

## Inhalt

B-Tree .....	2
Allgemein.....	2
Definition .....	2
Stolperfallen ¶ .....	2
Folgerungen ¶ .....	3
Besondere B-Bäume ¶ .....	3
Suchen eines Schlüssels ¶ .....	3
Einfügen von Schlüsseln ¶ .....	4
Beispiel zu Fall 1: Es ist noch Platz ¶ .....	4
Beispiel zu Fall 2: Knotenüberlauf ¶ .....	5
Löschen eines Schlüssels ¶ .....	5
Fall 1: Schlüssel in Blatt ¶ .....	5
Fall 2 - 3 ¶ .....	6
Trivia ¶ .....	6
Beispiel ¶ .....	6
Hashing .....	6
Kleiner Überblick .....	<b>Fehler! Textmarke nicht definiert.</b>
Verwendungen .....	7
Hash Tables.....	7
Caches.....	7
Suchen von identischen Einträgen .....	7
Datenschutz.....	7
Suchen von ähnlichen Einträgen .....	7
Suchen von ähnlichen Teileinträgen .....	7
Eigenschaften .....	7
Data Normalization .....	8
Continuity .....	8
Non-invertible.....	8
Collision-resistant.....	8

## B-Tree

### Allgemein

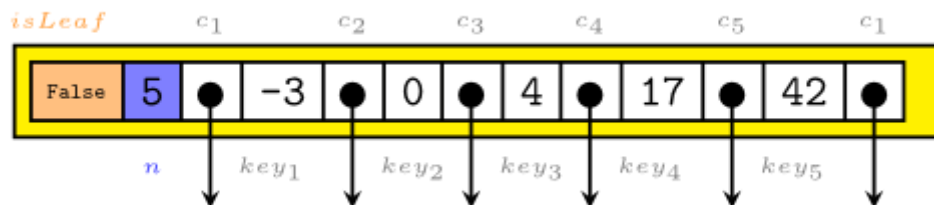
Ein B-Baum (englisch B-tree) ist in der Informatik eine Daten- oder Indexstruktur, die häufig in Datenbanken und Dateisystemen eingesetzt wird. Ein B-Baum ist ein immer vollständig balancierter Baum, der Daten nach Schlüsseln sortiert speichert. Er kann binär sein, ist aber im Allgemeinen kein Binärbaum. Das Einfügen, Suchen und Löschen von Daten in B-Bäumen ist in amortisiert logarithmischer Zeit möglich. B-Bäume wachsen und schrumpfen, anders als viele Suchbäume, von den Blättern hin zur Wurzel.

### Definition

Für einen B-Baum der Ordnung  $t$ ,  $t \in \mathbb{N} \setminus \{1\}$ , gilt:

1. Jeder Knoten  $xx$  hat die folgenden Attribute:
  1.  $n$ , die Anzahl der Schlüssel die im Knoten  $xx$  gespeichert wird,
  2. die  $n$  Schlüssel, die in aufsteigender Reihenfolge gespeichert werden (also  $key_1 \leq key_2 \leq \dots \leq key_n$ ),
  3.  $isLeaf$ , ein boolescher Wert der  $True$  ist, falls  $xx$  ein Blatt ist und  $False$  ist, falls  $xx$  ein innerer Knoten ist
2. Jeder innere Knoten hat  $n+1$  Zeiger  $c_1, c_2, \dots, c_{n+1}$  auf seine Kinder. Blattknoten haben keine Kinder, also sind ihre  $c_i$ -Attribute undefiniert.

Ein Knoten eines B-Baumes sieht also so aus:



Die Schlüssel  $key_i$  setzen Grenzen für die Werte der Schlüssel, die in den einzelnen Subbäumen gespeichert sind.

Falls  $k$  ein Schlüssel im Subbaum mit der Wurzel  $c_i$  ist, dann

gilt:  $k_1 \leq key_1 \leq k_2 \leq key_2 \leq \dots \leq key_n \leq k_n \leq key_{n+1}$

3. Alle Blätter haben die gleiche Tiefe.
4. Für die Anzahl der Schlüssel eines Knotens gilt:
  1. Jeder Knoten (bis auf die Wurzel) hat mindestens  $t-1$  Schlüssel.
  2. Jeder Knoten hat höchstens  $2t-1$  Schlüssel

### Stolperfallen

Ein Schlüssel ist etwas anderes als ein Zeiger!

## Folgerungen

- $(2) \Rightarrow (2) \Rightarrow$  Die Wurzel hat min. 2 Kinder, falls der Baum nicht leer ist.
- $(2), (5.1) \Rightarrow (2), (5.1) \Rightarrow$  Jeder innere Knoten (bis auf die Wurzel) hat min.  $t$  Kinder.
- $(2), (5.2) \Rightarrow (2), (5.2) \Rightarrow$  Jeder innere Knoten hat maximal  $2t$  Kinder.
- Wenn man an den Zeigern die Kinder in die Elterknoten zieht, sodass am Ende alle Knoten in der Wurzel sind, entsteht wegen (3) eine sortierte Liste.
- $h \leq \log_{t+1} n + 1$  (Cormen, S. 489)
- Ein B-Baum hat in der Tiefe  $h$  min.  $2^{h-1}$  Knoten.

## Besondere B-Bäume

Ein B-Baum der Ordnung  $t = 2$  wird auch 2-3-4-Baum genannt, da jeder Knoten entweder 2, 3 oder 4 Kinder hat.

Ich habe ja ausgeschlossen, dass es einen B-Baum der Ordnung  $t = 1$  gibt. Warum eigentlich? Aus (5.1) folgt: In einem B-Baum der Ordnung  $t = 1$  müssten einzelne Knoten keine Schlüssel haben. Das ist nicht sinnvoll. Also muss  $t \geq 2$  gelten.

## Suchen eines Schlüssels

Das Suchen eines Schlüssels funktioniert so:

*SEARCH-KEY(node, key):*

*int i = 0*

*while i < node.n and node.key[i].key < key:*

*i += 1*

*if node.key[i].key == key: # Schlüssel ist gesuchter Schlüssel*

*return node.key[i].satellitendaten*

*else if node.isLeaf: # Erfolgreiche Suche*

*return NIL*

*else: # Rekursiv weitersuchen*

*DISK-READ(node.c[i])*

*return SEARCH-KEY(node.c[i], key)*

Es wird also zuerst der Knoten durchsucht und dann gegebenenfalls der passende Subbaum.

Laut Vorlesung (Folie 97) gilt:

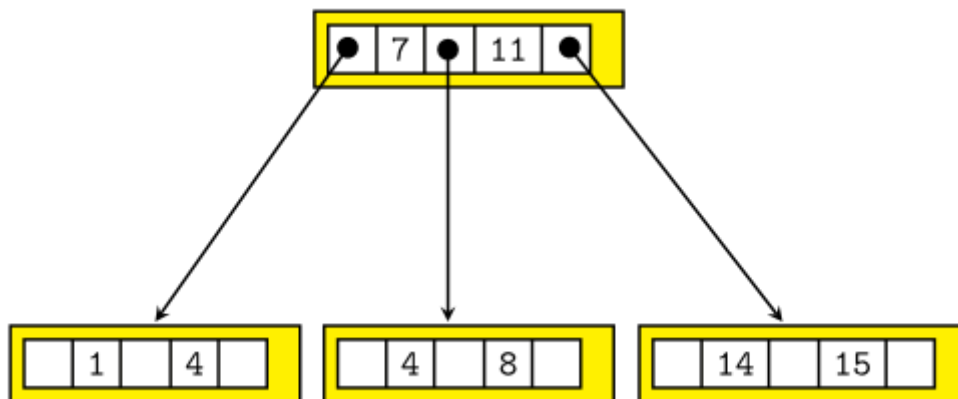
- Anzahl Zugriffe auf Hintergrundspeicher maximal:  $O(\log(n))O(\log_{t-1}(n))$
- Innerhalb eines Knotens:  $O(t)O(t)$
- Insgesamt also:  $O(t \cdot \log(n))O(t \cdot \log_{t-1}(n))$

Einfügen von Schlüsseln

Wenn ein Schlüssel in einen B-Baum eingefügt werden soll, dann muss man insbesondere auf Regel (5.1) und (5.2) achten: Jeder Knoten enthält  $n$  Schlüssel, mit  $t-1 \leq n \leq 2t-1$ .

Die Idee ist, dass man das Blatt sucht, in dem der Schlüssel sein müsste. Falls noch Platz ist, kann man den Schlüssel einfach einfügen. Falls nicht, muss man das Blatt aufsplitten.

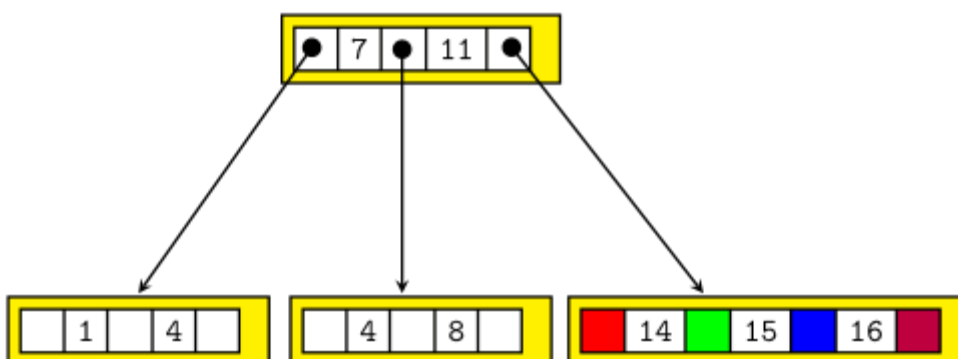
Beispiel zu Fall 1: Es ist noch Platz



B-Baum der Ordnung  $t = 2$

In den B-Baum aus Abb. 2 soll nun der Schlüssel 16 eingefügt werden. In einem B-Baum der Ordnung 2 hat jeder Knoten mindestens einen und höchstens 3 Schlüssel. Egal wo wir also landen würden, es würde noch in diesen Baum passen. Wir landen aber im Knoten rechts unten, da  $11 < 16$  ist.

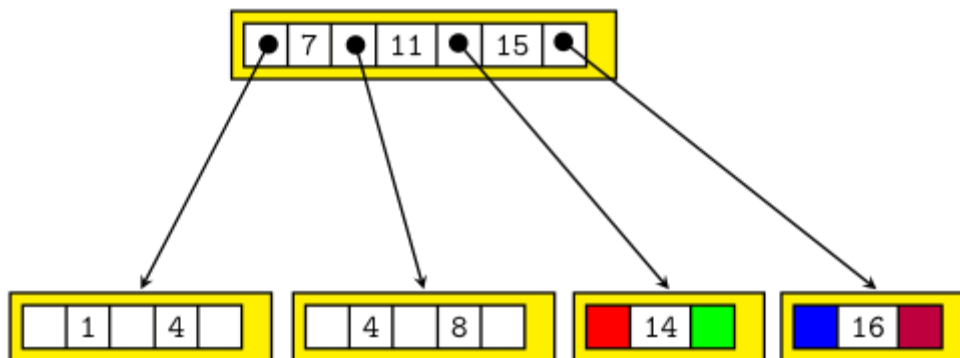
Das Ergebnis ist also:



B-Baum der Ordnung  $t = 2$

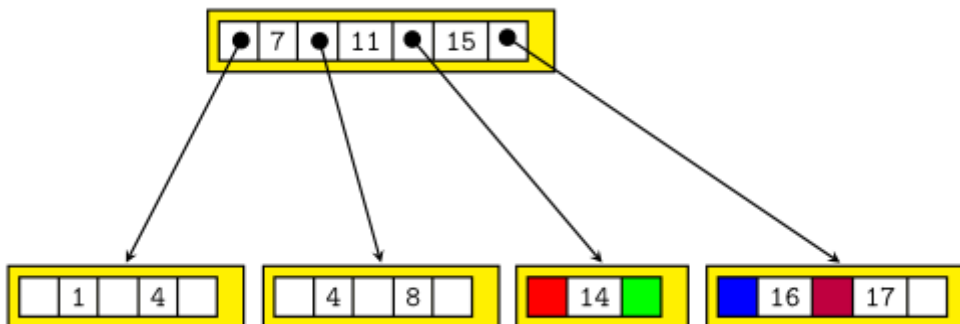
### Beispiel zu Fall 2: Knotenüberlauf

Will man nun in den B-Baum der Ordnung 2 aus Abb. 3 den Schlüssel 17 hinzufügen, so gibt es einen "Knotenüberlauf". Der Schlüssel müsste in den Knoten rechts unten. Damit hätte dieser 4 Schlüssel, er darf aber nur 3 haben. Also splitten wir zuerst den Knoten. Schlüssel 15 wandert zu dem Elternknoten hoch, die beiden einzelnen Schlüssel bilden eigene Knoten. Damit man sieht, was mit den Schlüsseln geschehen würde, wenn der Baum größer wäre, habe ich diese mal eingefärbt:



B-Baum der Ordnung  $t = 2$

Nun ist man beim Einfügen von 17 wieder in Fall 1. Das Ergebnis sieht so aus:



B-Baum der Ordnung  $t = 2$

### Löschen eines Schlüssels

Falls sich der Schlüssel in einem Blatt befindet, kann man ihn einfach löschen. Allerdings muss man darauf achten, dass mindestens  $t-1$  Schlüssel im Knoten verbleiben.

Ist der Schlüssel in einem inneren Knoten ist das ganze schwerer.

#### Fall 1: Schlüssel in Blatt

Der Einfachste Fall ist der 1. Fall des Einfügens, nur umgekehrt. Also aus dem B-Baum aus Abb. 3 die 16 entfernen. Dann entsteht der B-Baum aus Abb. 2.

## Fall 2 - 3

Für die anderen Fälle habe ich leider kein kleines Beispiel und will deshalb auf die Erklärung verzichten.

Falls ihr da Hilfe braucht: Cormen, dritte Ausgabe, S. 499ff war sehr hilfreich. (Link - Warum auch immer der eine PDF-Datei vom Buch hat. Die KIT-Bibliothek hat leider keine Online-Version.)

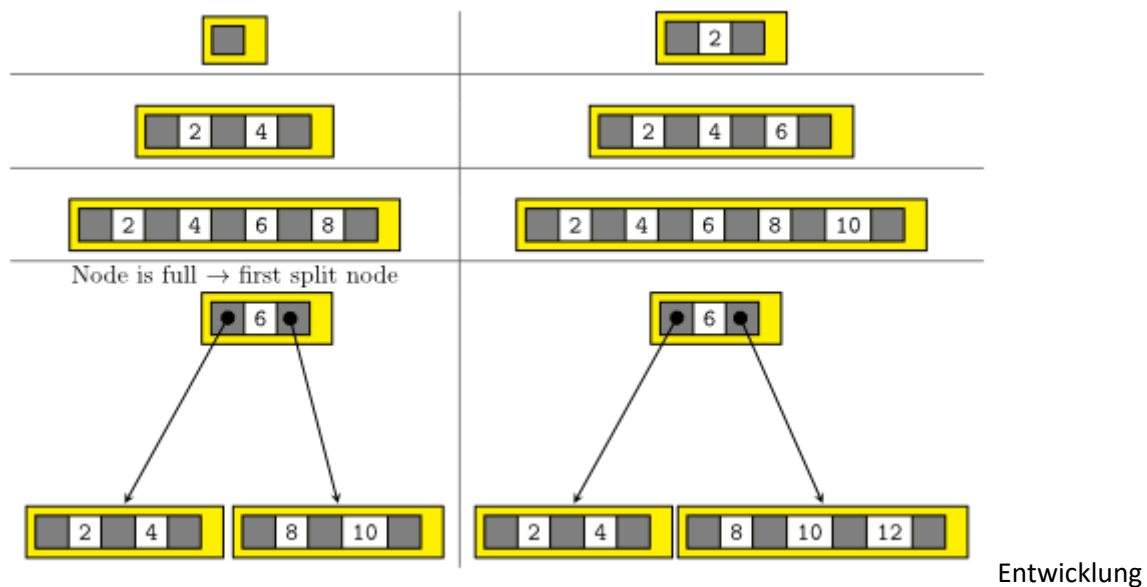
## Trivia

[...] standard B-trees had numerous characteristics that were at odds with the ext2 design philosophy of simplicity and robustness. For example, XFS's B-tree implementation was larger than all of ext2 or ext3's source files combined.

Source: [ext2.sourceforge.net](http://ext2.sourceforge.net)

## Beispiel

Fügt man die Schlüssel 2, 4, 6, 8, 10, 12 in einen anfangs leeren B-Baum ein, entwickelt sich dieser wie folgt:



eines B-Baumes der Ordnung  $t = 3$

## Hashing

## Allgemein

Eine Funktion kann eine Hash-Funktion genannt werden, wenn sie egal welche Daten von variierenden Größen einer Liste in eine Liste von Daten mit einer fixen Größe umwandeln kann. Die Werte die von einer Hash-Funktion zurückgegeben werden, werden „hash values“, „hash codes“,

„digests“ oder „hashes“ genannt, diese können identisch bei verschiedenen Eingabewerten sein, dieses Phänomen wird auch Collision genannt, welche in buckets gespeichert werden. Bucket ist einfach ein anderer Name für eine Liste, der sich bei Hash-Funktionen durchgesetzt hat.

Hashing wird öfters verwendet, um Daten schneller zu suchen.

## Verwendungen

### Hash Tables

Hash Tables werden benutzt um Dateneinträge schnell lokalisieren zu können. Sie sind ähnlich wie Dictionaries aufgebaut, wobei der key der Hash ist, währenddessen der Value eine Liste von möglichen dazugehörigen Speicherorte angibt.

Die Liste von möglichen dazugehörigen Speicherorten existiert, da eine Hash-Funktion normalerweise eine größere Reichweite an Eingabewerten verarbeiten kann, als er Hashes hat. Somit werden manche keys öfters verwendet, dies wird auch Collision genannt. Deshalb sind die Values einer Hash Table öfters auch bucket genannt, währenddessen die Hashes öfters bucket index oder bucket listing genannt werden.

### Caches

Hash-Funktionen können auch verwendet werden, um Caches aufzubauen. Wobei Collisions einfach überschrieben werden.

### Suchen von identischen Einträgen

Identische Einträge bekommen einer Hash-Funktion immer denselben Hash, was dazu führt das identische Einträge immer im gleichen bucket sind. Somit muss man nur alle Werte hashen und nur bei buckets, die mehr als einen Eintrag besitzen. Die Einträge vergleichen, um identische Einträge herauszufiltern.

### Datenschutz

Hash Values können verwendet werden, um geheime Informationen zu identifizieren. Dafür muss die Hash-Funktion collision-resistent sein.

### Suchen von ähnlichen Einträgen

Mit Hash-Funktionen, die die Eigenschaft von Continuity besitzen, können Einträge die sich ähneln herausgefiltert werden, indem man nach dem hashen der Daten, alle Daten rund um den bucket des Vergleich Eintrages vergleicht.

### Suchen von ähnlichen Teileinträgen

Funktioniert gleich wie das Suchen von ähnlichen Einträgen, nur dass nur der gesuchte Teil des Eintrages gehasht und verglichen wird.

## Eigenschaften

Gute Hash-Funktionen sollten normalerweise bestimmte Eigenschaften von der Liste besitzen. Man sollte aber aufpassen, nur Eigenschaften zu berücksichtigen, welche relevant für den gebrauchten Zweck sind. Da eine Hash-Funktion zum Indizieren von Daten höchstwahrscheinlich andere Eigenschaften besitzen muss, als eine Hash-Funktion für Verschlüsselungen.

### Determinism

Eine Hash-Funktion muss für denselben Eingabewert immer den gleichen Hash zurückgeben.

### Uniformity

Eine Hash-Funktion sollte die Hashes so gut wie möglich über ihre ganze output range verteilen.

### Defined Range

Eine Hashes einer Hash-Funktion sollen alle dieselbe Länge besitzen.

### Data Normalization

Die Eingabewerte der Hash-Funktion können auch irrelevante Werte besitzen für eine Vergleichsoperation. Zum Beispiel Groß- und Kleinschreibung bei Texten. Diese Eingabewerte sollten vor dem Verarbeiten normalisiert werden (z.B.: kleinschreiben aller Buchstaben).

### Continuity

2 Eingabewerte der Hash-Funktion, welche sich sehr ähneln, sollten zu ähnlichen Hashes umgewandelt werden. Diese Eigenschaft kann als eine Sicherheitslücke bezeichnet werden, da dadurch das Ausrechnen der Umkehrfunktion der Hash-Funktion erleichtert wird und sollte nur mitberücksichtigt werden, wenn man einen „nearest neighbor search“ durchführen will.

### Non-invertible

Die Umkehrfunktion der Hash-Funktion soll nicht leicht durch deren Hashes errechenbar sein. Natürlicherweise wird es die Umkehrfunktion immer geben, nur sollte man ohne dessen Wissen eine lange Zeit brauchen, um diese aus den Hashes auszurechnen.

### Collision-resistant

Eine Hash-Funktion ist collision-resistant, wenn sie eine sehr geringe Chance hat, aus zwei unterschiedlichen Eingabewerten, denselben Hash erzeugt. Dies wird erleichtert, indem sehr große Hashes generiert werden.