Institut for Matematik og Datalogi
Syddansk Universitet, Odense

5. maj 2020

DM507 — Algoritmer og Datastrukturer

Eksaminatorie-timer uge 18, Forår 2020

Instruktorerne for DM507

Indhold

1 Eksaminatorie-timer I **2**
1.1 Opgave 1 - Eksamen januar 2007, opgave 2 . . . 2
1.2 Opgave 2 - Eksamen juni 2012, opgave 5 . . . 4
1.3 Opgave 3 - Cormen et al. opgave 16-1 . . . 8
1.4 Opgave 4 - Eksamen juni 2011, opgave 3 . . . 13
1.5 Opgave 5 - Eksamen januar 2007, opgave 1 . . . 15

2 Eksaminatorie-timer II **17**
2.1 Opgave 1 - Cormen et al. opgave 22.1-1 . . . 17
2.2 Opgave 2 - Cormen et al. opgave 22.1-3 . . . 18
2.3 Opgave 3 - Cormen et al. 22.1-6 . . . 18
2.4 Opgave 4 - Cormen et al. opgave 22.2-1 . . . 21
2.5 Opgave 5 - Cormen et al. opgave 22.2-2 . . . 21
2.6 Opgave 6 - Cormen et al. opgave 22.2-3 . . . 24
2.7 Opgave 7 - Eksamen juni 2009, opgave 1a . . . 25
2.8 Opgave 8 - Eksamen januar 2005, opgave 5 . . . 26
2.9 Opgave 9 - Eksamen juni 2010, opgave 5 . . . 26

1 Eksaminatorie-timer I

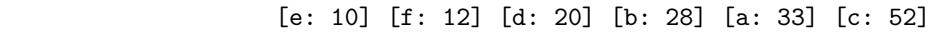
1.1 Opgave 1 - Eksamen januar 2007, opgave 2

Spørgsmål a): Betragt alfabetet med seks tegn a, b, c, d, e, f. Nedenstående tabel viser, hvor tit hvert enkelt tegn optræder i en given tekst

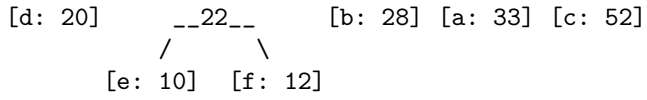
a	b	c	d	e	f
33	28	52	20	10	12

Tegn Huffman-træet, som repræsenterer Huffman-koderne for dette eksempel. Dette gøres ved at køre Huffmans algoritme:

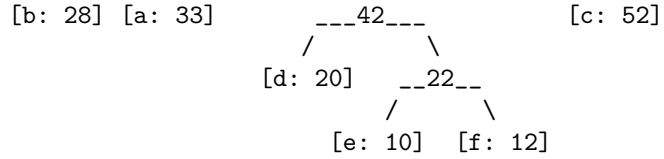
Gennemløb 0: Alle træer består af et tegn med tegnets frekvens.



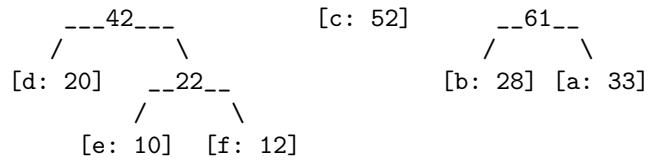
Gennemløb 1: Træerne med frekvenserne 10 og 12 slås sammen til et træ med frekvens 22.



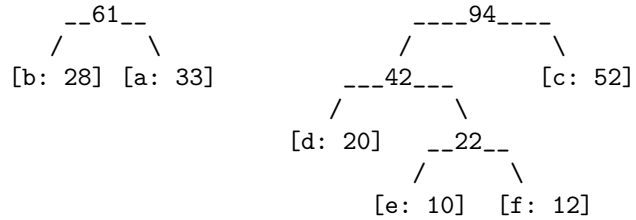
Gennemløb 2: Træerne med frekvenserne 20 og 22 slås sammen til et træ med frekvens 42.



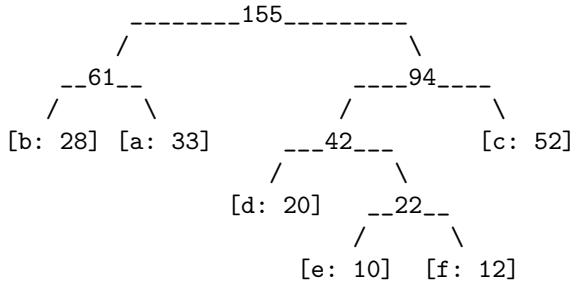
Gennemløb 3: Træerne med frekvenserne 28 og 33 slås sammen til et træ med frekvens 61.



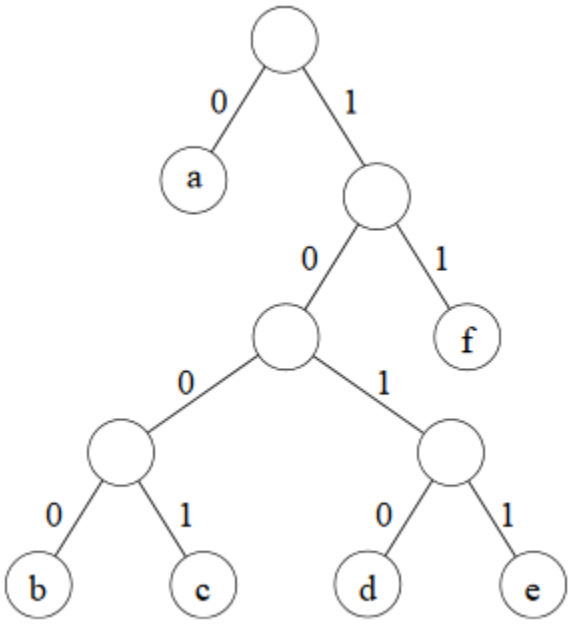
Gennemløb 4: Træerne med frekvenserne 42 og 52 slås sammen til et træ med frekvens 94.



Gennemløb 5: Træerne med frekvenserne 61 og 94 slås sammen til et træ med frekvens 155, hvilket er Huffman-træet, der repræsenterer Huffman-koderne for eksemplet.



Spørgsmål b) Brug følgende Huffman-træ

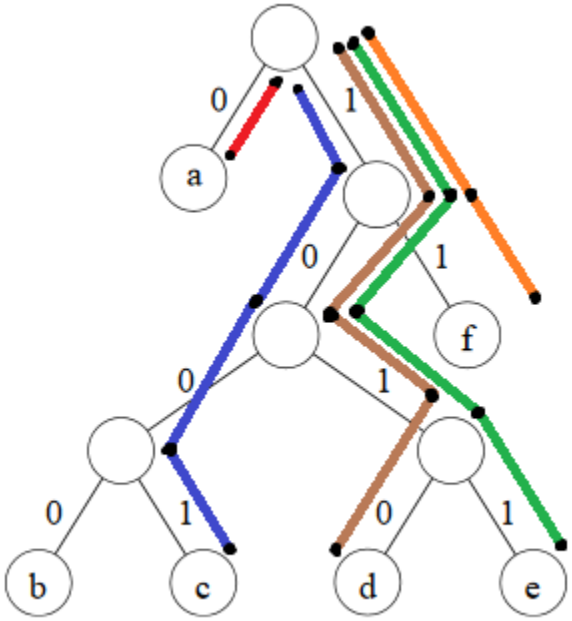


Figur 1

til at dekode $c = 1101001010101011$. Dette gøres ved at gennemløbe c samtidigt med man følger en rod-blad sti i Huffman-træet. Rod-blad stien er bestemt af hvilke tegn man møder i c under gennemløbet - ved 0 fortsætter man i venstre undertræ og ved 1 fortsætter man i højre undertræ. Når man når et blad, så har man dekodet et tegn (tegnet i bladet), og man genstarter fra roden før man fortsætter med at gennemløbe c. Den endelige dekodede besked er

$$\underbrace{11}_{\mathtt{f}}\ \underbrace{0}_{\mathtt{a}}\ \underbrace{1001}_{\mathtt{c}}\ \underbrace{0}_{\mathtt{a}}\ \underbrace{1010}_{\mathtt{d}}\ \underbrace{1011}_{\mathtt{e}}$$

og de brugte rod-blad stier er markeret i Figur 2.



Figur 2

1.2 Opgave 2 - Eksamen juni 2012, opgave 5

I denne opgave er vi givet en fil, som indeholder nedenstående tegn og tilhørende hyppigheder:

Tegn	a	b	c	d	e
Hyppighed	100	150	150	250	350

Tabel 1: De givne tegn i filen og deres tilhørende hyppigheder

Et muligt Huffman-træet for denne fil er som følger:

```
Huffman-træ H1:
          _______1000_______
        0/                  \1
     __400__               __600__
   0/       \1           0/       \1
[b: 150]  [d: 250] __250__      [e: 350]
                 0/       \1
              [a: 100]  [c: 150]
```

Når ovenstående træ anvendes i forbindelse med *afkodning* (decoding) eller *kodning* (encoding) af en fil, så læser vi 0 som et venstre skridt og 1 et som et højre skridt på en sti i træet fra roden til et blad.

Delopgave a Vi angiver hvor mange bits en fil fylder når den er kodet vha. Huffman-træet `H1`. Dette gøres ved (i) at tælle hvor mange bits der skal bruges til at kode et bestemt tegn og (ii) herefter tage højde for hvor mange af hvert tegn vi skal repræsentere, dvs. hyppigheden af tegnet i filen. Mere præcist observerer vi:

- Karakteren `a` kodes som 100, dvs. 3 bits og vi skal repræsentere 100 af disse karakterer
- Karakteren `b` kodes som 00, dvs. 2 bits og vi skal repræsentere 150 af disse karakterer
- Karakteren `c` kodes som 101, dvs. 3 bits og vi skal repræsentere 150 af disse karakterer
- Karakteren `d` kodes som 01, dvs. 2 bits og vi skal repræsentere 250 af disse karakterer
- Karakteren `e` kodes som 11, dvs. 2 bits og vi skal repræsentere 350 af disse karakterer

Dette resulterer endeligt i at filen vil fylde:

$$3 \cdot 100 + 2 \cdot 150 + 3 \cdot 150 + 2 \cdot 250 + 2 \cdot 350 = 2250 \text{ bits} \tag{1}$$

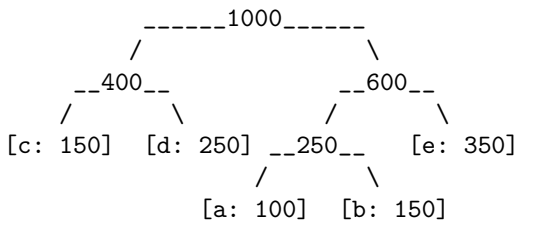
Delopgave b I denne delopgave anvender vi Huffman-træet `H1` til at afkode den givne sekvens: 1000000110110101. Dette gøres ved at aflæse sekvensen fra venstre mod højre og følge en sti i træet `H1` fra roden af indtil vi når til et blad med et korresponderende tegn. Vi genstarter så ved roden igen. Ved at gøre dette har vi følgende forløb ved afkodning:

$$\underbrace{100}_{a}\underbrace{00}_{b}\underbrace{00}_{b}\underbrace{11}_{e}\underbrace{01}_{d}\underbrace{101}_{c}\underbrace{01}_{d} \tag{2}$$

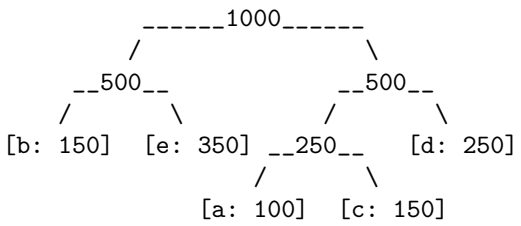
Den endelige afkodet streng er altså: abbedcd

Delopgave c Givet optimale træer `H2` til `H5`, for den givne fil, skal vi angive hvilke af disse træer som kan fremkomme ved anvendelse af Huffman's algoritme.

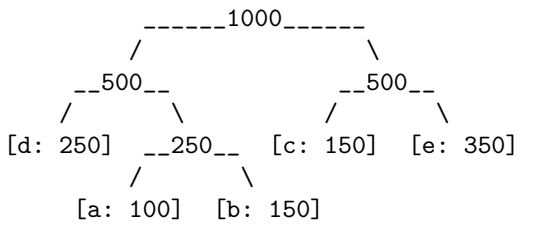
H2:



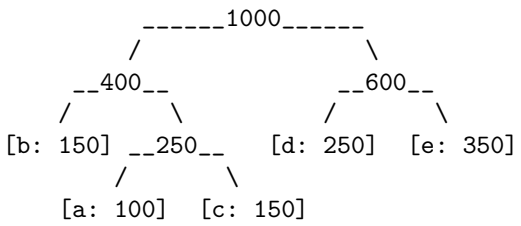
H3:



H4:



H5:

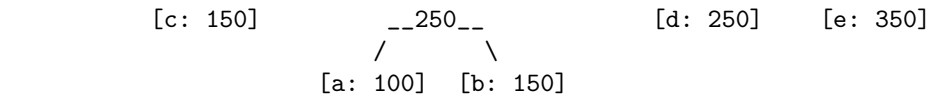


Givet tegnene og tilhørende hyppigheder i Tabel 1 konstruerer vi mulige Huffman-træer og ser hvilke af de 4 givne træer som fremkommer.

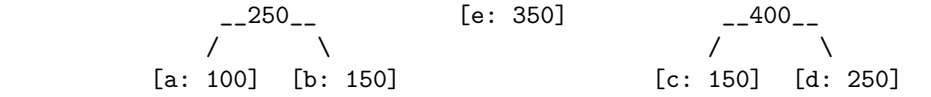
Vi konstruerer første mulige Huffman-træ ved følgende skridt:

1. Skridt: Start med alle tegn som værende blade i Huffman-træet

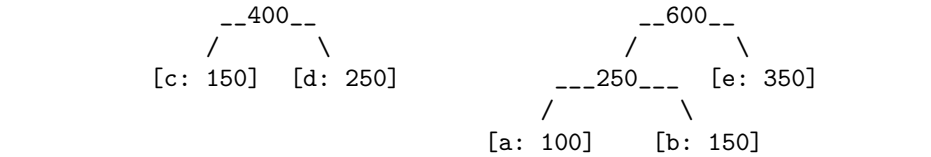
2. Skridt: Sammenlæg blade `[a: 100]` og `[b: 150]` til et nyt træ med rod `250`



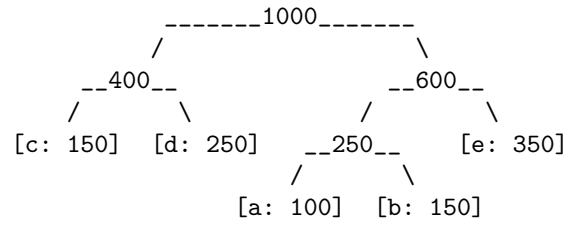
3. Skridt: Sammenlæg blade `[c: 150]` og `[d: 250]` til et nyt træ med rod `400`



4. Skridt: Sammenlæg træet med rod `250` med blad `[e: 350]`, så vi får et træ med rod `600`



5. Skridt: Sammenlæg træerne med rod `400` og `600`, så vi får et træ med rod `1000`



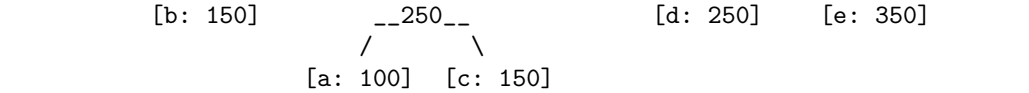
Vi ser at dette træ er identisk med træet `H2`.

Det kan bemærkes at i **3. Skridt** ville det også have været muligt at sammenlægge bladet `[c: 150]` med træet med rod `250`, således at et alternativt Huffman-træ vil fremkomme. I dette tilfælde kan det ses at ingen af træerne `H3` til `H5` vil fremkomme.

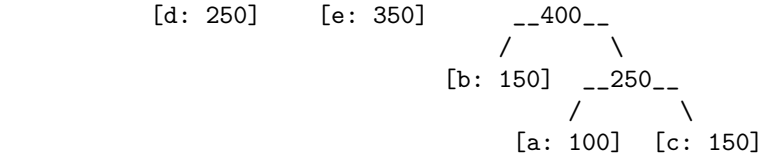
I **2. Skridt**, ovenfor, ville det også have været muligt at sammenlægge blade `[a: 100]` og `[c: 150]` til et nyt træ med rod `100 + 150 = 250`. Hvis vi gør dette i stedet, så får vi konstrueret et andet muligt Huffman-træ som følger:

1. Skridt: Start med alle tegn som værende blade

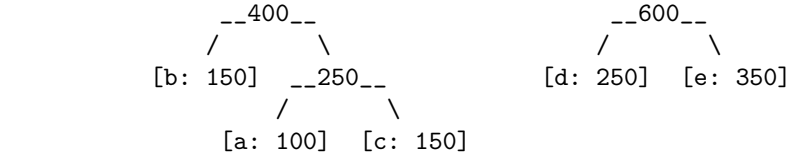
2. Skridt: Sammenlæg blade `[a: 100]` og `[c: 150]` til et nyt træ med rod `250`



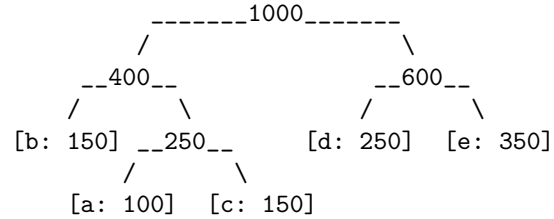
3. Skridt: Sammenlæg træet med rod `250` med blad `[b: 150]`, så vi får et træ med rod `400`



4. Skridt: Sammenlæg blade `[d: 250]` og `[e: 350]` til et nyt træ med rod `600`



5. Skridt: Sammenlæg træerne med rod `400` og `600`, så vi får et træ med rod `1000`



Vi ser at dette træ er identisk med træet `H5`.

Det kan igen bemærkes, at man i **3. Skridt** kan tage det alternative valg og sammenlægge blade `[b: 150]` og `[d: 250]`. I dette tilfælde vil et alternativt Huffman-træ fremkomme, som ikke vil være identisk med nogen af træerne `H2` til `H5` (det alternative Huffman-træ vil dog være identisk med træet `H1`, som er givet i opgavebeskrivelsen).

Til sidst kan det nævnes, at andre mulige Huffman-træer kan konstrueres ved at bytte rundt på højre og venstre barn i en eller flere knuder i de forrigt beskrevne Huffman-træer. Hvis vi gør dette ser vi, at vi ikke vil kunne få træerne `H3` eller `H4`. Vi kan endeligt konkludere at træerne `H3` og `H4` ikke kan fremkomme ved anvendelse af Huffman's algoritme. Det er derimod muligt at træerne `H2` og `H5` kan fremkomme.

1.3 Opgave 3 - Cormen et al. opgave 16-1

a)

Vi bruger følgende grådige algoritme, som altid tager den største mønt som har værdi højst det beløb der er tilbage, indtil vi har givet hele beløbet tilbage:

```
1 GreedyAlg(n):
2     rest = n
3     while rest > 0:
4         use largest coin m with value at most rest
5         rest = rest - m
```

Vi skal nu bevise, at denne algoritme er optimal hvis mønterne vi har til rådighed er $\langle 1, 5, 10, 25 \rangle$.

Vi beviser først følgende lemma omkring møntsættet vi ser på, som vi bruger i et senere bevis:

Lemma 1.1 *Enhver optimal løsning må indeholde en største mønt, m, som er højst det beløb n vi skal opdele for (dvs. $m \leq n$).*

Vi beviser nu dette lemma ved at bevise det for alle muligheder for n:

$\underline{1 \leq n < 5}$

Her er den største mønt vi kan bruge 1, og det er klart at 1 må være en del af løsningen, da vi ikke kan give tilbage på andre måder.

$\underline{5 \leq n < 10}$

Vi har 1'ere og 5'ere til rådighed. Antag at den største mønt, en 5'er, ikke er en del af en optimal løsning. Så må der være mindst 5 1'ere i denne optimale løsning, men så kan vi bytte 5 1'ere ud med en 5'er og få en bedre løsning, hvilket er i modstrid med at løsningen er optimal. Dvs. en optimal løsning må indeholde en 5'er.

$\underline{10 \leq n < 25}$

Vi har 1'ere, 5'ere og 10'ere til rådighed. Antag at der ikke er en 10'er med i en optimal løsning. Stil mønterne i denne optimale løsning op i faldende sorteret orden. Se på starten af denne opstilling. Vi ser nu på 3 muligheder for, hvordan de første mønter som giver 10 i værdi til sammen ser ud i denne opstilling.

- Der er 0 5'ere blandt disse mønter. Så må der være 10 1'ere, som kan udskiftes med en enkelt 10'er, og vi får en bedre løsning, hvilket er i modstrid med, at løsningen er optimal.
- Der er 1 5'er blandt disse mønter. Så må de resterende mønter blandt disse mønter være 5 1'ere. Alle mønterne i starten kan udskiftes med en enkelt 10'er, og vi får en bedre løsning, hvilket er i modstrid med, at løsningen er optimal.
- Der er 2 5'ere blandt disse mønter. Disse 2 5'ere kan udskiftes med en enkelt 10'er, og vi får en bedre løsning, hvilket er i modstrid med, at løsningen er optimal.

Der er ikke andre muligheder end de 3 muligheder, og i hver mulighed får vi en modstrid, dvs. den største mønt, en 10'er, må være en del af en optimal løsning.

$\underline{n \geq 25}$

Vi har alle mønter til rådighed. Antag at den største mønt, en 25'er, ikke er en del af en optimal løsning.

Hvis der er mindst 3 10'ere med i denne optimale løsning, så kan vi udskifte de 3 10'ere med en 25'er og en 5'er og få en bedre løsning, hvilket er i modstrid med, at løsningen er optimal.

Hvis der er højst 2 10'ere med i denne optimale løsning, må resten af mønterne i denne optimale løsning være 5'ere eller 1'ere. Opstil mønterne i denne optimale løsning i faldende sorteret orden. Hvis vi ser på mønterne fra starten af denne opstilling, kan vi ikke undgå at ramme præcist 25 når vi lægger værdien af mønterne fra starten sammen (hvis man ikke er overbevist, prøv at stille mulighederne op). Alle disse mønter kan vi udskifte med en 25'er og få en bedre løsning, hvilket er i modstrid med, at løsningen er optimal.

En optimal løsning må altså indeholde mindst én 25'er, og dermed den største mønt til rådighed.

Vi har nu argumenteret for, at enhver optimal løsning må indeholde den største mønt til rådighed, dvs. vi har bevist **Lemma 1.1**.

Dette lemma kan bruges til at bevise følgende løkke-invariant om løkken i vores grådige algoritme:

"Den møntmængde V som er valgt indtil videre er en delmængde af en optimal møntmængde OPT for hele beløbet n (dvs. $V \subseteq$ OPT)"

Vi beviser nu denne invariant som sædvanlig:

Initialization Inden første gang løkken kører er der ikke valgt nogen mønter, og V er dermed tom, dvs. $V = \emptyset \subseteq$ OPT, og invarianten er sand.

Maintenance Lad den valgte møntmængde inden løkke-iterationen være V og den valgte møntmængde efter løkke-iterationen være V'. Per invarianten ved vi, at $V \subseteq OPT$. Hvis vi ser på de mønter som der er i OPT og endnu ikke i V (dvs. OPT $\setminus V$), så må summen af disse mønter være præcist det beløb vi har tilbage og mangler at vælge mønter for i vores algoritme. Disse mønter må være et optimalt valg for det beløb vi har tilbage, for ellers ville vi kunne gøre OPT bedre ved at udskifte et ikke-optimalt valg af mønterne for det resterende beløb med et optimalt valg. Per **Lemma 1.1**, vil et optimalt valg af mønter for et beløb altid indeholde den største mønt der højst er dette beløb, dvs. sådan en mønt må være blandt

disse mønter som er i OPT og ikke i V. Algoritmen udvider i løkke-iterationen V til V' ved at tilføje denne største mønt til V. Vi har lige argumenteret for at denne største mønt også må være i OPT (blandt de mønter som er i OPT ikke i V), dvs. det gælder nu at $V' \subseteq \text{OPT}$, og invarianten er dermed også sand før næste løkke-iteration.

Termination Løkken slutter når det resterende beløb er 0 (og mønterne i V summer dermed til n, som vi er sikre på altid kan lade sig gøre, da mønten med værdi 1 er et muligt møntvalg). Da vi per vores invariant ved at $V \subseteq \text{OPT}$, og vi lige har argumenteret for at V summer til n og dermed er en korrekt løsning, vil denne løsning også være optimal (og faktisk vil $V = \text{OPT}$, da vores løsning ikke kan være bedre end OPT).

Vi har dermed bevist, at den grådige algoritme finder frem til en optimal løsning.

b)

Vi skal vise, at den grådige algoritme er optimal når møntsættet har den egenskab, at $m_1 = 1$ og for $i \geq 1$, så går m_i op i m_{i+1}. Der er k mønter i alt.

Her skal vi huske, at "går op i" er transitivt, dvs. hvis a går op i b og b går op i c, så går a også op i c. For vores problem betyder dette, at enhver mønt går op i alle mønter der er større end den selv. Dvs. alle mønter kan altså skrives som et multiplum af enhver mønt der er mindre end den selv.

Eksempel, hvor dette gælder er møntsættet: $\langle 1, 3, 9, 18 \rangle$.

Eksempel, hvor dette ikke gælder er møntsættet: $\langle 1, 5, 10, 25 \rangle$.

Her går 10 ikke op i 25. Men vi har lige i opgave a) bevist, at den grådige algoritme virker for dette møntsæt. Men dette er også i orden: det eneste vi skal vise er, at hvis et møntsæt *har* egenskaben, så virker den grådige algoritme. Vi siger ikke noget om hvorvidt den grådige algoritme ikke virker, hvis et møntsæt *ikke* har egenskaben.

For at bevise at den grådige algoritme er optimal, beviser vi først følgende lemma, som gælder for alle møntsæt som har den beskrevne egenskab:

Lemma 1.2 *Enhver optimal løsning må indeholde en største mønt, m, som er højst det beløb n vi skal opdele for (dvs. $m \leq n$).*

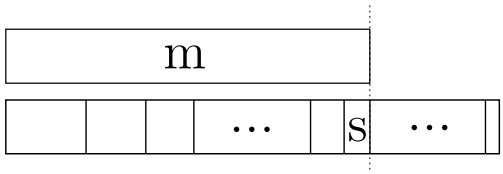
Bevis

Antag til modstrid, at et optimalt valg af mønter ikke indeholder denne største mønt m.

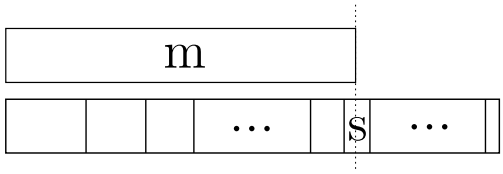
Vi ser nu på dette optimale valg af mønter. Vi sorterer mønterne i faldende orden (så den største står først). Vi ser nu på, hvordan den største mønt m ser ud i forhold til starten af vores opstilling af mønter. Vi repræsenterer mønterne som firkanter, hvor bredden af en firkant indikerer størrelsen af mønten.

Der er 2 tænkelige scenarier (vi viser senere, at scenarie nummer 2 ikke kan ske).

Scenarie 1: mønterne i opstillingen rammer præcist værdien m, og den sidste mønt blandt de første har værdi s:

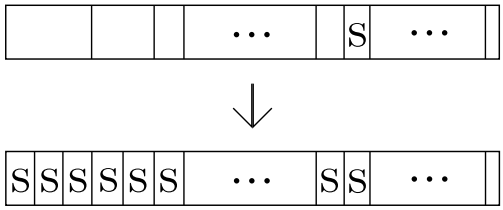


Scenarie 2: mønterne i opstillinger rammer *ikke* præcist værdien m, og den sidste mønt blandt de første har værdi s:



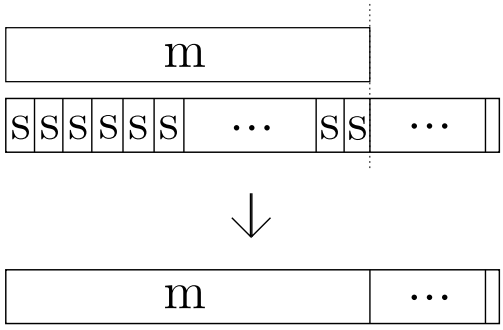
I området i begge tilfælde må der være mindst 2 mønter, da vi har antaget at den største mønt m ikke er blandt dem.

Da s er mindre end eller lig med alle mønterne før den, må den gå op i alle sammen, og hver af mønterne kan dermed skrives som et multiplum af s. Dvs. vi kan udskifte hver af disse mønter med et antal mønter af størrelse s:



Men det betyder at starten af vores opstilling udelukkende består af et antal mønter af størrelse s. Da s også går op i m, så må der præcist være x mønter i denne opstilling, således at $x \cdot s = m$. Dvs. scenarie 2 kan altså ikke ske.

Vi kan nu bytte disse x mønter af størrelse s ud med m:



Der var mindst 2 mønter i området i den optimale løsning. Efter udskiftningen af mønterne til mønter med størrelse s og efterfølgende udskiftning af mønterne med størrelse s til mønten med størrelse m, er der nu kun 1 mønt i området. Den optimale løsning har altså fået reduceret antallet af mønter med mindst 1. Dette er en modstrid, da vi ikke kan gøre en optimal løsning bedre. Vores antagelse, som har ført til modstriden er at en optimal løsning ikke indeholder denne største mønt m. Det må derfor være tilfældet, at en optimal løsning altid indeholder denne største mønt m. Vi har dermed bevist **Lemma 1.2**.

I slutningen af opgave a) beviste vi, at den grådige algoritme er optimal for møntsættet i opgave a). I det bevis gjorde vi kun brug af **Lemma 1.1** omkring møntsættet i opgave a). Vi har nu vist et tilsvarende lemma omkring møntsættet vi ser på nu, dvs. vi kan blot udskifte referencen til **Lemma 1.1** i beviset til **Lemma 1.2**, og vi har dermed bevist, at den grådige algoritme er optimal for det møntsæt vi ser på nu.

c)

Lad vores møntsæt være $\langle 1, 3, 4\rangle$, og beløbet vi gerne vil give mønter tilbage for 6.

Vores grådige algoritme vil vælge mønterne $\langle 4, 1, 1\rangle$.

Men den optimale løsning er $\langle 3, 3\rangle$.

Vores grådige algoritme er dermed ikke optimal for dette møntsæt (den vælger 3 mønter, og den optimale løsning indeholder kun 2 mønter).

d)

Vi er givet et beløb n som vi skal give mønter tilbage for, og et møntsæt $\langle m_1, m_2, ..., m_k\rangle$, som altid indeholder en mønt med værdi 1.

Vi laver nu en tabel, R, af længde $n + 1$, hvor $R[i]$ indeholder antal mønter i en optimal løsning for beløbet i, hvor $0 \leq i \leq n$.

Vi kan nu beskrive den rekursive formel:

$$R[i] = \begin{cases} 0 & i = 0 \\ \min\limits_{m \in M_i} \{1 + R[i - m]\} & i > 0 \end{cases}$$

Hvor M_i er mængden af alle de mønter, hvor møntværdien højst er i.

Med andre ord siger vores rekursive formel, at hvis beløbet er 0, så skal der klart bruges 0 mønter for at dække beløbet. Ellers så prøver vi alle mønter som er højst det beløb vi skal dække, og ser hvilken mønt som gør at vi bruger færrest mulige mønter for det beløb vi har tilbage (dette antal mønter findes i $R[i - m]$).

Denne rekursive formel er korrekt, da enhver optimal løsning må bestå af en "sidste mønt", samt en optimal opdeling af det resterende beløb (hvis den optimale løsning for hele beløbet ikke indeholdt en optimal løsning for det resterende beløb, kunne vi bytte den ikke-optimale løsning for det resterende beløb ud med en optimal løsning og dermed få en bedre løsning for hele beløbet, hvilket er en modstrid.) I vores rekursive formel tjekker vi alle mulige "sidste valg" af mønter, og vælger den mønt der giver os det mindste resultat, og vi finder dermed den optimale løsning.

Da vi nu har den rekursive formel, kan vi udfylde tabellen. Vi skal blot starte ved $i = 0$, og gå op til $i = n$. Hver gang vi udfylder et felt skal vi se på højst k andre felter (vi skal højst tjekke alle mønterne i møntsættet), og når vi udfylder et felt skal kun se på tidligere felter som allerede er blevet udfyldt. Da vi skal udfylde n felter og hver udfyldning højst tager k tid, bliver køretiden $O(nk)$. Pladsforbruget bliver $\Theta(n)$ da vi skal opbevare tabellen.

For at finde det antal mønter der skal bruges i en optimal løsning skal vi blot se på værdien i $R[n]$ efter vi har udfyldt tabellen.

For at finde en konkret løsning (dvs. hvilke mønter vi skal vælge), kan vi bagefter udfyldningen starte ved $R[n]$ og "gå baglæns", mens vi holder styr på hvor stort et beløb vi har givet penge tilbage for indtil videre, og tjekke alle mulige mønter for at finde ud af, hvad den optimale mønt at give tilbage var. F.eks. ved $R[n]$ har vi beløbet n vi skal give tilbage for. Vi tjekker derfor alle mønter med værdi $m \leq n$, og vælger at give den mønt tilbage, hvor $R[n-m]$ er mindst. Efter m er fundet gentages denne procedure, hvor vi nu er ved $R[n-m]$, og har resterende beløb $n-m$ tilbage. Dette fortsætter indtil vi ender i feltet $R[0]$.

1.4 Opgave 4 - Eksamen juni 2011, opgave 3

I denne opgave ser vi på sortering i to speciel-tilfælde:

1. Alle nøgler er forskellige, og input er omvendt sorteret; dvs. nøglerne optræder i faldende orden. Eks: {12, 9, 8, 5, 2}
2. Alle nøgler er ens. Eks: 5, 5, 5, 5, 5

I begge tilfælde antages det, at alle nøgler er heltal mellem 0 og n^5, hvor n er antallet af nøgler, der skal sorteres.

Spørgsmål a

Angiv køretiden for Insertion Sort i hvert af de to tilfælde.

Ved omvendt sorteret input (1) tager Insertion Sort $\Theta(n^2)$, da hver nøgle i input skal byttes en efter en med alle de nøgler, som ligger før den. Hver byt tager konstant tid, og køretiden kan udregnes ved at summere over antal byt:

$$0 + 1 + 2 + \ldots + (n-3) + (n-2) + (n-1) = \frac{(n-1) \cdot n}{2} = \frac{n^2 - n}{2} = \Theta(n^2)$$

Hvis nøglerne er ens (2), så er køretiden $\Theta(n)$, da ingen byt er nødvendige, så gennemløbet af alle nøglerne det eneste, der tager tid.

Spørgsmål b

Angiv køretiden for Quicksort i hvert af de to tilfælde.

Ved omvendt sorteret input (1) tager bogens Quicksort $\Theta(n^2)$, da når pivotten vælges til at være den sidste nøgle, så bliver nøglerne ikke ligeligt fordelt, de bliver faktisk maksimalt uligeligt fordelt. Dvs. at inputtene til de rekursive kald vil altid består af en tom liste og en liste med én mindre nøgle end forrige omgang. Da Partition løber igennem inputtet én

gang, så kan der siges, at Partition tager k tid, hvor k er input størrelsen. Ud fra dette kan køretiden af Quicksort på omvendt sorteret input (1) udregnes:

$$n + (n-1) + (n-2) + \ldots + 3 + 2 + 1 = \frac{n \cdot (n+1)}{2} = \frac{n^2 + n}{2} = \Theta(n^2)$$

Det samme gælder, når nøglerne er ens (2), altså er køretiden også $\Theta(n^2)$ i dette tilfælde.

Spørgsmål c

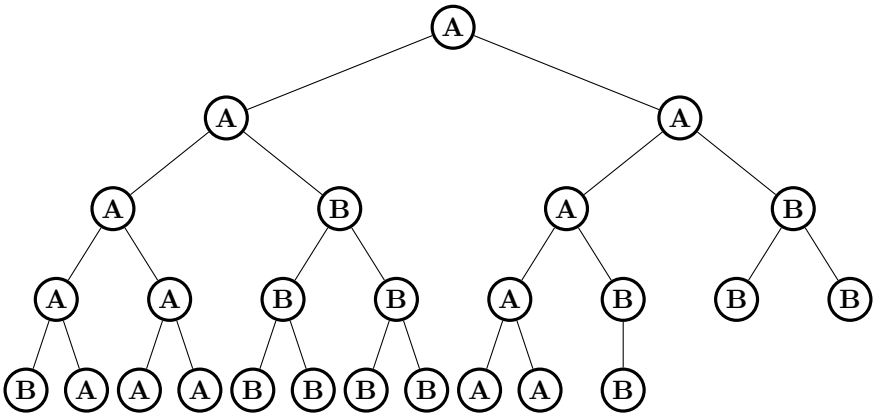
Angiv køretiden for Heapsort i hvert af de to tilfælde. Det skal noteres, at der bruges bogens Heapsort, som bruger en Max-Heap.

Ved omvendt sorteret input (1):

Observer først, at i træer med heapfacon er der mindst lige så mange blade som indre knuder.

Dette kan ses ved, at det sidste index som er en indre knude, er parent af sidste blad. Hvis vi bruger 1-indekserede arrays som i bogen, er sidste blad index n og sidste indre knude har så index $\lfloor n/2 \rfloor$. Dvs. at der er $\lfloor n/2 \rfloor$ indre knuder og dermed $\lceil n/2 \rceil$ blade.

Se på input som de n/2 største knuder (A) og de n/2 mindste knuder (B). Observér at på intet tidspunkt i heapsort kan en knude fra B have en knude fra A som efterkommer (pga. heaporder). Dvs. at knuderne fra B hele tiden udgør en samling deltræer af heapen (dvs. en samling knuder i heapen (= deltræernes rødder) og alle disse knuders efterkommere). Dette kan ses ved følgende eksempel:



Her kan ses, at ingen B knuder har nogle A knuder som efterkommer, og at disse B knuder former deltræer.

Disse træer har alle heapfacon. Så for hvert træ, og dermed for samlingen af træer, er mindst halvdelen af knuderne blade i heapen. Vi har vist, at til ethvert tidspunkt er mindst halvdelen af B blade i heapen.

Specielt gælder dette efter de første n/2 extractMax i Heapsort. Disse extractMax har først udtaget roden i heapen (et element i A), sat et element fra B[1] fra nederste lag op på rodens plads og derefter swappet dem nedad under heapify.

Elementerne fra B kan kun bevæge sig nedad ved at være med i et swap. Vi har ovenfor vist at halvdelen af B må være blade efter de n/2 extractMax. Højden af heapen er der $\lg(n/2) = \lg(n) - 1$. Derfor er der foregået mindst $n/4 \cdot (\lg(n) - 1)$ swaps indtil nu.

Derfor er Heapsort på dette input $\Omega(n \cdot \lg(n))$. Heapsort er altid $O(n \cdot \lg(n))$ for alle input. Derfor er Heapsort $\Theta(n \cdot \lg(n))$ på dette input.

Når alle nøglerne er ens (2), så er køretiden $\Theta(n)$, da *BuildHeap* tager $\Theta(n)$, og eftersom alle nøglerne er ens, så gør heapify intet. Derfor tager *extractMax* $\Theta(1)$ tid. Dette gøres n gange, altså tager dette også $\Theta(n)$. Derfor må Heapsort med ens input (2) tage $\Theta(n)$.

Spørgsmål d

Angiv den bedst mulige køretid for Radix Sort i hvert af de to tilfælde.

Radix Sorts køretid er udelukkende afhængig af input-længde, som er n, og det mulige interval for input, som er $[0, n^5[$. Heltal i intervallet $[0, n^5[$ kan ses som 5-cifrede tal i radix n. Der kan kigges på opgave 9 på ugeseddel 10, for forklaring om, hvordan man ser dette (med n^3 i stedet for n^5).

Radix Sort kører Counting Sort for hver af cifrene, i dette tilfælde bliver Counting Sort kørt 5 gange. Herfra kan køretiden udregnes: $5 \cdot (n + n) = \Theta(n)$.

Altså er køretiden for Radix Sort i begge tilfælde $\Theta(n)$

1.5 Opgave 5 - Eksamen januar 2007, opgave 1

Spørgsmål a

Her får vi givet et binært træ og vi skal så svare på om givne træ er et binært søgetræ. Husker man tilbage til forelæsningerne i starten af marts skal et binært søgetræ overholde følgende krav (fra slide 6 i [2]):

- Træet skal være et binært træ
- Knuderne skal være i inorder

Når knuderne er i inorder betyder det kort sagt for en knude v, at alle nøglerne i v's venstre undertræ er mindre end eller lig med v's nøgle og at alle nøglerne i v's højre undertræ er større end eller lig v's nøgle. For at være sikker på at det holder for det givne træ, bliver vi nødt til at tjekke at det holder for hver knude. Gør man det, vil man kunne se, at knuderne er i inorder, og det givne træ er derfor et binært søgetræ.

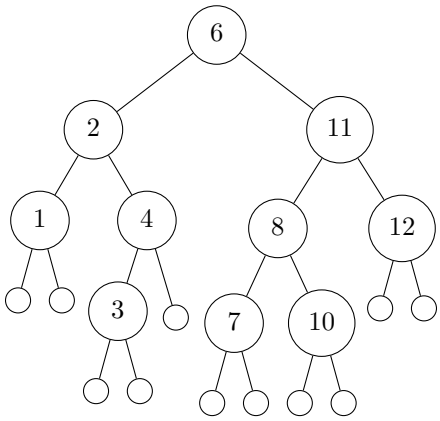
[1] Bemærk at elementer fra A kan kun bevæge sig opad her. Derfor ved vi, at under de første n/2 extractMax vil elementerne taget fra nederste lag og sat op i roden være fra B. Dette er implicit antaget i sætningen ovenfor. Elementerne fra B ender i nederste lag flere gange og kan derfor blive sat op flere gange, så situationen er **ikke** så simpel, at vi bare tager de n/2 mindste elementer efter tur, sådan som startopstillingen kunne give anledning til at tro.

Spørgsmål b

Vi får givet et nyt binært søgetræ, hvor vi skal slette knuden med nøglen 5 og derefter tegne det resulterende træ. Da 5 har både et højre og venstre barn kan vi ikke bare flytte et barn op på 5's plads, hvilket i forhold til algoritmen på side 298 i [1] betyder at vi skal springe de to første if-sætninger over.

Vi skal derfor først finde den mindste nøgle i 5's højre undertræ, hvilket vi gør ved at starte i knuden med nøglen 11 og gå så langt til venstre i undertræet som muligt. I det her træ vil det være knuden med nøglen 6, og da dens forældre ikke er 5 skal vi flytte 6's højre undertræ op på 6's plads for at frigøre 6. [2] I pseudokoden er det på linje 6-9.

Da knuden med nøglen 6 nu er frigjort skal vi sætte dens højre barn til at være 5's højre barn (11) og opdatere 11's forældre til at være 6 (det sker på linje 8-9). Til sidst flytter vi 6 op på 5's plads ved at opdatere dens forældre og sætte dens venstre barn til at være knuden med nøglen 2 (knuden med nøglens 5 venstre barn) og opdatere 2's forældre til at være 6, som man kan se på linje 10-12. Træet kommer derfor ud til at se ud som følgende figur.

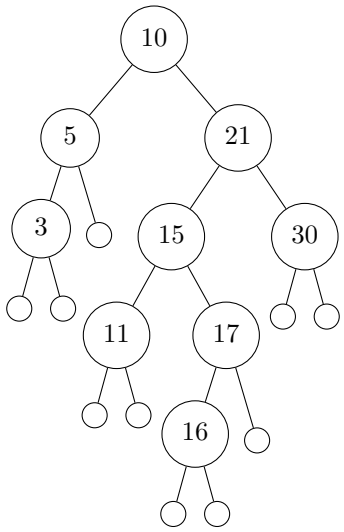


Figur 3: Træet efter knuden med nøglen 5 er blevet slettet

Spørgsmål c

I dette delspørgsmål får vi givet et tredje binært søgetræ, som vi skal indsætte en knude med nøglen 16 i og derefter tegne. Til dette skal vi bruge algoritmen på side 294 i [1]. På linjer 1-7 laver vi praktisk talt en søgning ned igennem træet for at finde en plads til den knude vi vil indsætte. I træet i opgaven gennemløbes knuderne 10, 21, 15, 17 (i den rækkefølge) og vi finder ud af 16 skal være knuden med nøglen 17's venstre barn. På linje 8-13 sætter vi 16's forældre til at være 17 og 16 til at være 17's venstrebarn (vi udfører if-sætning på linje 11). Derfor kommer træet efterfølgende til at se ud som følgende figur.

[2] Man kan her notere sig, at siden vi fandt 6 ved at gå så langt som muligt til venstre i undertræet med 11 som rod, så ved vi at 6 ikke har noget venstre barn.



Figur 4: Træet efter knuden med nøglen 16 er blevet indsat

2 Eksaminatorie-timer II

2.1 Opgave 1 - Cormen et al. opgave 22.1-1

Givet en naboliste-repræsentation for en graf skal vi angive hvor lang tid det tager at udregne ind-grad og ud-grad for alle knuder i grafen.

Herunder er givet pseudokode for en simpel algoritme der tæller ind- og ud-graden for alle knuder i en graf G med n knuder, m kanter og naboliste-repræsentation

```
1  function count_in_out_degree(G,n):
2         let in_degree[1...n] be a new array
3         let out_degree [1..n] be a new array
4         for i = 1 to n:
5             in_degree[i] = 0
6             out_degree[i] = 0
7         for v in G:
8             for u in G.adj(v):
9                 in_degree[u] += 1
10                out_degree[v] += 1
11        return(in_degree, out_degree)
```

På linjerne 2-3 laver vi nye lister der holder styr på ind- og ud-graden for hver knude. Knuden i's ind- og ud-grad vil så være i in_degree[i] og out_degree[i].

I for-løkken på linje 9 løber vi igennem alle knuder, $\Theta(n)$. For hver knude gennemløber vi dens ud-kanter. I alt besøger vi hver kant en gang, $\Theta(m)$. Dvs. vi ender med en køretid på $\Theta(n+m)$.

Bemærkning: hvis man er interesseret i ind- og ud-graden for en enkelt knude i stedet for alle knuder er der forskel på tiden for ind- og ud-graden. Tid for at udregne ud-graden for en knude v er lineært i antallet af knuder i knudens nabo-liste $\Theta(size(G.adj(v))$. Tid for

at udregne ind-graden bliver $\Theta(m)$ fordi vi er nødt til at besøge alle kanter for at være sikker.

2.2 Opgave 2 - Cormen et al. opgave 22.1-3

En transponeret orienteret graf $G = (V, E)$ er grafen $G^T = (V, E^T)$, hvor $E^T = \{(v, u) \in V \times V : (u, v) \in E\}$. Altså, G^T er G med alle kanter vendt rundt. Beskriv en effektiv algoritme for at bestemme G^T ud fra G, for både naboliste og nabomatrice repræsentationer af G. Analyser køretiden for algoritmerne.

Nabomatricen: Her skal man bare transponere nabomatricen. I pseudokode vil det svare til

```
1  transpose(G)
2  // lad G.adj[x, y] vaere pladsen i den x'te raekke og y'te kolonne i grafens nabomatrice
       .
3  G' = G.copy()
4  for i = 1 to |G.V|
5      for j = 1 to |G.V|
6          G'.adj[i, j] = G.adj[j, i]
7  return G'
```

Nabomatricen for en graf med $|V|$ knuder har størrelse $|V| \times |V|$. Da man i algoritmen laver konstant arbejde for hver plads i nabomatricen, så er køretiden $\Theta(|V|^2)$.

Naboliste: Lad G være en graf og G' være en kopi af G med tomme nabolister for knuderne. For hver knude u i $G.V$, gennemløb dens naboliste. For hver knude v i nabolisten, indsæt u i nabolisten for v i G'. I pseudokode vil det svare til

```
1  transpose(G)
2      G' = G.copy(empty_adj_list=true) // kopi af G med tomme nabolister
3      for u in G.V
4          for v in G.adj[u]
5              G'.adj[v].insert(u)
6      return G'
```

Først skal man lave en ny graf. Dette kræver man laver $|V|$ tomme nabolister, hvilket tager konstant tid for hver naboliste. Derefter skal man gennemløbe alle nabolister i den originale graf, og opdatere nabolister i den nye graf (indsætte knude i en naboliste). Der er $|E|$ kanter i alt og en opdatering kan foretages i konstant tid. Alt i alt bliver køretiden $\Theta(|V| + |E|)$.

2.3 Opgave 3 - Cormen et al. 22.1-6

I følgende opgave skal vi skitsere en algoritme der givet en orienteret graf $G = (V, E)$, i form af dens *adjacency matrix*, kan bestemme hvorvidt der findes en *universal sink* i grafen, i $O(|V|)$ tid. For at kunne gøre dette bemærker vi at:

1. En universal sink er kendetegnet ved at have in-degree $|V| - 1$ og out-degree 0.
2. Dette betyder at en knude der er *kandidat* til at være en universal sink ikke kan have en kant til sig selv (self-loop), da den så ellers ville have out-degree mindst 1, hvilket strider imod definitionen på en univsersal sink.

3. I en adjacency-matrix resulterer dette i at en knude v_k for $k \in \{1, \ldots, |V|\}$ er en universal sink, hvis kolonne k består af 1'ere (på nær værdien på diagonalen) og række k består af 0'ere. Dette er tilfældet, da out-degree for en knude v_k er summen af dens række k og in-degree for en knude v_k er summen af dens kolonne k.

På baggrund af disse punkter kan vi skitsere en algoritme der, givet en adjacency matrix, bestemmer om en bestemt knude v_k i den korresponderende graf er en universal sink:

```
# Lad A være en given adjacency matrix af størrelse |V| x |V|
# Lad k være et indeks på en kolonne og en række i A
Is-Universal-Sink(A, k):
    for i = 1 to |V|:
        if A[i][k] == 0 and i != k: # Tjek kolonne k
            return False
    for j = 1 to |V|:
        if A[k][j] == 1: # Tjek række k
            return False
    return True
```

For at kunne bestemme om en graf indeholder en universal sink og ikke bare om en enkelt knude er en universal sink eller ej, så bliver vi nødt til at tilføje et ekstra lag til algoritmen. Dette gør vi ved følgende algoritme:

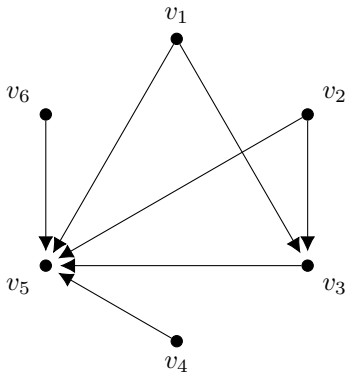
```
# Lad A være en given adjacency matrix af størrelse |V| x |V|
Contains-Universal-Sink(A):
    i = j = 1
    while i <= |V| and j <= |V|:
        if A[i][j] == 0:      # Gå mod højre i adjacency matricen
            j = j + 1
        else if A[i][j] == 1: # Gå nedad i adjacency matricen
            i = i + 1
    if i > |V|: # Bunden af matricen er nået
        return "Ingen universal sink eksisterer"
    else if Is-Universal-Sink(A, i) == False:
        return "Ingen universal sink eksisterer"
    else:
        return "Knude "i" er en universal sink!"
```

Denne algoritme løber en given adjacency matrix `A` igennem på en struktureret måde for at finde en knude der kan være en mulig kandidat til at være en universal sink. Efter en mulig kandidat er fundet, tjekkes denne om den faktisk er en universal sink eller ej.

Mere præcist, så starter algoritmen `Contains-Universal-Sink` ved element `A[1][1]` i adjacency matricen og skanner mod højre i denne så længe at `A[i][j] = 0`, dvs. forøg indeks j indtil vi møder et element `A[i][j] = 1`. Når vi endelig møder `A[i][j] = 1`, så begynd at skanne nedad i adjacency matricen, dvs. forøg indeks i indtil vi møder `A[i][j] = 0`. Hvis vi møder et element `A[i][j] = 0`, så begynd at skanne mod højre i matricen igen, som forklaret tidligere.

Skanprocessen må på et tidspunkt stoppe og vi må i sidste ende enten (i) nå ud over bunden af matricen, eller (ii) nå ud over højre side af matricen. I tilfælde (i) må alle knuderne have out-degree mindst 1, da vi har mødt et 1-tal i hver række i matricen som vi har bevæget os nedad i indtil vi er nået til bunden. Grafen kan derfor ikke indeholde en kandidat der kan være en universal sink. I tilfælde (ii) har vi fulgt en sti der består af en række vandrette stykker der har facon $0, \ldots, 1$ (et antal 0'ere efterfulgt af et 1-tal) skiftende med lodrette stykker

$1, \ldots, 0$ (et antal 1'ere efterfulgt af et 0). Dette er illustreret i Figur 5. Hvis man er i case (ii), vil stiens $0, \ldots, 0$-stykker tilsammen dække kolonnerne fra 1 til $|V|$. Disse stykker aflyser som kandidat indeks for alle de kolonner, som de passerer (da korresponderende knuder så må have in-degree mindre end $|V| - 1$), *undtagen* hvis stykket passerer kolonnen på diagonalen. Det vil sige, at et vandret stykke i række k kan efterlade indeks k som kandidat. Så umiddelbart har vi i case (ii) flere kandidater tilbage til slut, når vi er nået ud over højre side af matricen. Vi vil dog have, at hvis et vandret stykke på række k afsluttes med et 1-tal (dvs. har facon $0, \ldots, 1$), fjerne dette indeks k som kandidat, da out-degree for den korresponderende knude så mindst vil være 1. Alle vandrette stykker, *undtagen det sidste* har denne facon. Derfor er eneste mulige kandidat (ved en case (ii)-afslutning) række-indeks for det sidste vandrette stykke (og dette indeks er netop sidste værdi af i).



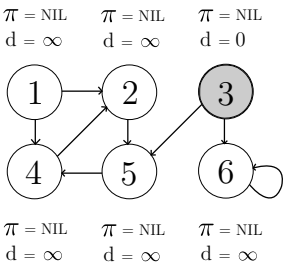
	v_1	v_2	v_3	v_4	v_5	v_6
v_1	$\rightarrow$	$\rightarrow$	$1\downarrow$		1	
v_2			$1\downarrow$		1	
v_3			$\rightarrow$	$\rightarrow$	$1\downarrow$	
v_4					$1\downarrow$	
v_5					$\rightarrow$	$\rightarrow$
v_6					1	

Figur 5: En given graf og dens korresponderende adjacency matrix (0'ere er undladt). Stien i adjacency matricen viser et eksempel på den beskrevne case (ii)-afslutning. I dette tilfælde er knude $i = k = 5$ en kandidat til at være en universal sink. Kolonne og række k kan herefter tjekkes for at konkludere om denne knude faktisk er en universal sink eller ej (i dette tilfælde er det).

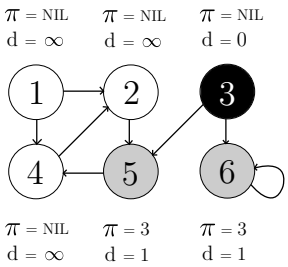
Køretiden for denne algoritme vil endeligt være $\Theta(|V|)$, da vi højest skal gennemløbe et antal elementer der svarer til længden + bredden af matricen $\Theta(|V| + |V|)$ (dette afspejler køretiden af while-løkken i `Contains-Universal-Sink`), mens tjekket til sidst også kun tager $\Theta(|V| + |V|) = \Theta(|V|)$ tid (dette afspejler køretiden af `Is-Universal-Sink`).

2.4 Opgave 4 - Cormen et al. opgave 22.2-1

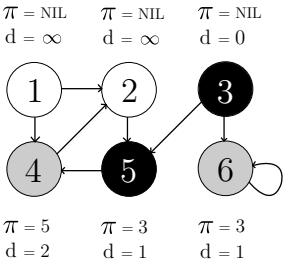
Vi følger pseudokoden som beskrevet på Rolfs slides. Hver figur er vist inden en iteration (den sidste inden løkken stopper):

(a)


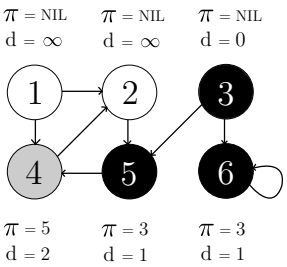
Q = [3]

(b)


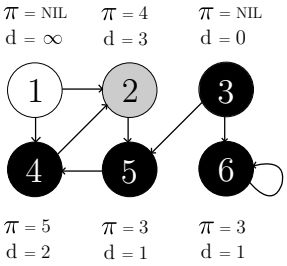
Q = [5,6]

(c)


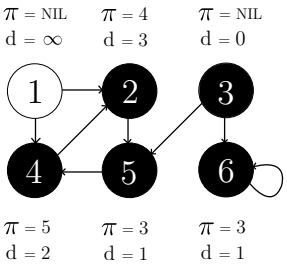
Q = [6,4]

(d)


Q = [4]

(e)


Q = [2]

(f)


Q = []

2.5 Opgave 5 - Cormen et al. opgave 22.2-2

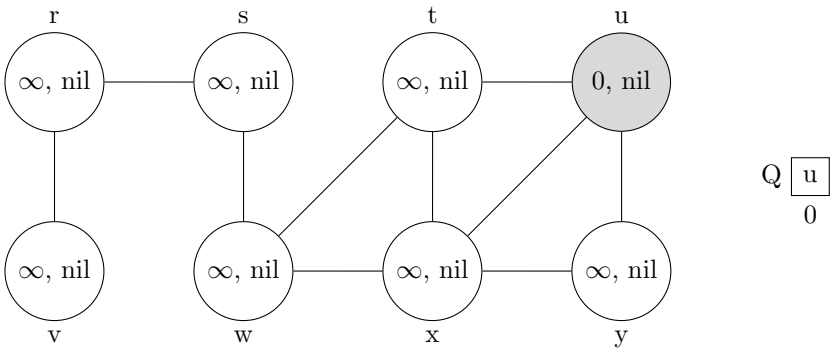
Vis d og π værdierne som resulterer ved et gennemløb af *Breadth-first Search* på graf 22.3[1, p.596], hvor knude u er begyndelsepunktet.

Først initialiseres algoritmen.

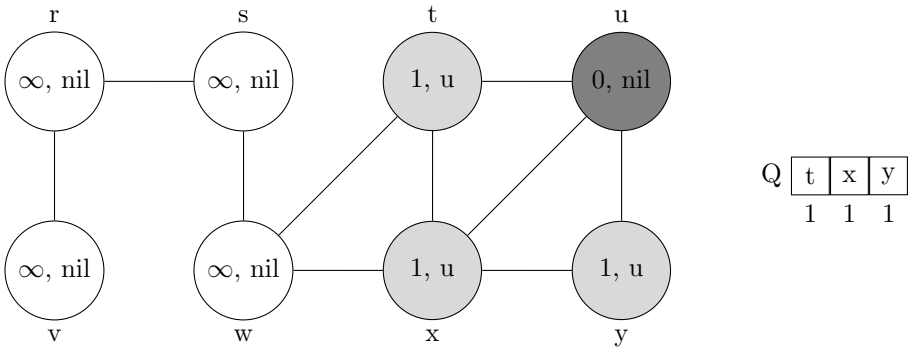
Linje 1-4 i algoritmen farver alle knuder, bortset fra begyndelsepunktet u, hvide, og sætter deres distance (d) til ∞ og deres predecessor (π) til nil.

Linje 5-7 farver vores begyndelsepunkt u gråt, sætter dens distance (d) til 0 og sætter dens predecessor (π) til nil.

Linje 8 skaber en kø Q, som indeholder alle grå knuder i sorteret orden efter deres distance (d), og linje 9 indsætter begyndelsepunktet u i Q.

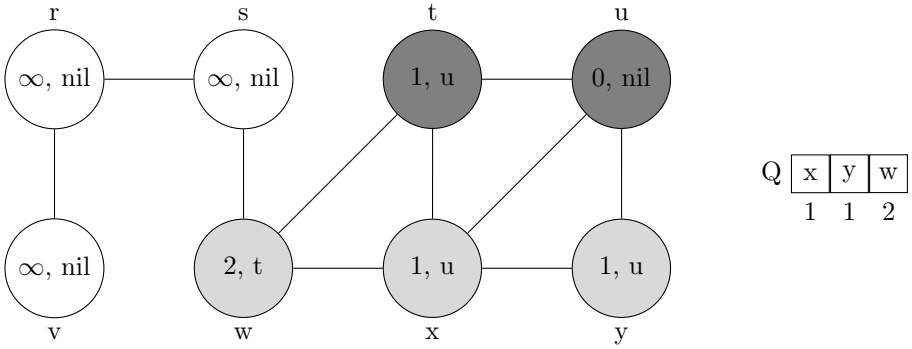


Derefter tages knuderne ud af køen Q en af gangen, indtil køen er tom. I første iteration af løkken tages u ud. Der kigges på u's naboer og finder dem, som er hvide: t, x og y. Disse tre knuders distance (d) sættes til u's distance +1, og deres predecessor (π) til u. Knuderne farves grå og bliver sat ind i Q, og til sidst farves u sort.

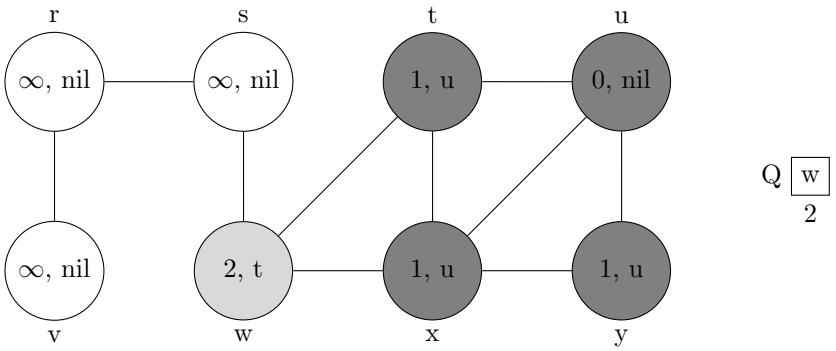


Her er der valgt at antage, at nabolister er ordnet alfabetisk, men dette er ikke et krav for BFS.

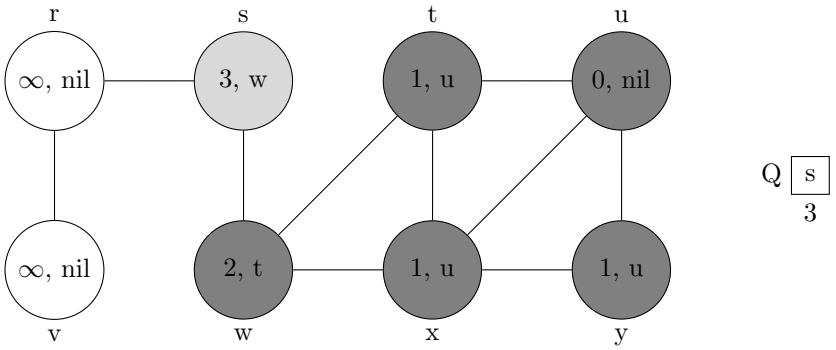
Derefter tages t ud af Q, og der kigges på t's naboer. w er den eneste hvide nabo t har. w's distance (d) sættes til t's distance +1, w's predecessor (π) til t, w farves grå, og w indsættes i Q. t farves sort.



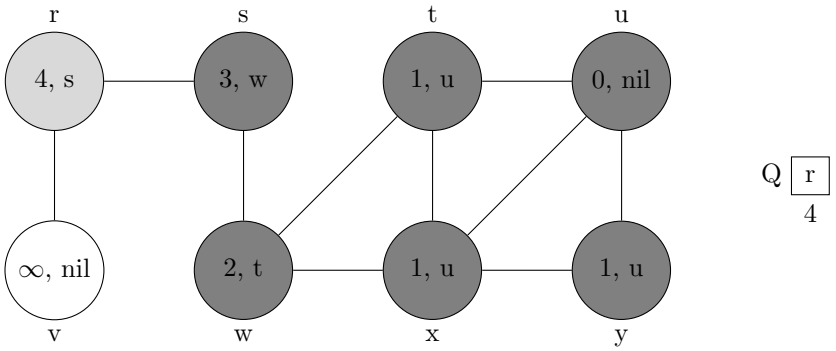
Her tages x ud af Q, der kigges på x's naboer, men da alle x's naboer er grå eller sorte, så sker der ikke noget med naboerne og x farves sort. Det samme sker derefter for y.



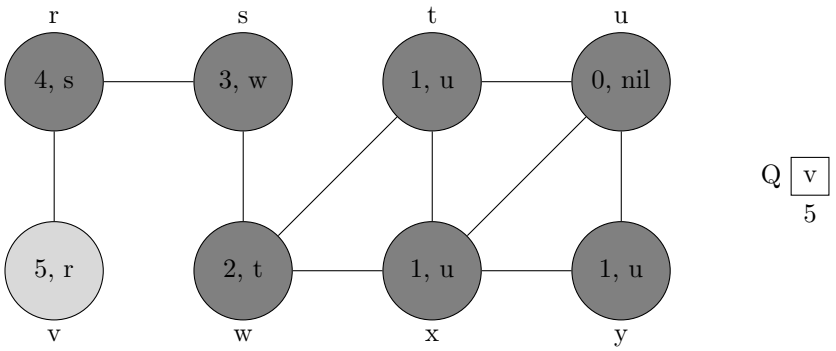
Derefter tages w ud af Q, og der kigges på w's naboer. s er den eneste hvide nabo w har. s's distance (d) sættes til w's distance +1, s's predecessor (π) til w, s farves sort, og s indsættes i Q. w farves sort.



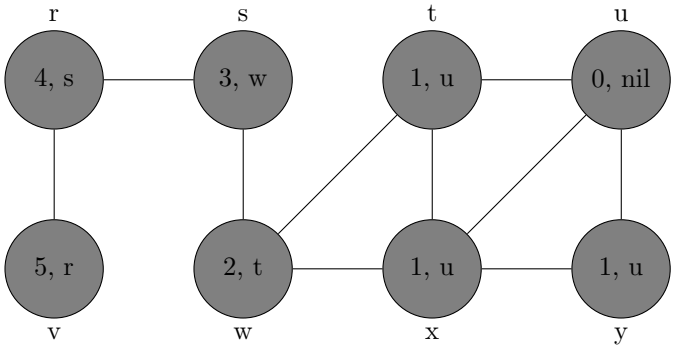
Dette gentages med s's naboer, hvor dens eneste hvide nabo er r.



Dette gentages med r's naboer, hvor dens eneste hvide nabo er v.



Derefter tages v ud af Q, der kigge på v's naboer, men da ingen af v's naboer er hvide, så sker der ikke andet end v farves sort.



Da Q nu er tom, så er algoritmen *Breadth-first Search* færdig, og dermed kan distancen (d) og predecessor (π) nu aflæses for alle knuderne.

2.6 Opgave 6 - Cormen et al. opgave 22.2-3

I denne opgave skal vi argumentere for at man kan nøjes med at bruge 1 bit til at gemme farven på knuderne i en graf når man kører `Breadth-first search`(side 595 i [1]) på den. Mere præcist skal vi argumentere for at man kan fjerne linje 18 i pseudokoden uden at ændre hvad algoritmen vil returnere.

Kigger man på pseudokoden, kan man hurtigt udelukke at det at fjerne linje 18 kan have nogen indflydelse på andet end while-løkken på linjerne 10-18. Inde i while-løkken sker der lidt groft sagt følgende: Vi tager en knude u ud af vores kø Q, hvorefter vi for alle dens hvide naboer i Q opdaterer deres farve til grå, deres afstand fra s til $u.d+1$ og deres predecessor til u, inden de bliver sat ind i Q. Det vigtige er her, at vi kigger på knuder som har farven hvid og intet sted bruger at nogle knuder kan have en bestemt farve udover hvid (og ikke-hvid). Da en knude bliver farvet grå inden den bliver sat ind i Q og først senere ville blive farvet sort, må algoritmen virke på samme måde med linje 18 fjernet. Da hvid og grå så er de eneste brugte farver, kan vi med en enkelt bit indikere hvilken af de to gælder for en bestemt knude.

2.7 Opgave 7 - Eksamen juni 2009, opgave 1a

I denne opgave skal vi udføre `Heap-Extract-Max` på hoben i tabel 2.

10	7	6	5	4	2	3	1	2	3	1	1

Tabel 2

Pseudokoden for `Heap-Extract-Max` kan findes i [1, side 163].

Første skridt er at gemme det største element i en variabel for sig selv og overskrive det største element med det sidste element i hoben og gøre hoben en mindre.

max											
10	1	7	6	5	4	2	3	1	2	3	1

Tabel 3

Derefter kaldes `Max-Heapify` på første element i hoben. Pseudokoden for `Max-Heapify` kan findes i [1, side 154].

max	i	l	r								
10	1	7	6	5	4	2	3	1	2	3	1

Tabel 4

I tabel 4 sammenligner vi i's højre barn (r) og venstre barn (l) med i og finder den største til at være det venstre barn. Vi bytter rundt på i og l og kalder derefter `Max-Heapify` på pladsen hvor i er flyttet hen.

max		i		l	r						
10	7	1	6	5	4	2	3	1	2	3	1

Tabel 5

I tabel 5 bytter vi rundt på i og l.

max				i				l	r		
10	7	5	6	1	4	2	3	1	2	3	1

Tabel 6

I tabel 6 bytter vi rundt på i og r.

max									i		
10	7	5	6	2	4	2	3	1	1	3	1

Tabel 7

I tabel 7 ser vi at i ikke længere har nogen børn; derfor er vi færdige og `Max-Heapify` terminerer.

I slutningen af `Heap-Extract-Max` returnere vi `max`.

2.8 Opgave 8 - Eksamen januar 2005, opgave 5

Beskriv en algoritme, der sorterer n heltal i tid $O(n \log(\log n))$, hvis der kun er $O(\log n)$ forskellige tal. Det antages, at to tal kan sammenlignes i konstant tid (dvs. i tid $O(1)$). Forklar, hvordan den sorterede følge af n heltal kan udskrives i tid $O(n)$.

Man er typisk ikke interesseret i at sortere tal. Som regel sorterer vi (*nøgle*, *data*), hvor sorteringen foretages ift. nøglen (dette kunne f.eks. være tal). Det er derfor vigtigt, at vi i nedenstående ikke mister relationen mellem nøgle og data.

Indsæt de n heltal ind i et rød-sort træ, men foretag følgende ændringer:

- Lad knudernes nøgler være unikke. Dermed er antallet af knuder i træet $O(\log n)$.
- Gem, som satellit information i hver knude, en *liste*, der kan indeholde den tilhørende data (eller pointere dertil).
- Ved indsættelse af et element med nøglen k, tjek først om k allerede er en nøgle i træet. Dette kan gøres i $O(\log(\log n)))$ tid, da søgning tager $O(\log n')$ tid, hvor n' er antallet af knuder i træet og $n' = O(\log n)$. Hvis k allerede er i træet, så indsættes det indsatte elements data i knudens liste i $O(1)$ tid (husk: knuden returneres fra søgningen)[3]. Hvis k ikke er i træet, så indsættes en ny knude med nøglen k og en liste indeholdende det indsatte elements data. Indsættelsen tager også $O(\log(\log n)))$ tid. Man kan evt. foretage indsættelse i et trin (husk man under indsættelse implicit foretager en søgning), men asymptotisk er køretiden ens, nemlig $O(\log(\log n))$.

Da der foretages n indsættelser i alt, så er køretiden $O(n \log(\log n))$.

For at udskrive tallene foretages et inorder gennemløb af træet, hvor hver knudes nøgle udskrives én gang for hvert element i denne knudes liste. Eftersom der er $O(\log n)$ knuder, det tager lineær tid at gennemløbe (samt udskrive nøglen én gang for hvert element) de individuelle knuders lister og den aggregerede størrelse af alle de individuelle lister er n, så tager det $O(n)$ tid at udskrive de indsatte heltal i sorteret orden.

2.9 Opgave 9 - Eksamen juni 2010, opgave 5

I denne opgave arbejder vi med udvidede binære søgetræer, hvor der i hver knude i træet er gemt ekstra informationer om den største afstand mellem en knude og dens *predecessor*. Yderligere informationer er også gemt i en knude, således at disse kan anvendes til at bestemme den føromtalte største afstand.

[3]Hvis vi kun er interesseret i at sortere nøgler uden noget data tilknyttet, så kan en tilfældig værdi indsættes (vigtigt eftersom listens længde bruges senere).

Delopgave a & b I denne delopgave skal vi angive hvorledes en knudes ekstra informationer kan bestemmes i $O(1)$ tid. Mere præcist, hvis vi ser på en knude v med et venstrebarn u og et højrebarn w (som begge er forskellige fra NIL), så er vi interesserede i at kunne bestemme følgende i $O(1)$ tid:

- Den største afstand $v.maxGap$ mellem en knude og dens predecessor. Per definition af in-order egenskaben i søgetræer er alle de søgte afstande i v's undertræ enten i (1) u's undertræ, eller i (2) w's undertræ, eller (3) afstanden mellem $v.key$ og $u.max$ (i dette tilfælde er dette værdien på nøglen af v's predecessor), eller (4) afstanden mellem $w.min$ (i dette tilfælde er dette værdien på nøglen af v's successor) og $v.key$. Derfor kan vi bestemme $v.maxGap$ i $O(1)$ tid som:

$$v.maxGap = \max\{u.maxGap,\ w.maxGap,\ v.key - u.max,\ w.min - v.key\} \quad (3)$$

  Vi bemærker at dette stadig er korrekt hvis u's eller v's undertræ kun indeholder én nøgle, fordi deres $maxGap$ da per definition er sat til nul (og derfor ikke kan dominere over de andre værdier der maksimeres over, eftersom afstande altid er ikke-negative).

- Værdierne $v.max$ og $v.min$. Disse kan henholdsvis bestemmes som $w.max$ og $u.min$. Hvis u eller w er NIL, så fås simplere versioner af ovenstående udregning. Hvis f.eks. u er NIL, mens w ikke er, skal vi i stedet bruge $v.maxGap = \max\{w.maxGap,\ w.min - v.key\}$, og $v.max$ og $v.min$ sættes til henholdsvis $w.max$ og $v.key$. Hvis begge er NIL, skal $v.maxGap$ sættes til nul, og $v.max$ og $v.min$ begge sættes til $v.key$.

Grundet sætning 14.1 i [1, s. 346] og ovenstående, dvs. at vi kan bestemme en knudes informationer i $O(1)$ tid kun vha. informationerne som er gemt i en knude selv og dens børn, så er det muligt at vedligeholde knudernes informationer under indsættelser og sletninger, uden at ændre køretiden på $O(\log n)$.

Delopgave c Den største afstand i træet mellem nøgler og deres predecessorer kan bestemmes ved at aflæse roden r's information $r.maxGap$. I denne forbindelse ønsker vi at kunne finde en konkret nøgle i træet som har denne afstand til sin predecessor.

Dette kan laves ved en rekursiv søgealgoritme, der starter i roden, og som for en besøgt knude v med venstrebarn u og højrebarn w (begge forskellige fra NIL) undersøger, hvilken af de fire værdier $(v.key - u.max)$, $(w.min - v.key)$, $u.maxGap$ og $w.maxGap$, der er lig $v.maxGap$ (mindst én af dem skal være det). I det første tilfælde returneres $v.key$. I det næste tilfælde returneres $w.min$.

I de to sidste tilfælde kalder algoritmen rekursivt sig selv på henholdsvis u og w. Hvis enten u eller w er NIL, er der blot to ud af de fire cases at betragte. (Hvis både u og w er NIL, bør der returneres en fejlkode - dette sker kun når hele træet har størrelse én, og da der ikke er nogle knuder som har en predecessor, er problemet ikke veldefineret).

Der bruges $O(1)$ tid i hver knude, og alle besøgte knuder befinder sig på én sti fra roden til et blad. Da et rød-sort træ har højde $O(\log n)$, er den samlede tid $O(\log n)$.

For til sidst at sætte det hele i kontekst er en mulig pseudokode for den beskrevne søgeprocess formuleret i det følgende:

```
1  # Lad v være en knude. Knude v er lig med roden ved første funktionskald
2  find-maxGap-key(v)
3      if v.u == NIL and v.w == NIL:
4          return Error
5      if v.u != NIL:
6          if (v.key - v.u.max) == v.maxGap: # Vi har fundet nøglen vi leder efter
7              return v.key
8          else if v.u.maxGap == v.maxGap: # Led efter nøgle i venstre undertræ til v
9              return find-maxGap-key(v.u)
10     if v.w != NIL:
11         if (v.w.min - v.key) == v.maxGap: # Vi har fundet nøglen vi leder efter
12             return v.w.min
13         else if v.w.maxGap == v.maxGap: # Led efter nøgle i højre undertræ til v
14             return find-maxGap-key(v.w)
```

Litteratur

[1] T.H. Cormen, C.E. Leiserson, R.L. Rivest, and C. Stein. *Introduction to Algorithms*. MIT Press, third edition, 2009.

[2] Rolf Fagerberg. Dictionaries. URL `https://imada.sdu.dk/~rolf/Edu/DM507/F20/dictionarySlides.pdf`, 2020.