TARTU ÜLIKOOL

TEADUSKOOL

PROGRAMMEERIMISE ALUSED

MITTELINEAARSED ANDMESTRUKTUURID

Õppevahend TK õpilastele

Ahto Truu

Tartu 2007

Peatükk 8

Mittelineaarsed andmestruktuurid

Sisukord

8.1 Põhilised mittelineaarsed struktuurid 3
8.1.1 Graaf 3
8.1.2 Puu 5
8.2 Puualgoritmid 7
8.2.1 Kahendpuu esitamine arvuti mälus 7
8.2.2 Kahendpuu läbimine 8
8.2.3 Kahendotsingu puu 10
8.2.4 Kahendkuhi 13
8.3 Graafialgoritmid 16
8.3.1 Graafi esitamine arvuti mälus 16
8.3.2 Graafi läbimine laiuti 18
8.3.3 Lühima tee leidmine 19
8.3.4 Graafi läbimine sügavuti 20
8.3.5 Topoloogiline sorteerimine 21

© 2001–2007, Ahto Truu & Tartu Ülikool
Käesolevat õppevahendit võib algallikale viidates kasutada ja levitada mistahes viisil ja eesmärkidel.

Õppevahend ilmub Tiigrihüppe SA toetusel.

Käesolevas peatükis jätkame tähtsamate andmestruktuuride ja nendega seotud algoritmide uurimist. Seekord võtame vaatluse alla objektid, mille sisemine struktuur on keerulisem kui lihtsalt nende elementide loend, ja mida seetõttu nimetatakse “mittelineaarseteks”.

8.1 Põhilised mittelineaarsed struktuurid

Tähtsaim mittelineaarne andmestruktuur on graaf. Arvutiteaduses on väga vähe ülesandeid, mis poleks ühel või teisel moel esitatavad graafiülesannetena. Graafi tähtsaim erijuht on puu. Nende kahega järgnevalt tutvumegi.

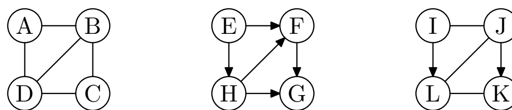
8.1.1 Graaf

Graaf (ingl *graph*) on matemaatiline objekt, mis koosneb mingist hulgast **tippudest** (ingl *vertex*) ja neid tippe paarikaupa ühendavatest **servadest** (ingl *edge*). Asjaolu, et graafi G tippude hulk on V ja servade hulk E, märgitakse tavaliselt $G = (V, E)$.

Kui graafi tippe u ja v ühendaval serval ei ole suunda määratud (seda serva pidi võib liikuda nii tipust u tippu v kui ka vastupidi), siis nimetatakse seda orienteerimata ehk suunamata servaks ehk lühemalt lihtsalt servaks ja tähistatakse $e = \{u, v\}$.

Kui aga serval on suund määratud, nimetatakse seda orienteeritud ehk suunatud servaks ehk **kaareks** (ingl *arc*) ja tähistatakse $e = (u, v)$.

Joonisel kujutatakse graafi tippe tavaliselt punktide, sõõride või kastikestena. Suunamata servi kujutatakse vastavaid tippe ühendavate joontena ja suunatud servi nooltena.



Joonis 8.1: Graafide näiteid

Graafi, mille kõik servad on suunamata, nimetatakse **suunamata graafiks** (ingl *undirected graph*) ehk lühemalt lihtsalt graafiks. Graafi, mille kõik servad on suunatud, nimetatakse **suunatud graafiks** (ingl *directed graph*). Graafi, mille osa servi on suunatud ja osa suunamata, nimetatakse **segagraafiks**. Näiteks joonisel 8.1 toodud graafidest on vasakpoolne suunamata, keskmine suunatud ja parempoolne segagraaf.

Teeks (ingl *path*) graafis $G = (V, E)$ nimetatakse "järjestikuste" servade jada $e_1 = \{v_0, v_1\}$, $e_2 = \{v_1, v_2\}$, ..., $e_n = \{v_{n-1}, v_n\}$, milles iga serva lõpptipp on talle järgneva serva algustipp.

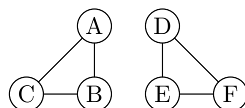
Ahelaks (ingl *chain*) nimetatakse teed, mis ei läbi ühtki serva korduvalt ja **lihtahelaks** (ingl *simple chain*) ahelat, mis ei läbi korduvalt ühtki tippu. **Tsükliks** (ingl *cycle*) nimetatakse ahelat ja **lihttsükliks** (ingl *simple cycle*) lihtahelat, mille algus- ja lõpptipp langevad kokku ($v_0 = v_n$). **Silmuseks** (ingl *loop*) nimetatakse ühest servast koosnevat tsüklit.[1]

Graafi $G = (V, E)$ **alamgraafiks** (ingl *subgraph*) nimetatakse mistahes graafi $G' = (V', E')$, mille korral kehtib $V' \subseteq V$ ja $E' \subseteq E$. See tähendab, et alamgraaf on saadav ülemgraafist mingi hulga tippude ja servade eemaldamise teel. Asjaolu, et G' on G alamgraaf, tähistatakse $G' \subseteq G$.

Graafi G alamgraafi G', mis ei ole võrdne graafi G endaga, nimetatakse tema **pärisalamgraafiks** (ingl *proper subgraph*) ja seda tähistatakse $G' \subset G$.

Tähtsal kohal on graafiteoorias sidususe mõiste. Suunamata graafi nimetatakse **sidusaks** (ingl *connected*), kui selle mistahes tipupaari v_1, v_2 jaoks leidub tee tipust v_1 tippu v_2.

Graafi G alamgraafi G' nimetatakse graafi G **sidususkomponendiks** (ingl *connected component*), kui G' on sidus ja graafis G ei leidu sidusat alamgraafi G'', mis sisaldaks graafi G' pärisalamgraafina. See tähendab, et sidususkomponent on maksimaalne sidus alamgraaf, seda ei saa kasvatada talle uute servade või tippude lisamisega ilma sidusust rikkumata.

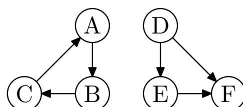


Joonis 8.2: Suunamata graafi sidususkomponendid

Näiteks joonisel 8.2 toodud graafis on kaks sidususkomponenti: tipud A, B, C ja kõik nendega seotud servad ning tipud D, E, F ja kõik nendega seotud servad.

[1] Teede, ahelate ja tsüklitega seonduv terminoloogia ei ole graafiteoorias väga hästi standardiseeritud, seetõttu tuleb enne põhjalikumate järelduste tegemist uurida, kuidas konkreetse raamatu autor need terminid enda jaoks defineerinud on. Näiteks kasutatakse mõnes ingliskeelses raamatus sõna 'path' meie 'ahela' tähenduses ja meie 'tee' jaoks sel juhul otsest vastet polegi. Mingil määral kehtib sama ka graafi enda mõiste kohta. Näiteks nimetatakse paljudes raamatutes graafideks ainult selliseid, milles mistahes kahe tipu vahel võib olla ülimalt üks serv ja silmused pole üldse lubatud. Programmeerimises tavaliselt graafidele selliseid piiranguid ei seata.

Suunatud või segagraafi G nimetatakse sidusaks, kui graafi G kõigi kaarte (v_1, v_2) suunamata servadega $\{v_1, v_2\}$ asendamisel saadud suunamata graaf G' on sidus, ja **tugevalt sidusaks** (ingl *strongly connected*), kui G mistahes tipupaari v_1 ja v_2 jaoks leidub nii tee tipust v_1 tippu v_2 kui ka tee tipust v_2 tippu v_1. Analoogiliselt võime eristada ka sidususkomponente ja **tugevalt sidusaid komponente** (ingl *strongly connected component*).[2]



Joonis 8.3: Suunatud graafi sidususkomponendid

Näiteks joonisel 8.3 toodud suunatud graafis on kaks sidususkomponenti: tipud A, B, C ja kõik nendega seotud kaared ning tipud D, E, F ja kõik nendega seotud kaared. (Kaarte asendamisel servadega saame sama graafi, mis on toodud joonisel 8.2.) Tugevalt sidusaid komponente on selles graafis aga tervelt neli: tipud A, B, C ja kõik nendega seotud kaared ning tipud D, E ja F igaüks eraldi. Tippe D, E ja F ühendavad kaared ei kuulu ühegi tugevalt sidusa komponendi koosseisu.

8.1.2 Puu

Puu (ingl *tree*) on sidus tsükliteta graaf.

Tegemist on mõnes mõttes piirjuhtumiga: puu on ühelt poolt minimaalne sidus graaf (ükskõik millise serva eemaldamine muudab puu mittesidusaks) ja teiselt poolt maksimaalne tsükliteta graaf (ükskõik millise uue serva lisamine tekitab tsükli).

Esimese väite kehtivuses veendumiseks vaatleme puu serva $e = \{u, v\}$. Oletame, et pärast selle serva eemaldamist on graaf endiselt sidus. See aga tähendab, et leidub mingi tee tipust u tippu v, mis ei läbi serva e. Lisades nüüd sellele teele serva e, saame tsükli. See on muidugi vastuolus eeldusega, et esialgne graaf oli tsükliteta, mistõttu peame järeldama, et sellist teed tipust u tippu v ei saa leiduda. Kuna arutelu ei sõltu serva e valikust, tähendab see, et puu muutub mittesidusaks ükskõik millise serva eemaldamisel.

Teise väite põhjendamine on sarnane: vaatleme puu tippe u ja v. Kuna puu on sidus, siis peab leiduma tee tipust u tippu v. See aga tähendab, et

[2]Peaks olema ilmne, et suunamata graafis langevad sidususe ja tugeva sidususe mõisted kokku, mistõttu sel juhul pole tugeva sidususe ja tugevalt sidusa komponendi mõiste järele vajadust.

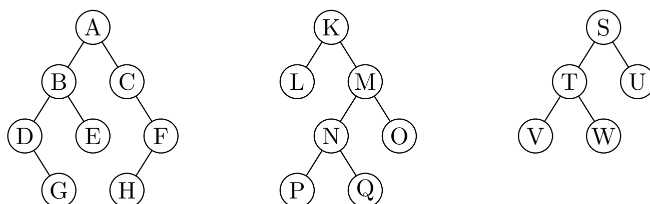
tippude u ja v vahele uue serva lisamine tekitaks tsükli. Kuna arutelu ei sõltu tippude u ja v valikust, tähendab see, et puusse tekib tsükkel mistahes serva lisamisel.

Programmeerimise seisukohalt pakuvad rohkem huvi **juurega puud** (ingl *rooted tree*). Juurega puu erineb tavalisest selle poolest, et üht selle tippu nimetatakse **juureks** (ingl *root*). Joonisel märgitakse juurtipp mõnikord rasvaselt, aga sagedamini paigutatakse tipud nii, et juurtipp on teistest kõrgemal, nagu näiteks tipud A, K ja S joonisel 8.4 toodud puudes.

Juurtipu esiletõstmine muudab puu hierarhiaks. Iga serva otstippudest on üks juurele lähemal kui teine. Juurele lähemat tippu nimetatakse teise **ülemuseks** (ingl *parent*), kaugemat tippu omakorda tema **alluvaks** (ingl *child*). Näiteks joonisel 8.4 vasakul toodud puus on tipp B tipu A alluv ja samal ajal tippude D ja E ülemus.

Puu tippu, millel on mõni alluv, nimetatakse **sõlmeks** (ingl *inner node*), alluvateta tippu aga **leheks** (ingl *leaf*). Näiteks joonisel 8.4 keskel kujutatud puus on K, M ja N sõlmed, aga L, O, P ja Q lehed.

Puu tippu koos tema kõigi (nii vahetute kui ka kaugemate) alluvatega nimetatakse **alampuuks** (ingl *subtree*). Näiteks joonisel 8.4 parempoolses puus moodustavad alampuu tipud T, V ja W. Muidugi on alampuu ka iga leht eraldi ja kogu puu tervikuna.



Joonis 8.4: Kahendpuude näiteid

Puud, mille igal sõlmel on maksimaalselt n alluvat, nimetatakse n-puuks. Näiteks kõik joonisel 8.4 kujutatud puud on **kahendpuud** (ingl *binary tree*). Puud, mille kõigil sõlmedel on sama arv alluvaid, nimetatakse **homogeenseks**. Näiteks joonisel 8.4 vasakul toodud puu ei ole homogeenne (tippudel C, D ja F on igaühel ainult üks alluv), aga keskmine ja parempoolne on.

Homogeenset puud, mille kõik lehed on juurest võrdsel kaugusel, nimetatakse **täielikuks**. Joonisel 8.4 kujutatud puudest pole ükski täielik, kuid parempoolset nimetatakse **kompaktseks** — tal puuduvad täielikkusest vaid mõned lehed alumise tippuderea lõpust.

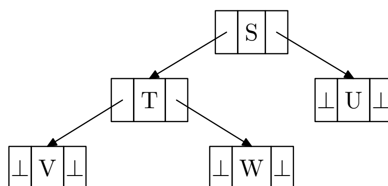
Kahendpuus on sageli kasulik eristatada, kas alluv on oma ülemuse vasak

või parem alluv, isegi juhul kui tegu on ainsa alluvaga. Kahendpuu joonistamisel paigutatakse vasak alluv oma ülemusest madalamale ja vasakule, parem alluv aga vastavalt madalamale ja paremale. Seega on joonisel 8.4 vasakul kujutatud puus tipp F tipu C parem ja tipp H omakorda tipu F vasak alluv.

8.2 Puualgoritmid

8.2.1 Kahendpuu esitamine arvuti mälus

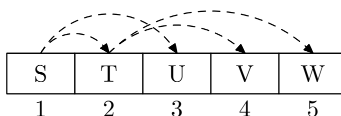
Kahendpuud on loomulik hoida rekursiivse struktuurina, kus iga tippu esitab kirje, milles on väljad selle tipu andmetega ja lisaks viidad selle tipu vasakule ja paremale alluvale. Kui tipul üht või teist (või ka mõlemat) alluvat pole, siis on vastavas väljas tühiviit. Kahendpuu loomuliku esituse näide on toodud joonisel 8.5.



Joonis 8.5: Kahendpuu loomulik esitus

Teine võimalus on hoida kahendpuud massiivis, nagu näidatud joonisel 8.6: puu juur on massiivi elemendis a_1; elemendis a_i hoitava tipu alluvad on elementides a_{2i} ja a_{2i+1} ning tema ülemus elemendis $a_{\lfloor i/2 \rfloor}$.[3]

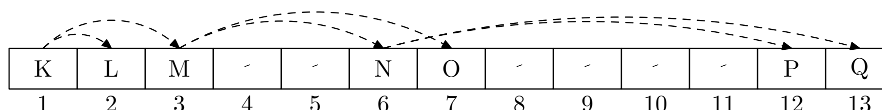
Massiiviesitus sobib hästi täielike ja kompaktsete kahendpuude hoidmiseks, sest siis täidab n-tipuline puu parajasti massiivi elemendid $a_{1...n}$.



Joonis 8.6: Kompaktse kahendpuu massiiviesitus

Nagu näha ka jooniselt 8.7, jääks mittetäieliku puu sellisel esitamisel osa massiivi elemente kasutamata. Esiteks oleks see muidugi asjatu mälukulu.

[3]Kirjutis $\lfloor i/2 \rfloor$ tähendab, et jagamisel tuleb võtta jagatise täisosa.



Joonis 8.7: Mittetäieliku kahendpuu massiiviesitus

Teiseks peaks me siis leidma mingi võimaluse neid tühje elemente tähistada, et me neid hiljem puu töötlemisel kogemata kasutada ei püüaks. Kuna joonisel näidatud nooled on vaid mõttelised seosed, mitte viidad, ei saa me nende puhul tühiviidaga võrdlemist kasutada!

8.2.2 Kahendpuu läbimine

Paljude puuülesannete lahendamisel on vaja puu tippe läbida mingis kindlas järjekorras. Vaatleme näiteks sellist ülesannet: antud puu, mille lehtedes on arvud; vaja on saada igasse sõlmtippu sellest algavas alampuus olevate lehtede arvude summa.

Muidugi saame igas sõlmes vajaliku summa leida tema vahetutes alluvates olevate arvude summana, aga seda ainult eeldusel, et alluvates on vajalikud summad juba välja arvutatud. See tähendab, et nõutud summade efektiivseks arvutamiseks tuleb iga tipu alluvad töödelda enne seda tippu ennast.

Puu niisugust läbimist nimetatakse **lõppjärjestuses läbimiseks** (ingl *post-order traversal*) ja selle üldkuju kahendpuude jaoks kirjeldab algoritm 8.1, mille täiendamine summaülesande lahenduseks oleks muidugi triviaalne.

Algoritm 8.1 LabiPuuLJ: Kahendpuu läbimine lõppjärjestuses
Sisend: t — viit puu juurele, kus $[\![t]\!].v$ ja $[\![t]\!].p$ on vasak ja parem alampuu

1. `kui` $t \neq \perp$ — tühja puu korral pole midagi läbida
2. LabiPuuLJ($[\![t]\!].v$) — töötleme vasaku alampuu
3. LabiPuuLJ($[\![t]\!].p$) — töötleme parema alampuu
4. — siin töötleme tipu $[\![t]\!]$
5. `lõppkui`

Kui summaülesandes liikus informatsioon puu läbimisel "alt üles" ja selle lahendamiseks oli vaja alluvad töödelda enne ülemust, siis paljudes teistes ülesannetes liigub informatsioon vastupidi, "ülalt alla".

Seda tüüpi ülesannete efektiivseks lahendamiseks on sageli mugavaim viis algoritm 8.2, puu läbimine **eesjärjestuses** (ingl *pre-order*).

Algoritm 8.2 LabiPuuEJ: Kahendpuu läbimine eesjärjestuses
Sisend: t — viit puu juurele, kus $[\![t]\!].v$ ja $[\![t]\!].p$ on vasak ja parem alampuu

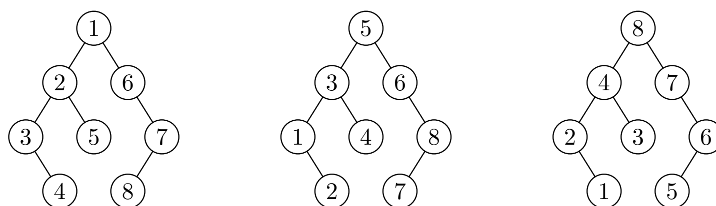
1. `kui` $t \neq \bot$ — tühja puu korral pole midagi läbida
2. — siin töötleme tipu $[\![t]\!]$
3. LabiPuuEJ($[\![t]\!].v$) — töötleme vasaku alampuu
4. LabiPuuEJ($[\![t]\!].p$) — töötleme parema alampuu
5. `lõppkui`

Kolmandat klassikalist tehnikat kujutav algoritm 8.3 läbib kahendpuu tipud **keskjärjestus** (ingl *in-order*): kõigepealt vasaku alampuu, siis juurtipu ja lõpuks parema alampuu. Erinevalt kahest eelmisest on see meetod loomulikul viisil kasutatav ainult kahendpuude korral, sest üldisemal juhul pole selge, millise kahe alluva vahel peaks töötlema alampuu juurtipu.

Algoritm 8.3 LabiPuuKJ: Kahendpuu läbimine keskjärjestuses
Sisend: t — viit puu juurele, kus $[\![t]\!].v$ ja $[\![t]\!].p$ on vasak ja parem alampuu

1. `kui` $t \neq \bot$ — tühja puu korral pole midagi läbida
2. LabiPuuKJ($[\![t]\!].v$) — töötleme vasaku alampuu
3. — siin töötleme tipu $[\![t]\!]$
4. LabiPuuKJ($[\![t]\!].p$) — töötleme parema alampuu
5. `lõppkui`

Joonis 8.8 illustreerib ühe kahendpuu läbimist nende kolme algoritmi poolt: joonisel vasakul, keskel ja paremal on puu tipud nummerdatud vastavalt nende ees-, kesk- ja lõppjärjestuses läbimise järjekorras.



Joonis 8.8: Kahendpuu kolm läbimist

8.2.3 Kahendotsingu puu

Kahendpuud T nimetatakse **kahendotsingu puuks** (ingl *binary search tree*), ehk lühemalt otsingupuuks, kui (a) T on tühi või (b.1) vasaku alampuu kõigi tippude väärtused on juurtipu omast väiksemad ja (b.2) parema alampuu kõigi tippude väärtused juurtipu omast suuremad ja (b.3) kumbki alampuu on ka ise otsingupuu.[4]

Mingi väärtuse otsimine otsingupuus (algoritm 8.4) on sarnane kahendotsinguga järjestatud massiivis: igal sammul võrdleme otsitavat väärtust puu juurtipus olevaga; kui puu tipus olev väärtus on otsitavast väiksem, võime lisaks juurtipule edasise vaatluse alt välja jätta ka kogu vasaku alampuu, see tähendab jätkata otsimist paremas alampuus; kui puu juurtipus olev väärtus on otsitavast suurem, võime otsimist jätkata vasakus alampuus.

Algoritm 8.4 OtsiPuus: Otsimine otsingupuus
Sisend: t — viit otsingupuu juurele; x — otsitav väärtus
Väljund: Tagastab leitud kirje, või $\perp$, kui x puus ei esine

1. `kui` $t = \perp$
2. `tagasta` $\perp$ — tühjas puus ei leia midagi
3. `muidukui` $x < [\![t]\!].x$
4. `tagasta` OtsiPuus($[\![t]\!].v, x$)
5. `muidukui` $x > [\![t]\!].x$
6. `tagasta` OtsiPuus($[\![t]\!].p, x$)
7. `muidu`
8. `tagasta` t — leidsime otsitava
9. `lõppkui`

Elemendi lisamine otsingupuusse (algoritm 8.4) on otsimisega üsna sarnane. See pole ka kuigi üllatav, sest algoritmi põhiline iva on uuele elemendile sobiva koha leidmine. Selle käigus saame ilma lisatööta tuvastada ka juhu, kui lisatav väärtus puus juba esineb.

[4] See definitsioon eeldab, et puus ei ole korduvaid väärtusi. Sellest eeldusest loobumisel peame täpsustama, kuidas me korduvaid väärtusi hoida tahame, ja muidugi ka järgnevaid algoritme vastavalt täiendama.

Algoritm 8.5 LISAPUUSSE: Lisamine otsingupuusse
Sisend: t — viit otsingupuu juurele; x — lisatav väärtus
Väljund: Lisab väärtuse x puusse, kui see seal juba ei esine

```
1. kui t = ⊥
2.     tee-uus t   — tühi puu, lisame uue lehe
3.     [[t]].x ← x
4.     [[t]].v ← ⊥; [[t]].p ← ⊥
5. muidukui x < [[t]].x
6.     LisaPuusse([[t]].v, x)
7. muidukui x > [[t]].x
8.     LisaPuusse([[t]].p, x)
9. muidu
       — väärtus juba on puus, pole midagi teha
10. lõppkui
```

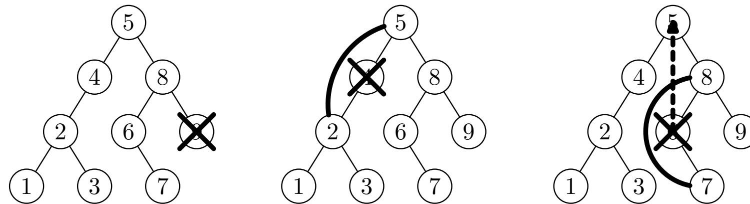
Pole raske näha, et mõlema algoritmi keerukus on võrdeline leitud või lisatud tipu kaugusega puu juurtipust — just nii palju kordi tuleb kummaski algoritmis teha rekursiivseid pöördumisi iseenda poole. Parimal juhul, kui töödeldav puu on täielik kahendpuu, on k-kihilises puus kokku $n = 2^k - 1$ tippu ja n-elemendilise andmehulga töötlemiseks kuluvad $O(k) = O(\log n)$ sammu on sel juhul kooskõlas juba varem viidatud kahendotsimise analoogiaga. Paraku kehtib see analoogia ainult selliste puude korral, mille iga tipu vasak ja parem alampuu on ligikaudu sama suured (sellisel juhul ütleme, et puu on tasakaalus).

Ilma spetsiaalseid pingutusi tegemata ei saa me üldjuhul eeldada, et meie otsingupuud tasakaalus püsivad. Näiteks elementide kasvavas või kahanevas järjekorras lisamisel saame tasakaalustatud puu asemel hoopis ahela, milles nii otsimine kui ka lisamine taanduvad lineaarseks ja võtavad $O(\log n)$ asemel $O(n)$ operatsiooni.

Selleks, et otsingupuud töö käigus jooksvalt tasakaalus hoida (ja seda piisavalt efektiivselt teha), on välja mõeldud mitmeid kavalaid võtteid. Tuntumad neist on **AVL-puud** (ingl *AVL-tree*) ja **puna-mustad puud** (ingl *red-black tree*). Nende algoritmide detailne kirjeldus on aga käesoleva materjali jaoks natuke liiga pikk.

Antud väärtuse x eemaldamine otsingupuust on veidi keerulisem kui otsimine või lisamine. Kui me oleme eemaldatavat väärtust sisaldava tipu puust üles leidnud, on põhimõtteliselt kolm võimalust:

- x asub lehes (näiteks $x = 9$, joonisel 8.9 vasakul); siis võime selle lehe lihtsalt puust eemaldada;
- x asub ühe alluvaga sõlmes ($x = 4$, joonisel keskel); siis võime sõlme puust eemaldada ja tema ainsa alluva tema asemele panna (on lihtne veenduda, et puu jääb endiselt otsingupuuks);
- x asub kahe alluvaga sõlmes ($x = 5$, joonisel paremal); siis ei saa me seda sõlme puust lihtsalt eemaldada, sest kahe vabaks jääva alluva hoidmiseks pole eemaldatava tipu ülemusel vabu viitasid; selle asemel leiame puust minimaalse eemaldatavast väärtusest suurema väärtuse x' (näites $x' = 6$), asendame eemaldatava väärtuse x tema asukohas väärtusega x' ja eemaldame selle asemel x' tema endisest asukohast (on lihtne veenduda, et puu jääb endiselt otsingupuuks; lisaks on lihtne veenduda, et x' ei saa asuda kahe alluvaga sõlmes, seega tema kustutamiseks pole enam vaja suuruselt järgmist elementi otsida; algoritmi õigsuse seisukohalt pole viimane ajaolu oluline).



Joonis 8.9: Elemendi eemaldamine otsingupuust

Algoritm 8.6 EEMALDAPUUST: Eemaldamine otsingupuust
Sisend: t — viit otsingupuu juurele; x — eemaldatav väärtus
Väljund: Eemaldab väärtuse x puust, kui see seal esineb

```
 1. kui t = ⊥
        — tühi puu, pole midagi teha
 2. muidukui x < ⟦t⟧.x
 3.     EemaldaPuust(⟦t⟧.v, x)
 4. muidukui x > ⟦t⟧.x
 5.     EemaldaPuust(⟦t⟧.p, x)
 6. muidu   — vaja on eemaldada tipp t
 7.     kui ⟦t⟧.p = ⊥
 8.         t' ← t; t ← ⟦t⟧.v   — haagime t puust lahti
 9.         kustuta t'   — ja kustutame mälust
10.     muidu
11.         ⟦t⟧.x ← Minimaalne(⟦t⟧.p)   — asendame naaberväärtusega
12.         EemaldaPuust(⟦t⟧.p, ⟦t⟧.x)   — ja eemaldame vana eksemplari
13.     lõppkui
14. lõppkui
```

Algoritm 8.7 MINIMAALNE: Leiab otsingupuust minimaalse väärtuse
Sisend: t — viit otsingupuu juurele, ei või olla $\bot$

```
1. kui ⟦t⟧.v = ⊥
2.     tagasta ⟦t⟧.x
3. muidu
4.     tagasta Minimaalne(⟦t⟧.v)
5. lõppkui
```

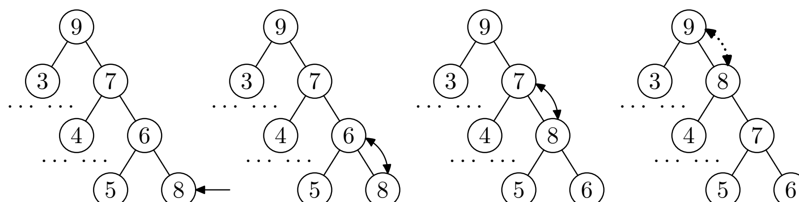
Teema lõpetuseks võiks veel märkida, et otsingupuu läbimisel keskjärjestuses külastame selle tippe nende väärtuste kasvamise järjekorras. See omadus järeldub üsna lihtsalt otsingupuu definitsioonist.

8.2.4 Kahendkuhi

Puud nimetatakse **kuhjaks** (ingl *heap*), kui tema ühegi tipu väärtus ei ületa selle tipu vahetu ülemuse väärtust. Kahendpuud, mis on samal ajal ka kuhi, nimetatakse muidugi kahendkuhjaks.

Vahetult kuhja definitsioonist järeldub, et mistahes kuhjas peab maksimaalse väärtusega element asuma juurtipus. See tähelepanek ongi aluseks, et konstrueerida kahendkuhjal põhinev andmestruktuur, millesse saab efektiivselt andmeid juurde lisada ja suvalisel hetkel kiiresti leida parasjagu kuhjas olevate elementide maksimumi.

Kuhja selliseks kasutamiseks on otstarbekas hoida kuhja elemente kompaktse kahendpuu massiiviesitusena. Uue elemendi lisamisel kuhja lisame selle kõigepealt massiivi lõppu (mis selle massiivi puuna tõlgendamise terminites tähendab, et me lisame puule uue lehe; joonisel 8.10 vasakul). Muidugi võib juhtuda, et uus leht ei rahulda kuhja tingimust.



Joonis 8.10: Elemendi lisamine kahendkuhja

Kuhja taastamiseks hakkame lisatud elementi puus ülespoole nihutama. Kui uus leht on oma ülemusest suurem, siis võime need kaks lelementi lihtsalt omavahel vahetada — siis on kuhja omadus vähemalt selles paaris taas rahuldatud. Kui vastlisatud element on suurem ka oma uuest ülemusest, vahetame ta veelkord ülespoole — nii jätkame, kuni ühel hetkel jõuame piisavalt suure ülemuseni või kuni uus element jõuab puu juurtippu.

Ei tohiks olla raske veenduda, et kui esialgne puu oli kuhi, siis on seda ka uue elemendi üles viimisel saadud puu.

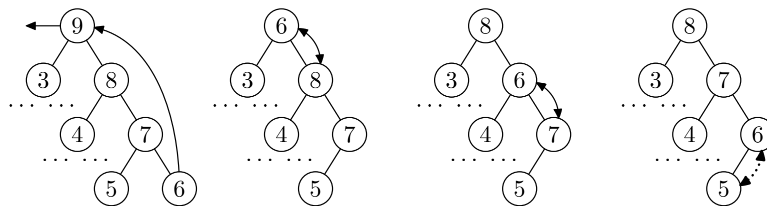
Algoritm 8.8 ViiÜles: Viib kahendkuhja lisatud lehe selle õigele kohale
Sisend: $a_{1\ldots n}$ — puu massiiviesitus, kus $a_{1\ldots n-1}$ moodustavad kuhja
Väljund: a_n viidud kuhjas oma õigele kohale

1. $i \leftarrow n$
2. `senikui` $i > 1$
   — invariant: $a_{1..n}$ miinus a_i alampuu on kuhi $\wedge$ a_i alampuu on kuhi
3. $k \leftarrow \lfloor i/2 \rfloor$
4. `kui` $a_k < a_i$
5. $a_i \leftrightarrow a_k$ — a_i on oma ülemusest suurem, tõrjub selle allapoole
6. `lõppkui`
7. $i \leftarrow k$
8. `lõppsenikui`

Kuhjast maksimaalse elemendi väljavõtmisel tekib samasugune probleem nagu otsingupuust elemendi eemaldamisel — me ei saa eemaldatud elemendi kohta tühjaks jätta. Esimene mõte — tuua juurtipu uueks väärtuseks tema

kahest alluvast suurem — ei ole pikas perspektiivis hea lahendus, sest siis võib uus vaba koht puu lehtede hulgas tekkida ükskõik kuhu ja järgmisel lisamisel on raske seda leida.

Selleks, et vaba koht jääks kindlasti puu massiiviesituse lõppu, toome puu juurtipu uueks väärtuseks justnimelt massiivi viimase elemendi (ja selle koht omakorda jääb vabaks; joonisel 8.11 vasakul). Kuna see on puu leht ja seega üsna väikese väärtusega, on tõenäoline, et sellega saab jälle puu kuhjaomadus rikutud.



Joonis 8.11: Elemendi eemaldamine kahendkuhjast

Kuhja taastamiseks hakkame nüüd vastset juurtipu kuhjas allapoole viima: igal sammul valime uue elemendi ja tema alluvate hulgast maksimaalse ning tõstame selle lokaalseks ülemuseks. Nagu elemendi üles viimisel, järgneme ka seekord vahetuse korral uuele elemendile ja jätkame võrdlemist ja vahetamist, kuni puu on taas kuhi.

Algoritm 8.9 ViiAlla: Viib kahendkuhja lisatud juure selle õigele kohale
Sisend: $a_{1...n}$ — puu massiiviesitus, kus $a_{2...n}$ moodustavad kaks kuhja
Väljund: a_1 viidud kuhjas oma õigele kohale

1. $i \leftarrow 1$
2. `senikui` $i < n$
   — invariant: $a_{1...i}$ on kuhi $\wedge$ a_i alampuu oleks kuhi, kui a_i oleks ∞
3. $\quad k \leftarrow i$
4. $\quad$ `kui` $2 \cdot i \leq n \wedge a_{2 \cdot i} > a_k$
5. $\quad\quad k \leftarrow 2 \cdot i$
6. $\quad$ `lõppkui`
7. $\quad$ `kui` $2 \cdot i + 1 \leq n \wedge a_{2 \cdot i+1} > a_k$
8. $\quad\quad k \leftarrow 2 \cdot i + 1$
9. $\quad$ `lõppkui`
10. $\quad$ `kui` $k > i$
11. $\quad\quad a_i \leftrightarrow a_k$ — a_i on oma alluvast väiksem, see tõrjub ta allapoole
12. $\quad$ `lõppkui`

```
13.     i ← k
14. lõppsenikui
```

Kahe eeltoodud võtte kombineerimisel saame efektiivse algoritmi massiivide sortimiseks: kõigepealt muudame massiivi kuhjaks, “lisades” selle elemente järjest kuhja, mida hoiame sama massiivi algusosas; seejärel võtame kuhjast maksimaalse elemendi, taastame kuhja, võtame allesjäänutest maksimaalse, taastame uuesti kuhja, ja nii edasi, kuni olemegi kõik elemendid järjestanud.

Algoritm 8.10 KuhjaSort
Sisend: $a_{1...n}$ — töödeldav massiiv
Väljund: $a_{1...n}$ — väärtused vahetatud nii, et $\forall i < n : a_i \le a_{i+1}$

```
   — muudame massiivi kuhjaks
1. korda i ← 1...n
       — invariant: a_{1...i−1} juba on kuhi
2.     ViiUles(a_{1...i})
3. lõppkorda
   — muudame kuhja järjestatud massiiviks
4. korda i ← n...1
       — invariant: a_{1...i} on kuhi
5.     a_i ↔ a_1
6.     ViiAlla(a_{1...i−1})
       — vahetingimus: a_i ≤ a_{i+1} ≤ ... ≤ a_n
7. lõppkorda
```

Kuna n-tipulises kompaktses puus võib element nii üles kui alla liikuda ülimalt $O(\log n)$ sammu, saame kummagi korduse ajalise keerukuse hinnanguks $O(n \log n)$.

8.3 Graafialgoritmid

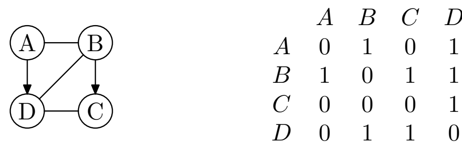
8.3.1 Graafi esitamine arvuti mälus

Kaks tähtsamat võtet graafide esitamiseks arvuti mälus on naabrusmaatriks ja kaareloendid. Mõlemad võtted sobivad paremini orienteeritud graafide esitamiseks. Orienteerimata servi esitatakse tavaliselt kahe vastassuunalise kaarena. See tähendab, et iga serva $\{v_1, v_2\}$ asemel pannakse graafi kaks kaart (v_1, v_2) ja (v_2, v_1). Kui pidada silmas ainult graafi ühelt tipult teisele liikumise võimalikkust, siis ei muuda selline asendus midagi. Kui aga mõnes ülesandes on vaja servi graafist eemaldada või vältida sama serva korduvat läbimist, tuleb sellised paariskaared kuidagi eraldi tähistada.

Graafi G **naabrusmaatriks** (ingl *adjacency matrix*) on ruutmaatriks, milles on üks rida ja üks veerg graafi G iga tipu kohta. Tipule v_1 vastava rea ja tipule v_2 vastava veeru ristumiskohas on info kaare (v_1, v_2) kohta.

Sellist maatriksit on kõige mugavam hoida 2-mõõtmelise massiivina. Lihtsaimal juhul on massiivi elemendid tõeväärtused või täisarvud, ning kaare olemasolu tähistab siis väärtus 1 ja selle puudumist väärtus 0 massiivi vastavas elemendis (nagu näha joonisel 8.12 toodud näites). Vajadusel võib luua ka massiivi, mille elemendid on keerukamad kirjed, ja hoida iga kaare kohta rohkem andmeid.

Täielikult orienteerimata graafi puhul kasutatakse naabrusmaatriksi asemel mõnikord "naabruskolmnurka" — maatriksist on kasutusel ainult selle peadiagonaali kohal olev osa. Kuna orienteerimata graafi naabrusmaatriks on alati peadiagonaali suhtes sümmeetriline, siis pole sel juhul diagonaali alla jääva osa andmetega täitmine vajalik.

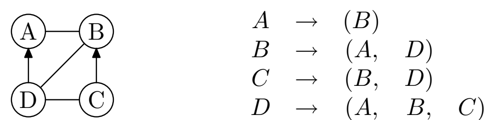


	A	*B*	*C*	*D*
A	0	1	0	1
B	1	0	1	1
C	0	0	0	1
D	0	1	1	0

Joonis 8.12: Segagraaf ja tema naabrusmaatriks

Graafi **esitus kaareloenditena** (ingl *adjacency-lists representation*) annab iga tipu kohta sellest tipust väljuvate servade loetelu.

Selliseid loetelusid on tavaliselt mugav hoida lihtahelates, ja kogu graafi esitus koosneb siis massiivist või ahelast, milles on üks element graafi iga tipu kohta. Lisaks otseselt tipu enda andmetele sisaldab see element ka viita tipust väljuvate servade loetelule (nagu näha joonisel 8.13 toodud näites), ja iga serva element omakorda viitab mingil viisil selle serva lõpptipule.



$A \rightarrow (B)$
$B \rightarrow (A, D)$
$C \rightarrow (B, D)$
$D \rightarrow (A, B, C)$

Joonis 8.13: Segagraaf ja tema kaareloendid

Kaareloendeid on parem kasutada sellistes algoritmides, kus on vaja kiiresti leida kõik antud tipust väljuvad kaared (või kõik tipud, kuhu antud tipust

edasi liikuda saab). Naabrusmaatriksid on tõhusamad algoritmides, kus on vaja kiiresti kontrollida, kas kahe antud tipu vahel on kaar või mitte.

Mälukasutuse seisukohalt on naabrusmaatriksid efektiivsemad tihedate ja kaareloendid hõredate graafide esitamisel.[5]

8.3.2 Graafi läbimine laiuti

Nagu puude korral, nii on ka paljude graafiülesannete lahenduse võti graafi tippude (ja servade) õiges järjekorras läbimises. Sageli osutub selleks õigeks järjekorraks graafi **läbimine laiuti** (ingl *breadth-first search*).

Laiuti läbimine (algoritm 8.11) algab mingist lähtetipust (joonisel 8.14 tipp A) ja vaatleb juba leitud tippudest väljuvaid servi või kaari, et leida uusi, seni avastamata tippe.[6] Uusi tippe töödeldakse nende avastamise järjekorras, seetõttu moodustavad avastatud, aga veel töötlemata tipud (algoritmis järjekorra Q sisu, joonisel tähistatud helehalliga) lähtetipu ümber omamoodi "laineharja" või "rindejoone", mis sellest igas suunas võrdse kiirusega eemaldub. Viimase asjaolu järgi ongi laiuti läbimine oma nime saanud.

Algoritm 8.11 LabiGraafL: Läbib graafi laiuti, alustades antud tipust
Sisend: graaf G; lähtetipp $v_0 \in V(G)$
Abimuutujad: G iga tipu v jaoks abimuutuja $v.olek$; tippude järjekord Q

1. `korda` $v \in V(G)$ — kõik V tipud
2. $\quad v.olek \leftarrow valge$
3. `lõppkorda`
4. $Q \leftarrow v_0$; $v_0.olek \leftarrow hall$
5. `senikui` $Q \neq \emptyset$ — järjekord pole tühi
6. $\quad v \leftarrow Q$ — võtame järjekorra esimese elemendi
7. $\quad$ `korda` $w \in V(v)$ — kõik v naabertipud
8. $\quad\quad$ `kui` $w.olek = valge$ — uus tipp
9. $\quad\quad\quad Q \leftarrow w$; $w.olek \leftarrow hall$
10. $\quad\quad$ `lõppkui`
11. $\quad$ `lõppkorda`
12. $\quad v.olek \leftarrow must$ — v töödeldud
13. `lõppsenikui`

[5]**Tihedaks** (ingl *dense*) nimetatakse graafi, mille mille tipupaaridest enamik on omavahel ühendatud. **Hõredaks** (ingl *sparse*) nimetatakse graafi, mille tipupaaridest enamik on omavahel ühendamata.

[6]Siin ja ka järgnevates näidetes eeldame, et igast tipust väljuvaid servi vaadeldakse naabertippude nimede tähestikulises järjekorras. Algoritmid töötavad õigesti ka ilma selle eelduseta, aga näited tuleks teise läbivaatusjärjekorra puhul veidi erinevad.

8.3.3 Lühima tee leidmine

Sidusa graafi laiuti läbimisel tippude avastamiseks kasutatud servad (joonisel 8.14 märgitud paksema joonega) moodustavad selle graafi **toesepuu** (ingl *spanning tree*) — puu, mille tippude hulk langeb kokku graafi G tippude hulgaga ja mille servade hulk on graafi servade hulga alamhulk.

Graafil võib olla mitmeid erinevaid toesepuid, kuid just laiuti läbimisel saadud toesepuu kasulik omadus on, et igast tipust mööda sellesse puusse kuuluvaid servi lähtetippu viiv tee ongi minimaalse võimaliku pikkusega, kui me loeme tee pikkuseks sellele jäävate servade arvu. Seega on graafis lühimate teede leidmiseks vaja laiuti läbimist vaid minimaalselt modifitseerida.

Algoritm 8.12 LyhTeed: Leiab lühimad teed, alustades antud tipust
Sisend: graaf G; lähtetipp $v_0 \in V(G)$
Abimuutujad: G iga tipu v jaoks abimuutujad $v.kaug$ ja $v.eelm$;
tippude järjekord Q

1. `korda` $v \in V(G)$ — kõik V tipud
2. $v.kaug \leftarrow \infty$
3. `lõppkorda`
4. $Q \leftarrow v_0$; $v_0.kaug \leftarrow 0$; $v_0.eelm \leftarrow \perp$
5. `senikui` $Q \neq \emptyset$ — järjekord pole tühi
6. $v \leftarrow Q$ — võtame järjekorra esimese elemendi
7. `korda` $w \in V(v)$ — kõik v naabertipud
8. `kui` $w.kaug = \infty$ — uus tipp
9. $Q \leftarrow w$; $w.kaug \leftarrow v.kaug + 1$; $w.eelm \leftarrow v$
10. `lõppkui`
11. `lõppkorda`
12. `lõppsenikui`

Algoritmi 8.12 õigsuse põhjendamiseks paneme tähele, et

- läbimise algatamine real 4 on ilmselt õige, sest lähtetipu v_0 kaugus iseendast on 0;

- real 5 algav kordus töötleb graafi tippe nende lähtetipust kaugenemise järjekorras; täpsemalt, kõik tipud, mille kaugus lähtetipust on k töödeldakse enne kui ükski tipp, mille kaugus on $k+1$;

- viitade $v_k.eelm = v_{k-1}$, $v_{k-1}.eelm = v_{k-2}$, ..., $v_1.eelm = v_0$ poolt näidatav tee mistahes tipust v_k lähtetippu v_0 on minimaalse sammude arvuga; kui see nii ei oleks, siis peaks sellel teel leiduma tipp v_i, mille viit $v_i.eelm$ osutab tipule v_{i-1} ja millel on olemas ka naaber v_j, mille kaugus lähtetipust on väiksem kui $i-1$; siis aga oleks v_j töödeldud

enne kui v_{i-1} ja tipp v_i avastatud serva $\{v_j, v_i\}$ kaudu; kuna seda ei juhtunud, siis järelikult sellist tippu v_j ei ole, ja leitud tee ongi lühim võimalik.

Kuna algoritm töötleb graafi G iga tippu täpselt ühe korra ja vaatleb iga serva täpselt kaks korda, siis kulub N tipu ja M servaga graafi töötlemiseks kokku $\mathrm{O}(N + M)$ operatsiooni.

Selle algoritmi tööd illustreerib joonis 8.15, kus tippudele on märgitud nende kaugused lähtetipust ja nooled osutavad igast tipust tema ülemusele toesepuus.

Teema lõpetuseks võiks veel märkida, et algoritmi 8.12 on võimalik üldistada ka juhule, kui graafi servadele on antud mittenegatiivsed kaalud ja tee pikkuseks loetakse mitte teel olevate servade arvu, vaid nende kaalude summat. Seda üldistust tuntakse autori järgi **Dijkstra algoritmi** nime all, aga selle täpsemaks kirjeldamiseks ja põhjendamiseks ei jätku siinkohal ruumi.

8.3.4 Graafi läbimine sügavuti

Laiuti läbimise kõrval on teine tähtsam viis graafi töötlemiseks selle **läbimine sügavuti** (ingl *depth-first search*). Sarnaselt laiuti läbimisega algab ka sügavuti läbimine (algoritmid 8.13 ja 8.14) mingist lähtetipust ja vaatleb uute tippude leidmiseks juba avastatud tippudest väljuvaid servi või kaari. Erinevus seisneb selles, et iga uue tipu avastamisel töödeldakse sellest tipust algav "haru" enne järgmise juurde asumist lõpuni.

Algoritm 8.13 LABIGRAAFS: Läbib graafi sügavuti, alustades antud tipust
Sisend: graaf G; lähtetipp $v_0 \in V(G)$
Abimuutujad: G iga tipu v jaoks abimuutuja $v.olek$, $v.alust$, $v.lopet$;
globaalne loendur t

1. `korda` $v \in V(G)$ — kõik V tipud
2. $\quad v.olek \leftarrow valge$
3. `lõppkorda`
4. $t \leftarrow 0$
5. TOOTLETIPPS(v_0)

Algoritm 8.14 TOOTLETIPPS: Läbib sügavuti antud tipu "alampuu"
Sisend: $v \in V(G)$

1. $v.olek \leftarrow hall$; $t \leftarrow t + 1$; $v.alust \leftarrow t$ — v töötlemise algus
2. `korda` $w \in V(v)$ — kõik v naabertipud
3. $\quad$ `kui` $w.olek = valge$ — uus tipp
4. $\qquad$ TOOTLETIPPS(w)

5. `lõppkui`
6. `lõppkorda`
7. $v.olek \leftarrow must$; $t \leftarrow t + 1$; $v.lopet \leftarrow t$ — v töötlemise lõpp

Sarnaselt laiuti läbimisega eraldab ka sügavuti läbimine töödeldavast graafist toesepuu, aga see puu hoopis teise kujuga (vrdl jooniseid 8.14 ja 8.16). Sarnaselt laiuti läbimisega on ka sügavuti läbimise keerukus $O(N + M)$.

8.3.5 Topoloogiline sorteerimine

Paljudes sügavuti läbimisel põhinevates algoritmides on kasulik tähele panna iga tipu töötlemise alguse (algoritmi 8.14 rida 1) ja lõpu (rida 7) aega. Nende aegade omavahelised suhted erinevate tippude vahel on tihedalt seotud graafi läbimise käigus eraldatava toesepuu struktuuriga.

Üks ülesandeid, mille lahendamisel tippude läbimise aegade võrdlemine kasuks tuleb, on suunatud graafi tippude sorteerimine **topoloogilisse järjekorda** (ingl *topological order*). Topoloogiliseks nimetatakse graafi tippude järjestust, mille korral kõik kaared osutavad järjekorras eespool olevatelt tippudelt tagapool olevatele, aga mitte kunagi vastupidi.

Topoloogilist sorteerimist on vaja näiteks üksteisest sõltuvate tööde järjekorra planeerimisel: kui me esitame iga töö graafi tipuna ja tõmbame igasse tippu kaared kõigist neist, mis peavad enne selle töö alustamist tehtud olema, siis saame töid topoloogilises järjekorras ette võttes kõik õigesti tehtud.

Algoritm 8.15 TOPOSORT: Leiab suunatud graafis topoloogilise järjestuse
Sisend: graaf G
Abimuutujad: G iga tipu v jaoks abimuutuja $v.olek$; tippude magasin S

1. `korda` $v \in V(G)$ — kõik V tipud
2. $v.olek \leftarrow valge$
3. `lõppkorda`
4. $S \leftarrow \emptyset$
5. `korda` $v \in V(G)$ — kõik V tipud
6. `kui` $v.olek = valge$ — uus juurtipp
7. TOPOTIPP(w)
8. `lõppkui`
9. `lõppkorda`
10. `senikui` $S \neq \emptyset$ — magasin pole tühi
11. $v \leftarrow S$ — võtame magasini tipmise elemendi
12. `väljasta` v
13. `lõppsenikui`

Algoritm 8.16 TopoTipp: Läbib sügavuti antud tipu "alampuu"
Sisend: $v \in V(G)$

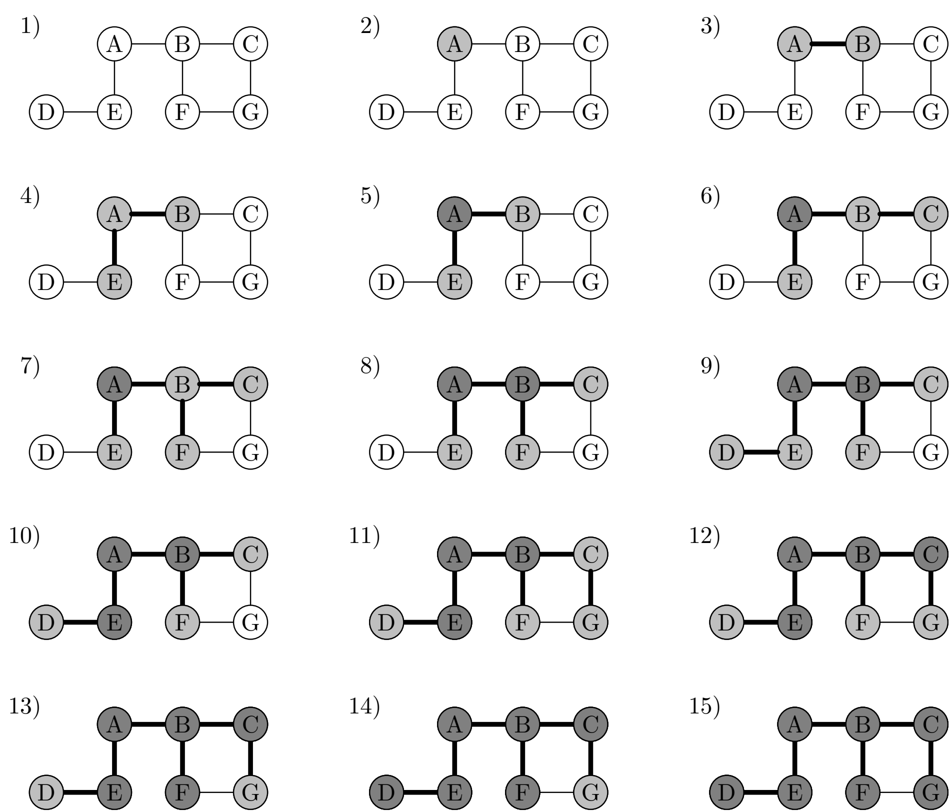
1. $v.olek \leftarrow hall$
2. `korda` $w \in V(v)$ — kõik v naabertipud
3.   `kui` $w.olek = valge$ — uus tipp
4.   TopoTipp(w)
5.   `lõppkui`
6. `lõppkorda`
7. $v.olek \leftarrow must$; $S \leftarrow v$ — töödeldud tipp magasini

Algoritmi 8.15 õigsuse põhjendamiseks tuleb tähele panna, et

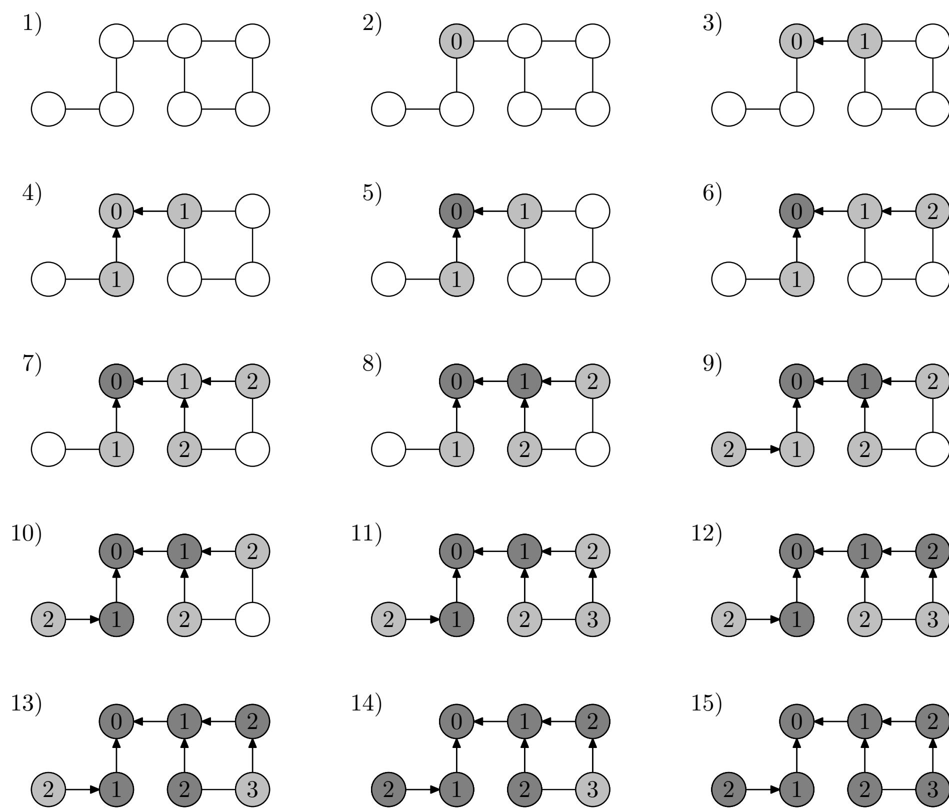
- väljakutse TopoTipp(v) töötleb kõik need tipud, mis on tipust v graafi servi (või kaari) pidi liikudes saavutatavad ja mis veel ei ole töödeldud;
- seega on väljakutse TopoTipp(v) täitmise lõpuks töödeldud kõik need tipud, mis ei tohi topoloogilises järjestuses tipust v eespool olla;
- seega tuleks topoloogilise järjestuse saamiseks tipud väljastada nende töötlemise lõpuaegade kahanemise järjekorras;
- seda aga TopoSort teebki, kuna TopoTipp väljakutsed panevad tipud nende töötlemise lõppedes magasini ja TopoSort väljastab nad magasinist väljavõtmise järjekorras (mis vastavalt magasini definitsioonile on vastupidine nende magasini panemise järjekorrale.

Algoritmi 8.15 tööd illustreerib joonis 8.17.

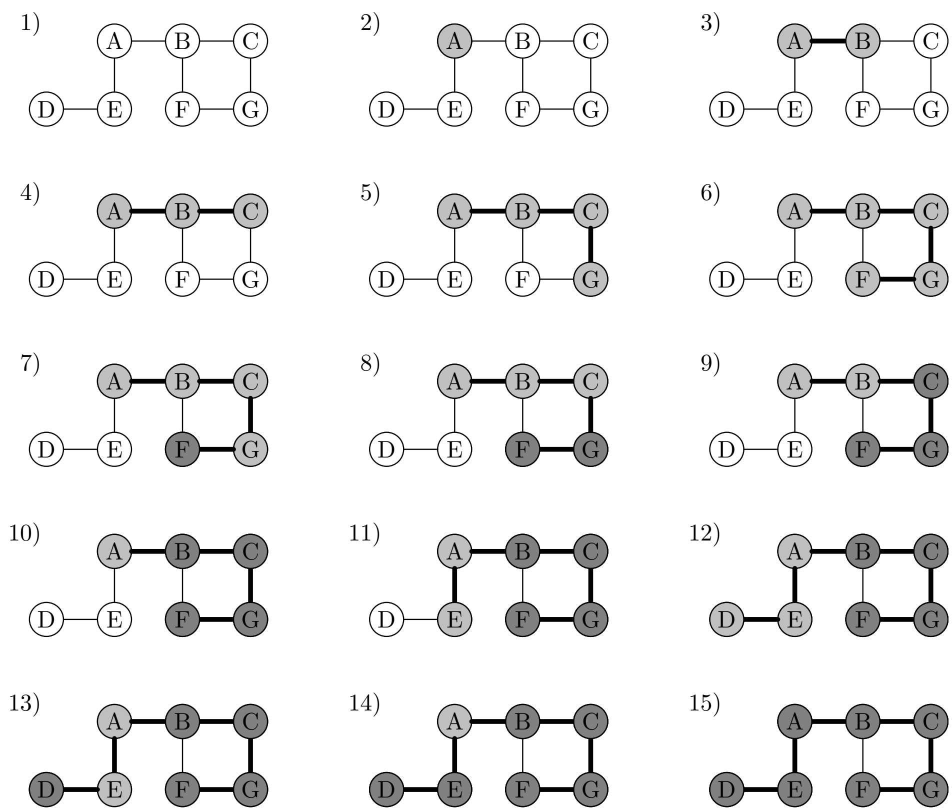
Muidugi ei saa topoloogiliselt järjestada graafi, milles on tsüklid. Näiteks graafis $G = (V, E)$, kus $V = \{A, B, C\}$ ja $E = \{(A, B), (B, C), (C, A)\}$, ei saa ükski tipp ei saa olla topoloogilises järjekorras esimene. Osutub, et efektiivne viis graafi tsüklilisuse kontrollimiseks ongi püüda teda topoloogiliselt sorteerida ja kontrollida, et selle käigus vastuolu ei teki.



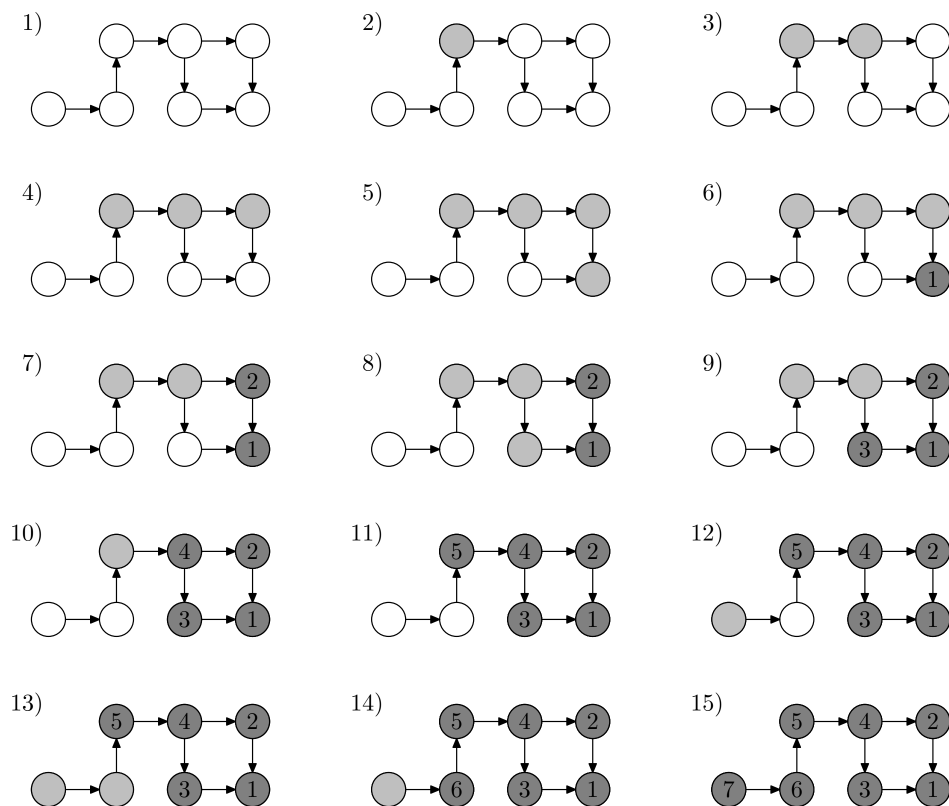
Joonis 8.14: Graafi läbimine laiuti



Joonis 8.15: Lühimate teede leidmine graafis



Joonis 8.16: Graafi läbimine sügavuti



Joonis 8.17: Graafi topoloogiline sorteerimine

Ülesanded

Ülesanne 8.1 *Kirjutada alamprogramm* TRYKIPUU*, mis saab kahendpuu loomuliku esituse ja väljastab suluavaldise, mis sobib selle puu uuesti sisselugemiseks näidete hulgas toodud alamprogrammi* LOEPUU *abil. (Vihje: millises järjekorras peavad puu tipud selles suluavaldises olema?)*

Ülesanne 8.2 *Koostada algoritm ja kirjutada programm loomulikus esituses antud kahendpuu tippude läbimiseks "sügavuse järjekorras": kõigepealt juurtipp, siis selle vahetud alluvad, siis nende vahetud alluvad jne. Näiteks kõigis kolmes joonisel 8.4 toodud puus tuleks tipud läbida nende märgendite tähestikulises järjekorras.*

Ülesanne 8.3 *Koostada algoritm, mis leiab naabrusmaatriksina antud suunatud graafi iga tipu jaoks temast väljuvate ja temasse sisenevate kaarte arvu. Milline on selle algoritmi tööaeg N tipu ja M servaga graafi töötlemisel?*

Kohandada see algoritm kaareloenditena antud graafile. Milline on tema tööaeg nüüd?

Ülesanne 8.4 *Koostada laiuti läbimisel põhinev algoritm ja kirjutada selle alusel programm, mis leiab naabrusmaatriksina antud suunamata graafi sidususkomponendid. Vastusena väljastada komponentide arv ja igasse komponenti kuuluvate tippude loetelu.*

Ülesanne 8.5 *Millise tulemuse annab algoritmi 8.15 rakendamine suunatud graafile, milles on tsükkel? Täiendada õppematerjali lisas toodud programmi* TOPOSORT *nii, et tsükleid sisaldava graafi töötlemisel oleks tulemuseks asjakohane veateade.*

Kirjandus

Kõik eelmises peatükis viidatud materjalid käsitlevad vähemalt mingil määral ka mittelineaarseid struktuure. Samuti võib puude ja graafidega seotud peatükke leida paljudest kombinatoorikaõpikutest, sealhulgas ka käesoleva õppevahendi kombinatoorikateemas viidatavatest. Seetõttu on allpool välja toodud vaid mõned konkreetselt graafiteooriale keskenduvad õpikud.

Peeter Puusemp. *Graafiteooria elemente.* Tallinna Tehnikaülikool, 2000.
Üldine sissejuhatus graafiteooriasse. 96 lk.

Ahto Buldas, Peeter Laud, Jan Villemson. *Graafid.* Tartu Ülikool, 2003.
Üldine sissejuhatus graafiteooriasse. 92 lk.

Oystein Ore. *Graafid ja nende kasutamine.* Valgus, 1976.
Keskendub rohkem graafiteooria matemaatilisele poolele, programmeerimisele eriti tähelepanu ei pööra. 139 lk.

Frank Harary. *Graph Theory.* Addison-Wesley, 1969.
Klassikaline graafiteooria õpik, samuti matemaatilise kallakuga. 274 lk.

Reinhard Diestel. *Graph Theory.* Springer, 2000.
Üsna uus klassikalise graafiteooria õpik. 315 lk.
http://www.math.uni-hamburg.de/home/diestel/books/graph.theory/