsemalt $O(dn + nk)$.

Kimbumeetodi puhul on eelduseks, et kõik väärtused kuuluvad poollõiku $[0, 1)$ ja on sellel ühtlaselt jaotunud. Algoritmis (vt. joonis 4.12) kasutatakse abistruktuuri, mis koosneb n lihtahelast. Ahelate päised salvestatakse massiivina $p_0, p_1, \ldots, p_{n-1}$. Kimbumeetodi ajaline keerukus on samuti $O(n)$.



Joonis 4.11: Positsioonimeetodil sorteerimise põhimõte.



Joonis 4.12: Kimbumeetod.

Peatükk 5

Sõnetöötlus

Olgu antud lõplik sümbolite hulk ehk **tähestik** S. **Sõneks** tähestikus S nimetatakse suvalist selle hulga sümbolitest moodustatud järjendit. Erijuhul võib sõnes olla ka null sümbolit; sellist **tühja sõnet** tähistame ϵ. Sõne t elemente tähistame $t[1], t[2], \ldots$, alamsõnet (t alamjärjendit) $t[i]t[i{+}1] \ldots t[j]$ aga tähisega $t[i \ldots j]$.

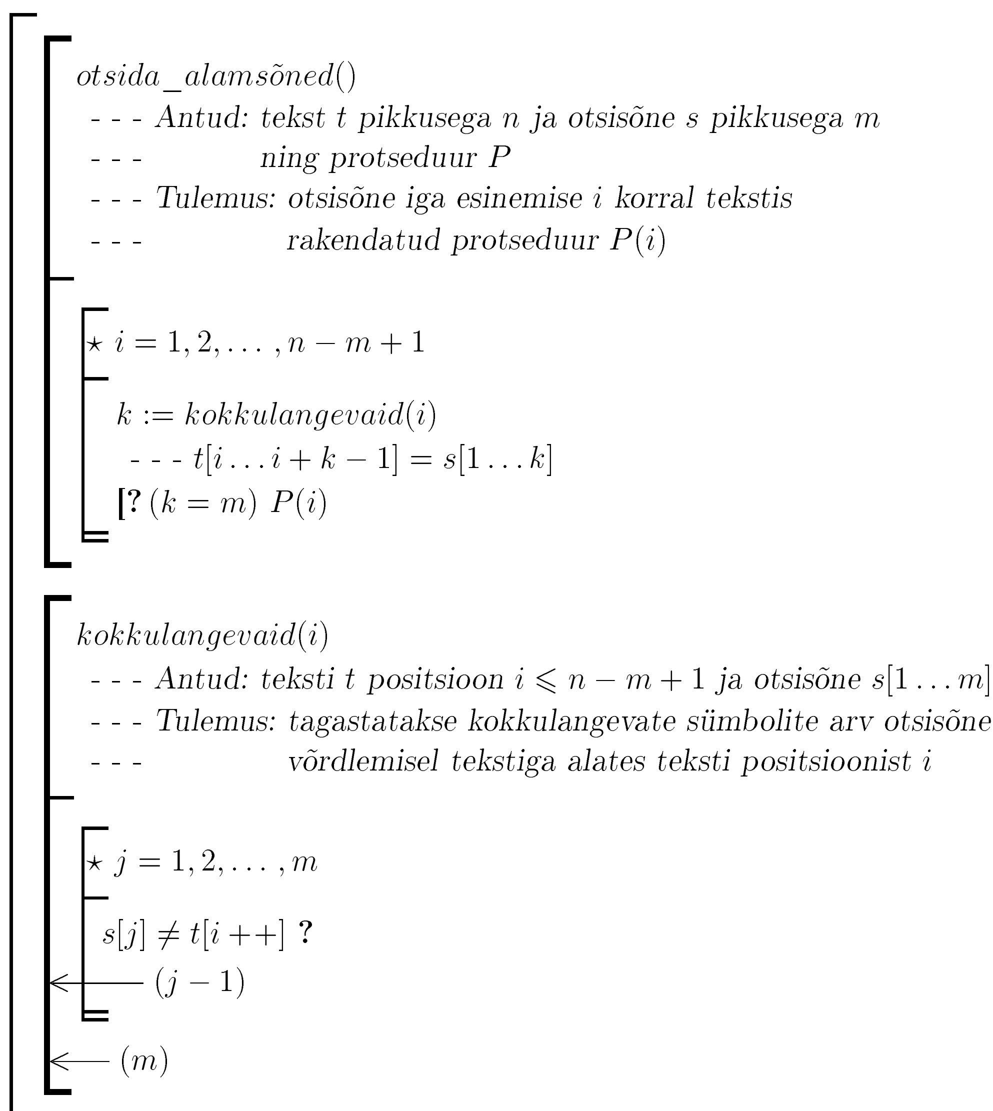
5.1 Alamsõne otsimine

Sõne s **esineb** sõnes t positsioonis i, kui leidub j nii, et $t[i \ldots j] = s$; sel korral indeksit i nimetame ka s esinemiskohaks ehk **esinemiseks** sõnes t.

Alamsõne otsimise ülesande korral on antud kaks sõnet t ja s. Tuleb leida kõik sõne s esinemised sõnes t ning iga esinemise puhul sooritada teatav tegevus P. Eeldatakse, et protseduuri P sooritamise käigus sõned t ja s ei muutu. Alamsõne otsimisega tuleb tihti tegeleda näiteks tekstitoimetites. Antud sõnet t, milles alamsõnet otsitakse, nimetame edaspidi **tekstiks**, otsitavat sõnet aga **otsisõneks**. Käesolevas jaotises kirjeldatavates algoritmides eeldatakse, et globaalsetena on antud

– tekst $t = t[1 \ldots n]$ ja selle pikkus n;
– otsisõne $s = s[1 \ldots m]$ ja selle pikkus m;
– protseduur $P(k)$, kus parameetriks k on otsisõne esinemine tekstis.

Joonisel 5.1 on esitatud lihtne otsimisalgoritm. Viimase ajaline keerukus on $O((n - m + 1)m)$, kuna põhitsüklit korratakse $n - m + 1$ korda ja



Joonis 5.1: Lihtne alamsõne otsimise algoritm.

funktsioonis *kokkulangevaid* tuleb halvimal juhul teha m sammu. Kui näiteks otsitakse alamsõnesid, mille pikkus on kuni $\lfloor n/2 \rfloor$, siis lahendusaeg on $O(n^2)$.

Leiduvad aga märksa tõhusamad moodused alamsõne otsimise ülesande lahendamiseks. Paremate meetodite ajaline keerukus on $O(n+m)$. Peamiseks

efektiivsuse tõstmise teeks on teksti ja otsisõne sümbolite varem sooritatud võrdlemistulemuste ärakasutamine parajasti käsiloleval otsimise etapil. Oletame näiteks, et algoritmi *otsida_alamsõned* (vt. joonis 5.1) tsükli i-ndal sammul leitakse $k : 0 < k < m$. Kuigi $k < m$ annab tunnistust otsisõne sobitamise katse ebaõnnestumisest (teksti positsioonile i), oleme tegelikult saanud siiski rohkem informatsiooni. Nimelt on ju nüüd tuvastatud, et otsisõne esimesed k sümbolit langevad kokku teksti vastava osaga: $s[1 \ldots k] = t[i \ldots i + k - 1]$. Vaadeldavas lihtsas algoritmis seda täiendavat teadmist ignoreeritakse ning alustatakse järgmist sobitamise katset lihtsalt teksti järgmisest sümbolist $t[i{+}1]$ (ja otsisõne algusest). Sellega seoses tehakse mõnevõrra tarbetut tööd: nüüd ju põhitsükli järgmisel sammul rakendatavas funktsioonis *kokkulangevaid* esimesed $k - 1$ võrdlemist tehakse sisuliselt otsisõne enda sümbolite vahel. Selle jao võrdlemistulemus sõltub aga ainult otsisõnest ning on võimalik kindlaks teha otsisõne eelneva analüüsimisega. Otsisõne eeltöötlus ongi kiiremate otsimismeetodite põhiideeks.

Olgu näiteks otsisõneks s = 'aabcaabcda'. Muuhulgas saab siis eelneva analüüsiga kindlaks teha järgmise tõiga: kui selle otsisõne sobitamise katsel alates teksti positsioonist i ilmneb erinevus tekstiga alles otsisõne üheksanda sümboli ('d') puhul, siis pole mõtet püüda seda otsisõnet sobitada alates teksti positsioonidest $i + 1, i + 2$ ega $i + 3$. Veelgi enam, otsisõne sisalduvust tuleb küll kontrollida alates teksti positsioonist $i + 4$, kuid seejuures võib vahele jätta nelja esimese sümboli võrdlemise, sest need on kindlasti teksti vastavate sümbolitega võrdsed:

		i								j					
tekst:	...	a	a	b	c	a	a	b	c	x	y	y	y	y	...
otsiõne:		a	a	b	c	a	a	b	c	d	a				
uus katse:						a	a	b	c	a	a	b	c	d	a

Järgmine võrreldav sümbolipaar on antud juhul seega $s[5]$ ja x. Kokkuvõttes kehtib antud otsisõne s puhul väide: kui sõne s võrdlemisel tekstiga esimene erinevus ilmneb sümbolite $s[9]$ ja $t[j]$ korral, siis järgmisena tuleb võrrelda sümboleid alates paarist $s[5]$ ja $t[j]$.

Analoogilise analüüsi saab teha otsisõne iga positsiooni jaoks. Üldiselt, kui otsisõne s esimesed k sümbolit langevad kokku tekstis sümbolile $t[j]$ eelnevate sümbolitega ja $s[k+1] \neq t[j]$, siis võib alates sümbolist $t[j]$ võrdlemist

Tabel 5.1: Prefiksfunktsiooni näide.

i	1	2	3	4	5	6	7	8	9	10
$s[i]$	a	a	b	c	a	a	b	c	d	a
$\pi(i)$	0	1	0	0	1	2	3	4	0	1

jätkata otsisõne sellisele prefiksile järgnevast sümbolist, mis on otsisõne osas $s[1 \dots k]$ sufiksiks. Arusaadavalt tuleb jätkata pikima sellise prefiksi viimasest sümbolist. Vaadeldud näites osutub sõnes $s[1 \dots 8]$ = 'aabcaabc' pikimaks prefiksiks, mis leidub sufiksina ka selle osa lõpus, sõne 'aabc'.

Prefiksfunktsiooniks antud otsisõne $s = s[1 \dots m]$ korral nimetatakse funktsiooni π, mis igale indeksile $k \in \{1, 2, \dots, m\}$ seab vastavusse $\pi(k) \in \{0, 1, \dots, k-1\}$ nii, et $\pi(k)$ on sõne $s[1 \dots k]$ pikima sellise prefiksi pikkus, mis võrdub sõne $s[1 \dots k]$ mingi (päris)sufiksiga.

Ülaltoodud näites vaadeldud otsisõne korral kasutasime tegelikult tema prefiksfunktsiooni väärtust $\pi(8) = 4$. Prefikfunktsiooni kõik väärtused selle otsisõne korral on toodud tabelis 5.1.

Joonisel 5.2 esitatud **Knuth-Morris-Pratti otsimisalgoritmis** arvutatakse kõigepealt antud otsisõne prefiksfunktsiooni väärtused massiivina $\pi_1, \pi_2, \dots, \pi_m$. Viimaseid kasutatakse järgnevas otsimise tsüklis kiiremaks edasiliikumiseks neil juhtudel, mil leitakse järjekordne erinevate sümbolite paar $s[j+1] \neq t[i]$ (ja $j > 0$). Saab näidata, et sellise otsimismeetodi ajaline keerukus on $O(n+m)$.

Prefiksfunktsiooni arvutamisel (vt. joonis 5.3) kiirendatakse järjekordse väärtuse π_k leidmist, kasutades juba leitud eelmisi väärtusi $\pi_1, \pi_2, \dots, \pi_{k-1}$. Sisuliselt on siin ju samuti tegemist alamsõne (prefiksi) otsimisega (sufiksi kohal).

Knuth-Morris-Pratti algoritmi idee rafineeritud edasiarenduseks võib pidada **Boyer-Moore'i algoritmi**. Selle kohaselt proovitakse otsisõne tekstiga sobitada paremalt vasakule, st. otsisõne viimasest sümbolist alustades. Teksti positsioone vaadeldakse ikka vasakult paremale liikudes. Niisugusel juhul on eeliseks see, et esimese erinevuse $s[k] \neq t[j]$ ilmnemisel saab võrdlemisi jätkata paarist $s[\lambda(t[j]) + 1], t[j+1]$, kus λ on funktsioon, mis vaadeldava

```
KMP_otsida_alamsõned()

  - - - abimassiiv: π_1, π_2, ..., π_m
  arvutada_prefiksfunktsioon(π);   - - - vt. joonis 5.3
  j := 0

  ⋆ i = 1, 2, ..., n

      - - - s[1...j] esineb t[1...i-1] lõpus
    j = 0 ?
      - - - otsisõnega tuleb alustada algusest

      s[1] ≠ t[i] ?
        - - - kohe erinevus
  <- - - - - - - - -

      - - - s[1] = t[i] ja j = 0
    <----
    s[j+1] = t[i]  ?
    <----
      - - - leitud erinevus s[j+1] ≠ t[i]
     j := π_j
      - - - järgmine võrdlemisele tulev sümbol otsisõnes on s[j+1]

    - - - on leitud kokkulangevus s[j+1] = t[i]
    [? (++j = m) P(i-m+1); j := π_j
```

Joonis 5.2: Knuth-Morris-Pratti algoritm alamsõne otsimiseks.

tähestiku igale sümbolile σ seab vastavusse σ viimase esinemise (indeksi) sõnes s; $\lambda(\sigma) = 0$, kui σ ei esine sõnes s. Nii nagu prefiksfunktsioon π, nii on ka λ eelnevalt tabuleeritav. Loomulikult arvestatakse erinevuse põhjustanud tekstisümboli viimase esinemiskohaga otsisõnes vaid siis, kui $\lambda(t[j]) < k$. Boyer-Moore'i algoritmis kombineeritakse λ väärtuste kasutamine prefiksfunktsioonil põhineva liikumisega otsisõnes. Täpsemalt me seda siis ei käsitle.

```
arvutada_prefiksfunktsioon(π)
 --- Antud: massiiv π_1, π_2, ..., π_m funktsiooni väärtuste salvestamiseks
 --- Tulemus: otsisõne s[1...m] prefiksfunktsiooni väärtused salvestatud,
 ---          π_i on funktsiooni väärtus kohal i

π_1 := 0;  j := 0
⋆ k = 2, 3, ..., m
   --- on leitud π_1, ..., π_{k-1}
      --- s[1...j] esineb s[1...k − 1] lõpus
    j = 0 ?
      --- alustada algusest
     [? (s[1] = s[k])  j ++
    ←
    s[j + 1] = s[k] ?
     j ++
    ←
      --- leitud erinevus s[j + 1] ≠ s[k]
     j := π_j;  --- järgmine võrdlemisele tulev sümbol prefiksis
   π_k := j
```

Joonis 5.3: Prefiksfunktsiooni arvutamine.

Hoopis teistsugust alamsõne otsimise moodust rakendatakse **Rabin-Karpi algoritmis**. Selle idee kohaselt käsitletakse m-sümbolilist sõnet tähestikus S kui m-kohalist arvu positsioonilises arvusüsteemis alusel $d = |S|$. Numbriteks on sümbolite järjenumbrid $0, 1, \ldots, |S| - 1$ tähestikus S. Teiste sõnadega, „originaalsõne“ asemel vaadeldakse vastavatest numbritest koosnevaid järjendeid ehk arve.

Näiteks, kui $S = \{\text{a, b, c, d}\}$, siis sõnet 'acddab' esindab neljandsüsteemi arv $023301_4 = 0 \times 4^5 + 2 \times 4^4 + 3 \times 4^3 + 3 \times 4^2 + 0 \times 4^1 + 1 \times 4^0 =$ $((((0 \times 4 + 2) \times 4 + 3) \times 4 + 3) \times 4 + 0) \times 4 + 1 = 753$.

Üldiselt, sõnele $u = u[1 \ldots k]$ vastavaks arvuks on

$\alpha(u) = (\ldots((u[1]d + u[2])d + u[3])d + \ldots + u[k-1])d + u[k]$,
mille väljaarvutamiseks kulub $k-1$ liitmist ja samapalju korrutamistehteid.

Alamsõne otsimise ülesande põhimõtteline lahendusskeem on siis väga lihtne: tuleb leida otsisõnele vastav arv $\alpha(s)$, seejärel teksti kõigile m-sümbolilistele alamsõnedele vastavad arvud $\alpha(t[1\ldots m])$, $\alpha(t[2\ldots m+1])$, $\ldots$, $\alpha(t[n-m+1\ldots n])$. Sõne esineb teksti positsioonis i parajasti siis, kui $\alpha(s) = \alpha(t[i\ldots i+m-1])$. Seejuures ainult esimese alamõne jaoks $\alpha(t[1\ldots m])$ arvutamise aeg on $O(m)$, iga järgmise saame ajaga $O(1)$, eemaldades eelmisest esimese numbri ja lisades saadud arvu lõppu uue numbri:
kui $\alpha(t[j\ldots j+m-1]) = y$, siis $\alpha(t[j+1\ldots j+m]) = (y - Dt[j])d + t[j+m]$, kus $D = d^{m-1}$.

Näiteks olgu tähestikus {a, b, c, d} tekstiks 'bcacddabac' ja otsisõneks ülalvaadeldud 'acddab', millele vastab arv 753. Teksti esimesed kuus sümbolit 'bcacdd' annavad arvu $120233_4 = ((((1\times4+2)\times4+0)\times4+2)\times4+3)\times4+3 = 1583$. Järgmisele kuuikule 'cacdda' (202330_4) vastav arv saadakse lihtsamalt: $(y - Dt[j])d + t[j+m] = (1583 - D\times1)\times4+0 = (1583-4^5)\times4 = 2236$. Analoogiliselt saadakse kolmandale kuuikule 'acddab' (023301_4) vastav arv $(y - Dt[j])d + t[j+m] = (2236 - D\times2)\times4+1 = (2236-4^5\times2)\times4+1 = 753$. Kuna viimane on võrdne otsisõne põhjal arvutatud väärtusega, siis on tuvastatud, et otsisõne esimeseks esinemiseks on 3, st. antud otsisõne esineb antud tekstis alates kolmandast positsioonist (ega esine eespool).

Praktiliselt seda skeemi muidugi rakendada ei saa, sest sõnedele vastavad arvud tulevad väga suured. Rabin-Karpi algoritmi (vt. joonis 5.4) kohaselt tehakse aga kõik arvutused (täpsemalt – omistamised) *modulo* q, kus q on sobivalt valitud algarv. Nüüd otsisõne s ja mingi tekstiosa u jaoks arvutatud suuruste $x = \alpha(s) \bmod q$ ja $y = \alpha(u) \bmod q$ erinevusest järeldub küll $s \neq u$, kuid $x = y$ korral tuleb veel teha sõnede s ja u täielik sümbolhaaval võrdlemine. Sellel vaatamata on Rabin-Karpi algoritmi ajaline keerukus praktiliselt $O(n)$.

Joonisel 5.4 toodud algoritmi realiseerimisel arvutiprogrammina tuleks q võtta võimalikult suur, kuid nii, et korrutamine $(d+1)q$ ei annaks ületäitumist. Esimese numbri eemaldamise operatsioon on soovitatav sooritada mitte kujul $y := (y - t[i]D) \bmod q$, vaid hoopis kujul $y := (y + dq - t[i]D) \bmod q$, et vältida jäägi leidmist negatiivsest arvust.

```
RK_otsida_alamsõned()
D := 1
[* (m − 1) D := (Dd) mod q
 - - - D = d^(m−1) mod q
x := 0;  y := 0
* i = 1, 2, ... , m
    x := (xd + s[i]) mod q
    y := (yd + t[i]) mod q
 - - - x on otsisõnele s vastav arv
 - - - y on teksti algusosale t[1 ... m] vastav arv
i := 1
    - - - y on arv, mis vastab teksti osale t[i ... i + m − 1]
    x = y ?
        - - - võrdsete arvude korral tuleb veel sõnesid võrrelda
        [? (kokkulangevaid(i) = m) P(i); - - - vt. joonis 5.1
    i = n − m + 1 ?
  <—
    - - - leida arv y järgmise tekstiosa jaoks
    y := (y − t[i]D) mod q;  - - - eemaldada esimene number
    y := (yd + t[i + m]) mod q;  - - - lisada lõppu uus number
    i ++;  - - - järgmine positsioon tekstis
```

Joonis 5.4: Rabin-Karpi algoritm.

5.2 Teksti pakkimine

Tavaliselt kasutatakse arvutis teksti säilitamisel fikseeritud pikkusega sümbolikoode. Näiteks ASCII kooditabelis vastab igale sümbolile 8-bitiline kahendkood. Sel korral n-sümbolilise teksti kujutiseks on bitisõne pikkusega $8n$. Tekstide kompaktsemaks esitamiseks ehk **pakkimiseks** arvuti välismälus võetakse kasutusele muutuva pikkusega sümbolikoodid. Efekt saavutatakse sel teel, et sagedamini esinevatele sümbolitele seatakse vastavusse lühemad koodid. Pakkimisprogrammides arvestatakse tavaliselt sümbolite esinemissagedust just pakitavas tekstis, koostades selle jaoks oma spetsiifilise kooditabeli. Viimane tuleb muidugi säilitada koos pakitud (st. selle tabeli alusel kodeeritud) tekstiga, et tagada dekodeerimise võimalus. Teatava kokkuhoiu annab juba see, et antud tekstiga seotud kooditabel on reeglina lühem kui üldine (ASCII) tabel.

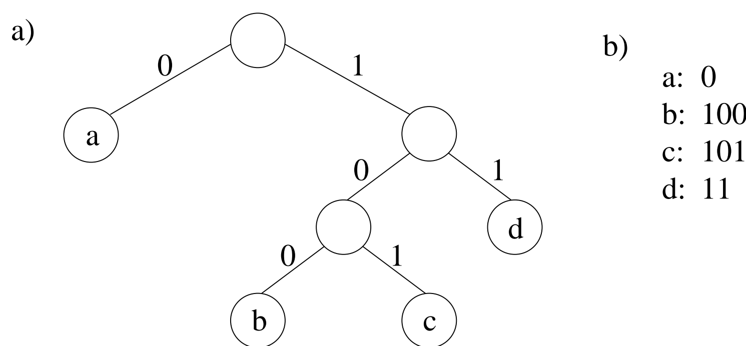
Olgu t 320-sümboliline tekst, milles on üldse kokku vaid 13 erinevat sümbolit. Selle teksti ASCII-kujutis on 2560-bitiline. Kodeerides need 13 sümbolit aga näiteks 4-bitiliste kahendkoodidena, saame pärast teksti t ümberkodeerimist 1280-bitilise kujutise. Mälu kokkuhoid on seega 50% (tõsi küll, seda ilma uue kooditabeli säilitamiseks tarvilikku mäluvajadust arvestamata).

Vaatleme nüüd teksti kujutamist muutuva pikkusega sümbolikoodide abil, millest tegelikult tuleneb peamine mälu kokkuhoid pakkimisel. Tavalise kooditabeli asemel on siin aluseks nn. koodipuu.

Sümbolite hulga S **koodipuuks** nimetatakse kahendpuud, mille lehtedeks on sümbolid hulgast S ja milles iga kaar vasakule alluvale on märgendatud numbriga 0 ning kaar paremale alluvale numbriga 1 (vt. joonis 5.5).

Sümboli $s \in S$ **prefikskoodiks** antud koodipuu korral nimetatakse märgendite (biti)järjendit teel juurtipust lehte s. Ilmselt ei pruugi prefikskoodid olla kõik ühepikkused.

Tegemist on tõepoolest kodeerimisega, sest (1) sümboli selline kood on ühene (kuna leidub vaid üks tee juurest lehte), (2) erinevatele sümbolitele vastavad erinevad koodid (liikudes juurtipust alla ja juhindudes ühesugustest bitijärjenditest ei saa jõuda erinevatesse lehtedesse). Nimetus *prefikskood* tuleneb omadusest, et ühegi sümboli kood ei saa olla teise sümboli koodis



c) ‘babcddabbbaa’ ⟶ 1000100101111010010010000

Joonis 5.5: Koodipuu (a), prefikskood (b) ja teksti kodeerimine (c).

prefiksiks. Tekstide kodeerimiseks sobib prefikskood eriti hästi seetõttu, et teksti saab kodeerida tavalisel viisil, kirjutades tekstis esinevate sümbolite prefikskoodid lihtsalt üksteise järele (mitte-prefikskood taolist kodeerimisviisi ei võimaldaks). Saadud bitijärjendi (s.o. teksti koodi) dekodeerimine toimub koodipuu alusel. Järjekordne sümbol leitakse liikudes koodipuu juurtipust alla, juhindudes seejuures teksti koodis leiduvatest bittidest (vasakult paremale); jõudnud leheni, on ka sümbol leitud: selleks ongi lehes olev sümbol. Järgmine bitt teksti koodis on juba esimeseks järgmise sümboli koodis ning vastavalt tuleb ka koodipuus jälle alustada juurest.

Ühe ja sama sümbolihulga S jaoks saab ehitada mitmeid koodipuid. Käesoleval juhul huvitab meid sellise koodipuu leidmine, kus suure esinemissagedusega sümbolid oleksid juurele võimalikult lähedal. Antud teksti korral nimetatakse **Huffmani puuks** sellist koodipuud, millele vastavalt prefikskoodidena kodeeritud teksti pikkus osutub minimaalseks.

Huffmani puu konstrueerimise algoritm, nn. Huffmani algoritm leidub joonisel 5.6 . Moodustatava puu tippudeks on kirjed $< s, f, vasak, parem >$, kus väljal s asub sümbol hulgast S, f on reaalarv (lehe korral sümboli s esinemissagedus), $vasak$ ja $parem$ – viidad alluvatele. Sõlmede korral ei kasutata välja s ning lehtede korral viitasid alluvatele. Tippude hulk kasvab

```
Huffman(S)
 - - - Antud: sümbolite hulk S, kus iga s ∈ S korral määratud veel tema
 - - -           sagedus s.f
 - - - Tulemus: Huffmani puu hulga S jaoks;
 - - -             tagastatakse konstrueeritud puu juur

 - - - abistruktuur: eelistusjärjekord Q (võtme f järgi)
 Q := S
 - - - Q on nende tippude hulk, millel pole veel ülemust

 ⋆ |S − 1|

   teha uus tipp r
    - - - selle alluvateks võtta kaks kõige väiksema f väärtusega tippu:
   r1 ⇐ Q;  r.vasak := r1
   r2 ⇐ Q;  r.parem := r2
   r.f := r1.f + r2.f;   - - - f uuele tipule
    - - - tipul r pole veel ülemust, lisada see tipp hulka Q:
   Q ⇐ r

←— (r)
```

Joonis 5.6: Huffmani algoritm optimaalse koodipuu leidmiseks.

algoritmi täitmise käigus. Eelistusjärjekorrana Q (võtme f suhtes) säilitatakse neid tippe, millel veel ülemust ei ole; Q algväärtuseks on hulgale S vastavate kirjete (tulevaste lehtede) hulk. Igal tsükli sammul eemaldatakse hulgast Q kaks kõige väiksema võtmega tippu ja lisatakse üks uus tipp (sõlm), mille võtmeks omistatakse eemaldatud tippude võtmete summa. Lisatud tipu alluvateks määratakse hulgast Q eemaldatud tipud. Kuna kokkuvõttes hulga Q elementide arv väheneb igal sammul ühe võrra, siis pärast $|S| - 1$ sammu on sinna jäänud vaid üks tipp. See tähendab, et kõikidele teistele tippudele on ülemused määratud, järelikult on puu valmis ning juurtipuks hulgas Q olev ainuke element.

Kirjeldatud Huffmani algoritm on tüüpiline nn. **ahne algoritm**, kus igal sammul valitakse see võimalus, mis hetkel parim näib. Antud juhul lisatakse varem puusse väiksema sagedusega sümbolid, lootuses, et siis nad ka lõpuks puus suhteliselt allapoole jäävad ning kokkuvõttes optimaalne koodipuu saadakse. Saab tõestada, et see lootus ennast siin tõepoolest õigustab (käesolevas me tõestust ei esita). Reeglina on ahne algoritm lihtne, kuid nõuab mahukat põhjendamist. On selge, et mitte alati ei vii ahne taktika sihile. Näiteks kui soovitakse odavalt sooritada pikem reis ühest linnast teise, kusjuures nende linnade vahel puudub otseühendus, siis ahne meetodi kohaselt tuleks ümberistumiseks valida see kolmas linn, kuhu esimesest kõige odavamini saab. Kokkuvõttes võib aga nii alustatud reis hoopis kulukamaks osutuda kui mõne teise valiku puhul.

Teksti pakkimise protseduur on põhimõtteliselt järgmine: arvutatakse iga ASCII sümboli esinemise suhteline sagedus antud tekstis, leitakse Huffmani puu, koostatakse selle alusel prefikskoodide tabel (jättes loomulikult välja tekstis puuduvad sümbolid) ja kodeeritakse tekst prefikskoodide abil. Pakitud tekst salvestatakse koos dekodeerimiseks tarviliku koodipuuga.

5.3 Pikima ühise osasõne otsimine

Antud sõne osasõne saadakse temast mõnede sümbolite väljajätmise teel. Osasõne erijuhtudeks on tühisõne (välja on jäetud kõik sümbolid) ja sõne ise (välja on jäetud null sümbolit).

Täpsemalt, sõne $u = u[1 \dots k]$ $(k \geqslant 0)$ on sõne $s = s[1 \dots m]$ **osasõneks**, kui $u = \epsilon$ või $u = s$ või sõnel s leidub rangelt kasvav indeksite järjend $i_1, i_2, \dots, i_k$ selliselt, et $s[i_j] = u[j]$ iga $j = 1, 2, \dots, k$ korral. Näiteks 'bbd' on sõne 'aabcabcda' osasõne (indeksitega 3,6,8 pikemal sõnel).

Antud kahe sõne s ja t korral öeldakse, et sõne u on nende **ühine osasõne** ehk **ühissõne**, kui u on nii s kui ka t osasõneks.

Käesolevas vaatleme ülesannet, mille kohaselt tuleb leida kahe sõne pikim ühissõne. Näiteks sõnede 'aabcbdab' ja 'bbdcaba' puhul on selle ülesande vastuseks 'bbdab'. Sõnede s ja t pikimat ühissõnet tähistame ka $P\ddot{U}(s, t)$. Kõigi võimaluste lihtne läbivaatamine, kus kontrollitakse sõne $s = s[1 \dots m]$ iga osasõne sisalduvust sõnes t, oleks antud juhul liiga aeganõudev. On ju m-sümbolilise sõne osasõnede arvuks indeksite hulga $\{1, 2, \dots, m\}$ alamhulkade arv 2^m. Õnneks osutub võimalikuks pikima ühissõne otsimise ülesanne taandada väiksema arvu (täpsemalt mn) lihtsamate alamülesannete lahendamisele.

Tähistame sõne x esimesest l sümbolist koosnevat prefiksit tähisega $x|_l$, st. $x|_l = x[1 \dots l]$. Pikima ühissõne otsimise ülesannet võimaldavad lihtsustada järgmised omadused.

Olgu sõnede $s = s[1 \dots m]$ ja $t = t[1 \dots n]$ korral leitud nende pikim ühissõne $u = u[1 \dots k]$.

Omadus 1. *Kui* $s[m] = t[n]$, *siis* $u[k] = s[m] = t[n]$ *ja* $P\ddot{U}(s|_{m-1}, t|_{n-1}) = u|_{k-1}$.

Tõepoolest, kui $u[k] \neq s[m]$, siis sõne $u[1] \dots u[k]s[m]$ oleks s ja t ühissõneks (pikkusega $k+1$), mis oleks vastuolus eeldusega $u = P\ddot{U}(s, t)$. Järelikult $u[k] = s[m] = t[n]$. Kui aga nii, siis $u|_{k-1}$ on $s|_{m-1}$ ja $t|_{n-1}$ ühine osasõne pikkusega $k-1$. Näitame, et $u|_{k-1}$ on ka $s|_{m-1}$ ja $t|_{n-1}$ pikimaks ühissõneks. Oletame vastuväiteliselt, et leidub $s|_{m-1}$ ja $t|_{n-1}$ ühissõne w, mille pikkus on suurem kui $k-1$. Sel korral osutub sümboliga $s[m] = t[n]$ täiendatud sõne $ws[m]$ (mille pikkus on suurem kui k) sõnede s ja t ühissõneks. See on aga

vastuolus eeldusega, et s ja t pikimaks ühissõneks on u (pikkusega k).

Omadus 2. *Kui* $s[m] \neq t[n]$, *siis* $u[k] \neq s[m]$ *korral* $P\ddot{U}(s|_{m-1}, t) = u$.

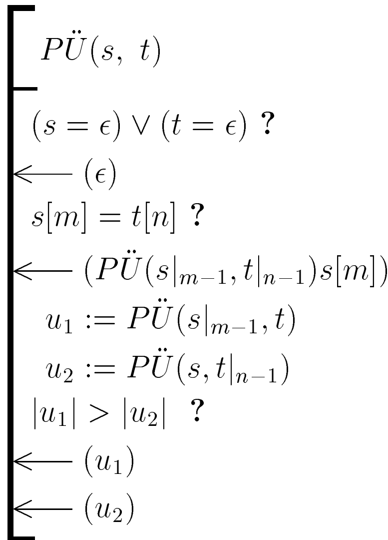
Tõepoolest, kui $u[k] \neq s[m]$, siis sõne u pikkusega k peab olema sõnede $s|_{m-1}$ ja t ühissõneks. Kui leiduks sõnede $s|_{m-1}$ ja t k-st suurema pikkusega ühissõne w, siis oleks w ka s ja t ühissõneks. See aga on vastuolus eeldusega, et s ja t pikimaks ühissõneks on u (pikkusega k).

Omadus 3. *Kui* $s[m] \neq t[n]$, *siis* $u[k] \neq t[n]$ *korral* $P\ddot{U}(s, t|_{n-1}) = u$. (Sümmeetriline omadusega 2.)

Tänu nendele omadustele saame sõnede $s = s[1 \ldots m]$ ja $t = t[1 \ldots n]$ pikima ühissõne u leida järgmise üldise skeemi alusel, vaadeldes vaid erineva pikkusega prefikseid:

kui $s[m] = t[n]$, siis $u = P\ddot{U}(s|_{m-1}, t|_{n-1})s[m]$;

kui $s[m] \neq t[n]$, siis u on pikim kahest sõnest $P\ddot{U}(s|_{m-1}, t)$ ja $P\ddot{U}(s, t|_{n-1})$.



Joonis 5.7: Pikima ühissõne otsimise eksponentsiaalse keerukusega algoritm.

Üldise lahendusskeemi otsene realiseerimine annab äärmiselt ebaefektiivse rekursiivse algoritmi (vt. joonis 5.7). Selle ajaline keerukus on eksponentsiaalne nö. "kattuvate" alamülesannete tõttu: algoritmi täitmisel lahendataks ühte ja sama konkreetset alamülesannet korduvalt. Näiteks selleks, et lei-

da $P\ddot{U}(s|_8, t|_8)$ võib tarvis minna nii suurust $P\ddot{U}(s|_7, t|_7)$ kui ka $P\ddot{U}(s|_7, t|_8)$ kui ka $P\ddot{U}(s|_8, t|_7)$. Rekursiivse algoritmi teeb siinjuures kohmakaks see, et niisuguste alamülesannete tulemusi läheb vaja korduvalt: ka $P\ddot{U}(s|_8, t|_7)$ arvutamisel võib vaja minna suurust $P\ddot{U}(s|_7, t|_7)$ jne. Rekursiooni puhul aga hakatakse kõiki vajalikke alamülesandeid lahendama nö. algusest peale.

Lihtsama näitena rekursiivse meetodi sobimatusest võib tuua n-nda Fibonacci arvu f_n leidmise ülesande. Vahetult definitsioonist

$f_0 = 0,$

$f_1 = 1,$

$f_n = f_{n-1} + f_{n-2}$, kui $n > 1$

lähtudes koostatud rekursiivne algoritm osutub tarbetult töömahukaks.

Kattuvate konkreetsete alamülesannete juhul on sobivaks alternatiiviks rekursiivsele meetodile nn. **dünaamilise kavandamise** meetod. Viimase all mõistetakse iteratiivset "alt-üles" algoritmi, milles igal sammul salvestatakse need konkreetsete alamülesannete lahendustulemused, mida järgmistel sammudel veel vaja võib minna. Fibonacci arvu leidmise juures tähendaks see arvutamist järjekorras $f_2, f_3, ..., f_n$, kusjuures kogu aeg peetakse meeles kahel eelmisel sammul leitud arvupaari.

Vaadeldaval juhul tekitab veel lisaraskusi see, et säilitatavateks vahetulemuseks on (osa)sõned. Viimaste meelespidamine on omaette probleem, näiteks ei ole otstarbekohane salvestada vastavaid indeksite järjendeid.

Algoritmis joonisel 5.8 on dünaamilisel kavandamisel abistruktuurideks kaks maatriksit c ja b. Maatriks $c_{ij} (i = 0, 1, \dots, m; j = 0, 1, \dots, n)$ on mõeldud vahetulemustena leitud sõnede pikkuste salvestamiseks. Selle elemendiks c_{ij} on $P\ddot{U}(s|_i, t|_j)$ pikkus. Pikima ühissõne leidmise üldise skeemi järgi

$c_{ij} = 0$, kui $i = 0$ või $j = 0$, st. kui üks argumentidest on tühisõne,

$c_{ij} = c_{i-1,j-1} + 1$, kui $i, j > 0$ ja $s[i] = t[j]$,

$c_{ij} = \max(c_{i-1,j}, c_{i,j-1})$, kui $i, j > 0$ ja $s[i] \neq t[j]$.

Maatriks $b_{ij} (i = 1, \dots, m; j = 1, \dots, n)$ on aga abivahendiks alamülesannetes leitavate pikimate ühissõnede meelespidamiseks. Osutub, et piisab sellest, kui iga $P\ddot{U}(s|_i, t|_j)$ korral on teada, millisel viisil (üldise skeemi mõttes) ta

on saadud:

$$b_{ij} = \begin{cases} 1, & \text{kui } P\ddot{U}(s|_i, t|_j) \text{ saadi } s[i] = t[j] \text{ lisamisel sõnele } P\ddot{U}(s|_{i-1}, t|_{j-1}), \\ 2, & \text{kui võeti } P\ddot{U}(s|_i, t|_j) = P\ddot{U}(s|_{i-1}, t|_j), \\ 3, & \text{kui võeti } P\ddot{U}(s|_i, t|_j) = P\ddot{U}(s|_i, t|_{j-1}) \end{cases}$$

```
kavandada()
 - - - Antud: m, n ja sõned s = s[1...m] ning t = t[1...n]
 - - - Tulemus: antud sõnede kõigi prefiksite paaride pikimaid ühissõnesid
 - - -            määravad maatriksid c ja b

 [⋆ (i = 1...m ) c_{i,0} := 0
 [⋆ (j = 1...n) c_{0,j} := 0

 ⋆ i = 1...m
 ⋆ j = 1...n

   s[i] = t[j] ?
     - - - kokkulangeva paari korral
     - - - pikim ühissõne pikeneb selle sümboli võrra
    c_{ij} := c_{i-1,j-1} + 1;
    b_{ij} := 1
 <- - - -
     - - - pikimaks ühissõneks valitakse eelmistest pikim
   c_{i-1,j} > c_{i,j-1} ?
    c_{ij} := c_{i-1,j};
    b_{ij} := 2
 <- - - -
    c_{ij} := c_{i,j-1};
    b_{ij} := 3
```

Joonis 5.8: Pikima ühissõne otsimise algoritmi põhiosa.

Kui antud sõnede s ja t korral arvutada maatriksid c ja b joonisel 5.8 toodud algoritmi kohaselt, siis saab nende põhjal hõlpsasti kindlaks teha soovi-

tud lahendi. Nimelt $P\ddot{U}(s,t)$ pikkuse leiame elemendist c_{mn}, sellese kuuluvad sümbolid (lõpust alates) aga maatriksi b läbivaatamisel lihtsa algoritmi järgi. Läbivaatust alustame elemendist b_{mn}, iga vaadeldava elemendi b_{ij} korral

kui $b_{ij} = 1$, siis $s[i] = t[j]$ on $P\ddot{U}(s,t)$ sümbol, liikuda edasi elemendile $b_{i-1,j-1}$;

kui $b_{ij} = 2$, siis liikuda edasi elemendile $b_{i-1,j}$;

kui $b_{ij} = 3$, siis liikuda edasi elemendile $b_{i,j-1}$.

Tabelis 5.2 on esitatud abimaatriksite c ja b elementide paarid pikima ühissõne $P\ddot{U}$('yywzxyx', 'xxyzywxy') = 'yywxy' leidmisel. Poolpaksus kirjas elemendid asuvad lahendi leidmise teel.

Tabel 5.2: Abimaatriksite $(c;b)$ elemendid pikima ühissõne leidmisel.

	j:	0	1	2	3	4	5	6	7	8
i			x	x	y	z	y	w	x	y
0		0;	0;	0;	0;	0;	0;	0;	0;	0;
1	y	0;	0;2	0;2	**1;1**	**1;3**	1;1	1;3	1;3	1;1
2	y	0;	0;2	0;2	1;2	1;2	**2;1**	2;3	2;3	2;1
3	w	0;	0;2	0;2	1;2	1;2	2;2	**3;1**	3;3	3;3
4	z	0;	0;2	0;2	1;2	2;1	2;2	**3;2**	3;2	3;2
5	x	0;	1;1	1;1	1;1	2;2	2;2	3;2	**4;1**	4;3
6	y	0;	1;2	1;2	2;1	2;2	3;1	3;2	4;2	**5;1**
7	x	0;	1;1	2;1	2;2	2;2	3;2	3;2	4;1	**5;2**

Kirjeldatud viisil dünaamilist kavandamist kasutava pikima ühissõne otsimise meetodi ajaline keerukus on $O(mn)$, mis tuleneb arvutatava(te) maatriksi(te) mõõtmetest. Vaadeldud algoritmi tegelikul realiseerimisel võib loobuda maatriksist b, sest tulemus on leitav ka üksnes maatriksi c põhjal.

Peatükk 6

Graafitöötlus

6.1 Graafi tippude topoloogiline sorteerimine

Orienteeritud graafi $G = (V, E)$ tippude **topoloogiliseks järjestuseks** nimetatakse sellist järjendit $S = (v_1, v_2, \ldots, v_n)$, et $V = \{v_1, v_2, \ldots, v_n\}$ ja iga $(v_i, v_j) \in E$ korral $i < j$ (st. iga tipu korral kõik eellased paiknevad järjendis temast eespool).

Topoloogiliselt saab järjestada vaid tsükliteta graafi tippe. Vastav üldine algoritm on esitatud joonisel 6.1. Igal tsükli sammul eemaldatakse graafist üks eellasteta tipp v (koos kõigi temast väljuvate kaartega) ja lisatakse moodustatava järjendi S lõppu. Selline sorteerimine vastab topoloogilise järjestuse nõuetele, sest tipul v kas polnudki graafis G eellasi või on need juba eemaldatud ning seega juba järjendisse kantud. Märgime, et tsükliteta orienteeritud graafis vähemalt üks tippudest on ilma eellasteta.

Topoloogilise järjestuse leidmiseks ei pruugi graafi tippe tegelikult eemaldadagi, piisab vaid iga tipu korral pidada tema (veel vaatlemata ja järjendisse lülitamata) eellaste loendurit (vt. joonis 6.2). Töö kiirendamiseks on mõistlik eraldi salvestada nende tippude hulk Q, millel loenduri väärtuseks on null ja mis pole veel järjendisse kantud.

Praktilist huvi pakuvad topoloogilise sorteerimise rakendused suurte projektide juhtimise valdkonnas. Nimelt osutub projekti realiseerimise kavandamisel otsarbekohaseks koostada projekti üksikute tööde **eeldusgraaf** ehk **võrkgraafik**. Eeldusgraafi tippudeks on projektis ettenähtud tööd (alam-

```
graafi_topoloogiline_sorteerimine(G, S)
 - - - Antud: orienteeritud, tsükliteta graaf G,
 - - -          järjend S tulemuse salvestamiseks
 - - - Tulemus: antud graafi tipud järjendis S topoloogilises järjestuses;
 - - -            antud graaf ei säili

S := ∅
   - - - lähtegraafi iga kaare (v', v'') korral: kui v'' ∈ S, siis v' ∈ S ja
   - - - tipp v' eelneb tipule v'' järjendis S
 G pole tühi ?
  v := üks G tippudest, millesse ei sisene kaari
  eemaldada graafist G tipp v koos kõigi temast väljuvate kaartega
  S ⇐ v
```

Joonis 6.1: Graafi tippude topoloogilise sorteerimise põhimõte.

projektid), kaarega (v, w) aga märgitakse asjaolu, et tööga w ei saa alustada enne, kui töö v on lõpetatud, st. töö v lõpetatus on töö w (üheks) eeldustingimuseks (ehk prerekvisiidiks). Taolise graafi tippude topoloogiline järjestus näitab, millises tööde järjekorras saaks vastavat projekti realiseerida, kui kõik tööd teha eraldi, ajaliselt üksteise järel. Veelgi olulisem on eeldusgraafi analüüs juhul, kui üksikuid töid soovitakse teha samaaegselt, muidugi eeldustingimustega lubatud piires. Ka selle ülesande lahendamisel on aluseks eeldusgraafi tippude topoloogiline sorteerimine.

Olgu eeldusgraafi $G = (V, E)$ iga tipuga v seotud veel vastava töö tegemiseks kuluv aeg $v.t$. Sel korral saab iga tipu v jaoks arvutada vastava töö varaseima lõputähtaja $v.varaseim$. Kogu projekti (varaseim) lõputähtaeg $valmis$ on siis nendest tööde lõputähtaegadest suurim:

$$valmis = \max_{v \in V}(v.varaseim).$$

Tavaliselt aga pole iga tööd ilmtingimata tarvis alustada nii varakult (ajal $v.varaseim - v.t$), et oleks tagatud tema võimalikult varajane valmimine; nii mõnegi töö algusega oleks võimalik viivitada, ilma et projekti lõputähtag ($valmis$) kannataks. Järelikult pakub huvi leida iga eeldusgraafi tipu v

jaoks veel suurus $v.hiliseim$, mis näitab vastava töö hiliseimat võimalikku lõputähtaega, mille puhul kogu projekti lõputähtajaks jääb ikka $valmis$.

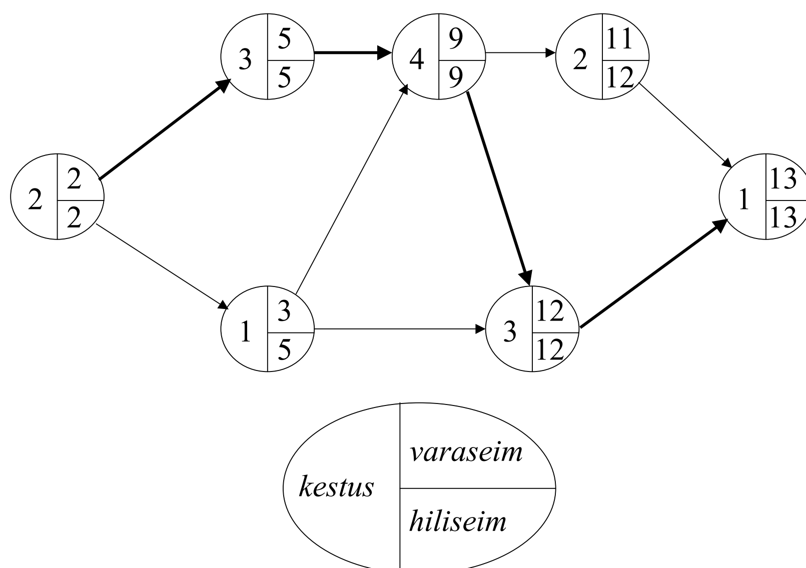
```
topoloogiline_sorteerimine(G, S)
 - - - Antud: orienteeritud, tsükliteta graaf G = (V, E), igal tipul on mää-
 - - -          ratud .sisendaste; järjend S tulemuse salvestamiseks
 - - - Tulemus: graafi G tipud järjendis S topoloogilises järjestuses

 - - - Abivahendid: tippude hulk Q,
 - - -              igal tipul v tööväli v.x – "eemaldamata" eellaste arv
 Q := ∅
 - - - tipud, millesse ei sisene kaari, kanda hulka Q:
 ⋆ ∀v, v ∈ V
    [? (v.sisendaste = 0)) Q ⇐ v
    v.x := v.sisendaste    - - - ühtlasi anda tööväljale algväärtus
 S := ∅
    - - - graafi G iga kaare (v′, v″) korral: kui v″ ∈ S, siis v′ ∈ S
    - - - Q koosneb tippudest, millel ei ole (veel "eemaldamata") eellasi
  Q pole tühi ?
   v ⇐ Q
   S ⇐ v
   ⋆ ∀w, w ∈ Gv    - - - tipu v iga järglase w korral
       - - - vähendada tipu w töövälja väärtust
     (− − w.x) = 0  ?
      Q ⇐ w   - - - kui see sai nulliks, siis lisada w hulka Q
```

Joonis 6.2: Graafi tippude topoloogilise sorteerimise algoritm.

Kuid alati leidub ka nn. **kriitilisi töid**, millel selline „lõtk“ on lubamatu: kui

kõiki töid saaks alustada viivitusega, siis poleks ju kogu projekti lõputähtaeg varaseim (seda saaks vähendada). Kriitilisele tööle vastavas tipus v kehtib võrdus $v.hiliseim = v.varaseim$, sellise omadusega tipud tipud paiknevad eeldusgraafis (ühel või mitmel) nn. **kriitilisel teel**, mis algab eellasteta tipust ja lõpeb (järglasteta) tipus w , kus $w.hiliseim = w.varaseim = valmis$.

Joonisel 6.4 esitatud algoritmis leitakse algul antud eeldusgraafi tippude topoloogiliselt sorteeritud järjend $s_1, s_2, \ldots, s_n$. Viimast päripidi läbi vaadates arvutatakse tööde varaseimad lõputähtajad (ühtlasi ka nende maksimum $valmis$) ning tagurpidi liikudes leitakse tööde hiliseimad lõputähtajad. Selliselt analüüsitud eeldusgraafis (vt. ka joonis 6.3) on juba lihtne leida kriitilisi töid ja kriitilisi teid.



Joonis 6.3: Analüüsitud eeldusgraaf.

Nii graafi tippude topoloogiline sorteerimine kui ka eeldusgraafi vastav analüüsimine on ajalise keerukusga $O(n + m)$, kus n on tippude arv ja m kaarte arv graafis.

```
eeldusgraafi_analüüs(G)
 --- Antud: n-tipuline eeldusgraaf G = (V, E), milles
 ---        iga tipu v korral on määratud suurus v.t (töö kestus),
 ---        iga tipuga v on seotud väljad v.varaseim ja v.hiliseim
 --- Tulemus: arvutatud v.varaseim ja v.hiliseim iga v ∈ V jaoks,
 ---          tagastatakse kogu projekti lõpetamistähtaeg
 ---          kui tipp v ei ole algustipust saavutatav või v = a,
 ---          siis v.d = +∞ ja v.eellane jääb määramata

 --- Abivahendid: funktsioon topoloogiline_sorteerimine,
 ---          muutujad m ja valmis, järjend (s1, s2, ... , sn) – koht
 ---          graafi tippude salvestamiseks topoloogilises järjestuses
 topoloogiline_sorteerimine(G, (s1, s2, ... , sn)); valmis := 0

 ⋆ v = s1, s2, ... , sn   --- päripidi
   leida varaseimad lõpetamisajad:
   m := 0
   ⋆ ∀w, w ∈ G⁻¹v   --- tipu v iga eellase w korral
      [? (w.varaseim > m) m := w.varaseim
   --- m on tähtaeg, millal lõpetatakse töö v viimane eeldustöö
   [? ((v.varaseim := v.t + m) > valmis) valmis := v.varaseim

 ⋆ v = sn, sn−1, ... , s1   --- tagurpidi
   leida hiliseimad lõpetamisajad:
   m := valmis
   ⋆ ∀w, w ∈ Gv   --- tipu v iga järglase w korral
      [? (w.hiliseim − w.t < m) m := w.hiliseim − w.t
   --- m on hiliseim tähtaeg, millal lõpetada töö v,
   --- et talle järgnevad tööd ei hilineks
   v.hiliseim := m
 ←— (valmis)
```

Joonis 6.4: Eeldusgraafi analüüsimise algoritm.

6.2 Lühimad teed graafis

Olgu graafi $G = (V, E)$ iga kaarega (v, w) seotud reaalarvuline **kaarepikkus** $c(v, w) \in \mathbb{R}^+$. Graafis leiduva tee $(v_1, v_2, \ldots, v_k)$ **pikkuseks** nimetatakse summat $c(v_1, v_2) + c(v_2, v_3) + \ldots + c(v_{k-1}, v_k)$. Tippude a ja b vaheliseks **kauguseks** $d(a, b)$ nimetatakse nendevahelise lühima tee $(a \ldots b)$ pikkust; seejuures tipu kauguseks iseendast loetakse 0, seostamata tippude vaheliseks kauguseks aga $+\infty$. Seega

$$d(a, a) = 0$$
$$d(a, b) = \min_{v \in G^{-1}b} (d(a, v) + c(v, b))$$

Kuna kahe tipu a ja b vahelise kauguse leidmiseks tuleb paratamatult vaadelda kõiki nendevahelisi teid ja arvestada nende teede pikkusi, siis osutub otstarbekohaseks koostada algoritmid üldisema ülesande lahendamiseks: iga tipu $v \in V$ korral leida selle kaugus antud **lähtetipust** a ning määrata vastav lühim tee.

Topoloogilise sorteerimisega üsna sarnasel viisil saab leida tsükliteta graafi kõigi tippude kaugused mingist etteantud lähtetipust ning ühtlasi ka määrata vastavad lühimad teed. Sellekohase algoritmi (vt. joonis 6.5) ajaline keerukus on samuti $O(|V| + |E|)$.

Algoritmi töö käigus hoitakse tipu väljal $.d$ selle tipu senini leitud („jooksvat“) kaugust lähtetipust. Leitavate teede „meelespidamiseks“ kasutatakse välju $.eellane$: väljal $v.eellane$ paikneb viit tipule, mis eelneb tipule v senini leitud teel $a \ldots v$.

Üldjuhul on aga kauguste leidmine mõnevõrra keerukam. **Dijkstra algoritmis** (vt. joonis 6.6) rakendatakse abivahendina eelistusjärjekorda Q selleks, et meeles pidada tippe, mille kõiki eellasi pole (võibolla) veel arvesse võetud. Algoritmi idee (vt. ka joonis 6.7) põhineb asjaolul, et vähima „jooksva“ kaugusega $(v.d)$ tipul $v \in Q$ on kõik eellased juba arvestatud ning järelikult $v.d$ enam ei muutu; sellise tipu v võib hulgast Q eemaldada, lisades hulka Q tema need järglased, mis veel vaatlusele (ja hulka Q) pole võetud. Seejuures korrigeeritakse tipu v järglaste w „jooksvat“ kaugust tipust a, kui osutub, et tippu w pääseb läbi v veel lühemat teed $(a \ldots v, w)$ pidi, kui seda on senini teadaolev lühim tee $a \ldots w$. Paraku suurendab selline korrigeerimi-

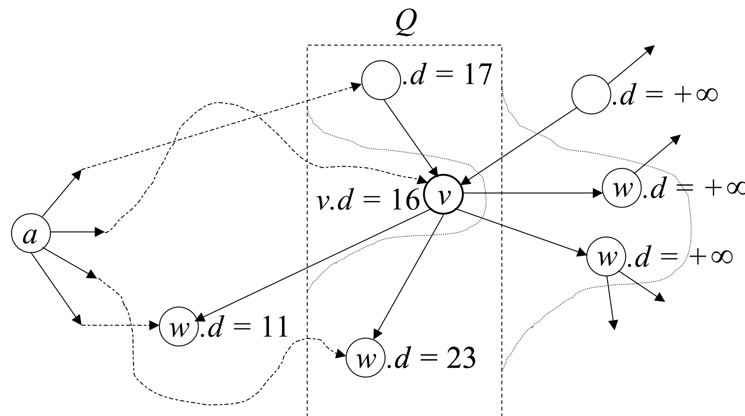
```
kaugused_tsükliteta_graafis(G, a)
 --- Antud: orienteeritud, tsükliteta graaf G = (V, E),
 ---        lähtetipp a, millest arvestada teiste tippude kaugusi;
 ---        igal kaarel (v, w) ∈ E on määratud suurus c(v, w) ⩾ 0 – selle
 ---        kaare pikkus,
 ---        iga tipuga v ∈ V on seotud väljad v.d ja v.eellane tulemuste
 ---        salvestamiseks
 --- Tulemus: iga tipu v ∈ V korral v.d = lühim teepikkus a ja v vahel;
 ---          kui tipp v ei ole algustipust saavutatav, siis v.d = +∞
 ---          ja v.eellane on määramata; ka a.eellane on määramata

 --- Abivahendid: tippude hulk Q,
 ---              igal tipul v töövali v.x – veel vaatlemata eellaste arv
a.d := 0; --- tipu a kaugus iseendast
Q := {a}
 --- algväärtused kaugustele ja tööväljadele:
[⋆ (∀v, v ∈ V\{a}) v.d := +∞; v.x := v.sisendaste

    --- hulk Q koosneb tippudest, millesse suubuvad kaared on kõik
    --- juba arvesse võetud;
    --- iga tipu v korral on v.d senini leitud lühima tee a...v pikkus
    --- (või +∞) ja v.x on sellesse tippu sisenevate veel vaatlemata
    --- kaarte arv
  Q pole tühi ?
   v ⇐ Q
   ⋆ ∀w, w ∈ Gv   --- iga tipust v väljuva kaare (v, w) korral
      --- võimalusel parandada w.d ja vastavalt ka w.eellane:
     [? (v.d + c(v, w) < w.d) w.d := v.d + c(v, w); w.eellane := v
      --- järjekordne tippu w sisenev kaar on vaadeldud:
     [? (−− w.x) = 0) Q ⇐ w
```

Joonis 6.5: Tsükliteta graafis kauguste leidmise algoritm.

```
kaugused_graafis(G, a)
 --- Antud: orienteeritud graaf G = (V, E) ja lähtetipp a ∈ V,
 ---          millest arvestada teiste tippude kaugusi;
 ---          igal kaarel (v, w) ∈ E on pikkus c(v, w) ⩾ 0, iga tipuga v
 ---          on seotud väljad v.d ja v.eellane tulemuste salvestamiseks
 --- Tulemus: iga tipu v ∈ V korral v.d = lühima tee a ... v pikkus ja
 ---            v.eellane on v-le eelnev tipp sellel teel;
 ---            kui tipp v ei ole algustipust saavutatav, siis v.d = +∞
 ---            ja v.eellane on määramata; ka a.eellane on määramata

 --- Abivahendid: eelistusjärjekord Q, kus tipu v ∈ Q võtmeks on v.d
Q := {a};   a.d := 0;  --- tipu a kaugus iseendast
[⋆ (∀v, v ∈ V\{a}) v.d := +∞;  --- algväärtus

    --- iga tipu v korral on v.d senini leitud lühima tee a ... v pikkus
    --- (või siis v.d = +∞);
    --- Q koosneb juba vaatlusele võetud tippudest, kus tee a ... w
    --- (w ∈ Q) pikkus w.d on küll leitud, kuid võib veel väheneda
    --- (kui leitakse lühem tee a ... w)
  Q pole tühi ?
   v ⇐ Q   --- võtta vähima võtmega element

   ⋆ ∀w, w ∈ Gv   --- iga tipust v väljuva kaare (v, w) korral

     uus_kaugus := v.d + c(v, w)   --- tee a ... v, w pikkus
    w.d = +∞  ?
      --- kaugus on veel määramata, seega w on veel vaatlemata
     w.eellane := v;  w.d := uus_kaugus;  Q ⇐ w
   <----
      --- w on juba vaatlusele võetud,
      --- võimalusel parandada w.d ja vastavalt ka w.eellane:
     [? (uus_kaugus < w.d) w.eellane := v;  w.d := uus_kaugus
      --- muudeti eelistusi Q-s vastavalt uuele w.d väärtusele
```

Joonis 6.6: Dijkstra algoritm lühimate teede leidmiseks.

ne algoritmi keerukust, sest tegemist on eelistusjärjekorra võtmete (seega ka eelistuste) muutmisega ja vastavalt Q ümberkorraldamisega. Dijkstra algoritmi ajalise keerukse hinnang sõltub oluliselt sellest, kuidas on korraldatud eelistusjärjekorra "pidamine". Kahendkuhja kasutamise korral on hinnanguks $O((n+m)\log n)$, kus $n = |V|$ ja $m = |E|$.



Joonis 6.7: Dijkstra algoritmi tööpõhimõte (hulgast Q eemaldatakse tipp v, selle „uued“ naabrid lisatakse hulka Q, kaugus $w.d = 23$ võib väheneda).

Paneme tähele, et vaadeldud kahes kauguste (ja lühimate teede) leidmise algoritmis joonistel 6.5 ja 6.6 on keskseks tegevuseks jooksva tipu (v) väljade $v.d$ ja $v.eellane$ nö. parandamine:

- - - *võimalusel parandada $w.d$ ja vastavalt ka $w.eellane$:*

[? $(v.d + c(v, w) < w.d)$ $w.d := v.d + c(v, w)$; $w.eellane := v$

Osutub, et selliste korrigeerivate tegevuste järjekord on oluline küll algoritmi efektiivsuse seisukohalt, kuid mitte lahenduse saavutamisel. Lahenduse saame ka siis, kui parandamisi teha lihtsalt piisaval arvul. Vastava algoritmi (vt. joonis 6.8) ajalise keerukuse hinnanguks on $O(|V| \times |E|)$, seega nt. $O(|V|^2)$ juhul, kui $|V| = |E|$. See algoritm on küll Dijkstra algoritmist aeglasem, kuid suureks eeliseks on rakendatavus ka juhul, kui graafis võib esineda negatiivseid kaarepikkusi.

Olgu $G = (V, E)$ graaf, milles tipp $v \in V$ on saavutatav tipust $a \in V$. Kui graafis G ei ole negatiivse pikkusega kaari, siis lühimaks teeks nende

tippude vahel saab olla vaid elementaartee. Piltlikult rääkides, on ju ilmne, et liikumisel ühest tipust teise igasugune vahepealne tsüklis „keerutamine“ ei saa kuidagi „jalavaeva“ (tee summaarset pikkust) vähendada. Kuid olukord on sootuks teine, kui graafis G leidub negatiivse pikkusega kaari. Siis võib osutuda võimalikuks, et tipust a saab liikuda tipp v nii, et läbitakse tsükkel, mille kaarte pikkuste summa on negatiivne. Taolisel juhul on tegemist graafiga, milles ei eksisteerigi tee $a \dots v$ pikkuse miinumumi: iga leitud tee $a \dots v$ korral saab konstrueerida veel lühema tee (milles nimetatud tsüklit läbitakse piisav arv kordi). Näiteks graafis joonisel 6.9(a) ei leidu lühimat teed tipust a tippu e, kuigi tipp e on saavutatav tipust a. Seevastu graafis 6.9(b) leiduvad lühimad teed tipust a kõikidesse teistesse tippudesse.

Bellman-Fordi algoritm joonisel 6.10 põhinebki parandamise protseduuril. Pärast viimase rakendamist on kas leitud kõik lühimad teed või tuvastatud, et mitte kõik lühimad teed ei ole leitavad. Viimasest asjaolust annab tunnistust see, et vähemalt ühte jooksvat kaugust ($.d$) saaks ikka veel parandada.

Graafi $G = (\{v_1, v_2, \dots, v_n\}, E)$ tippude **naabrusmaatriksiks** nimetatakse $n \times n$ maatriksit (a_{ij}, $i = 1, 2, \dots, n$, $j = 1, 2, \dots, n$), kus

$$a_{ij} = \begin{cases} c(v_i, v_j), & \text{kui } (v_i, v_j) \in E \\ +\infty, & \text{vastasel korral} \end{cases}$$

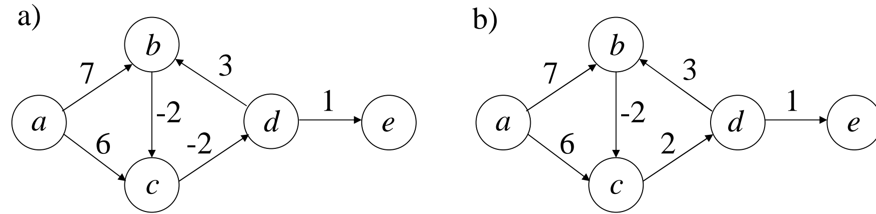
Leidub väga lihtsa struktuuriga algoritm kauguste leidmiseks kõigi tipupaaride korral (ilma lühimaid teid tuvastamata), mis seisneb antud graafi naabrusmaatriksi teisendamises nn. **kauguste maatriksiks**, milles

$$a_{ij} = \begin{cases} d(v_i, v_j), & \text{kui leidub tee } (v_i \dots v_j) \\ +\infty, & \text{vastasel korral.} \end{cases}$$

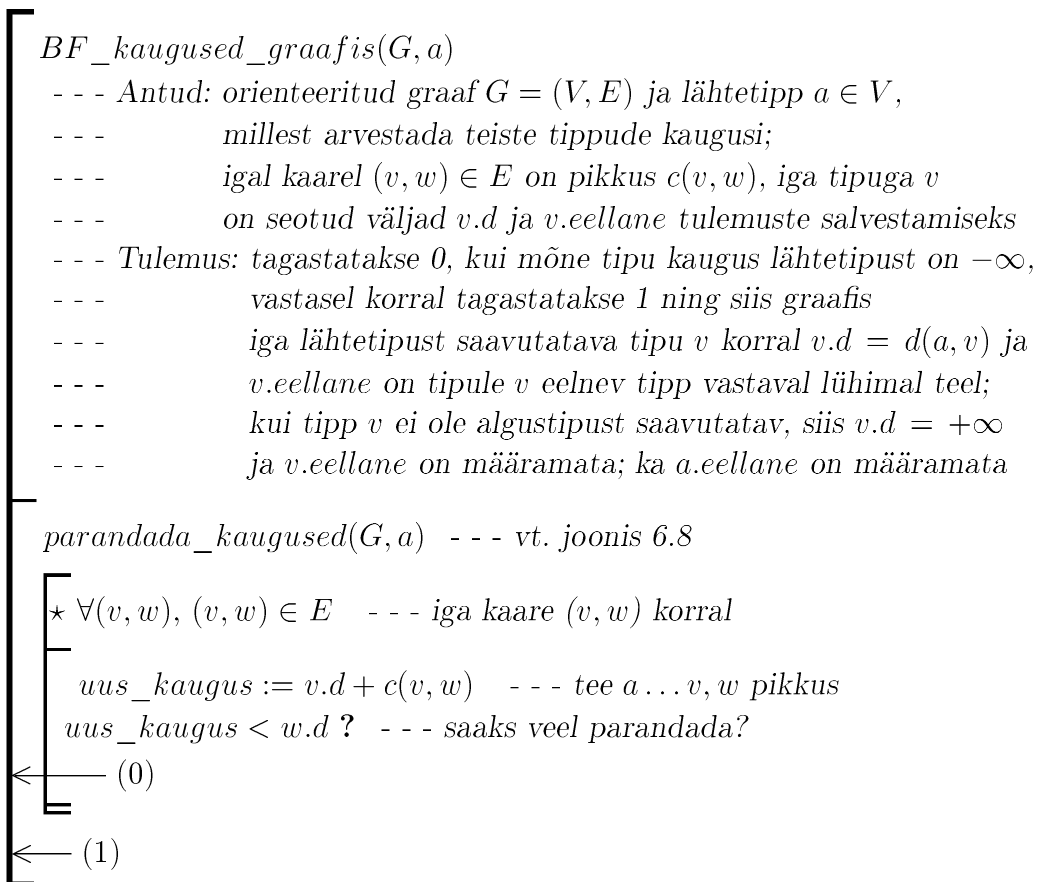
Floyd-Warshalli algoritmis (vt. joonis 6.11) algab välimise tsükli iga samm olukorras, kus maatriksi element a_{ij} tähendab tippude v_i ja v_j vahemaad mööda lühimat teed, mis läb ainult tippe $v_1, v_2, \dots, v_{t-1}$ (arvestamata siin tee otstippe v_i ja v_j). Sisemises kahekordses tsüklis korrigeeritakse maatriksi elemente, arvestades ka teid läbi tipu v_t. Taolisel viisil tippude kauguste arvutamise algoritmi ajaline keerukus on $O(n^3)$.

```
parandada_kaugused(G, a)
 --- Antud: orienteeritud graaf G = (V, E) ja lähtetipp a ∈ V,
 ---          millest arvestada teiste tippude kaugusi;
 ---          igal kaarel (v, w) ∈ E on pikkus c(v, w), iga tipuga v
 ---          on seotud väljad v.d ja v.eellane tulemuste salvestamiseks
 --- Tulemus: iga tipu v ∈ V korral, kui eksisteerib lühim tee a...v,
 ---            siis v.d = lühima tee a...v pikkus ja
 ---            v.eellane on v-le eelnev tipp sellel teel;
 ---            kui tipp v ei ole algustipust saavutatav, siis v.d = +∞
 ---            ja v.eellane on määramata; ka a.eellane on määramata

 a.d := 0;  --- tipu a kaugus iseendast
 [* ∀v, v ∈ V\{a}
    v.d := +∞;   --- algväärtus
 [* |V| − 1
    [* ∀(v, w), (v, w) ∈ E    --- iga kaare (v, w) korral
       uus_kaugus := v.d + c(v, w)    --- tee a...v, w pikkus
        --- võimalusel parandada w.d ja vastavalt ka w.eellane:
       [? (uus_kaugus < w.d) w.eellane := v; w.d := uus_kaugus
```

Joonis 6.8: Aeglasem algoritm lühimate teede leidmiseks.



Joonis 6.9: Kaks suvaliste kaarepikkustega graafi.



Joonis 6.10: Bellman-Fordi algoritm lühimate teede leidmiseks.

```
kauguste_maatriks(A, n)
 --- Antud: n-tipulise graafi tippude naabrusmaatriks
 ---          A = (a_ij, i = 1, 2, ..., n, j = 1, 2, ..., n)
 --- Tulemus: A on teisendatud tippudevaheliste kauguste maatriksiks

  * t = 1, 2, ..., n

     --- maatriksi A elemendiks a_ij on i-nda tipu ja j-nda tipu vaheline
     --- kaugus mööda teid, mis läbivad vähemalt ühte tippudest
     --- nr. 1, 2, ..., t − 1

     * i = 1, 2, ..., n
     * j = 1, 2, ..., n

        --- korrigeerida a_ij, kui tee i-ndast tipust j-ndasse tippu läbi
        --- t-nda tipu on lühem senini leitud lühimast ühendusteest nende
        --- tippude vahel
        [? (a_it + a_tj < a_ij) a_ij := a_it + a_tj
```

Joonis 6.11: Floyd-Warshalli algoritm graafi tippude kaugustemaatriksi arvutamiseks.

6.3 Graafi läbimine

Paljudes graafitöötlemise ülesannetes tuleb parajasti iga tipu korral sooritada teatav protseduur, liikudes mööda kaari tipust tippu. Niisugust graafi tippude süstemaatilist läbivaatamist nimetataksegi graafi läbimiseks.

Graafi läbimise meetodid jagunevad üldises plaanis kaheks mooduseks – laiuti läbimine ja sügavuti läbimine. Näiteks eespool vaadeldud lühima tee otsimise algoritmides rakendatakse just laiuti läbimist: põhitsükli igal sammul vaadeldakse juba vaatlusele võetud tippude hulga kõiki "naabreid". Joonisel 6.12 on esitatud veel üks seda liiki algoritm. Viimases lahendatakse nn. **otsetee** leidmise ülesanne. Lähtudes antud tipust a tehakse iga ülejäänud tipu v korral kindlaks selline tee $a \ldots v$, milles kaarte arv on minimaalne. Kuigi taolise ülesande saaks ka lahendada lühima tee otsimise erijuhuna, kus kõigi kaarte pikkused on võrdsed, osutub vahetu lahendusmeetod siiski märgatavamalt lihtsamaks.

Graafi sügavuti läbimise rekursiivne algoritm l_s leidub joonisel 6.13. Erinevalt laiuti läbimisest minnakse siin järjekordsest vaadeldavast tipust v kohe edasi ühele tema naabritest, asudes ka selle korral lahendama sama ülesannet. Kuna parajasti vaadeldavale tipule rakendatakse töötlemise funktsiooni f enne naabri juurde asumist, siis on tegemist töötlemisega eesjärjestuses. Algoritmi l_s ühe kasutusvõimalusega tutvume järgmises jaotises.

```
läbida_laiuti(G, a, f)
 - - - Antud: orienteeritud graaf G = (V, E),
 - - -        lähtetipp a, millest alustada G tippude läbimist,
 - - -        funktsioon f vaadeldava tipu töötlemiseks:
 - - -        iga tipuga v ∈ V on seotud väljad v.vaadeldud ja v.eellane
 - - -        lisatulemuste salvestamiseks
 - - - Tulemus: iga tipu v ∈ V, v ≠ a korral,
 - - -          kui v on saavutatav tipust a, siis
 - - -                on arvutatud f(v) ning lisaks
 - - -                v.vaadeldud = 1 ja
 - - -                v.eellane on v-le eelnev tipp otseseimal teel a...v,
 - - -          vastasel korral
 - - -                v.vaadeldud = 0 ja v.eellane on määramata;
 - - -          a.eellane pole määratud
 - - - Abistruktuur: tippude lihtjärjekord Q
 - - - algväärtused:
[* (∀v, v ∈ V) v.vaadeldud := 0
a.vaadeldud := 1
Q := {a}
   - - - Q koosneb juba vaatlusele võetud tippudest, mille järglasi
   - - - tuleb veel vaadelda
 Q pole tühi ?
  v ⇐ Q; f(v);  - - - võtta esimene ja töödelda
  * ∀w, w ∈ Gv    - - - tipu v iga järglase w korral
   w.vaadeldud = 0 ?
    w.vaadeldud := 1;  w.eellane := v
    Q ⇐ w;  - - - panna viimaseks
```

Joonis 6.12: Graafi laiuti läbimine otseteede leidmisel.

```
l_s(G, a, f)
 --- Antud: orienteeritud graaf G = (V, E),
 ---        lähtetipp a, millest alustada G tippude läbimist,
 ---        funktsioon f tippude töötlemiseks;
 ---        iga tipuga v ∈ V on seotud suurus v.vaadeldud ∈ {0; 1} ja
 ---        väli v.eellane
 --- Tulemus: arvutatud f(a), a.vaadeldud = 1;
 ---          lisaks iga tipu v ≠ a korral, mis on saavutatav tipust a
 ---          läbides vaid tippe, millel .vaadeldud = 0;
 ---          kui lähtegraafis oli v.vaadeldud = 0, siis
 ---          arvutatud f(v) ning v.vaadeldud = 1 ja
 ---          v.eellane on v-le eelnev tipp mingil teel a...v

 --- Abivahendid: funktsioon l_s (rekursiivselt)
a.vaadeldud := 1;  f(a);  --- töödeldakse antud tipp
  ⋆ ∀v, v ∈ Ga  --- tipu a iga (vaatlemata) naabri v korral
     [? (v.vaadeldud = 0) v.eellane := a;  l_s(G, v, f)
```

Joonis 6.13: Graafi sügavuti läbimine tippude töötlemisega eesjärjestuses.

6.4 Minimaalne toes

Sidusa orienteerimata graafi $G = (V, E)$ **toeseks** ehk **toesepuuks** nimetatakse sidusat atsüklilist graafi (V, T), kus $T \subseteq E$. Graafi sidusate komponentide toesepuude hulka nimetatakse graafi **toesemetsaks**. Nagu näidatud joonisel 6.14, saab toesemetsa leida graafi sidusate komponentide sügavuti läbimisel.

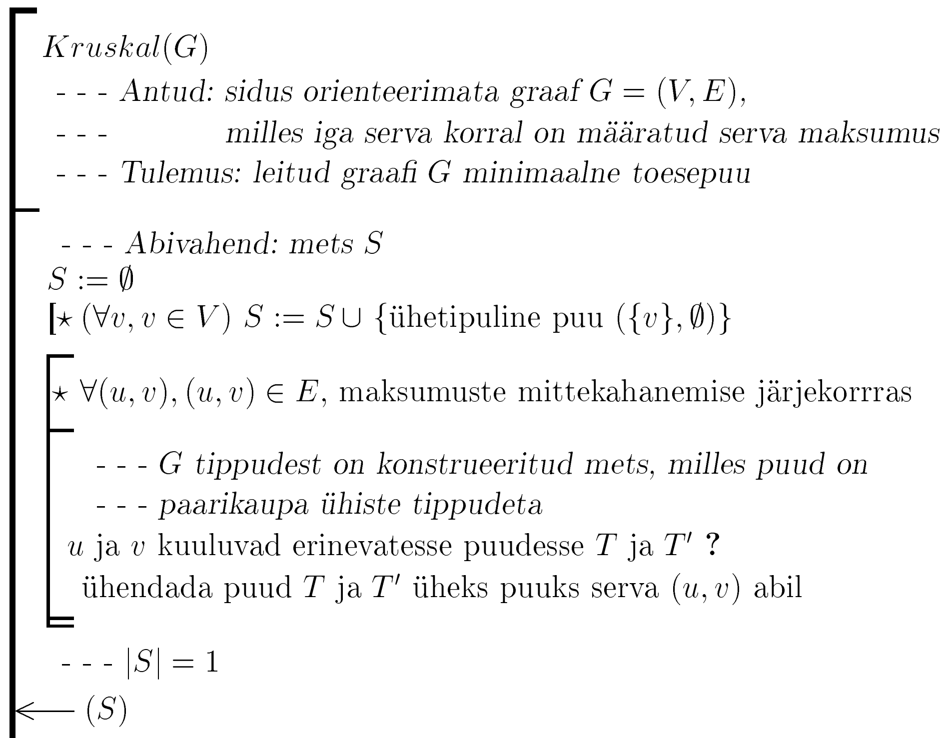
Kui graafi iga servaga on veel seotud serva (reaalarvuline) maksumus $.c \geqslant 0$, siis minimaalseks toeseks nimetatakse sellist toesepuud, millesse kuuluvate servade kogumaksumus $\sum_{e \in T} e.c$ on kõige väiksem. Vaatleme järgnevas kahte erinevat meetodit minimaalse toese leidmiseks.

```
toesemets(G)
 - - - Antud: graaf G = (V, E),
 - - - iga tipuga v ∈ V on seotud väljad v.eellane ja v.vaadeldud
 - - - Tulemus: graafi G toesemets määratud välja eellane kaudu:
 - - - kui v ∈ V on toesemetsa puu juur, siis v.eellane = Λ (on tühi),
 - - - vastasel korral v.eellane on v ülemus vastavas toesemetsa puus

 - - - Abivahendid: funktsioon l_s sügavuti läbimiseks (vt. joonis 6.13)
 - - - iga tipuga v ∈ V on seotud töoväli v.vaadeldud;
 - - - määrata algväärtused:
 [⋆ (∀v, v ∈ V) v.eellane := Λ;  v.vaadeldud := 0;
 [⋆ ∀v, v ∈ V
    [? (v.vaadeldud = 0) l_s(G, v, Λ)
```

Joonis 6.14: Graafi toesemetsa konstrueerimise algoritm.

Kruskali algoritmi ideed selgitab joonis 6.15. Otsitav puu konstrueeritakse järk-järgult puukujulistest osadest, kusjuures kõik tipud on kogu aeg "mängus" – leitud osapuude tippude hulgad on ühisosata, kuid nende ühendiks on kogu tippude hulk. Tulemusena saadakse minimaalsesse toesepuusse

kuuluvate servade hulk. Tegemist on ahne algoritmiga, igal sammul lisatakse servade hulka vähima maksumusega serv, mida saab lisada. Lisada aga saab neid servi, mis ei moodusta tsüklit mõnes juba leitud puus, st. lisamine tuleb arvesse ainult niisuguse serva korral, mille otstipud kuuluvad erinevatesse osapuudesse. Serva lisamisel moodustub vastavast kahest osapuust üks suurem osapuu.



Joonis 6.15: Kruskali algoritmi sisuline skeem.

Tegelikult Kruskali meetodi kohaselt osapuid ei konstrueeritagi, piisab vaadelda vaid nende tippude hulki kui klasse (vt. joonis 6.16). Selle tõttu praktilisel realiseerimisel lähtutakse Galler-Fisheri klasside kujutamise meetodist, mis tagab ajalise keerukuse $O(m \log n)$ n-tipulise ja m-servalise graafi jaoks.

Primi algoritm (vt. joonis 6.17) on tüüpiline graafi laiuti läbimisel põhinev graafitöötlemise moodus. Lähtudes etteantud tipust laiendatakse konstrueeritavat puud, üldmaksumust võimalikult väiksena hoides. Lisatava tipu

```
Kruskal_toes(G, A)
 - - - Antud: sidus orienteerimata graaf G = (V, E),
 - - -         milles iga serva korral on määratud serva maksumus;
 - - -         A on dünaamiline hulk tulemuse salvestamiseks
 - - - Tulemus: hulk A on graafi G minimaalse toesepuu servade hulk

 - - - abivahend: klasside komplekt S
S := ∅
[⋆ (∀v, v ∈ V) S := S ∪ {{v}}
A := ∅
sorteerida hulk E maksumuste järgi mittekahanevaks järjendiks
e_1, e_2, ... , e_m

 ⋆ (u, v) = e_1, e_2, ... , e_m
   - - - A ⊆ E on senini konstrueeritud puude servade hulk
   - - - igas komplekti S klassis on ühe puu tipud
   ühendada_klassid(u_0 := leida_esindaja(u), v_0 := leida_esindaja(v))
   [? (u_0 ≠ v_0) A := A ∪ {(u, v)}
```

Joonis 6.16: Kruskali algoritmi realiseerimine tipuklasse kasutades.

valimine toimub ka siin ahnel moel. Kui tippude eelistusjärjekord realiseerida kahendkuhjana, siis Primi algoritmi ajaline keerukerus on $O(n \log n)$.

Mõlemad algoritmid on rakendatavad ka mittesidusa graafi korral. Kruskali meetodi järgi leitakse siis iga sidusa komponendi minimaalne toes, Primi algoritmis aga selle komponendi toes, kuhu kuulub etteantud tipp.

```
Prim_toes(G,a)
 - - - Antud: orienteerimata graaf G = (V,E) ja
 - - -        lähtetipp a – tulevase toesepuu juur;
 - - -        igal serval on määratud maksumus .c ⩾ 0,
 - - -        igal tipul v ∈ V on väli v.eellane tulemuse salvestamiseks
 - - - Tulemus: antud graafi toes määratud väljadega .eellane

 - - - Abivahendid: tippude eelistusjärjekord Q, kus
 - - -              tipu v ∈ Q võtmeks on (v.eellane,v).c;
 - - -              lisaväljad tippudes – .läbitud, .puus, .c
[⋆ (∀v, v ∈ V\{a}) v.läbitud := 0
a.läbitud := 1;  a.puus := 0;  a.c := 0
Q := {a}

   - - - hulk Q koosneb juba vaatlusele võetud tippudest, kusjuures
   - - - tipu v ∈ Q korral v.c = (v.eellane,v).c
 Q pole tühi ?
  v ⇐ Q   - - - võtta vähima võtmega v.c element
  v.puus := 1

  ⋆ ∀w, w ∈ Gv   - - - tipu v iga naabri w korral

   w.läbitud = 0 ?
    w.läbitud := 1;  w.puus := 0;  w.eellane := v;
    w.c := (v,w).c
    Q ⇐ w
  <- - - -
   w.puus ?
  <- - - -
   (v,w).c < w.c ?
    w.eellane := v;  w.c := (v,w).c;  - - - muutub eelistus Q-s
```

Joonis 6.17: Primi algoritm minimaalse toese leidmiseks.

Peatükk 7

Algoritme planimeetriast

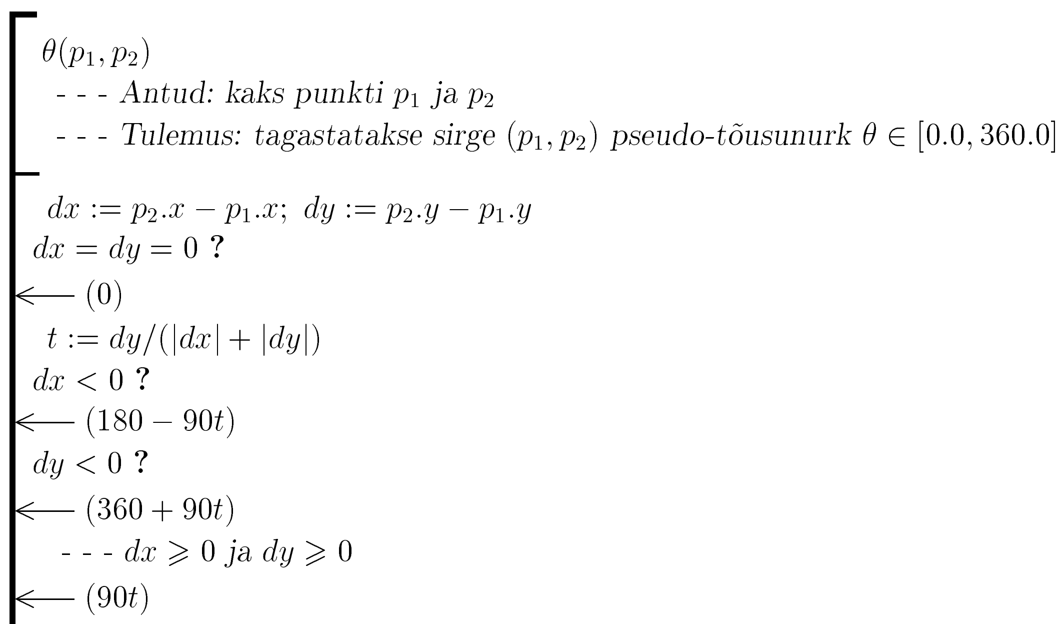
7.1 Lihtsamaid erivõtteid

Vaatleme esiteks mõningaid erivõtteid, mida planimeetria ülesannete algoritmide koostamisel on kasulik teada.

Sirgete (lõikude) tõusunurkade võrdlemise saab enamasti asendada nn. **pseudo-tõusunurkade** võrdlemisega. Viimaste leidmine osutub märksa lihtsamaks, kui seda on sirge tõusu arkustangensi arvutamine. Võrdlemisel saab piirduda pseudo-tõusunurkadega seetõttu, et nende järjestus ühtib vastavate tõusunurkade järjestusega. Pseudo-tõusunurga arvutamise eeskiri on esitatud joonisel 7.1.

Planimeetria algoritmides eeldame, et punkt on antud kirjena, milles leiduvad kindlasti abstsissiväli $.x$ ja ordinaadiväli $.y$.

Teiseks kasulikuks abivõtteks on **pöörde suuna** leidmine punktis p, kui liigutakse mööda lõike ühise otspunktiga p: $p_1 \to p \to p_2$. Funktsioonis vp joonisel 7.2 kontrollitakse üldiselt, kas toimub pööre vastupäeva (siis tagastatakse 1) või päripäeva (tagastatakse -1). Lisaks sellele aga eristatakse veel punktide vastastikuseid asendeid juhul, kui punktid osutuvad kollineaarse-



Joonis 7.1: Pseudo-tõusunurga arvutamine.

teks, st. kui tegelikult pööret punktis p ei toimugi:

$$vp(p_1, p, p_2) = \begin{cases} 1, & \text{kui punktis } p \text{ on pööre vastupäeva,} \\ & \text{või } p \text{ paikneb } p_1 \text{ ja } p_2 \text{ vahel samal sirgel} \\ -1, & \text{kui punktis } p \text{ on pööre päripäeva,} \\ & \text{või } p_1 \text{ paikneb } p_2 \text{ ja } p \text{ vahel samal sirgel} \\ 0, & \text{kui } p_2 \text{ paikneb } p_1 \text{ ja } p \text{ vahel samal sirgel} \\ & \text{või } p_2 = p \text{ või } p_2 = p_1 \end{cases}$$

Kasutades funktsiooni vp on nüüd lihtne koostada algoritm kontrollimaks, kas kahel antud lõigul leidub ühine punkt (vt. joonis 7.3). Selline **lõikude lõikumise** kontrollimise alamülesanne esineb omakorda mitmetes teistes planimeetria probleemides, sealhulgas järgmisena vaadeldavas ülesandes **punkti kuuluvusest hulknurka**.

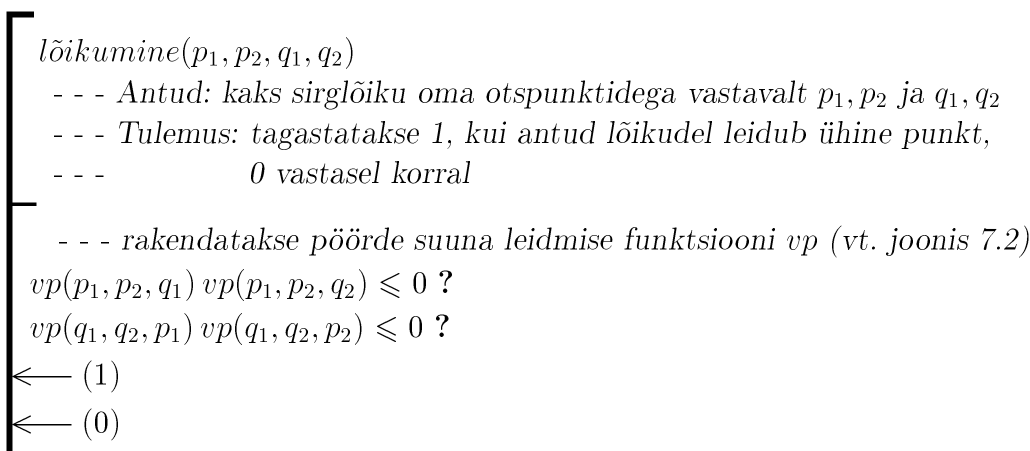
Olgu antud suvaline hulknurk. Selleks, et kindlaks teha, kas mingi punkt asub hulknurga sees, vaatleme sellest punktist suvalises suunas väljuvat kiirt.

$vp(p_1, p, p_2)$
- - - *Antud: kolm punkti p_1, p ja p_2*
- - - *Tulemus: uuritud pöörde suunda, mis tehakse punktis p liikumisel*
- - - *lühimat teed mööda punktist p_1 punkti p_2 läbi punkti p*

$dx_1 := p.x - p_1.x;\ dy_1 := p.y - p_1.y$
$dx_2 := p_2.x - p_1.x;\ dy_2 := p_2.y - p_1.y$
$dx_1 dy_2 > dy_1 dx_2$?
$\longleftarrow$ (1) - - - *pööre toimub vastupäeva*
$dx_1 dy_2 < dy_1 dx_2$?
$\longleftarrow$ (-1) - - - *pööre toimub päripäeva*
- - - *punktid on kollineaarsed, pööret ei toimu;*
- - - *uurida kolme antud punkti vastastikuseid asendeid sirgel:*
$dx_1 dx_2 < 0$?
$\longleftarrow$ (-1)
$dy_1 dy_2 < 0$?
$\longleftarrow$ (-1)
$dx_1^2 + dy_1^2 \geqslant dx_2^2 + dy_2^2$?
$\longleftarrow$ (0)
$\longleftarrow$ (1)

Joonis 7.2: Vastupäeva pöördumise kontrollimine teel $p_1 \to p \to p_2$.

Hulknurga sees asuvast punktist lähtuv kiir lõikab hulknurga külgi paaritu arv korda: pärast esimest lõikumist jõuab ta hulknurgast välja, pärast teist lõikumist naaseb sisse tagasi jne. Järelikult viimase lõikumise järjenumber on paaritu. Hulknurgast väljas asuva punkti korral on olukord vastupidine: liikudes mööda kiirt satutakse pärast esimest lõikumist hulknurga sisse, pärast teist aga jõutakse välja jne. Seega viimase lõikumise järjenumber on paarisarv. Kui aga kiir hulknurka üldse ei taba (see saab juhtuda vaid väljaspool hulknurka asuva punkti korral), siis on lõikumiste arv 0 samuti paaris. Paraku teeb ülesande lahendamise keerukamaks juht, kus kiir läbib parajasti ühte hulknurga tippudest. Siis võib näiteks väljastpoolt tulnud kiir sisse



Joonis 7.3: Lõikude lõikumise kontrollimine.

mitte jõuda, või hakata koguni mööda hulknurga külge edasi minema.

Algoritmis joonisel 7.4 eeldatakse, et hulknurk on antud oma tippude järjendiga loomulikus ümber hulknurga käimise järjekorras. Kiire asemel kasutatakse horisontaalset (x-teljega paralleelset) kontrolllõiku, mille üks otspunkt on uuritav punkt, teine otspunkt aga võimalikult suure abstsissiga (mis on suurem hulknurga suvalise punkti abstsissist). Liigutakse mitte mööda kontrolllõiku, vaid vaadeldakse järjekorras hulknurga tippe, loendades üldiselt kontrolllõigu lõikumisi hulknurga külgedega, täpsemalt lõikudega (p_j, p_i), kus p_i on järjekordne hulknurga tipp. Kui pole veel esinenud kontrolllõigul paiknevaid hulknurga tippe, siis $p_j = p_{i-1}$.

Kui aga esineb tipp p_i, mis asub kontrolllõigul, siis jäetakse see lihtsalt vahele, kuid järgmisena kontrollitava lõigu alguspunktiks jääb nüüd ikka p_j. Näiteks kui järgmine tipp p_{i+1} ei asu kontrolllõigul, siis järgmisel sammul tuleb uurimisele lõik (p_j, p_{i+1}), mitte hulknurga külg (p_i, p_{i+1}). Uuritakse aga (p_j, p_{i+1}) lõikumist kontrollõiku sisaldava sirgega (mitte kontrolllõiguga).

Vaadeldavas algoritmis toimub esimese sammuna teatava kindla tipu väljavalimine ja järjendis esikohale seadmine. Selline **lähtepunkti** ehk ankrupunkti fikseerimine osutub kasulikuks ka paljudes teistes planimeetria algoritmides. Käesolevas võetakse lähtepunktiks vähima ordinaadiga tipp (seda nimetatakse ka kõige alumiseks tipuks). Kui alumisi tippe on mitu, siis nendest valitakse vähima abstsissiga (vasakpoolseim) punkt.

```
punkt_hulknurgas(P, t)
 --- Antud: hulknurk P oma tippude järjendiga p1, p2, ... pn ja punkt t
 --- Tulemus: tagastatakse 1, kui punkt t kuulub hulknurka P,
 ---              0 vastasel korral

 --- kasutatakse veel lisaelementi p0
 --- kiirt kujutava kontrolllõigu otspunktideks on t ja t^{+∞} = (+∞, t.y)
 --- kiiresirget kujutab lõik otspunktidega t^{-∞} = (−∞, t.y) ja t^{+∞}

 [ leida hulknurga alumine tipp pm;
 [ kui alumisi tippe on mitu, siis pm on neist kõige vasakpoolsem

 [ tuua alumine esikohale järjendis,
 [ säilitades tippude loomuliku järjestuse

 loe := 0;  j := 0
 p0 := pn;  --- et pj oleks määratud ka tsükli esimesel sammul

 ⋆ i = 1, 2, ... , n
    --- pj on viimane selliste juba vaadeldud tippude seas,
    --- mis ei asu kontrolllõigul
   lõikumine(pi, pi, t, t^{+∞}) ?
    --- pi asub kontrolllõigul, loendajat ja indeksit j ei muudeta
  <----
    --- pi ei asu kontrolll oigul
   [? (j = i − 1) u := t : u := t^{-∞}
    --- (u, t^{+∞}) on kiir või kiiresirge
   [? (lõikumine(pj, pi, u, t^{+∞})) loe ++
   j := i;  --- uus j

<— (loe mod 2)
```

Joonis 7.4: Punkti hulknurka kuulumise kontrollimine.

7.2 Punktihulga töötlemine

Punktihulga töötlemise ülesannetele on iseloomulik, et ülesande püstitusest vahetult tuletatavad lihtsad, nn. "jõumeetodil" algoritmid on suure ajalise keerukusega. Ülesannete lähemal analüüsimisel aga osutub paljudel juhtudel võimalikuks leida märksa efektiivsemaid lahendusteid. Öeldu kehtib ka järgnevas vaadeldava kahe näite puhul.

Punktihulga Q **kumeraks katteks** nimetatakse vähimat kumerat hulknurka P, mille korral iga punkt hulgast Q asub kas hulknurgal P või selle sisealas.

Kumera katte leidmine jõumeetodil võiks toimuda sel teel, et kontrollitakse iga punktipaari korral, kas kõik ülejäänud punktid antud hulgas asuvad seda paari läbivast sirgest ühel pool. Kui see on nii, siis vaadeldavad kaks punkti kuuluvad kumerasse kattesse. Kuna n-elemendilises hulgas leidub $\Theta(n^2)$ paari ja iga paari korral uuritakse ülejäänud $n-2$ punkti, siis sellise algoritmi ajaline keerukus on $\Theta(n^3)$.

Üheks tüüpiliseks võtteks planimeetria ülesannete lahendusaja vähendamiseks on püüda vähendada vaatlusele võetavate punktide arvu, elimineerides lihtsa eelneva analüüsi põhjal need punktid, mis lahendustulemust ei saa mõjutada. Näiteks kumera katte leidmisel võib vaatlusest välja jätta suvalise sellise nelinurga sisealas asuvad punktid, mille tippudeks on valitud neli punkti antud punktihulgast. Üks võimlusi on valida need punktid, milleni (piltlikult öeldes) esimesena jõutakse, kui antud punktide kogumile "lähenetakse" neljast küljest sirgetega, mille tõus on 1 või -1. Nelinurga tippudeks on siis järgmised neli punkti: (1) vasak ülemine tipp – punkt, mille abstsissi ja ordinaadi vahe on minimaalne; (2) vasak alumine tipp – punkt, mille koordinaatide summa on minimaalne; (3) parem alumine ja (4) parem ülemine – punktid, millel vastavalt koordinaatide vahe või summa on maksimaalne. Olles leidnud nelinurga tipud, pole raske iga punkti korral määrata, kas ta kuulub sellesse nelinurka, st. elimineerimisele või mitte. Mõistagi saab osa punktide eelneva väljalülitamisega vähendada vaid keskmist keerukust. Halvimal juhul võib osutuda, et leitud nelinurga sisealasse ei kuulu mitte ühtegi lähtehulga punkti.

Antud punktihulga kumera katte efektiivsemaks leidmiseks on mitmeid

võimalusi. Käesolevas tutvume **Grahami seiremeetodiga**, mille ajaline keerukus on $O(n \log n)$. Vastavas algoritmis joonisel 7.5 fikseeritakse kõigepealt nö. vaatlus- ehk seirepunktiks kõige väiksema abstsissiga punkt vähima ordinaadiga punktide seas. See on leitava kumera katte esimene tipp ning ta tuuakse antud punktijärjendis ka esikohale. Igale ülejäänud punktile vastab nüüd teda läbiv (seire)kiir, mis lähtub seirepunktist. Punktide hulk (ilma seirepunktita) sorteeritakse vastavate seirekiirte (pseudo-)tõusunurkade järgi mittekahanevalt. Arvestades, et seirepunkt asub "all vasakul", on selge, et läbides niiviisi sorteeritud punktijärjendit vasakult paremale, peavad kumeral kattel asuvad punktid esinema kumera katte vastupäeva läbimise järjekorras. Kokkulangevate seirekiirte korral on üsna ilmne, et kumeral kattel saab neist esineda vaid seirepunktist kõige kaugemal asuv punkt.

Algoritmi põhitsükli i-nda sammu alguseks on punktijärjendi lõpuosa alates elemendist p_{i+1} veel vaatlemata (vt. ka joonis 7.7). Kuid järjendi algusossa on koondatud niisugused m punkti, mis osutuvad kumera katte tippudeks punktide p_2 ja p_m seirekiirte vahelisesse sektorisse jäävale punktide alamhulgale: teel $p_1 \to p_2 \to \ldots \to p_m$ pöördutakse igas sisemises tee punktis rangelt vastupäeva. Peamine ülesanne tsükli sammul on otsustada, kas selle tee lõppu sobib ka p_i, st. kas teel $p_{m-1} \to p_m \to p_i$ pöördutakse punktis p_m samuti rangelt vastupäeva. Sobivuse korral tuuaksegi punkt p_i kohale $m+1$. Kui aga p_i on selline, et teel $p_{m-1} \to p_m \to p_i$ ei pöörduta vastupäeva, siis just p_m ei saa olla kumera katte tipp. Punkti p_m "väljapraakimiseks" pole tarvis teha muud, kui vähendada m väärtust ühe võrra. Seejärel tuleb jätkata p_i lisamise võimaluse uurimist. Tsükli täitmine lõpeb seisus, kus järjendi algusosas leiduvad niisugused m punkti, mis on kumera katte tippudeks punktide p_2 ja p_n (st. vähima ja suurima tõusunurgaga) seirekiirte vahelisesse sektorisse jäävale punktide alamuhulgale, seega ka kogu punktide hulgale.

Vaatleme lõpuks veel mõningaid punktidevahelise kauguse ehk vahemaaga seotud küsimusi.

Antud punktihulgas **suurima vahemaaga** punktipaari otsimise ülesanne on taandatav kumera katte leidmisele. Nimelt saab näidata, et kaks teineteisest kõige kaugemal asuvat punkti peavad olema selle punktihulga kumera katte tippudeks. Leidub ka algoritm ajalise keerukusega $O(m \log m)$ otsimaks m-tipulise kumera hulknurga suurima vahemaaga tipupaari (käesolevas me

seda algoritmi ei esita). Järelikult saab suurima vahemaaga punktipaari n-tipulisest hulgast kätte ajaga $O(n \log n)$.

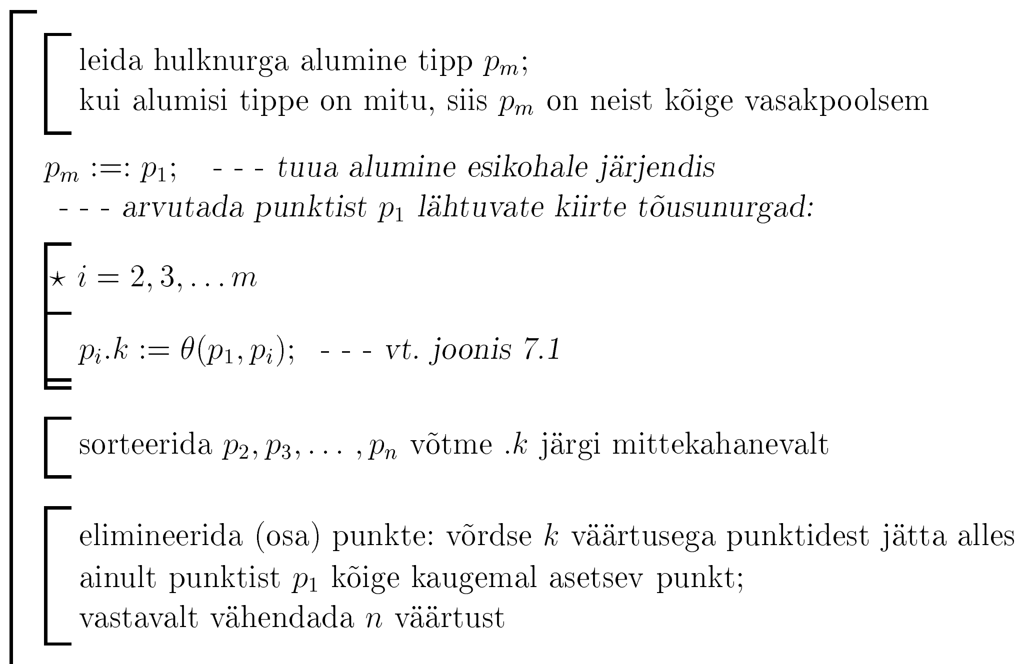
Ka **vähima vahemaaga punktipaari** ehk **lähima paari** leidmise ülesanne on lahendatav ajaga $O(n \log n)$. Jõumeetod, kus lihtsalt vaadatakse läbi vahemaad kõigi punktipaaride korral, on ilmselt $\Theta(n^2)$ keerukusega. Siin vaadeldava meetodi idee seisneb järgmises. Kõigepealt sorteeritakse punktid x-koordinaadi järgi mittekahanevalt. Saadud punktijärjend jaotatakse seejärel kaheks enamvähem võrdeks osaks. Olgu nende osade eraldusjooneks sirge $x = x_0$. Mõlemas osas eraldi leitakse lähim paar, edasisele vaatlusele jäetakse neist loomulikult väiksema vahemaaga paar (A, B). On selge, et kogu punktihulgas olev lähim paar on kas (A, B) või mõni sellistest paaridest, millest üks punkt asub ühel-, teine teiselpool eraldusjoont $x = x_0$. Viimasel juhul aga ei saa kumbki niisuguse paari punktidest olla eraldusjoonest kaugemal, kui δ=(punktide A ja B vaheline kaugus), sest vastasel korral oleks nende vahemaa suurem kui δ, ning see paar ei saaks konkureerida paariga (A, B). Järelikult tuleb peale (A, B) leidmist veel läbi vaadata punktid, mis asuvad vertikaalsel ribal ääresirgetega $x = x_0 - \delta$ ja $x = x_0 + \delta$. Selleks sorteeritakse ribal asuvad punktid y-koordinaatide järgi ja siis uuritakse iga punkti korral tema kaugust talle eelnevatest punktidest selles järjestuses. Kui nende hulgas leitud vähim kaugus on väiksem kui δ, siis on vastav paar ka otsitavaks lahendiks, vastasel korral jääb selleks paar (A, B).

Osutub, et riba iga punkti korral piisab vaadelda kaugusi vaid talle eelnevast neljast riba punktist. Seda geomeetrilist tõsiasja me siin ei tõesta, märgime vaid, et aluseks on asjaolu, et $\delta \times \delta$ ruudus saab neli üksteisele mitte lähemal kui δ ühikut asuvat punkti paikneda vaid ruudu tippudes.

Ülalkirjeldatud idee lähima paari otsimiseks on joonisel 7.8 realiseeritud rekursiivse algoritmina. Kuna jaotamine toimub triviaalselt, siis sobib ka sorteerimiseks kasutada just ühildusmeetodit. Sellest tulenevalt eeldatakse tippude järjendi esitust lihtahelana. Enamgi veel, riba punktide analüüsi saab sobitada y-koordinaadi järgi sorteerimise skeemi, selle saab sooritada kohe pärast järjekordse osa sorteerimist. Algoritmi töö käigus säilitatakse senini leitud lähim paar (A, B) ja sellele vastav kaugus δ globaalsetes muutujates.

```
Graham_seire(p, n)
 - - - Antud: punktide järjend p1, p2, ... pn; iga punktiga on peale
 - - -          koordinaatide x, y seotud veel väli .k
 - - - Tulemus: leitud antud punktidehulga kumer kate;
 - - -            sellesse kuuluvad punktid on järjendis esikohale toodud;
 - - -            tagastatakse punktide arv kumeras kattes

  [ - - - ettevalmistav osa, vt. joonis 7.6 ]

 m := 3
 - - - leida kumer kate:
 ⋆ i = 4, 5, ... n
    - - - punktide järjend koosneb järgmistest osadest:
    - - - p1, ... pm−1, pm – võimalikud kandidaadid kumerasse kattesse
    - - -    (liikumisel p1 → ... → pm−1 → pm pöördutakse vastupäeva)
    - - - pm+1 ... pi−1 – punktid, mis on juba välja praagitud, st.
    - - -    läbikukkunud kandidaadid
    - - - pi – käsiloleval sammul vaadeldav punkt
    - - - pi+1 ... pn – veel vaatlusele võtmata punktid
    liikumisel pm−1 → pm → pi ei pöörduta vastupäeva ?
      m −−;    - - - praakida välja punkt pm
    - - - liikumisel pm−1 → pm → pi pöörduti vastupäeva
   p++m :=: pi;  - - - pi lülitatakse kandidaatide rivi lõppu;
    - - - pi kohale tuli nüüd juba varem välja praagitud tipp
 ← (m)
```

Joonis 7.5: Kumera katte leidmine Grahami seiremeetodil.



Joonis 7.6: Ettevalmistav osa Grahami seiremeetodis.

Joonis 7.7: Tipujärjendi p struktuur Grahami seiremeetodis.

```
y_sorteerida(p, k)
 --- Antud: päisega lihtahel p, milles on k liiget (päist arvestamata),
 ---         ahela iga liige (l) kujutab ühte tasandi punkti (l.x; l.y);
 ---         ahel p on sorteeritud võtme x järgi mittekahanevalt;
 ---         globaalsetena on antud senini teadaolev lähim paar (A, B)
 ---         ja sellele vastav vahemaa δ
 --- Tulemus: antud lihtahel sorteeritud võtme y järgi;
 ---           kui selle sorteerimise käigus avastati veelgi lähem paar,
 ---           siis vastavalt muudetud (parandatud) (A, B) ja δ;
 ---           tagastatakse viit ahela viimasele liikmele

k = 1 ?
<— (p.viit)
 r := y_sorteerida(p, ⌊k/2⌋)
 x0 := r.viit.x;   --- x0 on lähtejärjendi keskmise punkti abstsiss
 s := y_sorteerida(r, ⌈k/2⌉)
  --- ühildada osad p...r ja r...s arvestades võtmega y:
 v := ühildada_ahelad(p, ⌊k/2⌋, r, ⌈k/2⌉))
  --- vt. ka joonis 4.9
  --- p...r...s sorteeritud võtme y järgi mittekahanevalt
  --- lähimaks paariks osades p...r ja r...s on (A, B) vahemaaga δ;
   [ --- uurida veel punkte 2δ laiusel ribal sirge x = x0 ümbruses,
   [ --- vt. joonis 7.9
<— (v)
```

Joonis 7.8: Lähima paari otsimine y-koordinaadi järgi ümberjärjestamise käigus.

```
u := p;  u1 := u2 := u3 := u4 := Λ
  ⋆ k
    u := u.viit
     - - - u on järjekordne vaadeldav tipp
      |u.x − x0| < δ ?
        - - - punkt u asub ribal, u võib moodustada uue lähima paari
        - - - ühega neljast talle ahelas eelnevast punktist u1, u2, u3, u4
       kontr(u, u1);  kontr(u, u2);  kontr(u, u3);  kontr(u, u4);
        - - - funktsioon kontr on kirjeldatud joonisel 7.10
     - - - eelnevate punktide nelik järgmise u jaoks:
    u1 := u2;  u2 := u3;  u3 := u4;  u4 := u
```

Joonis 7.9: Punktide uurimine ribal (lähima paari leidmise algoritmi osa).

```
kontr(a, b)
 - - - Antud: kaks punkti viitadega a ja b
 - - -        ning (globaalsetena) senini leitud lähima paari punktid A, B
 - - -        koos nende vahemaaga δ
 - - - Tulemus: kui (a, b) on veelgi lähem paar,
 - - -          siis vastavalt muudetud (parandatud) A, B ja δ
a ≠ Λ ∧ b ≠ Λ ?
 d := √((a.x − b.x)² + (a.y − b.y)²)
d < δ ?
 δ := d;
 A.x := a.x;  A.y := a.y;  B.x := b.x;  B.y := b.y
```

Joonis 7.10: Abifunktsioon lähima paari otsimisel.

Peatükk 8

Algoritmi korrektsus

Käesolevas peatükis tutvustatakse ühte võimalust algoritmide korrektsuse probleemi formaalseks käsitlemiseks, kus rangelt spetsifitseeritud algoritme vaadeldakse kui matemaatilisi objekte. Lihtsustavatel eeldustel selgitatakse, milles seisneb algoritmi korrektsuse matemaatilise tõestamise idee. Tõestustes kasutatavate formaalsete võtete lihtsus peaks lugejas tekitama mõningase ettekujutuse algoritmide (ja programmide) automaatse korrektsuse tõestamise põhimõttelisest võimalikkusest.

8.1 Algoritmi korrektsuse mõiste

Iga algoritm Π, mis on varustatud eeltingimusega (P) ja järeltingimusega (Q) kujutab endast teoreemi, mis väidab, et algoritm on korrektne, st. algoritmi sisemine spetsifikatsioon (Π) on kooskõlas tema välimise spetsifikatsiooniga (P, Q).

Paneme tähele, et eel- ja/või järeltingimuseta algoritmi (nagu näiteks algoritm joonisel 8.1) korrektsuse kohta ei ole võimalik midagi väita.

Käesolevas piirdume lihtsamate juhtudega, kus nii eel- kui ka järeltingimused on esitatavad predikaatidena (tingimustena) algoritmi olekute ruumis.

Loobume ka muutujate skoopide eristamisest, eeldades, et kõikide muutujate skoobiks on terve algoritm. Sel korral defineerime **olekute ruumi** kui muutjate väärtusehulkade otsekorrutise $V_1 \times V_2 \times \ldots \times V_m$, kus m on muutujate arv ja V_i on i-nda muutuja kõigi väärtuste hulk $(i = 1, 2, \ldots, m)$.

$m := 2$
$k := 1$
$k \leqslant n$ **?**
$m := m \times k$
$k := k + 1$

Joonis 8.1: Välise spetsifikatsioonita algoritm.

- - - $n \geqslant 1$
$m := 2$
$k := 1$
$k \leqslant n$ **?**
$m := m \times k$
$k := k + 1$
- - - $m = n!$

Joonis 8.2: Välise spetsifikatsiooniga algoritm.

- - - $n \geqslant 0$
$m := 1$
$k := 1$
$k = n$ **?**
$m := m \times k;\ k := k + 1$
- - - $m = n!$

Joonis 8.3: Mittenegatiivse täisarvu faktoriaal.

- - - $n > 0$
$m := 1$
$k := 2$
$k = n + 1$ **?**
$m := m \times k;\ \ k := k + 1$
- - - $m = n!$

Joonis 8.4: Naturaalarvu faktoriaal.

- - - $n \geqslant 0$
$m := 1$
$k := 2$
$k = n + 1$ **?**
$m := m \times k;\ \ k := k + 1$
- - - $m = n!$

Joonis 8.5: Mittenegatiivse täisarvu faktoriaal (vaid osaliselt korrektne).

Olekuks nimetatakse olekute ruumi elementi, st muutujate konkreetsete väärtuste järjendit.

Algoritmis joonisel 8.1 (samuti kui joonisel 8.2) on muutujateks n, m ja k, olekute näideteks järjendid $(0,0,0), (0,0,1), (3,6,6), (-5,1278,103)$. Algoritmi täitmise käigus omandavad muutujad erinevaid väärtusi, pärast iga operatsiooni täitmist tekkinud ehk **saavutatud** olekut nimetatakse ka **algoritmi olekuks** sellel täitmismomendil.

Olgu antud algoritm Π, eeltingimus P ja järeltingimus Q. Siis vastav **algoritmi korrektsuse teoreem** (ehk algoritmi täieliku korrektsuse teoreem) esitatakse kujul

- - - P
Π või $P \stackrel{\Pi}{\Longrightarrow} Q$
- - - Q

mis mõlemad loetakse järgmise teoreemi lühemaks üleskirjutiseks:

Teoreem. *Kui algoritmi Π täitmist alustatakse olekus, mis rahuldab tingimust P, siis algoritmi täitmine lõpeb pärast lõpliku arvu operatsioonide sooritamist ja lõpetamise hetkeks saavutatud olek rahuldab tingimust Q.*

Algoritm on täielikult korrektne, kui tema jaoks kehtib (on tõestatud) vastav korrektsuse teoreem. Enne tõestamisele asumist on muidugi otstarbekohane kontrollida teoreemi väite kehtivust mitmetel konkreetsetel erijuhtudel. Taoline kontrollimine vastab programmeerimise valdkonnast hästi tuntud testimistegevusele. Olles näiteks proovinud joonisel 8.2 olevat algoritmi "läbi teha" neljast eeltingimusega ($n \geqslant 1$) lubatud algolekust, vt. tabel 8.1, saame lõppolekud, milles kõigis kehtib $m = n!$. Järelikult võiks nagu üritada leida täieliku korrektsuse tõestust. See aga kindlasti ebaõnnestub, kuna vaadeldav algoritm ei ole tegelikult korrektne. Selleks, et näidata algoritmi ebakorrektsust ehk vigasust, piisab vaid ühe nö. kontranäite leidmisest. Sel puhul pole ilmselt mõtet tõestamisele asudagi. Ülaltehtud näitealgoritmi testimised kinnitasid selle algoritmi õiget tööd parajasti neljal konkreetsel erijuhul, ei midagi enamat. Nende katsetuste hulka ei sattunud paraku algoritmi ebakorrektsust tõestav kontranäide. Algoritm joonisel 8.2 osutubki vigaseks (vt. tabel 8.2), kuna nt. alustades lubatud olekust, kus $n = 1$ ja $m = 1$ ning $k = 1$, algoritm lõpetab olekus $n = 1, m = 2$ ja $k = 2$. Kuid selles olekus ei kehti $m = n!$, sest $m = 2$, aga $n! = 1! = 1$. Küll aga kehtivad joonistel 8.3

Tabel 8.1: Joonisel 8.2 oleva algortimi testimistulemusi.

	n	m	k	
algolek	2	1	1	kehtib: $n \geqslant 1$
lõppolek	2	2	3	kehtib: $m = n!$
algolek	3	-1	1	kehtib: $n \geqslant 1$
lõppolek	3	6	4	kehtib: $m = n!$
algolek	4	0	0	kehtib: $n \geqslant 1$
lõppolek	4	24	5	kehtib: $m = n!$
algolek	6	1	1	kehtib: $n \geqslant 1$
lõppolek	6	720	7	kehtib: $m = n!$

Tabel 8.2: Joonisel 8.2 oleva algortimi ebakorrektsuse tõestus.

	n	m	k	
algolek	1	1	1	kehtib: $n \geqslant 1$
lõppolek	1	2	2	!!! ei kehti: $m = n!$

ja 8.4 esitatud korrektsuse teoreemid, st. need kaks algoritmi on täielikult korrektsed (neist esimese tõestus antakse järgmises jaotises).

8.2 Algoritmi korrektsuse tõestamine

Täieliku korrektsuse teoreemi $P \stackrel{\Pi}{\Longrightarrow} Q$ tõestamiseks
1) lähtudes järeltingimusest Q ja algoritmist Π konstrueeritakse (leitakse) nõrgim eeltingimus P', nii et $P' \stackrel{\Pi}{\Longrightarrow} Q$;
2) näidatakse, et kehtib implikatsioon $P \Rightarrow P'$.

Skemaatiliselt:

- - - P (antud)

- - - P' (leida ja näidata $P \Rightarrow P'$)

Π (antud)

- - - Q (antud)

Järgnevates alajaotistes esitatakse algoritmides esinevatele lihtsamatele juhtimiskonstruktsioonidele vastavad ilmselt (aksiomaatiliselt) kehtivad tuletusreeglid ning tuuakse mõningaid nende rakendamise näiteid. Kaks esimest aksioomi (8.1 ja 8.2) esindavad väga lihtsaid konstruktsioone ega vaja põhjalikumat selgitamist.

8.2.1 Tühi algoritm

- - - P

Λ

- - - Q

Tühjas algoritmis (Λ) ei ole ühtegi operatsiooni, seega täitmisel olekumuutust ei toimu ning „saavutatud“ olekus saab olla rahuldatud vaid tingimus (Q), mis ei ole nõrgem kui eeltingimus (P). Tuletusreegel:

$$\frac{P \Rightarrow Q}{P \stackrel{\Lambda}{\Longrightarrow} Q} \tag{8.1}$$

Nõrgim eeltingimus on Q.

8.2.2 Järjestikalgoritm

- - - P
Π
- - - R ehk $P \stackrel{\Pi\Pi'}{\Longrightarrow} Q$
Π'
- - - Q

Tuletusreegel:

$$\frac{P \stackrel{\Pi}{\Longrightarrow} R, R \stackrel{\Pi'}{\Longrightarrow} Q}{P \stackrel{\Pi\Pi'}{\Longrightarrow} Q} \tag{8.2}$$

Kui R on nõrgim eeltingimus algoritmi
Π'
- - - Q
korrektsuseks ja ja P on nõrgim eeltingimus algoritmi
Π
- - - R
korrektsuseks, siis P on nõrgim eeltingimus antud järjestikalgoritmi korrektsuseks.

8.2.3 Omistamisdirektiiv

- - - P
$x := e$ ehk $P \stackrel{x:=e}{\Longrightarrow} Q$
- - - Q

Tuletusreegel (eeldusel, et avaldise e arvutamisel olek ei muutu):

$$\frac{P \Rightarrow Q_e^x}{P \stackrel{x:=e}{\Longrightarrow} Q} \tag{8.3}$$

Nõrgim eeltingimus Q_e^x saadakse järeltingimusest Q muutuja x kõigi esinemiste asendamisel avaldisega (e).

Tuletusreegli rakendamise näide:

Teoreem 1
- - - $x + y < y^2 \wedge y \geqslant 0 \wedge x \geqslant 0$
$x := \sqrt{x} - \sqrt{y}$
- - - $x^2 + 2x\sqrt{y} < y^2$

Tõestus.
Eeltingimus: $P \Leftrightarrow x + y < y^2 \wedge y \geqslant 0 \wedge x \geqslant 0$.
Järeltingimus: $Q \Leftrightarrow x^2 + 2x\sqrt{y} < y^2$.
Nõrgim eeltingimus Q_e^x saadakse järeltingimusest omistamise sihtmuutuja x kõigi esinemiste asendamisel omistamistatava avaldisega $(\sqrt{x} - \sqrt{y})$. Seega $Q_e^x \Leftrightarrow (\sqrt{x} - \sqrt{y})^2 + 2(\sqrt{x} - \sqrt{y})\sqrt{y} < y^2 \Leftrightarrow x - y < y^2$. Eeltingimusest järeldub nõrgim eeltingimus, $P \Rightarrow Q_e^x$: $x+y < y^2 \wedge y \geqslant 0 \wedge x \geqslant 0 \Rightarrow x-y < y^2$. □

Just asjaolu, et omistamisdirektiivi jaoks on järeltingimusest lähtudes väga kergesti, n.ö. „masinlikult“ konstrueeritav nõrgim eeltingimus, määrabki algoritmi korrektsuse tõestamise üldise alt-üles suuna. Kui püüda tõestus läbi viia ülevalt-alla, siis tuleks leida vaadeldava algoritmiosa jaoks tugevaim järeltingimus, lähtudes antud eeltingimusest. Omistamisdirektiivi korral oleks tugevaimaks järeltingimuseks P_x^e, mis saadakse tingimusest P kõigi osaavaldiste e asendamisel muutujaga x. Kuid see nõuaks tihtipeale eelnevat keerulist P teisendamist sellisele kujule, kus ilmutatatult esineksid osaavaldised e.

Näiteks teoreemi 1 ülevat-alla tõestamiseks tuleks esmalt teha teisendus $x + y < y^2 \wedge y \geqslant 0 \wedge x \geqslant 0 \Rightarrow (\sqrt{x} - \sqrt{y} + 2\sqrt{y})(\sqrt{x} - \sqrt{y}) < y^2$. Alles viimasest saab siis leida tugevaima järeltingimuse (asendamisel $(\sqrt{x} - \sqrt{y}) \rightarrow x$), milleks on $(x + 2\sqrt{y})x < y^2 \Leftrightarrow x^2 + 2x\sqrt{y} < y^2$.

8.2.4 Tingimuslik direktiiv

- - - P
Π
t **?**
Π' ehk $P \overset{\Pi}{\Longrightarrow} Q$
←
Π''
Π
- - - Q

Tuletusreegel (eeldusel, et tingimuse t kontrollimise käigus olek ei muutu):

$$\frac{t \wedge P \overset{\Pi'}{\Longrightarrow} Q, \neg t \wedge P \overset{\Pi''}{\Longrightarrow} Q}{P \overset{\Pi}{\Longrightarrow} Q} \tag{8.4}$$

Nõrgim eeltingimus on $(t \wedge R') \vee (\neg t \wedge R'')$, kus R' ja R'' on nõrgimad eeltingimused vastavalt algoritmide

Π' ja Π''
- - - Q - - - Q

korrektsuseks.
Erijuhul, kui teine osa puudub ($\Pi'' = \Lambda$), on tingimusliku direktiivi

t **?**
Π'

- - - Q
nõrgimaks eeltingimuseks $(t \wedge R') \vee (\neg t \wedge Q)$.

Teoreem 2
- - - $x > 0 \wedge y > 0$
$x > y$ **?**
$k := x - y$
←
$k := y - x$
- - - $k = |x - y|$

Tõestus.
Algoritmiks on tingimuslik direktiiv eeltingimusega $P \Leftrightarrow x > 0 \wedge y > 0$,

järeltingimusega $Q \Leftrightarrow k = |x - y|$ ning hargnemistingimusega $t \Leftrightarrow x > y$. Alternatiivselt täidetavateks osadeks on omistamisdirektiivid

$k := x - y;$ - - - Π'

ja

$k := y - x;$ - - - Π''.

Vajalikud vahetingimused R' ja R'' leiame omistamise reegli 8.3 põhjal:

- - - $x - y = |x - y| \Leftrightarrow Q^k_{x-y} \Leftrightarrow R'$

$k := x - y;$ - - - Π'

- - - $k = |x - y| \Leftrightarrow Q$

ning

- - - $y - x = |x - y| \Leftrightarrow Q^k_{y-x} \Leftrightarrow R''$

$k := x - y;$ - - - Π''

- - - $k = |x - y| \Leftrightarrow Q$

Antud tingimusliku direktiivi nõrgim eeltingimus järeltingimuse Q korral:
$P' \Leftrightarrow (x > y \wedge x - y = |x - y|) \vee (x \leqslant y \wedge y - x = |x - y|)$.
Kuna P' on samaselt tõene, siis $P \Rightarrow P'$.

□

Paneme tähele, et samaselt tõene nõrgim eeltingimus (P') tähendab tegelikult seda, et algoritm on korrektne suvalise eeltingimuse (P) korral (sest siis kindlasti $P \Rightarrow P'$) ehk soovitud lõppolek (mis rahuldab järeltingimust Q) saavutatakse suvalisest algolekust lähtudes. Seega ka vaadeldud tingimusdirektiiv lõpetab alati olekus, mis rahuldab tingimust $k = |x - y|$, sõltumata sellest, millises olekus täitmist alustatakse. Teiste sõnadega, tegemist on suvaliste väärtustega muutujate x ja y vahe absoluutväärtuse leidmise ja muutujale k omistamise algoritmiga.

Teoreem 3

- - - $x + y = m$

$x > y$ **?**
  $a := x$
  $x := y$
  $y := a$

- - - $x \leqslant y \wedge x + y = m$

Tõestus.
Algoritmiks on üheosaline tingimuslik direktiiv eeltingimusega $P \Leftrightarrow x + y = m$, järeltingimusega $Q \Leftrightarrow x \leqslant y \wedge x + y = m$ ning hargnemistingimusega $t \Leftrightarrow x > y$. Viimase rahuldatuse korral täidetakse kolmest omistamisdirektiivist koosnev järjestikalgoritm (Π'):

$a := x$
$x := y$
$y := a$

Vajaliku vahetingimuse R' leiame reegleid 8.3 ja 8.2 rakendades. Selleks, alustades järeltingimusest Q, konstrueerime järjekorras tingimused Q_1, Q_2 ja Q_3:

\- - - $Q_3 \Leftrightarrow y \leqslant x \wedge y + x = m \Leftrightarrow R'$
$a := x$
\- - - $Q_2 \Leftrightarrow y \leqslant a \wedge y + a = m$
$x := y$
\- - - $Q_1 \Leftrightarrow x \leqslant a \wedge x + a = m$
$y := a$
\- - - $Q \Leftrightarrow x \leqslant y \wedge x + y = m$

Antud tingimusliku direktiivi nõrgim eeltingimus järeltingimuse Q korral:
$P' \Leftrightarrow (t \wedge R') \vee (\neg t \wedge Q) \Leftrightarrow$
$\Leftrightarrow (x > y \wedge y \leqslant x \wedge y + x = m) \vee (x \leqslant y \wedge x \leqslant y \wedge x + y = m) \Leftrightarrow$
$\Leftrightarrow (x > y \wedge y + x = m) \vee (x \leqslant y \wedge x + y = m) \Leftrightarrow$
$\Leftrightarrow (x > y \vee x \leqslant y) \wedge (y + x = m) \Leftrightarrow$
$\Leftrightarrow y + x = m \Leftrightarrow$
$\Leftrightarrow P$.
Seega $P \Rightarrow P'$.

□

8.2.5 Tsüklidirektiiv

Seoses tsüklidirektiiviga kerkib algoritmi töö lõplikkuse probleem. Nimelt võib osutuda, et algoritmi täitmine ei lõpegi, kuigi alustati olekust, mis rahul-

dab eeltingimust. Näiteks, kui joonisel 8.5 esitatud algoritmi täitmist alustada lubatud olekust $n = 0, m = 1, k = 1$, siis paraku tsüklist väljumist ei saa toimuda, sest lõpetamistingimust ($k = n + 1$) rahuldavat olekut iialgi ei saavutata.

Otstarbekohane on sisse tuua algoritmi **osalise korrektsuse** mõiste.

Olgu antud algoritm Π, eeltingimus P ja järeltingimus Q. Siis vastav **algoritmi osalise korrektsuse teoreem** esitatakse kujul

- - - P
Π või $P \xrightarrow{\Pi} Q$
- - - $/Q/$

mis mõlemad loetakse järgmise teoreemi lühemaks üleskirjutiseks:

Teoreem. *Kui algoritmi* Π *täitmist alustatakse olekus, mis rahuldab tingimust P, siis algoritmi täitmise* l õ p p e d e s *on saavutatud olek, mis rahuldab tingimust Q.*

Näiteks saab tõestada, et joonisel 8.5 esitatud algoritm on osaliselt korrektne. Osaliselt korrektne algoritm ei anna kunagi nö. vale vastust: kui on alustatud lubatud algolekust, siis, kui algoritmi täitmine peaks lõppema, on kindlasti saavutatud järeltingimust rahuldav olek; kuid mitte igast lubatud algolekust alustades ei pruugi algoritmi täitmine lõppeda.

Algoritmi lõplikkuse teoreem esitatakse kujul

- - - P
Π või $P \xrightarrow{\Pi}|$
- - - |

mis mõlemad loetakse järgmise teoreemi lühemaks üleskirjutiseks:

Teoreem. *Kui algoritmi* Π *täitmist alustatakse olekus, mis rahuldab tingimust P, siis algoritmi täitmiseks kulub lõplik arv operatsioone.*

Tsüklidirektiivi

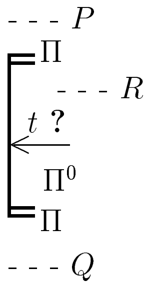
- - - P
Π
t ?
Π^0 ehk $P \xRightarrow{\Pi} Q$
Π
- - - Q

täieliku korrektsuse tõestamiseks näidatakse, et (1) tsükkel on osaliselt

korrektne, st. kehtib osalise korrektsuse teoreem, ja (2) tsükkel on lõplik, st. kehtib lõplikkuse teoreem – tsükkel täidetakse lõpliku ajaga (kui alustatakse olekust, mis rahuldab eeltingimust):

$$\frac{P \stackrel{\Pi}{\longrightarrow} Q, P \stackrel{\Pi}{\longrightarrow}\!|}{P \stackrel{\Pi}{\Longrightarrow} Q} \tag{8.5}$$

Kummagi „alamteoreemi“ tõestamiseks on paratamatult tarvis teada veel tsükli sammu alguses kehtivat lisatingimust – **tsükli invarianti** (R):



Tsükli invariandil on keskne koht tsükli olemuse ja tööpõhimõtte määratlemisel, samuti tsükli korrektsuse tõestamisel. Mõlemad olulised tuletusreeglid (8.6 ja 8.7) eeldavad, et koos tsüklidirektiiviga on antud ka s o b i v tsükli invariant. Viimane on tegelikult tsükli täielikust kirjeldusest lahutamatu, selle automaatne leidmine vaid tsükli spetsifikatsioonist (P, Π, Q) lähtudes ei ole üldiselt võimalik.

Invariandiga R varustatud tsükli osalise korrektsuse aksioom (eeldusel, et lõpetamistingimuse t kontrollimise käigus olek ei muutu):

$$\frac{P \Rightarrow R, \neg t \wedge R \stackrel{\Pi^0}{\longrightarrow} R, t \wedge R \Rightarrow Q}{P \stackrel{\Pi}{\longrightarrow} Q} \tag{8.6}$$

Seega, tsükkel on osaliselt korrektne, kui tsükli invariant R on rahuldatud alustamisolekus ($P \Rightarrow R$), tsükli sisu (Π^0) on osaliselt korrektne eeltingimuse $\neg t \wedge R$ ning järeltingimuse R korral ($\neg t \wedge R \stackrel{\Pi^0}{\longrightarrow} R$) ja lõpetamistingimus t konjunktsioonis invariandiga „tagab“ järeltingimust rahuldava lõppoleku ($t \wedge R \Rightarrow Q$).

Invariandiga R varustatud tsükli lõplikkuse tõestamiseks tuleb leida täisarvuline avaldis e, mille väärtus on positiivne enne tsükli sisu Π^0 iga täitmist

ja kindlasti väheneb sisu täitmise tulemusena. Seega, tsükli lõplikkuse aksioom:

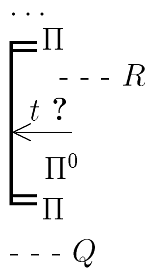
$$\frac{\neg t \wedge R \Rightarrow e > 0, \neg t \wedge R \wedge (0 < e = c) \stackrel{\Pi^0}{\Longrightarrow} (0 \leqslant e < c)}{P \stackrel{\Pi}{\longrightarrow\!\!|}} \tag{8.7}$$

kus e on avaldis ja c on konstant.

Vaadeldava tsüklikonstruktsiooni nõrgimaks eeltingimuseks on invariant R, mille korral on rahuldatud mõlema aksioomi (8.6 ja 8.7) eeldused.

Kui algoritmi üldises alt-üles tõestuskäigus on jõutud tsüklidirektiivini

...



- - - Q

...

siis võib võtta tingimuse R tsükli eeltingimuseks, kui tõestada $R \stackrel{\Pi}{\Longrightarrow} Q$; selleks aga tuleb omakorda näidata järgmise kolme väite kehtivus:

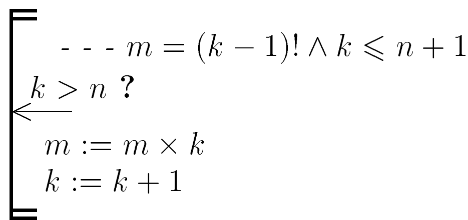
1) $\neg t \wedge R \stackrel{\Pi^0}{\Longrightarrow} R$ (invariantsus);

2) $t \wedge R \Rightarrow Q$ (sihipärasus);

3) $R \stackrel{\Pi}{\longrightarrow\!\!|}$ (lõplikkus).

Teoreem 4

- - - $m = (k-1)! \wedge k \leqslant n+1$



- - - $m = n!$

Tõestus.

$R \Leftrightarrow m = (k-1)! \wedge k \leqslant n+1$,

$Q \Leftrightarrow m = n!$.
$t \Leftrightarrow k > n$.
Π^0:

$m := m \times k$
$k := k + 1$

1) $\neg t \wedge R \stackrel{\Pi^0}{\Longrightarrow} R$ (invariantsus)

Leiame järjestikuselt (alt-üles) R_1 ja R_2, kasutades omistamise aksioomi (8.3):

- - - $R_2 \Leftrightarrow (R_1)^m_{m \times k} \Leftrightarrow m \times k = k! \wedge k \leqslant n \Leftrightarrow m = (k-1)! \wedge k \leqslant n$
$m := m \times k$
- - - $R_1 \Leftrightarrow R^k_{k+1} \Leftrightarrow m = k! \wedge k \leqslant n$
$k := k + 1$
- - - $R \Leftrightarrow m = (k-1)! \wedge k \leqslant n + 1$

Kehtib $\neg t \wedge R \Rightarrow R_2$:
$k \leqslant n \wedge m = (k-1)! \wedge k \leqslant n + 1 \Leftrightarrow k \leqslant n \wedge m = (k-1)! \Leftrightarrow R_2$.

Seega on invariantsus näidatud.

2) $t \wedge R \Rightarrow Q$ (sihipärasus)

Kehtib, sest
$k > n \wedge m = (k-1)! \wedge k \leqslant n+1 \Leftrightarrow m = (k-1)! \wedge k = n+1 \Rightarrow m = n! \Leftrightarrow Q$.

3) $R \stackrel{\Pi}{\longrightarrow}\!|$ (lõplikkus)

Lõplikkuse aksioomi eeldused on rahuldatud, kui avaldiseks e võtta $n - k + 1$.
Esiteks, $\neg t \wedge R \Leftrightarrow k \leqslant n \wedge m = (k-1)! \wedge k \leqslant n+1 \Rightarrow k \leqslant n \Rightarrow k < n+1 \Rightarrow n - k + 1 > 0$.
Teiseks, näitame, et $\neg t \wedge R \wedge (n - k + 1 = c) \stackrel{\Pi^0}{\Longrightarrow} (n - k + 1 < c)$.
Selleks leiame järjestikuselt R_3 ja R_4, kasutades omistamise aksioomi (8.3):

- - - $R_4 \Leftrightarrow n - k < c$
$m := m \times k$
- - - $R_3 \Leftrightarrow n - (k+1) + 1 < c \Leftrightarrow n - k < c$
$k := k + 1$
- - - $n - k + 1 < c$

Kehtib $n - k + 1 = c \Rightarrow R_4$, sest $n - k + 1 = c \Rightarrow n - k < c$.

Oleme invariandi R alusel tõestanud antud tsüklidirektiivi korrektsuse, seega R sobib ka selle tsükli eeltingimuseks.

□

Järeldus. *Mittenegatiivse täisarvu faktoriaali arvutamise algoritm (joonisel 8.3) on täielikult korrektne.*

Tõestus. Selles algoritmis oleva tsükli korrektsus on teoreemis uuritud. Peale selle kehtib

- - - $n \geqslant 0$
- - - P_2
$m := 1$
- - - P_1
$k := 1$
- - - $R \Leftrightarrow m = (k-1)! \wedge k \leqslant n+1$,

sest

$P_1 \Leftrightarrow m = 0! \wedge 1 \leqslant n+1 \Leftrightarrow m = 1 \wedge 0 \leqslant n$, $P_2 \Leftrightarrow 1 = 1 \wedge 0 \leqslant n \Leftrightarrow 0 \leqslant n$ ja $n \geqslant 0 \Rightarrow P_2$.

□

8.3 Näide: kiire astendamise algoritmi korrektsus

Teoreem

- - - $a, m \in \mathbb{N}$

- - - $m \geqslant 0$

$x := a;\ n := m;\ z := 1$

$n = 0$ **?** ←

[? $(paaritu(n))\ z := z \times x$

$n := \lfloor n/2 \rfloor;\ x := x^2$

- - - $z = a^m$

Tõestus.

Antud juhul pole tsükli invariant "kaasa antud". Selle leidmiseks tuleks tegelikult tsükkel üksikasjalikult "lahti mõtestada", et kindlaks teha muutujate väärtuste püsiv vahekord. Osutub, et invariandiks sobib

$R \Leftrightarrow zx^n = a^m \wedge n \geqslant 0.$

Invariandi R alusel tõestame algoritmis esineva tsüklidirektiivi (kus lõpetamistingimuseks t on $n = 0$).

1) $n \neq 0 \wedge R \overset{\Pi^0}{\Longrightarrow} R$ (invariantsus)

Täpsemalt, tuleb tõestada

- - - $n \neq 0 \wedge R$

[? $(paaritu(n))\ z := z \times x$

$n := \lfloor n/2 \rfloor;\ x := x^2$

- - - R

Leiame järjestikuselt $R_1,\ R_2,\ R_3,\ R_4$:

- - - R_4

$paaritu(n)$ **?**
  - - - R_3
  $z := z \times x$

- - - R_2
$n := \lfloor n/2 \rfloor$
- - - R_1
$x := x^2$
- - - $R \Leftrightarrow zx^n = a^m \wedge n \geqslant 0$
Omistamisaksioomi (8.3) põhjal saame $R_1,\ R_2,\ R_3$:
$R_1 \Leftrightarrow z(x^2)^n = a^m \wedge n \geqslant 0 \Leftrightarrow zx^{2n} = a^m \wedge n \geqslant 0,$
$R_2 \Leftrightarrow zx^{2\lfloor n/2 \rfloor} = a^m \wedge \lfloor n/2 \rfloor \geqslant 0,$
$R_3 \Leftrightarrow zxx^{2\lfloor n/2 \rfloor} = a^m \wedge \lfloor n/2 \rfloor \geqslant 0 \Leftrightarrow zx^{2\lfloor n/2 \rfloor+1} = a^m \wedge \lfloor n/2 \rfloor \geqslant 0.$
Tingimus R_4 on ühe poolega tingimusdirektiivi nõrgim eeltingimus:
$R_4 \Leftrightarrow (paaritu(n) \wedge R_3) \vee (\neg paaritu(n) \wedge R_2) \Leftrightarrow (paaritu(n) \wedge zx^{2\lfloor n/2 \rfloor+1} = a^m \wedge \lfloor n/2 \rfloor \geqslant 0) \vee (\neg paaritu(n) \wedge zx^{2\lfloor n/2 \rfloor} = a^m \wedge \lfloor n/2 \rfloor \geqslant 0).$
Kuna paaritu n korral $\lfloor n/2 \rfloor = (n-1)/2$ ja $(n-1)/2 \geqslant 0 \Rightarrow n \geqslant 0$ ning paaris n korral $\lfloor n/2 \rfloor = n/2$ ja $n/2 \geqslant 0 \Rightarrow n \geqslant 0$, siis
$R_4 \Leftrightarrow (zx^n = a^m \wedge n \geqslant 0) \vee (zx^n = a^m \wedge n \geqslant 0) \Leftrightarrow zx^n = a^m \wedge n \geqslant 0.$
Sellega on invariantsus näidatud, sest antud eeltingimusest järeldub nõrgim eeltingimus R_4:
$n \neq 0 \wedge R \Rightarrow n \neq 0 \wedge zx^n = a^m \wedge n \geqslant 0 \Rightarrow zx^n = a^m \wedge n > 0 \Rightarrow zx^n = a^m \wedge n \geqslant 0 \Leftrightarrow R_4.$

2) $t \wedge R \Rightarrow Q$ (sihipärasus)

Kehtib, sest
$t \wedge R \Rightarrow n = 0 \wedge zx^n = a^m \wedge n \geqslant 0 \Rightarrow n = 0 \wedge zx^n = a^m \Rightarrow z = a^m.$

3) $R \xrightarrow{\text{II}}\!\!|$ (lõplikkus)

Lõplikkuse aksioomi eeldused on rahuldatud, kui avaldiseks e võtta n.
Esiteks, $\neg t \wedge R \Leftrightarrow n \neq 0 \wedge zx^n = a^m \wedge n \geqslant 0 \Rightarrow n > 0.$

Teiseks, kehtib
- - - $\neg t \wedge R \wedge n = c$
[? $(paaritu(n))$ $z := z \times x$
$n := \lfloor n/2 \rfloor;\ x := x^2$
- - - $n < c$

Tõepoolest, järeltingimusest $n < c$ lähtudes saame selle algoritmi(osa) nõrgimaks eeltingimuseks $\lfloor n/2 \rfloor < c$ (kuna ainus muutujale n omistamine on $n := \lfloor n/2 \rfloor$). Saadud nõrgim eeltingimus aga järeldub antud eeltingimusest: $n \neq 0 \wedge R \wedge n = c \Rightarrow n \neq 0 \wedge zx^n = a^m \wedge n \geqslant 0 \wedge n = c \Rightarrow n > 0 \wedge n = c \Rightarrow \lfloor n/2 \rfloor < c$.

Kokkuvõttes, teoreemis antud algoritmis esineva tsükli nõrgimaks eeltingimuseks on R.

Tõestuse lõpuleviimiseks jääb üle vaid leida järjestikuselt (alt-üles) tingimused $P_1,\ P_2,\ P_3$:
- - - $P_3 \Leftrightarrow a^m = a^m \wedge m \geqslant 0 \Leftrightarrow m \geqslant 0$
$x := a$
- - - $P_2 \Leftrightarrow x^m = a^m \wedge m \geqslant 0$
$n := m$
- - - $P_1 \Leftrightarrow x^n = a^m \wedge n \geqslant 0$
$z := 1$
- - - $R \Leftrightarrow zx^n = a^m \wedge n \geqslant 0$

ja veenduda, et tingimus P_3 järeldub vaadeldavas algoritmis esitatud eeltingimusest $m \geqslant 0$.

□

Kirjandus

[1] T. H. Cormen, C. E. Leiserson, L. Rivest. *Introduction to Algorithms*, MIT, 1990.

[2] D. E. Knuth. *The Art of Computer Programming*, Addison-Wesley, osad 1-3, kolmas trükk, 1998.

[3] R. Sedgewick. *Algorithms*, Addison-Wesley, 1988.

[4] T. Budd. *Classic Data Structures in C++*, Addison-Wesley, 1994

[5] T. Standish. *Data Structures in Java*, Addison-Wesley, 1998.

[6] NIST. *Dictionary of Algorithms and Data Structures*, 2002 `http://www.nist.gov/dads/` (02.06.2003)

[7] J. Kiho. *Elementaaralgoritmid*, TÜ, Tartu, 1991.

[8] J. Kiho. *Arvutiõpetus II*, TÜ, Tartu, 1989.

Aineregister

O-relatsioon (*O-notation*), 13
Ω-relatsioon (*Ω-notation*), 15
Θ-relatsioon (*Θ-notation*), 14
m-rajaline otsimispuu (*m-way search tree*), 34

abstraktne andmestrktuur (*abstract data structure*), 25
ahne algoritm (*greedy algorithm*), 86
ajaline keerukus (*time complexity*), 11
alampuu (*subtree*), 27
alamsõne otsimine (*substring search*), 75
alluv (*child, proper descendant*), 27
andmestruktuur (*data structure*), 23
andmetüüp (*data type*), 23
asümptootiline hinnang (*asymptotic behavior*), 12
AVL-puu (*AVL-tree*), 33

B-puu (*B-tree*), 35
Bellman-Fordi algoritm, 102
binomiaalkuhi (*binomial heap*), 56
binomiaalpuu (*binomial tree*), 37
Boyer-Moore'i algoritm, 78

Dijkstra algoritm, 98
dünaamiline andmestruktuur (*dynamic data structure*), 24
dünaamiline järjend (*dynamic sequence*), 26
dünaamiline kavandamine (*dynamic programming*), 89

eeldusgraaf (*prerequisite graph*), 93
eelistusjärjekord (*priority queue*), 26
eelkäija (*ancestor*), 27, 40
eesjärjestus (*preorder*), 28
elementaartee (*simple path*), 40
elementaartsükkel (*simple loop*), 40
esinemine (*occurrence*), 75

Floyd-Warshalli algoritm, 102

Galler-Fischeri meetod (*Galler-Fischer representation*), 49
graaf (*graph*), 40
graafi läbimine (*graph traversal*), 106
Grahami seiremeetod (*Graham scan*), 119

Huffmani puu (*Huffman tree*), 84

jaga-ja-valitse (*divide-and-conquer*), 62
juur (*root*), 27

järeltulija (*descendant*), 27, 40
järjekord (*queue*), 26
järjestatud puu (*ordered tree*), 27
järjestikpaigutus (*sequential allocation*), 41

kaar (*edge*), 40
kahekäiguline algoritm (*two-pass algortihm*), 51
kahendkuhi (*binary heap*), 52
kahendotsimise puu (*binary search tree*), 28
kahendpuu (*binary tree*), 27
kaugus (*distance*), 98
keerukus halvimal juhul (*worst-case complexity*), 19
keskjärjestus (*postorder*), 28
keskmine keerukus (*average-case complexity*), 19
kiire astendamine (*fast exponentiation*), 140
kiirmeetod (*quicksort*), 62
kimbumeetod (*bucketsort*), 73
Knuth-Morris-Pratti algoritm, 78
kogukeerukus (*total complexity*), 18
kolleeg (*sibling*), 27
kollisioon (*collision*), 45
kommentaar (*comment*), 9
kompaktne kahendpuu (*compact binary tree*), 28
koodipuu (*prefix code tree*), 83
korrektsuse teoreem (*correctness theorem*), 127
kriitiline tee (*critical path*), 96
Kruskali algoritm, 109
kuhi (*heap*), 52
kuhjameetod (*heapsort*), 55
kuhjaomadus (*heap property*), 52
kuhjastama (*heapify*), 54
kumer kate (*convex hull*), 118
kuuluvus hulknurka (*inclusion in a polygon*), 114

lahtine adresseerimine (*open addressing*), 46
laiuti läbimine (*breadth-first traversal*), 106
leht (*leaf*), 27
lihtahel (*linked list*), 42
lihtskeem (*simple sketch*), 7
lisamine (*insertion*), 24
loendamismeetod (*counting sort*), 72
lõikude lõikumine (*line segment intersection*), 114
lõppjärjestus (*endorder*), 28
lähim paar (*closest pair*), 120
lähtepunkt (*anchor point*), 116
lähtetipp (*start node*), 98
lühim tee (*shortest path*), 98

magasin (*stack*), 26
mask (*mask, bitmap*), 16
mets (*forest*), 27
minimaalne toes (*minimal spanning tree*), 109
moodulskeem (*module sketch*), 8
muutuv eelistus (*changing priority*), 26

naaber (*neighbour*), 40

naabrusmaatriks (*adjacency matrix*), 102
naturaalesitus (*natural representation*), 42

olek (*state*), 127
olekute ruum (*state space*), 125
operatsioonivaru (*set of operations*), 23
orienteerimata graaf (*undirected graph*), 40
orienteeritud graaf (*directed graph, digraph*), 40
osaline korrektsus (*partial correctness*), 135
osasõne (*substring*), 87
otsetee (*direct path*), 106
otsisõne (*search string, pattern*), 75
otsustuste puu (*decision tree*), 70

paiskfunktsioon (*hash function*), 45
paisksalvestus (*hash coding*), 45
paisktabel (*hash table*), 45
pikim ühissõne (*longest common substring*), 87
pistemeetod (*insertion sort*), 60
polünomiaalne keerukus (*polynomial complexity*), 15
positsioonimeetod (*radix sort*), 72
prefiksfunktsioon (*prefix-function*), 78
prefikskood (*prefix code*), 83
Primi algoritm, 110
pseodo-tõusunurk, 113
punkt hulknurgas (*point in polygon*), 114
puu (*tree*), 27
põhioperatsioon (*characteristic operation*), 18
põhiteoreem (*master theorem*), 20
põimemeetod (*mergesort*), 66
põimimine (*merging*), 66
põrge (*collision*), 45
päripäeva (*clockwise*), 113
päristee (*proper path*), 40
päristsükkel (*proper loop*), 40
pöörde suund, 113
pöördkuhi (*inverted heap*), 52

Rabin- Karpi algoritm, 80
raskelt lahenduv (*intractable*), 15

saavutatav tipp (*reachable node*), 40
seljakotiülesanne (*knapsack problem*), 16
seotud paigutus (*linked allocation*), 41
serv (*edge*), 40
sisendaste (*in-degree*), 40
stabiilne sorteerimismeetod (*stable sorting*), 59
struktuurne tüüp (*structured type*), 23
suluesitus (*parenthesis notation*), 43
suurima vahemaaga punktid (*farthest points*), 119
sõlm (*internal node*), 27
sõne (*string*), 75

sõnetöötlus (*string processing*), 75
sügavuti läbimine (*depth-first traversal*), 106

tagasisidestatud (*threaded*), 42
tasakaalustatud (*balanced*), 31
tee (*path*), 40
tee õgvendamine (*path compression*), 51
teepikkus (*path length*), 98
tekst (*text*), 75
teksti pakkimine (*text compression*), 83
tipp (*node, vertex*), 40
toes (*spanning tree*), 109
toesemets (*spanning forest*), 109
toesepuu (*spanning tree*), 109
topeltpaiskamine (*double hashing*), 47
topoloogiline järjestus (*topological order*), 93
topoloogiline sorteerimine (*topological sort*), 93
traageldatud (*threaded*), 42
tsükkel (*loop*), 40
tsükli invariant (*loop invariant*), 136
tsükliskeem (*loop sketch*), 8
tähestik (*alphabet*), 75
täielik kahendpuu (*complete binary tree*), 28
täielik korrektsus (*complete correctness*), 135
tühisõne (*empty string*), 75

vahetipp (*internal node*), 27
vastupäeva (*counterclockwise*), 113
võrkgraafik (*prerequisite graph*), 93
võtmine (*extraction*), 24
vähima vahemaaga punktid (*closest points*), 120
välisahelate meetod (*external chaining*), 45
väljundaste (*out-degree*), 40
väärtusvaru (*domain, set of values*), 23

üherealine skeem (*one-line sketch*), 9
ühildamine (*merging*), 66
ühildusmeetod (*mergesort*), 66
ühine osasõne (*common substring*), 87
ühissõne (*common substring*), 87
ülemus (*parent, ancestor*), 27