Baumstrukturen, Navigation, Graphen

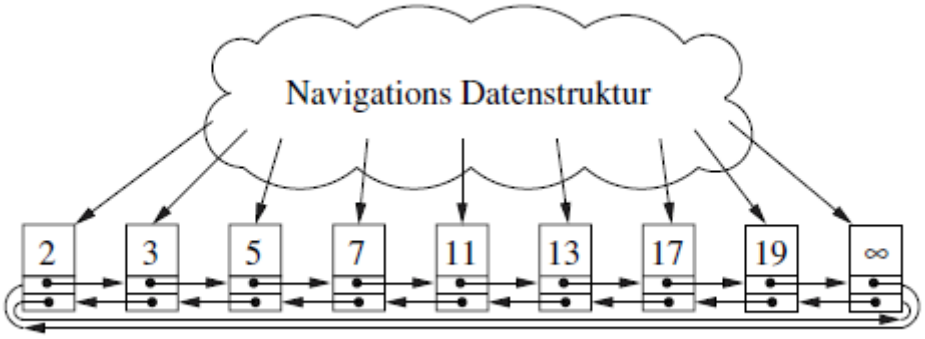
Themenverzeichnis

Baumstruktur-unterstützte sortierte Folgen

Die sortierten Folgen spielen offensichtlich eine wesentliche Rolle in zahlreichen praktischen Problemstellungen. Diesbezügliche Datenstrukturen sollten für die Navigation in der verwalteten Folge und für die Pflege ihres Inhaltes die folgenden Grundoperationen bereitstellen:

- $S.locate(k : Key)$:
  return Eintrag mit dem kleinsten Schlüssel in $\{e \in S : key(e) \geq k\}$.
- $S.insert(e : Element)$:
  Falls es ein $e' \in S$ mit $key(e') = key(e)$ gibt: $S := (S \setminus \{e'\}) \cup \{e\}$;
  andernfalls: $S := S \cup \{e\}$.
- $S.remove(k : Key)$:
  Falls es ein $e \in S$ mit $key(e) = k$ gibt: $S := S \setminus \{e\}$.

Die bisher thematisierten Ansätze (wie Arrays und verkettete Listen, ggf. kombiniert mit Hash-Tabellen) implementieren diese Operationen mit stark unterschiedlichen Zeitkomplexitäten, woraus sich das Bedürfnis nach einer zusätzlichen Optimierung ergibt.
Solche Optimierung kann erreicht werden, indem man z.B. die verkette Liste von Datenobjekten mit einer Baumstruktur verknüpft, die als „Navigation-leitender Überbau“ eingesetzt wird:



Somit wird bei einer Mutation (Einfügen/Löschen) mit Hilfe der Baumstruktur die Mutation-Stelle schnell gefunden und durch die verkettete Liste mit einem konstanten Zeitaufwand durchgeführt.
Die Effizienz von Baumstrukturen hängt allerdings wesentlich davon ab, wie gut *ausgeglichen* (*balanciert*) der jeweilige Baum ist.

Ein perfekt ausgeglichener binärer Baum ➔ in jedem Knoten beträgt der Gewichtunterschied zwischen seinen beiden Unterbäumen höchstens **1** .

Ein maximal unausgeglichener (= entarteter) Suchbaum kann z.B. entstehen, wenn eine sortierte Folge von **n** Elementen der Reihe nach hinzugefügt wird ➔ daraus resultiert die Höhe **n** und dementsprechend Zeitkomplexität **O(n)** für Suchvorgänge und Manipulationen.

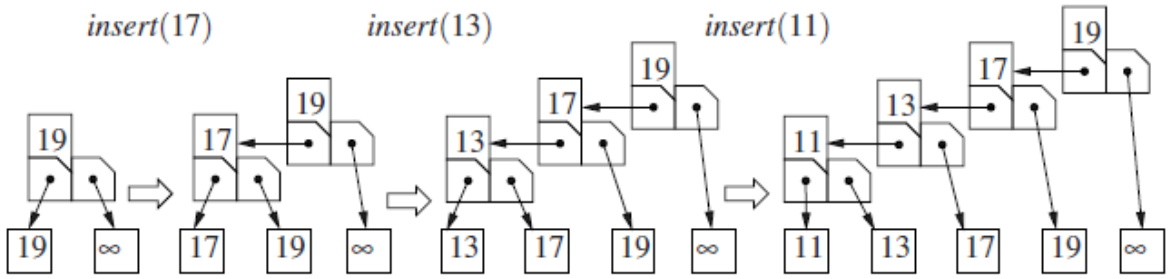
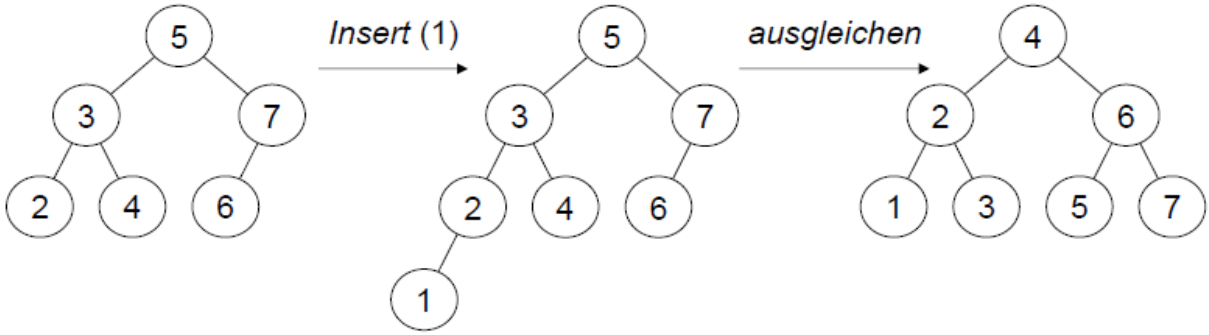


Abb. 7.4. Naives Einfügen von Einträgen in sortierter Reihenfolge liefert einen zu einer Liste entarteten Baum.

Ein perfekt ausgeglichener Suchbaum von insgesamt **n** Elementen hat ca. die Höhe $\mathbf{\log_2 n}$ ➔ Zeitkomplexität **O(log n)** beim Suchen. Das Einfügen/Löschen kann allerdings sehr aufwändig werden, weil u.U. sämtliche Elemente bewegt werden müssen, um die Ausgeglichenheit aufrechtzuerhalten:

Ausgeglichene Bäume

- Wie verhindert man, dass Suchbäume entarten?
- Jeweils ausgleichen ist zu aufwendig:



- … diese Folge zeigt, dass beim Ausgleichen eventuell jeder Knoten bewegt werden müsste!

(Die durchschnittliche Höhe von binären Bäumen liegt bei ca. $\mathbf{2 \log_2 n}$.)

AVL-Bäume

Eine bessere Alternative zu perfekt ausgeglichenen Bäumen entsteht, wenn das Ausgeglichenheitskriterium abgeschwächt wird ➔ es wird lediglich verlangt, dass in jedem Knoten der ***Höhen***unterschied zwischen seinen beiden Unterbäumen höchstens **1** beträgt.
Suchbäume, welche diese Einschränkung erfüllen, werden als AVL-Bäume klassifiziert.

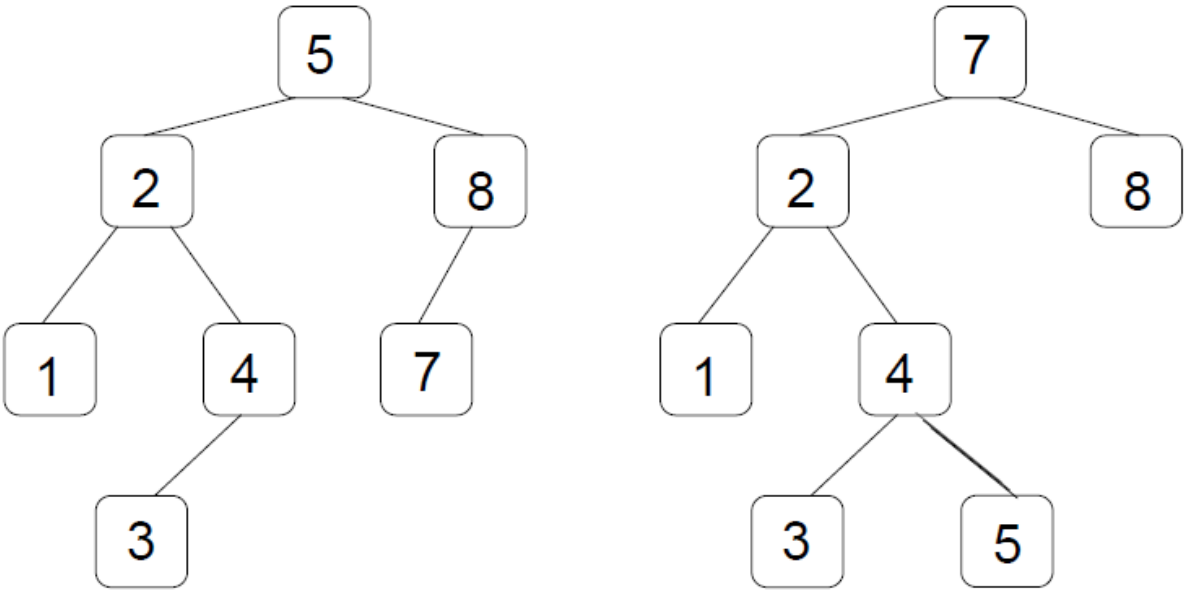
AVL-Bäume

- AVL für Adelson-Velskii und Landis (russische Mathematiker)
- binäre Suchbäume mit *AVL-Kriterium*:

  Für jeden (inneren) Knoten gilt: Höhe des linken und rechten Teilbaums differieren maximal um 1

- **Bemerkung:** Es reicht nicht, dieses nur für die Wurzel zu fordern!
  - Beide Teilbäume der Wurzel könnten entartet sein!

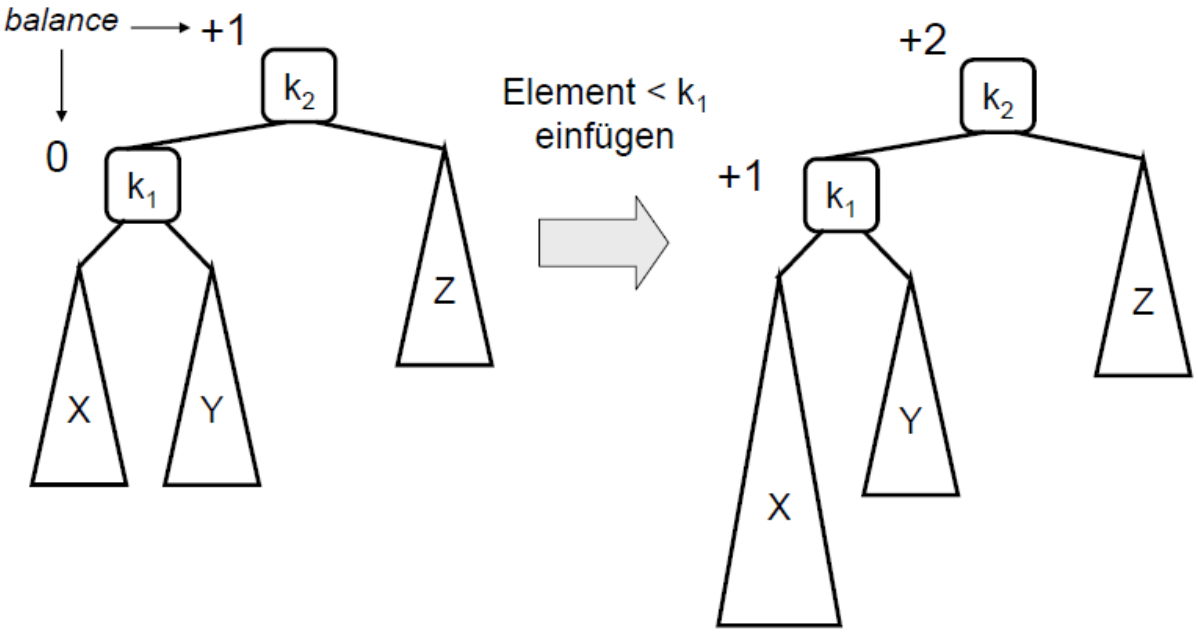
AVL-Eigenschaft am Beispiel



AVL nicht AVL

Einfügen in AVL-Bäume

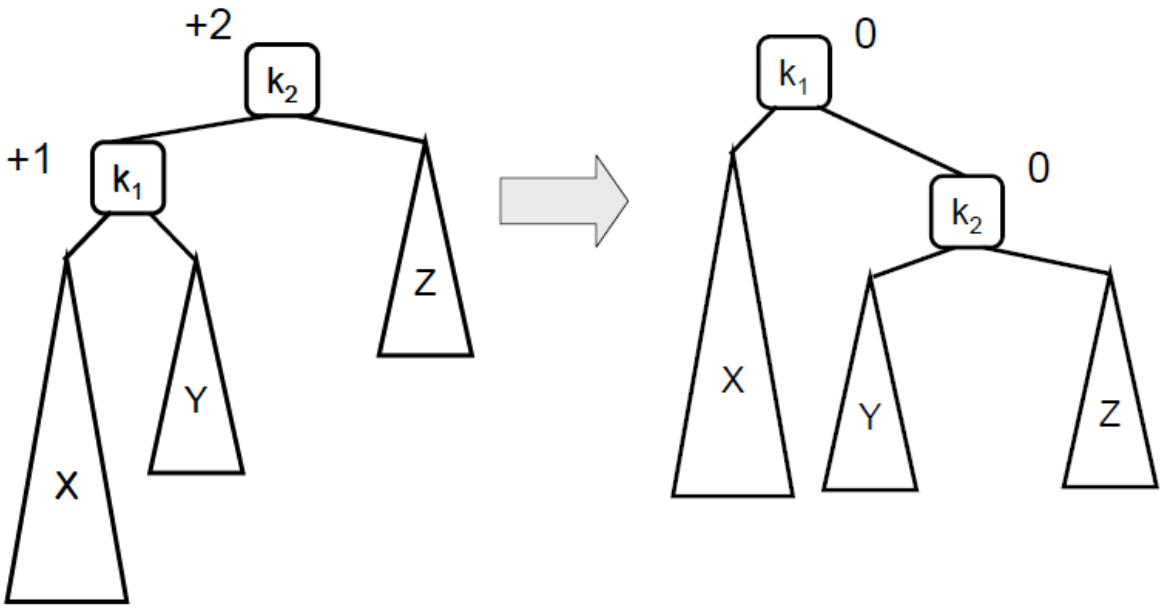
- Einfügen eines Schlüssels mit üblichem Algorithmus
- Danach kann die AVL-Eigenschaft verletzt sein:
  - *Balance* = *left.height* − *right.height*
  - AVL-Eigenschaft: *balance* $\in$ {−1, 0, +1}
  - nach Einfügen: *balance* $\in$ {−2, −1, 0, +1, +2}
- reparieren mittels *Rotation* und *Doppelrotation*



AVL Baum

Nicht AVL Baum

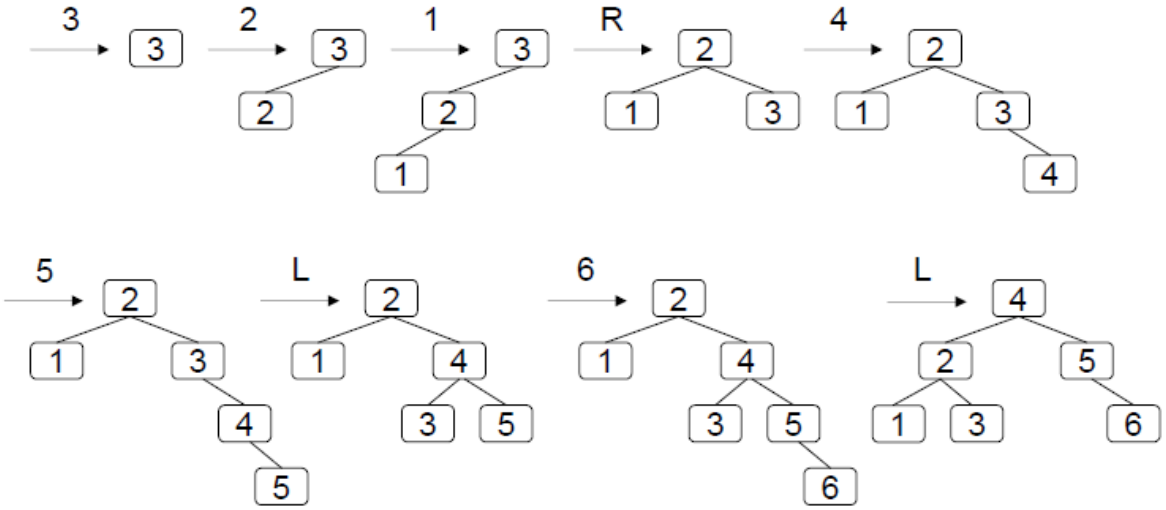
Einfache Rotation



Rotation mit linkem Kind nach rechts, analoge Operation nach links (spiegelbildlich)

Der Unterbaum **Y** des Knoten $\mathbf{k_1}$ (welcher nach oben verschoben wird) muss dabei unter $\mathbf{k_2}$ „umgehängt“ werden.

Aufbau eines AVL-Baums:




```
insert 3, 2, 1
       → einfache Rotation nach rechts (2,3)
insert 4, 5
       → einfache Rotation nach links (4,3)
insert 6
       → einfache Rotation (Wurzel) nach links (4,2)
...
```

In allen obigen Fällen kann die Verletzung des Gleichgewichts lokal korrigiert werden ➔ Zeitkomplexität **O(1)** (= konstant).

(a,b)-Bäume

Ein (a,b)-Baum wird durch folgende Eigenschaften definiert:

- jeder innere Knoten (d.h. ein Nicht-Blatt) ausser der Wurzel hat minimal **a** Kinder, die Wurzel darf weniger Kinder haben
- jeder Knoten hat maximal **b** Kinder
- ein innerer Knoten mit **k** Kindern beinhaltet **k – 1** Spaltschlüssel, wobei jeder Spaltschlüssel zwischen zwei benachbarten Kinderverweisen eingeordnet ist
- der Baum ist geordnet (ein Suchbaum)

Beispiel eines (a,b)-Baumes mit **a = 2**, **b = 4** :

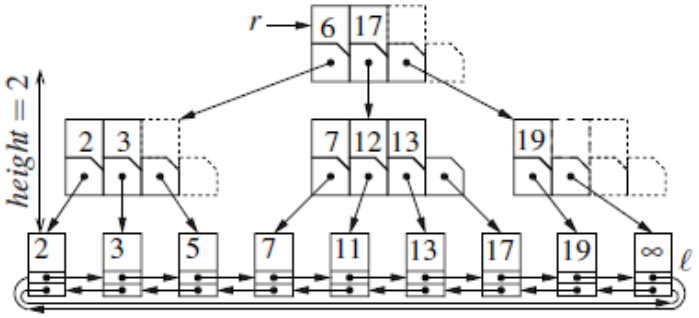
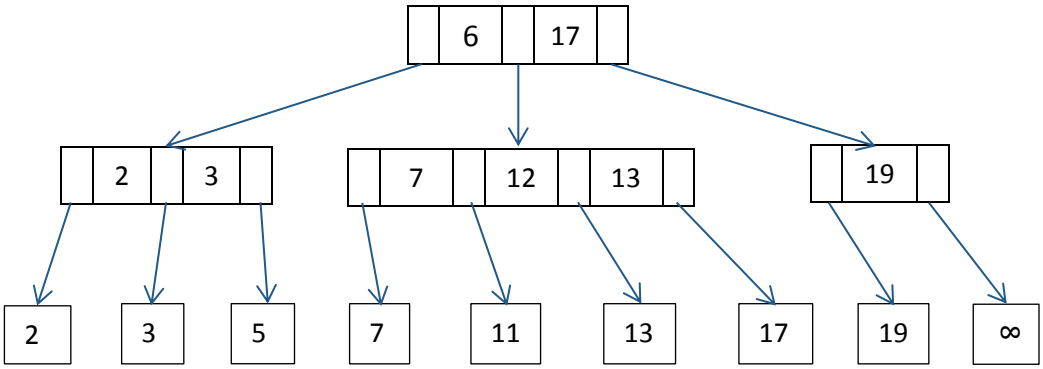


Abb. 7.5. Darstellung der Folge $\langle 2,3,5,7,11,13,17,19\rangle$ durch einen $(2,4)$-Baum. Der Baum hat Tiefe 2.

Im Rest dieses Skripts wird eine kompaktere Darstellungsart benutzt, mit der sich aus dem obigen Beispiel das folgende Diagramm ergibt:



B-Bäume

B-Bäume sind eine spezielle Ausprägung der oben beschriebenen Kategorie von (a,b)-Bäumen. Dabei wird die minimale Schlüsselzahl **a - 1** als Ordnung **m** des B-Baumes bezeichnet und die maximale Schlüsselzahl **b - 1** ist auf **2m** festgelegt. Eine weitere Einschränkung besteht darin, dass alle Blätter auf der gleichen Baumebene liegen.

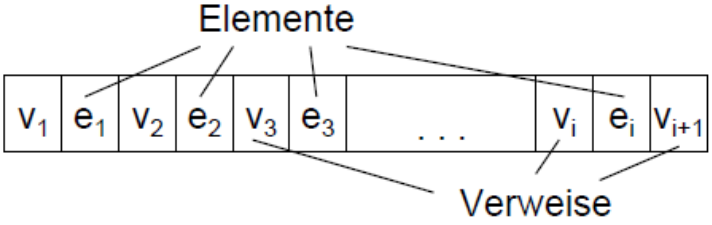
Der klassische Anwendungsbereich von B-Bäumen ist die Verwaltung von relationalen Datenbanken. In dieser Art von Anwendungen werden die Knoten von bei B-Bäumen *Seiten* genannt und die Schlüssel *Elemente.*
In den Blätterseiten wie auch in den inneren Seiten werden die Elemente mit Dateninhalten verknüpft, wobei ein Dateninhalt typisch die Adresse eines Datenblocks auf der Festplatte ist. (In bestimmten anderen Konzepten von (a,b)-Bäumen haben nur Blätter Dateninhalte, innere Knoten dienen ausschliesslich der Navigation).

Somit kann ein B-Baum kompakt wie folgt definiert werden:

Definition B-Baum

B-Baum der Ordnung *m*:

1. Jede Seite enthält höchstens 2*m* 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 (bzw. Verweise), falls *i* die Anzahl ihrer Elemente ist.
4. Alle Blattseiten liegen auf der gleichen Stufe.



Gemäss obiger Zeichnung bilden Verweise und Elemente eine alternierende Folge (sie beginnt und endet mit einem Verweis), die nach Schlüsselwerten sortiert ist.

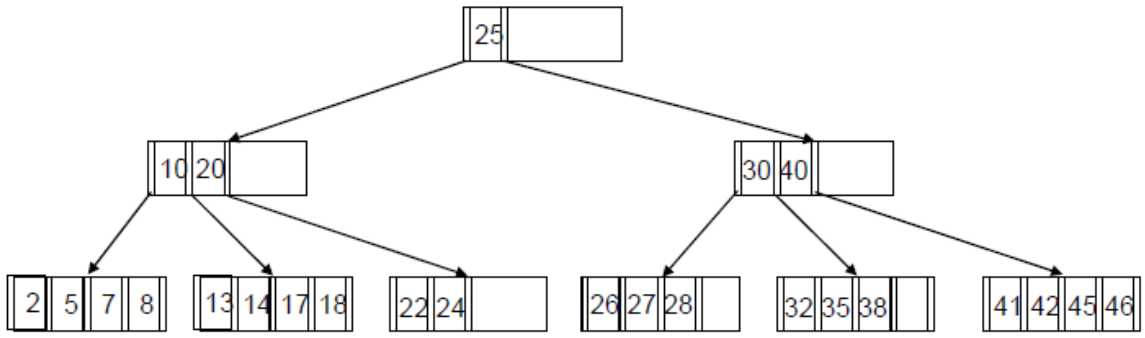
Da jede Blattseite auf der gleichen Ebene liegt, sind die Pfade Wurzel ➔ Blatt alle gleich lang (Baumhöhe). Bei einer minimalen Füllung (**m** Elemente pro Seite) beträgt für insgesamt **n** Elemente die Baumhöhe ca. $\log_m n$ ➔

➔ die B-Bäume sind hochgradig ausgeglichen.

Suchen in einem B-Baum

In der Wurzelseite beginnend wird top-down auf jeder Stufe nach dem ersten Schlüssel gesucht, welcher den gesuchten Schlüsselwert überdeckt.
Bei exakter Übereinstimmung ist die Suche beendet, ansonsten wird der vorherige Verweis zur nächsten Stufe befolgt und die Suche in der zugewiesenen Seite fortgesetzt. Wenn kein überdeckender Schlüsselwert in der Seite existiert, wird der letzte Verweis der Seite befolgt.
Wenn in der Blattseite keine exakte Schlüssel-Übereinstimmung gefunden wurde ➔ Suche erfolgslos.

Beispiel ➔ der Schlüsselwert **38** gesucht:



Einfügen in einen B-Baum

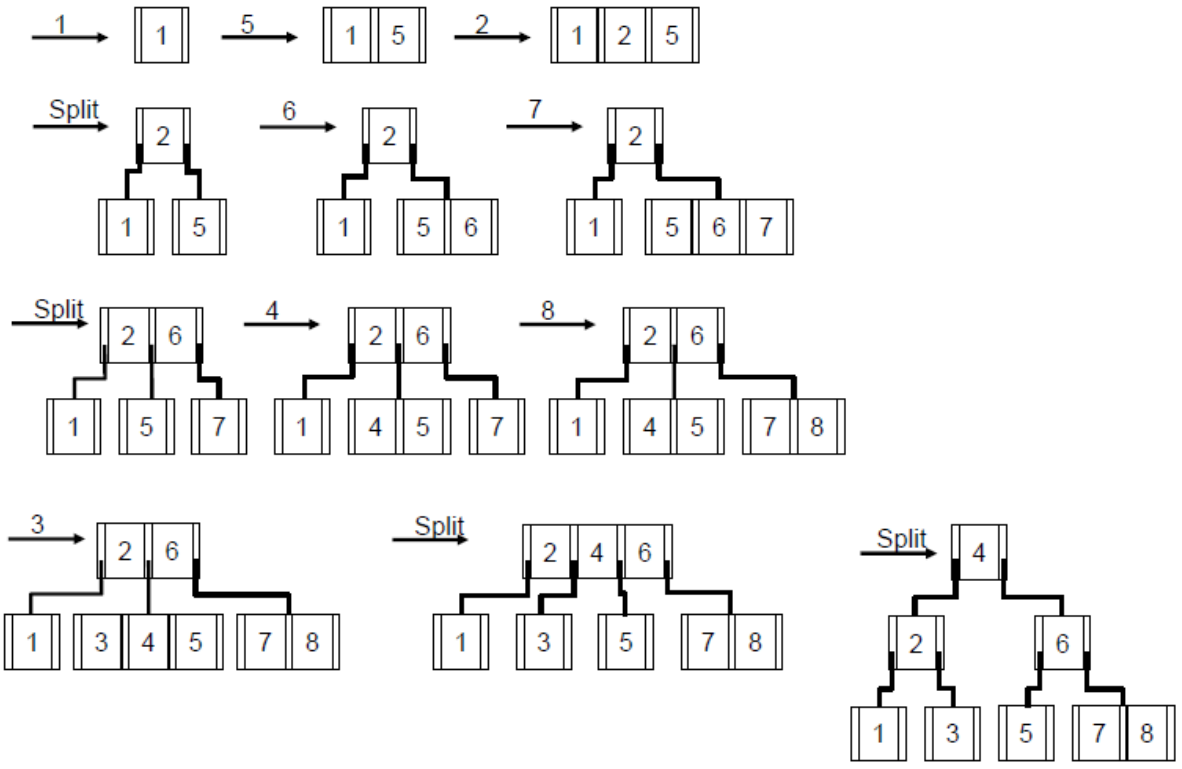
Zuerst muss die zugehörige Blattseite gefunden werden (➔ der erste überdeckende Schlüssel). Wenn in der Blattseite genug Platz vorhanden ist ➔ Einfügen unproblematisch.
Beim Seitenüberlauf wird die Blattseite in zwei ca. gleich grosse Seiten gespalten, was einen zusätzlichen Verweis in der Vaterseite erfordert.
Um diesen neuen Verweis vom vorherigen Verweis zu trennen, muss das mittlere Element der gespaltenen Doppelseite in die Vaterseite übertragen werden.
Falls dadurch die Vaterseite überläuft, muss der Spaltungsvorgang dort wiederholt und ggf. rekursiv bis zur Wurzelseite fortgesetzt werden.
Beim Überlauf der Wurzelseite wird eine neue Wurzelseite in der höheren Ebene erzeugt ➔ die Baumhöhe erhöht sich um **1** .

Einfügen-Schritte im Detail:

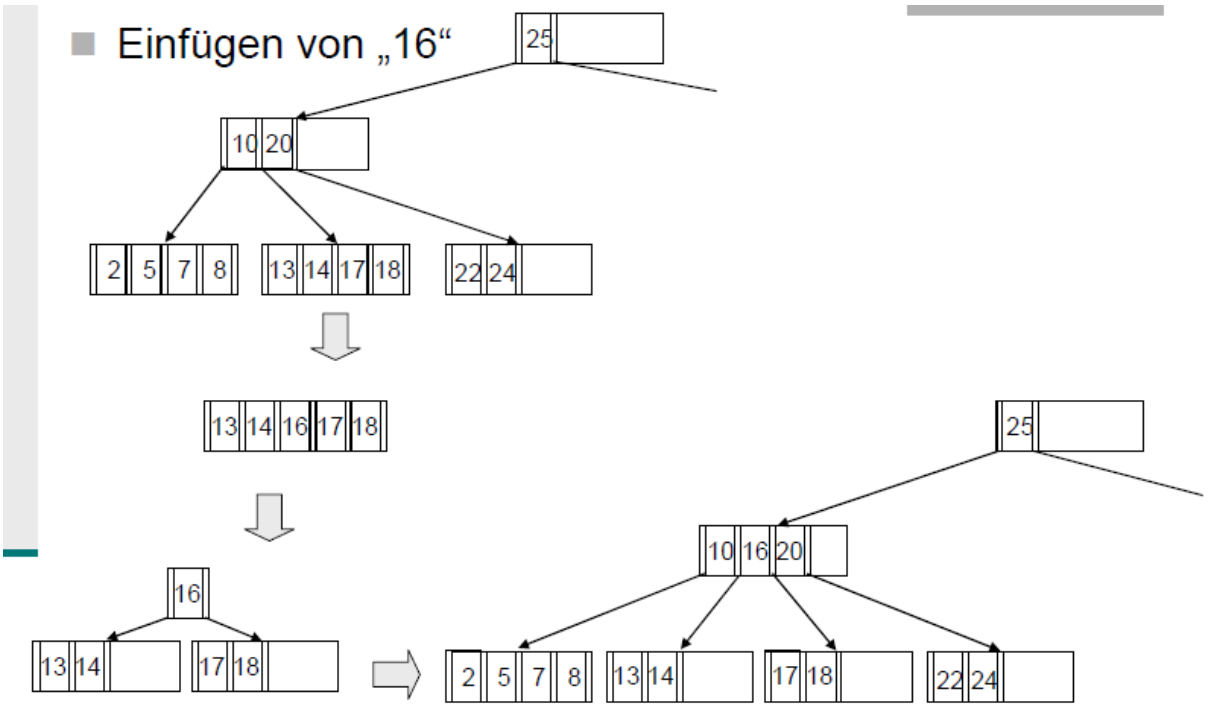
Einfügen in B-Bäumen

- Einfügen eines Wertes *w*
  - Blattseite suchen
  - passende Seite hat $n < 2m$ Elemente → *w* einsortieren
  - passende Seite hat $n = 2m$ Elemente → neue Seite erzeugen
    - ersten *m* Werte auf Originalseite
    - letzten *m* Werte auf neue Seite
    - mittleres Element in Vaterknoten nach oben
- eventuell dieser Prozeß rekursiv bis zur Wurzel
  - B-Bäume wachsen an der Wurzel!

Beispiel für **m = 1** :



Beispiel für **m = 2** :



N.B.: Es ist nicht notwendigerweise das eingefügte Element, das nach oben geschoben wird.

Löschen in B-Bäumen

- Problem: bei weniger als *m* Elementen auf Seite: Unterlauf
  - entsprechende Seite suchen
  - *w* auf Blattseite gespeichert

    → Wert löschen, eventuell Unterlauf behandeln
  - *w* nicht auf Blattseite gespeichert →
    - Wert löschen
    - durch lexikographisch nächst kleineres Element von einer Blattseite ersetzen
    - eventuell auf Blattseite Unterlauf behandeln

Behandlung von Seitenunterlauf

(i) Sich von links oder rechts ein Element borgen, wenn es dazu in einer Nachbarseite genug Elemente hat (Beispiel 1 unten).

(ii) Wenn **(i)** nicht möglich ist: die unterlaufende Seite mit einer Nachbarseite zusammenlegen. Dadurch entfällt ein Verweis in der Vaterseite und ein jetzt überzähliges Element der Vaterseite muss nach unten in die vereinte Seite übertragen werden (Beispiel 2 unten). Falls dadurch die Vaterseite unterläuft, muss die Unterlaufbehandlung dort wiederholt und ggf. rekursiv bis zur Wurzelseite fortgesetzt werden (die weniger als **m** Elemente haben darf ➔ wenn alle ihre Elemente entfernt sind, reduziert sich die Baumhöhe um **1**).

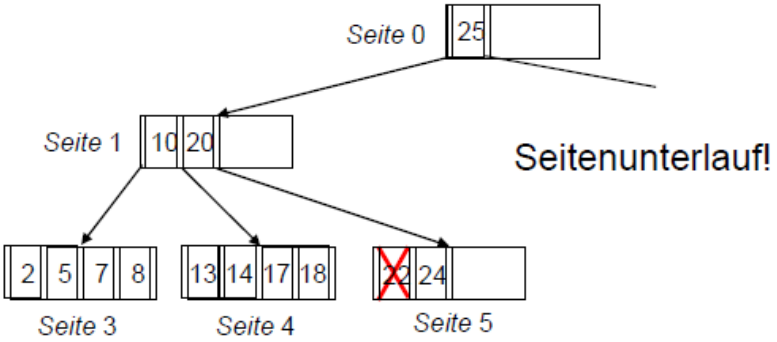
Unterlaufbehandlung im Detail:

- Unterlaufbehandlung
  - Ausgleichen mit einer benachbarten Seite (benachbarte Seite hat n Elemente mit $n > m$)
  - oder
    - Zusammenlegen zweier Seiten zu einer (Nachbarseite hat $n = m$ Elemente)
    - das passende “mittlere” Element vom Vaterknoten dazu
    - in Vaterknoten eventuell (rekursiv) Unterlauf behandeln

→ B-Bäume schrumpfen an der Wurzel!

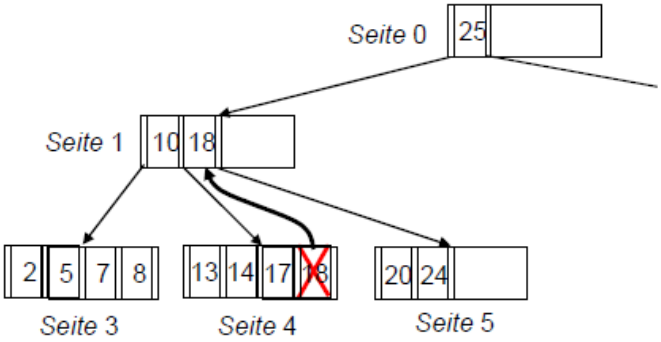
Unterlaufbehandlung, Beispiel 1 **(m = 2) :**

- Löschen von „22“



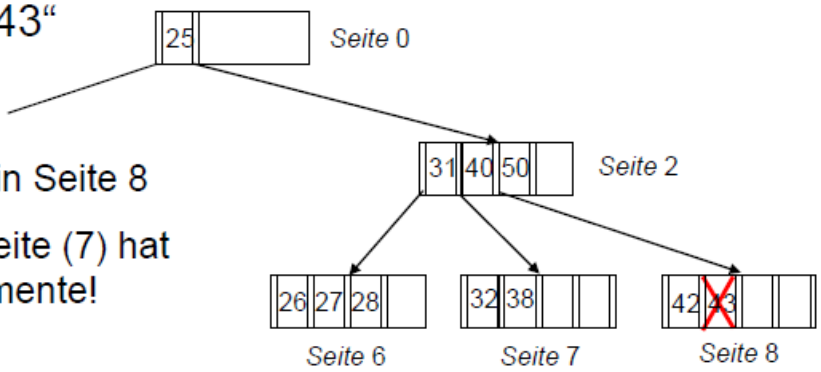
Seitenunterlauf!

→ Neuverteilen der Elemente in Seiten 4 und 5 und des Elements „20“ aus Elternseite 1

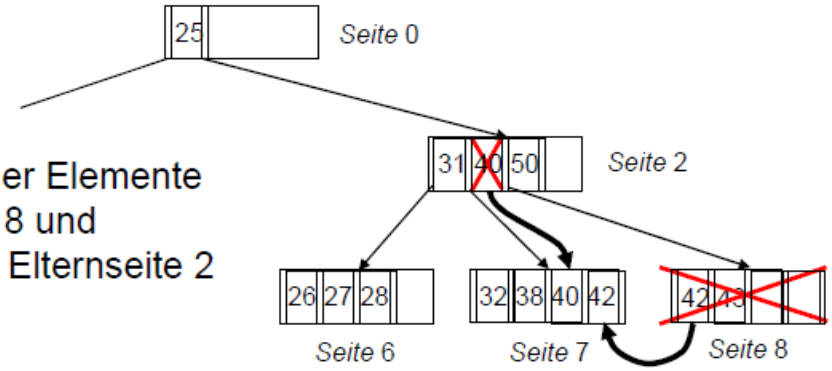


Unterlaufbehandlung, Beispiel 2 **(m = 2) :**

- Löschen von „43“



- Seitenunterlauf in Seite 8
- Linke Nachbarseite (7) hat genau n=m Elemente!



→ Zusammenlege der Elemente von Seiten 7 und 8 und Element „40“ aus Elternseite 2

Komplexität der Operationen

- Aufwand beim Einfügen, Suchen und Löschen im B-Baum immer $O(log_m(n))$ Operationen
- entspricht genau der “Höhe” des Baumes
- beliebt für sehr große Datenbestände (mit großer Knotengröße):
  - Konkret: Seiten der Größe 4 KB, Zugriffsattributwert 32 Bytes, 8-Byte-Zeiger: zwischen 50 und 100 Indexeinträge pro Seite; Ordnung dieses B-Baumes 50
  - 1.000.000 Datensätze: $\log_{50}(1.000.000) = 4$ Seitenzugriffe im schlechtesten Fall
  - optimiert Anzahl von der Festplatte in den Hauptspeicher zu transferierender Blöcke!

2,3,4-Bäume

Eine andere, häufig eingesetzte Unterkategorie von (a,b)-Bäumen mit **a = 2**, **b = 4** und der Einschränkung (ähnlich wie bei B-Bäumen), das alle Blätter auf der gleichen Baumebene liegen.
Wie die B-Bäume sind auch 2,3,4-Bäume hochgradig ausgeglichen.

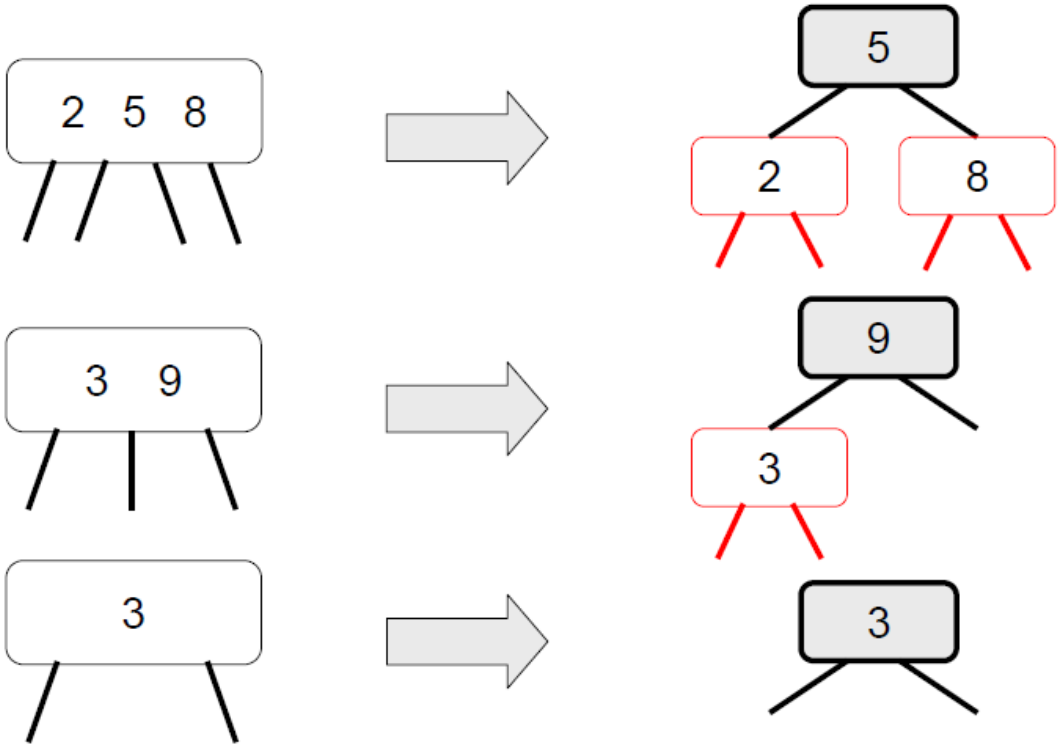
Das Suchverfahren ist für 2,3,4-Bäume gleich wie in B-Bäumen und auch für Einfügen/Löschen können die B-Baum-Techniken eingesetzt werden, solange keine 4-Knoten im Baum existieren (ein 2,3,4-Baum kann wie ein B-Baum der Ordnung **m = 1** aufgefasst werden, in dem aber zusätzlich 4-Knoten existieren). Demzufolge ist es von Vorteil, vor einer Mutation zuerst im betreffenden Pfad die 4-Knoten bottom-up oder top-down zu eliminieren:

- Suche analog zu binären Suchbäumen
- Einfügen:
  - erfolglose Suche liefert Blattknoten *b*
  - ist *b* ein 2- oder 3-Knoten: Einfügen
  - ist *b* ein 4-Knoten: Aufteilen (“split”), mittleres Element nach oben ziehen
  - Splitten kann sich bis zur Wurzel fortpflanzen! (bottom-up)
- Alternativ: Beim Einfügen werden vorsorglich alle 4-Knoten auf dem Pfad gesplittet (top-down)

Die durchs Einfügen **entstandenen** 4-Knoten werden allerdings so belassen und erst bei Bedarf im Rahmen einer zukünftigen Mutation beseitigt ➔ sie können auch als „lazy“ Knoten oder „Hypotheken“ interpretiert werden, die man zugunsten der Performance toleriert.

Rot-Schwarz-Bäume (RB-Bäume)

Um die Verwaltung von Verzweigungen in einem 2,3,4-Baum zu vereinfachen, kann diese Struktur auch als ein binärer Suchbaum dargestellt werden.
Dabei ersetzt man die 3- und 4-Knoten mit lokalen „Minibäumen“, welche aus 2-Knoten bestehen:



Rot-Schwarz-Bäume (red-black, kurz RB-Bäume)

Die zusätzlich eingeführten „virtuellen“ Knoten werden als *rote Knoten* bezeichnet, die übrigen nennt man *schwarze Knoten*.

Der so entstandene binäre Baum kann nach Bedarf durch lokale Rotationen (siehe AVL-Bäume oben) und durch Umfärbungen von Knoten umstrukturiert werden, wobei die folgenden Einschränkungen zu beachten sind:

Rot-Schwarz-Bäume sind binäre Suchbäume mit

1. Jeder Knoten ist entweder rot oder schwarz.
2. Jeder Blattknoten (Null-Knoten) ist per Definition schwarz.
3. Die Kinder jedes roten Knotens sind schwarz.
4. Für jeden Knoten *k* gilt: Jeder Pfad von *k* zu einem Blatt enthält die gleiche Anzahl schwarze Knoten („Schwarztiefe“).

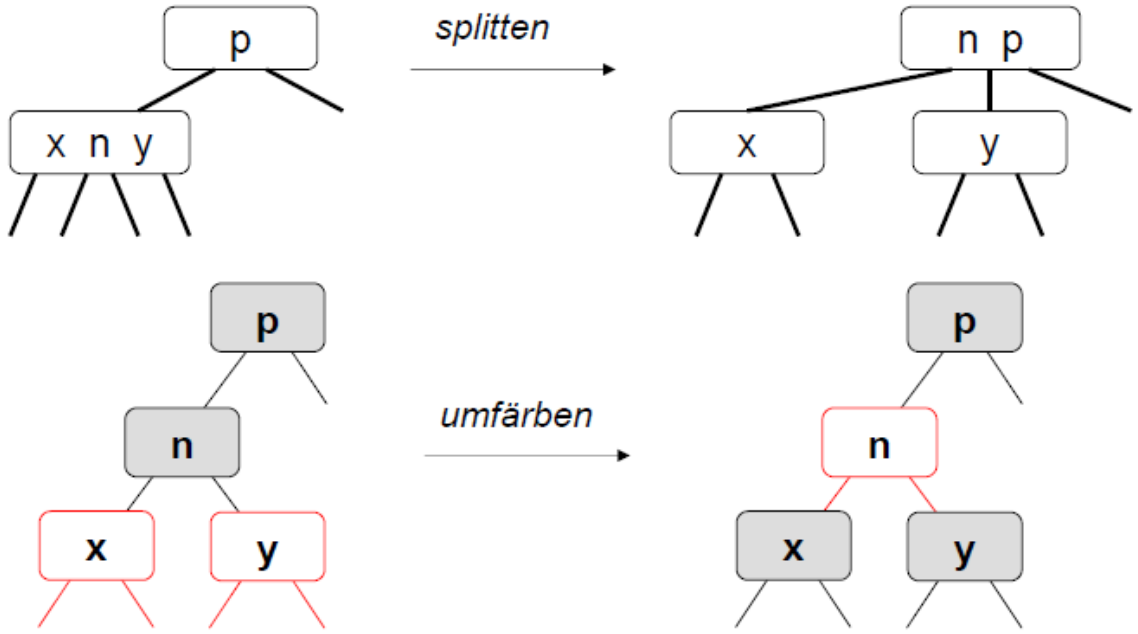
Beim Einhalten dieser Regel bleibt der mutierte RB-Baum gleichwertig mit einem 2,3,4-Baum und kann auch dementsprechend zurückverwandelt werden.

Ähnlich wie bei 2,3,4-Bäumen müssen vor Einfügen/Löschen die darin involvierten 4-Knoten (hier entsprechende Minibäume) zuerst gespalten werden.

Beim Einfügen wird wie folgt verfahren:

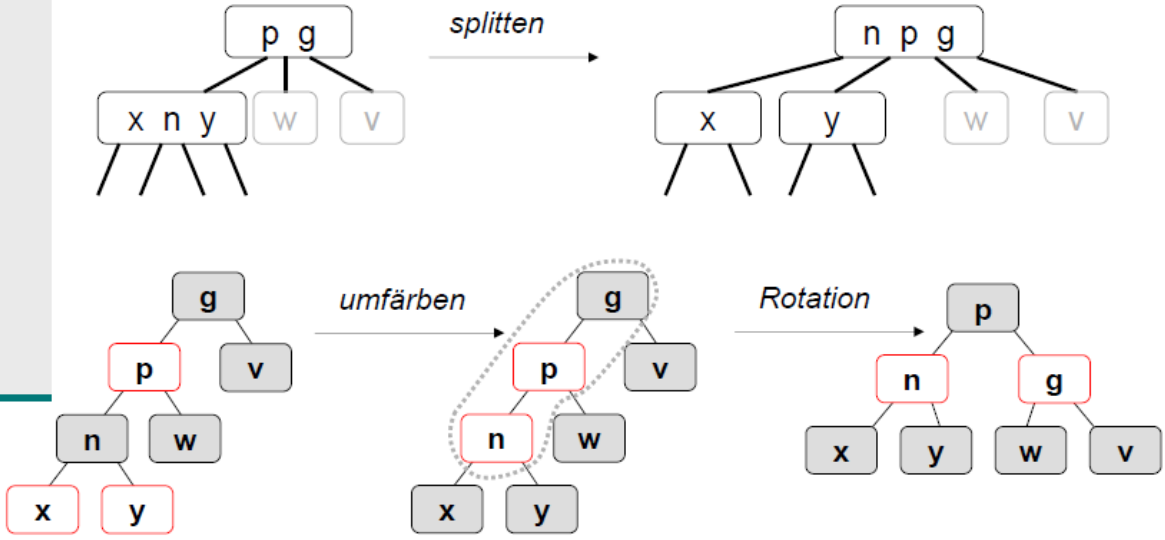
- Spalten der 4-Knoten, top-down, nichtrekursiv
- Dabei werden 3 folgende Fälle unterschiedlich behandelt
    - Der 4-Knoten hängt unter einem 2-Knoten
    - Der 4-Knoten hängt unter einem 3-Knoten links oder rechts
    - Der 4-Knoten hängt unter einem 3-Knoten in der Mitte
- Anschliessend wird das neue Element auf der Blattebene eingefügt

4-Knoten an 2-Knoten



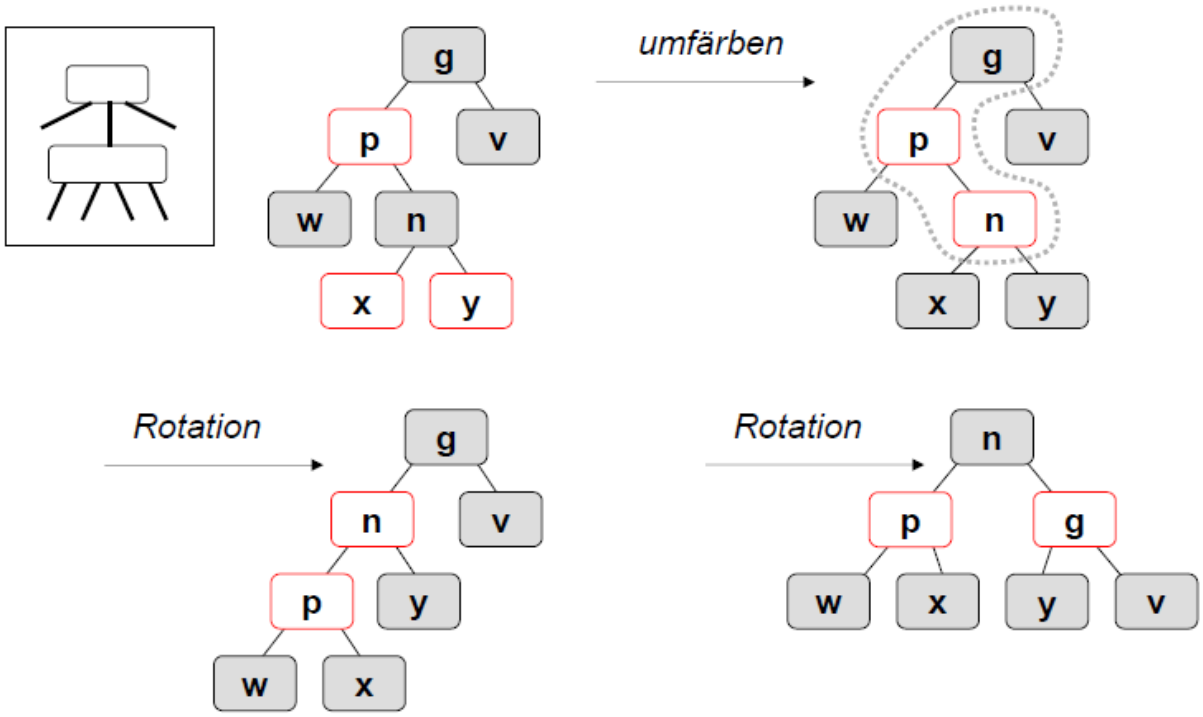
⇨ Umfärben **n**, **x**, **y**

4-Knoten unter 3-Knoten links oder rechts



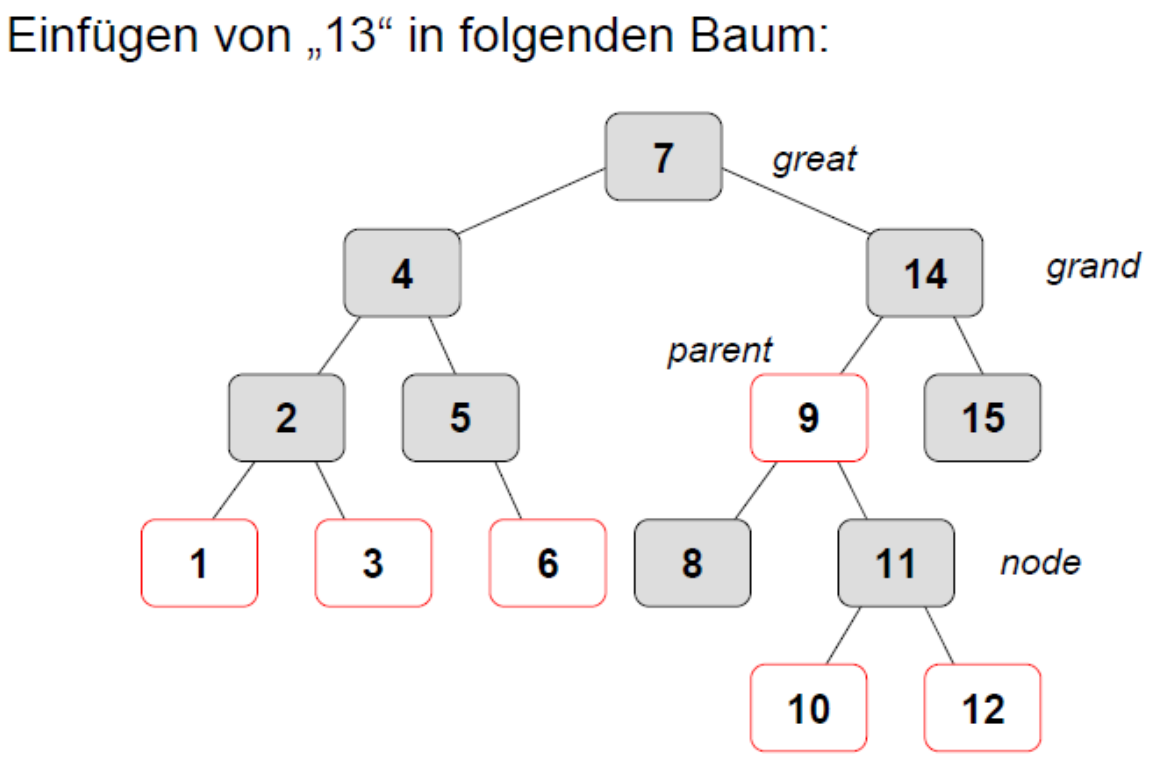
⇨ Umfärben **n**, **x**, **y** ➔ Rotation **n**>**p**>**g** im Uhrzeigersinn ➔
➔ Umfärben **p**, **g**

4-Knoten unter 3-Knoten mittig



⇨ Umfärben **n**, **x**, **y** ➔ Rotation **y**>**n**>**p** gegen Uhrzeigersinn ➔
➔ Rotation **p**>**n**>**g** im Uhrzeigersinn ➔ Umfärben **n**, **g**

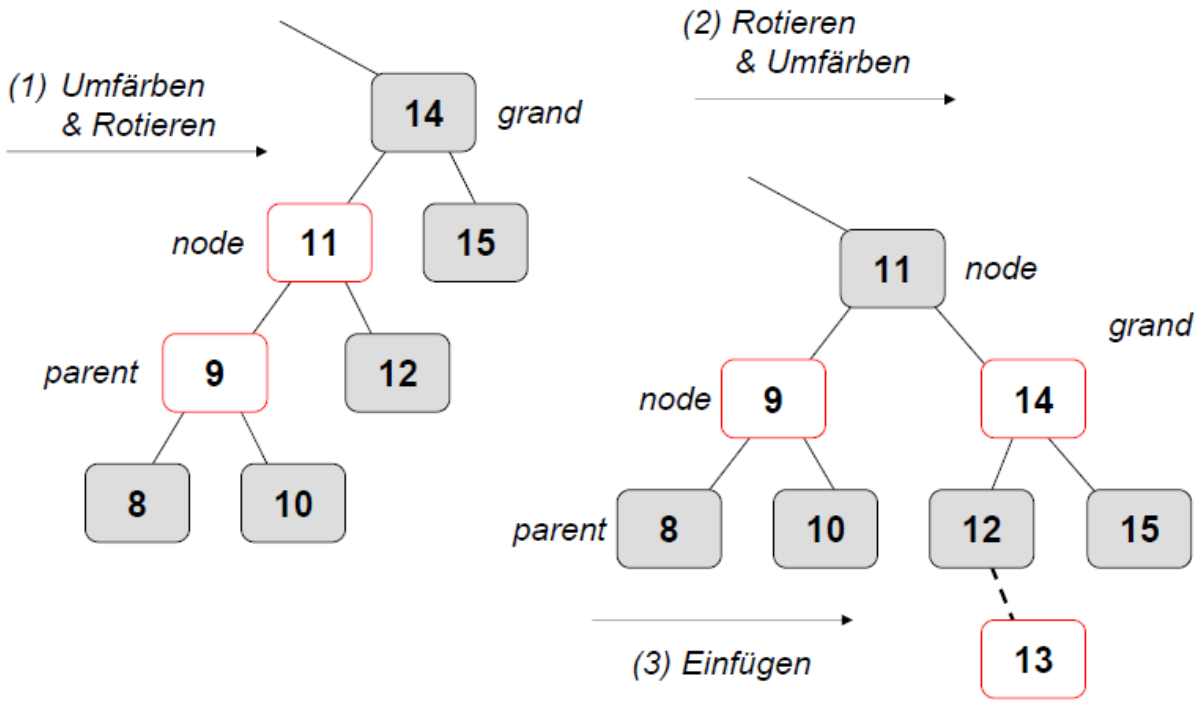
Beispiel:



13 kann nicht unter **12** rechts angehängt werden (➔ die Regel „gleiche Schwarztiefe für alle Blätter“ wäre verletzt), zuerst muss der 4-Knoten **node** gespalten werden.

Zur **node** – Spaltung wird das obige Schema „4-Knoten unter 3-Knoten mittig“ angewendet:

⇨ Umfärben **10**, **11**, **12** ➔ Rotation **12**>**11**>**9** gegen Uhrzeigersinn ➔
➔ Rotation **9**>**11**>**14** im Uhrzeigersinn ➔ Umfärben **11**, **14**



Anschliessend kann **13** ins Blatt **12** eingefügt werden.

Praktischer Einsatz der Verfahren

- AVL in Ausbildung und Lehrbüchern
- Rot-Schwarz-Bäume: Implementierung von Sets und Maps in Java-Bibliothek
  http://java.sun.com/j2se/1.5.0/docs/api/java/util/TreeMap.html
- B-Bäume “überall im Einsatz” (einfache Einfüge-Algorithmen; Knotengröße an Seitengröße von Hintergrundspeichern optimal anpassbar)

Graphen, Grundbegriffe

Graph: Ein System von Knoten und Kanten.

Kante (u,v): Eine Verbindung des Startknotens **u** mit dem Zielknoten **v**. Knoten **u** und **v** sind *adjazent*, **(u,v)** ist *inzident* zu **u** und zu **v**.

Gerichteter Graph (Digraph): Zu einer Kante **(u,v)** existiert nicht zwingend die Gegenkante **(v,u)**.

Ungerichteter (doppelt gerichteter) Graph: Zu einer Kante **(u,v)** existiert zwingend die Gegenkante **(v,u)**.

Pfad (Weg): Eine Folge von Knoten, die mit Kanten verbunden sind.

Einfacher Pfad: Seine Knoten sind paarweise verschieden.

Kreis (Zyklus): Ein Pfad mit Startknoten = Zielknoten.

Einfacher Kreis: Ausser Startknoten (= Zielknoten) sind seine Knoten paarweise verschieden.

Hamiltonkreis: Ein einfacher Kreis, der alle Knoten des Graphs umfasst.

Azyklischer Digraph (DAG): Ein Digraph, der keine Kreise beinhaltet.

Stark zusammenhängender Digraph: Von jedem Knoten existiert ein Pfad zu jedem anderen Knoten.

Starke Zusammenhangskomponente: Der maximale stark zusammenhängende Teilgraph, welcher von einer Knotenmenge ausgehend durch vorhandene Pfade gebildet werden kann.

Ungerichteter Baum: Ein ungerichteter Graph, in dem es zwischen zwei beliebigen Knoten jeweils *genau* einen Pfad gibt.

Ungerichteter Wald: Ein ungerichteter Graph, in dem es zwischen zwei beliebigen Knoten jeweils *höchstens* einen Pfad gibt.

Gerichteter Baum mit Wurzel (Out-Tree): Ein Digraph, in dem von der Wurzel zu jedem anderen Knoten genau ein Pfad führt.

Anmerkung: In den Kursunterlagen DA_PVA1.pdf – DA_PVA5.pdf wird mit „Baum“ ein *Out-Tree* bezeichnet.

Darstellung von Graphen

Die Knoten werden üblicherweise als Elemente eines Arrays dargestellt, wobei jeder Knoten mit seinem Index identifiziert wird. Zur Identifizierung einer Kante kann folgerichtig ein Index-Paar benutzt werden.
Die Kanten-Objekte werden in Kantenlisten oder in Adjazenzarrays/-listen aufbewahrt (siehe die Abbildung unten).

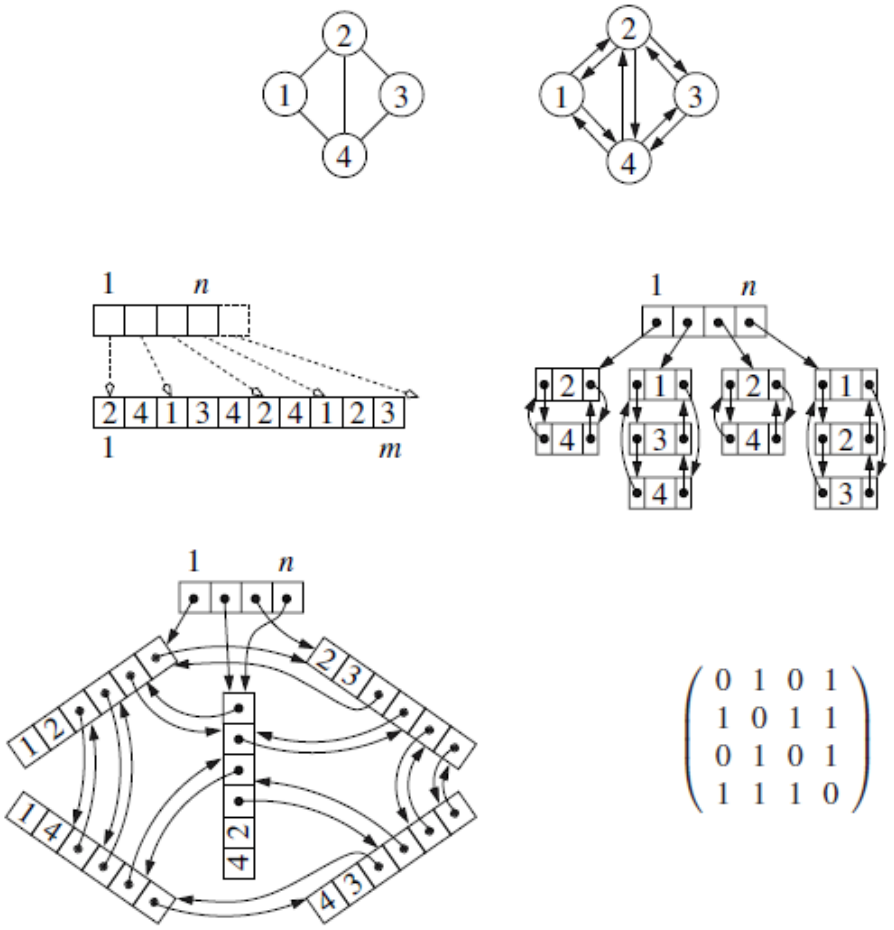


Abb. 8.1. *Oben*: Ein ungerichteter Graph und der entsprechende doppelt gerichtete Graph. *Mitte*: Die Darstellung des gleichen Graphen mit einem Adjazenzarray und mit Adjazenzlisten. *Unten*: Die Darstellung über verzeigerte Kantenobjekte und als Adjazenzmatrix.

Die zwei Varianten in der Mitte weisen jedem Knoten eine Sammlung von seinen adjazenten Knoten zu: links als ein Teil-Array, rechts als eine doppelt verkettete Liste.

Unten links wird jedem Knoten eine Liste von seinen inzidenten Kanten zugewiesen. Dabei wird jede Kante mit einem Kantenobjekt dargestellt, dass zu zwei Listen gehört und dementsprechend zwei Paare von *next*-/*prev*-Zeigern beinhaltet.
Beispielsweise gehört die Kante **(1,2)** via ihr erstes Zeigerpaar zur Liste **(1,2) <=> (1,4)** und via ihr zweites Zeigerpaar zur Liste **(1,2) <=> (2,4) <=> (2,3)** .

Unten rechts werden die Kanten des gleichen Graphs mit einer Adjazenzmatrix dargestellt.

Graph-Traversierung

Für die Erkundung und Verarbeitung eines Graphs ist ein Durchwandern von seinen Knoten und Kanten (Traversierung) notwendig. Die Traversierung kann nach verschiedenen Schemen erfolgen, hier werden zwei davon thematisiert:

- Breitensuche (Breadth-First Search, BFS)
- Tiefensuche (Depth-First Search, DFS)

In beiden Fällen spannt man den Pfaden folgend von einem Startknoten aus einen gerichteten Wald auf, dessen Kanten als *Baumkanten* registriert werden. Die übrigen inspizierten Kanten werden dabei als *Vorwärts*-, *Rückwärts*- oder *Querkanten* kategorisiert (siehe die Abbildung unten).

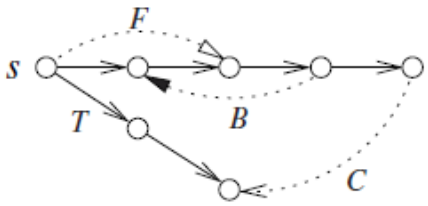


Abb. 9.1. Kanten eines Digraphen, klassifiziert als Baumkanten (*Tree*, durchgezogen), Vorwärtskanten (*Forward*), Rückwärtskanten (*Backward*) und Querkanten (*Cross*). Knoten *s* ist die Wurzel.

Die folgende Abbildung zeigt die Resultate der Traversierung eines Graph mit den obigen zwei Methoden.

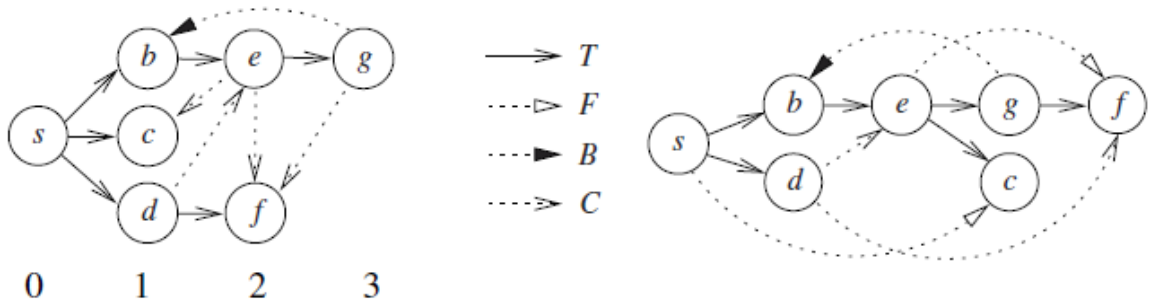


Abb. 9.3. Ein Beispiel für die bei Breitensuche (*links*, mit Schichten) und Tiefensuche (*rechts*) entstehende Einteilung der Kanten in Baumkanten (T), Vorwärtskanten (F), Rückwärtskanten (B) und Querkanten (C). (Bei BFS gibt es niemals Vorwärtskanten; der Algorithmus selbst unterscheidet nicht zwischen Rückwärts- und Querkanten.) BFS findet die Knoten in der Reihenfolge s, b, c, d, e, f, g; DFS findet die Knoten in der Reihenfolge s, b, e, g, f, c, d.

Breitensuche (BFS)

Der Algorithmus verfährt schichtweise: die Schicht **0** besteht aus dem Startknoten, Schicht **1** bilden die unmittelbaren Nachfolger des Startknoten. Schicht **2** besteht aus unmittelbaren Nachfolgern von Knoten der Schicht **1** usw. Dabei werden für die Schicht **i** nur diejenigen Knoten berücksichtigt, welche mindesten einen unmittelbaren Vorgänger in der Schicht **i – 1** haben und **keinen** Vorgänger in tieferen Schichten als **i – 1**.

Bei diesem Vorgehen können keine Vorwärtskanten entstehen und es wird nicht zwischen Rückwärts- und Querkanten unterscheidet (weil das viel zusätzliche Iteration erfordern würde).
BFS eignet sich gut für Anwendungen, in welchen möglichst kurze Pfade Vorgänger => Nachfolger von Vorteil sind.

Das Pseudocode-Schema der Breitensuche:

```
Function bfs(s : NodeId) : (NodeArray of 1..n) × (NodeArray of NodeId)
    d = ⟨∞,...,∞⟩ : NodeArray of NodeId                    // Abstand von der Wurzel
    parent = ⟨⊥,...,⊥⟩ : NodeArray of NodeId
    d[s] := 0
    parent[s] := s                                        // Schleife: signalisiert „Wurzel“
    Q = ⟨s⟩ : Set of NodeId                               // aktuelle Schicht des BFS-Baums
    Q' = ⟨⟩ : Set of NodeId                               // nächste Schicht des BFS-Baums
    for ℓ := 0 to ∞ while Q ≠ ⟨⟩ do                       // erkunde Schicht für Schicht
        invariant Q enthält alle Knoten mit Abstand ℓ von s
        foreach u ∈ Q do
            foreach (u,v) ∈ E do                          // durchlaufe Ausgangskanten von u
                if parent[v] = ⊥ then                     // bislang nicht erreichter Knoten gefunden
                    Q' := Q' ∪ {v}                        // für nächste Schicht merken
                    d[v] := ℓ + 1
                    parent[v] := u                        // aktualisiere BFS-Baum
        (Q,Q') := (Q',⟨⟩)                                 // schalte auf nächste Schicht um
    return (d,parent)                                     // der BFS-Baum ist jetzt {(v,w) : w ∈ V, v = parent[w]}
```

Abb. 9.2. Breitensuche von einem Knoten s aus.

Wenn der zu traversierende Graph ein Baum ist, reduziert sich die Traversierung auf die Verarbeitung von sämtlichen Knoten. In diesem Fall kann mit Hilfe einer Warteschlange ein viel effizienteres Vorgehen implementiert werden.

Beispiel (binärer Baum):

```
Procedure bbBfsTraversieren (wurzel : Node)      //  Wurzelknoten, die Klasse Node stellt =>
                                                 //  => einen Baumknoten dar
  q := leere Warteschlange
  q.pushBack(wurzel)                             // Wurzel in die Warteschlange aufnehmen

  while not q.isEmpty() do
    Node n := q.first()                          // den ersten Knoten inspizieren

    // den Inalt des Knoten verarbeiten

    if linkes Kind existiert then
      q.pushBack(n.getLeftChild())               // linkes Kind eintragen

    if rechtes Kind existiert then
      q. pushBack (n.getRightChild())            // rechtes Kind eintragen

    q.popFront()                                 // den verarbeiteten Knoten entfernen
```

Tiefensuche (DFS)

DFS ist ein rekursives Verfahren. Nach dem Entdecken eines Knotens **n** werden zuerst seine Nachfolger mit rekursiven Aufrufen vollständig traversiert, erst anschliessend erfolgt die Verarbeitung des Dateninhalts von **n** . Somit wird die Rekursion mit Vorrang bis zu maximal möglicher Tiefe fortgesetzt.
Jeder Knoten führt einen Status-Indikator, mit dem sein Verarbeitungszustand markiert wird. Der Knoten kann in einem von den folgenden drei Zuständen sein:

- *unmarkiert* (noch nicht entdeckt)
- *aktiv* (laufende Inspektion)
- *beendet* (Inspektion abgeschlossen)

Das Pseudocode-Schema der Tiefensuche:

Tiefensuche in einem gerichteten Graphen $G = (V, E)$
entferne alle Knotenmarkierungen
init
foreach $s \in V$ **do**
  if s ist nicht markiert **then**
    root(s) // Mache s zu einer Wurzel
    DFS(s, s) // Baue neuen DFS-Baum mit Wurzel s

Procedure *DFS*$(u, v : NodeId)$ // erkunde v, von u kommend.
  markiere v als *aktiv*
  foreach $(v, w) \in E$ **do**
    if w ist markiert **then** *traverseNonTreeEdge*(v, w) // w schon vorher erreicht
    else *traverseTreeEdge*(v, w) // w vorher nicht erreicht
      DFS(v, w)
  backtrack(u, v) // Aufräumen; Zusammenfassen und Rückgabe von Daten
  markiere v als *beendet*
  return // Rücksprung

Abb. 9.4. Ein Algorithmenschema für die Tiefensuche in einem gerichteten Graphen $G = (V, E)$. Wir sagen, dass ein Aufruf *DFS*$(*, v)$ den Knoten v *erkundet*. Diese Erkundung ist vollständig durchgeführt, wenn dieser Aufruf endet.

DFS ermittelt im Gegenteil zu BFS eine vollständige Klassifizierung der Kanten. Zu diesem Zweck wird jeder Knotenobjekt **n** mit den Zählern **dfsNum[n]** und **finNum[n]** versehen, die mit Hilfe von den globalen Zählern **dfsPos** und **finPos** wie folgt geführt werden:

init:	$dfsPos = 1 : 1..n; \quad finPos = 1 : 1..n$
$root(s)$:	$dfsNum[s] := dfsPos{+}{+}$
$traverseTreeEdge(v,w)$:	$dfsNum[w] := dfsPos{+}{+}$
$backtrack(u,v)$:	$finNum[v] := finPos{+}{+}$

Bei der Inspektion einer Kante wird ihre Kategorie mit dem folgenden Schema bestimmt:

Tabelle 9.1. Die Klassifikation einer Kante (v,w).

Typ	$dfsNum[v] < dfsNum[w]$	$finNum[w] < finNum[v]$	Markierung von w
Baumkante	ja	ja	unmarkiert
Vorwärtskante	ja	ja	*beendet*
Rückwärtskante	nein	nein	*aktiv*
Querkante	nein	ja	*beendet*

Im Vergleich zu BFS kann DFS komplexere Graphen traversieren mit umfassenderen Ergebnissen, die resultierenden Strukturen sind aber tendenziell weniger kompakt.
Es gibt auch eine nicht-rekursive Implementierung des DFS-Schema, welche die inspizierten Knoten in einem Stack verwaltet.

Auch die DFS-Traversierung reduziert sich für einen binären Baum zu einem viel einfacheren und effizienteren Schema:

```
Procedure bbDfsTraversieren (wurzel : Node)        //  Wurzelknoten, die Klasse Node stellt =>
                                                   //  => einen Baumknoten dar
    if linkes Kind existiert then
        bbDfsTraversieren(n.getLeftChild())        // linkes Kind traversieren

    if rechtes Kind existiert then
        bbDfsTraversieren(n.getRightChild())       // rechtes Kind traversieren

    verarbeiten(n.getInhalt())                     // Knoten n verarbeiten
```

Eine sehr nützliche Ausbau-Variante von DFS ist die Ermittlung von stark zusammenhängenden Komponenten. Die Implementierung davon wird durch das Konzept der *Vergröberung* vereinfacht. Unter Vergröberung versteht man die Aufteilung des Graphs in stark zusammenhängende Teilgraphen:

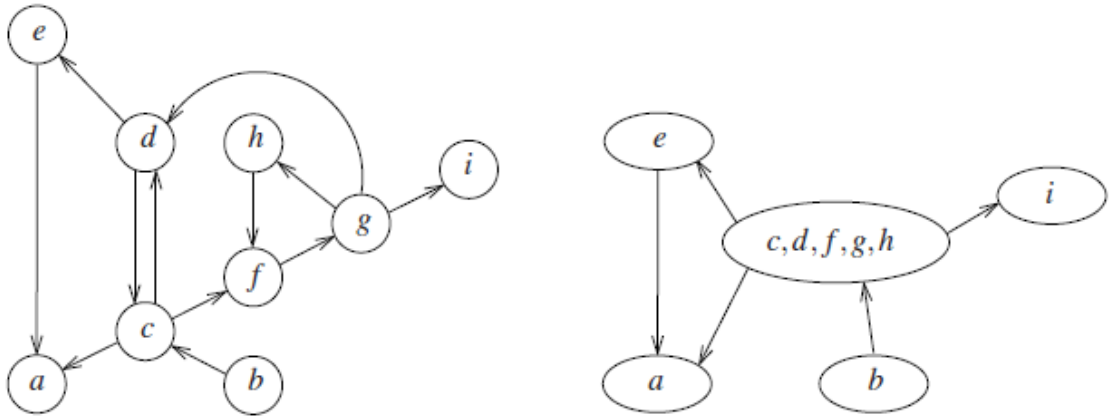


Abb. 9.5. Ein Digraph G und der zugehörige vergröberte Graph G^s. Die Komponenten von G haben Knotenmengen $\{a\}$, $\{b\}$, $\{c,d,f,g,h\}$, $\{e\}$ und $\{i\}$.

Jedem Teilgraphen wird ein von seinen Knoten als „Repräsentant“ zugewiesen, wobei die Repräsentanten im Stack **oReps** und ihre Knoten im Stack **oNodes** geführt werden.
Diese Erweiterung wird in die Unterprogramme von DFS (siehe oben) wie folgt integriert:

```
init:
    component : NodeArray of NodeId                    // Repräsentanten
    oReps = ⟨⟩ : Stack of NodeId                        // Repräsentanten offener Komponenten
    oNodes = ⟨⟩ : Stack of NodeId                       // alle Knoten in offenen Komponenten

root(w) oder traverseTreeEdge(v,w):
    oReps.push(w)                                       // neue offene
    oNodes.push(w)                                      // Komponente

traverseNonTreeEdge(v,w):
    if w ∈ oNodes then
        while w ≺ oReps.top do oReps.pop                // vereinige Komponenten auf Kreis

backtrack(u,v):
    if v = oReps.top then
        oReps.pop                                       // Komponente als
        repeat                                          // geschlossen erkannt,
            w := oNodes.pop                             // Behandlung abschließen
            component[w] := v
        until w = v
```

Abb. 9.10. Eine konkrete Version der Unterprogramme im DFS-Algorithmenschema aus Abb. 9.4, die die starken Zusammenhangskomponenten eines Digraphen $G = (V,E)$ berechnet.

Die Teilgraphen („offene Komponenten“) werden im Verlauf der Traversierung maximalisiert. Nach dem Abschluss der Prozedur beinhalten die beiden Stacks als Ergebnis die starken Zusammenhangskomponenten des Graphs.