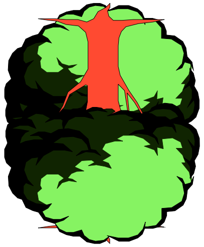
Kapitel 5
Bäume

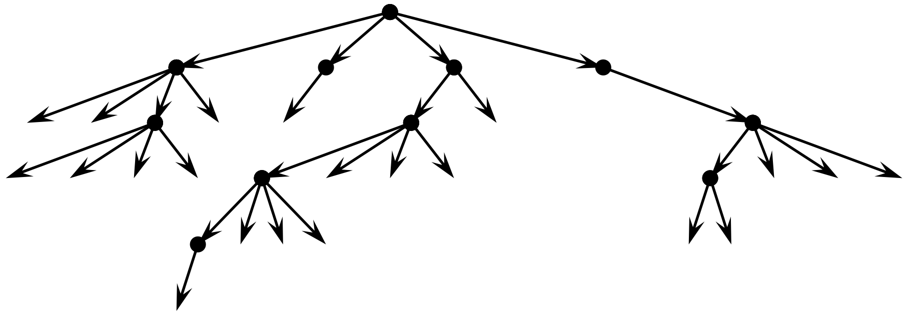
Ein **Baum** (tree) ist ein spezieller Graph, der eine hierarchische Struktur über eine Menge von Objekten definiert

Intuitives Vorstellungsmodell - Nicht-lineare Datenstruktur

Anwendungen

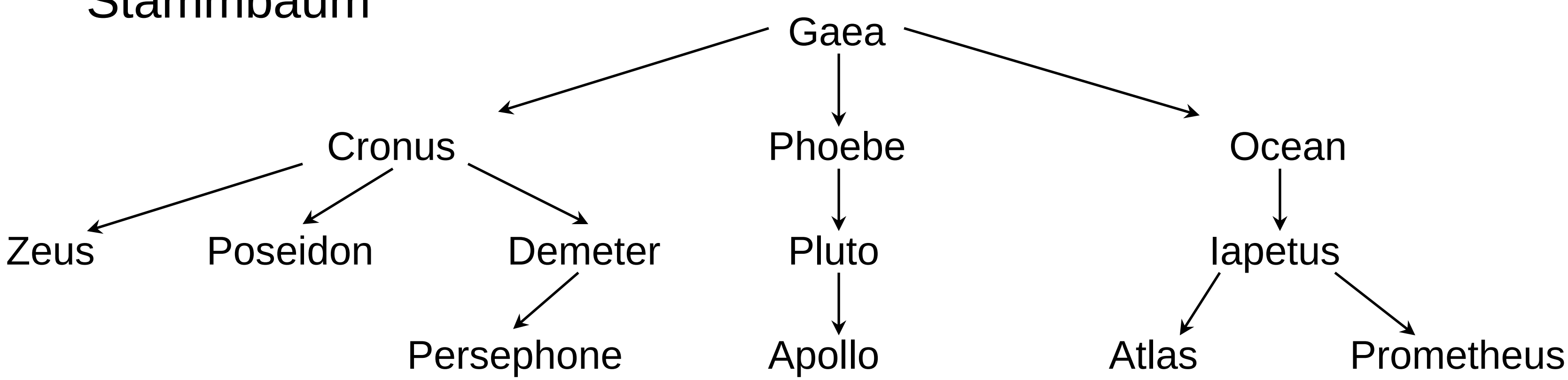
Repräsentieren Wissen, Darstellung von Strukturen, Abbildung von Zugriffspfaden, Analyse hierarchischer Zusammenhänge, ...





Prozessorfamilien

Stammbaum



Baum, Knoten, Kanten, Weg, Pfad

Ein **Baum** besteht aus einer Menge von **Knoten**, die durch **gerichtete Kanten** verbunden sind

Ein **Pfad** oder **Weg** ist eine Liste sich unterscheidender Knoten, wobei aufeinander folgende Knoten durch eine Kante verbunden sind

Definierende Eigenschaft eines Baumes

- Es gibt genau einen Pfad der 2 Knoten miteinander verbindet
- jeder Knoten hat nur einen direkten Vorgänger
- alle Vorgänger eines Knoten sind von ihm selbst verschieden

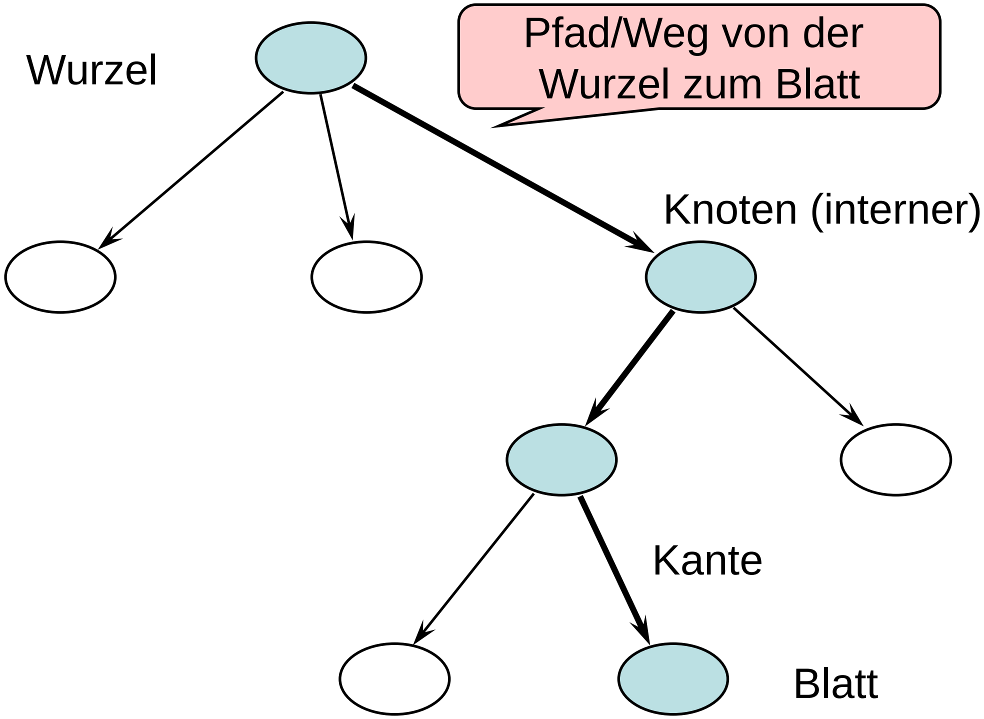
Ein Baum enthält daher **keine Kreise**, d.h. einen Pfad bei dem der Startknoten gleich dem Endknoten ist

Wurzel, Blatt, interne Knoten

Die **Wurzel** (**root**) ist der einzige Knoten mit nur wegführenden Kanten

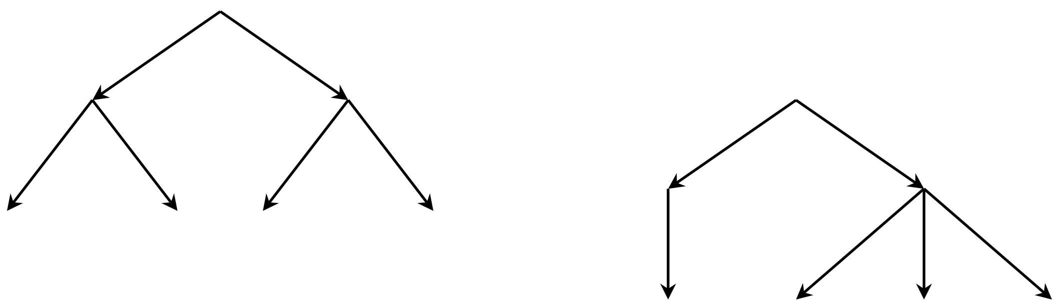
Ein **Blatt** (**leaf**) ist ein Knoten von denen keine Kanten wegführen

Knoten, in die eine Kante hineinführt und von denen Kanten wegführen, heißen **interne Knoten**

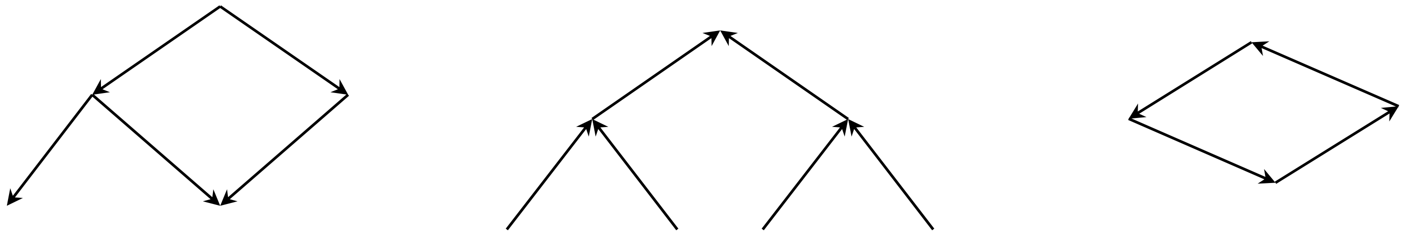


Gültige und ungültige Bäume

Gültige Bäume



Ungültige Bäume



- Es gibt genau einen Pfad der 2 Knoten miteinander verbindet
- jeder Knoten hat nur einen direkten Vorgänger
- alle Vorgänger eines Knoten sind von ihm selbst verschieden

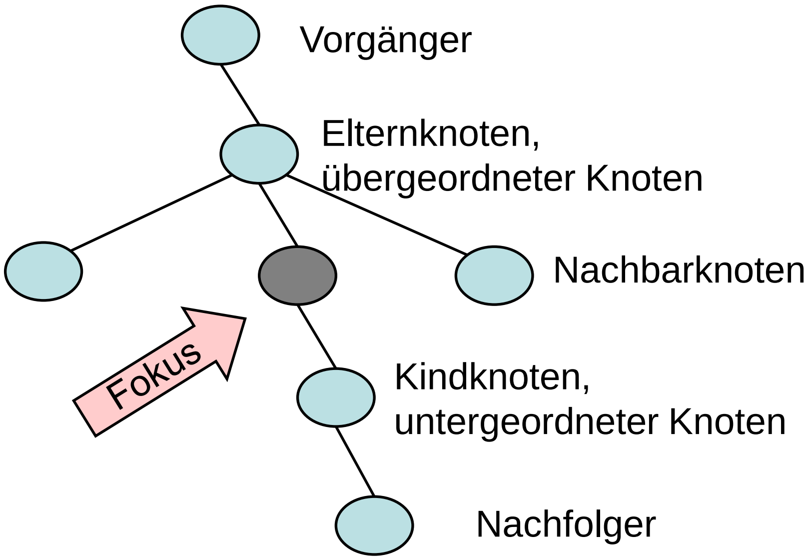
Spezielle Knoten

Jeder Knoten (mit Ausnahme der Wurzel) hat genau einen Knoten über sich, den man als **Elternknoten** oder **übergeordneten Knoten** bezeichnet

Die Knoten direkt unterhalb eines Knotens heißen **Kindknoten** oder **untergeordnete Knoten**

Transitive Eltern werden als **Vorgänger** bzw. transitive Kinder als **Nachfolger** bezeichnet

Räumlich nebeneinander-liegende Knoten auf derselben Tiefe heißen **Nachbarn**

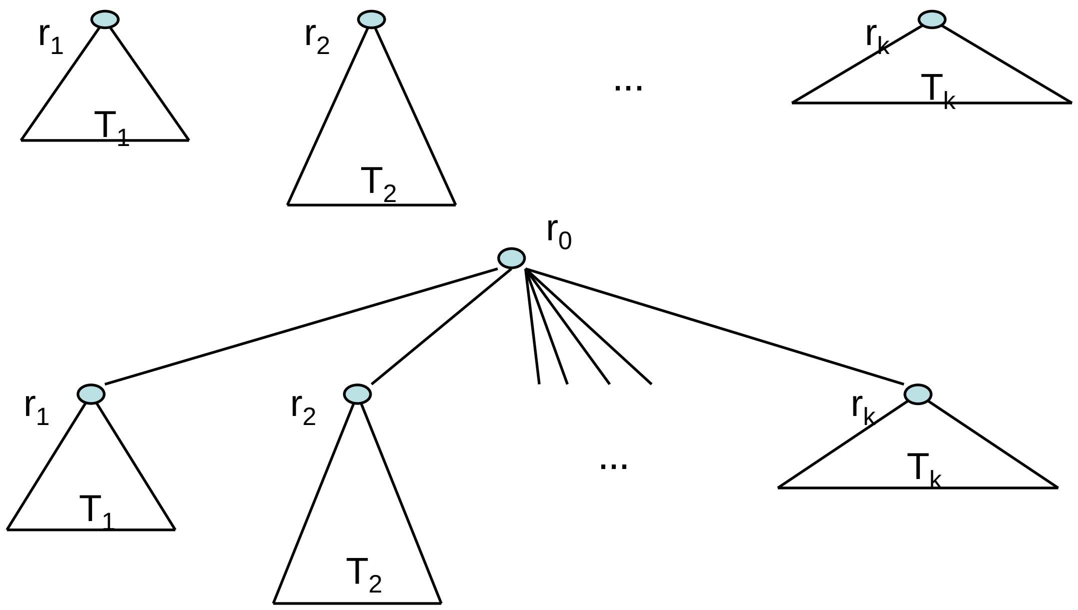


Rekursive Baum Definition

Ein einzelner Knoten ohne Kanten ist ein Baum

Seien $T_1, \ldots, T_k$ ($k > 0$) Bäume ohne gemeinsame Knoten. Die Wurzeln dieser Bäume seien $r_1, \ldots, r_k$. Ein Baum T_0 mit der Wurzel r_0 besteht aus den Knoten und Kanten der Bäume $T_1, \ldots, T_k$ und neuen Kanten von r_0 zu den Knoten $r_1, \ldots, r_k$

Der Knoten r_0 ist dann die neue Wurzel und $T_1, \ldots, T_k$ sind **Unterbäume** von T_0



r_i … Wurzel des Baumes i
T_i… Unterbaum i

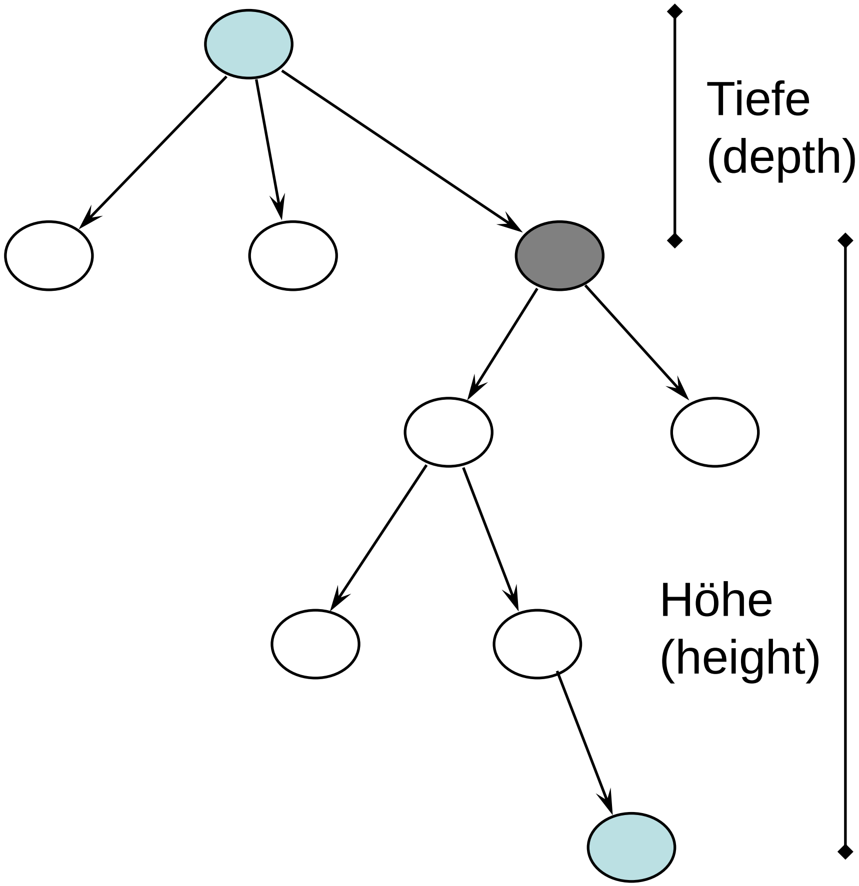
Länge, Höhe, Tiefe

Die **Länge eines Weges** zwischen 2 Knoten entspricht der Anzahl der Kanten auf dem Weg zwischen den beiden Knoten

Die **Höhe eines Knoten** ist die Länge des längsten Weges von diesem Knoten zu den erreichbaren Blättern

Die **Tiefe eines Knoten** ist die Länge des Weges zur Wurzel

Die **Höhe eines Baumes** entspricht der Höhe der Wurzel



Beispiel Parse Tree

Parsierungsbäume (**parse trees**) analysieren Anweisungen bzw. Programme einer Programmiersprache bezüglich einer gegebenen Grammatik

Eine Grammatik definiert Regeln, wie und aus welchen Elementen eine Programmiersprache aufgebaut ist

<*stmt*>	Nonterminal Symbole (werden aufgelöst)
`if`, `ident`	Terminalsymbole (nicht mehr aufgelöst)
::=, |	Grammatiksymbolik

Beispiel: C++ (Ausschnitt)

```
<stmt> ::= <select-stmt> | <expr>
<select-stmt> ::= if ( <expr> ) <stmt> else <stmt>
<expr> ::= <relational-expr> | <assign-expr> | ident
<relational-expr> ::= <expr> < <expr>
<assign-expr> ::= ident = <expr>
```

Beispiel Parse Tree (2)

`if (a < b) max = b else max = a`

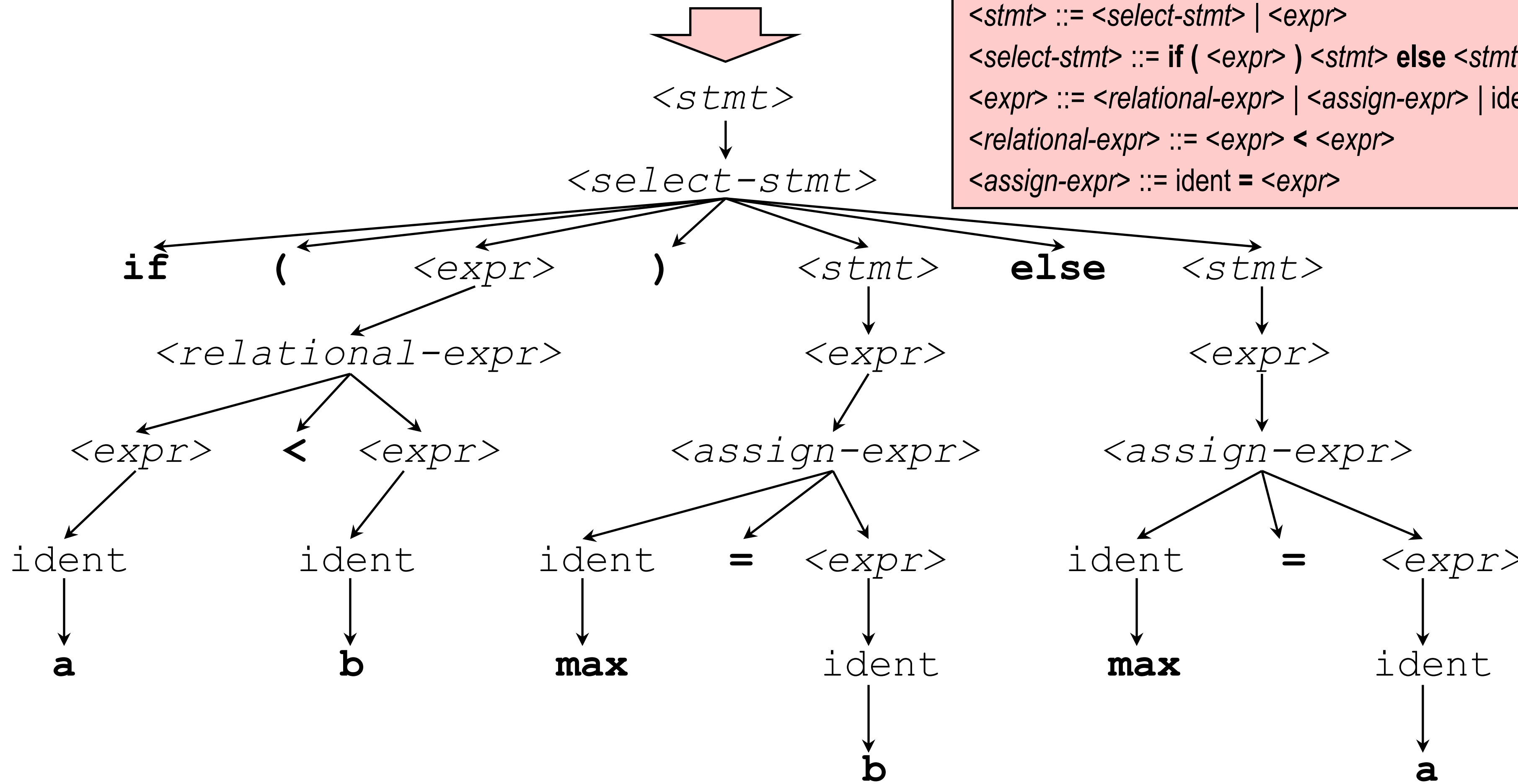
<*stmt*> ::= <*select-stmt*> | <*expr*>

<*select-stmt*> ::= **if (** <*expr*> **)** <*stmt*> **else** <*stmt*>

<*expr*> ::= <*relational-expr*> | <*assign-expr*> | ident

<*relational-expr*> ::= <*expr*> **<** <*expr*>

<*assign-expr*> ::= ident **=** <*expr*>



Spezielle Bäume

Die **Ordnung** legt die Position der Nachfolger eines Knotens in der grafischen Darstellung des Baumes fest (Knoten A links von Knoten B)

D.h. für die Kinder jedes Knotens ist eine Ordnungsrelation definiert

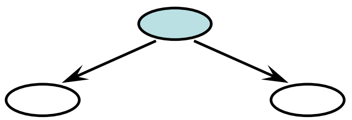
Bei diesem Baum handelt es sich um einen ungeordneten Baum, da die Reihenfolge der Nachfolger ohne semantische Bedeutung ist

- Bäume (trees)
  - ungeordnete Bäume (unordered trees)
  - geordnete Bäume (ordered trees)
    - binäre Bäume (binary trees)
    - Mehrweg Bäume (multiway trees)

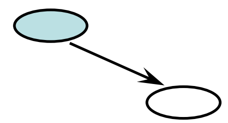
5.2 Binäre Bäume

Binäre Bäume sind Bäume, in denen jeder Knoten maximal 2 Kinder hat und die Ordnung der Knoten mit links und rechts festgelegt ist

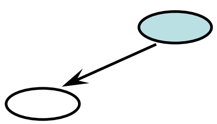
Binärer Baum mit 2 Kindern



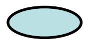
Binärer Baum mit rechtem Kind (right child)



Binärer Baum mit linkem Kind (left child)



Binärer Baum ohne Kind (Blatt)

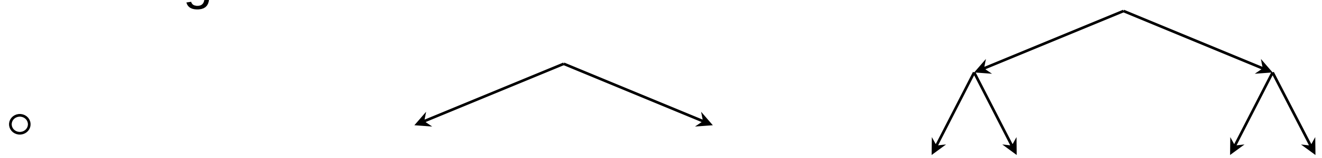


Binäre Bäume - Eigenschaften

Häufiger Einsatz in Algorithmen

Effiziente Manipulationsalgorithmen

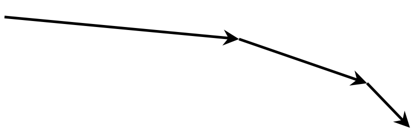
- Zusammenhang zw. Anzahl der Elemente und der Höhe



Höhe 0	Höhe 1	Höhe 2
1 Knoten	3 Knoten	7 Knoten
1 Blatt	2 Blätter	4 Blätter

- In jeder neuen Ebene wird im optimalen Fall die Anzahl der Blätter verdoppelt

Entartung möglich: Jeder Knoten hat genau ein Kind $\rightarrow$ entspricht linearer Liste



Formen binärer Bäume

Leerer binärer Baum

- Binärer Baum ohne Knoten

Voller binärer Baum

- Jeder Knoten hat keine oder 2 Kinder

Perfekter binärer Baum

- Ein voller binärer Baum bei dem alle Blätter dieselbe Tiefe besitzen

Kompletter binärer Baum

- Ein perfekter binärer Baum mit der Ausnahme, dass die Blattebene nicht vollständig, dafür aber von links nach rechts, gefüllt ist

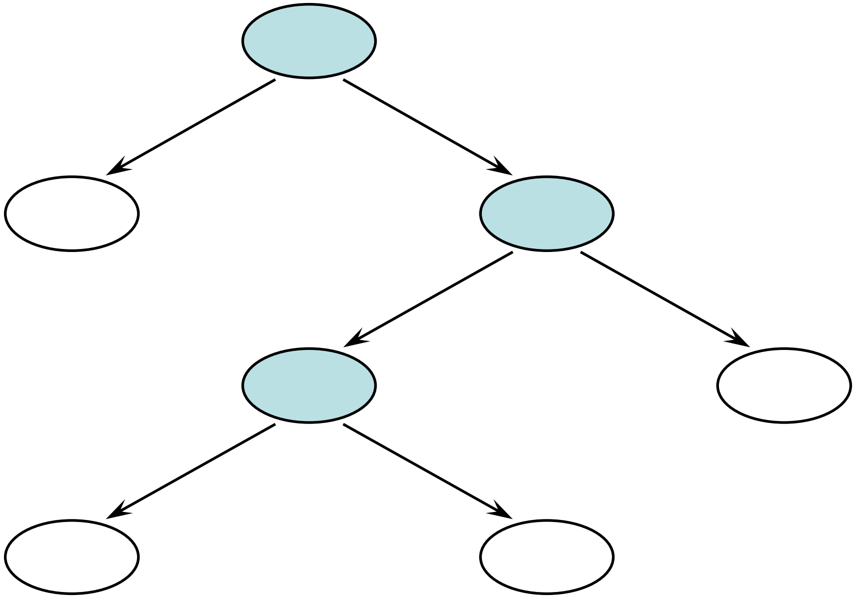
Höhen-balanzierter binärer Baum

- Für jeden Knoten ist der Unterschied der Höhe des linken und rechten Kindes maximal 1

Voller binärer Baum

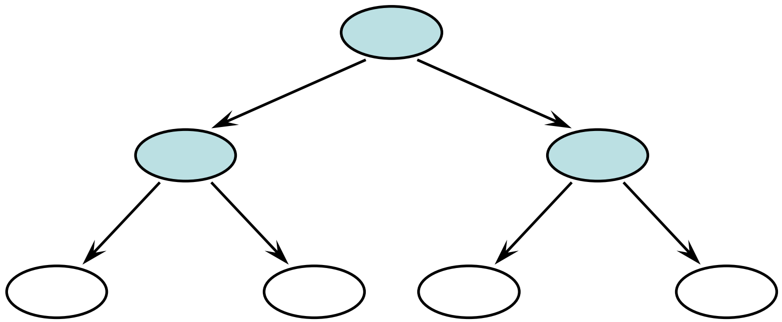
Full binary tree

- Jeder Knoten im Baum hat keine oder genau 2 Kinder.
  - Mit anderen Worten, kein Knoten besitzt nur 1 Kind



Perfect binary tree

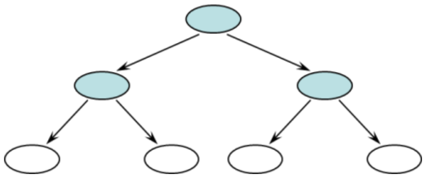
Ein voller binärer Baum (alle Knoten haben keine oder genau 2 Kinder) bei dem alle Blätter dieselbe Tiefe besitzen.



Eigenschaften des perfekten binären Baumes

Eigenschaften

Frage: welche Höhe h muss ein perfekter binärer Baum haben um n Blätter zu besitzen



In jeder Ebene Verdoppelung, d.h. $2^h = n$

$$h * \log 2 = \log n$$

$$h = \log_2 n = \text{ld } n$$

Ein perfekter binärer Baum der Höhe h besitzt $2^{h+1}-1$ Knoten

davon sind 2^h Blätter

Beweis mit vollständiger Induktion

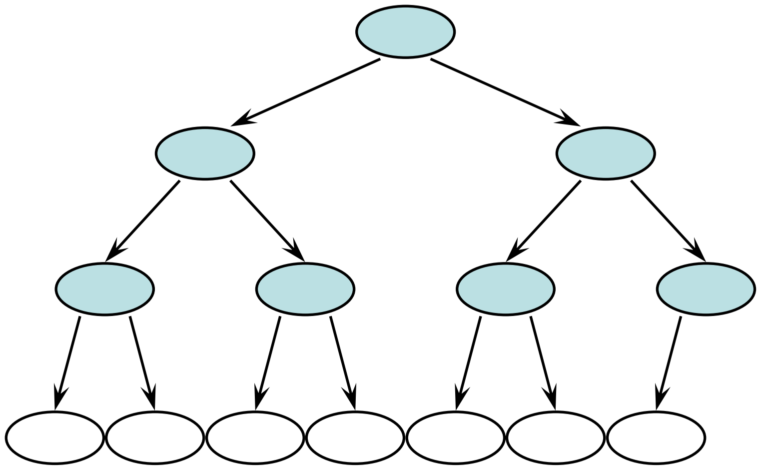
Zusammenhang zwischen Knoten/Blätter und Höhe:

wichtigste Eigenschaft von Bäumen

O(n) Knoten/Blätter : O(*log n)* Höhe

Complete binary tree

Ein kompletter binärer Baum ist ein perfekter binärer Baum mit der Ausnahme, dass die Blattebene nicht vollständig, dafür aber von links nach rechts, gefüllt ist



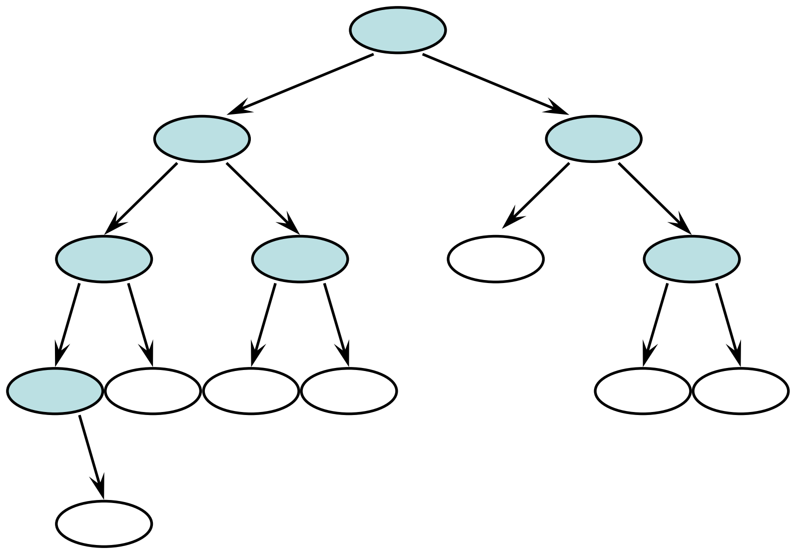
Eigenschaft

Ein kompletter binärer Baum mit n Knoten hat eine Höhe von maximal $h = \lfloor \log_2 n \rfloor$

Höhen-balanzierter binärer Baum

Height-balanced binary tree

- Für jeden Knoten ist der Unterschied der Höhe des linken und rechten Kindes maximal 1
  - Dies garantiert, dass lokal für jedes Kind die Balanzierungseigenschaft relativ gut erfüllt ist, global aber die gesamten Baumdifferenzen größer sein können $\rightarrow$ einfachere Algorithmen

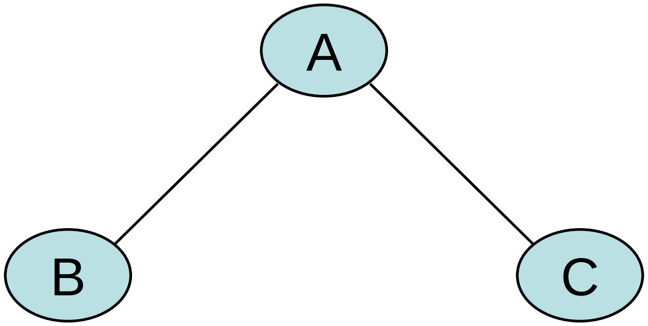


Traversieren eines Baumes bezeichnet das systematische Besuchen aller seiner Knoten

Unterschiedliche Methoden unterscheiden sich in der Reihenfolge der besuchten Knoten

Mögliche Reihenfolgen

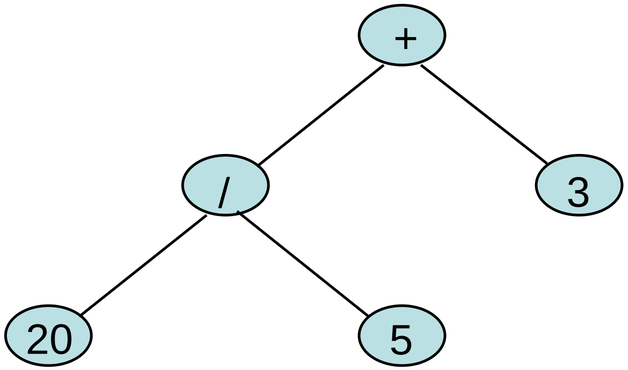
- A, B, C
- B, A, C
- B, C, A
- …



Systematische Auswertung eines mathematischen Ausdrucks

Ein mathematischer Ausdruck kann in Form eines Expression Trees angegeben werden

(20 / 5) + 3



Blätter repräsentieren Operanden (Zahlenwerte), interne Knoten Operatoren

Auswertung des Ausdrucks läuft von den Blättern zur Wurzel

Baumdarstellung erspart Klammernotation

Traversierungsalgorithmus

Traversierungsalgorithmen bestehen prinzipiell aus 3 verschiedenen Schritten

- Bearbeiten eines Knotens (process node)
- Rekursiv besuchen und bearbeiten des linken Kindes (visit left child)
- Rekursiv besuchen und bearbeiten des rechten Kindes (visit right child)

Durch unterschiedliches Anordnen der 3 Schritte unterschiedliche Reihenfolgen

3 Bearbeitungsreihenfolgen interessant

- Preorder Traversierung
- Postorder Traversierung
- Inorder Traversierung

Preorder Traversierung

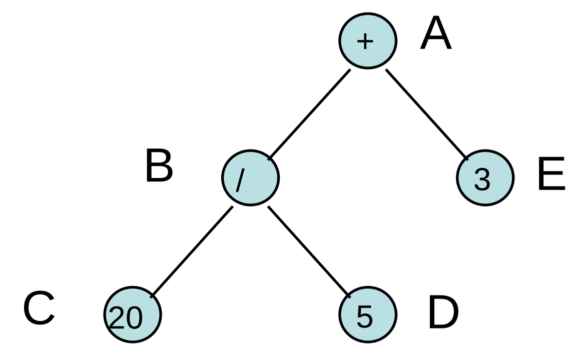
Algorithmus

```
preorder(node) {
  if(node != 0) {
    process(node)
    preorder(left child)
    preorder(right child)
  }
}
```

- Besucht die Knoten im Baum in *Prefix-Notation-Reihenfolge*
  - (Operator, Operand$_1$, Operand$_2$)
- Faustregel
  - Knoten bearbeiten beim 1. Besuch
- Anwendung
  - LISP, Assembler

Beispiel

20/5+3



Process-Reihenfolge: A B C D E

Notation-Reihenfolge: + / 20 5 3

Postorder Traversierung

Algorithmus

```
postorder(node) {
  if(node != 0) {
    postorder(left child)
    postorder(right child)
    process(node)
  }
}
```

Postfix-Notation-Reihenfolge

(Operand1, Operand2, Operator)

Faustregel

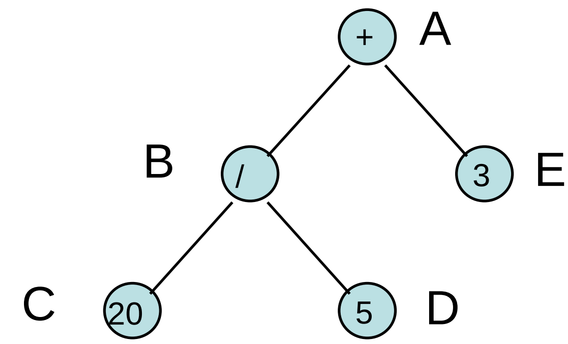
Knoten bearbeiten beim letzten Besuch

Anwendung

Invers Polish Notation, HP Taschenrechner, FORTH

Beispiel

20/5+3



Process-Reihenfolge: C D B E A

Notation-Reihenfolge: 20 5 / 3 +

Inorder Traversierung

Algorithmus

```
inorder(node) {
  if(node != 0) {
    inorder(left child)
    process(node)
    inorder(right child)
  }
}
```

Infix-Notation-Reihenfolge

(Operand$_1$, Operator, Operand$_2$)

Faustregel

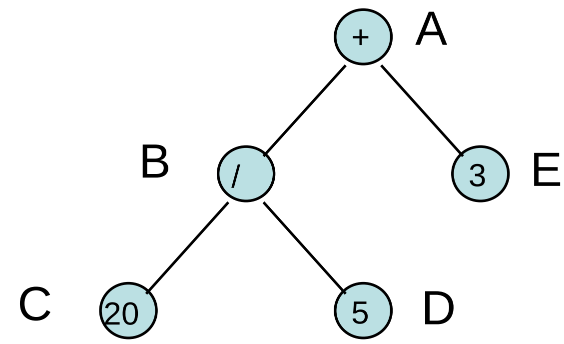
Knoten bearbeiten beim 2. oder letzten Besuch

Anwendung

einfacher algebraischer Taschen-rechner (ohne Klammern)

Beispiel

20/5+3



Process-Reihenfolge: C B D A E

Notation-Reihenfolge: 20 / 5 + 3

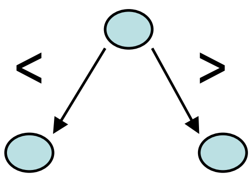
5.3 Binäre Suchbäume

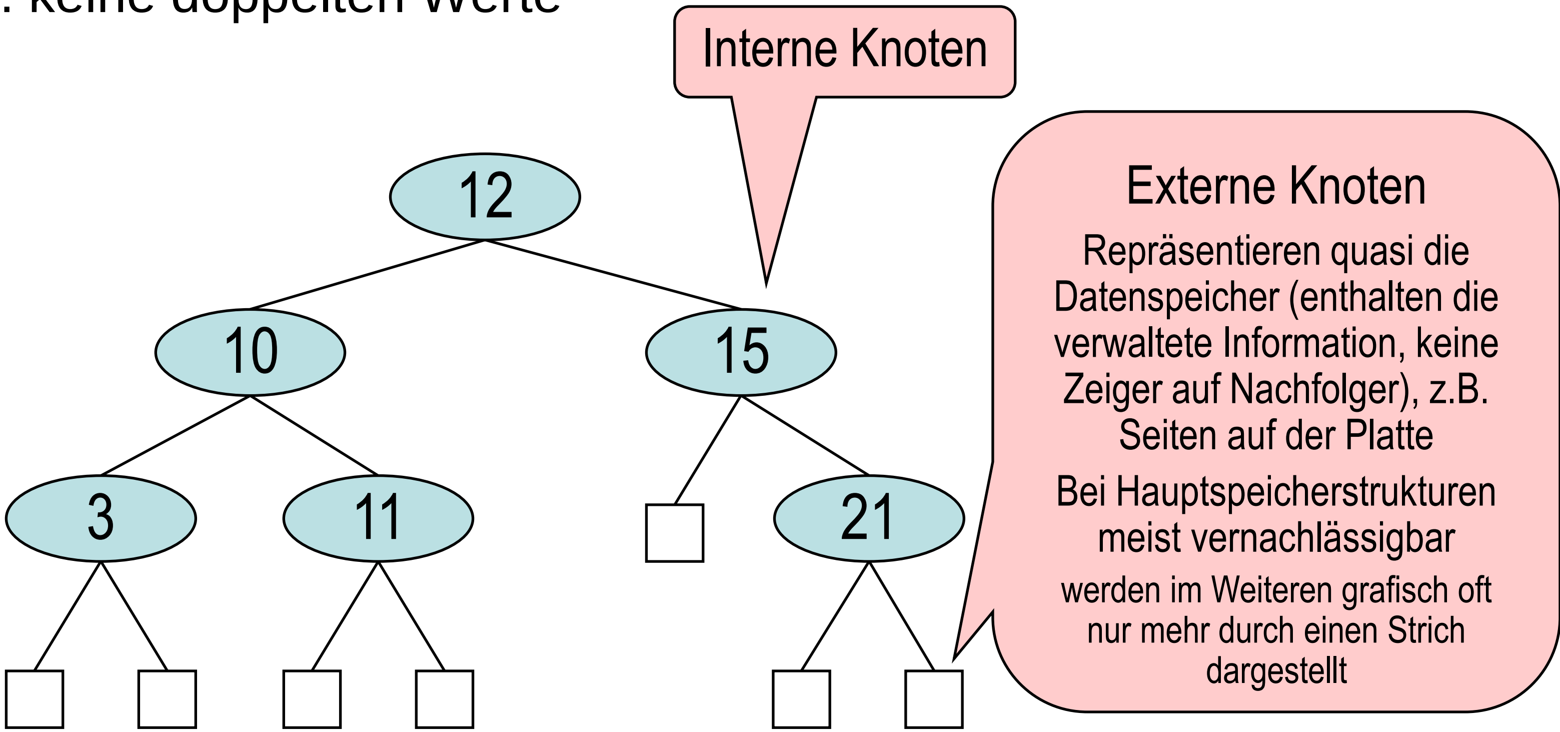
- In einem Binärbaum besitzt jeder (interne) Knoten eine linke und eine rechte Verbindung, die auf einen Binärbaum oder einen externen Knoten verweist (voller binärer Baum)
  - Verbindungen zu externen Knoten heißen Nullverbindungen, externe Knoten besitzen keine weiteren Verbindungen
- Ein binärer Suchbaum (binary search tree, BST) ist ein Binärbaum, bei dem jeder interne Knoten einen Schlüssel besitzt
  - externe Knoten besitzen keine Schlüssel
    - Externe Knoten enthalten die verwaltete Information, i.e. Datenspeicher (wird hier nicht weiter behandelt)
  - Auf den Schlüsseln ist eine lineare Ordnung “<“ definiert
    - Wenn x und y verschieden sind, so gilt x<y oder y<x; ist x<y und y<z, dann gilt auch x<z

Schlüsseleigenschaft im BST

Für jeden internen Knoten gilt, dass alle Werte der Nachfolger im linken Unterbaum kleiner als der Knotenwert und die Werte der Nachfolger im rechten Unterbaum größer als der Knotenwert sind

Annahme: keine doppelten Werte





Eigenschaften binärer Suchbäume

- Verwaltung beliebig großer Datenbestände
  - dynamische Struktur
- Effiziente Verwaltung
  - Aufwand proportional zur Höhe des Baumes (Erwartungswert!) und nicht zur Anzahl der Elemente
  - Einfügen, Zugriff und Löschen im Durchschnitt von O(log n)
  - Signifikante Verbesserung im Vergleich zum linearen Aufwand O(n) bei Liste
  - Zugriff auf Elemente in sortierter Reihenfolge durch inorder Traversierung

Operationen

Erstellen

- Erzeugen eines leeren Suchbaumes

Einfügen

- Einfügen eines Elementes in den Baum unter Berücksichtigung der Ordnungseigenschaft

Suche

- Test auf Inklusion

Löschen

- Entfernen eines Elementes aus dem Baum unter Erhaltung der Ordnungseigenschaft

Ausgabe

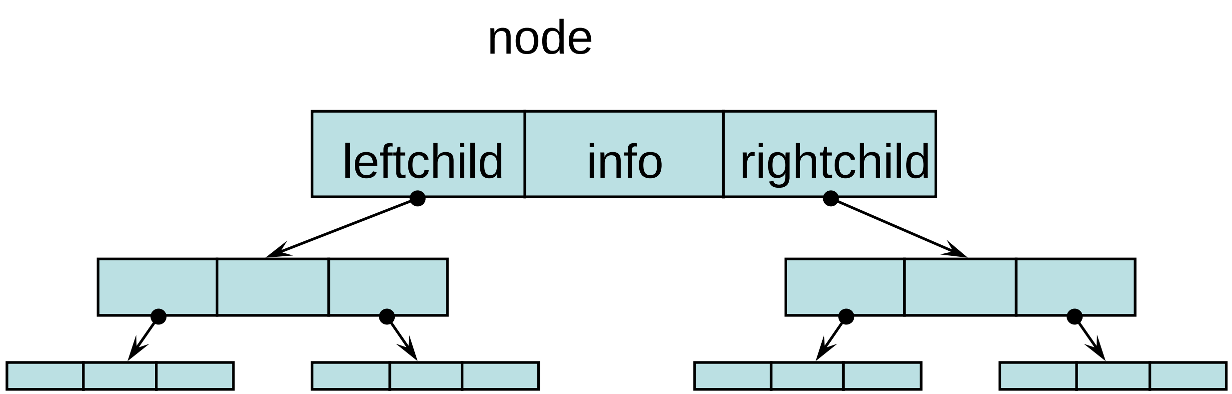
- Ausgabe aller Elemente in sortierter Reihenfolge

...

Klassendefinition

```
typedef int ItemType;
class SearchTree {
  class node {
  public:
    ItemType info;
    node * leftchild, * rightchild;
    node(ItemType x, node * l, node * r) {info=x; leftchild=l; rightchild=r;}
  };
  typedef node * link;
  link root;
  void AddI(ItemType);
  void AddR(link&, ItemType);
  bool MemberI(ItemType);
  bool MemberR(link, ItemType);
  void PrintR(link, int);
public:
  SearchTree(){root = 0;}
  void Add(ItemType);
  int Delete(ItemType);
  bool Member(ItemType);
  void Print();
};
```

Einfachheitshalber im **`node`** alles **`public`**

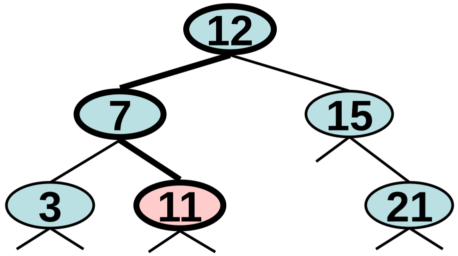


Suchen in einem binären Baum

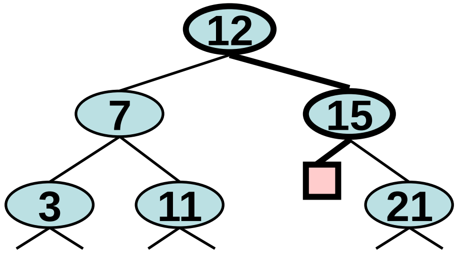
Beim Suchen eines Schlüssels wird ein Pfad von der Wurzel abwärts verfolgt

- Bei jedem internen Knoten wird der Schlüssel x mit dem Suchschlüssel s verglichen
- Falls x = s liegt ein Treffer vor, falls s < x suche im linken Teilbaum, sonst im rechten
- Wenn man einen externen Knoten erreicht, war die Schlüssel-Suche „erfolglos“

Erfolgreiche Suche (z.B. 11)



Erfolglose Suche (z.B. 13)



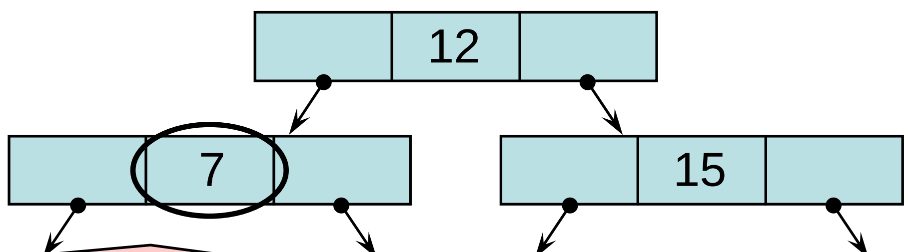
Suche (iterativ)

```
bool SearchTree::MemberI(ItemType a) {
  if(root) {
    link current = root;
    while(current) {
      if(current->info == a) return true;
      if(a < current->info)
        current = current->leftchild;
      else
        current = current->rightchild;
    }
  }
  return false;
}
bool SearchTree::Member(ItemType a) {
  return MemberI(a);
}
```

Aufruf:

```
SearchTree t;
…
t.Member(7);
```

Verzweigung auf current->leftchild, da a < current->info (d.I. 7 < 12)



return true, da current->info == a (gefunden!)

Suche (rekursiv)

```
bool SearchTree::MemberR(link h, ItemType a) {
  if(!h) return false;
  else {
    if(a == h->info) return true;
    if(a < h->info)
      return MemberR(h->leftchild, a);
    else
      return MemberR(h->rightchild, a);
  }
}
bool SearchTree::Member(ItemType a) {
  return MemberR(root, a);
}
```

Aufruf:

```
SearchTree t;
…
t.Member(7);
```

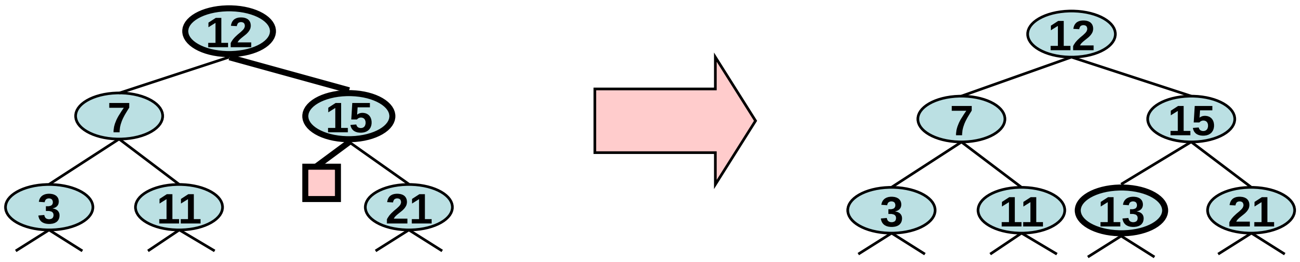
Einfügen in einem binären Suchbaum

Das Einfügen entspricht einer erfolglosen Suche und dem Anfügen eines neuen Knotens an der Nullverbindung wo die Suche endet (anstelle des externen Knotens)

Einfügen 13

Suche

Knoten anfügen



Einfügen (iterativ)

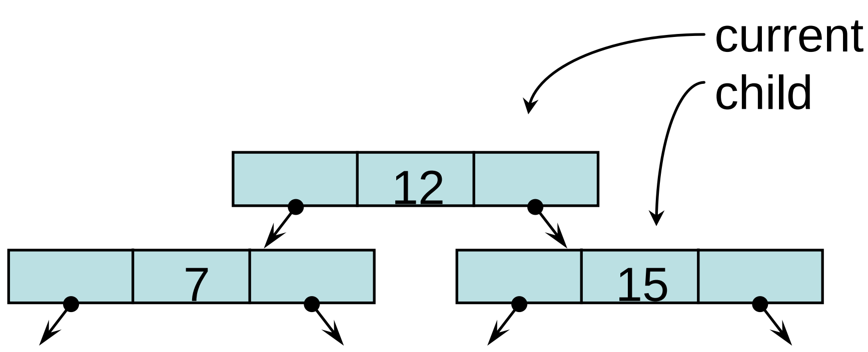
Ähnlich dem Einfügen in eine Liste (2 Hilfszeiger)

```
void SearchTree::AddI(ItemType a) {
  if(root) {
    link current = root;
    link child;
    while(1) {
      if(a == current->info) return;
      if(a < current->info) {
        child = current->leftchild;
        if(!child) {current->leftchild = new node(a, 0, 0); return;}
      } else {
        child = current->rightchild;
        if(!child) {current->rightchild = new node(a, 0, 0); return;}
      }
      current = child;
    }
  } else {
    root = new node(a, 0, 0);
    return;
  }
}
void SearchTree::Add(ItemType a) {
  …
  AddI(a);
}
```

Hilfszeiger:
`current, child`

Aufruf:

```
SearchTree t;
…
t.Add(7);
```



Einfügen (rekursiv)

```
void SearchTree::AddR(link& h, ItemType a) {
  if(!h) {h = new node(a, 0, 0); return;}
  if(a == h->info) return;
  if(a < h->info)
    AddR(h->leftchild, a);
  else
    AddR(h->rightchild, a);
}
void SearchTree::Add(ItemType a) {
  AddR(root, a);
}
```

Man beachte die Verwendung eines Referenzparameters (`link&`), erspart die 2 Hilfszeiger

Aufruf

```
SearchTree t;
…
t.Add(7);
```

Traversieren

```
void SearchTree::PrintR(link h, int n) {
  if(!h) return;
  PrintR(h->rightchild, n+2);
  for(int i = 0; i < n; i++) cout << " ";
  cout << h->info << endl;
  PrintR(h->leftchild, n+2);
}
void SearchTree::Print() {
  PrintR(root, 0);
}
```

Inorder Traversierung

Gibt Baum um 90 Grad gegen den Uhrzeigersinn verdreht aus

Aufruf

```
SearchTree t;
...
t.Print();
```

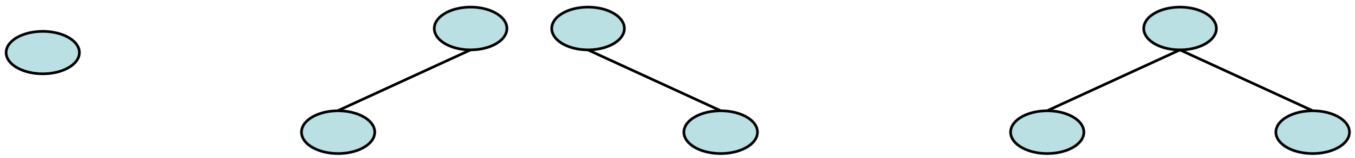
Löschen in einem binären Baum

Der Löschvorgang unterscheidet 3 Fälle

Löschen von internen Knoten mit

(1) keinem (2) einem (3) zwei

internen Knoten als Kind(er)

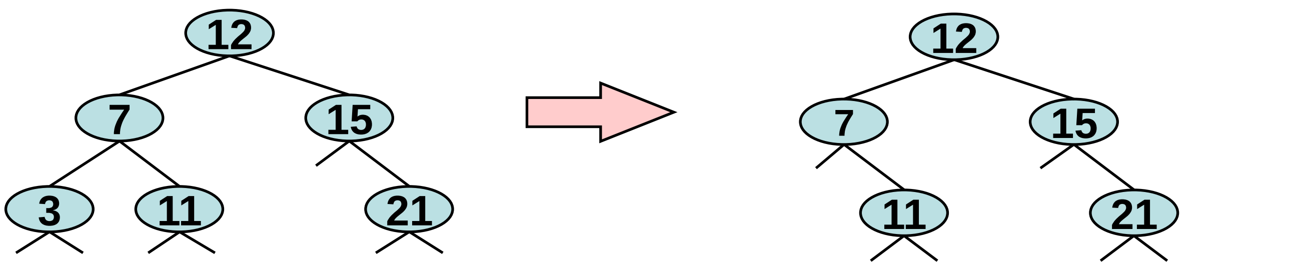


- Fälle 1 und 2 sind einfach durch Anhängen des verbleibenden Teilbaums an den Elternknoten des zu löschenden Knotens zu lösen.
- Für Fall 3 muss ein geeigneter Ersatzknoten gefunden werden. Hierzu eignen sich entweder der kleinste im rechten (Inorder Nachfolger) oder der größte im linken Teilbaum (Inorder Vorgänger)

Löschen (2)

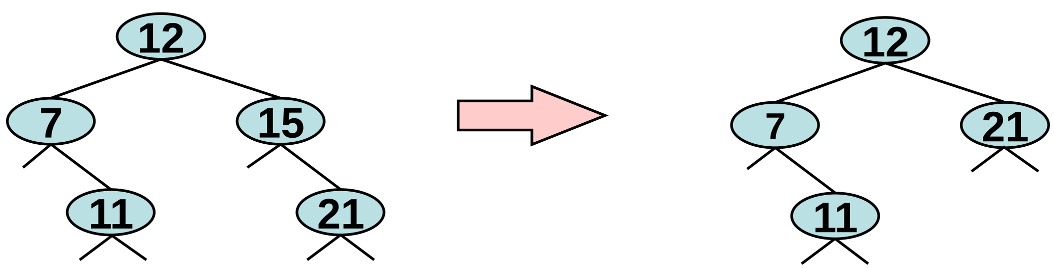
Fall 1

Löschen von Knoten 3 durch Ersetzen des linken Kindzeigers von Knoten 7 durch den externen Knoten (hier Nullverbindung)



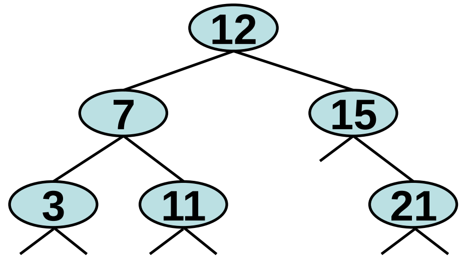
Fall 2

Löschen von Knoten 15 durch Einhängen des Knoten 21 als rechtes Kind von 12

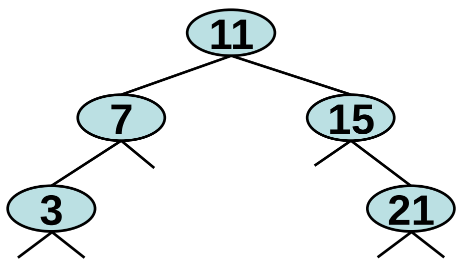


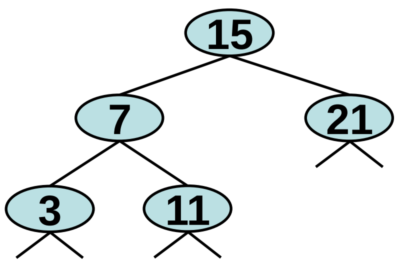
Löschen (3)

Fall 3



Im Falle des Löschens von 12 eignen sich als Ersatz die Knoten mit den Werten 11 (größte im linken) oder 15 (der kleinste im rechten Teilbaum).
Dies resultiert in 2 möglichen Bäumen:





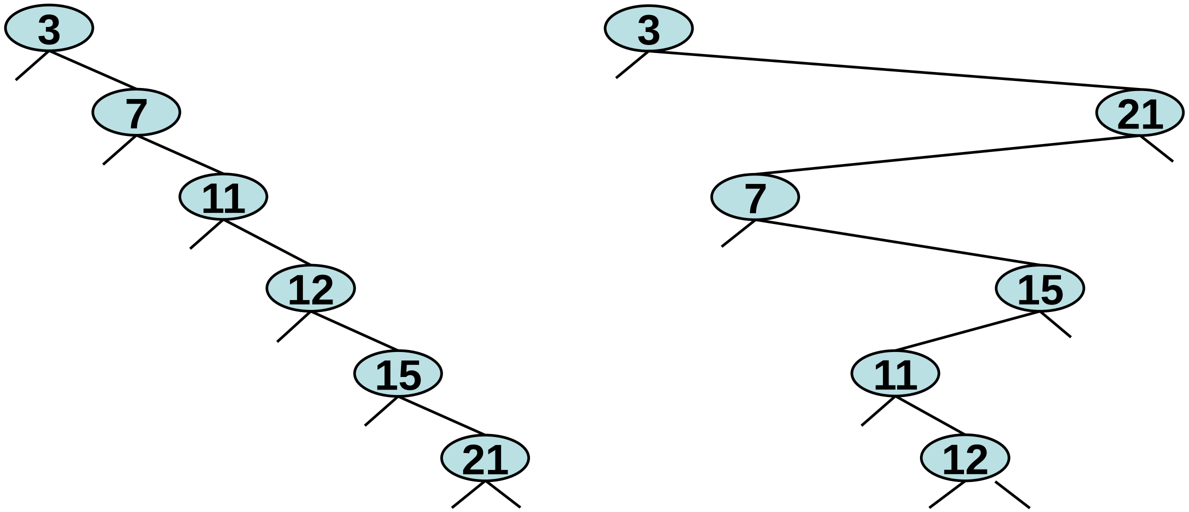
Somit muss der Löschvorgang für Fall 3 in zwei Teile zerlegt werden:

1. Finden eines geeigneten Ersatzknotens
2. Ersetzen des zu löschenden Knotens (was wiederum aus dem Entfernen des Ersatzknotens aus seiner ursprünglichen Position (entspricht Fall 1 oder 2) und dem Einhängen an der neuen Stelle besteht)

Erwartungswert der Einfüge- und Suchoperationen bei n zufälligen Schlüsselwerten ist ungefähr 1,39 ld n

Im ungünstigsten Fall kann der Aufwand zu ungefähr n Operationen „entarten“

Beispiele für ungünstige Suchbäume



Analyse binärer Suchbaum

Datenverwaltung

- unterstützt Einfügen und Löschen

Datenmenge

- unbeschränkt

Modelle

- Hauptspeicherorientiert
- Unterstützung komplexer Operationen
  - Bereichsabfragen, Sortierreihenfolge

Laufzeit

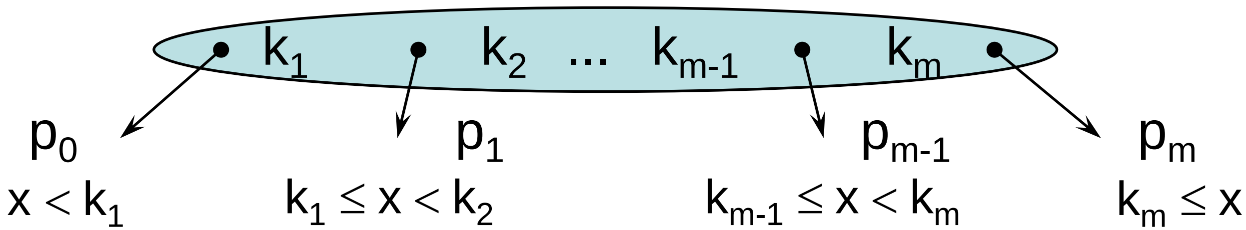
Speicherplatz	O(n)
Konstruktor	O(1)
Zugriff	O(log n)
Einfügen	O(log n)
Löschen	O(log n)
Sortierreihenfolge	O(n)

Erwartungswert!

Bitte beachten:
beträchtlicher konstanter Aufwand ist notwendig!

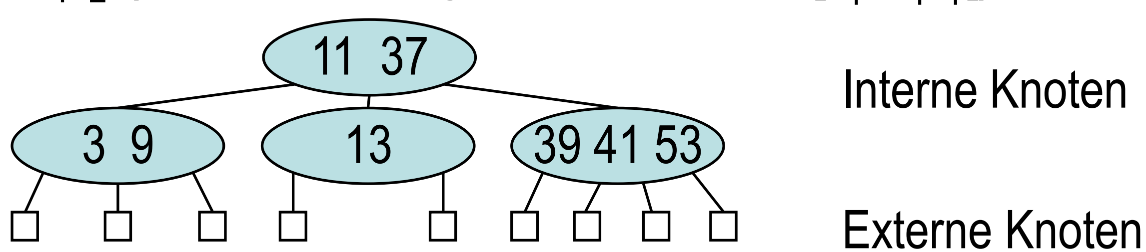
Mehrwegbäume sind Bäume mit Knoten, die mehr als 2 Kinder besitzen können

Intervallbasierten Suchdatenstrukturen



Knoten besteht aus einer Menge von **m Schlüsselwerten** k_1, k_2, …, k_m und **m+1 Verweisen** (Kanten) p_0, p_2, …, p_m, sodass für alle Schlüssel x_j im Unterbaum, der durch p_i referenziert wird, gilt $k_i \leq x_j < k_{i+1}$ (Schlüssel liegen im Intervall $[k_i, k_{i+1}[$).

Beispiel



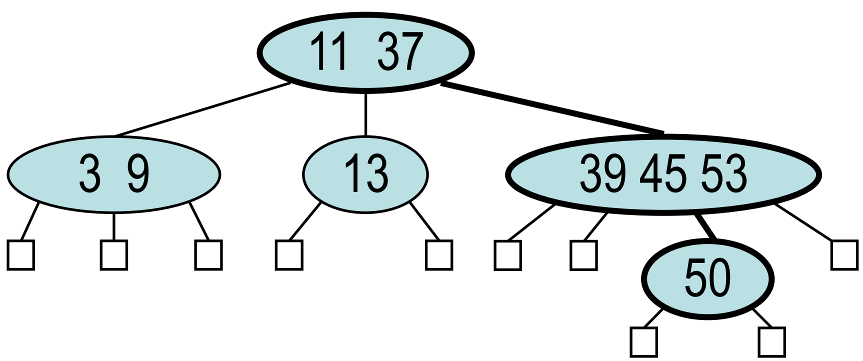
Suchen im Mehrwegbaum

Ähnlich zur Suche im binären Suchbaum

Bei jedem internen Knoten mit Schlüsseln $k_1, k_2, \ldots, k_m$ und Verbindungen $p_0, p_1, \ldots, p_m$

- Suchschlüssel $s = k_i$ ($i = 1, \ldots, m$): Suche erfolgreich
- $s < k_1$: fortsetzen im Unterbaum p_0
- $k_i \leq s < k_{i+1}$: fortsetzen im Unterbaum p_i
- $s \geq k_m$: fortsetzen im Unterbaum p_m
- Falls man einen externen Knoten erreicht: Schlüsselsuche erfolglos

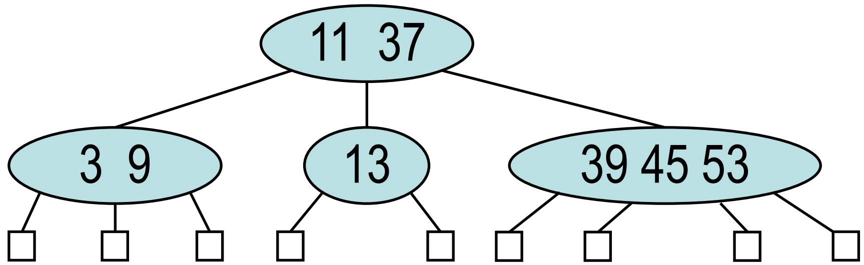
Beispiel: Suche 50



2-3-4 Bäume

Ein **2-3-4 Baum** ist ein Mehrwegbaum mit den folgenden Eigenschaften

- **Größeneigenschaft**: jeder interne Knoten hat mindestens 2 und maximal 4 Kinder
- **Tiefeneigenschaft**: alle externe Knoten besitzen dieselbe Tiefe
- Abhängig von der Anzahl der Kinder heißt ein interner Knoten 2-Knoten, 3-Knoten oder 4-Knoten



Höhe eines 2-3-4 Baumes

Satz: Ein 2-3-4 Baum, der n interne Knoten speichert, hat (immer) eine Höhe von O(log n)

Beweis

Die Höhe des 2-3-4 Baumes mit n internen Knoten sei h

Da es mindestens 2^i interne Knoten auf den Tiefen i = 0, … , h-1 gibt und keine internen Knoten auf Tiefe h, gilt

$$n \geq 1 + 2 + 4 + \ldots + 2^{h-1} = 2^h - 1$$

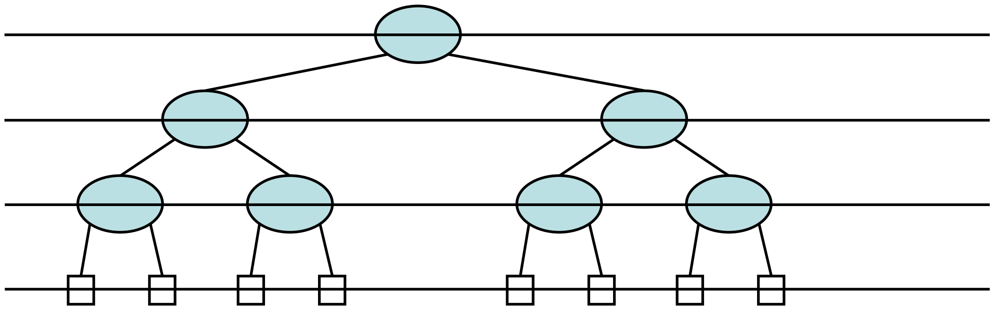
Daher gilt $h \leq \log_2 (n + 1)$

$n \geq 2^h - 1$

$n + 1 \geq 2^h$

$\log (n + 1) \geq h \log 2$

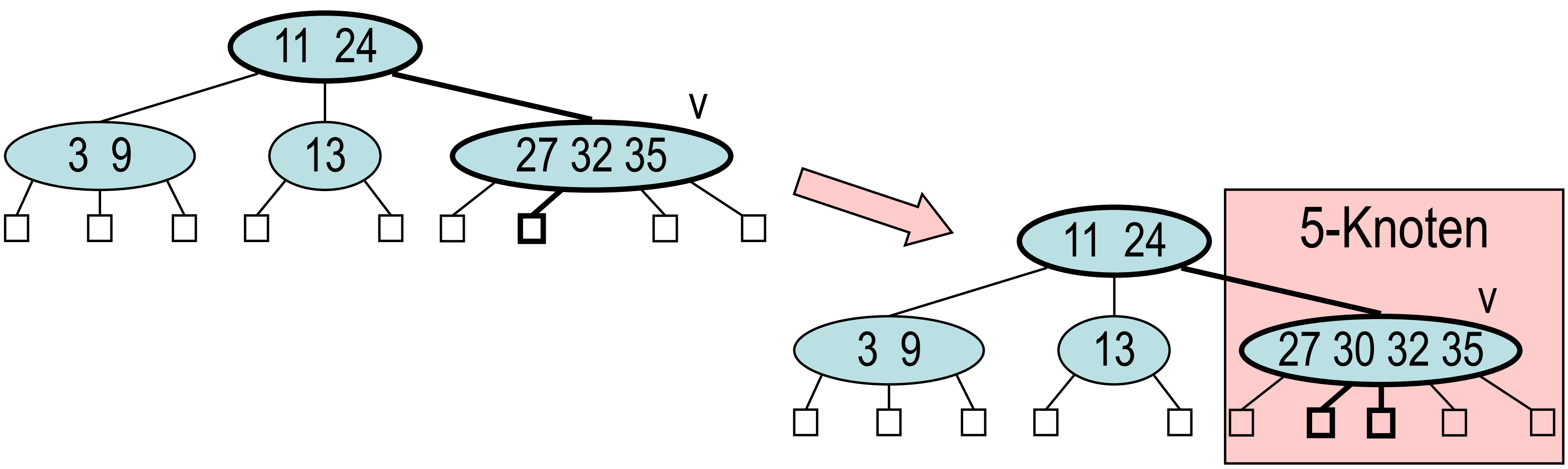
$h \leq \log (n +1) / \log 2 = \log_2 (n + 1)$



Knoten	Tiefe
1	0
2	1
2^{h-1}	h-1
0	h

Einfügen in einen 2-3-4 Baum

- Ein neuer Schlüssel s wird im Elternknoten v des externen Knotens eingefügt, den man bei der Suche nach s erreicht hat
  - Die Tiefeneigenschaft des Baumes wird erhalten, aber es wird möglicherweise die Größeneigenschaft verletzt
  - Ein (Knoten-) Überlauf ist möglich (es entsteht ein 5-Knoten)
- Beispiel: Einfügen von 30 erzeugt Überlauf

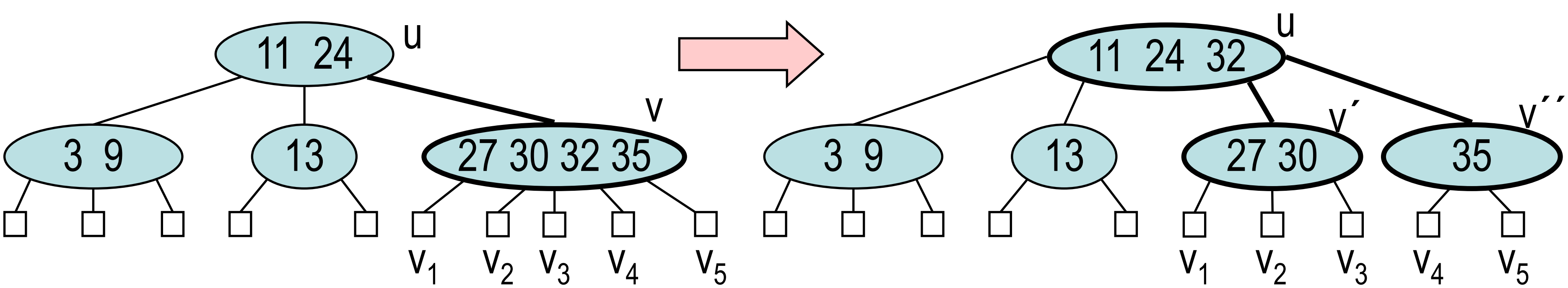


Überlauf und Split

Ein Überlauf (Overflow) bei einem 5-Knoten v wird durch eine Split Operation aufgelöst

- Die Kinder von v seien $v_1, \dots, v_5$ und die Schlüssel von v seien $k_1, \dots, k_4$
- Knoten v wird durch die Knoten v´ und v´´ ersetzt, wobei
  - v´ ein 3-Knoten mit den Schlüsseln k_1 und k_2 und Kindern v_1, v_2 und v_3,
  - v´´ ein 2-Knoten mit Schlüssel k_4 und Kindern v_4 und v_5 ist
- Schlüssel k_3 wird in den Elternknoten u von v eingefügt (dadurch kann eine neue Wurzel entstehen)

Ein Überlauf kann an die Vorgänger von u propagiert werden



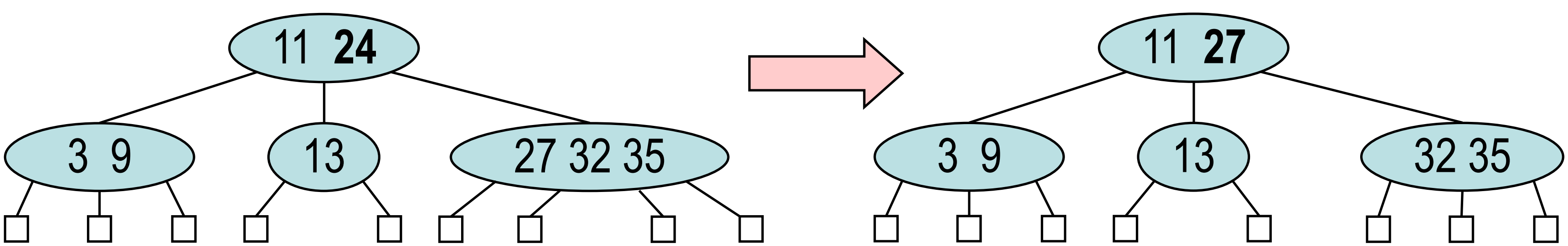
Löschen

Der Löschvorgang wird auf den Fall reduziert, wo der zu löschende Schlüsselwert in einem internen Knoten mit externen Knotenkindern liegt

Andernfalls wird der Schlüsselwert mit seinem Inorder Nachfolger (oder Inorder Vorgänger) ersetzt

- Analog Fall 3 binäre Suchbäume

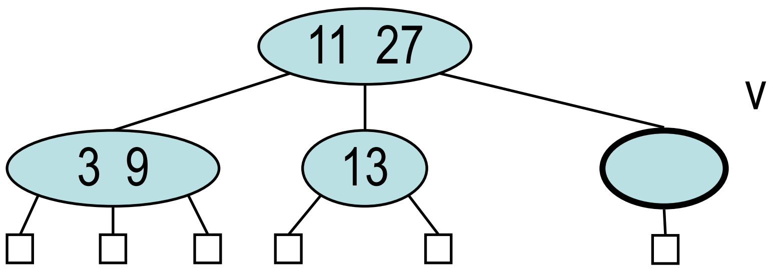
Beispiel Löschen Schlüssel 24



Unterlauf

Durch das Löschen eines Schlüssels in einem Knoten v kann es zu einem Unterlauf (underflow) kommen

- Knoten v degeneriert zu einem 1-Knoten mit einem Kind und keinem Schlüssel

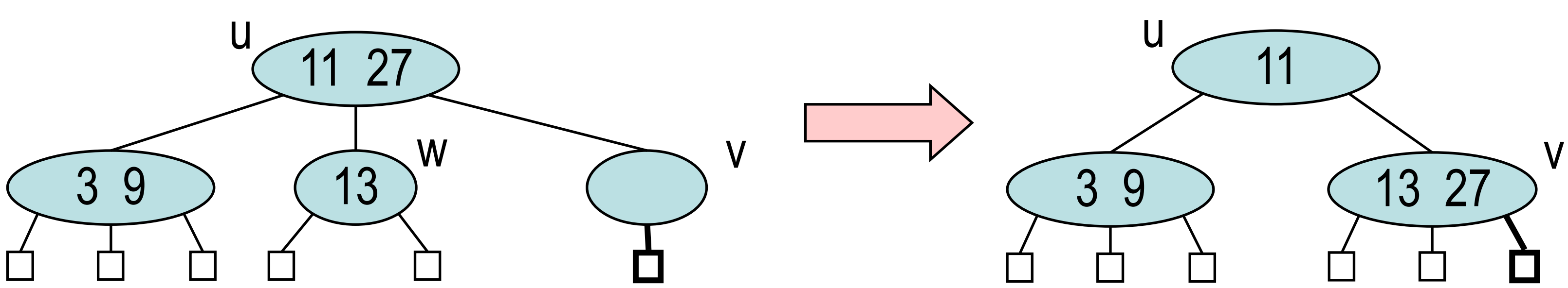


Bei einem Unterlauf kann man 2 Fälle unterscheiden

- Fall 1: Verschmelzen
  - Benachbarte Knoten werden zu einem erlaubten Knoten verschmolzen
- Fall 2: Verschieben
  - Schlüsselwerte und entsprechende Kinder werden zwischen Knoten verschoben

Fall 1: Verschmelzen von Knoten

- Bedingung: Alle benachbarten Knoten auf derselben Tiefe zum unterlaufenden Knoten v sind 2-Knoten
- Man verschmilzt v mit einem/dem Nachbarn w und verschiebt den nicht mehr benötigten Schlüssel vom Elternknoten u zu dem verschmolzenen Knoten v´
- Das Verschmelzen kann den Unterlauf zum Elternknoten propagieren



Verschieben im 2-3-4 Baum

Fall 2: Verschieben von Schlüsseln

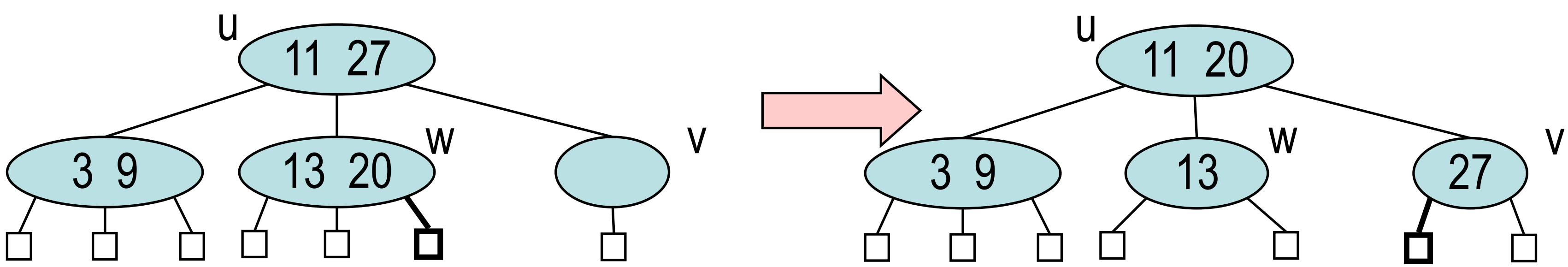
Bedingung: Ein benachbarter Knoten auf derselben Tiefe w zum unterlaufenden Knoten v ist ein 3-Knoten oder 4-Knoten

Man verschiebt ein Kind von w nach v

Man verschiebt einen Schlüssel von u nach v

Man verschiebt einen Schlüssel von w nach u

Nach dem Verschieben ist der Unterlauf behoben



Analyse 2-3-4 Baum

Datenverwaltung

- Einfügen und Löschen unterstützt

Datenmenge

- unbeschränkt

Modelle

- Hauptspeicherorientiert
- Unterstützung komplexer Operationen
  - Bereichsabfragen, Sortierreihenfolge

Laufzeit

Speicherplatz	O(n)
Verschmelzen, Verschieben	O(1)
Zugriff	O(log n)
Einfügen	O(log n)
Löschen	O(log n)
Sortierreihenfolge	O(n)

Garantiert!

Vergleich der Baumvarianten

Baumvariante	*Baumhöhe + Aufwand der Operationen*	*Balanzierungsform*	*Methode*
Allgemeiner Binärer Suchbaum	Im Durchschnitt log(n)	Abhängig von der Eingabe	Zufall, Gesetz der großen Zahlen
2-3-4 Baum	log(n)	perfektbalanziert	Split, Verschmelzen, Verschieben

Ziel

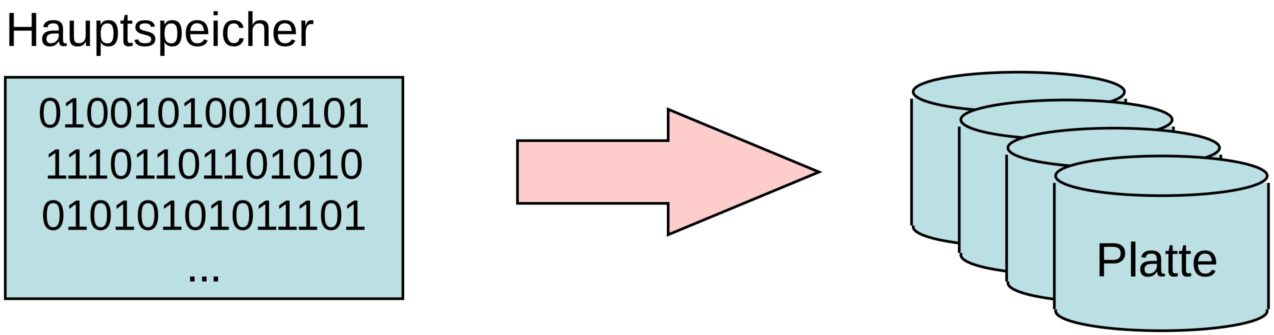
Erstellen eines Schlüsselbaumes für eine große Anzahl von Elementen

Problem

Hauptspeicher zu klein

Lösung

Speicherung der Knoten auf dem Externspeicher (Platte)



Plattenzugriffe

Ansatz

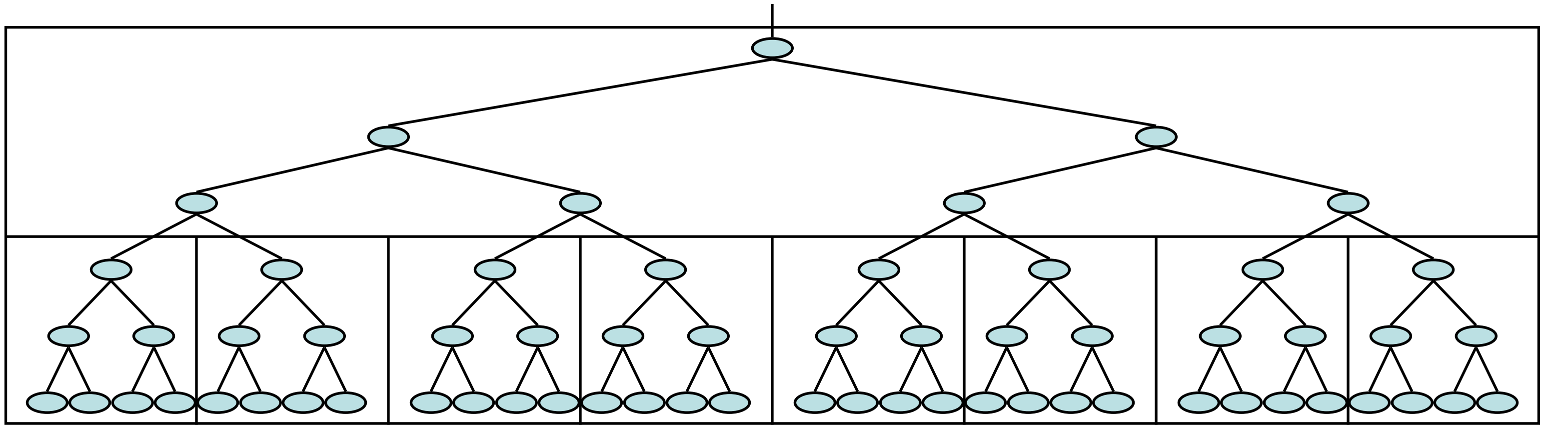
- Referenzieren eines Knotens entspricht Zugriff auf die Platte
  - Bei Aufbau eines binären Schlüsselbaumes für eine Datenmenge von z.B. einer Million Elementen ist die durchschnittliche Baumhöhe $\log_2 10^6 \approx 20$, d.h. 20 Suchschritte $\rightarrow$ 20 Plattenzugriffe

Dilemma

- Unterschied Aufwand Hauptspeicher- zu Plattenzugriff Faktor 10.000 (SSD) bis 100.000 (HD)
  - milli- zu nano-Sekunden (z.B. 60 ns zu 0,25-9ms)
- Jeder Suchschritt benötigt einen Plattenzugriff $\Rightarrow$ hoher Aufwand an Rechenzeit

Lösung

Zerlegung des binären Baumes in Teilbäume, diese in sog. *Seiten (pages)* speichern



$\Rightarrow$ Mehrwegbaum (multiway-tree)

Intervallbasierte Suchdatenstruktur

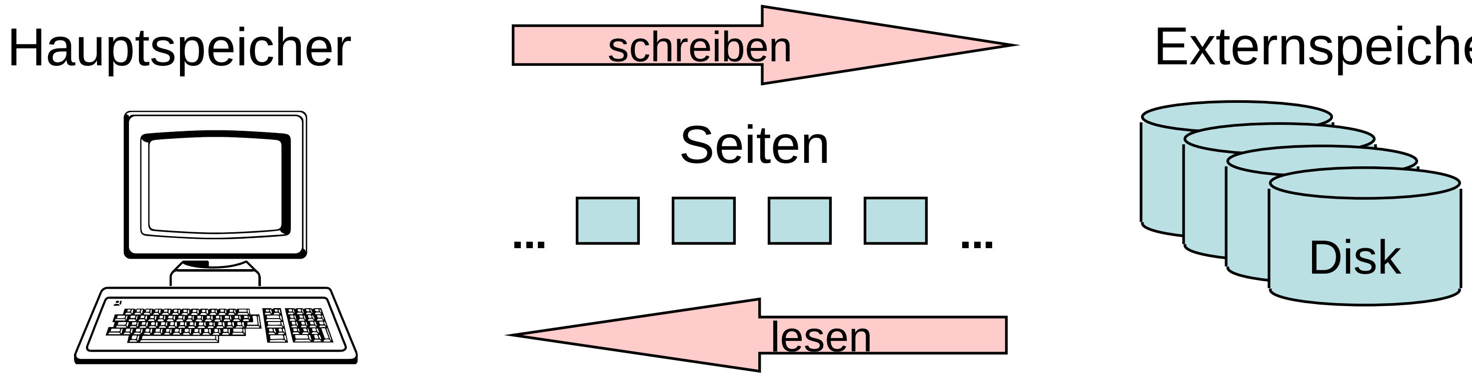
Bei 100 Knoten pro Seite für 1 Million Elemente nur mehr $\log_{100}10^6 = 3$ Seitenzugriffe

Im schlimmsten Fall (lineare Entartung) aber immer noch 10^4 Zugriffe (10^6 / 100)

Seitenverwaltung (2)

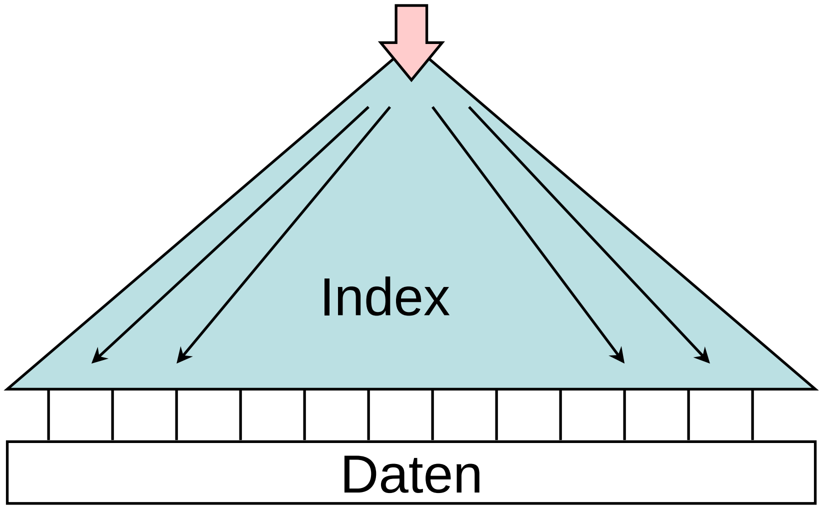
Knoten entsprechen Seiten (Blöcke)

- besitzen eine Größe, Seitengröße (Anzahl der enthaltenen Einträge bzw. Speichergröße in Bytes)
- repräsentieren die Transfereinheit zwischen Haupt- und Externspeicher
- Zur Effizienzsteigerung des Transfers wird die Größe der Seiten an die Blockgröße der Speichertransfereiheiten des Betriebssystems angepaßt (Unit of Paging/Swapping)



Indexstruktur besteht aus Schlüsselteil (Index) und Datenteil

Graphische Struktur



Index

ermöglicht den effizienten Zugriff auf die Daten

Daten

enthalten die gespeicherte Information (vergleiche externe Knoten)

2 Knotenarten

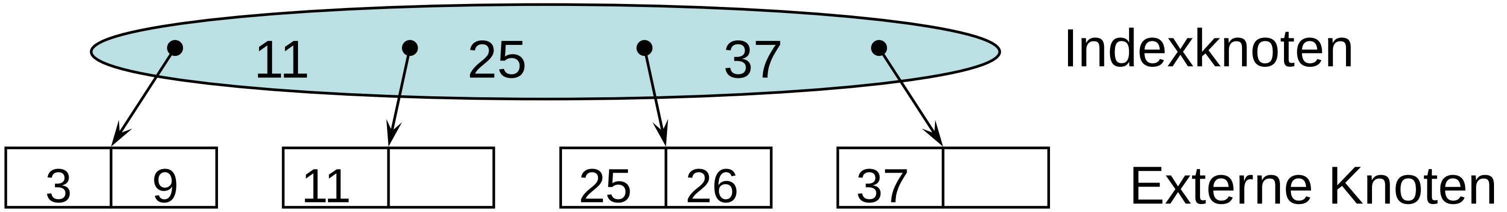
Indexknoten

- Erlauben effizienten Zugriff auf die externe Knoten
- Definieren die Intervallbereiche der Elemente, die im zugeordneten Teilbaum gespeichert sind.
- realisiert durch interne Knoten

Externe Knoten

- Enthalten den durch Schlüssel identifizierten **Datensatz (key, info)**
  - vergl. mit **Dictionary**
- realisiert durch Blattknoten
  - Im Beispiel verwenden wir nur Schlüsselwert, repräsentiert aber ganzen Datensatz

Beispiel



Höhenbalanzierter (perfektbalanzierter) Mehrwegbaum

Eigenschaften

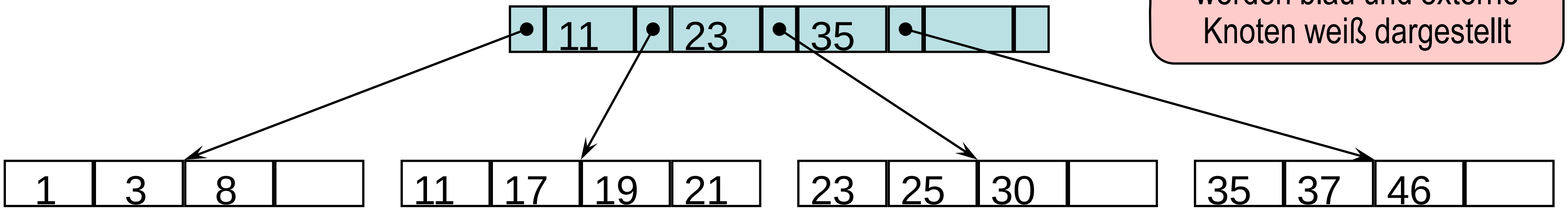
- Externspeicher-Datenstruktur
  - Eine der häufigsten Datenstrukturen in Datenbanksystemen
- Dynamische Datenstruktur (Einfügen und Löschen)
- Algorithmen für Einfügen und Löschen erhalten die Balanzierungs-eigenschaft
- Garantiert einen begrenzten (worst-case) Aufwand für Zugriff, Einfügen und Löschen
- Der Aufwand für die Operationen Zugriff, Einfügen und Löschen ist bedingt durch die Baumstruktur von der Ordnung log n (O(log n))
- Besteht aus Index und Daten
- Der Weg zu allen Daten ist gleich lang

Definition B+-Baum der Ordnung k

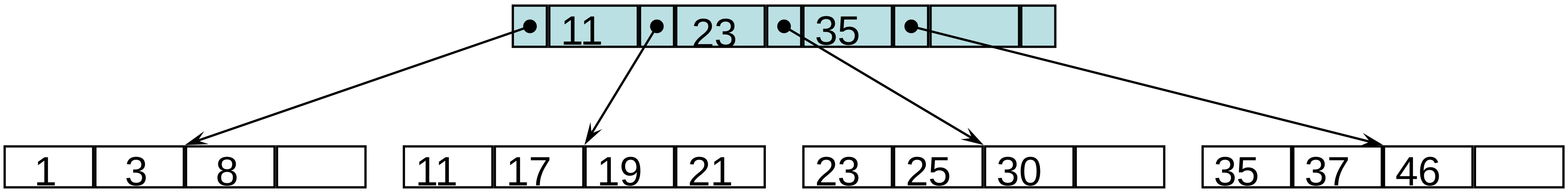
- Alle Blattknoten haben die gleiche Tiefe (gleiche Weglänge zur Wurzel)
- Die Wurzel ist entweder ein Blatt oder hat mind. 2 und max. 2k+1 Kinder
- Jeder (interne) Knoten besitzt mindestens k und maximal 2k Schlüsselwerte, daher mindestens k+1 und maximal 2k+1 Kinder
  - Für jeden Indexeintrag gilt, dass im linken Unterbaum nur Werte gespeichert sind, die kleiner sind als der Indexeintrag und im rechten Unterbaum nur Werte, die größer oder gleich dem Indexeintrag sind.
  - Intervallbasierte Suchdatenstruktur

Beispiel: B+-Baum der Ordnung 2

Anmerkung: Indexknoten werden blau und externe Knoten weiß dargestellt

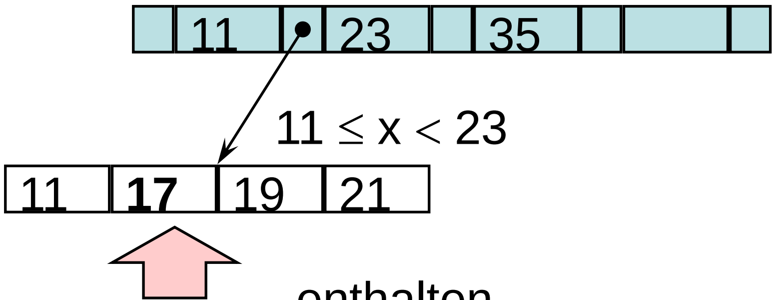


- Jeder Indexknoten hat mindestens 2 und maximal 4 Schlüsselwerte und folglich mindestens 3 und maximal 5 Kinder
- Ausnahme ist die Wurzel, kann auch nur 1 Schlüsselwert mit 2 Kindern enthalten
- Die Größe der externen Knoten ist eigentlich durch die Ordnung nicht definiert, wird aber üblicherweise in der Literatur gleichgesetzt



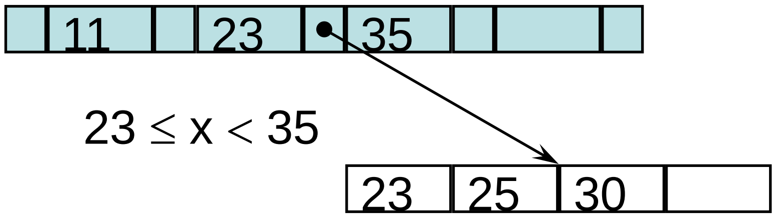
Suchen eines Datensatzes mit gegebenen Schlüssel

Datensatz mit Schlüssel 17



enthalten

Datensatz mit Schlüssel 27



nicht enthalten **?**

Aufwand der Suche

Höhe eine B^+-Baumes der Ordnung k mit *Datenblockgröße* b (mind. b, max 2b Elemente) ist maximal $\log_{k+1}(n/b)$.

Bereichsabfrage möglich

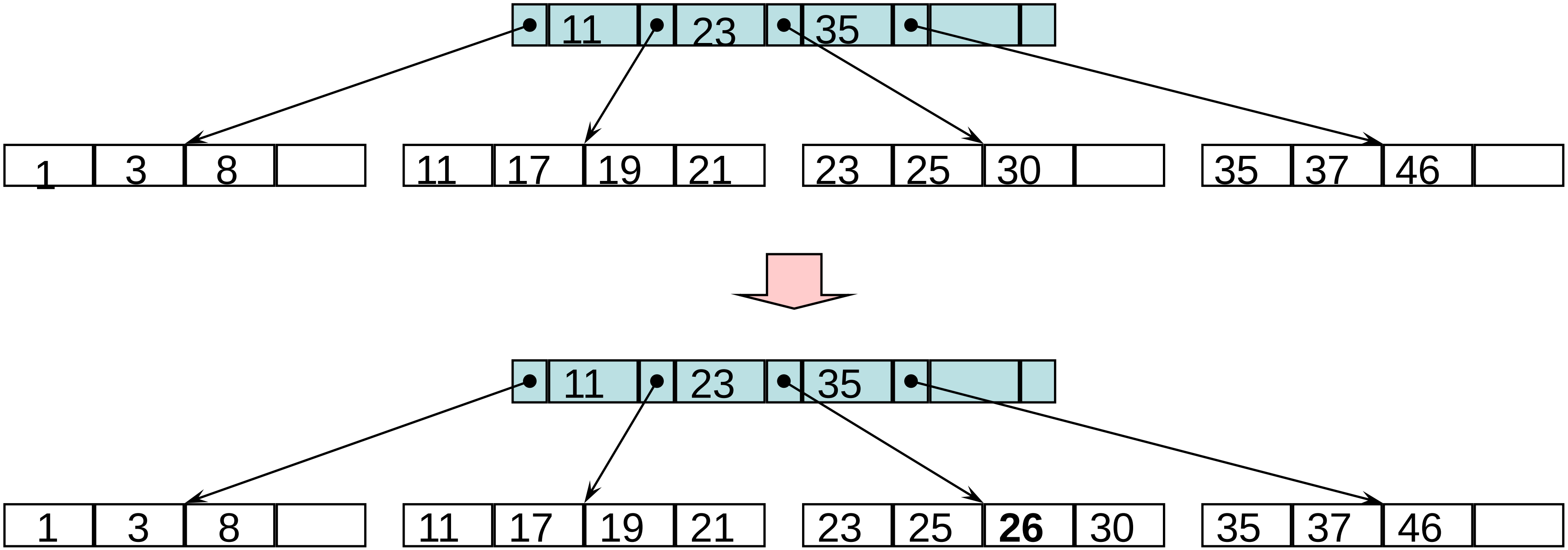
Suche alle Element im Bereich [Untergrenze, Obergrenze]

B^+-Baum Einfügen (1)

Einfügen bedeutet hier, dass ein Datensatz mit identifizierenden Schlüssel in einen externen Knoten eingefügt wird, wodurch gegebenenfalls Indexeinträge angepasst werden müssen

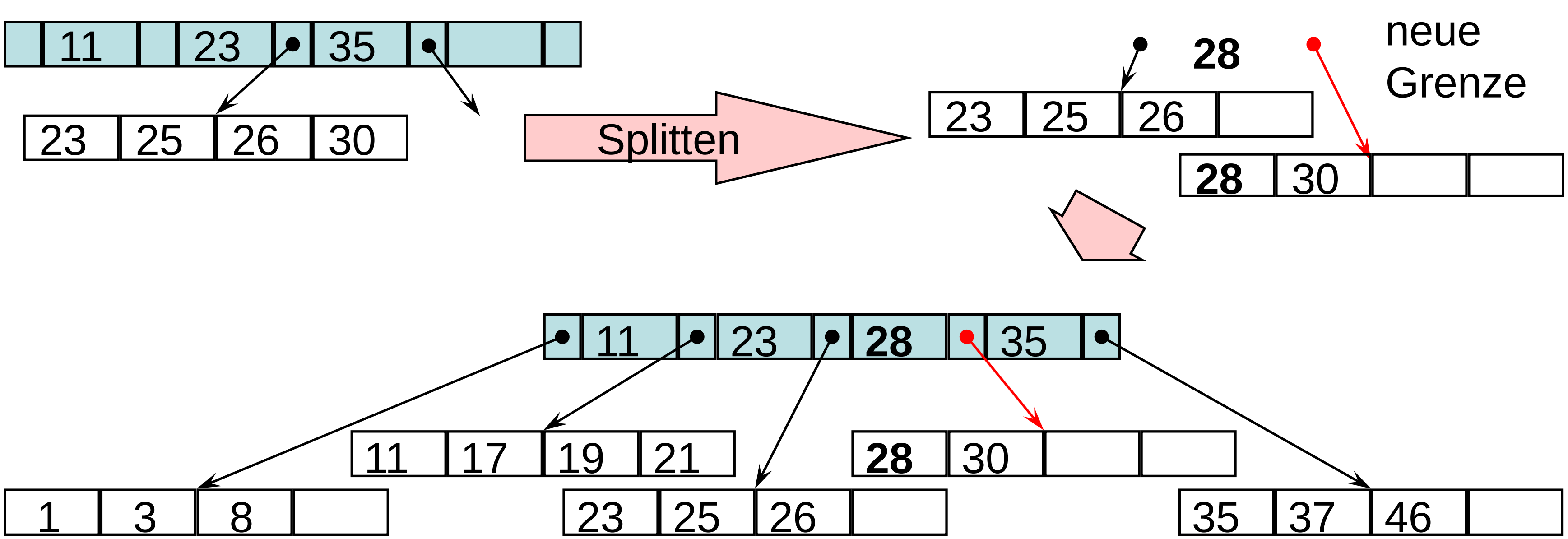
Fall 1: Einfügen Datensatz mit Schlüssel 26

Platz im externen Knoten vorhanden, einfaches Einfügen



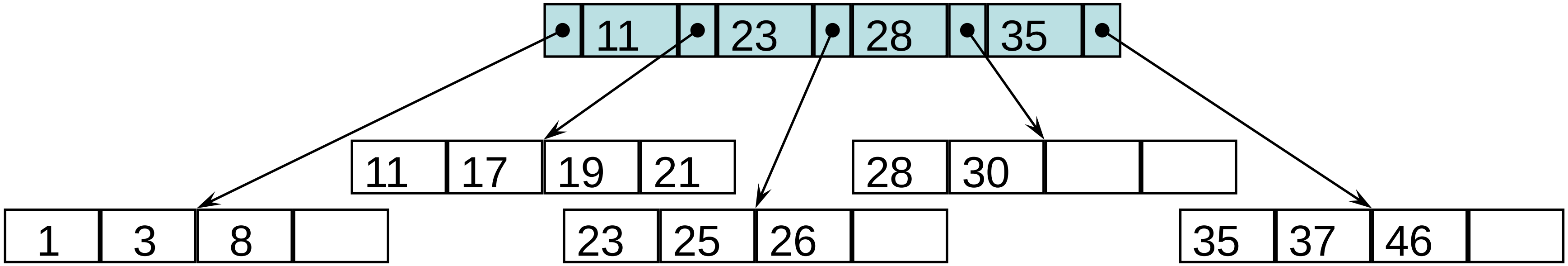
Fall 2: Einfügen Datensatz mit Schlüssel 28

Kein Platz mehr im externen Knoten (**Überlauf, Overflow**), externer Knoten muss geteilt werden ⇒ **Split**



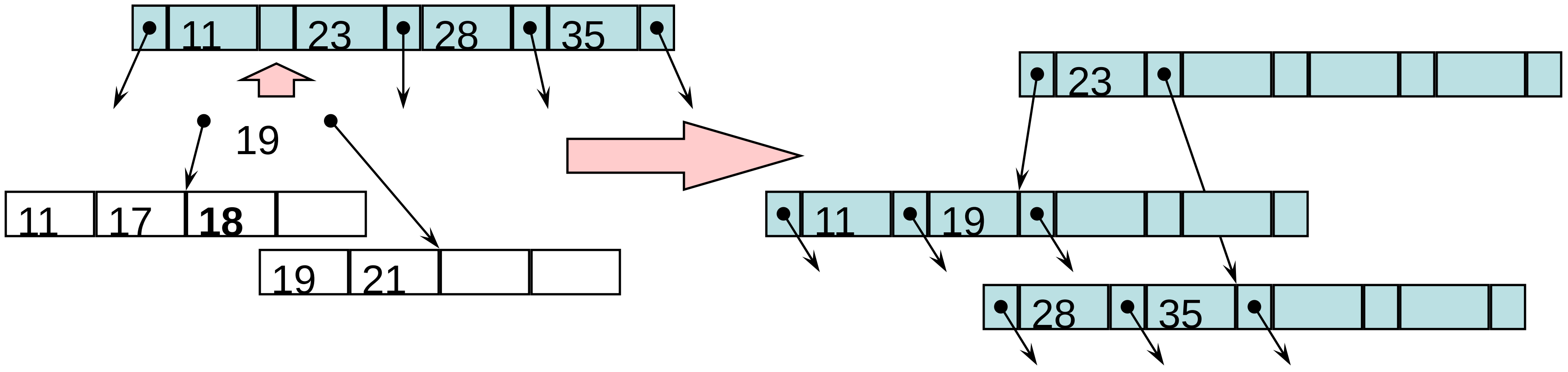
Fall 3: Einfügen Datensatz mit Schlüssel 18

Kein Platz mehr im Knoten und kein Platz mehr im darüber liegenden Indexknoten, Indexknoten muss gesplittet werden



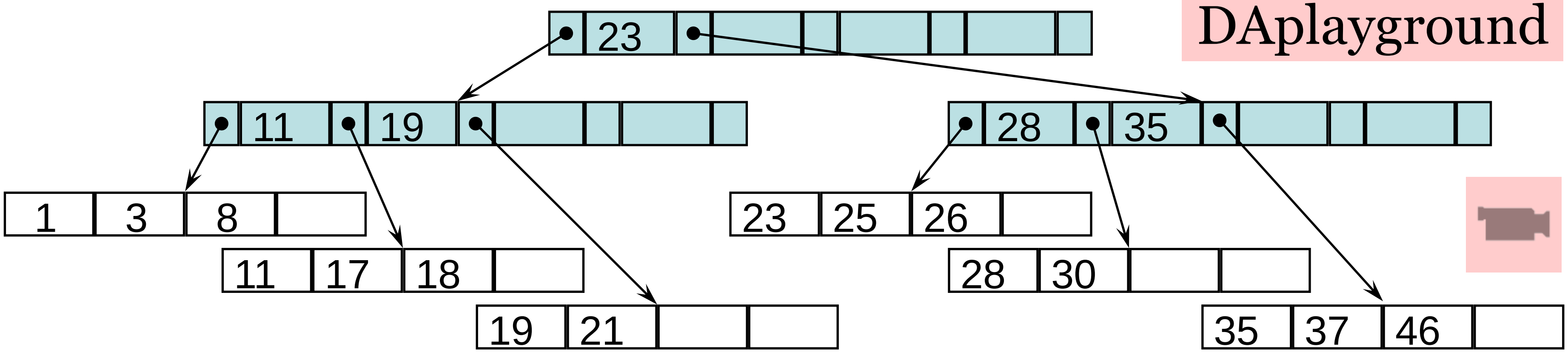
B^+-Baum Einfügen (4)

Beim Splitten eines Indexknoten wird das mittlere Indexelement im darüberliegenden Indexknoten eingetragen. Falls der nicht existiert (*Wurzelsplit*), wird eine neue Wurzel erzeugt



B^+-Baum Einfügen (5)

Der resultierende Baum hat folgendes Aussehen



DAplayground

Unterscheidung des Splits für externe und interne Knoten

Externe Knoten: 2k+1 Elemente werden auf 2 benachbarte externe Knoten aufgeteilt

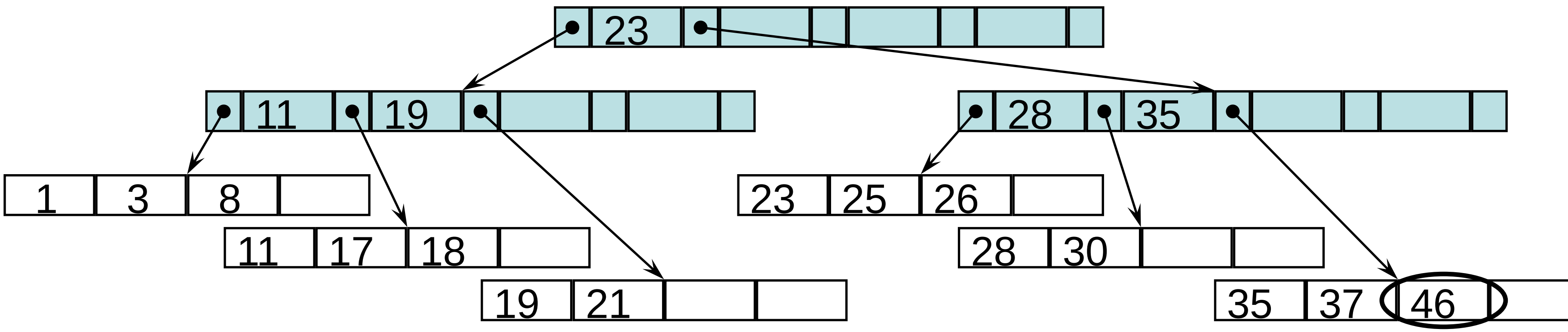
Interne Knoten: 2k+1 Indexelemente (Schlüsselwerte) werden auf 2 benachbarte Indexknoten zu je k Elemente aufgeteilt, das mittlere Indexelement wird in die darüberliegende Indexebene eingetragen.

Falls keine darüberliegende Ebene existiert, wird eine neue Wurzel erzeugt, der Baum wächst um eine Ebene ("Baum wächst von den Blättern zur Wurzel"), der Zugriffsweg zu den Daten erhöht sich um 1.

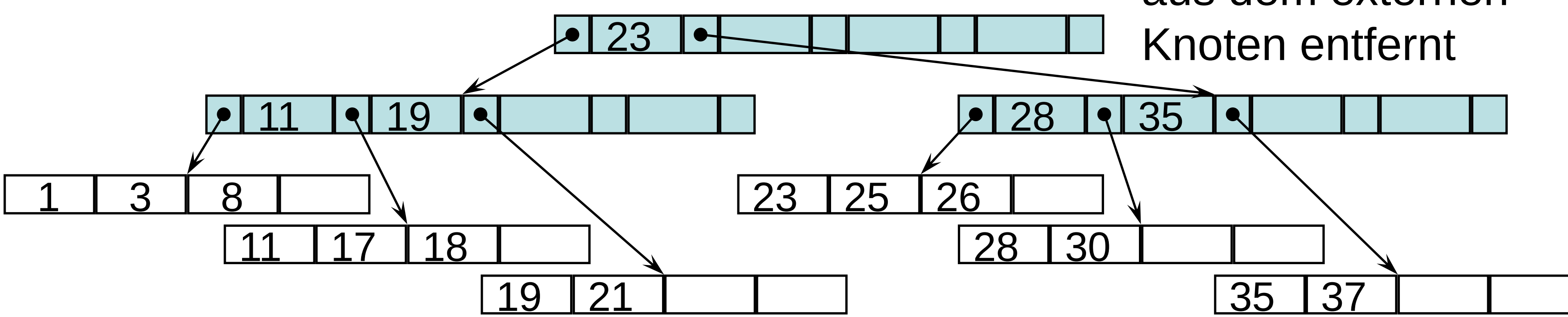
B^+-Baum Löschen (1)

Fall 1: Entfernen von Datensatz mit Schlüssel 46

Externer Knoten nach Löschen nicht unterbesetzt



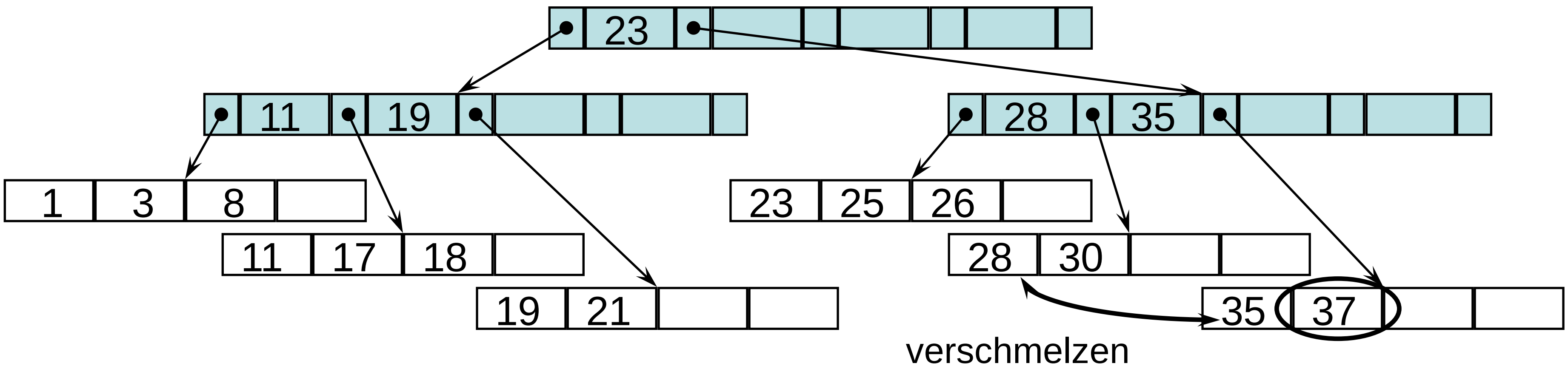
Datensatz wird einfach aus dem externen Knoten entfernt



B^+-Baum Löschen (2)

Fall 2: Entfernen von Datensatz mit Schlüssel 37

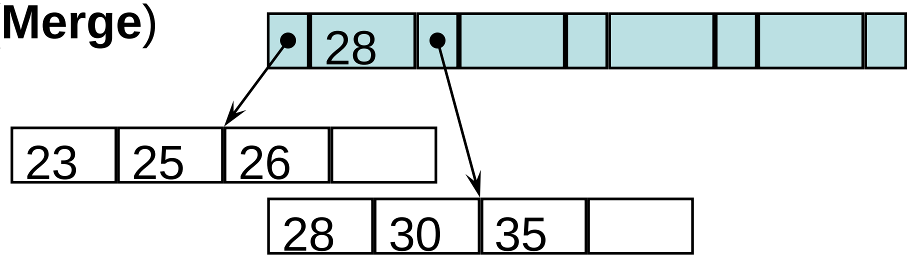
Externer Knoten nach dem Löschen unterbesetzt



Element wird entfernt, ist Knoten unterbesetzt (**Underflow**), Elementanzahl < k

Umgekehrter Vorgang zum Split $\Rightarrow$ benachbarte Knoten werden **verschmolzen**

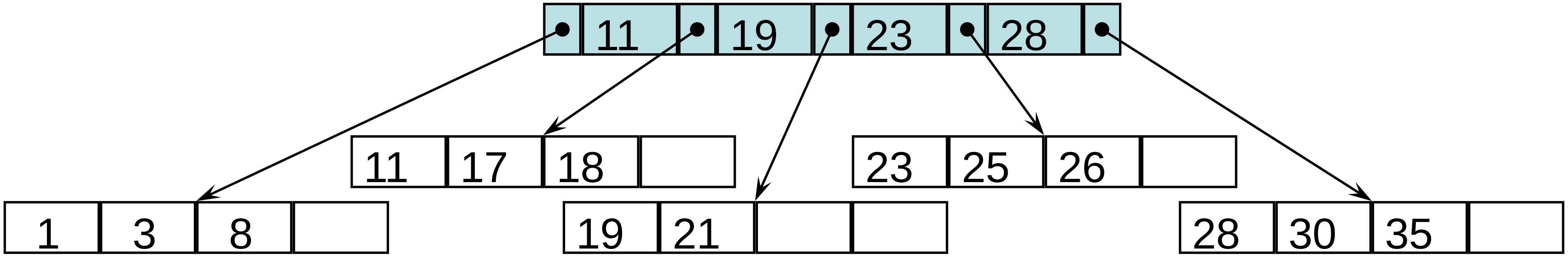
(**Merge**)



Interner Knoten unterbesetzt, muss ebenfalls mit Nachbarn verschmolzen werden.

B+-Baum Löschen (3)

Das Verschmelzen kann zur Verringerung der Baumhöhe führen, 2 Knoten werden zur neuen Wurzel verschmolzen, alte Wurzel wird entfernt.

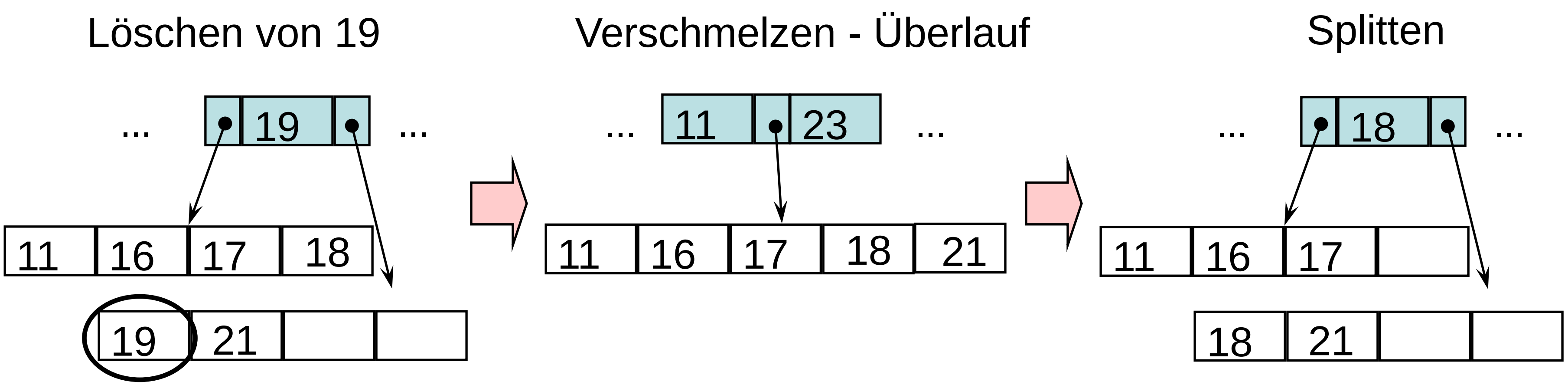


B^+-Baum Löschen (4)

Anmerkung

Das Verschmelzen zweier Knoten kann zu einem Überlauf (analog Einfügen) des neuen Knoten führen, was einen nachfolgenden Split notwendig macht.

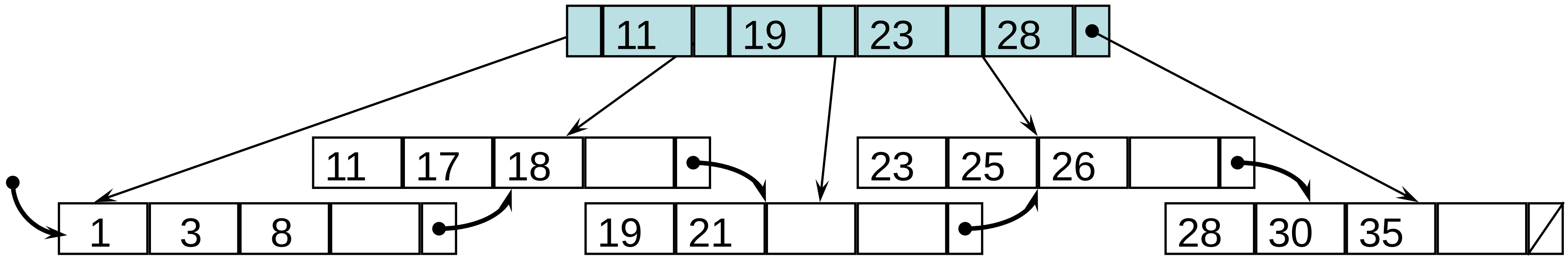
Beispiel: (Baumausschnitt)



Diese (Verschmelzen-Splitten) Sequenz erzeugt eine besseren Aufteilung der Datensätze zwischen benachbarten Knoten. Dies wird auch hier (siehe 2-3-4 Baum, eigentlich fälschlicherweise) als Datensatzverschiebung (shift) bezeichnet.

Datenblockverkettung

In der Praxis werden die externen Knoten (Datenblöcke) linear verkettet, um einen effizienten, sequentiellen Zugriff in der Sortierreihenfolge der gespeicherten Elemente zu ermöglichen



Analyse B+ - Baum

Datenverwaltung

- Einfügen und Löschen unterstützt

Datenmenge

- unbeschränkt
  - abhängig von der Größe des vorhandenen Speicherplatzes

Modelle

- **Externspeicherorientiert**
- Unterstützung komplexer Operationen
  - Bereichsabfragen, Sortierreihenfolge

Analyse B+ - Baum (2)

Speicherplatz	O(n)
Split, Verschmelzen	O(1)
Zugriff	O(log n)
Einfügen	O(log n)
Löschen	O(log n)
Sortierreihenfolge	O(n)

Garantiert!

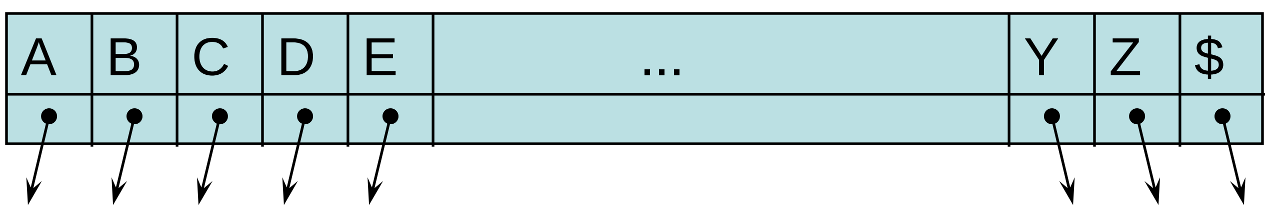
Bitte beachten: Ordnung des Baumes ist konstanter Faktor!

Aufgrund dieser Eigenschaften und der Fokussierung auf externe Speichermedien ist der B+ -Baum sehr gut für den **Einsatz in Datenbanksystemen** geeignet

5.7 Trie

Ein Trie ist ein digitaler Suchbaum oder Präfixbaum

- Dient zur Speicherung und Suche von Zeichenketten
  - Verwaltung von Wörterbüchern, Telefonbüchern, Spellcheckern, etc.
  - Bezeichnung wird abgeleitet von “retrieval”, wird aber wie “try” ausgesprochen
- Bei k Buchstaben ein “k+1-ary Tree”, wobei jeder Knoten durch eine Tabelle mit k+1 Kanten auf Kinder repräsentiert wird
- Ein Pfad beginnend bei der Wurzel bis zu einem Blatt im Trie stellt eine Zeichenkette dar
- Kann im weitesten Sinn auch als Intervallbasierte Suchdatenstruktur interpretiert werden, da Intervalle [A,B[, [B,C[, …

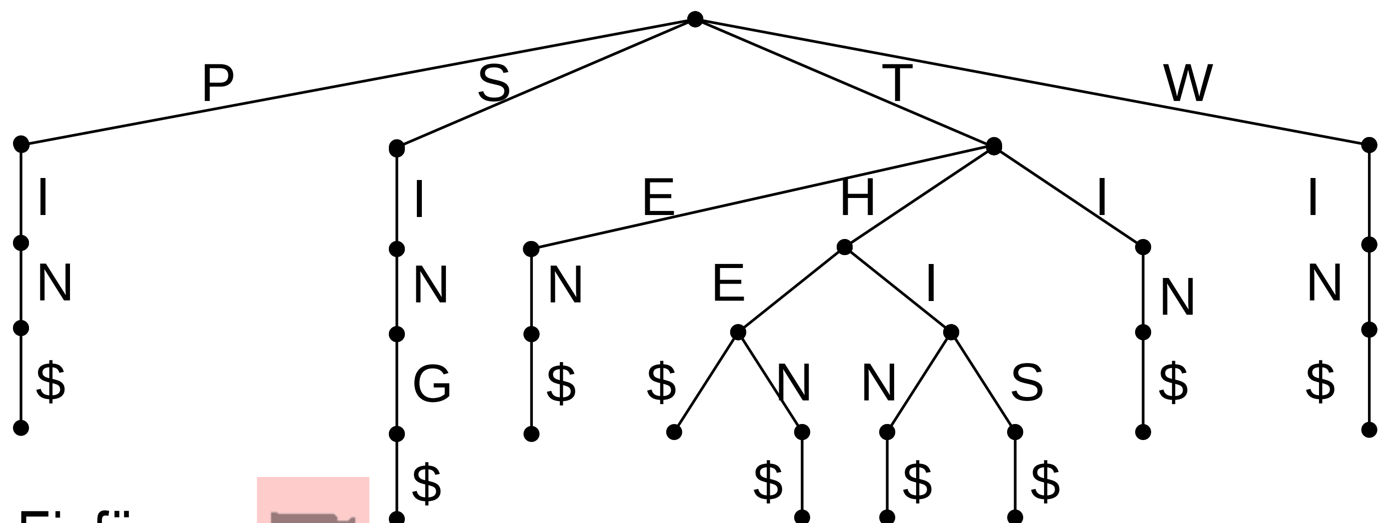


Spezialformen

- Patricia Tree, de la Briandais Tree

Trie

TEN
THE
THEN
THIN
THIS
TIN
SING
PIN
WIN

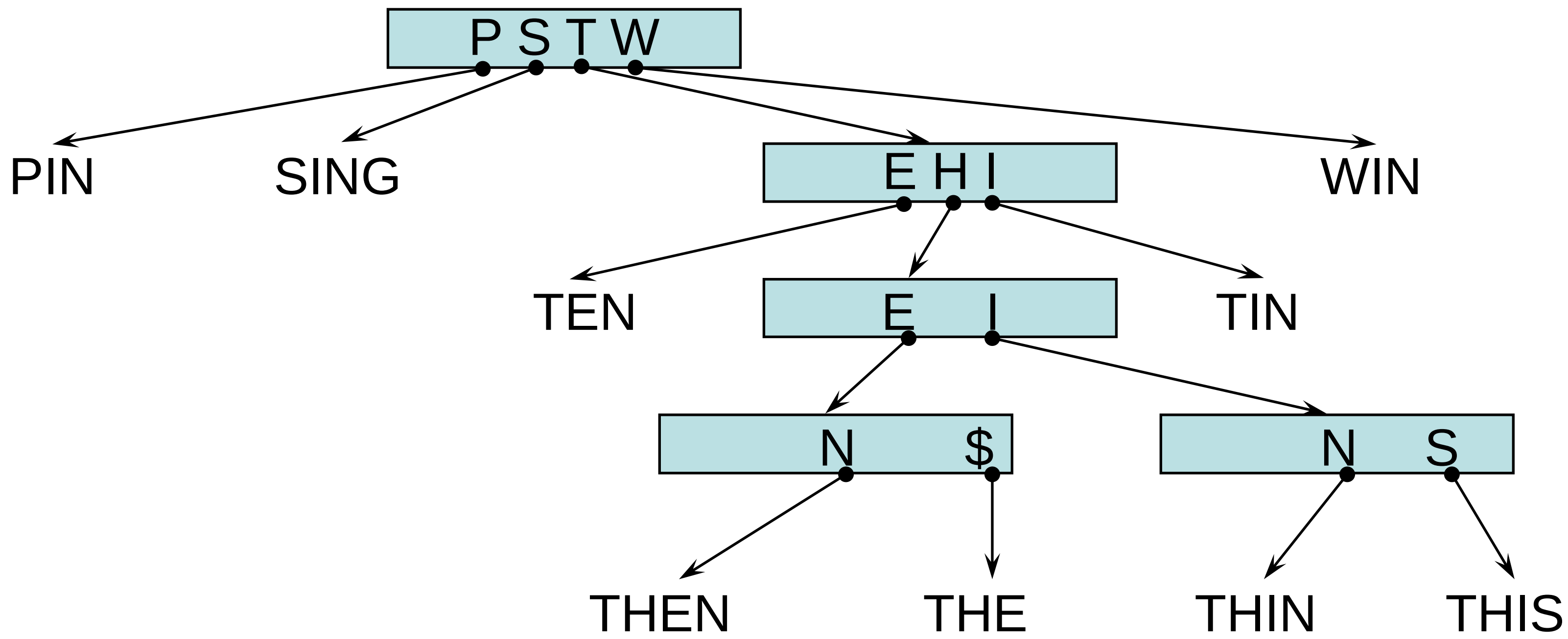


Einfügen

Suchen

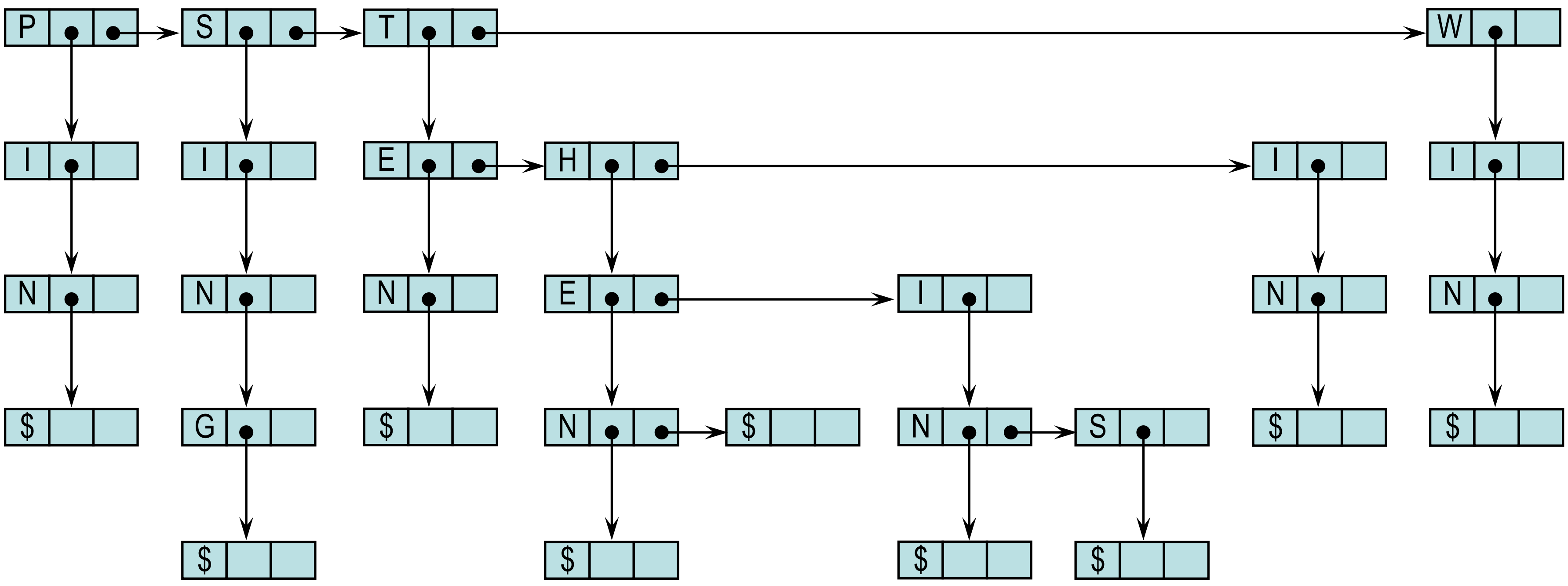
*P*ractical *A*lgorithm *to* *R*etrieve *I*nformation *C*oded *in* *A*lphanumeric

Ziel ist die Komprimierung des Tries



Listen-Repräsentation eines Tries

- statt der Tabellen im Knoten lineare Liste
  - Knoten besteht aus 2 Kinderkomponenten
    - Nächster Wert - Nächster Level



Analyse Trie

Datenverwaltung

- Einfügen und Löschen unterstützt
- Struktur des Tries unabhängig von der Einfügereihenfolge

Datenmenge

- unbeschränkt

Modelle

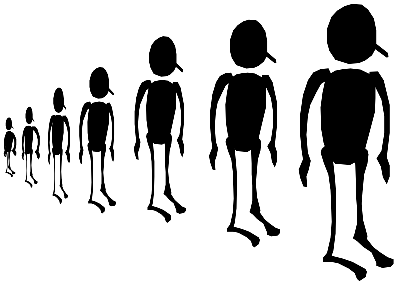
- Hauptspeicherorientiert
- Unterstützung komplexer Operationen
  - Bereichsabfragen, Sortierreihenfolge

Laufzeit

Speicherplatz	O(n)
Zugriff	O(log n)
Einfügen	O(log n)
Löschen	O(log n)
Sortierreihenfolge	O(n)

Bei der exakten Berechnung ist die Basis des Logarithmus abh. von der Kardinalität der Zeichenmenge, z.B. 26 (Grossbuchstaben), 2 (Bitstrings)

Datenstruktur zum wiederholten Finden und Entfernen des Elementes mit dem kleinsten (bzw. größten) Schlüsselwert aus einer Menge



Anwendungen

- Simulationssysteme (Schlüsselwerte repräsentieren Exekutionszeiten von Ereignissen)
- Prozessverwaltung (Prioritäten der Prozesse)
- Numerische Berechnungen (Schlüsselwerte entsprechen den Rechenfehlern, wobei zuerst die großen beseitigt werden)
- Grundlage für eine Reihe komplexer Algorithmen (Graphentheorie, Filekompression, etc.)
  - Anmerkung: Im folgenden betrachten wir o.B.d.A. nur Priority Queues die den Zugriff auf die kleinsten Elemente unterstützen

Operationen

Construct

Erzeugen einer leeren Priority Queue

IsEmpty

Abfrage auf leere Priority Queue

Insert

Einfügen eines Elementes

FindMinimum

Zurückgeben des kleinsten Elementes

DeleteMinimum

Löschen des kleinsten Elementes

Implementationsansätze

Ungeordnete Liste

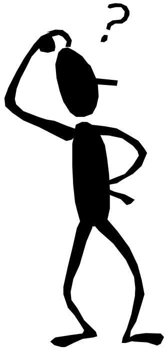
- Elemente werden beliebig in die Liste eingetragen (Aufwand O(1)).
- Beim Zugriff bzw. Löschen des ‘kleinsten’ Elementes muss die Liste abgesucht werden Aufwand → O(n).

Geordnete Liste

- Elemente werden in der Reihenfolge ihrer Größe eingetragen (Aufwand O(n)).
- Zugriff bzw. Löschen konstanter Aufwand → O(1)

Problem

- Aufwand O(n) schwer zu vermeiden.



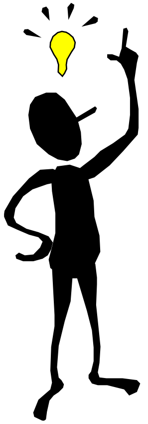
Priority Queue als Baumstruktur

Balanzierte Schlüsselbäume

- Einfügen im balanzierten Schlüsselbaum vom Aufwand O(log n)
- Zugriff realisieren durch Verfolgen des äußersten linken Weges im Baum (zum kleinsten Element) $\rightarrow$ Aufwand O(log n)

Günstigste Realisierung über ungeordnete, komplette Schlüsselbäume

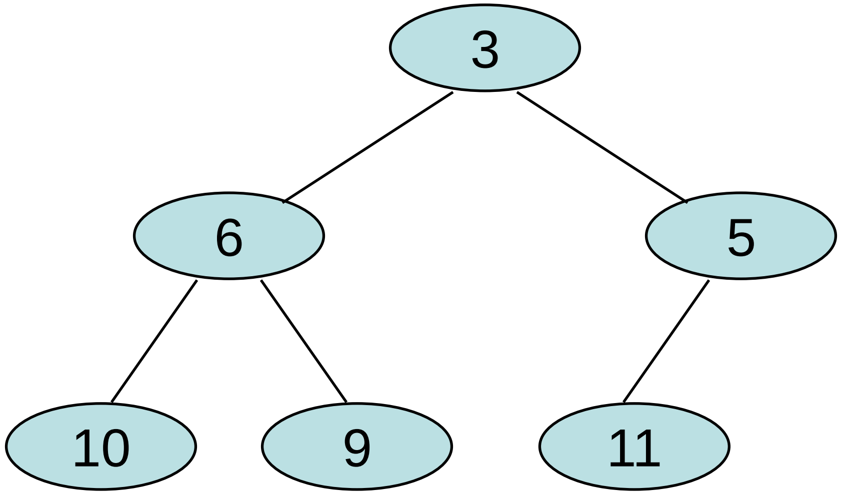
- mit der Eigenschaft, dass für alle Knoten die Schlüsselwerte ihrer Nachfolger größer (oder kleiner) sind



5.8.1 Heap

Wertemenge

3 6 5 10 9 11



Heap

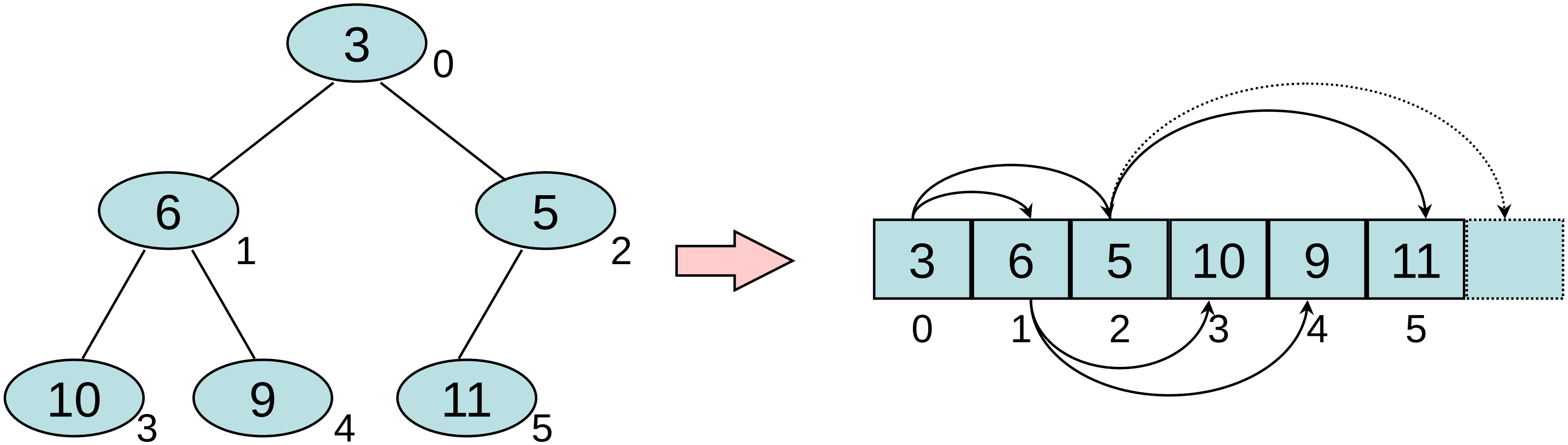
- Ungeordneter, binärer, kompletter Schlüsselbaum
- Wert jedes Knotens ist kleiner (größer) oder gleich den Werten seiner Nachfolgerknoten
- Wurzel enthält kleinstes (größtes) Element
- Anordnung der Unterbäume bez. Ihrer Wurzel undefiniert (ungeordneter Baum)

Realisierung eines Heaps effizient durch ein Feld

Die n Schlüsselwerte des Heaps können als Folge von Elementen x_0, x_1, x_2, … x_{n-1} interpretiert werden, wobei die Position der einzelnen Knotenwerte im Feld durch folgende Regel bestimmt wird:

- Wurzel in Position 0, die Kinder des Knotens i werden an Position 2i+1 und 2i+2 gespeichert.

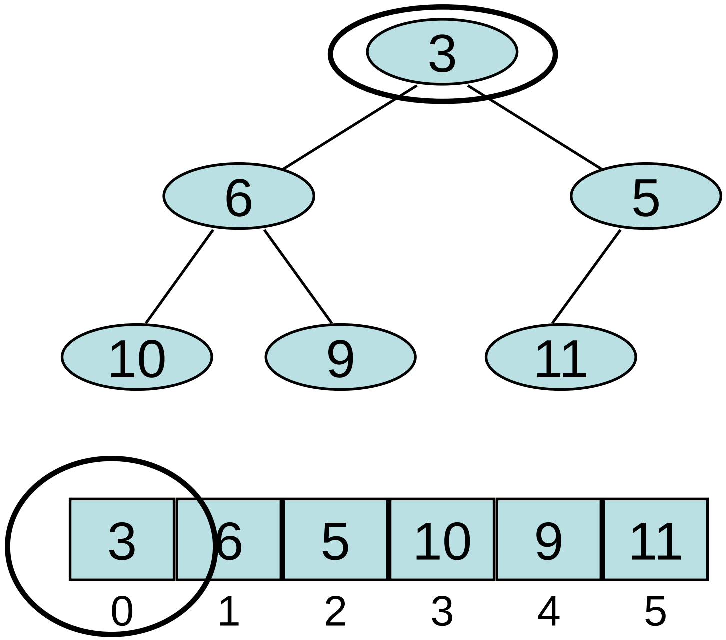
Beispiel



Zugriff

Zugriff auf das kleinste Element mit konstantem Aufwand: O(1)

→ Wurzel = erstes Element im Feld.



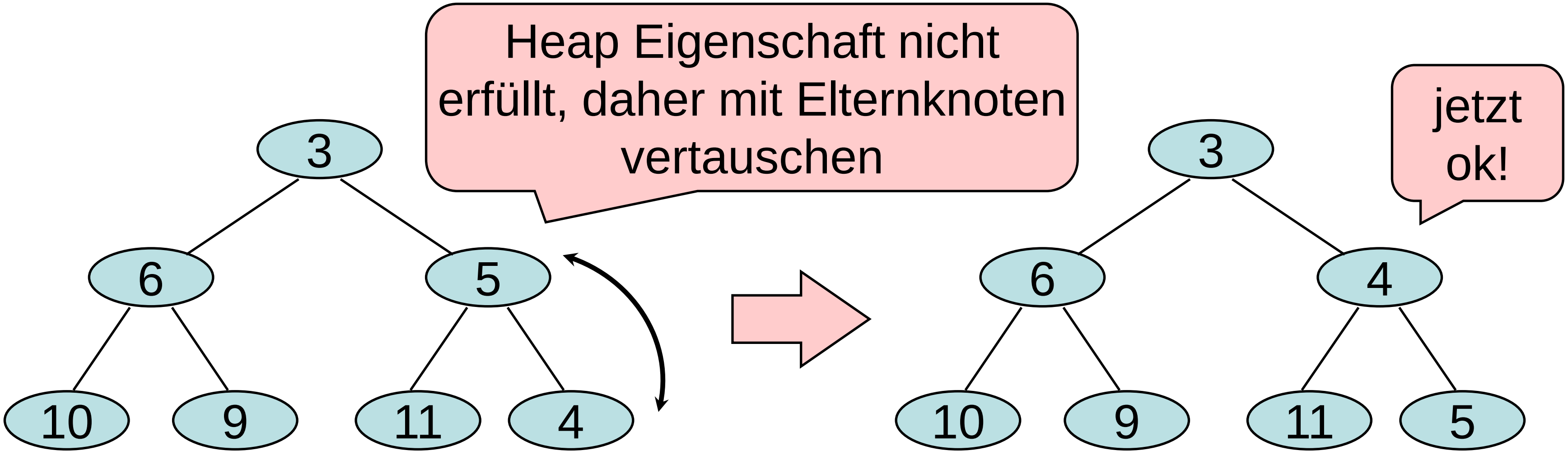
Klasse Priority Queue

```
class PQ {
  int *a, N;
public:
  PQ(int max) { a = new int[max]; N = -1; }
  ~PQ() { delete[] a; }
  int FindMinimum(){ return a[0]; }
  int IsEmpty() { return (N == -1); }
  void Insert(int);
  int DeleteMinimum();
  …
};
```

Einfügen in einen Heap

- Neues Element an der letzten Stelle im Feld eintragen
- Überprüfen, ob die Heap Eigenschaft erfüllt ist
- Wenn nicht, mit Elternknoten vertauschen und solange wiederholen bis erfüllt
- Eintragen von 4
  - (an der letzten Stelle im Feld)

Aufwand: O(log n)

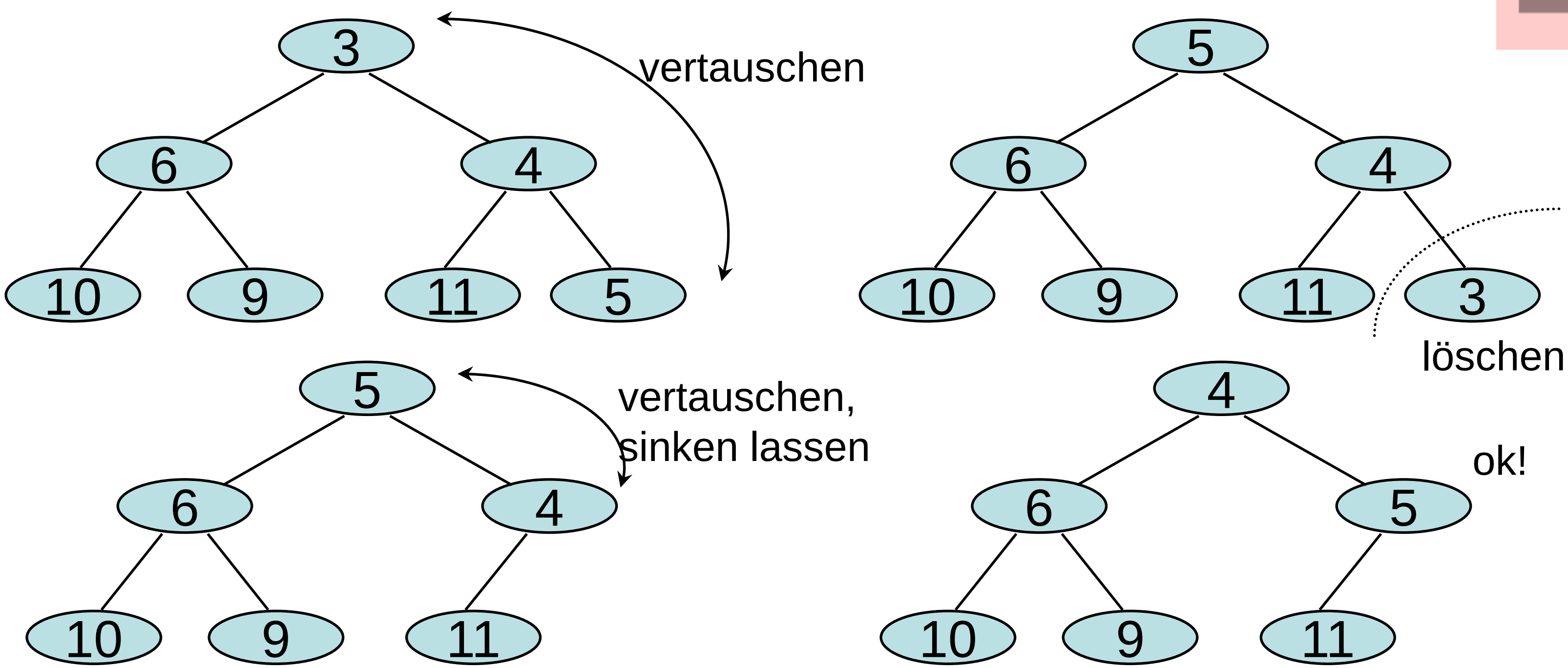


Methode Insert

```
void PQ::Insert(int v) {
  int child, parent;
  a[++N] = v;
  child = N;
  parent = (child - 1)/2;
  while(child != parent) {
    if(a[parent] > a[child]) {
      swap(a[parent], a[child]);
      child = parent;
      parent = (parent - 1)/2;
    } else break; // to stop the loop
  }
}
```

Löschen aus einem Heap

Wurzel mit äußerst rechten Blattknoten tauschen, diesen Blattknoten löschen, Wurzel in den Baum sinken lassen (mit kleinerem Kindknoten vertauschen), bis Heap Eigenschaft gilt.
Aufwand: O(log n)



Methode DeleteMinimum

```
int PQ::DeleteMinimum() {
  int parent = 0, child = 1;
  int v = a[0];
  a[0] = a[N];
  N--;
  while(child <= N) {
    if(a[child] > a[child+1]) child++;
    if(a[child] < a[parent]) {
      swap(a[parent], a[child]);
      parent = child;
      child = 2*child + 1;
    } else break; // to stop the loop
  }
  return v;
}
```

Analyse Heap

Datenverwaltung

- Einfügen und Löschen unterstützt

Datenmenge

- unbeschränkt

Modelle

- Hauptspeicherorientiert
- Nur simple Operationen

Laufzeit

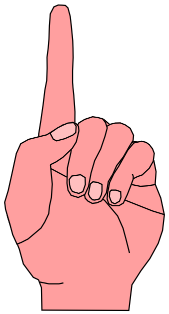
Speicherplatz	O(n)
Zugriff (Minimum/Maximum)	O(1)
Einfügen	O(log n)
Löschen	O(log n)

Der Heap stellt eine effiziente Basisdatenstruktur für viele weitere darauf aufbauende Datenstrukturen dar

Heaps und Heaps

Bitte zu unterscheiden! Der Begriff Heap wird in der Informatik zur Bezeichnung verschiedener Konzepte verwendet

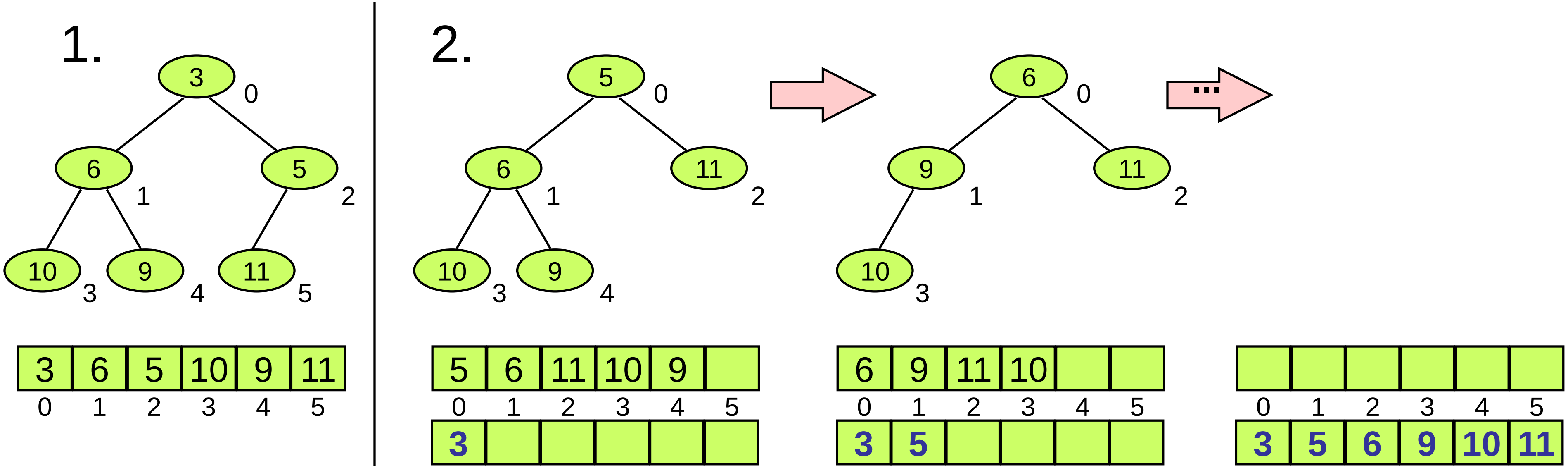
- Heap als Datenstruktur zur Speicherung von Priority Queues
- Heap als Speicherbereich zur Verwaltung (vom Benutzer selbst verwalteter) dynamisch allozierter Speicherbereiche
- Heap als Speicherform in Datenbanken



Eine Heap (Priority Queue) kann Basis zum Sortieren bilden (Williams 1964):

1. ein Element nach dem anderen in einen Heap einfügen
2. sukzessive das kleinste bzw. größte Element entfernt

die Element werden in aufsteigender bzw. absteigender Ordnung geliefert



Heapsort (modifizierter Ansatz)

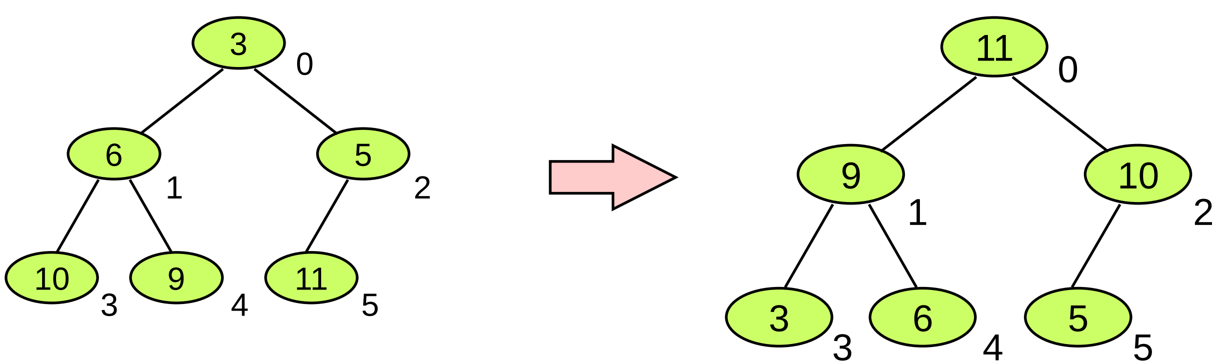
Kombination der beiden Arrays, Vermeidung von doppeltem Speicherplatz

6	9	11	10		
0	1	2	3	4	5
3	**5**				

⇨

6	9	11	10	**5**	**3**
0	1	2	3	4	5

Heap-Eigenschaft umdrehen: Wert jedes Knotens ist größer oder gleich den Werten seiner Kinderknoten



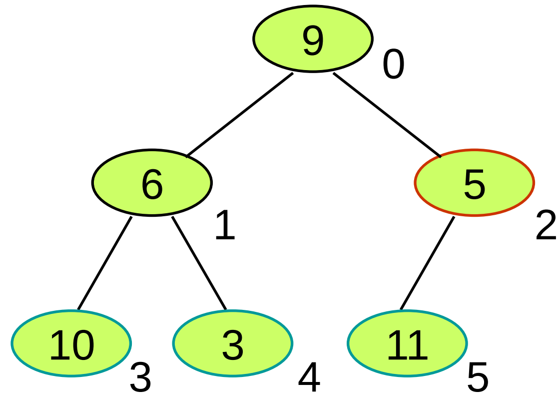
11	9	10	3	6	5
0	1	2	3	4	5

Beispiel: buildheap

Wertemenge

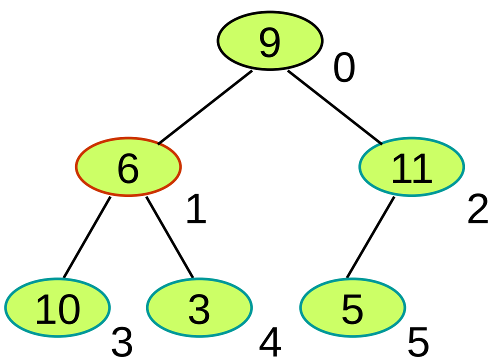
9 6 5 10 3 11

0	1	2	3	4	5
9	6	5	10	3	11

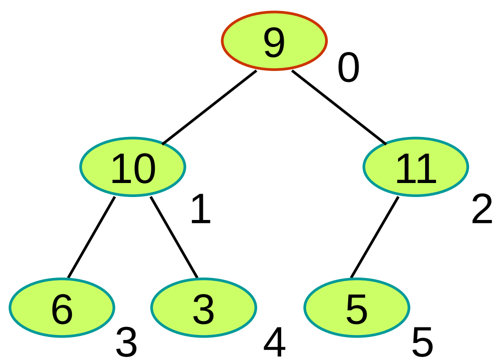


Idee: Heap-Eigenschaft erzeugen; für Blattknoten trivialerweise erfüllt

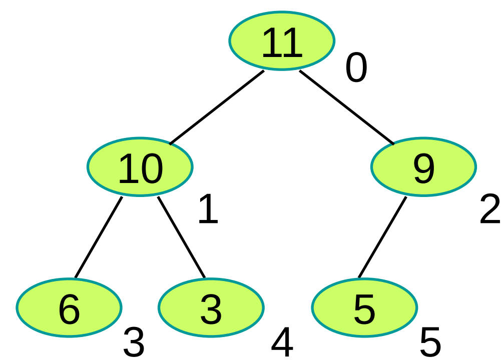
Nr. 2 erster zu prüfender Knoten, 11 passt nicht, in Baum sinken lassen, d.h. 5 mit 11 tauschen



Wert 6 (Nr. 1) passt nicht, in Teilbaum sinken lassen, d.h. mit größerem der Nachfolger vertauschen, d.h. 6 mit 10



Nr. 0 (9) in Baum sinken.

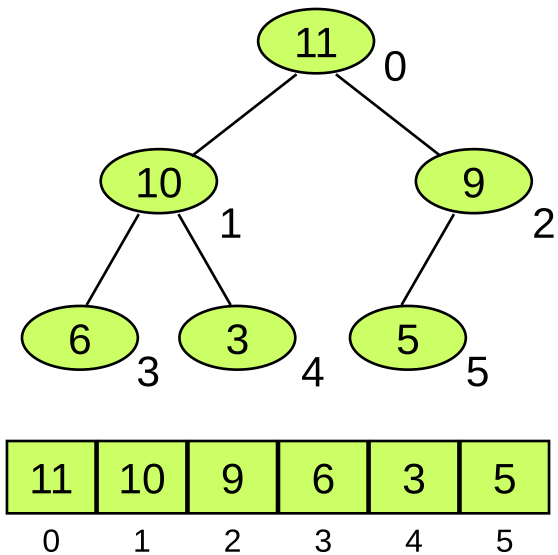


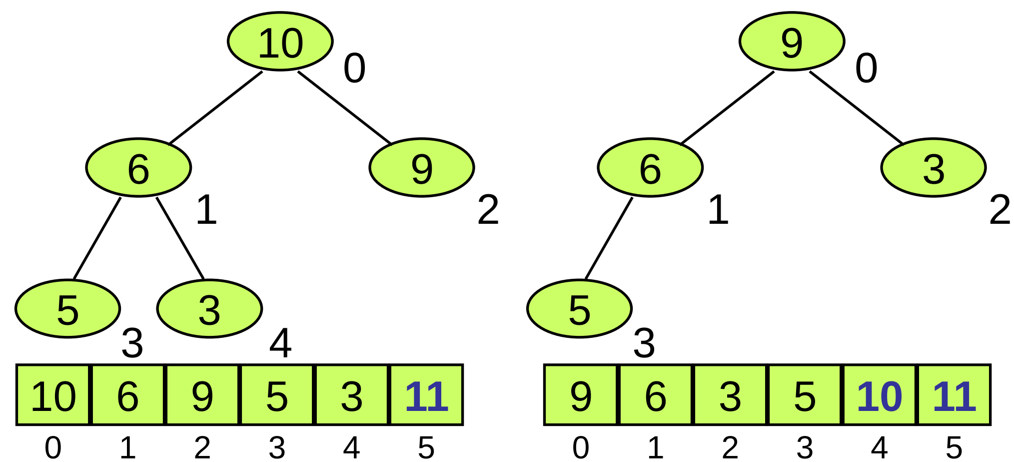
Heap-Eigenschaft erfüllt!

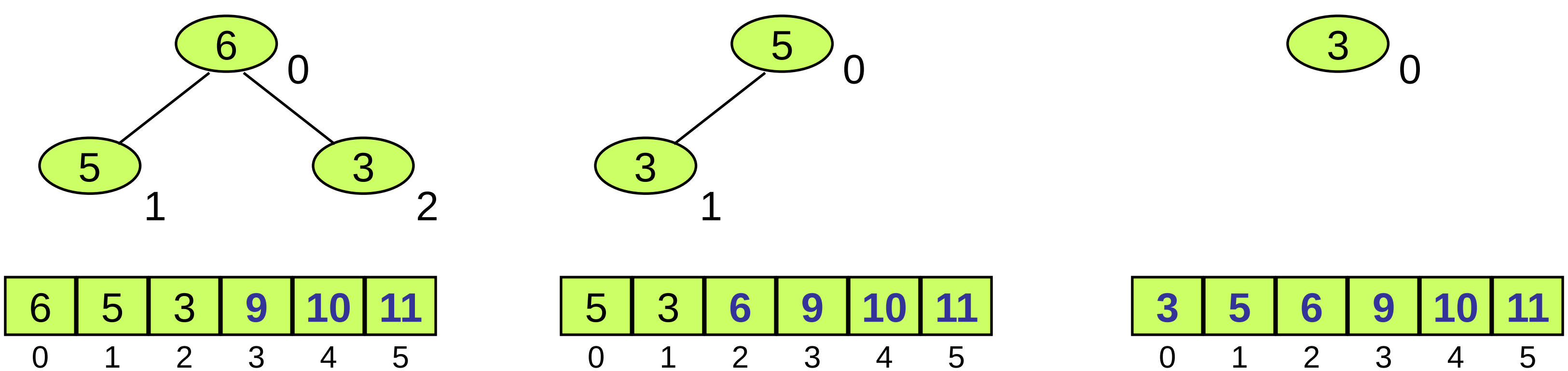
0	1	2	3	4	5
11	10	9	6	3	5

Beispiel: heapsort

Idee: Maximum "löschen" und am freiwerdenden Platz sichern







Heapsort Algorithmus (1)

```
void Vector::Heapsort() {
  int heapsize = Length();
  BuildMaxheap();
  for(int i = Length()-1; i >= 1; i--) {
    swap(a[0], a[i]);
    heapsize--;
    heapify(0, heapsize);
  }
}

void Vector::BuildMaxheap() {
  for(int i = Length()/2 - 1; i >= 0; i--)
    heapify(i, Length());
}
```

Heapsort Algorithmus (2)

```
void Vector::heapify(int i, int heapsize) {
  int left = 2*i + 1;
  int right = 2*i + 2;
  int largest;
  if (left < heapsize && a[left] > a[i])
    largest = left;
  else
    largest = i;
  if (right < heapsize && a[right] > a[largest])
    largest = right;
  if (largest != i) {
    swap(a[i], a[largest]);
    heapify(largest, heapsize);
  }
}
```

Eigenschaften der komplexen Verfahren

Quicksort

- Entartung zu $O(n^2)$ möglich
  - Wahl des Pivotelements!

Mergesort

- immer Laufzeit O(n * log(n))
- doppelter Speicherplatz notwendig

Heapsort

- keinen der obigen Nachteile
- aber höherer konstanter Aufwand

Was nehmen wir mit?

Baumstrukturen

- Notation
- spez. Eigenschaften, von O(n) auf O(log n)

Suchbäume

- Binäre Suchbäume
- AVL-Bäume
- 2-3-4 Bäume
- B^+-Baum
- Balanzierung

Trie

Priority Queues

- Heap