Informatik Seminar 2008 Iwanowski

Algorithmen – Thema 8:

Zeit vs. Platz III

Sebastian Barzyk

1.) B-Baum Geschichte

Der B-Baum wurde 1972 von Rudolf Bayer und Edward M. McCreight entwickelt.

Er erwies sich als ideale Datenstruktur zur Verwaltung von Indizes für das relationale Datenmodell, das im gleichen Jahr entwickelt wurde.

Diese Kombination führte zur Entwicklung des ersten SQL-Datenbanksystems „System R“ bei IBM.

Die Bedeutung des „B“ ist bis heute unklar – spekuliert wird unter anderem über Balanced und Bayer.

2.) Motivation

Wenn Datenstrukturen zur Programmlaufzeit vollständig oder Teilweise in einem Hintergrundspeicher liegen welcher einen vielfach langsameren Zugriff als der Hauptspeicher bietet, (z.B. Festplatte) ist es von entscheidender Bedeutung für die Effizienz des Systems das die Zugriffe auf den langsamen Hintergrundspeicher Effizient gestaltet sind.

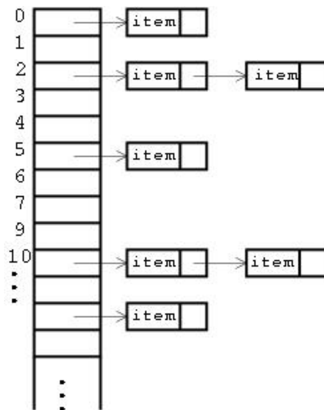
Beispiel:
Die Datensätze einer Datenbank liegen in der Regel fast vollständig auf einer Festplatte.

Die gängigste Anfrage an eine Datenbank ist das Suchen eines bestimmten Datensatzes.
Die Geschwindigkeit der Suche wird davon dominiert wie oft der Schreib-Lesekopf der Festplatte ausgerichtet werden muss um den gesuchten Datensatz zu finden.
Andere Geschwindigkeitsfaktoren sind relativ dazu in der Regel nicht signifikant.

Gesucht:
Eine Datenstruktur die die Physikalischen Datensatz-Adressen so verwaltet, dass bei der Suche nach ihnen der Schreib-Lesekopf möglichst wenig bewegt werden muss.

3.) Mögliche Datenstrukturen

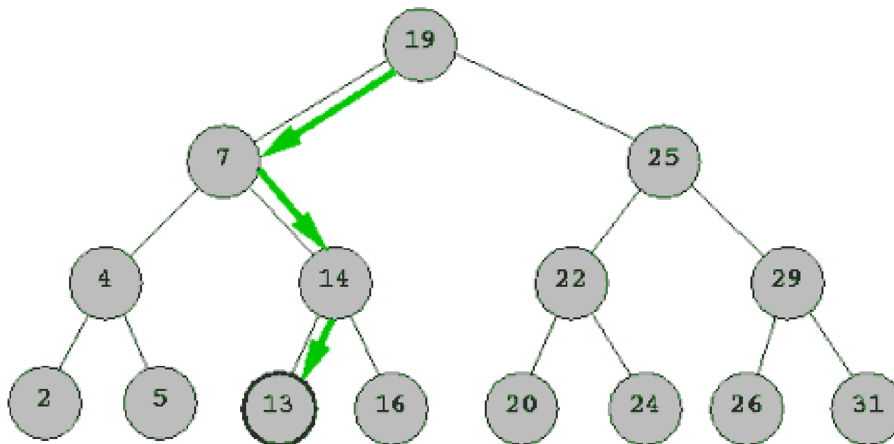
3.1) Hash-Tabelle



Datenobjekte werden in einer Tabelle/einem Array gespeichert dessen Index durch eine Funktion bestimmt wird.
Wenn die Funktion für verschiedene Datensätze den gleichen Index Berechnet, muss diese „Kollision" aufgelöst werden – z.B. dadurch dass in dem entsprechenden Index eine Neue Hashtabelle angelegt wird (Doppeltes Hashen).

Vorteil: Im Idealfall kann der gesuchte Datensatz sofort gefunden werden => O(1)
Nachteil: Treten Kollisionen vermehrt auf spricht man von „Entartungen", diese sind bei großen Datenbeständen kaum vermeidbar und können die Effizienz dramatisch verringern => Worst-Case: O(n)

3.2) Binärer Suchbaum

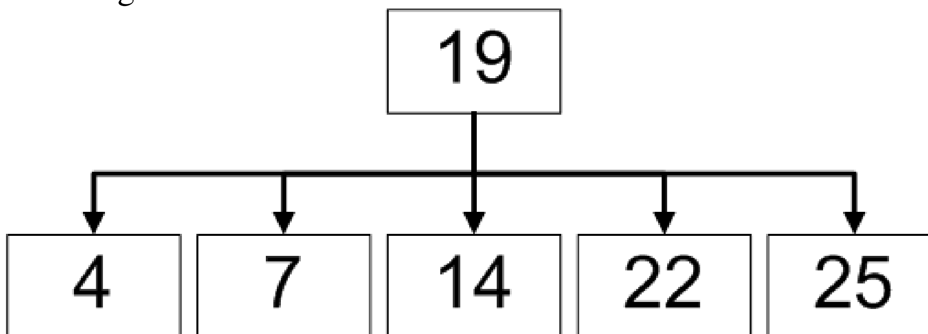


Der Binäre Suchbaum hat eine steile Hierarchie.
Bei der Suche nach einem Datenelement muss Der Schreib-Lesekopf der Festplatte jedes Mal neu ausgerichtet werden, wenn im Baum in einer Tieferen Ebene weitergesucht werden muss.
Komplexität: O(log n)

3.3) N-Baum

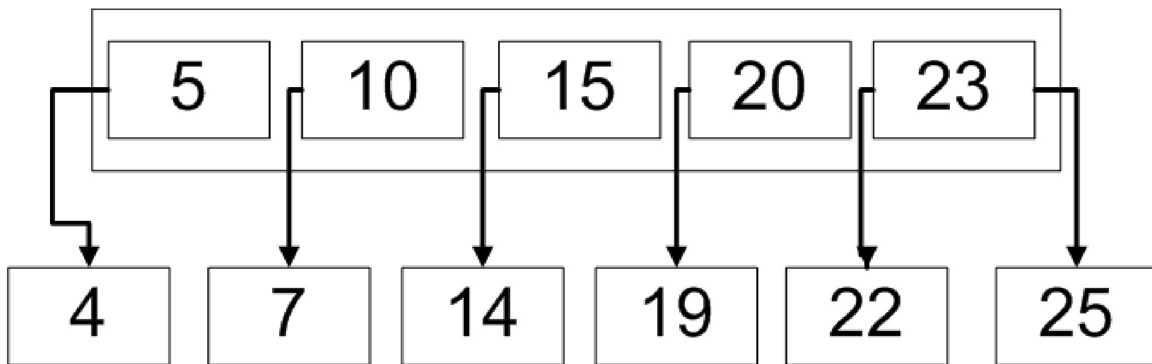
Für unsere Datenstruktur wäre also eine möglichst flache Hierarchie wünschenswert.

Dies kann erreicht werden, indem es erlaubt wird, dass ein Knoten mehr als 2 Nachfolger hat:



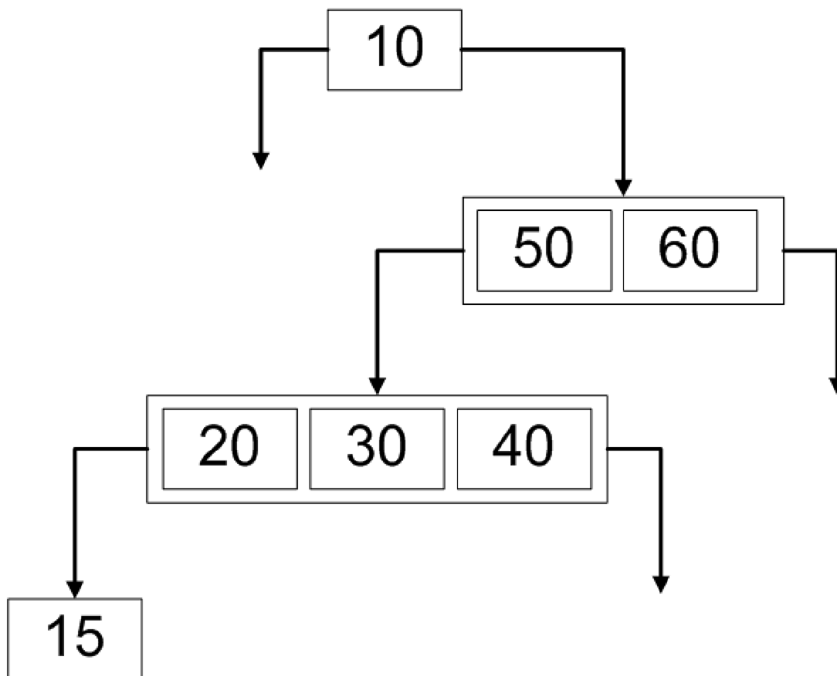
Leider eignet sich dieser N-Baum nicht als Suchbaum da mit dem vorhandenen Schlüssel pro Knoten nur zwischen zwei alternativen Sub-Bäumen gewählt werden kann.

3.4.1) B-Baum: Schlüssel



Um weiterhin eindeutig den Sub-Baum identifizieren zu können, in dem weitergesucht werden soll, sind weitere Schlüssel pro Knoten notwendig.
Für einen Knoten wird immer ein Schlüssel weniger gebraucht als er Kinder hat.

3.4.2) B-Baum: Balance



Wenn der Baum nicht balanciert wird kann er ungünstige Formen annehmen in denen unser Hauptziel – möglichst geringe Suchtiefe – mit Füßen getreten wird.

Das balancieren des Baumes verursacht selbstverständlich auch wahlfreie Zugriffe auf der Festplatte, ist aber gerade bei großen Bäumen hoher Ordnung relativ selten nötig und damit das kleinere Übel.

Ab wann soll balanciert werden?
Für B-Bäume ist die Invariante definiert das jeder Nicht-Wurzel-Knoten mindestens halb so viele Kinder haben muss wie die maximal mögliche Anzahl an Kindern.
So genannte B*-Bäume unterscheiden sich hier mit einer strengeren Invariante, bei Ihnen müssen die Nicht-Wurzel-Knoten nicht nur zur hälfte, sondern zu 2/3 gefüllt sein.

3.4.3) B-Baum: Ordnung

B-Bäume werden nach Ihrer Ordnung „t“ klassifiziert welche die maximal erlaubte Anzahl an Kindknoten angibt.
Die maximal erlaubte Anzahl an Schlüsseln ergibt sich entsprechend zu t - 1.
Teilweise wird unter t auch die minimal erlaubte Anzahl an Kindknoten verstanden – daraus ergibt sich dann die maximale Anzahl an schlüsseln 2t – 1.

Desto größer die Ordnung t des Baumes ist, desto flacher wird die Hierarchie sein – und desto weniger wahlfreie Festplatten-Zugriffe sind nötig um einen Blattknoten zu finden.
Eine hohe Ordnung führt dazu dass weniger oft Balanciert werden muss => weniger wahlfreie Zugriffe auf die Festplatte.

Eine möglichst hohe Ordnung ist also erstrebenswert!

Da die Größe eines Knotens von der Ordnung t abhängig ist bietet es sich an t so zu wählen das ein Baum-Knoten gerade immer in einen Block auf der Festplatte passt.

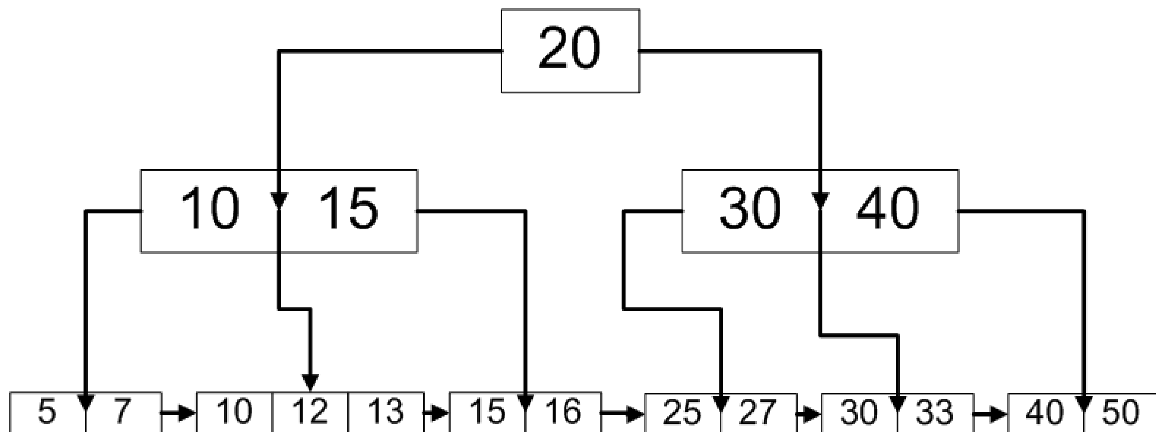
Eine weitere Anhebung der Ordnung ist nicht sinnvoll, da sonst der Schreib-Lesekopf beim lesen einzelner Knoten bewegt werden müsste.

Beispiel:

Ein Festplattenblock sei 1024 Byte groß.
Ein Schlüsselwert belege 1 Byte.
⇨ Es wird ein B+ Baum der Ordnung 1024 gewählt.
(Mit den Restrektionen:
Jeder innere Knoten hat höchstens 1024 Kinder.
Jeder innere Knoten hat mindestens 512 Kinder.)

Damit der Platz in den inneren Knoten voll für die Schlüssel verwendet werden kann (Um eine möglichst hohe Ordnung realisieren zu können) sollten hierbei Daten-Objekte lediglich an den Blattknoten erlaubt sein.
B-Bäume, die dieser Restrektion genügen werden als B+ Bäume bezeichnet und sind anscheinend besonders Für die Verwaltung von Datenbanken geeignet.

Ein B+ Baum:



Die Blätter der B+Bäume werden mitunter miteinander verkettet. Dadurch lassen sich Intervall-Suchen sehr effizient gestalten.

3.4.3) B-Baum: Definitionen

B-Baum mit der Ordnung t:

1.) Jeder Knoten hat <= t Kinder.
2.) Alle Nicht-Wurzel-Knoten haben >= t/2 Kinder.
3.) Alle Blätter liegen auf derselben Ebene.
4.) Anzahl der Schlüssel eines Knotens ist Anzahl der Kinder des Knotens – 1.
5.) Alle Schlüssel-Werte im linken Teilbaum sind kleiner, alle Schlüssel-Werte im rechten Teilbaum sind größer oder gleich.

B+ Baum:
B-Baum der Lediglich an den Blättern Nutz-Informationen trägt.
Die Blattknoten können miteinander verkettet sein.

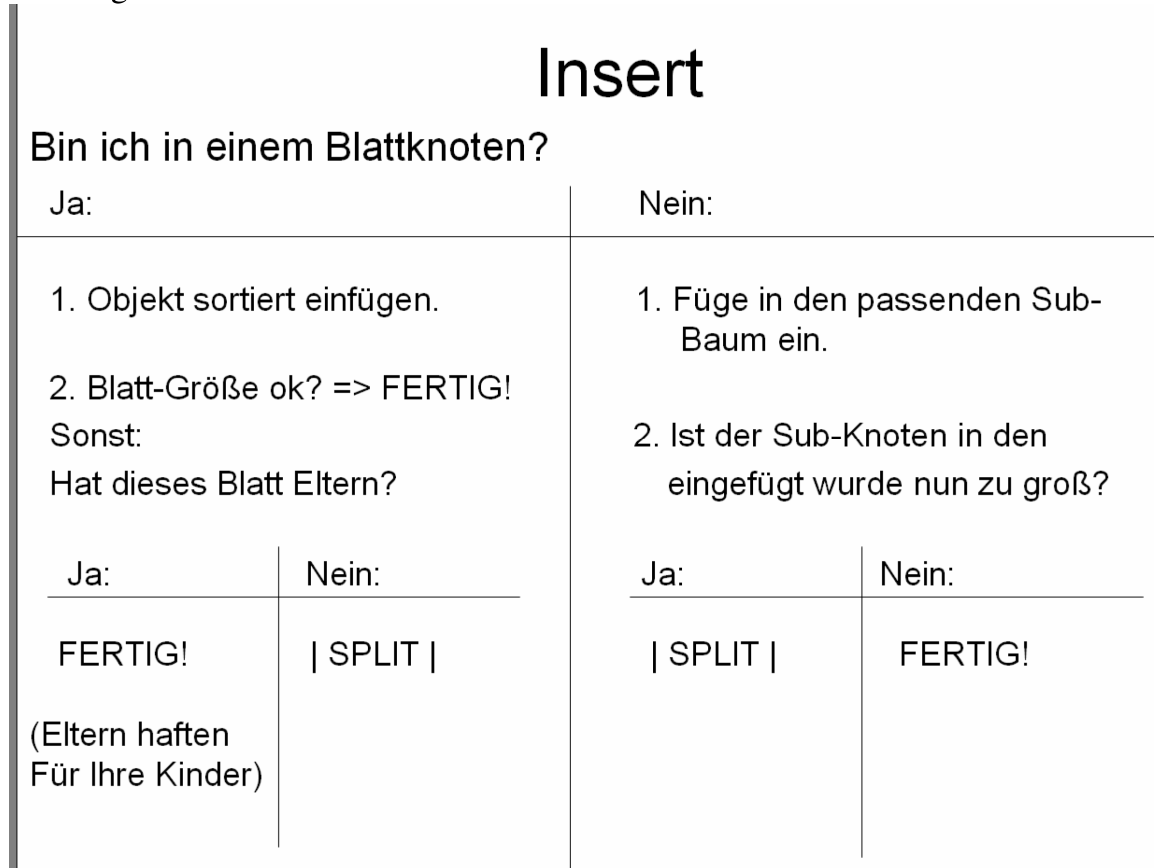
B* Baum:
B+ Baum dessen Nicht-Wurzel-Knoten mindestens zu 2/3 gefüllt sein müssen.

3.4.4) B-Baum: Komplexität

Die Komplexität für das finden eines Blatt-Knotens ist proportional zur Baum-Tiefe.
Für jeden Ebenen-Sprung im Baum muss 1 Mal der Schreib-Lesekopf bewegt werden.
In der O-Notation hat ein B-Baum allerdings dieselbe Komplexität wie ein binärer Suchbaum: O(log n).

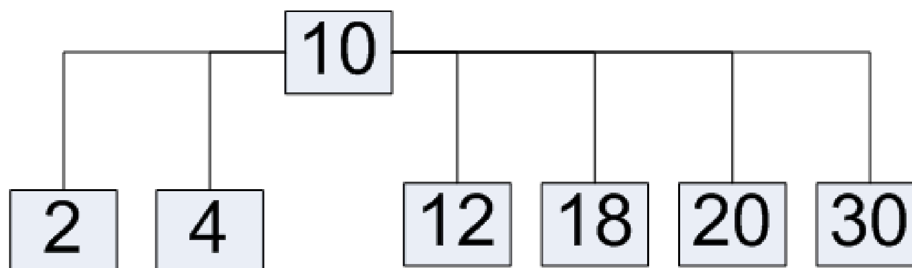
4.0) Insert

Struktogramm:



Nachdem in einen B-Baum ein Objekt eingefügt wurde, kann es passieren dass einer der Knoten zu viele Elemente enthält.

Beispiel: B-Baum der Ordnung 5:

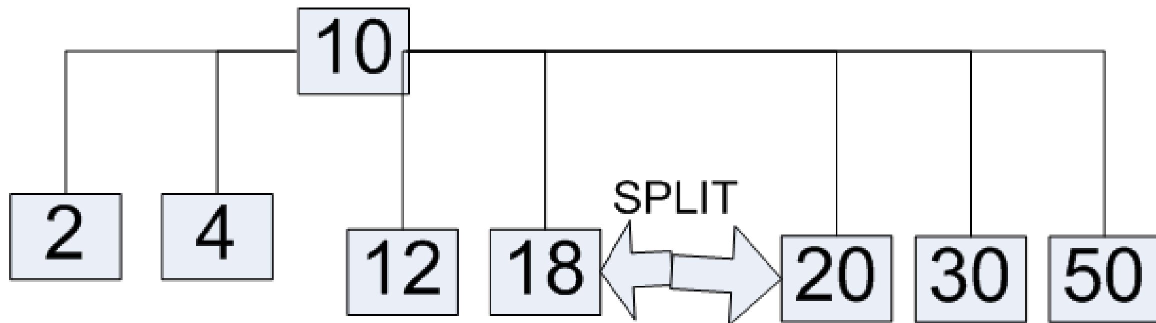


In jedem Knoten dürfen höchstens 4 Elemente enthalten sein.
Wenn eine Zahl > 10 – z.B. 50 eingefügt wird, hat der rechte Blattknoten zu viele Blätter.

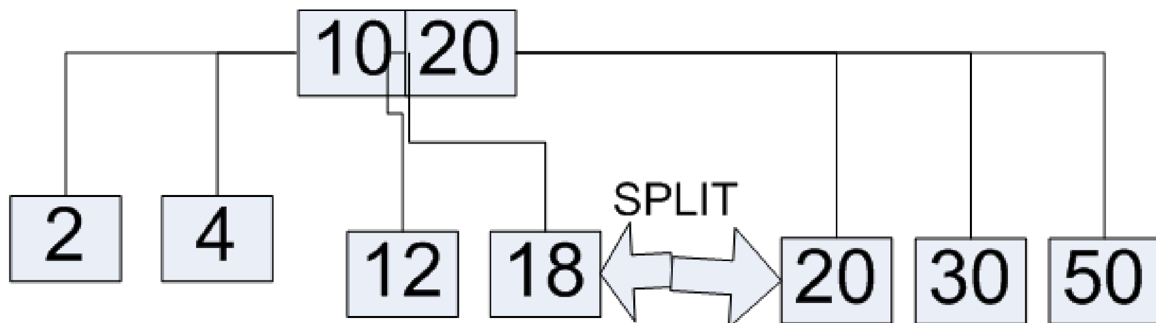
Um dies zu erkennen und den Fehler dann zu beseitigen achtet jeder Knoten darauf ob seine Kindknoten nach der Einfüge-Operation gegen die Regeln verstoßen: (Struktogramm rechte Seite – Kein Blattknoten)

1. Objekt in Subbaum einfügen
2. Subbaum fragen ob er zu groß ist

Meldet der Subbaum dass er zu groß ist, wird er gesplittet, so dass 2 neue Kindknoten entstehen: (Die 50 wurde eingefügt)



Da es jetzt 3 Sub-Knoten gibt wird ein weiterer Schlüssel/Seperator benötigt
Der Schlüssel des ersten Elements des rechten Teilbaums liefert den neuen Schlüssel: (20)



5.0) Delete

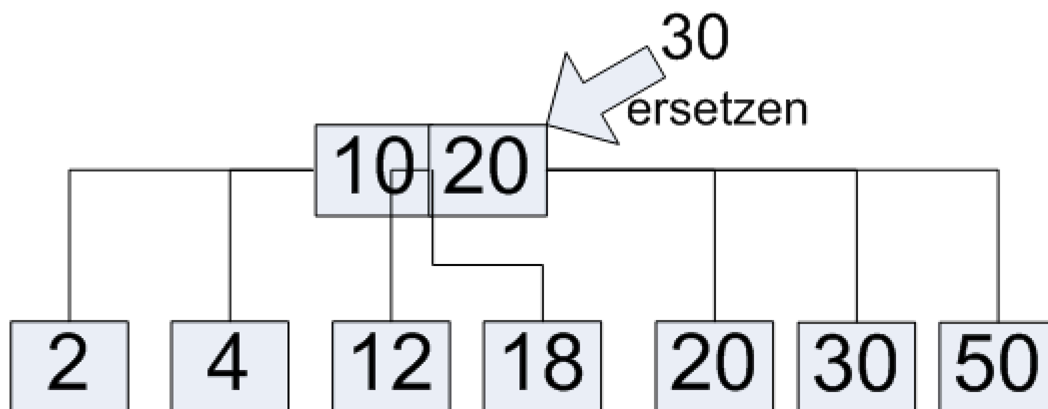
In Blattknoten: Element aus Element-Liste löschen => FERTIG.

In Inneren Knoten:

Schritt A.) Wenn Lösch-Key in Key-Liste: Ersetze ihn mit dem nächstem Key in seinem Blatt.

Wird der Schlüssel des zu löschenden Elementes in inneren Knoten verwendet – müssen diese vorkommen ebenfalls verschwinden – sie können aber nicht ersatzlos gelöscht werden da sie ja einen Zweck erfüllen – sie trennen 2 Subbäume. Deshalb werden sie ersetzt – und zwar durch das nächste Element im Blattknoten nach dem zu löschendem Element.

Bsp:

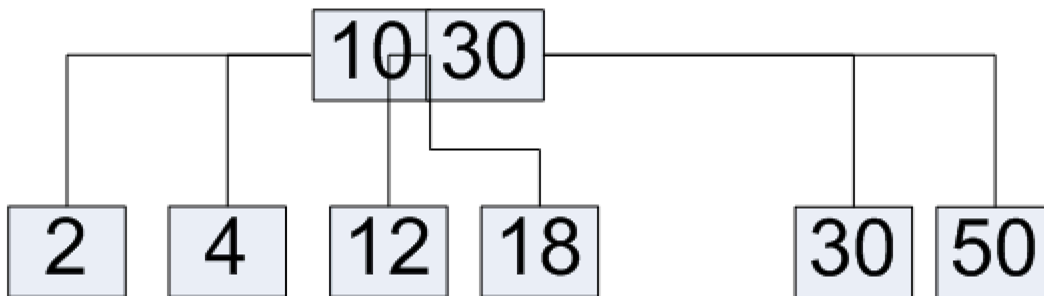


Die 20 soll gelöscht werden! Damit die 20 in der Wurzel verschwindet wird sie durch die 30 ersetzt.

Schritt B.) Delegiere das löschen an den geeigneten Sub-Baum.

Dies bedeutet nichts anderes als das der Sub-Baum in dem sich das zu löschende Element befindet mit der Delete-Funktion aufgerufen wird.

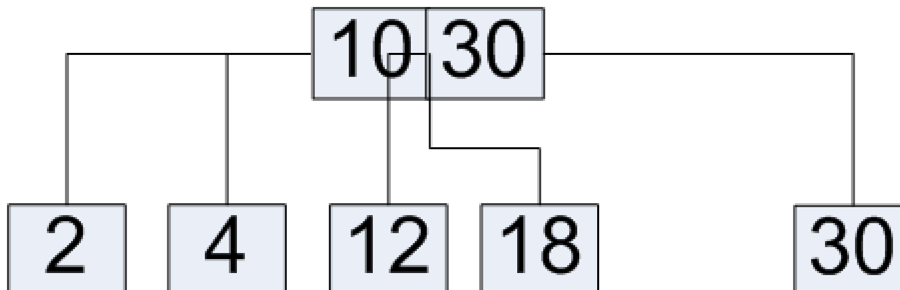
In dem vorherigem Beispiel würde also das löschen im rechten Blattknoten mit der 20 ausgeführt werden. Bei Blattknoten wird einfach nur das entsprechende Element entfernt so dass der neue Baum nun so aussieht:



Schritt C.) Wenn Sub-Baum zu klein ist -> | Balanciere |

Im Vorherigem Beispiel sind wir fertig, da der Sub-Baum aus dem gelöscht wurde weiterhin die Bedienung – mindestens 2 Elemente – erfüllt.

Löschen wir allerdings jetzt noch die 50:

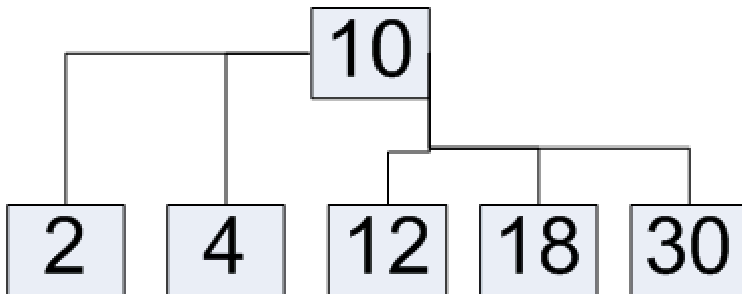


So ist der dritte Blattknoten jetzt unterbesetzt (Nur die 30).

Als erstes wird versucht von den Geschwistern überschüssige Elemente zu kriegen.
Elemente sind überschüssig wenn sie von ihrem Knoten entfernt werden können ohne das danach die Regel – mindestens t/2 Elemente – verletzt wird.

Im Beispiel gibt es nur ein linkes Geschwisterchen und dies hat bereits nur noch 2 Elemente – kann also keines entbehren.

Um die Unterdeckung an Elementen zu beheben wird der fehlerhafte Knoten jetzt mit seinem Geschwisterchen verschmolzen.



Der Seperator-Schlüssel der den fehlerhaften knoten und sein Geschwisterchen getrennt hat (die 30 in der Wurzel) wird nicht mehr benötigt und wird entfernt. Die Elemente des Fehlerhaften Knotens werden zum Geschwisterchen hinzugefügt. Wenn der fehlerhafte Knoten (30) kein Blatt wäre sondern Sub-Bäume hätte – könnte der Rechte Sub-Baum einfach mitgenommen werden – für den linken wäre allerdings kein Platz. Eine einfache Lösung des Problems wäre es, sämtliche Elemente des gesamten Teilbaums für den kein platzt ist einzelnd einzufügen (Siehe Implementierung). Da dies sehr ineffizient ist und eine Erhöhung der Komplexitätsklasse bewirkt sollte die Unterdeckung durch die Großeltern aufgehoben werden.

Im Worst-Case werden allerdings selbst die Großeltern dieses Problem haben (Sämtliche Knoten minimal gefüllt)

5.1) Delete Fazit

Lösch-Operationen können zu umfangreichen und aufwendigen neustrukturierungen im gesamten oder in großen Teilen des Baumes führen während denen der Index für andere Anfragen nicht zur Verfügung steht. In Datenbank-Systemen wollen aber mitunter eine große Anzahl Benutzer simultan auf Datensätze zugreifen.

Deshalb wird in der Praxis meist auf das verschmelzen von Bäumen verzichtet – und die resultierende Verletzung der B-Baum-Definition hingenommen - auf die Gefahr hin das der B-Baum im Worst-Case entarten kann.