Teil 1:
Suchen

- Problemstellung
- Elementare Suchverfahren
- Hashverfahren
- Binäre Suchbäume
- Ausgeglichene Bäume
- B-Bäume
- Digitale Suchbäume

Motivation

Bisher wurde angenommen, daß die gesamte Datenstruktur des Baumes im Hauptspeicher Platz findet.

Was passiert, wenn ein Teil oder die gesamte Datenstruktur auf der Festplatte gespeichert werden muß?

⇒ Bisherige O(f(n))-Analyse stimmt nicht mehr, da Plattenzugriffe viel länger dauern als Speicherzugriffe (ein Plattenzugriff entspricht der Dauer von mehreren 100 000 Speicherzugriffen).

⇒ Beispiel: Bei einem binären Suchbaum mit 10 Mio Einträgen würde eine durchschnittliche Suche 32 Plattenzugriffe (ca. 5s), oft aber 100 (ca. 16s) brauchen.

⇒ Beispiel: Bei einem AVL-Baum kommt man durchschnittlich nahe an log N, also typischerweise 25 Zugriffe (4s), maximal 35 Zugriffe (6s).

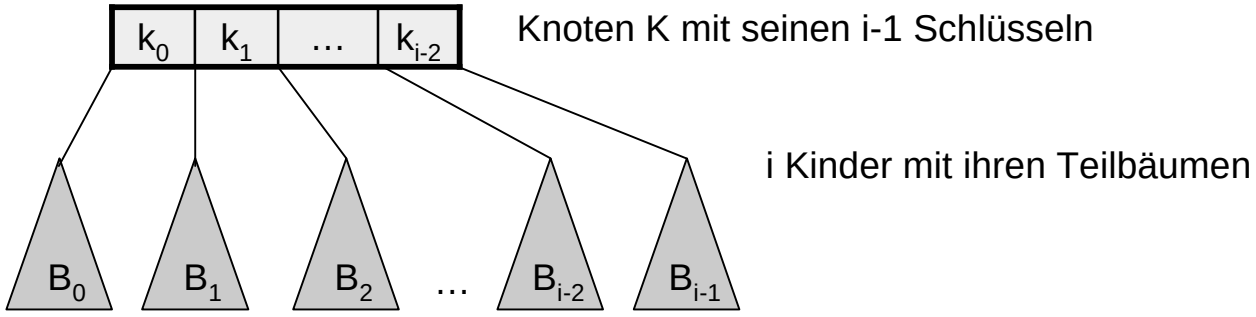
⇒ Ansatz: Minimiere Anzahl Zugriffe durch Bäume mit Knoten mit mehr als 2 Kindern.

Definition von B-Bäumen (1)

B-Baum der Ordnung m

ist ein Baum mit folgenden Eigenschaften:

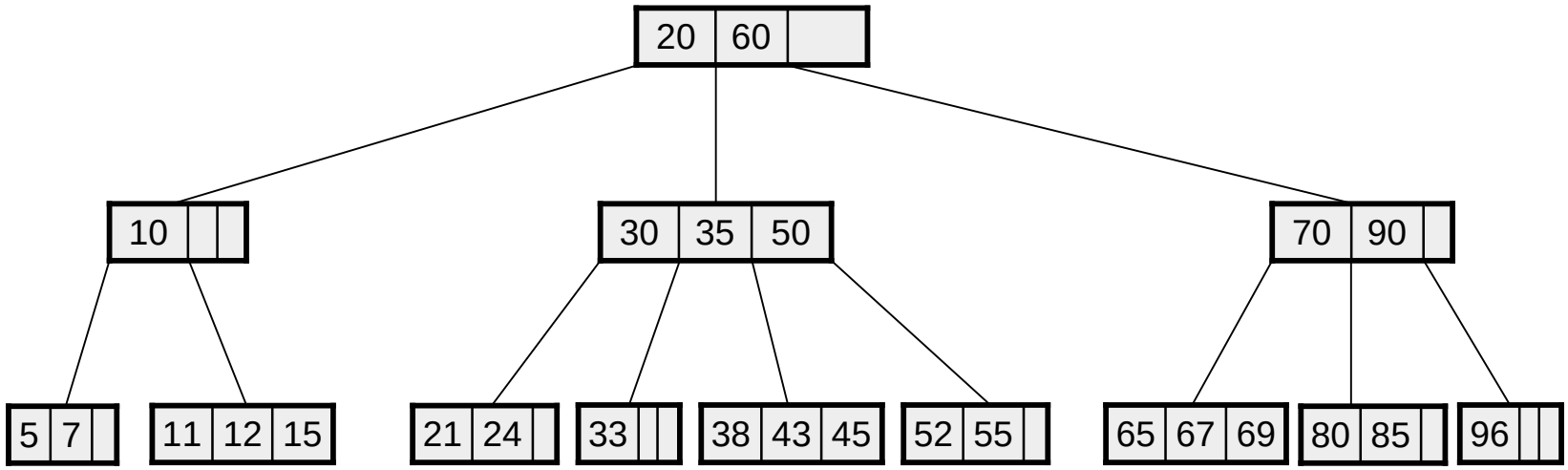
1. Jedes Blatt hat die gleiche Tiefe (d.h. der Baum ist perfekt ausgeglichen)
2. Anzahl Kinder:
   Die Wurzel hat zwischen 2 und m Kinder.
   Die anderen Knoten (mit Ausnahme der Blätter) haben zw. $\lceil m/2 \rceil$ und m viele Kinder.
3. Anzahl Schlüssel:
   Jeder Knoten mit i Kindern hat i-1 Schlüssel.
   Jedes Blatt hat zwischen $\lceil m/2 \rceil - 1$ und $m-1$ Schlüssel.
4. Schlüsselordnung: (ähnlich wie bei binären Suchbäumen)
   In jedem Knoten sind die Schlüssel sortiert und außerdem gilt für jeden Knoten K mit i Kindern und i–1 Schlüsseln $k_0, k_1, \ldots, k_{i-2}$ folgende Beziehung:



Schlüssel in $B_0 < k_0 <$ Schlüssel in $B_1 < k_1 <$ Schlüssel in $B_2 < \ldots$
Schlüssel in $B_{i-2} < k_{i-2} <$ Schlüssel in B_{i-1} .

Definition von B-Bäumen (2)

Beispiel für ein B-Baum der Ordnung m = 4:



Vergleiche mit Definition:

1. Der Baum ist perfekt ausgeglichen: Jedes Blatt hat die gleiche Tiefe; nämlich 2.
2. Anzahl Kinder: Die Wurzel hat 3 Kinder. Jeder andere Knoten (mit Ausnahme der Blätter) hat zwischen $\lceil m/2 \rceil = 2$ und $m = 4$ Kinder.
3. Anzahl Schlüssel: Jeder Knoten mit i Kindern hat i-1 Schlüssel und die Blätter haben zwischen $\lceil m/2 \rceil - 1 = 1$ und $m - 1 = 3$ Schlüssel.
4. Die Schlüsselordnung ist für jeden Knoten erfüllt.

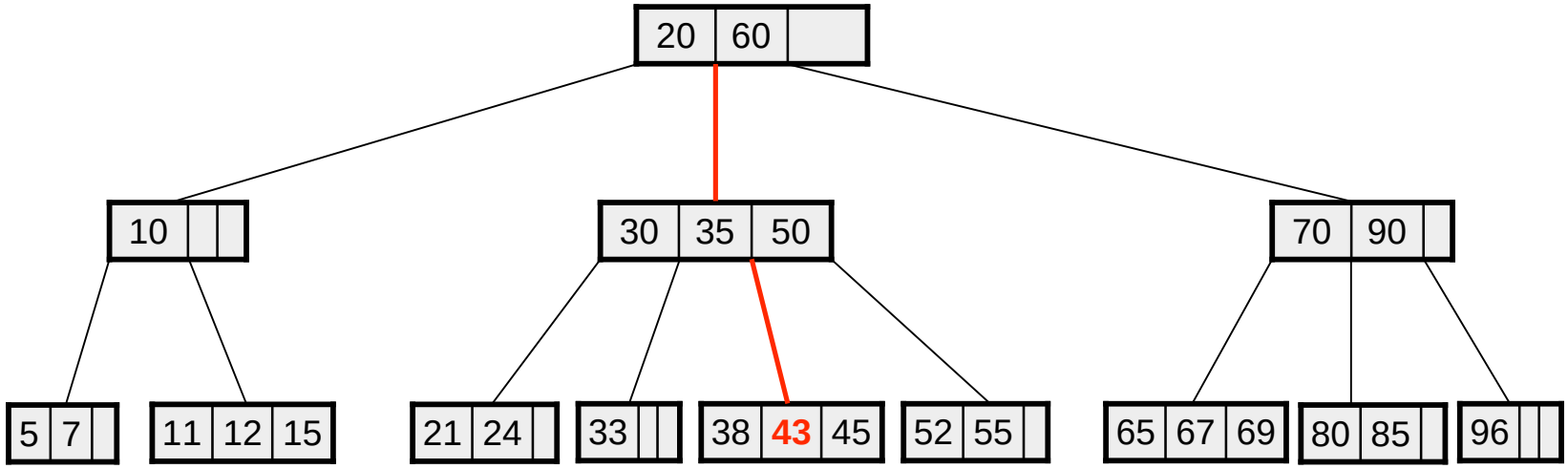
Suchen in B-Bäumen

Algorithmus:

```
bool search(Key k, Node* p)
{
      if (p == 0) return false; // nicht gefunden;

      suche k in Schlüsselmenge von Knoten p (beispielsweise mit binärer Suche);
      if (k gefunden) return true;

      return search(k, Teilbaum, in dem k liegen könnte);
}
```

Rekursiver Aufruf

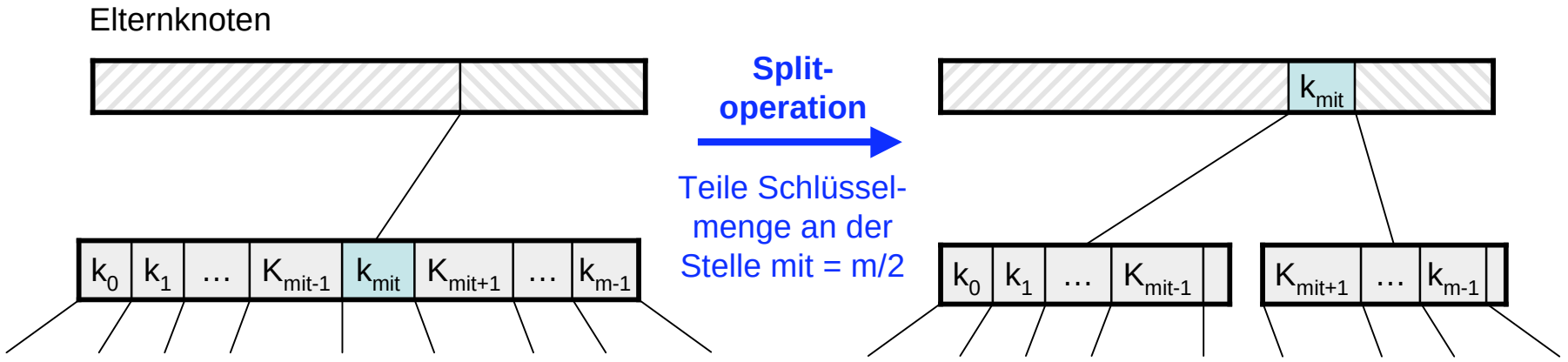
Beispiel: Suchen nach k = 43:



Einfügen in B-Bäumen (1)

Algorithmus:

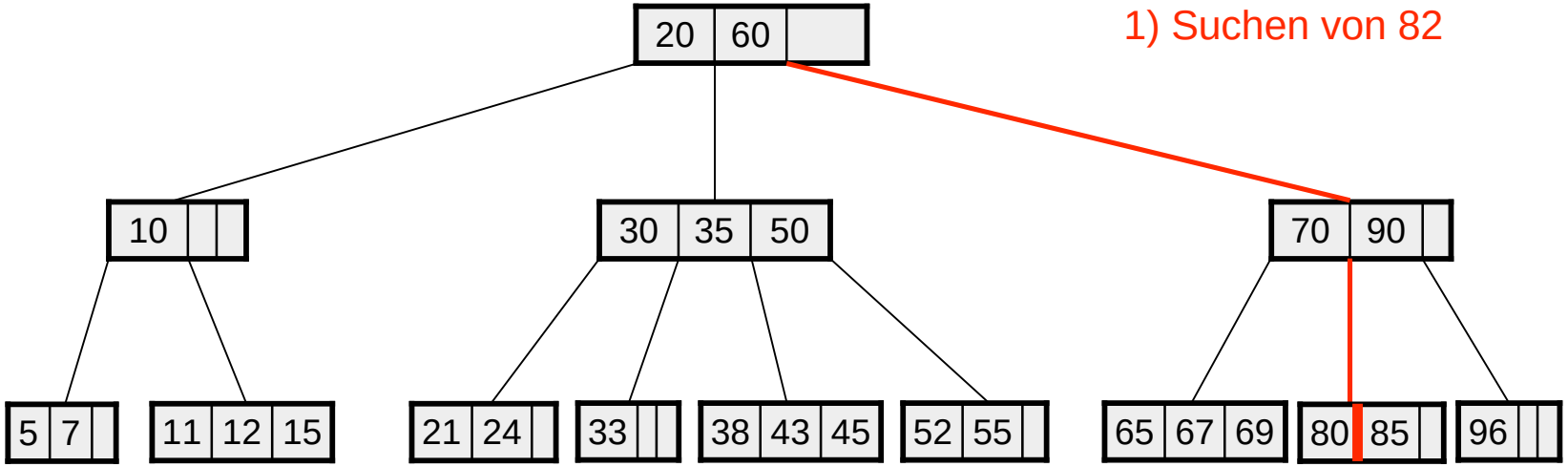
```
suche Schlüssel k im Baum;

if (k gefunden)
      k nicht einfügen;

füge k an die Blattstelle ein, wo die Suche beendet wurde; // wie bei binären Suchbäumen

if (Schlüsselüberlauf)    // d.h. Anzahl Schlüssel == m
      führe vom aktuellen Knoten bis zur Wurzel jeweils eine Splitoperation durch,
      falls die jeweilige Schlüsselmenge zu groß ist;
```



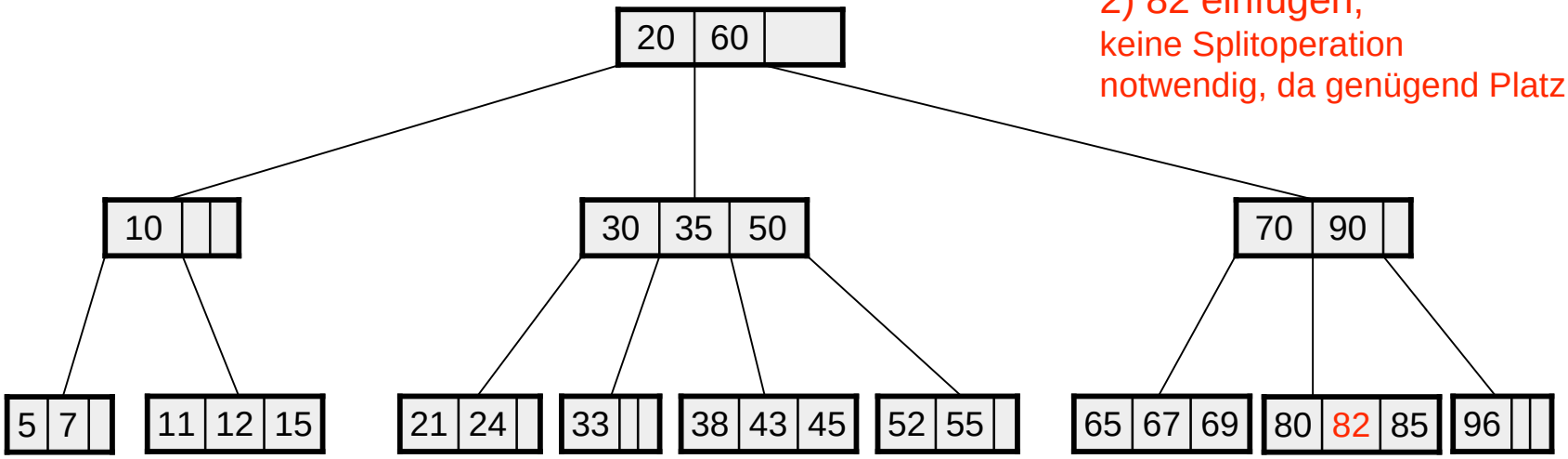
Aktueller Knoten;
Schlüsselmenge ist zu groß

Einfügen in B-Bäumen (2)

Beispiel 1: Einfügen von 82:



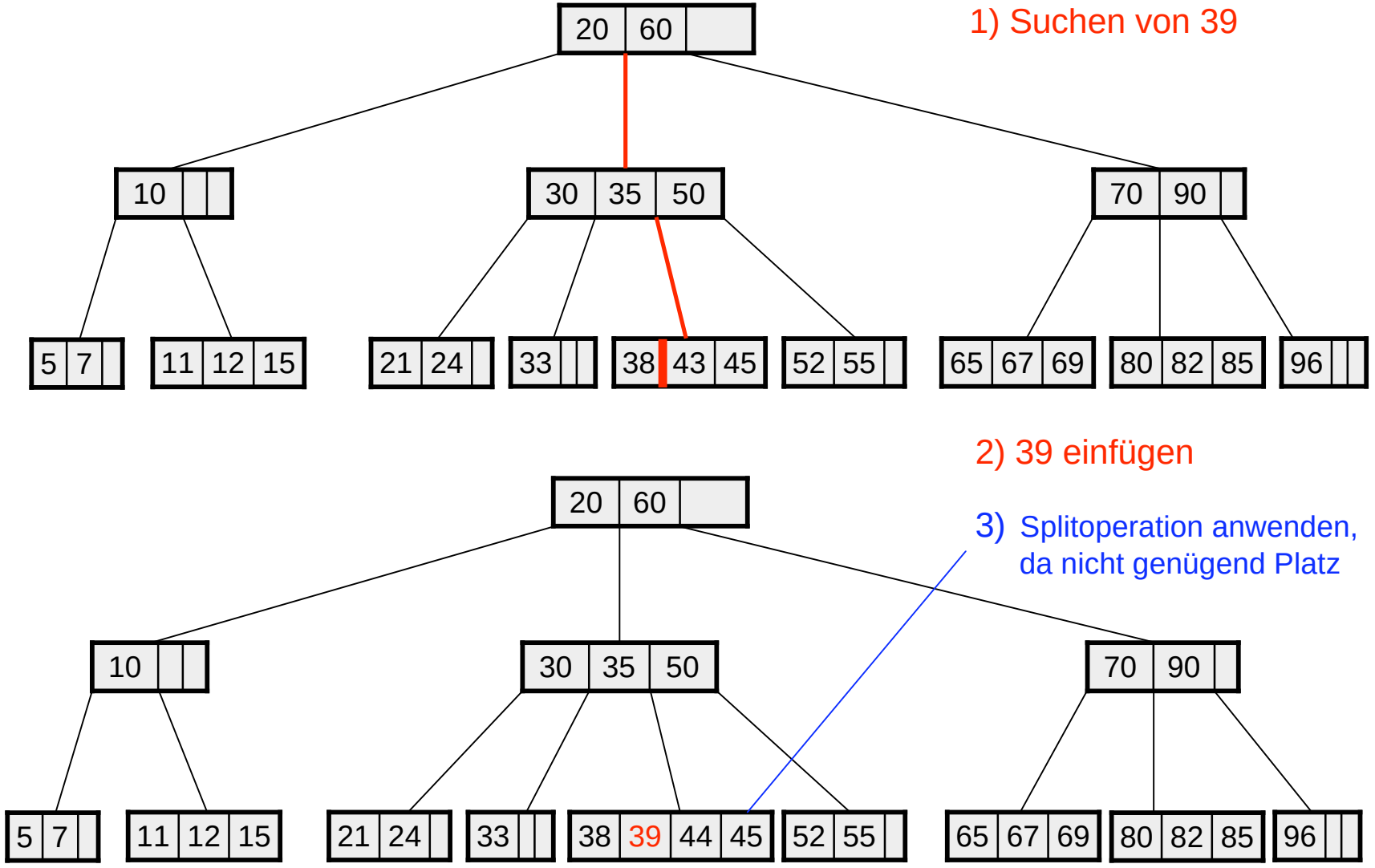
1) Suchen von 82



2) 82 einfügen;
keine Splitoperation
notwendig, da genügend Platz

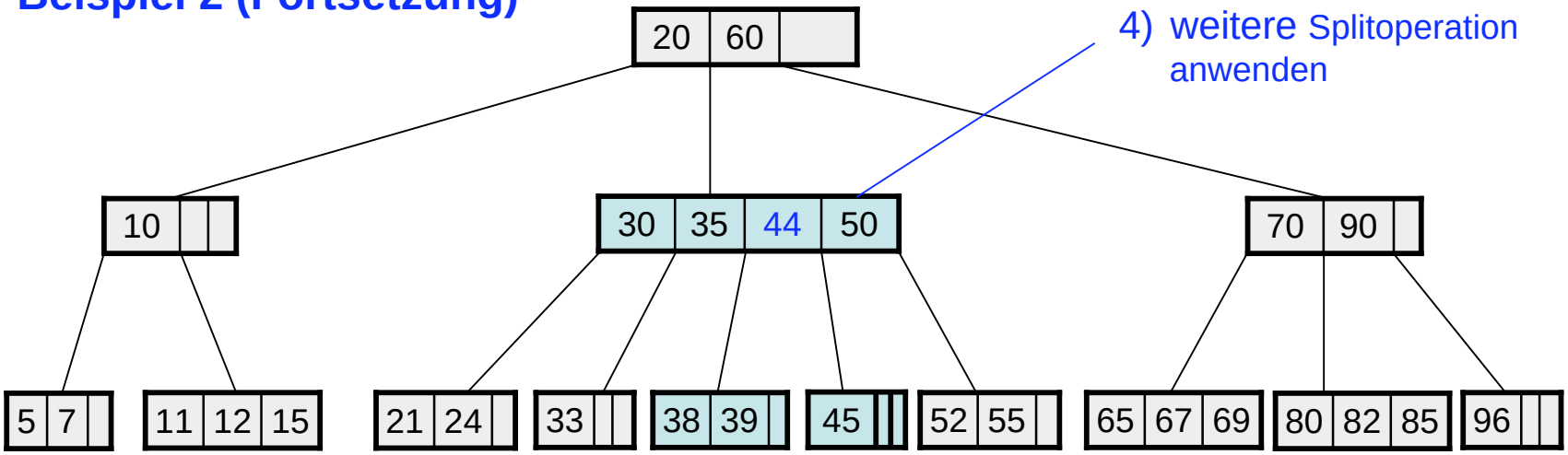
Einfügen in B-Bäumen (3)

Beispiel 2: Einfügen von 39:

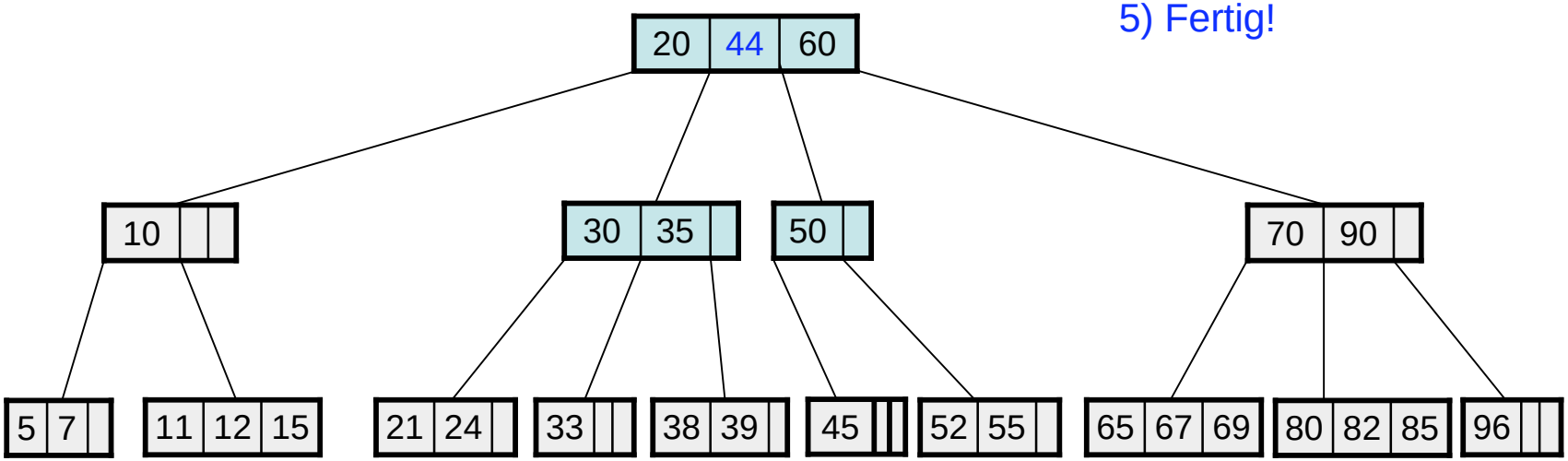


Einfügen in B-Bäumen (4)

Beispiel 2 (Fortsetzung)



4) weitere Splitoperation anwenden

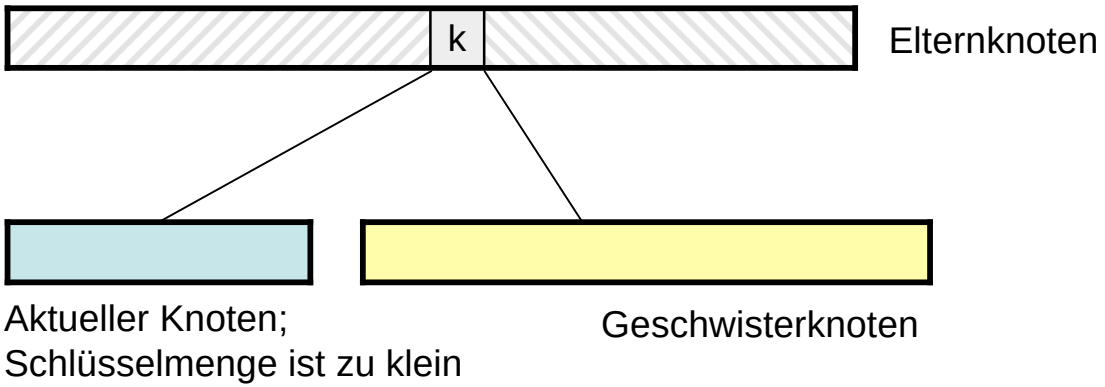


5) Fertig!

Löschen in B-Bäumen (1)

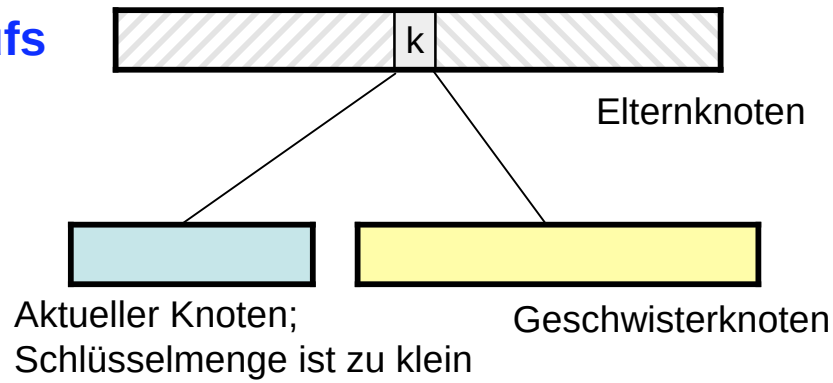
Algorithmus:

```
suche Schlüssel k im Baum;

if (nicht gefunden) nichts zu löschen;

if (k befindet sich im Blatt)
      lösche k;
else
      ersetze k durch nächst kleineren Schlüssel k' und lösche k';
      // k' befindet sich im nach k hängenden Teilbaum ganz links in einem Blatt

if (Schlüsselunterlauf)   // d.h. für Anzahl Schlüssel anz gilt: anz < ⌈m/2⌉ -1 (anz == 0 bei Wurzel)
      führe vom aktuellen Knoten bis zur Wurzel, falls die jeweilige Schlüsselmenge zu klein ist,
entweder Verschmelzung mit einem Geschwisterknoten oder
      Übernahme von Schlüsseln von einem Geschwisterknoten durch;
```



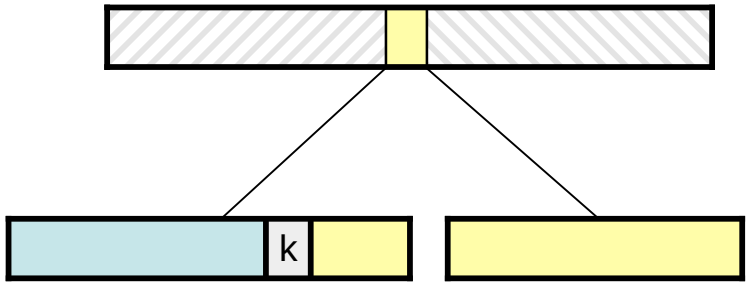
Löschen in B-Bäumen (2)

Behandlung eines Schlüsselunterlaufs



Übernahme von Schlüssel:

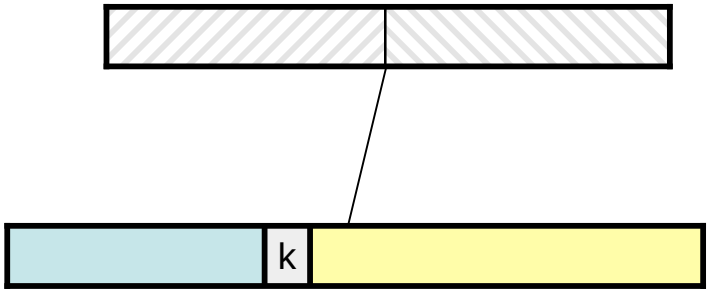
Linker bzw. rechter Geschwisterknoten kann genügend Schlüssel abgeben



Fertig! Keine weitere Unterlaufbehandlung notwendig.

Verschmelzung:

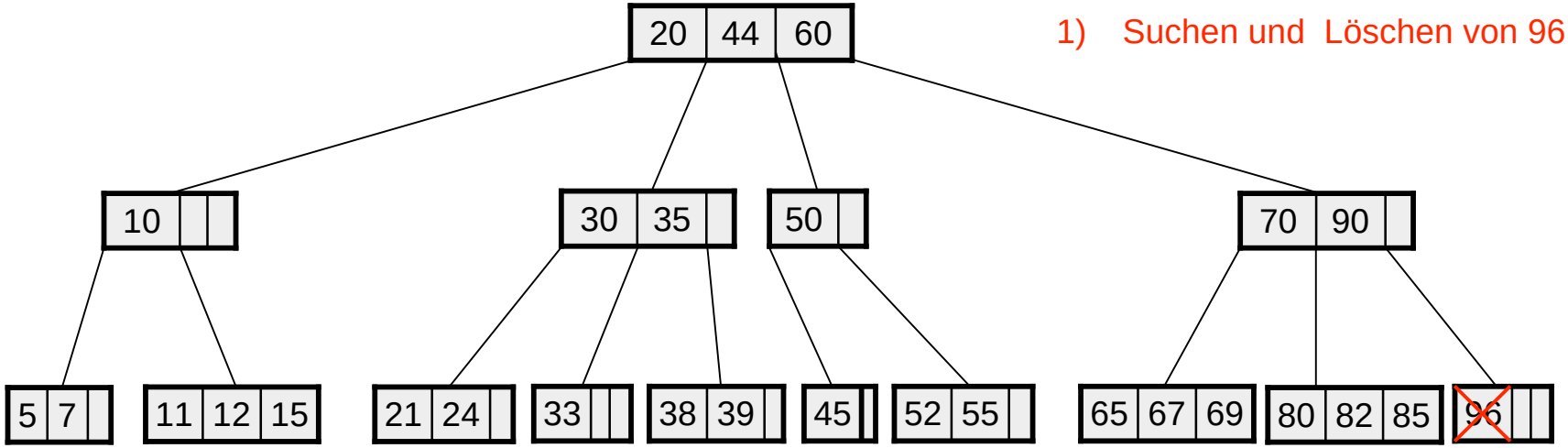
Beide Geschwisterknoten haben zuwenig Schlüssel. Fasse aktuellen Knoten mit linkem bzw. rechten Geschwisterknoten zusammen



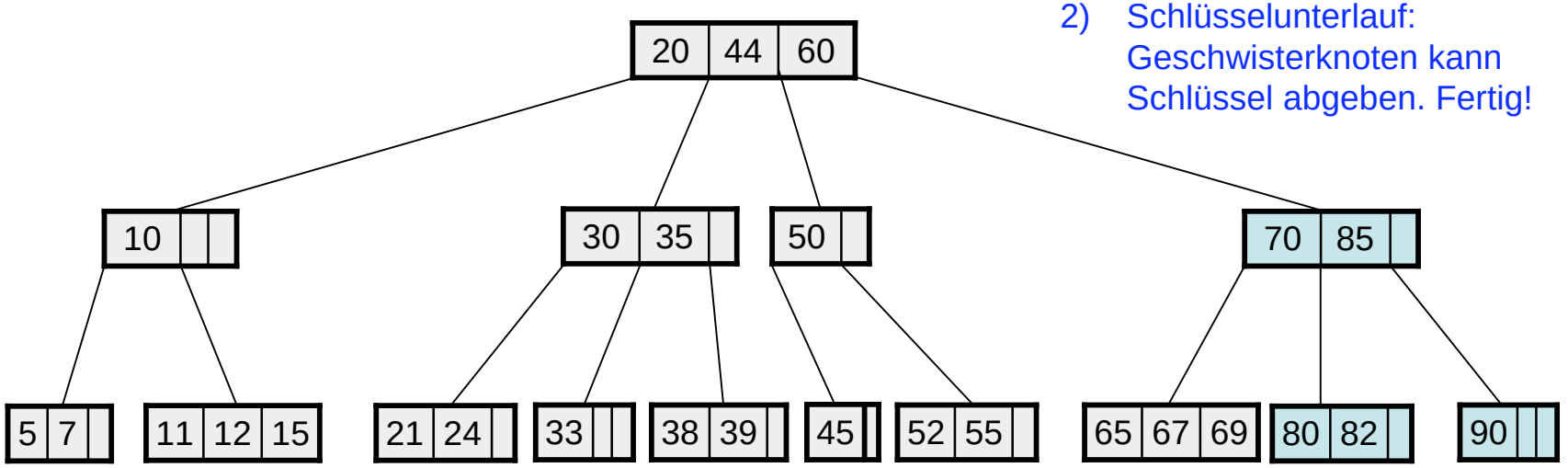
Elternknoten ist um ein Schlüssel kleiner geworden. Wende daher Unterlaufbehandlung auf Elternknoten an und mache entsprechend weiter.

Löschen in B-Bäumen (3)

Beispiel 1: Löschen von 96



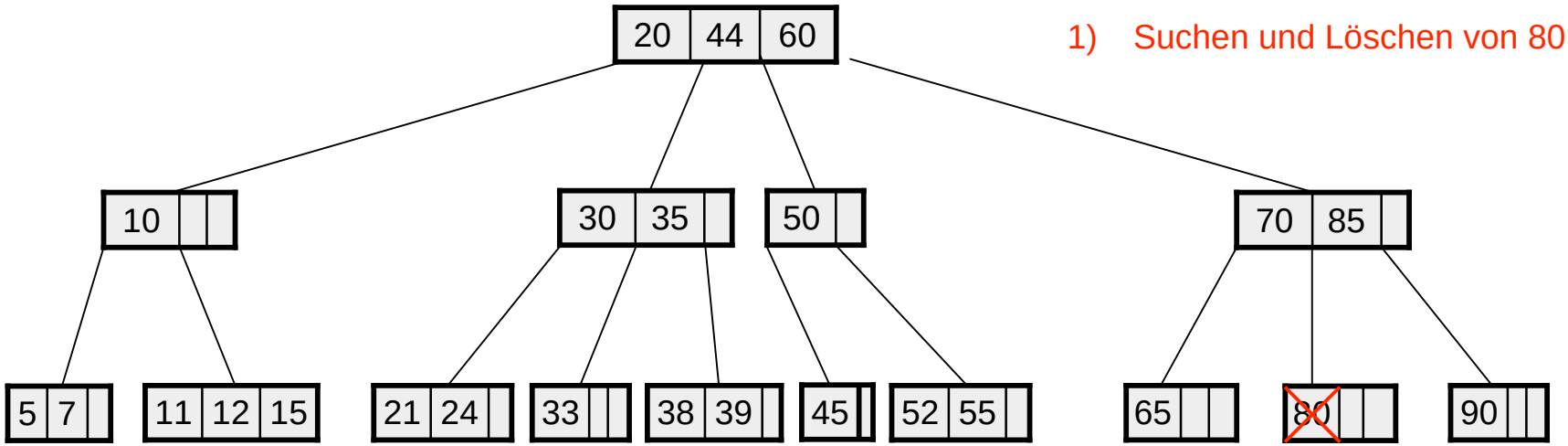
1) Suchen und Löschen von 96



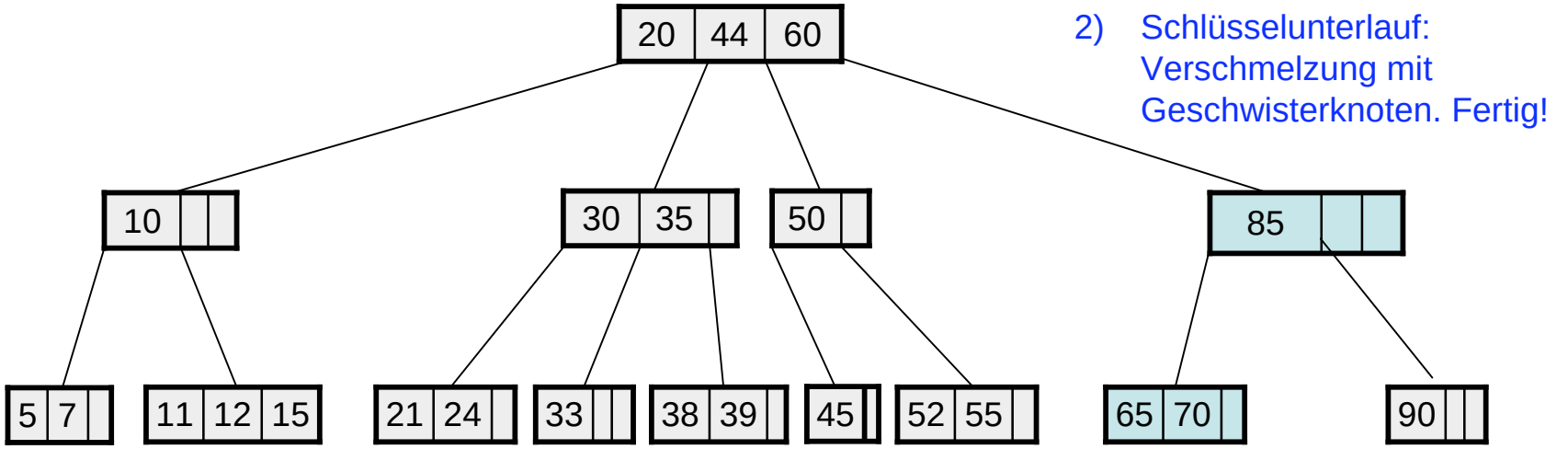
2) Schlüsselunterlauf: Geschwisterknoten kann Schlüssel abgeben. Fertig!

Löschen in B-Bäumen (4)

Beispiel 2: Löschen von 80



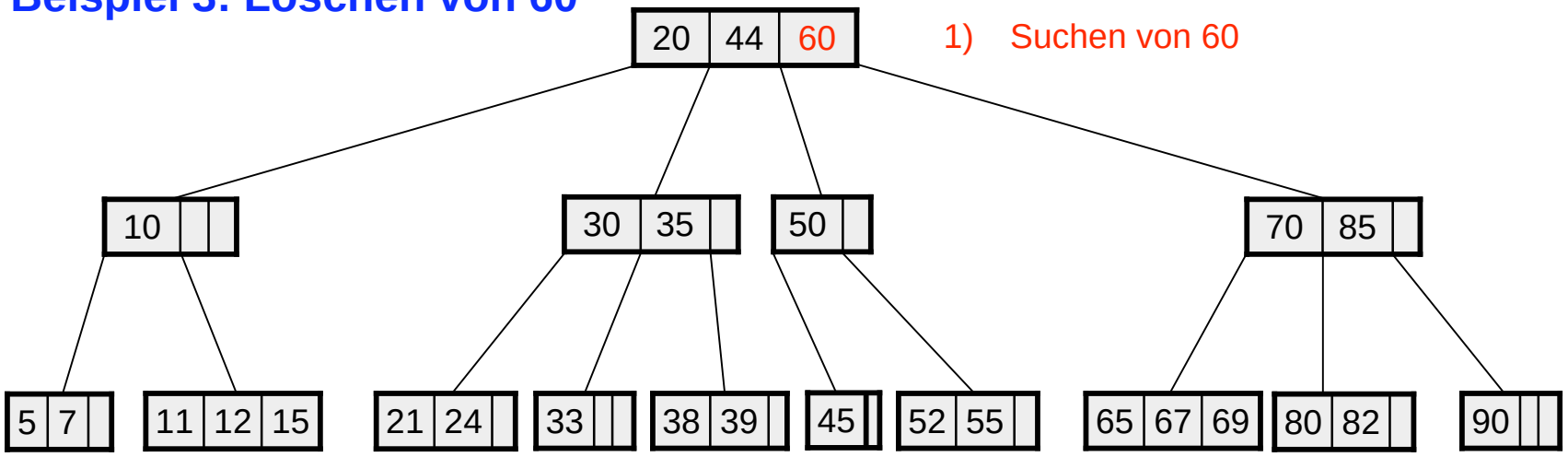
1) Suchen und Löschen von 80



2) Schlüsselunterlauf: Verschmelzung mit Geschwisterknoten. Fertig!

Löschen in B-Bäumen (5)

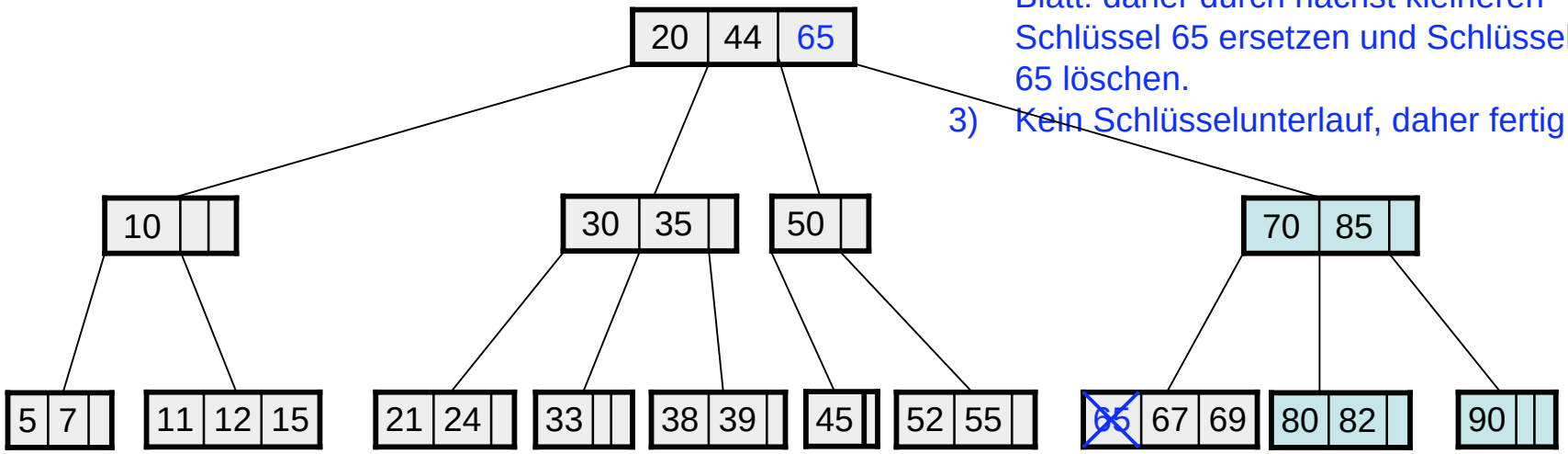
Beispiel 3: Löschen von 60



1) Suchen von 60

2) Schüssel 60 befindet sich nicht in Blatt: daher durch nächst kleineren Schlüssel 65 ersetzen und Schlüssel 65 löschen.

3) Kein Schlüsselunterlauf, daher fertig!



Analyse (1)

Höhe eines B-Baumes mit n Schlüsseln:

Im schlechtesten Fall hat die Wurzel 2 Kinder und jeder andere Nicht-Blatt-Knoten $\lceil m/2 \rceil$ Kinder. Damit ergibt sich eine maximale Höhe von (siehe [Ottmann u. Widmayer]):

$$h \leq 1 + \log_{\lceil m/2 \rceil}(\frac{n+1}{2})$$

Beispiel:
Für m = 256 und n = 10^9 ergibt sich eine maximale Höhe von h ≤ 4.1

Aufwand für Suchen, Einfügen und Löschen:

Da die Höhe eines B-Baums immer logarithmisch beschränkt ist, ergibt sich eine maximale Laufzeit von:

$$T(n) = O(\log_{\lceil m/2 \rceil}(n))$$

Analyse (2)

Speicherplatzausnutzung:

- Eine naheliegende Implementierung eines B-Baums sieht für jeden Knoten ein statisches Feld der Größe m für die Schlüssel und Zeiger auf die Kinder vor.

  Da jeder Knoten (außer Wurzel) zwischen $\lceil m/2 \rceil - 1$ und m-1 viele Schlüssel enthält, stellt sich die Frage, wie groß die Speicherplatzausnutzung ist.

  Es gilt folgende Eigenschaft (siehe [Ottmann u. Widmayer]):

  Wenn eine zufällig gewählte Folge von n Schlüsseln in einen anfangs leeren B-Baum der Ordnung m eingefügt werden, dann ist eine Speicherplatzausnutzung zu erwarten von:

  $\ln(2) \approx 69\ \%$.

- Falls eine absteigend oder aufsteigend sortierte Folge in einen leeren B-Baum einfügt wird, ergibt sich eine besonders schlechte Speicherplatzausnutzung, die aber immer noch bei ca. 50 % liegt.
  Etwas lindern lässt sich das Problem, indem nicht bei jedem Knotenüberlauf eine Splitoperation durchgeführt wird, sondern zuvor geprüft wird, ob Schlüssel an Geschwisterknoten abgegeben werden können.

Anwendungen

Externe Suchverfahren

Eine der wichtigsten Anwendungen von B-Bäume sind externe Suchverfahren (z.B. im Datenbankbereich). Die Menge der Datensätze ist dabei so groß, dass sie nur auf Hintergrundspeicher wie beispielsweise Festplatte abgespeichert werden kann.

Die Knotengröße m wird dann so gewählt, dass mit einem Festplattenzugriff die gesamten Daten eines Knotens (d.h. Schlüssel und Zeiger; Indexseite) gelesen werden können.

Die Datensätze selbst sind auf der Platte gespeichert.

Beispiel:

Wählt man beispielsweise m = 256 und sind n = 10^9 Datensätze zu verwalten, dann ergibt sich ein B-Baum etwa der Höhe 4. Speichert man die beiden obersten Ebenen des B-Baums (d.h. Wurzel und seine Kinder) im Hauptspeicher (das sind 256 + 256*256 ≈ 65000 Schlüssel), dann kommt man im schlechtesten Fall bei einer Suchoperation mit 3 Plattenzugriffe aus:

2 Knotentabellen lesen und den Datensatz selbst.

Varianten

B*-Baum

Zusätzliche Forderung: jeder Knoten (außer der Wurzel) muss mindestens zu 2/3 gefüllt sein.

Vorteil: bessere Speicherplatzausnutzung und geringere Höhe des Baumes.

Nachteil: bei den Operationen Einfügen und Löschen kommt ein Überschreiten der Maximal- bzw. Minimalzahl der Schlüssel je Knoten im Vergleich zum B-Baum häufiger vor.

B+-Baum

- Die Datensätze werden nur in den Blättern abgespeichert. In den Nicht-Blatt-Knoten stehen nur Schlüssel und Zeiger.
- Um die Bereichsuche zu unterstützen (finde alle Datensätze mit Schlüssel $k \in [k1, k2]$), werden die Datensätze in den Blättern miteinander verkettet.

Spezialfälle (1)

2-3- und 2-3-4 Bäume

Wichtige Sonderfälle für B-Bäume der Ordnung m sind:

- m = 3: 2-3-Bäume
- m = 4: 2-3-4-Bäume

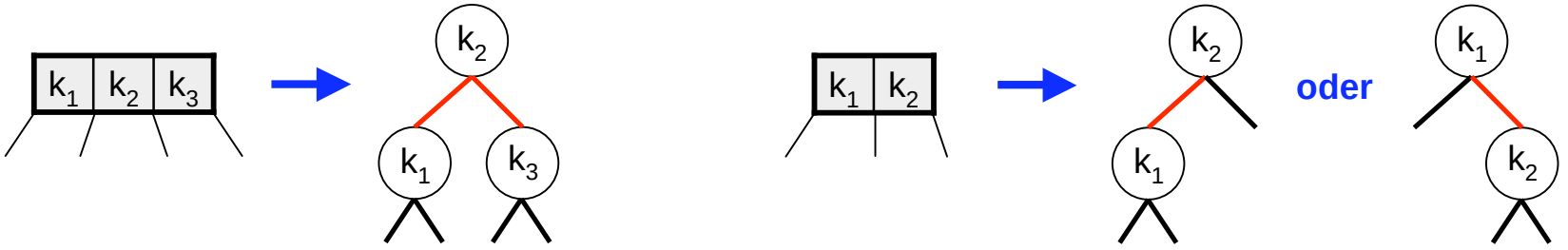
2-3- bzw. 2-3-4-Bäume sind sehr gut geeignete Alternativen für AVL-Bäume.

Da die Implementierung der B-Bäume für m = 3 bzw. 4 etwas überdimensioniert ist, werden sogenannte Rot-Schwarz-Bäume eingesetzt.

Rot-Schwarz-Bäume

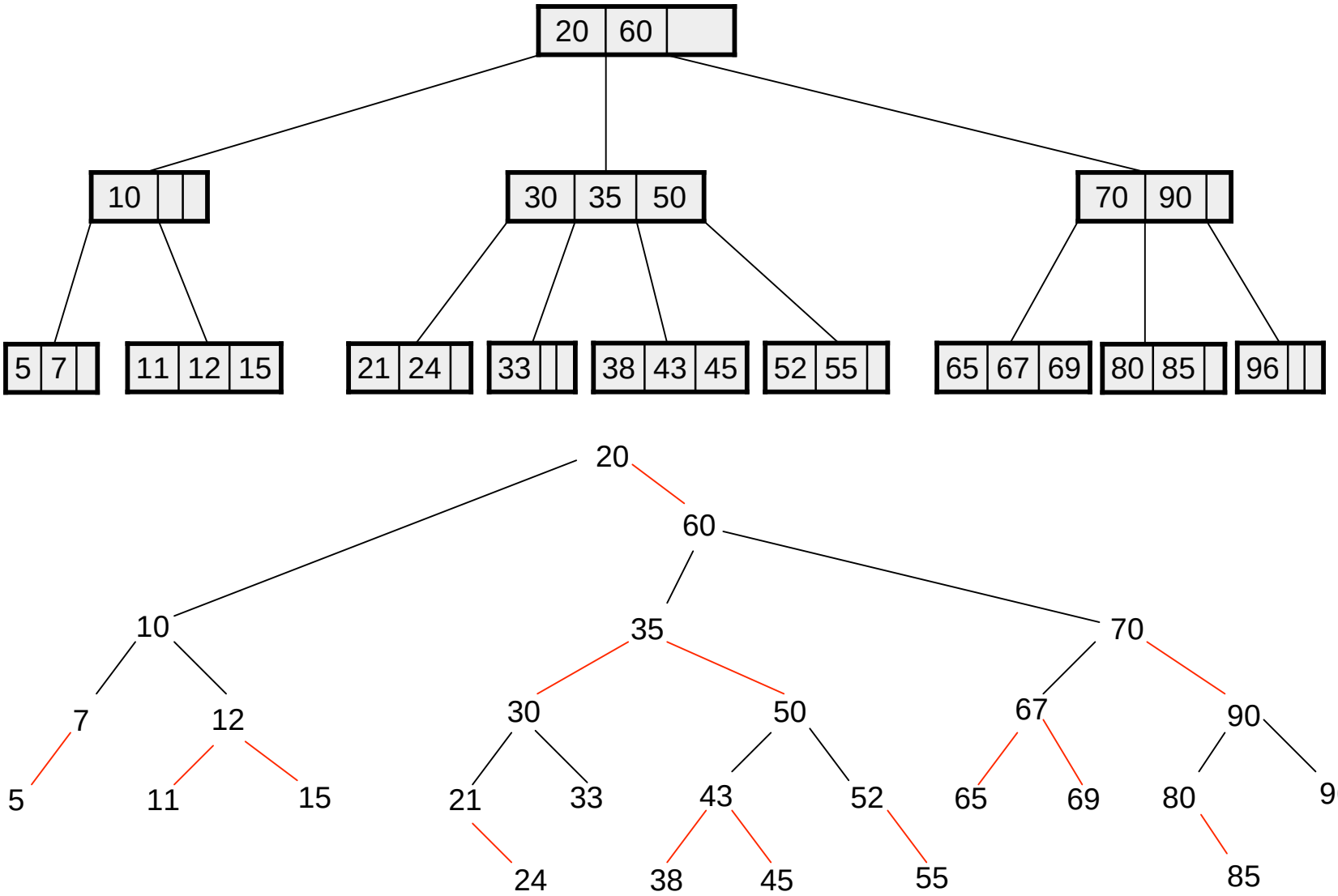
Rot-Schwarz-Bäume sind Binärbäume, dessen Kanten entweder rot oder schwarz sind (für die Färbungen gelten noch weitere Regeln, die hier weggelassen wurden).

2-3- bzw. 2-3-4-Bäume lassen sich dadurch realisieren, in dem ein Knoten mit 3 bzw. 4 Schlüssen durch einen kleinen Binärbaum mit roten Kanten dargestellt wird und die restlichen Verbinden durch schwarze Kanten:

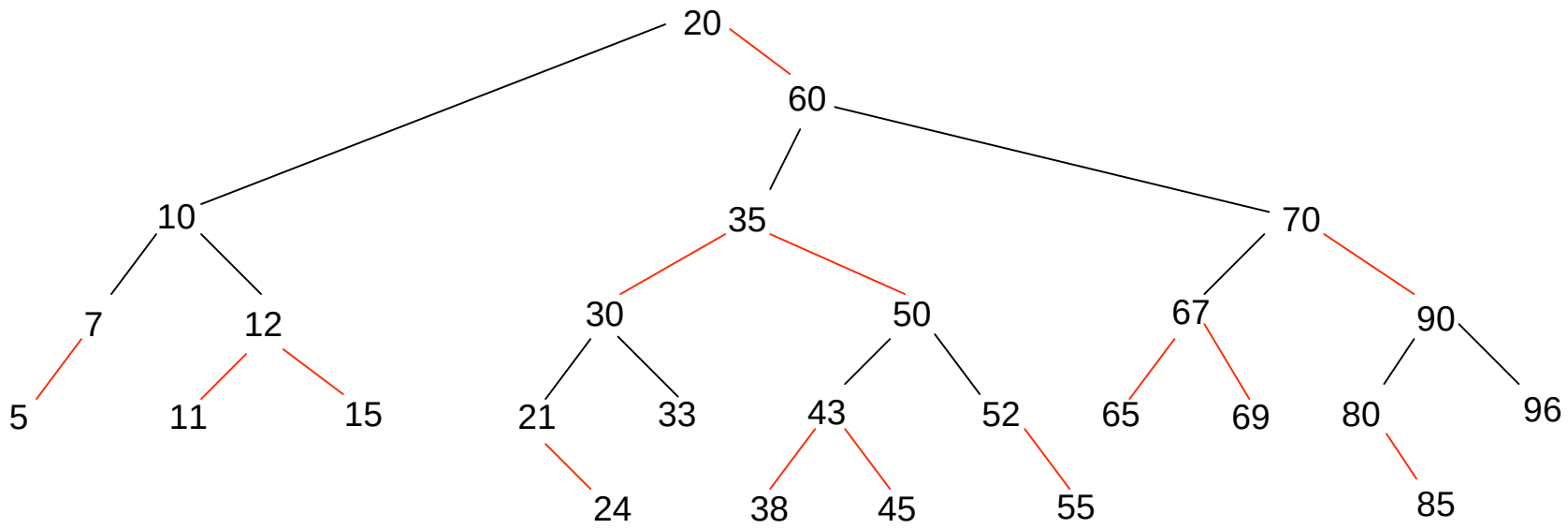


Beachte, dass sich die Kantenfarbe bei demjenigen Knoten, zu dem die Kante zeigt, als zusätzliches Bit abspeichern lässt.

Repräsentation eines 2-3-4-Baums als Rot-Schwarz-Baum



Balancierbedingungen für Rot-Schwarz-Bäume

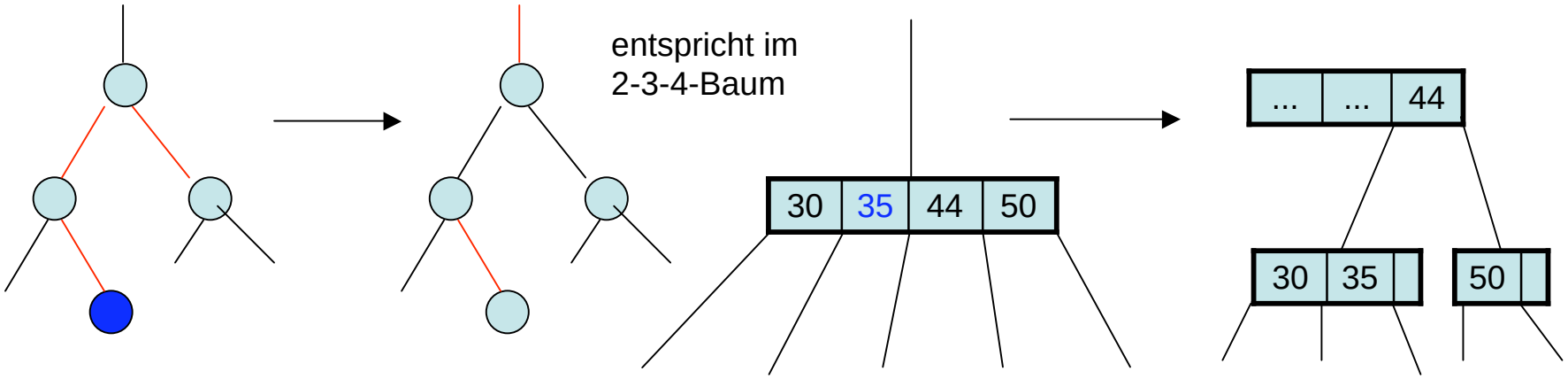


• Jeder Rot-Schwarz-Baum muß in allen Pfaden von der Wurzel zu einem Blatt dieselbe Anzahl von schwarzen Kanten haben (entspricht der Eigenschaft von B-Bäumen, daß alle Blätter denselben Abstand von der Wurzel haben müssen).

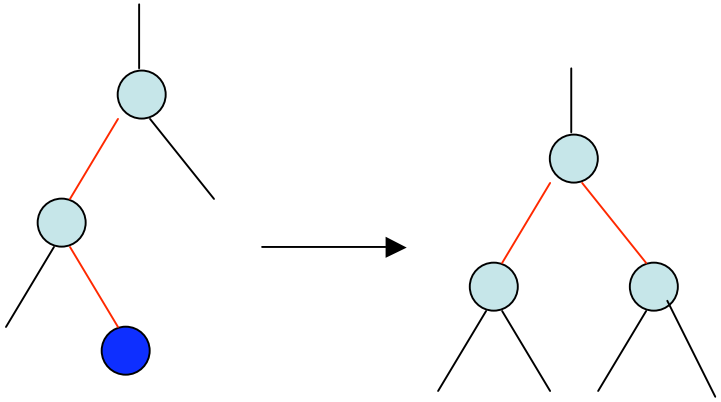
• Zur Repräsentation von 2-3-4-Bäumen wird zusätzlich verlangt, daß kein Pfad von einem inneren Knoten zu einem Blatt zwei aufeinanderfolgende rote Kanten haben darf.

Splitoperation: Farbwechsel und Rotation

Bei Einfügeoperationen wird neues Blatt mit roter Kante angehängt
=> kann es passieren, daß 2 rote Kanten aufeinanderfolgen.



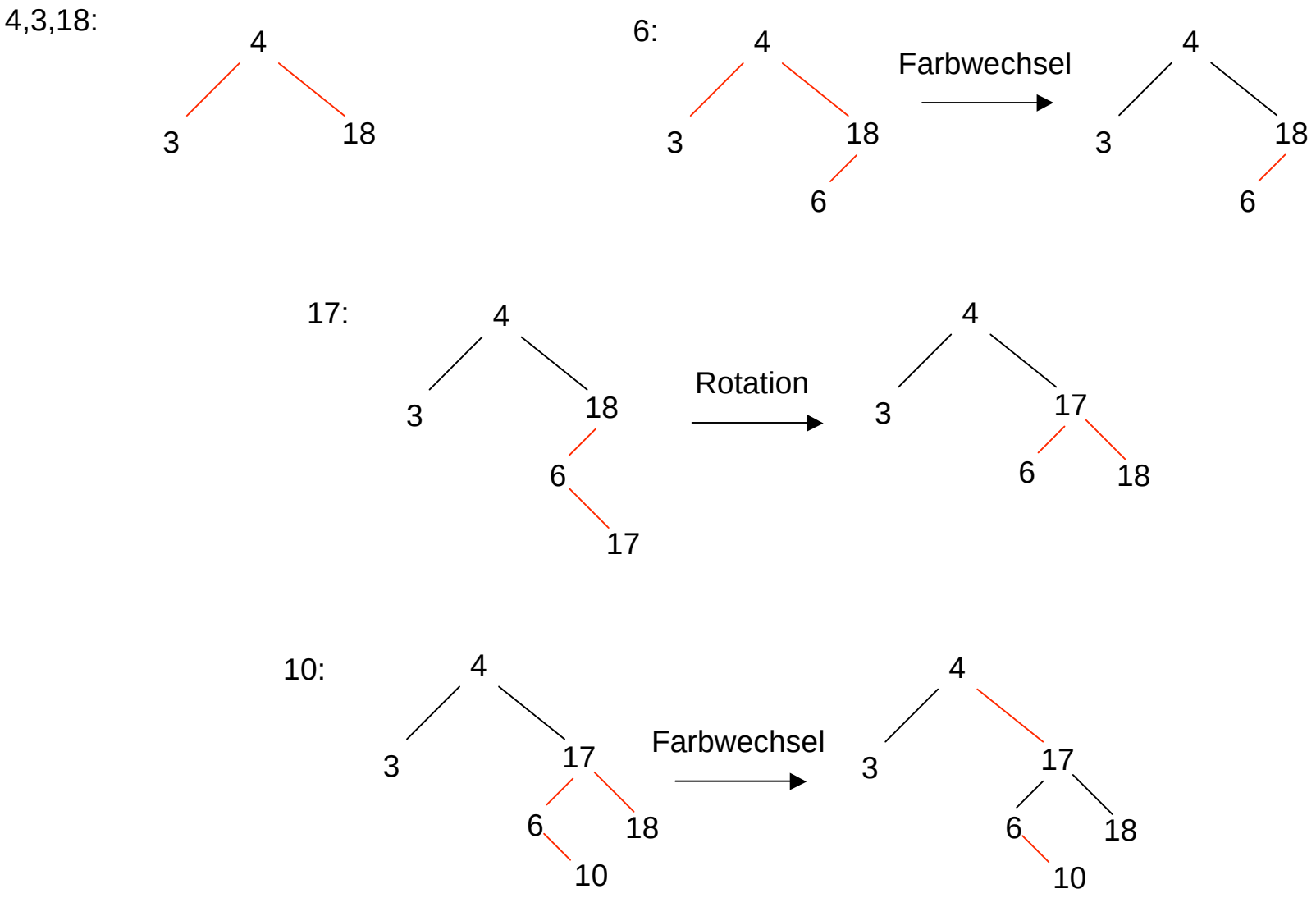
Farbwechsel entspricht Splitoperation



Rotation

(kommt nur in Repräsentation als Schwarz-Rot-Baum vor, wird automatisch in B-Baum-Repräsentation vermieden

Beispiel: Einfügen in Rot-Schwarz-Baum



Spezialfälle (2)

Top-Down Rot-Schwarz-Bäume

- Während es bei den AVL-Bäumen keine effiziente Alternative zum rekursiven Einfügen und Löschen gibt, lassen sich bei Rot-Schwarz-Bäumen das Einfügen und das Löschen iterativ (!) lösen:

  In einer Schleife wird der Baum von der Wurzel bis zum Blatt (einzufügender Knoten bzw. löschender Knoten) durchlaufen → Top-Down.

  Dabei werden prophylaktisch Split- bzw. Verschmelzungsoperationen durchgeführt. Split- und Verschmelzungsoperationen werden bei Rot-Schwarz-Bäumen im wesentlichen durch Rotationen und Umfärbungen durchgeführt.

  Ausführlich in [Weiss 2006] beschrieben. Operationen haben viele Sonderfälle und sind nicht ganz einfach zu programmieren.
  Es werden 3 Zeiger verwendet: aktueller Knoten, Elternknoten und Großelternknoten.

- Top-Down Rot-Schwarz-Bäume sind aufgrund der iterativen Implementierbarkeit sehr beliebt als Implementierungsvariante für die assoziativen STL-Container Set und Map. Zur Unterstützung der Iterator-Operationen werden sie zusätzlich mit einer Fädelung realisiert (→ gefädelte Suchbäume).

Teil 1:
Suchen

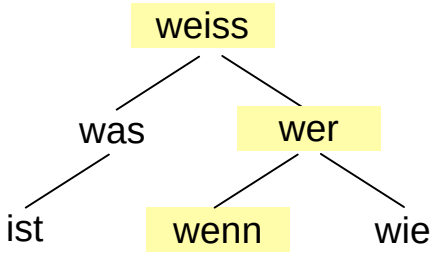
- Problemstellung
- Elementare Suchverfahren
- Hashverfahren
- Binäre Suchbäume
- Ausgeglichene Bäume
- B-Bäume
- Digitale Suchbäume

Tries

Problem mit den bisherigen Suchbäumen

Beim Suchen wird bei jedem durchlaufenen Knoten immer der komplette Schlüssel mit dem im Knoten abgespeicherten Schlüssel verglichen.

Bei langen Schlüsseln kann das sehr aufwendig werden.

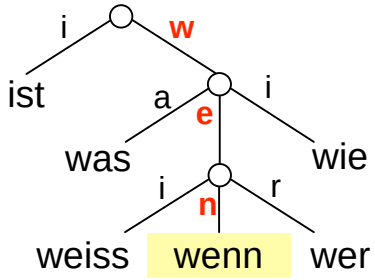


Suche nach „wenn“:

3 Vergleiche mit komplettem Schlüssel

Tries (aus retrieval; wird wie „try“ ausgesprochen)

Die Daten sind bei den Blättern abgespeichert. Der Schlüssel wird ziffern- bzw. zeichenweise betrachtet (daher auch der Name digitale Suchbäume) und entschieden, in welchem Teilbaum rekursiv weitergesucht wird. Erst beim Blatt wird der komplette Schlüssel verglichen.



Suche nach „wenn“:

3 Zeichen-Vergleiche und
1 Vergleich mit komplettem Schlüssel

Spezialfälle von Tries

Binäre Tries:

Die Schlüssel werden binär codiert. Bei jedem Knoten wird anhand des nächsten Bits entschieden, ob im linken oder rechten Teilbaum weitergesucht wird.

(Siehe Beispiel nächste Seite)

Alphabetische Tries:

Die Schlüssel sind Buchstabenfolgen (evtl. mit Sonderzeichen).
(siehe Beispiel vorige Seite).

Werden von einem Text sämtliche Suffixe in einem Trie abgespeichert, spricht man auch von einem Suffixbaum.

Suffixe sind „Nachsilben“ oder etwas allgemeiner formuliert Endteile von Strings. Beispielsweise enthält der String „abc“ die Suffixe „abc“, „bc“ und „c“.)

Suffixbäume werden als Indexierungsverfahren für große Texten eingesetzt. Typische Problemstellung: Finde in einem bekannten Text (z.B. die Bibel) alle Vorkommen eines Wortes (z.B. „Gott“).

Mehrwege-Tries:

Die Schlüssel werden als Zahl in einem Stellenwertsystem zur Basis m codiert.
(z.B. m = 26*26 = 256; Ziffern bestehen aus 2 Buchstaben).

Man hat dann im allgemeinen bei jedem Nicht-Blatt-Knoten m Teilbäume. Mehrwege-Tries sind eine mögliche Implementierungsvariante für externe Suchverfahren (d.h. Daten befinden sich zum größten Teil auf einen Hintergrundspeicher).

Binäre Tries (1)

Beispiel:

Binärer Trie mit den Schlüsseln (Beispiel aus [Sedgewick])

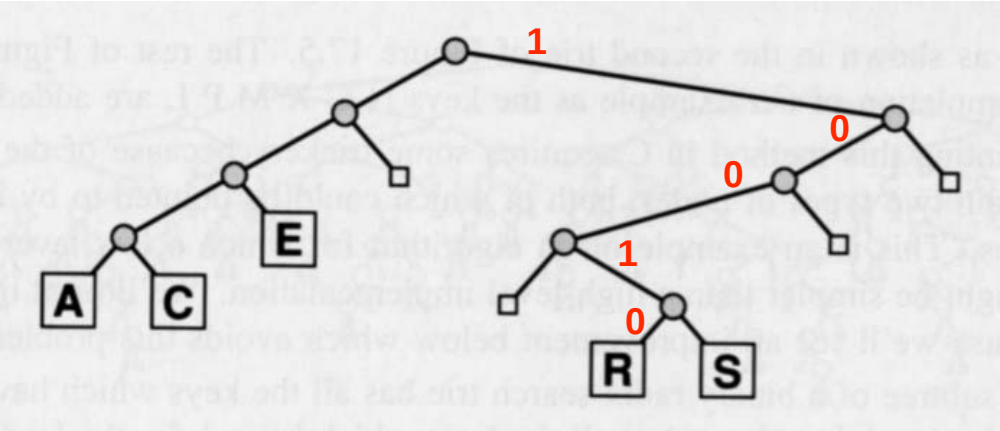
A = 00001

C = 00011

E = 00101

R = 10010

S = 10011

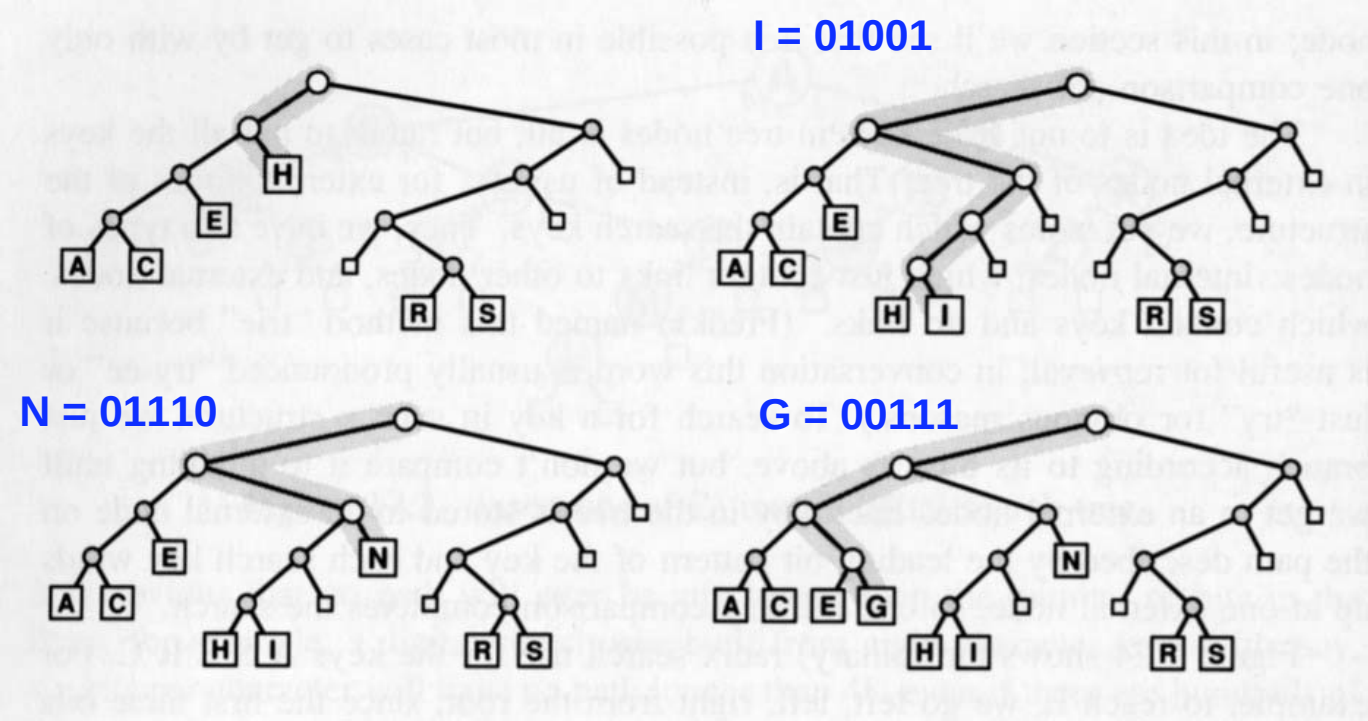


Suche nach R = 10010:

Bei Bit=1 wird im rechten und bei Bit=0 im linken Teilbaum weitergesucht.

Binäre Tries (2)

Einfügen:



Algorithmus:

Suche zuerst Schlüssel und füge dann neuen (bzw. neue) Knoten ein.

Endet die Suche bei einem leeren Baum (z.B. Einfügen von N = 01110), dann kann das neue Blatt direkt abgespeichert werden.

Endet die Suche bei einem Blatt (z.B. Einfügen von I = 01001), dann müssen eventuell so viele interne Knoten eingefügt werden, bis die Schlüsselbits sich unterscheiden.

Binäre Tries (3)

Eigenschaften:

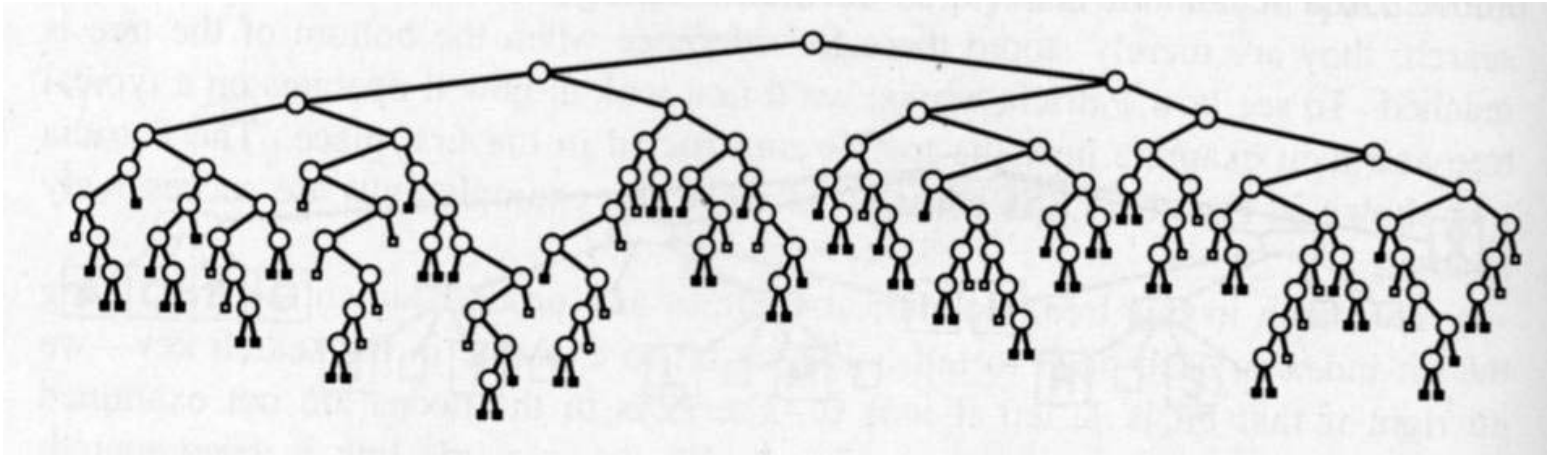
Laufzeit

Aufwand für Suchen eines Schlüssels:

- Im mittleren Fall: lg(n) Bit-Vergleiche, wobei n die Anzahl der Schlüssel im Trie ist.
- Im schlechtesten Fall: b Bit-Vergleiche, wobei die maximale Schlüssellänge b Bits beträgt.

Speicherplatz

Ein binärer Trie, der aus n zufälligen Schlüsseln mit m Bits besteht, hat ungefähr $n/\ln(2) \approx 1.44\ n$ viele interne Knoten

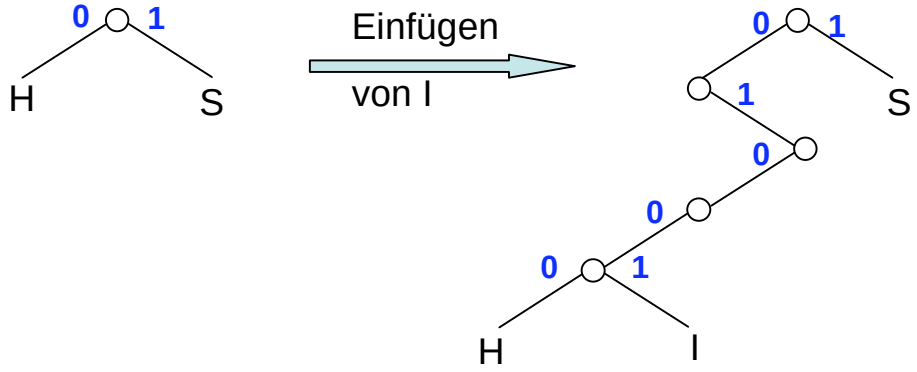


Ein Trie mit 95 zufälligen 10-Bit-Schlüsseln und 131 Knoten.

Patricia-Bäume (1)

Probleme mit bisherigen Tries:

- Das Einfügen eines neuen Schlüssel kann dazu führen, dass viele neue interne (nicht verzweigende) Knoten eingefügt werden müssen.
  Beispiel:



H = 01000
I = 01001
S = 10011

- Tries enthalten 2 Typen von Knoten:
  - Blattknoten mit Schlüssel
  - Interne Knoten (Nicht-Blatt-Knoten) mit Verweise auf linken und/oder rechten Teilbaum

  Nachteil: Aufwendig zu implementieren.

Patricia-Bäume (2)

Patricia

(Practical Algorithm to Retrieve Information Coded in Alpha-Numeric)

Ein Patricia-Baum vermeidet die beiden genannten Nachteile von Tries.

Ein Patricia-Baum weist gegenüber Tries folgende Unterschiede auf:

- Die nicht-verzweigenden internen Knoten aus den Tries (Knoten hat nur ein Kind) werden weggelassen. Damit hat jeder interne Knoten immer 2 Kinder.
  Jedoch muss dann in jedem Knoten die Stelle des Bits gespeichert werden, für das er zuständig ist.
- Statt die Schlüssel in Blätter zu speichern, werden die Schlüssel bei den internen Knoten abgespeichert. Statt Verweise auf Blätter erhält man dann nach oben zeigende Verweise.
  Man hat damit nur noch einen Knotentyp.

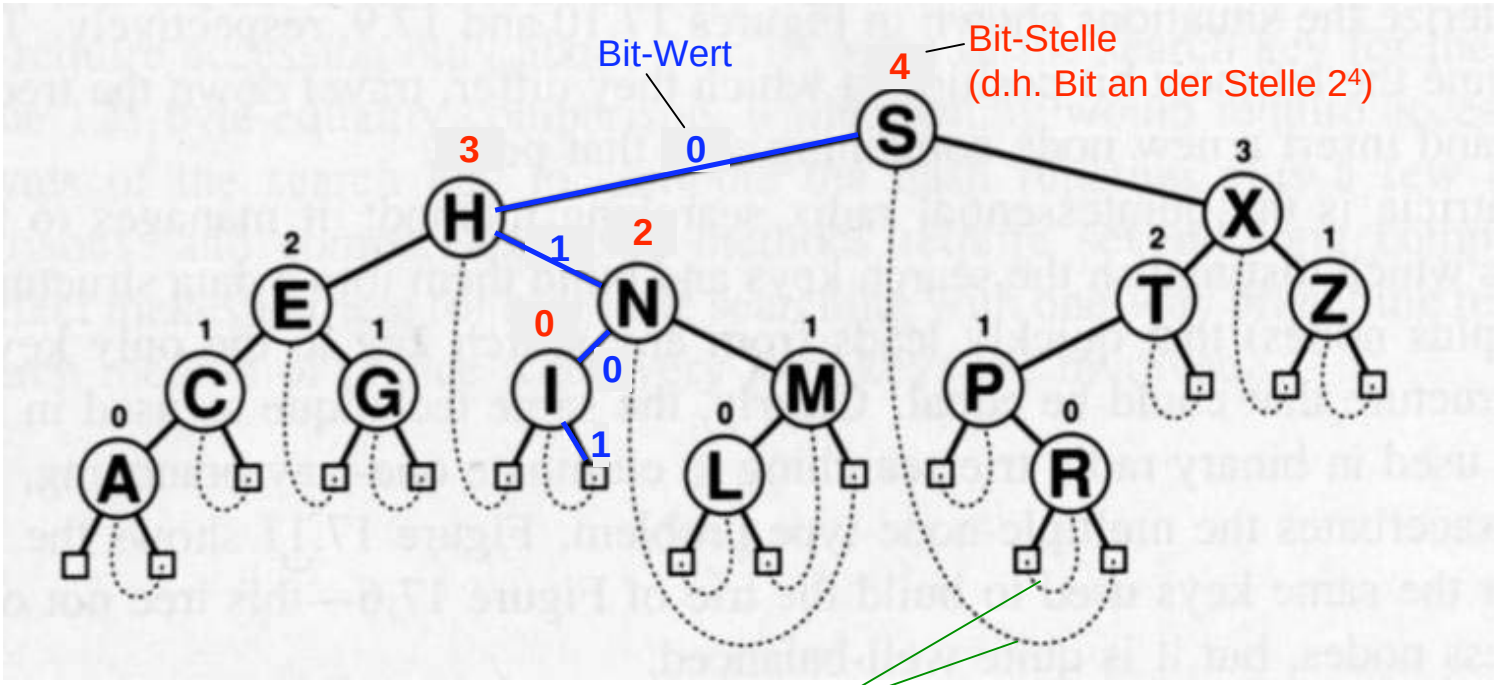
Patricia-Bäume (3)

Beispiel:

Schlüssel mit Binärcode

Um I = 01001 zu suchen, werden im Patriciabaum die Bit-Stellen 4, 3, 2 und 0 betrachtet.

A	00001
C	00011
E	00101
G	00111
H	01000
I	01001
L	01100
M	01101
N	01110
P	10000
R	10010
S	10011
T	10100
X	11000
Z	11010
43210	Bit-Stellen



nach oben zeigende Verweise.

Damit ist das rechte Kind von R ein Blatt und enthält den Schlüssel S.
Das linke Kind ist ebenfalls ein Blatt und enthält den Schlüssel R.