Programmeringsteori og Datastrukturer

Erik Meineche Schmidt
Michael I. Schwartzbach

Indhold

Introduktion **1**

1 Fundamentale begreber og notation **2**

1.1 Mængder 2

1.2 Følger 3

1.3 Funktioner 3

1.4 Prædikater 4

2 Elementære algoritmer **8**

2.1 Notation 8

2.2 Potensopløftning 10

2.3 Søgning 11

2.4 Fletning 15

2.5 Flytning 20

3 Kompleksitet **23**

3.1 Notation for størrelsesorden 24

3.2 Definition af tidskompleksitet 25

3.3 Direkte analyse 26

3.4 Analyse af nogle konkrete algoritmer 29

3.5 Analyse af en binær tæller 30

3.6 Tidskompleksitet af procedurer 32

3.7 Tidskompleksitet af rekursive procedurer 34

4 Specifikationer **38**

5 Stakke og køer **41**

5.1 Stakke 41

5.2 Køer 43

5.3 Implementationsovervejelser 45

5.4 En anvendelse: beregning af postfix udtryk 45

6 Prioritetskøer **47**

6.1 Bunker 48

6.2 En polymorf prioritetskø 54

6.3 En anvendelse: træsortering 57

7 Ordbøger **61**

7.1 Bitvektorer 61

7.2 Hash-tabeller 63

7.3 Søgetræer 66

7.4 Rød-sorte træer 70

7.5 Indsættelse i rød-sorte træer 72

7.6 Fjernelse fra rød-sorte træer 74

7.7 Andre højdebalancerede træer 77

7.8 Vægtbalancerede træer 78

7.9 Tider for balancerede søgetræer 80

8 Ækvivalensrelationer **81**

8.1 Kanoniske repræsentanter 81

8.2 Inverse træer 82

8.3 Vejforkortning 85

9 Amortiseret analyse **87**

9.1 Binær kilometertæller som datastruktur 87
9.2 Potentialfunktion 88
9.3 En monoton Sequence 91
9.4 Datatypen Sequence 95

10 Korrekthed af procedurer **97**
10.1 Ikke-rekursive procedurer 97
10.2 Rekursive procedurer 102

11 Eksempel: Erathostenes Si **105**
11.1 Den abstrakte algoritme 105
11.2 Den konkrete algoritme 107
11.3 Kompleksiteten 109
11.4 TRINE programmet 110
11.5 Kompleksiteten af TRINE programmet 113

12 Referencer **115**

Introduktion

Denne bog indeholder vigtige elementer af den del af programmeringsteorien, der beskæftiger sig med konstruktion af korrekte, effektive og let forståelige programmer.

Bogen starter med en introduktion til *algoritmer*, som er en slags abstrakte programstumper med formaliserede egenskaber. Efter en repetition af begreberne gyldighed og korrekthed, præsenteres de mest fundamentale algoritmer til søgning, fletning og sortering. Derefter etableres grundlaget for analyser af algoritmers tidskompleksitet, og dette anvendes på en række simple eksempler.

På denne baggrund introduceres nu de mest fundamentale *datastrukturer*, og det vises, hvordan forskellige implementationer fører til forskellige effektivitetsegenskaber. Vi beskæftiger os med køer, stakke, prioritetskøer, ordbøger og relationer, som implementeres v.hj.a. forskellige former for lister og (balancerede) træer. Datatypers operationer formaliseres v.hj.a. såkaldte *specifikationer*, som er et værktøj til formel beskrivelse af egenskaber ved ubestemte programstumper.

For en række vigtige datastrukturer er den sædvanlige *worst-case* kompleksitet et alt for pessimistisk kompleksitetsmål. Vi introducerer derfor begrebet *amortiseret* kompleksitet, som formaliserer begrebet "gennemsnitlig udførelsestid", og det vises v.hj.a. et par konkrete eksempler, hvordan dette intuitivt kan forbedre karakterisationen af en datatypes operationer.

Dernæst præsenteres en bid af teorien for ræsonnementer om programmer med boxe og procedurer, som vi sammenfatter i to såkaldte *bevisprincipper*.

Bogens sidste kapitel indeholder et eksempel, der viser, hvordan de fleste af de gennemgåede begreber og teknikker kan spille sammen i forbindelse med konstruktion af et konkret TRINE program. Der er tale om den såkaldte Erathostenes Si, som er en klassisk metode til at finde primtal.

1 Fundamentale begreber og notation

I dette afsnit repeteres/introduceres de grundlæggende matematiske begreber (med tilhørende notation), der vil blive brugt i dette hæfte.

1.1 Mængder

Vi bruger den sædvanlige mængdenotation (M_1 og M_2 er mængder og e er et element).

- $e \in M$: e tilhører M
- $e \notin M$: e tilhører ikke M
- $M_1 \cup M_2$: foreningsmængde
- $M_1 \cap M_2$: fællesmængde
- $M_1 \backslash M_2$: differens
- $M_1 \subseteq M_2$: inklusion
- $M_1 \subsetneq M_2$: ægte inklusion
- $|M|$: antal elementer i M
- $\emptyset$: den tomme mængde

De mængder, vi betragter, vil altid være delmængder af en mere eller mindre eksplicit defineret grundmængde G (G er ofte de hele tal, mængden af tegnfølger fra et givet alfabet o.l.). Enhver delmængde M af en grundmængde G er entydigt bestemt af sin *karakteristiske funktion*, som er en boolsk funktion (vi bruger $\underline{\mathrm{t}}$ og $\underline{\mathrm{f}}$ for de to Boolske værdier $\underline{\text{true}}$ og $\underline{\text{false}}$)

$$\chi_M : G \to \{\underline{\mathrm{t}}, \underline{\mathrm{f}}\}$$

defineret ved

$$\chi_M(e) = \begin{cases} \underline{\mathrm{t}} \text{ hvis } e \in M \\ \underline{\mathrm{f}} \text{ ellers} \end{cases}$$

Vi benytter følgende betegnelser for de mest almindelige mængder

- $\mathbb{N}$: de naturlige tal, $\mathbb{N} = \{1, 2, 3, \ldots\}$
- $\mathbb{N}_0$: de ikke-negative hele tal, $\mathbb{N}_0 = \{0, 1, 2, \ldots\}$
- $\mathbf{Z}$: de hele tal, $\mathbf{Z} = \{\ldots, -2, -1, 0, 1, 2, \ldots\}$

Da vi ofte får brug for at tale om mængder af heltal, hvori elementerne udgør et "interval", indfører vi følgende betegnelse

$$n..m = \begin{cases} \emptyset \text{ hvis } n \geq m \\ \{n, n+1, \ldots, m-1\} \text{ ellers} \end{cases}$$

Bemærk, at $n..m$ er "halvåbent", idet n er med, men m er ikke med.

1.2 Følger

Hvis M er en mængde, betegner M^* mængden af følger af elementer fra M. Følger er allerede introduceret i [TRINE] side 31 – vi bruger følgende yderligere notation

- den tomme følge ( ) betegnes også med λ
- hvis $f = (m_0, m_1, \ldots, m_{n-1})$ er en følge, betegner vi med $\mathrm{hd}(f)$ (head) *elementet* m_0, og med $\mathrm{tl}(f)$ (tail) *følgen* $(m_1, \ldots, m_{n-1})$. Såvel $\mathrm{hd}(f)$ som $\mathrm{tl}(f)$ er udefineret, hvis $f = \lambda$
- hvis $f' = (m'_0, m'_1, \ldots, m'_{n-1})$ og $f'' = (m''_0, \ldots, m''_{k-1})$ er to følger, betegnes deres sammensætning (eller *konkatenation*) med

$$f = f' \cdot f'' = ((m'_0, \ldots, m'_{n-1}, m''_0, \ldots, m''_{k-1})$$

1.3 Funktioner

Givet mængder A og B, betegner

$$A \to B$$

mængden af funktioner fra A til B. Vi angiver, at f er en sådan funktion ved enten at skrive

$$f \in A \to B$$

eller det mere konventionelle

$$f : A \to B$$

Hvis $A' \subseteq A$, betegner vi *billedet af* A' med

$$f(A') = \{f(a) \mid a \in A'\}$$

og hvis $B' \subseteq B$, er *urbilledet af* B'

$$f^{-1}(B') = \{a \in A \mid f(a) \in B'\}$$

1.4 Prædikater

Begrebet (program)udsagn er introduceret uformelt i kapitel 8 i [TRINE]. Formelt er et udsagn et *prædikat*, dvs. en Boolsk funktion skrevet på "ligningsform". Prædikatet

$$x + y = z$$

angiver den Boolske funktion F

$$F(x, y, z) = \begin{cases} \underline{\text{t}} \text{ hvis } x + y = z \\ \underline{\text{f}} \text{ ellers} \end{cases}$$

Prædikater bygges op v.hj.a.

- konstanter $0, 1, \ldots, \underline{\text{t}}, \underline{\text{f}}$
- variabler $x, y, z, \ldots$
- funktionssymboler $+, -, \leq, =, \ldots$
- konnektiver $\wedge, \vee, \neg, \Rightarrow, \Leftrightarrow$
- kvantorer $\forall$, $\exists$

Vi skal ikke formalisere de præcise regler for prædikaters udseende, men nøjes med at angive og diskutere følgende eksempler

1) $P_1 : (x - 2 * y) > 0$
2) $P_2 : (x \leq y) \wedge (z = 0)$
3) $P_3 : (5 < 3) \Rightarrow \forall x \in 0..1 : x = y$
4) $P_4 : \forall x \in \mathbf{Z} : x * x > 0$
5) $P_5 : \exists x \in \mathbf{Z} : (\forall y \in \mathbf{Z} : x * y = z)$
6) $P_6 : \forall x \in 0..10 : (\exists y \in 0..11 : x < y)$
7) $P_7 : (x \leq y) \vee ((\forall x \in \mathbb{N} : x > 0) \wedge (y = 3))$

De variabler, der forekommer i et prædikat, deles op i *frie* og *bundne* variabler. De bundne variabler er alle de variabler, der er "knyttet til" en kvantor ($\forall$ eller $\exists$). De spiller nogenlunde samme rolle som lokale variabler (i indskudte sætninger) i et TRINE program, og de har også samme virkefelter. De variabler, der ikke er bundet, kaldes frie, og et prædikat er en Boolsk funktion af sine frie variabler.

Vi angiver de frie variabler i eksemplerne ovenfor ved for hvert prædikatsymbol P_i eksplicit at angive dets argumenter på den sædvanlige form $P_i(x, y, \ldots)$

1) $P_1(x, y)$
2) $P_2(x, y, z)$
3) $P_3(y)$
4) $P_4(\ )$
5) $P_5(z)$
6) $P_6(\ )$
7) $P_7(x, y)$

P_4 og P_6 er altså Boolske funktioner af 0 variabler, dvs. de er *konstanter*. En konstant Boolsk funktion er enten lig med $\underline{\mathrm{t}}$ eller $\underline{\mathrm{f}}$ (fordi der ikke er andre Boolske værdier).

P_4 siger, at "kvadratet på ethvert heltal er positivt", hvilket er forkert, fordi $0 * 0 = 0$. Altså er $P_4 = \underline{\mathrm{f}}$.

P_6 siger, at "for ethvert element i 0..10 findes der et større element i 0..11, hvilket unægteligt er rigtigt. Altså er $P_6 = \underline{\mathrm{t}}$.

For de øvrige prædikater gælder, at værdierne afhænger af værdierne af

deres argumenter. Vi viser nogle eksempler

$$\begin{array}{lcl}
P_1(6,2) & = & (6-4) > 0 = \underline{\mathrm{t}} \\
P_3(2) & = & (5 < 3) \Rightarrow \forall x \in 0..1 : x = 2 \\
& = & \underline{\mathrm{f}} \Rightarrow (0 = 2) \wedge (1 = 2) \\
& = & \underline{\mathrm{t}} \\
& & \text{(fordi } \underline{\mathrm{f}} \text{ implicerer hvad som helst)} \\
P_5(0) & = & \exists x \in \mathbf{Z} : (\forall y \in \mathbf{Z} : x * y = 0) \\
& = & \underline{\mathrm{t}} \\
& & \text{(fordi } x = 0 \text{ giver } \forall y \in \mathbf{Z} : 0 * y = 0, \\
& & \text{hvilket er rigtigt)} \\
P_7(5,y) & = & (5 \leq y) \vee ((\forall x \in I\!N : x > 0) \wedge (y = 3)) \\
& = & (5 \leq y) \vee (y = 3) \\
& & \text{(fordi alle naturlige tal er større end 0)}
\end{array}$$

Som disse eksempler viser, finder man værdien af et prædikat på givne argumenter ved at erstatte forekomster af de frie variabler med de tilsvarende værdier, og derefter regne v.hj.a. de sædvanlige aritmetiske og logiske regler. Regnereglerne for $\wedge, \vee, \neg$ er som bekendt givet ved følgende tabel:

P	Q	$P \wedge Q$	$P \vee Q$	$\neg P$
$\underline{\mathrm{t}}$	$\underline{\mathrm{t}}$	$\underline{\mathrm{t}}$	$\underline{\mathrm{t}}$	$\underline{\mathrm{f}}$
$\underline{\mathrm{t}}$	$\underline{\mathrm{f}}$	$\underline{\mathrm{f}}$	$\underline{\mathrm{t}}$	$\underline{\mathrm{f}}$
$\underline{\mathrm{f}}$	$\underline{\mathrm{t}}$	$\underline{\mathrm{f}}$	$\underline{\mathrm{t}}$	$\underline{\mathrm{t}}$
$\underline{\mathrm{f}}$	$\underline{\mathrm{f}}$	$\underline{\mathrm{f}}$	$\underline{\mathrm{f}}$	$\underline{\mathrm{t}}$

Betydningen af $\Rightarrow$ og $\Leftrightarrow$ er nu som følger:

- $(P \Rightarrow Q) = (\neg P \vee Q)$
- $(P \Leftrightarrow Q) = (P \Rightarrow Q) \wedge (Q \Rightarrow P)$

Betydningen af kvantorerne $\forall$ og $\exists$ kan defineres på flere måder – vi vælger at gøre det rekursivt:

$$\begin{array}{lcl}
\forall x \in X : P(x) & = & \begin{cases} \underline{\mathrm{t}} \text{ hvis } X = \emptyset \\ P(x_0) \wedge (\forall x \in X \backslash \{x_0\} : P(x)) \text{ for } x_0 \in X \end{cases} \\
\exists x \in X : P(x) & = & \begin{cases} \underline{\mathrm{f}} \text{ hvis } X = \emptyset \\ P(x_0) \vee (\exists x \in X \backslash \{x_0\} : P(x)) \text{ for } x_0 \in X \end{cases}
\end{array}$$

Heraf følger den nyttige regneregel for negation

$$\neg(\forall x \in X : P(x)) = \exists x \in X : \neg P(x)$$

Bemærk også, at universel (eksistentiel) kvantificering over den tomme mængde altid er sand (falsk).

2 Elementære algoritmer

I dette kapitel konstruerer vi et antal simple algoritmer, som i forskellige iklædninger optræder igen og igen i praktisk programmering.

2.1 Notation

Vi skal anvende en notation for algoritmer, der er lidt mere abstrakt end fuldt færdige TRINE programmer og som illustreres af følgende udgave af Euklids algoritme (jfr. [TRINE] s. 93).

Algoritme : Euklid
Stimulans : n, m: $(n \geq 1) \wedge (m \geq 1)$
Respons : r: største fælles divisor af m og n
Metode : $p, q := n, m$
do $\{I\}$
$p > q \rightarrow p := p - q$
$\mid q > p \rightarrow q := q - p$
od
$r := p$

Denne notation vil blive brugt systematisk i det følgende til beskrivelse af abstrakte programmer, eller dele heraf, og er i det generelle tilfælde af formen

Algoritme : ☐
Stimulans : ☐
Respons : ☐
Metode : ☐

hvor stimulans og respons beskriver omgivelsernes påvirkninger af algoritmen og dennes reaktion herpå. Metodedelen er en abstrakt beskrivelse af algoritmen, normalt i stil med kroppen af en proces. Når vi bruger ordet *algoritme* i stedet for program, proces eller lignende, skyldes det at sidstnævnte indeholder en del elementer i form af erklæringer, initialiseringer,

ind- og udlæsning osv., som vi ofte vil være interesseret i at abstrahere væk fra. Der vil også være situationer, hvor vi ønsker at udtage en mindre del af en proces eller et system og betragte denne i isolation, og i sådanne tilfælde er det også hensigtsmæssigt med et nyt navn for beskrivelsen.

Vi anvender begreberne udsagn (som nu er prædikater), invariant og termineringsfunktion på præcis samme måde som i [TRINE]. Vi kan dermed tale om at (dekorerede) algoritmer er gyldige og korrekte på følgende måde.

En tilstand i en algoritme er (i lighed med en TRINE tilstand) en tilknytning af værdier til algoritmens variabler. En tilstand σ opfylder et prædikat P (hvis frie variabler er programvariabler), såfremt P har værdien $\underline{t}$ anvendt på de frie variablers værdier i σ.

Et udsagn i en algoritme er *gyldigt for en udførelse*, såfremt udsagnet opfyldes af algoritmens tilstand, hver gang det nås under den pågældende udførelse.

En *algoritme er gyldig*, såfremt alle dens udsagn er gyldige for alle de udførelser der starter med en tilstand, som opfylder algoritmens stimulansprædikat.

Vi kan nu præcisere kravene til algoritmers korrekthed på følgende måde.

En algoritme er *korrekt* såfremt enhver udførelse, der starter i en tilstand der opfylder stimulansprædikatet, er endelig, og slutter i en tilstand der opfylder responsprædikatet.

Med disse definitioner skulle det være klart, at hvis invarianten i Euklid ovenfor er

$$I : (sfd(p,q) = sfd(m,n)) \wedge (n \geq 1) \wedge (m \geq 1) \wedge (p \geq 1) \wedge (q \geq 1)$$

så er algoritmen både gyldig og korrekt. Termineringsfunktionen (som vi skal betegne med μ) er

$$\mu(p,q) = p + q$$

2.2 Potensopløftning

Potensopløftningsprogrammet side 96 i [TRINE] kan formuleres som følgende algoritme

Algoritme : Potensopløftning
Stimulans : $n, p : p \geq 0$
Respons : $r : r = n^p$
Metode : $\ll$Initialiser r og $q \gg$
do $\{r * n^q = n^p\}$
$q \neq 0 \rightarrow \ll$Opdater r og $q \gg$
od

og vi har allerede set, at følgende konkretiseringer giver en gyldig algoritme, som også standser.

$\ll$Initialiser r og $q \gg$ **is** $r, q := 1, p$
$\ll$Opdater r og $q \gg$ **is** $r, q := r * n, q - 1$

Det er også nævnt, at denne metode involverer et antal multiplikationer, der er proportionalt med p. Vi kan finde en væsentlig mere effektiv potensopløftningsmetode ved at ændre invarianten på følgende måde (h er en ny variabel)

$\ll$Initialiser r, q og $h \gg$
do $\{r * h^q = n^p\}$
$q \neq 0 \rightarrow \ll$Opdater r, q og $h \gg$
od

og samtidig observere, at når q er *lige*, kan invarianten opretholdes ved at *kvadrere* h og samtidigt *halvere* q. Den konkrete algoritme kan nu skrives på følgende måde

Algoritme : Logaritmisk potensopløftning
Stimulans : $n, p : p \geq 0$
Respons : $r : r = n^p$
Metode : $r, q, h := 1, p, n$
do $\{r * h^q = n^p\}$
$q \neq 0 \rightarrow$ **if** q ulige $\rightarrow r, q := r * h, q - 1$
| q lige $\rightarrow h, q := h * h, q/2$
fi
od

Argumentet for at invarianten er gyldig kan sammenfattes i følgende to ligninger, som viser, at hvis invarianten er opfyldt inden et gennemløb af løkken, så er den også opfyldt efter dette gennemløb.

q ulige: $r' * h'^{q'} = (r * h) * h^{q-1}$
q lige : $r' * h'^{q'} = r * (h^2)^{q/2}$

Læseren opfordres til at undersøge, hvor mange multiplikationer der udføres af denne algoritme.

2.3 Søgning

Enhver søgning er karakteriseret ved, at man blandt en samling af elementer leder efter ét eller flere med bestemte karakteristika. Samlingen af elementer der ledes i, kalder vi *søgedomænet*, og de elementer der ledes efter kalder vi *målgruppen*.

Vi skal betragte det på samme tid simple og forholdsvis generelle tilfælde, hvor søgedomænet er en *funktion* fra en indeksmængde X ind i en elementmængde E, og hvor målgruppen består af et enkelt element af type E. Vi ønsker at afgøre om dette element forekommer i funktionens billedmængde og i så fald ønsker vi at returnere (et af) dets indeks.

I følgende algoritme er S søgedomænet, m er målelementet, og K afgrænser *kandidatmængden*, dvs. den del af S's indeksmængde, hvis funktionsværdier stadig er målgruppekandidater. Resultatet angives i variablen r, hvis værdi enten er ?–X eller er lig med et indeks x, for hvilket $S(x) = m$.

Vi ønsker at opfatte r som en mængde, der enten er tom eller indeholder et indeks. Vi definerer derfor følgende *abstraktionsfunktion* α

$$\alpha(r) = \begin{cases} \emptyset & \text{hvis } r = ?\text{–X} \\ r & \text{ellers} \end{cases}$$

Ved hjælp heraf kan vi nu skrive den generelle søgealgoritme som følger:

Algoritme : Søgning

Stimulans	:	$S : X \to E$, søgedomæne
		$m : E$, målelement
Respons	:	$r : S(\alpha(r)) = S(S^{-1}(m))$
Metode	:	$\ll$Initialiser r og $K \gg$
		do $\{(\alpha(r) \subseteq S^{-1}(m) \subseteq K)\}$
		$(K \neq \emptyset) \wedge (r = ?\text{–X}) \to \ll$Opdater r og $K \gg$
		od

Denne algoritme er forholdsvis abstrakt og dens anvendelse i konkrete tilfælde forudsætter realiseringer af variablerne S og K. Det er imidlertid interessant, at vigtige egenskaber ved algoritmen kan vises selv på dette abstraktionsniveau. Mere præcist kan vi argumentere for, at hvis algoritmen er gyldig og den standser, så er den også korrekt. Argumentet er som følger.

Hvis algoritmen standser, er den kontrollerende betingelse falsk, dvs.

$$(K = \emptyset) \vee (\alpha(r) \neq \emptyset)$$

og der gælder yderligere at invarianten er opfyldt. Men så er responsudsagnet også opfyldt, fordi

$K = \emptyset$: Responsudsagnet følger trivielt fra invarianten.

$\alpha(r) \neq \emptyset$: Da $\alpha(r) \subseteq S^{-1}(m)$ gælder der $S(\alpha(r)) \subseteq S(S^{-1}(m)) \subseteq \{m\}$, men da $\alpha(r) \neq \emptyset$, må $S(\alpha(r))$ indeholde mindst ét element, hvorfor der også gælder, at $\{m\} \subseteq S(\alpha(r))$.

Vi viser nu to nyttige realiseringer af denne algoritme. I begge tilfælde realiseres de abstrakte variabler på følgende måde

- S er en liste af elementer af type E
- K repræsenteres som et interval lav..høj hvor lav og høj er to *konkrete* variable

Den første realisering er som følger

Algoritme : Lineær søgning

Stimulans : S : **List**(E), søgedomæne
$m : E$, målelement
Respons : $r : S(\alpha(r)) = S(S^{-1}(m))$
Metode : lav,høj,r $:= 0, |S|$,?–Int
do $\{I\}$
$(\text{lav} < \text{høj}) \wedge (r = \text{?–Int}) \rightarrow$
if $S.(\text{lav}) = m \rightarrow r := \text{lav}$
& true $\rightarrow$ lav $:=$ lav $+ 1$
fi
od

Invarianten, I, *den konkrete invariant*, fås ved at indsætte lav..høj for K i den abstrakte invariant ovenfor. Dette giver

$$I \ : \ \alpha(r) \subseteq S^{-1}(m) \subseteq \text{lav..høj}.$$

Det er nemt at se, at de to konkretiseringer af

$$\ll\text{Initialiser } K \text{ og } r \gg \text{ og } \ll\text{Opdater } K \text{ og } r \gg$$

sørger for at holde I opfyldt, dvs. at algoritmen er gyldig. At algoritmen terminerer indses ved at betragte følgende termineringsfunktion

$$\mu(\text{lav,høj,r}) = \text{høj-lav-}|\alpha(r)|$$

Pointen er nu, at når den konkrete algoritme således er gyldig og terminerer, så følger det af argumentet ovenfor (for den abstrakte algoritme), at den konkrete også er korrekt.

I det næste eksempel antager vi, at værditypen E er ordnet og at listen S er ordnet i ikke-aftagende orden, dvs. at S opfylder prædikatet $\forall i, j \in 0..|S| : i \leq j \Rightarrow S.(i) \leq S.(j)$. I dette tilfælde kan vi anvende *binær søgning*, dvs. en søgestrategi der består i at udvælge det midterste element

i kandidatintervallet lav..høj for derefter at bruge ordningen til at halvere kandidaterne, hvis det inspicerede element ikke er det, der ledes efter.

Idet repræsentationen er den samme som før (og vi for simpelheds skyld antager, at E er heltallene), får vi følgende algoritme, hvor invarianten I er den samme som ovenfor.

Algoritme : Binær søgning

Stimulans : S : **List** (Int), ordnet søgedomæne
m : Int, målelement

Respons : $r : S(\alpha(r)) = S(S^{-1}(m))$

Metode : lav,høj,r:= $0, |S|,$?–Int
do $\{I\}$
$(\text{lav} < \text{høj}) \ \wedge \ (r = ?\text{–Int}) \rightarrow$
midt := (lav + høj)/2
if $S.(\text{midt}) = m \rightarrow r := \text{midt}$
$\mid \ S.(\text{midt}) < m \rightarrow \text{lav} := \text{midt} + 1$
$\mid \ m < S.(\text{midt}) \rightarrow \text{høj} := \text{midt}$
fi
od

Det er nemt at vise, at denne algoritme også er gyldig. Med hensyn til terminering kan vi også genbruge termineringsfunktionen fra før. Her er det imidlertid ikke helt oplagt, at den altid aftager. Det detaljerede bevis herfor er som følger

- $S.(\text{midt}) = m$: $\mu(\text{lav}', \text{høj}', r') = \text{høj}' - \text{lav}' - |\alpha(r')| = \text{høj} - \text{lav} - 1$
  $< \text{høj} - \text{lav} = \text{høj} - \text{lav} - |\alpha(r)| = \mu(\text{lav}, \text{høj}, r)$
- $S.(\text{midt}) < m$: $\mu(\text{lav}', \text{høj}', r') = \text{høj}' - \text{lav}' = \text{høj} - (\text{midt} + 1)$
  $= \text{høj} - ((\text{lav} + \text{høj})/2 + 1) \leq \text{høj} - (\text{lav} + 1)$
  $< \text{høj} - \text{lav} = \mu(\text{lav}, \text{høj}, r)$
  fordi $\text{lav} \leq (\text{lav} + \text{høj})/2$ når $\text{lav} < \text{høj}$
- $m < S.(\text{midt})$: $\mu(\text{lav}', \text{høj}', r') = \text{høj}' - \text{lav}' = \text{midt} - \text{lav}$
  $= (\text{lav} + \text{høj})/2 - \text{lav} < \text{høj} - \text{lav} = \mu(\text{lav}, \text{høj}, r)$
  fordi $(\text{lav} + \text{høj})/2 < \text{høj}$ når $\text{lav} < \text{høj}$.

2.4 Fletning

Vi betragter nu en anden problemstilling, hvis løsning også til en vis grad kan "standardiseres". Problemet består i at *flette* to ordnede følger og kan illustreres v.hj.a. følgende eksempel. Fletningen af følgerne

(3,5,9,10,15)

og

(1,5,8,13,13,16)

er følgen

(1,3,5,5,8,9,10,13,13,15,16)

I almindelighed er fletningen af to ordnede følger en ordnet følge, som indeholder præcis de elementer, der findes i de to følger.

Vi betegner fletningen af følgerne f_1 og f_2 med

$$\mathrm{FLET}(f_1, f_2)$$

Vi får nu følgende algoritme for fletning, hvis idé er, at elementerne fra de to følger behandles ét for ét ved et symmetrisk *sekventielt* gennemløb af de to følger.

Algoritme: Fletning
Stimulans : $f_1, f_2 : E^*$, ordnede følger
Respons : $f : E^*,\ f = \mathrm{FLET}(f_1, f_2)$
Metode : $f, g_1, g_2 := \lambda, f_1, f_2$

do $\{f \cdot \mathrm{FLET}(g_1, g_2) = \mathrm{FLET}(f_1, f_2)\}$
$g_1 \neq \lambda \wedge g_2 = \lambda \rightarrow \ll Flyt_1 \gg$
| $g_1 = \lambda \wedge g_2 \neq \lambda \rightarrow \ll Flyt_2 \gg$
| $g_1 \neq \lambda \wedge g_2 \neq \lambda \rightarrow$ **if** $\mathrm{hd}(g_1) \leq \mathrm{hd}(g_2) \rightarrow \ll Flyt_1 \gg$
| $\mathrm{hd}(g_1) \geq \mathrm{hd}(g_2) \rightarrow \ll Flyt_2 \gg$
fi
od

hvor $\ll Flyt_1 \gg$ og $\ll Flyt_2 \gg$ er som følger

$\ll Flyt_1 \gg$ **is** $f, g_1 := f \cdot \mathrm{hd}(g_1), \mathrm{tl}(g_1)$
$\ll Flyt_2 \gg$ **is** $f, g_2 := f \cdot \mathrm{hd}(g_2), \mathrm{tl}(g_2)$

Vi overlader det til læseren at argumentere for at algoritmen er korrekt (argumentet består hovedsageligt i at observere, at invarianten er gyldig).

Følgende TRINE program viser en konkretisering af algoritmen, hvor de indgående følger er lister af heltal. `Sequence.tri` indeholder Boxen Sequence(T) (jfr. afsnit 10.5 i [TRINE]).

```
Process Fletning

(* Programmet indlæser to lister af heltal og kontrollerer
   at de er ordnede i ikke-aftagende orden. Derefter flettes
   de under anvendelse af flettealgoritmen. *)

   (+ @"sequence.tri"

      Box ISq
         Sequence(Int)
      end ISq

      Proc Indlæs[f: Vector]
         (+ Proc check[vec: Vector] → (Bool)
               (+ Var i: Int (* Søgning efter i med vec.(i-1) > vec.(i) *)
                  i := 1
                  do i < | vec | →
                        if vec.(i-1) ≤ vec.(i) → i := i+1
                        & true → return false
                        fi
                  od
                  return true
               +)
            end check

            write("Liste: ")
            read[f]
            do ¬ check[f] →
                  write(eol, "Om igen: ")
                  read[f]
            od
         +)
      end Indlæs
```

```
        Var f1, f2: Vector

        Indlæs[f1]
        Indlæs[f2] (* f1 og f2 er ordnede *)
        (+ Var f, g1, g2: ISq'Seq
           ISq'Con[f](Vector())
           ISq'Con[g1](f1)
           ISq'Con[g2](f2)
           do (* flettealgoritme *)
                (ISq'Len[g1] > 0) ∧ (ISq'Len[g2] = 0 ) → ≪ Flyt1 ≫
            | (ISq'Len[g1] = 0) ∧ (ISq'Len[g2] > 0) → ≪ Flyt2 ≫
            | (ISq'Len[g1] > 0) ∧ (ISq'Len[g2] > 0) →
                if ISq'Ltop[g1] ≤ ISq'Ltop[g2] → ≪ Flyt1 ≫
                 | ISq'Ltop[g1] ≥ ISq'Ltop[g2] → ≪ Flyt2 ≫
                fi
           od
           write(eol, "Resultat: ")
           ISq'Print[f]
        +)
    +)
end Fletning
where ≪ Flyt1 ≫ is
          ISq'Rpush[f](ISq'Ltop[g1])
          ISq'Lpop[g1]

where ≪ Flyt2 ≫ is
          ISq'Rpush[f](ISq'Ltop[g2])
          ISq'Lpop[g2]
```

Der findes en del algoritmer, som "næsten" er fletninger, dvs. de to følger gennemløbes på samme måde som ovenfor, men det er ikke sikkert, at alt kopieres "råt". Her er flettealgoritmen også nyttig, fordi det er nemt at modificere den i det konkrete tilfælde. Et eksempel er en algoritme, der givet to ordnede følger uden "dubletter" finder præcis de elementer, der optræder i dem begge. I dette tilfælde skal algoritmen kun "producere" noget, når hovedet af de to følger er ens, og resten skal ignoreres.

```
Process Snit

(* Programmet indlæser to lister af heltal og kontrollerer
   at de er ordnede i voksende orden. Derefter findes de
   elementer der forekommer i dem begge. *)

   (+ @"sequence.tri"

      Box ISq
         Sequence(Int)
      end ISq

      Proc Indlæs[f: Vector]
         (+ Proc check[vec: Vector] → (Bool)
               (+ Var i: Int (* Søgning efter i med vec.(i-1) > vec.(i) *)
                  i := 1
                  do i < | vec | →
                        if vec.(i-1) < vec.(i) → i := i+1
                        ff true → return false
                        fi
                  od
                  return true
               +)
            end check

            write("Liste: ")
            read[f]
            do ¬ check[f] →
                  write(eol, "Om igen: ")
                  read[f]
            od
         +)
      end Indlæs
```

```
        Var f1, f2: Vector

        Indlæs[f1]
        Indlæs[f2] (* f1 og f2 er voksende *)
        (+ Var f, g1, g2: ISq'Seq
           ISq'Con[f](Vector())
           ISq'Con[g1](f1)
           ISq'Con[g2](f2)
           do (* flettealgoritme *)
                (ISq'Len[g1] > 0) ∧ (ISq'Len[g2] = 0 ) → ≪ Flyt1 ≫
            | (ISq'Len[g1] = 0) ∧ (ISq'Len[g2] > 0) → ≪ Flyt2 ≫
            | (ISq'Len[g1] > 0) ∧ (ISq'Len[g2] > 0) →
                if ISq'Ltop[g1] < ISq'Ltop[g2] → ≪ Flyt1 ≫
                 | ISq'Ltop[g1] > ISq'Ltop[g2] → ≪ Flyt2 ≫
                 | ISq'Ltop[g1] = ISq'Ltop[g2] → ≪ Flyt12 ≫
                fi
           od
           write( "Resultat: ")
           ISq'Print[f]
        +)
    +)
end Snit
where ≪ Flyt1 ≫ is
          ISq'Lpop[g1]

where ≪ Flyt2 ≫ is
          ISq'Lpop[g2]

where ≪ Flyt12 ≫ is
          ISq'Rpush[f](ISq'Ltop[g1])
          ISq'Lpop[g1]
          ISq'Lpop[g2]
```

2.5 Flytning

Som den sidste problemstilling i dette kapitel betragter vi diverse former for *omflytningsalgoritmer*. En omflytningsalgoritme er karakteriseret ved, at den manipulerer elementerne i en struktur v.hj.a. *ombytninger*, dvs. den hverken fjerner eller tilføjer elementer, men nøjes med at flytte rundt på dem. Den struktur, der er relevant for omflytningsalgoritmer er en liste S, som indeholder elementer fra en ordnet mængde E, og som kun kan manipuleres v.hj.a. følgende operationer

$\|S\|$	: antal elementer i S
$S.(i)$	: det i'te element i S
$S.(i) < S.(j)$	: sammenligning af i'te og j'te element i S
$S.(i) :=: S.(j)$	: ombytning af i'te og j'te element i S

De vigtigste eksempler på omflytningsalgoritmer er sorteringsalgoritmer, som på grund af deres udbredte anvendelse i praksis er et af de områder inden for datalogien, der har været undersøgt mest intenst. Vi skal i dette afsnit nøjes med at betragte to af de simpleste og mest ligefremme sorteringsmetoder samt en omflytningsalgoritme, der bl.a. kan bruges som hjælpeprocedure i en mere interessant sorteringsalgoritme.

Det første eksempel er såkaldt *indsættelsessortering*, som kan beskrives på følgende måde (i resten af dette afsnit betyder *sorteret*, at vektoren er sorteret i ikke-aftagende orden)

Algoritme: Indsættelsessortering
Stimulans: S: vektor
Respons : S: sorteret
Metode : $i := 0$
do $\{(S(0..i)$ er sorteret$)\wedge(0 \leq i \leq |S|)\}$
$i < |S| \rightarrow$ $\ll$indsæt $S.(i)$ i $S(0..i) \gg$
$i := i + 1$
od

Det skulle være klart, at hvis denne algoritme er gyldig, så er den også korrekt. Selve indsættelsen kan beskrives som en (baglæns) søgning (ledsaget af en "skubning") efter det sidste element i $S(0..i)$, som er mindre end eller lig med $S.(i)$. Dette kan gøres på følgende måde.

Algoritme : Indsættelsessortering
Stimulans: S: vektor
Respons : S: sorteret
Metode : $i := 0$
do $\{(S(0..i)$ er sorteret$) \wedge (0 \leq i \leq |S|)\}$
$i < |S| \rightarrow$ j, fortsæt $:= i - 1$,true
do $\{$søgning efter $j : S.(j) \leq S.(j+1)\}$
$(0 \leq j) \wedge$ fortsæt $\rightarrow$ **if** $S.(j) \leq S.(j+1) \rightarrow$ fortsæt := false
$\mid S.(j) > S.(j+1) \rightarrow$ $S.(j) :=: S.(j+1)$
$j := j - 1$
fi
od
$i := i + 1$
od

Det andet eksempel på sortering er udvalgssortering, som kan beskrives som følger.

Algoritme : Udvalgssortering
Stimulans : S: vektor
Respons : S: sorteret
Metode : $i := 0$
do $\{(S(0..i)$ er sorteret$) \wedge (S(0..i) \leq S(i..|S|) \wedge (0 \leq i \leq |S|)\}$
$i < |S| \rightarrow$ $\ll$find det mindste element i $S(i..|S|)$ og
ombyt det med $S.(i) \gg$
$i := i + 1$
od

Vi overlader det til læseren at skrive $\ll$find det mindste $\ldots\gg$, og nøjes med at pege på, at uligheden $S(0..i) \leq S(i..|S|)$ betyder, at alle elementer i $S(0..i)$ er mindre end eller lig med alle elementer i $S(i..|S|)$.[1]

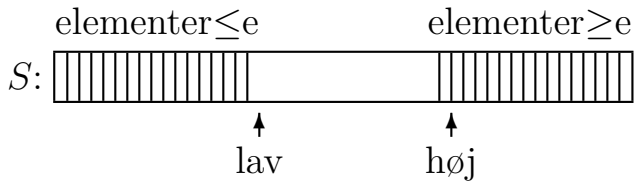
Det sidste eksempel på omflytning er en algoritme, som deler en vektor i to dele på en sådan måde, at alle elementer i den nedre del er mindre eller lig med en værdi e, og alle elementer i den øvre del er større end eller lig med e. Udover at foretage omflytningen, giver algoritmen også besked om, hvor delepunktet ligger.

Algoritmen ser ud som følger.

[1] Den præcise betydning af $S(0..i) \leq S(i..|S|)$ er $\forall j \in 0..i : (\forall k \in i..|S| : S.(j) \leq S.(k))$.

Algoritme : Opdel
Stimulans : S, e: S vektor, e element
Respons : $S, r : (0 \leq r \leq |S|) \wedge (S(0..r) \leq e \leq S(r..|S|))$
Metode : $\text{lav}, \text{høj} := 0, |S|$
do $\{(S(0..\text{lav}) \leq e \leq S(\text{høj}..|S|)) \wedge (0 \leq \text{lav} \leq \text{høj} \leq |S|)\}$
$\text{lav} < \text{høj} \rightarrow$**if** $S.(\text{lav}) \leq e \rightarrow \text{lav} := \text{lav} + 1$
$\mid S.(\text{høj} - 1) < e < S.(\text{lav}) \rightarrow S.(\text{lav}) :=: S.(\text{høj} - 1)$
$\text{lav}, \text{høj} := \text{lav} + 1, \text{høj} - 1$
$\mid e \leq S.(\text{høj} - 1) \rightarrow \text{høj} := \text{høj} - 1$
fi
od
$r := \text{lav}$

Vi kan illustrere invarianten for algoritmen v.hj.a. følgende tegning



Det skulle være nemt at se, at algoritmen er gyldig. Læseren opfordres til at overbevise sig om, at den også er korrekt.

3 Kompleksitet

I de foregående kapitler har vi beskæftiget os med de *kvalitative* aspekter af programmering, dvs. spørgsmål som korrekthed og simpelhed i de anvendte algoritmer. Algoritmer har som bekendt også *kvantitative* aspekter, som det er nødvendigt at beskæftige sig med, og blandt disse er først og fremmest spørgsmålet om, hvor lang tid det tager at udføre dem. Selv om moderne datamater er særdeles hurtige, er der en mangfoldighed af problemer, hvor det er afgørende, at det program, der løser problemet, er effektivt. Der er naturligvis mange eksempler på sådanne programmer, blandt hvilke såkaldte *realtidsprogrammer* er nogen af de bedste. Et sådant program har til formål at reagere på påvirkninger *inden for en bestemt maksimal tid*, og det er stærkt generende, måske ligefrem meningsløst, hvis det ikke sker. Et eksempel på et meningsløst program af denne type er et lønudbetalingssystem for ugelønnede, som bruger en måned på at gennemføre de nødvendige beregninger, eller (måske mere realistisk) en simulator til træning af piloter, som er flere minutter om at beregne konsekvenserne af en pilotreaktion, hvis resultat indfinder sig på sekunder i en rigtig flyvemaskine.

Omend programmers *pladsforbrug* ofte er lige så afgørende en faktor som deres tidsforbrug, skal vi i dette afsnit indskrænke os til at betragte sidstnævnte. Vi definerer *tidskompleksiteten* af et program eller en algoritme, som det antal *primitive operationer*, der udføres fra programmet (eller algoritmen) startes i en eller anden starttilstand og til det standser (hvis det da overhovedet standser).

Ved en primitiv operation vil vi forstå en operation, som med rimelighed kan siges at udgøre en tidsmæssig enhed, når et program afvikles på en datamaskine. Der er flere muligheder for at lægge sig fast på en sådan enhed, men det viser sig (jfr. kapitel 17 i [TRINE]), at disse alle mere eller mindre svarer til at definere en primitiv operation som en læsning, skrivning eller kopiering af enten en *simpel værdi* eller af en *reference* til en variabel (standardværdier og pointerværdier opfattes i denne sammenhæng også som simple værdier). Tidskompleksiteten af f.eks. et Trine program defineres således som det samlede antal læsninger, skrivninger og kopieringer af standardværdier, pointerværdier, værdier fra Unit, Int, Bool, Char, Real eller af variabelreferencer under programmets udførelse.

At opgøre denne tidskompleksitet eksakt er imidlertid, selv for små programmer, en overvældende sag, og vi skal derfor indskrænke os til at betragte dens *størrelsesorden*. Betimeligheden heraf er der argumenteret indgående for i bl.a. [Bentley] – her skal vi præcisere den notation, der bruges til at tale om størrelsesordener, ligesom vi skal se lidt på teknikker til at vurdere tidskompleksitet.

3.1 Notation for størrelsesorden

I det følgende betegner $\mathbb{R}_0$ ($\mathbb{R}_+$) de ikke-negative (positive) reelle tal. Vi betragter funktioner af formen

$$f : \mathbb{N}_0 \to \mathbb{R}_0$$

og definerer for en vilkårlig sådan funktion følgende klasse af funktioner

$$O(f) = \{g : \mathbb{N}_0 \to \mathbb{R}_0 \mid \exists k \in \mathbb{R}_+, n_0 \in \mathbb{N} : \forall n \geq n_0 : g(n) \leq kf(n)\}$$

$O(f)$ er klassen af funktioner, der *asymptotisk* (dvs. for store værdier af n) er opadtil begrænset af f "pånær en konstant faktor". Følgende er eksempler på anvendelse af notationen

$$4n^2 + 3n - 13 \in O(n^2)$$
$$\tfrac{1}{3}n + \sqrt{n} \in O(n)$$
$$35n + n(\sqrt{n} + 7) \in O(n^2)$$
$$2^n + 4n^4 \in O(2^n)$$

Det er klart, at vi altid vil være interesseret i at angive den mindst mulige begrænsende funktion. I den forbindelse skal vi lejlighedsvis også anvende følgende notation, som i analogi med $O(f)$ angiver funktioner, der er nedadtil begrænset.

$$\Omega(f) : \{g : \mathbb{N}_0 \to \mathbb{R}_0 \mid \exists k \in \mathbb{R}_+, n_0 \in \mathbb{N} : \forall n \geq n_0 : g(n) \geq kf(n)\}$$

Vi kan også tale om klassen af funktioner, der begrænses af f fra begge sider

$$\Theta(f) = O(f) \cap \Omega(f)$$

Vi skal ikke beskæftige os særlig meget med regneregler for disse funktionsklasser, men følgende er nyttige at kende.

Lad $g_1 \in O(f_1), g_2 \in O(f_2)$. Der gælder da

- $kg_1 \in O(f_1)$ for alle $k \geq 0$
- $g_1 \cdot g_2 \in O(f_1 \cdot f_2)$
- $g_1 + g_2 \in O(f_1 + f_2)$
- $f_1 \in O(f_2) \Rightarrow g_1 \in O(f_2)$

3.2 Definition af tidskompleksitet

Det fremgår af indledningen, at når tidskompleksiteten af et program eller en algoritme skal vurderes, skal man vurdere antallet af simple værdier og referencer, der "behandles" under programmets (algoritmens) udførelse. Det er nemt at indse, at denne "behandling" sker i forbindelse med

- tilordningssætninger
- betingelser
- kommunikationssætninger
- procedurekald
- variabelerklæringer

Vi vil derfor være godt hjulpet med at analysere omkostningerne for hver af disse programelementer. Her kan der være variationer fra sprog til sprog, og følgende diskussion handler derfor om TRINE.

TRINE's implementation på en fysisk datamat er beskrevet i kapitel 17 i [TRINE], og der er her argumenteret for at det vi er interesseret i, er summen af *størrelserne* af de værdier, der **udregnes** under afviklingen af programmet eller algoritmen. Størrelsen af en TRINE værdi defineres som følger (jfr. igen [TRINE])

- størrelsen af en simpel værdi er 1
- størrelsen af en sekvens er 2 + summen af størrelserne af dens elementer

Følgende eksempler illustrerer denne definition

- størrelsen af Int-konstanten 17 er 1
- størrelsen af List-konstanten Vector (0|100) er 102
- størrelsen af Prod-konstanten Point(7,0) er 4
- størrelsen af Sum-konstanten Res(x:7) er 4

Baseret på dette størrelsesmål, og på den tidligere nævnte rimelighed af at betragte manipulation af en simpel værdi eller reference som en primitiv operation, kan vi nu præcisere tidskompleksiteten af et TRINE program eller algoritme på følgende måde.

> Tidskompleksiteten af et program (en algoritme) i TRINE er summen af størrelserne af de værdier, der udregnes under udførelsen af programmet (algoritmen) med en starttilstand, der opfylder stimulansprædikatet. Såfremt der findes flere sådanne udførelser, er tidskompleksiteten maksimum over samtlige udførelser af summen af størrelserne.

3.3 Direkte analyse

Vi betragter nu nogle eksempler på vurdering af tidskompleksitet. Af bekvemmelighedsårsager skal vi benytte notationen $T[\![A]\!]$ til at angive tidskompleksiteten af algoritmen (programmet A).

Betragt algoritmen til potensopløftning fra side 10.

Algoritme : Lineær potensopløftning
Stimulans : $n, p : p \geq 0$
Respons : $r : r = n^p$
Metode : $r, q := 1, p$
do $\{r * n^q = n^p\}$
$q \neq 0 \rightarrow r, q := r * n, q - 1$
od

Det er klart, at alle værdier, der udregnes under udførelse af denne algoritme, har størrelse 1, og da løkken gennemløbes p gange, følger det at

$$T[\![\text{Lineær potensopløftning}]\!] \in \Theta(p)$$

Det er også klart, at af de to stimulans-variabler til Lineær potensopløftning, n og p, er det p der betyder noget for kompleksiteten, medens værdien af n er ligegyldig. I sådanne situationer skal vi fremhæve denne afhængighed eksplicit ved at tilføje den relevante parameter til $T[\![]\!]$ på følgende måde

$$T[\![\text{Lineær potensopløftning}]\!](p) \in \Theta(p)$$

Betragt nu som et andet eksempel på denne parametrisering følgende algoritmestump

$$\begin{array}{lll} A_1\text{:} & i := 1 & \\ & \textbf{do} & \\ & \quad i \leq n \rightarrow & j := 1 \\ & & \textbf{do} \\ & & \quad j \leq i \rightarrow j := j + 1 \quad \Big\} \; A_2 \\ & & \textbf{od} \\ & & i := i + 1 \\ & \textbf{od} & \end{array}$$

Det er klart, at kompleksiteten af A_1 afhænger af n, og at kompleksiteten af A_2 afhænger af i. Dette kan tydeliggøres på følgende måde. Det er klart at der findes en konstant k_1, så

$$T[\![A_1]\!](n) \leq k_1 * n + T[\![A_2]\!](1) + \cdots + T[\![A_2]\!](n)$$

og der findes også en konstant k_2

$$T[\![A_2]\!](i) \leq k_2 * i$$

men så er

$$\begin{array}{rcl} T[\![A_1]\!](n) & \leq & k_1 * n + k_2 * \sum_{i=1}^{n} i \\ & \leq & k_1 * n + k_2 * n^2 \\ & \in & O(n^2) \end{array}$$

Antag nu, at tilordningssætningen $i := i+1$ erstattes af sætningen $i := 2*i$. I så fald udføres der et antal iterationer i den yderste **do**-sætning, der er lig med det antal gange man skal multiplicere 1 med 2 for "at nå" n, dvs. $\log(n)$ gange[2]. Vi får da

$$\begin{aligned} T[\![A_1]\!] &\approx T[\![A_2]\!](1) + \ldots + T[\![A_2]\!](2^{\log(n)}) \\ &\approx 1 + \ldots + 2^i + \ldots + 2^{\log(n)} \\ &\approx 2n \\ &\in O(n) \end{aligned}$$

hvor vi har brugt $\approx$ til at angive, at vi regner med de tilnærmelser $O(\cdot)$ tillader.

Vi kan præcisere denne brug af parametre i kompleksitetsmålet på følgende måde. Hvis A er en (stump af) en algoritme, betegner vi med

$$T[\![A]\!](n)$$

tidskompleksiteten af A, når algoritmen startes i en tilstand, der er bestemt af den *karakteristiske parameter* n. Mere præcist skal der gælde, at $T[\![A]\!](n)$ skal være den mindste øvre grænse for summen af størrelserne af de værdier, der udregnes under udførelser, der starter i tilstande, som er karakteriseret af n. Vi skal ikke forsøge at formalisere hvad det i almindelighed betyder, at en tilstand er karakteriseret af en parameter, idet dette vil blive præciseret i hvert enkelt tilfælde. I de fleste tilfælde angiver n værdien af en af algoritmens variabler (som p og i ovenfor), og i så fald karakteriserer den enhver tilstand i hvilken den pågældende variabel har værdien n.

Det skal understreges, at kompleksitetsmålet er pessimistisk, idet $T[\![A]\!](n)$ dominerer tidskompleksiteten for *alle* udførsler af A i alle starttilstande, der karakteriseres af n. Dette kompleksitetsmål kaldes også *worst-case kompleksitet*, fordi det giver udførelsestiden for den værst tænkelige beregning. Et alternativt kompleksitetsmål er *gennemsnitlig* tidskompleksitet, hvor man under mere eller mindre realistiske antagelser om fordelingen af data,

[2]Alle logaritmer i disse noter har, med mindre andet nævnes eksplicit, grundtal 2, dvs. $\log(n) = \log_2(n)$.

udregner det forventede antal primitive operationer, der udføres i en beregning. Vi skal dog hovedsageligt interessere os for worst-case kompleksitet.

3.4 Analyse af nogle konkrete algoritmer

Vi anfører nu et antal eksempler på simple analyser af nogle af algoritmerne fra de foregående afsnit.

I algoritmen Logaritmisk potensopløftning er det igen klart, at alle forekommende værdier har størrelse 1, hvorfor vi kan vurdere dens tidskompleksitet ved antallet af iterationer i **do**-løkken. Her er det nemt at se, at variablen q halveres mindst hver anden gang løkken gennemløbes, hvorfor antallet af iterationer ligger mellem $\log(p)$ og $2\log(p)$. Altså gælder der

$$T[\![\text{Logaritmisk potensopløftning}]\!](p) \in \Theta(\log(p))$$

I algoritmen Binær søgning side 14 sker der en halvering af længden af søgeintervallet i hver iteration, hvorfor vi igen har (n er antallet af elementer i søgedomænet)

$$T[\![\text{Binær søgning}]\!](n) \in \Theta(\log(n))$$

For algoritmen Indsættelsessortering side 21 gælder der (n er længden af w og "søgning" angiver den inderste **do**-løkke).

$$\begin{aligned} T[\![\text{Indsættelsessortering}]\!](n) &\approx \sum_{i=1}^{n} T[\![\text{søgning}]\!](i) \\ &\in \sum_{i=1}^{n} O(i) \\ &= O(n^2) \end{aligned}$$

Vi viser nu, at disse grænser er "skarpe", dvs. at man kan erstatte O med Θ. For Indsættelsessortering skal vi altså vise, at

(*) $\quad T[\![\text{Indsættelsessortering}]\!](n) \in \Omega(n^2)$

Iflg. definitionen på tidskompleksitet skal vi finde en starttilstand for algoritmen, som er karakteriseret af n, og hvor der under udførelsen udregnes af størrelsesordenen n^2 (simple) værdier. Da n betegner længder af S, skal vi finde en liste af længde n, for hvilken algoritmens udførelsestid er ca. n^2. Men det er en simpel sag at indse, at indsættelsessortering af den omvendt sorterede liste

$$S = (n, n-1, n-2, \ldots, 2, 1)$$

involverer $\frac{(n-1)*n}{2}$ ombytninger, og derfor er det klart, at (*) er opfyldt.

Det overlades til læseren på tilsvarende måde at vise, at

$$T[\![\text{Udvalgssortering}]\!](n) \in \Omega(n^2)$$

3.5 Analyse af en binær tæller

I dette afsnit vil vi analysere en algoritme, der simulerer optælling i et tælleregister R, der svarer til en *binær* kilometertæller. Vi vil benytte en metode til vurdering af tidskompleksitet, som kan give skarpere analyser end "lige ud ad landevejen" regningerne fra de foregående afsnit. Metoden består i en slags debet-kredit regnskab, og vil i kapitel 9 blive formaliseret og generaliseret.

Algoritme: Optælling
Stimulans : $n : n > 0$
Respons : $R : \tilde{R}$ er den binære repræsentation af n
Metode : $i, R := 0$, Vector( )
do $\{\tilde{R}$ er den binære repræsentation af i og $(0 \leq i \leq n)\}$
$i < n \rightarrow p := 0$
do $(p < |R|) \wedge (R.(p) = 1) \rightarrow$
$R.(p), p := 0, p + 1$
od
if $p = |R| \rightarrow R := R + +\text{Vector}(0)$ **fi**
$R.(p) := 1$
$i := i + 1$
od

Hvis f.eks. n har værdien 14, vil algoritmen standse med følgende værdi for R

$$R = (0, 1, 1, 1)$$

hvilket passer med, at den binære repræsentation af 14 er 1110.

En grov analyse af algoritmen giver, at den yderste **do**-løkke gennemløbes præcis n gange. For hvert gennemløb

- udføres der højst $|R|$ gennemløb af den inderste **do**-løkke
- udføres sætningen $R := R + +\text{Vector}(0)$ eventuelt.

Da $R := R + +\text{Vector}(0)$ er den eneste sætning i algoritmen, hvor der udregnes en værdi hvis størrelse er forskellig fra 1, og da denne størrelse er $|R| + 1$, er det klart at

$$T[\![\text{Optælling}]\!](n) \in O(n \log(n))$$

idet $|R|$ højst kan blive $\log(n)$.

M.h.t. den "dyre" sætning $R := R + +\text{Vector}(0)$ er dette imidlertid alt for pessimistisk, fordi den faktisk udføres ret sjældent. Man kan godt udregne hvor sjældent, men tilbage står stadig at estimere det samlede antal gennemløb af den inderste **do**-løkke. Her anvender vi nu debet-kredit teknikken på følgende måde.

Vi vedtager, at det koster 1 krone hver gang vi ser på et element i R, og at det koster $|R|$ kroner, hver gang R udvides (ved udførelse af sætningen $R := R + +\text{Vector}(0)$). Vi kan nu "finansiere" hele algoritmen ved for hver optælling at *betale* 2 kroner samt ved for hver forlængelse af R at *låne* $|R|$ kroner.

De 2 kroner der betales for hver optælling er nemlig tilstrækkelige til at overholde følgende *bogførings-invariant*:

"Hvert element i R med værdi 1 er i besiddelse af 1 krone."

Dette indses ved at observere, at en optælling består i at gennemløbe et antal elementer i R, der alle har værdien 1, indtil man når et 0. 1'erne sættes

til 0 og 0'et til 1. Hvis nu R allerede overholder bogføringsinvarianten, kan alle 1'erne *selv* betale prisen for at besøge dem. De 2 kroner kan så bruges til dels at betale for at besøge 0'et, og dels til at give denne (som jo nu bliver en 1'er) den krone, den skal have for at overholde bogføringsinvarianten.

Når algoritmen standser, har de samlede udgifter været

- 2n kroner for optællingerne
- den gæld der er opbygget i forbindelse med forlængelserne af R.

R er imidlertid vokset med 1 element ad gangen, fra længde 0 op til slutværdien $\log(n)$, hvorfor gælden andrager

$$\left(\sum_{i=1}^{\log(n)} i \right) \approx \log(n)^2$$

De samlede udgifter beløber sig da til $\approx\ 2n + \log(n)^2$ kroner, og da antallet af kroner er proportionalt med summen af de udregnede værdier, er tidskompleksiteten givet ved

$$T[\![\text{Optælling}]\!](n) \in \Theta(n)$$

dvs. algoritmen er lineær.

Det skal understreges, at denne bogføringsinvariant er en tænkt invariant, som kun anvendes i forbindelse med analysen, og som ikke direkte har noget at gøre med invarianter der anvendes til korrekthedsargumenter.

3.6 Tidskompleksitet af procedurer

I dette og næste afsnit vil vi se på teknikker til at vurdere tidskompleksiteten af procedurer. Vi starter med ikke-rekursive procedurer.

Hvis P er en procedure, betegner vi med

$$T[\![\textbf{proc } P]\!](n)$$

tidskompleksiteten af at udføre P's *krop*[3], startende i en tilstand, der er karakteriseret af n.

Betragt som første eksempel følgende to procedurer til binær søgning (jfr. side 14), hvor det antages at argumentet S er en ordnet liste af længde n.

```
Proc Binær1(S: Vector, m: Int) → (Int)
    (+ Var lav, høj, midt: Int
       lav, høj := 0, | S |
       do lav<høj →
            midt := (lav+høj)/2
            if S.(midt) = m → return midt
            | S.(midt)<m → lav := midt+1
            | m<S.(midt) → høj := midt
            fi
       od
       return ?-Int
    +)
end Binær1

Proc Binær2[S: Vector](m: Int) → (Int)
    (+ Var lav, høj, midt: Int
       lav, høj := 0, | S |
       do lav<høj →
            midt := (lav+høj)/2
            if S.(midt) = m → return midt
            | S.(midt)<m → lav := midt+1
            | m<S.(midt) → høj := midt
            fi
       od
       return ?-Int
    +)
end Binær2
```

Det følger af analysen af Binær søgning, at

$$T[\![\textbf{proc Binær}_1]\!](n) = T[\![\textbf{proc Binær}_2]\!](n) \in O(\log(n))$$

[3]Omkostningerne ved parametermanipulation tælles altså ikke med her.

Hvis vi derimod ser på kompleksiteten af *kaldene*

$$\text{Binær}_1(S, x) \text{ og } \text{Binær}_2[S](x)$$

er der forskel, fordi der gælder

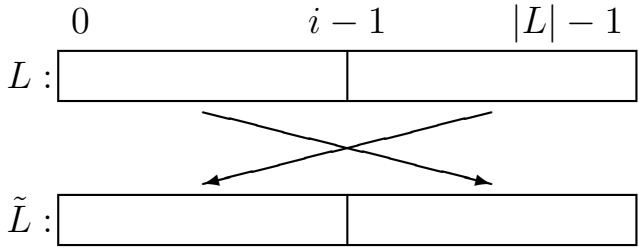
$$\begin{aligned} T[\![\text{Binær}_1(S,x)]\!] &\approx |S| + T[\![\textbf{proc } \text{Binær}_1]\!](|S|) \in O(|S|) \\ T[\![\text{Binær}_2[S](x)]\!] &\approx 1 + T[\![\textbf{proc } \text{Binær}_2]\!](|S|) \in O(\log(|S|)) \end{aligned}$$

Dette skyldes, at for en *værdiparameter* (S) er det som bekendt selve værdien (af størrelse $|S|$) der skal udregnes, medens det for *referenceparameteren* $[S]$ kun er referencen til den relevante variabel (af størrelse 1), der skal findes.

Vi ser altså et eksempel på, hvordan en uhensigtsmæssig parametermekanisme kan "ødelægge" en effektiv søgemetode.

3.7 Tidskompleksitet af rekursive procedurer

Når procedurer er rekursive, kan vi ikke udregne deres tidskompleksitet helt på samme måde som ovenfor. Betragt som eksempel følgende to procedurer, som spejler en liste L. Ideen i begge procedurer er, at der er følgende sammenhæng mellem L og dens spejlbillede $\tilde{L}$ (i er en vilkårlig værdi med $0 \leq i \leq |L|$).



Man kan spejle L ved at dele den i to, spejle hver del for sig og sætte dem sammen i omvendt rækkefølge – eller udtrykt mere kompakt

$$\tilde{L} = \widetilde{L(i..|L|)} + + \widetilde{L(0..i)}$$

Forskellen på de to procedurer er valget af i

```
Type liste = List(Char)

Proc Reverse1[L: liste]
    (+ Var hlp: liste
       if |L|>1 →
            hlp, L := L(0..1), L(1..|L|)
            Reverse1[L]
            L := L++hlp
       fi
    +)
end Reverse1

Proc Reverse2[L: liste]
    (+ Var hlp: liste
       if |L|>1 →
            hlp, L := L(0..|L|/2), L (|L|/2..|L|)
            Reverse2[hlp]
            Reverse2[L]
            L := L++hlp
       fi
    +)
end Reverse2
```

Da begge procedurer er rekursive, resulterer deres analyse i *rekursionsligninger* på følgende måde. For Reverse_1 gælder der, idet n betegner længden af parameteren L,

$$\begin{aligned} T[\![\textbf{proc } \text{Reverse}_1]\!](1) &\in O(1) \\ T[\![\textbf{proc } \text{Reverse}_1]\!](n) &\approx n + T[\![\text{Reverse}_1[L]]\!](n-1) + n \\ &\approx n + T[\![\textbf{proc } \text{Reverse}_1]\!](n-1) \end{aligned}$$

hvilket følger af, at størrelsen af værdierne af $L(1..|L|)$ og L++liste(hlp) begge tilhører $O(n)$.

Det er nemt at se, at løsningen til denne rekursionsligning er

$$\begin{aligned} T[\![\textbf{proc } \text{Reverse}_1]\!](n) &\in \sum_{i=1}^{n} O(i) \\ &= O(n^2) \end{aligned}$$

For proceduren Reverse_2 gælder der følgende analyse (for $n > 1$)

$$\begin{aligned} T[\![\textbf{proc } \text{Reverse}_2]\!](n) &\approx n + \\ &\quad T[\![\text{Reverse}_2[hlp]]\!](\frac{n}{2}) + \\ &\quad T[\![\text{Reverse}_2[L]]\!](\frac{n}{2}) + \\ &\quad n \\ &\approx 2(n + T[\![\textbf{proc } \text{Reverse}_2]\!](\frac{n}{2})) \\ &\approx \underbrace{2(n + 2(\frac{n}{2} + 2(\frac{n}{4} + \ldots + 2(1)\ldots)))}_{\log n} \\ &= 2(n + n + \ldots + n) \\ &\in O(n \log(n)) \end{aligned}$$

idet n "højst kan halveres $\log(n)$ gange". Bemærk, at det følger af analysen, at for store værdier af n er Reverse_2 væsentlig mere effektiv end Reverse_1 (der er stor forskel på n^2 og $n \log(n)$, når n er stor). Det kan altså betale sig at opdele problemet i to problemer, som hver er af halv størrelse, i stedet for at dele det i et stort og et lille.

Lad os for god ordens skyld også analysere proceduren KochLine fra afsnit 12.3 i [TRINE]. Idet den karakteristiske parameter denne gang er ordenen af den kurve der skal tegnes, skulle det være klart, at vi får følgende ligning

$$T[\![\text{KochLine}(n, r, l)]\!] \approx T[\![\textbf{proc } \text{KochLine}]\!](n)$$

samt at

$$T[\![\textbf{proc}\ \text{KochLine}]\!](0) \in O(1)$$
$$T[\![\textbf{proc}\ \text{KochLine}]\!](n) \approx 4 * T[\![\textbf{proc}\ \text{KochLine}]\!](n-1)$$

hvilket giver at

$$T[\![\textbf{proc}\ \text{KochLine}]\!](n) \in O(4^n)$$

4 Specifikationer

Når algoritmer opbygges v.hj.a. ubestemte stumper, herunder når der optræder procedurer i algoritmerne, bliver der behov for at få *egenskaber* ved sådanne ubestemte stumper ind i korrekthedsbeviserne.

Betragt som eksempel følgende situation, hvor K er en mængde af elementer af en eller anden type E.

Algoritme: Permutation
Stimulans: f: **List**(E), $\forall i, j \in 0..|f| : f.(i) \neq f.(j)$
Respons : f: permutation af listen
Metode : $K, i := \emptyset, 0$
do $i < |f| \rightarrow e, i := f.(i), i + 1$
$\ll$indsæt e i $K \gg$
od
$i := 0$
do $i < |f| \rightarrow \ll$udvælg e fra $K \gg$
$\ll$fjern e fra $K \gg$
$f.(i), i := e, i + 1$
od

Hvis vi skal vise, at denne algoritme er korrekt, er det klart, at vi skal bruge stumpernes egenskaber. Disse udtrykker vi nu v.hj.a. *specifikationer* på følgende måde

(*) $\ll$indsæt e i $K \gg$ **sat** $K' = K \cup \{e\}$

sat er en forkortelse for **satisfies** og angiver, at stumpen $\ll$indsæt e i $K \gg$ *opfylder* udsagnet $K' = K \cup \{e\}$. Dette udsagn adskiller sig fra de udsagn vi hidtil har beskæftiget os med ved at indeholde såvel mærkede (K') som umærkede (K, e) variabelnavne. Meningen er den samme som i [TRINE, kap. 8], at de umærkede (mærkede) navne refererer til variablernes værdier *før* (*efter*) udførelsen af stumpen. (*) siger således, at værdien af variablen K efter udførelsen er lig med foreningsmængden af $\{e\}$ og værdien af K før kaldet.

Specifikationen af $\ll$fjern e fra $K \gg$ er analog, hvorimod $\ll$udvælg e fra $K \gg$ har udseendet

$$\ll\text{udvælg } e \text{ fra } K \gg \textbf{sat } K \neq \emptyset \rightarrow (e' \in K) \wedge (K' = K)$$

Her har vi tilføjet en *betingelse* ($K \neq \emptyset$), som angiver under hvilke omstændigheder det giver mening at udføre stumpen.

Vi angiver nu den præcise betydning af en specifikation. En sådan er i almindelighed af formen

(**) $\quad \ll\text{Stump}\gg \textbf{sat } P \rightarrow U$

hvor der gælder, at P kun indeholder umærkede variabelnavne og U (kan) indeholde såvel umærkede som mærkede navne. Vi kalder U et *blandet udsagn.*

I afsnit 2.1 blev det defineret, at en tilstand σ opfylder et prædikat U, hvis U (som jo er en logisk funktion) udregnet i tilstanden σ, har værdien $\underline{t}$. I analogi hermed definerer vi, at et par af tilstande (σ, σ') opfylder et blandet prædikat U, hvis U har værdien $\underline{t}$, når det udregnes i tilstandene σ og σ', hvilket mere præcist betyder, at umærkede (mærkede) variabelnavne refererer til σ (σ').

Den formelle betydning af specifikationen (**) er nu som følger

> For enhver tilstand σ som opfylder P gælder, at enhver udførelse af $\ll$Stump$\gg$ med σ som starttilstand terminerer, og for enhver sluttilstand σ' gælder, at (σ, σ') opfylder U.

Med denne definition skulle det være klart, at

$$\ll\text{Stump}\gg \textbf{sat } P \rightarrow U$$

mekanismen ligger meget tæt på den hidtidige algoritmespecifikationsmekanisme. Ved at betragte definitionen på korrekthed af en algoritme side 9, er det nemt at se, at en algoritme af formen

Algoritme: Noget
Stimulans : P
Respons : U
Metode : $\ll$Noget$\gg$

er korrekt hvis og kun hvis der gælder

$$\ll\text{Noget}\gg \textbf{ sat } P \to U'$$

hvor U' er identisk med U bortset fra, at alle variabelnavne er blevet mærket.

Specifikationernes teknik til at udtale sig om sammenhængen mellem værdierne af variabler før og efter udførelsen af algoritmestumper muliggør opstillingen af følgende *bevisprincip*, der giver grundlaget for såvel konstruktion som korrekthedsbeviser for den type algoritmer, vi har beskæftiget os med i de foregående to kapitler. Her er $\bar{x} = x_1, \ldots, x_n$ programmets variabler, I er invarianten og μ termineringsfunktionen.

Bevisprincip for do-sætninger

Hvis der for sætningen

$$\textbf{do } B \to \ll \text{Krop} \gg \textbf{ od}$$

gælder

$$\ll \text{Krop} \gg \textbf{ sat } (B(\bar{x}) \wedge I(\bar{x})) \to I(\bar{x}') \wedge (\mu(\bar{x}) > \mu(\bar{x}') \geq 0)$$

så gælder der også

$$\textbf{do } B \to \ll \text{Krop} \gg \textbf{ od sat } I(x) \to I(\bar{x}') \wedge \neg B(\bar{x}')$$

Præcis ligesom vi udtrykker ubestemte stumpers egenskaber ved specifikationer, kan vi også udtrykke egenskaber ved procedurer. Dette beskrives nærmere i kapitel 10.

5 Stakke og køer

I dette og de følgende tre kapitler ser vi på de mest fundamentale *datastrukturer*.

Vi minder om, at en *datatype* er defineret til at være en værdimængde, der er udstyret med en samling karakteristiske værdi- og variabeludtryk.

En *datastruktur* vil vi i denne sammenhæng definere til at være en konkret implementation af en abstrakt datatype. Forskellige datastrukturer for den samme datatype vil føre til forskellige effektivitetsegenskaber.

I adskillige tilfælde vil vi observere, at de indlysende implementationer giver *lineære* udførelsestider, men at vi ved at introducere *træstrukturer* kan opnå *logaritmiske* tider.

I kapitel 2 er der vist et eksempel på anvendelsen af en abstraktionsfunktion (α), der forbinder værdien af en konkret variabel i en algoritme (r) med dens abstrakte betydning (en mængde, der evt. kan være tom). Vi har i princippet også brug for abstraktionsfunktioner når vi angiver specifikationer af datstrukturernes operationer men af hensyn til at holde symbolmængden under kontrol vil disse kun forekomme implicit i det følgende. En specifikation som

$$\textbf{Insert}[P](x) \;\; \textbf{sat} \;\; P' = P \cup \{x\}$$

skal således læses som

$$\textbf{Insert}[P](x) \;\; \textbf{sat} \;\; \alpha(P') = \alpha(P) \cup \{\alpha(x)\}$$

Denne oversættelse er altid entydig, da programvariabler kun giver mening i *abstraheret* form (dvs. som argument til α) til højre for **sat**.

5.1 Stakke

Vi er givet en elementtype E. En *stak* over E er en datatype, hvis værdimængde er følger af E-værdier. Lad S være en stakvariabel, og lad x være af type E. Så har stakken følgende udtryk

$$
\begin{aligned}
\textbf{Init}[S] \ \textbf{sat} \ & S' = () \\
\textbf{Push}[S](x) \ \textbf{sat} \ & S = (e_0, e_1, \ldots, e_{n-1}) \rightarrow S' = (e_0, e_1, \ldots, e_{n-1}, x) \\
\textbf{Pop}[S, x] \ \textbf{sat} \ & (S = (e_0, e_1, \ldots, e_{n-1})) \wedge (n > 0) \rightarrow \\
& \qquad (S' = (e_0, e_1, \ldots, e_{n-2})) \wedge (x' = e_{n-1}) \\
\textbf{Empty}[S] \ \textbf{sat} \ & (S' = S) \wedge (\text{Empty}' = (S = ()))
\end{aligned}
$$

En stak med fast øvre størrelse kan implementeres ved hjælp af en liste, så alle operationer finder sted i optimal tid. En polybox, der implementerer en stak med plads til N elementer, ser ud som følger

```
Box S(Element)
    Type Elist = List(Element)
    Type Stack = Prod(high: Int, data: Elist)

    Proc Init[S: Stack](n: Int)
        S := Stack(0, Elist(?-Element | n))
    end Init

    Proc Push[S: Stack](x: Element)
        if S.high<|S.data| →
            S.data.(S.high) := x
            S.high := S.high+1
        fi
    end Push

    Proc Pop[S: Stack, x: Element]
        if S.high>0 →
            S.high := S.high-1
            x := S.data.(S.high)
        fi
    end Pop

    Proc Empty[S: Stack] → (Bool)
        return S.high = 0
    end Empty
```

end S

Vi får med denne datastruktur følgende kompleksiteter

$$\begin{aligned} T[\![\mathbf{Init}[S](N)]\!] &\in O(N) \\ T[\![\mathbf{Push}[S](x)]\!] &\in O(1) \\ T[\![\mathbf{Pop}[S](x)]\!] &\in O(1) \\ T[\![\mathbf{Empty}[S]]\!] &\in O(1) \end{aligned}$$

5.2 Køer

En *kø* over en elementtype E er en datatype, hvis værdimængde er følger af E-værdier. Lad Q være en køvariabel, og lad x være af type E. Så har køen følgende udtryk

$$\begin{aligned} \mathbf{Init}[Q] \ \mathbf{sat} \ & Q' = () \\ \mathbf{Enter}[Q](x) \ \mathbf{sat} \ & Q = (e_0, e_1, \ldots, e_{n-1}) \rightarrow Q' = (e_0, e_1, \ldots, e_{n-1}, x) \\ \mathbf{Remove}[Q, x] \ \mathbf{sat} \ & (Q = (e_0, e_1, \ldots, e_{n-1})) \wedge (n > 0) \rightarrow \\ & \qquad (Q' = (e_1, \ldots, e_{n-1})) \wedge (x' = e_0) \\ \mathbf{Empty}[Q] \ \mathbf{sat} \ & (Q' = Q) \wedge (\mathrm{Empty}' = (Q = ())) \end{aligned}$$

En kø med fast øvre størrelse kan implementeres ved hjælp af en cyklisk liste, så alle operationer finder sted i optimal tid. En polybox, der implementerer en kø med plads til N elementer, ser ud som følger.

```
Box Q(Element)
    Type Elist = List(Element)
    Type Queue = Prod(low, high, size: Int, data: Elist)

    Proc Init[Q: Queue](n: Int)
        Q := Queue(0, 0, 0, Elist(?-Element | n))
    end Init

    Proc Enter[Q: Queue](x: Element)
        if Q.size<| Q.data | →
            Q.data.(Q.high) := x
            Q.high := (Q.high+1) mod | Q.data |
            Q.size := Q.size+1
        fi
    end Enter

    Proc Remove[Q: Queue, x: Element]
        if Q.size>0 →
            x := Q.data.(Q.low)
            Q.size := Q.size-1
            Q.low := (Q.low+1) mod | Q.data |
        fi
    end Remove

    Proc Empty[Q: Queue] → (Bool)
        return Q.size = 0
    end Empty
end Q
```

Vi får med denne datastruktur følgende kompleksiteter

$$
\begin{aligned}
T[\![\textbf{Init}[Q](N)]\!] &\in O(N) \\
T[\![\textbf{Enter}[Q](x)]\!] &\in O(1) \\
T[\![\textbf{Leave}[Q,x]]\!] &\in O(1) \\
T[\![\textbf{Empty}[Q]]\!] &\in O(1)
\end{aligned}
$$

5.3 Implementationsovervejelser

Hvis vi ikke ønsker at (eller ikke kan) angive den maksimale størrelse, så kan stakke og køer implementeres ved hjælp af pointere, så samtlige operationer får udførelsestider i $O(1)$. Vi viser implementationen af en stak

```
Box S(Element)
    Type Stack = Pointer(Node)
    Type Node = Prod(first: Element, next: Stack)

    Proc Init[S: Stack]
        S := nil
    end Init

    Proc Push[S: Stack](x: Element)
        S := Stack(Node(x, S))
    end Push

    Proc Pop[S: Stack, x: Element]
        if S ≠ nil →
            x := ref(S).first
            S := ref(S).next
        fi
    end Pop

    Proc Empty[S: Stack] → (Bool)
        return S = nil
    end Empty
end S
```

5.4 En anvendelse: beregning af postfix udtryk

Regneudtryk kan skrives på andre måder end med operatoren *imellem* argumenterne. I de såkaldte *postfix* udtryk skrives begge argumenterne *før* operatoren. Dette kendes blandt andet fra HP-lommeregnere, og kaldes også for *omvendt polsk notation*. Postfix udtryk har den fordel, at man aldrig får brug for parenteser. Fx vil regneudtrykket `(2+3)*7-4` i postfix

notation skrives som `2 3 + 7 * 4 -`. Følgende program benytter en stak af heltal til at beregne værdien angivet af et postfix udtryk; vi antager for simpelheds skyld, at talkonstanter kun har et enkelt ciffer.

```
Process Postfix
    (+ @"stack.tri"

        Box R
            S(Int)
        end R

        Var T: Text
        Var Z: R'Stack

        R'Init[Z]
        read[T]
        (+ Var i, r: Int
            i:=0
            do i<| T | →
                if '0' ≤ T.(i) ≤ '9' → R'Push[Z](ci(T.(i))-ci('0'))
                & true →
                    (+ Var x, y: Int
                        R'Pop[Z, y]
                        R'Pop[Z, x]
                        if T.(i) = '*' → R'Push[Z](x*y)
                        | T.(i) = '+' → R'Push[Z](x+y)
                        | T.(i) = '-' → R'Push[Z](x-y)
                        | T.(i) = '/' → R'Push[Z](x/y)
                        fi
                    +)
                fi
                i:=i+1
            od
            R'Pop[Z, r]
            write(r)
        +)
    +)
end Postfix
```

6 Prioritetskøer

Vi er givet en elementtype E, hvis værdier har en total ordning $\sqsubseteq$. En *prioritetskø* over E er en datatype, hvis værdimængde består af endelige multimængder af værdier af type E.

En *multimængde* er ligesom en mængde, bortset fra at den kan indeholde gentagelser af elementer.

Lad P være en prioritetskøvariabel, og x af type E. Så skal prioritetskøen have følgende operationer

$$\begin{array}{rll}
\textbf{Init}[P] & \textbf{sat} & P' = \emptyset \\
\textbf{Empty}[P] & \textbf{sat} & (P' = P) \wedge (\text{Empty}' = (P = \emptyset)) \\
\textbf{Insert}[P](x) & \textbf{sat} & P' = P \cup \{x\} \\
\textbf{DeleteMin}[P, x] & \textbf{sat} & P \neq \emptyset \rightarrow \\
 & & \quad (x' \in P) \wedge (P' = P - \{x'\}) \wedge (\forall a \in P : x' \sqsubseteq a)
\end{array}$$

Vi kan tænke på dette som en "udemokratisk" kø, hvor ankomster er vilkårlige, men hvor man altid lader den "fineste" ventende forlade køen først.

Prioritetskøen omfatter flere andre datatyper. Man kan få en almindelig *stak* over E ved at knytte *tidsinformation* til elementerne og definere $e_1 \sqsubset e_2$, hvis e_1 ankom *efter* e_2. Omvendt kan man konstruere en *kø* ved på tilsvarende måde at definere $e_1 \sqsubset e_2$, hvis e_1 ankom *før* e_2.

Vi ved, at disse to specialtilfælde kan implementeres optimalt ved hjælp af lister. I det generelle tilfælde er det vanskeligere. Der er to indlysende måder at bruge en liste i den generelle situation. For det første kan man ved indsættelse blot anbringe det nye element bagest i listen og ved fjernelse lede efter det mindste element. Det fører til følgende kompleksiteter (N er igen det maksimale antal elementer).

$$\begin{array}{rcl}
T[\![\textbf{Init}[P](N)]\!] & \in & O(N) \\
T[\![\textbf{Empty}[P]]\!] & \in & O(1) \\
T[\![\textbf{Insert}[P](x)]\!] & \in & O(1) \\
T[\![\textbf{DeleteMin}[P, x]]\!] & \in & \Omega(|P|)
\end{array}$$

Omvendt kan man sørge for hele tiden at have det mindste element forrest

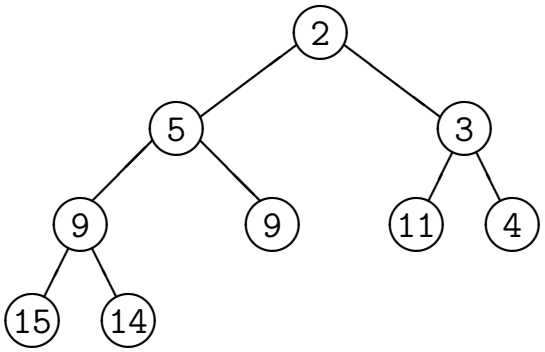
i listen. Så bliver fjernelsen billig og indsættelsen tilsvarende dyr

$$\begin{aligned} T[\![\mathbf{Init}[P](N)]\!] &\in O(N) \\ T[\![\mathbf{Empty}[P]]\!] &\in O(1) \\ T[\![\mathbf{Insert}[P](x)]\!] &\in \Omega(|P|) \\ T[\![\mathbf{DeleteMin}[P,x]]\!] &\in O(1) \end{aligned}$$

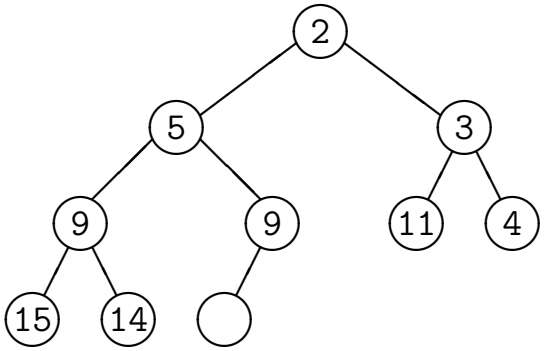
I mange situationer har man brug for en bedre balance mellem operationerne. Dette kan opnås ved hjælp af en træstruktur.

6.1 Bunker

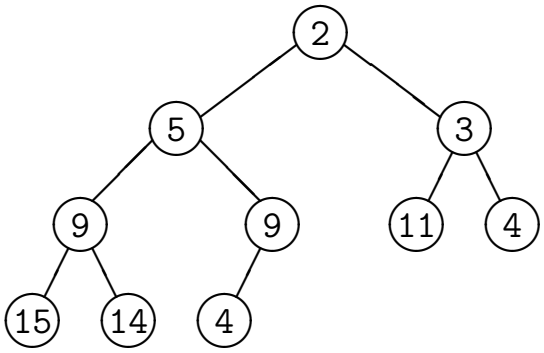
En *bunke* over E er et (balanceret) binært træ, hvis knuder indeholder værdier af type E. Endvidere gælder det for enhver knude, med indhold e_1, at hvis den har en efterfølger, med indhold e_2, så er $e_1 \sqsubseteq e_2$. Det vil sige, at elementerne på enhver vej fra roden er i ikke-aftagende orden. Det følger umiddelbart, at roden af bunken indeholder det mindste element. Det følgende er en bunke af heltal



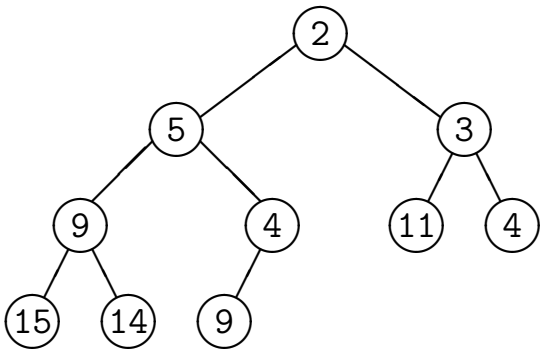
Lad os se hvordan man kan indsætte i en bunke. Fx kan vi indsætte tallet 4 i ovenstående eksempel. For det første skal træet have en ny knude. Dette kan klares ved at tilføje et blad, det første sted der "mangler" et.



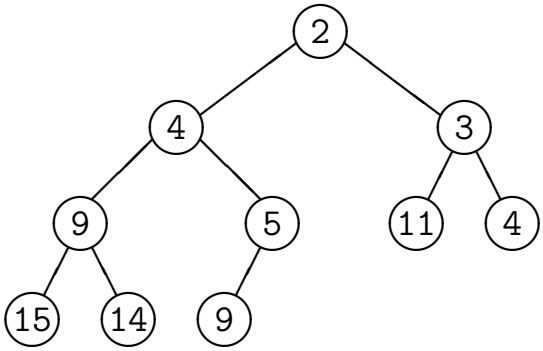
Desværre kan vi ikke blot indsætte det nye element her. Det kan jo, som i dette tilfælde, være for lille. Denne skavank er dog meget let at udbedre. Når vi har indsat det nye element, kan vi ombytte det med dets far, sålænge sønnen er mindre end faderen. På denne måde bliver det nye element "skubbet op" i bunken, indtil det falder på plads. Hvis det nye element er mindre end alle de gamle, vil det ende i roden af bunken. Indsættelsen af tallet 4 vil fortsætte i følgende skridt



Da 4 er mindre end 9 bytter vi rundt.

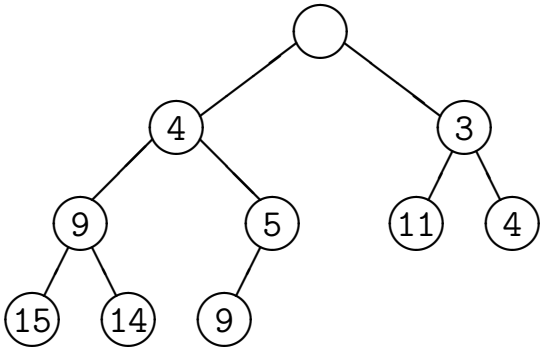


4-tallet er endnu ikke på plads, da dets far er et 5-tal.

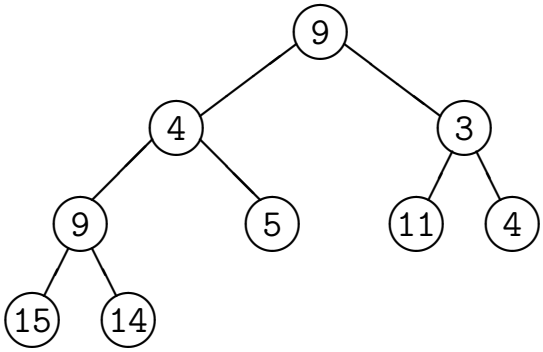


Til sidst har vi fået 4-tallet anbragt, hvor det hører hjemme. Bemærk, at vi aldrig kan komme til at skubbe et element længere end *højden* af bunken.

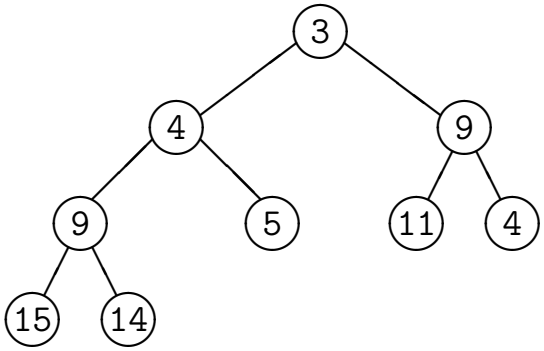
Fjernelsen af det mindste element foregår på tilsvarende facon. Som sagt, kan vi altid finde det mindste element i roden. Når vi har fjernet det, står vi desværre med et "hul", der skal fyldes ud



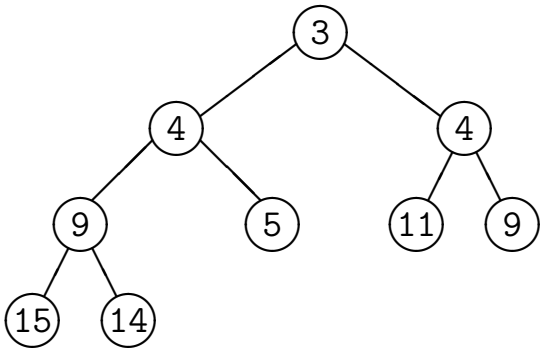
Vi fjerner bladet længst til højre og flytte dets indhold op i roden



Nu er situationen ligeså slem som før: det nye element i roden er for stort. Analogt med tidligere, kan vi dog "skubbe" 9-tallet ned i bunken, til det kommer på plads. Vi ombytter det med det *mindste* af dets sønner

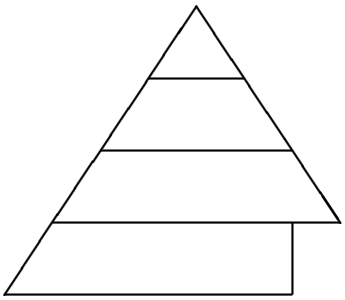


Vi kan heller ikke have et 9-tal siddende her, så vi ombytter igen med den mindste søn



Nu er situationen normaliseret. Som før kan vi aldrig skubbe elementet længere end *højden* af bunken.

Vi kan observere, at vores anstrengelser fører til, at bunken altid er et *perfekt balanceret* binært træ. Det vil sige, at træet altid er helt fyldt ud, undtagen måske i det nederste lag, der til gengæld er udfyldt fra venstre. Træet har altså følgende udseende



Hvis et perfekt balanceret træ har n knuder, så er dets højde $O(\log n)$. Antag nemlig, at træet har højde h; så må det have mindst

$$2^0 + 2^1 + \cdots + 2^{h-1} + 1 = 2^h$$

knuder. Så hvis træet har n knuder, så er $2^h \leq n$, det vil sige $h \leq \log n$. Kompleksiteterne bliver således i dette tilfælde

$$\begin{aligned} T[\![\mathbf{Init}[P](N)]\!] &\in O(N) \\ T[\![\mathbf{Empty}[P]]\!] &\in O(1) \\ T[\![\mathbf{Insert}[P](x)]\!] &\in O(\log |P|) \\ T[\![\mathbf{DeleteMin}[P, x]]\!] &\in O(\log |P|) \end{aligned}$$

Da bunken er perfekt balanceret, kan man implementere den på en særligt snedig måde. Hvis

$$e_0, e_1, e_2, \ldots, e_{|P|-1}$$

er en *lag-for-lag* udskrift af en bunke, gælder det, at e_0 er roden, og hvis e_i er en vilkårlig knude, så er dens to sønner e_{2i+1} og e_{2i+2}. Vi kan derfor repræsentere bunken i en liste i den ovennævnte lag-for-lag rækkefølge. Den sidste version af vores eksempelbunke ovenfor ville blive repræsenteret som

$$(3, 4, 4, 9, 5, 11, 9, 15, 14)$$

Denne repræsentation er mere kompakt end en rekursiv træstruktur. Desuden kan det sidste element findes i konstant tid, hvilket ikke er tilfældet for træer.

En box med en prioritetskø af heltal ser med dette valg af datastruktur ud som følger

```
Type Element = Int

Box Pri
    Type Elist = List(Element)
    Type Queue = Prod(size: Int, data: Elist)

    Proc Init[Q: Queue](N: Int)
        Q := Queue(0, Elist(?–Element | N))
    end Init

    Proc Empty[Q: Queue] → (Bool)
        return Q.size = 0
    end Empty

    Proc Insert[Q: Queue](x: Element)
        (+ Proc Up[Q: Queue](n: Int)
              (+ Var father: Int
                 father := (n-1)/2
                 if (n > 0) ∧ (Q.data.(father) > Q.data.(n)) →
                       Q.data.(father) :=: Q.data.(n)
                       Up[Q](father)
                 fi
              +)
           end Up

           if Q.size < | Q.data | →
                 Q.data.(Q.size) := x
                 Q.size := Q.size+1
                 Up[Q](Q.size-1)
           fi
        +)
    end Insert
```

```
    Proc DeleteMin[Q: Queue, x: Element]
        (+ Proc Down[Q: Queue](n: Int)
            (+ Var son: Int
               son := 2*n+1
               if son < Q.size →
                    if (son < Q.size-1) ∧ (Q.data.(son+1) < Q.data.(son)) →
                         son := son+1
                    fi
                    if Q.data.(n) > Q.data.(son) →
                         Q.data.(n) :=: Q.data.(son)
                         Down[Q](son)
                    fi
               fi
            +)
        end Down

        if Q.size > 0 →
             x := Q.data.(0)
             Q.size := Q.size-1
             if Q.size > 0 →
                  Q.data.(0) := Q.data.(Q.size)
                  Down[Q](0)
             fi
        fi
    +)
    end DeleteMin
end Pri
```

6.2 En polymorf prioritetskø

I dette afsnit beskrives en mere generelt anvendelig prioritetskø. Den implementeres som en *polybox*, således at elementerne kan have vilkårlige typer. Da man skal have en ordning mellem elementer, så kræver vi, at man sammen med et element angiver en *nøgle*, som er et heltal der angiver elementets prioritet. Til gengæld kan man ved initialisering, foruden køens maksimale størrelse, også angive hvilken vej ordningen på nøgler skal vende; DeleteMin er defor omdøbt til DeleteBest.

```
Box PolyPri(E)
    Type Element = Prod(val: E, key: Int)
    Type Elist = List(Element)
    Type Queue = Prod(size: Int, data: Elist, less: Bool)

    Proc Init[Q: Queue](N: Int, less: Bool)
        Q := Queue(0, Elist(?–Element | N), less)
    end Init

    Proc Order[Q: Queue](i, j: Int) → (Bool)
        if ¬ Q.less → i :=: j fi
        return Q.data.(i).key < Q.data.(j).key
    end Order

    Proc Empty[Q: Queue] → (Bool)
        return Q.size = 0
    end Empty

    Proc Insert[Q: Queue](val: E, key: Int)
        (+ Proc Up[Q: Queue](n: Int)
            (+ Var father: Int
               father := (n-1)/2
               if (n > 0) ∧ Order[Q](n, father) →
                     Q.data.(father) :=: Q.data.(n)
                     Up[Q](father)
               fi
            +)
          end Up

          if Q.size < | Q.data | →
               Q.data.(Q.size) := Element(val, key)
               Q.size := Q.size+1
               Up[Q](Q.size-1)
          fi
        +)
    end Insert
```

```
    Proc DeleteBest[Q: Queue, val: E, key: Int]
        (+ Proc Down[Q: Queue](n: Int)
            (+ Var son: Int
               son := 2*n+1
               if son < Q.size →
                    if (son < Q.size-1) ∧ Order[Q](son+1, son) →
                         son := son+1
                    fi
                    if Order[Q](son, n) →
                         Q.data.(n) :=: Q.data.(son)
                         Down[Q](son)
                    fi
               fi
            +)
        end Down

        if Q.size > 0 →
             val := Q.data.(0).val
             key := Q.data.(0).key
             Q.size := Q.size-1
             if Q.size > 0 →
                  Q.data.(0) := Q.data.(Q.size)
                  Down[Q](0)
             fi
        fi
      +)
    end DeleteBest
end PolyPri
```

Boxen Pri kan nu fås som

```
Box Pri
    PolyPri(Int)
end Pri
```

hvor man skal initialisere med

```
Pri'Init[P](N, true)
```

En prioritetskø uden maksimal størrelse kan som i de andre tilfælde implementeres ved hjælp af polyboxen Sequence.

6.3 En anvendelse: træsortering

En indlysende anvendelse af en sådan prioritetskø er til *sortering*. Man indsætter først alle tallene og fjerner dem derefter et ad gangen i størrelsesorden. Følgende program indlæser en vektor og udskriver den igen i sorteret orden.

```
Process Treesort
    (+ @"pri.tri"

        Var S: Vector
        Var H: Pri'Queue
        write("Indlæs vektor: ")
        read[S]
        Pri'Init[H](| S |)
        (+ Var i: Int
           i := 0
           do i<| S | →
                Pri'Insert[H](S.(i))
                i := i+1
           od
        +)
        (+ Var i: Int
           i := 0
           do i<| S | →
                Pri'DeleteMin[H,S.(i)]
                i := i+1
           od
        +)
        write(S)
    +)
end Treesort
```

Hvis vi indlæser n tal, så får vi

$$T[\![\text{Treesort}]\!](n) \approx \sum_{i=0}^{n-1} T[\![\textbf{Proc}\ \text{Insert}]\!](i) + \sum_{i=1}^{n} T[\![\textbf{Proc}\ \text{DeleteMin}]\!](i)$$

hvor $T[\![\textbf{Proc}\ \text{Insert}]\!](i)$ ($T[\![\textbf{Proc}\ \text{DeleteMin}]\!](i)$) betegner udførelsestiden for indsættelse (fjernelse) i (fra) en kø med i elementer. Heraf fås

$$T[\![\text{Treesort}]\!](n) \approx \sum_{i=1}^{n} O(\log i) = O(\sum_{i=1}^{n} \log i) \subseteq O(n \log n)$$

hvilket kan vises at være *optimalt* blandt sorteringsalgoritmer, der kun må sammenligne og ombytte elementerne.

Den første del af denne algoritme, indsættelsen af de n tal i bunken, kan gøres mere effektivt. Man kan starte med at bygge et perfekt balanceret træ med tallene i tilfældig orden. De mindste undertræer, bladene, er jo allerede små bunker i sig selv. Hvergang man har gjort to undertræer til bunker, kan man bygge dem sammen til en større bunke, ved at skubbe deres fælles rod på plads i bunken. Dette kan gøres med følgende forbløffende simple procedure, som vi kan tilføje til boxen Pri:

```
Proc Build[Q: Queue](v: Vector)
    Q := Queue(| v |, v)
    (+ Var i: Int
       i := Q.size
       do i>0 →
            i := i-1
            Down[Q](i)
       od
    +)
end Build
```

Det skulle være klart, at udførelsestiden for Build er givet ved

$$T[\![\textbf{Proc}\ \text{Build}]\!](n) \approx \sum_{i=0}^{n-1} T[\![\textbf{Proc}\ \text{Down}]\!](n, i)$$

hvor n er antallet af elementer i Q og i er værdien af parameteren i kaldet

Down[Q](i). For selve proceduren Down gælder der

$$T[\![\textbf{Proc}\ \text{Down}]\!](n,i) \approx \begin{cases} 1 & \text{hvis } i \geq \frac{n}{2} \\ 1 + T[\![\textbf{Proc}\ \text{Down}]\!](n, 2i+1) & \text{ellers} \end{cases}$$

Løsningen til denne ligning er

$$T[\![\textbf{Proc}\ \text{Down}]\!](n,i) \approx \log(\frac{n}{i+1})$$

hvorfor vi får

$$T[\![\textbf{Proc}\ \text{Build}]\!](n) \approx \sum_{i=1}^{n} \log(\frac{n}{i})$$

Denne sum kan "deles op i bidder" på følgende måde

$$\begin{aligned} & \sum_{i=\frac{n}{2}}^{n} \log(\frac{n}{i}) + \sum_{i=\frac{n}{4}}^{\frac{n}{2}-1} \log(\frac{n}{i}) + \cdots + \sum_{i=2}^{3} \log(\frac{n}{i}) + \log(n) \\ < \ & \frac{n}{2} \cdot 1 + \frac{n}{4} \cdot 2 + \frac{n}{8} \cdot 3 + \cdots + 2 \cdot (\log(n) - 1) + 1 \cdot \log(n) \\ = \ & n(\frac{1}{2} + \frac{2}{4} + \frac{3}{8} + \cdots + \frac{\log(n)}{n}) \\ < \ & n \sum_{j=1}^{\infty} \frac{j}{2^j} = 2n \end{aligned}$$

Vi kan således opbygge en bunke med n elementer i tid $O(n)$. Sorteringen er dog selvfølgelig stadig i $O(n \log(n))$, fordi elementerne skal tages ud af bunken igen.

Et sorteringsprogram, der benytter Build, ser ud som følger

```
Process BTreesort
    (+ @"pri.tri"

        Var S: Vector
        Var H: Pri'Queue
        write("Indlæs vektor: ")
        read[S]
        Pri'Build[H](S)
        (+ Var i: Int
```

```
            i:=0
            do i<| S | →
                  Pri'DeleteMin[H,S.(i)]
                  i:=i+1
            od
         +)
         write(S)
      +)
end BTreesort
```

7 Ordbøger

Vi er givet en elementtype E. En *ordbog* over E er en datatype, hvis værdimængde består af endelige mængder af værdier af type E. Lad D være en ordbogsvariabel og x en værdi af type E. Så skal ordbogen have følgende operationer

$$\begin{array}{rll} \mathbf{Init}[D] & \mathbf{sat} & D' = \emptyset \\ \mathbf{Insert}[D](x) & \mathbf{sat} & D' = D \cup \{x\} \\ \mathbf{Member}[D](x) & \mathbf{sat} & (D' = D) \wedge (\mathrm{Member}' = (x \in D)) \\ \mathbf{Delete}[D](x) & \mathbf{sat} & D' = D - \{x\} \end{array}$$

Den simpleste datastruktur for en ordbog ville igen være en liste, der indeholdt dens elementer. Det vil dog give følgende kompleksiteter

$$\begin{array}{rcl} T[\![\mathbf{Init}[D]]\!] & \in & O(N) \\ T[\![\mathbf{Insert}[D](x)]\!] & \in & O(1) \\ T[\![\mathbf{Member}[D](x)]\!] & \in & \Omega(|D|) \\ T[\![\mathbf{Delete}[D](x)]\!] & \in & \Omega(|D|) \end{array}$$

Man kan under specielle omstændigheder klare sig langt bedre.

7.1 Bitvektorer

Hvis E er intervallet $0..N$ hvor N er et passende lille heltal, så kan man repræsentere en ordbog over E som en *bitvektor*, der er en tabel af sandhedsværdier. Man kunne bruge typen

$$\mathbf{Type}\ \mathrm{Dictionary} = \mathbf{List}\,(\mathrm{Bool})$$

Hvis D er en variabel af denne type, er mængden den repræsenterer

$$\{i \in 0..N \mid D.(i) = \mathrm{true}\}$$

Med denne datastruktur opnår vi naturligvis optimale kompleksiteter

$$\begin{aligned} T[\![\mathbf{Init}[D]]\!] &\in O(N) \\ T[\![\mathbf{Insert}[D](x)]\!] &\in O(1) \\ T[\![\mathbf{Member}[D](x)]\!] &\in O(1) \\ T[\![\mathbf{Delete}[D](x)]\!] &\in O(1) \end{aligned}$$

Vi skal jo blot inspicere og opdatere en enkelt indgang i bitvektoren i hvert af disse tilfælde.

En box med en ordbog, implementeret som en bitvektor, ser i dette tilfælde således ud

```
Type Element = Int

Box B
    Type Dictionary = List(Bool)

    Proc Init[D: Dictionary](N: Int)
        D := Dictionary(false | N)
    end Init

    Proc Insert[D: Dictionary](x: Element)
        if 0≤x<|D| → D.(x) := true fi
    end Insert

    Proc Member[D: Dictionary](x: Element) → (Bool)
        if 0≤x<|D| → return D.(x) fi
    end Member

    Proc Delete[D: Dictionary](x: Element)
        if 0≤x<|D| → D.(x) := false fi
    end Delete
end B
```

Desværre vil grundmængden E i realistiske sammenhænge sjældent være tilstrækkeligt lille til, at man kan bruge bitvektorer.

7.2 Hash-tabeller

Selv om E er for stor til bitvektorer, kan man opnå en lignende effekt, hvis selve de mængder, man ønsker at behandle, ikke er for store.

En *hash-funktion* på E er en afbildning

$$h : E \to 0..M$$

for et passende M, der angiver størrelsordenen af kardinaliteten af de mængder, vi ønsker at behandle. Hash-funktionen har til opgave at "sprede" E-værdierne over hele tabellen. Da $|E| > M$ kan flere elementer blive sendt til samme indgang i tabellen. Vi må samle disse op i en liste, så vi kan definere

Type Elist = **List**(Int)
Type Dictionary = **List**(Elist)

Hvis D er en variabel af denne type, så er mængden den repræsenterer

$$\{e \in E \mid \exists i, j : D.(i, j) = e\}$$

Hvis elementerne er heltal, så er en typisk hash-funktion

$$h(e) = e \textbf{ mod } M$$

Bemærk, at hvis $0 < p \leq M$ er divisor i M *og* divisor i alle $e \in E$, så vil vi kun kunne bruge M/p indgange i tabellen. Ved at vælge M som et *primtal* minimaliserer vi risikoen for dette. Det præcise valg af hash-funktionen afhænger helt af den aktuelle anvendelse.

Hvis $M = 17$, så vil en hash-tabel med $h(e) = e$ **mod** 17, der indeholder elementerne

$$\{0, 3, 9, 11, 12, 17, 24, 28, 31, 43, 55, 68, 71, 88, 101\}$$

have følgende udseende

0	→	(0,17,68)
1	→	()
2	→	()
3	→	(3,71,88)
4	→	(55)
5	→	()
6	→	()
7	→	(24)
8	→	()
9	→	(9,43)
10	→	()
11	→	(11,28)
12	→	(12)
13	→	()
14	→	(31)
15	→	()
16	→	(101)

En box, der implementerer denne hashtabel, ser ud som følger

```
Box H
    Type Dictionary = List(Elist)
    Type Elist = List(Int)

    Proc Hash(e: Int) → (Int)
        return e mod 17
    end Hash

    Proc Init[D: Dictionary]
        D := Dictionary(Elist() | 17)
    end Init
```

```
    Proc Insert[D: Dictionary](x: Int)
        if ¬ Member[D](x) →
            (+ Var h: Int
               h := Hash(x)
               D.(h) := D.(h)++Elist(x)
            +)
        fi
    end Insert

    Proc Member[D: Dictionary](x: Int) → (Bool)
        (+ Var h, i: Int
           h, i := Hash(x), 0
           do i < | D.(h) | →
                if x = D.(h, i) → return true fi
                i := i+1
           od
           return false
        +)
    end Member

    Proc Delete[D: Dictionary](x: Int)
        (+ Var h, i: Int
           h, i := Hash(x), 0
           do (i < | D.(h) |) ∧ (x ≠ D.(h, i)) → i := i+1 od
           if i < | D.(h) | →
                D.(h) := D.(h)(0 .. i)++D.(h)(i+1 .. | D.(h) |)
           fi
        +)
    end Delete
end H
```

I værste fald, hvor hash-funktionen sender alle elementer til samme værdi, vil dette degenerere til kompleksiteterne

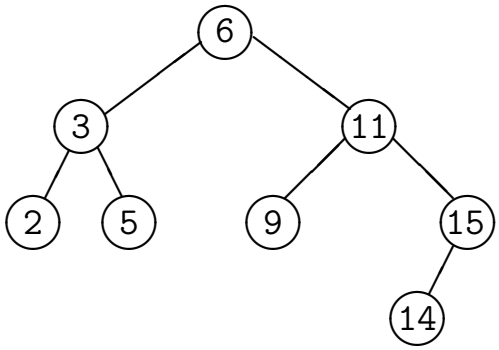
$$\begin{aligned} T[\![\textbf{Init}[D]]\!] &\in O(M) \\ T[\![\textbf{Insert}[D](x)]\!] &\in \Omega(|D|) \\ T[\![\textbf{Member}[D](x)]\!] &\in \Omega(|D|) \\ T[\![\textbf{Delete}[D](x)]\!] &\in \Omega(|D|) \end{aligned}$$

men med en rimelig fordeling af elementerne i tabellen, så vil alle de små lister kun have få elementer, og kompleksiteterne bliver

$$\begin{aligned} T[\![\mathbf{Init}[D]]\!] &\in O(M) \\ T[\![\mathbf{Insert}[D](x)]\!] &\in O(1) \\ T[\![\mathbf{Member}[D](x)]\!] &\in O(1) \\ T[\![\mathbf{Delete}[D](x)]\!] &\in O(1) \end{aligned}$$

7.3 Søgetræer

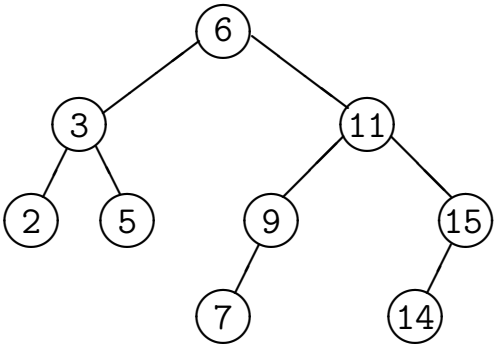
Hvis E har en total ordning $\sqsubseteq$, så kan vi igen få fordel af en træstruktur. Et *søgetræ* over E er et binært træ med E-værdier i knuderne. Hvis en knude, med indhold e, har en efterfølger i dens venstre undertræ, med indhold a, så skal det gælde, at $a \sqsubset e$; hvis knuden har en efterfølger i dens højre undertræ, med indhold b, så skal det gælde, at $e \sqsubset b$. Et søgetræ over heltallene kan se ud som følger



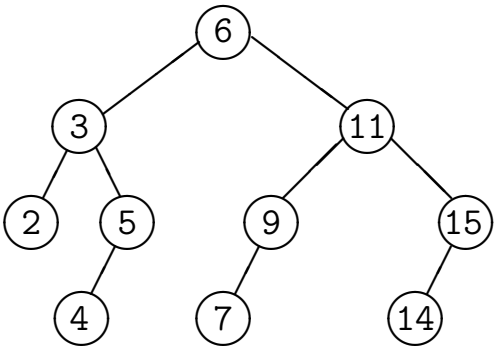
Det er meget let at finde et element i et søgetræ. Man sammenligner med værdien i den aktuelle knude. Hvis den er lig med den søgte værdi er man færdig. Hvis den er større end den søgte værdi skal man lede videre i venstre undertræ, og hvis den er mindre fortsættes søgningen i højre undertræ.

Indsættelse er ligeså simpel. Først leder man efter det aktuelle element, da det jo ikke skal indsættes to gange. Hvis man ikke finder det, er man til sidst i et blad, hvor det aktuelle element kan indsættes enten til højre eller til venstre.

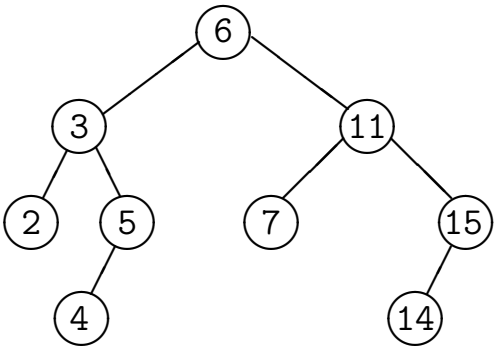
Hvis vi indsætter tallet 7 i ovenstående søgetræ får det udseendet



Indsættelse af et 4-tal resulterer i dette træ

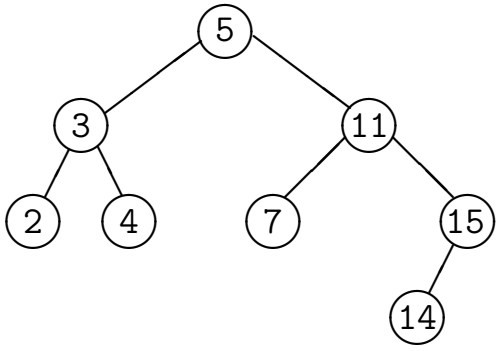


Man kan også fjerne et element e fra søgetræet. Hvis den uønskede knude højst har et enkelt undertræ er det ligetil. Så kan man blot erstatte knuden med roden i dens undertræ. I ovenstående eksempel kan vi fjerne tallet 9 og opnå træet



Hvis knuden har to undertræer er det knap så simpelt. Hvis man blot udtager knuden med elementet, så efterlader man et uklædeligt hul i træet. Ligesom ved bunken skal vi finde en passende erstatning. Det er heldigvis let at gøre. Søgetræets invariant bliver nemlig overholdt, hvis man erstatter elementet e med dets *umiddelbare forgænger*. Dette er det største element,

der er mindre end e, og det findes i den højreste knude i e's venstre undertræ. Denne knude kan ikke have noget højre undertræ og er således let at fjerne. Hvis vi fjerner tallet 6 fra eksempeltræet, observerer vi først, at dets umiddelbare forgænger er 5. Derfor bliver det nye træ som følger



Man kunne naturligvis også have valgt den *umiddelbare efterfølger*, hvilket er det mindste element, der er større.

Alle disse operationer tager tid proportionalt med træets højde. For et træ D med højde h fås derfor

$$\begin{aligned} T[\![\mathbf{Init}[D]]\!] &\in O(1) \\ T[\![\mathbf{Insert}[D](x)]\!](h) &\in O(h) \\ T[\![\mathbf{Member}[D](x)]\!](h) &\in O(h) \\ T[\![\mathbf{Delete}[D](x)]\!](h) &\in O(h) \end{aligned}$$

Man kan bevise, at hvis elementerne bliver indsat og fjernet i helt tilfældig orden, så er det rimeligt at antage, at h er proportional med $\log |D|$. Men hvis elementerne fx bliver indsat i *sorteret* orden, vil man få den værst tænkelige situation, hvor træet bliver *lineært*.

I dette triste tilfælde får vi disse deprimerende kompleksiteter

$$\begin{aligned} T[\![\mathbf{Init}[D]]\!] &\in O(1) \\ T[\![\mathbf{Insert}[D](x)]\!] &\in \Omega(|D|) \\ T[\![\mathbf{Member}[D](x)]\!] &\in \Omega(|D|) \\ T[\![\mathbf{Delete}[D](x)]\!] &\in \Omega(|D|) \end{aligned}$$

I næste afsnit skal vi betragte en variant af søgetræer, der *med garanti* forbliver pæne og balancerede.

En ordbog af heltal, implementeret som søgetræer, ser ud som følger

```
Type Element = Int

Box Search
    Type Tree = Prod(val: Element, left, right: Tree)

    Proc Init[T: Tree]
        T := ?-Tree
    end Init

    Proc Member[T: Tree](x: Element) → (Bool)
        if ¬is(T) → return false
        & true →
            if T.val = x → return true
            | T.val>x → return Member[T.left](x)
            | T.val<x → return Member[T.right](x)
            fi
        fi
    end Member

    Proc Insert[T: Tree](x: Element)
        if ¬is(T) → T := Tree(x, ?-Tree, ?-Tree)
        & true →
            if T.val>x → Insert[T.left](x)
            | T.val<x → Insert[T.right](x)
            fi
        fi
    end Insert

    Proc Delete[T: Tree](x: Element)
        (+ Proc DeleteMax[T: Tree, m: Element]
            if ¬is(T.right) →
                m := T.val
                T :=: T.left
            & true → DeleteMax[T.right, m]
            fi
        end DeleteMax
```

```
            if is(T) →
                 if T.val = x →
                      if ¬ is(T.left) → T:=:T.right
                      | ¬ is(T.right) → T:=:T.left
                      & true → DeleteMax[T.left, T.val]
                      fi
                 | T.val>x → Delete[T.left](x)
                 | T.val<x → Delete[T.right](x)
                 fi
            fi
        +)
    end Delete
end Search
```

Denne datatype kan naturligvis gøres polymorf i lighed med prioritetskøen.

7.4 Rød-sorte træer

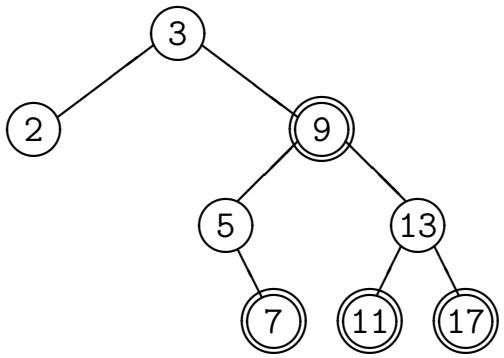
Problemet med ubalancerede søgetræer kan afhjælpes på mange måder. I alle tilfælde styrker man den invariant, som træerne skal opfylde. Udfra denne ekstra information skal man så kunne slutte, at træerne har logaritmisk højde.

Et *rødt-sort træ* er et søgetræ, hvori alle knuderne er farvet enten røde eller sorte. Roden kan kun være sort, og en rød knude kan aldrig have en rød søn. Herudover kræves det, at der er samme antal sorte knuder på alle veje fra roden til en knude med 0 efterfølgere (blade) eller 1 efterfølger.

Af typografiske årsager vil vi angive en rød knude som

og en sort knude som

Det følgende er et eksempel på et rødt-sort træ



Da mindst hveranden knude på en vej skal være sort er den længste vej til en knude med 0 eller 1 efterfølger højst dobbelt så lang som den korteste. Det følger deraf, at alle veje i træet har logaritmiske længder: Lad træet have n knuder, og lad antallet af knuder i den korteste og længste vej være henholdsvis k_{min} og k_{max}. Det følger, at

$$2^{k_{min}} \leq n + 1 \leq 2^{k_{max}}$$

og dermed

$$k_{min} \leq \log(n + 1) \leq k_{max}$$

Da $k_{max} \leq 2k_{min}$ får vi, at

$$k_{min} \leq \log(n + 1) \leq 2k_{min}$$

og til sidst

$$\frac{1}{2}\log(n + 1) \leq k_{min} \leq \log(n + 1)$$

så den korteste vej er logaritmisk. Da den længste vej højst er dobbelt så lang, er den (og dermed enhver anden vej) også logaritmisk.

Vores problem er nu at skrive operationerne, så de overholder den rød-sorte invariant. **Init** og **Member** kræver ingen ændringer; et rødt-sort træ har jo samme struktur som et almindeligt søgetræ. Problemerne ligger ved **Insert** og **Delete**; her kan de hidtidige implementationer føre til overtrædelser af den rød-sorte invariant.

7.5 Indsættelse i rød-sorte træer

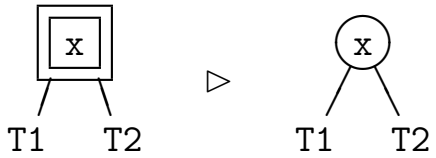
Når vi har indsat en ny knude (som et nyt blad), har vi samtidigt forøget en vej i træet med en ekstra knude. Vi kan ikke umiddelbart farve denne knude. Hvis den bliver rød kunne det være, at den havde en rød far. Hvis den bliver sort risikerer vi, at få en sort knude for meget på en vej.

Vi kan imidlertid altid slippe afsted med enten at farve knuden eller at sende problemet højere op i træet. Det følgende er et transitionssystem, der for forskellige situationer viser, hvordan vi skal bære os ad. På hver side af transitionsrelationen $\triangleright$ er der et rødt-sort træ. Ideen er, at hvis vi kan genkende en venstreside som et undertræ, så kan vi erstatte det med højresiden. Den "besværlige" knude, der skal farves, betegnes som

Algoritmen for indsættelse er nu, at vi først indsætter elementet som i et normalt søgetræ. Derefter mærker vi den nye knude som besværlig og anvender reglerne i følgende transitionssystem på træet. Hvis vi betegner den besværlige knude som en rød *uægte* knude (de øvrige knuder er *ægte*) vil følgende være en invariant for det "forfalskede" rød-sorte træ under udførelsen af transitionssystemet: (en ægte rod er sort) og (en ægte rød knude har en sort far) og (alle veje far roden til en knude med 0 eller 1 efterfølgere indeholder lige mange sorte knuder). Da vi altid gør fremskridt i transitionssystemet (fordi vi enten eliminerer den besværlige knude eller skubber den højere op i træet) og da mindst en af reglerne (eller de symmetriske tilfælde) altid kan anvendes, følger det at transitionssystemet er korrekt.

Det letteste tilfælde er hvis den besværlige knude er roden i træet.

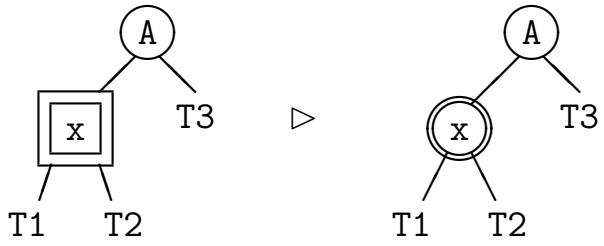
1)



I så fald står det os frit at farve den sort og dermed forøge antallet af sorte knuder i *samtlige* veje.

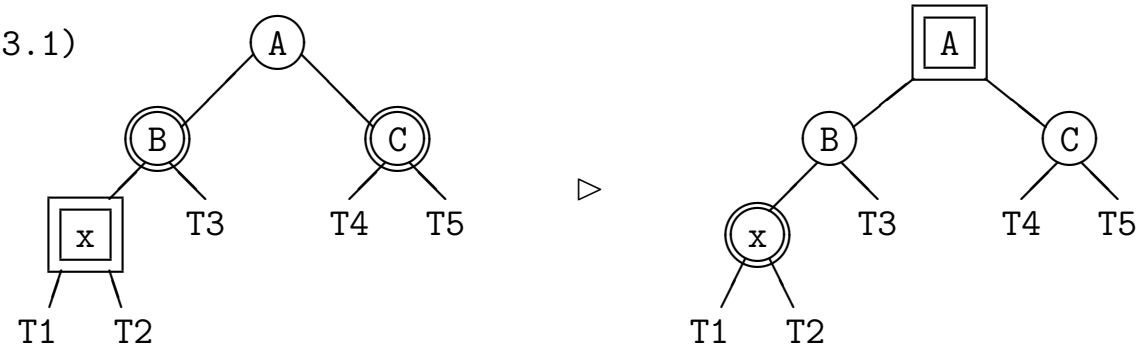
Hvis faderen er sort, kan vi uden videre farve knuden rød. Bemærk, at hvis `T1` og `T2` er tomme, svarer dette til situationen, hvor vi har indsat den nye knude som søn af en sort far.

2)



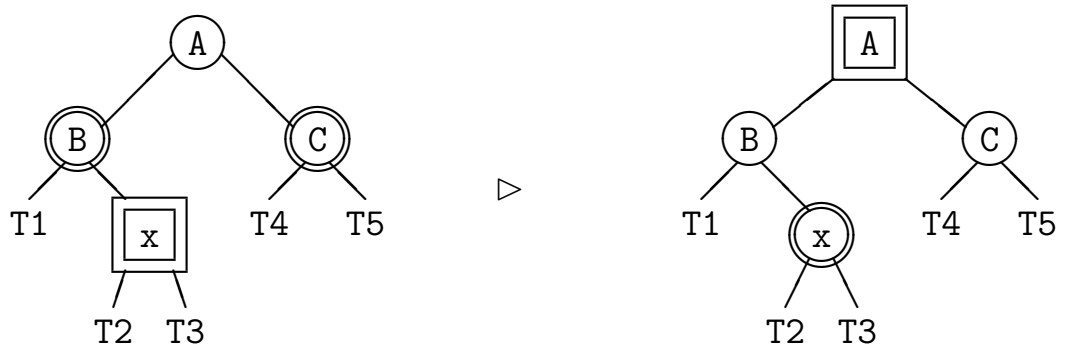
Hvis faderen er rød, er der fire tilfælde. Hvis farbroderen eksisterer og er rød har vi situationerne 3.1 og 3.2.

3.1)



Da `B` og `C` skifter farve fra rød til sort, har vejene gennem dem en overskydende sort knude. Den kan vi slippe af med, mod til gengæld at gøre `A` til en "besværlig" knude.

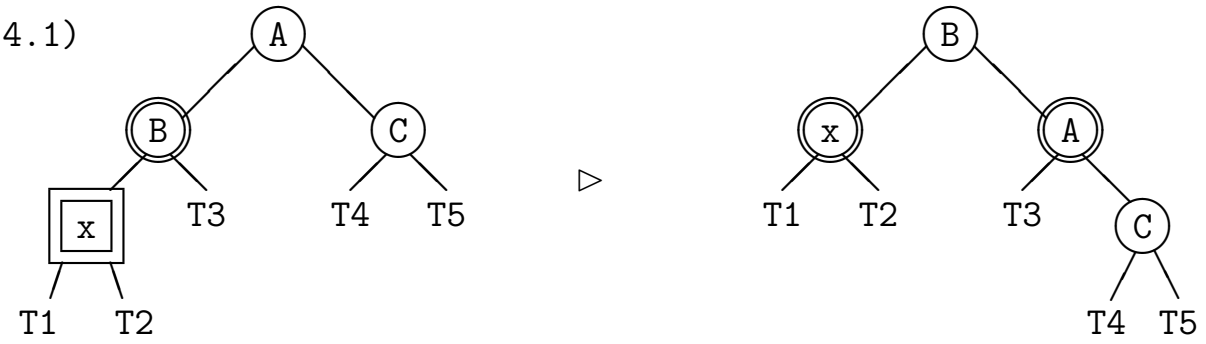
3.2)



Her kan vi igen nøjes med at sende Rød-Sorteper videre.

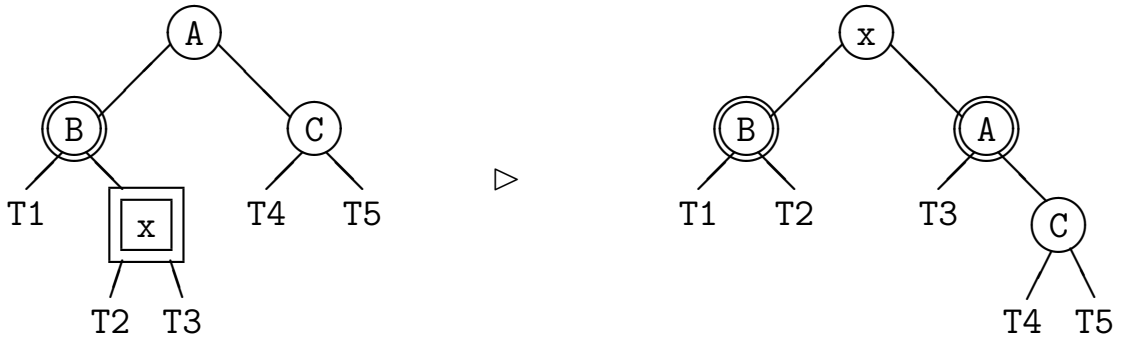
Hvis farbroderen er sort eller ikke-eksisterende har vi tilfældende 4.1 og 4.2.

4.1)



Bemærk, at træet bliver "roteret" omkring linjen `x-B-A`. En sådan rotation vil altid bevare den basale ordnings-invariant. Det er let at se, at også den rød-sorte invariant er bevaret.

4.2)



Bemærk, at denne omfattende transformation lader ordnings-invarianten intakt. Her kan vi tillade os at farve `x` sort uden at spolere den rød-sorte invariant.

7.6 Fjernelse fra rød-sorte træer

Når vi fjerner en knude fra et søgetræ, så har den altid højst ét undertræ. Hvis den havde to undertræer, så ombytter vi den jo først med dens umiddelbare efterfølger (*eller* dens umiddelbare forgænger), der jo ikke kan have noget venstre (højre) undertræ. Når vi fjerner elementet i knuden, så står vi med et "hul", der skal fjernes. Hvis hullet er rødt, så kan vi umiddelbart fjerne det. Hvis hullet er sort, men dets eneste søn er rød, så kan vi retablere invarianten ved at fjerne hullet og farve sønnen sort. Hvis sønnen også er sort, så må vi benytte os af et transitionssystem, ligesom vi gjorde ved indsættelsen. Vi angiver det som

og betegner det i analogi med ovenfor som en *uægte* sort knude (alle øvrige knuder er *ægte*). Vi eliminerer nu knuden ved hjælp af nedenstående transitionssystem. Invarianten er denne gang: (roden er sort) og (en rød knude har en sort far) og (alle veje fra roden til en ægte knude med 0 eller 1 efterfølgere indeholder lige mange sorte knuder). Da vi igen kan vise, at der altid gøres fremskridt og da der altid findes en regel der kan anvendes sålænge træet indeholder uægte knuder, er transitionssystemet igen korrekt.

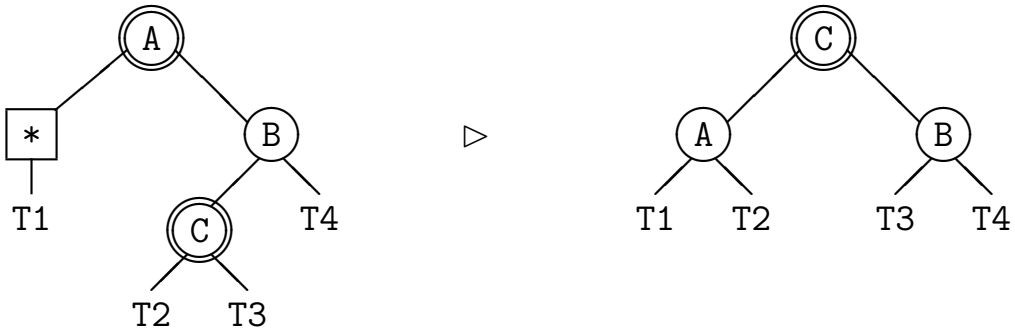
Transitionssystemet ser ud som følger; igen gælder det, at hullets søn er enten sort eller ikke-eksisterende.

Hvis hullet er ved roden, er sagen ganske enkel.

1)

Hvis faderen er rød, er der to tilfælde. Nevøen eksisterer og er rød.

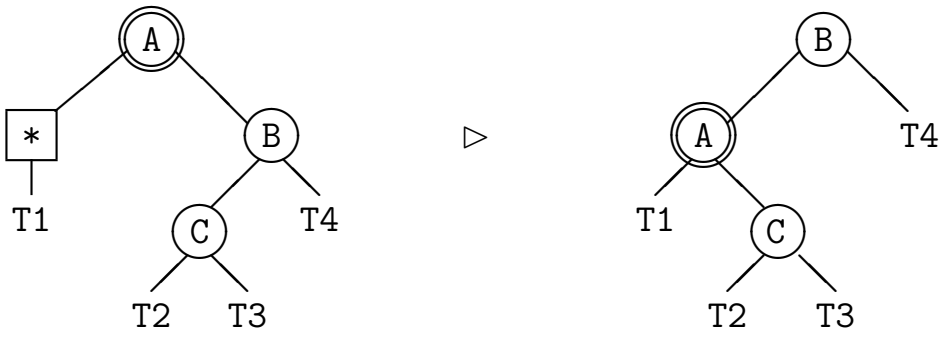
2.1)



Her kan vi slippe af med hullet, ved at omarrangere træet. Det ses let, at invarianten respekteres.

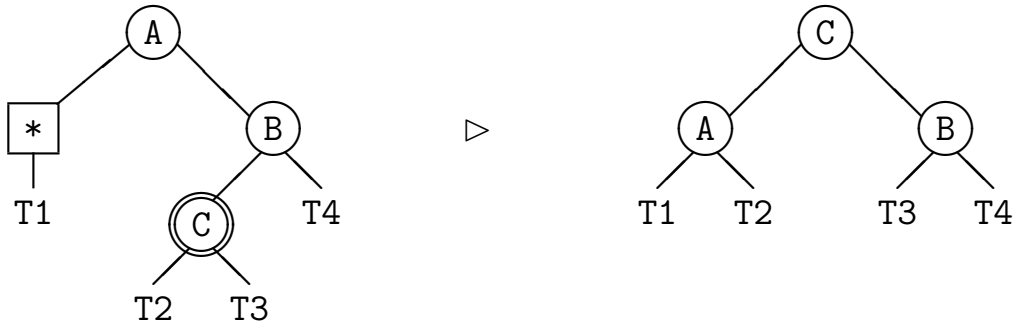
Nevøen er sort eller ikke-eksisterende.

2.2)



Hvis faderen er sort, er der tre tilfælde. Broderen er sort og nevøen eksisterer og er rød.

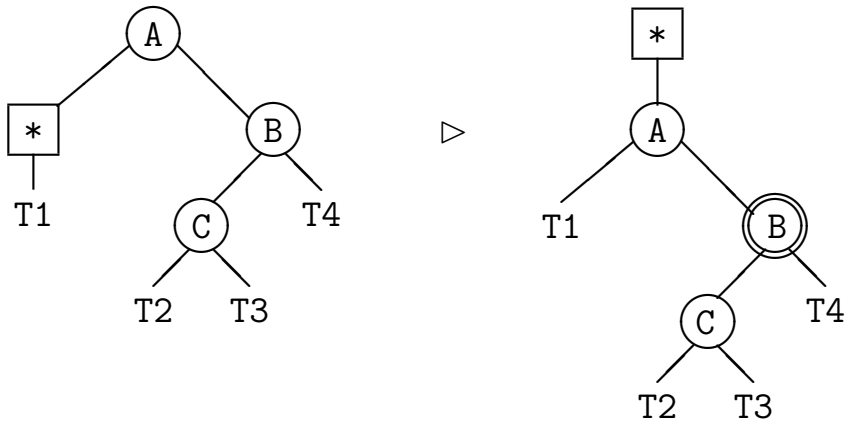
3.1)



Her forsvinder hullet ved omarrangering.

Broderen er sort og nevøen er sort eller ikke-eksisterende.

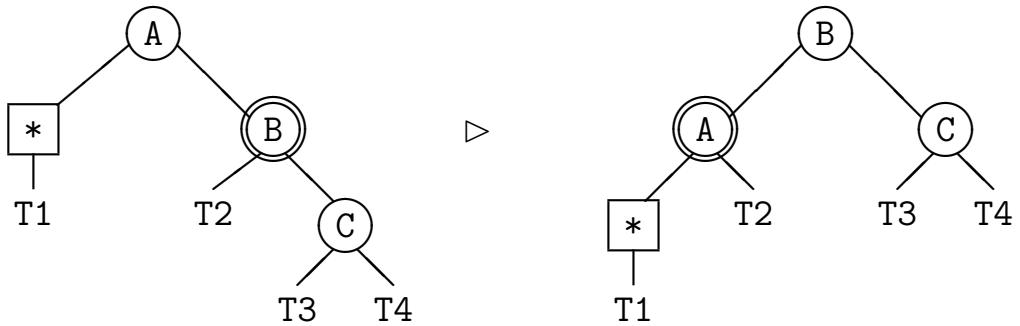
3.2)



Bemærk, at vi roligt kan farve `B` rød, da roden af `T4` må være sort. Ellers ville vi jo have det symmetriske tilfælde af `3.1)`.

Broderen er rød.

3.3)

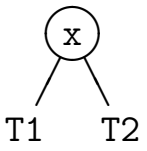


Denne transition ligner ikke nødvendigvis et fremskridt, men da roden af `T2` må være sort, kan vi gøre os færdige med enten regel `2.1` eller `2.2`.

Det overlades til den interesserede læser at realisere disse manipulationer i et TRINE-program.

7.7 Andre højdebalancerede træer

Rød-sorte træer er kun ét eksempel på højdebalancering. Der findes utallige andre. Et andet eksempel er følgende, hvor man kræver, at for ethvert undetræ af formen



skal det gælde, at

$$|\text{højde}(\texttt{T1}) - \text{højde}(\texttt{T2})| \leq k$$

for en konstant k. Hvis $k = 1$ benytter man også navnet *AVL-træer*. Ligesom med de rød-sorte træer kan man "rydde op" efter indsættelser og fjernelser og bevare invarianten.

Lad os her blot se, at denne invariant også fører til logaritmiske højder. Vi prøver at bygge et AVL-træ så magert, som vi kan. Definer M_h til at være et AVL-træ af højde h med så få knuder som overhovedet muligt. Det er let at se, at

$$|M_0| = 1 \text{ og } |M_1| = 2$$

I det generelle tilfælde er M_h et træ af højde h. Det ene undertræ må derfor være af højde $h - 1$. Vi tager selvfølgelig M_{h-1}. Hvor lille kan det andet undertræ være? Invarianten siger, at det må have højde mindst $h - 2$. Vi vælger derfor M_{h-2}. Men så kan vi slutte, at

$$|M_h| = |M_{h-1}| + |M_{h-2}| + 1$$

En simpel induktion viser, at $|M_h| \geq F_{h+2}$, hvor F_n er det n'te *Fibonacci-tal*, defineret som

$$\begin{aligned} F_1 &= 1 \\ F_2 &= 1 \\ F_n &= F_{n-1} + F_{n-2} \end{aligned}$$

Da det yderligere vides, at

$$F_n \in \Theta(\phi^n), \text{ hvor } \phi = \frac{\sqrt{5}+1}{2}$$

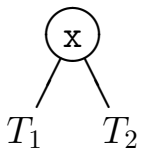
kan vi slutte, at det for ethvert AVL-træ T gælder, at

$$\text{højde}(T) \in O(\log_\phi |T|) = O(\log |T|)$$

så alle højder er logaritmiske.

7.8 Vægtbalancerede træer

I stedet for eksplicit at sætte begrænsninger på træernes højder, så kan man forlange, at *størrelserne* af en knudes undertræer ikke må være alt for forskellige. Et eksempel herpå er de såkaldte BB[α]-træer. Her står BB for Bounded Balance, og α er et reelt tal i intervallet $0 \leq \alpha \leq \frac{1}{2}$. Et binært træ tilhører BB[α] såfremt der for enhver knude gælder, at forholdet mellem antallet af knuder i dens undertræer er begrænset af α. Mere præcist forlanges det for et træ af formen

$T \quad = \quad$ 

at

i) $|T| \leq 1$, eller

ii) $\alpha \leq \dfrac{|T_1|+1}{|T|+1} \leq 1-\alpha$ og både T_1 og T_2 tilhører BB[α].

Det er nemt at indse, at denne definition er symmetrisk imellem de to undertræer, fordi det af ii) følger, at der også gælder

$$\alpha \leq \frac{|T_2|+1}{|T|+1} \leq 1-\alpha$$

At BB[α]-træer også har logaritmiske højder følger af, at der for et vilkårligt BB[α]-træ T gælder

$$(*) \qquad \text{højde}(T) \leq \frac{\log(|T|+1)-1}{\log(1/(1-\alpha))}$$

Dette vises ved induktion efter $|T|$ på følgende måde.

Basis $|T| = 1$: Når T kun indeholder en enkelt knude er højde$(T) = 0$. Men da log betyder 2-talslogaritmen er $\log(1+1) - 1$ også lig med 0.

Induktionsskridt $|T| = n + 1$: T har formen

$$T \quad = \quad \begin{array}{c} \texttt{x} \\ T_1 \quad T_2 \end{array}$$

hvor $|T_1| \leq n$ og $|T_2| \leq n$, dvs $(*)$ kan antages at gælde for T_1 og T_2. T's højde er givet ved

$$\begin{aligned} \text{højde}(T) &= 1 + \max\{\text{højde}(T_1), \text{højde}(T_2)\} \\ &\leq 1 + \max\{\frac{\log(|T_1|+1)-1}{\log(1/(1-\alpha))}, \frac{\log(|T_2|+1)-1}{\log(1/(1-\alpha))}\} \end{aligned}$$

på grund af induktionsantagelsen. Da såvel T_1 som T_2 er BB[α]-træer, og mindst et af dem er ikke-tomt, følger det at

$$\max\{\frac{\log(|T_1|+1)-1}{\log(1/(1-\alpha))}, \frac{\log(|T_2|+1)-1}{\log(1/(1-\alpha))}\} \leq \frac{\log((1-\alpha)(|T|+1))-1}{\log(1/(1-\alpha))}$$

hvorfor

$$\begin{aligned} \text{højde}(T) &\leq 1 + \frac{\log((1-\alpha)(|T|+1))-1}{\log(1/(1-\alpha))} \\ &= 1 + \frac{\log(1-\alpha)+\log(|T|+1)-1}{\log(1/(1-\alpha))} \\ &= \frac{\log(|T|+1)-1}{\log(1/(1-\alpha))} \end{aligned}$$

hvilket skulle vises.

Vi skal ikke komme yderligere ind på BB[α]-træers egenskaber, men blot bemærke, at det, i lighed med højdebalancerede træer, er muligt at opretholde BB[α] egenskaben under indsættelse og og fjernelse med en omkostning der er proportional med træets højde. Dette sker for såvel BB[α]-træer som for de højdebalancerede træer ved hjælp af *rotationer* og *dobbeltrotationer*, som er omorganiseringer af træet svarende til reglerne 4.1 og 4.2 for rød-sorte træer.

7.9 Tider for balancerede søgetræer

Det følger af ovenstående, at hvis en ordbog implementeres ved hjælp af balancerede søgetræer, så fås følgende kompleksiteter

$$\begin{aligned} T[\![\mathbf{Init}[D]]\!] &\in O(1) \\ T[\![\mathbf{Insert}[D](x)]\!] &\in O(\log |D|) \\ T[\![\mathbf{Member}[D](x)]\!] &\in O(\log |D|) \\ T[\![\mathbf{Delete}[D](x)]\!] &\in O(\log |D|) \end{aligned}$$

Læseren vil også have opdaget, at vi kan benytte (balancerede) søgetræer til en effektiv implementation af *prioritetskøer*. Som nævnt kan de ekstreme værdier findes som de yderligste knuder i søgetræet. Det leder til en version af prioritetskøer, hvor man kan fjerne både det største og mindste element med kompleksiteter

$$\begin{aligned} T[\![\mathbf{Init}[P]]\!] &\in O(1) \\ T[\![\mathbf{Insert}[P](x)]\!] &\in O(\log |P|) \\ T[\![\mathbf{DeleteMin}[P,x]]\!] &\in O(\log |P|) \\ T[\![\mathbf{DeleteMax}[P,x]]\!] &\in O(\log |P|) \end{aligned}$$

Forskellige udvidelser af søgetræer kan realisere mange andre operationer i logaritmisk tid.

8 Ækvivalensrelationer

Lad E være en type med endelig (og lille) værdimængde. En *ækvivalensrelation* over E er en datatype, hvis værdimængde består af klassedelinger af E's værdimængde.

En *klassedeling* af en mængde S er en samling af *klasser* $\{S_i\}$, således at

- $\forall i : (S_i \subseteq S) \wedge (\ S_i \neq \emptyset)$
- $\bigcup_i S_i = S$
- $i \neq j \Rightarrow S_i \cap S_j = \emptyset$

En klassedeling repræsenterer en matematisk ækvivalensrelation, hvor to elementer er relaterede, hvis og kun hvis de tilhører den samme klasse.

Denne datastruktur finder som regel anvendelse, når elementerne kan nummereres på en enkel måde. Vi vil derfor fremover antage, at

$$E = \{0, 1, 2, \ldots, N-1\}$$

Lad R være en variabel af type ækvivalensrelation og lad x_1,x_2 være værdier af type E. Der skal være følgende udtryk

$$\begin{array}{rll} \textbf{Init}[R] & \textbf{sat} & R' = \{\{0\}, \{1\}, \ldots, \{N-1\}\} \\ \textbf{Equiv}[R](x_1, x_2) & \textbf{sat} & (R' = R) \wedge (\text{Equiv}' = (\exists K \in R : x_1, x_2 \in K)) \\ \textbf{Join}[R](x_1, x_2) & \textbf{sat} & R' = R \setminus \{K_1, K_2\} \cup \{K_1 \cup K_2\} \\ & & \text{hvor } x_1 \in K_1 \in R \text{ og } x_2 \in K_2 \in R \end{array}$$

Man starter med den diskrete ækvivalensrelation, som man kan gøre grovere og grovere. Undervejs kan man spørge, om to elementer er relaterede.

8.1 Kanoniske repræsentanter

En simpel implementation får man, hvis man indfører *kanoniske repræsentanter* for klasserne, i.e. et specielt element fra hver klasse, der repræsenterer denne.

En klassedeling over E kan nu repræsenteres ved hjælp af en afbildning, $\rho : E \to E$, hvor $\rho(e)$ er den kanoniske repræsentant for den klasse som e tilhører. Vi kan repræsentere ρ ved hjælp af typen

$$\textbf{Type}\ \mathrm{EqRel} = \textbf{List}(\mathrm{Int})$$

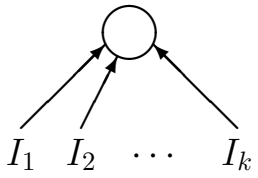
Hvis P er en variabel af denne type, der repræsenterer en ækvivalensrelation over E, så er $|P| = N$ og den kanoniske repræsentant for $e \in E$ er $P.(e)$. $\textbf{Init}[P]$ skal sætte P til identitetsafbildningen, $\textbf{Equiv}[P](x_1, x_2)$ skal teste om $P.(x_1) = P.(x_2)$, og $\textbf{Join}[P](x_1, x_2)$ skal ændre alle elementer med værdi $P.(x_1)$ til at have værdi $P.(x_2)$ (eller omvendt). Vi får følgende kompleksiteter

$$\begin{aligned} T[\![\textbf{Init}[P]]\!](N) &\in \Omega(N) \\ T[\![\textbf{Equiv}[P](x_1, x_2)]\!](N) &\in O(1) \\ T[\![\textbf{Join}[P](x_1, x_2)]\!](N) &\in \Omega(N) \end{aligned}$$

Som sædvanligt kommer træerne os til undsætning ved at tillade en mere effektiv implementation.

8.2 Inverse træer

Et *inverst træ (I-træ)* er enten *tomt*, eller det er en *knude*, der bliver peget på af et antal inverse træer



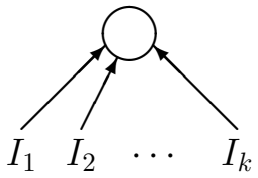
Vi kan repræsentere en klassedeling af E som en skov af I-træer med E-værdier i knuderne. Hvert I-træ svarer til en klasse, der består af elementerne i knuderne.

Ideen er, at man udnævner værdien i roden til kanonisk element for den tilsvarende klasse. Hvis vi kan holde I-træerne rimeligt balancerede, kan logaritmiske udførelsestider skimtes i horisonten.

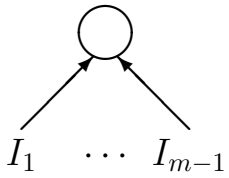
Det er let at udføre **Join**. Vi skal blot lade roden i det ene I-træ pege på roden af det andet. En indlysende optimalisering er at lade roden af det *letteste* I-træ pege på roden af det *tungeste*, hvor vi med *vægt* tænker på antallet af knuder. Vi må således holde rede på antallet af knuder i hvert I-træ.

Vi får brug for følgende observation: Et I-træ af højde h, opbygget ved hjælp af **Join**-operationer, indeholder mindst 2^h knuder.

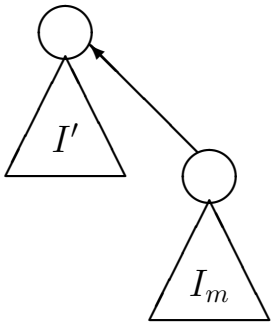
Dette kan vises ved induktion i højden. For $h = 0$ er antallet af knuder lig med $1 = 2^0$. Antag nu, at egenskaben gælder for I-træer med højde mindre end h. Betragt et generelt I-træ af højde h. Det ser ud som følger



Da I-træet har højde h, så må mindst ét undertræ have højde $h-1$. Antag, uden tab af generalitet, at I_j'erne er ordnet efter den rækkefølge, som de er blevet **Join**'et på, og at I_m har højde $h-1$. Kald I-træet



for I'. Så har vi **Join**'et



hvor I' så har flere knuder end I_m. Induktionsantagelsen giver os, at I_m indeholder mindst 2^{h-1} knuder, og da I' har mindst lige så mange, har vi ialt mindst

$$2^{h-1} + 2^{h-1} = 2^h$$

knuder. Vores oprindelige I-træ har alle disse knuder, plus hvad der blev **Join**'et til yderligere, nemlig $I_{m+1}, \ldots, I_k$.

Det er nu let at se, at alle I-træerne i repræsentationen af en klassedeling har logaritmiske højder. Antag, at vi har N forskellige elementer, og at et af I-træerne har højde $h > \log(N)$. Så må dette I-træ indeholde mindst

$$2^{\log(N)+1} = 2(2^{\log(N)}) = 2N$$

knuder, hvilket er dobbelt så mange som vi har. Derfor er højden af ethvert I-træ logaritmisk. Vi får således følgende kompleksiteter

$$\begin{aligned} T[\![\mathbf{Init}[P]]\!](N) &\in \Theta(N) \\ T[\![\mathbf{Equiv}[P](x_1, x_2)]\!](N) &\in O(\log N) \\ T[\![\mathbf{Join}[P](x_1, x_2)]\!](N) &\in O(\log N) \end{aligned}$$

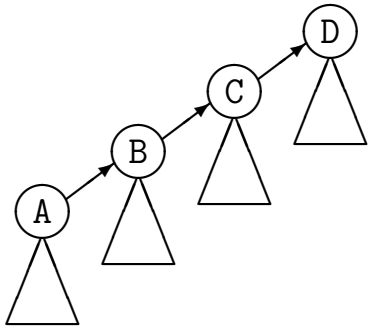
En box, der implementerer ækvivalensrelationer af heltal, ser ud som følger

```
Box Eq
    Type Rel = List(Data)
    Type Data = Prod(father: Int, weight: Int)

    Proc Init[R: Rel](n: Int)
        R := Rel(Data(0, 0) | n)
        (+ Var i: Int
           i := 0
           do i < n → R.(i), i := Data(i, 1), i+1 od
        +)
    end Init

    Proc Root[R: Rel](e: Int) → (Int)
        do R.(e).father ≠ e → e := R.(e).father od
        return e
    end Root
```

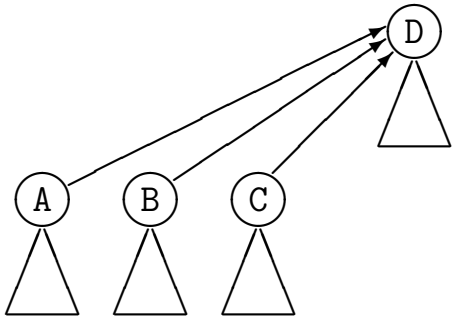
```
    Proc Equiv [R: Rel] (x1, x2: Int) → (Bool)
        return Root [R] (x1) = Root [R] (x2)
    end Equiv

    Proc Join [R: Rel] (x1, x2: Int)
        if ¬Equiv [R] (x1,x2) →
            (+ Var r1, r2: Int
               r1, r2 := Root [R] (x1), Root [R] (x2)
               if R.(r1).weight > R.(r2).weight → r1 :=: r2 fi
               R.(r1).father := r2
               R.(r2).weight := R.(r1).weight+R.(r2).weight
            +)
        fi
    end Join
end Eq
```

8.3 Vejforkortning

Man kan foretage en yderligere optimalisering af denne implementation. Når man har fundet roden for en knude, kan man uden yderligere omkostning lade alle knuderne på vejen pege direkte til roden. Det vil jo forkorte vejen ved senere rodsøgninger. Man transformerer altså et træ af formen



til et af formen



Man kan (med noget besvær) vise, at en vilkårlig sekvens af **Join**- og **Equiv**-operationer af længde m nu kan udføres i tid

$$O(m \log^*(N))$$

hvor vi definerer

$$\log^*(x) = \min\{i \mid \log^i(x) \leq 1\}$$
$$\log^0(x) = x$$
$$\log^i(x) = \log(\log^{i-1}(x))$$

Intuitivt er $\log^*(x)$ det antal gange man skal tage logaritmen for at komme fra x til 1. Dette er bedre end kompleksiteten

$$O(m \log(N))$$

som vi uden vejforkortning ville opnå for sekvensen af operationer. Faktisk er $\log^*(x)$ så langsomt voksende, at den ikke overstiger 5, selv om x er langt større end antallet af elementarpartikler i universet.

Vi kan implementere vejforkortningen ved blot at ændre rodsøgningen således

```
Proc Root[R: Rel](e: Int) → (Int)
    if R.(e).father ≠ e →
        R.(e).father := Root[R](R.(e).father)
    fi
    return R.(e).father
end Root
```

Vejen, der skal forkortes, ligger implicit gemt på rekursionsstakken. Bemærk, at proceduren repræsenterer et eksempel på en værdiprocedure med en hensigtsmæssig sideeffekt.

9 Amortiseret analyse

I de foregående kapitler har vi set en lang række eksempler på analyser af en datastrukturs operationer. Fælles for disse har været, at vi som sædvanligt har interesseret os for *worst-case* udførelsestiderne. I dette kapitel skal vi betragte en anden type udførelsestid, nemlig den såkaldte *amortiserede* udførelsestid. Dette er en formel måde at udtrykke, at en operation med høj worst-case udførelsestid godt kan være effektiv, hvis "worst-case optræder tilstrækkeligt sjældent".

9.1 Binær kilometertæller som datastruktur

Betragt som eksempel følgende datastruktur, der indeholder en binær tæller, som kan nulstilles, aflæses og tælles op med én. Dette er en datastruktur variant af algoritmen Optælling side 31.

```
Box Counter
    Type Bcounter = Vector

    Proc Init[R: Bcounter]
        R := Vector()
    end Init

    Proc ReadOut[R: Bcounter] → (Int)
        (+ Var i, x, r: Int
           i, x, r := 0, 1, 0
           do (i<| R |) →
                r := r+R.(i)*x
                x, i := 2*x, i+1
           od
           return r
        +)
    end ReadOut
```

```
    Proc Inc[R: Bcounter]
        (+ Var i: Int
           i := 0
           do(i<| R |) ∧ (R.(i)  = 1) → R.(i), i := 0, i+1 od
           if i = | R | → R := R++Vector(0) fi
           R.(i) := 1
        +)
    end Inc
end Counter
```

Det er klart, at de tre procedurer har følgende worst-case udførelsestider

$$\begin{array}{lcl} T[\![\text{Init}[R]]\!] & \in & O(1) \\ T[\![\text{ReadOut}[R]]\!] & \in & O(|R|) \\ T[\![\text{Inc}[R]]\!] & \in & O(|R|) \end{array}$$

Vi kan imidlertid i lighed med algoritmen Optælling argumentere for, at det er sjældent, at vi i proceduren Inc skal overføre menten hele vejen igennem R. Faktisk kunne vi gentage argumentet med bogføringsinvarianten og vise, at hvis vi betaler (noget der er proportionalt med) 2 kr. for hvert kald af Inc, så kan *samtlige* kald af Inc finansieres ad denne vej. Vi siger, at den *amortiserede* udførelsestid for Inc er konstant, hvilket vi skriver som

$$T_A[\![\text{Inc}[R]]\!] \in O(1)$$

9.2 Potentialfunktion

På basis af ovenstående datastruktur kan vi nu på én og samme tid præcisere begrebet amortiseret kompleksitet og generalisere idéen med bogføringsinvarianter. Midlet er begrebet *potentialfunktion*, som er en funktion, der er knyttet til en datastruktur (som Counter ovenfor), og som udtrykker, hvor megen "potentiel energi", der er opsparet fra tidlige operationer til brug for senere operationer. En potentialfunktion, Φ, afbilder (i lighed med en termineringsfunkion) datastrukturen ind i de hele (eller måske reelle) tal. Den anvendes i følgende generelle sammenhæng.

Vi betragter en datastruktur D, som har startværdi D_0 og "udsættes" for en følge af n operationer (som kald af procedurerne Init, ReadOut og Inc), som resulterer i, at den gennemløber værdierne

$$D_0, D_1, D_2, \ldots, D_i, \ldots, D_n$$

hvor den i'te operation transformerer D_{i-1} til D_i.

Hvis c_i er den *reelle* omkostning ved den i'te operation, definerer vi den *amortiserede omkostning* ved den i'te operation som

$$\bar{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1})$$

dvs. som summen af den reelle omkostning og stigningen i potentialet.

Hvis man nu sørger for at vælge Φ så den opfylder

(*) $\quad \Phi(D_0) \leq \Phi(D_n)$

gælder der

$$\begin{aligned}
\sum_{i=1}^{n} c_i &= \sum_{i=1}^{n} (\bar{c}_i + \Phi(D_{i-1}) - \Phi(D_i)) \\
&= (\sum_{i=1}^{n} \bar{c}_i) + \Phi(D_0) - \Phi(D_n) \\
&\leq \sum_{i=1}^{n} \bar{c}_i
\end{aligned}$$

dvs. at summen af de amortiserede omkostninger er en øvre grænse for summen af de reelle omkostninger. (I praksis opfyldes (*) ofte ved at sørge for, at $\Phi(D_0) \leq \Phi(D_i)$ for alle $i \leq n$.)

Anvendelsen af potentialfunktionen på datastrukturen Counter er som følger. Selve datastrukturen er vektoren R, på hvilken vi definerer potentialfunktionen ved

$$\Phi(R) = \alpha * (\text{antallet af 1'ere i } R)$$

hvor α er en proportionalitetsfaktor, vi bestemmer senere. Idet vi går ud fra, at R_0 er tom, er det klart, at $\Phi(R_0) \leq \Phi(R_i)$ for alle $i \geq 0$.

Den amortiserede omkostning for en operation P er nu

$$T_A[\![P[R]]\!] = T[\![P[R]]\!] + \Phi(R') - \Phi(R)$$

hvor R' (i lighed med anvendelsen i specifikationer) angiver værdien af R efter kaldet af P.

De amortiserede omkostninger ved operationerne Init, ReadOut og Inc er som følger.

Init

$$\begin{aligned} T_A[\![\text{Init}[R]]\!] &= T[\![\text{Init}[R]]\!] - \Phi(R) \\ &\leq T[\![\text{Init}[R]]\!] \\ &\in O(1) \end{aligned}$$

idet $R' = 0$.

ReadOut
ReadOut ændrer ikke på R, dvs.

$$T_A[\![\text{ReadOut}[R]]\!] = T[\![\text{ReadOut}[R]]\!] \in O(|R|))$$

Inc
Antag, at R har udseende

$$R = (\underbrace{1, 1, 1, \ldots, 1}_{x}, 0, ?, \ldots, ?)$$

Så er

$$R' = (\underbrace{0, 0, 0, \ldots, 0}_{x}, 1, ?, \ldots, ?)$$

hvorfor

$$\Phi(R') - \Phi(R) = \alpha - \alpha x = \alpha(1 - x)$$

Men nu følger det af kroppen af proceduren Inc, at udførelsestiden for Inc[R] er proportional med $x+1$, dvs. at der findes en konstant β, så

$$T[\![\text{Inc}[R]]\!] \leq \beta(x+1)$$

Vi har da

$$\begin{aligned} T_A[\![\text{Inc}[R]]\!] &= T[\![\text{Inc}[R]]\!] + \Phi(R') - \Phi(R) \\ &\leq \beta(x+1) + \alpha(1-x) \end{aligned}$$

Hvis vi nu vælger $\alpha = \beta$ får vi

$$T_A[\![\text{Inc}[R]]\!] \leq 2\alpha \in O(1)$$

9.3 En monoton Sequence

Lad os som et mere interessant eksempel betragte box'en Sequence(T) fra afsnit 10.5 i [TRINE]. Vi ser først på følgende "monotone delbox", hvor vi har fjernet Lpop og Rpop.

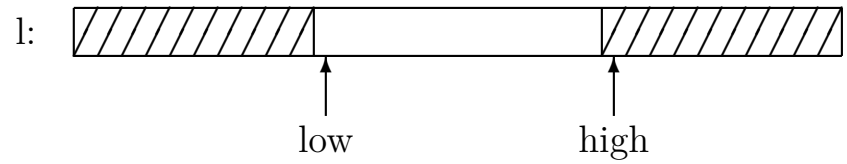
```
Box SS(T)
    Type Tlist = List(T)
    Type Seq = Prod(low, high: Int, l: Tlist)

    Proc Init[s: Seq]
        s := Seq(0, 0, Tlist())
    end Init

    Proc Len[s: Seq] → (Int)
        return s.high-s.low
    end Len

    Proc Lpush[s: Seq](t: T)
        if |s.l| = 0 → s := Seq(0, 1, Tlist(t))
        & s.low>0 →
            s.low := s.low-1
```

```
            s.l.(s.low) :=: t
        & true → s.l, s.low, s.high := Tlist(t | | s.l |)++s.l, | s.l |-1, s.high+| s.l |
        fi
    end Lpush

    Proc Ltop[s: Seq] → [T]
        if Len[s] = 0 → abort("Ltop on empty sequence") fi
        return s.l.(s.low)
    end Ltop


    Proc Rpush[s: Seq](t: T)
        if | s.l | = 0 → s := Seq(0, 1, Tlist(t))
        & s.high<| s.l | →
            s.l.(s.high) :=: t
            s.high := s.high+1
        & true → s.l, s.high: tt = s.l++Tlist(t | | s.l |), s.high+1
        fi
    end Rpush

    Proc Rtop[s: Seq] → [T]
        if Len[s] = 0 → abort("Rtop on empty sequence") fi
        return s.l.(s.high-1)
    end Rtop

    Proc Sel[s: Seq](i: Int) → [T]
        if 0 ≤ i<Len[s] → return s.l.(s.low+i)
        & true → abort("Seq index out of range")
        fi
    end Sel

    Proc Print[s: Seq]
        write(s.l(s.low .. s.high) )
    end Print
end SS
```

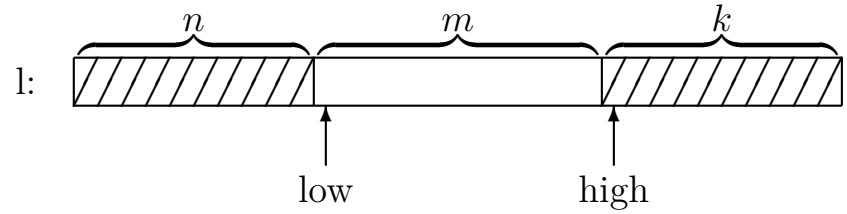
Det er klart, at der for procedurerne Init, Len, Ltop, Rtop og Sel gælder, at de har konstante udførelsestider. Det er også klart, at Print har lineær

udførelsestid.

Med hensyn til Rpush (og Lpush), kan vi udtrykke udførelsestiden på følgende måde. Typen Sequence er implementeret som en liste



hvor det skraverede er "spild". Rpush foretager sig kun noget "dyrt", når high peger på enden af listen, og i så fald er omkostningen proportional med længden af listen. Det er derfor relevant at udtrykke kompleksiteten af Rpush v.hj.a. følgende tre karakteristiske parametre n, m og k.



Hermed bliver tidskompleksiteten af Rpush

$$T[\![\textbf{proc Rpush}]\!](n,m,k) \in \begin{cases} O(1) & \text{hvis } k > 0 \\ O(n+m+1)^4 & \text{hvis } k = 0 \end{cases}$$

dvs. der findes en konstant α så

$$T[\![\textbf{proc Rpush}]\!](n,m,k) \leq \begin{cases} \alpha & \text{hvis } k > 0 \\ \alpha(n+m+1) & \text{hvis } k = 0 \end{cases}$$

Som potentialfunktion vælger vi nu

(*) $\quad \Phi(n,m,k) = \alpha(3m - (n+k))$

Vi får da følgende, idet vi betegner ændring i potentialet $\Phi(n',m',k') - \Phi(n,m,k)$ med $\Delta\Phi$

$$T_A[\![\textbf{proc Rpush}]\!] = T[\![\textbf{proc Rpush}]\!] + \Delta\Phi$$

[4] 1'tallet skyldes, at udsagnet også skal gælde, når $n = m = 0$

Vi ser nu på hvert af de to tilfælde .

$\underline{k > 0}$
Så er $n' = n$, $m' = m + 1$ og $k' = k - 1$, hvorfor

$$\Delta\Phi = \alpha(3(m+1) - (n+k-1)) - \alpha(3m - (n+k)) = 4\alpha$$

Men så er

$$\begin{aligned} T_A[\![\textbf{proc Rpush}]\!](n,m,k) &\leq 5\alpha \\ &\in O(1) \end{aligned}$$

$\underline{k = 0}$
I dette tilfælde fordobles listen, dvs. $n' = n$, $m' = m+1$ og $k' = n+m-1$. Vi har da

$$\begin{aligned} \Delta\Phi &= \alpha(3(m+1) - (2n+m-1)) - \alpha(3m-n) \\ &= \alpha(4 - (n+m)) \end{aligned}$$

Så er

$$\begin{aligned} T_A[\![\textbf{proc Rpush}]\!](n,m,k) &= T[\![\textbf{proc Rpush}]\!](n,m,k) + \Delta\Phi \\ &\leq \alpha(n+m+1) + \alpha(4-(n+m)) = 5\alpha \\ &\in O(1) \end{aligned}$$

Hvis vi nu også kan vise, at $\Phi(n,m,k) \geq 0$ har vi altså, at

$$T_A[\![\textbf{proc Rpush}]\!] \in O(1)$$

Det er klart, at der tilsvarende gælder, at

$$T_A[\![\textbf{proc Lpush}]\!] \in O(1)$$

For at vise, at $\Phi(n,m,k) \geq 0$, bemærker vi først, at initialværdien er lig med 0, fordi

$$\Phi(0,0,0) = 0$$

Dernæst observerer vi, at potentialfunktionens værdi *vokser*, når Rpush udføres for $k > 0$. Tilbage er da kun at vise, at Φ's værdi også er ikke-negativ efter en udførelse af Rpush, hvor $k = 0$. Efter en sådan udførelse er værdien af Φ

$$\Phi(n, m+1, n+m-1) = 2(m-n)+4$$

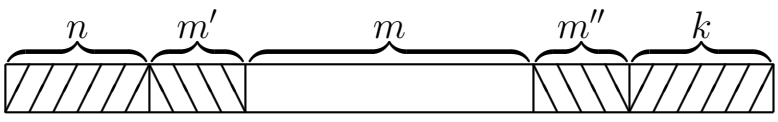
At den er ikke-negativ følger nu af den simple observation, at det *midterste* element i listen aldrig er spild. Når $k = 0$ betyder dette, at $m \geq n$, hvorfor $2(m-n)+4 > 0$, dvs.

$$\Phi(n, m+1, m+n-1) > 0.$$

9.4 Datatypen Sequence

Hvis vi betragter den "totale" box Sequence(T), dvs. den box vi får ved at tilføje Lpop og Rpop til SS, er det intuitivt klart, at de amortiserede udførelsestider for Lpush og Rpush vedbliver at være konstante. Begrundelsen er, at anvendelser af Lpop og Rpop blot betyder, at der går endnu længere tid mellem de "dyre" operationer Rpush og Lpush. Heraf følger det imidlertid ikke formelt, at Rpush og Lpush også har konstante amortiserede udførelsestider i box'en Sequence(T). Problemet er, at det ikke længere er rigtigt, at potentialfunktionen (*) er positiv, når vi også må anvende Lpop og Rpop.

Da vi kun kan have én potentialfunktion tilknyttet en datastruktur, skal vi derfor finde et alternativ til Φ ovenfor, som på samme tid er ikke-negativ og fører til konstante amortiserede udførelsestider for både Rpush og Lpush samt Rpop og Lpop. Betragt nu følgende inddeling af listen



hvor det skraverede stadig er spild, men hvor det nu er inddelt i "jomfrueligt" spild (n og k) og "almindeligt" spild (m' og m''). Forskellen på de to typer spild er, at jomfrueligt spild aldrig har indeholdt et "rigtigt" element, hvorimod almindeligt spild på et tidspunkt har indeholdt noget, som senere

er blevet pop'et. Hvis vi nu som alternativ definerer potentialfunktionen ved

$$\Phi(n, m', m, m'', k) = \alpha(3(m + m' + m'') - (n + k))$$

kan vi stort set gentage argumenterne ovenfor, dvs. vi kan dels vise, at

$$\Phi(n, m', m, m'', k) \geq 0$$

og dels at de amortiserede udførelsestider for såvel Rpush (Lpush) som Rpop (Lpop) er konstante.

Som afslutning på dette kapitel bemærker vi, at vi både her og i forrige afsnit har en oplagt analogi med "opsparingsfilosofien" fra bogføringsinvarianten (jfr. afsnit 3.5). Når vi udfører en Rpush i en situation, hvor $k > 0$ (og $m'' = 0$), fører dette til en *stigning* i potentialfunktionens værdi på 4α. Dette svarer til at spare 4 kr. op til brug for at betale kommende "dyre" Rpush'er.

Bemærk også, at vi med potentialfunktioner har indført endnu en målemekanisme, der hjælper os med at ræsonnere om algoritmer og datastrukturer. En potentialfunktion er, i lighed med termineringsfunktioner og udsagn, en funktion, hvis domæne er algoritmers eller datastrukturers tilstand, og som bruges i forbindelse med argumentation for korrekthed og/eller effektivitet. Bemærk også, at disse funktioner kun anvendes til at *måle* på tilstande, de påvirker dem ikke, dvs. hverken udsagn eller termineringsfunktioner har nogen indflydelse på beregningerne.

10 Korrekthed af procedurer

Når man har behov for at argumentere for korrekthed af algoritmer, der indeholder procedurekald, rejser der sig spørgsmålet om, hvordan man kan få procedurekaldenes egenskaber passet ind i korrekthedsbeviserne.

10.1 Ikke-rekursive procedurer

Vi kan igen betragte algoritmen Permutation fra kapitel 4, men denne gang skrevet v.hj.a. procedurer på følgende måde. (S er en mængdetype med elementer fra E).

Algoritme: Permutation
Stimulans : f: **List** (E), $\forall i, j \in 0..|f| : f.(i) \neq f.(j)$
Respons : f: permuteret
Metode : **proc** insert $[s : S](e : E)$
≪indsæt e i s ≫
end insert

proc delete $[s : S](e : E)$
≪fjern e fra s ≫
end delete

proc select $[s : S, e : E]$
if $s \neq \emptyset \rightarrow \ll e :=$ element i $s \gg$
& true →**abort**
fi
end select

$K, i := \emptyset, 0$
do $i < |f| \rightarrow$ insert $[K](f.(i))$
$i := i + 1$
od
$i := 0$
do $i < |f| \rightarrow$ select $[K, k]$
delete $[K](k)$
$f.(i), i := k, i + 1$
od

Procedurernes (eller rettere procedurekaldenes) egenskaber vil igen blive angivet v.hj.a. specifikationer på følgende måde

insert $[s](e)$ **sat** $s' = s \cup \{e\}$
delete $[s](e)$ **sat** $s' = s \backslash \{e\}$
select $[s, e]$ **sat** $s \neq \emptyset \rightarrow (e' \in s) \wedge (s = s')$

og betydningen er præcis den samme som i kapitel 4.

Der kan også blive behov for at specificere en værdiprocedure som følgende alternative udgave af select

proc select$[s : S] \rightarrow (E)$
  if $s \neq \emptyset \rightarrow \ll$returner et e i $s \gg$
  & true $\rightarrow$ **abort**
  fi
end select

I sådanne tilfælde bruger vi selve procedurenavnet til at angive værdien af procedurekaldet, dvs. vi får en specifikation af følgende form

select $[s]$ **sat** $s \neq \emptyset \rightarrow (\text{select}' \in s) \wedge (s = s')$.

Procedure*kaldenes* egenskaber beskrives altså v.hj.a. specifikationer, og disse kan så på sædvanlig vis anvendes i ræsonnementer om de algoritmer eller programmer, hvori kaldene optræder. Spørgsmålet er nu, hvordan man bærer sig ad med at vise, at et procedurekald opfylder en specifikation. Svaret er, at det gør man ved at vise, at procedure*kroppen* opfylder den givne specifikation. Dette forudsætter imidlertid, at specifikationen er udtrykt i termer af procedurens *formelle* parametre, så når den skal bruges i forbindelse med et kald, skal specifikationens formelle parametre erstattes af de *aktuelle* parametre fra kaldet. Vi kan udtrykke dette mere præcist på følgende måde.

Betragt en proceduredefinition af formen

proc $p[r : R](v : V) \rightarrow (E)$
  $\ll$Krop$\gg$
end p

Hvis man kan vise, at kroppen opfylder specifikationen

$$\ll\text{Krop}\gg \textbf{ sat } P(r,v) \to U(r,r',v,p')$$

hvor $P(r,v)$ ($U(r,r',v,p')$) er prædikater med frie variabler r,v (r,r',v,p'), så gælder der, at $p[r](v)$ opfylder specifikationen

$$p[r](v) \textbf{ sat } P(r,v) \to U(r,r',v,p')$$

Herudover gælder der for et vilkårligt andet kald

$$p[a](e)$$

hvor a er en variabel af type R og e er et værdiudtryk af type V, at dette kald opfylder den specifikation, der opnås ved i P og U at erstatte de formelle parametre med de aktuelle, dvs.

$$p[a](e) \text{ sat } P(a,e) \to U(a,a',e,p')$$

Vi kan angive dette mere skematisk v.hj.a. følgende såkaldte bevisprincip

Bevisprincip for ikke-rekursive procedurer

Hvis der for proceduren

proc $p[r : R](v : V) \to (E)$
  $\ll$Krop$\gg$
end p

gælder

$\ll\text{Krop}\gg$ **sat** $P(r,v) \to U(r,r',v,p')$

så gælder der for ethvert kald $p[a](e)$ at

$p[a](e)$ **sat** $P(a,e) \to U(a,a',e,p')$

Princippet gælder også for procedurer med flere parametre *forudsat* at ingen aktuel [ ]-parameter er identisk med eller delvariabel af andre aktuelle [ ]-parametre.

Betragt som eksempel på en anvendelse af bevisprincippet følgende algoritme til såkaldt *geometrisk søgning*, der anvender en kvadratrodsalgoritme. Algoritmen er næsten identisk med Binær Søgning side 14. Forskellen er kun, at i stedet for at beregne kandidatmængdens midtpunkt som det *aritmetiske* gennemsnit (lav+høj)/2, bruges nu det *geometriske* gennemsnit $\sqrt{\text{lav} * \text{høj}}$. Kvadratroden udregnes v.hj.a. en procedure, der returnerer heltalskvadratroden af sit argument.

Algoritme : Geometrisk søgning
Stimulans : S : **List** (Int), ordnet søgedomæne
m : Int, målelement
Respons : $r : S(\alpha(r)) = S\ (S^{-1}(m))$
Metode : **proc** $sqrt(n : \text{Int}) \to (\text{Int})$
(+**Var** r : Int
$r := n$
do $n < r^2 \to r := r - 1$ **od**
return r
+)
end *sqrt*

lav,høj,r := $0, |S|,$ Res(tom : #)
do
(lav < høj) $\wedge$ **is** $(r, \text{tom}) \to$
midt := $sqrt(\text{lav} * \text{høj})$
if $S.(\text{midt}) = m \to r :=$ Res$(x : \text{midt})$
| $S.(\text{midt}) < m \to \text{lav} := \text{midt} + 1$
| $m < S.(\text{midt}) \to \text{høj} := \text{midt}$
fi
od

Vi beviser nu, at proceduren sqrt tilfredsstiller specifikation

(*) sqrt(n) **sat** $n \geq 0 \to (\text{sqrt}')^2 \leq n < (\text{sqrt}' + 1)^2$

Iflg. bevisprincippet er det nok at vise, at

$$\left.\begin{array}{l} r := n \\ \textbf{do}\ n < r^2 \to r := r - 1\ \textbf{od} \\ \textbf{return}\ r \end{array}\right\} \textbf{sat}\ n \geq 0 \to (\text{sqrt}')^2 \leq n < (\text{sqrt}' + 1)^2$$

hvilket – da sqrt′ betegner den returnerede værdi og det er r der returneres – er det samme som at vise

$$\left.\begin{array}{l} r := n \\ \textbf{do } n < r^2 \rightarrow r := r - 1 \textbf{ od} \end{array}\right\} \textbf{sat } n \geq 0 \rightarrow (r')^2 \leq n < (r'+1)^2$$

At dette er rigtigt, indses ved at observere, at nedenstående invariant er gyldig og at beregningen standser

$$\begin{array}{l} r := n \\ \textbf{do } \{n < (r+1)^2\} \\ \quad n < r^2 \rightarrow r := r - 1 \\ \textbf{od} \end{array}$$

Altså kan vi konkludere, at (*) er opfyldt.

I et bevis for korrekthed af Geometrisk Søgning skal vi bruge kvadratrodsprocedurens egenskaber til at bevise terminering. Termineringsfunktionen er høj − lav − $|\alpha(r)|$, og for at sikre at den aftager, skal vi vise, at der efter udførelse af tilordningen

$$\text{midt} := \text{ sqrt}(\text{lav} * \text{høj})$$

gælder

$$\text{lav} \leq \text{midt} < \text{høj}$$

Beviset går som følger. Fra (*) og bevisprincippet fås

$$\text{midt}^2 \leq \text{lav} * \text{høj} < (\text{midt}+1)^2$$

og vi ved yderligere at

$$\text{lav} < \text{høj}$$

for ellers var vi ikke kommet ind i løkken.

Vi skal derfor blot vise, at

$$
\begin{array}{ll}
 & (\text{midt}^2 \leq \text{lav} * \text{høj} < (\text{midt}+1)^2) \wedge (\text{lav} < \text{høj}) \\
(**) & \Downarrow \\
 & \text{lav} \leq \text{midt} < \text{høj}
\end{array}
$$

Vi fører beviset indirekte. Antag først at midt $<$ lav. Så er midt $+ 1 \leq$ lav, hvoraf

$$(\text{midt}+1)^2 \leq \text{lav}^2 \leq \text{lav} * \text{høj}$$

hvilket giver modstriden. Hvis vi omvendt antager, at høj $\leq$ midt fås analogt

$$\text{midt}^2 \geq \text{høj}^2 > \text{lav} * \text{høj}$$

som igen giver modstrid. Herefter er (**) vist, og vi konkluderer, at Geometrisk Søgning er korrekt.

10.2 Rekursive procedurer

Når man skal bevise, at en rekursiv procedure tilfredsstiller en specifikation, er situationen på en vis måde analog til at analysere dens tidskompleksitet. Man må også her udtrykke en sammenhæng mellem proceduren selv og de rekursive kald. I forbindelse med korrekthedsbeviser har denne sammenhæng dog ikke form af en rekursionsligning med derimod af en *implikation*, hvor præmissen er en antagelse om, at de rekursive kald allerede har den ønskede egenskab.

Mere præcist gælder der følgende bevisprincip for rekursive procedurer[5] Funktionen μ er en sædvanlig termineringsfunktion, som skal bruges til at sikre, at rekursionen ikke bliver ved i det uendelige.

[5]Udvidelsen til at betragte værdiprocedurer er helt analog til sidste afsnit.

Bevisprincip for rekursive procedurer

Hvis der for den rekursive procedure

proc $p[r : R](v : V)$
  $\vdots$
  $p[s](u)$ $\ll$Krop$\gg$
  $\vdots$
end p

gælder

- $P(r, v) \Rightarrow \mu(r, v) > \mu(s, u) \geq 0$ for alle rekursive kald $p[s](u)$
- $p[s](u)$ **sat** $P(s, u) \rightarrow U(s, s', u)$
  $\Downarrow$
  $\ll$Krop$\gg$ **sat** $P(r, v) \rightarrow U(r, r', v)$

så gælder der for ethvert kald $p[a](e)$

$p[a](e)$ **sat** $P(a, e) \rightarrow U(a, a', e)$

Princippet gælder igen for procedurer med flere parametre og under samme forudsætning som for de iterative procedurer, dvs. der må ikke være overlap mellem forskellige aktuelle [ ]-parametre.

Lad os som eksempel på en anvendelse af princippet betragte proceduren Reverse_1. Hvis w er en Text-variabel, vil vi gerne vise at

$\text{Reverse}_1[w]$ **sat** $w' = \tilde{w}$

Iflg. princippet ovenfor skal vi så

- finde en termineringsfunktion μ
- bevise at

  $\text{Reverse}_1[L]$ **sat** $L' = \tilde{L}$
  $\Downarrow$
  $\ll$Krop$\gg$ **sat** $L' = \tilde{L}$

Termineringsfunktionen er simpel, idet vi blot tager

$$\mu(L) = |L|$$

Med hensyn til at vise implikationen er sagen klar i det tilfælde, hvor $|L| \leq 1$. Hvis $|L| \geq 1$ skal vibevise følgende

$$\left.\begin{array}{l} hlp, L := L(0..1), L(1..|L|) \\ \text{Reverse}_1\ [L] \\ L := L{+}{+}\ hlp \end{array}\right\} \textbf{sat}\ L' = \tilde{L}$$

Hvis vi bruger indicering til at betegne værdierne af variablerne før og efter de enkelte sætninger, får vi

$$\begin{array}{llcl} hlp_0 = ? & L_0 & = & L \\ hlp_1 = L_0(0..1) & L_1 & = & L_0(1..|L|) \\ hlp_2 = hlp_1 & L_2 & = & \tilde{L}_1 \text{ (p.gr.a. antagelsen om det lokale kald)} \\ hlp' = hlp_2 & L' & = & L_2{+}{+}hlp_2 \end{array}$$

som ved simpel udregning giver $L' = \tilde{L}$.

Det skal afslutningsvist bemærkes, at der er en nøje sammenhæng mellem det der i analyseafsnittet hedder *karakteristisk parameter* og den her anvendte termineringsfunktion. De har begge til formål at finde et aftagende *mål* for de tilstande, hvori proceduren kaldes rekursivt, og det vil ofte være sådan, at karakteristisk parameter og termineringsfunktion er identiske.

11 Eksempel: Erathostenes Si

I dette kapitel gennemgås et eksempel, som illustrerer de fleste af de begreber, der er introduceret i det foregående.

Eksemplet er den klassiske algoritme til at finde samtlige primtal mindre end eller lig med et givet tal $n \geq 2$, som går under navnet Erathostenes Si.

11.1 Den abstrakte algoritme

Fremgangsmåden kan illustreres på følgende måde for $n = 9$.

Vi opererer med to mængder P og K, som begge er delmængder af $\{2, \ldots, n\}$. P består af de hidtil fundne primtal, og K består af tal, der er kandidater til at blive indlemmet i P. Når beregningen starter, er situationen

$$P_0 = \emptyset$$
$$K_0 = \{2, 3, 4, 5, 6, 7, 8, 9\}$$

Det er klart, at K's mindste tal er et primtal, og det er også klart, at alle tal i K, der har dette tal som divisor, ikke er primtal. Første skridt består derfor i at flytte 2 over i P og at fjerne alle lige tal fra K, dvs. vi får

$$P_1 = \{2\}$$
$$K_1 = \{3, 5, 7, 9\}$$

K's mindste tal er igen et primtal, hvorfor vi får

$$P_2 = \{2, 3\}$$
$$K_2 = \{5, 7\}$$

Gentagelse af dette skridt giver

$$P_3 = \{2, 3, 5\}$$
$$K_3 = \{7\}$$

og endelig får vi

$P_4 = \{2, 3, 5, 7\}$
$K_4 = \emptyset$

hvorefter vi konkluderer, at 2, 3, 5 og 7 er primtallene blandt de første 9 tal.

Eksemplet antyder, at algoritmen skal bestå af en løkke med en invariant, som siger at P består af primtal, K af tal som ikke har divisorer fra P, samt at P og K tilsammen indeholder alle primtal blandt de n første tal. Dette kan udtrykkes formelt, hvis vi med $PRIM(n)$ betegner primtallene blandt $\{2, \ldots, n\}$, dvs.

$$PRIM(n) = \{p \mid (2 \leq p \leq n) \text{ og } p \text{ er et primtal}\}$$

og ligesom i Euklids Algoritme betegner største fælles divisor af to tal p og q med $sfd(p, q)$. Vi får følgende abstrakte algoritme

Algoritme : Erathostenes Si
Stimulans : $n : n \geq 2$
Respons : $P : P = PRIM(n)$
Metode : $P, K := \emptyset, \{2, 3, \ldots, n\}$
do $\{(P \subseteq PRIM(n) \subseteq P \cup K) \wedge (sfd(P, K) = 1)\}$[6]
$K \neq \emptyset \rightarrow h := min(K)$
$P := P \cup \{h\}$
$K := K - \{k \in K \mid h \text{ går op i } k\}$
od

Vi interesserer os nu lidt nærmere for denne algoritmes egenskaber, såvel de kvalitative som de kvantitative. Vi viser gyldighed, terminering og korrekthed på sædvanlig vis:

Terminering
Følger direkte ved at betragte termineringsfunktionen $\mu(K) = |K|$.

Gyldighed
Det er klart, at invarianten tilfredsstilles af initialiseringen af P og K. At invarianten bevares, reduceres til at bevise at

$$(P \subseteq PRIM(n) \subseteq P \cup K) \wedge (sfd(P, K) = 1) \wedge (K \neq \emptyset)$$

[6] $sfd(P, K)$ er et kompakt udtryk for prædikatet $\forall p \in P : \forall k \in K : sfd(p, k) = 1$.

$$\Downarrow$$

$$(P' \subseteq PRIM(n) \subseteq P' \cup K') \wedge (sfd(P', K') = 1)$$

hvor $P' = P \cup \{min(K)\}$
$K' = K - \{k \in K \mid min(K) \text{ går op i } k\}$

Denne implikation følger af at

a) $min(K)$ må være et primtal, for ellers ville det have en mindre primtalsfaktor, men denne måtte være med i P (fordi $PRIM(n) \subseteq P \cup K$), hvilket er i modstrid med at $sfd(P, K) = 1$

b) hvis $min(K)$ indsættes i P og det selv og alle dets multipla fjernes fra K, indeholder P' og K' de samme primtal som $P \cup K$, og vi har stadig, at $sfd(P', K') = 1$.

Korrekthed
Når algoritmen terminerer, gælder der

$$(P \subseteq PRIM(n) \subseteq P \cup K) \wedge (K = \emptyset)$$

hvoraf det direkte følger, at $P = PRIM(n)$.

11.2 Den konkrete algoritme

Vi kan ikke finde algoritmens udførelsestid ved at ræsonnere direkte på den foreliggende version. Problemet er, at de operationer der udføres på variablerne P og K *ikke* er primitive i kompleksitetsforstand. Vi må derfor realisere P og K (og deres operationer) på en sådan måde, at manipulationerne kommer tættere på de anerkendte primitiver. Dette kan gøres på følgende måde.

Variablen P skal kunne initialiseres til den tomme mængde, og derefter skal man kunne tilføje et element ad gangen. Begge disse operationer understøttes automatisk, hvis vi realiserer P som en heltals-sekvens på følgende måde

Box ISq
  Sequence(Int)
end ISq

Variablen K er mere interessant. Vi kan se, at den skal kunne initialiseres til $\{2, 3, \ldots, n\}$, man skal kunne finde dens mindste element, og man skal kunne fjerne "hvert h'te" element. Da K er monoton – mængden bliver aldrig større – kan vi realisere den effektivt v.hj.a. en bitvektor (jfr. kapitel 7), hvor vi yderligere vedligeholder en variabel, der indeholder antallet af elementer i K, samt én der peger på det sidst fundne primtal. Følgende tegning (B betegner bitvektoren, a antal elementer og p peger på det sidst fundne primtal) viser repræsentationen af mængden K_1 ovenfor.

B : $\boxed{\underline{\text{f}} \mid \underline{\text{f}} \mid \underline{\text{f}} \mid \underline{\text{t}} \mid \underline{\text{f}} \mid \underline{\text{t}} \mid \underline{\text{f}} \mid \underline{\text{t}} \mid \underline{\text{f}} \mid \underline{\text{t}}}$

a : 4

p : 2

Med disse realiseringer af P og M får den konkrete algoritme følgende udseende.

Algoritme : Erathostenes
Stimulans : $n : n \geq 2$
Respons : $P : P = PRIM(n)$
Metode :

```
ISq' Init[P]                                              } A
B, a, p := List(f | 2) ++ List(t | n − 1), n − 1, 0       }
do a > 0 →
        do ¬B.(p) → p := p + 1 od                         } B
        ISq' Rpush[P](p)
        i := p
        do i ≤ n →                                        }
            if B.(i) → B.(i), a := false, a − 1 fi        } C
            i := i + p                                    }
        od
od
```

11.3 Kompleksiteten

Analysen af denne algoritme er mere kompliceret end de eksempler, vi hidtil har set, bl.a. fordi det ikke er klart, hvor mange gange den yderste løkke gennemløbes. Dette antal kan vi imidlertid finde i følgende vigtige sætning fra talteorien.

Primtalssætningen
Lad $\pi(n)$ være antallet af primtal i intervallet $2..n+1$.
Der gælder[7]

$$\pi(n) \in \frac{n}{ln(n)} + O(\frac{n}{ln(n)^2})$$

Vi kan nu finde algoritmens udførelsestid som summen af udførelsestiderne for

a) initialiseringen A

b) løkken B

c) løkken C

d) resten af algoritmen

Vi påstår, at disse er som følger (p_i betegner det i'te primtal)

a) $\Theta(n)$

b) $\Theta(n)$

c) $\sum_{i=1}^{\pi(n)} \Theta(\frac{n}{p_i})$

d) $\Theta(\frac{n}{\log(n)})$

Med hensyn til a) og b) skulle analysen være oplagt. Med hensyn til c) følger resultatet af, at hver gang der er fundet et nyt primtal p, gennemløbes K i "spring af længde p". Endelig følger d) af primtalssætningen og det faktum,

[7] $ln(x)$ er den naturlige logaritme.

at den *amortiserede* kompleksitet af proceduren Rpush i Box'en Sequence tilhører $O(1)$ (jfr. 9.3).

Det er klart, at det dominerende led blandt a)-d) er c), som vi kan omskrive til

$$\Theta(n \sum_{i=1}^{\pi(n)} \frac{1}{p_i})$$

Med hensyn til værdien af denne sum, skal vi igen appellere til et resultat fra talteorien, som siger

$$\sum_{i=1}^{\pi(n)} \frac{1}{p_i} \approx ln(ln(n))$$

Vores analyse giver altså følgende samlede resultat

$$T[\![\text{Erathostenes Si}]\!](n) \in \Theta(n \log(\log(n)))$$

11.4 TRINE programmet

Lad os slutte med at præsentere et fuldt færdigt TRINE program, der finder primtal v.hj.a. Erathostenes Si. Her har vi, for at øge læseligheden og modificerbarheden, realiseret variablen K som en box, hvor hjælpevariablerne B, a, p osv. er skjult, og hvor det blot er de nødvendige procedurer, der stilles til rådighed. Boxen skal levere følgende fire procedurer

Init$[K](n)$
Size$[K]$
Min$[K]$
Delete$[K](k)$

Selve mængden K realiseres som et produkt af type

$$\textbf{Type } \text{Set} = \textbf{Prod}(B : \textbf{List } (\text{Bool}), a, m : \text{ Int})$$

hvor B angiver bitvektoren, a angiver antallet, men m nu angiver det *mindste* element i den mængde K repræsenterer. Dette er en lille forskel fra før,

hvor p angav det sidst fundne primtal. Denne ændring er en konsekvens af at realisere K som box, fordi boxen skal have selvstændig mening uafhængigt af, hvad den bliver brugt til.

Denne selvstændige mening udtrykker vi v.hj.a. en *box-invariant* (jfr. afsnit 10.2 i [TRINE]) på følgende måde

$$\begin{aligned} I(B,a,m) \quad : \quad & (a = |\{i \in 0..|B| \mid B.(i) = \underline{\mathrm{t}}\}|) \wedge (0 \leq m \leq |B|) \wedge \\ & (B(0..m) = \underline{\mathrm{f}}) \wedge ((B.(m) = \underline{\mathrm{t}}) \vee (m = |B|)) \end{aligned}$$

Vi skal så sørge for, at boxens initialiseringsprocedure etablerer I, og at de øvrige procedurer holder I invariant.

Vi ønsker imidlertid også at angive specifikationer for Init, Size, Min og Delete, og her får vi igen brug for en abstraktionsfunktion (jfr. side 12 og 41). Dette skyldes, at medens box-invarianten udtaler sig om de konkrete variabler (B, a og m), så skal specifikationerne udtale sig om den tilsvarende abstrakte variabel, som jo stadig skal angive en *mængde*. Det er egenskaber ved denne, vi ønsker at se udefra, og hele pointen er jo netop at holde egenskaberne ved B, a og m skjult. Korrespondancen mellem abstrakte og konkrete variabler udtrykkes ved følgende abstraktionsfunktion α, som afbilder variabler af type Set ind i mængder:

$$\alpha(K) = \{k \in 0..|K.B| \mid K.B.(k) = \underline{\mathrm{t}}\}$$

Ved hjælp af α kan vi nu angive følgende specifikationer af boxens operationer

$$\begin{aligned} &\mathrm{Init}[K](n)\ \mathbf{sat}\ n \geq 2 \rightarrow \alpha(K') = \{2,\ldots,n\} \\ &\mathrm{Size}[K]\ \mathbf{sat}\ (\mathrm{Size}' = |\alpha(K)|) \wedge (\alpha(K) = \alpha(K')) \\ &\mathrm{Delete}[K](p)\ \mathbf{sat}\ \alpha(K') = \alpha(K) \backslash \{p\} \\ &\mathrm{Min}[K]\ \mathbf{sat}\ \alpha(K) \neq \emptyset \rightarrow (\mathrm{Min}' = \min(\alpha(K))) \wedge (\alpha(K') = \alpha(K)) \end{aligned}$$

Det færdige TRINE program ser nu ud som følger.

```
Process Erathostenes
 (+ @"sequence.tri"

    Box PS
      Type Blist = List(Bool)
      Type Set = Prod(B: Blist, a, m: Int)

      { I(B, a, m) }

      Proc Init[K: Set](n: Int)
        K := Prod(Blist(false|2)++Blist(true|n-1), n-1, 2)
      end Init

      Proc Size[K: Set] → (Int)
        return K.a
      end Size

      Proc Min[K: Set] → (Int)
        return K.m
      end Min

      Proc Delete[K: Set](k: Int)
        if 0 ≤ k < |K.B| →
              if K.B.(k) → K.B.(k), K.a := false, K.a-1 fi
              do (K.m < |K.B|) ∧ ( ¬ K.B.(K.m)) → K.m := K.m+1 od
        fi
      end Delete
    end PS

    Box ISq
      Sequence(Int)
    end ISq

    Var P: ISq'Seq
    Var K: PS'Set
    Var n: Int
```

```
    write("n=")
    read[n]
    do  n < 2  →
          write("n=")
          read[n]
    od
    ISq'Init[P]
    PS'Init[K](n)
    do  { Erathostenes-invariant }
       PS'Size[K] > 0  →
          (+ Var  p, i: Int
             p := PS'Min[K]
             ISq'Rpush[P](p)
             i := p
             do  i ≤ n  →
                   PS'Delete[K](i)
                   i := i+p
             od
          +)
    od
    write("Primtal: ")
    ISq'Print[P]
  +)
end  Erathostenes
```

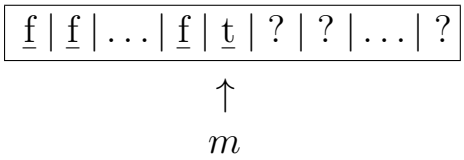
11.5 Kompleksiteten af TRINE programmet

Efter at have introduceret datastrukturen PS skal vi naturligvis argumentere for, at analysen fra afsnit 11.3 stadig holder.

Det er nemt at se, at dette er tilfældet såfremt såvel Rpush som Delete har konstante *amortiserede* udførelsestider.

Dette ved vi allerede for Rpush, medens vi for Delete er nødt til at finde en nyttig potentialfunktion, der kan knyttes til PS.

Det følger af box invarianten, at alle værdier i $B(0..m)$ er lig med $\underline{f}$, medens $B.(m) = \underline{t}$ (med mindre $m = |B|$). B har altså udseendet

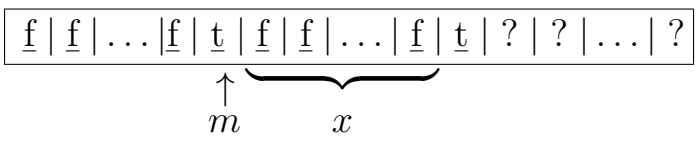
B: 

Det er klart, at udførelsestiden for Delete er konstant, men mindre det er elementet m, der fjernes; i så fald er den proportional med længden af den længste ubrudte sekvens af f'er, der står umiddelbart til højre for m. Inspireret heraf definerer vi følgende potentialfunktion

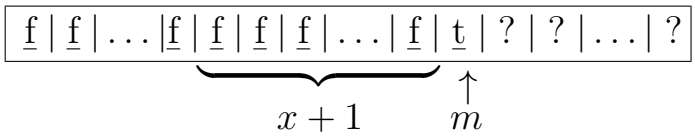
$$\Phi(B, m) = \alpha * (\text{ antal f'er i } B(m..|B|))$$

hvor α er den "sædvanlige" proportionalitetsfaktor.

Betragt nu kaldet Delete$[K](m)$ og antag, at B har udseendet

B: 

Efter kaldet vil vi da have

B': 

hvoraf det følger, at

$$T[\![\text{Delete}[K](m)]\!] \leq \alpha(x+1)$$

Samtidig gælder der

$$\Delta\Phi = -\alpha x$$

hvorfor

$$\begin{aligned} T_A[\![\text{Delete}[K](m)]\!] &= T[\![\text{Delete}[K](m)]\!] + \Delta\Phi \\ &\leq \alpha(x+1) - \alpha x = \alpha \\ &\in O(1) \end{aligned}$$

Alt i alt har Delete altså konstant amortiseret udførelsestid.

12 Referencer

[**TRINE**] E.M. Schmidt & M.I. Schwartzbach: Programmering og programmeringssproget Trine. DAIMI FN-36, Datalogisk Afdeling, Aarhus Universitet, august 1994.

[**Bentley**] J. Bentley: Programming Pearls. Addison-Wesley Publishing Company, 1986.

[**Transitionssystemer**] M. Nielsen, E.M. Schmidt, M.I. Schwartzbach: Introduktion til transitions systemer. Datalogisk Afdeling, Aarhus Universitet, februar 1995.

Algoritmer

Binær søgning, 14
Erathostenes si, 106, 108
Euklid, 8
Fletning, 15
Geometrisk søgning, 100
Indsættelsessortering, 20, 21
Lineær potensopløftning, 26
Lineær søgning, 13
Logaritmisk potensopløftning, 11
Opdel, 22
Optælling, 30
Permutation, 38, 97
Potensopløftning, 10
Søgning, 12
Udvalgssortering, 21

Programmer

Binær$_1$, 33
Binær$_2$, 33
Box B, 62
Box Counter, 87
Box Eq, 84
Box H, 64
Box PolyPri, 55
Box Pri, 53
Box PS, 112
Box Q, 44
Box S, 42, 45
Box Search, 69
Box SS, 91
BTreesort, 59
Erathostenes, 112
Fletning, 16
Postfix, 46
Reverse$_1$, 35
Reverse$_2$, 35
Snit, 18
Treesort, 57

Stikordsregister

abstraktionsfunktion, 12, 41
amortiseret omkostning, 89
amortiseret udførselstid, 87
aritmetiske gennemsnit, 100
asymptotisk begrænset, 24
AVL-træ, 77

balanceret binært træ, 48
BB[α]-træ, 78
besværlige knude, 72
bevisprincip, 40, 99, 103
billedet, 4
binær kilometertæller, 30
binær søgning, 29
bitvektor, 61
blandet udsagn, 39
bogførings-invariant, 31
boolske funktion, 4
box-invariant, 111
bunden variabel, 5
bunke, 48

cyklisk liste, 43

datastruktur, 41
datatype, 41
dobbeltrotation, 79
dublet, 17

eksistentiel, 7
Erathostenes si, 105, 110

Fibonacci-tal, 77
fletning, 15
fri variabel, 5
funktion, 3
følge, 3

gennemsnitlig tidskompleksitet, 28
geometriske gennemsnit, 100
gyldig algoritme, 9

hash-funktion, 63
hash-tabel, 63
head, 3
heltals-sekvens, 107
højdebalancering, 77

indsættelsessortering, 29
interval, 3
invariant, 9, 40
inverst træ, 82

kandidatmængden, 11
kanonisk repræsentant, 81
karakteristisk funktion, 2
karakteristisk parameter, 28, 104
klasse, 81
klassedeling, 81
KochLine, 36
konkatenation, 3
konnektiv, 4
korrekt algoritme, 9
kvantificering, 7
kvantor, 4
kø, 43

lineær potensopløftning, 27
$\log^*$, 86
logaritmisk potensopløftning, 29
logiske regneregler, 6

metode, 8
monoton mængde, 108
multimængde, 47

målgruppe, 11
mærket variabelnavn, 38

O, 24
ombytning, 20
Ω, 24
omflytningsalgoritme, 20
omvendt polsk notation, 45
opfylde, 9, 38
Optælling, 32
ordbog, 61

perfekt balanceret, 51
polymorf prioritetskø , 54
postfix udtryk, 45
potentialfunktion, 88
potentiel energi, 88
primitiv operation, 23
primtal, 105
Primtalssætningen, 109
prioritetskø, 47
proceduredefinition, 98
prædikat, 4

rekursionsligning, 35
rekursiv procedure, 102
respons, 8
responsprædikat, 9
Reverse_1, 103
rotation, 74, 79
rødt-sort træ, 70

sat, 38
sorteret vektor, 20
sorteringsalgoritme, 20
specifikation, 38, 98
stak, 41
stimulans, 8
stimulansprædikat, 9
størrelsesorden, 24
søgedomæne, 11
søgetræ, 66
søgning, 11

tail, 3
termineringsfunktion, 9, 40
Θ, 24
tidskompleksitet, 23, 26
tilstand, 9
transitionssystem, 72
træ, 48
træsortering, 57

udsagn, 9
umærket variabelnavn, 38
universel, 7
urbillede, 4
uægte knude, 72

variabelnavn, 38
vejforkortning, 85
vægtbalanceret træ, 78

worst-case kompleksitet, 28

ækvivalensrelation, 81