B-Bäume und Rot-Schwarz-Bäume

Frank Schönmann Falk Schubert

SS 2001

Inhaltsverzeichnis

1 Nachteile normaler Binärbäume

Normale Binärbäume sind für eine Menge von Anwendungen sehr gut geeignet, da sie relativ einfach zu implementieren sind und Operationen darauf im Idealfall sehr effizient ablaufen. Das Problem ist allerdings, dass mit einiger Wahrscheinlichkeit der ungünstigste Fall auftritt, nämlich ein komplett rechts- oder linkslastiger Binärbaum wie in Abbildung 1.

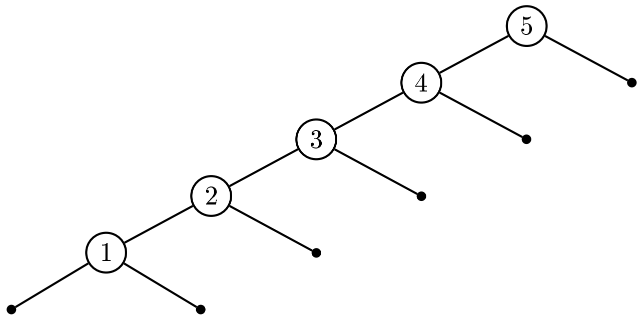

Abb. 1: Der *worst case* eines Binärbaums.

Im Gegensatz zur logarithmischen Effizienz im Bestfall, bei der sich die Anzahl der zu durchsuchenden Knoten in jedem Schritt halbiert, läuft eine Suche in einem solchen Baum nur in einem linearen Zeitaufwand ab, da sich die Zahl der Elemente bei jedem Schritt nur um eines verringert.

Dieser Fall kann zum Beispiel eintreten, wenn man bereits vorsortierte Daten, eine in umgekehrter Reihenfolge sortierte Liste oder Daten, die einen großen Abschnitt mit einfacher Struktur enthalten, auf normalen Weg in einen Binärbaum einfügt.

Es gibt jedoch Methoden, diesen *worst case* zu umgehen. Eine zum Beispiel im Fall des Sortieralgorithmus QuickSort häufig verwendete Technik ist die Randomisierung. Dabei wählt man das nächste in den Baum einzufügende Element zufällig aus. Auf diese Weise kann man die Gesetze der Stochastik ausnutzen, denn es ist sehr unwahrscheinlich, dass man dabei ausgerechnet das kleinste oder größte Element einer Liste erwischt. Genauer gesagt beträgt die Wahrscheinlichkeit, bis zum i-ten Schritt aus einer Liste der Länge n jedesmal das kleinste – oder das größte – Element zufällig auszuwählen:

$$\frac{1}{n} \cdot \frac{1}{n-1} \cdot \ldots \cdot \frac{1}{n-(i-1)} = \prod_{\nu=1}^{i} \frac{1}{n-(\nu-1)} = \prod_{\nu=0}^{i-1} \frac{1}{n-\nu}$$

Das heißt, die Wahrscheinlichkeit für einen solchen Fall nimmt sehr stark ab je länger die Liste wird.

Eine bessere Möglichkeit, die garantiert, dass der *worst case* niemals eintritt, ist das *balancing*. Darunter versteht man, den Baum bei jedem Einfüge- und Löschvorgang von Schlüsseln ausgeglichen zu halten. Auf diese Weise ist der Baum zu jedem Zeitpunkt balanciert[1], wodurch eine logarithmische Effizienz bei jeder Operation gewährleistet wird.

[1] D.h. jeder Knoten eines Baums hat an jeder Verkettung gleich viele Nachfolger.

2 Top-Down 2-3-4-Bäume

Binäre Suchbäume sind in manchen Fällen sehr ineffizient. Im *worst case* muss man beim Einfügen oder Löschen von Schlüsseln viele Rotationen von Knoten durchführen. Um dies zu verhindern, benötigt man eine flexiblere Datenstruktur, die es ermöglicht, mehrere Schlüssel pro Knoten zu speichern. 2-3-4 Bäume bieten diese flexible Datenstruktur. Die Knoten dürfen zwei bzw. drei Schlüssel enthalten. Da auch die 2-3-4 Bäume, wie die Binärbäume, geordnet und ausgeglichen sind, werden Verkettungen zwischen den Schlüsseln eines Knotens angeknüpft.

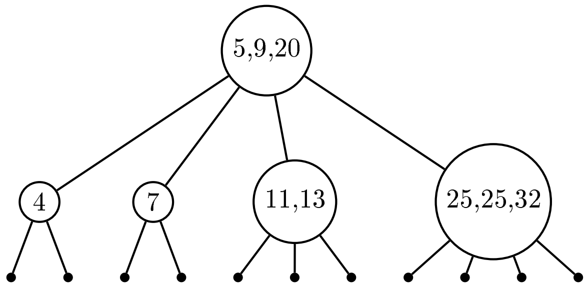

Abb. 2: Beispiel eines 2-3-4-Baums.

Das heißt, alle Schlüssel, die kleiner als der zweite Schlüssel und größer als der erste Schlüssel sind, befinden sich in einem Knoten des Unterbaums, der zwischen dem ersten und zweiten Schlüssel angeknüpft ist. Somit sind für Knoten mit einem Schlüssel zwei, für Knoten mit zwei Schlüsseln drei und für eine Knoten mit drei Schlüsseln vier Verknüpfungen möglich[2]. Einen normalen Binärbaum könnte man daher als einen 2-3-4-Baum auffassen, der nur aus 2-Knoten besteht.

Wir wollen nun die wichtigsten Operationen auf den *Top-Down* 2-3-4-Bäumen näher erklären.

2.1 Suche nach einem Schlüssel

Beim Suchen nach einem Schlüssel vergleicht man das Suchelement mit den Schlüsseln eines Knotens. Dabei überprüft man, zwischen welche Schlüssel eines Knotens das Suchelement passt, und verfolgt die dort angeknüpfte Verkettung zum Unterbaum. Ist das Element in einem Knoten enthalten, hat man den gesuchten Schlüssel gefunden.

2.2 Einfügen eines Schlüssels

Beim Einfügen von Knoten in einen 2-3-4-Baum wird eine Suche durchgeführt. Trifft man dabei auf einen 2- oder 3-Knoten kann das Element einfach eingefügt werden. Wenn man 4-Knoten hat, kann man trivialerweise das einzufügende Element einfach als Nachfolger anhängen. Dadurch wir der Baum allerdings

[2] Diese Knoten werden auch 2-Knoten, 3-Knoten bzw. 4-Knoten genannt, wodurch sich der Name der Baumstruktur ableitet.

unausgeglichen. Es ist daher besser, beim Suchen baumabwärts[3] alle 4-Knoten zu beseitigen. Durch dieses Verfahren gibt es nur sehr wenige 4-Knoten in einem Baum.

Die Beseitigung von 4-Knoten lässt sich folgendermaßen realisieren: begegnet man einem 4-Knoten, wird der darüber liegende Knoten um ein Element erweitert und der 4-Knoten in zwei 2-Knoten aufgespalten. Ein 4-Knoten an der Wurzel wird immer in drei 2-Knoten aufgespalten. Man erkennt leicht, daß lediglich das Aufspalten der Wurzel den Baum in seiner Tiefe um eine Ebene erweitert. Ein permanentes Aufspalten der Wurzel erweist sich als einfacher als zu warten, bis der reguläre Einfüge-Algorithmus die Auftrennung bewirkt.

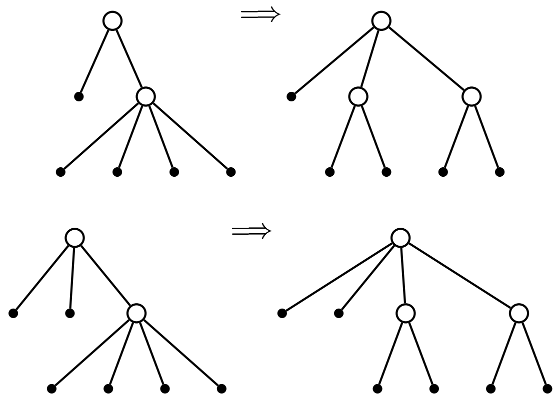

Abb. 3: Auftrennung von 3- bzw. 4-Knoten.

Der Vorteil, der sich durch die flexiblere Datenstruktur ergibt, besteht darin, dass der Baum nur lokal verändert werden muss. Dabei werden lediglich Schlüssel übergeben. Die Umstrukturierung ist ebenfalls verhältnismäßig einfach, da man lediglich die Zeiger bewegen muss. Der Aufspaltevorgang kann auch *Bottom-Up*[4] ausgeführt werden. Die entsprechende Implementierung ist aber wesentlich schwieriger.

Die Transformationen garantieren, dass die unterste Ebene frei von 4-Knoten ist, so dass man ein Element jederzeit einfach einfügen kann. Interessanterweise ergeben diese Transformationen immer wieder vollständig balancierte Bäume.

Die Tiefe eines 2-3-4-Baums ist im *worst case* $\log N$, nämlich genau dann, wenn der Baum lediglich aus 2-Knoten besteht. Jeder weitere 3- oder 4-Knoten verringert die Tiefe. Die Transformationen verändern auch nur an der Wurzel die Tiefe, dann aber für alle Knoten.

Die genannten Vorteile der 2-3-4-Bäume zeigen sich auch in der Effizienz der einzelnen Operationen auf den Bäumen.

2.3 Effizienz von 2-3-4-Bäumen

Da der Baum vollständig ausgeglichen ist, werden bei der Suche nach einem Element in einem 2-3-4-Baum maximal $\log N + 1$ Knoten betrachtet.

[3] *Top-Down*; ebenfalls für die Namensgebung verantwortlich

[4] also baumaufwärts

Beim Einfügen werden maximal $\log N + 1$ Aufspaltungen durchgeführt, wenn alle Knoten auf dem Weg nach unten 4-Knoten sind. Durchschnittlich muss man auf jedem Suchpfad anscheinend weniger als eine Aufspaltung eines Knotens durchführen[5]. Weiterhin befinden sich 4-Knoten fast nur auf der untersten Ebene. Auch dies wurde durch empirische Untersuchungen bestätigt.

2.4 Implementierung von 2-3-4-Bäumen

Die Implementierung ist generell schwierig. Die komplexen Knotenstrukturen mit einer variablen Zahl an Nachfolgern erzeugen bei den Operationen Ballast, der den Algorithmus langsamer machen kann als vergleichbare Operationen auf unausgeglichenen Binärbäumen.

[5] empirische Tests beweisen dies

3 Rot-Schwarz-Bäume

Das Problem mit der Implementierung von 2-3-4-Bäumen ist also der Overhead bei Operationen auf Bäumen mit mehr als zwei Verkettungen. Daher hat man mit den Rot-Schwarz-Bäumen eine bemerkenswerte Methode entwickelt, 2-3-4-Bäume als normale Binärbäume darzustellen. Dabei verfährt man auf die folgende Art und Weise:

Die 3- und 4-Knoten des 2-3-4-Baums werden als kleine binäre Bäume dargestellt, die durch rote Verkettungen miteinander verbunden sind. Im Gegensatz dazu stehen die schwarzen Verkettungen, die den 2-3-4-Baum zusammenhalten. Abbildung 4 soll dieses Verfahren anschaulich machen (rote Verkettungen werden von nun an fett dargestellt, schwarze Verkettungen normal).

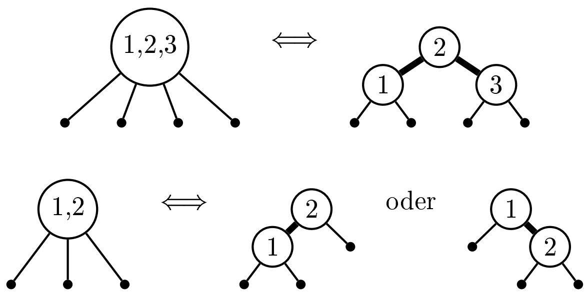

Abb. 4: Konvertierung von 2-3-4-Bäumen in Rot-Schwarz-Bäume.

Entscheidend bei diesem Verfahren ist, dass beide Darstellungsformen absolut äquivalent zueinander sind. Man kann also jederzeit aus einem 2-3-4-Baum einen korrespondierenden Rot-Schwarz-Baum erzeugen und umgekehrt. Daher gelten grundsätzlich für beide Arten von Bäumen die gleichen Eigenschaften. Die Darstellung als Binärbaum dient lediglich dem Zweck, Overhead bei der Arbeit auf Knoten mit einer variablen Zahl von Nachfolgern zu vermeiden.

Noch eine Bemerkung zur Konvertierung der 3-Knoten: die „Neigung" des entstehenden (Mini-)Binärbaums kann in diesem Fall dynamisch erfolgen. Es wäre zwar möglich, eine Regel einzuführen, nach der alle diese Knoten in der gleichen Weise geneigt sein müssten, aber dafür gibt es keinen Grund. Das bedeutet auch, dass es je 2-3-4-Baum mehrere äquivalente Rot-Schwarz-Bäume geben kann.

Bei der Implementierung der Datenstruktur für einen Rot-Schwarz-Baum erhöht sich der Speicherbedarf gegenüber normalen Binärbäumen nur um ein Bit pro Knoten, in dem die Farbe eines Knotens bzw. der Verkettung zu diesem Knoten gespeichert wird. Die restliche Datenstruktur ist absolut identisch.

3.1 Eigenschaften von Rot-Schwarz-Bäumen

Rot-Schwarz-Bäume haben viele Eigenschaften, die sich direkt aus ihrer Definition und ihrer Herleitung aus den 2-3-4-Bäumen ergeben. Die beiden wichtigsten Eigenschaften hängen mit den roten und schwarzen Verkettungen zusammen.

red condition: jeder rote Knoten hat einen schwarzen Vorgänger. Man könnte auch sagen, auf keinem Pfad treten zwei rote Verkettungen hintereinander auf.

Aus dieser Bedingung folgt auch, dass die Wurzel eines Rot-Schwarz-Baums niemals rot sein darf.

black condition: jeder Pfad von der Wurzel zu einem der Blätter hat gleich viele schwarze Verkettungen. Die Anzahl dieser Verkettungen nennt man *black height* oder *black depth*.

Diese Eigenschaften kann man sich am besten veranschaulichen, indem man einen typischen Rot-Schwarz-Baum wie in Abbildung 5 betrachtet.

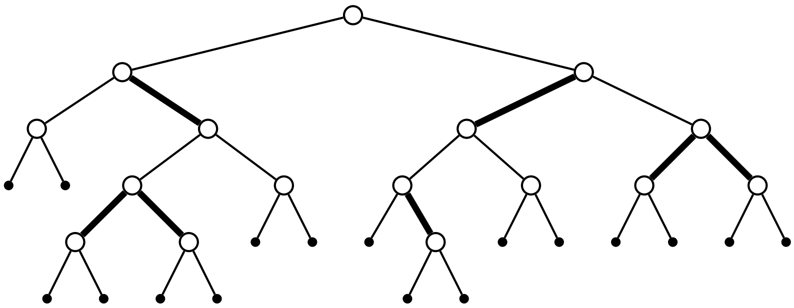

Abb. 5: Ein typischer Rot-Schwarz-Baum.

Eine weitere wichtige Eigenschaft folgt direkt aus den beiden eben genannten: obwohl es möglich ist, dass ein Pfad (schwarz-rot im Wechsel) doppelt so lang ist wie ein anderer (nur schwarz), sind trotzdem alle Pfadlängen nach wie vor proportional zu $\log N$, nämlich maximal $2 \log N$ lang.

Bemerkung: lässt man mehrfaches Auftreten eines Schlüssels zu – dies kann man umgehen, indem man zum Beispiel unter einem Schlüssel eine Liste statt eines einfachen Wertes speichert – kann dieser sowohl rechts als auch links eines Knotens existieren, da sonst lange Ketten von gleichen Schlüsseln den Baum aus der Balance bringen könnten. Allerdings ist es dadurch nicht möglich, alle diese Schlüssel durch Fortsetzung der Suchprozedur zu finden, da sie nicht unbedingt auf dem gleichen Pfad liegen.

Dadurch, dass die Datenstruktur sehr ähnlich zu der normaler Binärbäume implementiert werden kann, ist es unter Umständen[6] möglich, eine Suchfunktion auf Binärbäumen unverändert auf Rot-Schwarz-Bäumen laufen zu lassen, denn diese muss nur die bei beiden Daten-Arten vorhandenen Informationen nutzen, nicht aber die gespeicherte Farbe.

3.2 Implementation von Rot-Schwarz-Bäumen

Will man Rot-Schwarz-Bäume implementieren, ist vor allem bei Veränderungen des Baums zu beachten, dass die oben genannten Bedingungen nicht verletzt werden. Alle anderen Operationen arbeiten ähnlich oder genauso wie die entsprechenden Funktionen von reinen Binärbäumen. Da eine Implementation in imperativen Programmiersprachen relativ unübersichtlich würde, beschränken wir uns hier auf die Programmierung in Haskell. Tatsächlich basieren in einigen anderen Sprachen bestimmte Datentypen aus Effizienzgründen bereits auf Rot-

[6] D.h. in bestimmten Programmiersprachen, zum Beispiel C, C++ oder Java.

Schwarz-Bäumen. In Java ist dies beispielsweise die Klasse `Java.util.TreeMap`. Für weitere Informationen verweisen wir auf den frei verfügbaren Quellcode.

Wollen wir die Bäume in Haskell nutzen, benötigen wir zuallererst einmal eine geeignete Datenstruktur. Die folgende Definition von `RBTree` übernimmt diese Aufgabe.

```
data RBTree a = Nil
              | Br Color a (RBTree a) (RBTree a)
              deriving (Eq, Show)
data Color    = Blk
              | Red
              deriving (Eq, Show)
```

Wie man leicht sieht, ist die Datenstruktur rekursiv aufgebaut. Man kann auch die starke Ähnlichkeit zur Definition normaler Binärbäume erkennen. Eigentlich unterscheiden sich die beiden lediglich dadurch, dass hier jeder Knoten noch eine Farbe erhält. Ein einfacher Rot-Schwarz-Baum, der diesem Datentyp angehört, könnte etwa folgendermaßen aussehen:

```
tree :: RBTree Integer
tree  = (Br Blk 4
          (Br Red 2 (Br Blk 1 Nil Nil) (Br Blk 3 Nil Nil))
          (Br Blk 5 Nil Nil)
        )
```

Die entsprechende graphische Darstellung dieses Baums zeigt Abbildung 6.

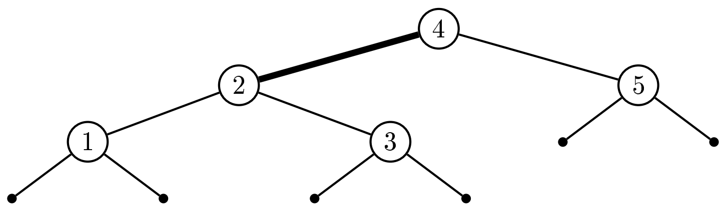

Abb. 6: Ein sehr einfacher Rot-Schwarz-Baum.

3.2.1 Einfache Operationen

Auf dieser Datenstruktur wollen wir nun einige wenige, einfache Funktionen definieren, um uns damit etwas vertrauter zu machen.

```
labels                :: RBTree a -> [a]
labels Nil             = []
labels (Br _ a l r)    = labels l ++ [a] ++ labels r

size                  :: RBTree a -> Integer
size Nil               = 0
size (Br _ _ l r)      = 1 + size l + size r
```

```
bheight                 :: RBTree a -> Integer
bheight Nil              = 0
bheight (Br Blk _ l _) = 1 + bheight l
bheight (Br Red _ l _) = bheight l
```

Diese Funktionen sind so einfach gehalten, dass sie eigentlich keiner weiteren Kommentierung bedürfen. `labels` erzeugt aus allen Schlüsseln eines Baums eine Liste, die – ebenso wie der Baum – sortiert ist. `size` berechnet die Anzahl der enthaltenen Schlüssel. Diesen Wert bezeichnen wir als Größe des Baums. die Funktion `bheight` ermittelt die *black height*, also die Anzahl schwarzer Verkettungen auf jedem Pfad von der Wurzel zu einem Blatt.

3.2.2 Suche

Die Suche nach Schlüsseln in einem Rot-Schwarz-Baum funktioniert absolut analog zur Suche in normalen Binärbäumen. Wie bereits oben erwähnt, kann man in einigen Programmiersprachen sogar diese Suchfunktion direkt übernehmen. Daher ist anzunehmen, dass sie auch in Haskell nicht viel anders aussieht:

```
search                 :: (Ord a) => a -> RBTree a -> Bool
search _ Nil            = False
search x (Br _ a l r)
    | x == a            = True
    | x < a             = search x l
    | x > a             = search x r
```

Die Funktion sucht einen Schlüssel vom Typ `a` in einem Rot-Schwarz-Baum dieses Typs. Findet sie den Schlüssel, wird `True` zurück gegeben. Hier kann man auch sehr schön sehen, dass bei den Knoten des Baums die Farbe nicht von Belang ist: `search x (Br _ a l r)` ignoriert nämlich einfach die Farbe des Knotens.

3.2.3 Einfügen

Einfügeoperationen auf Rot-Schwarz-Bäumen sind leider bedeutend komplizierter als bei normalen Bäumen und alle obigen Funktionen, da bei Veränderungen darauf geachtet werden muss, dass jeder Teilbaum balanciert bleibt.

Die Operation funktioniert nach folgendem Grundprinzip: im Baum wird ein geeigneter Pfad bis nach ganz unten verfolgt. Dort wird ein roter Knoten mit dem gewünschten Schlüssel als Wert eingefügt. Da der Baum nun unausgeglichen ist und evtl. zwei aufeinander folgende rote Knoten enthält, wird der Baum an jedem Knoten, der auf dem Pfad liegt, nach bestimmten Regeln rotiert. Danach wird die Wurzel auf jeden Fall schwarz gefärbt, weil dort kein roter Knoten stehen darf[7].

```
insert    :: (Ord a) => a -> RBTree a -> RBTree a
insert x s = Br Blk y a b
  where
    Br _ y a b = ins s
    ins                    :: (Ord a) => RBTree a -> RBTree a
```

[7] Dies ist der Fall, in dem sich die Höhe des äquivalenten 2-3-4-Baums vergrößern kann.

```
ins Nil               = Br Red x Nil Nil
ins (Br color y l r)
  | x < y             = balance color y (ins l) r
  | x > y             = balance color y l (ins r)
  | otherwise         = Br color y l r
```

Die Funktion geht den Baum nach unten, bis sie an einem leeren Blatt (`Nil`) den Schlüssel als roten Knoten einfügt. Auf dem Weg wird an jeder Abzweigung für den entsprechenden Knoten die `balance`-Funktion aufgerufen.

```
balance :: Color -> a -> RBTree a -> RBTree a -> RBTree a
balance Blk z (Br Red y (Br Red x a b) c) d
                    = Br Red y (Br Blk x a b) (Br Blk z c d)
balance Blk z (Br Red x a (Br Red y b c)) d
                    = Br Red y (Br Blk x a b) (Br Blk z c d)
balance Blk x a (Br Red z (Br Red y b c) d)
                    = Br Red y (Br Blk x a b) (Br Blk z c d)
balance Blk x a (Br Red y b (Br Red z c d))
                    = Br Red y (Br Blk x a b) (Br Blk z c d)
balance color a l r = Br color a l r
```

Die Funktion sieht zwar kompliziert aus, unterscheidet aber eigentlich nur vier Fälle. In allen anderen Fällen wird der Baum nicht verändert (letzte Zeile). Abbildung 7 zeigt die Möglichkeiten, die unterschieden werden.

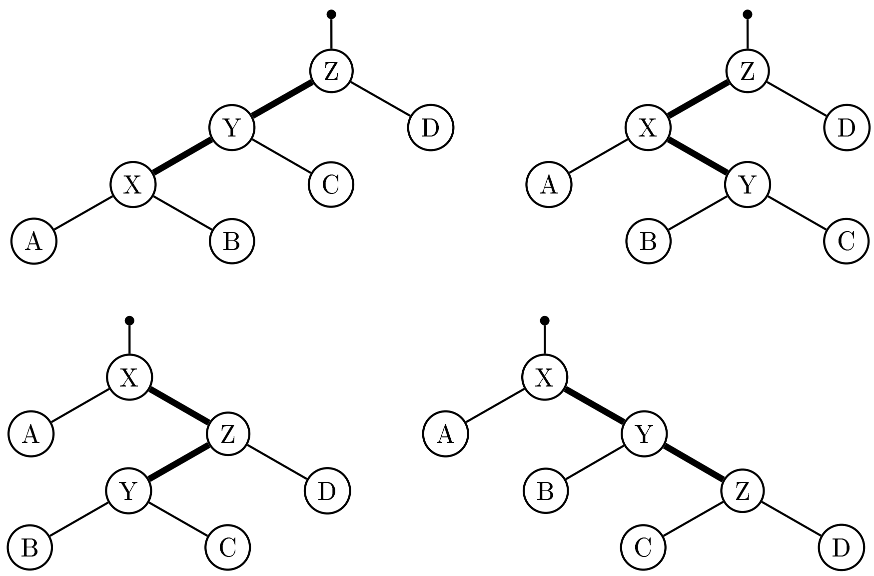

Abb. 7: Vier Fälle, die durch `balance` ausgeglichen werden.

Durch all diese Rotations-Operationen wird die Ausgeglichenheit gewährleistet, denn jede dieser Möglichkeiten werden durch den selben Teilbaum ersetzt, nämlich den in Abbildung 8 gezeigten.

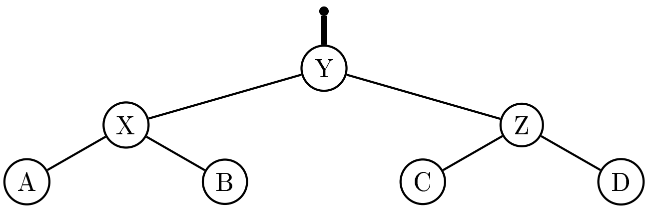

Abb. 8: Ergebnis-Teilbaum der `balance`-Funktion.

3.2.4 Löschen

Lösch-Operationen auf Rot-Schwarz-Bäumen sind mindestens genauso kompliziert wie eine Einfüge-Operation, da auch hier der Baum verändert wird, aber die Bedingungen nicht verfälscht werden dürfen. Das größte auftretende Problem ist, dass bei der Entfernung eines schwarzen Knotens die *black height* des Baums für alle Nachfolger verringert wird.

Um dieses Problem zu umgehen, verwandelt man es in ein anderes: man nimmt an dass ein geeigneter Knoten doppelt schwarz gefärbt ist – auch bezeichnet als *double black problem*. Diese *black*-Eigenschaft befördert man auf dem Pfad nach oben, bis man auf einen roten Knoten trifft, der dann in einen schwarzen umgefärbt werden kann. Das Fehlen des roten Knotens kann durch Rotation bzw. Umfärbung des anderen Nachfolgers behoben werden.

Da eine detaillierte Betrachtung des entsprechenden Algorithmus den Rahmen dieser Ausarbeitung sprengen würde, verweisen wir hier auf die Haskell-Implementierung von Rot-Schwarz-Bäumen in [3].

3.3 Effizienz von Rot-Schwarz-Bäumen

Die wichtigste Operation auf Rot-Schwarz-Bäumen ist im normalen Anwendungsfall die Suche, da ein Schlüssel nur einmal in einen Baum eingefügt, aber danach sehr häufig gesucht wird. Durchschnittlich treten dann kürzere Suchzeiten auf, da die Bäume immer ausgeglichen sind. Sogar im entarteten Fall, wie ihn Abbildung 9 zeigt[8], ist der Baum noch relativ gut ausgeglichen. Die Suche gehört also der Effizienzklasse $O(\log N)$ an, im *worst case* liegt die Effizienz der Suche in $\Theta(2 \log N)$, da der maximale Suchpfad dann doppelt so lang ist wie im *best case*.

Die Effizienz von Rot-Schwarz-Bäumen beim Einfügen lässt sich direkt aus der entsprechenden Effizienzklasse von 2-3-4-Bäumen ableiten. Zur Erinnerung: dort erfordern Einfügungen in Bäume mit N Knoten im ungünstigsten Fall weniger als $\log N + 1$ Aufspaltungen von Knoten[9]. Nur Aufspaltungen, die einem mit einem 4-Knoten verbundenen 3-Knoten im äquivalenten 2-3-4-Baum entsprechen, erfordern im Rot-Schwarz-Baum eine Rotation. Der *worst case* liegt also vor, wenn der Pfad zum Ort des Einfügens aus sich abwechselnden 3- und 4-Knoten besteht.

[8] Dieser Fall kann zum Beispiel beim Einfügen einer sortierten Liste in einen Rot-Schwarz-Baum auftreten.

[9] siehe Abschnitt 2.3, Seite 3

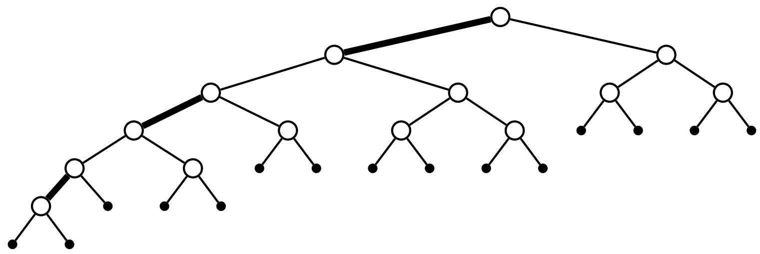

Abb. 9: Entarteter Fall eines Rot-Schwarz-Baums.

Grundsätzlich gehören alle Funktionen, die auf Rot-Schwarz-Bäumen arbeiten, der Effizienzklasse $O(\log N)$ an. Dies hängt damit zusammen, dass der Baum zu jedem Zeitpunkt ausgeglichen ist und der längste Pfad höchstens doppelt so lang ist wie der kürzeste Pfad (siehe Abbildung 9). Da es nur sehr wenige Suchalgorithmen mit dieser Eigenschaft gibt, und Rot-Schwarz-Bäume sich trotz dieser Eigenschaft recht einfach implementieren lassen, sind sie immer dort von Nutzen, wo der *worst case* auf jeden Fall vermieden werden soll.

4 B-Bäume

B-Bäume stellen eine verallgemeinerte Form der Binärbäume dar. Die Flexibilität der Knotenstruktur ist sehr hoch, denn die Knoten dürfen M viele Schlüssel beinhalten. Sie werden vorallem bei der Suche nach Schlüsseln benutzt, die auf mehreren Speichermedien verteilt liegen. Dabei ist in der Regel der Zugriff pro Datenblatt[10] der limitierende Faktor. Man versucht also die Zugriffe auf die Datenblätter so gering wie möglich zu halten. Somit hängt die genaue Knotenstruktur stark von dem Speichermedium[11] ab.

Bei der Definition gibt es verschiedene Ansichten. Bayer & McCreight beschreiben die B-Bäume als eine exakte Datenstruktur, andere Autoren verwenden die Bezeichnung B-Bäume allgemeine für externe ausgeglichene Bäume.

Die Knoten können 1 bis $M-1$ Schlüssel[12] und dementsprechend 2 bis M Verzweigungen pro Knoten enthalten.

4.1 Operationen auf B-Bäumen

Die Operationen auf den B-Bäumen sind im Prinzip eine Verallgemeinerung der Operationen auf 2-3-4-Bäumen[13]. Bei der Suche verfolgt man die Verkettungen im jeweils passenden Intervall des Knotens, bis man den Suchschlüssel gefunden hat.

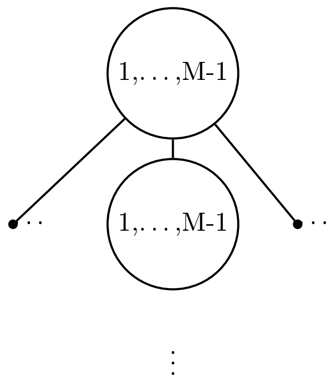

Abb. 10: Allgemeines Beispiel eines B-Baums.

Der Einfügevorgang gestaltet sich ebenfalls dem der 2-3-4-Bäume ähnlich. Trifft man auf einen Knoten der mit $M-1$ Knoten verbunden ist, so kann man den Schlüssel einfach einfügen. Ist der Knoten aber schon voll belegt, dann muss wieder ein Aufspaltevorgang erfolgen. Dabei wird bei dem darüber liegenden K-Knoten ein Schlüssel eingefügt, womit dieser zu einem $K+1$-Knoten wird. Der voll belegte M-Knoten kann dann in zwei Teile aufgespalten werden. Dadurch entsteht garantierter Platz für das Einfügen neuer Knoten.

[10] Man stelle sich ein Fesplatte vor, deren gesamter Inhalt zu groß für den Hauptspeicher ist, so dass nur Teile – also ein Datenblatt – in den Hauptspeicher geladen werden können.

[11] Festplatte, Festplatten-Raid, Magnetband, Optical Disk

[12] Im Regelfall handelt es sich um einige tausend Schlüssel pro Knoten.

[13] siehe Abschnitt 2.1, Seite 2

Ein Vorteil der B-Bäume ist der hohe Verzweigungsgrad, der eine maximale Höhe des Baums von $O(\log_M N)$ bewirkt.

4.2 Effizienz von B-Bäumen

Beim Suchen bzw. Einfügen in einen B-Baum der Ordnung M mit N Datensätzen benötigt man weniger als $log_{\frac{M}{2}} N$[14], denn alle Knoten im Inneren haben zwischen $\frac{M}{2}$ und M Schlüsseln, da sie höchstens größer sind oder aufgespalten werden. Der *worst case* ergibt sich genau dann, wenn alle Knoten $\frac{M}{2}$ Schlüssel haben.

Durch Vorverabeitung kann man sich einen Index über die einzelnen Knoten anlegen. Dieser passt in der Regel in den Hauptspeicher und erleichtert die Suche. Dies eignet sich allerdings nur für Bäume, deren Inhalt nicht ständig verändert wird, da sonst häufig diese Indizes erstellt werden müssten.

Zusätzlich kann man nun die Datensätze von den Verkettungen trennen. Dabei beinhalten die inneren Knoten nur Schlüssel und Verkettungen und die äußeren Knoten enthalten die eigentlichen Datensätze. Dies entspricht der Funktion eines Indexes und verbessert die Effizienz einer Suche, da weniger zeitaufwendige Plattenzugriffe durchgeführt werden müssen.

Für B-Bäume gibt es ebenfalls *Top-Down-* und *Bottom-Up-*Versionen, die Knoten aufzuspalten. Man kann die Spaltungsregeln verbessern, indem man zulässt, dass Knoteninhalte auch auf nicht volle Geschwisterknoten verlagert werden. Dies hat eine bessere Platzausnutzung zur Folge.

[14] N sei für praktische Zwecke und größere M konstant.

5 Alternative balancierte Bäume: AVL-Bäume

Damit die grundlegendene Operationen auf Binärbäumen in $O(\log \mathrm{N})$ ausführbar sind, darf die Höhe eines Baumes im Schnitt nicht größer als $O(\log \mathrm{N})$ sein. Weiterhin muss der Baum balanciert sein. Es gibt daher verschiedene Methoden Bäume balanciert zu halten. Der historisch erste Vorschlag wurde 1962 von Adelson-Velskij und Landis gemacht. Sie forderten, dass die Höhendifferenz der beiden Teilbäume eines jeden Knotens in ihrer Höhe nur um eins unterscheidet. Solche Bäume werden auch als AVL-Bäume bezeichnet.

Man kann beweisen, dass dadurch die Gesamthöhe des Baumes immer in $O(\log \mathrm{N})$ liegt und somit die Operationen wie Einfügen, Suchen und Löschen in logarithmischer Zeit durchführbar sind.

Die Implementierung der Operationen ist denen der Binärbäume sehr ähnlich. Man muss lediglich beachten, dass beim Einfügen bzw. Löschen die Höhendifferenz der Teilbäume erhalten bleibt. Gegebenfalls muss man dann den Baum einfach oder sogar doppelt rotieren. Bei der Implementierung benötigt man dazu nur einen Balancefaktor, der jedem Knoten zugeordnet wird. Dieser Faktor ergibt sich aus der Höhen-Differenz des rechten und des linken Teilbaums. Man erkennt leicht, dass der Balancefaktor nur 0, 1 oder -1 sein kann.

Literatur

[1] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*, chapter Red-Black Trees, pages 263–280. The MIT Press, 1990.

[2] Ralf Hinze. Constructing red-black trees, 1999.

[3] Okasaki et al. Implementation of red-black trees in haskell. `http://www.cs.ukc.ac.uk/people/staff/smk/redblack/rb.html`.

[4] Robert Sedgewick. *Algorithmen in C++*, chapter Ausgeglichene Bäume, pages 255–272. Addison-Wesley, München, 1992.

[5] Robert Sedgewick. *Algorithmen in C++*, chapter B-Bäume, pages 308–312. Addison-Wesley, München, 1992.