Fachhochschule
Dortmund

University of Applied Sciences and Arts
Fachbereich Informatik

Algorithmen und Datenstrukturen

VL08 – B-Bäume, Collections

Inhalt

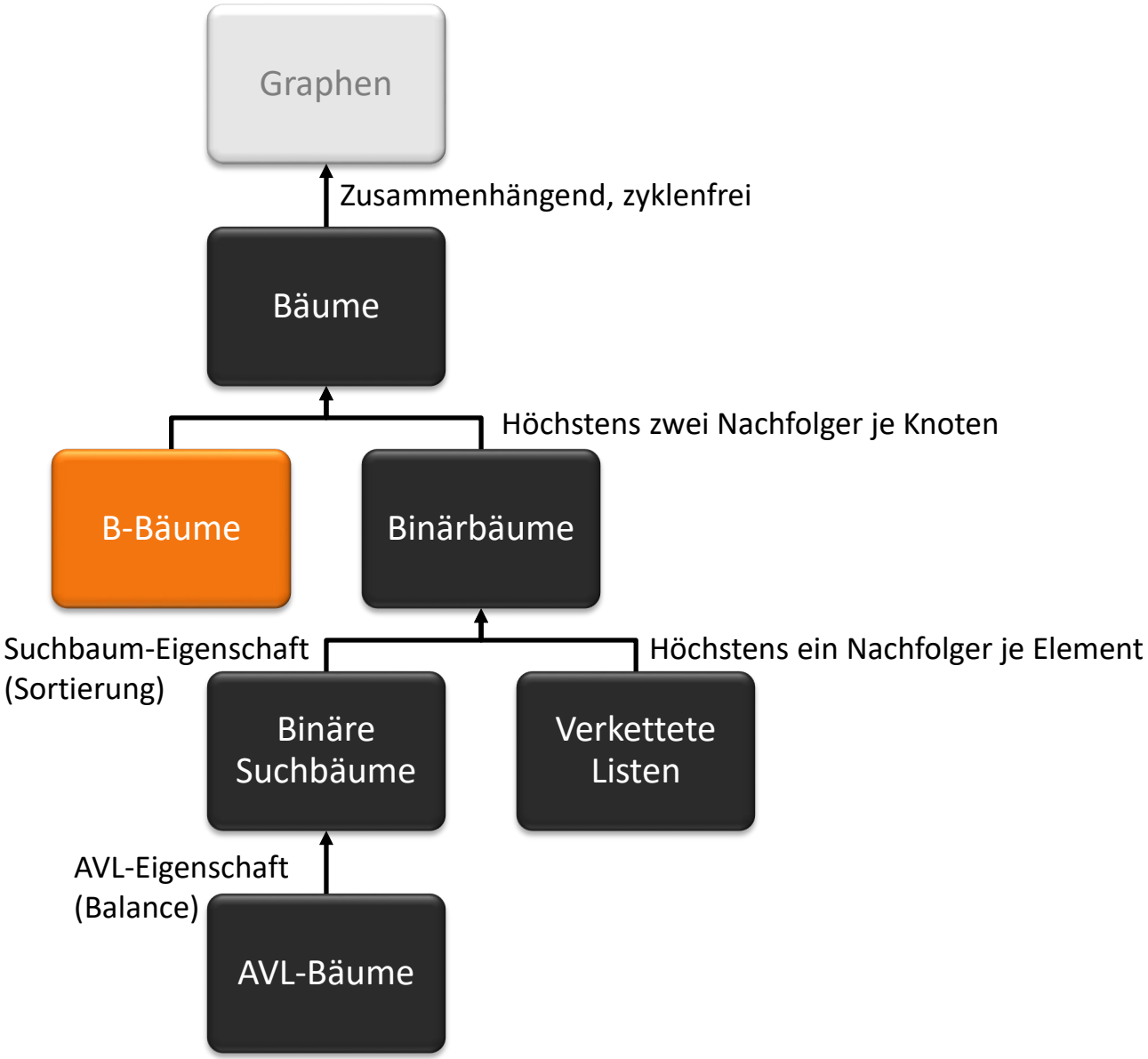
- B-Bäume
  - Motivation
  - Definition und Implementierung
  - Suchen und Einfügen
  - Entfernen
  - Komplexität

- Collections
  - `Set<E>`
  - `Map<K,V>`

B-Bäume

B-Bäume

Übersichtsgrafik



B-Bäume

Motivation

B-Bäume

Motivation

- Ein Knoten eines AVL-Baums benötigt 28 Byte Speicherplatz:
  - 8 Byte für den (64 Bit breiten) Zeiger auf das Nutzdaten-Objekt
  - 2∗8 Byte für die Zeiger auf die beiden Teilbäume
  - 4 Byte für den Balance-Faktor vom Typ `int`
  - Dazu wird in der Realität noch zusätzlicher Speicherbedarf für die Verwaltung des Objekts benötigt, z.B. für einen Referenz-Zähler für die Garbage Collection.
- AVL-Bäume werden im RAM-Speicher abgelegt:
  - Kleine Speicherblöcke können auf dem Heap dynamisch belegt und wieder freigegeben werden.
  - Der Zugriff auf den RAM-Speicher ist wahlfrei und schnell.

B-Bäume

Motivation

- Datenbank-Systeme arbeiten mit großen Datenmengen, die nicht im Arbeitsspeicher vorgehalten werden können. Sie sind stattdessen nur auf einer Festplatte oder SSD gespeichert:
  - Massenspeicher sind in **Blöcke** (**Sektoren**) eingeteilt, die in der Regel 512 oder 4096 Byte Speicherplatz für Nutzdaten bieten (siehe RuB).
  - Der Zugriff auf die Blöcke eine Festplatte/SSD ist zwar auch wahlfrei, aber im Vergleich zum Arbeitsspeicher extrem langsam.

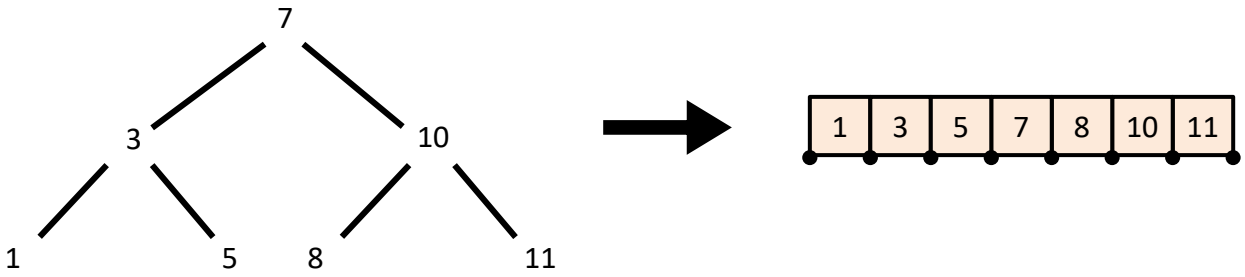
- Dasselbe gilt für den Arbeitsspeicher moderner Computer:
  - Zum Beispiel erfolgt der Zugriff auf den Arbeitsspeicher bei modernen x86-CPUs ebenfalls in **Seiten** mit einer Größe von 4096 Byte.
    - Suchbegriffe: **CPU cache**, **Paging**, **Page Table, Translation Lookaside Buffer**
  - Der Inhalt dieser Seiten wird im CPU-Cache gespiegelt. Das Nachladen einer derzeit nicht gespiegelten Seite dauert sehr lange – selbst wenn nur ein einziges Byte benötigt wird.

B-Bäume

Motivation

- Zusammenfassung:
  - Sowohl Festplatten und SSDs als auch der Arbeitsspeicher sind blockweise organisiert.
  - Sowohl beim Zugriff auf Festplatten und SSDs als auch beim Arbeitsspeicher dauert das Nachladen sehr lange.

- Idee:
  - Als Ziel wollen wir die Anzahl der Zugriffe auf den Speicher reduzieren, um die Geschwindigkeit der Datenstruktur zu erhöhen.
  - Dies können wir erreichen, indem wir den Platz besser ausnutzen. Jeder Knoten soll so viele Schlüssel enthalten wie ein Block fasst:



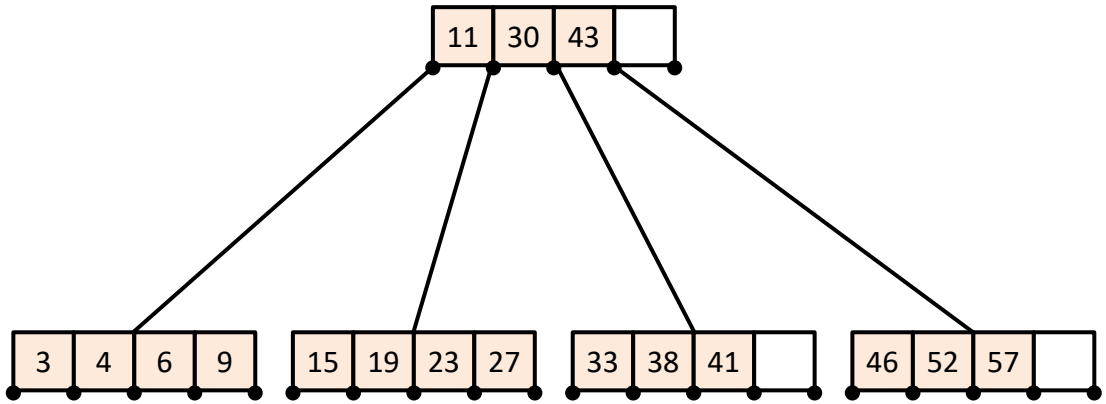
B-Bäume

Definition und Implementierung

B-Bäume

Definition

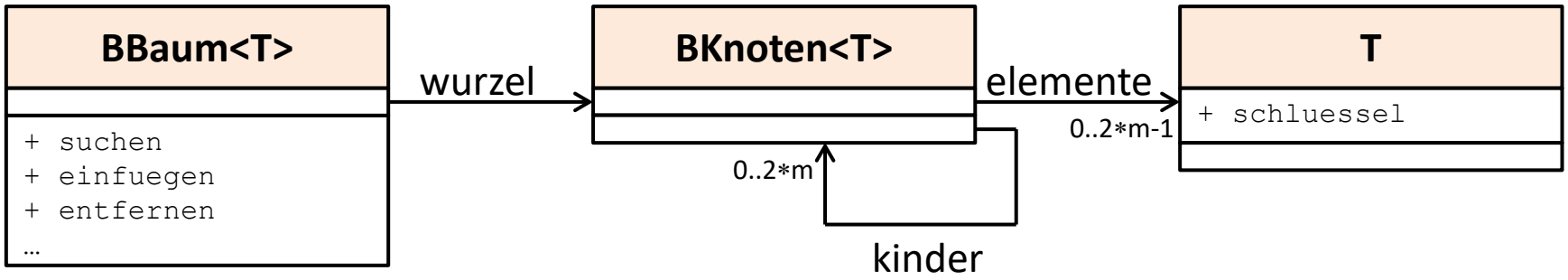
- Alle Blätter haben die gleiche Tiefe.
- Jeder B-Baum hat als Eigenschaft eine **Ordnung** m:
  - Jeder Knoten enthält mindestens m und höchstens $2m$ Schlüssel. Ausnahme: die Wurzel enthält zwischen 1 und $2m$ Schlüssel.
  - Innerhalb eines Knotens sind die Schlüssel aufsteigend sortiert.
- Alle Knoten außer den Blättern haben $c+1$ Nachfolger, wenn sie c Schlüssel enthalten:
  - Die Schlüssel im linken Teilbaum sind alle kleiner als der zugehörige Schlüssel im Elternknoten, die Schlüssel im rechten Teilbaum sind alle größer (analog zur Suchbaum-Eigenschaft aus VL06):



B-Bäume

Implementierung

- Ein B-Baum wird analog zu Binärbäumen, Binären Suchbäumen (VL06) und AVL-Bäumen (VL07) implementiert:
  - Ein `BKnoten` hat viele Vergleichselemente (`elemente`) und Teilbäume (`kinder`), die daher als Arrays gespeichert werden:



B-Bäume

Suchen und Einfügen

Suchen und Einfügen

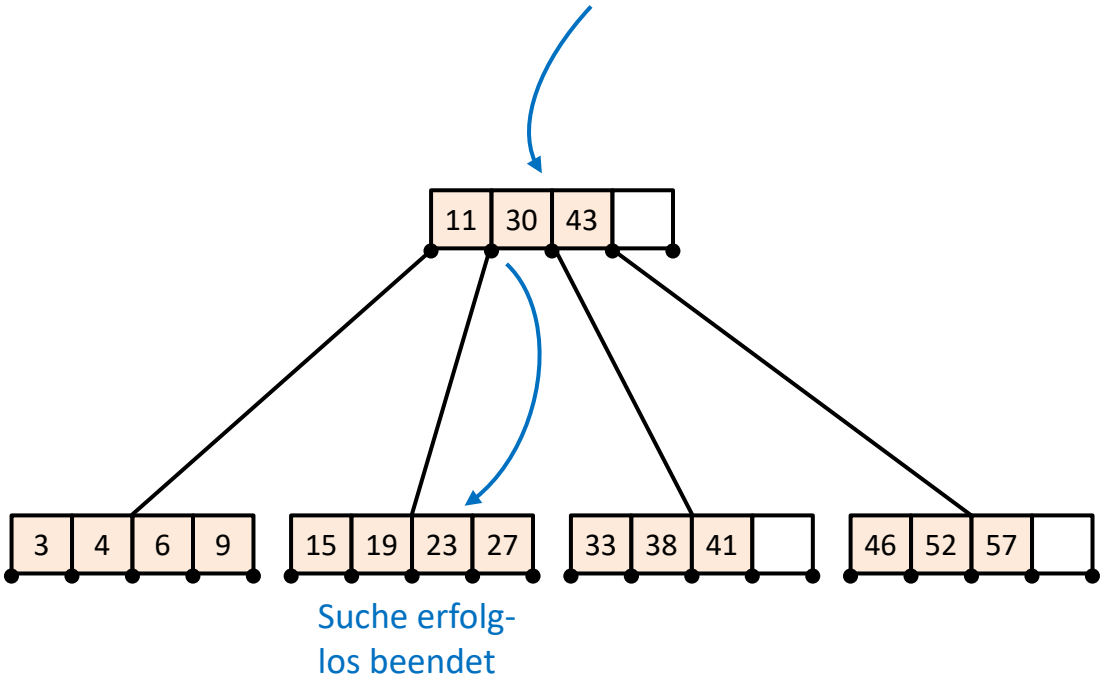
Suchen

- Beginnend mit der Wurzel als aktuellem Knoten `k` wiederhole:
  1. Prüfe, ob `x` im aktuellen Knoten `k` enthalten ist. Falls ja, dann ist `x` gefunden. → Ende
  2. Prüfe, ob `k` ein Blatt ist. Falls ja, dann ist `x` nicht vorhanden. → Ende
  3. Vergleiche `x` der Reihe nach mit allen Schlüsseln in `k`, beginnend mit dem kleinsten Schlüssel (Array `elemente`, siehe Folie 11).
  4. Beim ersten Schlüssel, der größer als `x` ist, setze `k` auf das Kind mit demselben Index (Array `kinder`, siehe Folie 11). → Fortfahren bei 1
  5. Wird kein solcher Schlüssel gefunden, wird `k` auf das letzte Kind gesetzt. → Fortfahren bei 1

Suchen und Einfügen

Beispiel

- Suchen des Schlüssels 24 in einem B-Baum der Ordnung $m=2$:



Suchen und Einfügen

Einfügen

- Einfügen eines noch nicht vorhandenen Schlüssels x:
  1. Wenn der B-Baum leer ist, wird ein neues Blatt mit x erzeugt und als Wurzel-Knoten eingefügt. → Ende
  2. Andernfalls wird nach x gesucht. Die Suche endet erfolglos in einem Blatt k.
  3. Es wird nun versucht, x in k einzufügen.
  4. Läuft das Blatt dabei über, d.h. es müsste $2m+1$ Schlüssel aufnehmen, so wird es in zwei Blätter mit je m Schlüsseln aufgespalten. Der Schlüssel mit dem mittleren Wert ist überzählig, und wird in den Elternknoten von k eingefügt.
     - Hier zeigt sich, wie genial B-Bäume sind: $2m+1$ Schlüssel sind gerade eben zu viele Schlüssel, um sie in einem einzigen Knoten (der ja maximal $2m$ Schlüssel speichern kann) unterzubringen. $2m+1$ Schlüssel sind aber gerade eben genug Schlüssel, um daraus 2 Knoten mit je m Schlüsseln zu machen (minimale Belegung) sowie einen Schlüssel als Trenner in den Elternknoten zu einzufügen.
  5. Läuft dabei der Elternknoten von k über, wird er nach demselben Prinzip aufgespalten. Wird schließlich die Wurzel aufgespalten, so wird eine neue Wurzel darüber erzeugt. Der Baum gewinnt an Höhe.

Suchen und Einfügen

Einfügen

- Interaktive Visualisierung:
  - Siehe https://www.cs.usfca.edu/~galles/visualization/BTree.html
  - **Achtung! Diese Webseite verwendet die Eigenschaft „Degree“ statt der Ordnung m, um den B-Baum zu beschreiben:**

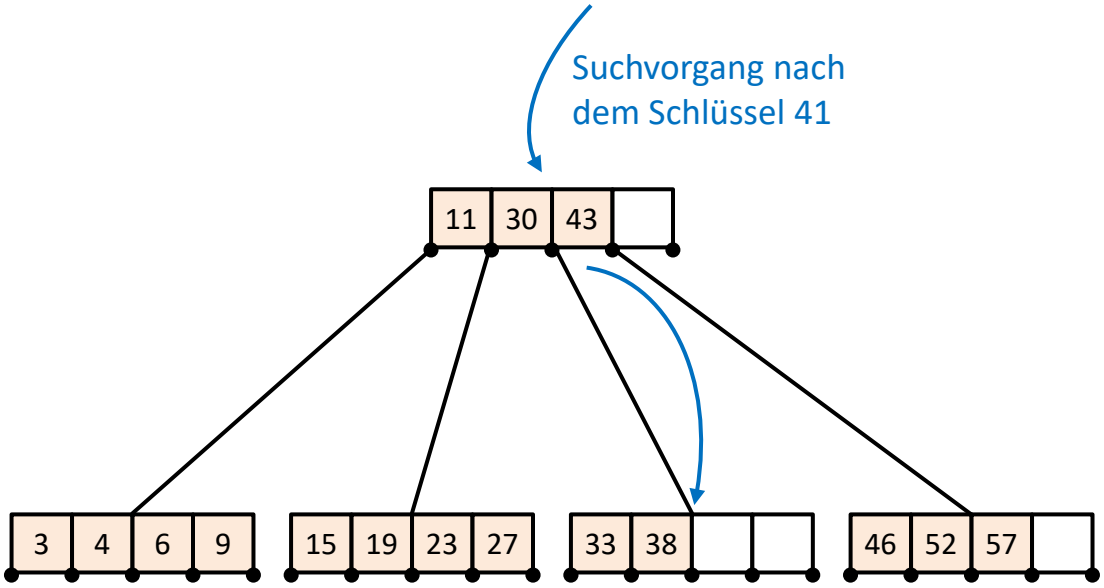
Ordnung m	„Degree“
1	3
2	5
3	7
…	…

Es handelt sich jedoch trotzdem um die beste interaktive Visualisierung, die uns bekannt ist.

Suchen und Einfügen

Beispiel

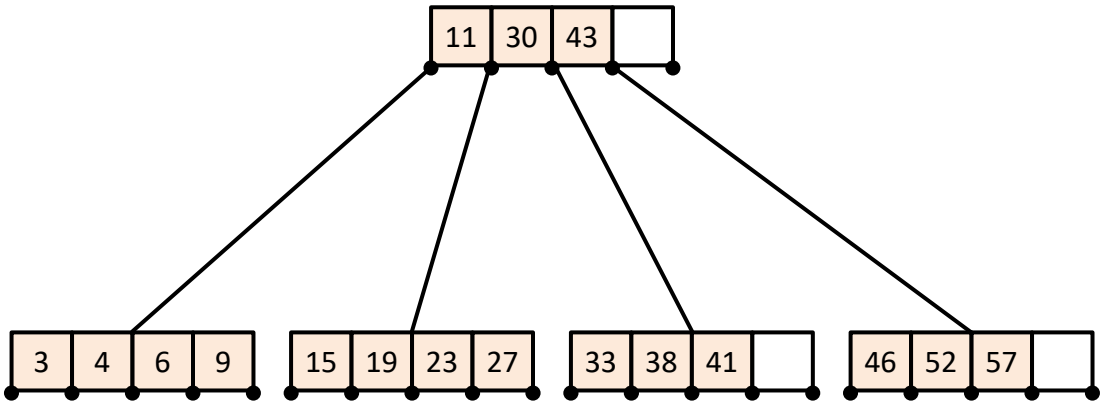
- Einfügen des Schlüssels 41 (B-Baum der Ordnung $m=2$):



Suchen und Einfügen

Beispiel

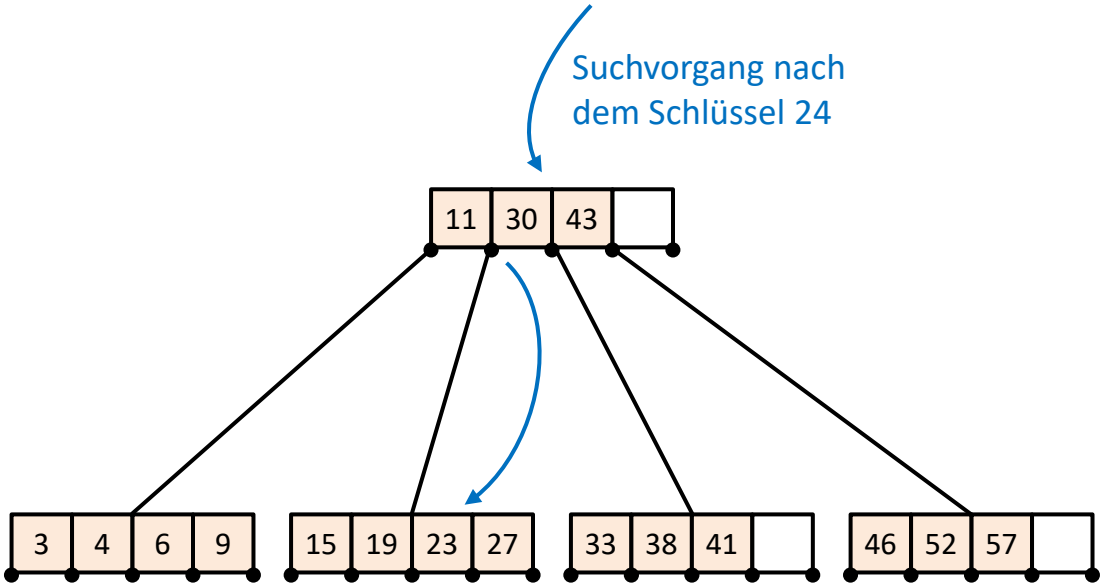
- Einfügen des Schlüssels 41 (Endzustand):



Suchen und Einfügen

Beispiel

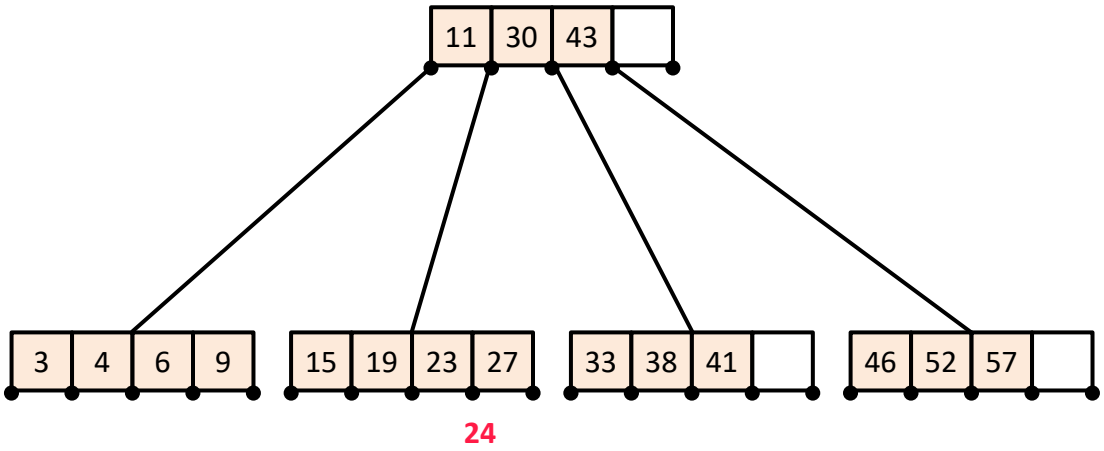
- Einfügen des Schlüssels 24:



Suchen und Einfügen

Beispiel

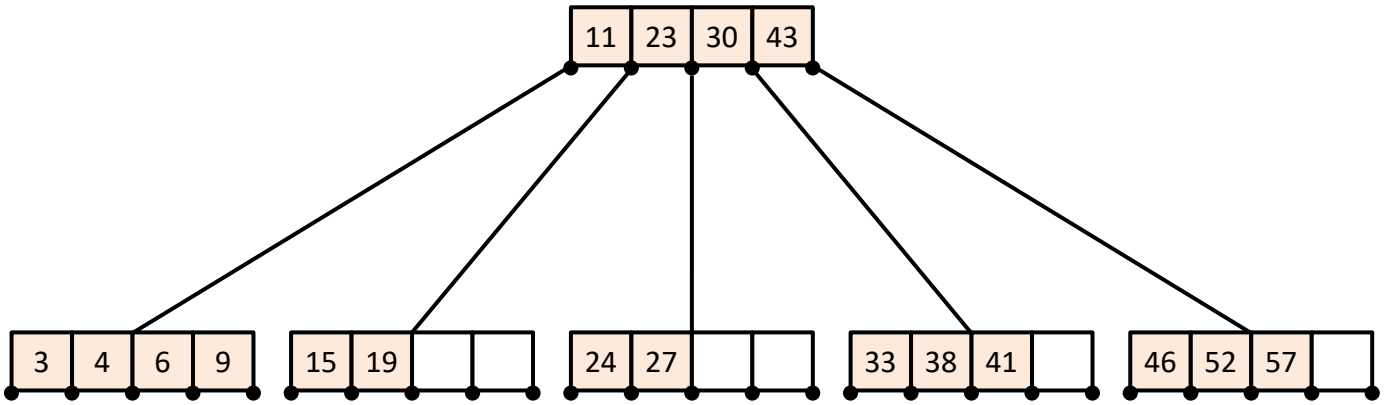
- Einfügen des Schlüssels 24 (**ungültiger** Zwischenzustand):



Suchen und Einfügen

Beispiel

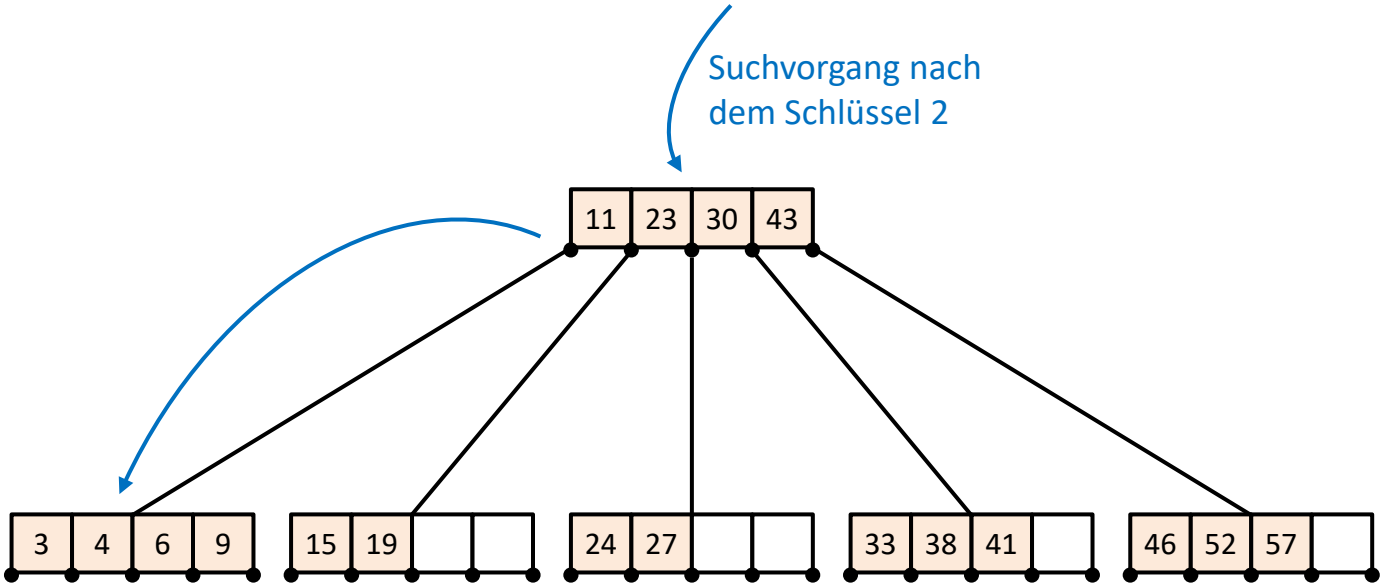
- Einfügen des Schlüssels 24 (Endzustand):



Suchen und Einfügen

Beispiel

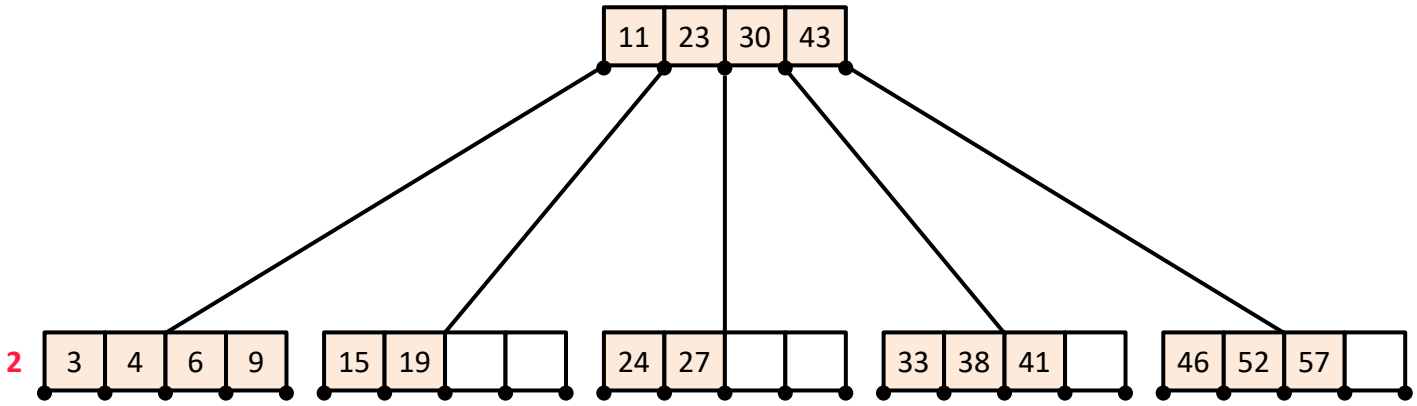
- Einfügen des Schlüssels 2:



Suchen und Einfügen

Beispiel

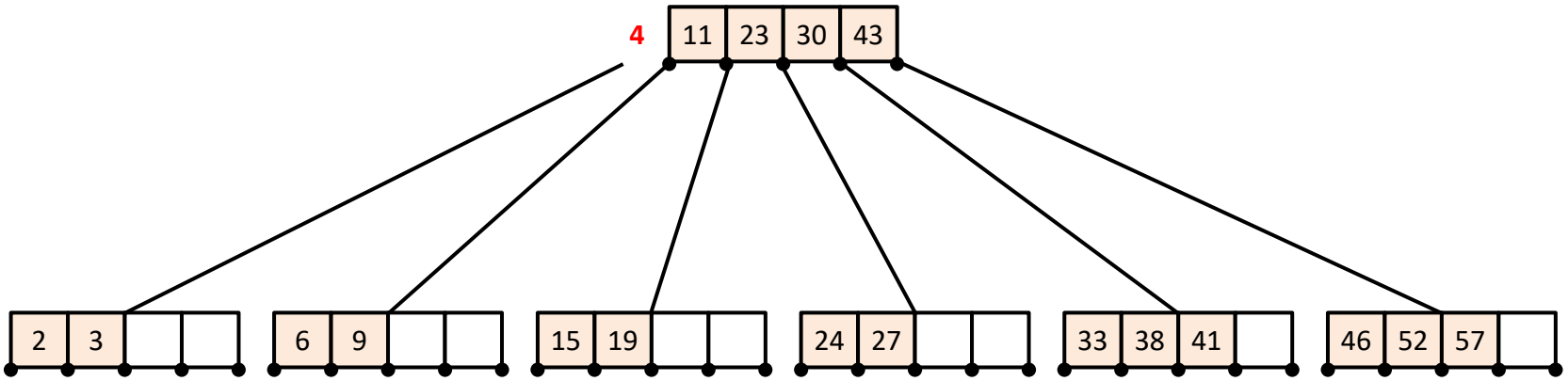
- Einfügen des Schlüssels 2 (**ungültiger** Zwischenzustand):



Suchen und Einfügen

Beispiel

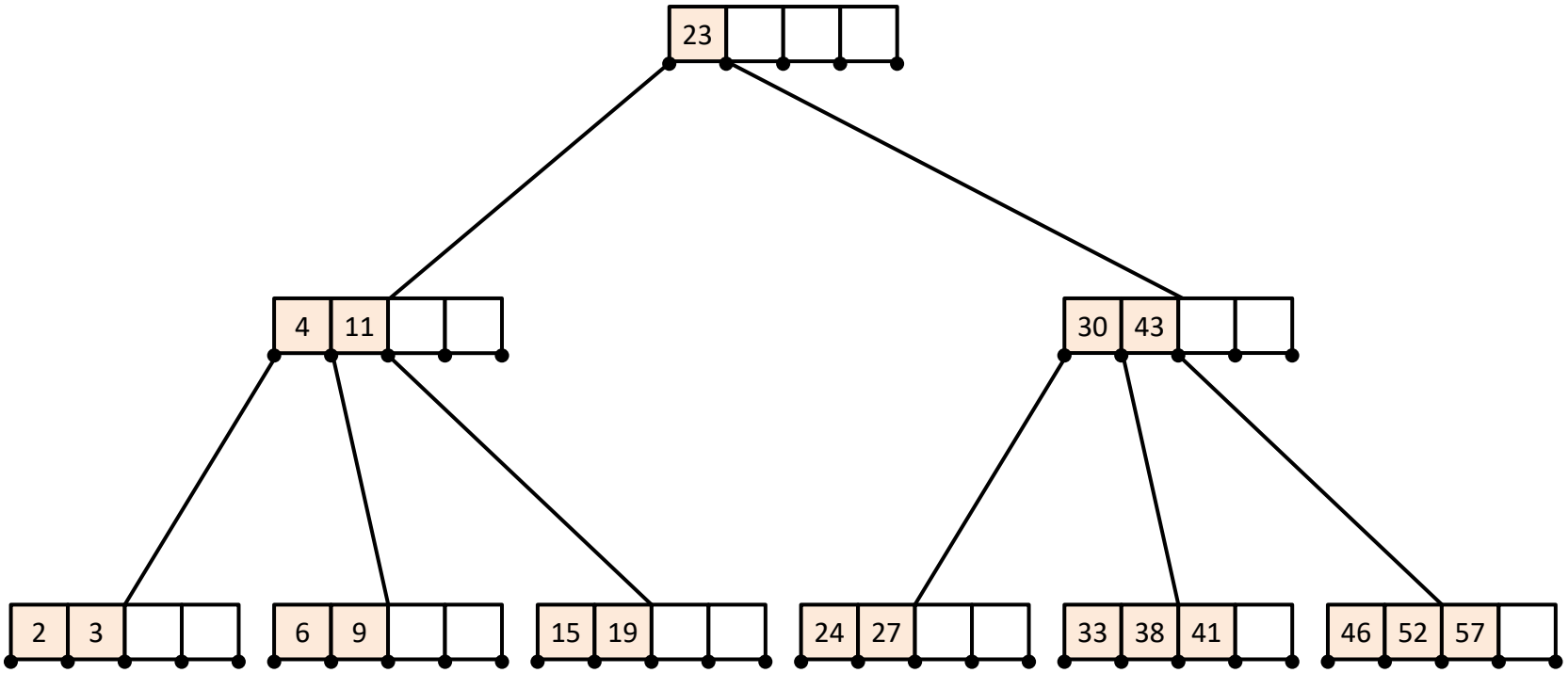
- Einfügen des Schlüssels 2 (**ungültiger** Zwischenzustand):



Suchen und Einfügen

Beispiel

- Einfügen des Schlüssels 2 (Endzustand):



Suchen und Einfügen

Aufspalten der Knoten

- Bei B-Bäumen erfolgt das ggf. notwendige Aufspalten der Knoten von unten nach oben am Ende der rekursiven Methode.

Für eine genauere algorithmische Beschreibung sei hier auf die Literatur (z.B. von Saake) verwiesen.

B-Bäume

Entfernen

Entfernen

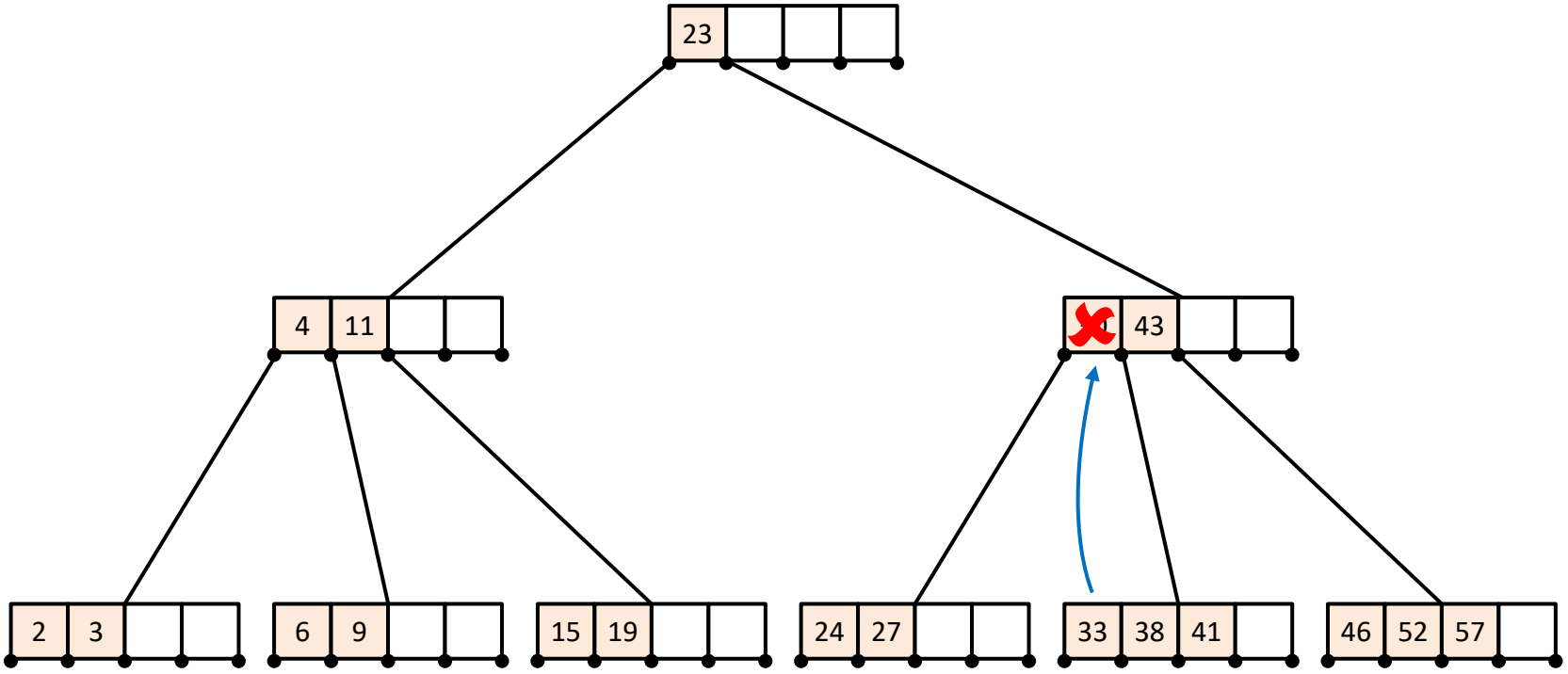
Teil 1: Entfernen am Blatt

- Entfernen des Opfers x aus dem B-Baum:
  1. Wenn x in einem Blatt ist, dann wird x direkt aus dem Blatt entfernt.
     → Weiter bei 3
  2. Wenn x nicht in einem Blatt ist, dann wird x zunächst durch seinen symmetrischen Vorgänger oder Nachfolger ersetzt und dadurch entfernt.
     Der symmetrische Vorgänger (größter Schlüssel im „linken“ Teilbaum bzgl. x) oder Nachfolger (kleinster Schlüssel im „rechten“ Teilbaum bzgl. x) befindet sich immer in einem Blatt. Er wird dort entfernt.
  3. Wenn das entsprechende Blatt nach dem Entfernen noch mindestens m Schlüssel enthält, so ist der Vorgang beendet. → Ende

Entfernen

Beispiel: Löschen in innerem Knoten

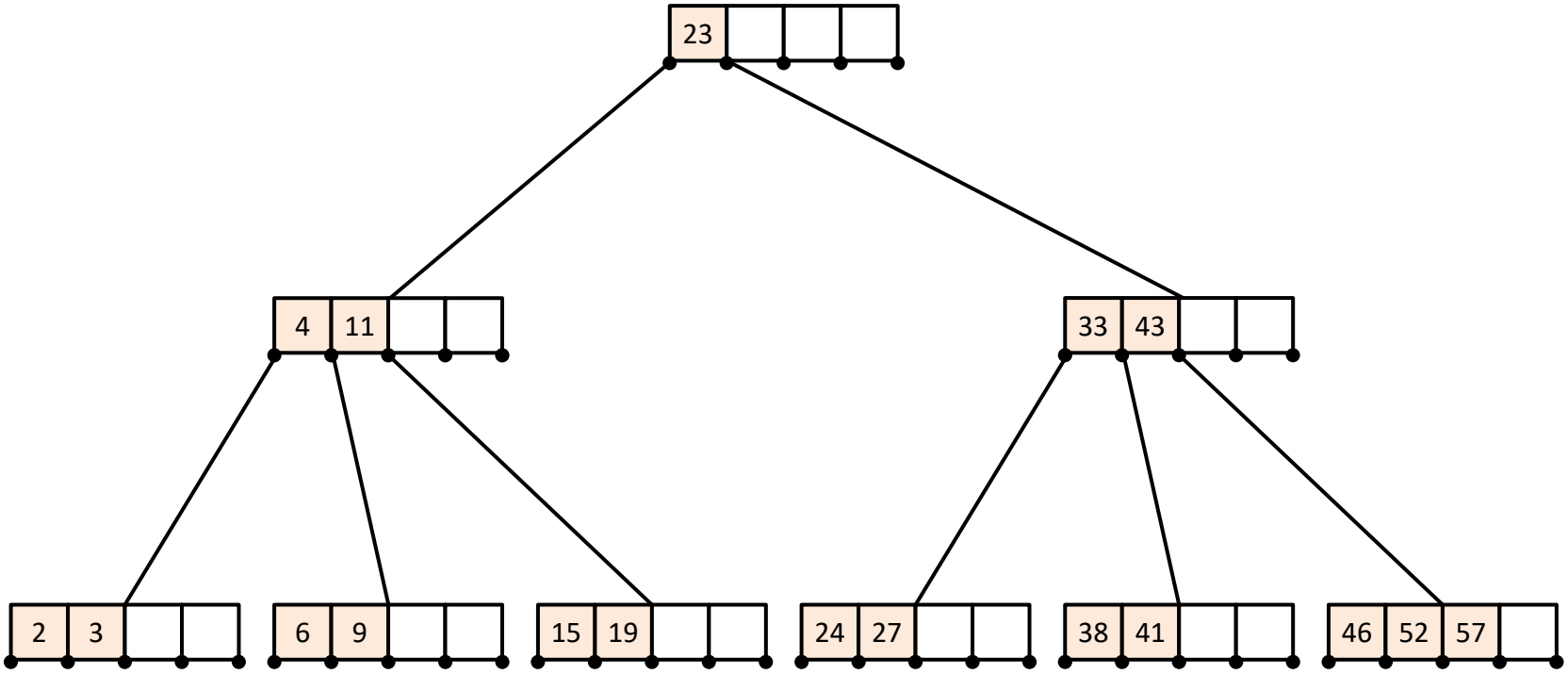
- Entfernen des Schlüssels 30 (Zwischenzustand):
  - Löschen in innerem Knoten, indem die Zahl durch ihren symmetrischen Nachfolger (hier: 33) ersetzt wird (Algorithmus Punkt 2)



Entfernen

Beispiel: Löschen in innerem Knoten

- Entfernen des Schlüssels 30:
  - Die 30 in innerem Knoten wurde durch ihren symmetrischen Nachfolger 33 aus einem Blatt ersetzt. Jetzt wird die 33 aus dem Blatt entfernt.



Entfernen

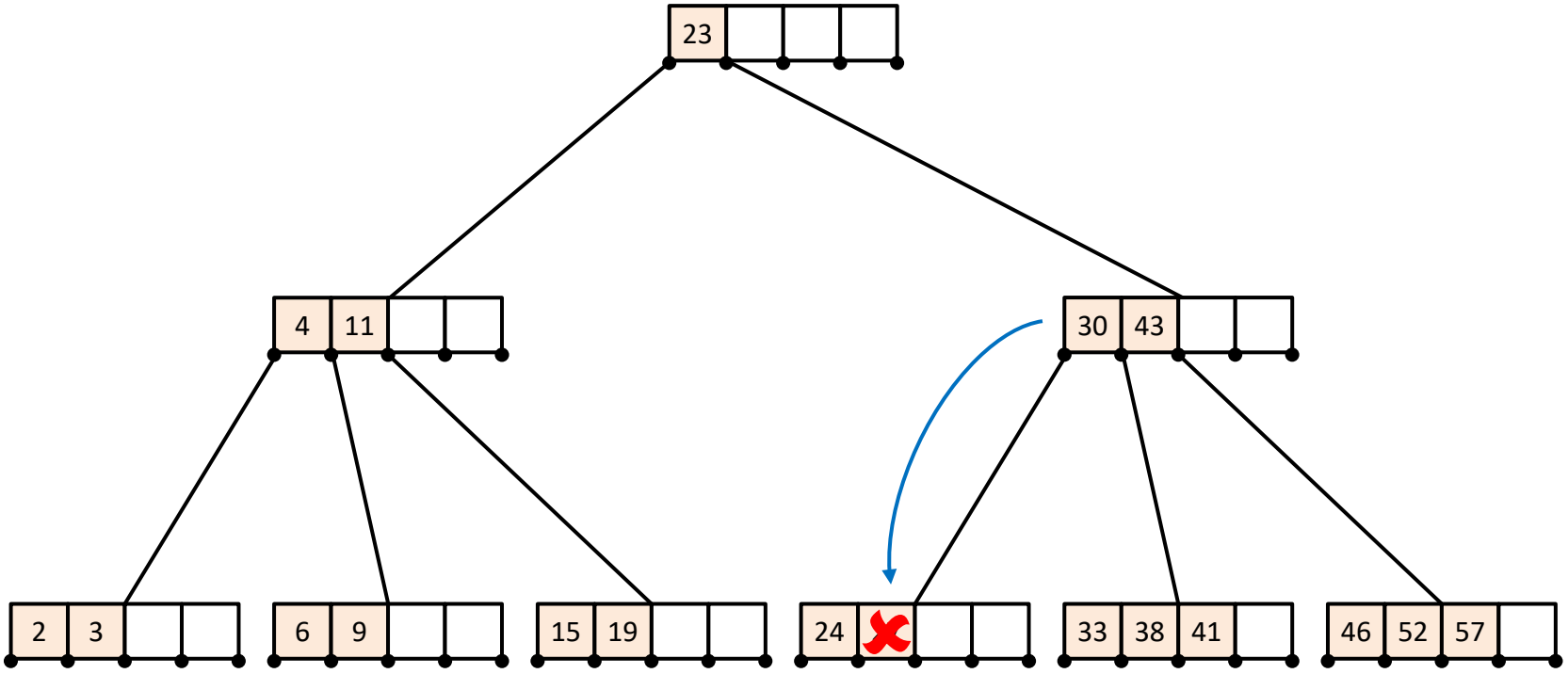
Teil 2: Unterlauf beim Entfernen

4. Wenn das entsprechende Blatt nach dem Entfernen weniger als m Schlüssel enthält, so hat ein **Unterlauf** stattgefunden. Dann wird versucht, diesen Unterlauf durch das Ausleihen von Schlüsseln aus den Nachbarknoten zu beseitigen.

Entfernen

Beispiel: Löschen im Blatt mit Unterlauf und Ausleihen

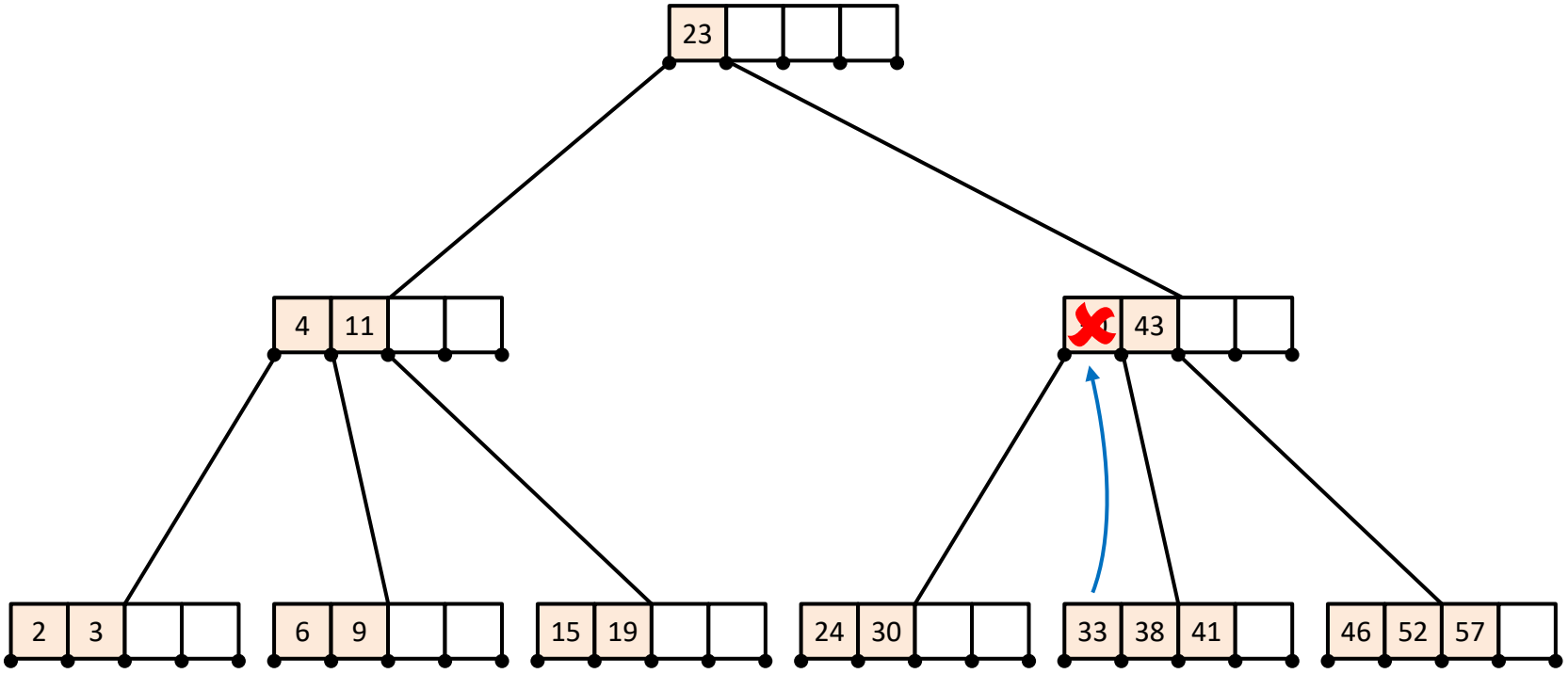
- Entfernen des Schlüssels 27:
  - Beim Löschen im Blatt entsteht ein Unterlauf. Der Ausgleich des Unterlaufs erfolgt durch Ausleihen von Schlüsseln.



Entfernen

Beispiel: Löschen im Blatt mit Unterlauf und Ausleihen

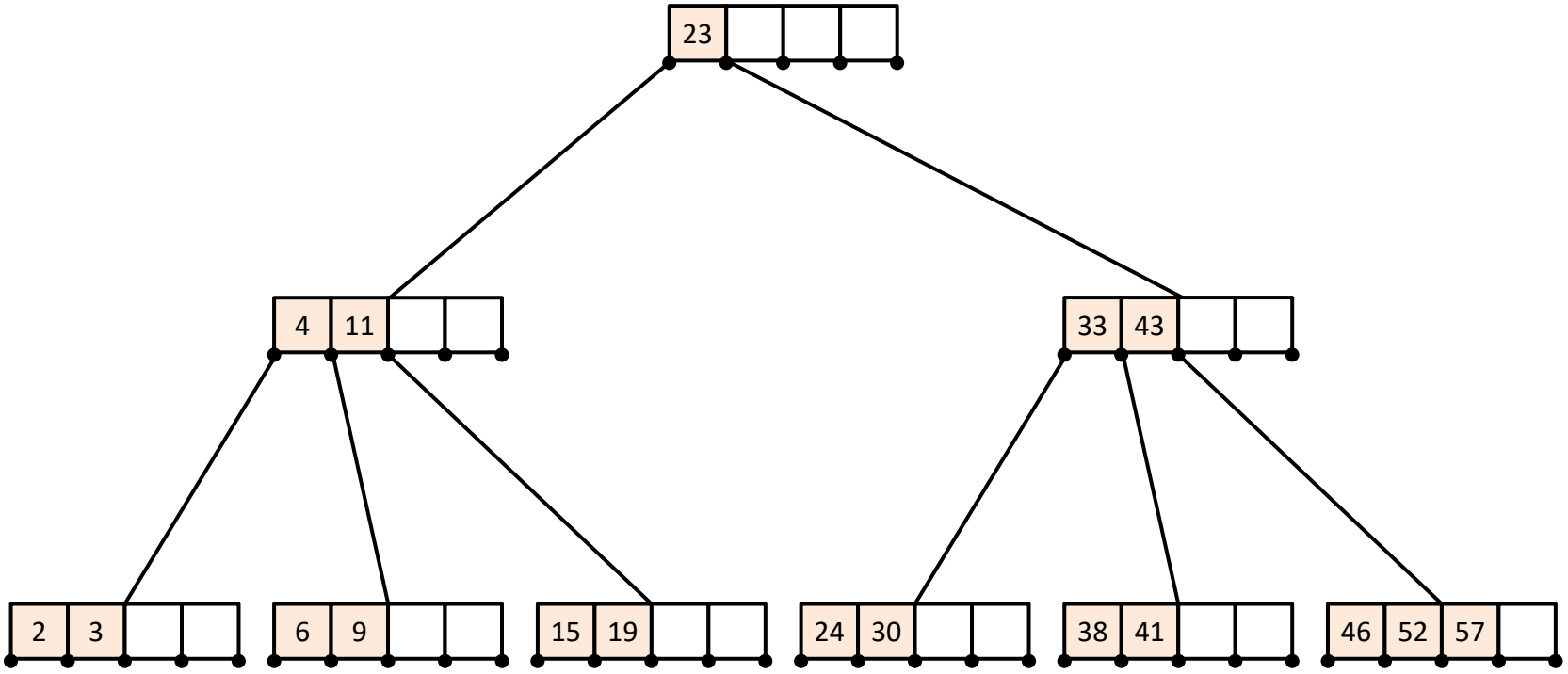
- Entfernen des Schlüssels 27 (Zwischenzustand):
  - Die 27 wurde im Blatt entfernt und die 30 aus dem Vaterknoten ausgeliehen. Die 30 nimmt jetzt den Platz der 27 im Blatt ein.



Entfernen

Beispiel: Löschen im Blatt mit Unterlauf und Ausleihen

- Entfernen des Schlüssels 27:
  - Die 33 aus dem Nachbarblatt wurde in den Vaterknoten an die ehemalige Stelle der 30 verschoben.



Entfernen

Teil 3: Unterlauf beim Entfernen

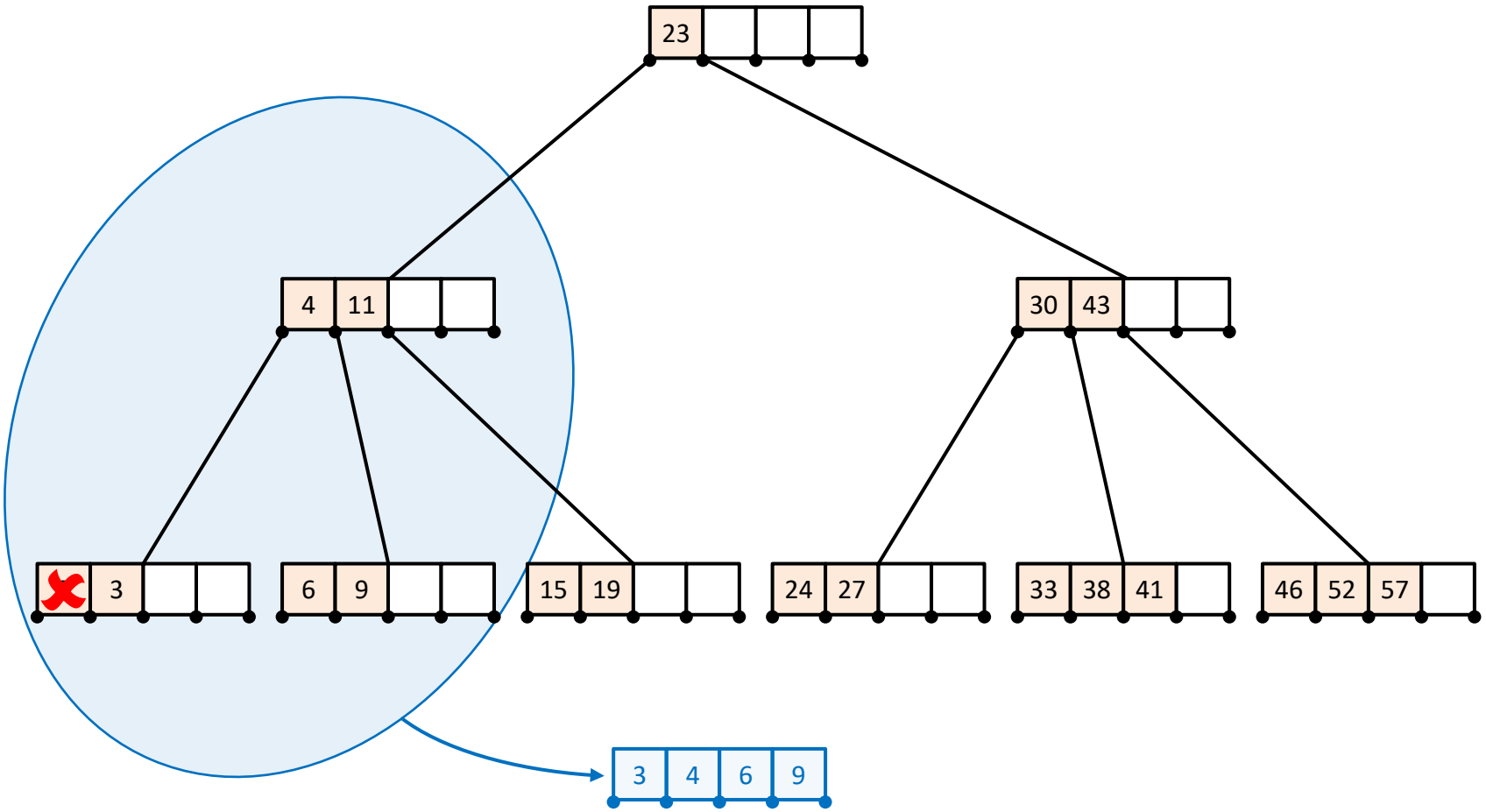
4. Wenn das entsprechende Blatt nach dem Entfernen weniger als m Schlüssel enthält, so hat ein Unterlauf stattgefunden. Dann wird versucht, diesen Unterlauf durch das Ausleihen von Schlüsseln aus den Nachbarknoten zu beseitigen.
5. Ist dies nicht möglich, weil die Nachbarknoten ebenfalls nur m Schlüssel haben, so wird der Knoten mit dem Unterlauf (d.h. mit m-1 Schlüsseln) mit einem Nachbarknoten (mit m Schlüsseln) und einem Schlüssel des gemeinsamen Elternknotens verschmolzen.

- In jedem Fall ist entweder Schritt 4 oder Schritt 5 möglich:
  - Das Blatt hat m-1 Schlüssel, ein Nachbar m+1 Schlüssel. Durch Ausleihen kann der Unterlauf behoben werden, es gilt jedoch: $m\text{-}1+m+1+1 > 2m$. Für ein Verschmelzen liegen zu viele Schlüssel vor.
  - Das Blatt hat m-1 Schlüssel, beide Nachbarn haben m Schlüssel. Dies sind zu wenige Schlüssel für das Ausleihen, der Unterlauf würde nur verschoben. Aber: $m\text{-}1+m+1 = 2m$. Es sind also gerade so wenige Schlüssel, dass ein Verschmelzen möglich wird.

Entfernen

Beispiel: Löschen mit Unterlauf und Verschmelzen

- Entfernen des Schlüssels 2:
  - Beim Löschen im Blatt entsteht ein Unterlauf.



Entfernen

Teil 4: Unterlauf beim Entfernen

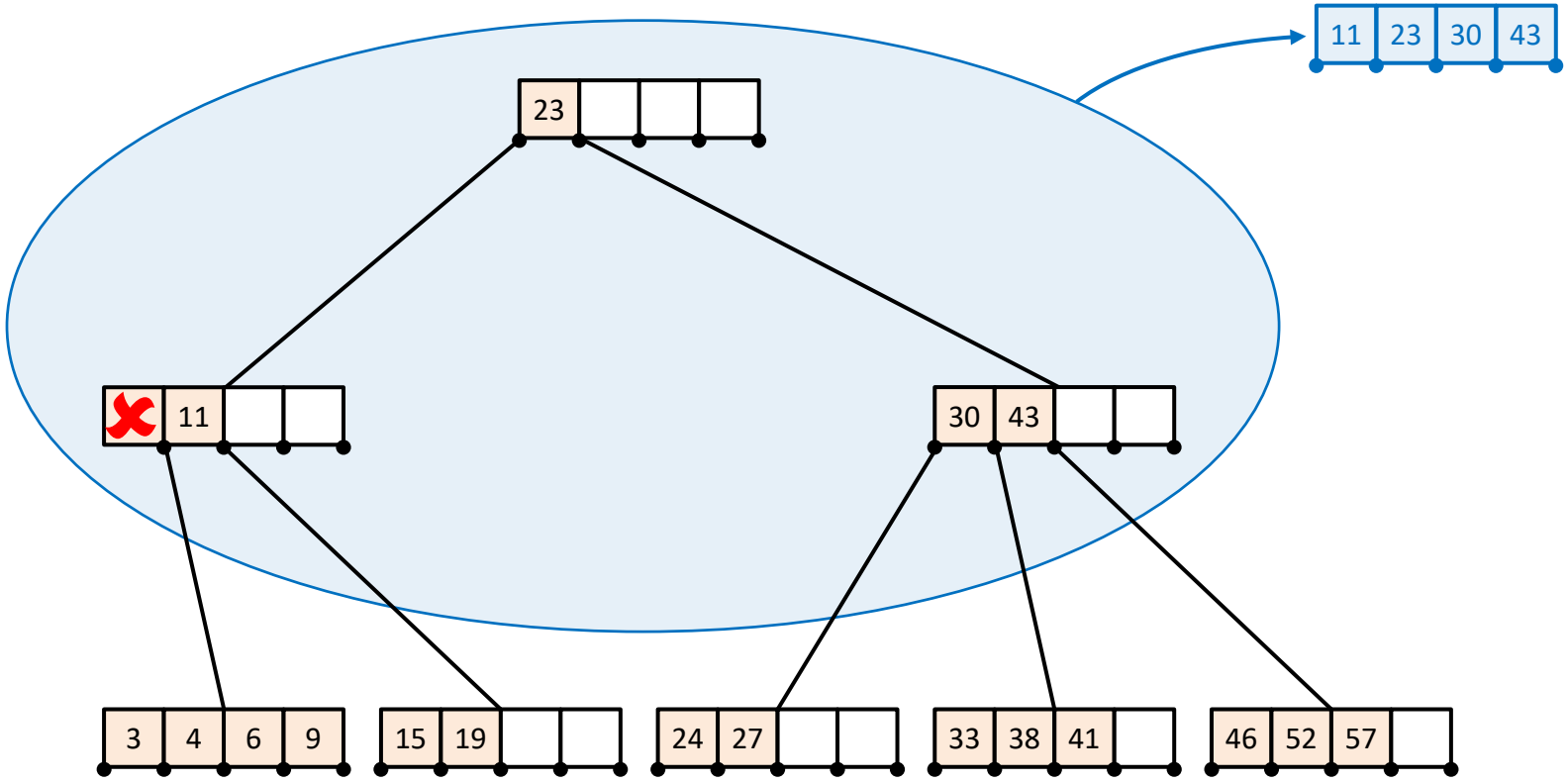
- Die Schlüsselzahl des Elternknotens reduziert sich durch Verschmelzen möglicherweise ebenfalls unter m, so dass das Verfahren ggf. bis zur Wurzel propagiert werden muss.
  - Dabei kann es sein, dass sich die Schlüsselzahl der Wurzel auf 0 reduziert. In diesem Fall gibt es auf der nächsten Ebene nur einen einzigen Knoten. Dieser wird zur neuen Wurzel.
  - Das ist die einzige Art und Weise, wie sich die Höhe eines B-Baumes verringern kann!

Eine genauere algorithmische Behandlung des Entfernens soll hier wiederum unterbleiben.

Entfernen

Beispiel: Löschen mit Unterlauf und Verschmelzen

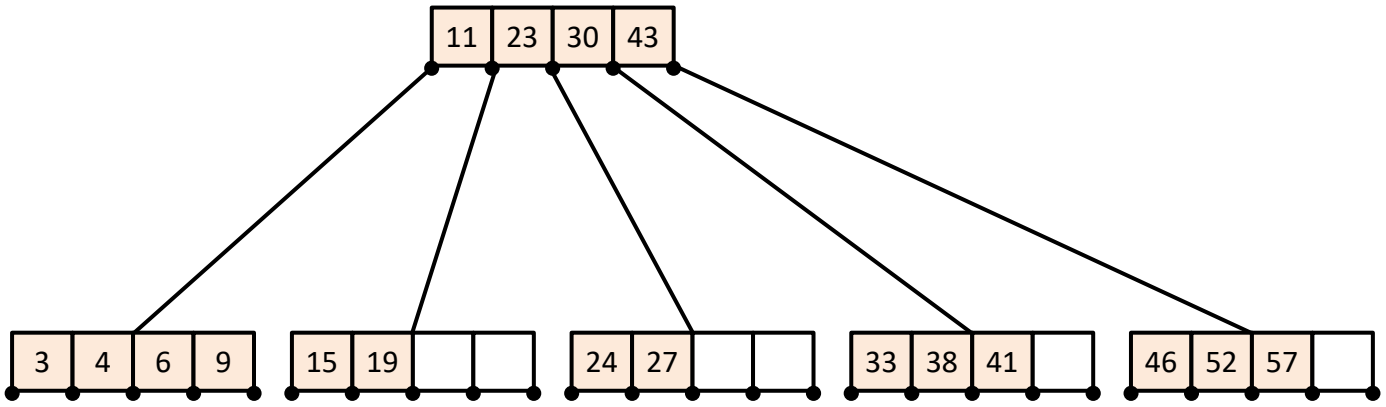
- Entfernen des Schlüssels 2 (Zwischenzustand):
  - Das Verschmelzen führt zu einem Unterlauf in einem inneren Knoten.



Entfernen

Beispiel: Löschen mit Unterlauf und Verschmelzen

- Entfernen des Schlüssels 2:
  - Durch Verschmelzen wird Wurzelknoten leer und entfernt. Die Höhe des B-Baums verringert sich dadurch.



B-Bäume

Komplexität

B-Bäume

Komplexität

- Der Aufwand für das Einfügen, Suchen und Löschen in B-Bäumen der Ordnung m ist $O(\log_m n)$.
- Beispiel:
  - Die typische Blockgröße sei 4096 Byte, für jeden Schlüssel plus Verweis auf die weiteren Nutzdaten sollen 16 Byte und für jeden Verweis auf Nachfolgeknoten 8 Byte benötigt werden.
  - Pro Block lassen sich also $m=85$ bis $2m=170$ Schlüssel speichern.
  - Der B-Baum hat somit die Ordnung $m=85$.
  - Bei 1.000.000 Datensätzen sind im Worst Case $\lceil \log_{85} 1.000.000 \rceil = 4$ Zugriffe notwendig. Bei einem Binärbaum werden dagegen $\lceil \log_2 1.000.000 \rceil = 20$ Zugriffe benötigt.

- B-Bäume sparen also kostbare Rechenzeit, was jedoch durch max. 50% Verschwendung von billigem Speicherplatz erkauft wird.

COLLECTIONS

Collections

Grundformen

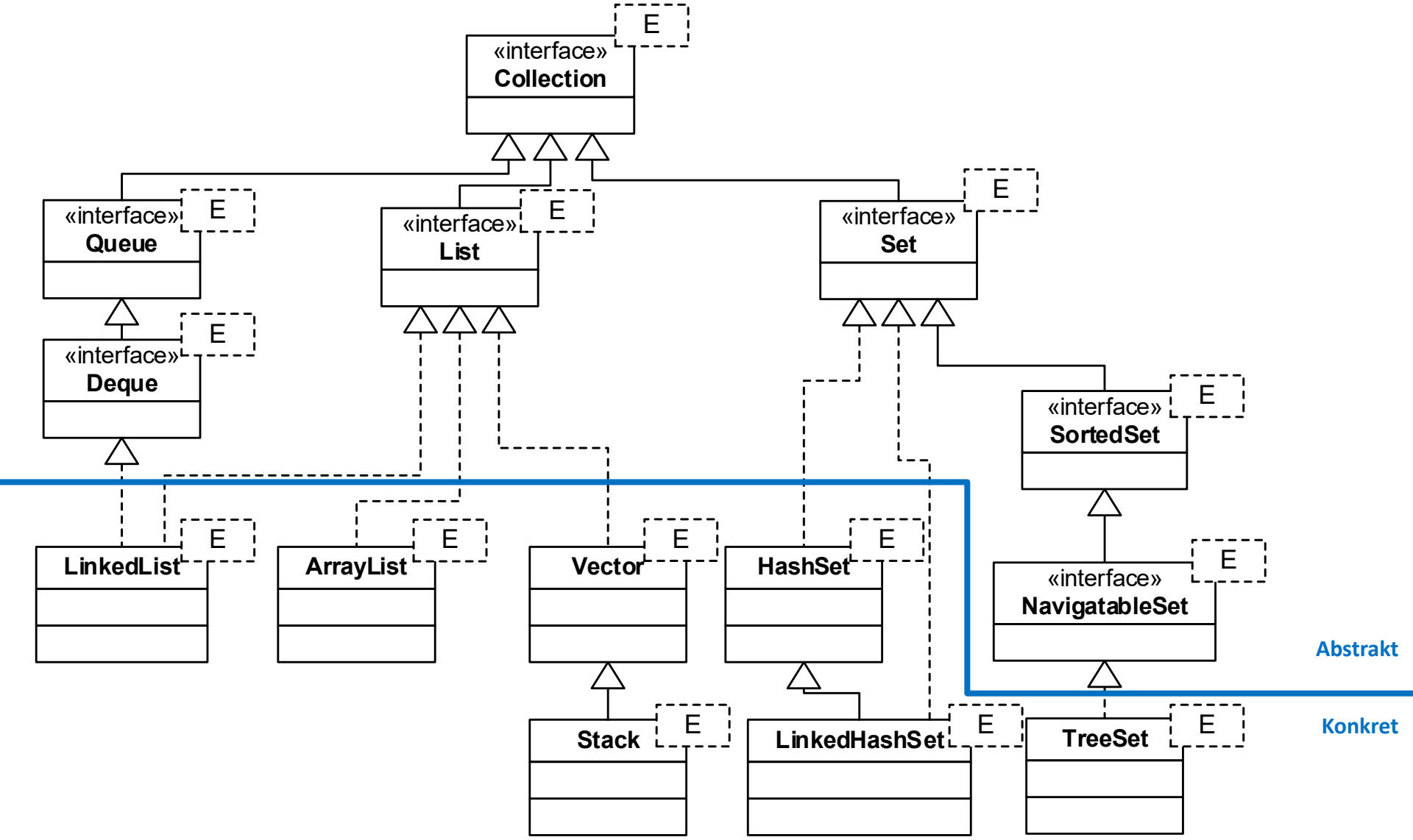
- `List<E>`:
  - Beliebig große Lineare Liste von Elementen des Typs E, auf die sowohl sequenziell als auch wahlfrei über einen Index zugegriffen werden kann
- `Queue<E>`:
  - Menge von Elementen, die in Form einer Warteschlange organisiert sind (typischerweise nach dem FIFO-Prinzip, aber auch z.B. nach LIFO oder Priorität)
- `Set<E>`:
  - Ansammlung von Elementen, auf die mit typischen Mengenoperationen zugegriffen werden kann, und die keine Elemente doppelt enthält
- `Map<K,V>`:
  - Abbildung von Elementen eines Typs `K` (Key) auf Elemente eines anderen Typs `V` (Value), also eine Menge zusammengehöriger Objekt-Paare (Schlüssel und Wert)

Collections

Set<E>

Collections

Collection-Interface



Set<E>

Eigenschaften

- Eine Collection vom Typ `Set<E>` ist die Java-Repräsentation einer mathematischen Menge von Objekten der Klasse `E`, wie z.B. `Set<String>`, `Set<Integer>`, `Set<Student>`, ...
- Sie enthält keine doppelten Elemente, d.h. es darf zu keinem Zeitpunkt zwei Elemente `a` und `b` geben, für die `a.equals(b)==true` ist.
- Die Elemente eines `Set<E>` haben im Gegensatz zu Listen keine definierte Reihenfolge.

Set<E>

Veränderliche Objekte

- **Besondere Vorsicht ist geboten, wenn Objekte, die in einem Set gespeichert sind, verändert werden:**
  - Objekte in einem Set müssen einzigartig sein!
  - Beispiel zur Sabotage:
    - Wir fügen zwei ungleiche Objekte zu einem leeren Set hinzu.
    - Danach verändern wir eines der Objekte so, dass `a.equals(b)==true` wird.
    - Das Set ist nun undefiniert. Derartige Zustände, z.B. aufgrund von Programmfehlern, müssen verhindert werden!

Set<E>

Implementierungen

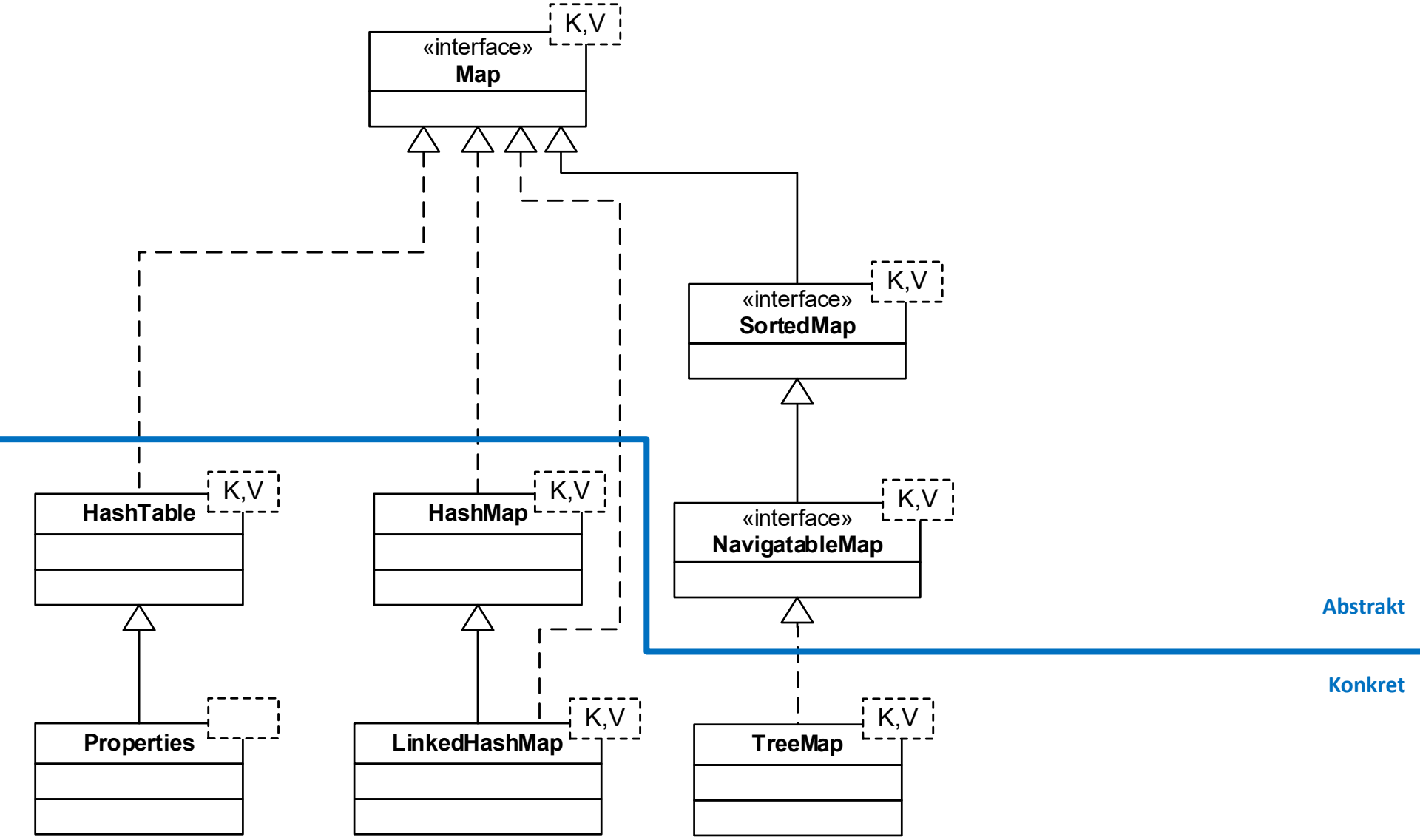
- Klasse `HashSet<E>`:
  - Implementiert die Menge durch eine Hash-Tabelle (VL11)
  - Vor jedem Einfügen wird geprüft, ob das einzufügende Element bereits in der Hash-Tabelle enthalten ist.
- Klasse `LinkedHashSet<E>`:
  - Implementierung als Hash-Tabelle mit zusätzlicher Verkettung der Elemente in der Reihenfolge, in der sie eingefügt wurden
- Klasse `TreeSet<E>`:
  - Implementierung als Rot-Schwarz-Baum (verwandt mit AVL-Bäumen und B-Bäumen der Ordnung $m=2$)
  - Schnelle Suche, Elemente werden sortiert gespeichert
  - `null` nicht erlaubt

Collections

Map<K,V>

Collections

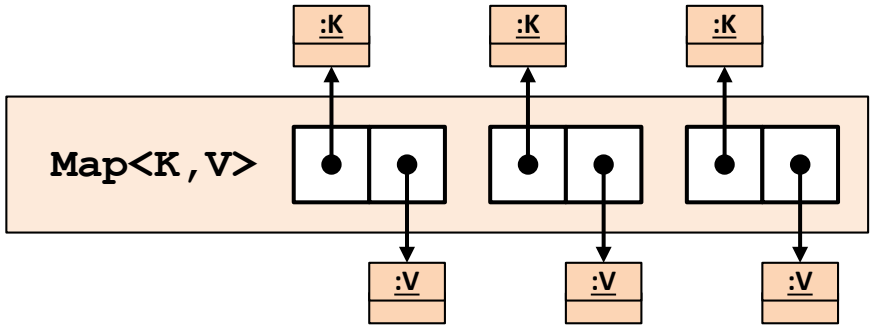
Map-Interface



Map<K,V>

Eigenschaften

- Eine Collection des Typs `Map<K,V>` realisiert einen assoziativen Speicher, der Schlüssel (**keys**) vom Typ `K` auf Werte (**values**) vom Typ `V` abbildet. Eine solche Struktur wird in der Informatik auch als Wörterbuch bezeichnet:



- Je Schlüssel gibt es entweder keinen oder genau einen Eintrag in der Map.
- Das Interface `Map<K,V>` ist nicht von `Collection<E>` abgeleitet:
  - Die Methoden `size`, `isEmpty` und `clear` (Entfernen aller Einträge) entsprechen jedoch den gleichnamigen Methoden aus `Collection<E>`.

Map<K,V>

Sichten

- Im Vergleich zum `Collection<E>`-Interface fällt auf, dass eine `Map<K,V>` keine Methode `iterator()` besitzt, um einen Iterator für den sequentiellen Durchlauf zu erzeugen.
- Stattdessen kann sie drei unterschiedliche Collections (**Sichten**) erzeugen, die dann über Iteratoren verfügen:
  - `Set<K> keySet()`
    Liefert die (mathematische) Menge aller Schlüssel zurück.
  - `Collection<V> values()`
    Liefert alle Werte der Map zurück.
  - `Set<Map.Entry<K,V>> entrySet()`
    Liefert alle Schlüssel-Wert-Paare der Map als Menge zurück.

Map<K,V>

Implementierungen

- Klasse `HashMap<K,V>`:
  - Implementiert die Map durch eine Hash-Tabelle (VL11)
  - Es ist erlaubt, `null` als Wert einzutragen.
  - Die Kollisionsauflösung erfolgt auch hier durch Verkettung.
  - Die Methoden sind nicht synchronisiert.
- Klasse `LinkedHashMap<K,V>`:
  - Implementierung als Hash-Tabelle mit zusätzlicher Verkettung der Elemente in der Reihenfolge, in der sie eingefügt wurden
- Klasse `TreeMap<K,V>`:
  - Implementierung als Rot-Schwarz-Baum
  - Schlüssel-Wert-Paare werden nach Schlüssel sortiert abgespeichert.

Lernziele

- Sie können entscheiden, ob es sich bei einem gegebenen Baum um einen B-Baum handelt oder nicht
- Sie sind in der Lage, aus einer Folge von Schlüsseln einen B-Baum zu konstruieren
- Sie kennen die Verfahren zum Löschen von Schlüsseln in einem B-Baum und können diese auf Beispiele anwenden
- Sie können begründen, warum Sie sich in der Praxis für einen AVL-Baum oder B-Baum entscheiden würden
- Sie können Problemstellungen für B-Bäume durch Bewegung in oder das Traversieren von B-Bäumen lösen und die Lösung in Java implementieren
- Sie kennen die charakteristischen Eigenschaften von Sets und Maps in Java
- Sie können für eine gegebene Problemstellung entscheiden und begründen, mit welchen Listen, Sets oder Maps Sie dieses Problem lösen würden

Literatur

- Quellen:
  - Ottmann, T. und Widmayer, P.: Algorithmen und Datenstrukturen (Kapitel 5.2: Balancierte Binärbäume)
  - Saake, G.: Algorithmen und Datenstrukturen (Kapitel 14.4: Ausgeglichene Bäume)