Inhalt

B-Tree

Allgemein

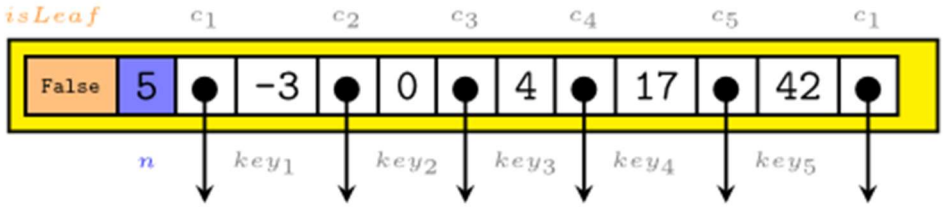
Ein B-Baum (englisch B-tree) ist in der Informatik eine Daten- oder Indexstruktur, die häufig in Datenbanken und Dateisystemen eingesetzt wird. Ein B-Baum ist ein immer vollständig balancierter Baum, der Daten nach Schlüsseln sortiert speichert. Er kann binär sein, ist aber im Allgemeinen kein Binärbaum. 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∈N\ {1}t∈N\ {1}, gilt:

1. Jeder Knoten xx hat die folgenden Attribute:
    1. nn, die Anzahl der Schlüssel die im Knoten xx gespeichert wird,
    2. die nn Schlüssel, die in aufsteigender Reihenfolge gespeichert werden (also key1≤key2≤...≤keynkey1≤key2≤...≤keyn),
    3. isLeafisLeaf, ein boolscher Wert der True ist, falls xx ein Blatt ist und False ist, falls xx ein innerer Knoten ist
2. Jeder innere Knoten hat n+1n+1 Zeiger c1, c2, ...cn+1c1, c2, ...cn+1 auf seine Kinder. Blattknoten haben keine Kinder, also sind ihre cici-Attribute undefiniert.

Ein Knoten eines B-Baumes sieht also so aus:



Die Schlüssel keyikeyi setzen grenzen für die Werte der Schlüssel, die in den einzelnen Subbäumen gespeichert sind.
Falls kiki ein Schlüssel im Subbaum mit der Wurzel cici ist, dann gilt: k1≤key1≤k2≤key2≤...≤keyn≤kn≤keyn+1k1≤key1≤k2≤key2≤...≤keyn≤kn≤keyn+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−12t−1 Schlüssel

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: # Schl&uuml;ssel ist gesuchter Schl&uuml;ssel
        return node.key[i].satelittendaten
    else if node.isLeaf: # Erfolglose Suche
        return NIL
    else: # Rekursiv weitersuchen
        DISK-READ(node.c[i])
        return SEARCH-KEY(node.c[i], key)
```

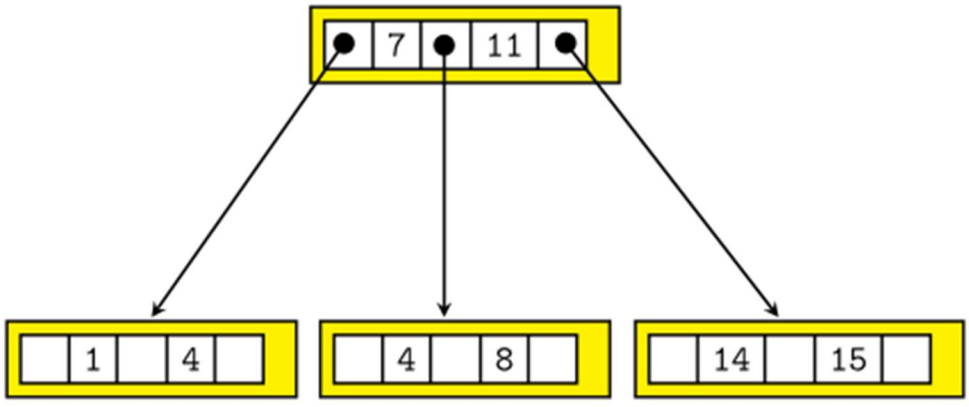
Es wird also zuerst der Knoten durchsucht und dann gegebenenfalls der passende Subbaum.

Einfügen von Schlüsseln

Jeder Knoten enthält n Schlüssel, mit t−1≤n≤2t−1t−1≤n≤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.

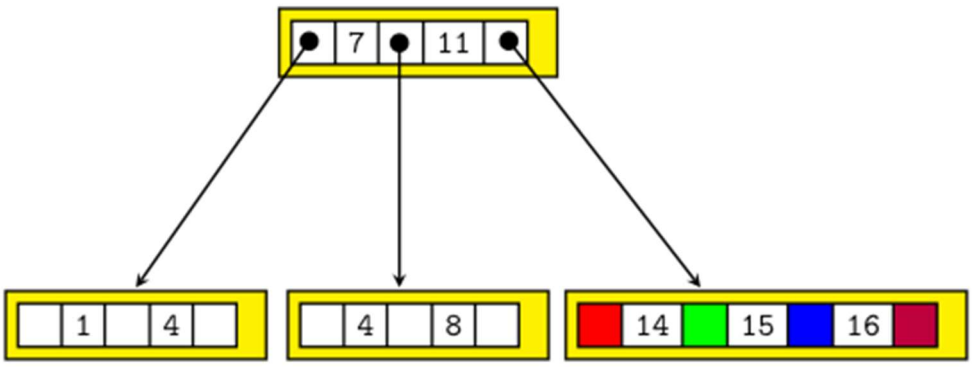
Beispiel zu Fall 1: Es ist noch Platz ¶



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<1611<16 ist.

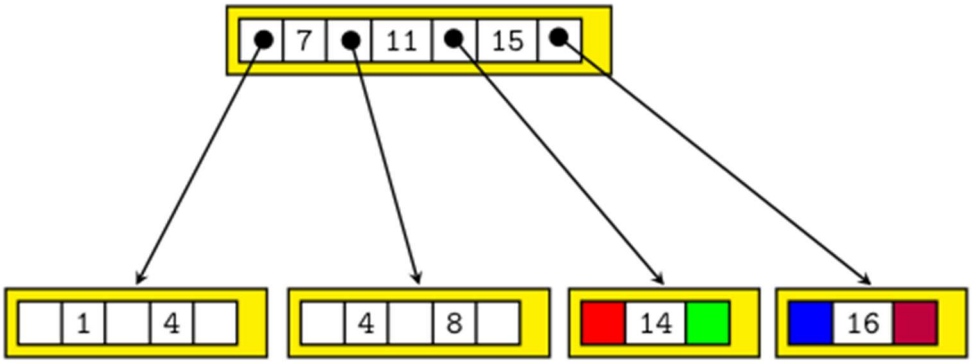
Das Ergebnis ist also:



B-Baum der Ordnung t = 2

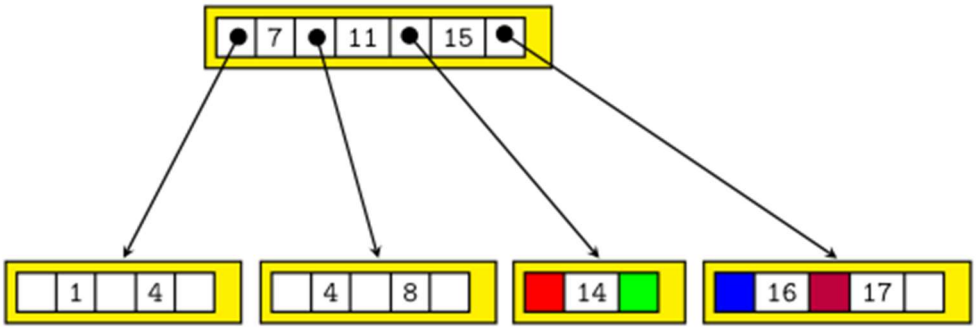
Beispiel zu Fall 2: Knotenüberlauf

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.

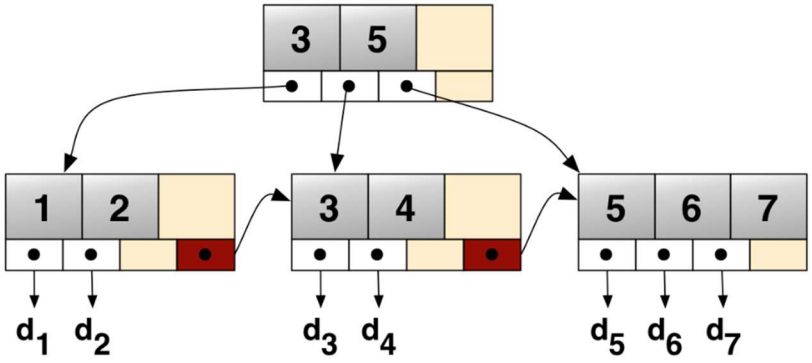
B+-Baum

Sie ist eine Erweiterung des B-Baumes. Bei einem B+-Baum werden die eigentlichen Datenelemente nur in den Blattknoten gespeichert, während die inneren Knoten lediglich Schlüssel enthalten. Die Schlüssel in den Verzeichnisseiten bezeichnet man auch als Separatoren.

Ziel dieses Verfahrens ist es, die Zugriffszeiten auf die Datenelemente zu verbessern. Dazu muss man die Baumhöhe verringern, was bedeutet, dass der Verzweigungsgrad des Baumes wachsen muss. Da die maximale Speicherbelegung eines Knotens begrenzt ist, gewinnt man durch das Verlegen der Daten in die Blätter mehr Platz für Schlüssel bzw. Verzweigungen in den inneren Knoten. Dies gilt insbesondere bei der Speicherung komplexer Objekte, die deutlich mehr Speicher belegen als die Schlüssel oder auch nicht über eine feste Größe verfügen. Die reduzierte Baumhöhe impliziert auch weniger innere Knoten. Diese können so leichter im Hauptspeicher gehalten werden, was die Leistung im wahlfreien Zugriff steigert.

Ein weiteres Ziel kann sein, die Operation Bereichssuche zu verbessern, bei der alle Daten in einem gewissen Schlüsselintervall sequentiell durchlaufen werden. Werden die Daten nämlich ausschließlich in den Blättern abgelegt, muss der jeweils nächste Datensatz der Sequenz nicht wieder von der Wurzel aus gesucht werden. So muss für einen Komplettdurchlauf der Daten nur der erste Schlüssel gesucht werden, ein Großteil des Baumes wird nicht gelesen. Um Nachfolger und Vorgänger eines Blattknoten effizient (d. h. in konstanter Zeit) zu finden, müssen die Blätter in einer doppelt verketteten Liste miteinander verbunden sein. Dieses Feature wird häufig in die Definition des B+-Baumes mit aufgenommen.

Wesentlicher Vorteil eines externen Suchbaums (Daten nur in den Blättern) ist die Möglichkeit des Einsatzes von Sekundärindizes. Sie stellen einen weiteren – nach anderen Kriterien sortierbaren – Suchbaum auf denselben Daten zur Verfügung.

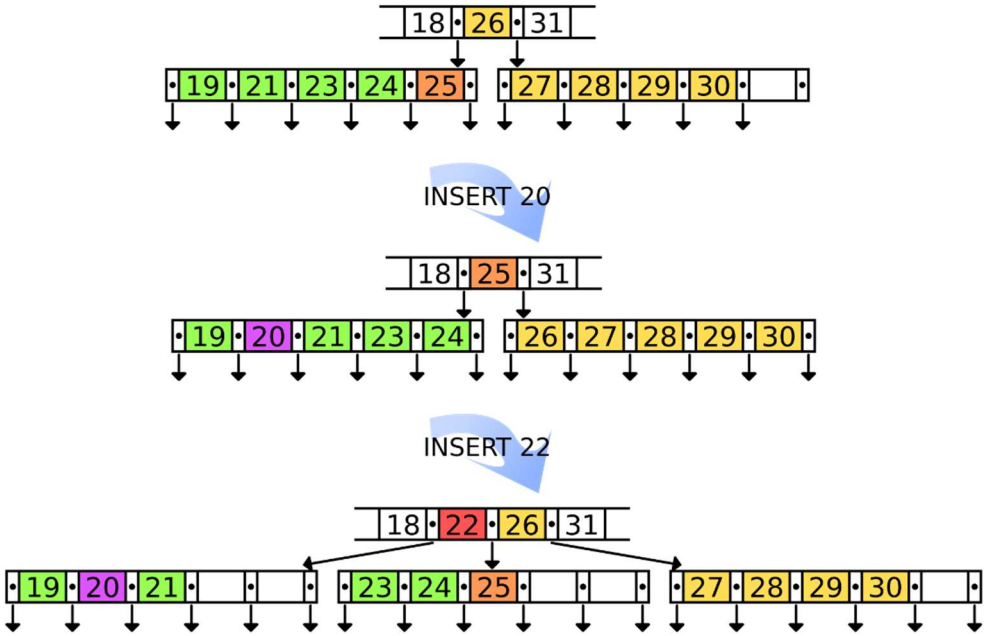


B*-Baum

Der Unterschied zum B-Baum besteht darin, dass Knoten mindestens zu 2/3 gefüllt sein müssen (anstatt nur 1/2 gefüllt).

Definition

1. alle Knoten außer die Wurzel haben maximal Kinder,
2. alle Knoten außer Wurzel und Blätter haben mindestens Kinder,
3. die Wurzel hat mindestens und maximal Kinder,
4. alle Blätter haben die gleiche Entfernung von der Wurzel und
5. alle Knoten, die keine Blätter sind, mit Kindern enthalten Schlüssel.



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, diese 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-resistent

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