Institut for Matematik og Datalogi
Syddansk Universitet, Odense

16. marts 2020

DM507 — Algoritmer og datastrukturer

Eksaminatorie-timer uge 11 II, Forår 2020

Instruktorerne for DM507

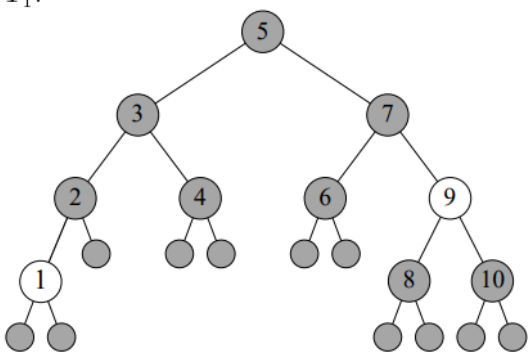
Indhold

1 Opgave 1 - Eksamen jan 2005, opgave 1

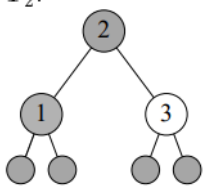
Spørgsmål a

Hvilke af følgende fire træer er rød-sorte træer? Begrund dine svar.

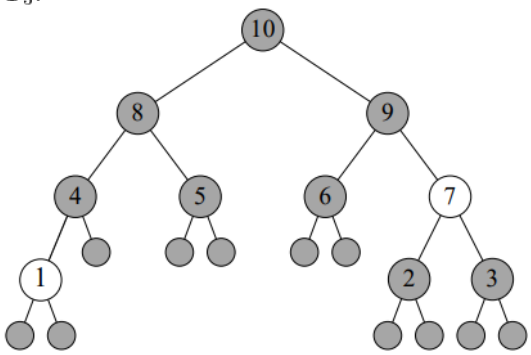
T_1:



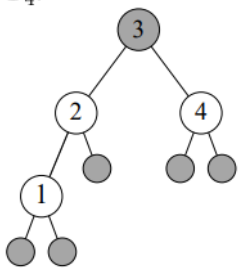
T_2:



T_3:



T_4:



Et gyldigt rød-sort træ har følgende 5 krav: [1, p. 308]

1. Træet skal overholde reglerne for et binært søgetræ
2. Hver knude skal være enten rød eller sort
3. Roden og bladene skal være sorte
4. Der må ikke være 2 røde knuder i streg på en rod-blad sti (dvs. som er forbundet)
5. Der skal være samme antal sorte knuder på alle rod-blad stier

T_1 er et rød-sort træ, da den ikke bryder nogen af de 5 givne regler.

T_2 er ikke et rød-sort træ, da rod-blad stien 2-3-nil rammer to sorte, 2 og nil, mens rod-blad stien 2-1-nil rammer tre sorte, 2, 1 og nil. Altså er krav 5 brudt.

T_3 er ikke et rød-sort træ, da det ikke er et binært søgetræ.
Der gælder, for alle binære søgetræer, at forholdet mellem en knude v's nøgle og alle nøgler i v's undertræer

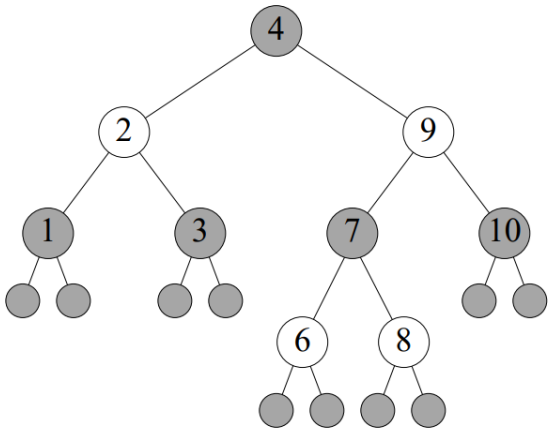
$$\text{nøgler i v's venstre undertræ} \ \leq \ \text{nøgle i v} \ \leq \ \text{nøgler i v's højre undertræ}$$

F.eks. i roden af T_3 er nøglen 10, mens i rodens højre barn er nøglen 9, som burde være lig eller større end roden. Altså er krav 1 brudt.

T_4 er ikke et rød-sort træ, da knuden 2 og 1 kommer i træk i rod-blad stien 3-2-1-nil, men både 2 og 1 er røde. Altså er krav 4 brudt.

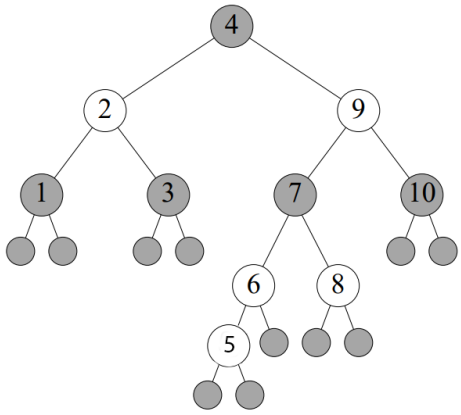
Spørgsmål b

Betragt følgende rød-sorte træ.

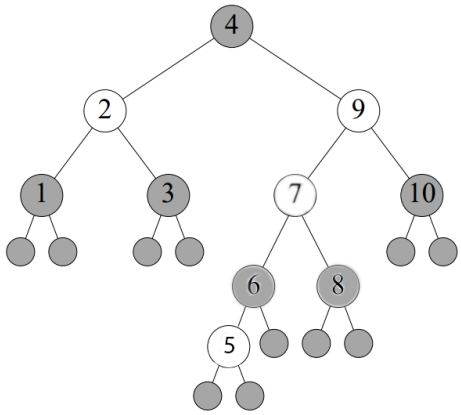


Nu indsættes et element med nøgle 5. Tegn træet, som det ser ud efter indsættelsen. Vis de enkelte trin i indsættelsen.

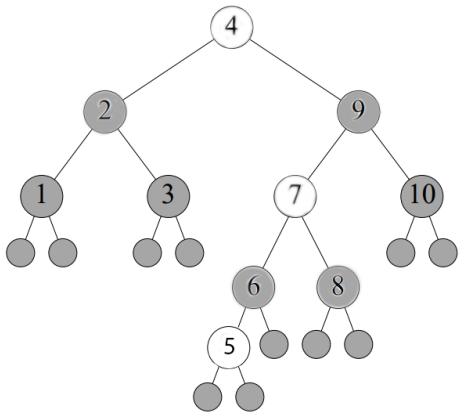
Indsættelse kan deles op i to dele. Først indsættes noden i træet, og derefter "fikses"træet. Den første del foregår på samme måde som ved indsættelse i et binært træ. Først sammenlignes 5 med roden 4. 5 er større end 4, så 4's højre barn er den næste, der bliver spurgt. 5 er mindre end 9, altså er 9's venstre barn den næste. 5 er mindre end 7, altså igen venstre. Til sidst, 5 er mindre end 6, men 6's venstre barn er nil, altså er der plads til knuden.



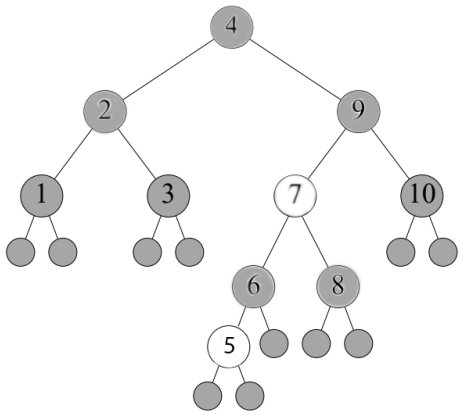
I rød-sorte træer bliver nye knuder farvet røde. Dette gøres, for at sikre sig at "Samme antal sorte på alle rod-blad stier"egenskaben ikke brydes. Der opstår dog nu et nyt problem, da 5 og 6 begge er røde og ligger i træk. Der kigges på hvilken en af de cases fra Dictionaries [2, p. 25], som opstår. Da 5's onkel, altså dens forælders søskende, er 8, og 8 er rød, så er det case 1. I case 1, skal 5's forælder, 6, og onkel, 8, farves sorte og 5's bedsteforælder, 7, skal farves rød.



Derefter kigges der på 7, for at tjekke om der opstår et "to røde i træk-problem mellem 7 og dens forælder. Det gør der, da både 7 og 9 er røde. 7's onkel, 2, undersøges. Da 2 er rød, så rammer vi case 1 igen. Altså skal 7's forælder, 9, og onkel, 2, farves sorte og 7's bedsteforælder, 4, skal farves rød.



Da vi nu har ramt roden, er vi færdige med at kigge efter "to røde i træk-problemer, vi skal dog farve roden sort igen efter indsættelse.

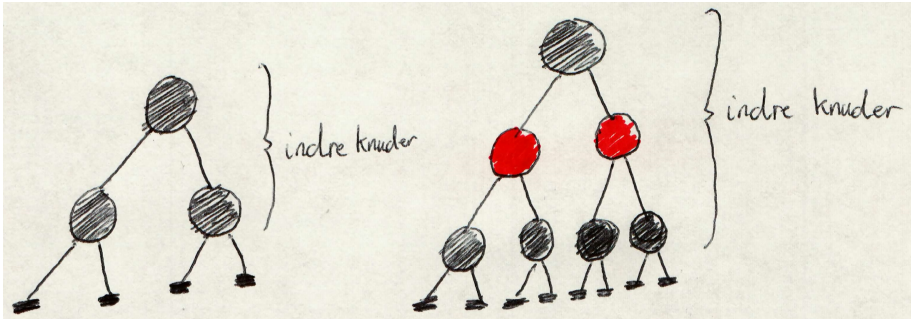


Dermed er vi færdige med at indsætte et element med nøglen 5.

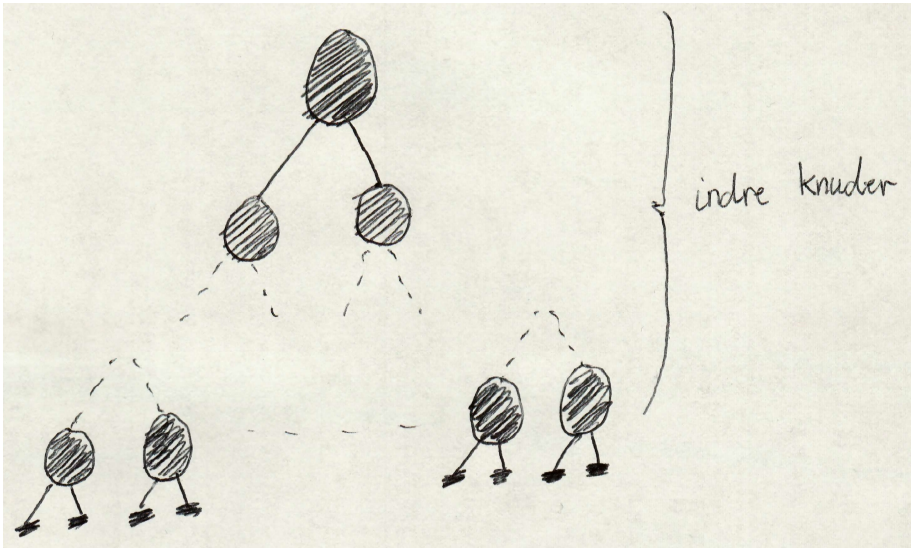
2 Opgave 2 - Cormen et al. opgave 13.1-6

Hvad er det største mulige antal indre knuder i et rød-sort træ med *sort-højde* k? Hvad er det mindst mulige antal indre knuder?

For at få lidt intuition, betragt da rød-sort træerne på Figur 1. Begge har en sort-højde på 2, men den ene har rod-blad stier med en rød knude, som bevirker, at antallet af indre



Figur 1: To rød-sorte træer med sort højde 3



Figur 2: Rød-sort træ med sort-højde k og mindst mulige antal indre knuder.

knuder stiger[1].

Det mindst mulige antal indre knuder opnås altså, når rød-sort træet kun består af sorte knuder. Da alle knuder er sorte og sort-højden er k, så vil der være $k+1$ fulde lag af knuder. Observer, at træet vi beskriver (se Figur 2) er et fuldt binært træ, da der ellers vil være en rod-blad sti med en anden sort-højde. Det sidste lag består af blade; altså, der er k lag af indre knuder. Jf. formlen fra [3, s. 23] er antallet af (indre) knuder i de første k lag

$$\underbrace{2^0 + 2^1 + 2^2 + \cdots + 2^{k-2} + 2^{k-1}}_{\text{k lag af knuder lagt sammen}} = 2^{(k-1)+1} - 1 = 2^k - 1$$

[1]Detalje: Bogen definerer *sort-højden* for en rod-blad sti som værende antallet af sorte knuder fra roden til bladet, men uden at tælle roden med. Rolf benytter ikke *sort-højde* sti i hans slides. Han benytter i stedet, at k er "det samlede antal sorte på en sti". Derfor er der en lille forskel mellem Rolfs slides og løsningen til denne opgave.

Tidligere opdagede vi, at røde knuder "strækker"en rod-blad sti ud uden at påvirke sort-højden. Det størst mulige antal indre knuder opnås altså når rød-sort træet skifter mellem lag af sorte og røde knuder. Observer følgende fra [2, s. 17]

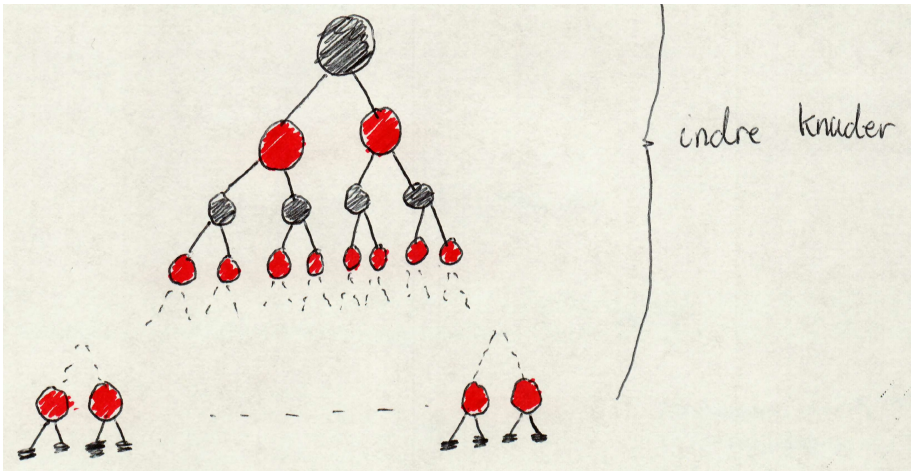
- Der er ikke to røde nabo-knuder.
- Det første (roden) og sidste lag (`NIL` blade) betår af sorte knuder.

Da der er $k+1$ sorte knuder på alle rod-blad stier (sort-højden er k, men roden tælles ikke med) og lagene skifter som

$$\underbrace{\text{Sort}}_{\text{første lag med sort rod}} \to \text{Rød} \to \text{Sort} \to \ldots \to \text{Rød} \to \underbrace{\text{Sort}}_{\text{sidste lag med sorte blade}}$$

er der $2k+1$ (ikke $2k+2$ da sidste lag ikke er rødt men sort) fulde lag af knuder i dette rød-sort træ. Der er igen tale om et fuldt binært træ, da alle rod-blad stier har samme antal knuder. Ligesom før består det sidste lag af blade; altså, der er $2k$ lag af indre knuder. Antallet af knuder i de første $2k$ lag er

$$\underbrace{2^0 + 2^1 + 2^2 + \cdots + 2^{2k-2} + 2^{2k-1}}_{\text{2k lag af knuder lagt sammen}} = 2^{(2k-1)+1} - 1 = 2^{2k} - 1$$



Figur 3: Rød-sort træ med sort-højde k og størst mulige antal indre knuder.

3 Opgave 3 - Eksamen juni 2011, opgave 1

Spørgsmål a

Her får vi givet 8 forskellige træer med farvede knuder som vi skal bedømme om er rød-sorte træer. For at kunne bedømme dette bliver man nødt til at have styr på definitionen af et rødsort træ, denne kan findes på slide 17 i [2].

For at opsummere så skal et rødsort træ overholde følgende:

- Træet skal overholde reglerne for et binært søgetræ
- Hver knude skal være enten rød eller sort
- Roden og bladene skal være sorte
- Der må ikke være 2 røde knuder i streg på en rod-blad sti (dvs. som er forbundet)
- Der skal være samme antal sorte knuder på alle rod-blad stier

Nu kan vi så begynde at kigge på om de 8 træer overholder denne definition ved slavisk at gennemgå dem for hvert træ.

For at finde ud af om t_1 er et binært søgetræ kan vi først tjekke om t_1 overholder reglerne for et binært søgetræ. Det kan vi gøre ved at kigge på roden og se om dens børn er inorder (man kan finde definitionen på inorder på slide 6 i [2]). Hvis man gør det kan man dog hurtigt se at 2 står til venstre for roden med nøgle 1, hvilket ikke er tilladt da nøgler i knudens venstre undertræ ikke må være større end knudens egen nøgle. **t_1 er derfor ikke et rødsort træ.**

Hvis man bruger samme metode på t_2 ser man at knuder alle er i inorder. Det næste krav til træet er så at alle blade skal være enten røde eller sorte, hvilket vi trivielt kan se gælder for alle træer. Så bliver vi nødt til at tjekke om roden og bladene er sorte i t_2, dog kan man observere at roden er rød og **t_2 er derfor heller ikke et rød-sort træ.**

Så er vi kommet til t_3, hvor vi hurtigt kan se at knuderne er i inorder samt at roden og bladene er sorte. Vi kan også observere at der ikke er 2 røde knuder som er forbundet, så mangler vi kunne at tjekke om der er det samme antal sorte knuder på alle rod-blad stier. Da dette ikke er et særligt stort træ kan vi bare tjekke alle stier fra roden til bladene manuelt, og vi kan verificere at alle stier har det samme antal sorte. **t_3 er derfor et rødsort træ**.

Samme process går vi igennem for **t_4 og vi kan verificere at dette også er et rødsort træ.**

Herefter er vi nået til t_5, hvor man kan se at der forekommer 2 røde i streg, og det betyder at **t_5 ikke er et rødsort træ**.

Hvis man sammenligner t_5 med t_6 kan man se at den eneste forskel er at knuden med nøglen 4 er blevet sort i stedet for rød i t_6. Dette løser problemet fra t_5, men det introducere et nyt problem da der nu ikke er lige mange sorte i alle stier fra roden til bladene. **t_6 er derfor heller ikke et rødsort træ.**

I t_6 var problemet at der var for mange sorte i rodens højre undertræ i forhold til det venstre, det kunne man prøve at løse ved at gøre knuden med nøglen 1 sort, hvilket ser ud til at være det der er sket i t_7 (knuder med nøglerne 3 og 4 er også byttet rundt, men det ændrer ikke noget). Der er dog stadig en sti fra roden til et blad med kun 1 sort, hvor de andre har 2. **t_7 er derfor stadig ikke et rød-sort træ.**

Til sidst er der i t_8 blevet byttet rundt på farverne på knuderne med nøglerne 3 og 4, hvilket endeligt ser ud til at have løst alle problemer og man kan ved at tjekke om træet overholder alle definitioner verificere at **t_8 er et rødsort træ**.

Spørgsmål b

I denne del opgave skal vi indsætte en knude med nøglen 3 i det givne træ og så finde det træ blandt 10 forskellige træer.

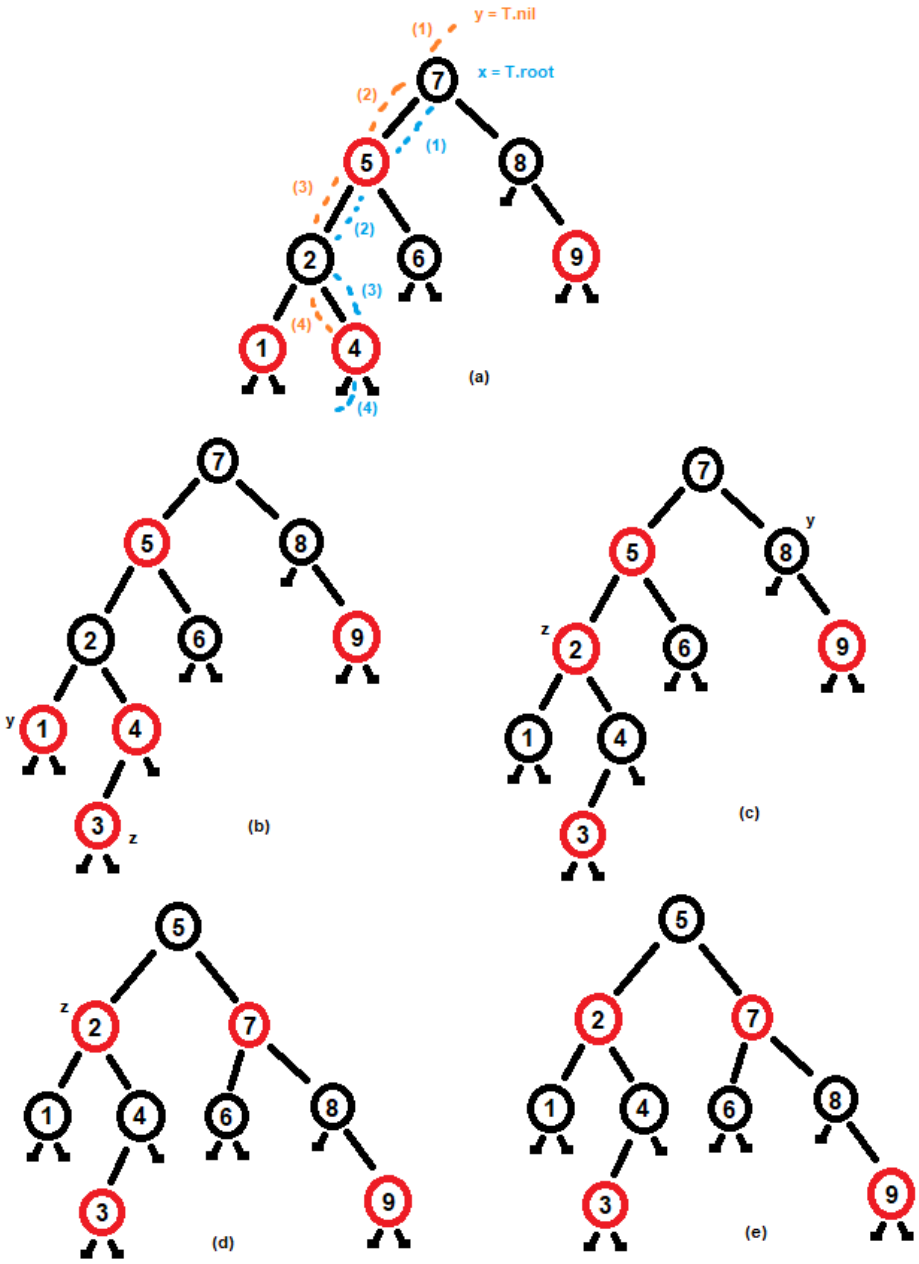
Den generelle strategi for at indsætte en ny knude i et rødsort træ er følgende:

1. Først indsættes knuden som i et normalt binært søgetræ
2. Herefter fjernes evt. opstået ubalance (overtrædelse af rød-sort strukturkravene)

Dette kan findes på slides 20-25 i [2]. En illustration som viser de forskellige trin kan ses i figuren på side 10. Når vi indsætter den nye knude som normalt laver man mere eller mindre binary tree search ned igennem træet indtil man finder en fri plads i træet. Gør man det i det givne træ med den nye knude med nøgle 3 ser man følgende sekvens 7, 5, 2, 4 hvorefter knuden bliver sat ind som venstre barn til knuden med nøglen 4. Det er her vigtigt, at man husker, at knuden skal farves rød ved efter man sat den på den rigtige plads.

Nu skal eventuelle overtrædelser af rød-sort strukturkravene fjernes, og siden vi har farvet den nye knude rød er der nu 2 rød knuder i streg på en blad-rod sti. Dette kan vi løse ved bytte om på nogle af farverne, som man også kan se på slide 25 i [2], da vi er i et tilfælde af case 1 eftersom at knuden med nøglen 4's søskende er rød (knuden med nøglen 1). Derfor bliver knuderne med nøglerne 1 og 4 sorte, mens at knuden med nøglen 2 bliver rød. Det resultere dog i at vi får en ny rød-rød overtrædelse mellem knuderne med nøglerne 2 og 5. Denne gang er vi i case 2, da knuden med nøglen 5 har en sort søskende, og vi bliver derfor nødt til at lave en højre rotation på 7 (kigger man på slide 25 på billedet i case 2 skal vi lave transformation fra hvor der står case 3). Det vil sige at vi flytter knuden med nøglen 5 op som roden og sætter knuden med nøglen 7 som dens højre barn, 5's gamle højre barn bliver nu 7's nye venstre barn (før var det 5). Til sidst skal farvene ændres sådan at knuden med nøglen 5 bliver sort og knuden med nøglen 7 bliver rød.

For at opsummere: Vi skal altså finde et træ med 5 som roden, hvor 1 og 4 er blevet sorte, men ellers har beholdt deres plads som børn af 2. **Det vil sige at t_7 er det rigtige svar.**



Figur 4: Illustration af de trin man skal igennem for at indsætte en ny knude med nøglen 3 i træet i opgave 3-b. (a) Vi laver en binary tree search ned igennem træet for at finde plads til 3. (b) Her er 3 sat ind i træet, læg mærke til rød-rød overtrædelsen. (c) Efter at have løst overtrædelse ved at bruge case 1 er der opstået en ny rød-rød overtrædelse. (d) Træet efter højre rotationen på 7. (e) Det færdige træ efter indsættelsen af 3

4 Opgave 4 - Eksamen juni 2013, opgave 5

Spørgsmål a

Ved sortering af n heltal med værdier mellem 0 og n^4 kigger vi på asymptotisk worst-case køretider for forskellige sorteringsalgoritmer.

1. **Counting sort:** På side 196 i [1] er der en analyse af counting sort, hvor det kan ses, at hvis man sorterer n heltal, som alle har værdi i intervallet $[0, k]$, så er køretiden $\Theta(n + k)$. Da vi her sorterer n heltal med værdier i intervallet $[0, n^4]$, så vil køretiden være $\Theta(n + n^4) = \Theta(n^4)$.
2. **Radix sort:** Her følger vi samme argumentation som vi brugte i opgave 9 fra ugeseddel 10. Vi kan repræsentere heltal i rækkevidden fra 0 til $(n^4 - 1)$ som 4 cifre i base n. Nu kan vi bruge radix sort med counting sort til at sortere de individuelle cifre, hvor de 4 cifre er i intervallet $[0, n - 1]$. Bruges Lemma 8.3 fra [1, p.198], så får vi køretiden til at være $\Theta(d(n + k)) = \Theta(4(n + (n - 1))) = \Theta(n)$.
3. **Quicksort:** Worst-case køretiden for Quicksort opstår f.eks. når stigende sorteret, faldende sorteret eller identiske elementer gives som input. I disse tilfælde vil køretiden være: $\Theta(n^2)$.
4. **Mergesort:** Mergesort laver altid samme arbejde og ender med køretid på $\Theta(n \cdot \log_2(n))$.
5. **Insertionsort:** Insertionsorts worst-case er faldende sorteret input som giver os en køretid på $\Theta(n^2)$ da alle tal skal sammenlignes med alle andre tal.

Spørgsmål b

CountingSort sortere listen: $A = (7, 4, 1, 2, 6, 4, 0, 4, 4, 4, 7, 2)$.
Skridt 1: C arrayet starter med at blive sat til 0 på alle index.

$$C = (0, 0, 0, 0, 0, 0, 0, 0)$$

Skridt 2: A gennemløbes og index i C tælles op undervejs.

$$C = (1, 1, 2, 0, 5, 0, 1, 2)$$

Skridt 3: index i C summes op fra venstre mod højre:

$$C = (1, 2, 4, 4, 9, 9, 10, 12)$$

Skridt 4: A gennemløbes bagfra og tallene skrives i den rigtige rækkefølge, index fra C tælles ned undervejs.

$$C = (0, 1, 2, 4, 4, 9, 9, 10)$$

5 Opgave 5 - Implementering af counting sort

Nedenunder findes der to gennemgange for en implementation af counting sort - én for Python og én for Java. Da Python og Java udgaverne er lavet på samme måde (dog sammenligner Python udgaven flere algoritmer til sidst), så giver det nok mest mening bare

at læse den ene. Nedenstående gennemgange går desuden mere i detaljen end hvad man normalt ville gøre i f.eks. en rapport. Dette er gjort for jeres skyld, da kurset henvender sig til mange studerende med forskellige grader af programmeringserfaring. Hvis man føler sig godt rustet i programmingskunsten, så kan man uden problemer læse (og køre) koden selv uden at læse (hele) gennemgangen nedenunder. Selvom nedenstående har mange detaljer, så er vi instruktorer mere vant til programmering og kan nemt glemme detaljer, som vi selv havde svært ved, da vi selv havde kurset - derfor må man meget gerne skrive spørgsmål til os inde på BlackBoard (se Spørgsmål 6).

5.1 Python

5.1.1 Fra pseudokode til Python implementation

Vi implementerer counting sort ud fra bogens pseudokode på s. 195:

```
COUNTING-SORT(A, B, k)
 1  let C[0..k] be a new array
 2  for i = 0 to k
 3      C[i] = 0
 4  for j = 1 to A.length
 5      C[A[j]] = C[A[j]] + 1
 6  // C[i] now contains the number of elements equal to i.
 7  for i = 1 to k
 8      C[i] = C[i] + C[i − 1]
 9  // C[i] now contains the number of elements less than or equal to i.
10  for j = A.length downto 1
11      B[C[A[j]]] = A[j]
12      C[A[j]] = C[A[j]] − 1
```

Som altid skal vi være opmærksomme på, at bogens arrays er indekseret startende fra 1 (hvis ikke andet er skrevet), mens Python indekserer lister startende fra 0. Vi skal dog være ekstra opmærksomme i lige præcis denne algoritme, da vi kan se at C arrayet (som vi bl.a. bruger til at holde styr på, hvor mange af hvert tal fra 0 til k der i input array) er indekseret fra 0. Vi går nu gennem algoritmen skridt for skridt, og ser hvordan vi kan lave pseudokoden om til en konkret Python implementation (vi bruger "array" til at benævne både arrays fra pseudokoden og lister fra Python). Algoritmen er implementeret i funktionen `counting_sort_internal(A, B, k)`:

Linje 1-3: her opretter vi et array C med pladserne fra 0 til k, og bagefter skriver vi 0 på alle pladserne. En nem måde at gøre dette på er ved:

```
C = [0] * (k+1)
```

dette giver os et array af størrelse k+1, hvor der står 0 på alle pladser (husk: dette betyder, at det første indeks i C er 0, og det sidste er k).

Linje 4-5: her går vi gennem A og hver gang vi ser et tal A[j] i A opdaterer vi plads A[j] i C, så vi får noteret at vi har set A[j] en gang mere. Her skal vi huske at A er indekseret fra

1 i pseudokoden. Da A i Python er indekseret fra 0 skal vi starte fra 0 i stedet i for-loop, og slutte ved A.length-1. Vi husker at C i pseudokoden er indekseret fra 0, så her behøver vi ikke ændre noget:

```
for j in range(len(A)):
    C[A[j]] = C[A[j]] + 1
```

Vi husker også, at `range(len(A))` giver os tallene fra 0 til len(A)-1 (`range` funktionen er eksklusiv det tal, vi giver som argument).

Linje 7-8: her ændrer vi C arrayet fra at C[i] indeholder hvor mange gange tallet `i` optræder i A, til at indeholde hvor mange tal der har værdi $\leq$ `i` i arrayet A. Da C både i pseudokode og implementation er indekseret fra 0, skal der ikke ændres på noget (udover at vi skal bruge `range(k+1)` fordi vi gerne vil have, at for-loop løber op til og med k):

```
for i in range(1,k+1):
    C[i] = C[i] + C[i-1]
```

Linje 10-12: vi har nu at C[i] indeholder hvor mange tal der har værdi $\leq$ `i` i arrayet A. Vi går nu baglæns gennem A, dvs. vi vil gerne have at vores variabel `j` går fra A.length-1 til 0 (da vores A er indekseret fra 0). En måde at gøre dette på er at bruge funktionen `reversed`.

Når vi møder tallet A[j], så slår vi op i C på denne plads (dvs. vi ser på C[A[j]]), og ser hvor A[j] skal stå i B. Her skal vi være lidt forsigtige. Lad os f.eks. se på det største tal i A (lad os kalde det `x`). Hvis vi til at starte med ser på C[x], så får vi hvor mange tal der er højst `x`. Det antal må være det samme som længden af A. Dvs. hvis vi bare fulgte bogens pseudokode, ville vi prøve på at skrive `x` på plads B[A.length], men siden B i vores implementation er indekseret fra 0, så er den sidste plads i B plads A.length-1, så det giver slet ikke mening at prøve på at skrive til plads A.length i B. Vi skal derfor huske at trække 1 fra det indeks vi prøver på at skrive til i B:

```
for j in reversed(range(len(A))):
    B[C[A[j]] - 1] = A[j]
    C[A[j]] = C[A[j]] - 1
```

5.1.2 Wrapper

(Følgende er én måde at designe interfacet til funktionen på. Der er også andre holdninger omkring om man skal lave en "wrapper" som vi gør her, eller lade være. Det afhænger også af, hvad funktionen skal bruges til. I vores tilfælde skal den sammenlignes med andre sorteringsalgoritmer, så her giver det mening at lave et "uniformt" interface som er det samme for alle algoritmerne).

Funktionen vi lige har beskrevet er i koden implementeret som
`counting_sort_internal(A, B, k)`. Grunden til dette er, at det er meget rart for andre der skal bruge vores funktion, at de ikke skal holde styr på et A, B og k som de skal give til vores funktion, men i stedet kan bruge funktionen som de ville bruge enhver anden sorteringsfunktion, dvs. give ét array med, og efter funktionskaldet er tallene i dette array sat i sorteret rækkefølge. (Man skal dog vide at ens array ikke kan indeholde negative tal).

Dvs. vi vil gerne give andre mulighed for at kalde en funktion `counting_sort(A)`. Vi gør dette ved at implementere denne funktion, hvor det eneste den funktion gør er at kalde den interne version vi lige har implementeret:

```
def counting_sort(A):
    counting_sort_internal(A.copy(), A, max(A) if len(A) > 0 else 0)
```

Siden vi gerne vil have at vi sorterer tallene i A, så giver vi en kopi af A med til den interne version som argumentet A, og som argumentet B giver vi det originale A vi har fået ind, så vi faktisk ender med at skrive det sorterede array i A igen. For at finde k bruger vi max(A) (og hvis A er tomt, giver vi 0 med). Alt dette koster lidt ekstra, men asymptotisk ændrer vi ikke på køretiden for algoritmen.

5.1.3 Test og tidtagning

(De forskellige tests bliver kaldt fra bunden af koden, og man skal udkommentere/afkommentere noget af koden for at køre den korrekte test.)

Vi tester vores funktion med en række forskellige input (se kode). Vi skal her huske på, at vi skal sørge for, at det input array, A, som vi giver med, kun må indeholde ikke-negative heltal. I tidtagning prøver vi med en række forskellige størrelser af A, og for hver størrelse n, tester vi for tre forskellige værdier af `k`. Efter gennemsnittet af 3 tidtagninger er fundet, dividerer vi gennemsnitstiden med $(n+k)$. Vi observerer, at de fremkomne tal er nogenlunde konstante, dvs. det tyder på, at vores analyse af køretiden som værende $\Theta(n+k)$ er korrekt.

5.1.4 Sammenligning med andre sorteringsalgoritmer

Vi skal også sammenligne køretiden for counting sort med de andre algoritmer vi har implementeret. Vi sammenligner også med den indbyggede sorteringsalgoritme i Python. Se i koden for, hvordan dette gøres. Konklusionen på test er (vi bruger $n = 50000$), at hvis $k = n$ eller $k = \frac{n}{50}$ så er counting sort hurtigere end de andre algoritmer vi har implementeret, og kun lidt langsommere end den indbyggede sorteringsalgoritme. For $k = 50n$ er det den langsommeste algoritme, hvilket også er hvad vi forventer, da 50 er væsentligt større end $\log(n) = \log_2(50000) = 15.6$, dvs. køretiden for counting sort (lidt handwavy) er $50n$, mens køretiden for f.eks. mergesort er $15.6n$ (giver kun mening at kigge på dette for intuitionens skyld; der er andre konstanter ganget på og lagt til hvis man laver en præcis analyse af algoritmerne, og i praksis er der også forskel på implementationen og hvordan hukommelse bliver brugt).

5.1.5 Til den nysgerrige studerende

Hvis man har kørt testen som sammenligner køretiderne for de algoritmer vi har implementeret og den sorteringsalgoritme der er indbygget i Python, så vil man se, at den indbyggede i Python er væsentligt hurtigere end vores algoritmer (her snakker vi en faktor 10 hurtigere). Hvorfor sker dette? Er det fordi den sorteringsalgoritme Python bruger er så optimeret, og godt implementeret, at vores algoritmer er håbløst bagud? Både ja og nej, men mest nej. Det er muligt at Pythons sorteringsalgoritme faktisk er hurtigere end vores, men den største årsag til at Pythons sortering er hurtigere end vores er pga. den måde Python fungerer på:

(En smule simplificeret) Python er et interpreted sprog, dvs. når vores Python program kører, så sker der i virkeligheden det, at der er et andet program som læser vores program én linje ad gangen, og så finder den ud af, hvad den skal bede computeren om at gøre, for at der sker det vi har skrevet i vores kode. Dvs. der er faktisk en "mellemmand" mellem vores kode og computeren, og dette gør at der er en del "overhead", altså ekstra tid der bliver brugt på dette. Men for en række funktioner der er indbygget i Python, så er disse funktioner faktisk slet ikke skrevet i Python, men de er en indbygget del af det program som læser vores Python kode, og når den indbyggede funktion kører, så skal den ikke læses af et andet program som skal bede computeren om at gøre noget, men funktionen fortæller selv direkte til computeren, hvad den skal gøre. I den indbyggede Python sorteringsfunktion er denne "mellemmand" altså skåret væk, og det er dét som gør at den indbyggede sorteringsalgoritme er så meget hurtigere end vores sorteringsalgoritmer.

Man kan altså sige at det på ingen måde er en fair sammenligning vi laver, når vi sammenligner køretiderne for vores implementeringer og Pythons sortering. En mere fair sammenligning af algoritmerne ville være hvis man så på algoritmen Pythons sortering bruger, og selv implementerede den i Python kode, og så lavede sammenligningen med denne implementering.

5.2 Java

5.2.1 Fra pseudokode til Java implementation

Vi implementerer counting sort ud fra bogens pseudokode [1, s. 195]:

```
COUNTING-SORT(A, B, k)
 1  let C[0..k] be a new array
 2  for i = 0 to k
 3      C[i] = 0
 4  for j = 1 to A.length
 5      C[A[j]] = C[A[j]] + 1
 6  // C[i] now contains the number of elements equal to i.
 7  for i = 1 to k
 8      C[i] = C[i] + C[i − 1]
 9  // C[i] now contains the number of elements less than or equal to i.
10  for j = A.length downto 1
11      B[C[A[j]]] = A[j]
12      C[A[j]] = C[A[j]] − 1
```

Som altid skal vi være opmærksomme på, at bogens arrays er indekseret startende fra 1 (hvis ikke andet er skrevet), mens Java indekserer arrays startende fra 0. Vi skal dog være ekstra opmærksomme i lige præcis denne algoritme, da vi kan se at C arrayet (som vi bl.a. bruger til at holde styr på, hvor mange af hvert tal fra 0 til k der i input array) er indekseret fra 0. Vi går nu gennem algoritmen skridt for skridt, og ser hvordan vi kan lave pseudokoden om til en konkret Java implementation. Algoritmen er implementeret i metoden `countingSortInternal(int[] a, int[] b, int k)`:

Linje 1: her opretter vi et array `c` med pladserne fra 0 til k.

```
int[] c = new int[k+1];
```

Pseudokoden overskriver derefter alle pladserne med 0, men dette er ikke nødvendigt i Java, da den selv sørger for dette (se her). Vær opmærksom på, at det første indeks i `c` er 0, og det sidste er `k`.

Linje 2-3: her går vi gennem `a` og hver gang vi ser et tal `a[i]` i `a` opdaterer vi plads `a[i]` i `c`, så vi får noteret at vi har set `a[i]` en gang mere. Her skal vi huske at A er indekseret fra 1 i pseudokoden. Da `a` i Java er indekseret fra 0 skal vi starte fra 0 i stedet i for-loop, og slutte ved `a.length-1`. Vi husker at C i pseudokoden er indekseret fra 0, så her behøver vi ikke ændre noget, da `c` i Java er ligeså:

```
for (int i = 0; i < a.length; i++)
    c[a[i]] += 1;
```

Linje 4-5: her ændrer vi `c` arrayet fra at `c[i]` indeholder hvor mange gange tallet `i` optræder i `a` til at indeholde hvor mange tal, der har værdi $\leq$ `i` i array `a`. Da C i pseudokoden og `c` i implementation er indekseret fra 0, skal der ikke ændres på noget.

```
for (int i = 1; i < c.length; i++)
    c[i] = c[i] + c[i-1];
```

Linje 6-9: vi har nu at `c[i]` indeholder hvor mange tal der har værdi $\leq$ `i` i arrayet `a`. Vi går nu baglæns gennem `a`, dvs. vi vil gerne have at vores variabel `i` går fra `a.length-1` til 0 (husk array `a` er indekseret fra 0).

Når vi møder tallet `a[i]`, så slår vi op i `c` på denne plads (dvs. vi ser på `c[a[i]]`), og ser hvor `a[i]` skal stå i `b`. Her skal vi være lidt forsigtige. Lad os f.eks. se på det største tal i `a` (lad os kalde det x). Hvis vi til at starte med ser på `c`[x], så får vi hvor mange tal der er højst x. Det antal må være det samme som længden af `a`. Dvs. hvis vi bare fulgte bogens pseudokode, ville vi prøve på at skrive x på plads `b[a.length]`, men siden `b` i vores implementation er indekseret fra 0, så er den sidste plads i `b` plads `a.length-1`, så det giver slet ikke mening at prøve på at skrive til plads `a.length` i `b`. Vi skal derfor huske at trække 1 fra det indeks vi prøver på at skrive til i `b`:

```
for (int i = a.length - 1; i >= 0; i--) {
    b[c[a[i]]-1] = a[i];
    c[a[i]]--;
}
```

5.2.2 Wrapper

(Følgende er én måde at designe interfacet til metoden på. Der er også andre holdninger omkring om man skal lave en "wrapper" som vi gør her, eller lade være. Det afhænger også af, hvad metoden skal bruges til. I vores tilfælde skal den sammenlignes med andre sorteringsalgoritmer, så her giver det mening at lave et "uniformt" interface som er det samme for alle algoritmerne).

Metoden vi lige har beskrevet er i koden implementeret som

`countingSortInternal(int[] a, int[] b, int k)`

Grunden til dette er, at det er meget rart for andre der skal bruge vores metode, at de ikke skal holde styr på et `a`, `b` og `k` som de skal give til vores metode, men i stedet kan bruge metoden som de ville bruge enhver anden sorteringsmetode, dvs. give ét array med, og efter metodekaldet er tallene i dette array sat i sorteret rækkefølge - man skal dog vide at ens array ikke kan indeholde negative tal. Dvs. vi vil gerne give andre mulighed for at kalde en metode `countingSort(v)`. Vi gør dette ved at implementere denne metode, hvor det eneste den metode gør er at kalde den interne version vi lige har implementeret:

```
private static void countingSort (int[] v) {
       if (v.length > 0) { // necessary to find max safely.
           countingSortInternal(Arrays.copyOf(v, v.length), v, Arrays.stream(v).max().
                getAsInt());
       }
   }
```

Siden vi gerne vil have, at vi sorterer tallene i `v`, så giver vi en kopi af `v` med til den interne version som argumentet `a`, og som argumentet `b` giver vi det originale `v` vi har fået, så vi faktisk ender med at skrive det sorterede array i `v` igen. For at finde det største element i k konverteres `v` til en `IntStream` og metoden `max` bruges. For folk der ikke følger DM563 svarer dette til at lave en lineær søgning gennem `v` for at finde det største element. Alt dette koster lidt ekstra, men asymptotisk ændrer vi ikke på køretiden for algoritmen.

5.2.3 Test og tidtagning

Vi tester vores metode med en række forskellige input (se kode). Vi skal her huske på, at vi skal sørge for, at det input array som vi giver med, kun må indeholde ikke-negative heltal. I tidtagning prøver vi med en række forskellige størrelser af input arrayet, og for hver størrelse n, tester vi for tre forskellige værdier af `k`. Efter gennemsnittet af 3 tidtagninger er fundet, dividerer vi gennemsnitstiden med $(n+k)$. Vi observerer, at de fremkomne tal er nogenlunde konstante, dvs. det tyder på, at vores analyse af køretiden som værende $\Theta(n+k)$ er korrekt.

5.2.4 Sammenligning med andre sorteringsalgoritmer

Vi sammenligner køretiden for counting sort med quicksort, som vi tidligere har implementeret. Se i koden for, hvordan dette gøres. Konklusionen på test er (vi bruger $n = 8000000$), at hvis $k = \frac{n}{50}$ så er counting sort hurtigere end quicksort. For $k = n$ er de nogenlunde ens. For $k = 50n$ er den langsommere end quicksort, hvilket også er hvad vi forventer, da 50 er væsentligt større end $\log_2(n) = \log_2(8000000) \approx 22.9$, dvs. køretiden for counting sort (lidt handwavy) er $50n$, mens køretiden for quicksort er $22.9n$ (giver kun mening at kigge på dette for intuitionens skyld; der er andre konstanter ganget på og lagt til hvis man laver en præcis analyse af algoritmerne, og i praksis er der også forskel på implementationen og hvordan hukommelse bliver brugt).

Vær opmærksom på, at programmet bruger en del hukommelse, da der laves mange store arrays. Når man udfører counting sort på et array af størrelse $n = 8 \cdot 10^6$, så fylder array A og B 32 MB. Derudover laver counting sort et C array, som for $k = 50n$ vil have en størrelse på ca. 1.6 GB. Derfor kan man få en `OutOfMemoryError` exception. Man burde kunne fikse dette ved at køre programmet på følgende måde: `java -Xmx4g CountingSort`

5.2.5 Til den nysgerrige studerende

En nysgerrig studerende ville måske undre sig over hvorfor counting sort ikke er klart bedre end quicksort når $k = n$. Hvis man endda vælger et mindre n, f.eks. $n = 500000$, så ville man nok observere dette (da vi kørte testen med $n = 500000$ tog counting sort 8 ms og quicksort 39 ms i gennemsnit) - er meningen med asymptotisk notation ikke at se på hvordan algoritmer udvikler sig når n bliver tilstrækkeligt stor? Jo, men computere er komplekse. Nogle mulige forklaringer kan være:

- Der er noget som hedder *locality of reference*. Det handler kort om, at man typisk gentagne gange tilgår samme områder i hukommelsen lige efter hinanden. Pga. den måde hukommelseshierarkiet er skruet sammen på, så er der store fordele at hente, hvis et program har en høj *locality of reference*. Counting sort hopper meget rundt i hukommelsen, og derfor kan den blive straffet ift. performance.
- Når $k = \frac{1}{50}n$, så er der en større sandsynlighed for at der optræder mange af de samme elementer i random input. Vi har tidligere set dette kan give mindre gode "splits"efter en partition. Når $k = n$, så falder sandsynligheden for mange ens tal. Derfor bliver quicksort på en måde straffet når $k = \frac{1}{50} \cdot n$, men dette sker i mindre grad når $k = n$, og kan være endnu en mulig forklaring på, hvorfor køretiderne ligger tæt på hinanden.

Pointen er, at der er mange ting, som spiller ind på køretiden i praksis - f.eks. brugen af hukommelsen. En typisk fremgangsmåde vil derfor være følgende: benytte asymptotisk analyse til en grovsortering af forskellige algoritmer, og derefter implementere dem for at kigge på hvordan de kører i praksis.

6 Opgave 6 - Spørgsmål fra tidligere ugesedler

Du kan stille spørgsmål til instruktorer ved at benytte det forum der er oprettet inde på BlackBoard. Det kan findes på følgende måde:

DM507 siden på BlackBoard → Course Materials → Discussion Board

Derudover kan du også sende os en mail (se *her*). Vi vil dog foreslå, at folk primært benytter BlackBoard, da alle dermed får glæde af det - man kan skrive anonymt.

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.

[3] Rolf Fagerberg. Sortering. URL `https://imada.sdu.dk/~rolf/Edu/DM507/F20/sortingSlides.pdf`, 2020.