

Baumbasierte Indexe

Baumbasierte Struktur / Organisation als Binärbaum

- Haufendateien oder sortierte Dateien → nützlich für statische Dateien
- Dateien organisiert als Binärbaum
 - Effizientes Einfügen und Löschen von Datensätzen
 - Benutzt binäre Suche
- Meist verwendete Indexstrukturen: B-, B⁺- und B^{*}-Bäume
- Speicherstruktur für eine Datei organisiert als Binärbaum:
 - Eine Menge von Knoten; Zeiger zu der Wurzel
 - Eine Liste von freien Knoten

Organisation als Binärbaum

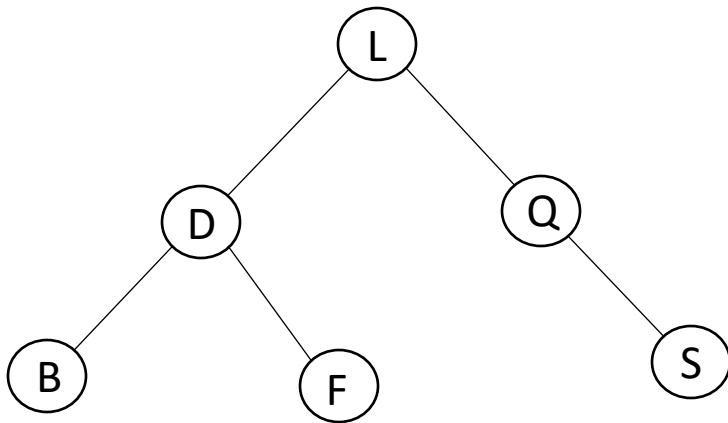
- Ein Eintrag in einem Knoten:

S_i	D_i	V_{i-1}	V_i
-------	-------	-----------	-------

- S_i – Suchschlüssel
- D_i – Datensatz oder Datensatzindetifikator
- V_{i-1} – Verweis auf Knoten, die kleinere Schlüsselwerte enthalten
- V_i – Verweis auf Knoten, die größere Schlüsselwerte enthalten

Organisation als Binärbaum

- Wurzel – Zeiger zu dem Wurzel
- Frei – Zeiger zu dem Listenkopf der freien Knoten
- Binärbaum:



Wurzel = 1 →

Frei = 3 →

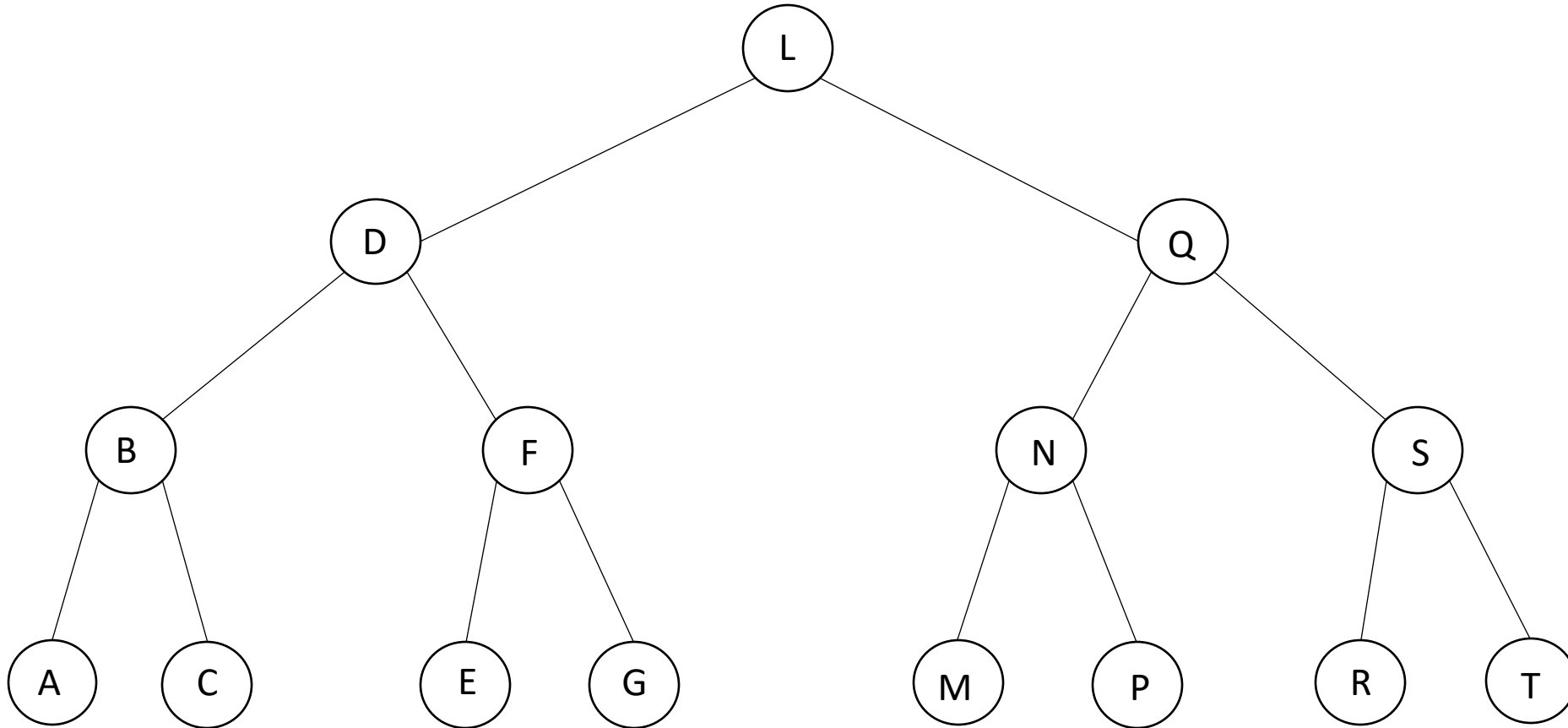
1	L	Daten _L	2	4
2	D	Daten _D	8	7
3			-6	NULL
4	Q	Daten _Q	NULL	5
5	S	Daten _S	NULL	NULL
6			-9	NULL
7	F	Daten _F	NULL	NULL
8	B	Daten _B	NULL	NULL
9			NULL	NULL

Einfügen und Löschen von Datensätzen

- Einen Datensatz einfügen:
 - Finde die Position wo der neue Datensatz eingefügt werden sollten
 - Speichere den Datensatz in einem freien Knoten
 - Füge die Zeiger für den neuen Knoten ein
- Einen Datensatz löschen
 - Suche den Datensatz
 - 3 Fälle:
 - Keine Kinder → Elterns Zeiger = NULL
 - 1 Kind → verbinde das Kind zu dem Elternteil
 - 2 Kinder → ersetze den Knoten mit dem nächsten Nachbarwert
 - Füge den Knoten, der frei geworden ist, zu der Liste mit freien Knoten ein

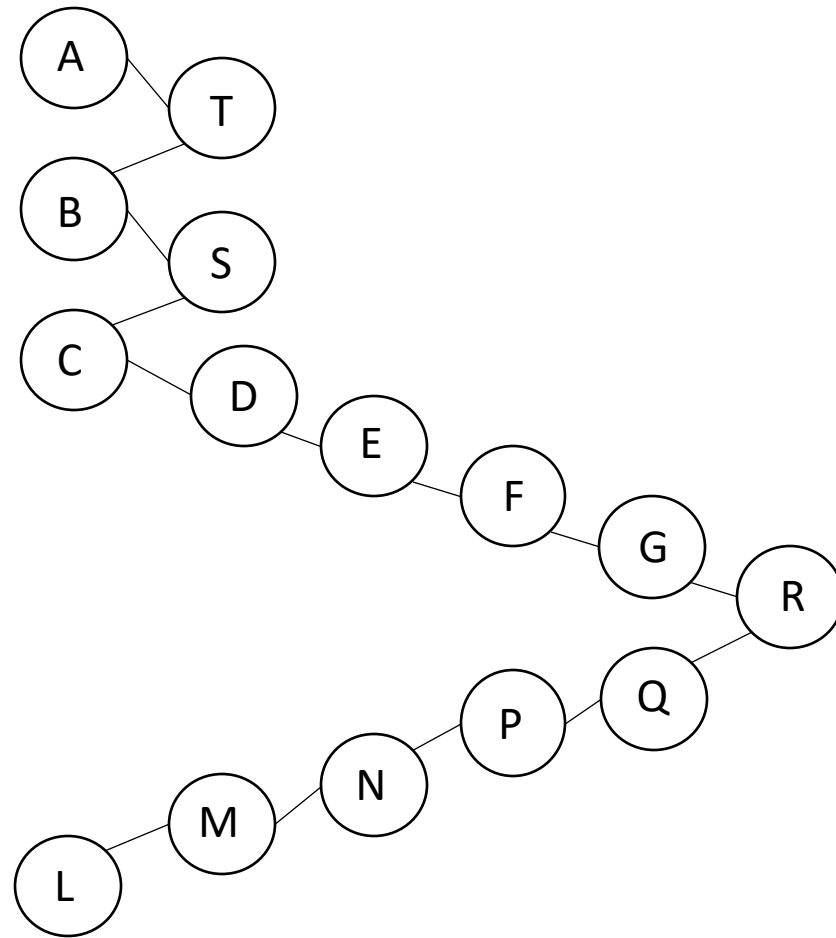
Einfügeanomalie im Binärsuchbaum

- L, D, B, Q, N, F, S, R, T, M, E, G, P, A, C



Einfügeanomalie im Binärsuchbaum

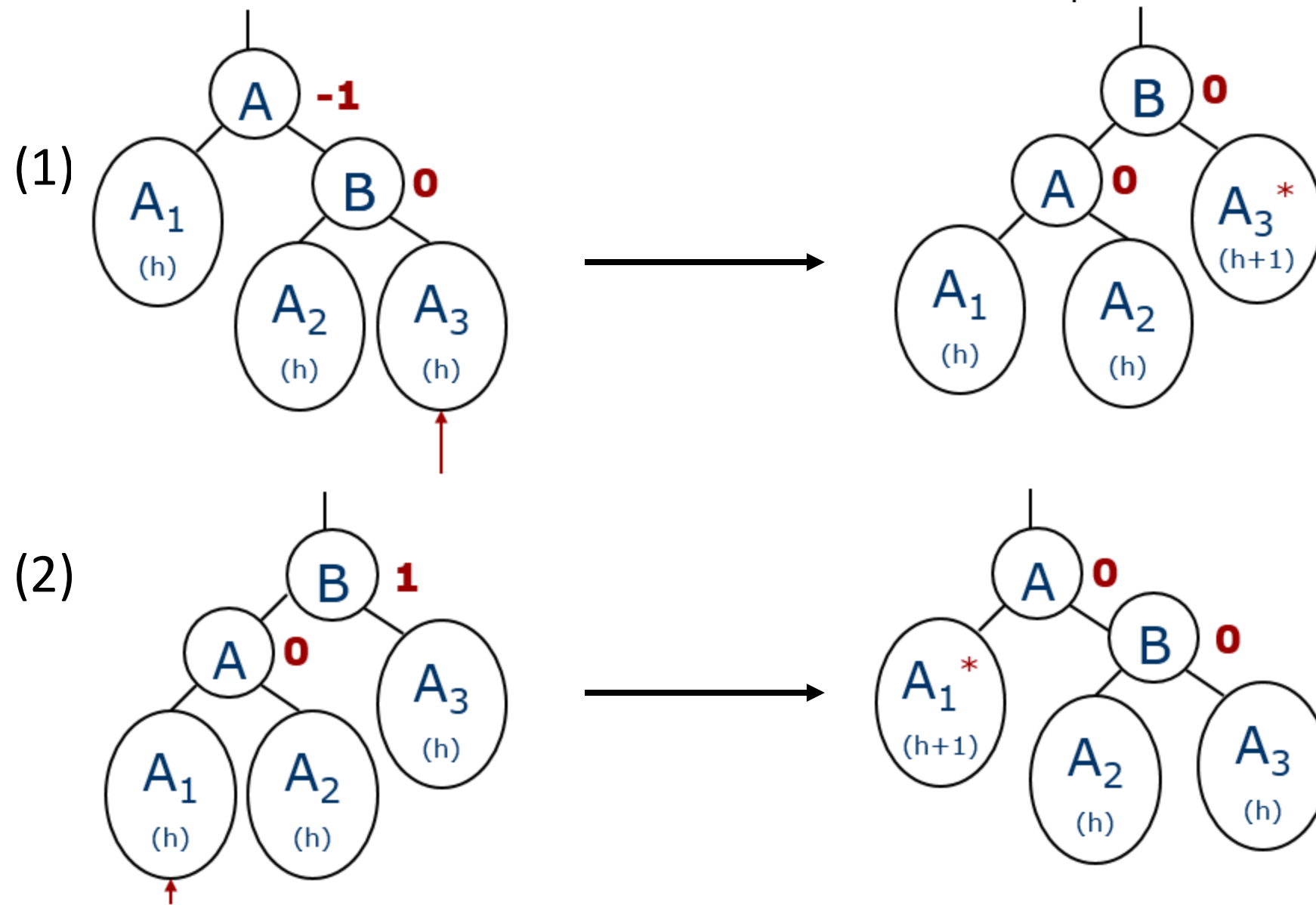
- A, T, B, S, C, D, E, F, G, R, Q, P, N, M, L



Optimale vs. Balancierte Binärbäume

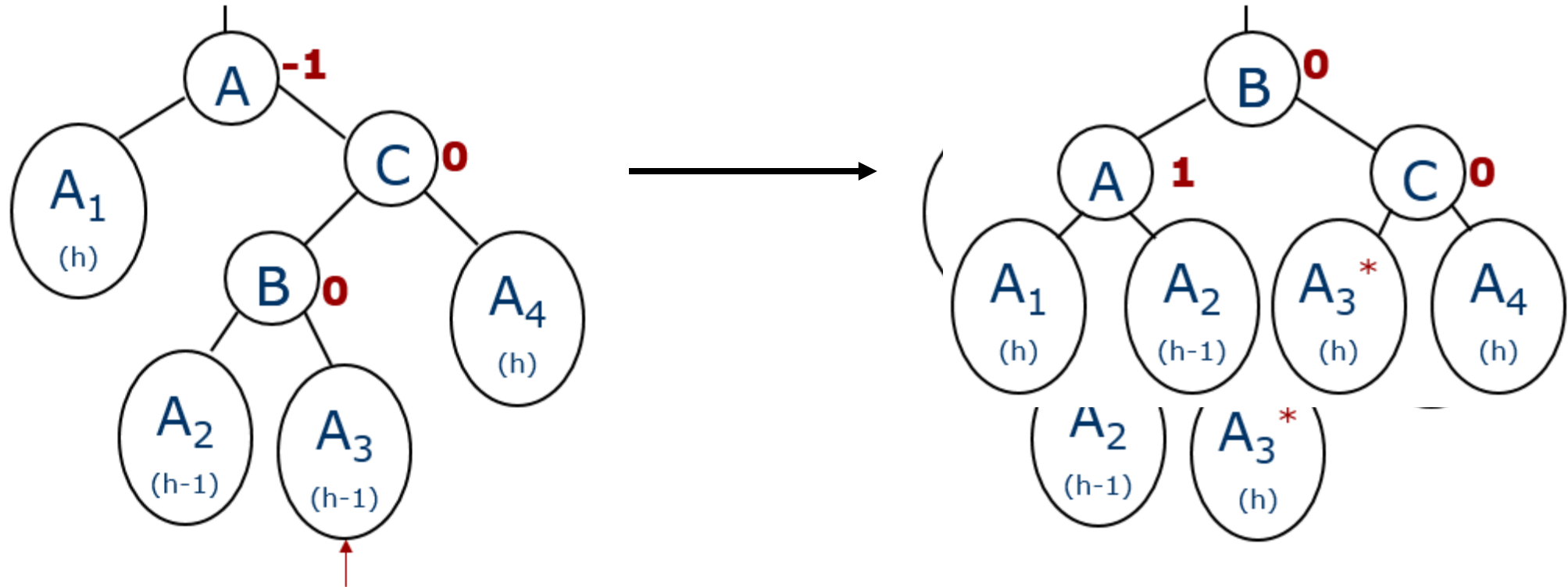
- Optimale Binärbäume:
 - Die Blätter befinden sich auf höchstens Level 2
 - Erhaltung ist aufwendig
- Balancierte Binärbäume:
 - für einen Knoten können die Höhen der zwei Teilbäume mit höchstens 1 voneinander abweichen (Höhe – Länge des längsten Pfades von der Wurzel zu den Blättern)
 - Wenigere Operationen um den Binärbaum balanciert zu behalten
 - Es gibt 6 Fälle wenn der Baum unbalanciert wird nach einem Einfügen

Erhaltung der Balancierten Binärbäumen

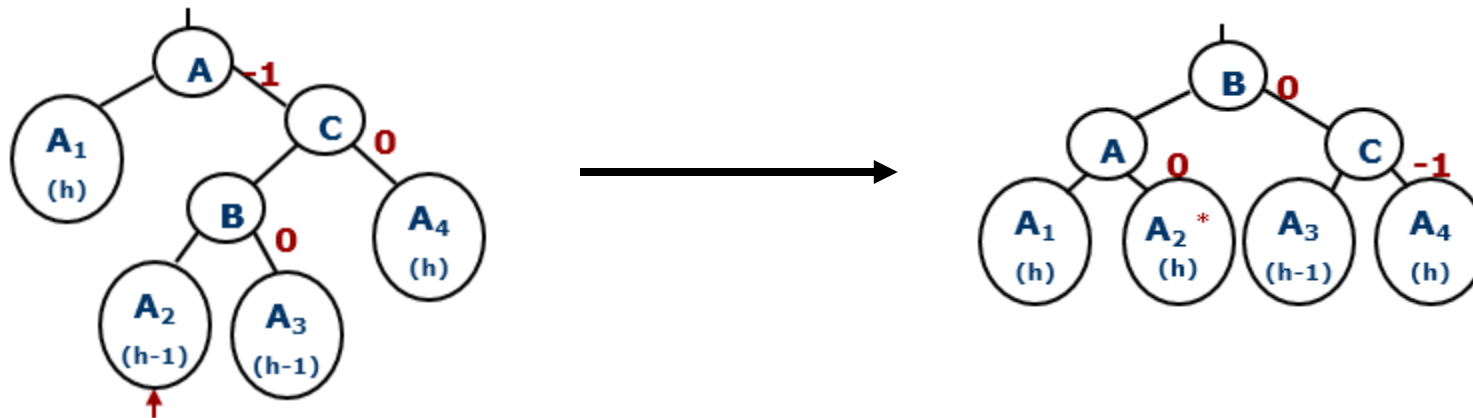


Erhaltung der Balancierten Binärbäumen

(3)

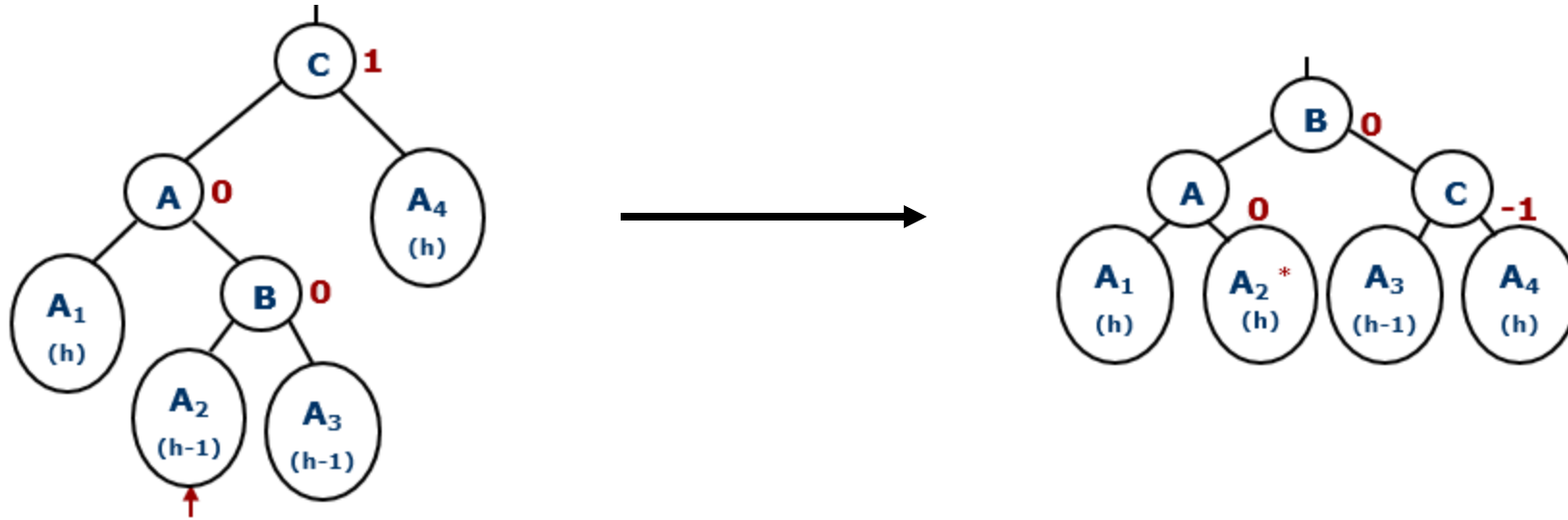


(4)

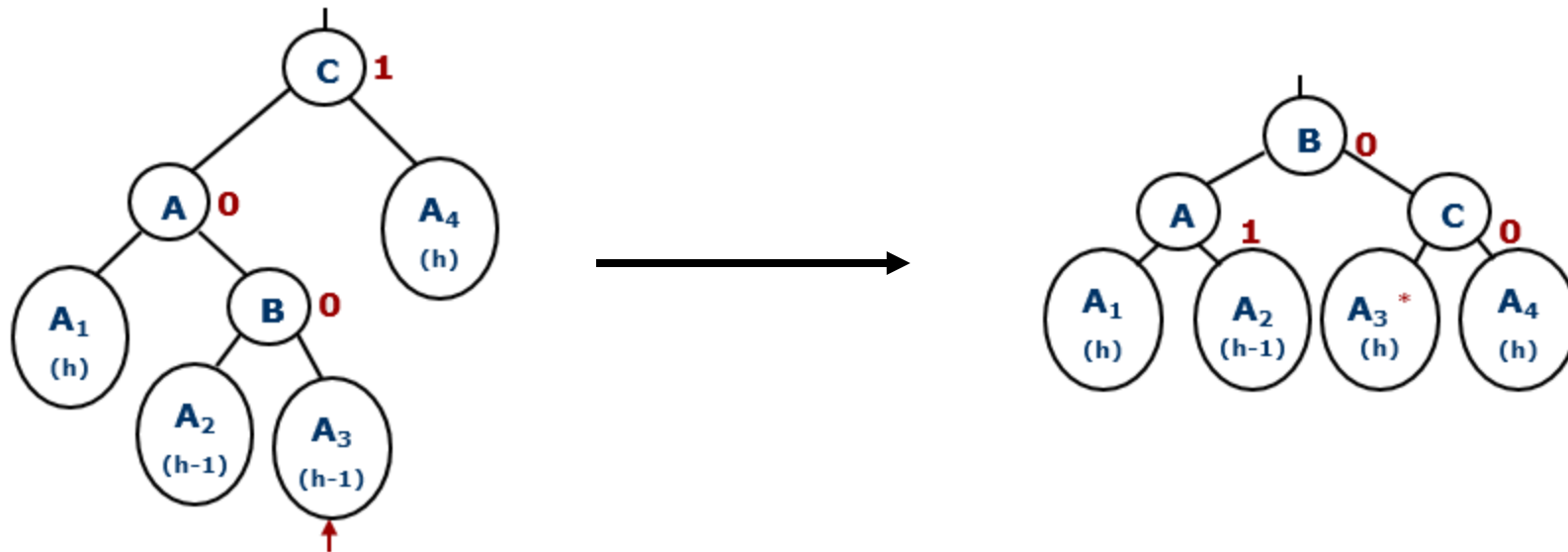


Erhaltung der Balancierten Binärbäumen

(5)

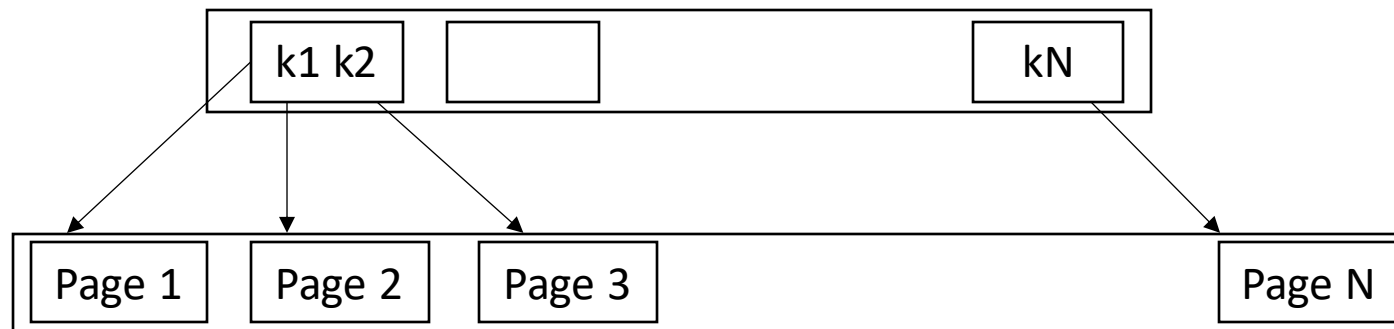


(6)



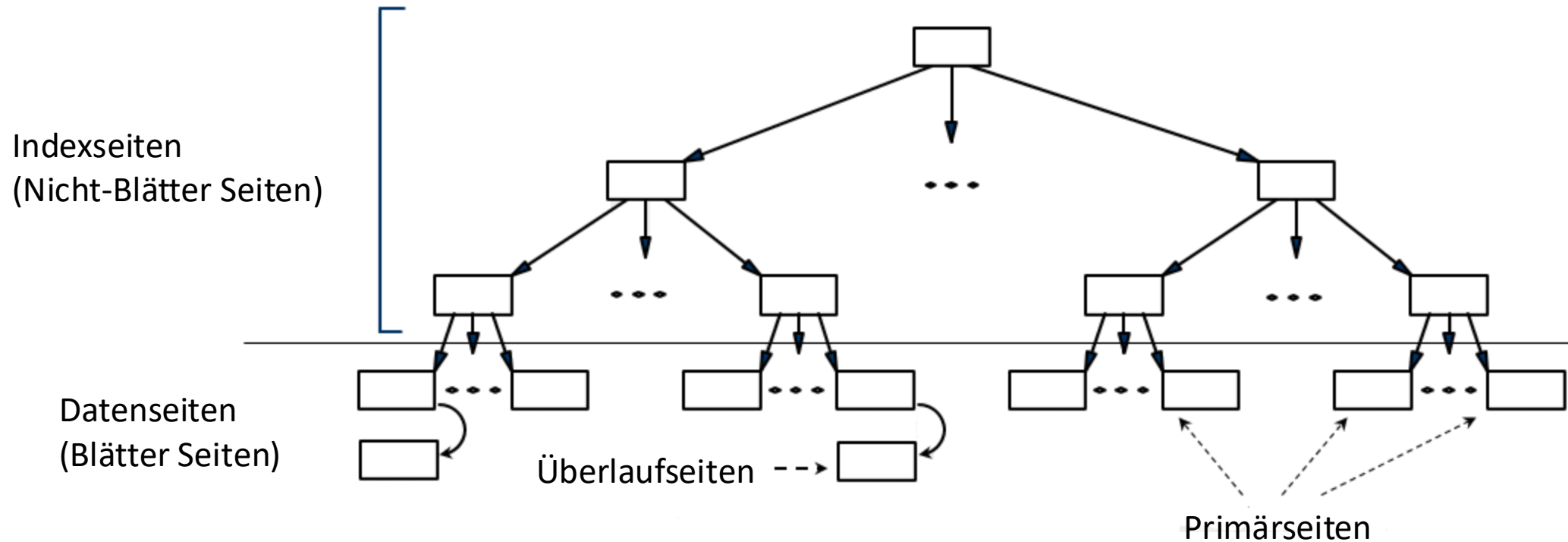
Bereichsanfragen

- Finde alle Studenten mit Note > 8
 - Wenn die Daten sortiert sind, dann Binärsuche um der erste Student zu finden und dann Scannen, um die anderen zu finden
 - Aber: Binärsuche kann teuer sein
- Einfache Idee:
 - Index erstellen → ISAM (Index-Sequential Access Method)
 - Binärsuche auf dem Index (kleiner als die Datei selber)



ISAM (Index-Sequential Access Method)

- Eine sehr einfache und manchmal auch sehr effektive Indexstruktur
- Wie beim Nachschlagen in einem Wörterbuch
- Die Indexdatei kann immer noch zu groß sein → wir können die Prozedur wiederholen (Index zu dem Index, usw.)



ISAM

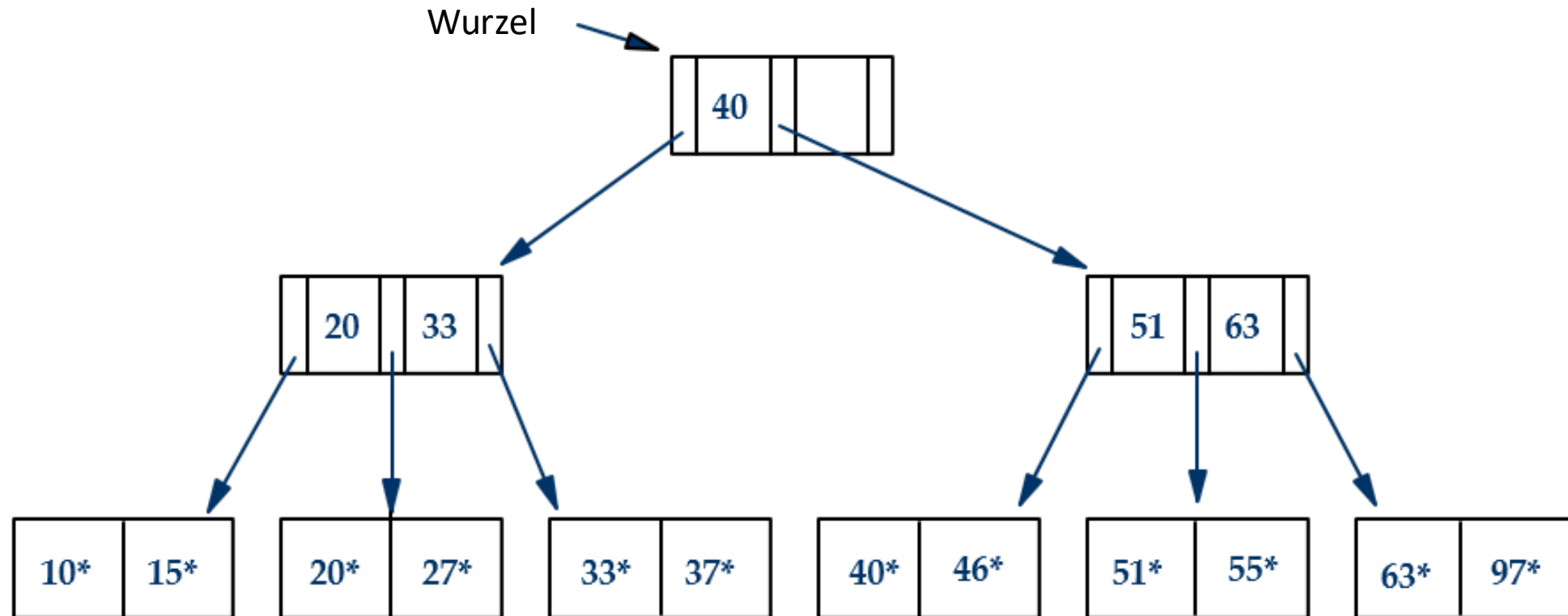
- Sowohl der Index als auch die Datensätze in den Datenseiten werden nach dem Suchschlüssel geordnet abgespeichert
- Der Index befindet sich auf Seiten, die sequentiell hintereinander auf dem Sekundärspeicher abgelegt sind
- Die Erstellung der Dateien:
 - Erst werden die Datenseiten erstellt und sequentiell abgespeichert
 - Dann die Indexseiten
 - Dann wird Speicherplatz für Überlaufseiten reserviert
- Indexeinträge der Form <Suchschlüsselwert, SeitenID>
- Statische Baumstruktur: Einfügen und Löschen von Datensätzen beeinflusst nur die Blätterseiten

ISAM

- Suche eines Schlüssels:
 - beginne bei der Wurzel
 - Vergleiche den Suchschlüssel mit den Suchschlüsseln aus den Indexeinträgen und verfolge die Verweise, um eine Blattseite zu erreichen
 - Von dieser Datenseite an kann man wegen der Sortierung alle weiteren Datenseiten lesen, bis der gewünschte Datensatz gefunden wurde
 - Kosten: $\log_F N$, wobei F = Anzahl von Indexseiten, N = Anzahl von Datenseiten
- Einfügen: finde die Datenseite, wo der neue Datensatz gehören sollte und, wenn es freier Platz gibt, füge den Datensatz ein
 - Falls es kein freier Platz in der Datenseite gibt → Überlaufseite
- Löschen: finde die Datenseite und lösche den Datensatz
 - Falls eine Überlaufseite frei wird → Speicherplatz freigeben

ISAM Baum - Beispiel

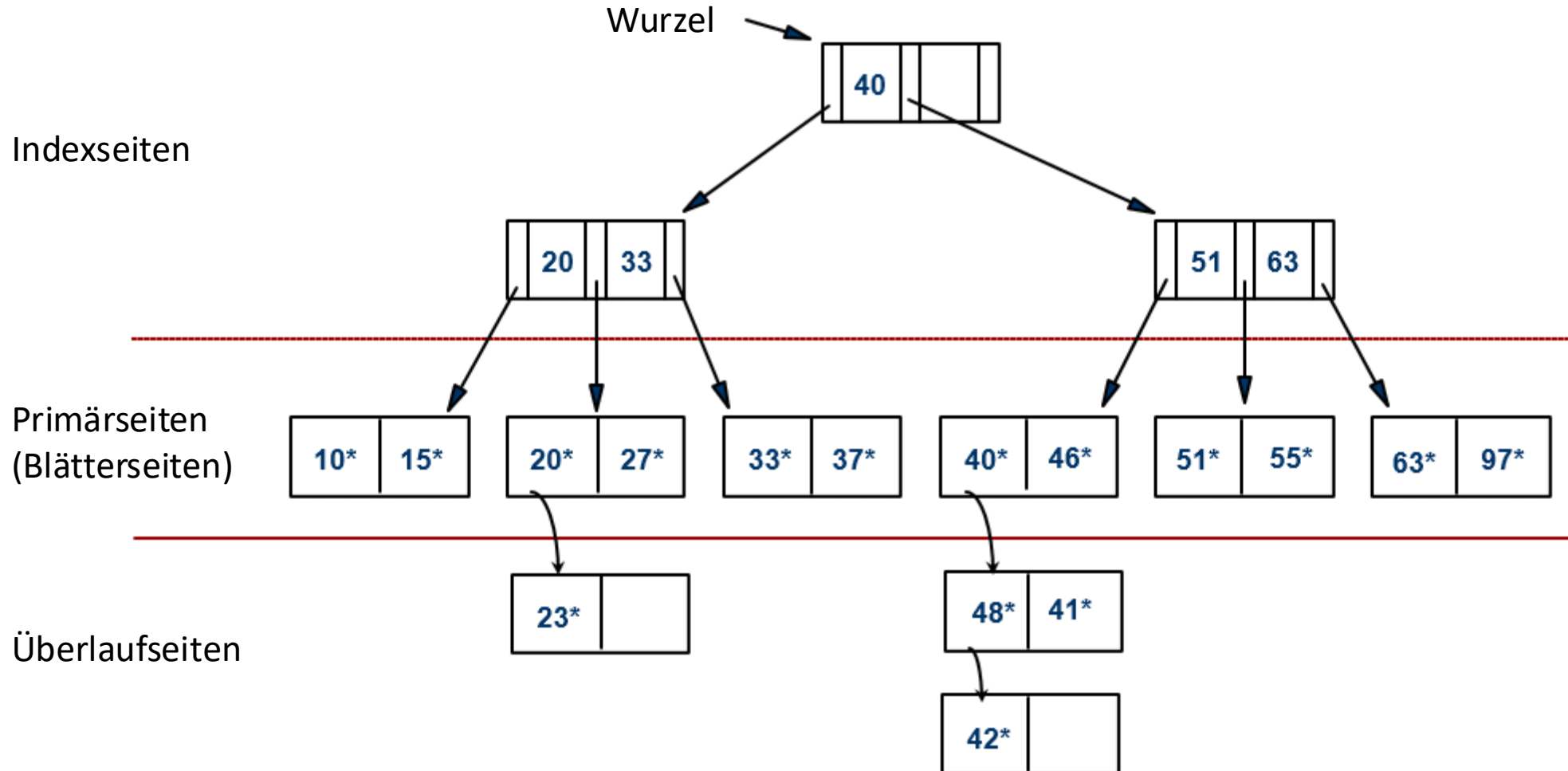
- In jedem Knoten passen zwei Einträge



- Füge folgende Werte ein: 23*, 48*, 41*, 42*

ISAM Baum - Beispiel

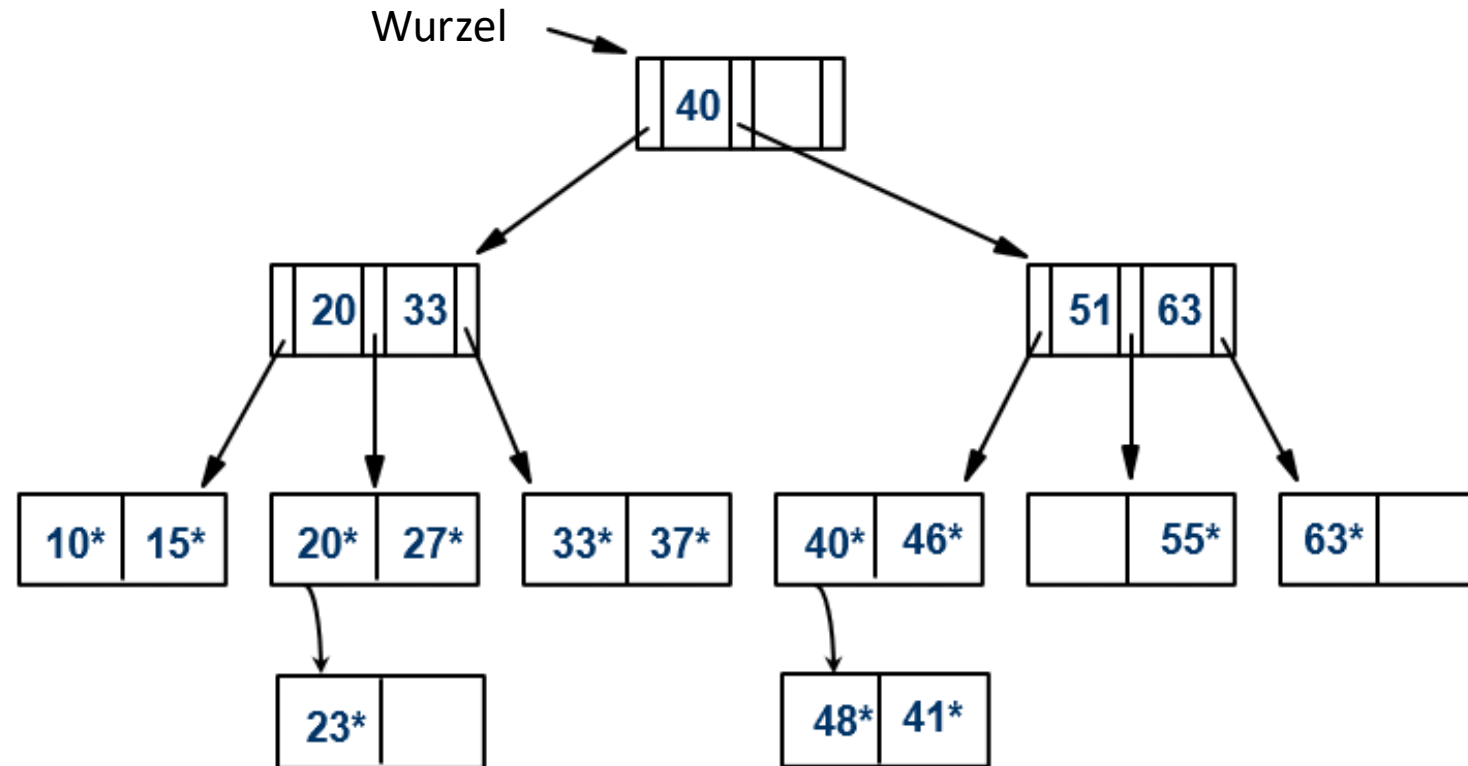
- Nach dem Einfügen von 23^* , 48^* , 41^* , 42^*



- Lösche: 42^* , 51^* , 97^*

ISAM Baum - Beispiel

- Nach dem Löschen von 42*, 51*, 97*
- Bemerkung. Nach dem Löschen von 51* kommt dieser Wert in den Indexseiten immer noch vor (obwohl er von den Datenseiten gelöscht wurde)

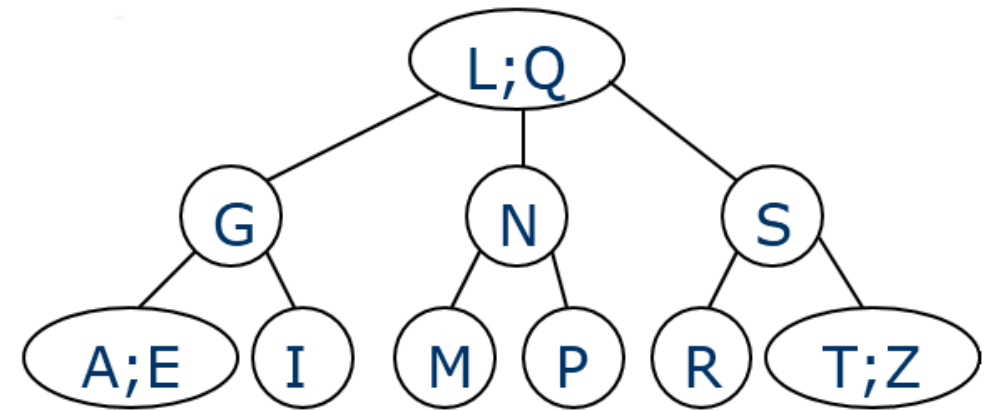


Vor- und Nachteile von ISAM

- Es kann unbalanciert werden nach vielen Einfügen oder Löschen von Datensätzen → ungleichförmiger Suchzeit
- Datensätze in den Überlaufseiten sind meistens nicht sortiert
- Effizientes Einfügen und Löschen (keine Änderungen in den Knoten des Baumes außer den Blättern, keine Baum Balancierung)
- Effizienter gleichzeitiger Zugriff – die Knoten des Baumes werden nicht gesperrt
- Geeignet für Dateien, die nicht viel geändert werden

B-Bäume

- B-Baum – sortierter Baum
- Am meisten verwendete Indexstruktur
- „B“ kommt von „balanced“ oder „broad“
- Definiert in 1970 von R. Bayer und E. M. McCreight von Boeing
- Normale Binärbäume wurden als Suchstruktur für den Hauptspeicher konzipiert und eignen sich nicht für den Sekundärspeicher (in diesem Fall, die Festplatte des Datenbankenservers)
- Für den Hintergrundpeicher: Mehrwegbäume, deren Knotengrößen auf die Seitenkapazität abgestimmt werden
- Ein Knoten des Baumes entspricht einer Seite des Sekundärspeichers



Die Struktur eines Knotens mit m Einträge

V_0	S_1	V_1	S_2	V_2	...	V_{m-1}	S_m	V_m
-------	-------	-------	-------	-------	-----	-----------	-------	-------

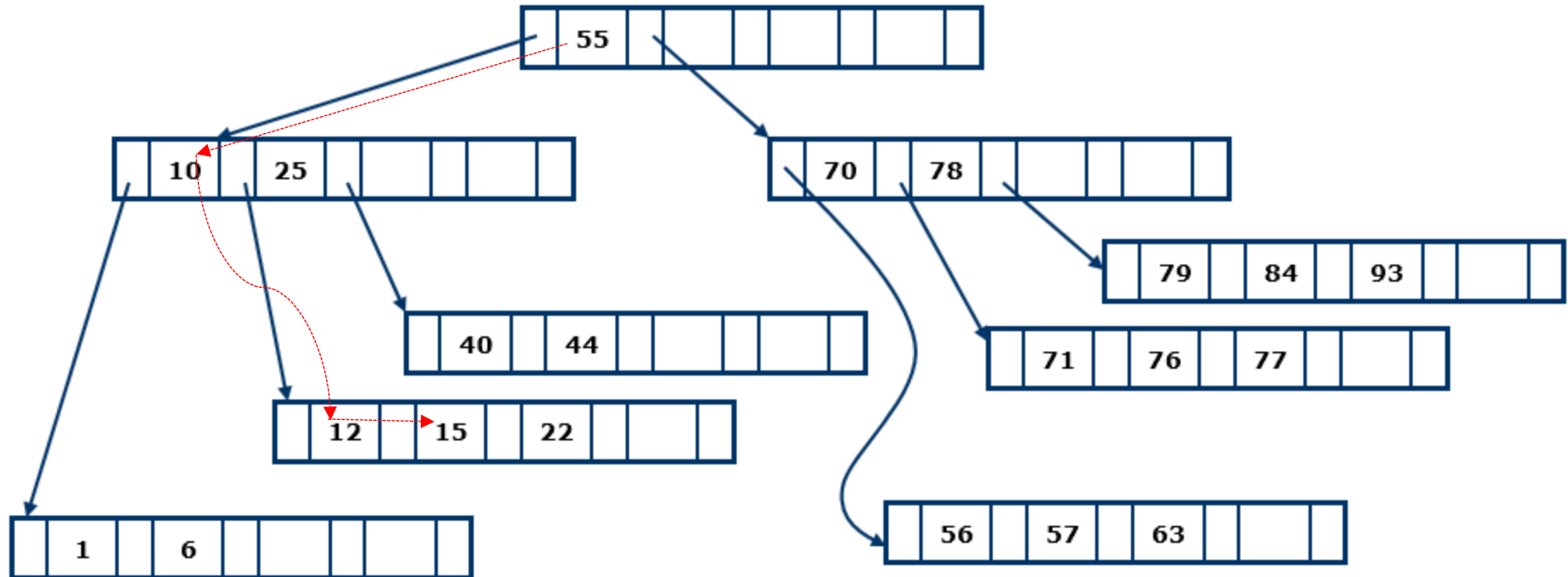
- S_i – Suchschlüsselwerte
- V_i – Zeiger zu einem Teilbaum
- $S_1 < S_2 < \dots < S_m$

B-Baum von Ordnung k - Eigenschaften

- Jeder Weg von der Wurzel zu einem Blatt hat die gleiche Länge
- Jeder Knoten außer der Wurzel hat mindestens k und höchstens $2 \cdot k$ Einträge
- Die Wurzel hat zwischen einem und $2 \cdot k$ Einträgen
- Die Einträge werden in allen Knoten sortiert behalten
- Alle Knoten mit n Einträgen, außer den Blättern, haben $n+1$ Kinder
- Seien S_1, S_2, \dots, S_n die Schlüssel eines Knotens mit $n+1$ Kindern; die Verweise auf diese Kinder: V_0, V_1, \dots, V_n
 - V_0 – weist auf den Teilbaum mit Schlüssel kleiner als S_1
 - V_i ($i = 1, \dots, n-1$) weist auf den Teilbaum, dessen Schlüssel zwischen S_i und S_{i+1} liegen
 - V_n weist auf den Teilbaum mit Schlüssel größer als S_n
- In den Blattknoten sind die Zeiger nicht definiert

Suchen in einem B-Baum

- In jedem Knoten muss man für den Pfad zwischen zwei Zeiger wählen
- Wir suchen „15“:

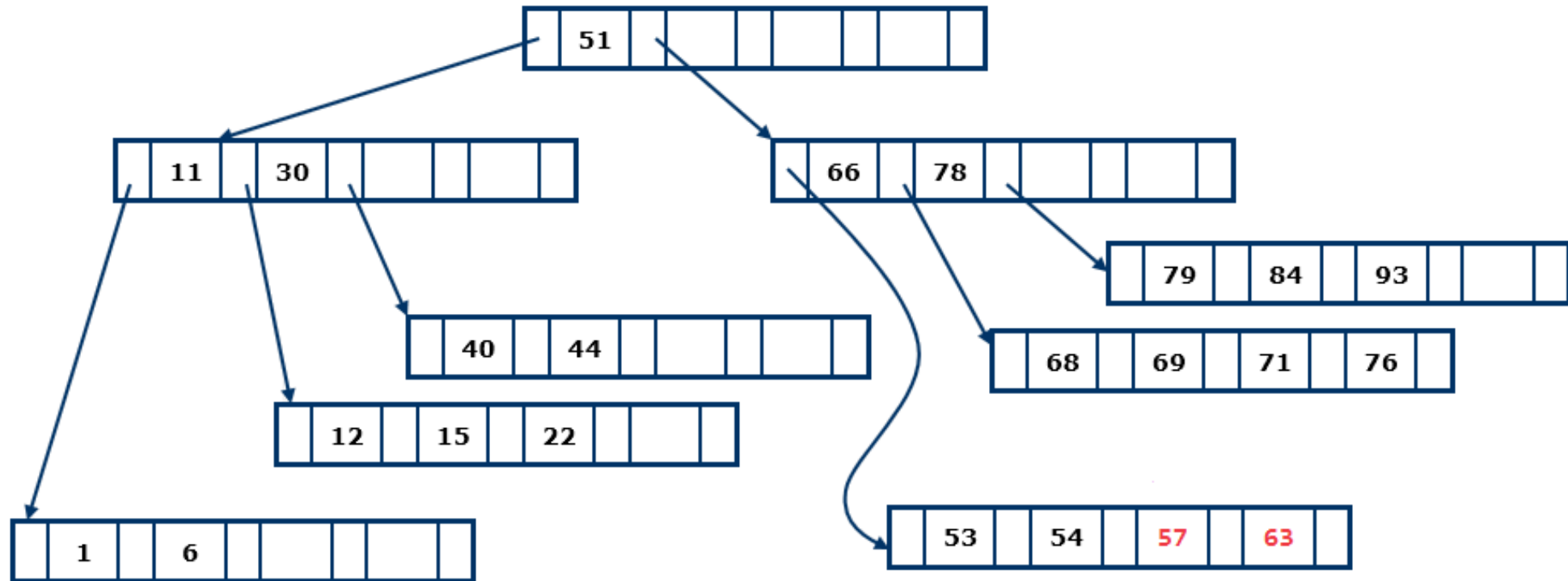


Einfügen in einem B-Baum

- Algorithmus für den Einfügevorgang:
 - Führe eine Suche nach dem Schlüssel, um die Einfügestelle zu finden
 - Füge den Schlüssel dort ein
 - Ist der Knoten überfüllt, teile ihn:
 - Erzeuge einen neuen Knoten und belege ihn mit den Einträgen des überfüllten Knotens, deren Schlüssel größer ist als der des mittleren Eintrags
 - Füge den mittleren Eintrag im Vaterknoten des überfüllten Knotens ein
 - Verbinde den Verweis rechts des neuen Eintrags im Vaterknoten mit dem neuen Knoten
 - Ist der Vaterknoten jetzt überfüllt?
 - Handelt es sich um die Wurzel, so lege eine neue Wurzel an
 - Wiederhole Schritt 3 mit dem Vaterknoten

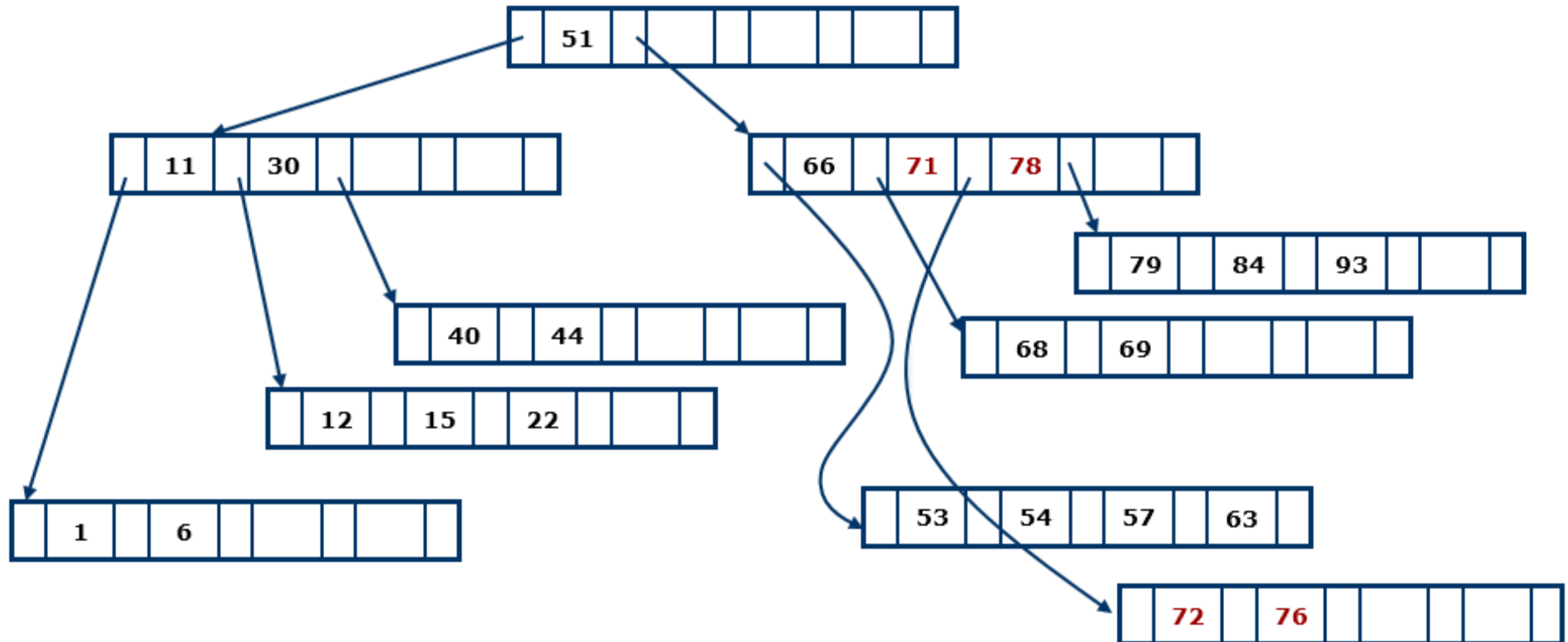
Einfügen in einem B-Baum

- Füge den Schlüssel „57“ ein



Einfügen in einem B-Baum

- Füge den Schlüssel „72“ ein



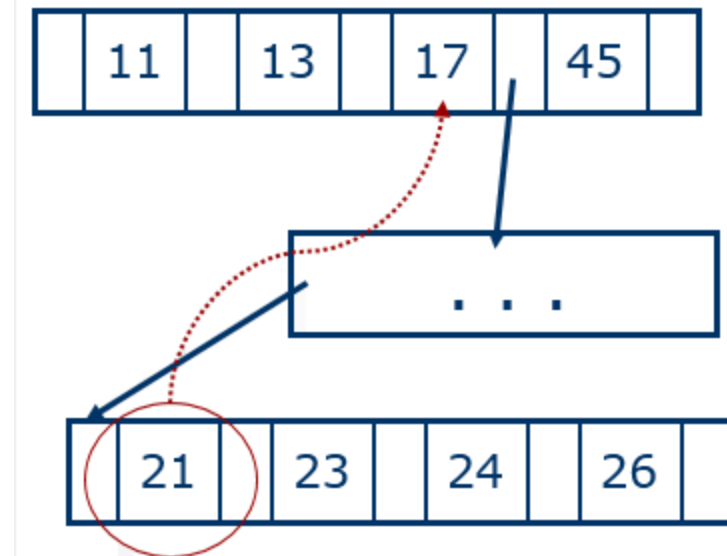
Löschen in einem B-Baum von Ordnung k

- Schritte beim Löschen:
 - Finde den Knoten, der den gewünschten Schlüssel enthält
 - Wenn der Knoten kein Blatt ist, dann kann man ein Schlüssel aus einem Kind anstatt dem gelöschten Schlüssel kopieren
 - Falls es einen Unterlauf gibt in dem Knoten p :
 - Balance – wenn einer der Nachbar genügend Suchschlüssel hat ($> k$), wird seine Suchschlüsselreihe mit der von p ausgeglichen
 - Merge – p wird mit dem Nachbarn zu einem einzigen Knoten verschmolzen

Algorithmus für Löschen in einem B-Baum von Ordnung k

1. Finde den Schlüssel, der gelöscht werden muss

- Wenn sich der Schlüssel in einem inneren Knoten (Nichblattknoten) befindet, dann:
 - Ersetze den Schlüssel mit seinem Nachfolger/ größeren Nachbar (der Schlüssel am weitesten links aus dem Blatt am weitesten links aus dem rechten Teilbaum)

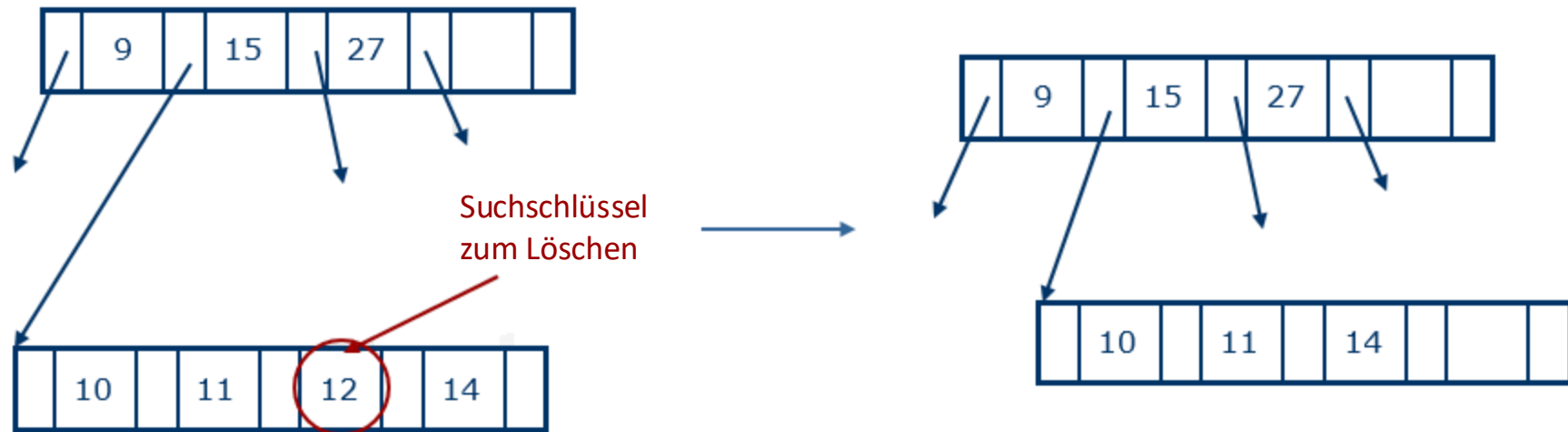


Algorithmus für Löschen in einem B-Baum von Ordnung k

2. Wiederhole diesen Schritt bis man zu Fall A oder Fall B/C gelangt

A. Wenn sich der Suchschlüssel, den wir grade löschen wollen, in der Wurzel befindet **oder** die Anzahl der gebliebenen Schlüssel $\geq k$ ist:

- Lösche den gewünschte Suchschlüssel
- Die Zeiger zu den Suchschlüssel in dem Knoten müssen neu angeordnet werden
- Algorithmus wird beendet

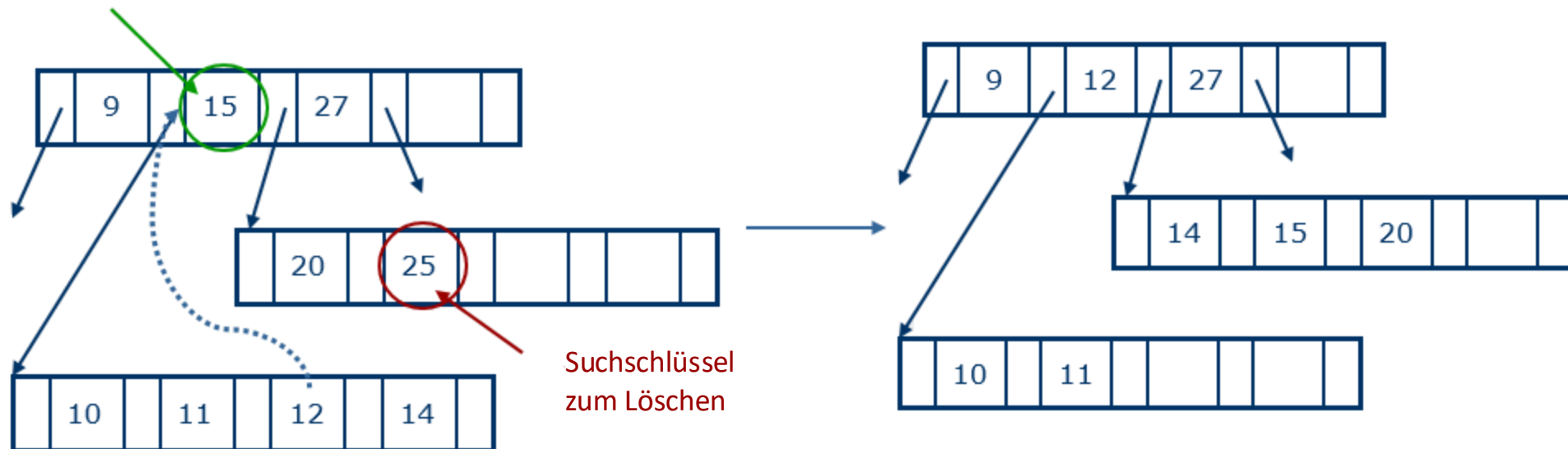


Algorithmus für Löschen in einem B-Baum von Ordnung k

B. Wenn die Anzahl der gebliebenen Suchschlüssel $< k$ ist und ein Nachbarknoten $> k$ Suchschlüssel enthält \rightarrow *Balance*

- Die Schlüssel der zwei Knoten und der Separatorschlüssel aus dem Elternteil werden gleichmäßig umverteilt
- Wähle den mittleren Schlüssel um den Separatorschlüssel in dem Elternteil (übergeordnete Knoten) zu ersetzen
- Algorithmus wird beendet

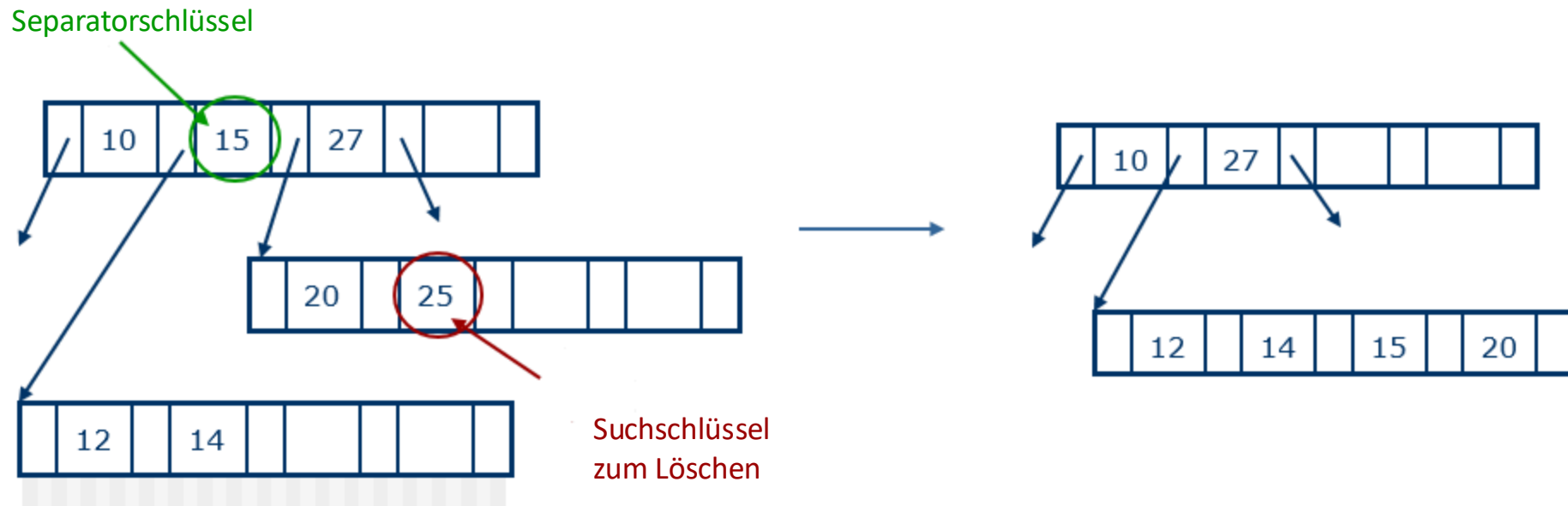
Separatorschlüssel



Algorithmus für Löschen in einem B-Baum von Ordnung k

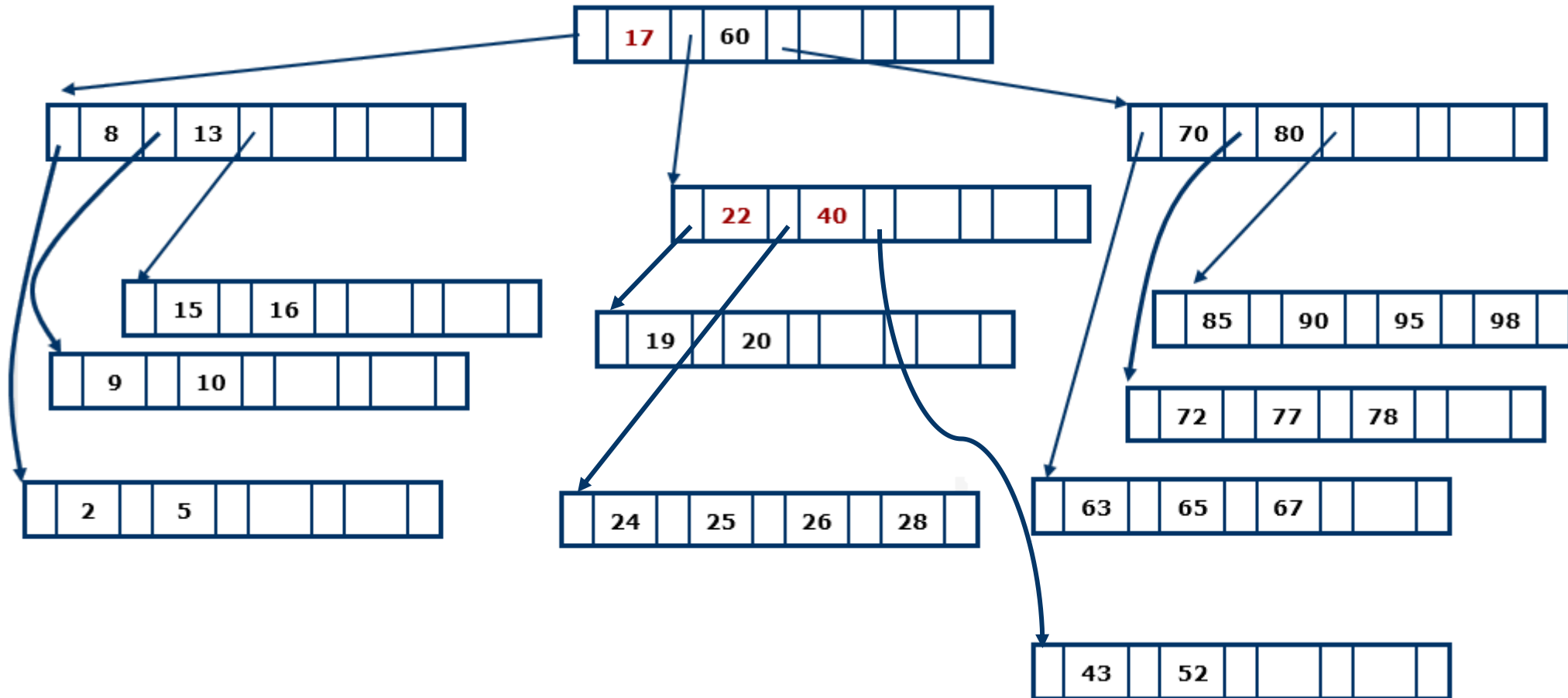
C. Wenn die Anzahl der übriggebliebenen Suchschlüssel zusammen mit der Anzahl der Suchschlüssel in jedem Nachbarknoten k ist: *Merge*

- Verschmelze die zwei Knoten, zusammen mit dem Separatorschlüssel
- Wiederhole Schritt 2 für den **übergeordneten** Knoten
- Wenn der übergeordnete Knoten die Wurzel ist und keine Schlüssel mehr enthält, dann wird der aktuelle Knoten zur Wurzel



Löschen in einem B-Baum

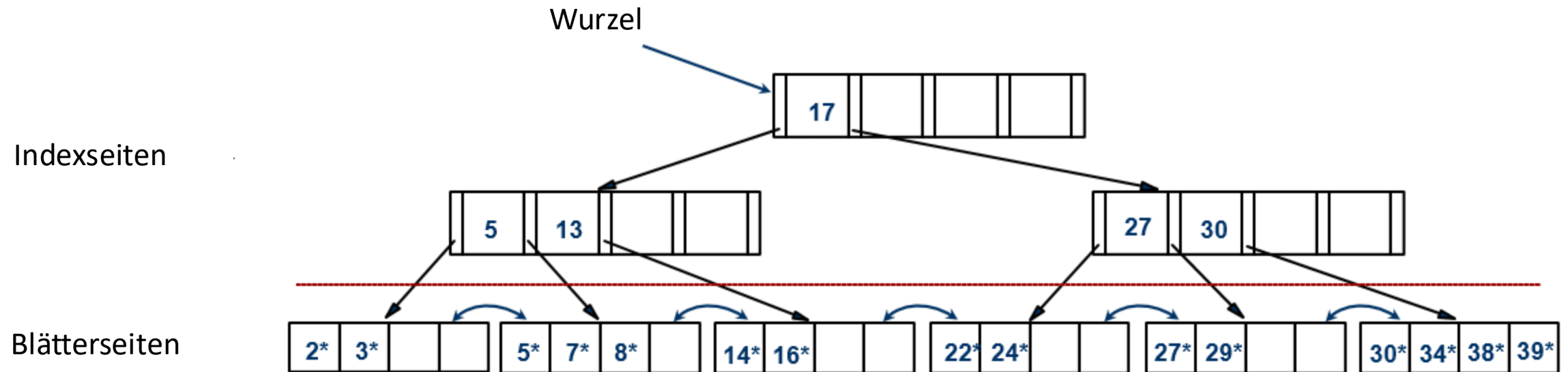
Schlüssel



Nachfolger

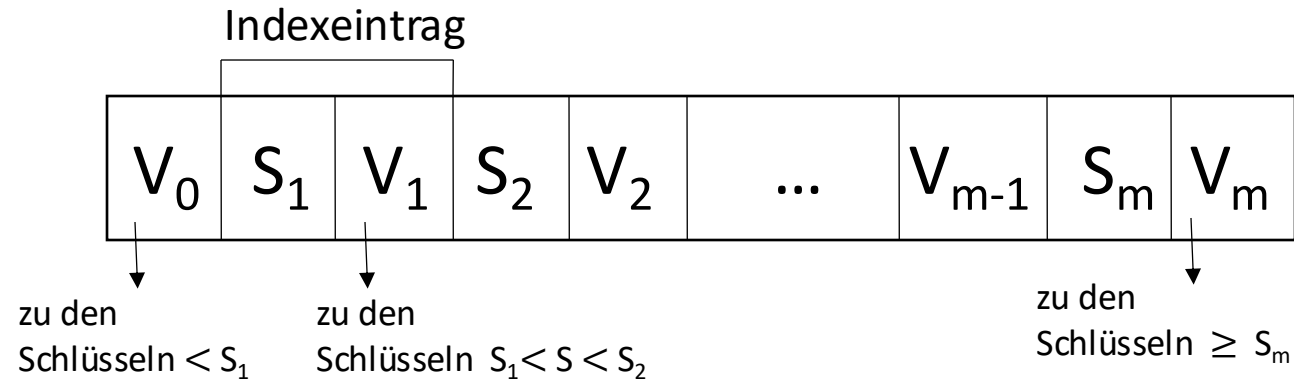
B⁺-Bäume

- Eine Modifikation von B-Bäumen:
 - Alle Zeiger zu den Datensätzen befinden sich in den Blättern
 - Die Suchschlüssel der inneren Knoten dienen als Wegweiser oder Separatoren bei der Suche der Datensätzen
- Ein B⁺-Baum kann weniger Levels als der entsprechende B-Baum enthalten

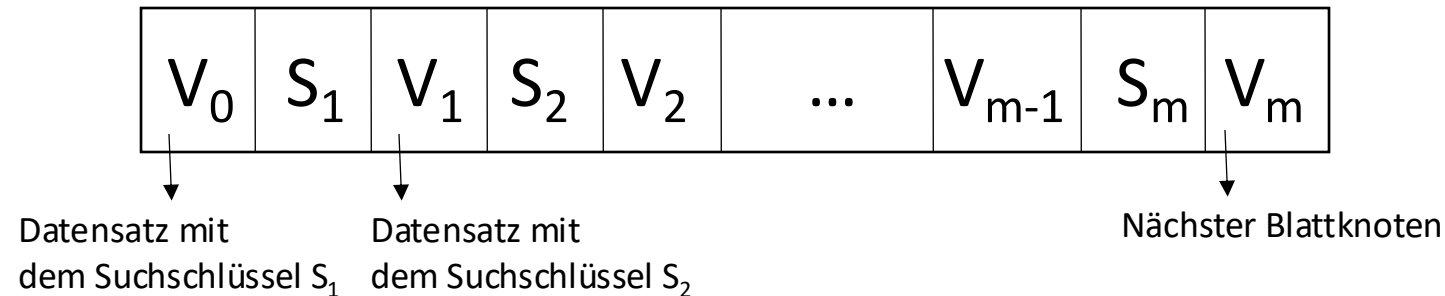


Die Struktur der Knoten in einem B⁺-Baum

- Innere Knoten:



- Blätterknoten:



B⁺-Bäume in der Praxis

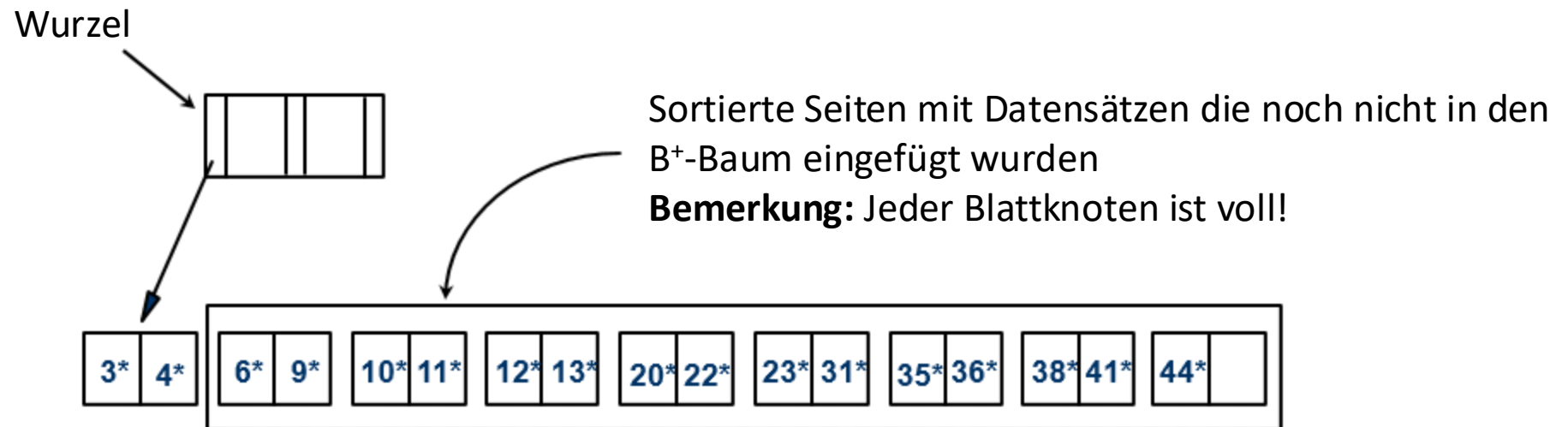
- Typisches Verzweigungsgrad/Ordnung: 100, Typisches Füllgrad: 67%
 - Typisches Ausgangsgrad/Fan-out (Anzahl von Indexeinträge pro Indexseite) $F = 133$
 - $F = \lceil \text{Seitengröße} / (\text{Schlüsselgröße} + \text{Zeigergröße}) \rceil$ - in Bytes
 - Größe einer Seite auf der Platte: 4, 8, 16, oder 32 KB
 - Physische Größe eines Zeigers: 4 oder 8 Bytes
 - Schlüsselgröße variiert stark (int, string, float, usw)
- Typische Größe:
 - Höhe 4: $133^4 = 312,900,700$ Datensätze
 - Höhe 3: $133^3 = 2,352,637$ Datensätze
- Oft können diese in den Top-Levels des Buffer Pools sein:
 - Level 1: 1 Seite = 8 KBytes
 - Level 2: 133 Seiten = 1 MByte
 - Level 3: 17,689 Seiten = 133 MBytes

Vor- und Nachteile der B⁺-Bäume

- Der Index bleibt balanciert → gleichformige Suchzeit
- Selten mehr als 3-5 Levels → man kann den Datensatz in 2-3 I/O Operationen finden
- Am meisten benutzt für Indexe in DBMS wegen der Flexibilität (auch am meisten optimierte in DBMS)
- B⁺-Bäume können für folgende Indexe benutzt werden:
 - geclustert, dünner Index, wenn die Daten sortiert sind
 - nicht-geclustert, dichter Index, wenn die Daten nicht sortiert sind

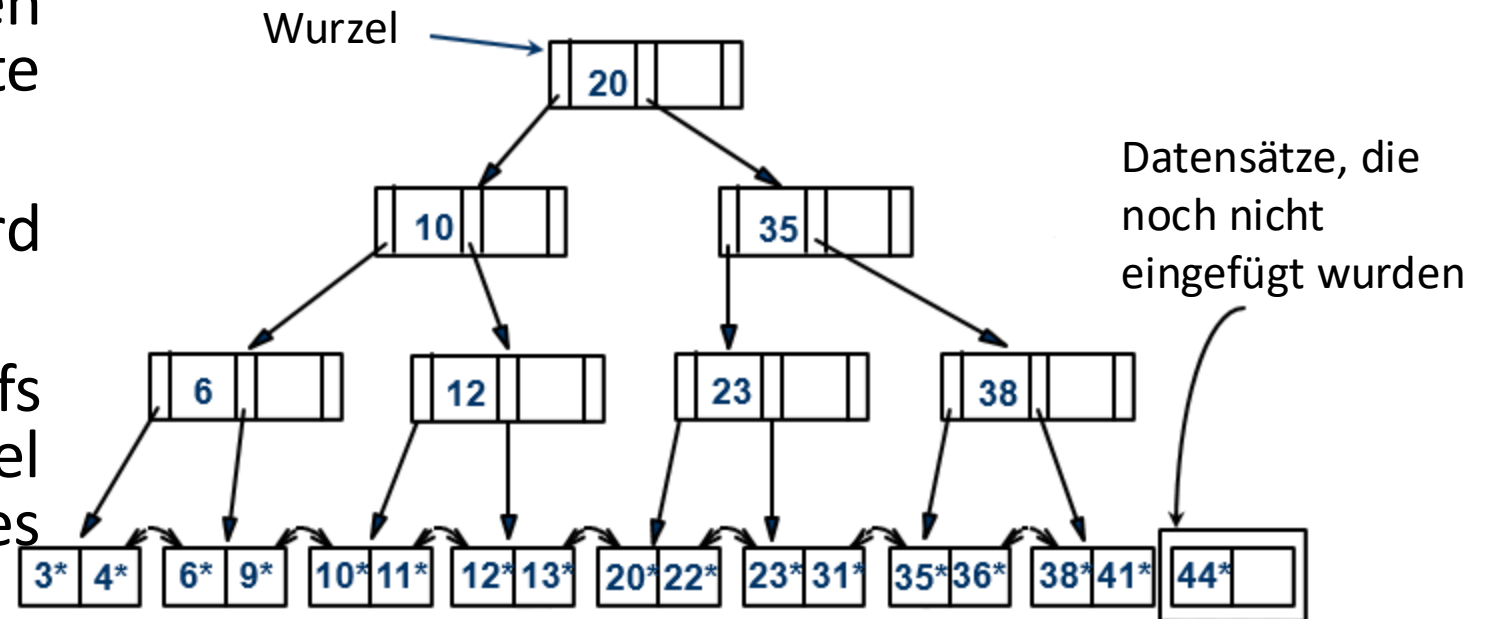
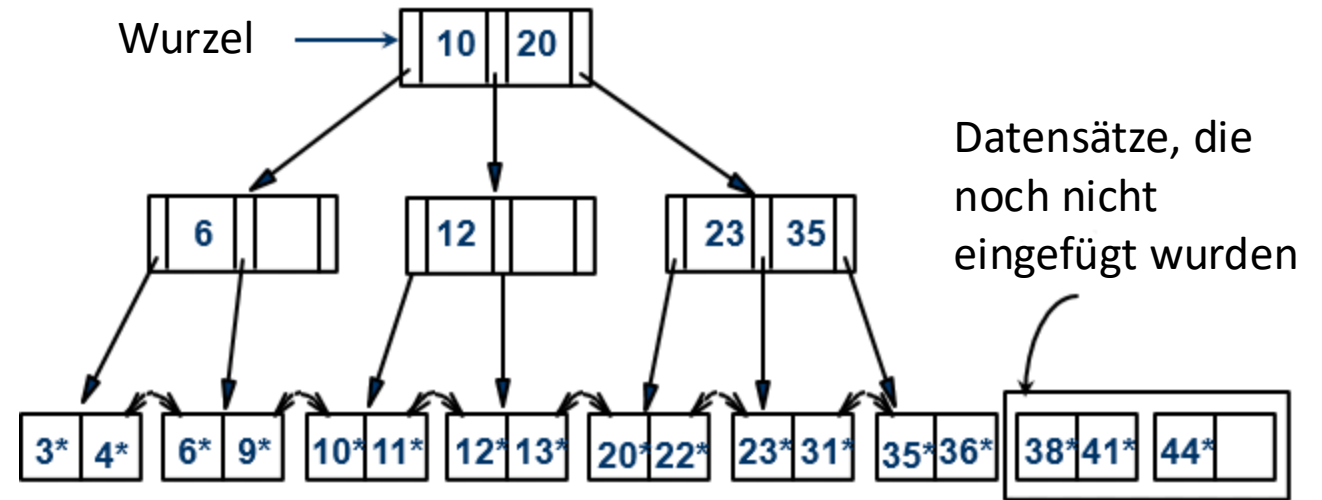
Bulk Einfügen in einem B⁺- Baum

- Wenn wir einen B⁺ - Baum auf einen Datenfeld für eine große Datenmenge erstellen wollen, dann ist es nicht effizient jeden Datensatz einzeln einzufügen → **Bulk Einfügen** ist viel effizienter
- Initialisierung:
 - Sortiere alle Datensätze
 - Erstelle eine neue Wurzelseite, die ein Zeiger zu der ersten Blattseite enthält



Bulk Einfügen

- Der kleinste Wert jedes Blattes (außer dem ersten) wird zum Separatorknoten
- Indexeinträge die auf Blätterseiten verweisen werden immer in die Indexseite eingefügt, die am weitesten rechts und der Blätterseite übergeordnet ist
- Wenn diese voll ist, dann wird sie geteilt
- Die Behandlung des Überlaufs kann sich bis zu der Wurzel fortpflanzen und die Höhe des Baumes um eins erhöhen



Bulk Einfügen vs. Einzelne Einfügeoperationen

- 1. Möglichkeit: mehrere einzelne Einfügeoperationen (ein einziger Datensatz)
 - Langsam
 - Die Blätter werden nicht sequentiell abgespeichert
- 2. Möglichkeit: Bulk Einfügen
 - Hat Vorteile für Mehrbenutzersynchronisation (wenigere Sperrungen)
 - Wenigere I/O Operationen um den B⁺- Baum zu erstellen
 - Blätter werden sequentiell abgespeichert
 - Man kann den Füllgrad beeinflussen

Präfix B⁺-Bäume

- Die Größe der Separatoreinträge \Rightarrow bestimmt die Anzahl der Indexeinträge in einer Seite \Rightarrow bestimmt die Höhe des B⁺-Baumes
- Um die Höhe des B⁺-Baumes zu minimieren und den Ausgangsgrad/fan-out zu maximieren müssen möglichst viele Separatoren auf eine Seite passen
- Die Suchschlüssel haben nur die Funktion eines Separators, die eine Suche zu den Blattknoten führt \rightarrow können meistens komprimiert werden (ein Präfix ist ausreichend)
- Einfügen und Löschen müssen entsprechend geändert werden

Präfix B⁺-Bäume - Beispiel

- z.B. Benachbarte Indexeinträge mit den Suchschlüsseln:
Dan Yogurt, David Smith, Demi Moore, Davey Jones
- Wir kürzen diese ab: *David Smith* Dav., *Davey Jones* → *Dave.*, ...
- Wie vergleichen wir jetzt *David Smith* mit *Davey Jones*?
- Ist jetzt "*Dav.*" < „*Dave.*“?
- Wenn man eine Schlüsselkompression auswählt, dann muss die Ordnung zwischen den Suchschlüsseln aufbewahrt werden

Verzweigungsgrad/Ordnung des B⁺-Baumes in der Praxis

- In der Praxis ersetzt man das Konzept von Verzweigungsgrad/Ordnung mit **Füllgrad** (wenigstens halb voll)
- Die Indexseiten können viel mehrere Einträge enthalten als die Blätterseiten
- Wenn die Länge der Suchschlüssel und der Datensätze variable ist, dann können Seiten unterschiedliche Anzahl von Einträgen enthalten
- Auch wenn die Länge nicht variable ist, kann es sein dass zwei Datensätze denselben Suchschlüsselwert haben → mit Alternative (3) wird die Länge der Indexeinträge variable

Zusammenfassung

- Baum-strukturierte Indexe sind sehr effizient für Bereichsanfragen, aber auch gut für Gleichheitsanfragen
- ISAM (Index-Sequential Access Method) ist eine statische Struktur
 - Nur die Blätterseiten ändern sich, man braucht Überlaufsseiten
 - Überlaufsseiten können die Effizienz erniedrigen, außer wenn sich die Daten nicht oft ändern
- B⁺-Baum ist eine dynamische Struktur
 - Nach einem Einfügen oder Löschen bleibt der Baum balanciert
 - Großer Ausgangsgrad/fan-out → oft ist die Tiefe nicht mehr als 3 oder 4
 - Fast immer besser als die Datei sortiert zu behalten
 - Typischerweise Füllgrad 67%
 - Meistens besser als ISAM

Zusammenfassung

- Präfix B⁺-Baum
 - Schlüsselkompression vergrößert den Ausgangsgrad/fan-out und verkleinert die Höhe
- Bulk-Einfügen kann viel schneller sein als wiederholte einzelne Einfüge-Operationen bei dem Erstellen eines B⁺-Baumes
- B⁺-Bäume sind die am meisten benutzte Indexe in DBMS wegen der Flexibilität und der hohen Optimierung, Default für die meisten Systeme