# Algorithmen und Datenstrukturen

*Praktische Einführung und Programmierung*

Stefan Bosse

Universität Koblenz - FB Informatik

# Bäume

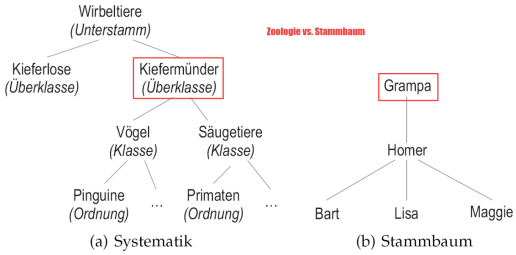
- Übergang von linearen Listen auf Baumstrukturen; Skip-Listen als "Vermittler"
- Vor- und Nachteile
- Algorithmen
- Anwendungen

# Motivation

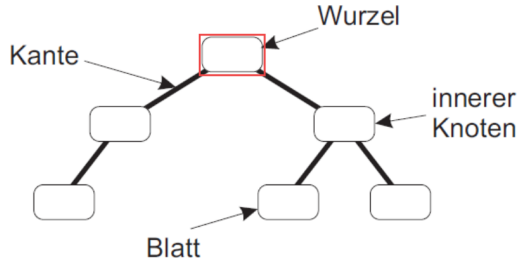
- Die bisher betrachteten Datenstrukturen waren im Wesentlichen **eindimensional oder linear**: sie weisen mit der Vorgänger-Nachfolger-Relation nur eine Form von Beziehungen zwischen den einzelnen Elementen auf.
    - Das hat Auswirkungen auf die Algorithmik und deren Komplexitätsklassen, vor allem die Suche mit O(N).
- In diesem Kapitel wollen wir mit den Bäumen eine Klasse von Datenstrukturen kennen lernen, die eine **zweite Dimension** einbeziehen und dadurch auch die Darstellung von hierarchischen Strukturen ermöglichen.
- Baumstrukturen sind nicht nur in der Informatik von fundamentaler Bedeutung, etwa zur Organisation von Daten, sondern sind auch im Alltag in vielfältiger Form wiederzufinden.

# Taxonomie

## Allgemeines Beispiel




## Formales Modell




- Bäume sind hierarchische Strukturierungshilfsmittel oder rin Organisationsprinzip für Daten, Klassen, Fakten.
- Ein typisches Beispiel ist etwa der Stammbaum einer Familie, der die Vererbungsbeziehungen darstellt, oder die Systematik in der Biologie

# Taxonomie

Allgemein können wir unter einem Baum eine Menge von Knoten $\mathbb{N}$ (Nodes) und Kanten $\mathbb{E}$ (Edges) verstehen, die besondere Eigenschaften aufweisen:

- So besitzt jeder Baum genau einen ausgezeichneten Knoten, der als Wurzel bezeichnet wird.
- Weiterhin ist jeder Knoten außer der Wurzel durch genau eine Kante mit seinem Vorgängerknoten verbunden.
- Jeder Knoten $n_{\mathrm{a}}$ kann $0..k$ Nachfolgerknoten $u$ besitzen die durch Kanten $k_{\mathrm{a,u}}$ verbunden sind.
- Die Kanten sind (zunächst noch) gerichtet ("von oben nach unten")
- Ein Knoten oder der Baum haben eine Ordnung $d$: Die maximale Anzahl von Kanten pro Knoten.

# Taxonomie

[https://ac.informatik.uni-freiburg.de/lak_teaching/ss_06/info2/Slides/18_Baeume_Grundlagen_Natuerliche_Suchbaeume.pdf]

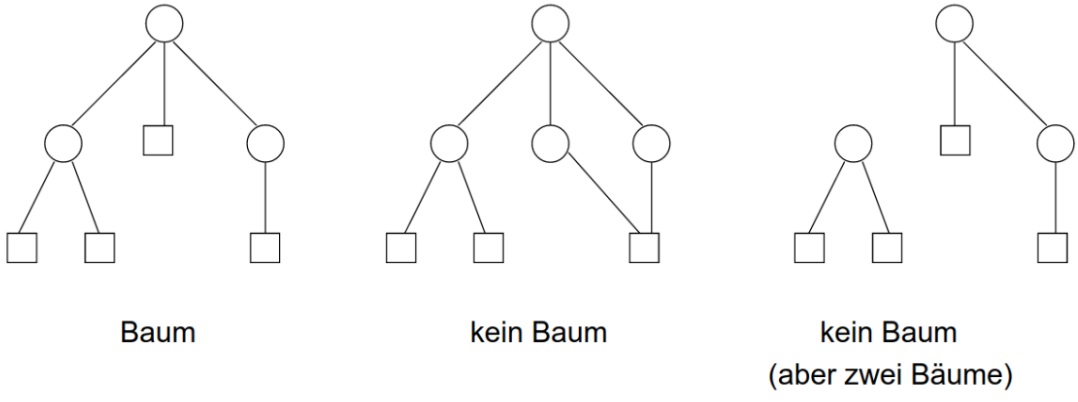



Abb. 1. Nicht jede Struktur ist ein Baum (aber ein Graph)

# Taxonomie

- Ein Knoten ohne Nachfolger kann als **Blatt**(knoten) bezeichnet werden.
- Ein **Pfad** in einem Baum ist eine Folge von unterschiedlichen Knoten, in der die aufeinander folgenden Knoten durch Kanten miteinander verbunden sind.
- Mithilfe des Pfades lässt sich auch die grundlegende Eigenschaft eines Baumes definieren: Zwischen jedem Knoten und der Wurzel gibt es genau einen Pfad. Dies bedeutet, dass:

1. ein Baum zusammenhängend ist und
2. es keine Zyklen gibt.

Weitere Eigenschaften (Parameter) von Bäumen: Höhe und Niveau.

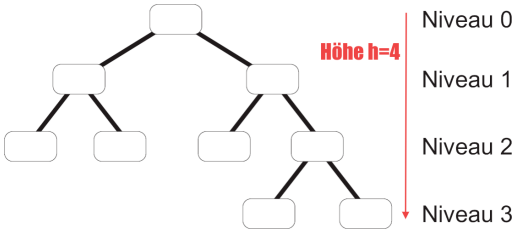
- Unter dem **Niveau** eines Knotens versteht man die Länge des Pfades von der Wurzel zu diesem Knoten. Die **Höhe** eines Baumes entspricht dem größtem Niveau eines Blattes plus 1.

# Taxonomie

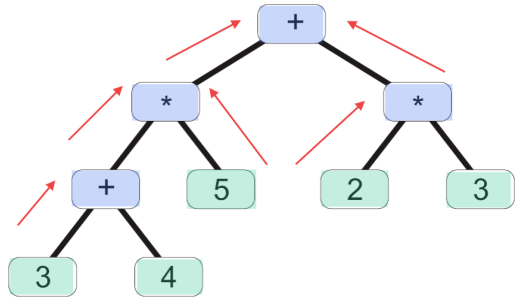
- Es muss im Gegensatz zu lineare Listen Entscheidungsfunktionen *f* geben, die einen Übergang von Knoten $n_a$ auf einen Nachfolgerknoten $n_b$ berechenn.
  - Anders ausgedrückt: Jeder Knoten muss mit Daten verknüpft sein die entscheiden welche Kante als nächstes ausgewählt wird.
  - Ein Knoten kann mit einer Variable $x_i$ verknüpft sein, eine Kante (a,b) die zwei Knoten $n_a$ mit $n_b$ verbindet mit eine Aktivierungsfunktion $f(x_i)_{a,b}: n_a \rightarrow n_b$
  - Beispiel: Jeder Knoten ist mit dem Alter eines Menschen verknüpft, und die Kanten bestimmen Altersanschnitte.
  - Ein Stammbaum hat keine Kantenfunktionen!!!
- Ein Baum ist neben einer Beziehungsrelation (Stammbaum) eine **Suchmaschine**.
  - Die Knoten können mit den Ergebnisdaten verknüpft sein, müssen aber nicht.
  - Es kann Blätter mit den Ergebnisdaten geben (Blattsuchbaum), oder auch nicht.

# Taxonomie

## Höhe und Niveau von Bäumen




## Bottom-up Berechnungsbaum




- Die Richtung von Bäumen muss nicht immer top-down sein. Ein Berechungsbaum für arithmetische Ausdrücke fängt bei den Blättern an und liefert ein Ergebnis im Wurzelknoten (Bottom-up).

# Taxonomie

- Ist mit jedem Knoten ein Datensatz verbunden $D$ und ein Schlüssel $k$ verbunden, so spricht man von einem **Suchbaum**.
- Die Kanten bestimmen Bereiche (Intervalle) von Schlüsseln $[k_1,k_2]$

Jeder Baum kann in Teilbäume zerlegt werden. Der atomare Baum besteht aus einem Knoten. Bäume sind rekursive Datenstrukturen. Unterbäume sind auch Bäume.

- Hat man einen Bereichsbaum so sind die Datensätze nach ihrem Schlüssel sortiert, aber nicht mehr linear, sondern in Teilbäumen. Jeder Knoten ist mit einem Intervall $[k_1,k_2]$ verknüpft.

# Taxonomie

- In Such- und Bereichsbäumen werden die Kanten mit einer Relation zum aktuellen Knotenschlüssel $k_i$ ausgewählt. Z.B.
  - Linke Kante wenn $k<l$, Rechte Kante wenn $k>l$ ⇒ **Binärbaum**

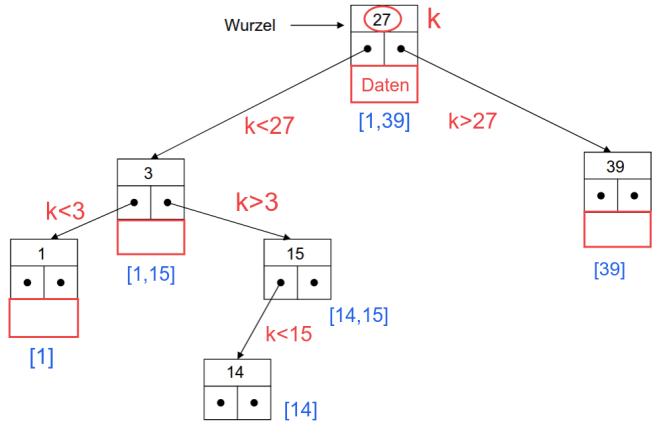



Abb. 2. Suche nach Schlüssel *s* endet beim Suchbaum in Knoten *k* mit *k*.key == *s* oder in Blatt, dessen Intervall *s* enthält.

# Taxonomie

Zusammenfassung:

> Baumstrukturen dienen im Wesentlichen zur hierarchischen Repräsentation und Organisation von Daten. Eine der wichtigsten Einsatzgebiete von Bäumen ist jedoch die Unterstützung einer effizienten Suche. Hierfür werden in den einzelnen Knoten eines Baumes neben den eigentlichen Nutzdaten (bzw. einen Verweis darauf) zusätzlich noch Schlüsselwerte gespeichert, über die die Suche erfolgt.

# Abstrakter Datentyp Binärer Baum

- Da der ADT Tree parametrisierbar bezüglich des Typs der Elemente sein soll, wird in der folgenden Spezifikation wieder ein Sortenparameter T verwendet.
  - Die entscheidende Funktion ist bin, die einen binären Baum aus einem linken und einem rechten Teilbaum sowie einem Element des Datentyps T konstruiert.
  - Für einen gegebenen Baum liefern daher left und right den linken bzw. rechten Teilbaum, während value das Wurzelelement dieses Baumes ermittelt.

Für einen aus den Teilbäumen x und y sowie dem Element b konstruierten Baum bin(x,b,y) liefert left demnach den Teilbaum x, right den Teilbaum y und value das Element b. Die beiden weiteren Funktionen empty und is_empty werden zur Konstruktion eines leeren Baumes bzw. zum Test auf einen leeren Baum benötigt.

# Abstrakter Datentyp Binärer Baum

```
type Tree(T)
import Bool
operators
  empty : → Tree
  bin : Tree × T × Tree → Tree
  left : Tree → Tree
  right : Tree → Tree
  value : Tree → T
  is_empty : Tree → Bool
axioms ∀x,y: Tree, ∀b: T
  left (bin (x, b, y)) = x
  right (bin (x, b, y)) = y
  value (bin (x, b, y)) = b
  is_empty (empty) = true
  is_empty (bin (x, b, y)) = false
```

Def. 1. ADT Tree(T) mit Operatoren und Axiomen die das Verhalten der Operationen definieren

# Datenstrukturen

## Suchbaum

Knoten sind mit Daten und Schlüssel verknüpft

```
typeof root = node
typeof node = {
  left  : node,
  right : node,
  data  : data,
  key   : number
}
```

## Blattsuchbaum

Knoten sind nur mit Schlüsseln verknüpft, d.h. innere Knoten sind Wegweiser, die Daten sind nur in den Blättern

```
typeof root = node
typeof node = {       leaf = {
  left  : node|leaf,    key  : number,
  right : node|leaf,    data : data
  key   : number      }
}
```

# Algorithmen

Es gibt wieder den bisher bekannten Satz an **Basisoperationen** die auf Bäume angewendet werden:

`search(k)`
: Suche nach einem Datensatz mit dem Schlüssel *k*

`insert(node | ⟨data,k⟩)`
: Einfügen eines neues Datensatzes mit dem Schlüssel *k* oder eine bereits erstellten Knotens.

`remove(node|data|k)`
: Entfernen eines eines Knotens mit dem Schlüssel *k*, Suche nach dem Datensatz, Schlüssel, oder Knoten.

# Algorithmen

## Suche (Binärbaum)

```
function Node(data,k) { return { left:null,right:null,data:data,key:k }}
function search (node,k) {
  if (node.key==k) return node
  if (node.left && node.left.key<k) return search(node.left,k)
  if (node.right && node.right.key<k) return search(node.right,k)
  throw NOTFOUND
}
search(root,100)
```

Alg. 1. Suche in einem Bereichsbaum nach dem Schlüüsel *k*

# Algorithmen

## Suche (k-stelliger Baum)

- Hier gibt es mehr als zwei Nachfolger. Jeder Nachfolgeknoten ist ein Teilbaum mit einem Intervallbereich $[k_1,k_2]$. Dieses Intervall wird in einem Knoten als *lowerBound* und *upperBound* in *keyRange* gespeichert.

```
function node(data,k) { return { children:[],data:data,key:k,keyRange:[k,k] }}
function search (node,k) {
  if (node.key==k) return node
  for(i=0;i<length(node.children);i++) {
    if (node.children[i].keyRange[0]>=k && node.children[i].keyRange[1]<=k)
      return search(node.children[i],k)
  }
  throw NOTFOUND
}
```

Alg. 2. Suche in einem Bereichsbaum nach dem Schlüüsel *k*

# Algorithmen

## Strategien zur Traversierung

Es gibt verschiedene Strategien zur Traversierung (für Suche, Einfügen, Löschen von Knoten) eines Baums.

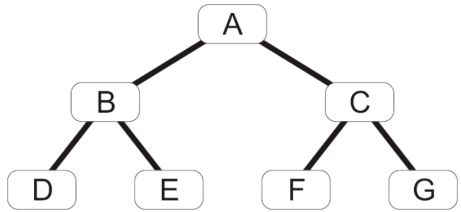



Abb. 3. Ein Beispielsbaum mit den Knoten A,B,C,D,E,F und G

### Inorder-Durchlauf

Hier wird zuerst rekursiv der linke Teilbaum besucht, dann der Knoten selbst und anschließend der rechte Teilbaum. Für den binären Baum aus der Abbildung entspricht das der Reihenfolge:

### Inorder-Durchlauf

D → B → E → A → F → C → G

Ein Beispiel für die Anwendung dieser Strategie ist das Auslesen eines Baumes für einen arithmetischen Ausdruck in Infixschreibweise.

Der Algorithmus für einen Inorder-Durchlauf lässt sich am einfachsten rekursiv formulieren. Der gesamte Baum wird nach der Inorder-Strategie durchlaufen, wenn der Algorithmus mit der Wurzel des Baumes als Parameter aufgerufen wird.

```
function Inorder (k) {
  // Eingabe: Knoten k eines binären Baumes mit Verweis auf
  // linken (k.left) und rechten (k.right) Teilbaum sowie
  // dem Element k seleber (Daten data und Schlüssel k)

  Inorder (k.left)  /* besuche linken Teilbaum */
  Process (k.data,k.key)  /* Verarbeite aktuellen Knoten */
  Inorder (k.right) /* besuche rechten Teilbaum */
}
```

Alg. 3. Inorder-Traversierung

**Preorder-Durchlauf**

Bei dieser Strategie wird zuerst der Knoten selbst besucht und erst danach erfolgt das Traversieren des linken bzw. rechten Teilbaumes. Für den Beispielbaum liefert das die Reihenfolge:

`A → B → D → E → C → F → G`

Neben der Ausgabe eines arithmetischen Ausdrucks in Präfixschreibweise ist die Erzeugung einer eingerückten Baumdarstellung für ein Dateisystem ein typisches Anwendungsbeispiel. Hier wird zuerst der Name des Verzeichnisses ausgegeben und darunter eingerückt eine Liste der darin enthaltenen Dateien und Verzeichnisse.

Der Algorithmus für den Preorder-Durchlauf kann wiederum rekursiv formuliert werden

## Preorder-Durchlauf

```
function Preorder (k) {
  // Eingabe: Knoten k eines binären Baumes mit Verweis auf
  // linken (k.left) und rechten (k.right) Teilbaum sowie
  // dem Element k seleber (Daten data und Schlüssel k)

  Process  (k.data,k.key)  /* Verarbeite aktuellen Knoten */
  Preorder (k.left)  /* besuche linken Teilbaum */
  Preorder (k.right) /* besuche rechten Teilbaum */
}
```

Alg. 4. Preorder-Traversierung

**Postorder-Durchlauf**

Beim Postorder-Durchlauf werden erst beide Teilbäume durchlaufen, bevor der Knoten selbst besucht wird. Dies führt in unserem Beispiel zur Reihenfolge

`D → E → B → F → G → C → A`

Ein Beispiel für die Anwendung dieser Strategie ist die Erstellung einer Stückliste mit Aufsummierung, etwa beim UNIX-Kommando du (disk usage), das den Platzbedarf eines Verzeichnisses und aller Unterverzeichnisse im Dateisystem ermittelt. Hierbei werden zuerst die Größen der Einträge in den Blättern (d.h. der Dateien) bestimmt und für den direkten Vaterknoten (d.h. das enthaltende Verzeichnis) addiert. Der Platzbedarf dieser Verzeichnisse wird wiederum zum Platzbedarf des übergeordneten Verzeichnisses zusammengefasst usw., bis die Gesamtgröße der Wurzel bestimmt werden kann.

Wie bereits bei den anderen beiden Strategien wird der Algorithmus zum Postorder-Durchlauf am einfachsten rekursiv formuliert

## Postorder-Durchlauf

```
function Postorder (k) {
  // Eingabe: Knoten k eines binären Baumes mit Verweis auf
  // linken (k.left) und rechten (k.right) Teilbaum sowie
  // dem Element k seleber (Daten data und Schlüssel k)

  Postorder (k.left)  /* besuche linken Teilbaum */
  Postorder (k.right) /* besuche rechten Teilbaum */
  Process   (k.data,k.key)  /* Verarbeite aktuellen Knoten */
}
```

Alg. 5. Postorder-Traversierung

### Levelorder-Durchlauf

Während bei den ersten drei Strategien der Baum zuerst in der Tiefe durchwandert wird, entspricht der Levelorder-Durchlauf einer Breitensuche, d.h., auf jedem Niveau des Baumes werden erst alle Knoten besucht, bevor auf das nächste Niveau gewechselt wird. Danach ergibt sich die Reihenfolge:

A → B → C → D → E → F → G

Der Algorithmus zum Levelorder-Durchlauf ist im Gegensatz zu den bisher betrachteten Strategien nicht rekursiv. Stattdessen wird eine Warteschlange verwendet, in der die noch zu verarbeitenden Knoten abgelegt werden. Der Baum wird durchlaufen, indem der nächste zu besuchende Knoten aus der Warteschlange entnommen wird. Nach dem Besuch dieses Knotens werden die Kinder am Ende der Warteschlange eingetragen, bevor mit dem nächsten Knoten fortgefahren wird.

Die Levelorder-Strategie wird iterativ implementiert.

## Levelorder-Durchlauf

```
function Levelorder (k) {
  // Eingabe: Knoten k eines binären Baumes

  q = [] // leere Warteschlange;
  addTail (q, k) /* Wurzel in die Warteschlange aufnehmen */
  while (length(q)!=0) {
    n = removeHead(q);
    Process (n.data, n.key)
    addTail(q, n.left)  /* linken Knoten eintragen */
    addTail(q, n.right) /* rechten Knoten eintragen */
  }
}
```

Alg. 6. Levelorder-Traversierung

# Algorithmen

Wenn der Baum nicht geordnet oder die Knoten nicht mit Schlüssel assoziert sind (generischer Baum) müssen alle Teilbäume und Pfade durchsucht werden.

- Das klinkt sinnlos, da das Suchen scheinbar wieder einer lineare Liste entspricht.
- Aber bei **Parallelisierung** der Suche in Teilbäume (Divide-and-Conquer Methode) könnte dennoch eine geringere Suchzeit notwendig sein!

```
function parSearch(node,k) {
  if (node.key==k) terminateSearchWith(node)
  startProcess(parSearch,node.left,k)
  startProcess(parSearch,node.right,k)
}
parSearch(root,k)
```

Alg. 7. Parallele Suche. Es werden bis zu $N$-1-$L$ Prozesse gestartet, bei $N$ Knoten und $L$ Endknoten ohne Nachfolger.

# Algorithmen

Bäume sind daher selber Organisationsstrukturen für Algorithmen die einen Programmablauf steuern können!

- Z.B. auch Suche im Internet (Suchmaschine!)
- Verteilung von Berechnungen auf Rechner mittels hierarchischer Teilung
- Baumstrukturen stellen verteilte Kommunikationsnetze dar (Verbindung von Rechnern), die Kanten sind Kommunikationskanäle.
- In Betriebssystemen (UNIX) können Prozesse weietere Prozesse starten. Die neuen Prozesse werden Kindprozesse des Elternprozesses ⇒ Baumstruktur!
- Dateisysteme sind Baumstrukturen

# Algorithmen

- Höhe eines symmetrischen ideal besetzten Binärbaums: H=O(log N).
- Die Baum-Struktur hängt von Einfügereihenfolge in anfangs leeren Baum ab
  - Höhe kann linear zunehmen, sie kann aber auch in O(log N) sein, genau $\lceil \log_2 (N+1) \rceil$.

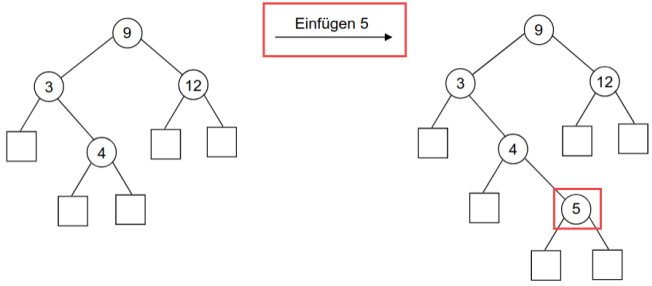



Abb. 4. Einfügen eines neues Elements (Schlüssel *k*=5) in einen bestehenden Baum verändert die Höhe des Baums und dessen Symmetrie

# Algorithmen

## Einfügen

- Das Einfügen in einem geordneten Baum hängt von seiner Stelligkeit (Ordnung) ab.
  - Bei Binärbäumen sind häufig schon alle Kanten belegt, und der neue Knoten muss am Ende eingefügt werden
  - Bei k-stelligen Bäumen könnten noch freie Kanten für das Einhängen des neuen Knoten vorhanden sein, sofern die Ordnungsrelation noch gegeben ist (hie rnicht trival zu bestimmen).

## Einfügen (Binärbaum)

- Gegeben ist ein neuer Datensatz *D* mit dem Schlüssel *k*
- Es muss zuerst die korrekte Einfügeposition gefunden werden, so dass die Ordnung der Elemente nicht verletzt wird.
- Diese Position muss ein Knoten sein, dessen Schlüsselwert größer als der des einzufügenden Elementes ist und der noch keinen linken Nachfolger hat oder einen Knoten mit einem kleineren Schlüsselwert und einen freien Platz für den rechten Nachfolger. Dabei sind im Prinzip zwei Fälle zu unterscheiden:

1. Der Baum ist leer, d.h., der einzufügende Knoten wird die neue Wurzel.
2. Es gibt bereits Knoten im Baum. In diesem Fall ist der Knoten zu identifizieren, der Elternknoten des neuen Elementes werden soll.

# Einfügen (Binärbaum)

```
function Node(data,k) { return { left:null,right:null,data:data,key:k }}
function insert(node,data,k) {
  if (node.key==k) throw Error;
  if (k>node.key) {
    if (!node.right) node.right=Node(data,k)
    else insert(node.right,data,k)
  } else {
    if (!node.left) node.left=Node(data,k)
    else insert(node.left,data,k)
  }
}
if (!root) root=Node(data,k)
else insert(root,data,k)
```

Alg. 8. Einfügen eines neuen Datensatzs in einen Binärbaum

Hier wird die Preorder Traversierung angewendet.

# Einfügen (Binärbaum)

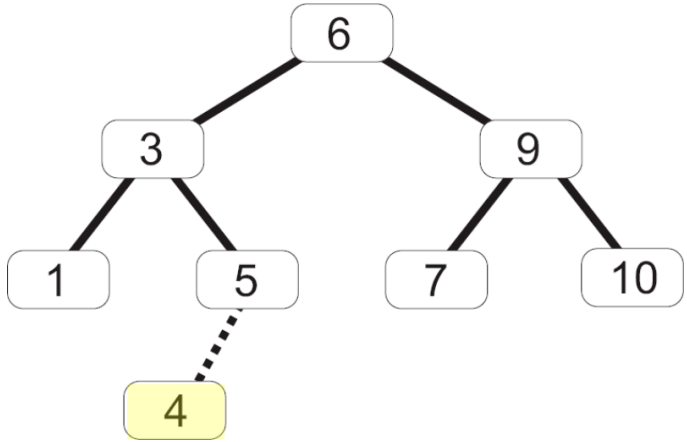



Abb. 5. Beispiel des Einfügens eines neuen Knotens mit dem Schlüssel k=4

# Algorithmen

## Entfernen eines Knotens

Die komplizierteste Operation im binären Suchbaum ist das Löschen. Dies liegt daran, dass beim Entfernen eines inneren Knotens einer der beiden Teilbäume des Knotens »hochgezogen« und gegebenenfalls umgeordnet werden muss.

Grundsätzlich müssen beim Löschen eines Knotens k drei Fälle berücksichtigt werden:

1. Der Knoten k ist ein Blatt. Dieser Fall ist der einfachste, da hier nur der Elternknoten zu bestimmen ist und dessen Verweis auf Knoten k entfernt werden muss.
2. Der Knoten k besitzt nur ein Kindknoten. In diesem Fall ist der Verweis vom Elternknoten auf den Kindknoten von k umzulenken.
3. Der Knoten k ist ein innerer Knoten mit zwei Kindknoten. Hierbei muss der Knoten durch den am weitesten links stehenden Knoten des rechten Teilbaumes ersetzt werden, da dieser in der Sortierreihenfolge der nächste Knoten ist. Alternativ kann auch der am weitestens rechts stehende Knoten des linken Teilbaumes verwendet werden.

## Entfernen eines Knotens

Für das Ersetzen eines Knotens gibt es wiederum zwei Varianten:

1. Entweder werden die Daten der Knoten ausgetauscht oder
2. es werden nur die Verweise auf die Knoten aktualisiert.

- Die erste Variante ist einfacher zu implementieren, die zweite Methode vermeidet ein unter Umständen aufwendiges Kopieren.

# Entfernen eines Knotens

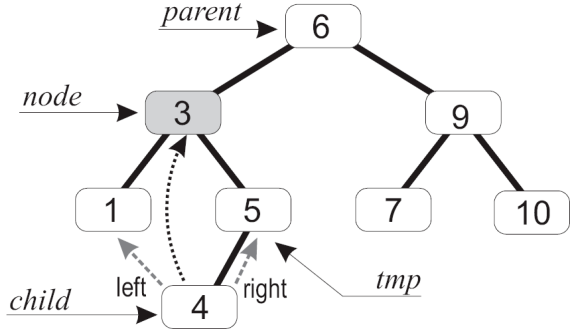
```
function removeNode(root,k) {
  // Eingabe: Wurzel vom Suchbaum T, Schlüssel k des zu löschenden Elementes
  node = searchNode (root, k)
  if (node = null) throw Error
  parent = searchParent (root, node) // Arrg, sollte mit Search kombiniert werden
  /* 1. Fall */
  if (!node.left && !node.right) { // node ist Blattknoten
    if (parent.left===node) // node ist linkes Kind
      parent.left = null
    else
      parent.right = null
  /* 2. Fall */
  } else if (!node.left)  {
    if (parent.left===node)  // node ist linkes Kind
      parent.left = node.right
    else
      parent.right = node.right
  } else if (!node.right) {
    if (parent.left===node) // node ist linkes Kind
      parent.left  = node.left
    else
      parent.right = node.left
  } else /* 3. Fall */
    child = minElement (node.right)
    replaceNode(node,child) // Ersetze node durch child
  }
```

Alg. 9. Entfernen eines Knotens mit dem Schlüssel k

# Entfernen eines Knotens

```
function minElement(node) {
  if (!node.left && !node.right) return node;
  if (node.left) return minElement(node.left)
  else           return node
}
```

Alg. 10. Minimales Element eines Teilbaums




[asdp]

Abb. 6. Löschen im binären Suchbaum

# Komplexität

.. it depends.

- Die (erfolglose) Suchzeit als Anzahl der Knoteniterationen ist bei einem ausgeglichenen und symmetrischen Binärbaum einfach gleich der Höhe, und diese ist O(log N) bei N Knoten.
- Der Extremfall eines Baums (nicht ausgegelichen und asymmetrisch) ist die lineare Liste (jeder Baum kann in einer lineare Liste transformiert werden). Dann ist die Suchzeit wieder O(N)
- In der Realität hat man etwas dazwischen liegendes
- Die noch einzuführenden Operationen zum EInfügen und Entfernen von Knoten benötigen i.A. auch eine Suche. Für diese giltg die gleiche Komplexitäsklasse.

# Beispiele

## Abstrakter Syntaxbaum (AST)

... oder nur Syntaxbaum

Worum geht es? Um Compiler und Syntaxanalyse von Programmen die in einer bestimmten Programmiersrpache geschrieben wurden.

Ein Compiler ist typischweise aus folgenden Komponenten im Schichtenmodell aufgebaut:

Analysephase:

1. Ein Lexer, der einen Textzeichenstrom in eine **lineare Liste** von Tokens (Symbolen) und Werten transformiert.
2. Ein Parser, der diese lineare Liste aus Token zu einem (Abstrakten) Syntaxbaum wieder zusammensetzt.

3. Vereinfachung des AST (Konstantenfaltung), Kompaktierung
4. Erzeugung einer Zwischenrepräsentation (IR)
5. Optimierung
6. Erzeugung von Zielcode (Maschinencode, VM Bytecode oder andere Programmiersprache; Transpiler)

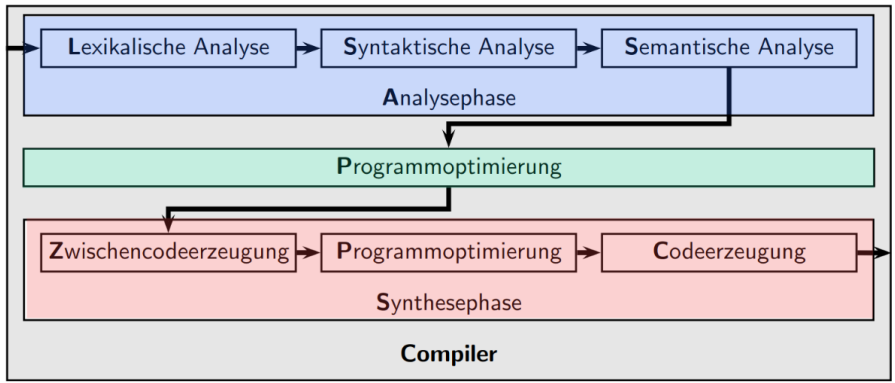



Abb. 7. Typisches Schichten- und Phasenmodell eines Compielrs

# Abstrakter Syntaxbaum (AST)

Man kann einen Baum auch als geschachtelte Liste (Array) von Listen verstehen.

Daher kann ein Tokenstrom (TOkenliste) mit bestimmten Regeln wieder in eine Baumstruktur transformiert werden.

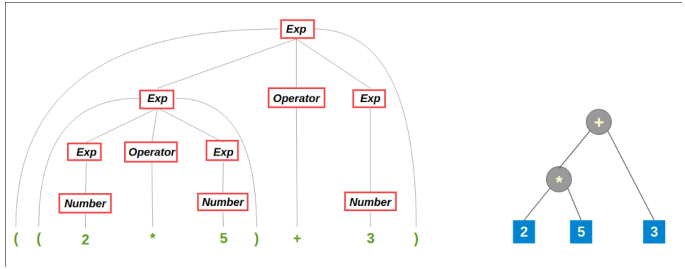



Abb. 8. Konkerter versa abstrakten Syntaxbaum

# Abstrakter Syntaxbaum (AST)

Beispiel: Arithmetische Berechnungen wie `(x*n+1)/y`

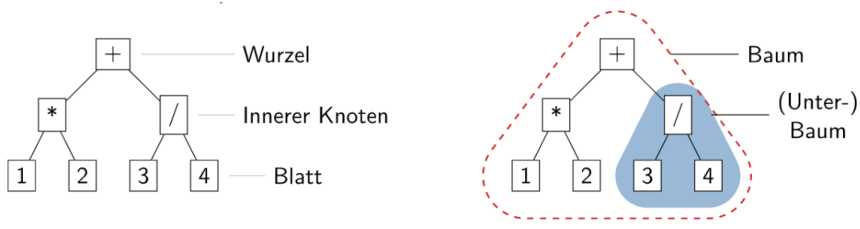
```
code   = "(x*n+1)/y"
tokens = lexer(code)
tokens := [LPAR,LIT(x),MUL,LIT(n),ADD,NUM(1),RPAR,DIV,LIT(y)]
tree   = parser(tokens)
tree   := [DIV,[ADD,[MUL,VAR(x),VAR(n)],[NUM(1)]]],[VAR(y)]]
```

Bsp. 1. Der Lexer und Parser im Zusammenspiel

# Abstrakter Syntaxbaum (AST)

Beim Abstrakten Syntaxbaum fehlen alle unwesentlichen, grammatikspezifischen Elemente:

- Der Abstrakte Syntaxbaum gibt die logische Struktur des Ausdrucks wieder
- Der Syntaxbaum kann einfach (rekursiv) interpretiert werden
- Deklarationen von Bezeichnern (Variablen, Typen, Funktionen) erfordern eine Symboltabelle
- Der Syntaxbaum kann Reihenfolgen und Bindungen bewahren, im Gegensatz zur linearen Liste




Bsp. 2. Strukturen und Bindungen im Syntaxbaum am Beispiel von Ausdrücken. Token werden hierarchisch geordnet, implizite (Bindungs- und Reihenfolge) Regeln werden explizit.

## Der Tokenstrom

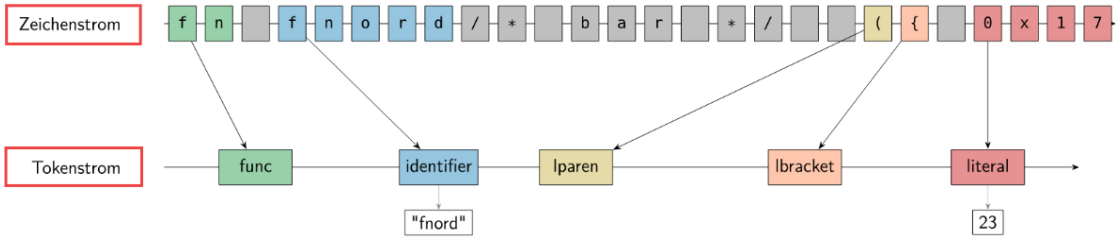
Der Tokenstrom (oder Liste):

- Gruppierung von Textzeichen zu sprachspezifischen Bausteinen
- Der Tokenstrom enthält keine Leerzeichen oder Kommentare
- Token haben eine Klasse und ggfs. eine "Nutzlast" (Funktionale, Argumente)
- Unterschiedliche Schreibweisen sind vereinheitlicht (0x18,1.2,1E-9,100)

Ziele der Abstraktion:

1. Komplexitätsreduktion (Es gibt immer weniger Token als Textzeichen)
2. Fokussierung aus Sprachelemente und erste lexikalische Syntaxprüfung
3. Vereinfachung des Parsers und des Parsens

# Der Tokenstrom




Bsp. 3. Beispiel der Ezeugung eines Tokenstroms

## Algorithmen auf AST

- Die Suche ist eher sekundär.
- Die Transformationen dieses Baumes ist primär und steht im Vordergrund
- Ein Beispiel für eine Transformation ist die Konstantenfaltunf, d.h. die Umstellung von Ausdrücken so dass Konstanten zusammengefasst und zur Kompilierungszeit evaluiert werden können.

```
x = (a+2)-(b+4)
  =  a+2 + (-b) + (-4)
  =  a-b + 2-4
  =  a-b - 2
```

Bsp. 4. Beispiel der Konstantenfaltung

- Ziel ist es einfache Elementaroperationen wiederholend auf Teilbäume anzuwenden bis keine weriter Vereinfachung eines Ausdricks möglich ist

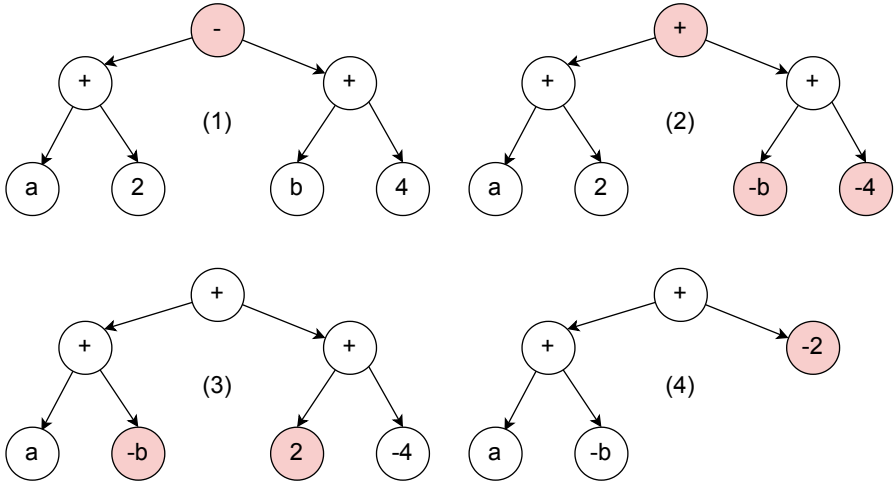
# Algorithmen auf AST

Das Problem bei der Konstantenfaltung: Die Konstanten können in verschiedenen Teilbäumen verteilt sein. Ein Transformation der Baumstruktur kann die Konstanten dann in einem Knoten zusammenbringen.

- Dazu müssen Auswerte- und Binderegeln beachtet werden.
- Eine Möglichkeit ist die Normalisierung von komplimentären Ausdrücken
  - Subtrkation in Addition transformieren
  - Division in Multiplikation transformieren
  - Dann Kommutativgesetz anwenden und Operanden / Knoten vertauschen

Wir werden weitere Transformationen von Bäumen kennen lernen (da vor allem zur Symmetrisierung), wie LR und RL Transformationen.

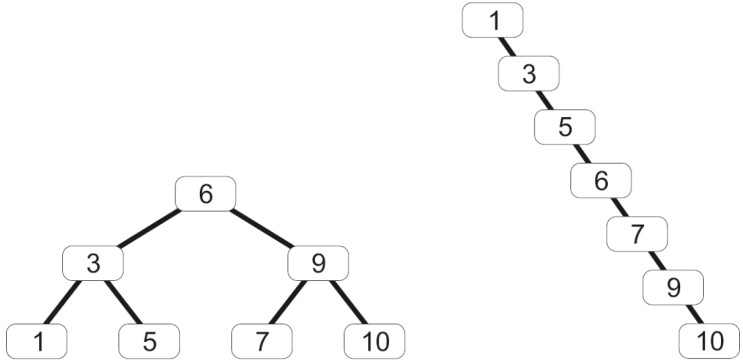
# Algorithmen auf AST




Bsp. 5. Konstantefaltung durch Baumtransformation mit Operationsnormalisierung

# Transformation von Bäumen

- Wir hatten bereits spezielle Transformationen von Syntaxbäumen kennen gelernt
- Jetzt geht es um allgemeine Transformationen
- Bei den Komplexitätsbetrachtungen fiel schon auf dass Bäume optimal balanziert sein müssen um O(log(N)) zu erreichen




(a) ausgeglichen (b) entartet

Abb. 9. »Gute« und »schlechte« Suchbäume

# Transformation von Bäumen

## Balanzierung von Bäumen

- Ziel ist die Balanzierung von Bäumen um "optimale" Bäume zu erhalten, also balanziert oder ausgeglichen.
- Neben vollständig balanzierten Bäumen gibt es schwächere Varianten wie AVL Bäume:
  - Beim balanzierten Baum ist der **Höhenunterschied benachbarter Teilbäume Null**
  - Bei einem AVL (Adelson-Velskij und Landis) Baum ist der **Höhenunterschied maximal 1**
  - B-Bäume besitzen eine ausgeglichene Höhe, lassen aber einen unausgeglichenen Verzweigungsgrad zu.

Für die Höhe h eines AVL-Baums mit n Knoten gilt:

$$h_{\text{bal}} \leq h \leq h_{\text{avl}}$$
$$\lfloor \log 2(n) \rfloor \leq h \leq 1.44 \cdot \log 2(n+2) - 1.33$$

# Balanzierung von Bäumen

[O. Bittel, HTWG Konstanz]

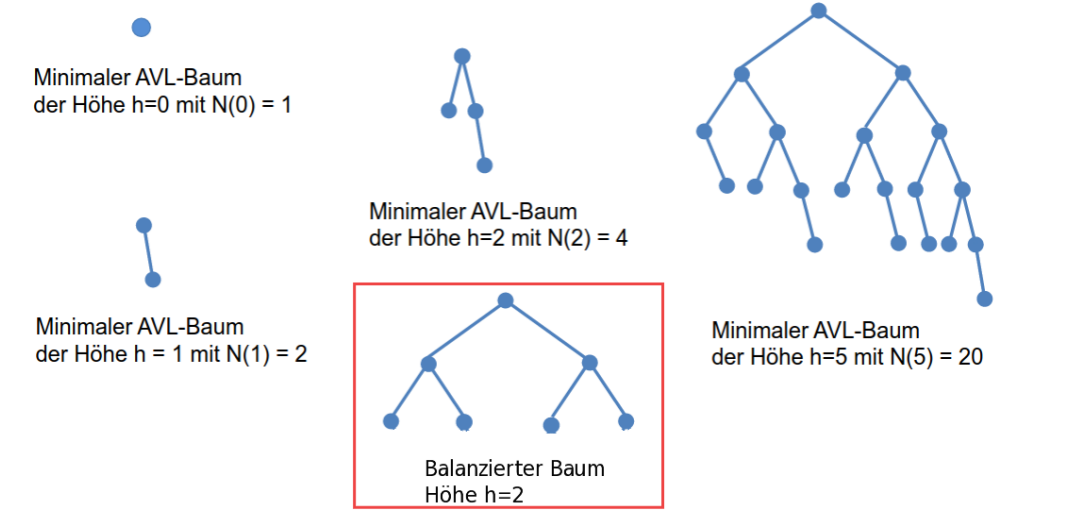



Abb. 10. Verschiedene AVL Bäume im Vergleich zu einem vollständig balanzierten Baum

# Balanzierung von Bäumen

Bei Binärbäumen geht die Balanzierung sehr einfach durch Vertauschen von Knoten, wie das bereits bei der Einfügeoperation zu sehen war.

[addp]

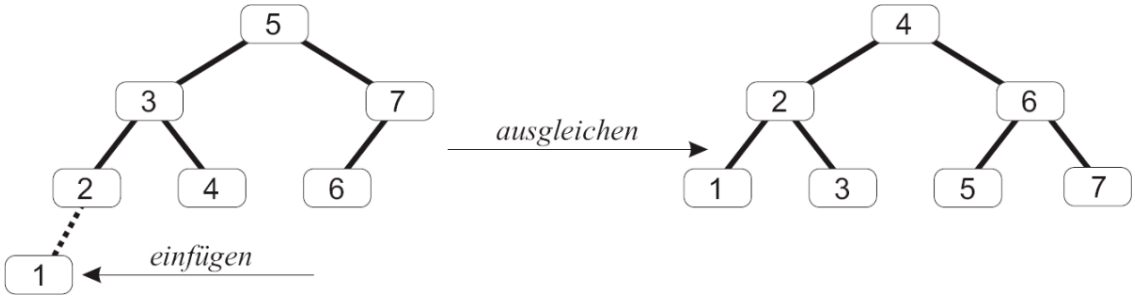



Abb. 11. Vollständiges Ausgleichen eines Baumes nach Einfügen eines neuen Knotens: Kein Knoten bleibt an seinem Platz! Eine teure Operation!

# Rechts-Rotation

- Fall A: Baum ist linkslastig, d.h. Höhenunterschied = -2
- Unterfall A1: linker Teilbaum hat Höhenunterschied -1 oder 0

[O. Bittel, HTWG Konstanz]

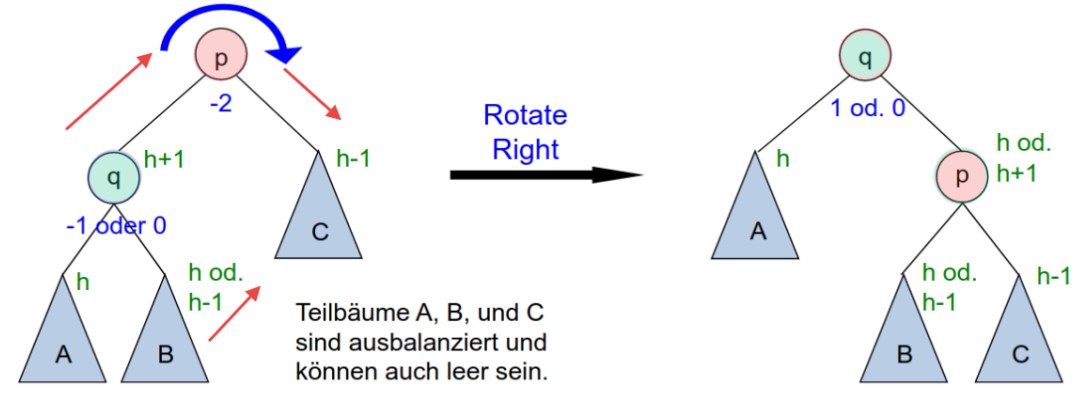



Abb. 12. Rechts-Rotation

# Rechts-Rotation

```
function Node(data,k) { return { left:null,right:null,data:data,key:k }}
function rotateRight(p) {
  q = p.left
  A = q.left
  B = q.right
  C = p.right
  pp = parentOf(p)
  q.left  = A
  q.right = q
  p.left  = B
  p.right = C
  if (pp.left === p) pp.left = q
  else               pp.right= q
}
```

Alg. 11. Rechts-Rotation mit Elternsuche von Rotationsknoten p

# Rechts-Rotation

```
function Node(data,k) { return { left:null,right:null,data:data,key:k }}
function swap(p,q) {
  data=p.data; key=p.key;
  p.data=q.data; p.key=q.key; q.data=data; q.key=key
}
function rotateRight(p) {
  q = p.left
  A = q.left
  B = q.right
  C = p.right
  // tauscht Daten und Key, damit wir keinen Elternknoten benötigen!!!
  // Die parent-p Relation bleibt erhalten
  swap(p,q)
  // p ist nun q, q ist nun p
  p.left  = A
  p.right = q
  q.left  = B
  q.right = C
}
```

Alg. 12. Rechts-Rotation: Optimiert ohne Elternsuche vom Rotationsknoten p

# Links-Rotation

Fall B: Baum ist rechtslastig, d.h. Höhenunterschied = +2

Unterfall B1: rechter Teilbaum hat Höhenunterschied 0 oder +1:

[O. Bittel, HTWG Konstanz]

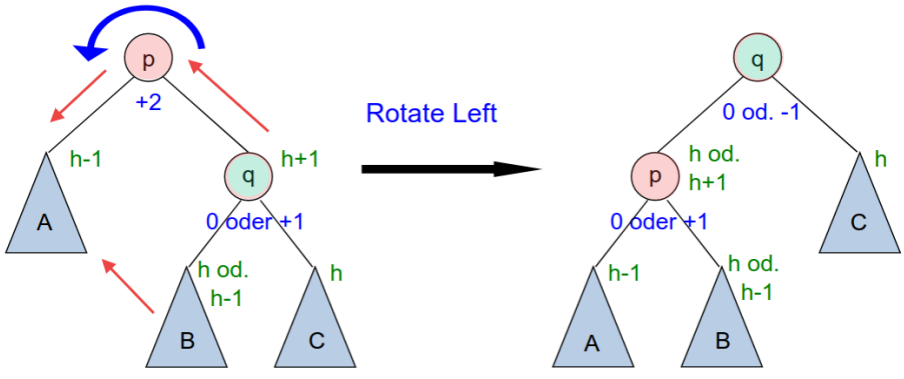



Abb. 13. Links-Rotation

# Links-Rotation

```
function Node(data,k) { return { left:null,right:null,data:data,key:k }}
function rotateLeft(p) {
  q = p.right
  A = p.left
  B = q.left
  C = q.right
  pp = parentOf(p)
  p.left  = A
  p.right = B
  q.left  = p
  q.right = C
  if (pp.left === p) pp.left = q
  else               pp.right= q
}
```

Alg. 13. Links-Rotation mit Elternsuche von Rotationsknoten p

# Rechts-Links-Rotation

Die Rechts- (genauso analog die Links-) Rotation verändert den Baum, aber ausgeglichen ist er nicht. Dazu wird eine Kombination aus Links- und Rechts-Rotation verwendet.

Fall B: Baum ist rechtslastig, d.h. Höhenunterschied = +2
Unterfall B2: rechter Teilbaum hat Höhenunterschied -1

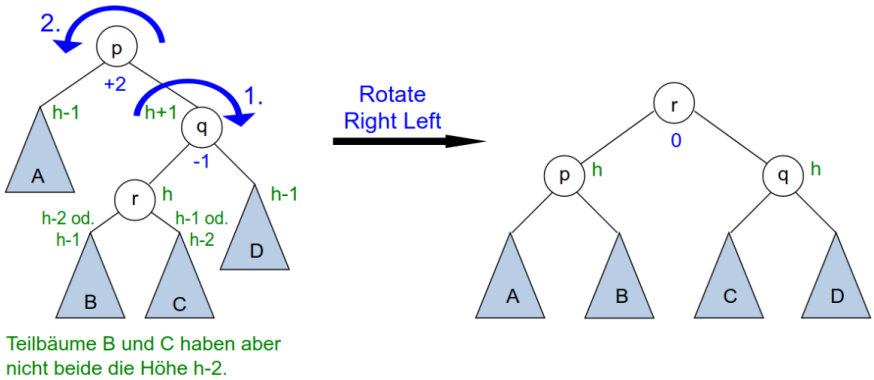



Abb. 14. Rechts-Links-Rotation

# Rechts-Links-Rotation

```
function Node(data,k) { return { left:null,right:null,data:data,key:k }}
function rotateLeft(p) {
  pp = parentOf(p)
  q = p.right
  r = q.left
  A = p.left  ; B = r.left
  C = r.right ; D = q.right
  p.left  = A ; p.right = B
  q.left  = C ; q.right = D
  r.left  = p ; r.right = q
  if (pp.left === p) pp.left = r
  else               pp.right= r
}
```

Alg. 14. Rechts-Links-Rotation mit Elternsuche von Rotationsknoten p

- Analog für die Links-Rechts-Rotation bei linkslastigen Baum (oder Teilbaum)

# Rotation

- Da der Baum vor dem Einfügen bzw. Löschen balanciert war, haben alle Teilbäume einen Höhenunterschied von -1, 0 oder +1.
- Damit können nach einem Einfüge- oder Löschschritt nur die Fälle A1, A2, B1 oder B2 auftreten.
- Es lässt sich zeigen, dass beim Einfügen insgesamt maximal eine der 4 Rotationsoperationen notwendig ist.
- Beim Löschen müssen i.a. mehrere Rotationsoperationen durchgeführt werden (siehe Beispiel später)
- Das Einfügen bedarf bei AVL Bäumen u.U. nur eine Links- oder Rechts-Rotation, bei vollständig balanzierten Bäumen aber i.A. Kombinationen!

# Einfügen und Rotation

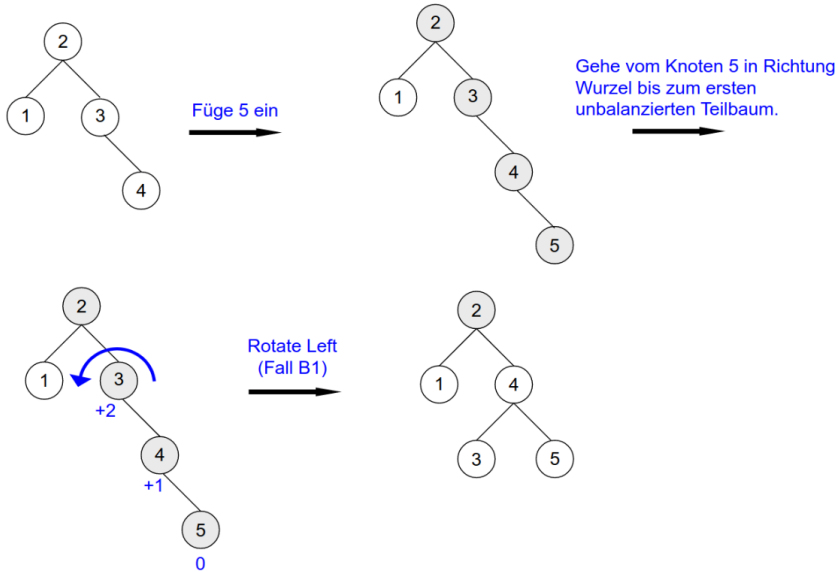



Abb. 15. Beispiel 1 zu Einfügen in AVL-Bäumen

# Einfügen und Rotation

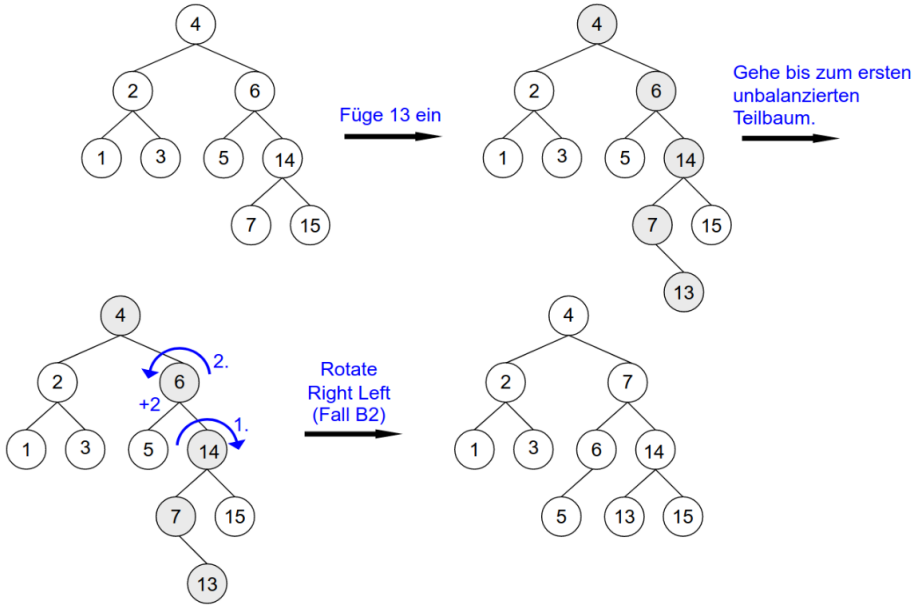



Abb. 16. Beispiel 2 zu Einfügen in AVL-Bäumen

# Balanzierung eines gesamten Baums

Ein iterativer und rekursiver Vorgang!

# B-Bäume

- AVL-B äume werden annähernd balanziert gehalten.
- Eine weitere Variante einer ausgeglichenen Baumstruktur ist der von R. Bayer und E. McCreight entwickelte B-Baum.
- Hierbei steht der Name »B« für balanciert, breit, buschig oder auch Bayer, nicht jedoch für binär.
    - Die Grundidee des B-Baumes ist es gerade, dass der Verzweigungsgrad variiert, während die Baumhöhe vollständig ausgeglichen ist.
- Den Ausgangspunkt bildet somit auch wieder ein ausgeglichener Suchbaum, bei dem alle Pfade von der Wurzel zu den Blättern gleich lang sind.
- Eine Variation des Verzweigungsgrades bedeutet nun, dass ein Knoten mehrere Elemente enthalten kann – wir sprechen daher von Mehrwegebäumen.

# B-Bäume

## Prinzip des B-Baumes

Wendet man das Kriterium der Ausgeglichenheit auf einen Mehrwegebaum an, so müssten folgende Forderungen für einen vollständig ausgeglichenen Baum erfüllt sein:

- Alle Wege von der Wurzel bis zu den Blättern sind gleich lang.
- Jeder Knoten besitzt gleich viele Einträge.

Das vollständige Ausgleichen in einem solchen Baum wäre jedoch zu aufwendig. Daher wird das B-Baum-Kriterium eingeführt:

- Jeder Knoten außer der Wurzel enthält zwischen m und 2m Schlüsselwerte.

# B-Bäume

- Im Kontext von B-Bäumen spricht man auch von **Seiten** anstelle von Knoten.
  - Seiten entsprechen dabei Speichereinheiten, die auf einem externen Medium verwaltet werden und mit einer Operation ein- bzw. ausgelagert werden können.
  - Daher sind B-Bäume besonders gut für Datenstrukturen geeignet, die aufgrund ihrer Größe nicht mehr im Hauptspeicher, sondern extern gespeichert werden.

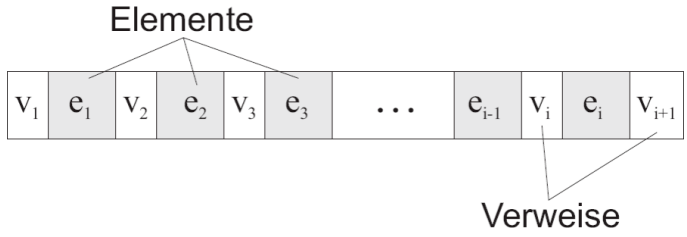



Abb. 17. Struktur einer Seite im B-Baum

- Die Höhe eines B-Baumes ergibt sich danach bei minimaler Füllung aus $\log_m n$.
- Gleichzeitig lassen sich somit n Datensätze mit $\log_m n$ Seitenzugriffen von der Wurzel zum Blatt lesen.

# B-Bäume

- Jede Seite enthält i geordnete Elemente oder Schlüsselwerte, wobei es einen Wert m gibt, so dass $m \leq i \leq 2m$ gilt.
  - Dieses m wird auch als Ordnung des B-Baumes bezeichnet.
  - Zusätzlich zu den Schlüsselwerten enthält jede Seite noch Verweise auf die Kindknoten mit den Unterbäumen, wobei für einen inneren Knoten jeweils i + 1 solcher Verweise vorhanden sind.

1. Jede Seite enthält höchstens 2m Elemente.
2. Jede Seite außer der Wurzelseite enthält mindestens m Elemente.
3. Jede Seite ist entweder eine Blattseite ohne Nachfolger oder hat i + 1 Nachfolger, falls i die Anzahl ihrer Elemente ist.
4. Alle Blattseiten liegen auf der gleichen Stufe.

Def. 2. Für einen B-Baum der Ordnung m gilt:

# B-Bäume

## Suchen im B-Baum

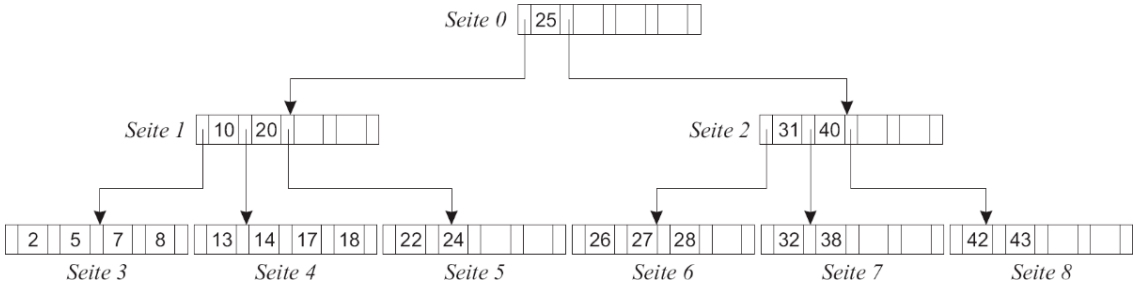



Abb. 18. Suchen im B-Baum

## Einfügen und Löchen

Bei den Einfüge- und Löschoperationen muss die Forderung nach mindestens m und maximal 2m Elementen pro Seite eingehalten werden.

# Bäume Live