DM507 — Algoritmer og datastrukturer

Eksaminatorie-timer uge 10 # 1, Forår 2021

Løsninger

Del A

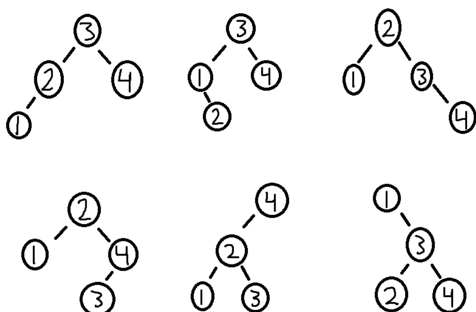
Opgave 1 - Eksamen juni 2008, opgave 4 b

Tegn alle mulige binære søgetræer, som har højde 2 og indeholder fire knuder med nøglerne 1, 2, 3 og 4.

Følgende observationer kan hjælpe:

- Man kan ikke have en kæde pga. højden ≤ 2.
- Hvis 1 eller 4 skal være rod, så skal hhv. højre og venstre undertræ være et fuldt binært træ af højde 1.
- Rotationer: Ligesom med Rød-Sorte træer, så laver rotationer ikke om på *in-order*.

Alle de mulige binære søgetræer er



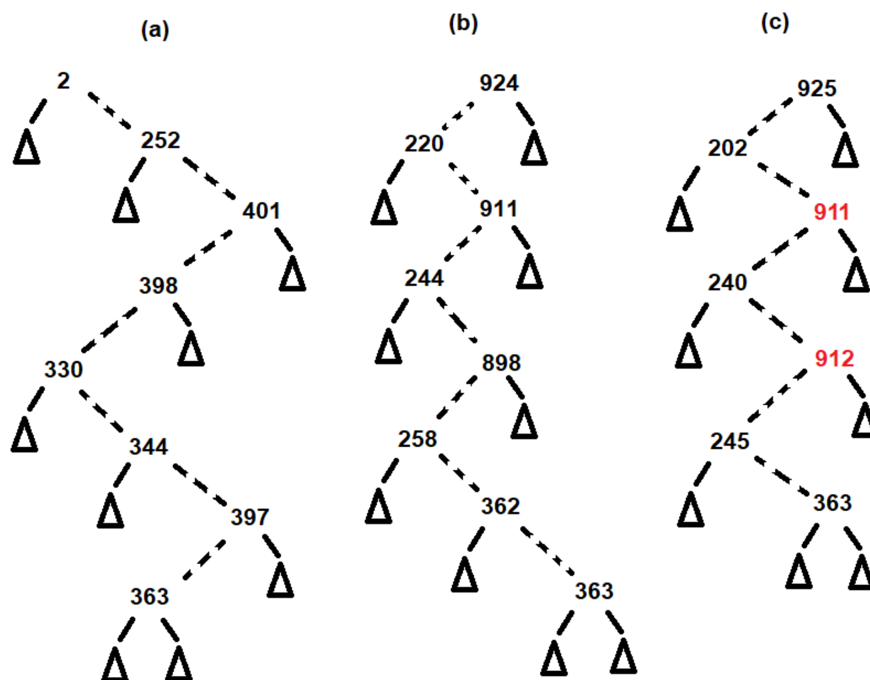
Opgave 2 - Cormen et al. øvelse 12.2-1, a–c

Antag vi har tallene mellem 1 og 1000 i et binært søgetræ, og vi søger efter tallet 363. Hvilken af de følgende følger vil ikke være en mulig følge af tal undersøgt i søgningen.
Fremgangsmåde: Fra en given knude x, hvis søgevej

- ...går til venstre, så er alle efterfølgende knuder undersøgt $\leq$ `x.key`
- ...går til højre, så er alle efterfølgende knuder undersøgt $\geq$ `x.key`

Det gælder af et BST's knuder er i *in-order*. Når vi søger efter *key*, så går vi til venstre ved x hvis $key \leq x.key$ eller til højre hvis $key \geq x.key$.
Herved fås

- $2, 252, 401, 398, 330, 344, 397, 363$: Mulig
- $924, 220, 911, 244, 898, 258, 362, 363$: Mulig
- $925, 202, 911, 240, 912, 245, 363$: Umulig, da vi undersøger en knude med key 911 og går til venstre – alle efterfølgende knuder's key skal være ≤ 911. Problem: senere undersøges en knude med key 912.

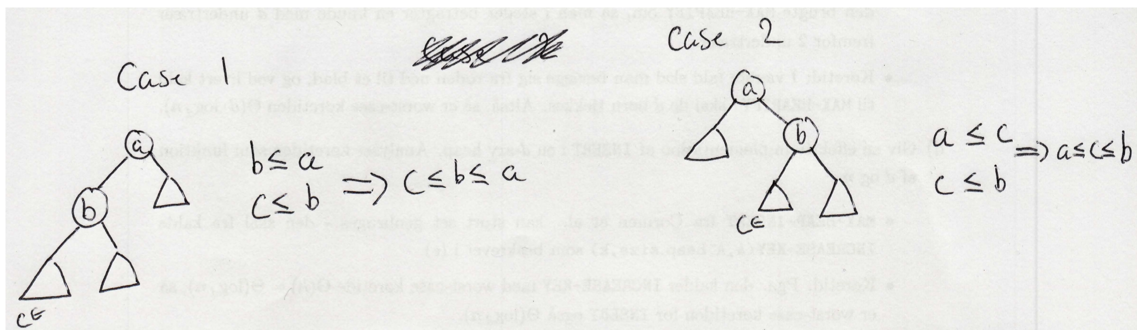


Opgave 3 - Cormen et al. øvelse 12.2-3

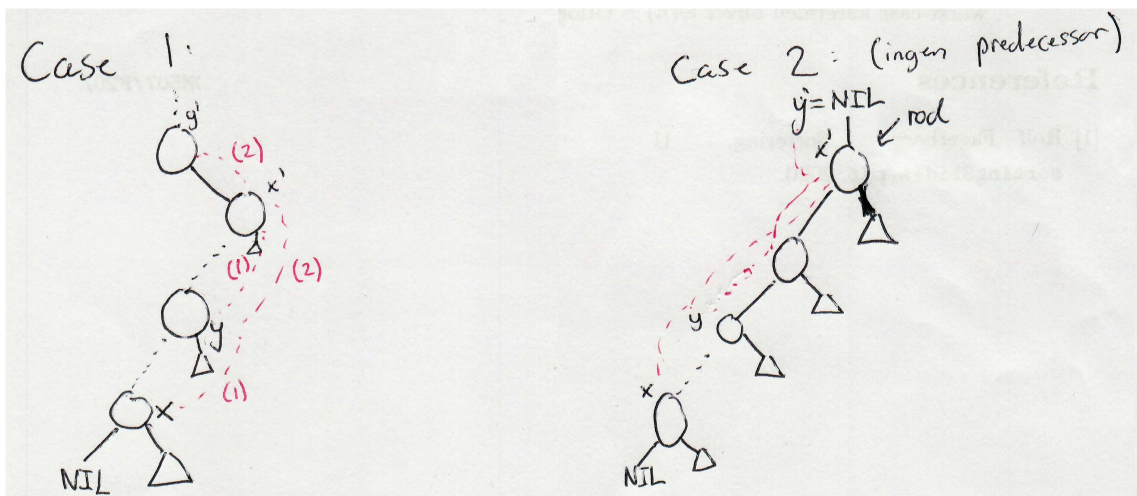
Skriv `Tree-Predecessor` proceduren.

Ideen er følgende, når vi ønsker at finde predecessor for en knude x:

- Venstre undertræ: Eksisterer dette, så må predecessor være knuden med den højste key i venstre undertræ (alle knuder i venstre undertræ har key mindre end $x.key$).
  - Korrekthed: Predecessoren er ikke længere oppe i træet – uanset om x er et højre eller venstre barn.
  - Illustration: Vi ønsker at finde predecessor for b.



- Tomt venstre undertræ: Bevæg sig opad i træet vha. x's forfædre. Stop ved den forfader, som er en (højre) forældre til enten x selv eller en anden af x's forfædre.
  - Korrekthed/Princip: Se at på stien fra x til roden kan ingen side-træer indeholde det søgte element (pga. in-order). Mens x og y pointere bevæger sig opad gælder $x.key \leq y.key$. Når man stopper skyldes det enten ingen predecessor eksisterer ($y ==$ NIL) eller y er en (højre) forældre til x (altså; $x.key \geq y.key$).
  - Illustration: Vi ønsker at finde predecessor af x.




```
Tree-Predecessor(x)
  // If possible, find node with highest key in left subtree
  if x.left ≠ NIL
    return Tree-Maximum(x.left)

  // Move up towards root (ie. while x.key ≤ y.key), and
  // stop when y is NIL or y is (right) parent of x.
  y = x.p
  while y ≠ NIL and x == y.left
    x = y
    y = y.p
  return y
```

Opgave 4 - Cormen et al. øvelse 12.3-3

Vi kan sortere en given mængde af n tal ved først at bygge et BST med disse tal og så printe tallene ud vha. en in-order tree walk. Hvad er best-case og worst-case køretiden for disse sorteringsalgoritmer? Hvad hvis vi bruger Rød-Sorte træer?

Vi starter med at kigge på ubalancerede BST.

Worst case: Dette sker når vi indsætter i stigende rækkefølge; altså, $1, 2, 3, \ldots$. Efter n indsættelser kan træet højst have højde $n-1$. Worst case køretiden er

$$\overbrace{0}^{\text{pga. første indsættelse (tomt træ)}} + 1 + 2 + \cdots + \underbrace{(n-1)}_{\text{pga. sidste indsættelse}} = \Theta(n^2)$$

Best case: Dette sker når vi fylder et lag ad gangen[1]. Observér, at indsatte knuder forbliver i den indsatte dybde.

Det endelige træ vil have højde $O(\lg n)$, så køretiden er $O(n \lg n)$.

Et binært træ af højde h kan have højst $2^{h+1} - 1$ knuder. Observér, hvis man indsætter i dybde højst $\lg n - 2$, kan der højst indsættes $2^{(\lg(n)-2)+1} - 1 = \frac{1}{2} \cdot n - 1$ knuder. Altså, i dybde mindst $\lg n - 1$ vil mindst $\frac{n}{2}$ knuder indsættes, og heraf følger køretiden er $\Omega(n \lg n)$.

Køretiden er $\Theta(n \lg n)$.

Lad os kigge på Rød-sorte træer.

Worst case: Vi laver n indsættelser og hver indsættelse er $O(\lg n)$. Dvs. $O(n \lg n)$.

Best case: Når antallet af sorte på stierne er k, så er højden af Rød-sort træet højst $2(k-1)$. Dvs. der er højst $2^{(2(k-1)-1)+1} - 1 = 2^{2(k-1)} - 1$ knuder [2]

Hvis $k \leq (\lg n)/2$, så kan der højst indsættes $2^{((\lg n/2)-1)+1} = 2^{\lg n/2} = 2^{\lg n - 2} = n/4$ knuder. Dvs. for $3n/4$ knuder vil $k \geq (\lg n)/2$ og dermed vil højden (med bladene) være mindst $2((\lg n)/2 - 1) = \lg n - 2$. Heraf følger best case er $\Omega(n \lg n)$.

Eftersom best case er højst så dårlig som worst case, så er både best case og worst case $\Theta(n \lg n)$.

For at få tallene ud i sorteret orden, så skal vi udføre en in-order tree walk. Dette tager $\Theta(n)$ tid.

Opgave 5 - Cormen et al. øvelse 12.1-5

Argumenter for, at siden sortering af n elementer tager $\Omega(n \lg n)$ i worst-case i den sammenligningsbaserede model, så må enhver sammenligningsbaseret algoritme for at konstruere BST fra n tilfældige elementer tage $\Omega(n \lg n)$ i worst-case.

Hvordan sorterer man vha. BST? Fremgangsmåden er, at konstruere et BST og derefter lave et in-order gennemløb. Worst-case, så tager konstruering af BST **???** og in-order gennemløb tager

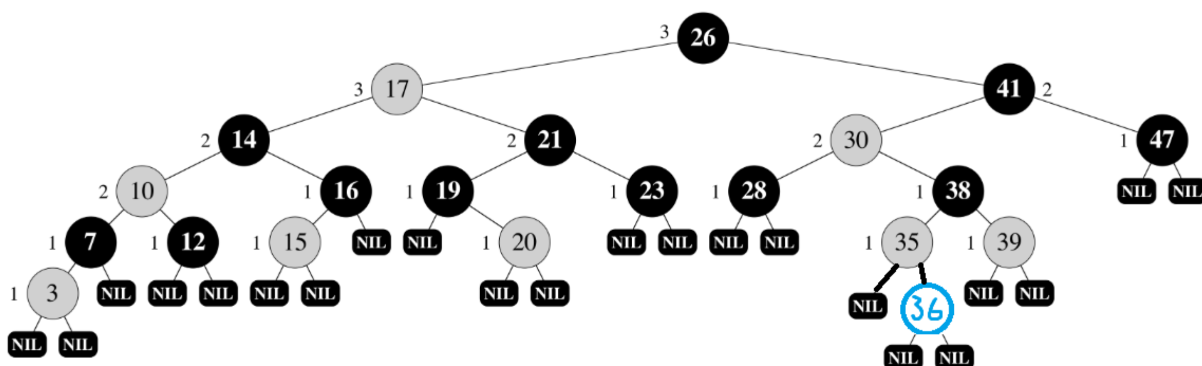
[1] Ligesom i en heap men uden heap-facon krav
[2] Vi bruger $2(k-1)-1$, da højden stiger med én pga. definitionen af blade ift. ubalancerede træer.

$\Theta(n)$. Hvad er køretiden for konstruering af BST (som er sammenligningsbaseret)? Jf. **Sætning 8.1** [1, p. 193], så er worst-case køretiden for en sammenligningsbaseret sorteringsalgoritme $\Omega(n \lg n)$. Derfor må konstruerings af BST i worst-case tage $\Omega(n \lg n)$, da man ellers vil kunne sortere hurtigere end $\Omega(n \lg n)$ i worst case (hvilket vil være i modstid med Sætning 8.1)

Opgave 6 - Cormen et al. øvelse 13.1-2

Tegn det Rød-Sorte træ efter `Tree-Insert` kaldes med træet på Figur 13.1 og key 36. Hvis den indsatte knude farves rød, er det resulterende træ et Rød-Sort træ? Hvad hvis den farves sort?

Følgende er resultatet efter insættelse af 36 vha. `Tree-Insert`:



Hvis knuden farves...

- ...rød, så er der to røde knuder i træk ⇒ ikke Rød-Sort træ
- ...sort, så har ikke alle rod-blad stiger samme antal sorte knuder ⇒ ikke Rød-Sort træ

Note: Problemet kan løses ved at bruge `RB-Insert(T, z)` fremfor `Tree-Insert([...])`, som er lavet til (almindelige) ubalancerede BST.

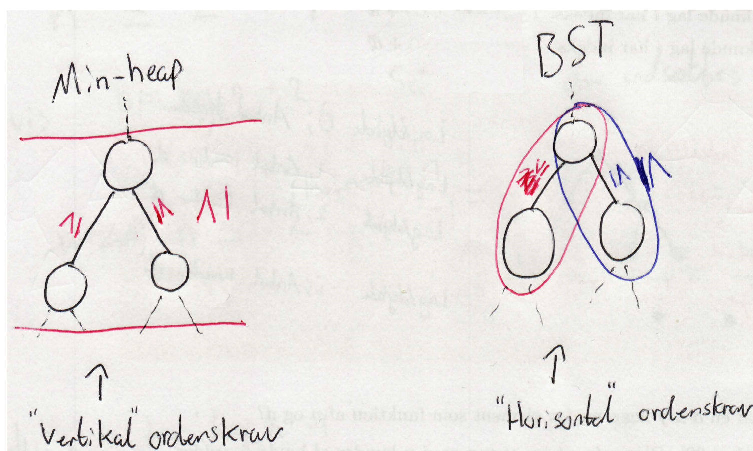
Del B

Opgave 1 - Cormen et al. øvelse 12.1-2

Hvad er forskellen på BST-egenskaben og min-heap-egenskaben? Kan min-heap egenskaben bruges til at printe nøglerne for et n-knude træ ud i sorteret orden i $O(n)$? Vis hvorfor eller hvorfor ikke.

Forskellen på BST-egenskaben og min-heap-egenskaben er:

- In-order/BST-egenskaben: nøgler i v's venstre undertræ $\leq$ nøgle i v $\leq$ nøgler i v's højre undertræ
- Min-heap-egenskaben: `Parent(`i`)'s nøgler` $\leq$ i's nøgler
- Illustration: *min-heap-egenskaben* giver "vertikal" orden og *in-order* giver "horisontal" orden.



Hvordan sorterer man vha. min-heap? Fremgangsmåde: (1) Konstruering af min-heap (2) Udskrivning fra min-heap i sorteret orden. Worst case, så tager konstruering af min-heap $O(n)$ og udskrivning i sorteret orden tager **???**. Hvad er køretiden for udskrivning fra min-heap i sorteret orden? Jf. **Sætning 8.1** [1, p. 193], så er worst-case køretiden for en sammenligningsbaseret sorteringsalgoritme $\Omega(n \lg n)$. Derfor må enhver sammenligningsbaseret algoritme, der udskriver fra en min-heap tage $\Omega(n \lg n)$ tid, da man ellers vil kunne sortere hurtigere end $\Omega(n \lg n)$ i worst case (hvilket vil være i modstid med Sætning 8.1).

Fortolkning: En forskel på min-heap-order og in-orden er

- In-order: Elementerne er essentielt allerede i total orden, når in-order er opnået.
- Min-heap-orden: Der er "en smule" orden i den forstand, at vi ved hvor det mindste element er, men elementerne er langt fra sorteret.

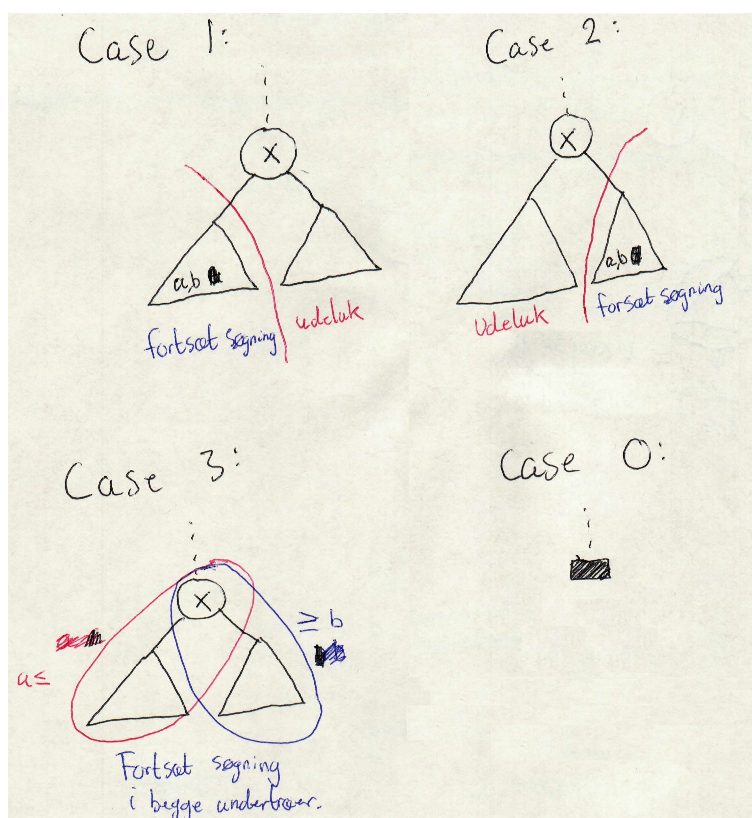
Opgave 2 - Cormen et al. øvelse 14.2-4

Beskriv en algoritme kaldet `RangeSearch(x, a, b)` som udskriver alle nøgler k, $a \leq k \leq b$, i RB-sort træet givet ved x. Beskriv hvordan denne kan implementeres i $\Theta(m + \lg n)$ tid, hvor m

er antallet af nøgler der udskrives og n er antallet af indre knuder i træet. (opgavebeskrivelsen er lidt anderledes i bogen)

Består af 4 cases:

- Case 0 (base case): x er et blad (`NIL` undetræ).
  - Action: return
- Case 1: a, b er $<$ `x.key`
  - Action: Søg videre i venstre undertræ; x og knuder i dets højre undertræ er ligegyldige.
- Case 2: a, b er $>$ `x.key`
  - Action: Søg videre i højre undertræ; x og knuder i dets venstre undertræ er ligegyldige.
- Case 3: $a \leq$ `x.key` $\leq b$
  - Action: (1) Søg rekursivt videre i venstre undertræ. (2) Udskriv `x.key` (3) Søg rekursivt videre i højre undertræ.



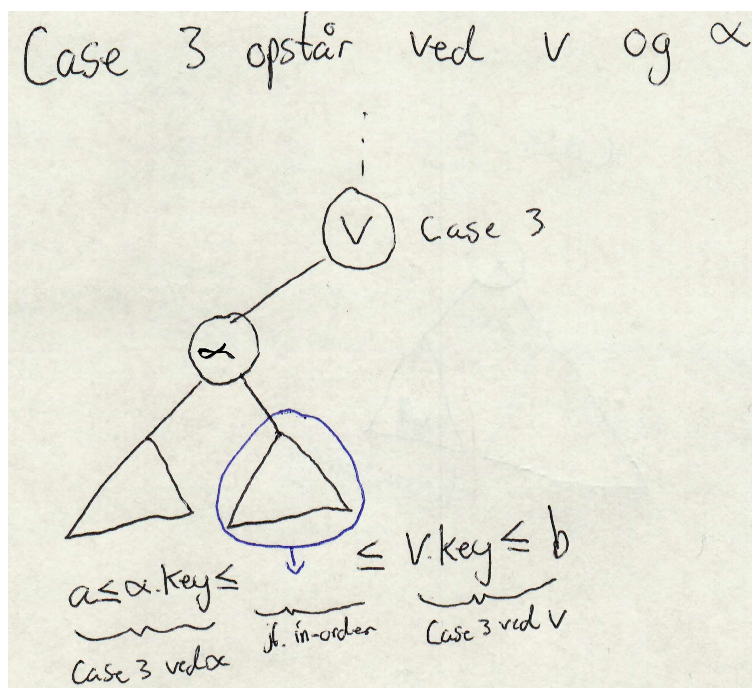
Køretid: Betragt det inducerede rekursionstræ R (dette er isomorft med den del af søgetræet, der gennemløbes). Køretidsanalyse trick: Lad $v \in R$ være første knude hvor Case 3 opstår; der søges rekursivt videre i begge undertræer.

- Venstre undertræ kaldes α:

- Case 0, Case 2 og Case 3 kan opstå, men ikke Case 1 da $b < \alpha$`.key` ville modstride at Case 3 opstod for v (alle nøgler i α's undertræ er $\leq$ `v.key`)
- Observation: opstår Case 3 for α, så skal hele α højre undertræ udskrives. Hvorfor? Da $a \leq \alpha$`.key` og v`.key` $\leq b$, så må alle knuder der ligger mellem α og v i in-order være mellem a og b; altså, dette svarer til α's højre undertræ. At gennemløbe denne del af træet "betales af" m (dvs. antallet af knuder/nøgler i $[a, b]$).

• Højre undertræ: Symmetrisk argument.

Observation: bortset fra de ydre stier til højre og venstre i rekusionstræet, så er resten "betalt" af output (m i $\Theta(m + \lg n)$). Da træet er balanceret, så er de to ydre stier $O(\lg n)$.



Korrekthed: Vi tjekker alle undertræer, som kan indeholde nøgler i intervallet, og udskriver en nøgle, hvis den er i intervallet.

References

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