Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

ETH Institut für Verhaltenswissenschaften
CAB G 83.2
Universitätsstrasse 6
CH-8092 Zürich
http://www.ifvll.ethz.ch

Leitprogramm zum Thema

Indexierung in Datenbanken

Eine Einführung in die Indexierung von Datenbanken

Ein Leitprogramm im Fachgebiet Informatik

Zielgruppe: Technikerschule / Informatik-Berufsschule 4. Semester

Vorkenntnisse: Grundkenntnisse in SQL, einfache Datenbank-Administrationsaufgaben, wie das Kreieren und Löschen von Tabellen

Onlinematerial: http://db.logging.ch

Bearbeitungsdauer: ca. 20 Lektionen

Autor: Franco Hug
ETH Zürich (D-INFK)

Betreuer: Josef Wetzel
Technische Berufsschule Zürich (TBZ)

Fassung: Version 1.3 vom 16. Januar 2014

Dieses Manual wurde erstmals im April 2010 an der Technischen Berufsschule Zürich (TBZ) erprobt.

Einführung: Um was geht es in diesem Leitprogramm?

Sie haben in der Datenbank-Vorlesung oder in Ihrem Arbeitsalltag bestimmt schon einige SELECT-Befehle abgesetzt, ohne sich dabei Gedanken zu machen, was die Datenbank alles erledigen muss, um Ihnen das gewünschte Resultat anzuzeigen. Doch haben Sie sich schon einmal gefragt, wie es möglich ist, dass die Datenbank Ihnen aus einer grossen Datenbank-Tabelle mit einigen 100'000 Datensätzen genau das richtige Suchresultat zurückliefert? Und das erst noch in einem Bruchteil einer Sekunde?

Die Datenbank schafft das nur dank der Indexierung, welche Sie in diesem Leitprogramm kennen lernen werden.

Im **Kapitel 1** lernen Sie den Begriff "Indexierung" im allgemeinen kennen, d.h. unabhängig von Datenbanken. Sie werden sehen, dass wir Indexierung im Alltag verwenden, ohne dass wir uns dessen bewusst sind. Nach der Bearbeitung des Kapitels werden Sie ein Grundverständnis für die Thematik entwickelt haben.

Im **Kapitel 2 und 3** lernen Sie die theoretischen Grundlagen kennen, welche für das Verständnis der Datenbank-Indexierung wichtig sind. Sie werden sehen, wie ein Index aufgebaut ist. Dies wird Ihnen helfen zu erkennen, mit welchen Problemen sich eine Datenbank tagtäglich herumschlagen muss. Vielleicht sind Sie nach dem Bearbeiten dieser zwei Kapitel etwas nachsichtiger mit Ihrer Datenbank, wenn das Suchresultat einmal nicht so schnell erscheint wie erwartet.

Im **Kapitel 4 und 5** lernen Sie die Indexierung in der Praxis kennen. Sie werden den Unterschied von Datenbankabfragen mit und ohne Indexierung hautnah miterleben. Sie werden lernen, wie eine Abfrage schneller gemacht werden kann, wenn die Indexierung richtig eingesetzt wird.

Nach der Bearbeitung aller 5 Kapitel werden Sie ein gutes Verständnis der Indexierung erlangt haben. Dies wird Ihnen helfen, Datenbankabfragen zu optimieren und allfällige Probleme zu erkennen. Zudem wird Ihnen spätestens dann klar sein, wieso das Thema Indexierung in Datenbanken sehr wichtig ist – es hängt direkt mit der Suchgeschwindigkeit Ihrer Datenbank zusammen.

Nun wünsche ich Ihnen viel Spass und Erfolg beim Bearbeiten dieses Leitprogrammes!

Franco Hug / Zürich, 04.. Mai 2010

Inhaltsverzeichnis

Arbeitsanleitung

Wie gehen Sie mit dem vorliegenden Leitprogramm am besten um? In den nächsten Lektionen werden Sie weitgehend selbständig lernen und arbeiten. Alles, was Sie am Schluss wissen müssen, können Sie alleine und in Ihrem eigenen Tempo mit diesem Leitprogramm erarbeiten.

Der Einfachheit halber, und wegen der besseren Übersicht, sind alle Kapitel gleich aufgebaut. Die folgenden Symbole zeigen Ihnen, was Sie als nächstes erwartet:

Übersicht

Wie ein Englein von oben, erhalten Sie hier eine Übersicht, um was es in diesem Kapitel geht.

Lernziele

Dieser Abschnitt beschreibt, was Sie nach dem Bearbeiten dieses Kapitels können.

Nun sind Sie gefragt!

Jetzt geht's zur Sache, hier lernen Sie etwas! Lesen Sie den Abschnitt gut durch, so dass Sie am Schluss die Fragen von Homer Simpson beantworten können.

Zusammenfassung

Am Schluss des Kapitels präsentiert Ihnen Marge eine Zusammenfassung vom Kapitel.

Wissenssicherung

Homer übernimmt die Wissenssicherung: Er wird von Zeit zu Zeit kleine Fragen oder Aufgaben einstreuen, die Sie beantworten sollen. Die Fragen sollen Ihr Verständnis fördern. Wenn Sie gut aufgepasst haben, dann schaffen Sie das mit Links!

Lösungen zur Wissenssicherung

Am Schluss des Kapitels präsentiert Ihnen Homer die Lösungen zu den Wissenssicherungsfragen, prägen Sie sich diese gut ein. Vergleichen Sie Ihre Lösung mit der Lösung von Homer!

Lernkontrolle

Bart Simpson übernimmt die Lernkontrolle. Hier können Sie selbst überprüfen, ob Sie das Kapitel verstanden haben, indem Sie die Fragen von Bart beantworten.

Lösungen zur Lernkontrolle

Am Schluss des Kapitels präsentiert Ihnen Bart seine Lösungen zu den Lernkontrollfragen. Überprüfen Sie Ihre Resultate anhand der Lösungen. Falls Sie nicht weiter kommen, finden Sie hier die Lösungen

Kapitel 1: Was ist Indexierung?

1.1 Übersicht: Worum geht es?

Haben Sie sich schon gefragt, was ein Index ist, wozu er gebraucht wird, woher der Begriff kommt, und was Indexierung bedeutet? In diesem Kapitel erfahren Sie es! Indexierung treffen wir nicht nur im Datenbanken-Umfeld an, sondern tagtäglich – wenn wir z.B. eine Telefonnummer nachschauen oder wissen wollen, wann der nächste Zug fährt.

Wenn Sie dieses Kapitel bearbeitet haben, haben Sie einen guten Überblick über die Indexierung erhalten. Im Kapitel 2 werden Sie dann sehen, was für verschiedene Indexe es gibt.

1.2 Lernziele

In diesem Kapitel lernen Sie folgendes:

- Sie wissen was ein Index ist – ganz generell und im Kontext von Datenbanken
- Sie kennen Beispiele aus dem Alltag, wo Indexe vorkommen
- Sie verstehen, auf was es ankommt, wenn man in einem Index/Verzeichnis etwas effizient suchen will
- Sie haben verstanden, wozu die Indexierung in Datenbanken gut sein soll

1.3 Telefonbücher und andere Verzeichnisse

Das Wort "Index" kommt ursprünglich aus dem Lateinischen und bedeutet "Zeigefinger". Dies erklärt auch, wieso der Zeigefinger auf Englisch **index finger** heisst – weil man mit ihm auf etwas zeigen kann. Weitere Bedeutungen sind "Übersicht", "Titel", "Anzeiger" oder "Inhaltsverzeichnis". Je nach Fachgebiet gibt es weitere Bedeutungen von diesem Wort. Im Datenbanken-Umfeld ist "**Inhaltsverzeichnis**" die zutreffende Bezeichnung. Der "Landesindex der Konsumentenpreise", welcher monatlich vom Bundesamt für Statistik (BfS) heraus gegeben wird, ist ein gutes Beispiel für einen "Anzeiger": er zeigt die Teuerung und die damit verbundene Stimmung der Konsumenten an.

Für viele Anwendungen, insbesondere im Datenbank-Bereich, meint man mit einem Index nichts anderes als ein **Verzeichnis**. Beispiele aus dem Alltag dafür sind z.B. das Telefonbuch, der SBB-Fahrplan, das Stichwortverzeichnis eines Fachbuches, oder der öffentliche Autoindex, wo Sie auf Grund der Autonummer den Fahrzeughalter nachschauen können. Den Autoindex gibt es sowohl als gebundenes Buch, als auch online im Internet.

Der Zweck eines Verzeichnisses ist, uns die **Suche** nach den gewünschten Daten zu vereinfachen. Deshalb nützen Verzeichnisse meistens nur etwas, wenn sie **sortiert** sind. Stellen Sie sich vor, das Telefonbuch wäre nicht alphabetisch nach Städten, Nachnamen und Vornamen sortiert... es wäre wohl nutzlos. Das gleiche gilt für die anderen genannten Beispiele. Wenn Sie im Stichwortverzeichnis Ihres Fachbuches einen Begriff suchen, dann erwarten Sie, dass dieses alphabetisch sortiert ist – ansonsten würde es wohl nichts nützen, dann wären Sie vermutlich schneller, wenn Sie das Buch durchblättern würden. Im Fall vom Autoindex ist es hilfreich, wenn dieser alphanumerisch nach den Autonummern sortiert ist, besonders wenn er in der Form eines Buches benutzt wird.

Wissenssicherung: Beantworten Sie die folgenden Fragen

a) Nennen Sie zwei weitere Beispiele aus dem Alltag oder Ihrem Berufsleben, wo Sie schon Verzeichnisse resp. Indexe angetroffen haben.

b) Stellen Sie sich vor, dass Sie ein 50seitiges Buch geschrieben haben. Sie überlegen sich nun, ob Sie Ihr Buch mit einem Stichwortverzeichnis ergänzen sollen. Was spricht für ein solches Register, was dagegen?

Die obige Erklärung des Begriffs "Index" hat mit Datenbanken noch nicht viel zu tun. In den folgenden Abschnitten werden wir sehen, wie der Begriff "Index" in die Datenbank-Welt passt.

1.4 Eine einfache Tabelle

Dieser Abschnitt hat zum Ziel, Ihre Datenbank-Kenntnisse etwas aufzufrischen.

Der Einfachheit halber verwenden wir für die kommenden Beispiele die Tabelle `SIMPSONS`, welche wie folgt definiert ist:

```
CREATE TABLE SIMPSONS (
    ID              NUMBER(19) PRIMARY KEY,
    FIRSTNAME       VARCHAR2(30),
    LASTNAME        VARCHAR2(30),
    SSN             NUMBER(19)
);
```

Die Tabelle enthält 4 Kolonnen (Deutsch: **Spalte**, Englisch: **Columns**):

- `ID` ist der Primärschlüssel (**Primary Key**) – jede Zahl kommt in dieser Tabellenspalte höchstens 1 Mal vor.
- `FIRSTNAME` ist der Vorname einer Person aus der Fernsehserie "The Simpsons"
- `LASTNAME` ist der Nachname einer Person der aus Fernsehserie "The Simpsons"
- `SSN` sei die Sozialversicherungsnummer (social security number) der Person

Die Daten einer Datenbank-Zeile werden **Tupel** oder **Record** genannt. Ein Tupel erkennt man daran, dass es üblicherweise in Klammern geschrieben wird: (ID,FIRSTNAME,LASTNAME,SSN). Der/die Primary Key(s) wird/werden oftmals unterstrichen, um ihn/sie speziell zu kennzeichnen.

Ein Primary Key (Primärschlüssel) ist ein eindeutiger Schlüsselwert, der es erlaubt, die zu ihm gehörende Tabellenzeile (English: **Row**) eindeutig zu identifizieren. Das bedeutet, dass jeder Wert in der Kolonne ID höchstens 1 Mal vorkommen darf. Ein Primary Key kann sich auch über mehrere Tabellenspalten erstrecken. So wäre es im obigen Beispiel möglich, dass der Primary Key aus dem Tupel (ID,SSN) besteht. In diesem Fall muss das Tupel eindeutig sein, in den einzelnen Spalten des Tupels dürfen jedoch die gleichen Werte mehrfach vorkommen.

Im Gegensatz zum Primary Key (ID) dürfen in den anderen Kolonnen (FIRSTNAME, LASTNAME, SSN) die gleichen Werte mehrfach vorkommen.

Ein **Identifier** ist eine Zeichenkette (String), welche ein Objekt in der Datenbank eindeutig identifiziert. Daher kommt wohl auch sein Name. Datenbank-Objekte sind z.B. Tabellen, die Kolonnen der Tabellen, Views oder Stored Procedures. Der Identifier muss innerhalb einer Objektklasse eindeutig sein, d.h. es darf in der Datenbank nur eine Tabelle SIMPSONS geben.

Kolonnen müssen innerhalb einer Tabelle eindeutig sein, jedoch darf der gleiche Kolonnenname in einer anderen Tabelle verwendet werden. Dies ist deshalb so, weil der Kolonnenname sich hierarchisch mit dem Tabellennamen zusammen setzt:

SIMPSONS.FIRSTNAME und THE_SIMPSONS.FIRSTNAME bezeichnen beide eine Kolonne FIRSTNAME, jedoch in zwei verschiedenen Tabellen.

Die Tabelle SIMPSONS enthält die folgenden 10 Personen:

	SIMPSONS			
	ID	FIRSTNAME	LASTNAME	SSN
Zeile 01:	13	Milhouse	van Houten	13259
Zeile 02:	45	Waylon	Smithers	89241
Zeile 03:	82	Marge	Simpson	71430
Zeile 04:	07	Monty	Burns	52367
Zeile 05:	93	Bart	Simpson	38172
Zeile 06:	68	Seymour	Skinner	94891
Zeile 07:	23	Homer Jay	Simpson	27272
Zeile 08:	73	Moe	Szyslak	06585
Zeile 09:	58	Lisa	Simpson	66914
Zeile 10:	36	Chief Clancy	Wiggum	40103

Tabelle 1.1: Die SIMPSONS Tabelle

Wie Sie sehen können, ist die obige Tabelle nicht sortiert – weder nach der ID, noch nach Vorname/Nachname, und auch nicht nach der Sozialversicherungsnummer. Dies ist in Datenbanken üblich, ja sogar der Normalfall. Je mehr Mutationen (Insert/Delete) es auf einer Tabelle gibt, desto weniger sind die Daten in der gewünschten Reihenfolge. Die Tabelle wird mit der Zeit **fragmentiert**, d.h. sie ist evtl. nicht mehr zusammenhängend gespeichert.

Die Reihenfolge der Datensätze in einer Tabelle hängt demnach von der Reihenfolge der Insert und Delete Statements ab.

Wir möchten nun die Sozialversicherungsnummer der Person mit der ID=23 (Homer Jay Simpson) heraus finden. Bei dieser kleinen Tabelle sehen Sie auf den ersten Blick, dass sich die gewünschten Daten in der Zeile 07 befinden. Doch bedenken Sie: Datenbanken speichern üblicherweise sehr grosse Datenmengen, so dass das mit dem ersten Blick nicht mehr funktioniert.

Wissenssicherung: Beantworten Sie die folgenden Fragen

c) Wie würden Sie in der obigen Tabelle nach der SSN von Homer Jay Simpson suchen? Beschreiben Sie, wie Sie vorgehen würden.

d) Wie viele Suchschritte brauchen Sie in Aufgabe c) minimal, wieviele maximal?

e) Nehmen Sie nun an, dass die obige Tabelle nach der ID sortiert ist: Wie viele Suchschritte brauchen Sie jetzt für die Suche nach Homer Jay Simpson's SSN im Minimum, wieviele im Maximum? Beschreiben Sie, wie Sie bei der Suche vorgehen.

In der Aufgabe e) haben Sie die maximale Anzahl Suchschritte durch Probieren heraus gefunden, oder allenfalls durch systematisches Überlegen oder Anwendung einer Methode. Am schnellsten geht es wohl, wenn wir das Suchintervall immer halbieren. Für kleine Tabellen geht das noch gut "von Hand", für Tabellen mit mehr als 1'000 Rows wird das etwas umständlich. Doch wir können die maximale Anzahl nötiger Suchschritte auch berechnen!

Da wir das Suchintervall so lange halbieren, bis wir das Resultat gefunden haben oder nur noch eine Zahl übrig bleibt, können wir die Frage etwas umformulieren:

Wieviel Mal können wir eine Zahl durch 2 dividieren, bis ≤ 1 raus kommt?

Aus der Mathematik kennen wir die 3 Funktionen Potenz, Wurzel und Logarithmus. Sie hängen wie folgt zusammen:

$$y = x^n \qquad x = \sqrt[n]{y} \qquad n = \log_x y$$

Wie Sie vielleicht gemerkt haben, spielt die Zahl 2 dabei eine wichtige Rolle. Das werden wir dann im nächsten Kapitel genauer anschauen. Wenn man für x=2 einsetzt, erhält man die folgenden Formeln:

$$y = 2^n \qquad 2 = \sqrt[n]{y} \qquad n = \log_2 y$$

y entspricht dabei der Anzahl Rows in der Tabelle, und n der maximalen Anzahl Suchschritte. x=2 kommt daher, weil wir das Suchintervall immer halbieren. Die allgemeine Formel für die Ermittlung der maximalen Anzahl Suchschritte lautet demnach:

$$\textbf{Anzahl Suchschritte} = \lceil \log_2 (\textbf{Anzahl Rows}) \rceil$$

Die Zeichen $\lceil \ldots \rceil$ sagen, dass man das Resultat auf die nächste ganze Zahl aufrunden muss.

Nun können wir also den 2er-Logarithmus von 10 ziehen, und erhalten die Anzahl maximaler Suchschritte:

$n = \log_2 10 = 3.3219\ldots$ → aufgerundet ergibt das max. **4 Suchschritte**.

Mit dem Taschenrechner können wir normalerweise den 2er-Logarithmus nicht direkt ziehen. Normale Taschenrechner kennen nur den 10er-Logarithmus (log) und den natürlichen Logarithmus (ln). Aber dafür gibt es ja den Logarithmensatz aus der Mathematik:

$$n = \log_x y = \frac{\ln y}{\ln x} = \frac{\log y}{\log x}$$

Die Suchmethode mit dem fortlaufenden Halbieren des Intervalls verhält sich demnach logarithmisch. Somit ist die Komplexität (Ordnung) dieser Suchmethode logarithmisch. Das gibt man in der Regel mit dem **Landau-Symbol** O(...) an:

Komplexität (Ordnung): **O(log n)**

Wissenssicherung: Beantworten Sie die folgenden Fragen

f) Berechnen Sie die maximal nötige Anzahl Suchschritte für eine Tabelle, welche 100 Rows enthält.

g) Wieso verwenden wir für die Berechnungen den 2er-Logarithmus ($\log_2$), und nicht den 10er-Logarithmus (log resp. $\log_{10}$)?

1.5 Verzeichnisse in Datenbanken

Vielleicht fragen Sie sich jetzt: was hatte der letzte Abschnitt mit Indexierung zu tun? Die Antwort ist einfach: noch nicht viel! Das Ziel war, Ihre SQL-Kenntnisse etwas aufzufrischen.

Wir haben gesehen, dass die Suche wesentlich schneller ist, wenn eine Tabelle sortiert ist. Nachdem wir die Tabelle nach der ID sortiert hatten, benötigten wir im Maximum nur noch 4 Suchschritte anstatt 10, wie dies bei der unsortierten Tabelle der Fall gewesen ist.

Allerdings haben wir auch gelernt, dass neue Daten immer am Ende einer Tabelle angefügt werden, was soviel bedeutet, dass die Tabelle niemals automatisch sortiert wird. Die Datenbank wüsste auch gar nicht, nach welchen Kolonnen sie die Daten sortieren soll.

Was nun? Um dieses Problem zu lösen gibt es Indexe! **Indexe sind die Verzeichnisse einer Datenbank**. Wenn wir in der Tabelle SIMPSONS z.B. nicht nach der ID, sondern nach der Sozialversicherungsnummer (SSN) der Person suchen wollen, dann können wir für diese Kolonne einen Index anlegen.

```
CREATE INDEX SSN_INDEX ON SIMPSONS(SSN);
```

Die Datenbank baut nun eine Art Stichwortverzeichnis für die Werte in der Kolonne SSN auf. Wenn man zukünfig in der Tabelle SIMPSON nach einer Sozialversicherungsnummer sucht, dann weiss die Datenbank, dass es auf dieser Kolonne einen Index gibt. Sie verwendet dann für die Suche den Index, anstatt einen Full Table Scan zu machen.

Wenn wir nun nach der Person suchen wollen, welche die Sozialversicherungsnummer 27272 hat,

```
SELECT * FROM SIMPSONS WHERE SSN=27272;
```

wird die Datenbank selbst entscheiden, wie sie schneller zum Resultat kommt: Entweder verwendet sie den Index SSN_INDEX, oder sie macht einen Full Table Scan. Auf jeden Fall bekommen wir von der Datenbank schnell das Resultat!

Übrigens, für den Primary Key einer Tabelle legt die Datenbank immer automatisch einen Index an, da davon auszugehen ist, dass dieser Schlüssel oftmals für die Suche resp. Identifizierung einer Row gebraucht wird. Diese Arbeit nimmt uns die Datenbank ab.

Falls wir einen Index nicht mehr brauchen, können wir ihn wieder löschen:

```
DROP INDEX SSN_INDEX;
```

Beachten Sie, dass der obige Befehl nur den Index löscht, nicht jedoch die Nutzdaten! Dies darf nicht verwechselt werden: Die Nutzdaten – d.h. die Daten in der Tabelle SIMPSONS – werden durch das Kreieren oder Löschen eines Indexes niemals verändert. Ein Index ist lediglich ein **Hilfsmittel** für die Datenbank, um die gewünschte Row schneller finden zu können. Dafür baut die Datenbank eigene Indexstrukturen auf, die dem Anwender verborgen bleiben. Sie merken höchstens, dass eine Suche schneller wird.

Wissenssicherung: Beantworten Sie die folgenden Fragen

h) Wieso macht die Datenbank nicht automatisch auf jeder Kolonne einen Index?

i) Nennen Sie 2 Gründe, wieso es sinnvoll ist, dass die Datenbank auf dem Primary Key automatisch einen Index anlegt.

1.6 Zusammenfassung

In diesem Kapitel haben Sie gelernt, was ein Index ist – und zwar ganz generell, sowie im Speziellen im Datenbank-Bereich. Sie haben gesehen, dass Indexe auch in unserem Alltag eine wichtige Rolle spielen, obwohl wir uns dessen meistens gar nicht bewusst sind. Begriffe wie Index, Verzeichnis, Register, Anzahl Suchschritte oder Landau-Symbol sind Ihnen nun geläufig. Zudem haben Sie Ihre Datenbank-Kenntnisse etwas aufgefrischt, so dass Sie wieder wissen, was eine Tabelle, eine Kolonne, eine Row oder ein Primary Key ist. Sie haben gelernt, auf was es ankommt, wenn man in einem Index etwas suchen will.

Im folgenden Kapitel beschäftigen wir uns damit, wie ein Index aufgebaut ist.

1.7 Lösungen zu den Wissenssicherungsfragen

a) *Nennen Sie zwei weitere Beispiele aus dem Alltag oder Ihrem Berufsleben, wo Sie schon Verzeichnisse resp. Indexe angetroffen haben.*

Tramfahrplan an der Tramhaltestelle: Für jede Tramlinie gibt es ein separates Anschlagsbrett. Und auf jedem Anschlagsbrett sind die Abfahrtszeiten nach Stunden und Minuten sortiert. Dies erlaubt es Ihnen, schnell heraus zu finden, wann das nächste Tram fährt.

Google: Bestimmt haben Sie schon einmal die Internet-Suchmaschine Google verwendet (http://www.google.com). Sie geben einen Suchbegriff ein, und Sekunden später haben Sie die Antwort. Google scheint für diese Arbeit irgend welche Verzeichnisse zu verwenden, sonst wäre die Antwortzeit nicht so schnell.

Domainnamen: Auf der Internetseite von SWITCH (http://www.nic.ch) – das ist die Domainnamen-Vergabestelle für Domains mit der Endung .ch – gibt es ein sogenanntes WHOIS-Verzeichnis, worin Sie z.B. nachschauen können, wer für den Domainnamen schweiz.ch verantwortlich ist. Im Hintergrund sind die Daten in einem Verzeichnis abgelegt, das nach Domainnamen sortiert ist.

Coop Passabene: Haben Sie schon einmal bei Coop etwas eingekauft, und dabei die Produkte selbst gescannt? Sekundenschnell erscheint auf Ihrem Terminal der Name des Produktes sowie der Preis. Coop führt im Hintergrund eine Liste (Verzeichnis) von sämtlichen Produkten und Preisen, welche nach der auf dem Produkt angebrachten Strichcode-Nummer sortiert ist.

Dies sind nur ein paar Beispiele, bestimmt finden Sie weitere!

b) *Stellen Sie sich vor, dass Sie ein 50seitiges Buch geschrieben haben. Sie überlegen sich nun, ob Sie Ihr Buch mit einem Stichwortverzeichnis ergänzen sollen. Was spricht für ein solches Register, was dagegen?*

Pro: Es ist für den Leser Ihres Buches praktisch, wenn er etwas zu einem bestimmten Stichwort nachlesen möchte, ohne gleich das ganze Buch lesen zu müssen. Dies gilt besonders für Fach-/Sachbücher, weniger für Romane oder Kommödien. Es ist gut möglich, dass Sie Ihr Buch besser verkaufen, wenn es ein gutes Stichwortverzeichnis aufweist.

Contra: Sie müssen das Stichwortverzeichnis erstellen, das bedeutet Arbeit! Ein gutes Stichwortverzeichnis zu erstellen ist nicht ganz einfach, man kann leicht Stichworte vergessen. Dann werden sich die Leser beklagen und das Stichwortverzeichnis evtl. gar nicht verwenden. Denken Sie auch an den Unterhalt: Wenn Sie auf Seite 25 ein neues Kapitel einfügen, müssen Sie diese Stichworte ebenfalls indizieren, und dafür Sorge tragen, dass die Referenzen im Stichwortverzeichnis auf die Seiten 26-50 noch stimmen.

c) *Wie würden Sie in der obigen Tabelle nach der SSN von Homer Jay Simpson suchen? Beschreiben Sie, wie Sie vorgehen würden.*

Da gibt es nicht viel zu vereinfachen. Weil die Tabelle nicht sortiert ist, kommt man nicht darum herum, die ganze Tabelle so lange zu durchsuchen, bis man das Resultat gefunden hat. Ob man mit der Suche von oben oder von unten her beginnt, spielt statistisch keine Rolle. Für die Suche nach "Homer Jay Simpson" wäre es besser, mit der Suche von unten her zu beginnen. Die Suche nach seiner Frau "Marge Simpson" würden wir wohl besser von oben her beginnen. Das Datenbanksystem kennt jedoch den Inhalt der Zeilen nicht, und kann deshalb nicht entscheiden, ob besser von oben oder von unten her begonnen wird. Deshalb müssen wir davon ausgehen, dass im schlimmsten Fall die ganze Tabelle durchsucht werden muss. Dieses Verfahren nennt man **Full Table Scan**.

d) *Wie viele Suchschritte brauchen Sie in Aufgabe c) minimal, wieviele maximal?*

Minimum: 1 Suchschritt. Wenn Sie Glück haben, dann befindet sich die Daten in der ersten Zeile. Hätten Sie nach "Milhouse van Houten" gesucht, wären Sie sofort fündig geworden.

Maximum: Der schlimmste Fall (worst case scenario) ist, wenn sich die gesuchten Daten in der letzten Zeile befinden, welche wir durchsuchen. Wenn wir z.B. nach "Chief Clancy Wiggum" suchen, und die Tabelle von oben nach unten durchsuchen, dann brauchen wir 10 Suchschritte.

Im schlimmsten Fall muss also mit einem Full Table Scan die gesamte Tabelle durchsucht werden.

e) *Nehmen Sie nun an, dass die obige Tabelle nach der ID sortiert ist: Wie viele Suchschritte brauchen Sie jetzt für die Suche nach Homer Jay Simpson's SSN im Minimum, wieviele im Maximum? Beschreiben Sie, wie Sie bei der Suche vorgehen.*

Minimum: 1 Suchschritt. Auch hier gilt: wenn wir Glück haben, befinden sich die gesuchten Daten im ersten Record.

Maximum: Da die Daten sortiert sind, können wir die Suche optimieren. Wir können zuerst in die Mitte der Tabelle springen und die Zahlen vergleichen. Ist die ID kleiner als die Zahl, dann wissen wir, dass das Resultat in der ersten Hälfte der Datensätze liegen muss. Falls die ID grösser ist, dann befindet sich das Resultat entsprechend in der zweiten Tabellenhälfte. Falls die ID mit der Zahl überein stimmt, dann haben wir die Row bereits gefunden.

Eine Suche nach "Chief Clancy Wiggum" (ID=36) sieht demnach wie folgt aus:

07	13	23	36	45	**58**	68	73	82	93	
07	13	**23**	36	45	58	68	73	82	93	
07	13	23	36	**45**	58	68	73	82	93	
07	13	23	**36**	45	58	68	73	82	93	← Resultat nach 4 Suchschritten

Das ist doch schon wesentlich besser als der Full Table Scan bei Aufgabe d) wo wir für die Suche nach dem "Chief" noch 10 Suchschritte gebraucht haben!

Die Suche nach "Homer Jay Simpson" (ID=23) sieht wie folgt aus:

07	13	23	36	45	**58**	68	73	82	93	
07	13	**23**	36	45	58	68	73	82	93	← Resultat nach 2 Suchschritten

Hier kommen wir sogar mit nur 2 Suchschritten zum Ziel.

f) *Berechnen Sie die maximal nötige Anzahl Suchschritte für eine Tabelle, welche 100 Rows enthält.*

Für die Berechnung können wir die folgende Formel verwenden:

$$\text{Anzahl Suchschritte} = \lceil \log_2 (\text{Anzahl Rows}) \rceil$$

$$\text{Anzahl Suchschritte} = \lceil \log_2 (100) \rceil$$

Damit wir das mit dem Taschenrechner berechnen können, verwenden wir den Logarithmensatz:

$$\text{Anzahl Suchschritte } n = \log_2 100 = \frac{\ln 100}{\ln 2} = \frac{\log 100}{\log 2} = 6.6438...$$

Wenn wir das Resultat aufrunden, erhalten wir **7 Suchschritte**.

g) *Wieso verwenden wir für die Berechnungen den 2er-Logarithmus ($\log_2$), und nicht den 10er-Logarithmus (log resp. $\log_{10}$)?*

Weil wir das Suchintervall immer halbieren, d.h. duch 2 teilen – und nicht durch 10.

h) *Wieso macht die Datenbank nicht automatisch auf jeder Kolonne einen Index?*

Indexe brauchen Platz auf der Datenbank! Die Datenbank legt für jeden Index neue Datenstrukturen an. Deshalb sollte man niemals Indexe auf Vorrat anlegen, sondern nur, wenn man sie wirklich braucht – z.B. für eine effiziente Suche.

i) *Nennen Sie 2 Gründe, wieso es sinnvoll ist, dass die Datenbank auf dem Primary Key automatisch einen Index anlegt.*

Grund 1: Suche

Weil der Primary Key eine Row eindeutig identifiziert, ist davon auszugehen, dass viele Suchanfragen über den Primary Key gehen werden.

Grund 2: JOIN

Nehmen Sie an, dass die Sozialversicherungsnummer ein einer separaten Tabelle gespeichert ist. Die Tabellen seien wie folgt definiert:

```
CREATE TABLE SIMPSONS (
    ID              NUMBER(19) PRIMARY KEY,
    FIRSTNAME       VARCHAR2(30),
    LASTNAME        VARCHAR2(30)
);

CREATE TABLE SIMPSONS_SSN (
    ID_SSN          NUMBER(19) PRIMARY KEY,
    SSN             NUMBER(19)
);
```

Wenn Sie nun die Sozialversicherungsnummer von "Homer Jay Simpson" wissen wollen, können Sie folgende Anfrage absetzen:

```
SELECT s.SSN
  FROM SIMPSONS p JOIN SIMPSONS_SSN s
 WHERE p.FIRSTNAME='Homer Jay'
   AND p.LASTNAME='Simpson'
   AND p.ID=s.ID_SSN;                    ← JOIN-Bedingung
```

Diese Abfrage verbindet die Tabellen SIMPSONS und SIMPSONS_SSN mittels einem JOIN. Als Bedingung werden die Primary Keys ID und ID_SSN verwendet.

Die Datenbank muss nun für jeden Wert in der Kolonne ID den passenden Wert in der Kolonne ID_SSN suchen – das sagt die JOIN-Bedingung p.ID=s.ID_SSN.

Folglich ist es hilfreich, wenn auf den Kolonnen einer JOIN-Bedingung Indexe definiert sind. Da JOINs oftmals auf Primary Keys gemacht werden, ist dies ebenfalls ein gutes Argument, um auf Primary Keys automatisch einen Index zu erstellen.

1.8 Lernkontrolle: Hier können Sie überprüfen, ob Sie das Kapitel beherrschen

Aufgabe 1

Wäre es nicht einfacher, eine Tabelle gleich von Anfang an sortiert abzuspeichern? Die Suche wäre in der Folge doch viel einfacher! Wieso macht man das nicht?

Aufgabe 2

Was ist der Unterschied zwischen einem (Primary) Key und einem Identifier? Und für was braucht's diese?

Aufgabe 3

Erklären Sie die folgenden Begriffe aus der Datenbank-Welt:

a) Column

b) Row

c) Record

d) Primary Key

e) Full Table Scan

Aufgabe 4

Berechnen Sie die maximal nötige Anzahl Suchschritte für eine Tabelle, welche 1'000 Rows enthält. Für welche Tabellengrössen hat Ihr Resultat Gültigkeit?

Aufgabe 5

Was bedeutet die Angabe **$O(n^2)$** bei einem Suchalgorithmus? Ist das ein guter Suchalgorithmus? Begründen Sie!

Hilfe zum Landau-Symbol O(...) bekommen Sie zum Beispiel hier:
http://de.wikipedia.org/wiki/O-Notation

Aufgabe 6

Wir wollen in der Tabelle SIMPSONS nach Vor- und Nachname suchen können. Erzeugen Sie dazu einen Index, der sich über die 2 Kolonnen (LASTNAME,FIRSTNAME) erstreckt! Wie wird der Index wieder gelöscht, wenn er nicht mehr gebraucht wird?

1.9 Lösungen zu den Lernkontrollfragen

Aufgabe 1

Wäre es nicht einfacher, eine Tabelle gleich von Anfang an sortiert abzuspeichern? Die Suche wäre in der Folge doch viel einfacher! Wieso macht man das nicht?

In der Tat würde dies die Suche wesentlich vereinfachen. In der Praxis geht das jedoch nicht: Es würde zu lange dauern, wenn nach jeder Insert-Operation die ganze Tabelle neu sortiert werden müsste. Bei kleinen Tabellen, welche nur wenig Daten pro Record speichern, würde das evtl. noch gehen. Bei grösseren Tabellen oder grossen Records ist dies nicht mehr praktikabel. Das hat damit zu tun, wie die Daten physisch auf der Disk gespeichert werden. Angenommen, Sie möchten in der Mitte einer Tabelle, welche 10GB gross ist, einen Record einfügen: dann müsste das Datenbanksystem die hinteren 5GB zuerst verschieben, bevor der Insert durchgeführt werden könnte. Das dauert einfach zu lange. Deshalb werden neue Daten einfach an die Tabelle angehängt, das ist schnell.

Um die Suche können wir uns dann später kümmern, genau dafür gibt es das Konzept der Datenbank-Indexierung!

Aufgabe 2

Was ist der Unterschied zwischen einem (Primary) Key und einem Identifier? Und für was braucht's diese?

Ein (Primary) Key identifiziert eine Row in einer Tabelle eindeutig. Ein Identifier ist ein "Kennzeichner", welcher eine beliebige Ressource in der Datenbank eindeutig bezeichnet.

Aufgabe 3

Erklären Sie die folgenden Begriffe aus der Datenbank-Welt:

a) *Column:* Das ist eine Spalte in einer Datenbank-Tabelle.

b) *Row:* Das ist eine Zeile in einer Datenbank-Tabelle. Die darin enthaltenen Daten werden Record genannt. Vielleicht kennen Sie diesen Begriff aus einer Programmiersprache, z.B. Java, C oder Pascal.

c) *Record:* Das sind die Daten, welche in einer Row gespeichert sind.

d) *Primary Key:* Das ist ein eindeutiger Schlüssel, resp. Wert, aus dem Wertebereich des dazu gehörenden Identifiers. Ein Primary Key kann aus mehreren Identifiern (Columns) zusammen gesetzt sein.

e) *Full Table Scan:* Für die Suche muss potentiell die ganze Tabelle durchforstet werden. Dies tritt immer dann auf, wenn die Tabelle nicht sortiert ist

Aufgabe 4

Berechnen Sie die maximal nötige Anzahl Suchschritte für eine Tabelle, welche 1'000 Rows enthält. Für welche Tabellengrössen hat Ihr Resultat Gültigkeit?

Für die Berechnung können wir die folgende Formel verwenden:

$$\text{Anzahl Suchschritte} = \lceil \log_2 (\text{Anzahl Rows}) \rceil$$

$$\text{Anzahl Suchschritte} = \lceil \log_2 (1'000) \rceil$$

Damit wir das mit dem Taschenrechner berechnen können, verwenden wir den Logarithmensatz:

$$\text{Anzahl Suchschritte } n = \log_2 1'000 = \frac{\ln 1'000}{\ln 2} = \frac{\log 1'000}{\log 2} = 9.96578...$$

Wenn wir das Resultat aufrunden, erhalten wir **10 Suchschritte**.

Um den Gültigkeitsbereich zu berechnen, verwenden wir die Umkehrfunktion zum Logarithmus – die Potenz:

Minimale Tabellengrösse: $> 2^{10-1} + 1 = 2^9 + 1 =$ **513 Rows**

Maximale Tabellengrösse: $= 2^{10} =$ **1'024 Rows**

Das heisst, für eine Tabelle mit 513 ... 1'024 Rows benötigen wir im Maximum 10 Suchschritte.

Aufgabe 5

*Was bedeutet die Angabe **$O(n^2)$** bei einem Suchalgorithmus? Ist das ein guter Suchalgorithmus? Begründen Sie!*

Die O-Notation gibt die Komplexität (die Ordnung) eines Algorithmus' oder Verfahrens an. Dies sagt etwas über die Laufzeit aus.

Der Suchalgorithmus geht Zeile für Zeile vor – wie beim Verfahren der fortlaufenden Halbierung des Suchintervalles. Allerdings werden für eine Zeile sämtliche davor gelesenen Zeilen nochmals gelesen. Z.B. beim Verarbeiten der Zeile 05 werden die Zeilen 01 bis 04 nochmals gelesen. Somit steigt der Aufwand **quadratisch** an!

Im Vergleich zu $O(\log n)$ ist dies ein schlechter Suchalgorithmus.

Aufgabe 6

Wir wollen in der Tabelle SIMPSONS nach Vor- und Nachname suchen können. Erzeugen Sie dazu einen Index, der sich über die 2 Kolonnen (LASTNAME,FIRSTNAME) erstreckt! Wie wird der Index wieder gelöscht, wenn er nicht mehr gebraucht wird?

Index erzeugen:

```
CREATE INDEX SEARCH_BY_NAME ON SIMPSONS(LASTNAME,FIRSTNAME);
```

Index löschen:

```
DROP INDEX SEARCH_BY_NAME ON SIMPSONS;
```

Kapitel 2: Binärer (Such-)Baum

2.1 Übersicht: Worum geht es?

Spätestens seit dem Bearbeiten des ersten Kapitels wissen Sie, was Indexierung ist – sowohl ganz allgemein, als auch im Kontext von Datenbanken. Doch haben Sie sich schon die Frage gestellt, wie ein Index einer Datenbank aussieht, wie er technisch implementiert werden kann? In diesem Kapitel erfahren Sie es! Ein Datenbank-Index ist nämlich etwas komplizierter aufgebaut als der Index eines Buches, damit die Datenbank im Index schnell suchen kann.

Wenn Sie dieses Kapitel bearbeitet haben, wissen Sie, was binäre (Such-)Bäume mit Indexen zu tun haben. Im Kapitel 3 werfen wir einen Blick auf einen Spezialfall des binären Baumes, welcher in Datenbanken sehr oft angetroffen wird.

2.2 Lernziele

In diesem Kapitel lernen Sie folgendes:

- Sie wissen, was ein binärer Baum ist
- Sie kennen den Unterschied zwischen einem binären Baum und einem binären Suchbaum
- Sie wissen, auf was es bei binären Bäumen ankommt
- Sie verstehen, weshalb der binäre Suchbaum in Datenbanken eine wichtige Rolle spielt

2.3 Der binäre Baum

Im ersten Kapitel haben wir den Index einer Datenbank-Tabelle mit dem Index eines Buches verglichen. Leider ist es so, dass die Datenbank mit so einem Index, wie er bei Büchern vorkommt, nicht viel anfangen kann. Der Index eines Buches gleicht nämlich einer linearen Liste, welche alphabetisch (oder numerisch) sortiert ist. Wenn nun die Datenbank in so einem Index einen Schlüssel suchen muss, dann müsste sie die Liste vom ersten Element an so lange durchsuchen, bis sie den gesuchten Wert gefunden hätte. Dies wäre dann wieder eine "normale" Suche, dann könnten wir ja gleich in der Tabelle selbst suchen... genau um das zu verhindern, gibt es binäre (Such-)Bäume!

Im ersten Kapitel haben wir gelernt, dass ein Schlüssel durch fortlaufende Halbierung des Suchintervalles schneller gefunden werden kann, als mit der Suche von A–Z. Genau das ist das Ziel der binären Bäume.

Ein **binärer Baum** besteht aus einer **Wurzel** (Root), aus **inneren Knoten**, sowie aus **Blattknoten** (Blätter des Baumes). Die Wurzel und die inneren Knoten haben maximal 2 Kindknoten – daher kommt das Wort "binär".

Eine wichtige Kenngrösse des binären Baumes ist die (Baum-)**Höhe h**. Sie sagt aus, in welcher **Tiefe t** der tiefste Knoten des Baumes liegt. Die Tiefe t ihrerseits sagt aus, auf welcher Tiefe (Ebene) sich ein spezifischer Knoten befindet. Die Wurzel befindet sich immer auf der Tiefe

t=0. Anders als ein Baum in der freien Natur, werden binäre Bäume meistens auf dem Kopf dargestellt.

Die **maximale Tiefe** des Baumes entspricht demnach der Baumhöhe h. Sie sagt aus, wie viele Schritte man machen muss, wenn man von der Wurzel zum tiefsten Blattknoten kommen will. Die Höhe des Baumes wird also von dem Blattknoten bestimmt, welcher sich am weitesten von der Wurzel entfernt befindet – wie bei einem reellen Baum.

Nachfolgend ist ein binärer Baum abgebildet, welcher die Schlüssel der Tabelle SIMPSONS aus dem ersten Kapitel speichert. Die Reihenfolge der Schlüssel entspricht der Reihenfolge in der Tabelle (von oben nach unten, resp. von links nach rechts). Als Wurzel (rot) wurde der erste Schlüssel (ID=13) verwendet:

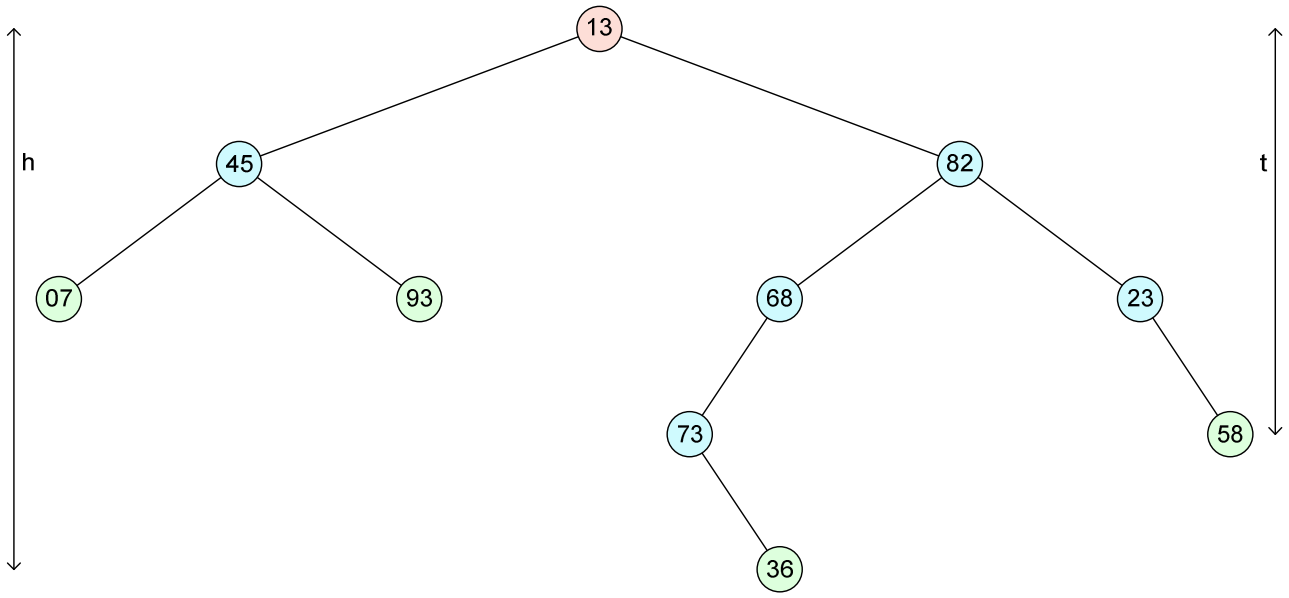


Abbildung 2.1: Binärer Baum in Tabellenreihenfolge

Legende: rot = Wurzel
blau = innere Knoten
grün = Blattknoten

h = Baumhöhe (Höhe = 4)
t = Tiefe der Knoten mit ID=58 und ID=73 (Tiefe = 3)

Wissenssicherung: Beantworten Sie die folgenden Fragen

a) Zeigen Sie, wie Sie im obigen binären Baum die Baumhöhe ermitteln.

b) Wie viele Blätter hat ein binärer Baum der Höhe h=3 im Maximum?

c) Wie viele Knoten (Wurzel, innere Knoten und Blattknoten) hat ein binärer Baum der Höhe h=3 im Maximum?

d) Wie würden Sie im obigen binären Baum nach der ID=36 suchen? Beschreiben Sie eine Variante.

Der binäre Baum ist neben der linearen Liste (Tabellenform) eine weitere Möglichkeit, wie Daten in einem Index – oder ganz generell – gespeichert werden können. Allerdings wollen wir die Daten nicht nur speichern, sondern auch wieder effizient abrufen können. Mit der Reihenfolge, welche im obigen binären Baum gewählt wurde, ist keine effiziente Suche möglich – wie Sie vielleicht in Aufgabe d) gemerkt haben. Wir waren zwar schnell mit dem Speichern (einfach die IDs schön der Reihe nach abspeichern), doch die Suche gestaltet sich schwierig. Genau dafür gibt es die binären Suchbäume!

2.4 Suchen im binären Baum

Wie wir im letzten Kapitel gesehen haben, ist in einem unsortierten binären Baum keine effiziente Suche möglich. Der Aufwand ist gleich, wie wenn wir eine Datenbank-Tabelle ganz durchsuchen müssen: im schlimmsten Fall müssen wir den ganzen Baum durchsuchen. Dieses "Abgrasen" des ganzen Baumes nennt man **traversieren**.

Beim Traversieren eines binären Baumes unterscheidet man zwischen der **Breitensuche** und der **Tiefensuche**. Wie die Namen bereits sagen, geht die Breitensuche zuerst in die Breite, und erst später in die Tiefe. Bei der Tiefensuche ist es genau umgekehrt: zuerst wird in der Tiefe gesucht, danach wird in die Breite gegangen.

Breitensuche: der Baum wird pro Baumebene abgesucht (**level-order**). Die Nummern der Knoten geben die Suchreihenfolge an, die gestrichelten Linien kennzeichnen die Baumebenen (Levels):

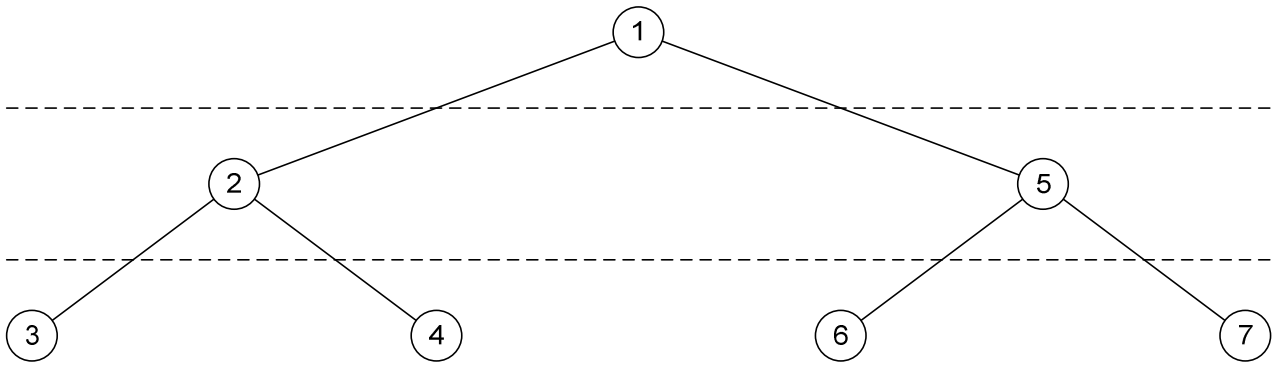


Abbildung 2.2: Breitensuche

Tiefensuche: der Baum wird zuerst in der Tiefe durchsucht, d.h. man versucht als erstes, so tief wie möglich in der Baumhierarchie abzusteigen. Die gestrichelten Linien kennzeichnen die Teilbäume:

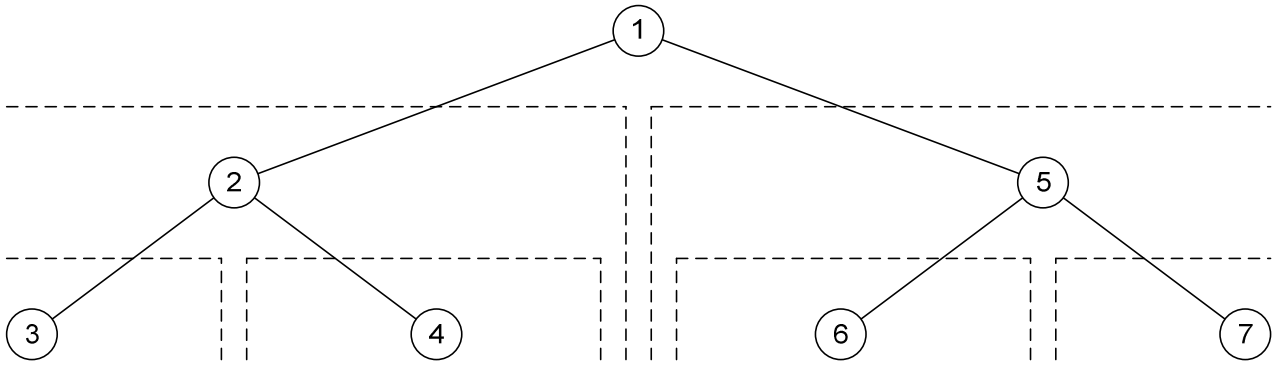


Abbildung 2.3: Tiefensuche

Bei der Tiefensuche gibt es 3 Möglichkeiten, wie man in die Tiefe absteigen kann:

a) Hauptreihenfolge

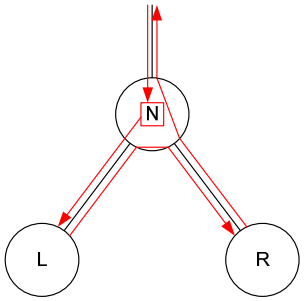


Abb. 2.4: pre-order (N-L-R)

b) Symmetrische Reihenfolge

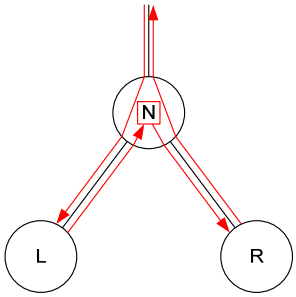


Abb. 2.5: in-order (L-N-R)

c) Nebenreihenfolge

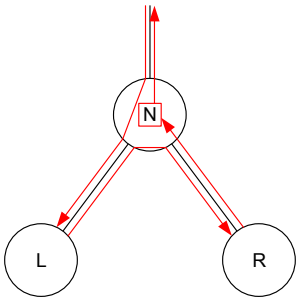


Abb. 2.6: post-order (L-R-N)

a) **pre-order** (N-L-R): Zuerst wird der Knoten **N** (Node) untersucht. Wenn man weiter suchen muss, steigt man zuerst in den linken Teilbaum **L** ab, danach in den rechten Teilbaum **R**.

   pre heisst auf Lateinisch "vor" und bedeutet hier, dass der Node N **vor** dem Absteigen in die Teilbäume überprüft wird.

b) **in-order** (L-N-R): Bevor der Knoten N (Node) untersucht wird, steigt man in den linken Teilbaum **L** ab. Erst dann wird der Knoten **N** untersucht. Anschliessend steigt man in den rechten Teilbaum **R** ab.

   in bedeutet hier, dass der Node N nach dem Absteigen in den linken Teilbaum, jedoch vor dem Absteigen in den rechten Teilbaum überprüft wird.

c) **post-order** (L-R-N): Der Knoten N (Node) wird zuletzt untersucht. Zuerst steigt man direkt in den linken Teilbaum **L** ab, danach in den rechten Teilbaum **R**. Erst jetzt wird der Knoten **N** (Node) untersucht.

   post heisst auf Lateinisch "nach" und bedeutet hier, dass der Node N **nach** dem Absteigen in die Teilbäume überprüft wird.

Wissenssicherung: Beantworten Sie die folgenden Fragen

e) Wir haben 3 Varianten für die Tiefensuche kennen gelernt. Gibt es weitere Varianten?

f) Was ist der Vorteil von pre-order gegenüber post-order?

g) Nehmen Sie den binären Baum aus dem Kapitel 2.3 und traversieren Sie ihn mit pre-order, in-order und post-order. Notieren Sie sich die Schlüssel der Reihe nach.

In diesem Kapitel haben wir 4 Verfahren kennen gelernt, wie man einen binären Baum traversieren kann. Wir haben auch gesehen, dass uns ein binärer Baum für die effiziente Suche nicht viel bringt... doch das ändert sich im nächsten Kapitel!

2.5 Der binäre Suchbaum

Ein **binärer Suchbaum** ist genau gleich aufgebaut wie ein binärer Baum. Der Unterschied ist, dass der binäre Suchbaum **sortiert** ist. Die folgenden 2 **Regeln** gelten immer:

1. Der Wert des linken Kindknotens ist immer kleiner als der Wert des Knotens selbst
2. Der Wert des rechten Kindknotens ist immer grösser als der Wert des Knotens selbst

Beim Speichern der Daten im binären Suchbaum müssen wir also darauf achten, dass wir diese beiden Regeln einhalten. Dies bedeutet zwar etwas mehr Aufwand beim Speichern, dafür ist die Suche umso schneller. Beim Speichern dürfen wir das Datenelement nicht mehr "irgendwo" ablegen, sondern müssen zuerst die richtige Position im binären Baum suchen, an der wir das Datenelement abspeichern dürfen:

1. Korrekte Position im binären Suchbaum **suchen**
2. Datenelement **abspeichern**

Nachfolgend ist ein binärer Suchbaum abgebildet, welcher wiederum die Schlüssel der Tabelle SIMPSONS aus dem ersten Kapitel speichert. Allerdings entspricht die Reihenfolge der Knoten nun den Regeln des binären Suchbaumes. Als Wurzel (rot) wurde der Schlüssel mit der ID=58 verwendet:

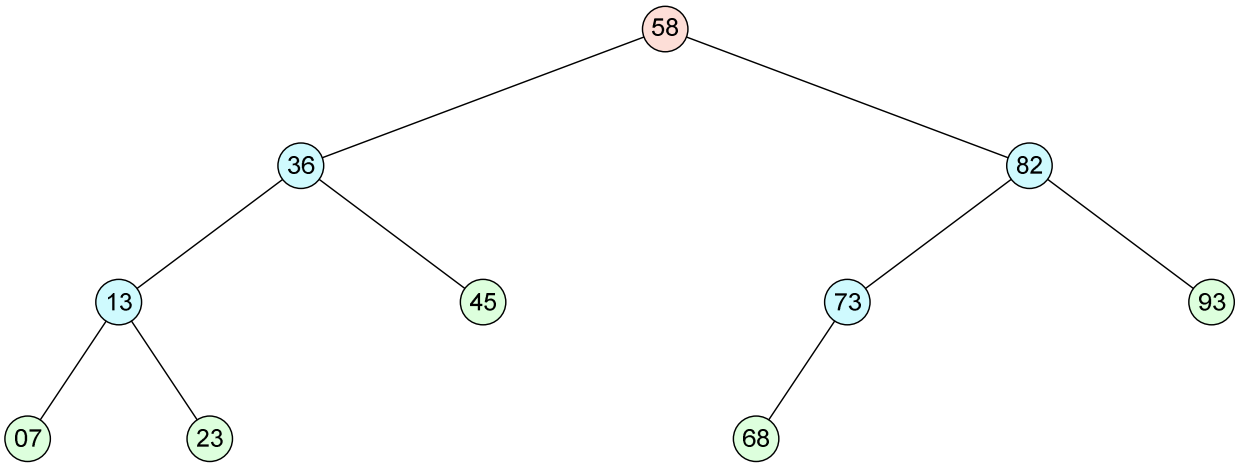


Abbildung 2.7: Binärer Baum in Suchreihenfolge

Legende: rot = Wurzel blau = innere Knoten grün = Blattknoten

Wissenssicherung: Beantworten Sie die folgenden Fragen

h) Wie hoch (resp. tief) kann ein binärer (Such-)Baum, welcher 15 Knoten enthält, im Maximum werden?

i) Wie hoch (resp. tief) kann ein binärer (Such-)Baum mit 15 Knoten im Minimum sein? Wie lautet die Antwort, wenn der Baum 32'000 Knoten enthält?

j) Nehmen Sie ein Blatt Papier, und zeichnen Sie einen binären Suchbaum. Als Wurzel verwenden Sie ID=45. Fügen Sie die Schlüssel in der folgenden Reihenfolge in den Suchbaum ein: 93, 58, 13, 73, 07, 36, 68, 23, 82. Was stellen Sie fest?

k) Welches Element verwenden Sie in einem binären Suchbaum am besten als Wurzel? Weshalb?

2.6 Suchen im binären Suchbaum

Wie wir im Kapitel 2.5 gesehen haben, ist der binäre Suchbaum sortiert: der linke Teilbaum eines Knotens enthält nur kleinere Schlüssel als der Knoten selbst, der rechte Teilbaum nur grössere. Damit wir schön der Reihe nach suchen können, drängt sich die Variante b) der Tiefensuche aus dem letzten Kapitel auf: die Suche in symmetrischer Reihenfolge (L-N-R), auch **in-order** Suche genannt. L-N-R sagt eigentlich schon alles: zuerst wird der **L**inke Teilbaum durchsucht, danach der **N**ode (Knoten) selbst, und danach der **R**echte Teilbaum. Weil der binäre Suchbaum sortiert ist, haben wir somit Gewähr, dass die Suche schön der Reihe nach von Statten geht.

Die Suche in einem sortierten binären Baum – dem binären Suchbaum – ist wesentlich schneller als die Suche in einem nicht sortierten binären Baum. Wenn wir in einem Knoten entscheiden, dass wir in einen Teilbaum absteigen müssen, können wir bereits sicher sein, dass das Suchresultat sich nicht im anderen Teilbaum befindet. Somit können wir den anderen Teilbaum komplett ignorieren. Damit halbieren wir in jedem Knoten die Anzahl der zu überprüfenden Knoten.

Dieses Verfahren kommt uns doch irgendwie bekannt vor, oder? Im ersten Kapitel hatten wir in der Tabelle SIMPSONS gesucht, indem wir das Suchintervall immer halbiert hatten. Genau diese Methode kommt auch im binären Suchbaum zum Einsatz. Somit sind die Formeln aus dem ersten Kapitel auch für den binären Suchbaum gültig.

Wissenssicherung: Beantworten Sie die folgende Frage

l) Suchen Sie im binären Suchbaum aus Kapitel 2.5 nach dem Knoten mit der ID=68. Notieren Sie sich die einzelnen Schritte. Wie viele Suchschritte brauchen Sie?

Da ein binärer Baum eigentlich nur etwas nützt, wenn er sortiert ist, verwendet man in der Praxis vorwiegend binäre Suchbäume oder Weiterentwicklungen des binären Suchbaumes. Im nächsten Kapitel werden wir so eine Weiterentwicklung kennen lernen.

2.7 Probleme: Borkenkäfer im Wald

In der Aufgabe h) hatten wir festgestellt, dass es Probleme geben kann, wenn wir die Knoten sortiert in einen binären (Such-)Baum einfügen. So kann es passieren, dass aus einem binären Baum auf einmal eine lineare Liste wird. Das gleiche Problem kann entstehen, wenn wir auf einem binären Baum viele Einfüge- oder Löschoperationen ausführen.

Diese ungewollte Umwandlung eines binären Baumes in eine lineare Liste nennt man **Entartung**: der Baum verliert die Eigenschaften seiner ursprünglichen Art. Man sagt auch, dass der Baum **degeneriert**.

Wie wir das Degenerieren verhindern können, werden wir im nächsten Kapitel sehen.

Eine weitere wichtige Thematik bei binären Bäumen ist das **Balancieren**. Wenn wir die Knoten in einer unglücklichen Reihenfolge in den Baum einfügen, kann es passieren, dass ein Teilbaum eines Knotens viel grösser ist als der andere Teilbaum. Je höher im Baum dieses Problem auftritt, desto schwerwiegender ist es. So kann es sein, dass ein binärer Baum wie eine kranke Tanne im Wald aussieht: auf der einen Seite hat sie viele Äste, auf der anderen Seite hat der Borkenkäfer alles verfressen. Man sagt auch, dass der Baum nicht mehr balan-

ciert ist. Dieses Problem kann auch nach häufigen Einfüge- und Löschoperationen auftreten. Das ist sozusagen der Anfang vom Ende, resp. der Anfang des Degenerierens.

Wie wir einen binären Baum in Balance halten können, werden wir im nächsten Kapitel sehen.

2.8 Zusammenfassung

In diesem Kapitel haben Sie gelernt, was ein binärer Baum ist. Wir haben gesehen, dass ein binärer Baum uns an sich noch nicht viel nützt, und dass es dafür den binären Suchbaum gibt. Sie haben 4 verschiedene Methoden für die Suche im binären Suchbaum kennen gelernt. Zudem sind für Sie die Kenngrössen eines binären Baumes keine Fremdwörter mehr, z.B. Tiefe, Anzahl Knoten, Füllgrad, pre-order, in-order oder post-order. Sie wissen nun auch, was balancieren bedeutet, und dass Entartung nichts Abartiges ist – jedoch unerwünscht.

Im nächsten Kapitel nehmen wir eine Weiterentwicklung des binären Baumes unter die Lupe, welche verschiedene Vorteile bietet, und zudem die erwähnten Probleme des Balancierens und der Entartung lösen soll.

2.9 Lösungen zu den Wissenssicherungsfragen

a) *Zeigen Sie, wie Sie im obigen binären Baum die Baumhöhe ermitteln.*

Der binäre Baum hat die **Höhe h=4**. Die Höhe ist durch den Blattknoten gegeben, welcher sich am weitesten von der Wurzel entfernt befindet, d.h. in der grössten Tiefe. In diesem binären Baum handelt es sich um den Knoten mit der ID=36. Von der Wurzel her gelangt man mit 4 Schritten zu diesem Knoten, über die Knoten mit den IDs 82, 68 und 73.

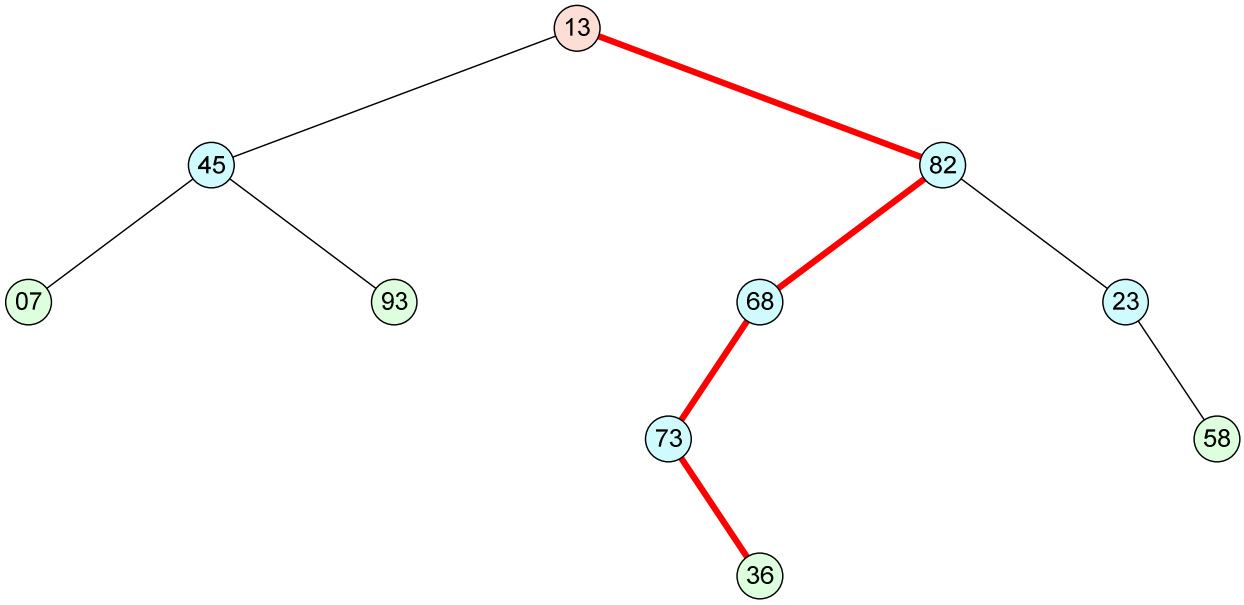


Abbildung 2.8: Baumhöhe (Lösung)

b) *Wie viele Blätter hat ein binärer Baum der Höhe h=3 im Maximum?*

Grafisch: Man kann sich einen binären Baum mit der Höhe h=3 aufzeichnen und die Blattknoten zählen. In der unten stehenden Abbildung entspricht die Lösung der Summe der grünen Blattknoten: **8 Blätter**.

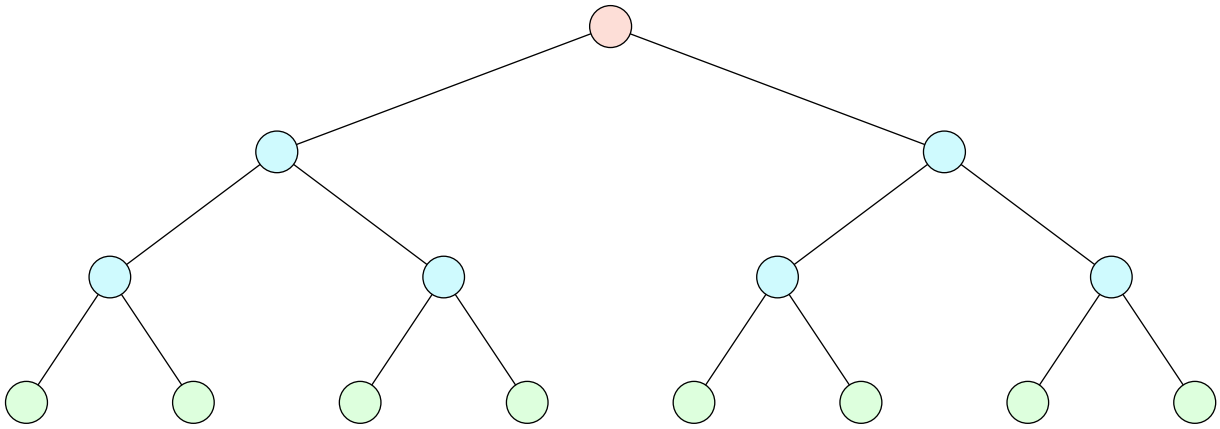


Abbildung 2.9: Anzahl Blätter (grafische Lösung)

Berechnung: Bei binären Bäumen mit einer grossen Höhe, z.B. h=5, ist das Aufzeichnen nicht mehr so einfach. Bei noch grösseren Bäumen, z.B. h=50, wird das Aufzeichnen ein Ding der Unmöglichkeit. Doch zum Glück können wir die Anzahl Blattknoten auch berechnen:

$$\text{Anzahl Blattknoten} = 2^{\text{Baumhöhe}} = 2^3 = \textbf{8 Blätter}$$

Weil es sich um einen binären Baum handelt, verwenden wir die Basis 2. Für einen ternären oder quaternären Baum würden wir entsprechend die Basis 3 oder 4 verwenden.

c) *Wie viele Knoten (Wurzel, innere Knoten und Blattknoten) hat ein binärer Baum der Höhe h=3 im Maximum?*

Grafisch: Man kann sich einen binären Baum mit der Höhe h=3 aufzeichnen und die Knoten zählen. In der oben stehenden Abbildung entspricht die Lösung der Summe aller Knoten (rot, blau und grün): **15 Knoten**.

Berechnung: Die Summe aller Knoten im Baum entspricht der Summe aller Knoten einer bestimmten Tiefe:

$$\text{Anzahl Knoten} = 2^0 + 2^1 + 2^2 + \ldots + 2^{\text{Baumhöhe}} = 2^{\text{Baumhöhe}+1} - 1$$

$$\text{Anzahl Knoten} = 2^{3+1} - 1 = 2^4 - 1 = \textbf{15 Knoten}$$

d) *Wie würden Sie im obigen binären Baum nach der ID=36 suchen? Beschreiben Sie eine Variante.*

Da der binäre Baum nicht sortiert ist, gibt es kein effizientes Verfahren für die Suche. Man muss den ganzen Baum nach der ID=36 absuchen. Dazu sagt man auch, man muss den Baum **traversieren**. Dafür gibt es verschiedene Vorgehensweisen, welche Sie im Kapitel 2.4 kennen lernen werden.

e) Wir haben 3 Varianten für die Tiefensuche kennen gelernt. Gibt es weitere Varianten?

Man kann die Reihenfolge des Absteigens ändern:

- N-L-R $\rightarrow$ **N-R-L**
- L-N-R $\rightarrow$ **R-N-L**
- L-R-N $\rightarrow$ **R-L-N**

Statt zuerst in den linken Teilbaum abzusteigen, steigt man zuerst in den rechten Teilbaum ab. Wenn dann der rechte Teilbaum an der Reihe wäre, steigt man in den linken Teilbaum ab. Vom Prinzip her ändert sich nichts.

f) Was ist der Vorteil von pre-order gegenüber post-order?

Bei pre-order wird der Knoten N am Anfang verglichen, bei post-order findet der Vergleich erst am Schluss statt. Wenn N das gewünschte Suchresultat ist, dann kommt pre-order schneller zum Ziel. Zudem spart sich der Computer das Absteigen in einen möglicherweise grossen Teilbaum. was die Suche schneller macht.

g) Nehmen Sie den binären Baum aus dem Kapitel 2.3 und traversieren Sie ihn mit pre-order, in-order und post-order. Notieren Sie sich die Schlüssel der Reihe nach.

pre-order: 13, 45, 07, 93, 82, 68, 73, 36, 23, 58

in-order: 07, 45, 93, 13, 73, 36, 68, 82, 23, 58

post-order: 07, 93, 45, 36, 73, 68, 58, 23, 82, 13

Wie Sie sehen, kommt bei jedem Verfahren eine andere Reihenfolge heraus. Der Vollständigkeit halber sei hier noch die Lösung des level-order Verfahrens aufgeführt:

level-order: 13, 45, 82, 07, 93, 68, 23, 73, 58, 36

h) *Wie hoch (resp. tief) kann ein binärer (Such-)Baum, welcher 15 Knoten enthält, im Maximum werden?*

Der schlimmste Fall tritt ein, wenn die Elemente vor dem Einfügen in den Baum bereits sortiert sind, und sie dann der Reihe nach eingefügt werden:

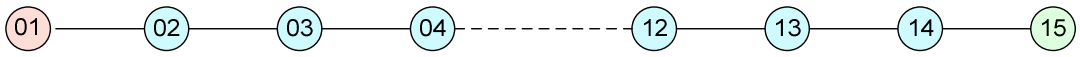


Abbildung 2.10: Binärer Baum als lineare Liste

In diesem Fall verkommt der binäre Suchbaum zu einer linearen Liste. Achtung: auch die lineare Liste ist per Definition ein binärer (Such-)Baum!

i) *Wie hoch (resp. tief) kann ein binärer (Such-)Baum mit 15 Knoten im Minimum sein? Wie lautet die Antwort, wenn der Baum 32'000 Knoten enthält?*

Die minimale Baumhöhe ist auf jeden Fall dann erreicht, wenn der Baum voll besetzt ist. Voll besetzt heisst, dass bis zur maximalen Tiefe des Baumes alle Knoten vorhanden sind. Da es sich um einen binären Baum handelt, können wir die Baumhöhe mit dem 2er-Logarithmus berechnen:

$$\text{Anzahl Ebenen} = \log_2(\text{Anz. Knoten} + 1) = \frac{\ln 16}{\ln 2} = \frac{\log 16}{\log 2} = 4$$

$$\text{Minimale Baumtiefe} = \text{Anzahl Ebenen} - 1 = 4 - 1 = 3$$

Oder für 32'000 Knoten:

$$\text{Anzahl Ebenen} = \log_2(\text{Anz. Knoten} + 1) = \frac{\ln 32'001}{\ln 2} = \frac{\log 32'001}{\log 2} = 14.96$$

$$\text{Minimale Baumtiefe} = \text{Anzahl Ebenen} - 1 = 14.96 - 1 = 13.96$$

Die minimale Baumtiefe (d.h. die Baumhöhe) beträgt 14 – weniger tief geht nicht.

j) *Nehmen Sie ein Blatt Papier, und zeichnen Sie einen binären Suchbaum. Als Wurzel verwenden Sie ID=45. Fügen Sie die Schlüssel in der folgenden Reihenfolge in den Suchbaum ein: 93, 58, 13, 73, 07, 36, 68, 23, 82. Was stellen Sie fest?*

Wir können feststellen, dass der rechte Teilbaum tiefer ist, als er eigentlich sein müsste. Obwohl der Baum nur 10 Knoten enthält, hat er eine Tiefe von t=4, d.h. maximal 2^4=16 Blattknoten. Somit hätte der Baum Platz für 2^{4+1}-1=31 Knoten. Das ergibt eine Ausnüt-

zung von 10/31 = 0.3225, d.h. **32.25%**. Man nennt dies auch den **Füllgrad** des Baumes. Der Platz im Baum wird nicht effizient genutzt.

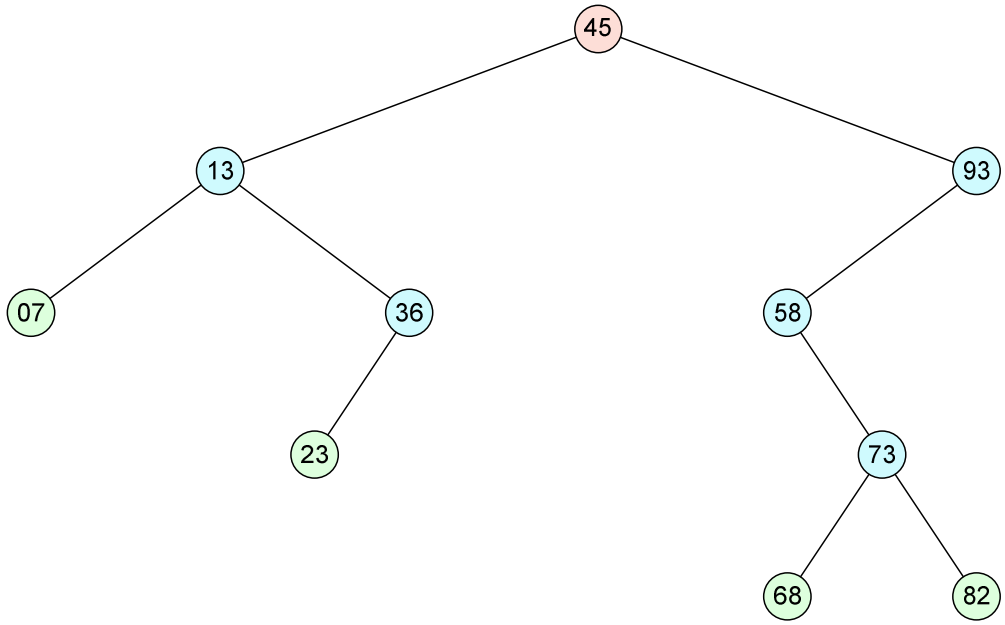


Abbildung 2.11: Unbalancierter binärer Baum (Lösung)

k) *Welches Element verwenden Sie in einem binären Suchbaum am besten als Wurzel? Weshalb?*

Das kommt darauf an, ob man vor dem Einfügen in den Baum weiss, wie viele Knoten einzufügen sind.

Wenn man die Anzahl einzufügender Knoten kennt, sortiert man die Knoten am besten vor dem Einfügen. Dann wählt man den Knoten in der Mitte als Wurzel. Nun kann man die restlichen Knoten in beliebiger Reihenfolge in den Baum einfügen. Aber Achtung: man muss aufpassen, dass nicht das Problem aus Aufgabe h) auftritt und der Baum zu einer linearen Liste verkommt! Genau das würde passieren, wenn wir die restlichen Knoten vorsortiert in den Baum einfügen. Diese Beschreibung ist demnach nur gut, um einen Wurzelknoten zu bestimmen. Das Einfügen sollte zufällig erfolgen.

Wenn wir die Anzahl einzufügender Knoten nicht kennen, spielt die Wahl des Wurzelknotens nicht so eine grosse Rolle, da der Baum mit zunehmender Anzahl Einfüge- und Löschoperationen **degenerieren** wird. Degenerieren bedeutet z.B., dass der binäre Baum zu einer linearen Liste verkommt. Wie wir das verhindern können, werden Sie im Kapitel 2.7 sehen.

l) *Suchen Sie im binären Suchbaum aus Kapitel 2.5 nach dem Knoten mit der ID=68. Notieren Sie sich die einzelnen Schritte. Wie viele Suchschritte brauchen Sie?*

1. Knoten (Wurzel):	68 > 58	→ weiter im rechten Teilbaum mit ID=82
2. Knoten:	68 < 82	→ weiter im linken Teilbaum mit ID=73
3. Knoten:	68 < 73	→ weiter im linken Teilbaum mit ID=68
4. Knoten:	68 = 68	→ Resultat nach 4 Vergleichen gefunden, d.h. nach **3 Suchschritten**

2.10 Lernkontrolle: Hier können Sie überprüfen, ob Sie das Kapitel beherrschen

Aufgabe 1

Gegeben ist der folgende binäre Suchbaum, welcher 15 Knoten enthält. Wie hoch ist der Baum? Wandeln Sie diesen Baum in einen neuen binären Suchbaum um. Der neue Suchbaum soll nur so hoch wie wirklich nötig sein. Zeigen Sie auf, wie Sie vorgehen.

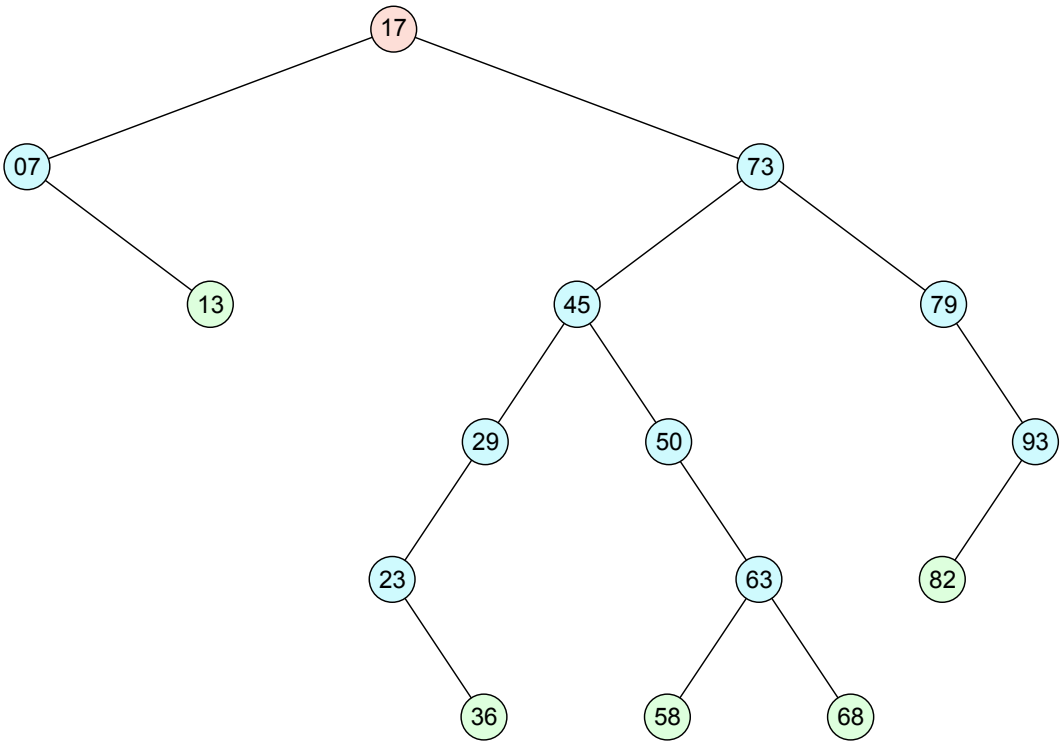


Abbildung 2.12: Binären Suchbaum balancieren

Aufgabe 2

Bestimmen Sie den Füllgrad des binären Suchbaumes aus Aufgabe 1.

Aufgabe 3

Ein binärer Baum der Höhe h=7 sei voll besetzt. Wie viele Knoten enthält der Baum? Wie viele Knoten befinden sich in der Tiefe t=5?

Aufgabe 4

Aus dem folgenden binären Suchbaum wollen Sie den Knoten mit der ID=80 löschen. Wie machen Sie das?

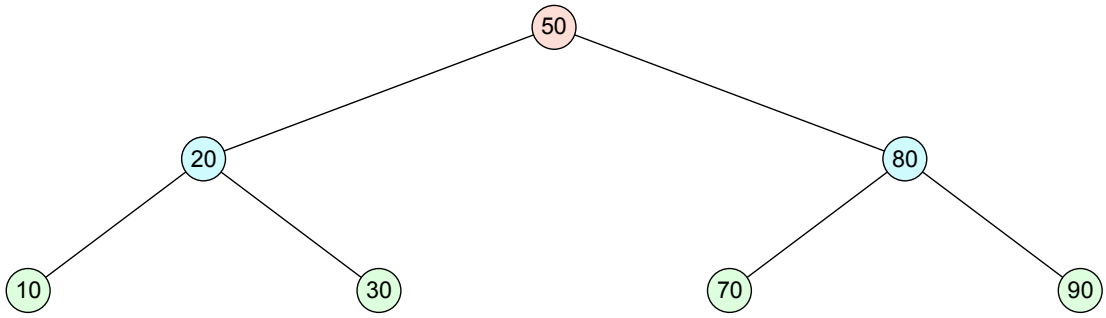


Abbildung 2.13: Löschen aus binärem Baum

2.11 Lösungen zu den Lernkontrollfragen

Aufgabe 1

Gegeben ist der folgende binäre Suchbaum, welcher 15 Knoten enthält. Wie hoch ist der Baum? Wandeln Sie diesen Baum in einen neuen binären Suchbaum um. Der neue Suchbaum soll nur so hoch wie wirklich nötig sein. Zeigen Sie auf, wie Sie vorgehen.

Der Baum hat die Höhe **h=5**.

Schritt 1: Bestimmen Sie die Höhe des neuen Suchbaumes, und zeichnen Sie den vorerst leeren Baum auf. Der gegebene Suchbaum hat 15 Knoten. Daraus können wir die minimale Tiefe berechnen:

Minimale Tiefe = $\log_2$ (Anzahl Knoten + 1) = $\log_2$ (15 + 1) = **4 Ebenen**

Schritt 2: Schreiben Sie sich die Werte des alten Suchbaumes der Reihe nach auf:

Reihenfolge: 07, 13, 17, 23, 29, 36, 45, 50, 58, 63, 68, 73, 79, 82, 93

Schrit 3: Bestimmen Sie die neue Wurzel. Damit der binäre Suchbaum am Schluss balanciert da steht, verwenden wir das Element in der Mitte der Liste als neue Wurzel: **50**. So ist gewährleistet, dass jeder Teilbaum links und rechts von der Wurzel gleich gross ist, d.h. gleich viele Knoten enthält.

Schritt 4: Füllen Sie nun die Werte in den neuen binären Suchbaum ein, welchen Sie zuvor aufgezeichnet haben. Dabei gehen Sie so vor, dass Sie das Intervall links und rechts von der neuen Wurzel (50) fortlaufend halbieren, und dann jeweils die mittleren Elemente als Kindknoten wählen:

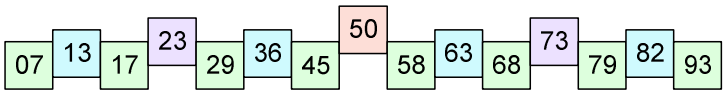


Abbildung 2.14: Schlüssel auf Knoten aufteilen (Lösung)

Danach sieht der binäre Suchbaum so aus:

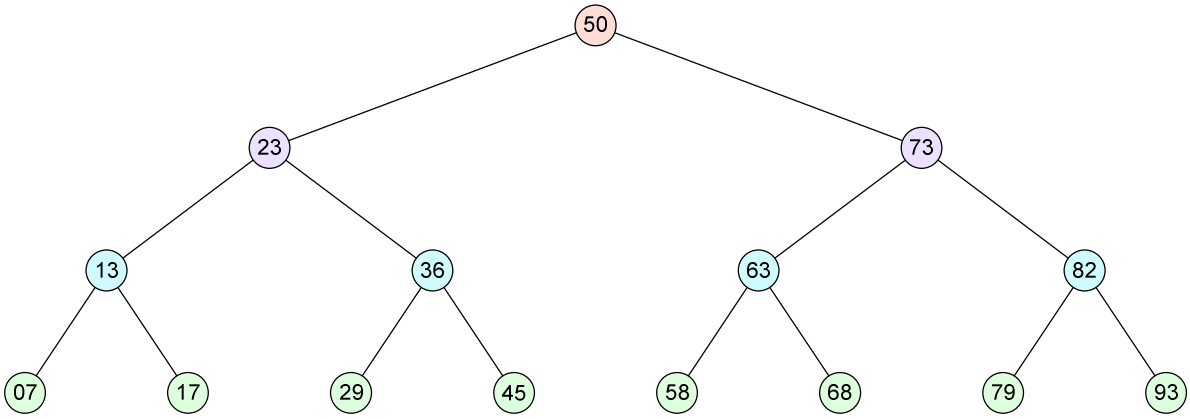


Abbildung 2.15: Neuer balancierter Suchbaum (Lösung)

Aufgabe 2

Bestimmen Sie den Füllgrad des binären Suchbaumes aus Aufgabe 1.

Füllgrad = Anzahl Knoten / Max. Anzahl Knoten = 15 / $(2^6 - 1)$ = 15 / 63 = 0.238 → **23.8%**

Aufgabe 3

Ein binärer Baum der Höhe h=7 sei voll besetzt. Wie viele Knoten enthält der Baum? Wie viele Knoten befinden sich in der Tiefe t=5?

Anzahl Knoten = $2^{\text{Baumhöhe}+1} - 1 = 2^{7+1} - 1 = 2^8 - 1 =$ **255 Knoten**

Anzahl Knoten auf Tiefe t=5: $2^{\text{Tiefe}} = 2^5 =$ **32 Knoten**

Aufgabe 4

Aus dem folgenden binären Suchbaum wollen Sie den Knoten mit der ID=80 löschen. Wie machen Sie das?

Man löscht den Knoten mit der ID=80, und fügt an dessen Stelle eines seiner Kindknoten ein. Es gibt also 2 Varianten.

Variante 1: ID=90 wählen

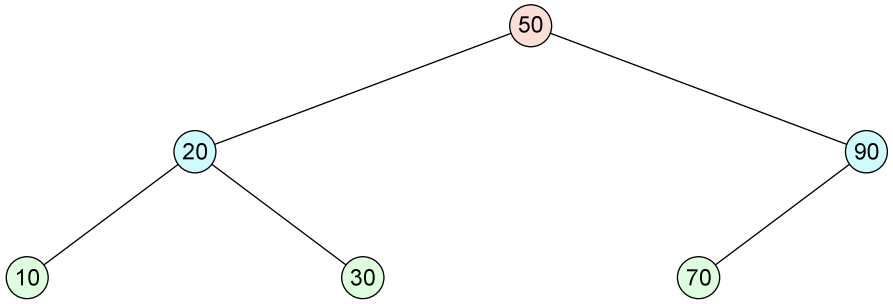


Abbildung 2.16: Löschen aus binärem Suchbaum (1/2)

Variante 2: ID=70 wählen

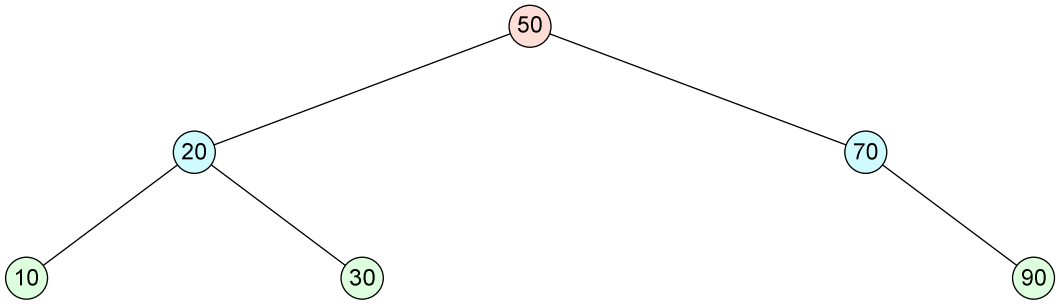


Abbildung 2.17: Löschen aus binärem Suchbaum (2/2)

Problematisch wird es dann, wenn z.B. der Knoten mit der ID=90 auch wieder Kinder hat. Dann hätten wir auf einmal einen Knoten mit 3 Kindern, was im binären Baum natürlich nicht geht. Überlegen Sie, was man da tun könnte. Lösung: Kapitel 3.

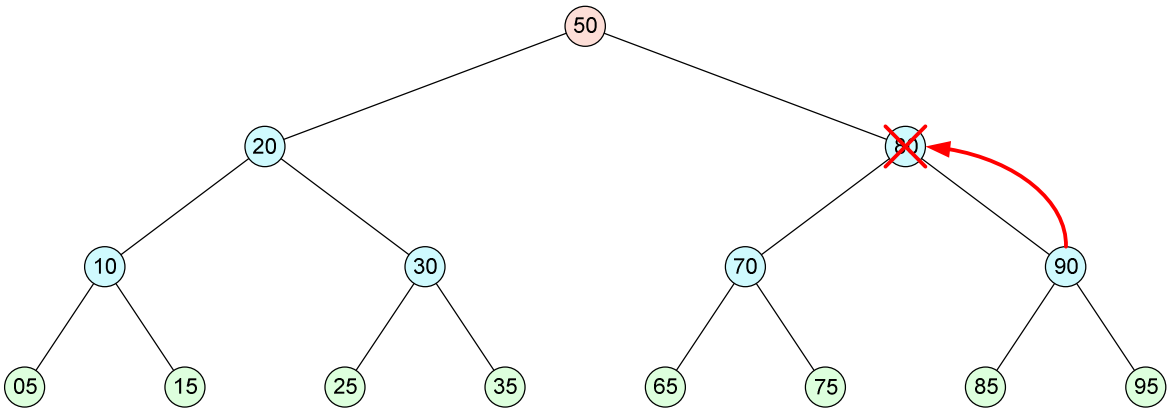


Abbildung 2.18: Probleme beim Löschen

Kapitel 3: B-Baum

3.1 Übersicht: Worum geht es?

Im letzten Kapitel haben wir die binären (Such-)Bäume kennen gelernt, und dabei gesehen, dass diese ein paar Unschönheiten aufweisen: Verlust der Balance und die damit verbundene Entartung. In diesem Kapitel werden wir sehen, wie der B-Baum als Weiterentwicklung der binären Bäume mit diesen Mängeln umgehen kann. Wir werden sehen, weshalb der B-Baum im Datenbank-Umfeld und in der Informatik ganz generell sehr oft zum Einsatz kommt.

3.2 Lernziele

In diesem Kapitel lernen Sie folgendes:

- Sie wissen, was ein B-Baum ist
- Sie kennen die Vorteile eines B-Baumes gegenüber den binären Bäumen
- Sie wissen, weshalb B-Bäume in der Praxis oft eingesetzt werden
- Sie kennen die wichtigsten Kenngrössen eines B-Baumes
- Sie kennen ein paar Weiterentwicklungen des B-Baumes

3.3 Der B-Baum

Der B-Baum (Englisch: **B-Tree**) wurde bereits 1972 von Rudolf Bayer und Edward M. McCreight entwickelt. Er ist eine Weiterentwicklung des binären Suchbaumes. Allerdings ist nicht klar, woher der B-Baum seinen Namen hat. Die häufigste Erklärung ist, dass das "B" für **balanciert** steht. Andere Spekulationen gehen dahin, dass das "B" für Bayer steht (den Entwickler), oder für Barbara (die Frau des Entwicklers), oder für Boeing (das war der Arbeitgeber von Rudolf Bayer). Da der B-Baum die angenehme Eigenschaft hat, dass er immer balanciert ist, können wir davon ausgehen, dass der erste Erklärungsversuch am meisten Sinn macht – obwohl auch die anderen Erklärungen möglich wären. Wir wissen es schlicht und einfach nicht.

Nachfolgend sehen Sie einen B-Baum mit der Höhe h=2, welcher voll besetzt ist:

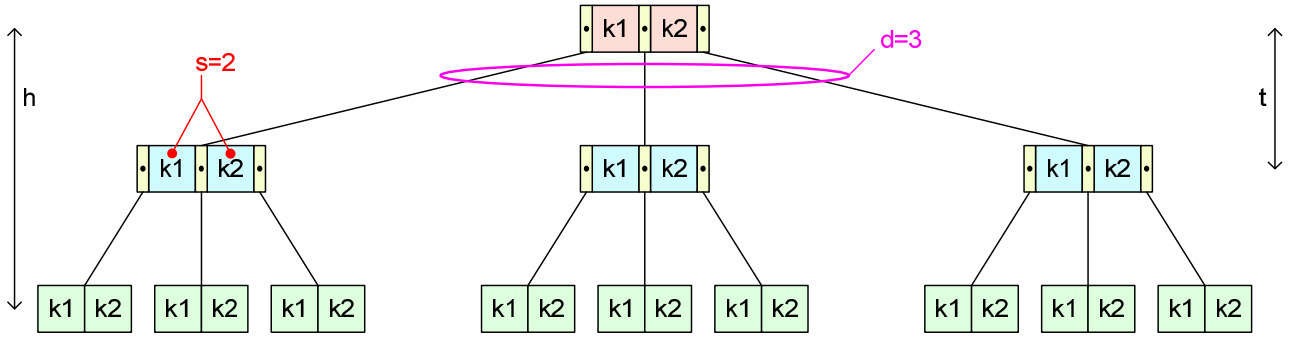


Abbildung 3.1: Voll besetzter B-Baum

Legende:	rot = Wurzel	gelb = Verweise auf Kinder	kx = Schlüssel (key)
	blau = innere Knoten	h = Höhe	s = Anz. Schlüssel (size)
	grün = Blattknoten	t = Tiefe	d = Anz. Kinder (Ordnung)

Im Gegensatz zum binären Baum speichern die Knoten eines B-Baumes mehr als nur einen Schlüssel. Im obigen Beispiel beträgt die Anzahl Schlüssel **s=2**. Die Schlüssel in einem Knoten sind immer aufsteigend sortiert: **k1 < k2 < ... < kn**. Zudem können die Knoten mehr als nur 2 Kinder haben. Die maximale Anzahl Kinder eines Knotens im Baum entspricht der **Ordnung** des Baumes. Beim binären Baum beträgt die Ordnung 2, der oben abgebildete B-Baum hat die Ordnung **d=3**.

Wie Sie sehen können, hat der B-Baum gegenüber dem binären (Such-)Baum ein paar weitere Regeln, die es zu beachten gilt:

Regeln für Knoten

1. Ein Knoten kann mehr als einen Schlüssel (**key k**) speichern. Die maximale Anzahl Schlüssel in einem Knoten nennen wir **s** (**size**).
2. Die Schlüssel in einem Knoten sind immer aufsteigend **sortiert**: k1 < k2 < ... < kn
3. Die Wurzel (rot) kann zwischen **0** Schlüssel (dann ist der B-Baum leer) und **d-1** Schlüssel enthalten.
4. Die inneren Knoten (blau) und die Blattknoten (grün) sind immer **mindestens halb voll**: Sie enthalten mindestens **⌈d/2⌉ – 1** und im Maximum **d-1** Schlüssel.
5. Sämtliche Blattknoten (grün) befinden sich immer in der **gleichen Tiefe t**. Deshalb entspricht die Tiefe der Blattknoten gleich der **Höhe h** des B-Baumes.

Regeln für Kinder

6. Ein Knoten kann mehr als 2 Kinder haben. Die maximale Anzahl Kinder eines Knotens nennen wir die **Ordnung** des Baumes, und kürzen sie mit **d** (**density**) ab. Die Ordnung wird oftmals auch als **Verzweigungsgrad** bezeichnet.
7. Die Wurzel (rot) hat entweder gar keine Kinder, oder aber mindestens **2** und im Maximum **d = s + 1** Kinder.
8. Die inneren Knoten (blau) haben im Minimum **⌈d/2⌉** Kinder, und im Maximum **s + 1** Kinder. Die Verweise auf Kinder werden auch als **Pointer** (Zeiger) bezeichnet.

Diese Regeln erscheinen auf den ersten Blick etwas kompliziert, und machen für Sie im Moment vermutlich noch nicht viel Sinn. So kompliziert ist es aber gar nicht! Auf den folgenden Seiten werden Sie die Regeln noch genauer kennen lernen, so dass Sie auch ihren Sinn erkennen werden.

Im B-Baum sind die folgenden Zahlen wichtig, wir werden ihnen immer wieder begegnen:

$\lceil d/2 \rceil$	minimale Anzahl Kinder Ausnahme: die Wurzel hat keine oder mindestens 2 Kinder
d	maximale Anzahl Kinder
$\lceil d/2 \rceil - 1$	minimale Anzahl Schlüssel Ausnahme: die Wurzel kann auch weniger Schlüssel haben
$d - 1$	maximale Anzahl Schlüssel

Die Schreibweise $\lceil d/2 \rceil$ sagt aus, dass wir das Resultat von $d/2$ nach der Berechnung aufrunden werden, falls das Resultat keine ganze Zahl ist.

Um die Zahlen noch etwas zu verdeutlichen, sind sie in der folgenden Grafik nochmals dargestellt:

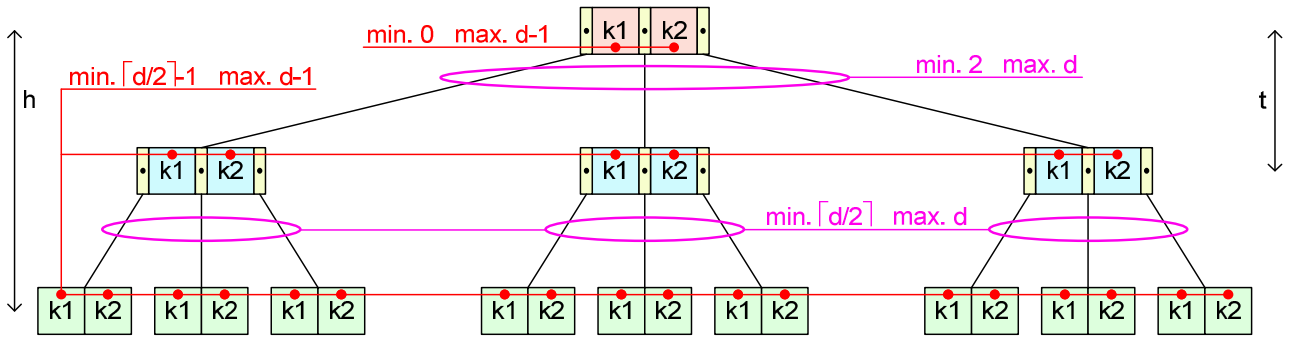


Abbildung 3.2: Kenngrössen des B-Baumes

Beachten Sie, dass die Blattknoten keine Zeiger (Pointer) beinhalten. Dies ist auch nicht nötig, da Blattknoten ja per Definition keine Kinder haben.

Wissenssicherung: Beantworten Sie die folgenden Fragen

a) Gegeben ist ein B-Baum mit der Ordnung **d=5** und der Höhe **h=2**:

1. Wie viele Schlüssel enthält die Wurzel im Minimum / im Maximum?
2. Wie viele Schlüssel enthält ein innerer Knoten im Minimum / im Maximum?
3. Wie viele Schlüssel enthält ein Blattknoten im Minimum / im Maximum?
4. Wie viele Kindknoten hat die Wurzel im Minimum / im Maximum?
5. Wie viele Kindknoten hat ein innerer Knoten im Minimum / im Maximum?
6. Wie viele Kindknoten hat ein Blattknoten im Minimum / im Maximum?

3.4 Der B-Baum im Detail

Nun wollen wir die Regeln des B-Baumes etwas genauer unter die Lupe nehmen.

Regel: Knotengrösse

Der B-Baum wurde speziell für Informatiksysteme optimiert, d.h. Computer, Datenbanken, etc. Wie wir wissen, bearbeiten die meisten Computer die Daten **blockorientiert**. Wenn man z.B. 1 Byte von der Harddisk einlesen möchte, dann geht das nicht so einfach: man muss den ganzen Datenblock einlesen, welcher das gewünschte Byte enthält. Erst kann man auf das Byte zugreifen. Den Rest des Datenblockes, den wir vermutlich gar nicht brauchen, müssen wir dann wegwerfen. Datenblöcke haben in der Regel eine Grösse von 512, 1'042, 2'048 oder 4'096 Bytes. Bei grossen Informatiksystemen können die Datenblöcke eine noch grössere 2er-Potenz einnehmen, z.B. 2^{13} = 8'192 Bytes oder gar 2^{16} = 65'536 Bytes. Das ist der Grund, weshalb das Lesen und Schreiben von Daten von und auf die Harddisk relativ langsam ist, verglichen mit ähnlichen Lese- und Schreiboperationen im Memory (Hauptspeicher). Es spielt von der Geschwindigkeit her keine Rolle, ob wir 1 Byte oder 512 Bytes von der Harddisk lesen – wir müssen so oder so den ganzen Datenblock lesen.

Genau hier setzt die Idee der B-Bäume an! Wenn wir es so arrangieren können, dass die Grösse eines Knotens genau so gross ist wie die Grösse eines Datenblockes, dann verlieren wir nichts – im Gegenteil. Wir können dann mit einem einzigen Lesebefehl einen ganzen Knoten des B-Baumes von der Harddisk in den Hauptspeicher einlesen. So haben wir relativ schnell Zugriff auf relativ viele Schlüssel.

Regel: Knoten sortieren

Die Schlüssel in einem Knoten werden aufsteigend sortiert. Das hat zur Folge, dass jeder Teilbaum nur Schlüssel in einem bestimmten Wertebereich enthält:

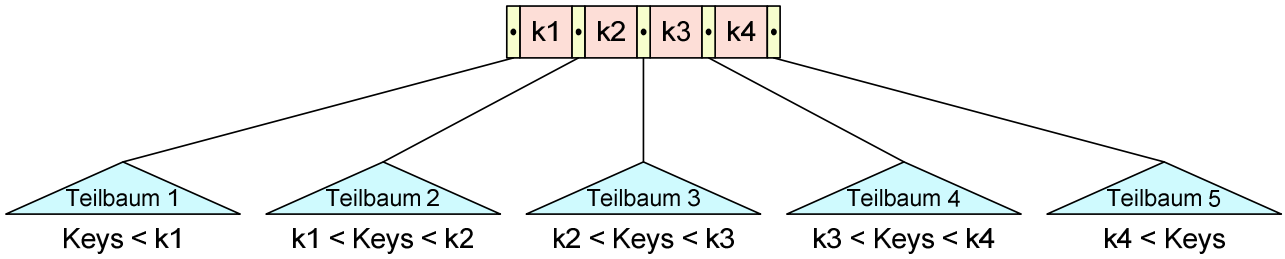


Abbildung 3.3: Trennschlüssel und Aufteilung in Teilbäume

So enthält der Teilbaum 1 nur Schlüssel die kleiner sind als der Schlüssel k1. Der Teilbaum 2 hingegen enthält nur Schlüssel, die grösser sind als der Schlüssel k1. Der Schlüssel k1 selbst ist im übergeordneten Knoten gespeichert. Er wird auch als **Trennschlüssel** bezeichnet, da er die beiden Teilbäume voneinander trennt. Wenn diese Regel im ganzen B-Baum eingehalten wird, ist der Baum sortiert.

Die Eigenschaft, dass die Schlüssel nicht nur in der Baumhierarchie sortiert sind, sondern auch in den einzelnen Knoten, erlaubt uns eine effiziente Suche oder Traversierung. Innerhalb eines Knotens verwenden wir die **Breitensuche**, welche wir im letzten Kapitel kennen gelernt hatten. Die Breitensuche entspricht der Suche in einer linearen, sortierten Liste. Da die Liste sortiert ist, können wir die Methode der Halbierung der Suchintervalle aus Kapitel 1 verwenden. Erst wenn wir feststellen, dass der Knoten den gesuchten Schlüssel nicht ent-

hält, behelfen wir uns mit der **Tiefensuche** und steigen zum entsprechenden Kindknoten ab. Den zu verwendenden Pointer finden wir mit Hilfe der obigen Darstellung.

Die **Traversierung** eines B-Baumes läuft wie folgt ab: zuerst steigen wir in der Hierarchie so tief wie möglich ab, d.h. bis zum Blattknoten ganz links (in T1). Nachdem wir alle Schlüssel des Blattknotens inspiziert haben, kommen wir einen Level zurück, und besuchen den Schlüssel K1. Danach steigen wir in den nächsten Teilbaum (T2) ab, und so weiter. Wenn wir alle Teilbäume besucht haben, können wir wieder einen Level aufsteigen. Dies geht so lange weiter, bis wir letzten Endes wieder bei der Wurzel angelangt sind. Da wir dort nicht mehr weiter aufsteigen können, ist die Traversierung beendet.

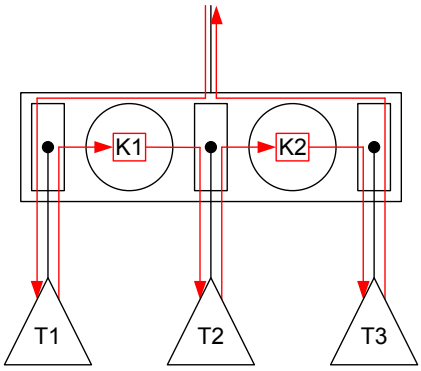


Abbildung 3.4: Traversierung eines B-Baumes

Regel: Ordnung und Anzahl Kinder resp. Anzahl Pointer

Ein Knoten eines B-Baumes kann so viele Kinder haben, wie er Schlüssel enthält – plus noch eines mehr. Diseser zusätzliche Teilbaum kommt daher, weil jeder Schlüssel links und rechts einen Pointer hat. Das erinnert uns an das Gartenzaun-Problem: ein Zaun hat 17 Zwischenräume (entspricht den Schlüsseln), wie viele Lättchen (entsprechen den Pointern) hat der Zaun?

Für die Ordnung des Baumes gibt es 2 verschiedene Definitionen:

Ordnung nach Bayer: Rudolf Bayer, der Erfinder der B-Bäume, hat die Ordnung eines B-Baumes definiert als "die minimale Anzahl Schlüssel in einem Nicht-Wurzel-Knoten". Dies entspricht unserer Formel $\lceil d/2 \rceil - 1$ welche wir bereits kennen gelernt haben.

Diese Definition hat zwei Nachteile:

1. Sie ist nicht eindeutig. Da wir aufrunden, haben gemäss Bayer B-Bäume mit 5 oder 6 Schlüsseln pro Knoten die gleiche Ordnung:

   s=5 $\lceil 5/2 \rceil - 1 = 3 - 1 = 2$

   s=6: $\lceil 6/2 \rceil - 1 = 3 - 1 = 2$.

2. Sie ist nicht sehr intuitiv

Deshalb gibt es eine zweite Definition für die Ordnung von B-Bäumen:

Ordnung nach Knuth: Donald Knuth, ehemaliger Professor an der Stanford University, hat die Ordnung des B-Baumes definiert als die maximale Anzahl Kindknoten, die ein Knoten haben kann. Man nennt dies auch den **Verzweigungsgrad** eines Baumes.

Ohne es zu erwähnen, haben wir bis jetzt die zweite Definition der Ordnung verwendet, weil sie einfacher und eindeutig ist. Wir werden auch weiterhin mit der zweiten Definition arbei-

ten. Allerdings werden Sie in der Literatur hin und wieder die erste Definition gemäss Bayer finden, also wundern Sie sich nicht.

Regel: Alle Blattknoten sind gleich tief

Diese Regel gewährleistet, dass der Baum immer balanciert ist. Wie wir das bewerkstelligen, sehen wir im nächsten Kapitel.

Wissenssicherung: Beantworten Sie die folgenden Fragen

b) Nehmen Sie ein leeres A4-Blatt quer, und zeichnen Sie einen leeren B-Baum auf, welcher 2 Schlüssel pro Knoten (s=2) speichern kann und die Baumhöhe h=2 hat. Wie gross ist die maximale Anzahl Schlüssel (N), die dieser B-Baum speichern kann? Alle Knoten seien voll besetzt. Ordnen (nicht Einfügen!) Sie anschliessend die Schlüssel 1, 2, 3, 4, ..., N den Knoten Ihres B-Baumes zu, so dass dieser optimal besetzt ist. Beachten Sie dabei die Regeln des B-Baumes! Wie sieht der B-Baum aus?

c) Welches Verfahren der Tiefensuche wird in den B-Bäumen verwendet?

d) Ein Knoten eines B-Trees soll genau in einen Datenblock auf der Harddisk passen. Die Harddisk habe eine Blockgrösse B von 512 Bytes. Ein Schlüssel und ein Pointer im Knoten belegen je 4 Bytes. Zudem soll im Datenblock die Anzahl belegter Schlüssel im Knoten gespeichert werden, welche ebenfalls 4 Bytes Platz braucht. Wie viele Schlüssel hat ein Knoten, und welche Ordnung hat der B-Tree?

3.5 Ein paar Formeln

Die Formeln für die Berechnung der verschiedenen Kenngrössen eines B-Baumes sind zum Teil ähnlich wie die Formeln für den binären Baum. Allerdings müssen wir einen zusätzlichen Faktor berücksichtigen: die Knotengrösse s. Bei den binären Bäumen war s = 1, bei den B-Bäumen ist s > 1. Nachfolgend finden Sie eine Gegenüberstellung der Formeln:

Formel	Binärer Baum	B-Baum
Max. Anz. Schlüssel	$2^0 + 2^1 + ... + 2^h = 2^{h+1} - 1$	$s * (d^0 + d^1 + ... + d^h)$
Max. Anz. Knoten	$2^0 + 2^1 + ... + 2^h = 2^{h+1} - 1$	$d^0 + d^1 + ... + d^h$
Max. Anz. innere Knoten	$2^0 + 2^1 + ... + 2^{h-1} = 2^h - 1$	$d^0 + d^1 + ... + d^{h-1}$
Max. Anz. Knoten in Tiefe t	2^t	d^t
Max. Anz. Schlüssel in Tiefe t	2^t	$s * d^t$
Max. Anz. Blattknoten	2^h	d^h
Min. Baumhöhe (best case)	$\lceil \log_2 (N+1) \rceil$	$\lceil \log_d (N+1) \rceil$
Max. Baumhöhe (worst case)	$N - 1$	$\lceil \log_{\lceil d/2 \rceil} (N+1) \rceil$

N entspricht der Anzahl Schlüssel im Baum, h bezeichnet die Höhe und d die Ordnung des Baumes. Die Schreibweise $\lceil ... \rceil$ wird "obere Gaussklammer" genannt. Sie bedeutet, dass wir das Resultat aufrunden, falls es Nachkommastellen gibt. Wie Sie den Logarithmus zur Basis d berechnen, haben Sie bereits im Kapitel 2 bei den binären Bäumen kennen gelernt.

3.6 Einfügen

Wir wollen nun den Schlüssel mit der ID=45 in den folgenden B-Baum einfügen:

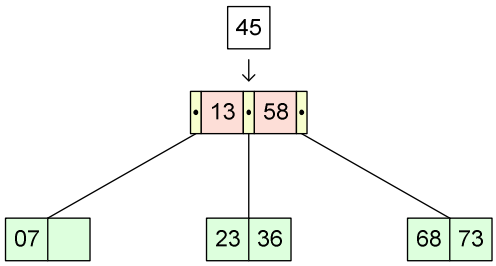


Abbildung 3.5: Einfügen von ID=45

Achtung: im Gegensatz zu den binären Bäumen werden im B-Baum neue Schlüssel immer in den Blattknoten hinzugefügt!

Dazu suchen wir als erstes im B-Baum nach dem Blattknoten, in welchen der der neue Schlüssel eingefügt werden soll, und fügen den Schlüssel sortiert ein. Nach dem Einfügen stellen wir fest, dass der betroffene Blattknoten zu gross geworden ist:

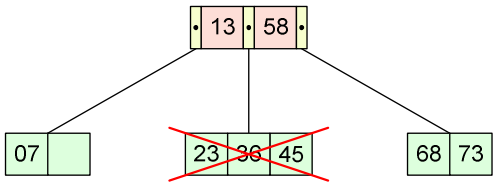


Abbildung 3.6: Knoten wird zu gross

Da der B-Baum die Ordnung d=3 hat, darf ein Knoten maximal 2 Schlüssel beherbergen. Um den B-Baum-Bedingungen wieder gerecht zu werden, müssen wir etwas unternehmen. Wir könnten z.B. den Knoten in einen inneren Knoten und zwei neue Blattknoten aufteilen:

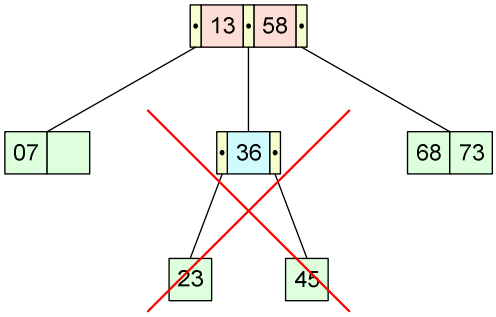


Abbildung 3.7: Auslagerung in Kindknoten ist nicht erlaubt

Allerdings ist dies auch eine irreguläre Anordnung. Wir haben ja gelernt, dass im B-Baum alle Blattknoten die gleiche Tiefe haben, was nun nicht mehr gegeben ist.

Die Lösung ist, dass wir den mittleren Schlüssel mit der ID=36 in den übergeordneten Knoten verschieben:

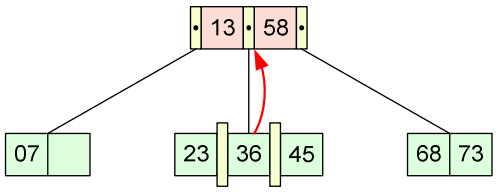


Abbildung 3.8: Schlüssel in übergeordneten Knoten verschieben

Aus den Schlüsseln links und rechts vom Schlüssel mit der ID=36 (gelbe Abtrennung) bilden wir je einen neuen Blattknoten. Im Falle eines inneren Knotens entsprechen die Abtrennungen den Pointern auf die Kindknoten:

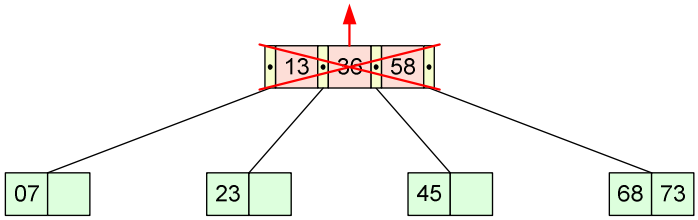


Abbildung 3.9: Übergeordneter Knoten wird zu gross

Nun haben wir die B-Baum-Bedingungen auf der untersten Ebene wieder hergestellt. Allerdings hat sich nun ein neues Problem ergeben: die Wurzel enthält zu viele Schlüssel! Um dies zu beheben, gehen wir genau gleich vor: wir verschieben den mittleren Schlüssel in den übergeordneten Knoten, und bilden aus den Schlüsseln links und rechts zwei neue innere Knoten. Da die Wurzel kein übergeordnetes Element hat, ergibt sich so ein neuer Wurzelknoten:

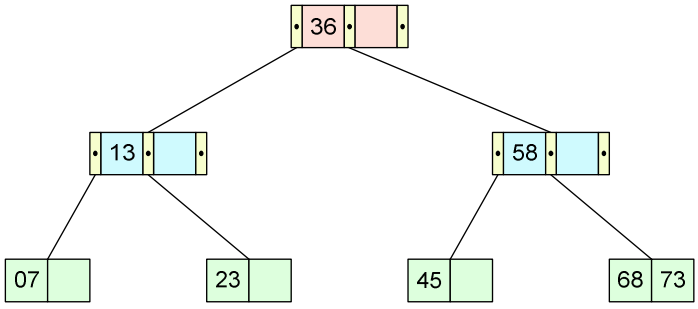


Abbildung 3.10: B-Baum mit neuer Wurzel

Nun sind wir fertig, die B-Baum-Bedingungen sind wieder hergestellt!

MERKE:

→ B-Bäume wachsen "von unten nach oben" ←

→ Schlüssel werden immer in Blattknoten eingefügt ←

Wissenssicherung: Beantworten Sie die folgenden Fragen

e) Die wichtigste Regel beim Einfügen in den B-Baum ist die Regel "Knoten teilen". Erklären Sie, wie das Teilen eines Knotens funktioniert.

f) Papierübung: Fügen Sie die IDs der Tabelle SIMPSONS der Reihe nach in einen leeren B-Baum dritter Ordnung ein. Reihenfolge: 13, 45, 82, 07, 93, 68, 23, 73, 58, 36. Zeigen Sie alle Schritte der Reihe nach auf.

g) Überprüfen Sie Ihre Lösung mit dem B-Baum Applet auf http://db.logging.ch: Fügen Sie die IDs aus Aufgabe f) einem leeren B-Baum hinzu. Beachten Sie, dass das Applet für die Ordnung die Definition von Bayer verwendet.

h) Fügen Sie in die Abbildung 3.10 noch die fehlenden Schlüssel 82 und 93 ein. Wie sieht der B-Baum nach dem Einfügen aus? Wieso sieht er nicht gleich aus wie der B-Baum aus Aufgabe f), obwohl er die gleichen Schlüssel enthält? Erklären Sie, wie es zu diesem Unterschied kommt.

i) Wie gehen Sie vor, wenn Sie im B-Baum aus Aufgabe f) nach Bart Simpson suchen wollen?

3.7 Löschen

Das Löschen eines Schlüssels aus dem B-Baum gestaltet sich etwas schwieriger als das Einfügen. Wir müssen auch den Fall berücksichtigen, dass wir einen Schlüssel aus einem inneren Knoten oder aus der Wurzel löschen. Zudem müssen wir immer sicher stellen, dass jeder Knoten mindestens halb voll ist (genau: $\lceil d/2 \rceil$). Wie wir bereits wissen, ist die Wurzel von dieser Regel ausgenommen: sie darf auch weniger Schlüssel enthalten.

Beim Löschen kann es passieren, dass ein Knoten nach dem Löschen nicht mehr die geforderte minimale Anzahl Schlüssel $\lceil d/2 \rceil$ enthält. Um dies zu verhindern, gibt es 2 Strategien:

1. Wir suchen das zu löschende Element, und löschen es einfach mal. Falls wir nach dem Löschen feststellen, dass der Knoten die B-Baum Regeln verletzt, führen wir eine →**Reorganisation** durch.
2. Wir suchen das zu löschende Element. Während der Suche überprüfen wir jeden Knoten, an dem wir vorbei kommen, ob seine Anzahl beherbergter Schlüssel bereits minimal ist. Falls ja, könnten wir keinen Schlüssel aus diesem Knoten löschen. Wir führen dann ad hoc eine Reorganisation für diesen Knoten durch, so dass seine Anzahl Schlüssel auf $\lceil d/2 \rceil + 1$ erhöht wird. Dann fahren wir mit der Suche fort. Wenn wir den zu löschenden Schlüssel gefunden haben, können wir ihn gefahrlos entfernen.

Für die nachfolgenden Beschreibungen verwenden wir die Variante 1: wir löschen zuerst, und führen dann eine Reorganisation durch, falls nötig. Das ist wie im wilden Westen: zuerst schiessen, dann die Fragen stellen.

Beachten Sie, dass in den folgenden Abbildungen die Blattknoten ebenfalls Zeiger (Pointer) beinhalten. Dies wäre im Prinzip nicht nötig, da Blattknoten ja keine Kinder haben. Wir werden später sehen, wieso man das trotzdem tun kann – für den Moment können Sie diese Gegebenheit ignorieren.

A) Schlüssel aus Blattknoten löschen

Das Löschen eines Schlüssels aus einem Blattknoten ist der einfachste Fall. Oftmals kann der Schlüssel einfach gelöscht werden. Dies ist dann der Fall, wenn der Blattknoten mehr als das geforderte Minimum an Schlüsseln enthält. Die unten stehende Grafik zeigt, welche Schlüssel direkt gelöscht werden können.

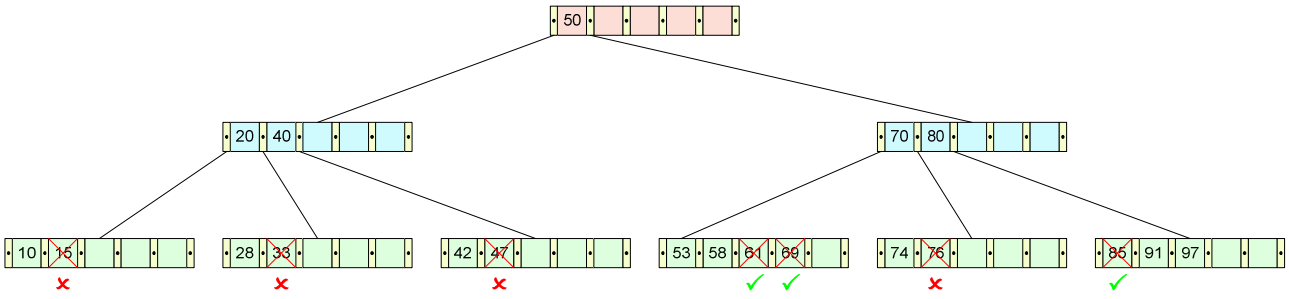


Abbildung 3.11: Schlüssel aus Blattknoten löschen

✓ Diese Schlüssel können direkt gelöscht werden
× Diese Schlüssel können nicht direkt gelöscht werden

Der B-Baum hat die Ordnung d = 6 (Ordnung gemäss Bayer: $\lceil 6/2 \rceil = 3$), so dass die minimale Anzahl Schlüssel in einem Knoten $s = \lceil 6/2 \rceil - 1 = 2$ beträgt. Bei 4 Blattknoten ist dies der Fall, so dass wir aus diesen Knoten nicht direkt löschen dürfen. Aus dem 4ten Knoten von links dürfen wir 2 Schlüssel direkt löschen, z.B. ID=61 und ID=69. Aus dem Blattknoten ganz rechts dürfen wir einen Schlüssel direkt löschen, z.B. ID=85. Wenn wir die mit einem ✕ markierten Schlüssel löschen, erfüllt der B-Baum die Regeln nicht mehr. Dann müssen wir eine →**Reorganisation** vornehmen.

Reorganisation

Eine Reorganisation nehmen wir dann vor, wenn der B-Baum nach dem Löschen eines Schlüssels die B-Baum Bedingungen nicht mehr erfüllt. Eine Reorganisation kann eine **Verschiebung** oder eine **Verschmelzung** von Knoten sein. Die Reorganisation nennt man auch **Rebalancieren**, weil der B-Baum so wieder in Balance gebracht wird.

1. Verschiebung (Rotation)

Wenn wir den Schlüssel mit der ID=76 löschen, dann enthält der Blattknoten nur noch einen Schlüssel, was gegen die B-Baum Bedingungen ist. Um dies zu beheben, können wir von einem Nachbarknoten einen Schlüssel in diesen Knoten verschieben:

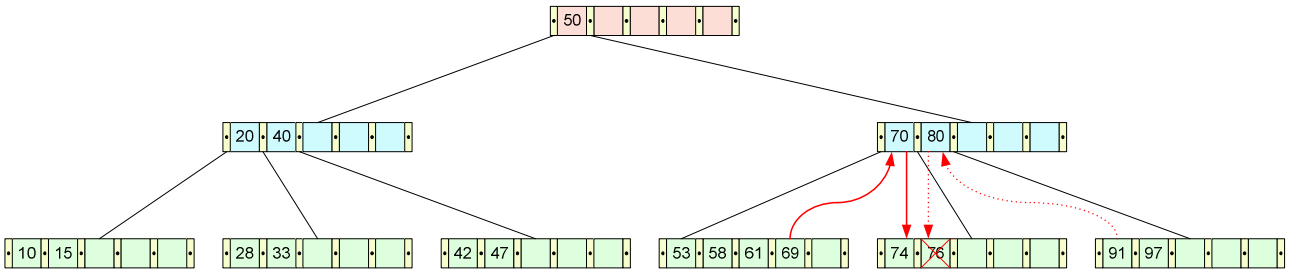


Abbildung 3.12: Schlüssel von einem Nachbarknoten holen

Wir könnten z.B. vom Nachbarknoten auf der rechten Seite den Schlüssel mit der ID=91 in den gemeinsamen Vaterknoten verschieben, und dann vom Vaterknoten den Schlüssel mit der ID=80 zu uns holen. In der Grafik ist dies mit den gestrichelten Linien gekennzeichnet. Allerdings geht das nicht, weil dann im Knoten zu unserer Rechten nur noch 1 Schlüssel enthalten wäre. Also versuchen wir es mit dem Blattknoten zu unserer Linken: wir verschieben den Schlüssel mit der ID=69 in den gemeinsamen Vaterknoten, und holen den Schlüssel mit der ID=70 vom Vaterknoten zu uns:

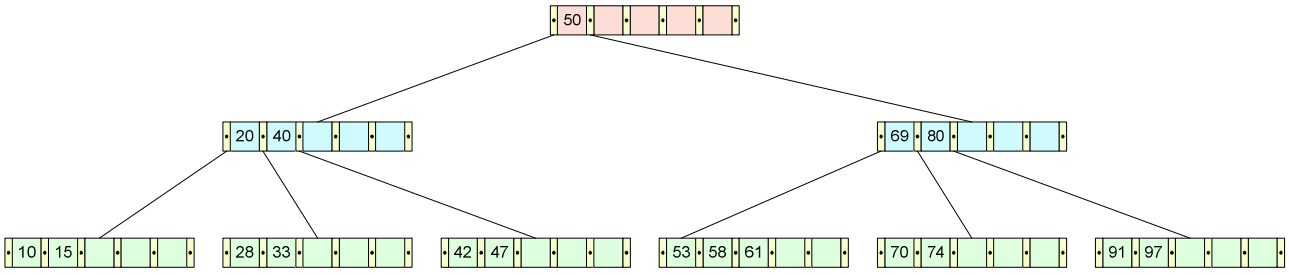


Abbildung 3.13: B-Baum nach dem Löschen und Verschieben (Rotieren)

Beachten Sie, dass vom Verschieben/Rotieren auch allfällige Kinder betroffen sind: wäre der Schlüssel mit ID=69 ein Trennschlüssel für 2 Kindknoten, dann würde einer der Kindknoten mit verschoben. Er wäre dann neu ein Kind unter dem Schlüssel mit ID=70.

So haben wir die B-Baum Bedingungen erfolgreich wieder hergestellt: alle Knoten enthalten mindestens 2 Schlüssel. So eine Verschiebung nennt man auch **Rotation**, weil es so aussieht, als ob die Elemente im Kreis rotiert würden. Doch was wäre passiert, wenn auch der Knoten auf der linken Seite nur 2 Schlüssel enthalten hätte? Dann wäre eine Verschiebung gar nicht möglich gewesen... Dafür gibt es die Verschmelzung!

2. Verschmelzung

Wenn wir den Schlüssel mit der ID=33 löschen, dann enthält der Blattknoten nur noch einen Schlüssel, was gegen die B-Baum Regeln ist. Als erstes versuchen wir eine Verschiebung: wir holen uns einen Schlüssel vom Knoten auf der linken oder der rechten Seite:

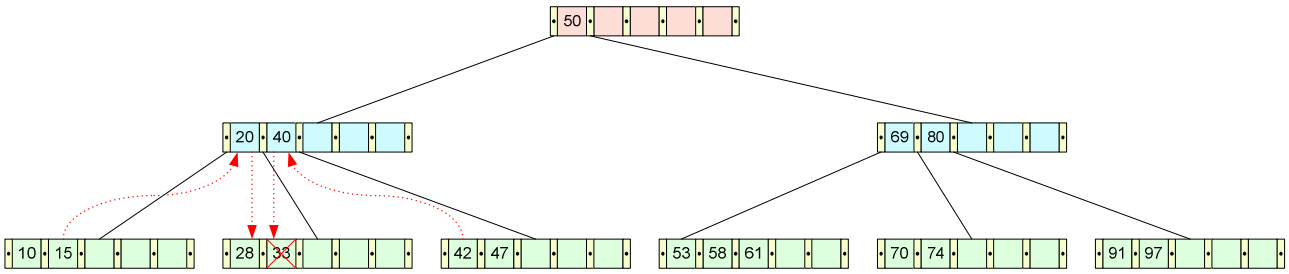


Abbildung 3.14: Verschieben (Rotieren) ist nicht möglich

Leider ist das in diesem Fall nicht möglich: sowohl der linke als auch der rechte Blattknoten enthalten bereits die minimale Anzahl Schlüssel, eine Verschiebung würde die B-Baum Regeln nicht wieder herstellen. Deshalb führen wir eine Verschmelzung durch:

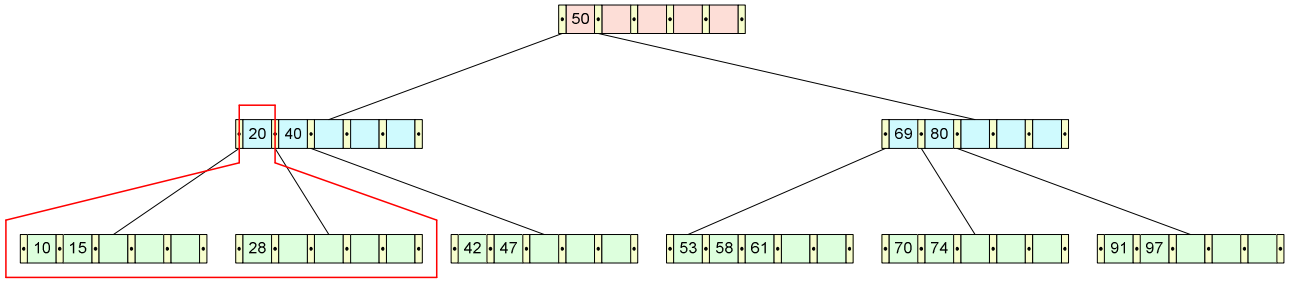


Abbildung 3.15: Blattknoten verschmelzen

Wir verschmelzen unseren Knoten – welcher im Moment zu wenig Schlüssel enthält – mit einem Knoten auf unserer linken oder rechten Seite, welcher lediglich die minimale Anzahl Schlüssel enthält, zu einem neuen Knoten. Das dazwischen liegende Element aus dem Vaterknoten nehmen wir dazu. Das Motto heisst also "aus 2 mach 1": wir haben nun einen Knoten erhalten, welcher wieder die minimale Anzahl Schlüssel enthält. Somit haben wir die B-Baum Bedingungen auf der untersten Stufe wieder hergestellt.

Allerdings ergibt sich jetzt ein neues Problem auf dem nächsthöheren Level: der innere Knoten, welcher den Schlüssel mit der ID=40 beherbergt, enthält nur noch 1 Element. Das ist gegen die B-Baum Bedingungen!

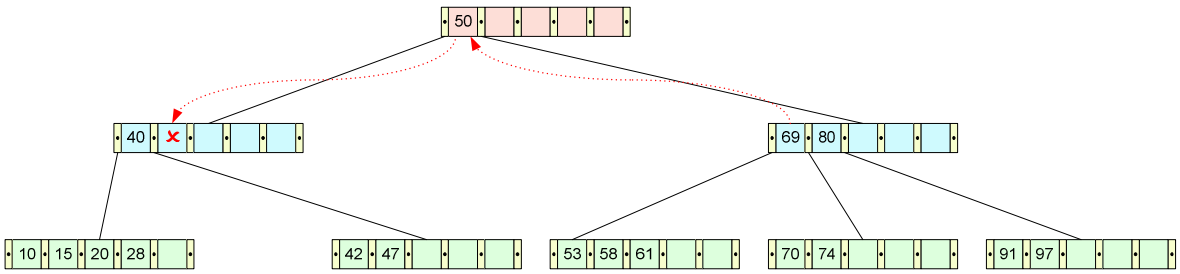


Abbildung 3.16: Innerer Knoten enthält zuwenig Schlüssel

✘ Da fehlt ein Schlüssel!

Nun machen wir wieder das Gleiche wie vorher: wir versuchen eine Verschiebung. Wir verschieben ein Element des Knotens zu unserer Linken oder zu unserer Rechten in den gemeinsamen Vaterknoten, und holen ein Element vom Vaterknoten zu uns.

Allerdings ist das hier nicht möglich, da der Knoten, welcher die ID=69 beherbergt, lediglich die minimale Anzahl Schlüssel enthält. Deshalb führen wir wieder eine Verschmelzung durch:

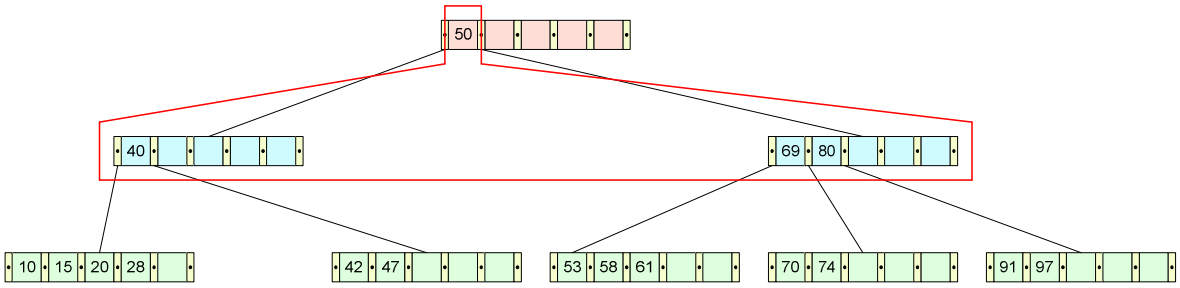


Abbildung 3.17: Innere Knoten verschmelzen

Wir verschmelzen unseren Knoten mit einem Knoten auf der linken oder rechten Seite, welcher nur die minimale Anzahl Schlüssel enthält, zu einem neuen inneren Knoten. Dann nehmen wir noch das dazwischen liegende Element aus dem Vaterknoten dazu, und fertig:

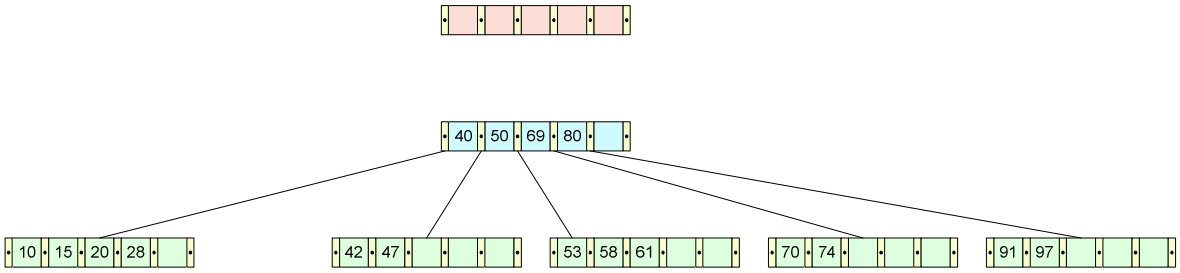


Abbildung 3.18: Wurzel und innerer Knoten nach der Verschmelzung

Da die Wurzel nun leer geworden ist, hat sie keine Verbindung mehr zum restlichen Baum. Deshalb können wir sie ganz einfach löschen:

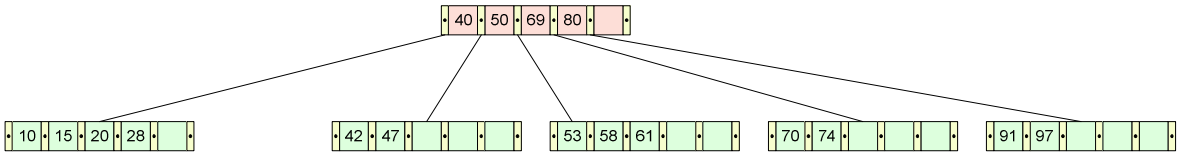


Abbildung 3.19: B-Baum nach dem Entfernen der Wurzel

Aus den beiden inneren Knoten ist eine neue Wurzel entstanden! Nun sind wir fertig, die B-Baum-Bedingungen sind wieder hergestellt!

MERKE:

→ B-Bäume schrumpfen "von oben nach unten" ←

B) Schlüssel aus innerem Knoten oder der Wurzel löschen

Das Löschen eines Schlüssels aus einem inneren Knoten oder aus der Wurzel stellt uns vor ein neues Problem:

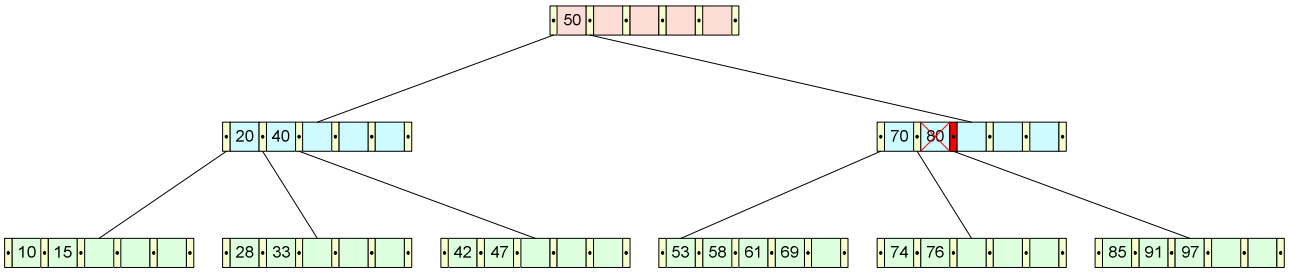


Abbildung 3.20: Löschen aus innerem Knoten – Pointer ohne Trennschlüssel

Wenn wir den Schlüssel einfach löschen, dann hat der Pointer zum Blattknoten (rot markiert) auf der rechten Seite keinen Trennschlüssel mehr. Das ist ein illegaler Zustand.

Symmetrischer (in-order) Vorgänger / Nachfolger

Um das Problem des fehlenden Trennschlüssels zu vermeiden, vertauschen wir den zu löschenden Schlüssel mit seinem symmetrischen (in-order) Vorgänger oder Nachfolger, welcher sich per Definition in einem Blattknoten befindet.

Der symmetrische Vorgänger ist jener Schlüssel, welcher direkt vor dem aktuellen Schlüssel gefunden wird, wenn man den B-Baum traversiert. Dem zu Folge ist der symmetrische Nachfolger jener Schlüssel, welcher beim Traversieren des B-Baumes direkt nach dem aktuellen Schlüssel gefunden wird.

Dass sich der symmetrische Vorgänger/Nachfolger in einem Blattknoten befindet, ergibt sich aus der Art und Weise, wie der B-Baum traversiert wird. Siehe dazu Abbildung 3.4: Traversierung eines B-Baumes. Deshalb werden die symmetrischen Vorgänger/Nachfolger auch in-order Vorgänger und in-order Nachfolger genannt.

Der symmetrische Vorgänger befindet sich in Bezug auf den zu löschenden Schlüssel (Trennschlüssel) im linken Teilbaum, und zwar im Blattknoten ganz rechts.

Der symmetrische Nachfolger befindet sich im rechten Teilbaum, nämlich im Blattknoten ganz links.

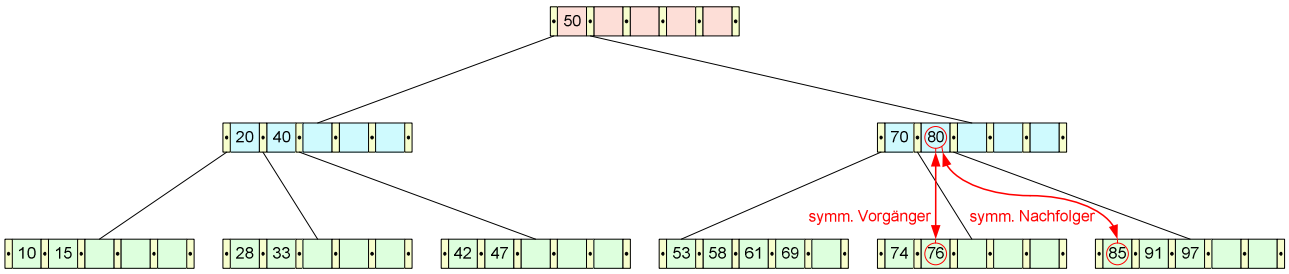


Abbildung 3.21: Löschen aus innerem Knoten – symmetrischer Vorgänger/Nachfolger

Für die Vertauschung haben wir 2 Möglichkeiten: entweder verwenden wir den symmetrischen Vorgänger, oder den symmetrischen Nachfolger. Dabei berücksichtigen wir den Füllgrad der betroffenen Blattknoten. Wir müssen darauf achten, dass nach der Löschung aus dem Blattknoten kein illegaler Zustand auftritt.

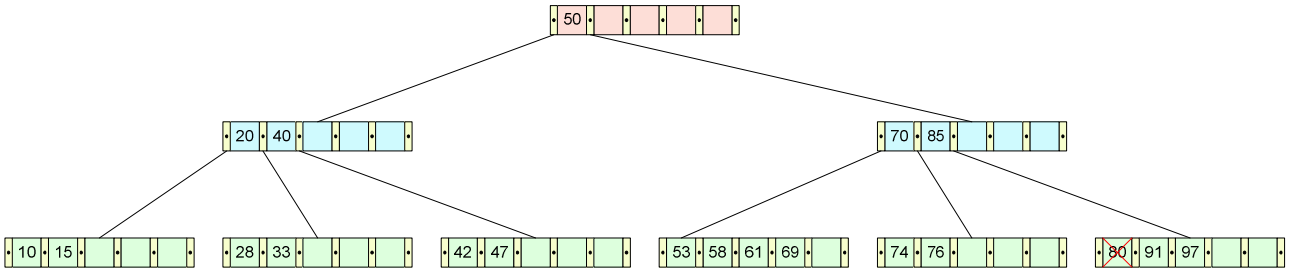


Abbildung 3.22: Löschen aus innerem Knoten – Schlüssel vom Kindknoten holen

Der Schlüssel mit ID=76 eignet sich nicht für eine Vertauschung, da nach dem Löschen zu wenig Schlüssel im Blattknoten wären. Wir müssten dann eine **→Reorganisation** durchführen, die wir uns ersparen können. Deshalb wählen wir ID=85 für die Vertauschung.

Nach dem Vertauschen können wir den Schlüssel mit ID=80 einfach aus dem Blattknoten löschen. Dann sieht der B-Baum so aus:

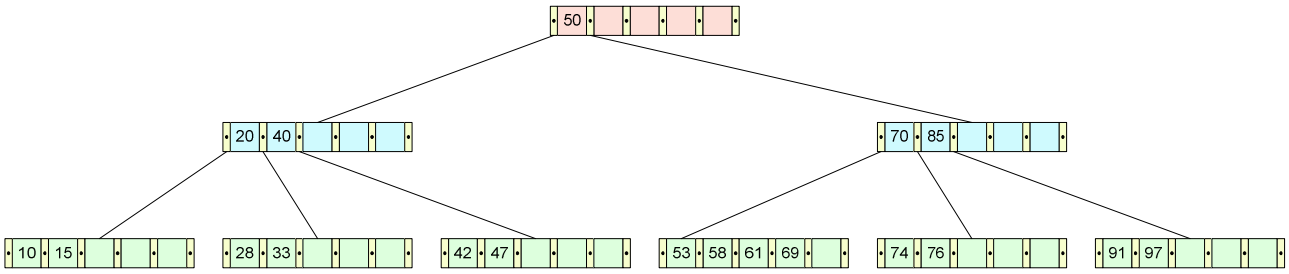


Abbildung 3.23: B-Baum nach Löschen aus innerem Knoten

Nun sind wir fertig, wir haben erfolgreich einen Schlüssel aus einem inneren Knoten gelöscht. Die B-Baum Bedingungen sind nach wie vor intakt.

Doch was wäre, wenn all unsere Kinder nur die minimale Anzahl Schlüssel enthalten würden? Im folgenden Beispiel wollen wir den Schlüssel mit der ID=40 löschen:

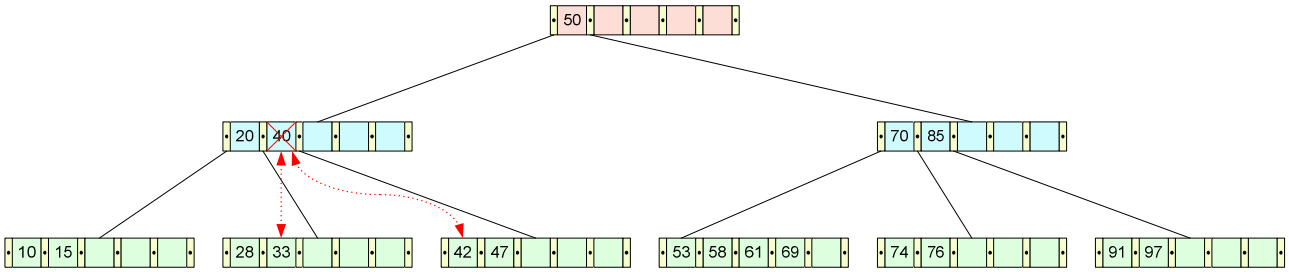


Abbildung 3.24: Löschen aus innerem Knoten – Schlüssel vom Kindknoten holen nicht möglich

In diesem Fall können wir keine Vertauschung mit dem symmetrischen Vorgänger/Nachfolger durchführen, ohne dass die B-Baum Bedingungen nach dem Löschen verletzt sind. Wir tun es trotzdem, und kümmern uns nachher um die Wiederherstellung der B-Baum Bedingungen. Vertauschen wir also die Schlüssel mit ID=40 und ID=42. Nach dem Löschen der ID=40 aus dem Blattknoten enthält dieser nur noch ein Element mit der ID=47:

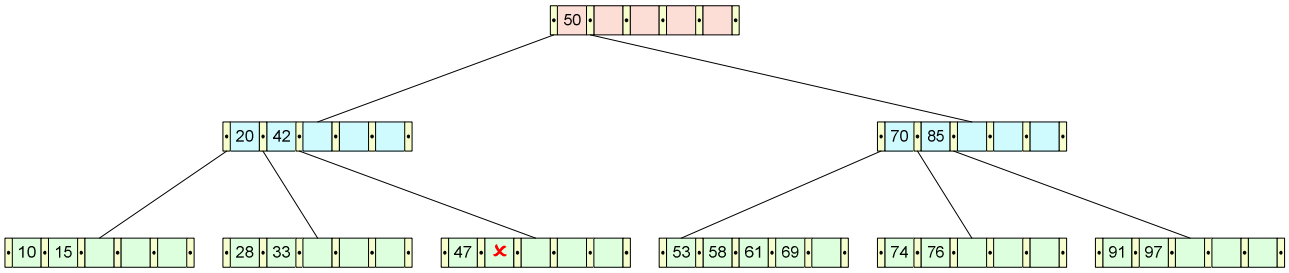


Abbildung 3.25: Blattknoten enthält zuwenig Schlüssel

Wir hätten genau so gut die ID=33 für die Vertauschung verwenden können. Da die B-Baum Bedingungen nach dem Löschen so oder so verletzt werden, spielt es keine Rolle.

Um die B-Baum Bedingungen im betreffenden Blattknoten wieder herzustellen, drängt sich eine **→Reorganisation** (Verschiebung oder Verschmelzung) auf. Da wir gesehen haben, dass alle 3 Blattknoten bereits vor der Löschung nur die minimale Anzahl Elemente enthalten hatten, kommt eine Verschiebung nicht in Frage. Wir führen also eine Verschmelzung durch, die wir ja bereits kennen:

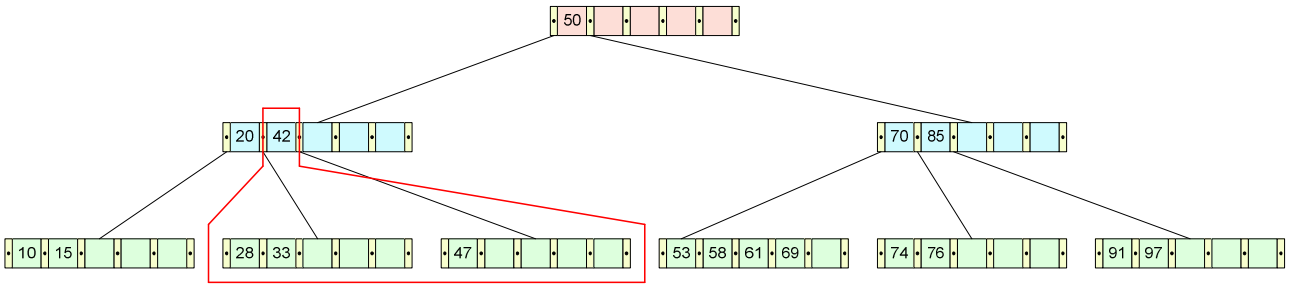


Abbildung 3.26: Blattknoten verschmelzen

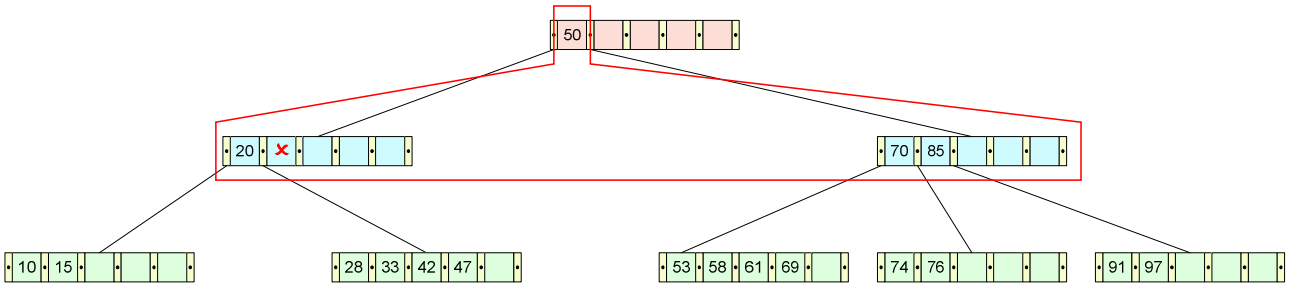


Abbildung 3.27: Innere Knoten verschmelzen

So entsteht aus 2 Blattknoten ein neuer Blattknoten, welcher die IDs 28, 33, 42 und 47 enthält. Nun stellt sich wieder ein Problem, das wir bereits angetroffen haben: der innere Knoten enthält zu wenig Elemente. Wir versuchen zuerst eine Verschiebung, was nicht geht, weil der Nachbarknoten nur die minimale Anzahl Elemente enthält. Also führen wir nochmals eine Verschmelzung durch. Es entsteht eine neue Wurzel, der B-Baum ist um einen Level kleiner geworden:

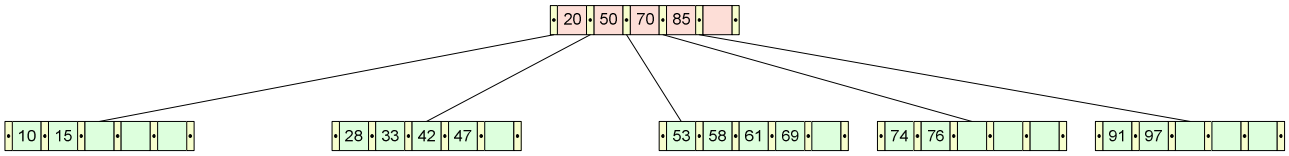


Abbildung 3.28: B-Baum nach Löschen aus innerem Knoten

Nun sind wir fertig, die B-Baum-Bedingungen sind wieder hergestellt!

MERKE:

→ B-Bäume schrumpfen "von oben nach unten" ←

→ Schlüssel werden immer aus Blattknoten gelöscht ←

Wissenssicherung: Beantworten Sie die folgenden Fragen

j) Welche Verfahren zur Wiederherstellung (Reorganisation) der B-Baum Struktur haben Sie kennen gelernt?

k) Wieso können Schlüssel nur aus Blattknoten gelöscht werden?

l) Wann erfolgt eine Verschiebung (Rotation) von Knoten?

m) Wann erfolgt eine Verschmelzung von Knoten?

n) Wie geht man vor, wenn man einen Schlüssel aus einem inneren Knoten löschen will?

o) Wie geht man vor, wenn man die Wurzel löschen will?

3.8 Wo bleiben die Daten?

Bis jetzt haben wir im binären Baum und im B-Baum nur Schlüssel gespeichert. Vielleicht fragen Sie sich: wo bleiben denn die Daten? Was bringt es, wenn wir nur Schlüssel speichern, aber keine Daten? Und was ist überhaupt der Zusammenhang zwischen einem B-Baum, einem Index und einer Datenbank-Tabelle?

In der Tat haben wir das bis jetzt der Einfachheit halber unterschlagen. Ein B-Baum speichert nebst den Schlüsseln auch ein Datenelement, welches z.B. eine Adresse im Speicher sein kann. Es wird also immer die Zuordnung von einem Schlüssel zu einem Datenelement gespeichert (**key/value pair**). Bezogen auf das Beispiel aus Kapitel 1 könnte das Datenelement z.B. ein Pointer in den Hauptspeicher sein, wo die zum Schlüssel gehörende Row gespeichert ist.

Die Tabelle SIMPSONS sei im Speicher ab der Adresse 0x0100 abgespeichert:

Adresse im Speicher (**value**)	SIMPSONS			
	ID (**key**)	FIRSTNAME	LASTNAME	SSN
0x0100:	13	Milhouse	van Houten	13259
0x0200:	45	Waylon	Smithers	89241
0x0300:	82	Marge	Simpson	71430
0x0400:	07	Monty	Burns	52367
0x0500:	93	Bart	Simpson	38172
0x0600:	68	Seymour	Skinner	94891
0x0700:	23	Homer Jay	Simpson	27272
0x0800:	73	Moe	Szyslak	06585
0x0900:	58	Lisa	Simpson	66914
0x0A00:	36	Chief Clancy	Wiggum	40103

Der Index auf der Kolonne ID der Tabelle SIMPSONS sieht im B-Baum dann z.B. wie folgt aus:

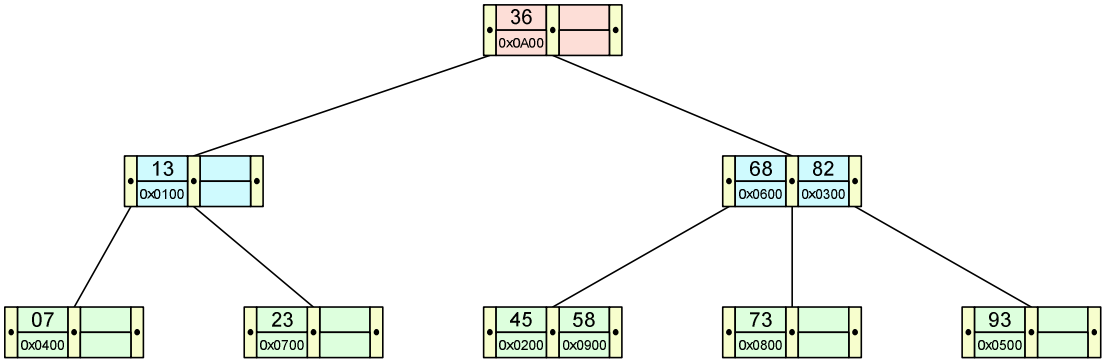


Abbildung 3.29: B-Baum mit Schlüssel und Daten

Wenn wir nun die Row von Homer Jay Simpson (ID=23) finden wollen, dann suchen wir zuerst im Index (d.h. im B-Baum) nach der ID=23, und finden dort das Datenelement 0x0700. Dieser Pointer sagt uns, dass die Row an der Speicherstelle 0x0700 zu finden ist. So haben wir einen sehr schnellen Zugriff auf die Daten!

3.9 Weiterentwicklungen: B$^+$-Baum und B*-Baum

Die Erfindung des B-Baumes war ein grosser Meilenstein in der Entwicklung der Informatik und in der Optimierung von Systemen. Der B-Baum spielt in der Praxis eine grosse Rolle, so dass aus ihm zahlreiche Weiterentwicklungen entstanden sind.

B$^+$-Baum

Eine Weiterentwicklung des B-Baumes ist der B$^+$-Baum. Seine Besonderheit ist, dass er die Datenelemente ausschliesslich in den Blattknoten speichert. In den inneren Knoten und in der Wurzel (sofern sie Kinder hat) werden nur Schlüssel gespeichert. Dies hat zur Folge, dass gewisse Schlüssel im Baum mehrfach vorkommen: alle Schlüssel der Wurzel und der inneren Knoten kommen auch in den Blattknoten vor.

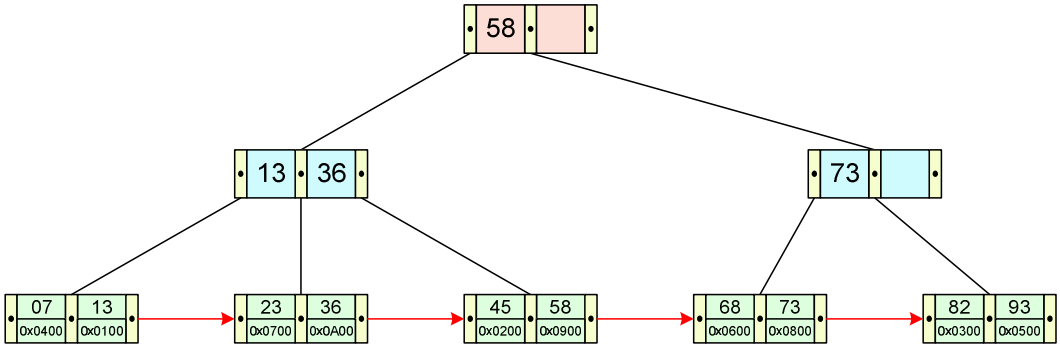


Abbildung 3.30: B$^+$-Baum: Blätter sind verlinkt, Schlüssel kommen mehrfach vor

Der Vorteil ist, dass die Wurzel und die inneren Knoten weniger Speicherplatz belegen, da sie keine Datenelemente speichern müssen. Dies ermöglicht es uns, den **Verzweigungsgrad** des Baumes zu erhöhen, so dass wir einen flacheren Baum erhalten. Die Blattknoten sind zudem untereinander verlinkt, was eine effiziente **Bereichssuche** ermöglicht. Fragen wie z.B. "gib mir alle Rows mit $45 \leq ID \leq 73$ zurück" sind so einfach zu beantworten, ohne dass man während der Suche wieder bis zur Wurzel aufsteigen muss.

Die Verkettung der Elemente der Blattknoten ist die Erklärung dazu, weshalb man in Blattknoten trotzdem Zeiger (Pointer) verwenden kann: die Elemente werden zu einer linearen Liste verkettet, jeder Pointer zeigt auf das nächste Element in der Liste.

Weil der B$^+$-Baum alle Eigenschaften des B-Baumes besitzt, und zudem eine sehr effiziente Bereichssuche zulässt, wird der B$^+$-Baum sehr oft in Datenbank-Systemen eingesetzt.

B*-Baum

Eine weitere Weiterentwicklung ist der B*-Baum. Er ist sehr ähnlich wie der B$^+$-Baum, mit dem Unterschied, dass seine Knoten immer zu 2/3 gefüllt sein müssen – im Gegensatz zum B-Baum, bei dem der Füllgrad $\lceil d/2 \rceil$ beträgt.

Weitere Bäume

Aus dem B-Baum ist mit der Zeit eine Vielzahl von weiteren Bäumen hervor gegangen. Eine nicht abschliessende Auflistung finden Sie im Anhang A: "Weiterführende Quellen". Wir gehen an dieser Stelle nicht mehr weiter auf diese Bäume ein, da dies den Rahmen dieses Leitprogrammes sprengen würde.

3.10 Zusammenfassung

In diesem Kapitel haben Sie gelernt, was ein B-Baum ist, und wie er sich vom binären Baum unterscheidet. Wir haben gesehen, dass die Regeln des B-Baumes etwas komplexer sind als die Regeln des binären (Such-)Baumes. Zudem ist der B-Baum etwas aufwändiger im Unterhalt, was z.B. das Einfügen und Löschen von Elementen betrifft. Dafür hat der B-Baum den grossen Vorteil, dass er **immer balanciert** ist, und somit den vorhandenen Speicher effizient nutzt. Damit sich der B-Baum automatisch balanciert, haben wir Rekonstruktionsmassnahmen kennen gelernt, wie z.B. die **Verschiebung** oder das **Verschmelzen** von Knoten.

Sie kennen nun auch die wichtigsten Kenngrössen des B-Baumes, und wissen, wie diese zusammen hängen. Wir haben z.B. gesehen, dass die maximale Anzahl Schlüssel pro Knoten direkt mit der Ordnung des B-Baumes zusammen hängt. Auch haben wir gelernt, von was die Baumhöhe und die Grösse der Knoten abhängt. Durch **Erhöhen des Verzweigungsgrades** und entsprechender Anpassung an die Blockgrösse eines Systems erreichen wir einen flacheren Baum, womit die Effizienz weiter gesteigert werden kann.

Speziell ist auch, wie ein B-Baum wächst, beziehungsweise schrumpft – nämlich genau umgekehrt als der binäre Baum, d.h. "von unten nach oben" resp. "von oben nach unten".

In den nächsten zwei Kapiteln widmen wir uns der Praxis: wir werden sehen, wie Indexe in einem Datenbanksystem eingesetzt werden, und wie sie helfen, den Zugriff auf die Daten zu beschleunigen.

3.11 Lösungen zu den Wissenssicherungsfragen

a) *Gegeben ist ein B-Baum mit der Ordnung **d=5** und der Höhe **h=2***

1. *Wie viele Schlüssel enthält die Wurzel im Minimum / im Maximum?*
2. *Wie viele Schlüssel enthält ein innerer Knoten im Minimum / im Maximum?*
3. *Wie viele Schlüssel enthält ein Blattknoten im Minimum / im Maximum?*
4. *Wie viele Kindknoten hat die Wurzel im Minimum / im Maximum?*
5. *Wie viele Kindknoten hat ein innerer Knoten im Minimum / im Maximum?*
6. *Wie viele Kindknoten hat ein Blattknoten im Minimum / im Maximum?*

Anzahl Schlüssel:

1.	Wurzel Minimum:	**s = 0**	(trivial, der B-Baum ist leer...)
		s = 1	(der B-Baum hat Höhe h ≥ 0)
	Wurzel Maximum:	**s = 4**	(s = d – 1 = 5 – 1 = 4)
2.	Innere Knoten Minimum:	**s = 2**	(s = ⌈d/2⌉ – 1 = ⌈5/2⌉ – 1 = 3 – 1 = 2)
	Innere Knoten Maximum:	**s = 4**	(s = d – 1 = 5 – 1 = 4)
3.	Blattknoten Minimum:	**s = 2**	(s = ⌈d/2⌉ – 1 = ⌈5/2⌉ – 1 = 3 – 1 = 2)
	Blattknoten Maximum:	**s = 4**	(s = d – 1 = 5 – 1 = 4)

Anzahl Kindknoten:

4.	Wurzel Minimum:	**d = 0**	(trivial, der B-Baum hat Höhe h = 0)
		d = 2	(per Definition, wenn Höhe h > 0)
	Wurzel Maximum:	**d = 5**	(Anzahl = Ordnung)
5.	Innere Knoten Minimum:	**d = 3**	(d = ⌈d/2⌉ = ⌈5/2⌉ = 3)
	Innere Knoten Maximum:	**d = 5**	(Anzahl = Ordnung)
6.	Blattknoten Minimum:	**d = 0**	(Blätter haben keine Kinder)
	Blattknoten Maximum:	**d = 0**	(Blätter haben keine Kinder)

b) *Nehmen Sie ein leeres A4-Blatt quer, und zeichnen Sie einen leeren B-Baum auf, welcher 2 Schlüssel pro Knoten (s=2) speichern kann und die Baumhöhe h=2 hat. Wie gross ist die maximale Anzahl Schlüssel (N), die dieser B-Baum speichern kann? Alle Knoten seien voll besetzt. Ordnen Sie anschliessend die Schlüssel 1, 2, 3, 4, ..., N den Knoten Ihren Baum zu. Beachten Sie dabei die Regeln des B-Trees! Wie sieht der B-Baum aus?*

Wenn Sie den B-Baum aufzeichnen, erkennen Sie, dass der B-Baum total 26 Schlüssel speichern kann:

Wurzel:	1 x 2 Schlüssel
Innere Knoten	3 x 2 Schlüssel = 6 Schlüssel
Blattknoten	9 x 2 Schlüssel = 18 Schlüssel

Wir können die maximale Anzahl Schlüssel N auch berechnen:

Anz. Schlüssel Total = Anz. Schlüssel pro Knoten * Anz. Knoten

Anz. Schlüssel Total = $2 * (3^0 + 3^1 + 3^2) = 2 * (1 + 3 + 9) = 2 * 13 = 26$

Die Zuordnung der Schlüssel 1..N (1..26) sieht wie folgt aus:

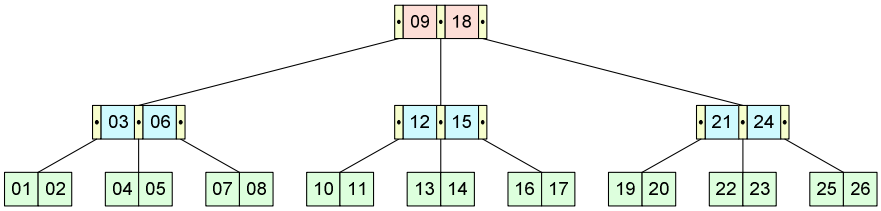


Abbildung 3.31: Voll besetzter B-Baum mit Schlüsseln 01 – 26 (Lösung)

c) *Welches Verfahren der Tiefensuche wird in den B-Bäumen verwendet?*

Das kann man nicht exakt mit der Tiefensuche bei den binären Suchbäumen vergleichen. Auf jeden Fall suchen wir zuerst im Knoten selbst, bevor wir zu einem Kindknoten absteigen. Somit fallen die in-order und die post-order Suche als Vergleich weg: diese Verfahren steigen zuerst in den linken Teilbaum ab. Bei den B-Bäumen gibt es jedoch keinen "linken und rechten" Teilbaum, da ein Knoten eines B-Baumes mehr als einen Wert speichern kann. Deshalb kann der B-Baum mehr als 2 Teilbäume haben.

Am ähnlichsten ist die **pre-order** Suche: zuerst durchsuchen wir den Knoten selbst, danach steigen wir bei Bedarf in den jeweiligen Teilbaum ab. Da ein Knoten eines B-Baumes mehr als einen Wert speichern kann, müssen wir den Knoten wirklich durchsuchen. Bei den binären Suchbäumen reduziert sich die "Suche im Knoten" auf einen einfachen Vergleich, da der binäre Suchbaum nur einen Wert im Knoten speichern kann.

d) *Ein Knoten eines B-Trees soll genau in einen Datenblock auf der Harddisk passen. Die Harddisk habe eine Blockgrösse B von 512 Bytes. Ein Schlüssel und ein Pointer im Knoten belegen je 4 Bytes. Zudem soll im Datenblock die Anzahl belegter Schlüssel im Knoten gespeichert werden, welche ebenfalls 4 Bytes Platz braucht. Wie viele Schlüssel hat ein Knoten, und welche Ordnung hat der B-Tree?*

Da ein Schlüssel und ein Pointer je 4 Bytes belegen, brauchen wir ca. die Hälfte der 512 Bytes für die Schlüssel, und die andere Hälfte für die Pointer.

Berechnung:

Blockgrösse = (Anz. Keys * 4 Bytes) + (Anz. Pointer * 4 Bytes) + 4 Bytes für Anz. Keys

B = s * 4 Bytes *+ (s+1) * 4 Bytes + 4 Bytes // 4 Bytes ausklammern

B = 4 Bytes * (s + (s+1) + 1)

B = 4 Bytes * (2s + 2)

Auflösen nach s:

s = (B – 2) / (4 Bytes * 2) = (B – 2) / (8 Bytes) = 510 / 8 Bytes = 63.75 → Abrunden: **63**

d = s + 1 = 63 + 1 = **64**

Kontrolle:

512 Bytes = **63 Schlüssel** * 4 Bytes + **64 Pointer** * 4 Bytes + 4 Bytes für Anz. Schlüssel

512 Bytes = 252 Bytes für Schlüssel + 256 Bytes für Pointer + 4 Bytes für Anz. Schlüssel

Da der B-Tree 64 Pointer hat, ist die **Ordung 64**. In einem einzigen Knoten können wir bis zu **63 Schlüssel** speichern.

e) *Die wichtigste Regel beim Einfügen in den B-Baum ist die Regel "Knoten teilen". Erklären Sie, wie das Teilen eines Knotens funktioniert.*

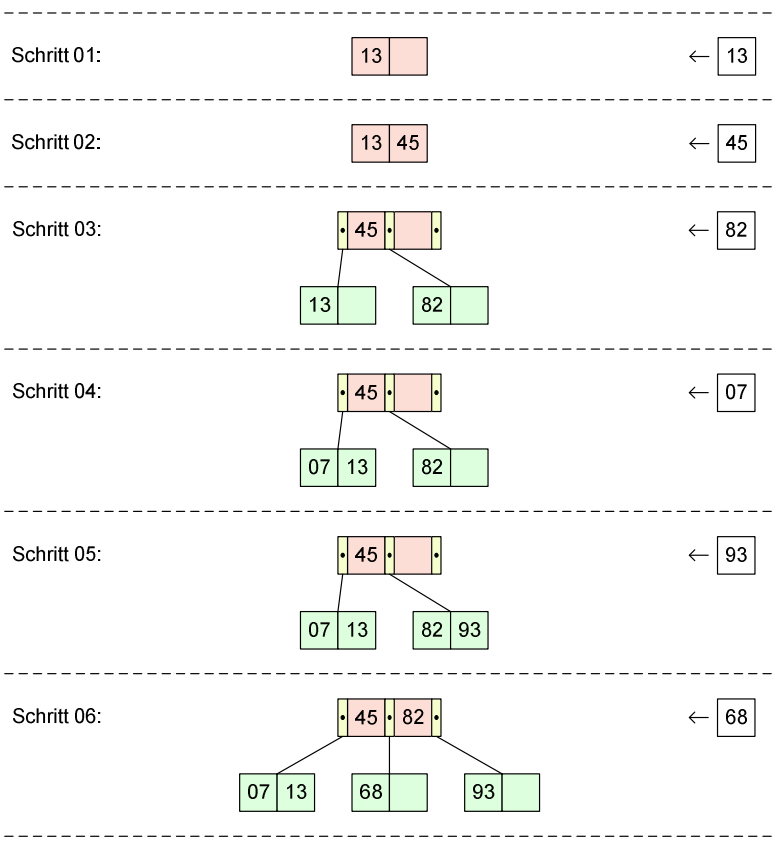
Ein Knoten muss immer dann geteilt werden, wenn er zu gross wird. Dies ist immer dann der Fall, wenn man einen Schlüssel in einen (Blatt-)Knoten einfügt, obwohl der Knoten bereits voll ist.

Vorgehen:

1. Man fügt den Schlüssel einfach mal ein. Das Einfügen erfolgt sortiert.
2. Man wählt den mittleren Schlüssel als sogenannten **Trennschlüssel** aus.
3. Aus den Elementen links und rechts des Trennschlüssels erzeugen wir je einen neuen Knoten. Es entstehen also 2 Knoten.
4. Den Trennschlüssel verschieben wir in den darüber liegenden Vaterknoten. Falls es keinen Vaterknoten gibt, entsteht ein neuer Wurzelknoten.
5. Falls der Vaterknoten durch das Verschieben des Trennschlüssels zu gross geworden ist, wiederholen wir dieses Verfahren mit dem Vaterknoten.

Am Schluss haben wir erfolgreich einen Schlüssel eingefügt. Eventuell ist es beim Einfügen nötig, mehrere Knoten entlang des Pfades vom Blattknoten zu der Wurzel zu teilen.

f) *Papierübung: Fügen Sie die IDs der Tabelle SIMPSONS der Reihe nach in einen leeren B-Baum dritter Ordnung ein. Reihenfolge: 13, 45, 82, 07, 93, 68, 23, 73, 58, 36. Zeigen Sie alle Schritte der Reihe nach auf.*



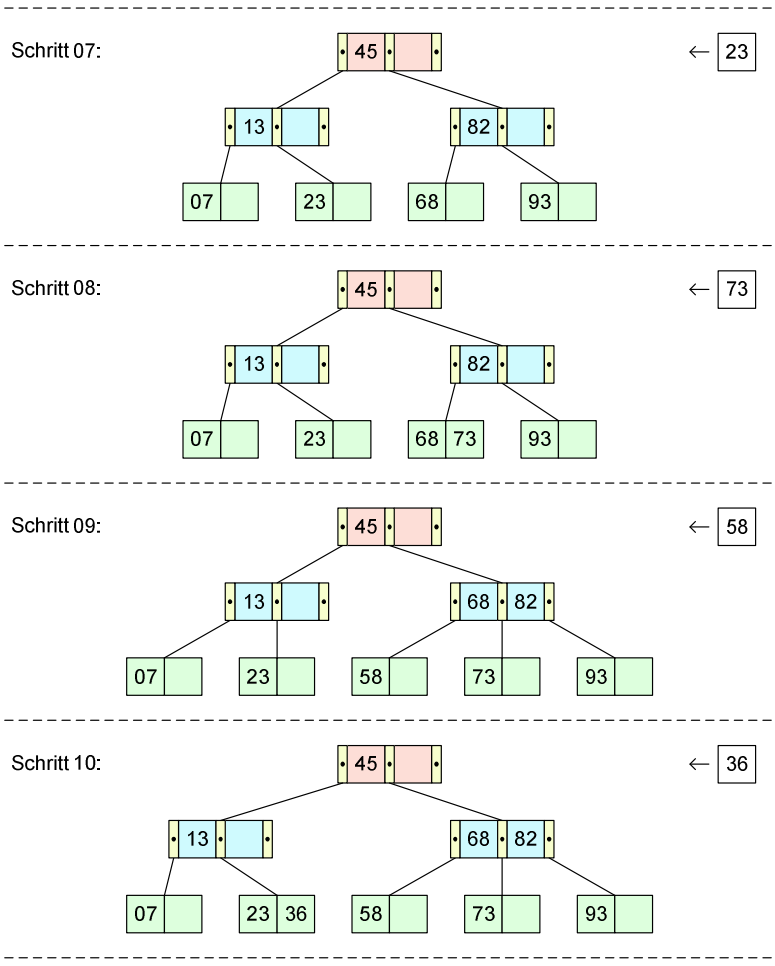


Abbildung 3.32: Einfügen der Tabelle SIMPSONS in einen B-Baum (Lösung)

g) *Überprüfen Sie Ihre Lösung mit dem B-Baum Applet auf db.logging.ch: Fügen Sie die IDs aus Aufgabe f) einem leeren B-Baum hinzu. Beachten Sie, dass das Applet zur Bestimmung der Ordnung die Definition von Bayer verwendet.*

Vorgabe: Ordnung d=3 (nach Knuth)

Ordnung gemäss Bayer: $\lceil d/2 \rceil - 1 = \lceil 3/2 \rceil - 1 = 2 - 1 = \mathbf{1}$.

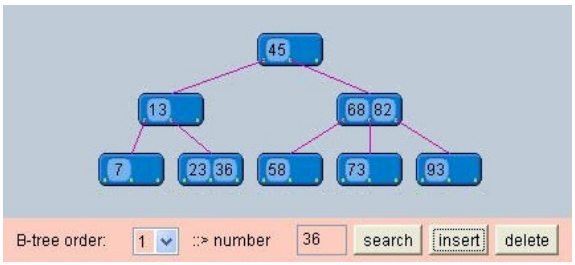


Abbildung 3.33: Lösung mit Applet

h) *Fügen Sie in die Abbildung 3.10 noch die fehlenden Schlüssel 82 und 93 ein. Wie sieht der B-Baum nach dem Einfügen aus? Wieso sieht er nicht gleich aus wie der B-Baum aus Aufgabe f), obwohl er die gleichen Schlüssel enthält? Erklären Sie, wie es zu diesem Unterschied kommt.*

Der B-Baum sieht wie folgt aus:

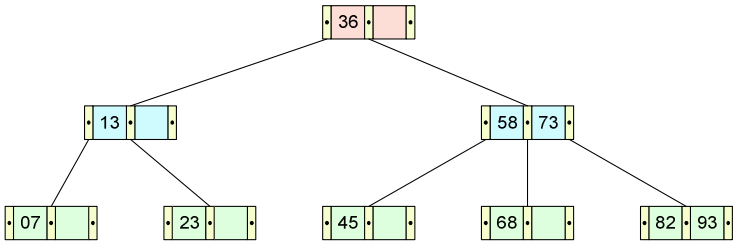


Abbildung 3.34: B-Baum aus Abbildung 3.10 mit IDs 82 und 93

Die beiden B-Bäume enthalten die gleichen Elemente, und sehen trotzdem unterschiedlich aus. Vermutlich wurden die Elemente in einer anderen Reihenfolge hinzugefügt, oder es wurden zwischendurch andere Elemente eingefügt und wieder gelöscht. Ein Einfügen und Löschen verursacht eventuell eine Reorganisation des B-Baumes, wie wir im nächsten Kapitel noch sehen werden.

i) *Wie gehen Sie vor, wenn Sie im B-Baum aus Aufgabe f) nach Bart Simpson suchen wollen?*

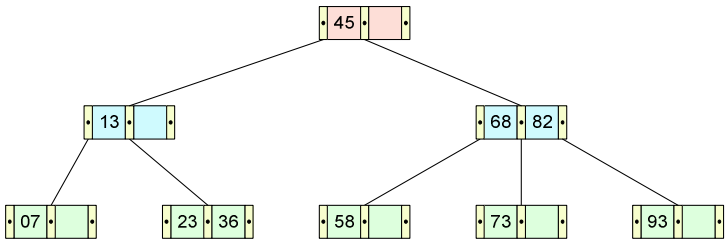


Abbildung 3.35: Suchen nach Bart Simpson (ID=93)

Bart Simpson hat ID=93: Zuerst die Wurzel durchsuchen, dann in den rechten inneren Knoten absteigen, dann diesen durchsuchen, und zum Schluss in den Blattknoten ganz rechts absteigen, welcher die ID=93 enthält.

j) *Welche Verfahren zur Wiederherstellung (Reorganisation) der B-Baum Struktur haben Sie kennen gelernt?*

1. Synchroner Vorgänger/Nachfolger

Diese Methode wenden wir an, wenn wir einen Schlüssel aus einem inneren Knoten oder aus der Wurzel (sofern sie Kinder hat) löschen wollen. Das Löschen aus inneren Knoten oder der Wurzel (sofern sie Kinder hat) ist verboten! Deshalb suchen wir den synchronen Vorgänger oder Nachfolger des zu löschenden Schlüssels. Dieser befindet sich per Definition in einem Blattknoten. Dann vertauschen wir den zu löschenden Schlüssel mit seinem synchronen Vorgänger oder Nachfolger. Anschliessend kann der zu löschende Schlüssel aus dem betreffenden Blattknoten gelöscht werden, unter Anwendung allfälliger weiterer Reparaturmassnahmen.

Ob wir den synchronen Vorgänger oder Nachfolger für die Vertauschung wählen, hängt vom **Füllgrad** der betroffenen Blattknoten ab. Für die Vertauschung wählen wir jenen Blattknoten, welcher nicht die minimale Anzahl Schlüssel enthält. Dadurch er-

sparen wir uns Reparaturmassnahmen nach dem Löschen. Falls beide Blattknoten mehr als die minimale Anzahl Schlüssel enthalten, spielt es keine Rolle, welchen Knoten wir verwenden. Das Gleiche gilt, wenn beide Blattknoten nur die minimale Anzahl Schlüssel enthalten. In diesem Fall müssen wir nach dem Löschen weitere Reparaturmassnahmen ergreifen, z.B. eine **Verschiebung (Rotation)** oder eine **Verschmelzung**.

2. **Knoten verschieben (rotieren)**

   Blattknoten: Wenn ein Blattknoten nach dem Löschen eines Schlüssels weniger als die geforderte minimale Anzahl Schlüssel enthält, dann ist dies ein illegaler Zustand, der eine Reparaturmassnahme erfordert. Wir versuchen, durch eine Rotation von Schlüsseln die B-Baum Bedingungen wieder herzustellen. Wir schauen, ob ein angrenzender Geschwisterknoten mehr als die minimale Anzahl Schlüssel enthält. Falls ja, können wir eine Rotation durchführen, falls nein, führen wir eine **Verschmelzung** durch.

   Die Rotation funktioniert so: wir holen den Schlüssel aus dem Vaterknoten zu uns (in das fehlerhafte Blatt), welcher sich zwischen uns und dem Geschwisterknoten befindet, welchen wir für die Rotation verwenden. Anschliessend verschieben wir einen Schlüssel des Geschwisterknotens in den gemeinsamen Vaterknoten. Um welchen Schlüssel es sich handelt, hängt davon ab, ob es sich um den linken oder den rechten Geschwisterknoten handelt. Im Falle des linken Geschwisterknotens verschieben wir den grössten Schlüssel in den Vaterknoten, im Falle des rechten Geschwisterknoten verwenden wir den kleinsten Schlüssel für die Verschiebung.

   Innerer Knoten: Eine Rotation auf inneren Knoten ist dann nötig, wenn ein innerer Knoten zu wenig Schlüssel enthält. Dies kann z.B. durch Verschmelzung von zwei Knoten zustande kommen, bei der ein Schlüssel mit zwei Kindknoten verschmolzen wird (wodurch ein Schlüssel aus dem inneren Knoten verschwindet, resp. gelöscht wird). Die Rotation an sich funktioniert genau gleich wie die Rotation von Blattknoten, mit einem kleinen Unterschied: der Kindknoten des Geschwisterknotens, welchen wir für die Rotation verwenden, wird ebenfalls mit verschoben! Dies ist nötig, weil sonst für das Kind kein Trennschlüssel mehr existieren würde. Der Kindknoten wird an den inneren Knoten angehängt, welcher ursprünglich zu wenig Schlüssel enthalten hatte.

3. **Knoten verschmelzen**

   Wenn ein Blattknoten nach dem Löschen eines Elementes die minimale Anzahl Schlüssel unterschreitet, versuchen wir als erstes, diesen illegalen Zustand mit einer **Rotation (Knoten verschieben)** zu beheben. Dasselbe versuchen wir, wenn ein innerer Knoten zuwenig Schlüssel enthält, z.B. nach einer zuvor ausgeführten Verschmelzung.

   Falls eine Rotation nicht möglich ist, weil die Geschwisterknoten nur die minimale Anzahl Knoten enthalten, führen wir eine Knotenverschmelzung durch. Die Knotenverschmelzung kann man als die **Umkehrfunktion des Knotenteilens** (beim Einfügen) ansehen.

   Ein Knoten wird mit einem Geschwisterknoten auf der linken oder rechten Seite des Knotens zu einem einzigen neuen Knoten verschmolzen. Der Geschwisterknoten muss exakt die minimale Anzahl erforderlicher Schlüssel enthalten – sonst würde man eine Verschiebung (Rotation) durchführen. Nach dem Verschmelzen der 2 Knoten verschieben wir den Trennschlüssel aus dem zuvor gemeinsamen Vaterknoten in den neu entstandenen Knoten. Somit enthält der neue Knoten die maximal mögliche Anzahl Schlüssel.

   Nach der Verschmelzung ist es möglich, dass der Vaterknoten zuwenig Schlüssel enthält – da wir ja ein Element in den Kindknoten verschoben haben. Wenn dieser Fall

eintritt, versuchen wir auf dem Vaterknoten eine Rotation mit dessen Geschwistern – und falls das fehlschlägt, wieder eine Verschmelzung. So machen wir weiter, bis wir schliesslich bei der Wurzel angelangt sind.

k) *Wieso können Schlüssel nur aus Blattknoten gelöscht werden?*

Wenn wir einen Schlüssel aus einem inneren Knoten (oder aus der Wurzel, falls sie Kinder hat) löschen, dann bedeutet dies, dass wir den Trennschlüssel löschen, welcher die Pointer auf der linken und der rechten Seite des gelöschten Schlüssels abtrennt. Das wäre ein illegaler Zustand, die Pointer müssen durch einen Schlüssel getrennt sein!

Deshalb ist das Löschen eines Schlüssels aus einem inneren Knoten so nicht möglich. Damit die Löschung trotzdem erfolgen kann, bedient man sich der Methode der Vertauschung des Schlüssels mit dem symmetrischen Vorgänger/Nachfolger.

l) *Wann erfolgt eine Verschiebung (Rotation) von Knoten?*

Eine Rotation kann auf Blattknoten oder auf inneren Knoten erfolgen. Sie erfolgt dann, wenn ein Knoten zuwenig Schlüssel enthält. Voraussetzung ist, dass ein Geschwisterknoten mehr als die minimale Anzahl Schlüssel enthält. Sonst würde man eine Verschmelzung durchführen.

m) *Wann erfolgt eine Verschmelzung von Knoten?*

Eine Verschmelzung kann auf Blattknoten oder auf inneren Knoten (und der Wurzel) erfolgen. Sie erfolgt dann, wenn eine Rotation nicht durchgeführt werden kann, d.h. wenn beide Geschwisterknoten lediglich die minimale Anzahl Schlüssel enthalten. Die Verschmelzung ist die "Umkehrfunktion" zu der Knotenteilung (beim Insert) – auf diese Weise schrumpft der B-Baum wieder.

n) Wie geht man vor, wenn man einen Schlüssel aus einem inneren Knoten löschen will?

Man sucht den symmetrischen Vorgänger oder den symmetrischen Nachfolger des zu löschenden Schlüssels, und vertauscht die beiden Schlüssel. Der symmetrische Vorgänger/Nachfolger befindet sich per Definition immer in einem Blattknoten. Nach der Vertauschung kann der Schlüssel einfach vom Blattknoten gelöscht werden. Falls der Blattknoten nach der Löschung weniger als die minimale Anzahl Schlüssel enthält, muss eine Rotation oder allenfalls eine Verschmelzung durchgeführt werden.

o) *Wie geht man vor, wenn man die Wurzel löschen will?*

Das Vorgehen ist gleich wie bei n)

3.12 Lernkontrolle: Hier können Sie überprüfen, ob Sie das Kapitel beherrschen

Aufgabe 1

Knobelaufgabe: Gegeben sei der folgende voll besetzte B-Baum der Ordnung d=3 (nach Knuth) und der Höhe h=1:

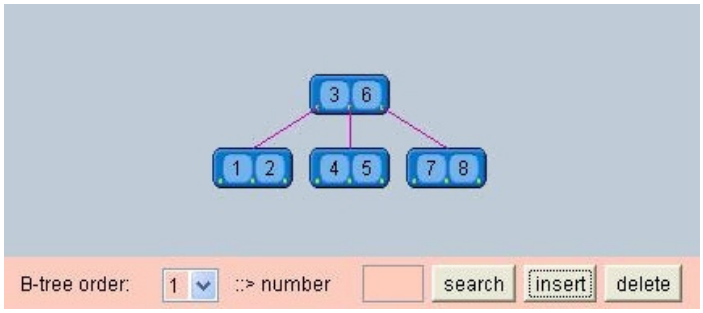


Abbildung 3.36: Voll besetzter B-Baum mit Höhe h=1

In welcher Reihenfolge müssen Sie die Schlüssel 1, 2, 3, 4, 5, 6, 7 und 8 in einen leeren B-Baum einfügen, um obige Abbildung zu erhalten? Probieren Sie es mit dem B-Baum Applet auf http://db.logging.ch aus, und begründen Sie!

Aufgabe 2

Gegeben sei der folgende B-Baum der Höhe h=2 und der Ordnung d=3 (nach Knuth):

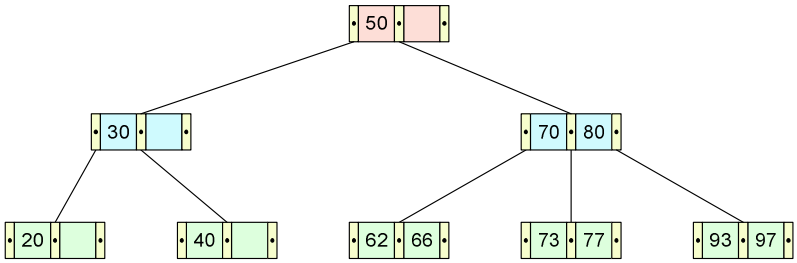


Abbildung 3.37: Löschen aus dem B-Baum

Löschen Sie alle 12 Schlüssel in folgender Reihenfolge aus dem B-Baum. Zeichnen Sie nach jedem Schritt den B-Baum auf, und beschreiben Sie, wie Sie vorgegangen sind.

Löschreihenfolge:

Schritt 01:	ID=20	Schritt 07:	ID= 73
Schritt 02:	ID= 50	Schritt 08:	ID= 62
Schritt 03:	ID= 40	Schritt 09:	ID= 70
Schritt 04:	ID= 66	Schritt 10:	ID= 93
Schritt 05:	ID= 80	Schritt 11:	ID= 30
Schritt 06:	ID= 97	Schritt 12:	ID= 77

3.13Lösungen zu den Lernkontrollfragen

Aufgabe 1

Knobelaufgabe: Gegeben sei der folgende voll besetzte B-Baum der Ordnung d=3 (nach Knuth) und der Höhe h=1. In welcher Reihenfolge müssen Sie die Schlüssel 1, 2, 3, 4, 5, 6, 7 und 8 in einen leeren B-Baum einfügen, um obige Abbildung zu erhalten? Probieren Sie es mit dem B-Baum Applet auf http://db.logging.ch aus, und begründen Sie!

Da neue Schlüssel immer in die Blätter eingefügt werden, muss man beim Einfügen auf die Reihenfolge achten, so dass die richtigen Schlüssel (3 und 6) zur Wurzel aufsteigen werden, wenn sich ein Knoten teilt.

Schritt 1: Einfügen der Schlüssel 1, 3 und 8
Der (Blatt-)Knoten teilt sich, die 3 steigt in die neue Wurzel auf

Schritt 2: Einfügen der Schlüssel 5 und 6
Der Blattknoten ganz rechts teilt sich, die 6 steigt in die Wurzel auf

Schritt 3: Einfügen der restlichen Schlüssel 2, 4 und 7
Dank der richtigen Reihenfolge können diese Schlüssel nun direkt eingefügt werden, ohne dass ein Knoten geteilt werden muss

Der schlechteste Fall stellt sich ein, wenn wir die Schlüssel der Reihe nach einfügen:

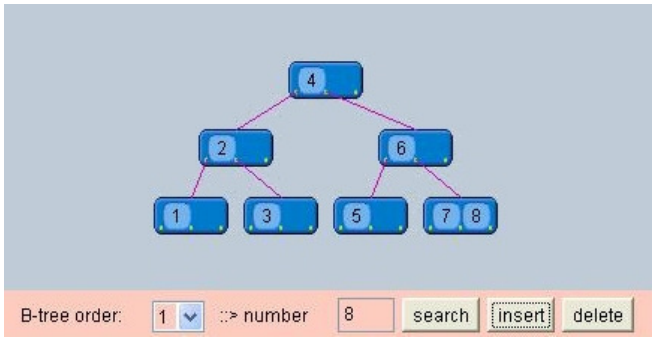


Abbildung 3.38: B-Tree nach dem sortierten Einfügen

Dieser B-Baum braucht zwar etwas mehr Speicherplatz, da er eine Ebene mehr hat als der voll besetzte B-Baum. In der Praxis ist das jedoch nicht unbedingt schlecht, vor allem dann, wenn wir viele Einfüge- und Löschoperationen erwarten. Der obige B-Baum hat den Vorteil, dass er Platz für weitere Schlüssel bietet, so dass wir in Zukunft weniger Knoten teilen müssen. Dies hat eine schnellere Geschwindigkeit zur Folge, natürlich auf Kosten des benötigten Speicherplatzes.

Aufgabe 2

Gegeben sei der folgende B-Baum der Höhe h=2 und der Ordnung d=3 (nach Knuth). Löschen Sie alle 12 Schlüssel in folgender Reihenfolge aus dem B-Baum. Zeichnen Sie nach jedem Schritt den B-Baum auf, und beschreiben Sie, wie Sie vorgegangen sind. Löschreihenfolge: 20, 50, 40, 66, 80, 97, 73, 62, 70, 93, 30, 77

Ausgangslage:

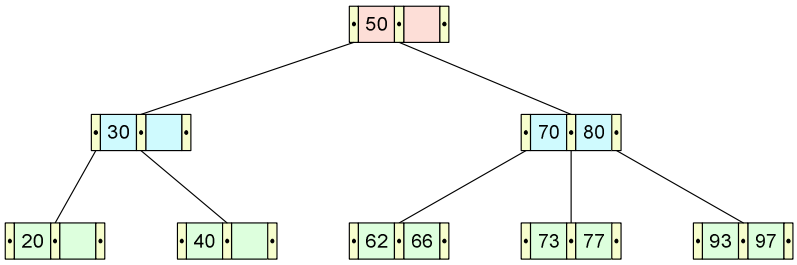


Abbildung 3.39: Löschen aus dem B-Baum – Ausgangslage

Schritt 01: Löschen von ID=20

- Rotation funktioniert nicht, da Geschwisterknoten nur 1 Schlüssel enthält
- Verschmelzung von ID=30 und ID=40 durchführen
- Innerer Knoten hat zuwenig Schlüssel
- Rotation durchführen: ID=70 wird in die Wurzel verschoben, ID=50 wird in den inneren Knoten verschoben
- Zu beachten: der Kindknoten [62,66] wird mitverschoben!

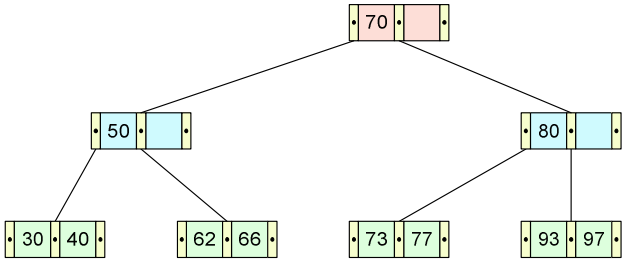


Abbildung 3.40: Löschen von ID=20

Schritt 02: Löschen von ID=50

- Symmetrischen Vorgänger/Nachfolger suchen: ID=40 oder ID=62
- Wir verwenden den symmetrischen Vorgänger (ID=40): Vertauschung von ID=50 und ID=40
- Danach kann ID=50 einfach aus dem Blattknoten gelöscht werden

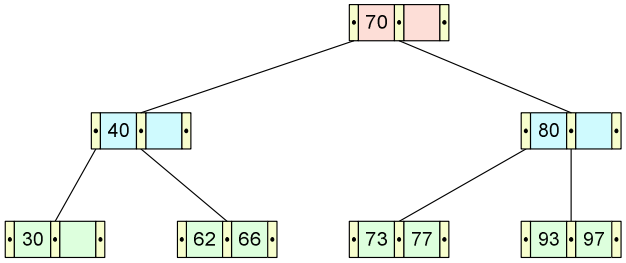


Abbildung 3.41: Löschen von ID=50

- Wir hätten auch den symmetrischen Nachfolger (ID=62) verwenden können. Dann wäre jetzt ID=62 an der Stelle von ID=40 → spielt keine Rolle

Schritt 03: Löschen von ID=40

- Symmetrischen Vorgänger/Nachfolger suchen: ID=30 oder ID=62
- Den symmetrischen Vorgänger (ID=30) können wir nicht verwenden, da sonst nach dem Löschen der Blattknoten leer ist
- Wir verwenden den symmetrischen Nachfolger (ID=62): Vertauschung von ID=40 und ID=62
- Danach kann ID=40 einfach aus dem Blattknoten gelöscht werden

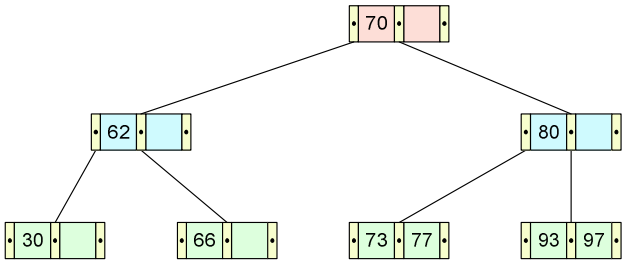


Abbildung 3.42: Löschen von ID=40

- Wir könnten auch ID=30 (symmetrischer Vorgänger) verwenden. Weil der Blattknoten danach leer ist, müssten wir eine Rotation durchführen. Das käme letzten Endes auf das Gleiche heraus. Durch Verwenden der ID=62 (symmetrischer Nachfolger) ersparen wir uns jedoch die zusätzliche Rotation.

Schritt 04: Löschen von ID=66

- Nach dem Löschen ist der Blattknoten leer → illegaler Zustand!
- Eine Verschiebung (Rotation) ist nicht möglich, da der Geschwisterknoten nur die minimale Anzahl Schlüssel (ID=30) enthält
- Wir führen eine Verschmelzung durch: Die Schlüssel mit der ID=30 und ID=62 werden zu einem neuen Blattknoten verschmolzen
- Nun ist der innere Knoten leer → illegaler Zustand!
- Eine Verschiebung (Rotation) ist nicht möglich, da der Geschwisterknoten nur die minimale Anzahl Schlüssel (ID=80) enthält
- Wir führen wiederum eine Verschmelzung durch: Die Schlüssel mit der ID=70 und ID=80 werden zu einem neuen Knoten verschmolzen
- Die Wurzel ist nun leer: sie kann einfach gelöscht werden
- Es ist eine neue Wurzel entstanden!

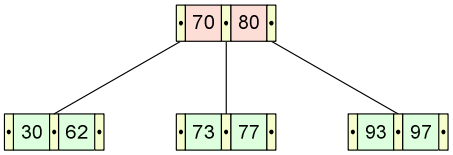


Abbildung 3.43: Löschen von ID=66

Schritt 05: Löschen von ID=80

- Symmetrischen Vorgänger/Nachfolger suchen: ID=77 oder ID=93
- Wir verwenden den symmetrischen Nachfolger (ID=93): Vertauschung von ID=80 und ID=93
- Danach kann ID=80 einfach aus dem Blattknoten gelöscht werden

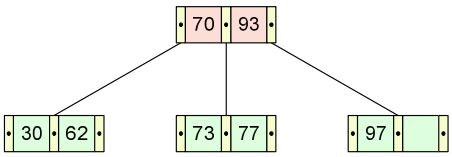


Abbildung 3.44: Löschen von ID=80

- Wir hätten auch den symmetrischen Vorgänger (ID=77) verwenden können. Dann wäre jetzt ID=77 an der Stelle von ID=93 → spielt keine Rolle

Schritt 06: Löschen von ID=97

- Nach der Löschung der ID=97 ist der Blattknoten leer → illegaler Zustand!
- Wir führen eine Rotation durch: wir holen ID=93 zu uns, und verschieben ID=77 des Geschwisterknotens in den Vaterknoten

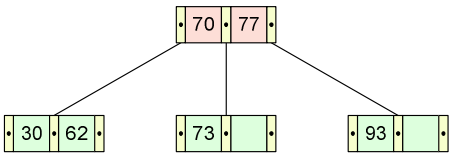


Abbildung 3.45: Löschen von ID=97

Schritt 07: Löschen von ID=73

- Nach der Löschung der ID=73 ist der Blattknoten leer → illegaler Zustand!
- Wir führen eine Rotation durch: wir holen ID=70 zu uns, und verschieben ID=62 des Geschwisterknotens in den Vaterknoten

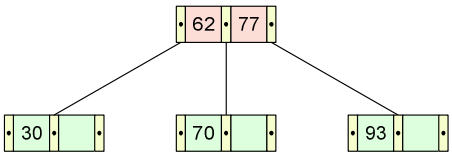


Abbildung 3.46: Löschen von ID=73

Schritt 08: Löschen von ID=62

- Symmetrischen Vorgänger/Nachfolger suchen: ID=30 oder ID=70
- Wir verwenden den symmetrischen Nachfolger (ID=70): Vertauschung von ID=62 und ID=70
- Danach löschen wir ID=62 aus dem Blattknoten
- Der Blattknoten ist nun leer → illegaler Zustand!
- Eine Rotation ist nicht möglich, da die Geschwisterknoten nur die minimale Anzahl Knoten enthalten (links: ID=30 / rechts: ID=93)
- Wir führen eine Verschmelzung durch. Es spielt keine Rolle, ob wir dazu den Geschwisterknoten auf der linken oder der rechten Seite verwenden. Wir verwenden den rechten Geschwisterknoten mit ID=93

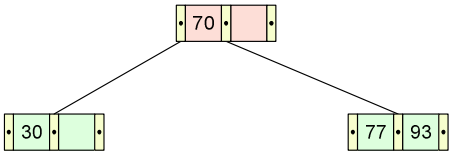


Abbildung 3.47: Löschen von ID=62

Schritt 09: Löschen von ID=70

- Symmetrischen Vorgänger/Nachfolger suchen: ID=30 oder ID=77
- Den symmetrischen Vorgänger (ID=30) können wir nicht verwenden, da sonst nach dem Löschen der Blattknoten leer ist
- Wir verwenden den symmetrischen Nachfolger (ID=77): Vertauschung von ID=70 und ID=77
- Danach kann ID=70 einfach aus dem Blattknoten gelöscht werden

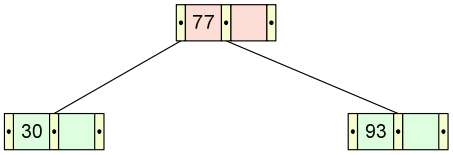


Abbildung 3.48: Löschen von ID=70

- Wir könnten auch ID=30 (symmetrischer Vorgänger) verwenden. Weil der Blattknoten danach leer ist, müssten wir eine Rotation durchführen. Das käme letzten Endes auf das Gleiche heraus. Durch Verwenden der ID=77 (symmetrischer Nachfolger) ersparen wir uns jedoch die zusätzliche Rotation.

Schritt 10: Löschen von ID=93

- Nach dem Löschen von ID=93 ist der Blattknoten leer → illegaler Zustand!
- Eine Rotation ist nicht möglich, da der Geschwisterknoten nicht die minimale Anzahl Knoten enthält (nur ID=30)
- Wir führen eine Verschmelzung durch: der leere Blattknoten wird mit seinem Geschwisterknoten (ID=30) und dem Trennschlüssel aus der Wurzel (ID=77) verschmolzen.
- Die Wurzel ist nun leer, sie kann einfach gelöscht werden
- Es ist eine neue (kinderlose) Wurzel entstanden!

Abbildung 3.49: Löschen von ID=93

Schritt 11: Löschen von ID=30

- Die Wurzel ist jetzt gleichzeitig auch ein Blatt!
- Die Löschung aus einem Blattknoten ist trivial

Abbildung 3.50: Löschen von ID=30

Schritt 11: Löschen von ID=77

- Die Löschung aus einem Blattknoten ist trivial
- Nach der Löschung von ID=77 ist der B-Baum leer

Abbildung 3.51: Löschen von ID=77

Kapitel 4: Indexierung – SELECT

4.1 Übersicht: Worum geht es?

In den letzten 2 Kapiteln haben Sie den binären (Such-)Baum und den B-Baum im Detail kennen gelernt. Binäre Bäume sind in der Informatik eine wichtige Datenstruktur, denen Sie immer wieder begegnen werden. Die B-Bäume im besonderen sind eine wichtige Grundlage für die Indexierung in Datenbanken, da die Indexe der meisten Datenbanksysteme auf diesem Baum beruhen – so auch die Indexe von Oracle.

In diesem Kapitel lernen Sie anhand von praktischen Beispielen, was es bedeutet, wenn ein Index benutzt wird – oder eben nicht. Sie werden feststellen, dass die Geschwindigkeitsunterschiede zum Teil dramatisch sein können.

4.2 Lernziele

In diesem Kapitel lernen Sie folgendes:

- Sie lernen, wie Sie die Ausführungszeit von einem Query messen
- Sie lernen, was ein Query-Plan ist, und wie Sie sich diesen Anzeigen lassen können
- Sie verstehen den Unterschied von einem Query mit Index und einem Query ohne Index
- Sie verstehen, wann die Verwendung eines Indexes angebracht ist – und wann nicht

4.3 Beispieltabelle `MYTEST`

Für die Übungen in diesem und dem nächsten Kapitel verwenden wir die Tabelle `MYTEST`, welche wie folgt definiert ist:

```
CREATE TABLE MYTEST (
    ID          NUMBER(19) PRIMARY KEY,
    TXT         VARCHAR2(50),
    FLAG        CHAR(1),
    FK          NUMBER(19)
);
```

Sie müssen diese Tabelle nicht kreieren, sie existiert bereits.

Die Kolonne `ID` bezeichnet den **Primary Key**, d.h. einen eindeutigen Schlüssel. Die meisten Datenbanken kreieren automatisch einen Index auf einer Kolonne, welche als `PRIMARY KEY` gekennzeichnet ist – so auch Oracle. Die Idee dahinter ist, dass nach dem Primärschlüssel oft gesucht werden muss, weshalb so eine Suche schnell sein soll. Auf der Kolonne `ID` existiert demnach ein Index.

Die Kolonne `TXT` ist ein **Freitextfeld** der Länge 50 Zeichen. Auf dieser Kolonne existiert kein Index, da wir dies im `CREATE TABLE` Statement nicht angegeben haben. Wenn wir nichts angeben, erzeugt Oracle keinen Index – ausser bei Primärschlüsseln, wie wir gerade gesehen haben.

Die Kolonne `FLAG` kann genau 1 Zeichen enthalten – `FLAG` darf auch `NULL` sein. Auch auf dieser Kolonne existiert kein Index. Die Kolonne wurde so initialisiert, dass sie für ungerade `ID`s ein `'U'` enthält, und für gerade `ID`s ein `'G'`.

Die Kolonne `FK` enthält einen **Foreign Key** (Fremdschlüssel). Auch auf dieser Kolonne existiert kein Index. Die Tabelle wurde so initialisiert, dass `FK` immer den gleichen Wert enthält wie `ID`.

Loggen Sie sich nun mit Ihrem Benutzernamen und Ihrem Passwort auf dem Login-Screen von iSQL*Plus ein. Falls Sie noch über keine Zugangsdaten verfügen, erhalten Sie diese von Ihrer Lehrperson.

Abbildung 4.1: iSQL*Plus Login Screen

Nach dem erfolgreichen Login sehen Sie den "Work Screen", welcher Ihnen vermutlich vom Unterricht her bereits bekannt ist.

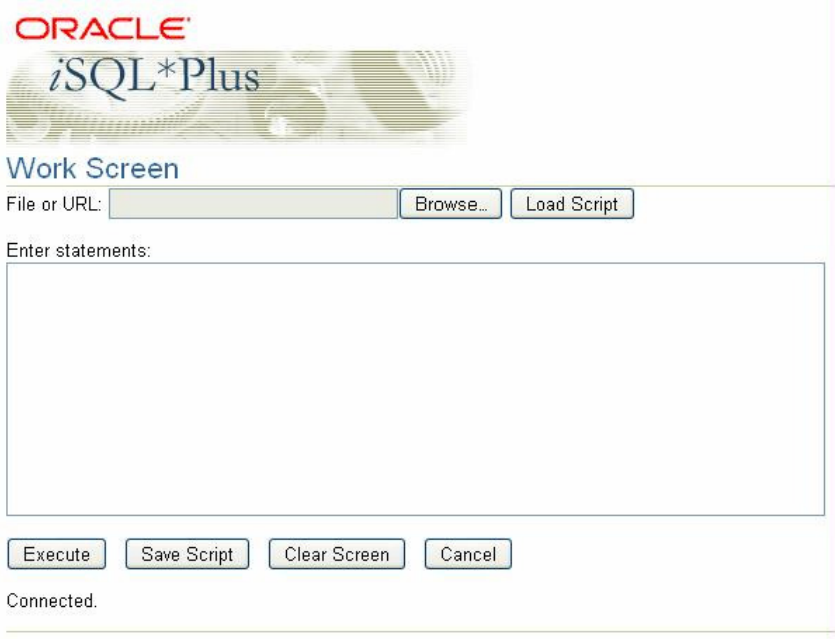


Abbildung 4.2: iSQL*Plus Work Screen

Nun sind wir bereit, um die SQL-Befehle abzusetzen!

Lassen Sie sich als erstes die Definition der Tabelle MYTEST anzeigen:

Enter statements:

```
DESCRIBE MYTEST
```

Sie sollten die folgende Ausgabe erhalten:

Output:

```
Name  Null?     Type
====  =====     ====
ID    NOT NULL  NUMBER(19)
TXT             VARCHAR2(30)
FLAG            CHAR(1)
FK              NUMBER(19)
```

Dies entspricht dem CREATE TABLE Statement auf der vorhergehenden Seite:

- Die Kolonne ID ist als NOT NULL markiert. Dies bedeutet, das in jeder Row für ID ein Wert gesetzt sein muss (ID darf nicht NULL sein). Zudem muss der Wert UNIQUE sein, d.h. eindeutig, wie es sich für einen Primärschlüssel gehört. Jeder Wert in der Kolonne ID darf demnach höchstens 1 Mal vorkommen.
- Die Kolonne TXT darf NULL sein, d.h. leer. Wir erkennen das daran, weil im Output bei Null? nichts steht.
- Die Kolonne FLAG darf ebenfalls NULL sein – ist sie in unserem Beispiel jedoch nie, da sie so initialisiert wurde, dass sie entweder ein 'G' für gerade IDs oder ein 'U' für ungerade IDs enthält.
- Die Kolonne FK darf ebenfalls NULL sein – ist sie in unserem Beispiel jedoch nicht, da FK immer den gleichen Wert wie ID enthält. Die Tabelle wurde so initialisiert.

Die Datentypen stimmen ebenfalls mit dem CREATE TABLE Statement überein.

Mit folgendem Befehl können Sie überprüfen, welche Indexe für die Tabelle MYTEST bereits existieren:

Enter statements:

```
SELECT i.index_name,
       i.index_type,
       c.table_name,
       c.column_name
  FROM all_indexes i,
       all_ind_columns c
 WHERE i.table_name='MYTEST'
   AND i.index_name=c.index_name
```

Sie sollten die folgende Ausgabe erhalten:

Output:

```
INDEX_NAME      INDEX_TYPE  TABLE_NAME  COLUMN_NAME
==========      ==========  ==========  ===========
SYS_C00118636   NORMAL      MYTEST      ID
```

Wir sehen, dass Oracle für die Kolonne `ID` (Primary Key) automatisch einen Index vom Typ "`NORMAL`" angelegt hat. "`NORMAL`" bedeutet, dass Oracle einen Index vom Typ B-Tree angelegt hat. B-Trees haben wir im Kapitel 3: B-Bäume kennen gelernt.

Oracle hat dem Index automatisch den Namen `SYS_C00118636` gegeben. Diesen Namen könnten wir verwenden, um den Index zu droppen (löschen):

```
DROP INDEX SYS_C00118636;
```

Versuchen Sie nicht, dieses Statement auszuführen – Sie werden eine Fehlermeldung erhalten, da Sie keine Berechtigung haben, diesen Index zu löschen.

4.4 Ausführungszeit

Ausgabe von Timing-Informationen

Damit wir messen können, wie schnell Oracle ein Statement ausführt, müssen wir Oracle anweisen, uns die Timing-Informationen auszugeben. Dies erreichen wir mit dem folgenden Befehl:

Enter statements:

```
SET TIMING ON
```

Wenn Sie die folgende Ausgabe erhalten, wird Oracle von nun an ausgeben, wie lange ein Statement gedauert hat.

Output:

```
SP2-0863: iSQL*Plus processing completed
```

Sie können die Timing-Information auch einschalten, indem Sie im iSQL*Plus auf "Preferences" klicken, und dann auf "Set system variables". Setzen Sie "Timing" auf "On", und klicken Sie auf OK. Diese Einstellung ist für die ganze Session gültig, oder solange, bis Sie die Timing-Information wieder ausschalten (via Menü oder mit `SET TIMING OFF`).

Zählen Sie nun die Anzahl Rows in der Tabelle `MYTEST`:

Enter statements:

```
SELECT COUNT(*) FROM MYTEST;
```

Sie sollten ungefähr so eine Ausgabe erhalten:

Output:

```
COUNT(*)
========
 5000000

Elapsed: 00:00:04.70
```

Wir sehen, dass die Tabelle `MYTEST` 5'000'000 Rows enthält, und dass das Statement fast 5 Sekunden gebraucht hat – kein Wunder, bei dieser grossen Anzahl Zeilen!

Wissenssicherung: Beantworten Sie die folgenden Fragen

a) Zählen Sie die Anzahl gerader und ungerader `IDs` in der Tabelle `MYTEST`! Verwenden Sie dazu die Kolonne `FLAG`. Messen und notieren Sie die dafür benötigte Zeit!

b) Zählen Sie die Anzahl gerader und ungerader `IDs` in der Tabelle `MYTEST`! Verwenden Sie dazu die Kolonne `ID`. Messen und notieren Sie die dafür benötigte Zeit!

c) Zählen Sie die Anzahl gerader und ungerader `IDs` in der Tabelle `MYTEST`! Verwenden Sie dazu die Kolonne `FK`. Messen und notieren Sie die dafür benötigte Zeit!

d) Wählen Sie eine zufällige `ID` ($1 \leq$ `ID` $\leq 5'000'000$) und selektieren Sie entsprechende Row. Wie lange dauerte das Statement?

e) Wählen Sie eine andere zufällige `ID` ($1 \leq$ `ID` $\leq 5'000'000$) und selektieren Sie die entsprechende Row. Verwenden Sie jedoch nicht `ID`, sondern `FK` für das Selektionskriterium. Wie lange dauerte das Statement?

f) Vergleichen Sie die Messungen mit den Messungen Ihres Nachbarn. Haben Sie ähnliche Ergebnisse gemessen? Wie erklären Sie sich allfällige Abweichungen? Wieso dauert das Statement in Aufgabe e) länger als das Statement in Aufgabe d)?

4.5 Query-Plan

Bevor die Datenbank ein Query ausführt, legt sie sich eine Strategie zurecht, wie sie die gewünschten Informationen am schnellsten finden kann. Diese Strategie nennt man **Query-Plan** oder **Execution Plan** – oder einfach kurz **Plan**. Dieser Plan sieht je nach Query und Tabelle unterschiedlich aus. Falls es auf einer Kolonne einen Index gibt, wird der Query-Plan anders aussehen, als wenn der Index fehlen würde.

Da die Erstellung eines Execution Plans für die Datenbank Arbeit bedeutet, merkt sie sich die Query-Pläne für eine gewisse Zeit. Wenn in Zukunft ein ähnliches Query abgesetzt wird, dann kann die Datenbank auf den früheren Query-Plan zurück greifen, und ihn für die Abfrage verwenden. So spart Oracle Zeit!

Den Query-Plan für ein Statement können Sie sich anzeigen lassen. Dazu müssen Sie einmalig die spezielle Tabelle `PLAN_TABLE` in Ihrem Datenbank-Schema erstellen. Oracle benötigt diese Tabelle, um den Query-Plan und statistische Informationen des gerade ausgeführten Queries abzuspeichern, um das Query zu analysieren, und um Ihnen letzten Endes den Query-Plan anzeigen zu können.

Um die Tabelle `PLAN_TABLE` zu erstellen, führen Sie das `CREATE TABLE` Statement auf der folgenden Seite aus. Sie finden das Script auch unter http://db.logging.ch, unter dem Link "Script plan-table.sql". Diese Tabelle müssen Sie nur 1 Mal erstellen – von nun an unterstützt Ihr Datenbank-Schema die Anzeige von Query-Plänen.

Damit Sie nun den Query-Plan eines Statements zu Gesicht bekommen, müssen Sie Oracle noch mitteilen, dass Sie den Query-Plan sehen wollen. Dies erreichen Sie mit dem folgendem Befehl:

Enter statements:

```
SET AUTOTRACE ON
```

Wenn Sie die folgende Ausgabe erhalten, wird Ihnen von nun an für jedes Query der Query-Plan angezeigt:

Output:

```
SP2-0863: iSQL*Plus processing completed
```

Sie können die Ausgabe der Query-Pläne auch einschalten, indem Sie im iSQL*Plus auf "Preferences" klicken, und dann auf "Set system variables". Setzen Sie "Autotrace" auf "On", machen Sie ein Häckchen bei "Explain" und "Statistics", und klicken Sie anschliessend auf OK. Diese Einstellung ist für die ganze Session gültig, oder solange, bis Sie die Ausgabe der Query-Pläne wieder ausschalten (via Menü oder mit `SET AUTOTRACE OFF`).

Erstellung der Tabelle `PLAN_TABLE`

Enter statements:

```
CREATE TABLE PLAN_TABLE (
    STATEMENT_ID                    VARCHAR2(30),
    TIMESTAMP                       DATE,
    REMARKS                         VARCHAR2(80),
    OPERATION                       VARCHAR2(30),
    OPTIONS                         VARCHAR2(30),
    OBJECT_NODE                     VARCHAR2(128),
    OBJECT_OWNER                    VARCHAR2(30),
    OBJECT_NAME                     VARCHAR2(30),
    OBJECT_INSTANCE                 NUMBER(38),
    OBJECT_TYPE                     VARCHAR2(30),
    OPTIMIZER                       VARCHAR2(255),
    SEARCH_COLUMNS                  NUMBER,
    ID                              NUMBER(38),
    PARENT_ID                       NUMBER(38),
    POSITION                        NUMBER(38),
    COST                            NUMBER(38),
    CARDINALITY                     NUMBER(38),
    BYTES                           NUMBER(38),
    OTHER_TAG                       VARCHAR2(255),
    PARTITION_START                 VARCHAR2(255),
    PARTITION_STOP                  VARCHAR2(255),
    PARTITION_ID                    NUMBER(38),
    OTHER                           LONG,
    DISTRIBUTION                    VARCHAR2(30)
);
```

Output:

```
Table created.
```

Führen Sie nun das folgende Statement aus, welches die Row mit der `ID=2359124` selektiert. Die Wahl von ID ist zufällig:

Enter statements:

```
SELECT * FROM MYTEST WHERE ID=2359124
```

Die Ausgabe sollte wie folgt aussehen:

Output:

```
ID       TXT                                        F  FK
==       ===                                        =  ==
2359124  TEST #115912: wCMazLrSVfMOwqBSzGQXrqk  G  2359124

Elapsed: 00:00:00.19

Execution Plan
----------------------------------------------------------
  0   SELECT STATEMENT Optimizer=CHOOSE
  1 0   TABLE ACCESS (BY INDEX ROWID) OF 'MYTEST'
  2 1     INDEX (UNIQUE SCAN) OF 'SYS_C00118633' (UNIQUE)
```

- Zeile 0 zeigt uns an, dass es sich beim Statement um einen `SELECT`-Befehl handelt.
- Zeile 1 zeigt uns an, dass ein Zugriff auf die Tabelle `MYTEST` erfolgt. Der Zugriff erfolgt via Index.
- Zeile 2 zeigt uns an, dass für die Suche der Index `SYS_C00118633` verwendet wird. Das ist der gleiche Index, den wir früher in diesem Kapitel bereits angetroffen haben. Es handelt sich um den Index der Kolonne `ID`, d.h. um den Primary Key, welcher in der `WHERE`-Clause (`WHERE`-Klausel) des `SELECT`-Befehls vorkommt.

Wenn wir nun ein ähnliches Query ausführen, z.B.

```
SELECT * FROM MYTEST WHERE ID=1234567;
```

dann erkennt Oracle, dass es sich um ein ähnliches Statement handelt. Oracle kann dann den gleichen Query-Plan wieder verwenden, und muss ihn nicht neu erstellen. Das bedeutet eine Geschwindigkeitssteigerung!

Zum Vergleich führen wir nun die gleiche Abfrage nochmals aus, allerdings unter Verwendung der Kolonne `FK` anstatt der Kolonne `ID`:

Enter statements:

```
SELECT * FROM MYTEST WHERE FK=2359124
```

Die Ausgabe sieht nun etwas anders aus:

Output:

```
ID       TXT                                          F  FK
==       ===                                          =  ==
2359124  TEST #115912: wCMazLrSVfMOwqBSzGQXrqk  G  2359124

Elapsed: 00:00:04.54

Execution Plan
----------------------------------------------------------
  0   SELECT STATEMENT Optimizer=CHOOSE
  1 0   TABLE ACCESS (FULL) OF 'MYTEST'
```

Der Query-Plan ist nun kürzer, was jedoch nicht heisst, dass er schneller oder besser ist!

- Zeile 0 zeigt uns an, dass es sich beim Statement wieder um einen `SELECT`-Befehl handelt.
- Zeile 1 zeigt uns an, dass ein Zugriff auf die Tabelle `MYTEST` erfolgt. Der Zugriff erfolgt jedoch nicht via Index, da es auf der Kolonne `FK` keinen Index gibt. Der Datenbank bleibt nichts anderes übrig, als die gesamte Kolonne zu durchsuchen. Dies nennt man einen **Full Table Scan** – deshalb steht `TABLE ACCESS (FULL)` im Query-Plan.

Full Table Scans sind nicht immer schlecht. Bei kleinen Tabellen ist das für Oracle kein Problem. Bei Tabellen mit mehr als 50'000 Rows muss man jedoch versuchen, Full Table Scans zu vermeiden, sonst leidet die Performance! Dies sehen wir auch daran, dass das Query ca. 4.5 Sekunden gebraucht hat, um die gleichen Informationen abzurufen wie das Query zuvor, welches nur ca. 2/10 Sekunden gebraucht hat. D.h. in diesem Fall ist das Query ohne Index ca. **20 Mal langsamer** als das Query mit Index. Ein Index zahlt sich also aus!

Wissenssicherung: Beantworten Sie die folgenden Fragen

g) Was ist ein Query-Plan, und wozu braucht's den überhaupt?

h) Das Query **`SELECT * FROM MYTEST WHERE ID=2359124 OR ID=1923472;`** liefert den folgenden Query-Plan:

```
0   SELECT STATEMENT Optimizer=CHOOSE
1 0   CONCATENATION
2 1     TABLE ACCESS (BY INDEX ROWID) OF 'MYTEST'
3 2       INDEX (UNIQUE SCAN ) OF 'SYS_C00118633' (UNIQUE)
4 1     TABLE ACCESS (BY INDEX ROWID) OF 'MYTEST'
5 4       INDEX (UNIQUE SCAN ) OF 'SYS_C00118633' (UNIQUE)
```

Erklären Sie diesen Query-Plan!

i) Wie könnten Sie das Query **`SELECT * FROM MYTEST WHERE FK=4253978;`** schneller machen?

j) Wie sieht der Query-Plan des folgenden Statements aus?

```
SELECT * FROM MYTEST WHERE ID=2359124 OR FK=1923472;
```

k) Das Query **`SELECT COUNT(*) FROM MYTEST WHERE FLAG='G';`** ist langsam. Könnten wir das Query schneller machen, wenn wir auf der Kolonne FLAG einen Index anlegen? Weshalb?

4.6 Zusammenfassung

In diesem Kapitel haben Sie gelernt, wie Sie die Ausführungszeit von einem Query messen. Sie haben gesehen, dass die Ausführungszeiten variieren können – besonders dann, wenn ein Index verwendet wird, oder eben nicht! Sie haben zudem gelernt, dass sich die Datenbank für die Ausführung eines Queries einen Schlachtplan zurecht legt – den Query Plan. Sie können sich die Query-Pläne anzeigen lassen, und verstehen es, diese zu interpretieren. Zudem haben wir gesehen, dass ein Index nicht immer das Allerheilmittel gegen langsame Queries ist. Wenn eine Kolonne nur wenig unterschiedliche Werte enthält, bringt ein Index nicht das, was wir erwarten.

Im folgenden Kapitel beschäftigen wir uns mit etwas komplexeren Queries. Dann werden Sie sehen, dass die Verwendung eines Indexes wirklich sehr nützlich sein kann – und manchmal sogar zwingend ist.

4.7 Lösungen zu den Wissenssicherungsfragen

a) Zählen Sie die Anzahl gerader und ungerader IDs in der Tabelle MYTEST! Verwenden Sie dazu die Kolonne FLAG. Messen und notieren Sie die dafür benötigte Zeit!

Enter statements:

```
SELECT COUNT(*) FROM MYTEST WHERE FLAG='G';
SELECT COUNT(*) FROM MYTEST WHERE FLAG='U';
```

Sie sollten ungefähr so eine Ausgabe erhalten:

Output:

```
COUNT(*)
========
 2500000

Elapsed: 00:00:04.74

COUNT(*)
========
 2500000

Elapsed: 00:00:04.52
```

b) Zählen Sie die Anzahl gerader und ungerader IDs in der Tabelle MYTEST! Verwenden Sie dazu die Kolonne ID. Messen und notieren Sie die dafür benötigte Zeit!

Enter statements:

```
SELECT COUNT(*) FROM MYTEST WHERE MOD(ID,2) = 0;
SELECT COUNT(*) FROM MYTEST WHERE MOD(ID,2) = 1;
```

Sie sollten ungefähr so eine Ausgabe erhalten:

Output:

```
COUNT(*)
========
 2500000

Elapsed: 00:00:05.70

COUNT(*)
========
 2500000

Elapsed: 00:00:05.65
```

c) *Zählen Sie die Anzahl gerader und ungerader `IDs` in der Tabelle `MYTEST`! Verwenden Sie dazu die Kolonne `FK`. Messen und notieren Sie die dafür benötigte Zeit!*

Enter statements:

```
SELECT COUNT(*) FROM MYTEST WHERE MOD(FK,2) = 0;
SELECT COUNT(*) FROM MYTEST WHERE MOD(FK,2) = 1;
```

Sie sollten ungefähr so eine Ausgabe erhalten:

Output:

```
COUNT(*)
========
 2500000

Elapsed: 00:00:05.93

COUNT(*)
========
 2500000

Elapsed: 00:00:06.12
```

d) *Wählen Sie eine zufällige `ID` ($1 \leq ID \leq 5'000'000$) und selektieren Sie entsprechende Row. Wie lange dauerte das Statement?*

Enter statements:

```
SELECT * FROM MYTEST WHERE ID=1532978
```

Sie sollten ungefähr so eine Ausgabe erhalten:

Output:

```
ID       TXT                          F  FK
==       ===                          =  ==
1532978  TEST #332978: ZOWnoUjtFVKw  G  1532978

Elapsed: 00:00:00.09
```

e) *Wählen Sie eine andere zufällige `ID` ($1 \leq ID \leq 5'000'000$) und selektieren Sie die entsprechende Row. Verwenden Sie jedoch nicht `ID`, sondern `FK` für das Selektionskriterium. Wie lange dauerte das Statement?*

Enter statements:

```
SELECT * FROM MYTEST WHERE FK=3982714
```

Sie sollten ungefähr so eine Ausgabe erhalten:

Output:

```
ID       TXT                                    F  FK
==       ===                                    =  ==
3982714  TEST #282714: NSmzwIlNoFaLwLJEqjYXj  G  3982714

Elapsed: 00:00:04.66
```

f) Vergleichen Sie die Messungen mit den Messungen Ihres Nachbarn. Haben Sie ähnliche Ergebnisse gemessen? Wie erklären Sie sich allfällige Abweichungen? Wieso dauert das Statement in Aufgabe e) länger als das Statement in Aufgabe d)?

Geringfügige Abweichungen sind normal. Sie hängen mit der Systemauslastung des Datenbank-Servers zusammen, und allenfalls damit, welches Query zuvor ausgeführt worden ist. Oracle merkt sich nämlich die ausgeführten Queries, so dass eine gleiche oder ähnliche Abfrage beim zweiten Mal schneller beantwortet werden kann.

Man möchte meinen, dass die Queries in Aufgabe d) und e) in etwa gleich schnell sein sollten, da sie sehr ähnlich sind: es wird eine einzelne `ID` selektiert. In Tat und Wahrheit ist das Query in Aufgabe e) viel langsamer als das Query in Aufgabe d). Der Grund dafür ist, weil es auf der Kolonne `FK` keinen Index gibt, auf der Kolonne `ID` hingegen schon. Bei Aufgabe e) muss die Datenbank die ganze Tabellenkolonne durchsuchen, um die richtige `ID` (`FK`) zu finden. Bei Aufgabe d) geht das wesentlich schneller, da die Datenbank den vorhandenen Index `SYS_C00118636` für die Suche verwenden kann. Wir haben ja im Kapitel 3: B-Bäume gesehen, dass die Suche in logarithmischer Zeit möglich ist, wenn die Schlüssel sich sortiert in einem Index befinden.

g) Was ist ein Query-Plan, und wozu braucht's den überhaupt?

Ein Query-Plan ist die Strategie der Datenbank, wie sie ein Query verarbeiten wird – sozusagen der Schlachtplan. Query-Pläne werden aufbewahrt, so dass sie in Zukunft wieder verwendet werden können, falls ein ähnliches Query abgesetzt wird. Ein Query-Plan ist nur für ein bestimmtes Statement gültig (ausser für ähnliche Statements) – für ein komplett anderes Statement sieht der Query-Plan auch komplett anders aus. Vor jeder Ausführung eines Queries erstellt die Datenbank einen Execution Plan (Query-Plan), oder sie sucht nach einem bereits vorhandenen Plan, den sie sich früher mal gemerkt hatte. Das Caching der Query-Pläne dient der Performance-Steigerung, da die Erstellung eines Query-Planes auch Zeit benötigt!

*h) Das Query **SELECT * FROM MYTEST WHERE ID=2359124 OR ID=1923472;** liefert den folgenden Query-Plan:*

```
0   SELECT STATEMENT Optimizer=CHOOSE
1 0   CONCATENATION
2 1     TABLE ACCESS (BY INDEX ROWID) OF 'MYTEST'
3 2       INDEX (UNIQUE SCAN ) OF 'SYS_C00118633' (UNIQUE)
4 1     TABLE ACCESS (BY INDEX ROWID) OF 'MYTEST'
5 4       INDEX (UNIQUE SCAN ) OF 'SYS_C00118633' (UNIQUE)
```

Erklären Sie diesen Query-Plan!

- Zeile 0 zeigt uns an, dass es sich beim Query um ein `SELECT`-Statement handelt.
- Zeile 1 zeigt uns an, dass 2 Resultate miteinander verbunden (konkateniert) werden: die Abfrage nach `ID=2359124` und `ID=1923472`. Das `CONCATENATION` entspricht dem `OR` im `SELECT`-Befehl.

- Zeile 2 zeigt uns an, dass auf die Tabelle `MYTEST` zugegriffen wird, und zwar unter Verwendung eines Indexes.
- Zeile 3 zeigt uns an, dass der Index `SYS_C00118633` verwendet wird – der Index der Kolonne `ID`.
- Die Zeilen 4 und 5 entsprechen den Zeilen 2 und 3: die Zeilen 2 und 3 stellen die Suche nach `ID=2359124` dar, die Zeilen 4 und 5 die Suche nach `ID=1923472` – oder umgekehrt. Die Suchreihenfolge können wir im Query-Plan nicht erkennen, doch das spielt auch keine Rolle.

Die zweite Kolonne von Zahlen gibt die Einrückungstiefe der Zeilen an. Dies zeigt uns an, was die Reihenfolge der Statements ist.

i) Wie könnten Sie das Query **`SELECT * FROM MYTEST WHERE FK=4253978;`** schneller machen?

Sie könnten einen Index auf der Kolonne FK anlegen:

```
CREATE INDEX MYTEST_FK ON MYTEST(FK);
```

Den Indexnamen können Sie frei wählen – er darf in Ihrem Datenbank-Schema jedoch nur 1 Mal vorkommen. Deshalb können wir den Indexnamen z.B. aus dem Tabellennamen und dem Kolonnennamen zusammensetzen. So können wir sicher sein, dass der Indexname noch nicht verwendet wird. Dieser Indexname ist auch beschreibender als z.B. der Indexname **`SYS`**`_C00118633` , welcher von Oracle (vom **`SYS`**tem) automatisch vergeben worden ist.

j) Wie sieht der Query-Plan des folgenden Statements aus?

`SELECT * FROM MYTEST WHERE ID=2359124 OR FK=1923472;`

Der Query-Plan sieht wie folgt aus:

```
0   SELECT STATEMENT Optimizer=CHOOSE
1 0   TABLE ACCESS (FULL) OF 'MYTEST'
```

Das wird Sie nun vielleicht etwas enttäuschen... Man möchte meinen, dass die Datenbank für die Suche nach `ID=2359124` den Index verwendet, und für die Suche nach `FK=1923472` einen Full Table Scan macht. Die Datenbank hat sich jedoch entschieden, auch für die Suche nach `ID=2359124` einen Full Table Scan zu machen. Wieso macht sie das?

Die Datenbank hat gemerkt, dass sie für die Suche nach `FK=1923472` sowieso die ganze Tabelle durchsuchen muss. Offenbar ist die Datenbank schneller, wenn sie während dem Full Table Scan gleich auch nach der `ID=2359124` sucht.

k) Das Query ***`SELECT COUNT(*) FROM MYTEST WHERE FLAG='G';`*** *ist langsam. Könnten wir das Query schneller machen, wenn wir auf der Kolonne FLAG einen Index anlegen? Weshalb?*

Wir können einen Index auf der Kolonne `FLAG` anlegen, doch vermutlich wird das Query dadurch nicht wesentlich schneller. Grund dafür ist die Art der Daten, welche sich in der Kolonne befinden: es gibt nur 2 verschiedene Werte! 2'500'000 Mal `'G'` und 2'500'000 Mal `'U'`. Da hilft ein Index nicht viel weiter, weil die Daten zuwenig selektiv sind.

4.8 Lernkontrolle: Hier können Sie überprüfen, ob Sie das Kapitel beherrschen

Aufgabe 1

Gegeben sei das folgende Query:

```
SELECT * FROM MYTEST WHERE ID BETWEEN 1000001 AND 1000005;
```

Wie sieht Ihrer Meinung nach der Query-Plan aus? Überprüfen Sie Ihre Vermutung, indem Sie das Query ausprobieren und sich den Query-Plan anzeigen lassen.

Aufgabe 2

Nehmen Sie an, dass es auf der Kolonne `TXT` einen Index gibt.

a) Gegeben sei das folgende Query:

```
SELECT * FROM MYTEST WHERE TXT LIKE 'TEST #500000%';
```

welches das folgende Resultat liefert:

```
ID         TXT                            F    FK
==         ===                            =    ==
500000     TEST #500000: vbfHgsVriLOF     G    500000
1700000    TEST #500000: vbfHgsVriLOF     G    1700000
3000000    TEST #500000: vbfHgsVriLOF     G    3000000
4200000    TEST #500000: vbfHgsVriLOF     G    4200000
```

Wird die Datenbank den Index verwenden, oder wird sie einen Full Table Scan durchführen? Wie sieht der Query-Plan Ihrer Meinung nach aus?

b) Gegeben sei das folgende Query:

```
SELECT * FROM MYTEST WHERE TXT LIKE '%riLOF';
```

Die Ausgabe sieht dabei gleich aus wie bei a)

Wird die Datenbank den Index verwenden, oder wird sie einen Full Table Scan durchführen? Wie sieht der Query-Plan Ihrer Meinung nach aus?

4.9 Lösungen zu den Lernkontrollfragen

Aufgabe 1

Gegeben sei das folgende Query: ***SELECT * FROM MYTEST WHERE ID BETWEEN 1000001 AND 1000005;*** *Wie sieht Ihrer Meinung nach der Query-Plan aus? Überprüfen Sie Ihre Vermutung, indem Sie das Query ausprobieren und sich den Query-Plan anzeigen lassen.*

Die obige Abfrage könnte man auch schreiben als:

```
SELECT *
  FROM MYTEST
 WHERE ID=1000001
    OR ID=1000002
    OR ID=1000003
    OR ID=1000004
    OR ID=1000005;
```

Bei der Wissenssicherungsfrage h) haben wir gesehen, wie ein Query-Plan aussieht, wenn ein OR vorkommt. Demnach könnte man vermuten, dass der Query-Plan für das obige Statement irgendwie so aussieht:

```
 0      SELECT STATEMENT Optimizer=CHOOSE
 1   0    CONCATENATION
 2   1      TABLE ACCESS (BY INDEX ROWID) OF 'MYTEST'
 3   2        INDEX (UNIQUE SCAN ) OF 'SYS_C00118633' (UNIQUE)
 4   1      TABLE ACCESS (BY INDEX ROWID) OF 'MYTEST'
 5   4        INDEX (UNIQUE SCAN ) OF 'SYS_C00118633' (UNIQUE)
 6   1      TABLE ACCESS (BY INDEX ROWID) OF 'MYTEST'
 7   6        INDEX (UNIQUE SCAN ) OF 'SYS_C00118633' (UNIQUE)
 8   1      TABLE ACCESS (BY INDEX ROWID) OF 'MYTEST'
 9   8        INDEX (UNIQUE SCAN ) OF 'SYS_C00118633' (UNIQUE)
10   1      TABLE ACCESS (BY INDEX ROWID) OF 'MYTEST'
11  10        INDEX (UNIQUE SCAN ) OF 'SYS_C00118633' (UNIQUE)
```

Für die Abfrage mit OR stimmt das auch, der Query-Plan sieht genau so aus. Die Abfrage mit BETWEEN hat jedoch einen anderen Query-Plan:

```
 0    SELECT STATEMENT Optimizer=CHOOSE
 1  0   TABLE ACCESS (BY INDEX ROWID) OF 'MYTEST'
 2  1     INDEX (RANGE SCAN) OF 'SYS_C00118633' (UNIQUE)
```

In Zeile 2 sehen wir, dass die Datenbank einen **Range Scan** durchführt. Den Range Scan hatten wir beim **B^+-Baum** kurz angesprochen: die Blattknoten sind miteinander verlinkt, so dass ein Range Scan sehr effizient ist. Bei der Abfrage mit OR ist ein Range Scan nicht möglich, da die Datenbank nicht weiss, dass die IDs direkt hintereinander liegen. Für das Erstellen des Query-Planes werden die Daten nämlich nicht berücksichtigt, da Oracle diese noch nicht kennt. Deshalb ist hier die Abfrage mit BETWEEN schneller.

Den gleichen Query-Plan hat übrigens die folgende Abfrage:

```
SELECT *
  FROM MYTEST
 WHERE ID >= 1000001
   AND ID <= 1000005;
```

Das bedeutet, dass die Abfrage mit BETWEEN und <= / >= gleich schnell sind. In Tat und Wahrheit übersetzt Oracle für dem Ausführen des Queries die erste Schreibweise in die zweite.

Aufgabe 2

Nehmen Sie an, dass es auf der Kolonne `TXT` einen Index gibt.

a) Gegeben sei das folgende Query:

```
SELECT * FROM MYTEST WHERE TXT LIKE 'TEST #500000%';
```

welches das folgende Resultat liefert:

```
ID        TXT                          F   FK
==        ===                          =   ==
500000    TEST #500000: vbfHgsVriLOF   G   500000
1700000   TEST #500000: vbfHgsVriLOF   G   1700000
3000000   TEST #500000: vbfHgsVriLOF   G   3000000
4200000   TEST #500000: vbfHgsVriLOF   G   4200000
```

Wird die Datenbank den Index verwenden, oder wird sie einen Full Table Scan durchführen? Wie sieht der Query-Plan Ihrer Meinung nach aus?

b) Gegeben sei das folgende Query:

```
SELECT * FROM MYTEST WHERE TXT LIKE '%riLOF';
```

Die Ausgabe sieht dabei gleich aus wie bei a)

Wird die Datenbank den Index verwenden, oder wird sie einen Full Table Scan durchführen? Wie sieht der Query-Plan Ihrer Meinung nach aus?

Wenn die Datenbank für die Kolonne TXT einen Index aufbauen muss, geht sie wie folgt vor: sie wandelt alle Buchstaben in Zahlen um (z.B. ASCII-Zeichensatz). Dann behandelt sie den String wie eine Folge von Zahlen. Diese Zahlen werden dann für die Indexierung verwendet.

Dies bedeutet, dass die Zeichenketten `'ABCW'`, `'ABCX'`, `'ABCY'` und `'ABCZ'` quasi nebeneinander liegen, d.h. man kann sie sortieren. Die Zeichen `'W'`, `'X'`, `'Y'` und `'Z'` sind sozusagen Kinder von `'C'`, `'C'` ist ein Kind von `'B'`, und `'B'` ist ein Kind von `'A'` (in B-Baum Schreibweise). Der leere String `''` ist dabei die Wurzel.

Wenn man also nach `'TEST #500000%'` suchen will, dann befinden sich alle weiteren Zeichen (`'%'`) unter dem gleichen Knoten (`'TEST #500000'`). Der Knoten (`'TEST #500000'`) ist demnach die Wurzel für einen Teilbaum, der alle Strings enthält, welche mit `'TEST #500000'` beginnen. Folglich kann man in diesem Teilbaum das kleinste und das grösste Element suchen (im B-Baum entweder immer links oder immer rechts in die Kindknoten absteigen), und dann einen **Range Scan** durchführen.

a) Genau das macht auch Oracle. Der Query-Plan sieht wie folgt aus:

```
0   SELECT STATEMENT Optimizer=CHOOSE
1 0    TABLE ACCESS (BY INDEX ROWID) OF 'MYTEST'
2 1       INDEX (RANGE SCAN) OF 'MYTEST_TXT' (NON-UNIQUE)

Elapsed: 00:00:00.13
```

Die Datenbank kann den Index verwenden, das Query ist sehr schnell! In Zeile 2 sehen wir, dass die Datenbank einen Range Scan auf dem Index `MYTEST_TXT` macht. `MYTEST_TXT` ist der Name des Indexes auf der Kolonne `TXT`.

b) Der Query-Plan sieht wie folgt aus:

```
0   SELECT STATEMENT Optimizer=CHOOSE
1 0    TABLE ACCESS (FULL) OF 'MYTEST'

Elapsed: 00:00:06.54
```

Die Datenbank kann den Index nicht verwenden, deshalb ist das Query sehr langsam!

Der Suchstring `'%riLOF'` bedeutet, dass wir Strings suchen wollen, die mit `'riLOF'` enden. Das `'%'` bedeutet, dass wir alle inneren Knoten des B-Baumes finden wollen (in B-Baum Schreibweise). Wir müssen also praktisch den ganzen B-Baum traversieren. Ein Range Scan ist nicht möglich. Der Range Scan kann nur auf Blattknoten angewendet werden, da diese miteinander verlinkt sind.

Fazit: Prefix-Suche ist schnell, Infix- und Postfix-Suche sind Langsam. Auch bei diesem Query (Infix-Suche):

```
SELECT * FROM MYTEST WHERE TXT LIKE '%aDZfX%';
```

muss die Datenbank einen Full Table Scan machen.

Kapitel 5: Indexierung – SELECT und JOIN / Query Hints

5.1 Übersicht: Worum geht es?

Im letzen Kapitel haben Sie die Ausführungszeiten von einfachen `SELECT`-Abfragen gemessen und sich mit den Query-Plänen vertraut gemacht. Dabei haben Sie gelernt, dass die Verwendung eines Indexes die Suche erheblich beschleunigen kann. In diesem Kapitel werden Sie sehen, dass die Auswirkungen eines Indexes noch viel drastischer sind, wenn in einer `SELECT`-Abfrage zwei Tabellen mittels einer **`JOIN`**-Verknüpfung miteinander verbunden werden. Im zweiten Teil des Kapitels werden Sie lernen, wie man Oracle einen Hinweis (**Hint**) geben kann, ob bei einem Query ein Index (falls vorhanden) verwendet werden soll – oder doch lieber nicht.

5.2 Lernziele

In diesem Kapitel lernen Sie folgendes:

- Sie werden sich bewusst, dass die Verwendung von Indexen beim Verknüpfen von Tabellen (mit `JOIN`) besonders wichtig sind
- Sie wissen auch, weshalb das so ist
- Sie wissen, was der Optimizer ist, und welche Aufgaben er wahr nimmt
- Sie lernen, wie man Oracle einen Hint (Hinweis) für die Verwendung eines Indexes geben kann

5.3 Query mit SELECT und JOIN

Im letzten Kapitel haben Sie gesehen, dass die Antwortzeiten von einem Query stark variieren können, selbst wenn das Resultat letzten Endes gleich lautet. Abhängig davon, ob für ein Query ein Index verwendet werden kann, wird ein Query schneller oder langsamer ausgeführt.

Noch extremer werden die Laufzeitunterschiede, wenn zwei grosse Tabellen mittels einem **`JOIN`** miteinander verbunden werden. Wir machen dies, indem wir die Tabelle `MYTEST` mit sich selbst `JOIN`en. Dieses Vorgehen nennt man **Autojoin**.

Achtung: Die Tabelle `MYTEST` enthält 5'000'000 Rows! `JOIN` bedeutet, dass die Datenbank das Kreuzprodukt **`MYTEST x MYTEST`** berechnen muss, d.h. $5 * 10^6 * 5 * 10^6 = 25 * 10^{12}$. Das entspricht **25 Billionen Zeilen!** D.h. seien Sie etwas vorsichtig, und lassen Sie sich nicht alle Zeilen von so einem `JOIN` anzeigen...

Query 1: Zugriff via Index

Führen Sie nun das folgende Statement aus:

Enter statements:

```
SELECT *
  FROM MYTEST t1,
       MYTEST t2,
 WHERE t1.ID=2500000
   AND t1.ID=t2.ID;
```

Als Resultat erwarten wir genau einen (oder gar keinen) Treffer. Die Ausgabe sollte wie folgt aussehen (zu Gunsten einer übersichtlicheren Darstellung werden die Kolonnen `TXT` im unten stehenden Output nicht dargestellt):

Output:

```
ID       TXT  F  FK       ID       TXT  F  FK
==       ===  =  ==       ==       ===  =  ==
2500000  ...  G  2500000  2500000  ...  G  2500000

Elapsed: 00:00:00.15

Execution Plan
----------------------------------------------------------
  0   SELECT STATEMENT Optimizer=CHOOSE
  1 0   NESTED LOOPS
  2 1     TABLE ACCESS (BY INDEX ROWID) OF 'MYTEST'
  3 2       INDEX (UNIQUE SCAN) OF 'SYS_C00118633' (UNIQUE)
  4 1     TABLE ACCESS (BY INDEX ROWID) OF 'MYTEST'
  5 4       INDEX (UNIQUE SCAN) OF 'SYS_C00118633' (UNIQUE)
```

Erstaunlicherweise dauerte die Abfrage **nur 150 Millisekunden** – obwohl die Datenbank so ein grosses Kreuzprodukt bilden musste! Dies ist nur möglich, weil der **Optimizer** erkannt hat, dass für die Abfrage der Index `SYS_C00118633` (der Primary Key) verwendet werden kann. Dies können Sie dem Execution Plan (oben) entnehmen. Das Suchen in einem Index erfolgt in logarithmisch amortisierter Zeit, wie wir bereits bei den B-Bäumen gesehen hatten. Deshalb ist das Query trotz der grossen Anzahl Rows sehr schnell.

Der **Optimizer** ist der Teil der Datenbank, welcher ein Query vor dessen Ausführung analysiert und optimiert, und dann entweder einen neuen Query Plan erstellt, oder einen bereits bestehenden Query Plan wieder verwendet. Erst danach führt die Datenbank das Query tatsächlich aus.

Der Optimizer hat erkannt, dass `t1.ID=2500000` und `t2.ID=2500000` gelten muss. Deshalb kann die Datenbank sowohl in `t1` als auch in `t2` nach der `ID=2500000` suchen, und dazu den Index `SYS_C00118633` verwenden. Eine Berechnung des vollständigen Kreuzproduktes ist nicht nötig, weshalb Oracle die Antwort zu Ihrer Anfrage sehr schnell findet.

Etwas anders sieht der Query Plan aus, wenn wir die gleiche Suche über die Kolonnen `FK` ausführen (siehe Query 2).

Query 2: Zugriff via Full Table Scan

Probieren Sie nun das folgende (an sich gleiche) Statement aus:

Enter statements:

```
SELECT *
  FROM MYTEST t1,
       MYTEST t2,
 WHERE t1.FK=2500000
   AND t1.FK=t2.FK;
```

Da die Kolonne `FK` die gleichen Werte enthält wie die Kolonne `ID`, erwarten wir das gleiche Ergebnis wie im ersten, schnellen Query. Der Unterschied ist, dass auf der Kolonne `FK` kein Index existiert.

Als Ausgabe sollten Sie folgendes erhalten:

Output:

```
ID       TXT  F  FK       ID       TXT  F  FK
==       ===  =  ==       ==       ===  =  ==
2500000  ...  G  2500000  2500000  ...  G  2500000

Elapsed: 00:00:32.48

Execution Plan
----------------------------------------------------------
  0   SELECT STATEMENT Optimizer=CHOOSE
  1 0   MERGE JOIN
  2 1     SORT (JOIN)
  3 2       TABLE ACCESS (FULL) OF 'MYTEST'
  4 1     SORT (JOIN)
  5 4       TABLE ACCESS (FULL) OF 'MYTEST'
```

Nun dauert die Abfrage **mehr als 30 Sekunden**, d.h. wesentlich länger. In einem produktiven System wäre eine solche Abfrage nicht tragbar, da der Benutzer in der Regel nicht gewillt ist, derart lange auf sein Resultat warten zu müssen!

Aus dem Query Plan sehen wir, dass die Datenbank 2 Mal einen **Full Table Scan** auf der Tabelle `MYTEST` macht: einen für `t1` und einen für `t2`. Der Grund dafür ist, weil auf der Kolonne kein Index erstellt worden ist. Somit bleibt der Datenbank nichts anderes übrig, als die Tabellen komplett zu durchsuchen. Die Datenbank kann ja nicht wissen, dass wir in der Kolonne `FK` die gleichen Werte gespeichert haben wie in der Kolonne `ID` – das könnte gerade so gut auch anders sein! Die Datenbank muss hier vom allgemeinen Fall ausgehen.

Eine Mischung zwischen Zugriff via Index und einem Full Table Scan ist das nachfolgende Query.

Query 3: Zugriff via Index und Full Table Scan

Führen Sie nun das folgende Statement aus:

Enter statements:

```
SELECT *
  FROM MYTEST t1,
       MYTEST t2,
 WHERE t1.ID=2500000
   AND t1.FK=t2.FK;
```

Als Ausgabe sollten Sie folgendes erhalten:

Output:

```
ID       TXT  F  FK       ID       TXT  F  FK
==       ===  =  ==       ==       ===  =  ==
2500000  ...  G  2500000  2500000  ...  G  2500000

Elapsed: 00:00:06.73

Execution Plan
----------------------------------------------------------
  0   SELECT STATEMENT Optimizer=CHOOSE
  1 0   MERGE JOIN
  2 1     TABLE ACCESS (BY INDEX ROWID) OF 'MYTEST'
  3 2       INDEX (UNIQUE SCAN) OF 'SYS_C00118633' (UNIQUE)
  4 1     FILTER
  5 4       TABLE ACCESS (FULL) OF 'MYTEST'
```

Aus dem Query-Plan sehen Sie, dass der erste Zugriff auf die Tabelle `MYTEST` (`t1`) via den Index `SYS_C00118633` (den Primary Key) geschieht. Dieser Zugriff entspricht der ersten Bedingung in der `WHERE`-Clause. Der Zugriff auf `MYTEST` (`t2`) kann jedoch nicht via Index geschehen, da wir in der zweiten Bedingung in der `WHERE`-Clause die Kolonne `FK` verwenden.

Daraus ergibt sich, dass der erste Zugriff auf `t1` schnell ist (wie beim ersten Statement), der Zugriff auf `t2` hingegen langsam (wie beim zweiten Statement). Dies wiederspiegelt sich auch in der Laufzeit des Querys. Mit fast **7 Sekunden** ist das Query deutlich langsamer als das erste Query, jedoch ebenso deutlich schneller als das zweite Query.

Wissenssicherung: Beantworten Sie die folgenden Fragen

a) Wieso muss man bei einem `JOIN` besonders darauf achten, dass man auf den richtigen Kolonnen einen Index angelegt hat?

b) Gegeben seien die folgenden Tabellen `T1` und `T2`:

```
CREATE TABLE T1 (
    A   VARCHAR2(10);
    B   VARCHAR2(10);
    C   VARCHAR2(10);
    D   VARCHAR2(10);
);
```

```
CREATE TABLE T2 (
    W   VARCHAR2(10);
    X   VARCHAR2(10);
    Y   VARCHAR2(10);
    Z   VARCHAR2(10);
);
```

Nun möchten Sie das folgende Query ausführen:

```
SELECT D,
       W,
       X
  FROM T1,
       T2
 WHERE A='HALLO'
   AND B=Y
   AND C=Z
```

Frage: auf welchen Kolonnen sollten Sie einen Index erstellen, und weshalb?

c) Stellen Sie eine allgemein gültige Regel auf, auf welchen Kolonnen man immer einen Index erstellen sollte.

5.4 Tipps für die Datenbank: Query Hints

Ein Query Hint (kurz: **Hint**) ist ein Tipp resp. Hinweis an den Optimizer, wie dieser das Query resp. den Query Plan optimieren soll. Wichtig: Ein Hint hat keinen Einfluss auf das Resultat, lediglich auf den Query Plan – und somit auf die Ausführungsgeschwindigkeit!

Ein Query Hint wird direkt nach dem einleitenden Schlüsselwort für das Statement (`SELECT`, `INSERT`, `UPDATE` oder `DELETE`) angegeben. Die Schreibweise für ein `SELECT` ist z.B. wie folgt:

```
SELECT /*+ ... */ c1, c2 FROM table WHERE c1='...' and c2='...';
```

Die Zeichen **`/* ... */`** kennzeichnen einen Kommentar. Kommentare sind nicht Bestandteil eines Queries, sie werden ignoriert! Das Spezielle an diesem Kommentar ist das Plus-Zeichen (**`+`**), welches direkt und ohne Abstand nach dem Einleiten des Kommentares (**`/*`**) folgt. Dieses Plus-Zeichen weist den Optimizer an, dass dieser Kommentar für ihn bestimmt ist. Aus Sicht des Optimizers handelt es sich demzufolge nicht um einen Kommentar, sondern um einen Hint! Für das Statement an und für sich ist diese Angabe jedoch "nur" ein Kommentar, welcher ignoriert wird.

Es gibt viele verschiedene Arten von Query-Hints. Die wichtigsten sind nachfolgend beschrieben:

Query Hints, welche indirekt die Verwendung von Indexen beeinflussen

`/*+ ALL_ROWS */`

Gibt dem Optimizer den Tipp, dass wir an allen Zeilen des Resultats interessiert sind. Oracle entscheidet dann selbst, ob ein Full Table Scan oder die Verwendung von Indexen zielführender ist.

`/*+ FIRST_ROWS(<integer>) */`

Gibt dem Optimizer den Tipp, dass wir in erster Linie nur an den ersten N Zeilen des Resultates interessiert sind, und dass wir möglichst schnell auf diese N Zeilen zugreifen können wollen. Oracle entscheidet dann selbst, ob ein Full Table Scan oder die Verwendung von Indexen zielführender ist. Wenn wir trotzdem mehr als N Zeilen des Resultates verwerten, dann wäre evtl. eine andere Suchstrategie (ein anderer Query Plan) zielführender gewesen (z.B. `ALL_ROWS`).

Query Hints, welche direkt die Verwendung von Indexen beeinflussen

`/*+ FULL([@queryblock] <tablespec>) */`

Gibt dem Optimizer den Tipp, dass wir auf der Tabelle `<tablespec>` einen Full Table Scan erzwingen wollen. Je nach Suchstrategie kann das sinnvoll sein. Die Angabe von `@queryblock` ist freiwillig. Man kann so einem Query Block einen Namen geben, z.B. `@MYBLOCK` (siehe `QB_NAME` weiter unten). Ein Query Block ist das aktuelle Statement, auf das sich der Hint bezieht, z.B. ein `SELECT`-, `INSERT`-, `UPDATE`- oder `DELETE`-Statement.

`/*+ NO_INDEX([@queryblock] <tablespec> <indexspec>) */`

Gibt dem Optimizer den Tipp, dass der Index `<indexspec>` auf der Tabelle `<tablespec>` auf keinen Fall werden soll. Optional ist die Angabe eines zuvor definierten Namens (`@queryblock`), welcher einen Query Block bezeichnet.

`/*+ INDEX([@queryblock] <tablespec> <indexspec>) */`

Gibt dem Optimizer den Tipp, dass der Index `<indexspec>` auf der Tabelle `<tablespec>` unbedingt verwendet werden soll.

/*+ INDEX_ASC([@queryblock] <tablespec> <indexspec>) */

Gibt dem Optimizer den Tipp, dass der Index `<indexspec>` auf der Tabelle `<tablespec>` unbedingt verwendet werden soll. Falls ein Range Scan involviert ist, dann wird der Index in aufsteigender Reihenfolge traversiert.

/*+ INDEX_DESC([@queryblock] <tablespec> <indexspec>) */

Gibt dem Optimizer den Tipp, dass der Index `<indexspec>` auf der Tabelle `<tablespec>` unbedingt verwendet werden soll. Falls ein Range Scan involviert ist, dann wird der Index in absteigender Reihenfolge traversiert.

/*+ QB_NAME(<query_block_name>) */

Gibt einem Query Block einen eindeutigen Namen, auf den man später verweisen kann. Ein Query Block ist das aktuelle Statement, auf das sich der Hint bezieht, z.B. ein `SELECT`-, `INSERT`-, `UPDATE`- oder `DELETE`-Statement. Query Block Namen werden vor allem bei komplizierten Queries eingesetzt.

Weitere Query Hint Typen finden Sie hier: http://psoug.org/reference/hints.html

In den folgenden 4 Beispielen lernen Sie die Query-Hints `FIRST_ROW`, `FULL`, `INDEX` und `NO_INDEX` kennen.

Beispiel 1: FIRST_ROWS

Wir nehmen das Query 2 aus dem ersten Teil dieses Kapitels, und versehen es mit dem Hint **`FIRST_ROW(1)`**. Sie erinnern sich: das Query benötigte ursprünglich mehr als 30 Sekunden!

Führen Sie nun das folgende Statement aus:

Enter statements:

```
SELECT /*+ FIRST_ROWS(1) */ *
  FROM MYTEST t1,
       MYTEST t2,
 WHERE t1.FK=2500000
   AND t1.FK=t2.FK;
```

Als Ausgabe sollten Sie ungefähr folgendes erhalten:

Output:

```
ID       TXT  F  FK       ID       TXT  F  FK
==       ===  =  ==       ==       ===  =  ==
2500000  ...  G  2500000  2500000  ...  G  2500000

Elapsed: 00:00:09.96

Execution Plan
----------------------------------------------------------
  0   SELECT STATEMENT Optimizer=HINT: FIRST_ROWS (Cost=4 Card=1 Bytes=112)
  1 0   MERGE JOIN (CARTESIAN) (Cost=4 Card=1 Bytes=112)
  2 1     TABLE ACCESS (FULL) OF 'MYTEST' (Cost=2 Card=1 Bytes=56)
  3 1     BUFFER (SORT) (Cost=2)
  4 3       TABLE ACCESS (FULL) OF 'MYTEST' (Cost=2)
```

Wenn Sie den obigen Query Plan mit dem Query Plan aus dem ersten Teil des Kapitels (Query 2) vergleichen, dann werden Sie feststellen, dass es Unterschiede gibt. Dank unserem Query Hint benötigte das Statement **nur noch ca. 10 Sekunden**, anstatt der ursprünglichen 30 Sekunden. D.h. mit dem Query Hint ist das Statement **66% schneller**!

Aus der Zeile 0 erkennen wir, dass der Optimizer unseren Hint berücksichtigt hat. Es steht nun **`Optimizer=HINT: FIRST_ROWS`** anstatt **`Optimizer=CHOOSE`**. `CHOOSE` bedeutet, dass der Optimizer die Optimierungsstrategie selbst wählt – das ist der Normalfall. `Optimizer=XXX` bedeutet, dass wir dem Optimizer explizit die Optimierungsstrategie `XXX` vorgegeben haben.

Die Datenbank macht zwar immer noch auf beiden Tabellen einen Full Table Scan, wählt dafür jedoch eine etwas andere Vorgehensweise: `BUFFER (SORT)` anstatt `SORT (JOIN)`. Den genauen Unterschied betrachten wir an dieser Stelle nicht weiter, dies ist nicht Ziel des vorliegenden Leitprogrammes. Sie sollen lediglich lernen, dass man mit Query Hints Einfluss auf die Optimierungsstrategie des Optimizers nehmen kann.

Beispiel 2: `FULL`

Wir nehmen das Query 1 aus dem ersten Teil dieses Kapitels, und versehen es mit dem Hint **`FULL(<table_name>)`**. Sie erinnern sich: das Query benötigte ursprünglich nur gerade 150 Millisekunden.

Führen Sie nun das folgende Statement aus:

Enter statements:

```
SELECT /*+ FULL(t1) FULL(t2) */ *
  FROM MYTEST t1,
       MYTEST t2,
 WHERE t1.ID=2500000
   AND t1.ID=t2.ID;
```

Die Ausgabe sieht nun etwas anders aus:

Output:

```
ID       TXT  F  FK       ID       TXT  F  FK
==       ===  =  ==       ==       ===  =  ==
2500000  ...  G  2500000  2500000  ...  G  2500000

Elapsed: 00:00:09.82

Execution Plan
----------------------------------------------------------
  0   SELECT STATEMENT Optimizer=CHOOSE (Cost=108008760 Card=1 Bytes=112)
  1 0   NESTED LOOPS (Cost=108008760 Card=1 Bytes=112)
  2 1     TABLE ACCESS (FULL) OF 'MYTEST' (Cost=3567 Card=30279 Bytes=1695624)
  3 1     TABLE ACCESS (FULL) OF 'MYTEST' (Cost=3567 Card=30279 Bytes=1695624)
```

Die Ausgabe sieht immer noch gleich aus wie die Ausgabe vom Query 1. Allerdings sieht der Execution Plan anders aus, und das Statement benötigt wesentlich mehr als 150 Millisekunden – nämlich beinahe **10 Sekunden!** Mit dem Query Hint weisen wir den Optimizer an, auf den Tabellen `t1` und `t2` in jedem Fall einen Full Table Scan durchzuführen. Dies können Sie aus dem obigen Query Plan heraus lesen. Allfällig vorhandene Indexes werden ignoriert. Das bedeutet, dass der Index `SYS_C00118633` (Primary Key) für die abfrage nicht verwendet wird! Deshalb dauert das ursprünglich schnelle Query nun wesentlich länger.

Dieser Query Hint ist im vorliegenden Fall nicht sinnvoll, da er das Query langsamer macht. Es kann jedoch Situationen geben, in denen man einen Full Table Scan erzwingen möchte. Wir gehen an dieser Stelle nicht weiter darauf ein. Sie sollen lediglich lernen, dass es diese Möglichkeit gibt.

Die folgenden Beispiele 3 und 4 können Sie nicht selbst ausprobieren, da Sie keine Berechtigung haben, den Index `MYTEST_TXT` zu erstellen.

Beispiel 3: `INDEX`

Der Query Hint **`/*+ INDEX([@queryblock] <tablespec> <indexspec>) */`** ermöglicht es Ihnen, dem Optimizer die Verwendung eines Indexes explizit vorzuschreiben. Als Beispiel erstellen wir auf der Kolonne TXT den Index `MYTEST_TXT`:

```
CREATE INDEX MYTEST_TXT ON MYTEST(TXT);
```

Die Erstellung dieses Indexes dauert ca. 2.5 Minuten. Kein Wunder, denn die Tabelle enthält ja 5'000'000 Rows!

Wenn wir nun das folgende Statement ausführen ...

Enter statements:

```
SELECT /*+ INDEX(MYTEST MYTEST_TXT) */ *
  FROM MYTEST
 WHERE TXT LIKE 'TEST #2500000%';
```

... erhalten wir die folgende Ausgabe:

Output:

```
ID       TXT  F  FK       ID       TXT  F  FK
==       ===  =  ==       ==       ===  =  ==
2500000  ...  G  2500000  2500000  ...  G  2500000

Elapsed: 00:00:00.09

Execution Plan
----------------------------------------------------------
  0   SELECT STATEMENT Optimizer=CHOOSE (Cost=10 Card=151394 Bytes=8478064)
  1 0   TABLE ACCESS (BY INDEX ROWID) OF 'MYTEST' (Cost=10 Card=151394 Bytes=8478064)
  2 1     INDEX (RANGE SCAN) OF 'MYTEST_TXT' (NON-UNIQUE) (Cost=2 Card=27251)
```

Mit dem Query Hint teilen wir dem Optimizer mit, dass er auf der Tabelle `MYTEST` unbedingt den von uns zuvor erstellten Index `MYTEST_TXT` verwenden soll. Sie erkennen dies an der zweiten Zeile im Query Plan. Wenn wir keinen Query Hint angeben, entscheidet der Optimizer selbst, ob er für die Abfrage den Index verwenden soll oder nicht. Oftmals benutzt Oracle den Index nur für grosse Tabellen, und entscheidet sich für kleine Tabellen (z.B. bis 30'000 Rows) für einen Full Table Scan.

Da wir im obigen Statement den Optimizer anweisen, den Index `MYTEST_TXT` zwingend zu verwenden, ist das Statement auch entsprechend schnell: es benötigt nur ca. **10 Millisekunden!**

Beispiel 4: NO_INDEX

Sozusagen das Gegenteil des Query Hints INDEX ist der Query Hint NO_INDEX. Damit weisen wir den Optimizer an, einen bestimmten Index explizit **nicht** zu verwenden. Der Optimizer ist nun immer noch frei, einen anderen Index zu verwenden oder einen Full Table Scan zu machen. Im Unterschied zum Beispiel 3 schreiben wir hier die Verwendung eines Indexes nicht vor, sondern überlassen die Wahl dem Optimizer. Wir teilen dem Optimizer lediglich mit, dass er diesen Index nicht verwenden soll.

Wenn wir nun das folgende Statement ausführen ...

Enter statements:

```
SELECT /*+ NO_INDEX(MYTEST MYTEST_TXT) */ *
  FROM MYTEST,
 WHERE TXT LIKE 'TEST #2500000%';
```

... erhalten wir die folgende Ausgabe:

Output:

```
ID       TXT  F  FK       ID       TXT  F  FK
==       ===  =  ==       ==       ===  =  ==
2500000  ...  G  2500000  2500000  ...  G  2500000

Elapsed: 00:00:05.78

Execution Plan
----------------------------------------------------------
  0   SELECT STATEMENT Optimizer=CHOOSE (Cost=3567 Card=151394 Bytes=8478064)
  1 0   TABLE ACCESS (FULL) OF 'MYTEST' (Cost=3567 Card=151394 Bytes=8478064)
```

Aus dem obigen Query Plan sehen wir, dass die Datenbank nun einen Full Table Scan auf der Tabelle MYTEST durchführt – obwohl das Statement an und für sich genau das gleiche ist wie das Statement im Beispiel 3. Den Unterschied macht der Query Hint aus! Das Statement braucht nun mit fast **6 Sekunden** wesentlich länger als das gleiche Statement im Beispiel 3.

Wie beim Beispiel 2 ist auch dieser Query Hint für sich alleine nicht sinnvoll, da er das Statement langsamer macht. Trotzdem kann es hin und wieder sinnvoll sein, dem Optimizer die Verwendung eines Indexes zu verbieten.

Betrachten Sie z.B. die folgenden Indexes:

```
CREATE INDEX MYTEST_TXT_1 ON MYTEST(TXT);
CREATE INDEX MYTEST_TXT_2 ON MYTEST(TXT, FLAG);
```

Den ersten Index kennen Sie bereits, er wurde im Beispiel 3 angelegt. Der zweite Index wird über die Kolonnen TXT und FLAG erstellt.

Wenn Sie nun die folgende Abfrage machen ...

```
SELECT *
  FROM MYTEST
 WHERE TXT LIKE 'TEST #2500000%'
   AND FLAG='G';
```

... dann muss der Optimizer entscheiden, welchen der beiden Indexes er verwenden soll. Sie können dem Optimizer jedoch explizit verbieten, den Index MYTEST_TXT_1 zu verwenden,

und dem Optimizer trotzdem die Wahl überlassen, ob er den Index MYTEST_TXT_2 verwenden oder einen Full Table Scan durchführen soll:

```
SELECT /*+ NO_INDEX(MYTEST MYTEST_TXT_1) */ *
  FROM MYTEST
 WHERE TXT LIKE 'TEST #2500000%'
   AND FLAG='G';
```

Natürlich können Sie dem Optimizer auch direkt vorschreiben, den Index MYTEST_TXT_2 zu verwenden:

```
SELECT /*+ INDEX(MYTEST MYTEST_TXT_2) */ *
  FROM MYTEST
 WHERE TXT LIKE 'TEST #2500000%'
   AND FLAG='G';
```

Welche Variante Sie verwenden, hängt oftmals noch von weiteren Faktoren ab, auf die wir an dieser Stelle jedoch nicht mehr weiter eingehen.

Wissenssicherung: Beantworten Sie die folgenden Fragen

d) Wenn Sie den Query Hint FIRST_ROWS(N) oder ALL_ROWS verwenden, dann sehen Sie im Query Plan die Angabe Optimizer=FIRST_ROWS resp. Optimizer=ALL_ROWS. Beim Query Hint INDEX(...) ist das nicht der Fall, es steht immer Optimizer=CHOOSE. Weshalb ist das so?

e) Schreiben Sie ein Query, welches die Kolonne TXT für die IDs 1'000'001...1'000'010 ausgibt. Verwenden Sie einen Query Hint, so dass der Index in aufsteigender Reihenfolge traversiert wird. Schreiben Sie anschliessend das Query um, so dass der Index in absteigender Richtung traversiert wird. Was fällt Ihnen an der Ausgabe auf?

f) Was ist an folgendem Query falsch?

```
SELECT /*+ NO_INDEX(MYTEST SYS_C00118633) */ *
  FROM MYTEST t1,
       MYTEST t2
 WHERE t1.ID='5000000'
   AND t1.FK=t2.ID;
```

5.5 Zusammenfassung

In diesem Kapitel haben Sie gelernt, auf was es ankommt, wenn Sie zwei Tabellen mit einem **JOIN** verbinden wollen. Sie haben gesehen, dass es für die Performance wesentlich ist, auf den richtigen Kolonnen einen Index anzulegen. Deshalb legt man auf **Foreign Key** Kolonnen oftmals aus Prinzip einen Index an.

Sie haben nun auch den **Optimizer** kennen gelernt – ein wichtiger Teil von Oracle, wenn es um Performance geht. Sie haben gesehen, dass man mit **Query Hints** direkt Einfluss auf den Optimizer nehmen kann. Sie können dem Optimizer eine Optimierungsstrategie vorgeben, und sogar definieren, welche Indexes für das Query verwendet werden sollen.

Sie haben nun die wesentlichen Werkzeuge kennen gelernt, welche für das **Tuning** eines Querys nötig sind.

5.6 Lösungen zu den Wissenssicherungsfragen

a) *Wieso muss man bei einem `JOIN` besonders darauf achten, dass man auf den richtigen Kolonnen einen Index angelegt hat?*

Bei einem `JOIN` (z.B. `A JOIN B`) muss die Datenbank das **Kreuzprodukt `A x B`** bilden, d.h. im Extremfall (ohne `JOIN`-Bedingung) muss sie alle Zeilen der Tabelle `A` mit jeder Zeile in der Tabelle `B` verknüpfen:

```
TABLE A
a1
a2
a3

TABLE B
b1
b2
b3
```

`A x B` →

```
A JOIN B
a1   b1
a1   b2
a1   b3
a2   b1
a2   b2
a2   b3
a3   b1
a3   b2
a3   b3
```

Mit der `JOIN`-Bedingung **`ON (...)`** kann man die Menge der Rows einschränken, auf denen das Kreuzprodukt gebildet werden soll. Dies bedeutet jedoch, dass die Datenbank sowohl in der Tabelle `A` als auch in der Tabellle `B` die richtigen Zeilen finden muss, bevor sie das Kreuzprodukt bilden kann.

Es ist also eine **Suche in `A` und in `B`** nötig. Damit diese Suche schnell durchgeführt werden kann, ist es nötig, auf jenen Kolonnen einen Index anzulegen, welche in der `JOIN`-Bedingung verwendet werden. Normalerweise handelt es sich dabei um **Foreign Keys**, da die Tabellen meistens via Foreign Keys verknüpft (ge`JOIN`ed) werden.

b) *Gegeben seien die folgenden Tabellen `T1` und `T2`:*

```
CREATE TABLE T1 (
    A   VARCHAR2(10);
    B   VARCHAR2(10);
    C   VARCHAR2(10);
    D   VARCHAR2(10);
);
```

```
CREATE TABLE T2 (
    W   VARCHAR2(10);
    X   VARCHAR2(10);
    Y   VARCHAR2(10);
    Z   VARCHAR2(10);
);
```

Nun möchten Sie das folgende Query ausführen:

```
SELECT D,
       W,
       X
  FROM T1,
       T2
 WHERE A='HALLO'
   AND B=Y
   AND C=Z
```

Frage: auf welchen Kolonnen sollten Sie einen Index erstellen, und weshalb?

Wie Sie bereits in Aufgabe a) gesehen haben, sollten Sie einen Index auf den Kolonnen anlegen, welche in der JOIN-Bedingung vorkommen. Es sind dies:

`T1.A`, **`T1.B`**, **`T1.C`** sowie **`T2.Y`**, **`T2.Z`**

Der Grund dafür ist, weil die Datenbank die Anzahl Rows einschränken muss, auf welchen das Kreuzprodukt gebildet werden soll (Kolonnen in der `JOIN`-Bedingung).

c) Stellen Sie eine allgemein gültige Regel auf, auf welchen Kolonnen man immer einen Index erstellen sollte.

1. Auf den **Foreign Keys** sollten Sie prinzipiell einen Index erstellen, da diese Kolonnen sehr of in der `JOIN`-Bedingung verwendet werden.
2. Auf allen zusätzlichen Kolonnen, welche ebenfalls in der `JOIN`-Bedingung verwendet werden, sollten Sie ebenfalls einen Index erstellen.

d) Wenn Sie den Query Hint `FIRST_ROWS(N)` oder `ALL_ROWS` verwenden, dann sehen Sie im Query Plan die Angabe `Optimizer=FIRST_ROWS` resp. `Optimizer=ALL_ROWS`. Beim Query Hint `INDEX(...)` ist das nicht der Fall, es steht immer `Optimizer=CHOOSE`. Weshalb ist das so?

`FIRST_ROWS(N)` und `ALL_ROWS` sind Query Hints, welche dem Optimizer direkt die **Strategie** vorgeben, wie er das Query optimieren soll. `Optimizer=XXX` wiederspiegelt die Strategie `XXX`, welche der Optimizer für die Optimierung des Querys verwendet. Der Optimizer ist dabei frei in der Wahl der Indexes, welche er für das Query verwenden soll, um der Strategie so gut wie möglich gerecht zu werden.

Im Gegensatz dazu ist `INDEX(...)` ein Query Hint, welcher dem Optimizer direkt den zu verwendenden Index vorgibt – und **nicht die Strategie!** Deshalb hat dieser Query Hint keinen Einfluss auf die Angabe `Optimizer=XXX`. Die Auswirkungen des Query Hints sehen Sie weiter unten im Query Plan: bei der betreffenden Tabelle sehen Sie, ob für die Suche ein Index verwendet wird oder nicht.

`FIRST_ROWS(N)` / `ALL_ROWS` und `INDEX(...)` nehmen demnach auf unterschiedliche Art und Weise Einfluss auf den Optimizer, was sich im Query Plan an unterschiedlichen Orten wiederspiegelt.

e) Schreiben Sie ein Query, welches die Kolonne `TXT` für die `ID`s 1'000'001...1'000'010 ausgibt. Verwenden Sie einen Query Hint, so dass der Index in aufsteigender Reihenfolge traversiert wird. Schreiben Sie anschliessend das Query um, so dass der Index in absteigender Richtung traversiert wird. Was fällt Ihnen an Ausgabe auf?

Query ohne Hints

```
SELECT TXT                          SELECT TXT
  FROM MYTEST                         FROM MYTEST
 WHERE ID >= 1000001                 WHERE ID BETWEEN 1000001
   AND ID <= 1000010;                      AND 1000010;
```

Beide Queries sind gleichwertig. Der Optimizer wird das Query auf der rechten Seite in das Query auf der linken Seite umformen. Deshalb verwenden wir im folgenden das Query auf der linken Seite.

Query 1: mit Hint `INDEX_ASC(...)`

```
SELECT /*+ INDEX_ASC(MYTEST SYS_C00118633) */
       TXT
  FROM MYTEST
 WHERE ID >= 1000001
   AND ID <= 1000010;
```

Query 2: mit Hint `INDEX_DESC(...)`

```
SELECT /*+ INDEX_DESC(MYTEST SYS_C00118633) */
       TXT
  FROM MYTEST
 WHERE ID >= 1000001
   AND ID <= 1000010;
```

Auffallend ist, dass die Ausgabe beim Query 1 aufsteigend sortiert ist, beim Query 2 jedoch absteigend – ohne dass wir ein `ORDER BY` angegeben haben! Wenn wir kein `ORDER BY` angeben, dann ist Oracle prinzipiell frei, in welcher Reihenfolge die Ausgabe erfolgen soll. Da wir mit dem Query Hint dem Optimizer die Richtung vorgeben (`ASC` / `DESC`), in welcher der Index traversiert werden soll, entspricht dies gleich der Ausgabe von Oracle, da dies am effizientesten ist. Man sollte sich jedoch nicht darauf verlassen: wenn man die Ausgabe sortiert haben möchte, sollte man ein `ORDER BY` angeben. Oracle wird das dann selbst optimieren – im vorliegenden Fall müsste Oracle nichts mehr sortieren, da dies bereits geschehen ist.

f) Was ist an folgendem Query falsch?

```
SELECT /*+ NO_INDEX(MYTEST SYS_C00118633) */ *
  FROM MYTEST t1,
       MYTEST t2
 WHERE t1.ID='5000000'
   AND t1.FK=t2.ID;
```

Die Tabelle `MYTEST` kommt im Query 2 Mal vor. Da wir im Query Hint die Tabelle `MYTEST` angegeben haben, weiss der Optimizer nicht, welche Tabelle (`t1` oder `t2`) wir meinen. Deshalb ignoriert der Optimizer unseren Hint, die Ausgabe sieht wie folgt aus:

Output:

```
ID       TXT  F  FK       ID       TXT  F  FK
==       ===  =  ==       ==       ===  =  ==
5000000  ...  G  5000000  5000000  ...  G  5000000

Elapsed: 00:00:00.15

Execution Plan
----------------------------------------------------------
  0   SELECT STATEMENT Optimizer=CHOOSE (Cost=3 Card=1 Bytes=112)
  1 0   NESTED LOOPS (Cost=3 Card=1 Bytes=112)
  2 1     TABLE ACCESS (BY INDEX ROWID) OF 'MYTEST' (Cost=2 Card=1 Bytes=56)
  3 2       INDEX (UNIQUE SCAN) OF 'SYS_C00118633' (UNIQUE) (Cost=1 Card=100)
  4 1     TABLE ACCESS (BY INDEX ROWID) OF 'MYTEST' (Cost=1 Card=1 Bytes=56)
  5 4       INDEX (UNIQUE SCAN) OF 'SYS_C00118633' (UNIQUE)
```

Beachten Sie, dass Oracle **keine Fehlermeldung** ausgibt! Der Query Hint wird stillschweigend ignoriert. Deshalb: wenn Sie Query Hints verwenden, sollten Sie in jedem Fall den Query Plan anschauen, um sicher zu gehen, dass der Optimizer auch das macht was Sie erwarten.

Wenn Sie den Index `SYS_C00118633` auf beiden Tabellen nicht verwenden wollen, müssen Sie im Query Hint beide Tabellen angeben, unter Verwendung der **Table Aliases** `t1` und `t2`:

Enter statements:

```
SELECT /*+ NO_INDEX(t1 SYS_C00118633)
           NO_INDEX(t2 SYS_C00118633) */ *
  FROM MYTEST t1,
       MYTEST t2
 WHERE t1.ID='5000000'
   AND t1.FK=t2.ID;
```

Die Ausgabe sieht nun etwas anders aus:

Output:

```
ID       TXT  F  FK       ID       TXT  F  FK
==       ===  =  ==       ==       ===  =  ==
5000000  ...  G  5000000  5000000  ...  G  5000000

Elapsed: 00:00:17.54

Execution Plan
----------------------------------------------------------
  0   SELECT STATEMENT Optimizer=CHOOSE (Cost=15548 Card=1 Bytes=112)
  1 0   HASH JOIN (Cost=15548 Card=1 Bytes=112)
  2 1     TABLE ACCESS (FULL) OF 'MYTEST' (Cost=3567 Card=30279 Bytes=1695624)
  3 1     TABLE ACCESS (FULL) OF 'MYTEST' (Cost=3567 Card=3027878 Bytes=169561168)
```

Aus dem Query Plan sehen Sie, dass nun auf beiden Tabellen ein Full Table Scan gemacht wird. Die Abfrage dauert jetzt mit **17 Sekunden** wesentlich länger.

5.7 Lernkontrolle: Hier können Sie überprüfen, ob Sie das Kapitel beherrschen

Die folgenden Aufgaben repräsentieren eine zusammenhängende Übung. Beginnen Sie bei Aufgabe 1, und arbeiten Sie sich bis zur Aufgabe 10 durch.

Aufgabe 1

Erstellen Sie in Ihrem Datenbank-Schema die Tabellen `PERSON` und `ADRESSE`, welche die folgenden Kolonnen enthalten:

```
PERSON                                  (Primary Key)
P_ID            NUMBER(19)
ANREDE          CHAR(1)
VORNAME         VARCHAR2(30)
NAME            VARCHAR2(30)
GEB_DAT         DATE

ADRESSE
A_ID            NUMBER(19)              (Primary Key)
P_ID            NUMBER(19)              (Foreign Key auf PERSON.P_ID)
STRASSE         VARCHAR2(30)
HAUSNR          NUMBER(19)
PLZ             NUMBER(19)
ORT             VARCHAR2(30)
```

Aufgabe 2

Erstellen Sie auf der Kolonne `P_ID` der Tabelle `ADRESSE` einen Index. Der Index soll den Namen `IX_ADRESSE_P_ID` haben. Wieso brauchen wir diesen Index?

Aufgabe 3

Fügen Sie die folgenden Personen in die Tabelle `PERSON` ein:

PERSON				
P_ID	ANREDE	VORNAME	NAME	GEB_DAT
13	H	Milhouse	van Houten	13.02.1983
45	H	Waylon	Smithers	29.10.1957
82	F	Marge	Simpson	20.04.1959
07	M	Monty	Burns	27.12.1952
93	M	Bart	Simpson	10.05.1983
68	M	Seymour	Skinner	05.07.1960
23	M	Homer Jay	Simpson	12.05.1956
73	M	Moe	Szyslak	14.09.1959
58	F	Lisa	Simpson	23.11.1985
36	M	Chief Clancy	Wiggum	02.03.1958

Aufgabe 4

Fügen Sie die folgenden Adressen in die Tabelle ADRESSE ein:

ADRESSE					
A_ID	P_ID	STRASSE	HAUSNR	PLZ	ORT
1301	13	Main St.	150	45931	Springfield City
4501	45	Atom St.	49	55556	Power Plant
4502	45	Michigan Av.	392	39247	Chicaco IL
8201	82	Main St.	152	45931	Springfield City
0701	07	Atom St.	55	55555	Power Plant
0702	07	Burn's Home	1	89322	New York City
9301	93	Main St.	152	45931	Springfield City
6801	68	Scool St.	22	45937	Springfield Edu
6802	68	Rodeo St.	91	24742	Austin TX
2301	23	Main St.	152	45931	Springfield City
7301	73	Beer St.	88	49287	Springfield Downtown
5801	58	Main St.	152	45931	Springfield City
3601	36	Police Office	911	29883	Springfield County
3602	36	Vacation St.	99	90481	Bozeman MT

Aufgabe 5

Finden Sie die Indexnamen heraus, welche auf den Tabellen PERSON und ADRESSE existieren. Tipp: verwenden Sie die System-View USER_INDEXES.

Aufgabe 6

Schreiben Sie ein Query, welches alle Personen inklusive deren Adresse ausgibt. Die Ausgabe soll die folgenden Kolonnen enthalten:

P_ID A_ID ANREDE NAME VORNAME GEB_DAT STRASSE HAUSNR PLZ ORT

Lassen Sie sich auch den Query Plan und die benötigte Zeit anzeigen. Welche Indexes werden benutzt?

Aufgabe 7

Weisen Sie nun den Optimizer an, dass er das Query so optimieren soll, dass die erste Row so schnell als möglich retourniert wird. Wie sieht das Query nun aus? Hat sich der Query Plan verändert?

Aufgabe 8

Sie wollen alle Rows lesen. Weisen Sie den Optimizer so an, dass er das Query diesbezüglich optimiert. Wie sieht das Query nun aus? Hat sich der Query Plan verändert?

Aufgabe 9

In Aufgabe 6 haben Sie gesehen, dass der Optimizer die Indexes der Tabelle `ADRESSE` nicht verwendet. Weisen Sie nun den Optimizer an, dass er für das Query alle 3 Indexes verwenden soll! Wie sehen das Query und der Query Plan aus, und wie wirkt sich das auf die Performance aus?

Aufgabe 10

Räumen Sie Ihr Datenbank-Schema wieder auf, indem Sie die erstellten Tabellen wieder löschen!

5.8 Lösungen zu den Lernkontrollfragen

Die folgenden Aufgaben repräsentieren eine zusammenhängende Übung. Beginnen Sie bei Aufgabe 1, und arbeiten Sie sich bis zur Aufgabe 10 durch.

Aufgabe 1

Erstellen Sie in Ihrem Datenbank-Schema die Tabellen `PERSON` und `ADRESSE`, welche die folgenden Kolonnen enthalten:

```
PERSON                                  (Primary Key)
P_ID            NUMBER(19)
ANREDE          CHAR(1)
VORNAME         VARCHAR2(30)
NAME            VARCHAR2(30)
GEB_DAT         DATE

ADRESSE
A_ID            NUMBER(19)              (Primary Key)
P_ID            NUMBER(19)              (Foreign Key auf PERSON.P_ID)
STRASSE         VARCHAR2(30)
HAUSNR          NUMBER(19)
PLZ             NUMBER(19)
ORT             VARCHAR2(30)
```

Führen Sie die folgenden Statements aus:

```
CREATE TABLE PERSON (
    P_ID          NUMBER(19) PRIMARY KEY,
    ANREDE        CHAR(1),
    VORNAME       VARCHAR2(30),
    NAME          VARCHAR2(30),
    GEB_DAT       DATE
);

CREATE TABLE ADRESSE (
    A_ID          NUMBER(19) PRIMARY KEY,
    P_ID          NUMBER(19) REFERENCES PERSON(P_ID),
    STRASSE       VARCHAR2(30),
    HAUSNR        NUMBER(19),
    PLZ           NUMBER(19),
    ORT           VARCHAR2(30)
);
```

Aufgabe 2

Erstellen Sie auf der Kolonne `P_ID` der Tabelle `ADRESSE` einen Index. Der Index soll den Namen `IX_ADRESSE_P_ID` haben. Wieso brauchen wir diesen Index?

Erstellen Sie den Index wie folgt:

```
CREATE INDEX IX_ADRESSE_P_ID ON ADRESSE(P_ID);
```

Den Index braucht es, wenn wir die Personen mit den dazu gehörenden Adressen ausgeben wollen. Dann müssen wir die Tabellen `PERSON` und `ADRESSE` miteinander verbinden (mit einem `JOIN`). Dies geschieht über den Foreign Key `P_ID` (`JOIN`-Bedingung), für welchen wir gerade eben den Index erstellt haben.

Aufgabe 3

Fügen Sie die folgenden Personen in die Tabelle `PERSON` ein:

PERSON				
P_ID	*ANREDE*	*VORNAME*	*NAME*	*GEB_DAT*
13	*H*	*Milhouse*	*van Houten*	*13.02.1983*
45	*H*	*Waylon*	*Smithers*	*29.10.1957*
82	*F*	*Marge*	*Simpson*	*20.04.1959*
07	*M*	*Monty*	*Burns*	*27.12.1952*
93	*M*	*Bart*	*Simpson*	*10.05.1983*
68	*M*	*Seymour*	*Skinner*	*05.07.1960*
23	*M*	*Homer Jay*	*Simpson*	*12.05.1956*
73	*M*	*Moe*	*Szyslak*	*14.09.1959*
58	*F*	*Lisa*	*Simpson*	*23.11.1985*
36	*M*	*Chief Clancy*	*Wiggum*	*02.03.1958*

Fügen Sie die Rows wie folgt ein:

```
INSERT INTO PERSON VALUES (13, 'H', 'Milhouse', 'van Houten', '13.02.1983');
INSERT INTO PERSON VALUES (45, 'H', 'Waylon', 'Smithers', '29.10.1957');
INSERT INTO PERSON VALUES (82, 'F', 'Marge', 'Simpson', '20.04.1959');
INSERT INTO PERSON VALUES (07, 'H', 'Monty', 'Burns', '27.12.1952');
INSERT INTO PERSON VALUES (93, 'H', 'Bart', 'Simpson', '10.05.1983');
INSERT INTO PERSON VALUES (68, 'H', 'Seymour', 'Skinner', '05.07.1960');
INSERT INTO PERSON VALUES (23, 'H', 'Homer Jay', 'Simpson', '12.05.1956');
INSERT INTO PERSON VALUES (73, 'H', 'Moe', 'Szyslak', '14.09.1959');
INSERT INTO PERSON VALUES (58, 'F', 'Lisa', 'Simpson', '23.11.1985');
INSERT INTO PERSON VALUES (36, 'H', 'Chief Clancy', 'Wiggum', '02.03.1958');
COMMIT;
```

Aufgabe 4

Fügen Sie die folgenden Adressen in die Tabelle `ADRESSE` ein:

ADRESSE					
A_ID	*P_ID*	*STRASSE*	*HAUSNR*	*PLZ*	*ORT*
1301	*13*	*Main St.*	*150*	*45931*	*Springfield City*
4501	*45*	*Atom St.*	*49*	*55556*	*Power Plant*
4502	*45*	*Michigan Av.*	*392*	*39247*	*Chicaco IL*
8201	*82*	*Main St.*	*152*	*45931*	*Springfield City*
0701	*07*	*Atom St.*	*55*	*55555*	*Power Plant*
0702	*07*	*Burn's Home*	*1*	*89322*	*New York City*
9301	*93*	*Main St.*	*152*	*45931*	*Springfield City*
6801	*68*	*Scool St.*	*22*	*45937*	*Springfield Edu*
6802	*68*	*Rodeo St.*	*91*	*24742*	*Austin TX*
2301	*23*	*Main St.*	*152*	*45931*	*Springfield City*
7301	*73*	*Beer St.*	*88*	*49287*	*Springfield Downtown*
5801	*58*	*Main St.*	*152*	*45931*	*Springfield City*
3601	*36*	*Police*	*911*	*29883*	*Springfield County*
3602	*36*	*Vacation St.*	*99*	*90481*	*Bozeman MT*

Fügen Sie die Rows wie folgt ein:

```
INSERT INTO ADRESSE VALUES (1301, 13, 'Main St.', 150, 45931, 'Springfield City');
INSERT INTO ADRESSE VALUES (4501, 45, 'Atom St.', 49, 55556, 'Power Plant');
INSERT INTO ADRESSE VALUES (4502, 45, 'Michigan Av.', 392, 39247, 'Chicago IL');
INSERT INTO ADRESSE VALUES (8201, 82, 'Main St.', 152, 45931, 'Springfield City');
INSERT INTO ADRESSE VALUES (0701, 07, 'Atom St.', 55, 55555, 'Power Plant');
INSERT INTO ADRESSE VALUES (0702, 07, 'Burn''s Home', 1, 89322, 'New York City');
INSERT INTO ADRESSE VALUES (9301, 93, 'Main St.', 152, 45931, 'Springfield City');
INSERT INTO ADRESSE VALUES (6801, 68, 'Scool St.', 22, 45937, 'Springfield Edu');
INSERT INTO ADRESSE VALUES (6802, 68, 'Rodeo St.', 91, 24742, 'Austin TX');
INSERT INTO ADRESSE VALUES (2311, 23, 'Main St.', 152, 45931, 'Springfield City');
INSERT INTO ADRESSE VALUES (7301, 73, 'Beer St.', 88, 49287, 'Springfield Downtown');
INSERT INTO ADRESSE VALUES (5801, 58, 'Main St.', 152, 45931, 'Springfield City');
INSERT INTO ADRESSE VALUES (3601, 36, 'Police', 911, 29883, 'Springfield County');
INSERT INTO ADRESSE VALUES (3602, 36, 'Vacation St.', 99, 90481, 'Bozeman MT');
COMMIT;
```

Aufgabe 5

Finden Sie die Indexnamen heraus, welche auf den Tabellen `PERSON` und `ADRESSE` existieren. Tipp: verwenden Sie die System-View `USER_INDEXES`.

Lassen Sie sich die Definition von `USER_INDEXES` anzeigen:

```
DESCRIBE USER_INDEXES

Name                          Null?      Type
====                          =====      ====
INDEX_NAME                    NOT NULL   VARCHAR2(30)  ← für uns interessant
INDEX_TYPE                               VARCHAR2(27)
TABLE_OWNER                   NOT NULL   VARCHAR2(30)
TABLE_NAME                    NOT NULL   VARCHAR2(30)  ← für uns interessant
TABLE_TYPE                               VARCHAR2(11)
UNIQUENESS                               VARCHAR2(9)
COMPRESSION                              VARCHAR2(8)
PREFIX_LENGTH                            NUMBER
TABLESPACE_NAME                          VARCHAR2(30)
INI_TRANS                                NUMBER
MAX_TRANS                                NUMBER
INITIAL_EXTENT                           NUMBER
NEXT_EXTENT                              NUMBER
MIN_EXTENTS                              NUMBER
MAX_EXTENTS                              NUMBER
PCT_INCREASE                             NUMBER
PCT_THRESHOLD                            NUMBER
INCLUDE_COLUMN                           NUMBER
FREELISTS                                NUMBER
FREELIST_GROUPS                          NUMBER
PCT_FREE                                 NUMBER
LOGGING                                  VARCHAR2(3)
BLEVEL                                   NUMBER
LEAF_BLOCKS                              NUMBER
DISTINCT_KEYS                            NUMBER
AVG_LEAF_BLOCKS_PER_KEY                  NUMBER
AVG_DATA_BLOCKS_PER_KEY                  NUMBER
CLUSTERING_FACTOR                        NUMBER
STATUS                                   VARCHAR2(8)
NUM_ROWS                                 NUMBER
SAMPLE_SIZE                              NUMBER
LAST_ANALYZED                            DATE
DEGREE                                   VARCHAR2(40)
INSTANCES                                VARCHAR2(40)
PARTITIONED                              VARCHAR2(3)
TEMPORARY                                VARCHAR2(1)
GENERATED                                VARCHAR2(1)
SECONDARY                                VARCHAR2(1)
BUFFER_POOL                              VARCHAR2(7)
USER_STATS                               VARCHAR2(3)
DURATION                                 VARCHAR2(15)
PCT_DIRECT_ACCESS                        NUMBER
ITYP_OWNER                               VARCHAR2(30)
ITYP_NAME                                VARCHAR2(30)
PARAMETERS                               VARCHAR2(1000)
GLOBAL_STATS                             VARCHAR2(3)
DOMIDX_STATUS                            VARCHAR2(12)
DOMIDX_OPSTATUS                          VARCHAR2(6)
FUNCIDX_STATUS                           VARCHAR2(8)
JOIN_INDEX                               VARCHAR2(3)
```

Die Ausgabe liefert uns 50 Kolonnen, von denen für uns nur 2 Kolonnen interessant sind: **INDEX_NAME** und **TABLE_NAME**. Mit diesen Kolonnen gelangen Sie an die nötigen Informationen.

Führen Sie nun das folgende Statement aus:

Enter statements:

```
  SELECT TABLE_NAME,
         INDEX_NAME
    FROM USER_INDEXES
   WHERE TABLE_NAME='PERSON'
      OR TABLE_NAME='ADRESSE';
```

Die Ausgabe sollte in etwa so aussehen:

Output:

```
TABLE_NAME    INDEX_NAME
==========    ==========
ADRESSE       IX_ADRESSE_P_ID
PERSON        SYS_C00118862
ADRESSE       SYS_C00118863
```

- `IX_ADRESSE_P_ID` ist der Name des Indexes auf dem Foreign Key `P_ID` der Tabelle `ADRESSE`, welchen Sie in Aufgabe 2 erstellt hatten
- `SYS_C00118863` ist der Index auf dem Primary Key `A_ID` der Tabelle `ADRESSE`
- `SYS_C00118862` ist der Index auf dem Primary Key `P_ID` der Tabelle `PERSON`

Die Indexnamen für die Primary Keys (`SYS_.....`) wurden von Oracle automatisch vergeben. Deshalb werden diese Indexe in Ihrem Datenbank-Schema mit grosser Sicherheit anders heissen.

Schreiben Sie sich diese Namen auf – Sie werden sie in den weiteren Aufgaben benötigen.

Aufgabe 6

Schreiben Sie ein Query, welches alle Personen inklusive deren Adresse ausgibt. Die Ausgabe soll die folgenden Kolonnen enthalten:

P_ID A_ID ANREDE NAME VORNAME GEB_DAT STRASSE HAUSNR PLZ ORT

Lassen Sie sich auch den Query Plan und die benötigte Zeit anzeigen. Welche Indexes werden benutzt?

Enter statements:

```
  SELECT p.P_ID,
         a.A_ID,
         p.ANREDE,
         p.NAME,
         p.VORNAME,
         p.GEB_DAT,
         a.STRASSE,
         a.HAUSNR,
         a.PLZ,
         a.ORT
    FROM PERSON p,
         ADRESSE a
   WHERE p.P_ID=a.P_ID;
```

Die Ausgabe sollte in etwa so aussehen:

Output:

```
P_ID A_ID A NAME       VORNAME      GEB_DAT  STRASSE      HAUSN   PLZ ORT
==== ==== = ====       =======      =======  =======      =====   === ===
  13 1301 H van Houten Milhouse     13.02.83 Main St.       150 45931 Springfield City
  45 4501 H Smithers   Waylon       29.10.57 Atom St.        49 55556 Power Plant
  45 4502 H Smithers   Waylon       29.10.57 Michigan Av.   392 39247 Chicago IL
  82 8201 F Simpson    Marge        20.04.59 Main St.       152 45931 Springfield City
   7  701 H Burns      Monty        27.12.52 Atom St.        55 55555 Power Plant
   7  702 H Burns      Monty        27.12.52 Burn's Home      1 89322 New York City
  93 9301 H Simpson    Bart         10.05.83 Main St.       152 45931 Springfield City
  68 6801 H Skinner    Seymour      05.07.60 Scool St.       22 45937 Springfield Edu
  68 6802 H Skinner    Seymour      05.07.60 Rodeo St.       91 24742 Austin TX
  23 2311 H Simpson    Homer Jay    12.05.56 Main St.       152 45931 Springfield City
  73 7301 H Szyslak    Moe          14.09.59 Beer St.        88 49287 Springfield Downtown
  58 5801 F Simpson    Lisa         23.11.85 Main St.       152 45931 Springfield City
  36 3601 H Wiggum     Chief Clancy 02.03.58 Police         911 29883 Springfield County
  36 3602 H Wiggum     Chief Clancy 02.03.58 Vacation St.    99 90481 Bozeman MT

Elapsed: 00:00:00.09

Execution Plan
----------------------------------------------------------
  0   SELECT STATEMENT Optimizer=CHOOSE
  1 0   NESTED LOOPS
  2 1     TABLE ACCESS (FULL) OF 'ADRESSE'
  3 1     TABLE ACCESS (BY INDEX ROWID) OF 'PERSON'
  4 3       INDEX (UNIQUE SCAN) OF 'SYS_C00118862' (UNIQUE)
```

Es wird lediglich der Index `SYS_C00118862` benutzt, d.h. der Index des Primary Keys `P_ID` der Tabelle `PERSON`. Auf der Tabelle `ADRESSE` macht Oracle einen Full Table Scan – dies erkennen Sie an der Zeile 2 im Query Plan.

Aufgabe 7

Weisen Sie nun den Optimizer an, dass er das Query so optimieren soll, dass die erste Row so schnell als möglich retourniert wird. Wie sieht das Query nun aus? Hat sich der Query Plan verändert?

Verwenden Sie den Query Hint `FIRST_ROWS(1)`. Führen Sie das folgende Statement aus:

Enter statements:

```
SELECT /*+ FIRST_ROWS(1) */
       p.P_ID,
       a.A_ID,
       p.ANREDE,
       p.NAME,
       p.VORNAME,
       p.GEB_DAT,
       a.STRASSE,
       a.HAUSNR,
       a.PLZ,
       a.ORT
  FROM PERSON p,
       ADRESSE a
 WHERE p.P_ID=a.P_ID;
```

Die Ausgabe sollte in etwa so aussehen (das Resultat wird hier nicht angezeigt):

Output:

```
Elapsed: 00:00:00.11

Execution Plan
----------------------------------------------------------
  0   SELECT STATEMENT Optimizer=HINT: FIRST_ROWS (Cost=3 Card=1 B ytes=195)
  1 0   NESTED LOOPS (Cost=3 Card=1 Bytes=195)
  2 1     TABLE ACCESS (FULL) OF 'ADRESSE' (Cost=2 Card=1 Bytes=136)
  3 1     TABLE ACCESS (BY INDEX ROWID) OF 'PERSON' (Cost=1 Card=1 Bytes=59)
  4 3       INDEX (UNIQUE SCAN) OF 'SYS_C00118862' (UNIQUE)
```

In Zeile 0 sehen Sie, dass der Optimizer Ihre Anweisung befolgt hat.

Resultat: der Query Plan sieht genau gleich aus wie der Query Plan in Aufgabe 6! Zum Vergleich hier nochmals das Resultat aus Aufgabe 6:

Output:

```
Elapsed: 00:00:00.09

Execution Plan
----------------------------------------------------------
  0   SELECT STATEMENT Optimizer=CHOOSE
  1 0   NESTED LOOPS
  2 1     TABLE ACCESS (FULL) OF 'ADRESSE'
  3 1     TABLE ACCESS (BY INDEX ROWID) OF 'PERSON'
  4 3       INDEX (UNIQUE SCAN) OF 'SYS_C00118862' (UNIQUE)
```

Der Unterschied ist lediglich, dass der Output mehr Informationen über die Kosten enthält.

Schlussfolgerung: der Query Hint hat nichts gebracht. Die Tabelle enthält derart wenig Rows, so dass Oracle in Aufgabe 6 bereits selbst diese Strategie gewählt hat!

Aufgabe 8

Sie wollen alle Rows lesen. Weisen Sie den Optimizer so an, dass er das Query diesbezüglich optimiert. Wie sieht das Query nun aus? Hat sich der Query Plan verändert?

Verwenden Sie den Query Hint `ALL_ROWS`. Führen Sie das folgende Statement aus:

Enter statements:

```
SELECT /*+ ALL_ROWS */
       p.P_ID,
       a.A_ID,
       p.ANREDE,
       p.NAME,
       p.VORNAME,
       p.GEB_DAT,
       a.STRASSE,
       a.HAUSNR,
       a.PLZ,
       a.ORT
  FROM PERSON p,
       ADRESSE a
 WHERE p.P_ID=a.P_ID;
```

Die Ausgabe sollte in etwa so aussehen (das Resultat wird hier nicht angezeigt):

Output:

```
Elapsed: 00:00:00.15

Execution Plan
----------------------------------------------------------
  0   SELECT STATEMENT Optimizer=HINT: ALL_ROWS (Cost=5 Card=82 By tes=11152)
  1 0   HASH JOIN (Cost=5 Card=82 Bytes=11152)
  2 1     TABLE ACCESS (FULL) OF 'PERSON' (Cost=2 Card=82 Bytes=4838)
  3 1     TABLE ACCESS (FULL) OF 'ADRESSE' (Cost=2 Card=82 Bytes=6314)
```

In Zeile 0 sehen Sie, dass der Optimizer Ihre Anweisung befolgt hat.

Nun hat sich der Query Plan geändert! Zu unserem Erstaunen macht Oracle nun auf beiden Tabellen einen Full Table Scan, was Sie an den Zeilen 2 und 3 erkennen.

Auf den ersten Blick mag das komisch erscheinen, da wir ja gelernt haben, dass der Zugriff mit Index schneller ist als ohne. Oracle hat jedoch erkannt, dass es keine `JOIN`-Bedingung gibt, welche die Anzahl Rows in `PERSON` oder `ADRESSE` einschränken würde. Jede Row von jeder Tabelle muss für den JOIN gelesen werden. Deshalb entscheidet sich Oracle (resp. der Optimizer) dafür, direkt einen **Full Table Scan** zu machen. Das ist in diesem Fall sogar schneller als der Zugriff via Index, da sowieso jede Row "besucht" werden muss. Der Index würde hier nur einen unnötigen Overhead verursachen, da wir nicht nach einer bestimmten ID suchen.

Aufgabe 9

In Aufgabe 6 haben Sie gesehen, dass der Optimizer die Indexes der Tabelle ADRESSE nicht verwendet. Weisen Sie nun den Optimizer an, dass er für das Query alle 3 Indexes verwenden soll! Wie sehen das Query und der Query Plan aus, und wie wirkt sich das auf die Performance aus?

Verwenden Sie den Query Hint **INDEX(...)**, sowie die in Aufgabe 5 ermittelten Indexnamen.

Führen Sie nun das folgende Query aus:

Enter statements:

```
  SELECT /*+ INDEX(p SYS_ C00118862)
             INDEX(a SYS_C00118863)
             INDEX(a IX_ADRESSE_P_ID) */
         p.P_ID,
         a.A_ID,
         p.ANREDE,
         p.NAME,
         p.VORNAME,
         p.GEB_DAT,
         a.STRASSE,
         a.HAUSNR,
         a.PLZ,
         a.ORT
    FROM PERSON p,
         ADRESSE a
   WHERE p.P_ID=a.P_ID;
```

Die Ausgabe sollte in etwa so aussehen (das Resultat wird hier nicht angezeigt):

Output:

```
Elapsed: 00:00:00.09

Execution Plan
----------------------------------------------------------
  0   SELECT STATEMENT Optimizer=CHOOSE (Cost=908 Card=82 Bytes=11152)
  1 0   NESTED LOOPS (Cost=908 Card=82 Bytes=11152)
  2 1     TABLE ACCESS (BY INDEX ROWID) OF 'ADRESSE' (Cost=826 Card=82 Bytes=6314)
  3 2       INDEX (FULL SCAN) OF 'IX_ADRESSE_P_ID' (NON-UNIQUE) (Cost=26 Card=82)
  4 1     TABLE ACCESS (BY INDEX ROWID) OF 'PERSON' (Cost=1 Card=1 Bytes=59)
  5 4       INDEX (UNIQUE SCAN) OF 'SYS_C00118862' (UNIQUE)
```

Nun haben wir das Ziel erreicht, Oracle verwendet die Indexe:

- In Zeile 2 sehen Sie, dass der Zugriff auf die Tabelle ADRESSE via den von uns erstellten Index IX_ADRESSE_P_ID (Foreign Key P_ID) geschieht (Zeile 3).
- In Zeile 4 sehen Sie, dass der Zugriff auf die Tabelle PERSON via den Index des Primary Keys P_ID geschieht (SYS_C00118862, Zeile 5)
- Der Index auf der Kolonne A_ID (SYS_C00118863) wird nach wie vor nicht gebraucht. Dies ist aus nicht nötig, da wir A_ID in der JOIN-Bedingung ja gar nicht verwenden, d.h. es muss auch nicht danach gesucht werden.

Allerdings, einen Haken hat die Sache... die Indexes werden nun verwendet, doch ist das Query nun tatsächlich schneller als in den vorhergehenden Aufgaben?

In Zeile 3 des Query Plans sehen Sie:

```
3 2       INDEX (FULL SCAN) OF 'IX_ADRESSE_P_ID' (NON-UNIQUE) (Cost=26 Card=82)
```

Dies bedeutet, dass Oracle einen **Full Index Scan** auf dem Index IX_ADRESSE_P_ID machen muss. Dies ist nicht weiter erstaunlich, da wir ja alle Rows der Tabelle ADRESSE mit den Rows der Tabelle PERSON JOINen wollen. Das heisst, dass wir auch alle Rows besuchen müssen, was auf einen Full Index Scan heraus läuft.

In diesem Fall ist es schneller, einen **Full Table Scan** zu machen, anstatt einen Full Index Scan. Da wir so oder so alle Rows "besuchen" müssen, können wir uns den Overhead (Suche im Index) sparen, und gleich alle Rows besuchen, indem wir die Tabelle "von oben nach unten" durchgehen.

Oracle hatte demnach mit der ursprünglichen Wahl des Query Plans Recht – ein Full Table Scan ist hier besser als ein Full Index Scan!

Aufgabe 10

Räumen Sie Ihr Datenbank-Schema wieder auf, indem Sie die erstellten Tabellen wieder löschen!

Führen Sie die folgenden Statements aus:

```
DROP TABLE ADRESSE;
DROP TABLE PERSON;
```

Der von Ihnen erstellte Index IX_ADRESSE_P_ID wird mit dem Löschen der Tabelle ADRESSE automatisch gelöscht.

Beachten Sie die Reihenfolge des Löschens! Die Tabelle ADRESSE müssen Sie vor der Tabelle PERSON löschen, da von der Tabelle ADRESSE **Foreign Key Constraints** zu der Tabelle PERSON bestehen!

Wenn Sie die Tabelle PERSON vor der Tabelle ADRESSE löschen wollen, erhalten Sie den folgenden Fehler, welcher Sie auf den existierenden Constraint (Einschänkung) hinweist:

```
drop table person
           *

ERROR at line 1:
ORA-02449: unique/primary keys in table referenced by foreign keys
```

Alternativ zu den obigen DROP Statements könnten Sie auch das folgende Statement absetzen:

```
DROP TABLE PERSON CASCADE CONSTRAINTS;
```

Dieses DROP-Statement löscht die Tabelle PERSON, folgt allen Primary/Foreign Key Beziehungen, und löscht auch die damit verbundenen Tabellen (kaskadiertes Löschen).

Aber Achtung, diese Art des DROPpens kann gefährlich sein! Besser ist es, die erste Variante mit 2 DROP-Statements zu verwenden.

Anhang A: Weiterführende Quellen

A.1 Online Begleitmaterial zum Leitprogramm

Leitprogramm Homepage	http://db.logging.ch

A.2 Bedeutung von Indexierung

Bedeutung von "Indexierung"	http://de.wikipedia.org/wiki/Index
Simpsons Familie	http://en.wikipedia.org/wiki/List_of_characters_in_The_Simpsons

A.3 Bäume in der Informatik

Binärer Baum	http://de.wikipedia.org/wiki/Binärbaum
Suchbaum	http://de.wikipedia.org/wiki/Suchbaum
Binärer Suchbaum	http://de.wikipedia.org/wiki/Binärer_Suchbaum
B-Baum	http://de.wikipedia.org/wiki/B-Baum
B+-Baum	http://de.wikipedia.org/wiki/B+-Baum
B*-Baum	http://de.wikipedia.org/wiki/B*-Baum
B#-Baum	http://en.wikipedia.org/wiki/B_sharp_tree
2-3-4-Baum	http://de.wikipedia.org/wiki/2-3-4-Baum
Rot-Schwarz-Baum	http://de.wikipedia.org/wiki/Rot-Schwarz-Baum
R-Baum	http://de.wikipedia.org/wiki/R-Baum
R*-Baum	http://de.wikipedia.org/wiki/R*-Baum
k-d-Baum	http://de.wikipedia.org/wiki/K-d-Baum
Quadtree	http://de.wikipedia.org/wiki/Quadtree
UB-Baum	http://de.wikipedia.org/wiki/UB-Baum
Bereichsbaum	http://de.wikipedia.org/wiki/Bereichsbaum
Gridfile	http://de.wikipedia.org/wiki/Gridfile
AVL-Baum	http://de.wikipedia.org/wiki/AVL-Baum
Fibonacci-Baum	http://de.wikipedia.org/wiki/Fibonacci-Baum

A.4 Landau-Symbole

Landau-Symbole / Order Notation	http://de.wikipedia.org/wiki/O-Notation

A.5 Datenbanksysteme

Oracle	http://www.oracle.com
MySQL	http://www.mysql.com
PostgreSQL	http://www.postgresql.org
DB2	http://www.ibm.com/db2
Apache Derby	http://db.apache.org/derby
SQLite	http://www.sqlite.org

A.6 Query Hints

Query Hints	http://psoug.org/reference/hints.html

Anhang B: Referenzen

[1] Henry F. Korth, Abraham Silberschatz. *Database System Concepts*. McGraw-Hill, Inc. Serving the Need for Knowledge®. McGraw-Hill, 2nd edition, 1991.

[2] Gavin Powell. *Oracle® High Performance Tuning for 9i and 10g*. Elsevier Digital Press, Elsevier Inc., 2004.

[3] Kathy Rich. *Oracle® Database Reference 10g Release 2 (10.2)*. Oracle Inc., 2009.

[4] A. Kemper, A. Eickler. *Datenbanksysteme – eine Einführung*. R. Oldenburg Verlag, München. Oldenburg, 1996.

[5] St. Feuerstein. *Oracle PL/SQL Best Practices – Optimizing Oracle Code*. O'Reilly & Associates, Inc. O'Reilly, 2001.

[6] D. Ensor, I. Stevenson. *Oracle Design – Database & Code Design*. O'Reilly & Associates, Inc. O'Reilly, 1997.

[7] R. J. Niemiec. *Oracle Performance Tuning*. Osborne / McGraw-Hill, Inc.. Serving the Need for Knowledge®, Osborne / McGraw-Hill, 1999.