Red-Black Trees

Algorithms and Datastructures, F25, Lecture 6

Andreas Holck Høeg-Petersen

Department of Computer Science
Aalborg University

March 27, 2025

Opdateringer

AALBORG UNIVERSITET

- Næste programmeringsopgave er ude — har I alle set den?

Opdateringer

- Næste programmeringsopgave er ude — har I alle set den?
  - Der vil være lidt ekstra tid til exercises i dag, og det er blandt andet, så I eventuelt kan arbejde lidt med opgaven

Outline

1. Introduktion til Red-Black Trees
2. Insertion i Red-Black trees
3. Exercises
4. Deletion i RBT

Outline

1. Introduktion til Red-Black Trees
2. Insertion i Red-Black trees
3. Exercises
4. Deletion i RBT

Recap

Udfordringer ved BST'er

Vi så til sidste forelæsning på binære søgetræer

Recap

Udfordringer ved BST'er

Vi så til sidste forelæsning på binære søgetræer

- De var super seje

Recap

Udfordringer ved BST'er

Vi så til sidste forelæsning på binære søgetræer

- De var super seje
- De understøttede alle vores elskede operationer — Insert, Delete, Search, Maximum/Minimum og lignende — og kunne gøre det i $\log n$ tid

Recap

Udfordringer ved BST'er

Vi så til sidste forelæsning på binære søgetræer

- De var super seje
- De understøttede alle vores elskede operationer — Insert, Delete, Search, Maximum/Minimum og lignende — og kunne gøre det i $\log n$ tid
- Eller det vil sige...

Recap

Udfordringer ved BST'er

Vi så til sidste forelæsning på binære søgetræer

- De var super seje
- De understøttede alle vores elskede operationer — Insert, Delete, Search, Maximum/Minimum og lignende — og kunne gøre det i $\log n$ tid
- Eller det vil sige...
  - Alle procedurerne kørte i tid proportionelt med træets højde h

Recap

Udfordringer ved BST'er

Vi så til sidste forelæsning på binære søgetræer

- De var super seje
- De understøttede alle vores elskede operationer — Insert, Delete, Search, Maximum/Minimum og lignende — og kunne gøre det i $\log n$ tid
- Eller det vil sige...
  - Alle procedurerne kørte i tid proportionelt med træets højde h
  - For et balanceret træ med n knuder har vi at $h = \Theta(\log n)$

Recap

Udfordringer ved BST'er

Vi så til sidste forelæsning på binære søgetræer

- De var super seje
- De understøttede alle vores elskede operationer — Insert, Delete, Search, Maximum/Minimum og lignende — og kunne gøre det i $\log n$ tid
- Eller det vil sige...
  - Alle procedurerne kørte i tid proportionelt med træets højde h
  - For et balanceret træ med n knuder har vi at $h = \Theta(\log n)$
  - Men i worst case er $h = n$

Recap

Udfordringer ved BST'er

Vi så til sidste forelæsning på binære søgetræer

- De var super seje
- De understøttede alle vores elskede operationer — Insert, Delete, Search, Maximum/Minimum og lignende — og kunne gøre det i $\log n$ tid
- Eller det vil sige...
  - Alle procedurerne kørte i tid proportionelt med træets højde h
  - For et balanceret træ med n knuder har vi at $h = \Theta(\log n)$
  - Men i worst case er $h = n$
- Kan vi gøre noget ved det?

Bedre tider

See what I did there??

Red-Black Trees

Egenskaber

Vi introducerer nu en ny datastruktur, som er en lettere modificeret udgave BST'er.

Red-Black Trees

Egenskaber

Vi introducerer nu en ny datastruktur, som er en lettere modificeret udgave BST'er.

Red-Black Trees
Egenskaber

Vi introducerer nu en ny datastruktur, som er en lettere modificeret udgave BST'er.

- Alle knuder indeholder nu også en attribut *color* $\in \{\text{RED}, \text{BLACK}\}$

Red-Black Trees

Egenskaber

Vi introducerer nu en ny datastruktur, som er en lettere modificeret udgave BST'er.

- Alle knuder indeholder nu også en attribut *color* $\in \{\text{RED}, \text{BLACK}\}$
- Vi repræsenter nu blade som NIL-pointers (så alle knuder med nøgler er 'interne knuder')

Red-Black Trees

Egenskaber

Vi introducerer nu en ny datastruktur, som er en lettere modificeret udgave BST'er.

- Alle knuder indeholder nu også en attribut *color* $\in \{\text{RED}, \text{BLACK}\}$
- Vi repræsenter nu blade som NIL-pointers (så alle knuder med nøgler er 'interne knuder')
- Et red-black tree er et BST, som overholder følgende red-black egenskaber:

Red-Black Trees

Egenskaber

Vi introducerer nu en ny datastruktur, som er en lettere modificeret udgave BST'er.

- Alle knuder indeholder nu også en attribut *color* $\in \{\text{RED}, \text{BLACK}\}$
- Vi repræsenter nu blade som NIL-pointers (så alle knuder med nøgler er 'interne knuder')
- Et red-black tree er et BST, som overholder følgende red-black egenskaber:
  1. Alle knuder er enten røde eller sorte

Red-Black Trees

Egenskaber

Vi introducerer nu en ny datastruktur, som er en lettere modificeret udgave BST'er.

- Alle knuder indeholder nu også en attribut $\textit{color} \in \{\text{RED}, \text{BLACK}\}$
- Vi repræsenter nu blade som NIL-pointers (så alle knuder med nøgler er 'interne knuder')
- Et red-black tree er et BST, som overholder følgende red-black egenskaber:
    1. Alle knuder er enten røde eller sorte
    2. Roden er sort

Vi introducerer nu en ny datastruktur, som er en lettere modificeret udgave BST'er.

- Alle knuder indeholder nu også en attribut $color \in \{\text{RED}, \text{BLACK}\}$
- Vi repræsenter nu blade som NIL-pointers (så alle knuder med nøgler er 'interne knuder')
- Et red-black tree er et BST, som overholder følgende red-black egenskaber:
  1. Alle knuder er enten røde eller sorte
  2. Roden er sort
  3. Alle blade er er sorte

Red-Black Trees

Egenskaber

Vi introducerer nu en ny datastruktur, som er en lettere modificeret udgave BST'er.

- Alle knuder indeholder nu også en attribut *color* $\in \{\text{RED}, \text{BLACK}\}$
- Vi repræsenter nu blade som NIL-pointers (så alle knuder med nøgler er 'interne knuder')
- Et red-black tree er et BST, som overholder følgende red-black egenskaber:
    1. Alle knuder er enten røde eller sorte
    2. Roden er sort
    3. Alle blade er er sorte
    4. Hvis en knude er rød, så er begge dens børn sorte

Red-Black Trees

Egenskaber

Vi introducerer nu en ny datastruktur, som er en lettere modificeret udgave BST'er.

- Alle knuder indeholder nu også en attribut *color* $\in \{\text{RED}, \text{BLACK}\}$
- Vi repræsenter nu blade som NIL-pointers (så alle knuder med nøgler er 'interne knuder')
- Et red-black tree er et BST, som overholder følgende red-black egenskaber:
  1. Alle knuder er enten røde eller sorte
  2. Roden er sort
  3. Alle blade er er sorte
  4. Hvis en knude er rød, så er begge dens børn sorte
  5. Alle simple stier fra en knude til et blad i et af dets subtræer indeholder den samme mængde sorte knuder

Red-Black Trees

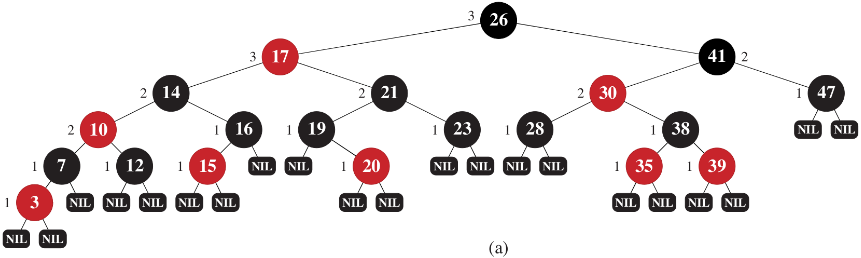
Egenskaber

Vi introducerer nu en ny datastruktur, som er en lettere modificeret udgave BST'er.

- Alle knuder indeholder nu også en attribut $color \in \{\mathrm{RED}, \mathrm{BLACK}\}$
- Vi repræsenter nu blade som NIL-pointers (så alle knuder med nøgler er 'interne knuder')
- Et red-black tree er et BST, som overholder følgende red-black egenskaber:
  1. Alle knuder er enten røde eller sorte
  2. Roden er sort
  3. Alle blade er er sorte
  4. Hvis en knude er rød, så er begge dens børn sorte
  5. Alle simple stier fra en knude til et blad i et af dets subtræer indeholder den samme mængde sorte knuder
- Vi kalder antallet af sorte knuder på en simpel sti fra en knude x (eksklusiv x selv) til et blad for knudes 'sorte højde', $bh(x)$

Red-Black Trees

Illustrationer

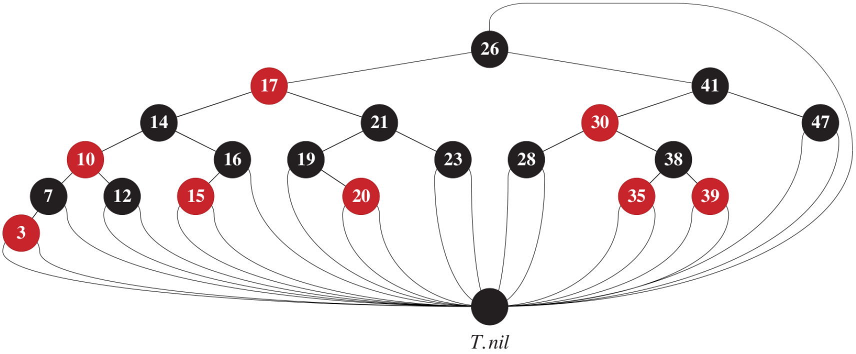


(a)

Et RBT hvor alle blade er sorte NIL-knuder

Red-Black Trees

Illustrationer

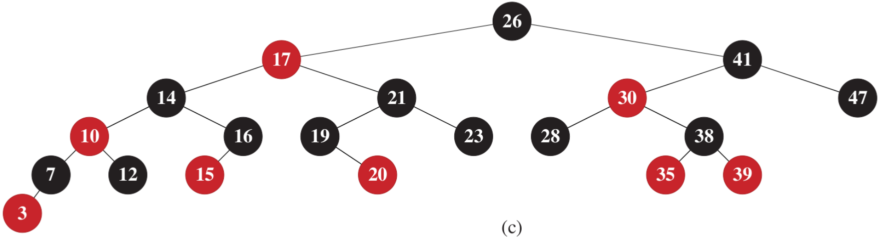


(b)

Vi bruger en sentinel NIL-knude til at repræsentere alle blade (og forældren til roden)

Red-Black Trees

Illustrationer



(c)

Men vi gider ikke tegne det hver gang, så man skal bare huske, at der egentlig altid er pointers til disse sorte NIL-knuder

Red-Black Trees

Garanti for højden

Med disse egenskaber på plads kan vi garantere, at højden for et RBT med n interne knuder højst er $2\log(n+1)$, hvilket er i $O(\log n)$. Hvorfor?

Red-Black Trees

Garanti for højden

Med disse egenskaber på plads kan vi garantere, at højden for et RBT med n interne knuder højst er $2\log(n+1)$, hvilket er i $O(\log n)$. Hvorfor?

Lemma 13.1: Højden på et RBT er højst $2\log(n+1)$.

- Først, bemærk at et sub-træ med rod i en knude x har mindst $2^{bh(x)} - 1$ interne knuder

$\square$

Red-Black Trees

Garanti for højden

Med disse egenskaber på plads kan vi garantere, at højden for et RBT med n interne knuder højst er $2\log(n+1)$, hvilket er i $O(\log n)$. Hvorfor?

Lemma 13.1: Højden på et RBT er højst $2\log(n+1)$.

- Først, bemærk at et sub-træ med rod i en knude x har mindst $2^{bh(x)} - 1$ interne knuder
- Dernæst, det følger af den 4. egenskab (en rød knude skal have 2 sorte børn), at mindst halvdelen af knuderne på enhver sti fra roden til et blad er sorte

$\square$

Red-Black Trees

Garanti for højjden

Med disse egenskaber på plads kan vi garantere, at højden for et RBT med n interne knuder højst er $2\log(n+1)$, hvilket er i $O(\log n)$. Hvorfor?

Lemma 13.1: Højden på et RBT er højst $2\log(n+1)$.

- Først, bemærk at et sub-træ med rod i en knude x har mindst $2^{bh(x)} - 1$ interne knuder
- Dernæst, det følger af den 4. egenskab (en rød knude skal have 2 sorte børn), at mindst halvdelen af knuderne på enhver sti fra roden til et blad er sorte
- Det giver os

$$
\begin{aligned}
n &\geq 2^{h/2} - 1 \\
n + 1 &\geq 2^{h/2} \\
\log(n+1) &\geq h/2
\end{aligned}
$$

$\square$

Red-Black Trees

Garanti for højden

Med disse egenskaber på plads kan vi garantere, at højden for et RBT med n interne knuder højst er $2\log(n+1)$, hvilket er i $O(\log n)$. Hvorfor?

Lemma 13.1: Højden på et RBT er højst $2\log(n+1)$.

- Først, bemærk at et sub-træ med rod i en knude x har mindst $2^{bh(x)} - 1$ interne knuder
- Dernæst, det følger af den 4. egenskab (en rød knude skal have 2 sorte børn), at mindst halvdelen af knuderne på enhver sti fra roden til et blad er sorte
- Det giver os

$$\begin{aligned} n &\geq 2^{h/2} - 1 \\ n + 1 &\geq 2^{h/2} \\ \log(n+1) &\geq h/2 \end{aligned}$$

- Hvilket vi også kan skrive som $h \leq 2\log(n+1)$

□

Rotationer

Basal operation

Eftersom insertion og deletion laver ændringer i træet, skal vi bruge en basal operation kaldet Left-Rotate[1] til at rette op på de eventuelle brud med RBT-egenskaberne.

Left-Rotate(T, x)

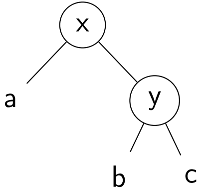
```
y = x.right
x.right = y.left
if y.left ≠ T.nil
    y.left.p = x
y.p = x.p
if x.p == T.nil
    T.root = y
elseif x == x.p.left
    x.p.left = y
else x.p.right = y
y.left = x
x.p = y
```

[1] Eller tilsvarende, Right-Rotate.

Rotationer

Basal operation

Eftersom insertion og deletion laver ændringer i træet, skal vi bruge en basal operation kaldet Left-Rotate[1] til at rette op på de eventuelle brud med RBT-egenskaberne.



Left-Rotate(T, x)

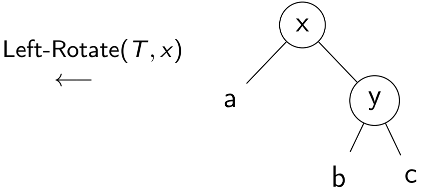
```
y = x.right
x.right = y.left
if y.left ≠ T.nil
    y.left.p = x
y.p = x.p
if x.p == T.nil
    T.root = y
elseif x == x.p.left
    x.p.left = y
else x.p.right = y
y.left = x
x.p = y
```

[1] Eller tilsvarende, Right-Rotate.

Rotationer

Basal operation

Eftersom insertion og deletion laver ændringer i træet, skal vi bruge en basal operation kaldet Left-Rotate[1] til at rette op på de eventuelle brud med RBT-egenskaberne.



Left-Rotate(T, x)

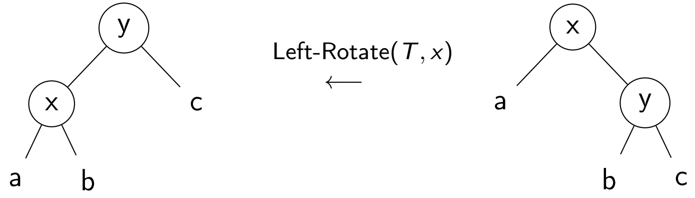
$y = x.right$
$x.right = y.left$
if $y.left \neq T.nil$
  $y.left.p = x$
$y.p = x.p$
if $x.p == T.nil$
  $T.root = y$
elseif $x == x.p.left$
  $x.p.left = y$
else $x.p.right = y$
$y.left = x$
$x.p = y$

[1] Eller tilsvarende, Right-Rotate.

Rotationer

Basal operation

Eftersom insertion og deletion laver ændringer i træet, skal vi bruge en basal operation kaldet Left-Rotate[1] til at rette op på de eventuelle brud med RBT-egenskaberne.



Left-Rotate(T, x)

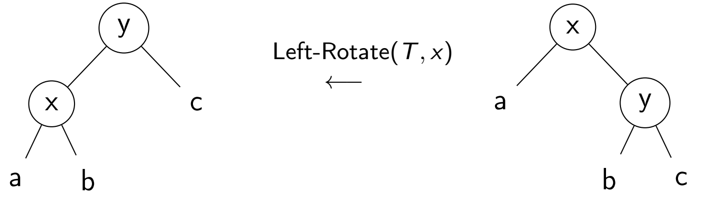
```
y = x.right
x.right = y.left
if y.left ≠ T.nil
        y.left.p = x
y.p = x.p
if x.p == T.nil
        T.root = y
elseif x == x.p.left
        x.p.left = y
else x.p.right = y
y.left = x
x.p = y
```

[1] Eller tilsvarende, Right-Rotate.

Rotationer

Basal operation

Eftersom insertion og deletion laver ændringer i træet, skal vi bruge en basal operation kaldet Left-Rotate[1] til at rette op på de eventuelle brud med RBT-egenskaberne.



Left-Rotate udnytter, at hvis x er ys mor, så er alt i ys venstre sub-træ stadig større end x og kan dermed gøres til xs højre barn, mens x selv kan gøres til ys nye venstre barn.

Left-Rotate(T, x)

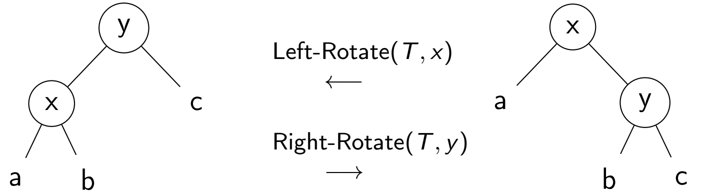
```
y = x.right
x.right = y.left
if y.left ≠ T.nil
    y.left.p = x
y.p = x.p
if x.p == T.nil
    T.root = y
elseif x == x.p.left
    x.p.left = y
else x.p.right = y
y.left = x
x.p = y
```

[1] Eller tilsvarende, Right-Rotate.

Rotationer

Basal operation

Eftersom insertion og deletion laver ændringer i træet, skal vi bruge en basal operation kaldet Left-Rotate[1] til at rette op på de eventuelle brud med RBT-egenskaberne.



Left-Rotate udnytter, at hvis x er ys mor, så er alt i ys venstre sub-træ stadig større end x og kan dermed gøres til xs højre barn, mens x selv kan gøres til ys nye venstre barn.

Left-Rotate(T, x)

```
y = x.right
x.right = y.left
if y.left ≠ T.nil
    y.left.p = x
y.p = x.p
if x.p == T.nil
    T.root = y
elseif x == x.p.left
    x.p.left = y
else x.p.right = y
y.left = x
x.p = y
```

[1] Eller tilsvarende, Right-Rotate.

Rotationer

Left-Rotate

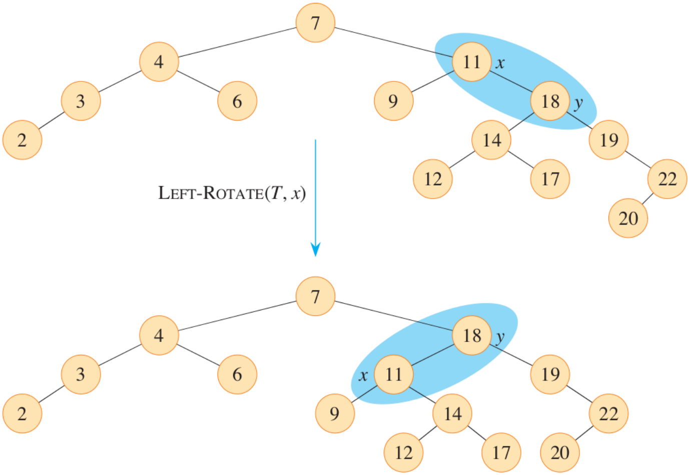


Figure 13.3 An example of how the procedure LEFT-ROTATE(T, x) modifies a binary search tree. Inorder tree walks of the input tree and the modified tree produce the same listing of key values.

Outline

1 Introduktion til Red-Black Trees

2 Insertion i Red-Black trees

3 Exercises

4 Deletion i RBT

Insertion

Med rotationer på plads er vi klar til at se på, hvordan insertion virker i et RBT. Vi starter med at sammenligne med insertion for almindelige BST'er:

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
```

RB-Insert(T, z)

```
x = T.root
y = T.nil
while x ≠ T.nil
    y = x
    if z.key < x.key
        x = x.left
    else x = x.right
z.p = y
if y == T.nil
    T.nil = z
elseif z.key < y.key
    y.left = z
else y.right = z
z.left = T.nil
z.right = T.nil
z.color = RED
RB-Insert-Fixup(T, z)
```

Insertion

Violations

Nu er spørgsmålet, hvilke af RBT-egenskaberne kan ødelægges ved insertion?

RB-Insert(T, z)

```
x = T.root
y = T.nil
while x ≠ T.nil
    y = x
    if z.key < x.key
        x = x.left
    else x = x.right
z.p = y
if y == T.nil
    T.nil = z
elseif z.key < y.key
    y.left = z
else y.right = z
z.left = T.nil
z.right = T.nil
z.color = RED
RB-Insert-Fixup(T, z)
```

Insertion

Violations

Nu er spørgsmålet, hvilke af RBT-egenskaberne kan ødelægges ved insertion?

1. Alle knuder er enten røde eller sorte

RB-Insert(T, z)

```
x = T.root
y = T.nil
while x ≠ T.nil
    y = x
    if z.key < x.key
        x = x.left
    else x = x.right
z.p = y
if y == T.nil
    T.nil = z
elseif z.key < y.key
    y.left = z
else y.right = z
z.left = T.nil
z.right = T.nil
z.color = RED
RB-Insert-Fixup(T, z)
```

Insertion

Violations

Nu er spørgsmålet, hvilke af RBT-egenskaberne kan ødelægges ved insertion?

1. Alle knuder er enten røde eller sorte
   - Nej - alle knuder er fortsat røde eller sorte

RB-Insert(T, z)

```
x = T.root
y = T.nil
while x ≠ T.nil
    y = x
    if z.key < x.key
        x = x.left
    else x = x.right
z.p = y
if y == T.nil
    T.nil = z
elseif z.key < y.key
    y.left = z
else y.right = z
z.left = T.nil
z.right = T.nil
z.color = RED
RB-Insert-Fixup(T, z)
```

Insertion

Violations

Nu er spørgsmålet, hvilke af RBT-egenskaberne kan ødelægges ved insertion?

1. Alle knuder er enten røde eller sorte
   - Nej - alle knuder er fortsat røde eller sorte
2. Roden er sort

RB-Insert(T, z)

```
x = T.root
y = T.nil
while x ≠ T.nil
    y = x
    if z.key < x.key
        x = x.left
    else x = x.right
z.p = y
if y == T.nil
    T.nil = z
elseif z.key < y.key
    y.left = z
else y.right = z
z.left = T.nil
z.right = T.nil
z.color = RED
RB-Insert-Fixup(T, z)
```

Insertion

Violations

Nu er spørgsmålet, hvilke af RBT-egenskaberne kan ødelægges ved insertion?

1. Alle knuder er enten røde eller sorte
   - Nej - alle knuder er fortsat røde eller sorte
2. Roden er sort
   - Ja - hvis træet er tomt, så bliver den nye, røde knude roden

RB-Insert(T, z)

```
x = T.root
y = T.nil
while x ≠ T.nil
    y = x
    if z.key < x.key
        x = x.left
    else x = x.right
z.p = y
if y == T.nil
    T.nil = z
elseif z.key < y.key
    y.left = z
else y.right = z
z.left = T.nil
z.right = T.nil
z.color = RED
RB-Insert-Fixup(T, z)
```

Insertion

Violations

Nu er spørgsmålet, hvilke af RBT-egenskaberne kan ødelægges ved insertion?

1. Alle knuder er enten røde eller sorte
   - Nej - alle knuder er fortsat røde eller sorte
2. Roden er sort
   - Ja - hvis træet er tomt, så bliver den nye, røde knude roden
3. Alle blade er er sorte

RB-Insert(T, z)

```
x = T.root
y = T.nil
while x ≠ T.nil
    y = x
    if z.key < x.key
        x = x.left
    else x = x.right
z.p = y
if y == T.nil
    T.nil = z
elseif z.key < y.key
    y.left = z
else y.right = z
z.left = T.nil
z.right = T.nil
z.color = RED
RB-Insert-Fixup(T, z)
```

Insertion

Violations

Nu er spørgsmålet, hvilke af RBT-egenskaberne kan ødelægges ved insertion?

1. Alle knuder er enten røde eller sorte
   - Nej - alle knuder er fortsat røde eller sorte
2. Roden er sort
   - Ja - hvis træet er tomt, så bliver den nye, røde knude roden
3. Alle blade er er sorte
   - Nej - den nye knude har $T.nil$ (sort) som blade/børn

RB-Insert(T, z)

```
x = T.root
y = T.nil
while x ≠ T.nil
    y = x
    if z.key < x.key
        x = x.left
    else x = x.right
z.p = y
if y == T.nil
    T.nil = z
elseif z.key < y.key
    y.left = z
else y.right = z
z.left = T.nil
z.right = T.nil
z.color = RED
RB-Insert-Fixup(T, z)
```

Insertion

Violations

Nu er spørgsmålet, hvilke af RBT-egenskaberne kan ødelægges ved insertion?

1. Alle knuder er enten røde eller sorte
   - Nej - alle knuder er fortsat røde eller sorte
2. Roden er sort
   - Ja - hvis træet er tomt, så bliver den nye, røde knude roden
3. Alle blade er er sorte
   - Nej - den nye knude har $T.nil$ (sort) som blade/børn
4. Hvis en knude er rød, så er begge dens børn sorte

RB-Insert(T, z)

```
x = T.root
y = T.nil
while x ≠ T.nil
    y = x
    if z.key < x.key
        x = x.left
    else x = x.right
z.p = y
if y == T.nil
    T.nil = z
elseif z.key < y.key
    y.left = z
else y.right = z
z.left = T.nil
z.right = T.nil
z.color = RED
RB-Insert-Fixup(T, z)
```

Insertion

Violations

Nu er spørgsmålet, hvilke af RBT-egenskaberne kan ødelægges ved insertion?

1. Alle knuder er enten røde eller sorte
   - Nej - alle knuder er fortsat røde eller sorte
2. Roden er sort
   - Ja - hvis træet er tomt, så bliver den nye, røde knude roden
3. Alle blade er er sorte
   - Nej - den nye knude har $T.nil$ (sort) som blade/børn
4. Hvis en knude er rød, så er begge dens børn sorte
   - Ja - hvis den nye knudes mor er rød

RB-Insert(T, z)

```
x = T.root
y = T.nil
while x ≠ T.nil
    y = x
    if z.key < x.key
        x = x.left
    else x = x.right
z.p = y
if y == T.nil
    T.nil = z
elseif z.key < y.key
    y.left = z
else y.right = z
z.left = T.nil
z.right = T.nil
z.color = RED
RB-Insert-Fixup(T, z)
```

Insertion

Violations

Nu er spørgsmålet, hvilke af RBT-egenskaberne kan ødelægges ved insertion?

1. Alle knuder er enten røde eller sorte
   - Nej - alle knuder er fortsat røde eller sorte
2. Roden er sort
   - Ja - hvis træet er tomt, så bliver den nye, røde knude roden
3. Alle blade er er sorte
   - Nej - den nye knude har $T.nil$ (sort) som blade/børn
4. Hvis en knude er rød, så er begge dens børn sorte
   - Ja - hvis den nye knudes mor er rød
5. Alle simple stier fra en knude til et blad i et af dets subtræer indeholder den samme mængde sorte knuder

RB-Insert(T, z)

```
x = T.root
y = T.nil
while x ≠ T.nil
    y = x
    if z.key < x.key
        x = x.left
    else x = x.right
z.p = y
if y == T.nil
    T.nil = z
elseif z.key < y.key
    y.left = z
else y.right = z
z.left = T.nil
z.right = T.nil
z.color = RED
RB-Insert-Fixup(T, z)
```

Insertion

Violations

Nu er spørgsmålet, hvilke af RBT-egenskaberne kan ødelægges ved insertion?

1. Alle knuder er enten røde eller sorte
   - Nej - alle knuder er fortsat røde eller sorte
2. Roden er sort
   - Ja - hvis træet er tomt, så bliver den nye, røde knude roden
3. Alle blade er er sorte
   - Nej - den nye knude har $T.nil$ (sort) som blade/børn
4. Hvis en knude er rød, så er begge dens børn sorte
   - Ja - hvis den nye knudes mor er rød
5. Alle simple stier fra en knude til et blad i et af dets subtræer indeholder den samme mængde sorte knuder
   - Nej - den nye knude er rød og bidrager dermed ikke til den sorte højde

RB-Insert(T, z)

```
x = T.root
y = T.nil
while x ≠ T.nil
    y = x
    if z.key < x.key
        x = x.left
    else x = x.right
z.p = y
if y == T.nil
    T.nil = z
elseif z.key < y.key
    y.left = z
else y.right = z
z.left = T.nil
z.right = T.nil
z.color = RED
RB-Insert-Fixup(T, z)
```

Fiks insertions

RB-Insert-Fixup

Vi ser nu på RB-Insert-Fixup proceduren, der genopretter RBT-egenskaberne efter insertion:

RB-Insert-Fixup(T, z)

```
while z.p.color == RED
    if z.p == z.p.p.left
        y = z.p.p.right
        if y.color == RED
            z.p.color = BLACK
            y.color = BLACK
            z.p.p.color = RED
            z = z.p.p
        else
            if z == z.p.right
                z = z.p
                Left-Rotate(T, z)
            z.p.color = BLACK
            z.p.p.color = RED
            Right-Rotate(T, z.p.p)
    else same, but swap 'left' and 'right'
T.root.color = BLACK
```

- z er rød, så vi går kun ind i while-løkken, hvis dens mor også er rød

Fiks insertions

RB-Insert-Fixup

Vi ser nu på RB-Insert-Fixup proceduren, der genopretter RBT-egenskaberne efter insertion:

RB-Insert-Fixup(T, z)

```
while z.p.color == RED
    if z.p == z.p.p.left
        y = z.p.p.right
        if y.color == RED
            z.p.color = BLACK
            y.color = BLACK
            z.p.p.color = RED
            z = z.p.p
        else
            if z == z.p.right
                z = z.p
                Left-Rotate(T, z)
            z.p.color = BLACK
            z.p.p.color = RED
            Right-Rotate(T, z.p.p)
    else same, but swap 'left' and 'right'
T.root.color = BLACK
```

- z er rød, så vi går kun ind i while-løkken, hvis dens mor også er rød
- Vi tjekker, om z's mor er et venstre barn

Fiks insertions

RB-Insert-Fixup

Vi ser nu på RB-Insert-Fixup proceduren, der genopretter RBT-egenskaberne efter insertion:

RB-Insert-Fixup(T, z)

```
while z.p.color == RED
    if z.p == z.p.p.left
        y = z.p.p.right
        if y.color == RED
            z.p.color = BLACK
            y.color = BLACK
            z.p.p.color = RED
            z = z.p.p
        else
            if z == z.p.right
                z = z.p
                Left-Rotate(T, z)
            z.p.color = BLACK
            z.p.p.color = RED
            Right-Rotate(T, z.p.p)
    else same, but swap 'left' and 'right'
T.root.color = BLACK
```

- z er rød, så vi går kun ind i while-løkken, hvis dens mor også er rød
- Vi tjekker, om z's mor er et venstre barn
- Hvis z's moster (y) er rød, så skal både den og z's mor farves sorte, mens bedstefar gøres rød — og vi fortsætter nu fra bedstefar

Fiks insertions

RB-Insert-Fixup

Vi ser nu på RB-Insert-Fixup proceduren, der genopretter RBT-egenskaberne efter insertion:

RB-Insert-Fixup(T, z)

```
while z.p.color == RED
    if z.p == z.p.p.left
        y = z.p.p.right
        if y.color == RED
            z.p.color = BLACK
            y.color = BLACK
            z.p.p.color = RED
            z = z.p.p
        else
            if z == z.p.right
                z = z.p
                Left-Rotate(T, z)
            z.p.color = BLACK
            z.p.p.color = RED
            Right-Rotate(T, z.p.p)
    else same, but swap 'left' and 'right'
T.root.color = BLACK
```

- z er rød, så vi går kun ind i while-løkken, hvis dens mor også er rød
- Vi tjekker, om z's mor er et venstre barn
- Hvis z's moster (y) er rød, så skal både den og z's mor farves sorte, mens bedstefar gøres rød — og vi fortsætter nu fra bedstefar
- Hvis moster ikke var rød, og z er et højre barn, så skal vi lave en Left-Rotate

Fiks insertions

RB-Insert-Fixup

Vi ser nu på RB-Insert-Fixup proceduren, der genopretter RBT-egenskaberne efter insertion:

RB-Insert-Fixup(T, z)

```
while z.p.color == RED
    if z.p == z.p.p.left
        y = z.p.p.right
        if y.color == RED
            z.p.color = BLACK
            y.color = BLACK
            z.p.p.color = RED
            z = z.p.p
        else
            if z == z.p.right
                z = z.p
                Left-Rotate(T, z)
            z.p.color = BLACK
            z.p.p.color = RED
            Right-Rotate(T, z.p.p)
    else same, but swap 'left' and 'right'
T.root.color = BLACK
```

- z er rød, så vi går kun ind i while-løkken, hvis dens mor også er rød
- Vi tjekker, om z's mor er et venstre barn
- Hvis z's moster (y) er rød, så skal både den og z's mor farves sorte, mens bedstefar gøres rød — og vi fortsætter nu fra bedstefar
- Hvis moster ikke var rød, og z er et højre barn, så skal vi lave en Left-Rotate
- Var z venstre barn laver vi en Right-Rotate

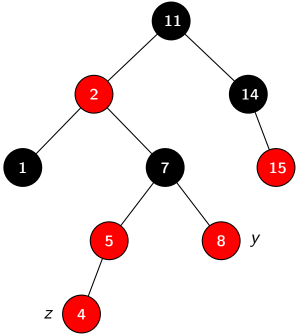
Fiks insertions

RB-Insert-Fixup

Vi ser nu på RB-Insert-Fixup proceduren, der genopretter RBT-egenskaberne efter insertion:

RB-Insert-Fixup(T, z)

```
while z.p.color == RED
    if z.p == z.p.p.left
        y = z.p.p.right
        if y.color == RED
            z.p.color = BLACK
            y.color = BLACK
            z.p.p.color = RED
            z = z.p.p
        else
            if z == z.p.right
                z = z.p
                Left-Rotate(T, z)
            z.p.color = BLACK
            z.p.p.color = RED
            Right-Rotate(T, z.p.p)
    else same, but swap 'left' and 'right'
T.root.color = BLACK
```



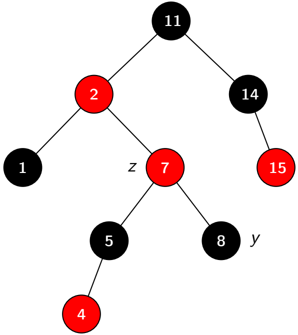
Fiks insertions

RB-Insert-Fixup

Vi ser nu på RB-Insert-Fixup proceduren, der genopretter RBT-egenskaberne efter insertion:

RB-Insert-Fixup(T, z)

```
while z.p.color == RED
    if z.p == z.p.p.left
        y = z.p.p.right
        if y.color == RED
            z.p.color = BLACK
            y.color = BLACK
            z.p.p.color = RED
            z = z.p.p
        else
            if z == z.p.right
                z = z.p
                Left-Rotate(T, z)
            z.p.color = BLACK
            z.p.p.color = RED
            Right-Rotate(T, z.p.p)
    else same, but swap 'left' and 'right'
T.root.color = BLACK
```



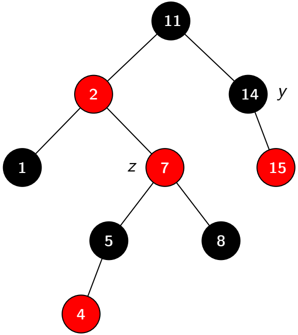
Fiks insertions

RB-Insert-Fixup

Vi ser nu på RB-Insert-Fixup proceduren, der genopretter RBT-egenskaberne efter insertion:

RB-Insert-Fixup(T, z)

```
while z.p.color == RED
    if z.p == z.p.p.left
        y = z.p.p.right
        if y.color == RED
            z.p.color = BLACK
            y.color = BLACK
            z.p.p.color = RED
            z = z.p.p
        else
            if z == z.p.right
                z = z.p
                Left-Rotate(T, z)
            z.p.color = BLACK
            z.p.p.color = RED
            Right-Rotate(T, z.p.p)
    else same, but swap 'left' and 'right'
T.root.color = BLACK
```



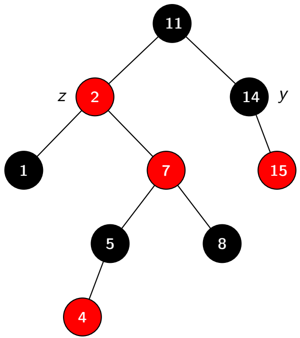
Fiks insertions

RB-Insert-Fixup

Vi ser nu på RB-Insert-Fixup proceduren, der genopretter RBT-egenskaberne efter insertion:

RB-Insert-Fixup(T, z)

```
while z.p.color == RED
    if z.p == z.p.p.left
        y = z.p.p.right
        if y.color == RED
            z.p.color = BLACK
            y.color = BLACK
            z.p.p.color = RED
            z = z.p.p
        else
            if z == z.p.right
                z = z.p
                Left-Rotate(T, z)
            z.p.color = BLACK
            z.p.p.color = RED
            Right-Rotate(T, z.p.p)
    else same, but swap 'left' and 'right'
T.root.color = BLACK
```



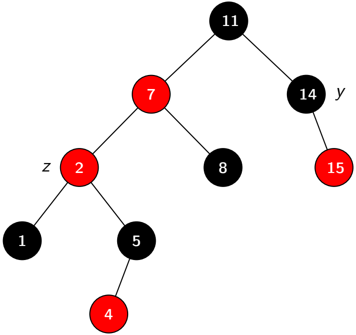
Fiks insertions

RB-Insert-Fixup

Vi ser nu på RB-Insert-Fixup proceduren, der genopretter RBT-egenskaberne efter insertion:

RB-Insert-Fixup(T, z)

```
while z.p.color == RED
    if z.p == z.p.p.left
        y = z.p.p.right
        if y.color == RED
            z.p.color = BLACK
            y.color = BLACK
            z.p.p.color = RED
            z = z.p.p
        else
            if z == z.p.right
                z = z.p
                Left-Rotate(T, z)
            z.p.color = BLACK
            z.p.p.color = RED
            Right-Rotate(T, z.p.p)
    else same, but swap 'left' and 'right'
T.root.color = BLACK
```



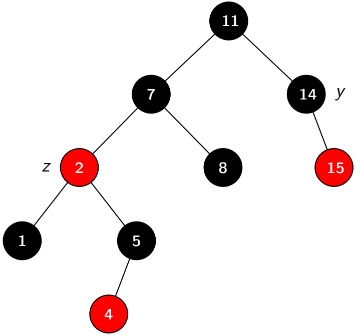
Fiks insertions

RB-Insert-Fixup

Vi ser nu på RB-Insert-Fixup proceduren, der genopretter RBT-egenskaberne efter insertion:

RB-Insert-Fixup(T, z)

```
while z.p.color == RED
    if z.p == z.p.p.left
        y = z.p.p.right
        if y.color == RED
            z.p.color = BLACK
            y.color = BLACK
            z.p.p.color = RED
            z = z.p.p
        else
            if z == z.p.right
                z = z.p
                Left-Rotate(T, z)
            z.p.color = BLACK
            z.p.p.color = RED
            Right-Rotate(T, z.p.p)
    else same, but swap 'left' and 'right'
T.root.color = BLACK
```



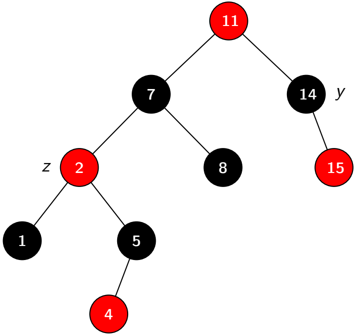
Fiks insertions

RB-Insert-Fixup

Vi ser nu på RB-Insert-Fixup proceduren, der genopretter RBT-egenskaberne efter insertion:

RB-Insert-Fixup(T, z)

```
while z.p.color == RED
    if z.p == z.p.p.left
        y = z.p.p.right
        if y.color == RED
            z.p.color = BLACK
            y.color = BLACK
            z.p.p.color = RED
            z = z.p.p
        else
            if z == z.p.right
                z = z.p
                Left-Rotate(T, z)
            z.p.color = BLACK
            z.p.p.color = RED
            Right-Rotate(T, z.p.p)
    else same, but swap 'left' and 'right'
T.root.color = BLACK
```



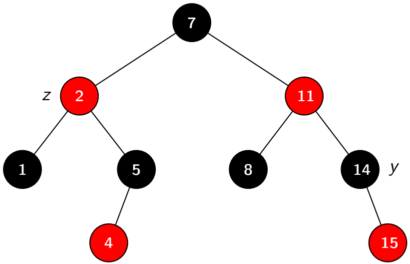
Fiks insertions

RB-Insert-Fixup

Vi ser nu på RB-Insert-Fixup proceduren, der genopretter RBT-egenskaberne efter insertion:

RB-Insert-Fixup(T, z)

```
while z.p.color == RED
    if z.p == z.p.p.left
        y = z.p.p.right
        if y.color == RED
            z.p.color = BLACK
            y.color = BLACK
            z.p.p.color = RED
            z = z.p.p
        else
            if z == z.p.right
                z = z.p
                Left-Rotate(T, z)
            z.p.color = BLACK
            z.p.p.color = RED
            Right-Rotate(T, z.p.p)
    else same, but swap 'left' and 'right'
T.root.color = BLACK
```



Outline

1 Introduktion til Red-Black Trees

2 Insertion i Red-Black trees

3 Exercises

4 Deletion i RBT

Exercises

Super fedt! <3

På Moodle! Go! Fungerer det fint?

Outline

1. Introduktion til Red-Black Trees
2. Insertion i Red-Black trees
3. Exercises
4. Deletion i RBT

Deletion i RBT

Mere komplekst

Vi vender nu blikket mod det at skulle slette et element z fra et RBT.

Deletion i RBT

Mere komplekst

Vi vender nu blikket mod det at skulle slette et element z fra et RBT.

- Ligesom for binære søgetræer er det en noget mere kompliceret procedure

Deletion i RBT

Mere komplekst

Vi vender nu blikket mod det at skulle slette et element z fra et RBT.

- Ligesom for binære søgetræer er det en noget mere kompliceret procedure
- Der er modificeret udgave af Transplant-proceduren — kaldet RB-Transplant — som ligesom for BST'er erstatter en knude u med en anden knude v

Deletion i RBT

Mere komplekst

Vi vender nu blikket mod det at skulle slette et element z fra et RBT.

- Ligesom for binære søgetræer er det en noget mere kompliceret procedure
- Der er modificeret udgave af Transplant-proceduren — kaldet RB-Transplant — som ligesom for BST'er erstatter en knude u med en anden knude v
- Til at genoprette RBT-egenskaberne efter deletion findes der en procedure RB-Delete-Fixup

Deletion i RBT

Mere komplekst

Vi vender nu blikket mod det at skulle slette et element z fra et RBT.

- Ligesom for binære søgetræer er det en noget mere kompliceret procedure
- Der er modificeret udgave af Transplant-proceduren — kaldet RB-Transplant — som ligesom for BST'er erstatter en knude u med en anden knude v
- Til at genoprette RBT-egenskaberne efter deletion findes der en procedure RB-Delete-Fixup
  - Den kommer vi ikke til at gå ind i

Deletion

RB-Delete(T, z)

```
y = z
yOrgColor = y.color
if z.left == T.nil
    z = z.right
    RB-Transplant(T, z, z.right)
elseif z.right == T.nil
    x = z.left
    RB-Transplant(T, z, z.left)
else y = Tree-Minimum(z.right)
    yOrgColor = y.color
    x = y.right
    if y ≠ z.right
        RB-Transplant(T, y, y.right)
        y.right = z.right
        y.right.p = y
    else x.p = y
    RB-Transplant(T, z, y)
    y.left = z.left
    y.left.p = y
    y.color = z.color
if yOrgColor == BLACK
    RB-Delete-Fixup(T, x)
```

Deletion

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
```

RB-Delete(T, z)

```
y = z
yOrgColor = y.color
if z.left == T.nil
        z = z.right
        RB-Transplant(T, z, z.right)
elseif z.right == T.nil
        x = z.left
        RB-Transplant(T, z, z.left)
else y = Tree-Minimum(z.right)
        yOrgColor = y.color
        x = y.right
        if y ≠ z.right
                RB-Transplant(T, y, y.right)
                y.right = z.right
                y.right.p = y
        else x.p = y
        RB-Transplant(T, z, y)
        y.left = z.left
        y.left.p = y
        y.color = z.color
if yOrgColor == BLACK
        RB-Delete-Fixup(T, x)
```

Deletion

RB-Delete(T, z)

```
1   y = z
2   yOrgColor = y.color
3   if z.left == T.nil
4       z = z.right
5       RB-Transplant(T, z, z.right)
6   elseif z.right == T.nil
7       x = z.left
8       RB-Transplant(T, z, z.left)
9   else y = Tree-Minimum(z.right)
10      yOrgColor = y.color
11      x = y.right
12      if y ≠ z.right
13          RB-Transplant(T, y, y.right)
14          y.right = z.right
15          y.right.p = y
16      else x.p = y
17      RB-Transplant(T, z, y)
18      y.left = z.left
19      y.left.p = y
20      y.color = z.color
21  if yOrgColor == BLACK
22      RB-Delete-Fixup(T, x)
```

[a] Bemærk at linie 16 er mærkelig, men er nødvendig for RB-Delete-Fixup senere

Deletion

- Som i Tree-Delete vil vi slette knude z

RB-Delete(T, z)

```
1  y = z
2  yOrgColor = y.color
3  if z.left == T.nil
4      z = z.right
5      RB-Transplant(T, z, z.right)
6  elseif z.right == T.nil
7      x = z.left
8      RB-Transplant(T, z, z.left)
9  else y = Tree-Minimum(z.right)
10     yOrgColor = y.color
11     x = y.right
12     if y ≠ z.right
13         RB-Transplant(T, y, y.right)
14         y.right = z.right
15         y.right.p = y
16     else x.p = y
17     RB-Transplant(T, z, y)
18     y.left = z.left
19     y.left.p = y
20     y.color = z.color
21 if yOrgColor == BLACK
22     RB-Delete-Fixup(T, x)
```

[a] Bemærk at linie 16 er mærkelig, men er nødvendig for RB-Delete-Fixup senere

Deletion

- Som i Tree-Delete vil vi slette knude z
- Linie 1-8 tjekker om z kun har 1 barn, og erstatter så bare z med det barn

RB-Delete(T, z)

```
1   y = z
2   yOrgColor = y.color
3   if z.left == T.nil
4       z = z.right
5       RB-Transplant(T, z, z.right)
6   elseif z.right == T.nil
7       x = z.left
8       RB-Transplant(T, z, z.left)
9   else y = Tree-Minimum(z.right)
10      yOrgColor = y.color
11      x = y.right
12      if y ≠ z.right
13          RB-Transplant(T, y, y.right)
14          y.right = z.right
15          y.right.p = y
16      else x.p = y
17      RB-Transplant(T, z, y)
18      y.left = z.left
19      y.left.p = y
20      y.color = z.color
21  if yOrgColor == BLACK
22      RB-Delete-Fixup(T, x)
```

[a] Bemærk at linie 16 er mærkelig, men er nødvendig for RB-Delete-Fixup senere

Deletion

- Som i Tree-Delete vil vi slette knude z
- Linie 1-8 tjekker om z kun har 1 barn, og erstatter så bare z med det barn
- Hvis vi rammer linie 9, så har z to børn, og vi finder dets successor y

RB-Delete(T, z)

```
1   y = z
2   yOrgColor = y.color
3   if z.left == T.nil
4       z = z.right
5       RB-Transplant(T, z, z.right)
6   elseif z.right == T.nil
7       x = z.left
8       RB-Transplant(T, z, z.left)
9   else y = Tree-Minimum(z.right)
10      yOrgColor = y.color
11      x = y.right
12      if y ≠ z.right
13          RB-Transplant(T, y, y.right)
14          y.right = z.right
15          y.right.p = y
16      else x.p = y
17      RB-Transplant(T, z, y)
18      y.left = z.left
19      y.left.p = y
20      y.color = z.color
21  if yOrgColor == BLACK
22      RB-Delete-Fixup(T, x)
```

[a]Bemærk at linie 16 er mærkelig, men er nødvendig for RB-Delete-Fixup senere

Deletion

- Som i Tree-Delete vil vi slette knude z
- Linie 1-8 tjekker om z kun har 1 barn, og erstatter så bare z med det barn
- Hvis vi rammer linie 9, så har z to børn, og vi finder dets successor y
- I linie 12-15 tjekker vi, om y er længere nede i træet — i så fald erstatter vi y med dets højre barn, og lader det derefter pege på z's højre barn

RB-Delete(T, z)

```
1   y = z
2   yOrgColor = y.color
3   if z.left == T.nil
4       z = z.right
5       RB-Transplant(T, z, z.right)
6   elseif z.right == T.nil
7       x = z.left
8       RB-Transplant(T, z, z.left)
9   else y = Tree-Minimum(z.right)
10      yOrgColor = y.color
11      x = y.right
12      if y ≠ z.right
13          RB-Transplant(T, y, y.right)
14          y.right = z.right
15          y.right.p = y
16      else x.p = y
17      RB-Transplant(T, z, y)
18      y.left = z.left
19      y.left.p = y
20      y.color = z.color
21  if yOrgColor == BLACK
22      RB-Delete-Fixup(T, x)
```

[a] Bemærk at linie 16 er mærkelig, men er nødvendig for RB-Delete-Fixup senere

Deletion

- Som i Tree-Delete vil vi slette knude z
- Linie 1-8 tjekker om z kun har 1 barn, og erstatter så bare z med det barn
- Hvis vi rammer linie 9, så har z to børn, og vi finder dets successor y
- I linie 12-15 tjekker vi, om y er længere nede i træet — i så fald erstatter vi y med dets højre barn, og lader det derefter pege på z's højre barn
- Herefter kan vi erstatte z og y med hinanden og give z's venstre barn til y (som ikke havde et venstre barn, da det er z's successor i højre sub-træ)[a]

RB-Delete(T, z)

```
1   y = z
2   yOrgColor = y.color
3   if z.left == T.nil
4       z = z.right
5       RB-Transplant(T, z, z.right)
6   elseif z.right == T.nil
7       x = z.left
8       RB-Transplant(T, z, z.left)
9   else y = Tree-Minimum(z.right)
10      yOrgColor = y.color
11      x = y.right
12      if y ≠ z.right
13          RB-Transplant(T, y, y.right)
14          y.right = z.right
15          y.right.p = y
16      else x.p = y
17      RB-Transplant(T, z, y)
18      y.left = z.left
19      y.left.p = y
20      y.color = z.color
21  if yOrgColor == BLACK
22      RB-Delete-Fixup(T, x)
```

[a] Bemærk at linie 16 er mærkelig, men er nødvendig for RB-Delete-Fixup senere

Deletion

- Som i Tree-Delete vil vi slette knude z
- Linie 1-8 tjekker om z kun har 1 barn, og erstatter så bare z med det barn
- Hvis vi rammer linie 9, så har z to børn, og vi finder dets successor y
- I linie 12-15 tjekker vi, om y er længere nede i træet — i så fald erstatter vi y med dets højre barn, og lader det derefter pege på z's højre barn
- Herefter kan vi erstatte z og y med hinanden og give z's venstre barn til y (som ikke havde et venstre barn, da det er z's successor i højre sub-træ)[a]
- Til sidst tjekker vi, om farven på den knude, vi har pillet ved (enten z eller dens successor y) oprindeligt var sort — for så skal vi reparere træet!

[a] Bemærk at linie 16 er mærkelig, men er nødvendig for RB-Delete-Fixup senere

RB-Delete(T, z)

```
1   y = z
2   yOrgColor = y.color
3   if z.left == T.nil
4       z = z.right
5       RB-Transplant(T, z, z.right)
6   elseif z.right == T.nil
7       x = z.left
8       RB-Transplant(T, z, z.left)
9   else y = Tree-Minimum(z.right)
10      yOrgColor = y.color
11      x = y.right
12      if y ≠ z.right
13          RB-Transplant(T, y, y.right)
14          y.right = z.right
15          y.right.p = y
16      else x.p = y
17      RB-Transplant(T, z, y)
18      y.left = z.left
19      y.left.p = y
20      y.color = z.color
21  if yOrgColor == BLACK
22      RB-Delete-Fixup(T, x)
```

Deletion

Violations

Hvad kan gå galt?

Deletion

Violations

Hvad kan gå galt?

1. Alle knuder er enten røde eller sorte

Deletion

Violations

Hvad kan gå galt?

1. Alle knuder er enten røde eller sorte
   - Nej - alle knuder er fortsat røde eller sorte

Deletion

Violations

Hvad kan gå galt?

1. Alle knuder er enten røde eller sorte
   - Nej - alle knuder er fortsat røde eller sorte
2. Roden er sort

Deletion

Violations

Hvad kan gå galt?

1. Alle knuder er enten røde eller sorte
   - Nej - alle knuder er fortsat røde eller sorte
2. Roden er sort
   - Ja - hvis z var roden og et rødt barn bliver ny rod

Deletion

Violations

Hvad kan gå galt?

1. Alle knuder er enten røde eller sorte
   - Nej - alle knuder er fortsat røde eller sorte
2. Roden er sort
   - Ja - hvis z var roden og et rødt barn bliver ny rod
3. Alle blade er er sorte

Deletion

Violations

Hvad kan gå galt?

1. Alle knuder er enten røde eller sorte
   - Nej - alle knuder er fortsat røde eller sorte
2. Roden er sort
   - Ja - hvis z var roden og et rødt barn bliver ny rod
3. Alle blade er er sorte
   - Nej - ingen blade skifter farve

Deletion

Violations

Hvad kan gå galt?

1. Alle knuder er enten røde eller sorte
   - Nej - alle knuder er fortsat røde eller sorte
2. Roden er sort
   - Ja - hvis z var roden og et rødt barn bliver ny rod
3. Alle blade er er sorte
   - Nej - ingen blade skifter farve
4. Hvis en knude er rød, så er begge dens børn sorte

Deletion

Violations

Hvad kan gå galt?

1. Alle knuder er enten røde eller sorte
   - Nej - alle knuder er fortsat røde eller sorte
2. Roden er sort
   - Ja - hvis z var roden og et rødt barn bliver ny rod
3. Alle blade er er sorte
   - Nej - ingen blade skifter farve
4. Hvis en knude er rød, så er begge dens børn sorte
   - Ja - hvis y, da den blev flyttet, blev erstattet af en rød knude

Deletion

Violations

Hvad kan gå galt?

1. Alle knuder er enten røde eller sorte
   - Nej - alle knuder er fortsat røde eller sorte
2. Roden er sort
   - Ja - hvis z var roden og et rødt barn bliver ny rod
3. Alle blade er er sorte
   - Nej - ingen blade skifter farve
4. Hvis en knude er rød, så er begge dens børn sorte
   - Ja - hvis y, da den blev flyttet, blev erstattet af en rød knude
5. Alle simple stier fra en knude til et blad i et af dets subtræer indeholder den samme mængde sorte knuder

Deletion

Violations

Hvad kan gå galt?

1. Alle knuder er enten røde eller sorte
   - Nej - alle knuder er fortsat røde eller sorte
2. Roden er sort
   - Ja - hvis z var roden og et rødt barn bliver ny rod
3. Alle blade er er sorte
   - Nej - ingen blade skifter farve
4. Hvis en knude er rød, så er begge dens børn sorte
   - Ja - hvis y, da den blev flyttet, blev erstattet af en rød knude
5. Alle simple stier fra en knude til et blad i et af dets subtræer indeholder den samme mængde sorte knuder
   - Ja - vi har fjernet en sort knude

Deletion

Violations

Hvad kan gå galt?

1. Alle knuder er enten røde eller sorte
   - Nej - alle knuder er fortsat røde eller sorte
2. Roden er sort
   - Ja - hvis z var roden og et rødt barn bliver ny rod
3. Alle blade er er sorte
   - Nej - ingen blade skifter farve
4. Hvis en knude er rød, så er begge dens børn sorte
   - Ja - hvis y, da den blev flyttet, blev erstattet af en rød knude
5. Alle simple stier fra en knude til et blad i et af dets subtræer indeholder den samme mængde sorte knuder
   - Ja - vi har fjernet en sort knude

```
RB-DELETE-FIXUP(T, x)
1  while x ≠ T.root and x.color == BLACK
2      if x == x.p.left                // is x a left child?
3          w = x.p.right               // w is x's sibling
4          if w.color == RED
5              w.color = BLACK                              }
6              x.p.color = RED                              } case 1
7              LEFT-ROTATE(T, x.p)                          }
8              w = x.p.right                                }
9          if w.left.color == BLACK and w.right.color == BLACK
10             w.color = RED                                } case 2
11             x = x.p                                      }
12         else
13             if w.right.color == BLACK
14                 w.left.color = BLACK                     }
15                 w.color = RED                            } case 3
16                 RIGHT-ROTATE(T, w)                       }
17                 w = x.p.right                            }
18             w.color = x.p.color                          }
19             x.p.color = BLACK                            }
20             w.right.color = BLACK                        } case 4
21             LEFT-ROTATE(T, x.p)                          }
22             x = T.root                                   }
23     else // same as lines 3–22, but with "right" and "left" exchanged
24         w = x.p.left
25         if w.color == RED
26             w.color = BLACK
27             x.p.color = RED
28             RIGHT-ROTATE(T, x.p)
29             w = x.p.left
30         if w.right.color == BLACK and w.left.color == BLACK
31             w.color = RED
32             x = x.p
33         else
34             if w.left.color == BLACK
35                 w.right.color = BLACK
36                 w.color = RED
37                 LEFT-ROTATE(T, w)
38                 w = x.p.left
39             w.color = x.p.color
40             x.p.color = BLACK
41             w.left.color = BLACK
42             RIGHT-ROTATE(T, x.p)
43             x = T.root
44 x.color = BLACK
```

RB-Delete-Fixup(T, x)

```
1  while x ≠ T.root and x.color == BLACK
2      if x == x.p.left
3          w = x.p.right
           ...
4      else // Når x == x.p.right
5          w = x.p.left
           ...
6  x.color = BLACK
```

- Proceduren tager træet T og en knude x, som vi skal reparere fra

Reparere træet

RB-Delete-Fixup

RB-Delete-Fixup(T, x)

```
while x ≠ T.root and x.color == BLACK
    if x == x.p.left
        w = x.p.right
        ...
    else // Når x == x.p.right
        w = x.p.left
        ...
x.color = BLACK
```

- Proceduren tager træet T og en knude x, som vi skal reparere fra
- Proceduren består af en while-løkke, som kører så længe, at x ikke er roden og så længe, at x er sort

Reparere træet

RB-Delete-Fixup

RB-Delete-Fixup(T, x)

```
while x ≠ T.root and x.color == BLACK
    if x == x.p.left
        w = x.p.right
        ...
    else // Når x == x.p.right
        w = x.p.left
        ...
x.color = BLACK
```

- Proceduren tager træet T og en knude x, som vi skal reparere fra
- Proceduren består af en while-løkke, som kører så længe, at x ikke er roden og så længe, at x er sort
- Koden består af 2 næsten identiske bidder, der kun adskiller sig ved, om x er et venstre eller et højre barn

Reparere træet

RB-Delete-Fixup

RB-Delete-Fixup(T, x)

```
while x ≠ T.root and x.color == BLACK
    if x == x.p.left
        w = x.p.right
        ...
    else // Når x == x.p.right
        w = x.p.left
        ...
x.color = BLACK
```

- Proceduren tager træet T og en knude x, som vi skal reparere fra
- Proceduren består af en while-løkke, som kører så længe, at x ikke er roden og så længe, at x er sort
- Koden består af 2 næsten identiske bidder, der kun adskiller sig ved, om x er et venstre eller et højre barn
- w angiver x's søster

Reparere træet

RB-Delete-Fixup

RB-Delete-Fixup(T, x)

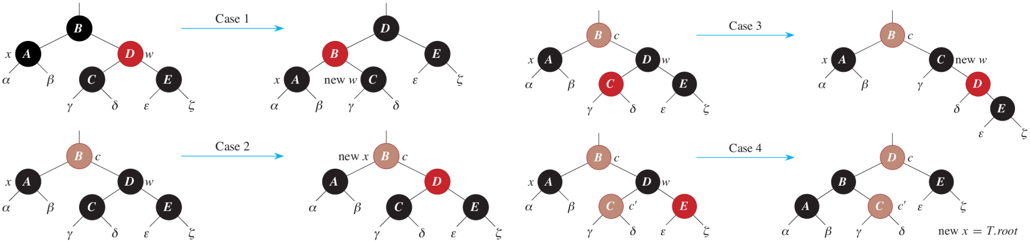
```
while x ≠ T.root and x.color == BLACK
    if x == x.p.left
        w = x.p.right
        ...
    else // Når x == x.p.right
        w = x.p.left
        ...
x.color = BLACK
```

- Proceduren tager træet T og en knude x, som vi skal reparere fra
- Proceduren består af en while-løkke, som kører så længe, at x ikke er roden og så længe, at x er sort
- Koden består af 2 næsten identiske bidder, der kun adskiller sig ved, om x er et venstre eller et højre barn
- w angiver x's søster
- Det hele slutter med, at vi farver x sort

Reparere træet

4 cases

Resten af koden håndterer 4 cases for, hvordan der kan være opstået problemer.



Reparere træet

Men fordi det næsten er fredag, så gider vi ikke gå længere ind i det!

Tak for i dag!

Flere exercises..

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