Inhaltsverzeichnis, Name, Beispiele für Hash-Funktionen (modulo als einfachste Fu.)

B-Tree (Balanced Search Tree)

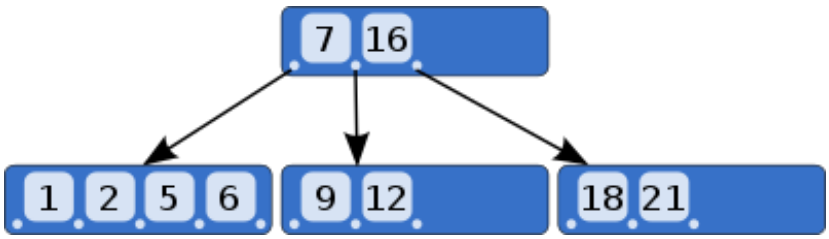
Laufzeitverhalten analysieren O(1), O(log N)

Kleiner Überblick

Der B-Tree wurde von Rudolf Bayer und Ed McCreight erfunden, während sie bei Boing Research Labs in 1972 gearbeitet haben. Es wird spekuliert, dass das B vom B-Tree für „balanced“, „broad“, „bushy“, „Boing“ oder „Bayer“ steht, aber die Entwickler haben nie erklärt wofür das B im B-Tree steht.

Der B-Tree ist ein selbst regulierende Baum-Datenstruktur, der seine Daten ordnet und das Suchen, Hinzufügen, Löschen und Anschauen dessen Daten in logarithmischen Zeiten[1] zulässt. Der B-Tree ist nicht zu verwechseln mit dem Binary-Tree, da er eine Generalisierung dessen ist, da der B-Tree mehr als nur zwei Kinder besitzen kann. Der B-Tree ist für Systemen geeignet, die mit großen Datenblöcken arbeiten. Sie sind ein gutes Beispiel für Datenstrukturen von externen Speichermedien und wird oft von Filesystemen und Datenbanken verwendet.

Aufbau



Der B-Tree besteht aus einem Root, (non-leave) nodes und leaves.

Der Root ist im Grunde genommen, nur ein node der kein Elternteil besitzt und deswegen das Zentrum des ganzen B-Tree‘s ist.

Ein node besitzt eine variierende Anzahl von (non-leave) nodes die innerhalb einer davor eingestellten Reichweite (pre-defined range) liegt. Damit der B-Tree sortiert bleibt, muss man deswegen in jedem node keys hinterlegen. Die keys sind dafür da, damit Daten am richtigen Platz gespeichert werden, damit sie dann später auch wiedergefunden werden können. Die Anzahl der keys ist optional. Die Anzahl der keys muss zwischen der pre-defined range – 1 bei der maximal, als auch bei der minimalen Anzahl liegen.

Ein leave hat ein node als Elternteil und besitzt keine Kinder, sondern nur die Daten die abgespeichert werden sollen. In einem Node zählen sie als solches.

Beispiel: Ein Node hat einer pre-defined range von 2-4, somit muss der Node 1-3 keys (k_1, k_2, k_3) besitzen, damit die Daten auf die 4 möglichen Nodes (n_1, n_2, n_3, n_4) richtig aufgeteilt werden können.

- Wenn der gefragte Wert kleiner als k_1 ist, kommt er zu n_1.
- Wenn der gefragte Wert größer als k_1 ist und kleiner als k_2, kommt er zu n_2.
- Wenn der gefragte Wert größer als k_2 ist und kleiner als k_3, kommt er zu n_3.
- Wenn der gefragte Wert größer als k_3 ist, kommt er zu n_4.

Keys innerhalb eines nodes können Splits und Joints ausführen. Dabei wird entweder 2 Keys zusammengeführt zu einem neuen key, oder ein key in zwei geteilt.

[1] Zeiten die man errechnen kann durch Logarithmen

Varianten

B-Tree

Basis, wobei die Keys in den Nodes gespeichert wird, aber diese nicht in den Eigenschaften von den leaves gespeichert werden.

B+-Tree

Erweiterung von B-Tree, die Keys werden in den Eigenschaften von den leaves gespeichert und es kann sogar noch ein Pointer zum nächsten leave gespeichert werden, um die sequentielle Suche schneller zu machen.

B*-Tree

Bei dem B*-Tree werden die nodes stärker gefüllt, damit der Tree kompakter ist.

Datenbankverwendungen

Bei größeren Datenbanken werden die Daten auf mehreren Speichermedien gespeichert (Disk Drives), welches einem B-Tree stark ähnelt. Deswegen weist man Datenblöcke Indexe hinzu, damit man dann in einem B-Tree die Indexe (Speicherorte) einspeichert und somit alle Operationen über den B-Tree auszuführen. Somit muss man nicht mehr in alle Speichermedien nach dem gewünschten Datenblock suchen, sondern muss man nur noch den Index im B-Tree suchen, der dann den Speicherort des Datenblockes zurückgibt.

Wenn bei der Datenbank Insert und Delete ausgeführt werden, sollte man nötigenfalls den Index anpassen, um unnötige Zeitverluste zu verhindern.

Vorteile von B-Tree

- Keys sind sortiert
- Ist hierarchisch, was den Zeitaufwand reduziert
- Kann Datenblöcke verwenden, um Insert und Delete schneller zu machen
- Haltet den Tree balanced, damit der Zeitaufwand für alles ungefähr gleich ist.

Algorithmen

Search

Der Suchalgorithmus ist sehr ähnlich, wie der vom Binary Search Tree. Es wird vom Root ausgegangen und den Baum Level zu Level hinuntergegangen, bis man am Ziel angekommen ist. Der einzige Unterschied im Gegensatz zum Binary Search Tree ist, dass man bei jedem Level mehrere Wege hat, also somit mehrere Vergleichswerte die man berücksichtigen muss.

Insert

Der Hinzufügungsalgorithmus ist ein bisschen komplizierter. Als erstes sucht man den leaf node, wo das neue Element hinzugefügt werden soll. Nach diesem Schritt wird je nach Bedingungen anders vorgegangen.

- Falls der Node weniger als die maximale Anzahl von Elementen enthält, wird das neue Element einfach bei diesem Node hinzugefügt.
- Sonst wenn der Node voll ist, wird der Node gleichmäßig in 2 Nodes gespaltet:
    - Zu aller Erst wird ein Element des Nodes ausgewählt oder das neue Element und als Medium platziert
    - Alle Elemente die Wertemäßig kleiner als das Medium sind, werden in das neue linke Node eingefügt, währenddessen alle Elemente die Wertemäßig größer als das Medium sind, werden in das neue rechte Node eingefügt.
    - Das Medium selber wird in das Parent Node hinzugefügt, falls aber dort auch schon alle Plätze gefüllt sind, wird dieses kleine Separation-Spiel im Parent Node erneut ausgeführt.
        - Falls dieses Separation-Spiel bis zum Root hinaufgeht, wird ein neuer Root erstellt, der 2 Child Nodes besitzt.

Ein verbesserter Algorithmus muss nur einmal den Baum hinuntergehen um ein Element hinzuzufügen. Dies wird geschafft, indem auf dem Weg zur richtigen Hinzufügungsstelle alle vollen Nodes schon aufgetrennt werden in 2 Nodes (Achtung: dabei darf man nicht das neue Element als Kandidat für das Medium verwenden). Somit kann man sich Zeit ersparen, den Baum wieder hinaufzuklettern, da zum Beispiel die Parent Nodes voll gewesen waren.

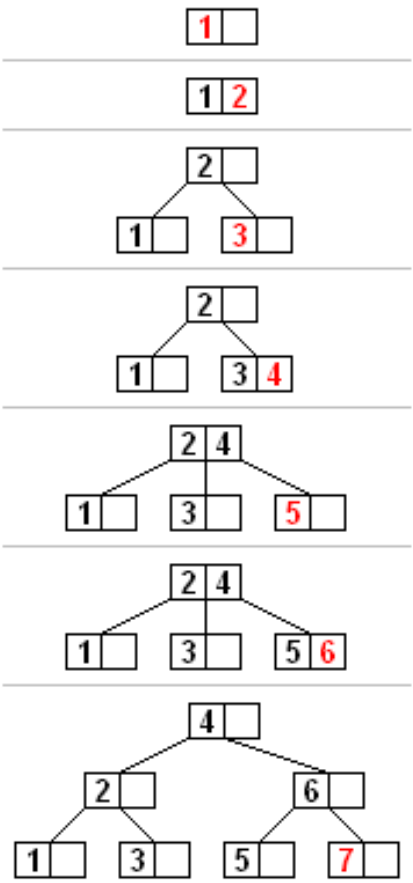


Abbildung 1: https://en.wikipedia.org/wiki/B-tree:

Delete

Wie beim Hinzufügen von Elementen gibt es auch beim Löschen von Elementen 2 Strategien. Zu aller Erst wird das Element gesucht, welches gelöscht werden sollte. Danach löscht man das Element und überprüft man den Node, ob er unterhalb des Minimums von Element gestoßen ist. Falls ja, muss er neu balanciert werden.

- Falls es möglich ist von einem Sibling Node (nur der direkt links oder rechts davon liegende) ein Element zu nehmen, da dieser Sibling Node nicht am Minimum ist, wird zu aller erst der Separator, der den aktuellen Node und den Sibling Node so versetzt, dass entweder das linkeste oder das rechteste Element rechtsmäßig ins aktuelle Node gehört. Danach wird das linkeste oder das rechteste Element vom Sibling Node zum aktuellen Node umgelagert und somit ist der Baum wieder balanciert.
- Falls es möglich ist von einem Sibling Node (nur der direkt links oder rechts davon liegende) ein Element zu nehmen, da diese Sibling Nodes am Minimum sind, wird der Separator der den linken Sibling Node und den aktuellen Node so verschoben, dass alle Elemente des aktuellen Nodes rechtsmäßig in das linke Sibling Node gehören. Anschließend lagere alle Elemente des aktuellen Nodes in das linke Sibling Node um und lösche den verschobenen Separator und den jetzigen leeren Node (der aktuelle Node). Achtung, durch diesen Merge werden nicht nur der Sibling Node und der aktuelle Node beeinflusst, sondern auch der Parent Node, indem er ein Element verliert.
    - Somit falls der Parent unter der minimalen Anzahl an Elementen ist, muss man diesen auch wieder zusammenführen mit einer seiner Sibling Nodes.
    - Oder falls der Parent der Root war und jetzt nur noch ein Child Node hätte, mach dieses Child Node den neuen Root.

Von diesem Algorithmus gibt es wieder eine verbesserte Variante, die der verbesserten Variante des Inserts stark ähnelt. Nur dass beim vorzeitigen neu balancieren alle Nodes deren Anzahl von Elementen der minimalen Anzahl entsprechen vorzeitig mit einem Sibling Node zusammengeführt werden.

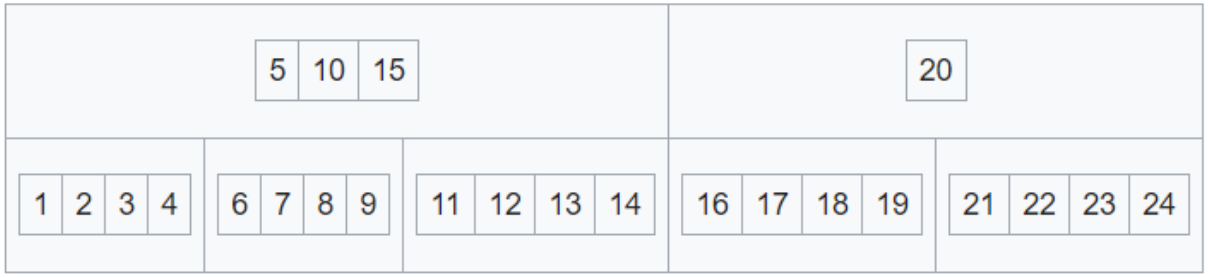
Initial Construction

Es gibt 2 Varianten, wie man einen B-Tree initialisieren kann. Zu aller Erst könnte man ein Element nach dem Anderen hinzufügen, welches eine recht stabile Variante wäre, nur sehr lange dauern könnte.

Die 2.Variante nennt sich bulkloading.

- Dabei werden zu aller erst alle vorhandenen Elemente sortiert.
- Danach werden sie in gleich große Teile aufgeteilt. Diese Teile besitzen eine Anzahl von 1 Element mehr als die eingestellte maximale Anzahl von Elementen.
    - Falls es einen Rest nach der Aufteilung gibt, wird der Rest in einen eigenen Teil hinzugefügt.
    - Falls es keinen Rest nach der Aufteilung gibt, wird das letzte Teil der Aufteilung in ungefähr 2 gleich großen Teilen aufgeteilt.

- Als nächstes nimmt man von jedem Teil das letzte heraus und schiebt es in ein Parent Node. Dieses Parent Node vollgefüllt, bis es eine Anzahl besitzt von 1 Element mehr als die eingestellte maximale Anzahl von Elementen. Der Rest wird gleichbehandelt wie bei dem Schritt zuvor.



- Der vorhergegangene Schritt wird jetzt solange wiederholt, bis man den Root gefunden hat.

Hashing

Kleiner Überblick

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.

Quellen

https://en.wikipedia.org/wiki/Hash_function
https://en.wikipedia.org/wiki/B-tree