Datenstrukturen, Algorithmen und Programmierung 2

Amin Coja-Oghlan

June 27, 2023

Lehrstuhl Informatik 2
Fakultät für Informatik

Red black trees

Worum geht es?

- binäre Suchbäume sind nur dann effizient, wenn sie von geringer Höhe sind
- red black trees sind eine Variante, die sich selbst “trimmt”
- die Höhe bei n Elementen ist dabei stets $O(\log n)$
- wir müssen die Operationen `Insert` und `Delete` verändern

Red black trees

Red black-Regeln

Ein red black tree ist ein binärer Suchbaum, dessen Knoten als zustätzliches Attribut eine Farbe haben.

RB1: jeder Knoten ist entweder rot oder schwarz gefärbt

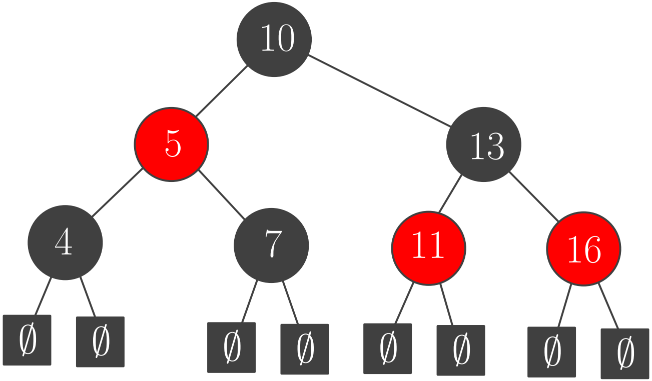
RB2: die Wurzel ist schwarz

RB3: jeder $\emptyset$-Zeiger auf ein Kind zählt als schwarzer Knoten (Blatt)

RB4: rote Knoten haben nur schwarze Kinder

RB5: jeder Pfad von der Wurzel zu einem Blatt enthählt dieselbe Zahl schwarzer Knoten

Red black trees



Red black trees

Implementation

- jeder Knoten benötigt ein zusätzliches Bit für die Farbe
- wir speichern *einen* zusätzlichen Knoten als $\emptyset$-Knoten ab
- dessen Elternzeiger ist also *nicht* korrekt gesetzt!

Red black trees

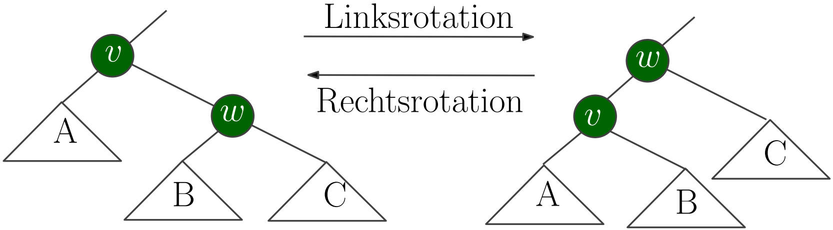
Lemma

Ein red black tree mit n Knoten hat Höhe $O(\log n)$.

Beweis

Dies folgt unmittelbar aus **RB4** und **RB5**.

Red black trees

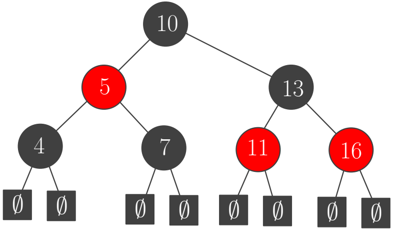


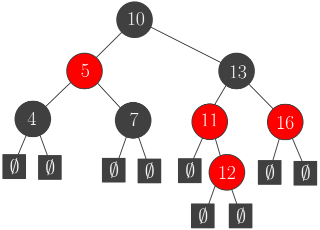
Rotationen

Linksrotation um einen Knoten v

Rechtsrotation um einen Knoten w

Red black trees





`Insert`: Einfügen eines neuen Knotens

- zunächst verfahren wir genau wie bei “gewöhnlichen” binären Suchbäumen
- der neue Knoten wird rot gefärbt
- die $\varnothing$-pointer werden auf das $\varnothing$-Objekt gesetzt
- Vorsicht: RB2 oder RB4 könnten verletzt sein!

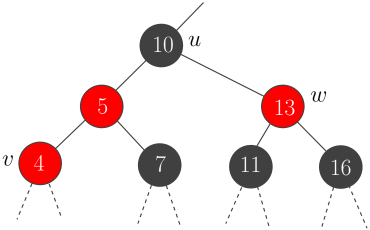
Red black trees

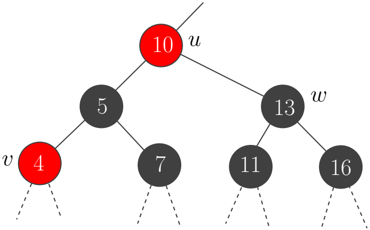
Wiederherstellen von RB2 und RB4

- solange der Elternknoten des eingefügten Knotens v rot, ist, betrachten wir drei Fälle
- Fall 1: die “Tante” w von v ist rot
- Fall 2: w ist schwarz und v ist ein rechtes Kind
- Fall 3: w ist schwarz und v ist ein linkes Kind

die drei Fälle lassen sich am besten anhand von Beispielen erklären; dabei nehmen wir an, daß der Elternknoten von v ein linkes Kind ist; sonst sind “links” und “rechts” jeweils zu vertauschen!

Red black trees

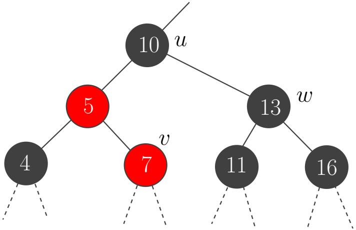


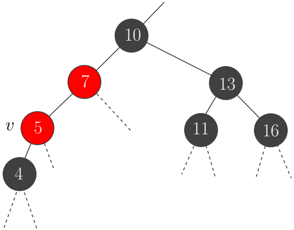


Fall 1: roter Elternknoten und rote Tante

- färbe Elternknoten und die Tante schwarz
- färbe den Großelternknoten u rot
- führe die Wiederherstellung rekursiv für u aus

Red black trees

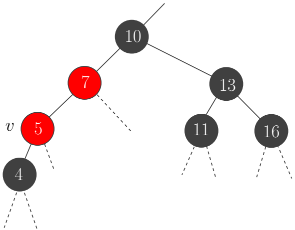


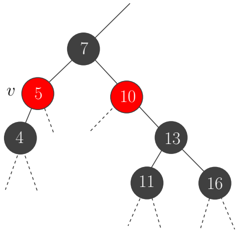


Fall 2: roter Elternknoten, schwarze Tante, rechtes Kind

- führe eine Linksrotation um v durch
- weiter mit Fall 3

Red black trees





Fall 3: roter Elternknoten, schwarze Tante, linkes Kind

- färbe den Elternknoten von v schwarz und den Großelternknoten u rot
- dann Rechtsrotation um den Großelternknoten u

Red black trees

Abschluß Wiederherstellung

- am Ende der Wiederherstellungsoperation wird die Wurzel schwarz gefärbt
- dadurch wird RB2 garantiert

Red black trees

Proposition

Die Einfügeoperation inkl. Wiederherstellung hat Laufzeit $O(\log n)$ und stellt die Eigenschaften RB1–RB5 her.

Beweis

- in den Fällen 2 und 3 ist die Gesamtlaufzeit $O(1)$
- in Fall 1 bewegt sich v auf die Wurzel zu
- wir stoppen also nach $O(\log n)$ Schritten

Red black trees

Entfernen eines Knotens z

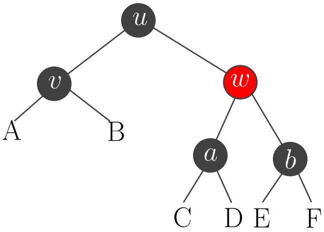
- verfahre wie beim Löschen von z in einem binären Suchbaum
- sei y der Knoten, der die Stelle von z einnimmt
- y übernimmt die Farbe von z
- sei v das Kind von y, bzw. $\emptyset$, falls y kein Kind hat
- *dabei identifizieren wir $\emptyset$ mit dem $\emptyset$-Objekt des Baums*

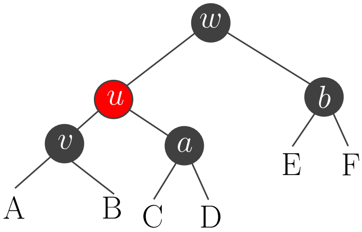
Red black trees

Wenn y schwarz war, können Verletzungen von RB1–RB5 eintreten

- möglicherweise war y die Wurzel; das Kind v tritt an die Stelle von y, wird somit die neue Wurzel, ist aber womöglich rot; also RB2 verletzt
- wenn v und der Elternknoten u von z rot sind, ist RB4 verletzt
- die Pfade, die vormals y enthalten haben, enthalten jetzt einen schwarzen Knoten weniger; somit RB5 verletzt
- wir können uns v, der an die Stelle von y tritt, als einen “doppelt schwarzen” Knoten vorstellen
- *wir unterscheiden vier Fälle, die nach der Farbe des Schwesterknotens w von v und den Farben der Kinder von w unterscheiden*
- *dabei nehmen wir an, daß v ein linkes Kind ist; andernfalls sind “links” und “rechts” zu vertauschen*

Red black trees

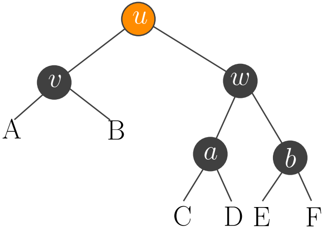


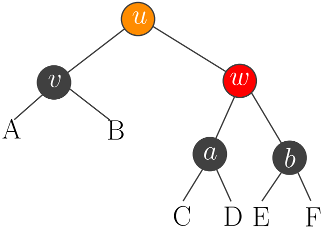


Fall 1: roter Schwesterknoten w

- vertausche die Farben von w und u
- führe eine Linksrotation um u aus
- fahre mit Fall 2/3/4 fort, wobei nun $w = a$ der neue Schwesterknoten ist

Red black trees

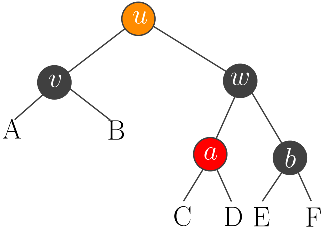


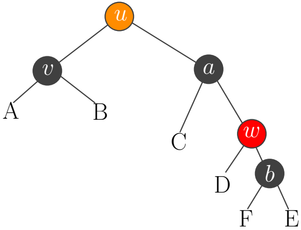


Fall 2: schwarzer Schwesterknoten w, beide Kinder von w sind schwarz

- färbe den Schwesterknoten w rot
- fahre rekursiv mit $v = u$ fort
- (der Knoten u kann rot oder schwarz sein)

Red black trees

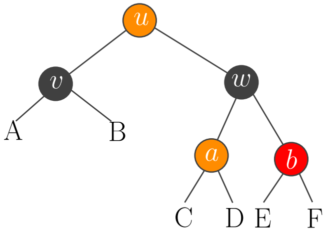


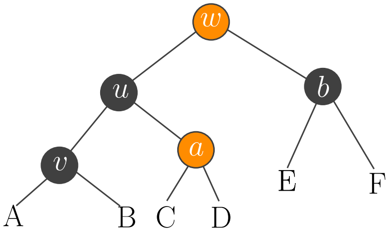


Fall 3: schwarzer Schwesterknoten w, linkes Kind von w rot, rechtes Kind schwarz

- vertausche die Farben von w und seinem linken Kind
- Rechtsrotation um w
- fahre fort mit Fall 4, mit neuem Schwesterknoten $w = a$

Red black trees





Fall 4: schwarzer Schwesterknoten w, rechtes Kind von w rot

- Linksrotation um u
- Farben anpassen $\rightsquigarrow$ Bedingung RB5 nun erfüllt; setze v auf die Wurzel des Baums

Red black trees

Abschluß

- sobald der aktuelle Knoten v schwarz gefärbt ist, sind wir fertig
- wenn v die Wurzel ist, färben wir v schwarz und sind dann ebenfalls fertig

Red black trees

Proposition

Die Laufzeit zum Entfernen eines Knotens ist $O(\log n)$.

Beweis

- in den Fällen 1,3 und 4 hält der Algorithmus nach $O(1)$ Schritten
- in Fall 2 bewegen wir uns auf die Wurzel zu

Red black trees

Zusammenfassung

- red black trees sind extrem effiziente selbstbalancierende binäre Suchbäume
- Beispielanwendung: Prozessorscheduling im Linux-Kern