Binary Search Trees

Algorithms and Datastructures, F25, Lecture 5

Andreas Holck Høeg-Petersen

Department of Computer Science
Aalborg University

March 13, 2025

Opdateringer

- Næste programmeringsopgave kommer til at handle om binære søgetræer (og måske lidt priority queues)

Opdateringer

- Næste programmeringsopgave kommer til at handle om binære søgetræer (og måske lidt priority queues)
- Den skal afleveres i starten af april (nærmere info senere)

Opdateringer

- Næste programmeringsopgave kommer til at handle om binære søgetræer (og måske lidt priority queues)
- Den skal afleveres i starten af april (nærmere info senere)
- Fra evaluering:

Opdateringer

- Næste programmeringsopgave kommer til at handle om binære søgetræer (og måske lidt priority queues)
- Den skal afleveres i starten af april (nærmere info senere)
- Fra evaluering:
  - Det 'dårlige':

Opdateringer

- Næste programmeringsopgave kommer til at handle om binære søgetræer (og måske lidt priority queues)
- Den skal afleveres i starten af april (nærmere info senere)
- Fra evaluering:
  - Det 'dårlige':
    - 'Meget læsestof til forelæsningen'

Opdateringer

AALBORG UNIVERSITET

- Næste programmeringsopgave kommer til at handle om binære søgetræer (og måske lidt priority queues)
- Den skal afleveres i starten af april (nærmere info senere)
- Fra evaluering:
  - Det 'dårlige':
    - 'Meget læsestof til forelæsningen'
    - 'Flere konkrete eksempler' + 'De sidste slides gik lige hurtigt nok' + 'ville ønske jeg kunne sætte playback speed x1.25'

Opdateringer

- Næste programmeringsopgave kommer til at handle om binære søgetræer (og måske lidt priority queues)
- Den skal afleveres i starten af april (nærmere info senere)
- Fra evaluering:
  - Det 'dårlige':
    - 'Meget læsestof til forelæsningen'
    - 'Flere konkrete eksempler' + 'De sidste slides gik lige hurtigt nok' + 'ville ønske jeg kunne sætte playback speed x1.25'
    - 'Svært at se laser nogle gange'

Opdateringer

- Næste programmeringsopgave kommer til at handle om binære søgetræer (og måske lidt priority queues)
- Den skal afleveres i starten af april (nærmere info senere)
- Fra evaluering:
  - Det 'dårlige':
    - 'Meget læsestof til forelæsningen'
    - 'Flere konkrete eksempler' + 'De sidste slides gik lige hurtigt nok' + 'ville ønske jeg kunne sætte playback speed x1.25'
    - 'Svært at se laser nogle gange'
  - Det 'gode':

Opdateringer

- Næste programmeringsopgave kommer til at handle om binære søgetræer (og måske lidt priority queues)
- Den skal afleveres i starten af april (nærmere info senere)
- Fra evaluering:
  - Det 'dårlige':
    - 'Meget læsestof til forelæsningen'
    - 'Flere konkrete eksempler' + 'De sidste slides gik lige hurtigt nok' + 'ville ønske jeg kunne sætte playback speed x1.25'
    - 'Svært at se laser nogle gange'
  - Det 'gode':
    - 'Interessant emne' <3

Opdateringer

- Næste programmeringsopgave kommer til at handle om binære søgetræer (og måske lidt priority queues)
- Den skal afleveres i starten af april (nærmere info senere)
- Fra evaluering:
  - Det 'dårlige':
    - 'Meget læsestof til forelæsningen'
    - 'Flere konkrete eksempler' + 'De sidste slides gik lige hurtigt nok' + 'ville ønske jeg kunne sætte playback speed x1.25'
    - 'Svært at se laser nogle gange'
  - Det 'gode':
    - 'Interessant emne' <3
    - 'er glad for at du prøver at sørge for alle er med' + 'kan godt lide at du spørger ud og er meget opmærksom på os studerende'

Opdateringer

- Næste programmeringsopgave kommer til at handle om binære søgetræer (og måske lidt priority queues)
- Den skal afleveres i starten af april (nærmere info senere)
- Fra evaluering:
  - Det 'dårlige':
    - 'Meget læsestof til forelæsningen'
    - 'Flere konkrete eksempler' + 'De sidste slides gik lige hurtigt nok' + 'ville ønske jeg kunne sætte playback speed x1.25'
    - 'Svært at se laser nogle gange'
  - Det 'gode':
    - 'Interessant emne' <3
    - 'er glad for at du prøver at sørge for alle er med' + 'kan godt lide at du spørger ud og er meget opmærksom på os studerende'
    - Illustrationer og tegninger på tavlen

Opdateringer

- Næste programmeringsopgave kommer til at handle om binære søgetræer (og måske lidt priority queues)
- Den skal afleveres i starten af april (nærmere info senere)
- Fra evaluering:
  - Det 'dårlige':
    - 'Meget læsestof til forelæsningen'
    - 'Flere konkrete eksempler' + 'De sidste slides gik lige hurtigt nok' + 'ville ønske jeg kunne sætte playback speed x1.25'
    - 'Svært at se laser nogle gange'
  - Det 'gode':
    - 'Interessant emne' <3
    - 'er glad for at du prøver at sørge for alle er med' + 'kan godt lide at du spørger ud og er meget opmærksom på os studerende'
    - Illustrationer og tegninger på tavlen
    - 'Hjælpelærer Jakob var vildt god, specielt i første time'

Opdateringer

- Næste programmeringsopgave kommer til at handle om binære søgetræer (og måske lidt priority queues)
- Den skal afleveres i starten af april (nærmere info senere)
- Fra evaluering:
  - Det 'dårlige':
    - 'Meget læsestof til forelæsningen'
    - 'Flere konkrete eksempler' + 'De sidste slides gik lige hurtigt nok' + 'ville ønske jeg kunne sætte playback speed x1.25'
    - 'Svært at se laser nogle gange'
  - Det 'gode':
    - 'Interessant emne' <3
    - 'er glad for at du prøver at sørge for alle er med' + 'kan godt lide at du spørger ud og er meget opmærksom på os studerende'
    - Illustrationer og tegninger på tavlen
    - 'Hjælpelærer Jakob var vildt god, specielt i første time'
    - 'opdeling i 2 blokke er en fantastisk formular'

Opdateringer

- Næste programmeringsopgave kommer til at handle om binære søgetræer (og måske lidt priority queues)
- Den skal afleveres i starten af april (nærmere info senere)
- Fra evaluering:
  - Det 'dårlige':
    - 'Meget læsestof til forelæsningen'
    - 'Flere konkrete eksempler' + 'De sidste slides gik lige hurtigt nok' + 'ville ønske jeg kunne sætte playback speed x1.25'
    - 'Svært at se laser nogle gange'
  - Det 'gode':
    - 'Interessant emne' <3
    - 'er glad for at du prøver at sørge for alle er med' + 'kan godt lide at du spørger ud og er meget opmærksom på os studerende'
    - Illustrationer og tegninger på tavlen
    - 'Hjælpelærer Jakob var vildt god, specielt i første time'
    - 'opdeling i 2 blokke er en fantastisk formular'
- Næste uge: self-study session med rigtigt eksamenssæt!

Outline

Outline

Repræsentation af træer

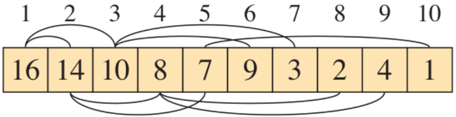
Arrays

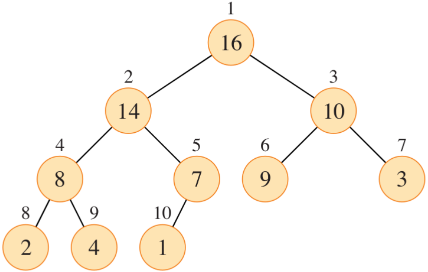
Sidste gang så vi på binære heaps, hvor vi fortolkede et array $A[1:n]$ som et binært træ.

Repræsentation af træer

Arrays

Sidste gang så vi på binære heaps, hvor vi fortolkede et array $A[1:n]$ som et binært træ.

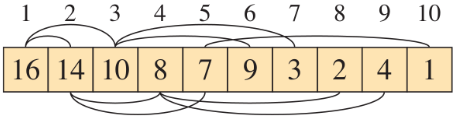


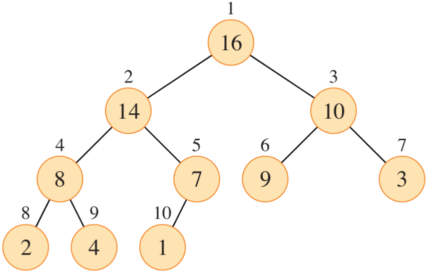


Repræsentation af træer

Arrays

Sidste gang så vi på binære heaps, hvor vi fortolkede et array $A[1:n]$ som et binært træ.





I dette tilfælde antager vi, at træet er næsten komplet, og at et element $A[i]$ er forældreknude til elementerne $A[2i]$ og $A[2i+1]$ (så længe $2i \leq n$ og $2i+1 \leq n$).

Repræsentation af træer

Begrænsninger for arrays

Hvilke begrænsninger giver dette os?

Repræsentation af træer

Begrænsninger for arrays

Hvilke begrænsninger giver dette os?

- Træet skal fyldes op fra ‘venstre mod højre’

Repræsentation af træer

Begrænsninger for arrays

Hvilke begrænsninger giver dette os?

- Træet skal fyldes op fra 'venstre mod højre'
- Træets størrelse er begrænset til n (og skal være kendt på forhånd)

Repræsentation af træer

Begrænsninger for arrays

Hvilke begrænsninger giver dette os?

- Træet skal fyldes op fra 'venstre mod højre'
- Træets størrelse er begrænset til n (og skal være kendt på forhånd)
- Hver knude kan kun have 2 børn (eller skal i hvert fald have det samme antal børn)

Repræsentation af træer

Begrænsninger for arrays

Hvilke begrænsninger giver dette os?

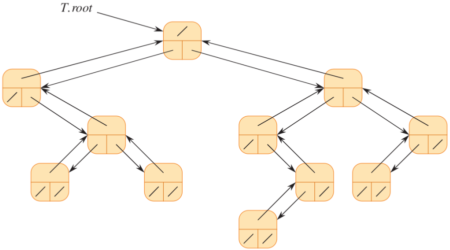
- Træet skal fyldes op fra 'venstre mod højre'
- Træets størrelse er begrænset til n (og skal være kendt på forhånd)
- Hver knude kan kun have 2 børn (eller skal i hvert fald have det samme antal børn)

En mere generisk løsning? Pointers!

Repræsentation af træer

Generalisering med pointers

Når vi repræsenterer et binært træ T med pointers, så gør vi følgende:

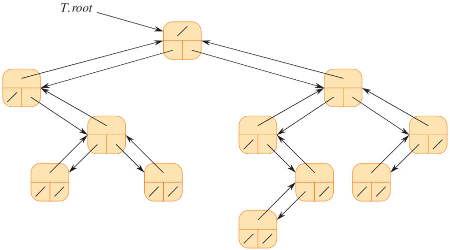


Repræsentation af træer

Generalisering med pointers

Når vi repræsenterer et binært træ T med pointers, så gør vi følgende:

- Hver knude x i træet er et objekt med 4 attributter:

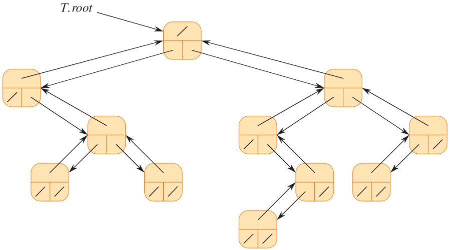


Repræsentation af træer

Generalisering med pointers

Når vi repræsenterer et binært træ T med pointers, så gør vi følgende:

- Hver knude x i træet er et objekt med 4 attributter:
  - $x.key$ indeholder knudens værdi/nøgle

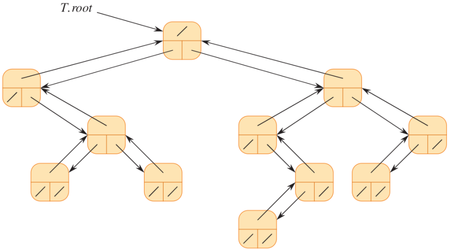


Repræsentation af træer

Generalisering med pointers

Når vi repræsenterer et binært træ T med pointers, så gør vi følgende:

- Hver knude x i træet er et objekt med 4 attributter:
  - $x.key$ indeholder knudens værdi/nøgle
  - $x.p$ peger på knudes forældre (parent)

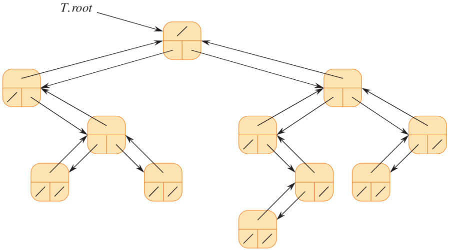


Repræsentation af træer

Generalisering med pointers

Når vi repræsenterer et binært træ T med pointers, så gør vi følgende:

- Hver knude x i træet er et objekt med 4 attributter:
  - $x.key$ indeholder knudens værdi/nøgle
  - $x.p$ peger på knudes forældre (parent)
  - $x.left$ og $x.right$ peger hhv. på knudens venstre og højre barn

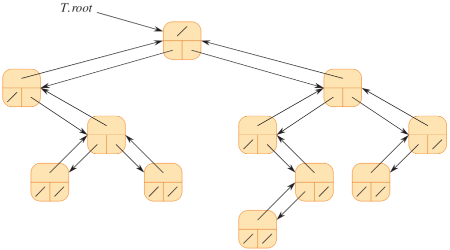


Repræsentation af træer

Generalisering med pointers

Når vi repræsenterer et binært træ T med pointers, så gør vi følgende:

- Hver knude x i træet er et objekt med 4 attributter:
  - $x.key$ indeholder knudens værdi/nøgle
  - $x.p$ peger på knudes forældre (parent)
  - $x.left$ og $x.right$ peger hhv. på knudens venstre og højre barn
- $T.root$ peger på træets rod — bemærk at $x.p = \text{NIL}$ hvis og kun hvis $x = T.root$



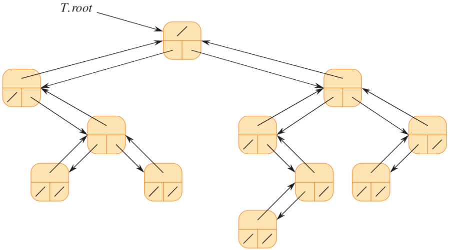
Repræsentation af træer

Generalisering med pointers

Når vi repræsenterer et binært træ T med pointers, så gør vi følgende:

- Hver knude x i træet er et objekt med 4 attributter:
  - $x.key$ indeholder knudens værdi/nøgle
  - $x.p$ peger på knudes forældre (parent)
  - $x.left$ og $x.right$ peger hhv. på knudens venstre og højre barn
- $T.root$ peger på træets rod — bemærk at $x.p = \text{NIL}$ hvis og kun hvis $x = T.root$

Nu kan vi nemt indsætte nye knuder, hvor vi vil (bare opdater pointers) og træet kan gro til en arbitrær størrelse.



Repræsentation af træer

Bonus: ubegrænset forgrening

I vores repræsentation af et træ kan hver knude stadig kun have 2 børn. Kan vi løse det?

Repræsentation af træer

Bonus: ubegrænset forgrening

I vores repræsentation af et træ kan hver knude stadig kun have 2 børn. Kan vi løse det?

- Vi kunne give hver knude k børn, som vi kan tilgå med $x.child_1, x.child_2, \ldots, x.child_k$

Repræsentation af træer

Bonus: ubegrænset forgrening

I vores repræsentation af et træ kan hver knude stadig kun have 2 børn. Kan vi løse det?

- Vi kunne give hver knude k børn, som vi kan tilgå med $x.child_1, x.child_2, \ldots, x.child_k$
  - Men så er vi tilbage til, at alle knuder maksimalt kan have k børn og...

Repræsentation af træer

Bonus: ubegrænset forgrening

I vores repræsentation af et træ kan hver knude stadig kun have 2 børn. Kan vi løse det?

- Vi kunne give hver knude k børn, som vi kan tilgå med $x.child_1, x.child_2, \ldots, x.child_k$
  - Men så er vi tilbage til, at alle knuder maksimalt kan have k børn og...
  - at vi skal sætte plads af til k børn i alle knuder, uanset at de fleste knuder måske har meget færre

Repræsentation af træer

Bonus: ubegrænset forgrening

I vores repræsentation af et træ kan hver knude stadig kun have 2 børn. Kan vi løse det?

- Vi kunne give hver knude k børn, som vi kan tilgå med $x.child_1, x.child_2, \ldots, x.child_k$
  - Men så er vi tilbage til, at alle knuder maksimalt kan have k børn og...
  - at vi skal sætte plads af til k børn i alle knuder, uanset at de fleste knuder måske har meget færre
- En bedre løsning med ubegrænset forgrening kaldes left-child, right-sibling repræsentationen

Repræsentation af træer

Bonus: ubegrænset forgrening

I vores repræsentation af et træ kan hver knude stadig kun have 2 børn. Kan vi løse det?

- Vi kunne give hver knude k børn, som vi kan tilgå med $x.\mathit{child}_1, x.\mathit{child}_2, \ldots, x.\mathit{child}_k$
    - Men så er vi tilbage til, at alle knuder maksimalt kan have k børn og...
    - at vi skal sætte plads af til k børn i alle knuder, uanset at de fleste knuder måske har meget færre
- En bedre løsning med ubegrænset forgrening kaldes left-child, right-sibling repræsentationen
    - Vi erstatter $x.\mathit{left}$ og $x.\mathit{right}$ med

Repræsentation af træer

Bonus: ubegrænset forgrening

I vores repræsentation af et træ kan hver knude stadig kun have 2 børn. Kan vi løse det?

- Vi kunne give hver knude k børn, som vi kan tilgå med $x.child_1, x.child_2, \ldots, x.child_k$
  - Men så er vi tilbage til, at alle knuder maksimalt kan have k børn og...
  - at vi skal sætte plads af til k børn i alle knuder, uanset at de fleste knuder måske har meget færre
- En bedre løsning med ubegrænset forgrening kaldes left-child, right-sibling repræsentationen
  - Vi erstatter $x.left$ og $x.right$ med
    - ⋆ $x.left\text{-}child$, der peger på knudens barn længst til venstre

Repræsentation af træer

Bonus: ubegrænset forgrening

I vores repræsentation af et træ kan hver knude stadig kun have 2 børn. Kan vi løse det?

- Vi kunne give hver knude k børn, som vi kan tilgå med $x.\mathit{child}_1, x.\mathit{child}_2, \ldots, x.\mathit{child}_k$
  - Men så er vi tilbage til, at alle knuder maksimalt kan have k børn og...
  - at vi skal sætte plads af til k børn i alle knuder, uanset at de fleste knuder måske har meget færre
- En bedre løsning med ubegrænset forgrening kaldes left-child, right-sibling repræsentationen
  - Vi erstatter $x.\mathit{left}$ og $x.\mathit{right}$ med
    - $x.\mathit{left\text{-}child}$, der peger på knudens barn længst til venstre
    - $x.\mathit{right\text{-}sibling}$, der peger på knudens søskende umiddelbart til højre

Repræsentation af træer

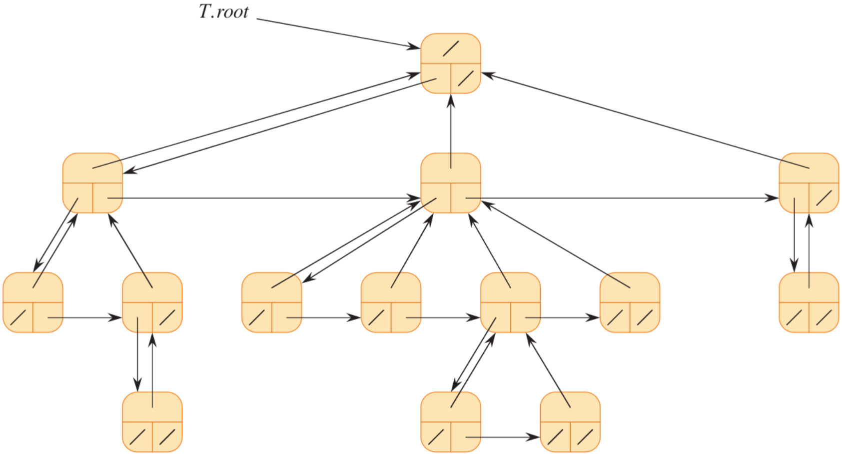
Bonus: ubegrænset forgrening

I vores repræsentation af et træ kan hver knude stadig kun have 2 børn. Kan vi løse det?

- Vi kunne give hver knude k børn, som vi kan tilgå med $x.child_1, x.child_2, \ldots, x.child_k$
  - Men så er vi tilbage til, at alle knuder maksimalt kan have k børn og...
  - at vi skal sætte plads af til k børn i alle knuder, uanset at de fleste knuder måske har meget færre
- En bedre løsning med ubegrænset forgrening kaldes left-child, right-sibling repræsentationen
  - Vi erstatter $x.left$ og $x.right$ med
    - $x.left\text{-}child$, der peger på knudens barn længst til venstre
    - $x.right\text{-}sibling$, der peger på knudens søskende umiddelbart til højre
  - Søskende-relationen bliver dermed en slags linked list repræsentation, der tillader forskelligt antal børn for hver enkelt knude

Repræsentation af træer

Left-child, right-sibling



Repræsentation af træer

Back to basics

Men da vi nu udelukkende skal se på binære træer, så holder vi os til den simple repræsentation med $x.\mathit{left}$ og $x.\mathit{right}$...

Outline

En klassisk og udbredt datastruktur er de binære søgetræer eller binary search trees eller bare BST'er.

- Understøtter operationer som Insert, Search, Delete, Minimum og Maximum
- De kan således bruges som
  - dictionaries, der associerer en nøgle (key) til en værdi (value)
  - priority queues, der opretholder en orden på basis af nøgler
- Operationer på BST'er er proportionelle til træets højde h
  - For et balanceret træ med n knuder er højden $O(\log n)$
  - I worst-case er træet dog så ubalanceret, at højden er $O(n)$

Binære søgetræer

BST-egenskaben

Ligesom vi havde en heap-egenskab, som alle knuder i et heap skulle overholde, så har vi også en egenskab for BST'er:

Binære søgetræer
BST-egenskaben

Ligesom vi havde en heap-egenskab, som alle knuder i et heap skulle overholde, så har vi også en egenskab for BST'er:

Binary-search-tree property

For to knuder x, y i et binært søgetræ:

- Hvis y er en knude i det venstre sub-træ af x, så skal $y.key \leq x.key$
- Hvis y er en knude i det højre sub-træ af x, så skal $y.key \geq x.key$

Binære søgetræer

BST-egenskaben

Ligesom vi havde en heap-egenskab, som alle knuder i et heap skulle overholde, så har vi også en egenskab for BST'er:

Binary-search-tree property

For to knuder x, y i et binært søgetræ:

- Hvis y er en knude i det venstre sub-træ af x, så skal $y.key \leq x.key$
- Hvis y er en knude i det højre sub-træ af x, så skal $y.key \geq x.key$

Hvordan adskiller det sig fra et heap?

Binære søgetræer

BST-egenskaben

Ligesom vi havde en heap-egenskab, som alle knuder i et heap skulle overholde, så har vi også en egenskab for BST'er:

Binary-search-tree property

For to knuder x, y i et binært søgetræ:

- Hvis y er en knude i det venstre sub-træ af x, så skal $y.key \leq x.key$
- Hvis y er en knude i det højre sub-træ af x, så skal $y.key \geq x.key$

Hvordan adskiller det sig fra et heap?

Max-Heap property

For alle knuder $i > 1$ gælder $A[\text{Parent}(i)] \geq A[i]$

Binære søgetræer

BST-egenskaben

Mere uformelt siger vi, at alt til venstre skal være mindre end eller lig med, og at alt til højre skal være større end eller lig med.

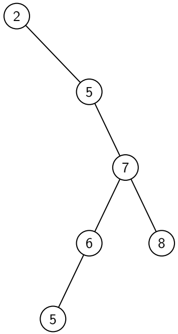
Binære søgetræer

BST-egenskaben

Mere uformelt siger vi, at alt til venstre skal være mindre end eller lig med, og at alt til højre skal være større end eller lig med.

Er disse træer BST'er?

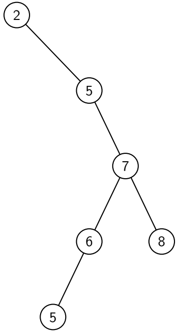
Mere uformelt siger vi, at alt til venstre skal være mindre end eller lig med, og at alt til højre skal være større end eller lig med.
Er disse træer BST'er?



Binære søgetræer

BST-egenskaben

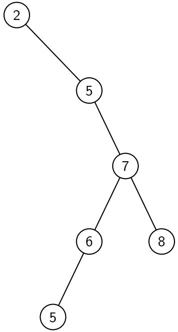
Mere uformelt siger vi, at alt til venstre skal være mindre end eller lig med, og at alt til højre skal være større end eller lig med.
Er disse træer BST'er?



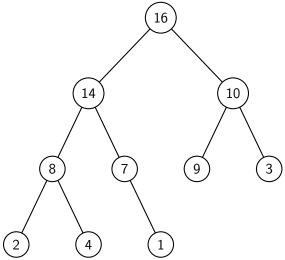
Ja!

Binære søgetræer
BST-egenskaben

Mere uformelt siger vi, at alt til venstre skal være mindre end eller lig med, og at alt til højre skal være større end eller lig med.
Er disse træer BST'er?



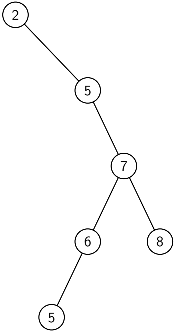
Ja!



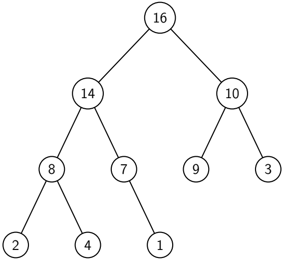
Binære søgetræer

BST-egenskaben

Mere uformelt siger vi, at alt til venstre skal være mindre end eller lig med, og at alt til højre skal være større end eller lig med.
Er disse træer BST'er?



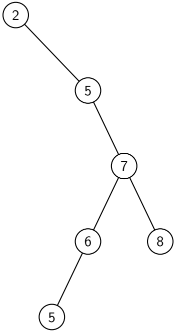
Ja!



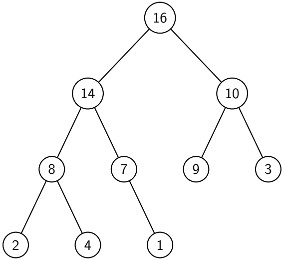
Nej!

Binære søgetræer
BST-egenskaben

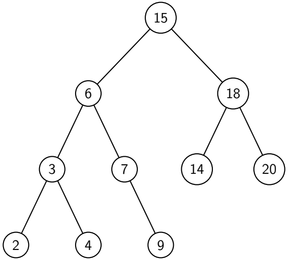
Mere uformelt siger vi, at alt til venstre skal være mindre end eller lig med, og at alt til højre skal være større end eller lig med.
Er disse træer BST'er?



Ja!



Nej!

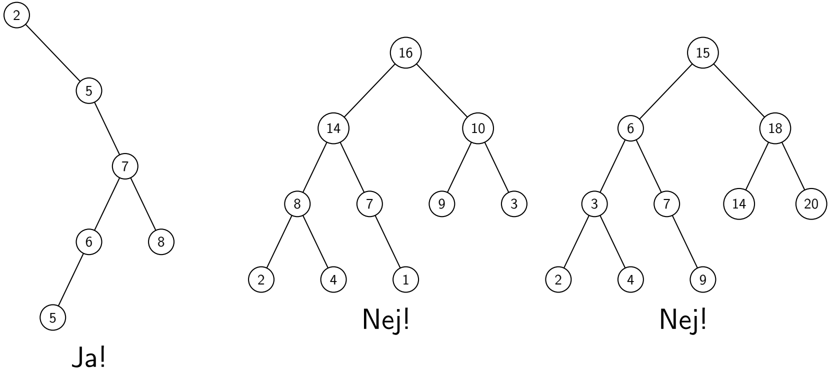


Binære søgetræer

BST-egenskaben

Mere uformelt siger vi, at alt til venstre skal være mindre end eller lig med, og at alt til højre skal være større end eller lig med.

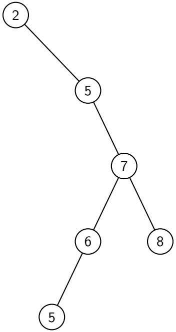
Er disse træer BST'er?



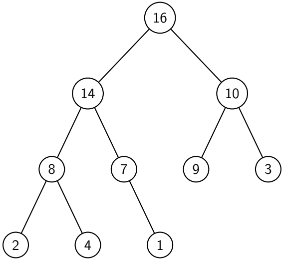
Binære søgetræer

BST-egenskaben

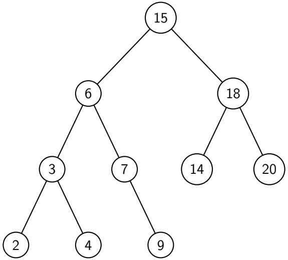
Mere uformelt siger vi, at alt til venstre skal være mindre end eller lig med, og at alt til højre skal være større end eller lig med.
Er disse træer BST'er?



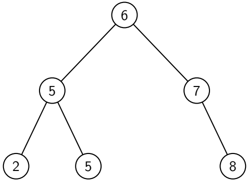
Ja!



Nej!



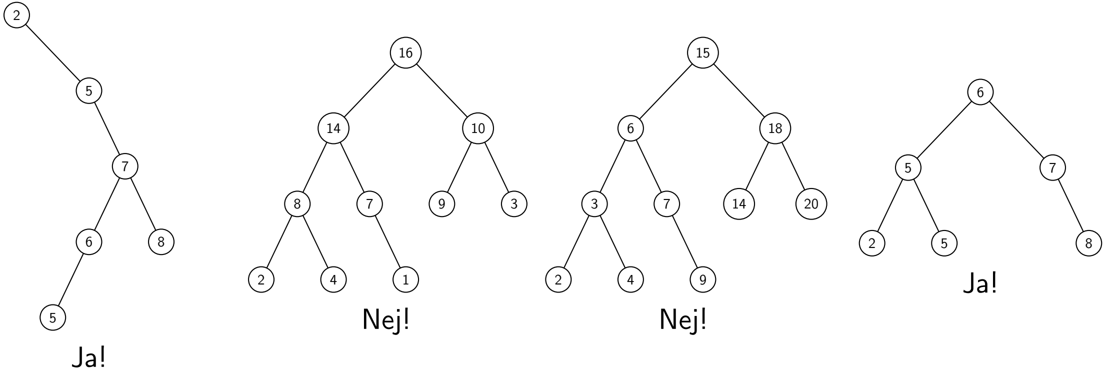
Nej!



Binære søgetræer

BST-egenskaben

Mere uformelt siger vi, at alt til venstre skal være mindre end eller lig med, og at alt til højre skal være større end eller lig med.
Er disse træer BST'er?



Binære søgetræer

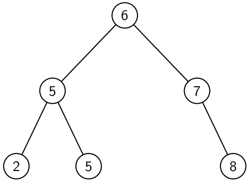
Traversere træet

Vi ser senere på, hvordan vi indsætter og sletter fra et BST. Først, hvad kan vi med et BST?

- Vi kan udnytte BST-egenskaben til at printe alle nøglerne i sorteret rækkefølge

Inorder-Tree-Walk(x)

if $x \neq$ NIL
        Inorder-Tree-Walk($x.\mathit{left}$)
        print $x.\mathit{key}$
        Inorder-Tree-Walk($x.\mathit{right}$)



Binære søgetræer

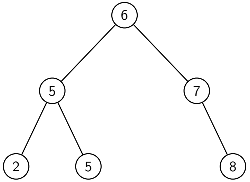
Traversere træet

Vi ser senere på, hvordan vi indsætter og sletter fra et BST. Først, hvad kan vi med et BST?

- Vi kan udnytte BST-egenskaben til at printe alle nøglerne i sorteret rækkefølge
- Inorder-Tree-Walk(x) tager en knude x og printer først alle nøgler i det venstre (lille) sub-træ, så $x.key$ og til sidst alle nøgler i det højre (store) sub-træ

Inorder-Tree-Walk(x)

```
if x ≠ NIL
    Inorder-Tree-Walk(x.left)
    print x.key
    Inorder-Tree-Walk(x.right)
```



Binære søgetræer

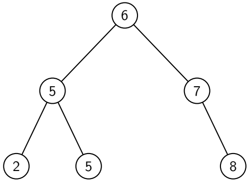
Traversere træet

Vi ser senere på, hvordan vi indsætter og sletter fra et BST. Først, hvad kan vi med et BST?

- Vi kan udnytte BST-egenskaben til at printe alle nøglerne i sorteret rækkefølge
- Inorder-Tree-Walk(x) tager en knude x og printer først alle nøgler i det venstre (lille) sub-træ, så $x.key$ og til sidst alle nøgler i det højre (store) sub-træ
- Kompleksitet?

Inorder-Tree-Walk(x)

if $x \neq$ NIL
    Inorder-Tree-Walk($x.left$)
    print $x.key$
    Inorder-Tree-Walk($x.right$)



Binære søgetræer

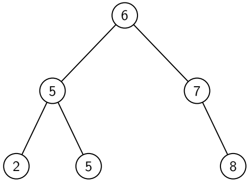
Traversere træet

Vi ser senere på, hvordan vi indsætter og sletter fra et BST. Først, hvad kan vi med et BST?

- Vi kan udnytte BST-egenskaben til at printe alle nøglerne i sorteret rækkefølge
- Inorder-Tree-Walk(x) tager en knude x og printer først alle nøgler i det venstre (lille) sub-træ, så $x.key$ og til sidst alle nøgler i det højre (store) sub-træ
- Kompleksitet? $\Theta(n)$ (for et træ med n knuder)

Inorder-Tree-Walk(x)

```
if x ≠ NIL
    Inorder-Tree-Walk(x.left)
    print x.key
    Inorder-Tree-Walk(x.right)
```

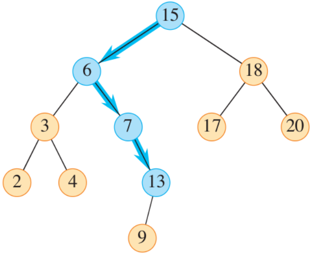


Søgning

For søgning i træet har vi følgende simple procedure:

Iterative-Tree-Search(T, k)

```
while x ≠ NIL and x.key ≠ k
    if k < x.key
        x = x.left
    else x = x.right
return x
```



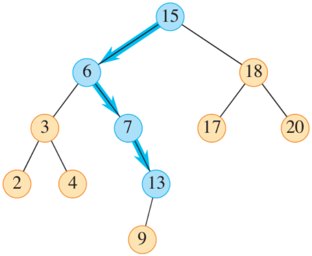
Søgning

For søgning i træet har vi følgende simple procedure:

Iterative-Tree-Search(T, k)

```
while x ≠ NIL and x.key ≠ k
    if k < x.key
        x = x.left
    else x = x.right
return x
```

- While-løkken tester, om x er NIL (søgningen har fejlet) eller $x.key = k$ (søgningen er successfuld)



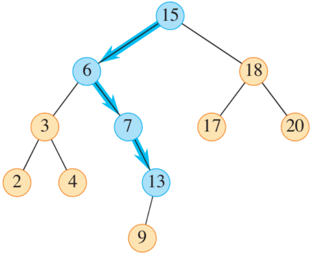
Søgning

For søgning i træet har vi følgende simple procedure:

Iterative-Tree-Search(T, k)

while $x \neq$ NIL and $x.key \neq k$
  if $k < x.key$
    $x = x.left$
  else $x = x.right$
return x

- While-løkken tester, om x er NIL (søgningen har fejlet) eller $x.key = k$ (søgningen er successfuld)
- Hvis k er mindre end $x.key$ skal vi lede videre i det venstre sub-træ (vi sætter $x = x.left$) — og ellers skal vi lede i det højre sub-træ



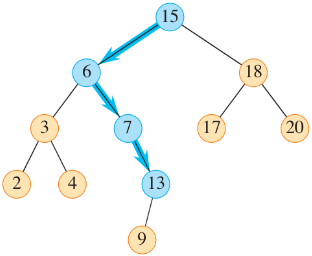
Søgning

For søgning i træet har vi følgende simple procedure:

Iterative-Tree-Search(T, k)

while $x \neq$ NIL and $x.key \neq k$
    if $k < x.key$
        $x = x.left$
    else $x = x.right$
return x

- While-løkken tester, om x er NIL (søgningen har fejlet) eller $x.key = k$ (søgningen er successfuld)
- Hvis k er mindre end $x.key$ skal vi lede videre i det venstre sub-træ (vi sætter $x = x.left$) — og ellers skal vi lede i det højre sub-træ
- Vi slutter, når vi enten har fundet det rigtige element eller x er sat til NIL

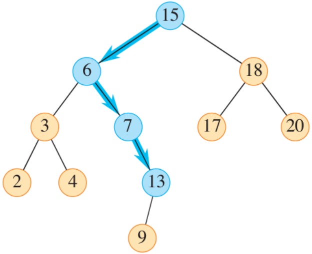


Søgning

For søgning i træet har vi følgende simple procedure:

Iterative-Tree-Search(T, k)

```
while x ≠ NIL and x.key ≠ k
    if k < x.key
        x = x.left
    else x = x.right
return x
```



- While-løkken tester, om x er NIL (søgningen har fejlet) eller $x.key = k$ (søgningen er successfuld)
- Hvis k er mindre end $x.key$ skal vi lede videre i det venstre sub-træ (vi sætter $x = x.left$) — og ellers skal vi lede i det højre sub-træ
- Vi slutter, når vi enten har fundet det rigtige element eller x er sat til NIL
- Kompleksitet?

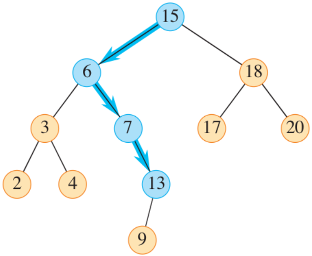
Søgning

For søgning i træet har vi følgende simple procedure:

Iterative-Tree-Search(T, k)

```
while x ≠ NIL and x.key ≠ k
    if k < x.key
        x = x.left
    else x = x.right
return x
```

- While-løkken tester, om x er NIL (søgningen har fejlet) eller $x.key = k$ (søgningen er successfuld)
- Hvis k er mindre end $x.key$ skal vi lede videre i det venstre sub-træ (vi sætter $x = x.left$) — og ellers skal vi lede i det højre sub-træ
- Vi slutter, når vi enten har fundet det rigtige element eller x er sat til NIL
- Kompleksitet?
    - Hver iteration bevæger sig et niveau ned i træet

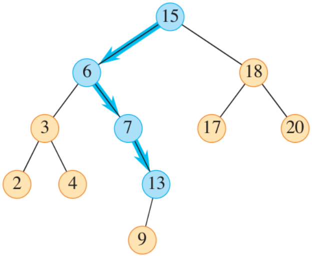


Søgning

For søgning i træet har vi følgende simple procedure:

Iterative-Tree-Search(T, k)

while $x \neq$ NIL and $x.key \neq k$
  if $k < x.key$
    $x = x.left$
  else $x = x.right$
return x



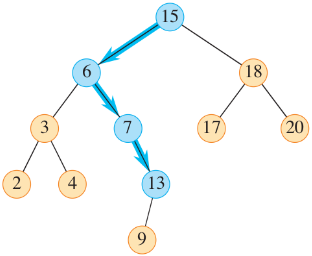
- While-løkken tester, om x er NIL (søgningen har fejlet) eller $x.key = k$ (søgningen er successfuld)
- Hvis k er mindre end $x.key$ skal vi lede videre i det venstre sub-træ (vi sætter $x = x.left$) — og ellers skal vi lede i det højre sub-træ
- Vi slutter, når vi enten har fundet det rigtige element eller x er sat til NIL
- Kompleksitet?
  - Hver iteration bevæger sig et niveau ned i træet
  - Der er maks h niveauer, så kompleksiteten er $O(h)$

Søgning

For søgning i træet har vi følgende simple procedure:

Iterative-Tree-Search(T, k)

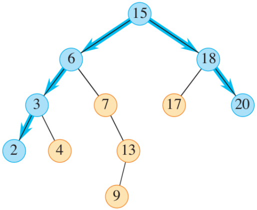
while $x \neq$ NIL and $x.key \neq k$
  if $k < x.key$
    $x = x.left$
  else $x = x.right$
return x



- While-løkken tester, om x er NIL (søgningen har fejlet) eller $x.key = k$ (søgningen er successfuld)
- Hvis k er mindre end $x.key$ skal vi lede videre i det venstre sub-træ (vi sætter $x = x.left$) — og ellers skal vi lede i det højre sub-træ
- Vi slutter, når vi enten har fundet det rigtige element eller x er sat til NIL
- Kompleksitet?
  - Hver iteration bevæger sig et niveau ned i træet
  - Der er maks h niveauer, så kompleksiteten er $O(h)$
- En rekursiv udgave findes også, men den vil typisk være mindre effektiv

Minimum og Maximum

Vi har også to simple procedurer til at finde det største og mindste element i et træ.



Tree-Minimum(x)

1 **while** $x.left \neq$ NIL
2 $\quad x = x.left$
3 **return** x

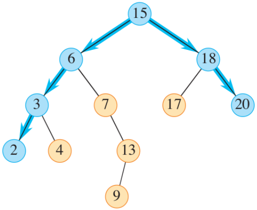
Tree-Maximum(x)

1 **while** $x.right \neq$ NIL
2 $\quad x = x.right$
3 **return** x

Minimum og Maximum

Vi har også to simple procedurer til at finde det største og mindste element i et træ.

- Tree-Minimum(x) følger blot stien, der fremkommer ved at fortsætte til venstre i træet indtil, at den når et blad



Tree-Minimum(x)

```
while x.left ≠ NIL
      x = x.left
return x
```

Tree-Maximum(x)

```
while x.right ≠ NIL
      x = x.right
return x
```

Minimum og Maximum

Vi har også to simple procedurer til at finde det største og mindste element i et træ.

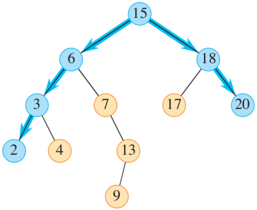
- Tree-Minimum(x) følger blot stien, der fremkommer ved at fortsætte til venstre i træet indtil, at den når et blad
- Tree-Maximum(x) gør det samme, denne gang bare ved at gå mod højre

Tree-Minimum(x)

1 **while** $x.left \neq$ NIL
2 $\quad x = x.left$
3 **return** x

Tree-Maximum(x)

1 **while** $x.right \neq$ NIL
2 $\quad x = x.right$
3 **return** x



Minimum og Maximum

Vi har også to simple procedurer til at finde det største og mindste element i et træ.

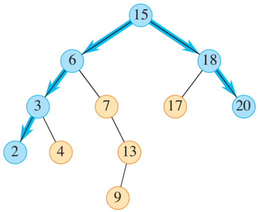
- Tree-Minimum(x) følger blot stien, der fremkommer ved at fortsætte til venstre i træet indtil, at den når et blad
- Tree-Maximum(x) gør det samme, denne gang bare ved at gå mod højre
- BST-egenskaben garanterer korrektheden af disse metoder

Tree-Minimum(x)

```
while x.left ≠ NIL
    x = x.left
return x
```

Tree-Maximum(x)

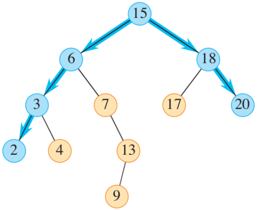
```
while x.right ≠ NIL
    x = x.right
return x
```



Minimum og Maximum

Vi har også to simple procedurer til at finde det største og mindste element i et træ.

- Tree-Minimum(x) følger blot stien, der fremkommer ved at fortsætte til venstre i træet indtil, at den når et blad
- Tree-Maximum(x) gør det samme, denne gang bare ved at gå mod højre
- BST-egenskaben garanterer korrektheden af disse metoder
- Begge kører i $O(h)$



Tree-Minimum(x)

```
1  while x.left ≠ NIL
2      x = x.left
3  return x
```

Tree-Maximum(x)

```
1  while x.right ≠ NIL
2      x = x.right
3  return x
```

Predecessor og successor

De to foregående procedurer kan vi blandt andet benytte til at konstruere procedurer for at finde en knudes predecessor og successor.

- Givet en knude x i et BST T returnerer Tree-Successor den knude, som har den mindste nøgle, der er større end $x.key$

Tree-Successor(x)

```
if x.right ≠ NIL
    return Tree-Minimum(x.right)
else
    y = x.p
    while y ≠ NIL and x == y.right
        x = y
        y = y.p
return y
```

Predecessor og successor

De to foregående procedurer kan vi blandt andet benytte til at konstruere procedurer for at finde en knudes predecessor og successor.

- Givet en knude x i et BST T returnerer Tree-Successor den knude, som har den mindste nøgle, der er større end $x.key$
- Der er to cases:

Tree-Successor(x)

```
if x.right ≠ NIL
    return Tree-Minimum(x.right)
else
    y = x.p
    while y ≠ NIL and x == y.right
        x = y
        y = y.p
return y
```

Predecessor og successor

De to foregående procedurer kan vi blandt andet benytte til at konstruere procedurer for at finde en knudes predecessor og successor.

- Givet en knude x i et BST T returnerer Tree-Successor den knude, som har den mindste nøgle, der er større end $x.key$
- Der er to cases:
  - 1) Enten er successoren det mindste element i det højre (altså store) sub-træ til x

Tree-Successor(x)

```
if x.right ≠ NIL
    return Tree-Minimum(x.right)
else
    y = x.p
    while y ≠ NIL and x == y.right
        x = y
        y = y.p
return y
```

Predecessor og successor

De to foregående procedurer kan vi blandt andet benytte til at konstruere procedurer for at finde en knudes predecessor og successor.

- Givet en knude x i et BST T returnerer Tree-Successor den knude, som har den mindste nøgle, der er større end $x.key$
- Der er to cases:
  - 1) Enten er successoren det mindste element i det højre (altså store) sub-træ til x
  - 2) Ellers skal vi kravle op i træet indtil, at vi finder den første knude, som har x i sit venstre (lille sub-træ)

Tree-Successor(x)

```
if x.right ≠ NIL
    return Tree-Minimum(x.right)
else
    y = x.p
    while y ≠ NIL and x == y.right
        x = y
        y = y.p
return y
```

Predecessor og successor

De to foregående procedurer kan vi blandt andet benytte til at konstruere procedurer for at finde en knudes predecessor og successor.

- Givet en knude x i et BST T returnerer Tree-Successor den knude, som har den mindste nøgle, der er større end $x.key$
- Der er to cases:
  - 1) Enten er successoren det mindste element i det højre (altså store) sub-træ til x
  - 2) Ellers skal vi kravle op i træet indtil, at vi finder den første knude, som har x i sit venstre (lille sub-træ)
- Der er en helt analog procedure for at finde predecessor (største element, der er mindre end x)

Tree-Successor(x)

```
if x.right ≠ NIL
    return Tree-Minimum(x.right)
else
    y = x.p
    while y ≠ NIL and x == y.right
        x = y
        y = y.p
return y
```

Successor

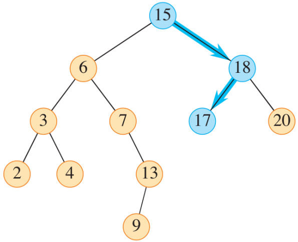
Eksempel

Tree-Successor(x)

```
if x.right ≠ NIL
    return Tree-Minimum(x.right)
else
    y = x.p
    while y ≠ NIL and x == y.right
        x = y
        y = y.p
return y
```

Successor
Eksempel

Tree-Successor(15)



Successoren til knuden med nøgle 15 er det mindste element i det højre sub-træ.

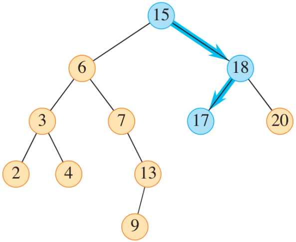
Tree-Successor(x)

if $x.right \neq$ NIL
    return Tree-Minimum($x.right$)
else
    $y = x.p$
    while $y \neq$ NIL and $x == y.right$
        $x = y$
        $y = y.p$
return y

Successor

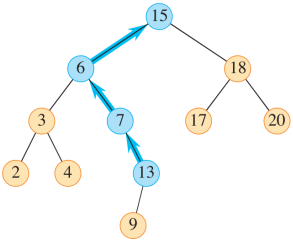
Eksempel

Tree-Successor(15)



Successoren til knuden med nøgle 15 er det mindste element i det højre sub-træ.

Tree-Successor(13)



Knuden med nøgle 13 har ikke noget sub-træ til højre, så dens successor er den mindste ancestor ('bedsteforældre'), hvis venstre barn også er en ancestor til 13.

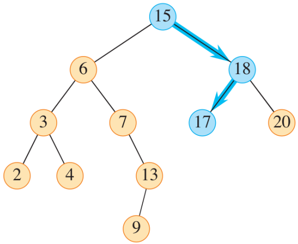
Tree-Successor(x)

if $x.right \neq$ NIL
    return Tree-Minimum($x.right$)
else
    $y = x.p$
    while $y \neq$ NIL and $x == y.right$
        $x = y$
        $y = y.p$
return y

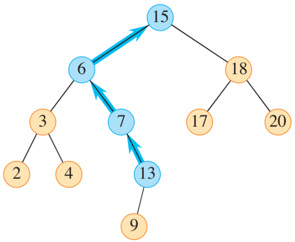
Successor

Eksempel

Tree-Successor(15)



Tree-Successor(13)



Tree-Successor(x)

```
if x.right ≠ NIL
    return Tree-Minimum(x.right)
else
    y = x.p
    while y ≠ NIL and x == y.right
        x = y
        y = y.p
return y
```

Successoren til knuden med nøgle 15 er det mindste element i det højre sub-træ.

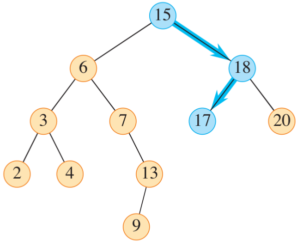
Knuden med nøgle 13 har ikke noget sub-træ til højre, så dens successor er den mindste ancestor (‘bedsteforældre’), hvis venstre barn også er en ancestor til 13.

Kompleksitet?

Successor

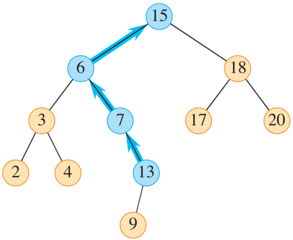
Eksempel

Tree-Successor(15)



Successoren til knuden med nøgle 15 er det mindste element i det højre sub-træ.

Tree-Successor(13)



Knuden med nøgle 13 har ikke noget sub-træ til højre, så dens successor er den mindste ancestor ('bedsteforældre'), hvis venstre barn også er en ancestor til 13.

Tree-Successor(x)

```
if x.right ≠ NIL
    return Tree-Minimum(x.right)
else
    y = x.p
    while y ≠ NIL and x == y.right
        x = y
        y = y.p
return y
```

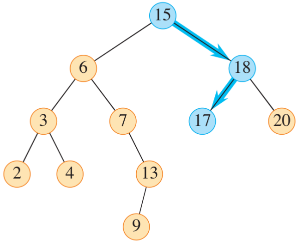
Kompleksitet?

- Tree-Minimum kører i $O(h)$

Successor

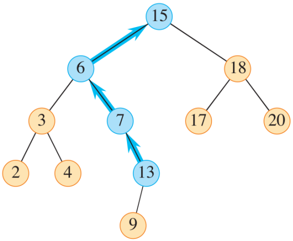
Eksempel

Tree-Successor(15)



Successoren til knuden med nøgle 15 er det mindste element i det højre sub-træ.

Tree-Successor(13)



Knuden med nøgle 13 har ikke noget sub-træ til højre, så dens successor er den mindste ancestor ('bedsteforældre'), hvis venstre barn også er en ancestor til 13.

Tree-Successor(x)

1 **if** $x.right \neq$ NIL
2   **return** Tree-Minimum($x.right$)
3 **else**
4     $y = x.p$
5   **while** $y \neq$ NIL and $x == y.right$
6         $x = y$
7         $y = y.p$
8 **return** y

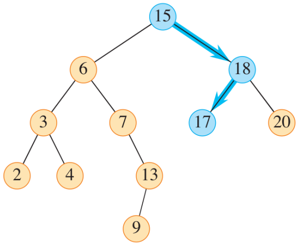
Kompleksitet?

- Tree-Minimum kører i $O(h)$
- Linie 3-8…

Successor

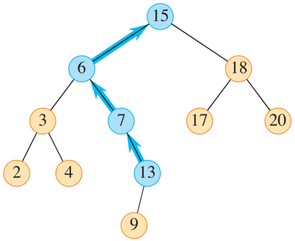
Eksempel

Tree-Successor(15)



Successoren til knuden med nøgle 15 er det mindste element i det højre sub-træ.

Tree-Successor(13)



Knuden med nøgle 13 har ikke noget sub-træ til højre, så dens successor er den mindste ancestor ('bedsteforældre'), hvis venstre barn også er en ancestor til 13.

Tree-Successor(x)

1 **if** $x.right \neq$ NIL
2   **return** Tree-Minimum($x.right$)
3 **else**
4     $y = x.p$
5   **while** $y \neq$ NIL and $x == y.right$
6         $x = y$
7         $y = y.p$
8 **return** y

Kompleksitet?

- Tree-Minimum kører i $O(h)$
- Linie 3-8... traverserer træet og kører dermed også i $O(h)$

Outline

Exercises

Super fedt! <3

På Moodle! Go! Fungerer det fint?

Kuriosum

ChatGPT snyder dig nemt

Kan I huske det gode spørgsmål fra 2. forelæsning?

Kuriosum

ChatGPT snyder dig nemt

Kan I huske det gode spørgsmål fra 2. forelæsning?

> *Er der et eksempel på en algoritme, hvor der er forskel på upper og lower bound i worst case?*

Kuriosum

ChatGPT snyder dig nemt

Kan I huske det gode spørgsmål fra 2. forelæsning?

> *Er der et eksempel på en algoritme, hvor der er forskel på upper og lower bound i worst case?*

Lad os spørge ChatGPT!

Binære søgetræer

Hvad har vi set so far?

Lad os lige samle op:

Binære søgetræer

Hvad har vi set so far?

Lad os lige samle op:

- Inorder-Tree-Walk(x) printer alle elementer i træet med rod i x ud i sorteret rækkefølge

Binære søgetræer

Hvad har vi set so far?

Lad os lige samle op:

- Inorder-Tree-Walk(x) printer alle elementer i træet med rod i x ud i sorteret rækkefølge
- Tree-Search(T, k) returnerer det element, der har nøglen k (eller NIL, hvis det ikke findes)

Binære søgetræer

Hvad har vi set so far?

Lad os lige samle op:

- Inorder-Tree-Walk(x) printer alle elementer i træet med rod i x ud i sorteret rækkefølge
- Tree-Search(T, k) returnerer det element, der har nøglen k (eller NIL, hvis det ikke findes)
- Tree-Minimum(x) returnerer det mindste element i træet med rod i x (og Tree-Maximum(x) returnerer omvendt det største element)

Binære søgetræer

Hvad har vi set so far?

Lad os lige samle op:

- Inorder-Tree-Walk(x) printer alle elementer i træet med rod i x ud i sorteret rækkefølge
- Tree-Search(T, k) returnerer det element, der har nøglen k (eller NIL, hvis det ikke findes)
- Tree-Minimum(x) returnerer det mindste element i træet med rod i x (og Tree-Maximum(x) returnerer omvendt det største element)
- Tree-Successor(x) returnerer det mindste element, der er større end x

Binære søgetræer

Hvad har vi set so far?

Lad os lige samle op:

- Inorder-Tree-Walk(x) printer alle elementer i træet med rod i x ud i sorteret rækkefølge
- Tree-Search(T, k) returnerer det element, der har nøglen k (eller NIL, hvis det ikke findes)
- Tree-Minimum(x) returnerer det mindste element i træet med rod i x (og Tree-Maximum(x) returnerer omvendt det største element)
- Tree-Successor(x) returnerer det mindste element, der er større end x
- Udover Inorder-Tree-Walk, som kører i $\Theta(n)$ kører alle procedurerne i $O(h)$, hvor h er træets højde

Binære søgetræer

Hvad har vi set so far?

Lad os lige samle op:

- Inorder-Tree-Walk(x) printer alle elementer i træet med rod i x ud i sorteret rækkefølge
- Tree-Search(T, k) returnerer det element, der har nøglen k (eller NIL, hvis det ikke findes)
- Tree-Minimum(x) returnerer det mindste element i træet med rod i x (og Tree-Maximum(x) returnerer omvendt det største element)
- Tree-Successor(x) returnerer det mindste element, der er større end x
- Udover Inorder-Tree-Walk, som kører i $\Theta(n)$ kører alle procedurerne i $O(h)$, hvor h er træets højde
- Alt dette lader sig kun gøre pga BST-egenskaben, der siger,at alle elementer til venstre for x skal være mindre end x og alle elementer til højre skal være større end x

Binære søgetræer

Hvad har vi set so far?

Lad os lige samle op:

- Inorder-Tree-Walk(x) printer alle elementer i træet med rod i x ud i sorteret rækkefølge
- Tree-Search(T, k) returnerer det element, der har nøglen k (eller NIL, hvis det ikke findes)
- Tree-Minimum(x) returnerer det mindste element i træet med rod i x (og Tree-Maximum(x) returnerer omvendt det største element)
- Tree-Successor(x) returnerer det mindste element, der er større end x
- Udover Inorder-Tree-Walk, som kører i $\Theta(n)$ kører alle procedurerne i $O(h)$, hvor h er træets højde
- Alt dette lader sig kun gøre pga BST-egenskaben, der siger,at alle elementer til venstre for x skal være mindre end x og alle elementer til højre skal være større end x

Vi ser nu på, hvordan vi kan indsætte og slette fra BST'er, så vi fortsat opretholder BST-egenskaben.

Outline

Manipulering med BST'er

Insertion

At indsætte i et BST er relativt simpelt.

- Proceduren tager et træ T og en knude z, som vi vil indsætte som et nyt blad i træet

Tree-Insert(T, z)

```
x = T.root
y = NIL
while x ≠ NIL
      y = x
      if z.key < x.key
            x = x.left
      else x = x.right
z.p = y
if y == NIL
      T.root = z
elseif z.key < y.key
      y.left = z
else y.right = z
```

Manipulering med BST'er

Insertion

At indsætte i et BST er relativt simpelt.

- Proceduren tager et træ T og en knude z, som vi vil indsætte som et nyt blad i træet
- Vi definerer x til at være den knude, vi sammenligner med z (til at starte med $T.root$), og y er bare forældren til x

Tree-Insert(T, z)

```
x = T.root
y = NIL
while x ≠ NIL
    y = x
    if z.key < x.key
        x = x.left
    else x = x.right
z.p = y
if y == NIL
    T.root = z
elseif z.key < y.key
    y.left = z
else y.right = z
```

Manipulering med BST'er

Insertion

At indsætte i et BST er relativt simpelt.

- Proceduren tager et træ T og en knude z, som vi vil indsætte som et nyt blad i træet
- Vi definerer x til at være den knude, vi sammenligner med z (til at starte med $T.root$), og y er bare forældren til x
- Sålænge x ikke er NIL sammenligner vi $z.key$ med $x.key$, og går enten til højre eller venstre i træet

Tree-Insert(T, z)

```
x = T.root
y = NIL
while x ≠ NIL
    y = x
    if z.key < x.key
        x = x.left
    else x = x.right
z.p = y
if y == NIL
    T.root = z
elseif z.key < y.key
    y.left = z
else y.right = z
```

Manipulering med BST'er

Insertion

At indsætte i et BST er relativt simpelt.

- Proceduren tager et træ T og en knude z, som vi vil indsætte som et nyt blad i træet
- Vi definerer x til at være den knude, vi sammenligner med z (til at starte med $T.root$), og y er bare forældren til x
- Sålænge x ikke er NIL sammenligner vi $z.key$ med $x.key$, og går enten til højre eller venstre i træet
- Når x er NIL, sætter vi z's parent til at være y

Tree-Insert(T, z)

```
x = T.root
y = NIL
while x ≠ NIL
    y = x
    if z.key < x.key
        x = x.left
    else x = x.right
z.p = y
if y == NIL
    T.root = z
elseif z.key < y.key
    y.left = z
else y.right = z
```

Manipulering med BST'er

Insertion

At indsætte i et BST er relativt simpelt.

- Proceduren tager et træ T og en knude z, som vi vil indsætte som et nyt blad i træet
- Vi definerer x til at være den knude, vi sammenligner med z (til at starte med $T.root$), og y er bare forældren til x
- Sålænge x ikke er NIL sammenligner vi $z.key$ med $x.key$, og går enten til højre eller venstre i træet
- Når x er NIL, sætter vi z's parent til at være y
- Hvis y er NIL, er træet tomt, og z skal være den nye rod — ellers indsætter vi x som enten det venstre eller højre barn af y

Tree-Insert(T, z)

```
x = T.root
y = NIL
while x ≠ NIL
    y = x
    if z.key < x.key
        x = x.left
    else x = x.right
z.p = y
if y == NIL
    T.root = z
elseif z.key < y.key
    y.left = z
else y.right = z
```

Manipulering af BST'er

Insertion eksempel

Lad os prøve at indsætte nøglerne $\langle 12, 18, 5, 2, 15, 9, 17, 19, 13 \rangle$:

Tree-Insert(T, z)

```
x = T.root
y = NIL
while x ≠ NIL
    y = x
    if z.key < x.key
        x = x.left
    else x = x.right
z.p = y
if y == NIL
    T.root = z
elseif z.key < y.key
    y.left = z
else y.right = z
```

Manipulering af BST'er

Insertion eksempel

Lad os prøve at indsætte nøglerne $\langle 12, 18, 5, 2, 15, 9, 17, 19, 13 \rangle$:

- Tree-Insert(T, 12) — kommer ind som rod

Tree-Insert(T, z)

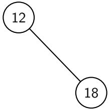
```
x = T.root
y = NIL
while x ≠ NIL
    y = x
    if z.key < x.key
        x = x.left
    else x = x.right
z.p = y
if y == NIL
    T.root = z
elseif z.key < y.key
    y.left = z
else y.right = z
```

Manipulering af BST'er

Insertion eksempel

Lad os prøve at indsætte nøglerne $\langle 12, 18, 5, 2, 15, 9, 17, 19, 13 \rangle$:

- Tree-Insert$(T, 12)$ — kommer ind som rod
- Tree-Insert$(T, 18)$



Tree-Insert(T, z)

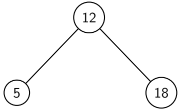
```
x = T.root
y = NIL
while x ≠ NIL
    y = x
    if z.key < x.key
        x = x.left
    else x = x.right
z.p = y
if y == NIL
    T.root = z
elseif z.key < y.key
    y.left = z
else y.right = z
```

Manipulering af BST'er

Insertion eksempel

Lad os prøve at indsætte nøglerne $\langle 12, 18, 5, 2, 15, 9, 17, 19, 13 \rangle$:

- Tree-Insert(T, 12) — kommer ind som rod
- Tree-Insert(T, 18)
- Tree-Insert(T, 5)



Tree-Insert(T, z)

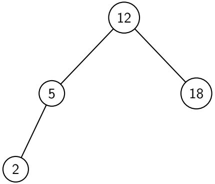
```
x = T.root
y = NIL
while x ≠ NIL
        y = x
        if z.key < x.key
                x = x.left
        else x = x.right
z.p = y
if y == NIL
        T.root = z
elseif z.key < y.key
        y.left = z
else y.right = z
```

Manipulering af BST'er

Insertion eksempel

Lad os prøve at indsætte nøglerne $\langle 12, 18, 5, 2, 15, 9, 17, 19, 13 \rangle$:

- Tree-Insert$(T, 12)$ — kommer ind som rod
- Tree-Insert$(T, 18)$
- Tree-Insert$(T, 5)$
- Tree-Insert$(T, 2)$



Tree-Insert(T, z)

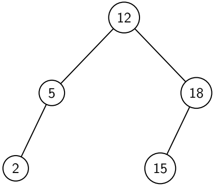
```
x = T.root
y = NIL
while x ≠ NIL
    y = x
    if z.key < x.key
        x = x.left
    else x = x.right
z.p = y
if y == NIL
    T.root = z
elseif z.key < y.key
    y.left = z
else y.right = z
```

Manipulering af BST'er

Insertion eksempel

Lad os prøve at indsætte nøglerne $\langle 12, 18, 5, 2, 15, 9, 17, 19, 13 \rangle$:

- Tree-Insert(T, 12) — kommer ind som rod
- Tree-Insert(T, 18)
- Tree-Insert(T, 5)
- Tree-Insert(T, 2)
- Tree-Insert(T, 15)



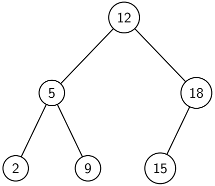
Tree-Insert(T, z)

```
x = T.root
y = NIL
while x ≠ NIL
    y = x
    if z.key < x.key
        x = x.left
    else x = x.right
z.p = y
if y == NIL
    T.root = z
elseif z.key < y.key
    y.left = z
else y.right = z
```

Manipulering af BST'er
Insertion eksempel

Lad os prøve at indsætte nøglerne $\langle 12, 18, 5, 2, 15, 9, 17, 19, 13 \rangle$:

- Tree-Insert$(T, 12)$ — kommer ind som rod
- Tree-Insert$(T, 18)$
- Tree-Insert$(T, 5)$
- Tree-Insert$(T, 2)$
- Tree-Insert$(T, 15)$
- Tree-Insert$(T, 9)$



Tree-Insert(T, z)

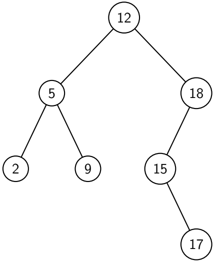
```
x = T.root
y = NIL
while x ≠ NIL
        y = x
        if z.key < x.key
                x = x.left
        else x = x.right
z.p = y
if y == NIL
        T.root = z
elseif z.key < y.key
        y.left = z
else y.right = z
```

Manipulering af BST'er

Insertion eksempel

Lad os prøve at indsætte nøglerne $\langle 12, 18, 5, 2, 15, 9, 17, 19, 13 \rangle$:

- Tree-Insert(T, 12) — kommer ind som rod
- Tree-Insert(T, 18)
- Tree-Insert(T, 5)
- Tree-Insert(T, 2)
- Tree-Insert(T, 15)
- Tree-Insert(T, 9)
- Tree-Insert(T, 17)



Tree-Insert(T, z)

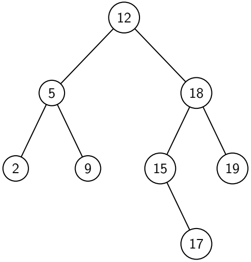
```
x = T.root
y = NIL
while x ≠ NIL
    y = x
    if z.key < x.key
        x = x.left
    else x = x.right
z.p = y
if y == NIL
    T.root = z
elseif z.key < y.key
    y.left = z
else y.right = z
```

Manipulering af BST'er

Insertion eksempel

Lad os prøve at indsætte nøglerne $\langle 12, 18, 5, 2, 15, 9, 17, 19, 13 \rangle$:

- Tree-Insert(T, 12) — kommer ind som rod
- Tree-Insert(T, 18)
- Tree-Insert(T, 5)
- Tree-Insert(T, 2)
- Tree-Insert(T, 15)
- Tree-Insert(T, 9)
- Tree-Insert(T, 17)
- Tree-Insert(T, 19)



Tree-Insert(T, z)

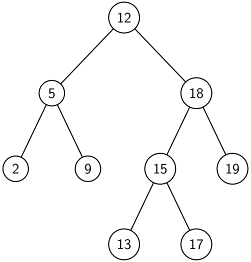
```
x = T.root
y = NIL
while x ≠ NIL
    y = x
    if z.key < x.key
        x = x.left
    else x = x.right
z.p = y
if y == NIL
    T.root = z
elseif z.key < y.key
    y.left = z
else y.right = z
```

Manipulering af BST'er

Insertion eksempel

Lad os prøve at indsætte nøglerne $\langle 12, 18, 5, 2, 15, 9, 17, 19, 13 \rangle$:

- Tree-Insert(T, 12) — kommer ind som rod
- Tree-Insert(T, 18)
- Tree-Insert(T, 5)
- Tree-Insert(T, 2)
- Tree-Insert(T, 15)
- Tree-Insert(T, 9)
- Tree-Insert(T, 17)
- Tree-Insert(T, 19)
- Tree-Insert(T, 13)



Tree-Insert(T, z)

```
x = T.root
y = NIL
while x ≠ NIL
    y = x
    if z.key < x.key
        x = x.left
    else x = x.right
z.p = y
if y == NIL
    T.root = z
elseif z.key < y.key
    y.left = z
else y.right = z
```

Manipulering af BST'er

Insertion worst case

Men vent... Hvad nu, hvis vi havde indsat elementerne i sorteret rækkefølge?

Tree-Insert(T, z)

```
x = T.root
y = NIL
while x ≠ NIL
    y = x
    if z.key < x.key
        x = x.left
    else x = x.right
z.p = y
if y == NIL
    T.root = z
elseif z.key < y.key
    y.left = z
else y.right = z
```

Manipulering af BST'er

Insertion worst case

Men vent... Hvad nu, hvis vi havde indsat elementerne i sorteret rækkefølge?

- Tree-Insert(T, 2)

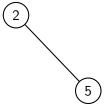
Tree-Insert(T, z)

```
x = T.root
y = NIL
while x ≠ NIL
    y = x
    if z.key < x.key
        x = x.left
    else x = x.right
z.p = y
if y == NIL
    T.root = z
elseif z.key < y.key
    y.left = z
else y.right = z
```

Manipulering af BST'er

Insertion worst case

Men vent... Hvad nu, hvis vi havde indsat elementerne i sorteret rækkefølge?



- Tree-Insert(T, 2)
- Tree-Insert(T, 5)

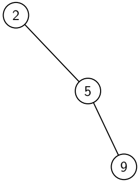
Tree-Insert(T, z)

```
x = T.root
y = NIL
while x ≠ NIL
    y = x
    if z.key < x.key
        x = x.left
    else x = x.right
z.p = y
if y == NIL
    T.root = z
elseif z.key < y.key
    y.left = z
else y.right = z
```

Manipulering af BST'er

Insertion worst case

Men vent... Hvad nu, hvis vi havde indsat elementerne i sorteret rækkefølge?



- Tree-Insert$(T, 2)$
- Tree-Insert$(T, 5)$
- Tree-Insert$(T, 9)$

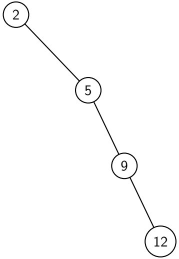
Tree-Insert(T, z)

```
x = T.root
y = NIL
while x ≠ NIL
    y = x
    if z.key < x.key
        x = x.left
    else x = x.right
z.p = y
if y == NIL
    T.root = z
elseif z.key < y.key
    y.left = z
else y.right = z
```

Manipulering af BST'er

Insertion worst case

Men vent... Hvad nu, hvis vi havde indsat elementerne i sorteret rækkefølge?



- Tree-Insert$(T, 2)$
- Tree-Insert$(T, 5)$
- Tree-Insert$(T, 9)$
- Tree-Insert$(T, 12)$

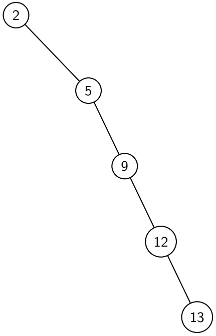
Tree-Insert(T, z)

```
x = T.root
y = NIL
while x ≠ NIL
    y = x
    if z.key < x.key
        x = x.left
    else x = x.right
z.p = y
if y == NIL
    T.root = z
elseif z.key < y.key
    y.left = z
else y.right = z
```

Manipulering af BST'er

Insertion worst case

Men vent... Hvad nu, hvis vi havde indsat elementerne i sorteret rækkefølge?



- Tree-Insert(T, 2)
- Tree-Insert(T, 5)
- Tree-Insert(T, 9)
- Tree-Insert(T, 12)
- Tree-Insert(T, 13)

Tree-Insert(T, z)

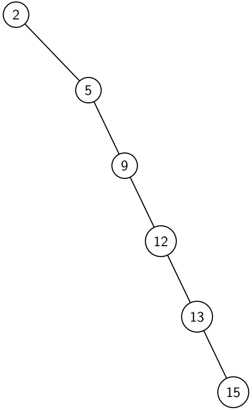
```
x = T.root
y = NIL
while x ≠ NIL
    y = x
    if z.key < x.key
        x = x.left
    else x = x.right
z.p = y
if y == NIL
    T.root = z
elseif z.key < y.key
    y.left = z
else y.right = z
```

Manipulering af BST'er

Insertion worst case

Men vent... Hvad nu, hvis vi havde indsat elementerne i sorteret rækkefølge?

- Tree-Insert(T, 2)
- Tree-Insert(T, 5)
- Tree-Insert(T, 9)
- Tree-Insert(T, 12)
- Tree-Insert(T, 13)
- Tree-Insert(T, 15)



Tree-Insert(T, z)

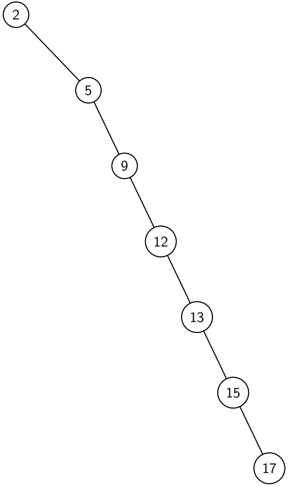
```
x = T.root
y = NIL
while x ≠ NIL
    y = x
    if z.key < x.key
        x = x.left
    else x = x.right
z.p = y
if y == NIL
    T.root = z
elseif z.key < y.key
    y.left = z
else y.right = z
```

Manipulering af BST'er

Insertion worst case

Men vent... Hvad nu, hvis vi havde indsat elementerne i sorteret rækkefølge?

- Tree-Insert(T, 2)
- Tree-Insert(T, 5)
- Tree-Insert(T, 9)
- Tree-Insert(T, 12)
- Tree-Insert(T, 13)
- Tree-Insert(T, 15)
- Tree-Insert(T, 17)



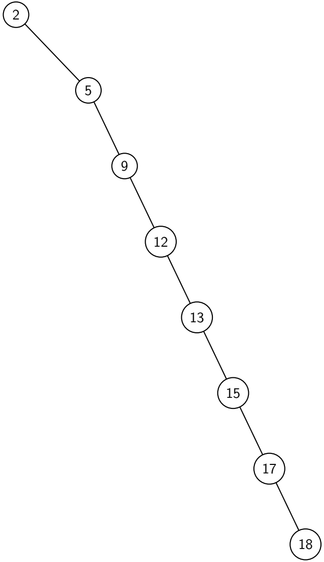
Tree-Insert(T, z)

```
x = T.root
y = NIL
while x ≠ NIL
    y = x
    if z.key < x.key
        x = x.left
    else x = x.right
z.p = y
if y == NIL
    T.root = z
elseif z.key < y.key
    y.left = z
else y.right = z
```

Manipulering af BST'er

Insertion worst case

Men vent... Hvad nu, hvis vi havde indsat elementerne i sorteret rækkefølge?



- Tree-Insert(T, 2)
- Tree-Insert(T, 5)
- Tree-Insert(T, 9)
- Tree-Insert(T, 12)
- Tree-Insert(T, 13)
- Tree-Insert(T, 15)
- Tree-Insert(T, 17)
- Tree-Insert(T, 18)
  NOOOOOOOOO...

Tree-Insert(T, z)

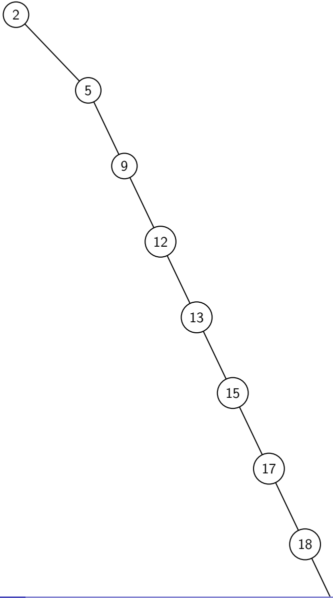
```
x = T.root
y = NIL
while x ≠ NIL
    y = x
    if z.key < x.key
        x = x.left
    else x = x.right
z.p = y
if y == NIL
    T.root = z
elseif z.key < y.key
    y.left = z
else y.right = z
```

Manipulering af BST'er

Insertion worst case

Men vent... Hvad nu, hvis vi havde indsat elementerne i sorteret rækkefølge?

- Tree-Insert(T, 2)
- Tree-Insert(T, 5)
- Tree-Insert(T, 9)
- Tree-Insert(T, 12)
- Tree-Insert(T, 13)
- Tree-Insert(T, 15)
- Tree-Insert(T, 17)
- Tree-Insert(T, 18)
  NOOOOOOOOO...
- Tree-Insert(T, 19)
  ...OOOOOOO!!



Tree-Insert(T, z)

```
x = T.root
y = NIL
while x ≠ NIL
    y = x
    if z.key < x.key
        x = x.left
    else x = x.right
z.p = y
if y == NIL
    T.root = z
elseif z.key < y.key
    y.left = z
else y.right = z
```

Manipulering af BST'er

Insertion worst case

Med andre ord — hvis vores data kommer uhensigtsmæssigt ind (f.eks. sorteret), så bliver højden på vores træ n, og så har vi basically bare en linked liste.

Manipulering af BST'er

Insertion worst case

Med andre ord — hvis vores data kommer uhensigtsmæssigt ind (f.eks. sorteret), så bliver højden på vores træ n, og så har vi basically bare en linked liste.

Dermed bliver alle vores tidligere operationer også $O(n)$.

Manipulering af BST'er

Insertion worst case

Med andre ord — hvis vores data kommer uhensigtsmæssigt ind (f.eks. sorteret), så bliver højden på vores træ n, og så har vi basically bare en linked liste.

Dermed bliver alle vores tidligere operationer også $O(n)$.

Det kan heldigvis vises, at hvis data kommer ind i tilfældig rækkefølge, så er den forventede højde på træet $O(\log n)$.

Manipulering af BST'er

Insertion worst case

Med andre ord — hvis vores data kommer uhensigtsmæssigt ind (f.eks. sorteret), så bliver højden på vores træ n, og så har vi basically bare en linked liste.

Dermed bliver alle vores tidligere operationer også $O(n)$.

Det kan heldigvis vises, at hvis data kommer ind i tilfældig rækkefølge, så er den forventede højde på træet $O(\log n)$.

Og næste gang ser vi på en modifikation af BST'er, der garanterer, at træet er balanceret.

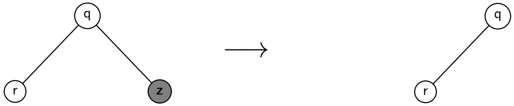
Manipulering af BST'er

Deletion

Hvor insertion er en relativt simpel procedure, så kræver deletion lidt flere tricks. Lad os sige, at vi vil slette en knude z:

Hvor insertion er en relativt simpel procedure, så kræver deletion lidt flere tricks. Lad os sige, at vi vil slette en knude z:

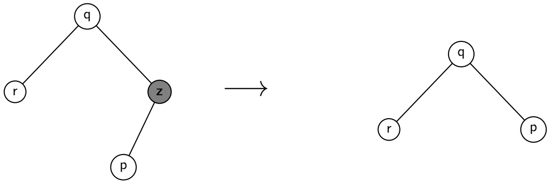
- Det er simpelt nok, hvis z er et blad — så sletter vi den bare



Manipulering af BST'er
Deletion

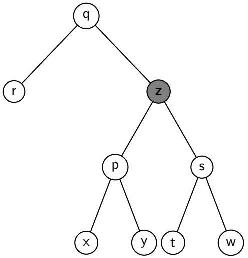
Hvor insertion er en relativt simpel procedure, så kræver deletion lidt flere tricks. Lad os sige, at vi vil slette en knude z:

- Det er simpelt nok, hvis z er et blad — så sletter vi den bare
- Hvis z kun har 1 barn, så er det også nemt — så flytter vi bare barnet op på z's plads

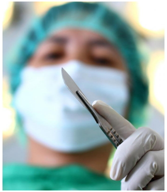


Hvor insertion er en relativt simpel procedure, så kræver deletion lidt flere tricks. Lad os sige, at vi vil slette en knude z:

- Det er simpelt nok, hvis z er et blad — så sletter vi den bare
- Hvis z kun har 1 barn, så er det også nemt — så flytter vi bare barnet op på z's plads
- Men hvis z har 2 børn, så skal vi ind med en skalpel og rode lidt



$\longrightarrow$



Manipulering af BST'er

Deletion når der er 2 børn

Vi kigger nu på, hvordan vi sletter en knude z, når den har 2 børn (drabeligt!)

Manipulering af BST'er

Deletion når der er 2 børn

Vi kigger nu på, hvordan vi sletter en knude z, når den har 2 børn (drabeligt!)

- Først skal vi finde z's successor y. Vi ved, at y ligger i z's højre sub-træ og ikke har noget venstre barn

Manipulering af BST'er

Deletion når der er 2 børn

Vi kigger nu på, hvordan vi sletter en knude z, når den har 2 børn (drabeligt!)

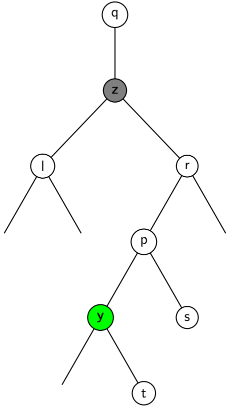
- Først skal vi finde z's successor y. Vi ved, at y ligger i z's højre sub-træ og ikke har noget venstre barn
  - Hvorfor?

Manipulering af BST'er

Deletion når der er 2 børn

Vi kigger nu på, hvordan vi sletter en knude z, når den har 2 børn (drabeligt!)

- Først skal vi finde z's successor y. Vi ved, at y ligger i z's højre sub-træ og ikke har noget venstre barn
    - Hvorfor?
    - y skal være større end z, og dermed er den i højre sub-træ

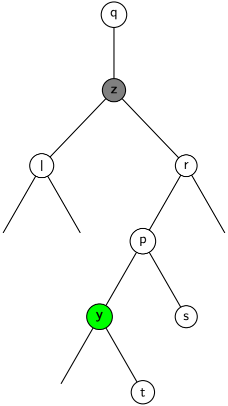


Manipulering af BST'er

Deletion når der er 2 børn

Vi kigger nu på, hvordan vi sletter en knude z, når den har 2 børn (drabeligt!)

- Først skal vi finde z's successor y. Vi ved, at y ligger i z's højre sub-træ og ikke har noget venstre barn
  - Hvorfor?
  - y skal være større end z, og dermed er den i højre sub-træ
  - Hvis y har et venstre barn, så er det mindre end y selv, og så kan y ikke være z's successor

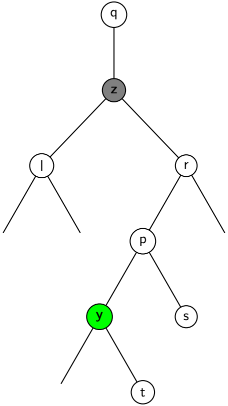


Manipulering af BST'er

Deletion når der er 2 børn

Vi kigger nu på, hvordan vi sletter en knude z, når den har 2 børn (drabeligt!)

- Først skal vi finde z's successor y. Vi ved, at y ligger i z's højre sub-træ og ikke har noget venstre barn
  - Hvorfor?
  - y skal være større end z, og dermed er den i højre sub-træ
  - Hvis y har et venstre barn, så er det mindre end y selv, og så kan y ikke være z's successor
- y skal nu sættes ind på z's plads, men...

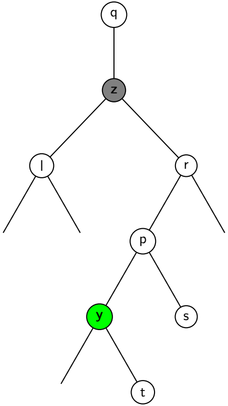


Manipulering af BST'er

Deletion når der er 2 børn

Vi kigger nu på, hvordan vi sletter en knude z, når den har 2 børn (drabeligt!)

- Først skal vi finde z's successor y. Vi ved, at y ligger i z's højre sub-træ og ikke har noget venstre barn
  - Hvorfor?
  - y skal være større end z, og dermed er den i højre sub-træ
  - Hvis y har et venstre barn, så er det mindre end y selv, og så kan y ikke være z's successor
- y skal nu sættes ind på z's plads, men...
  - Hvis y er z's højre barn, flyttes det op på z's plads, og vi lader dets højre barn være

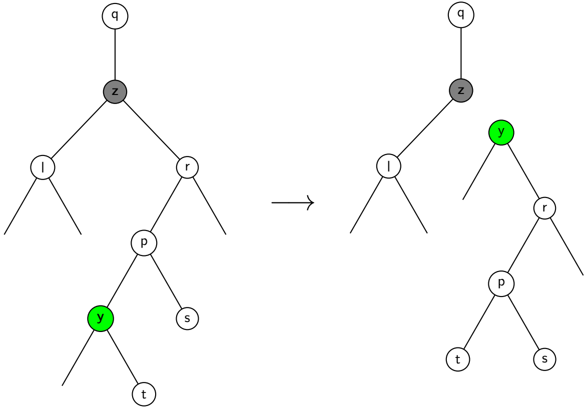


Manipulering af BST'er

Deletion når der er 2 børn

Vi kigger nu på, hvordan vi sletter en knude z, når den har 2 børn (drabeligt!)

- Først skal vi finde z's successor y. Vi ved, at y ligger i z's højre sub-træ og ikke har noget venstre barn
  - Hvorfor?
  - y skal være større end z, og dermed er den i højre sub-træ
  - Hvis y har et venstre barn, så er det mindre end y selv, og så kan y ikke være z's successor
- y skal nu sættes ind på z's plads, men...
  - Hvis y er z's højre barn, flyttes det op på z's plads, og vi lader dets højre barn være
  - Hvis y ikke er z's højre barn, skal vi først erstatte y med dets eget højre barn, og herefter erstatte z med y

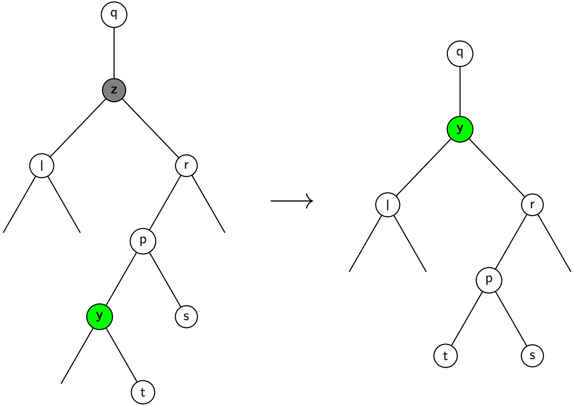


Manipulering af BST'er

Deletion når der er 2 børn

Vi kigger nu på, hvordan vi sletter en knude z, når den har 2 børn (drabeligt!)

- Først skal vi finde z's successor y. Vi ved, at y ligger i z's højre sub-træ og ikke har noget venstre barn
  - Hvorfor?
  - y skal være større end z, og dermed er den i højre sub-træ
  - Hvis y har et venstre barn, så er det mindre end y selv, og så kan y ikke være z's successor
- y skal nu sættes ind på z's plads, men...
  - Hvis y er z's højre barn, flyttes det op på z's plads, og vi lader dets højre barn være
  - Hvis y ikke er z's højre barn, skal vi først erstatte y med dets eget højre barn, og herefter erstatte z med y



Manipulering af BST'er

Deletion pseudo-kode

Vi prøver nu at tage et kig på koden for Tree-Delete(T, z).

Tree-Delete(T, z)

```
if z.left == NIL
    Transplant(T, z, z.right)
elseif z.right == NIL
    Transplant(T, z, z.left)
else y = Tree-Minimum(z.right)
    if y ≠ z.right
        Transplant(T, y, y.right)
        y.right = z.right
        y.right.p = y
    Transplant(T, z, y)
    y.left = z.left
    y.left.p = y
```

Manipulering af BST'er

Deletion pseudo-kode

Vi prøver nu at tage et kig på koden for Tree-Delete(T, z).

- Først, bemærk at vi har en sub-procedure Transplant(T, u, v), som erstatter knuden u med knuden v i træet T

Tree-Delete(T, z)

```
if z.left == NIL
    Transplant(T, z, z.right)
elseif z.right == NIL
    Transplant(T, z, z.left)
else y = Tree-Minimum(z.right)
    if y ≠ z.right
        Transplant(T, y, y.right)
        y.right = z.right
        y.right.p = y
    Transplant(T, z, y)
    y.left = z.left
    y.left.p = y
```

Manipulering af BST'er

Deletion pseudo-kode

Vi prøver nu at tage et kig på koden for Tree-Delete(T, z).

- Først, bemærk at vi har en sub-procedure Transplant(T, u, v), som erstatter knuden u med knuden v i træet T
- Linie 1-4 dækker de tilfælde, hvor z kun har et barn, og vi bare skal flytte barnet op

Tree-Delete(T, z)

```
1   if z.left == NIL
2       Transplant(T, z, z.right)
3   elseif z.right == NIL
4       Transplant(T, z, z.left)
5   else y = Tree-Minimum(z.right)
6       if y ≠ z.right
7           Transplant(T, y, y.right)
8           y.right = z.right
9           y.right.p = y
10      Transplant(T, z, y)
11      y.left = z.left
12      y.left.p = y
```

Manipulering af BST'er

Deletion pseudo-kode

Vi prøver nu at tage et kig på koden for Tree-Delete(T, z).

- Først, bemærk at vi har en sub-procedure Transplant(T, u, v), som erstatter knuden u med knuden v i træet T
- Linie 1-4 dækker de tilfælde, hvor z kun har et barn, og vi bare skal flytte barnet op
- Hvis z har to børn finder vi dets successor y i linie 5

Tree-Delete(T, z)

```
1   if z.left == NIL
2       Transplant(T, z, z.right)
3   elseif z.right == NIL
4       Transplant(T, z, z.left)
5   else y = Tree-Minimum(z.right)
6       if y ≠ z.right
7           Transplant(T, y, y.right)
8           y.right = z.right
9           y.right.p = y
10      Transplant(T, z, y)
11      y.left = z.left
12      y.left.p = y
```

Manipulering af BST'er

Deletion pseudo-kode

Vi prøver nu at tage et kig på koden for Tree-Delete(T, z).

- Først, bemærk at vi har en sub-procedure Transplant(T, u, v), som erstatter knuden u med knuden v i træet T
- Linie 1-4 dækker de tilfælde, hvor z kun har et barn, og vi bare skal flytte barnet op
- Hvis z har to børn finder vi dets successor y i linie 5
- Linie 6-9 dækker den case, hvor y ikke er z's højre barn

Tree-Delete(T, z)

```
1   if z.left == NIL
2       Transplant(T, z, z.right)
3   elseif z.right == NIL
4       Transplant(T, z, z.left)
5   else y = Tree-Minimum(z.right)
6       if y ≠ z.right
7           Transplant(T, y, y.right)
8           y.right = z.right
9           y.right.p = y
10      Transplant(T, z, y)
11      y.left = z.left
12      y.left.p = y
```

Manipulering af BST'er

Deletion pseudo-kode

Vi prøver nu at tage et kig på koden for Tree-Delete(T, z).

- Først, bemærk at vi har en sub-procedure Transplant(T, u, v), som erstatter knuden u med knuden v i træet T
- Linie 1-4 dækker de tilfælde, hvor z kun har et barn, og vi bare skal flytte barnet op
- Hvis z har to børn finder vi dets successor y i linie 5
- Linie 6-9 dækker den case, hvor y ikke er z's højre barn
  - Linie 7 erstatter y med y's højre barn

Tree-Delete(T, z)

```
1   if z.left == NIL
2       Transplant(T, z, z.right)
3   elseif z.right == NIL
4       Transplant(T, z, z.left)
5   else y = Tree-Minimum(z.right)
6       if y ≠ z.right
7           Transplant(T, y, y.right)
8           y.right = z.right
9           y.right.p = y
10      Transplant(T, z, y)
11      y.left = z.left
12      y.left.p = y
```

Manipulering af BST'er

Deletion pseudo-kode

Vi prøver nu at tage et kig på koden for Tree-Delete(T, z).

- Først, bemærk at vi har en sub-procedure Transplant(T, u, v), som erstatter knuden u med knuden v i træet T
- Linie 1-4 dækker de tilfælde, hvor z kun har et barn, og vi bare skal flytte barnet op
- Hvis z har to børn finder vi dets successor y i linie 5
- Linie 6-9 dækker den case, hvor y ikke er z's højre barn
  - Linie 7 erstatter y med y's højre barn
  - Linie 8-9 flytter pointers, så y's højre barn nu er z's højre barn

Tree-Delete(T, z)

```
1   if z.left == NIL
2       Transplant(T, z, z.right)
3   elseif z.right == NIL
4       Transplant(T, z, z.left)
5   else y = Tree-Minimum(z.right)
6       if y ≠ z.right
7           Transplant(T, y, y.right)
8           y.right = z.right
9           y.right.p = y
10      Transplant(T, z, y)
11      y.left = z.left
12      y.left.p = y
```

Manipulering af BST'er

Deletion pseudo-kode

Vi prøver nu at tage et kig på koden for Tree-Delete(T, z).

- Først, bemærk at vi har en sub-procedure Transplant(T, u, v), som erstatter knuden u med knuden v i træet T
- Linie 1-4 dækker de tilfælde, hvor z kun har et barn, og vi bare skal flytte barnet op
- Hvis z har to børn finder vi dets successor y i linie 5
- Linie 6-9 dækker den case, hvor y ikke er z's højre barn
  - Linie 7 erstatter y med y's højre barn
  - Linie 8-9 flytter pointers, så y's højre barn nu er z's højre barn
- I Linie 10-12 er y med sikkerhed z's højre barn, og vi blot kan erstatte z med y og rykke lidt pointers

Tree-Delete(T, z)

```
1   if z.left == NIL
2       Transplant(T, z, z.right)
3   elseif z.right == NIL
4       Transplant(T, z, z.left)
5   else y = Tree-Minimum(z.right)
6       if y ≠ z.right
7           Transplant(T, y, y.right)
8           y.right = z.right
9           y.right.p = y
10      Transplant(T, z, y)
11      y.left = z.left
12      y.left.p = y
```

Manipulering af BST'er

Deletion pseudo-kode

Vi prøver nu at tage et kig på koden for Tree-Delete(T, z).

- Først, bemærk at vi har en sub-procedure Transplant(T, u, v), som erstatter knuden u med knuden v i træet T
- Linie 1-4 dækker de tilfælde, hvor z kun har et barn, og vi bare skal flytte barnet op
- Hvis z har to børn finder vi dets successor y i linie 5
- Linie 6-9 dækker den case, hvor y ikke er z's højre barn
  - Linie 7 erstatter y med y's højre barn
  - Linie 8-9 flytter pointers, så y's højre barn nu er z's højre barn
- I Linie 10-12 er y med sikkerhed z's højre barn, og vi blot kan erstatte z med y og rykke lidt pointers
- Kompleksitet?

Tree-Delete(T, z)

```
1   if z.left == NIL
2       Transplant(T, z, z.right)
3   elseif z.right == NIL
4       Transplant(T, z, z.left)
5   else y = Tree-Minimum(z.right)
6       if y ≠ z.right
7           Transplant(T, y, y.right)
8           y.right = z.right
9           y.right.p = y
10      Transplant(T, z, y)
11      y.left = z.left
12      y.left.p = y
```

Manipulering af BST'er

Deletion pseudo-kode

Vi prøver nu at tage et kig på koden for Tree-Delete(T, z).

- Først, bemærk at vi har en sub-procedure Transplant(T, u, v), som erstatter knuden u med knuden v i træet T
- Linie 1-4 dækker de tilfælde, hvor z kun har et barn, og vi bare skal flytte barnet op
- Hvis z har to børn finder vi dets successor y i linie 5
- Linie 6-9 dækker den case, hvor y ikke er z's højre barn
  - Linie 7 erstatter y med y's højre barn
  - Linie 8-9 flytter pointers, så y's højre barn nu er z's højre barn
- I Linie 10-12 er y med sikkerhed z's højre barn, og vi blot kan erstatte z med y og rykke lidt pointers
- Kompleksitet? $O(h)$

Tree-Delete(T, z)

```
1   if z.left == NIL
2       Transplant(T, z, z.right)
3   elseif z.right == NIL
4       Transplant(T, z, z.left)
5   else y = Tree-Minimum(z.right)
6       if y ≠ z.right
7           Transplant(T, y, y.right)
8           y.right = z.right
9           y.right.p = y
10      Transplant(T, z, y)
11      y.left = z.left
12      y.left.p = y
```

Manipulering af BST'er

Transplant proceduren

For god ordens skyld tager vi også lige Transplant-proceduren.

Transplant(T, u, v)

```
if u.p == NIL
        T.root = v
elseif u == u.p.left
        u.p.left = v
else u.p.right = v
if v ≠ NIL
        v.p = u.p
```

Manipulering af BST'er

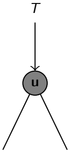
Transplant proceduren

For god ordens skyld tager vi også lige Transplant-proceduren.

Transplant(T, u, v)

```
1  if u.p == NIL
2        T.root = v
3  elseif u == u.p.left
4        u.p.left = v
5  else u.p.right = v
6  if v ≠ NIL
7        v.p = u.p
```

- Linie 1-2 tjekker, om vi skal opdatere T's rod (hvis u ikke har nogen parent)



Manipulering af BST'er

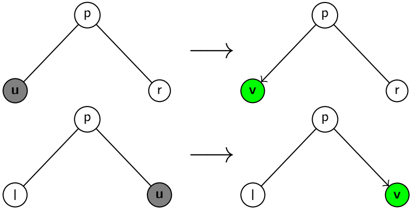
Transplant proceduren

For god ordens skyld tager vi også lige Transplant-proceduren.

Transplant(T, u, v)

```
1  if u.p == NIL
2      T.root = v
3  elseif u == u.p.left
4      u.p.left = v
5  else u.p.right = v
6  if v ≠ NIL
7      v.p = u.p
```

- Linie 1-2 tjekker, om vi skal opdatere T's rod (hvis u ikke har nogen parent)
- Linie 3-5 tjekker, om u er venstre eller højre barn af sin parent og indsætter v på rette plads



Manipulering af BST'er

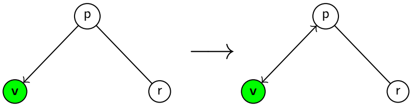
Transplant proceduren

For god ordens skyld tager vi også lige Transplant-proceduren.

Transplant(T, u, v)

```
1  if u.p == NIL
2      T.root = v
3  elseif u == u.p.left
4      u.p.left = v
5  else u.p.right = v
6  if v ≠ NIL
7      v.p = u.p
```

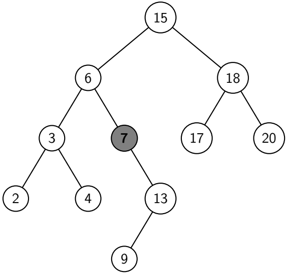
- Linie 1-2 tjekker, om vi skal opdatere T's rod (hvis u ikke har nogen parent)
- Linie 3-5 tjekker, om u er venstre eller højre barn af sin parent og indsætter v på rette plads
- Linie 6-7 sætter v's parent til at være u's parent, hvis ikke v er NIL



Deletion i et BST

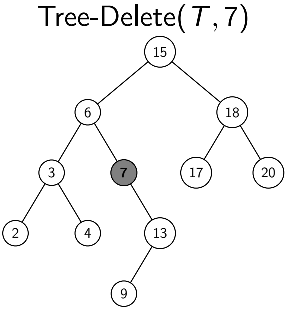
Eksempel

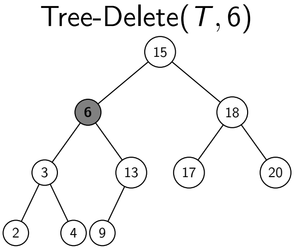
Tree-Delete($T, 7$)



Deletion i et BST

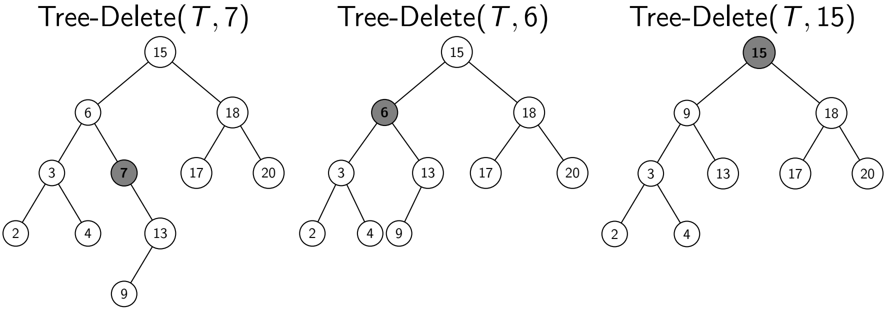
Eksempel





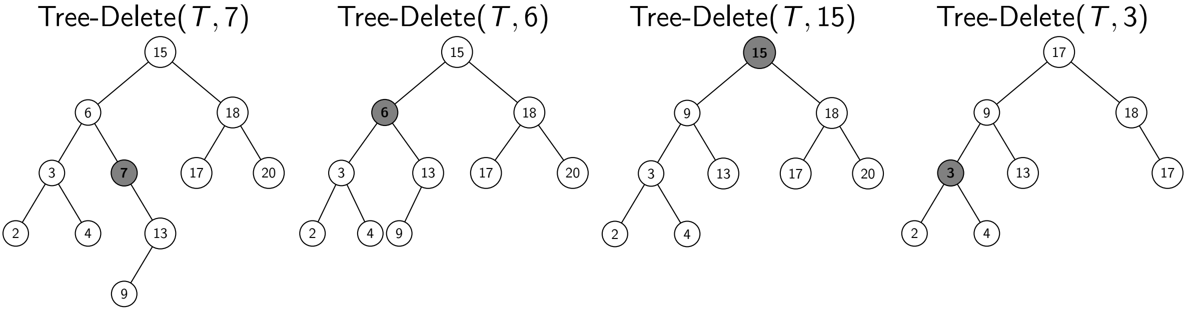
Deletion i et BST

Eksempel



Deletion i et BST

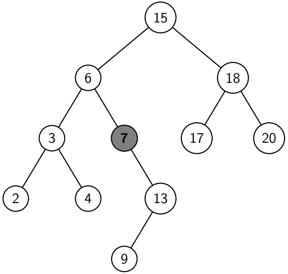
Eksempel



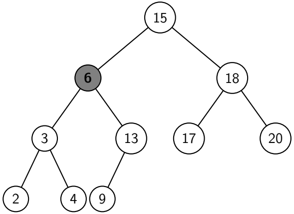
Deletion i et BST

Eksempel

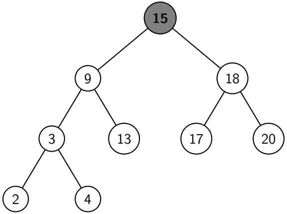
Tree-Delete(T, 7)



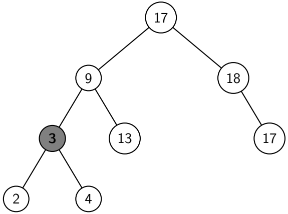
Tree-Delete(T, 6)



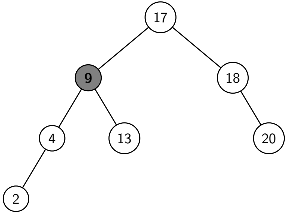
Tree-Delete(T, 15)



Tree-Delete(T, 3)



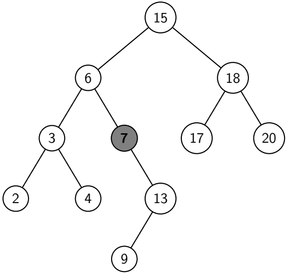
Tree-Delete(T, 9)



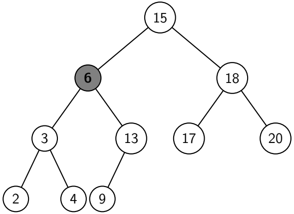
Deletion i et BST

Eksempel

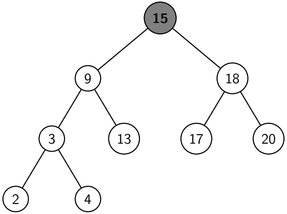
Tree-Delete(T, 7)



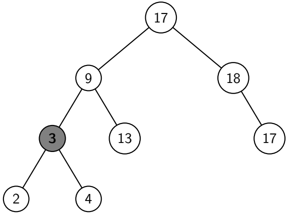
Tree-Delete(T, 6)



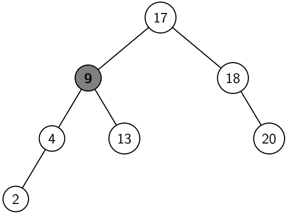
Tree-Delete(T, 15)



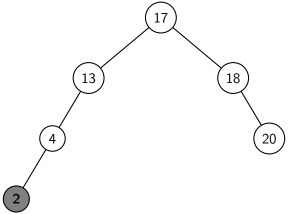
Tree-Delete(T, 3)



Tree-Delete(T, 9)



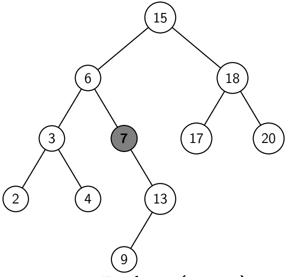
Tree-Delete(T, 2)



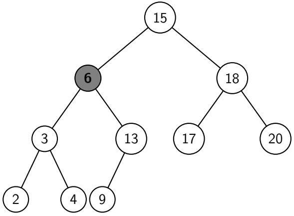
Deletion i et BST

Eksempel

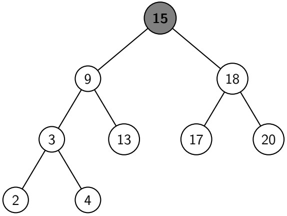
Tree-Delete(T, 7)



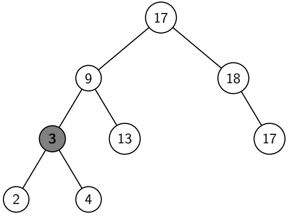
Tree-Delete(T, 6)



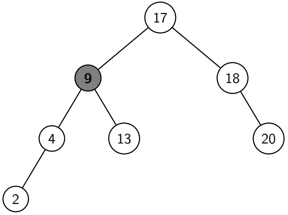
Tree-Delete(T, 15)



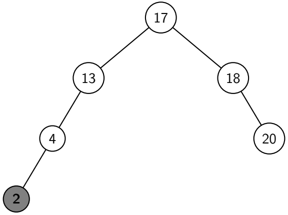
Tree-Delete(T, 3)



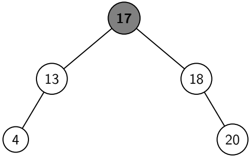
Tree-Delete(T, 9)



Tree-Delete(T, 2)



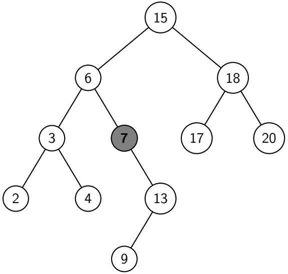
Tree-Delete(T, 17)



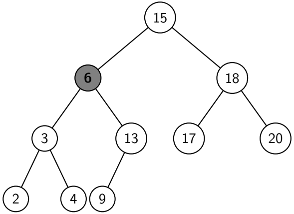
Deletion i et BST

Eksempel

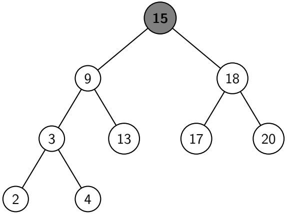
Tree-Delete(T, 7)



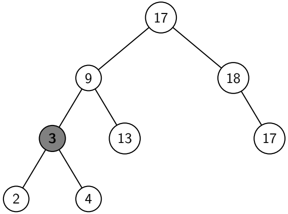
Tree-Delete(T, 6)



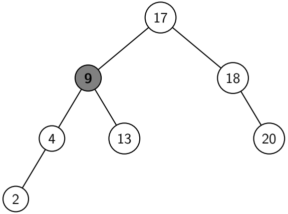
Tree-Delete(T, 15)



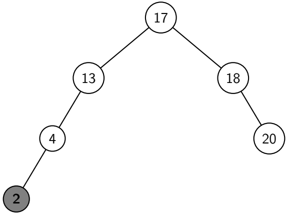
Tree-Delete(T, 3)



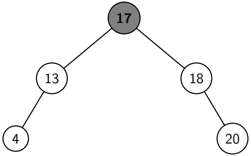
Tree-Delete(T, 9)



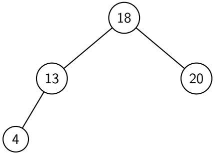
Tree-Delete(T, 2)



Tree-Delete(T, 17)



Finito



Opsamling

Vi har brugt hele dagen på at snakke om binære søgetræer.

Opsamling

Vi har brugt hele dagen på at snakke om binære søgetræer.

- Det er en smuk datastruktur

Opsamling

Vi har brugt hele dagen på at snakke om binære søgetræer.

- Det er en smuk datastruktur
- De kan bruges som både dictionaries (key/value pairs) og priority queues

Opsamling

Vi har brugt hele dagen på at snakke om binære søgetræer.

- Det er en smuk datastruktur
- De kan bruges som både dictionaries (key/value pairs) og priority queues
- Alle operationer kører i tid proportionelt med højden h

Opsamling

Vi har brugt hele dagen på at snakke om binære søgetræer.

- Det er en smuk datastruktur
- De kan bruges som både dictionaries (key/value pairs) og priority queues
- Alle operationer kører i tid proportionelt med højden h
  - For et balanceret træ med n knuder er højden $\sim \log n$

Opsamling

Vi har brugt hele dagen på at snakke om binære søgetræer.

- Det er en smuk datastruktur
- De kan bruges som både dictionaries (key/value pairs) og priority queues
- Alle operationer kører i tid proportionelt med højden h
  - For et balanceret træ med n knuder er højden $\sim \log n$
  - I værste fald er højden dog ligeså stor som antallet af knuder

Opsamling

Vi har brugt hele dagen på at snakke om binære søgetræer.

- Det er en smuk datastruktur
- De kan bruges som både dictionaries (key/value pairs) og priority queues
- Alle operationer kører i tid proportionelt med højden h
  - For et balanceret træ med n knuder er højden $\sim \log n$
  - I værste fald er højden dog ligeså stor som antallet af knuder
- Vi har også set, at ChatGPT kan være meget misvisende

Næste gang. . .

. . . er lidt anderledes

I næste uge har vi ikke nogen forelæsning. I stedet, så vil der være en såkaldt 'self-study' session, hvor I får mulighed for at prøve kræfter med et eksamenssæt.

- I får 2 timer til at lave et halvt eksamenssæt (den rigtige eksamen er 4 timer)
- Så bruger vi tiden bagefter på at gennemgå løsningen sammen
- I kan enten komme her til lokalet og lave det i en setting, der minder om den rigtig eksamenssituation (jeg vil være her, og kan hjælpe lidt, selvom det selvfølgelig bryder med eksamenssimuleringen)
- Eller I kan lave det i jeres gruppe-lokaler, og så bare komme herop kl 14.30, hvis I vil være med til at gennemgå løsningen

Tak for i dag!

Flere exercises..

Den bedste måde ikke at snyde sig selv på er lave exercises!