Kapitel 3

Suchen

In diesem Kapitel behandeln wir Algorithmen für das Auffinden von Elementen in einer Menge bzw. den Test, ob das Element überhaupt in der Menge enthalten ist. Die Algorithmen basieren darauf, dass die Menge in einer speziell für diese Art Anfrage gewählten Datenstruktur verwaltet wird.

Grundsätzlich betrachten wir also Implementationen eines Datentyps `Dictionary` zur Verwaltung von Elementen gleichen Typs (oder zumindest mit Schlüsseln gleichen Typs).

	Dictionary
`item`	find(`key` k)
	insert(`item` x)
	remove(`key` x)

Voraussetzung ist dabei lediglich, dass wir über der Grundmenge, aus der die Elemente stammen, eine Ordnung $\leq$ gegeben haben. Auch wenn die Elemente in den Beispielen der Einfachheit halber wieder Zahlen werden, wird also in der Regel nicht ausgenutzt, um wieviel sich zwei Werte unterscheiden. Verfahren, die stärkere Voraussetzungen machen, behandeln wir im nächsten Kapitel.

3.1 Folgen

In einer unsortierten Folge M kann durch *lineare Suche*, d.h. Durchlaufen aller Positionen, in linearer Zeit festgestellt werden, ob, wo und wie oft ein bestimmtes Element in M vorkommt.

3.1 Satz
Lineare Suche benötigt auch im mittleren Fall Zeit $\Theta(n)$.

Beweis. Kommt das gesuchte Element nicht vor, weiß man das erst nach Durchlaufen der ganzen Folge. Kommt das Element vor, dann an jeder Stelle $i = 0, \ldots, n-1$ mit gleicher Wahrscheinlichkeit $1/n$. Die lineare Suche durchläuft zunächst alle davor liegenden Stellen, prüft also insgesamt $i+1$ Folgenelemente. Die mittlere Laufzeit ist damit

$$\sum_{i=0}^{n-1} \frac{1}{n}(i+1) = \frac{n(n+1)}{2n} = \frac{n+1}{2} \in \Theta(n) \ .$$

□

Unter der Annahme, dass die Reihenfolge und Vorkommen der zulässigen Werte im Array zufällig sind und dass jedes Element mit gleicher Wahrscheinlichkeit gesucht wird, kann keine bessere Strategie angegeben werden. Das gesuchte Element könnte dann an jeder Stelle stehen, und wir lernen nichts daraus, es an bestimmten anderen Stellen nicht gefunden zu haben.

Wir behandeln zwei Methoden, durch Anordnung der Elemente in M eine bessere Laufzeit zu bekommen.

- Speicherung von M in einem sortierten Array
- Umsortierung von M aufgrund von Anfragen

3.1.1 Sortierte Arrays

Können wir voraussetzen, dass M sortiert ist, dann bedeutet das Finden eines anderen Elements an einer bestimmten Stelle, dass das gesuchte davor oder dahinter stehen muss – je nachdem, ob es kleiner oder größer als das

gefundene ist. Bei der *binären Suche* in Algorithmus 17 wird bei jedem Vergleich mit einem Array-Element die Hälfte der verbleibenden Elemente von der weiteren Suche ausgeschlossen.

Algorithmus 17: Binäre Suche

```
binsearch(M[0,...,n−1], x)  begin
│ l ← 0;   r ← n − 1
│ while x ≥ M[l] && x ≤ M[r] do
│ │ m ← ⌊(l+r)/2⌋
│ │ if x > M[m] then
│ │ │ l ← m + 1
│ │ else if x < M[m] then
│ │ │ r ← m − 1
│ │ else
│ │ └ return „x ist in M“
│ └
│ return „x ist nicht in M“
end
```

3.2 Beispiel

*[**Interaktiv:** 0,2,5,5,8,10,12,13,18,23,36,42,57,60,64,666, Anfragen nach 18,9.]*

3.3 Bemerkung

Wir könnten sogar angeben, an welcher Stelle x im Array M auftritt, gehen aber hier davon aus, dass die aufrufende Stelle nicht wissen kann, dass die Elemente in einem Array verwaltet werden und daher auch mit der Positionsinformation nichts anfangen kann. Zum einen verwenden die Verfahren in den nächsten Abschnitten andere Datenstrukturen, und zum anderen ändern sich die Positionen im Allgemeinen, wenn Elemente eingefügt und gelöscht werden (da das Array immer sortiert sein muss).

3.4 Bemerkung

Da M sortiert ist, kann auch die Anzahl der Vorkommen eines Elementes leicht bestimmt werden. Wir müssen wegen der Sortierung von der gefundenen Stelle in beide Richtungen nur solange nach weiteren Vorkommen suchen, bis jeweils zum ersten Mal ein anderes Element auftritt.

3.5 Satz

Binäre Suche auf einem sortierten Array der Länge n benötigt $\Theta(\log n)$ Schritte, um ein Element x zu finden bzw. festzustellen, dass $x \notin M$.

Beweis. Die Anzahl der Schleifendurchläufe ist immer $\Theta(\log n)$, da das Intervall $M[l, \ldots, r]$ nach jedem Durchlauf eine Länge hat, die um höchstens eins von der Hälfte der vorherigen abweicht. □

3.6 Bemerkung

Moderne Prozessoren verwenden Pipelining, sodass Sprünge in bedingten Verzweigungen in der Regel zu Zeitverlust führen. Bei im Wesentlichen gleichverteilten Eingabe kann es daher günstiger sein, statt $m \leftarrow \lfloor \frac{l+r}{2} \rfloor$ *eine ungleiche Aufteilung wie z.B.* $m \leftarrow \lfloor \frac{1}{3}(l+r) \rfloor$ *zu wählen, weil die Bedinung in der Schleife dann voraussichtlich häufiger zutrifft als nicht und der Inhalt der Pipeline dann nicht verloren geht.*

Eine ähnliche Idee wie in der voraus gegangenen Bemerkung liegt auch der *Interpolationssuche* zu Grunde. Handelt es sich bei den Elementen des Arrays um Zahlen und liegt x deutlich näher an einem der Inhalte der Randzellen, macht es unter Umständen Sinn, nicht in der Mitte, sondern entsprechend näher an diesem Rand einen Vergleich vorzunehmen und damit gleich mehr al die Hälfte der verbleibenden auszuschließen.

ausreichend: intervallskalierte Daten

Algorithmus 18: Interpolationssuche

```
interpolationsearch(M[0,...,n-1], x) begin
    l ← 0;  r ← n-1
    while x ≥ M[l] && x ≤ M[r] do
        m ← l + ⌊ (x-M[l]) / (M[r]-M[l]) · (r-l) ⌋
        if x > M[m] then
            l ← m+1
        else if x < M[m] then
            r ← m-1
        else
            return „x ist in M“
    return „x ist nicht in M“
end
```

3.7 Beispiel

[Gleiche zwei Anfragen wie oben – hat's was gebracht?]

Binäre und Interpolationssuche setzen voraus, dass man die Länge des Arrays kennt. Wenn die Länge unbekannt (oder zumindest sehr groß) und das

gesuchte Element auf jeden Fall (insbesondere an eher kleiner Indexposition) enthalten ist, bietet sich an, den Suchbereich zunächst vorsichtig von vorne einzugrenzen.

Algorithmus 19: Exponentielle Suche

```
expsearch(M[0,...], x) begin
  r ← 1
  while x > M[r] do r ← 2r
  binsearch(M[⌊r/2⌋,...,r], x)
end
```

Die Korrektheit des Verfahrens ergibt sich daraus, dass nach Abbruch der **while**-Schleife sicher $x \in M[\lfloor \frac{r}{2} \rfloor, \ldots, r]$ gilt. Für die Laufzeit beachte, dass die Teilfolge, auf der binär gesucht wird, in genau so vielen Schritten bestimmt wird, wie die Suche dann anschließend auch braucht. Natürlich kann statt binärer Suche auch jedes andere Suchverfahren für sortierte Folgen benutzt werden.

3.1.2 Selbstanordnende Folgen

Folge M gespeichert als Liste, Aufwand für Zugriff auf i-tes Element ist i (lineare Suche).

Idee Umordnung so, dass häufiger angefragte Elemente weiter vorne stehen.

Sind die Anfragehäufigkeiten bekannt, dann kann man die Folge einfach sortieren. Für den Fall, dass die Häufigkeiten unbekannt sind, gibt es drei übliche Adaptionsstrategien:

- *MF* (move to front): Das Element, auf das gerade zugegriffen wurde, wird an die erste Stelle gesetzt.
- *T* (transpose): Das Element, auf das gerade zugegriffen wurde, wird mit seinem Vorgänger vertauscht.
- *FC* (frequency count): Sortiere die Liste immer entsprechend einem Zähler für die Zugriffshäufigkeiten.

3.8 Beispiel
$M = \{1, \ldots, 8\}$, Zwei Zugriffsfolgen

1. $10 \times (1, 2, \ldots, 8)$
2. $10 \times 1, 10 \times 2, \ldots, 10 \times 8$

Bei beliebiger statischer Anordnung wird auf alle Elemente $10\times$ zugegriffen, die Laufzeit ist daher immer

$$10 \cdot \sum_{i=1}^{8} i = 10 \cdot \frac{8 \cdot 9}{2} = 360$$

im Mittel also $360/80 = 4.5$.

MF: beginne mit $M = (1, \ldots, 8)$

1. *die ersten 8 Zugriffe benötigen $\sum_{i=1}^{8} i = 36$, wonach $M = (8, 7, 6, \ldots, 1)$. Jeder weitere Zugriff benötigt 8 Schritte, da das gesuchte Element immer gerade am Ende der Liste ist. Insgesamt im Mittel $(\sum_{i=1}^{8} i + 9 \cdot 8 \cdot 8)/80 = 7.65$.*
2. *die ersten 10 Zugriffe erfolgen auf das erste Element, dann 1 Zugriff auf das Zweite, 9 auf das Erste, dann 1 Zugriff auf das Dritte, 9 auf das Erste, usw. Insgesamt im Mittel also $(\sum_{i=1}^{8} i + 8 \cdot 9 \cdot 1)/80 = 1.35$.*

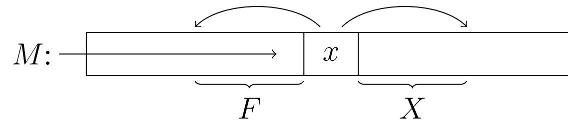
□

MF kann also gut oder schlecht sein, je nach Zugriffsreihenfolge; ähnlich für T und FC (hier evtl. noch zusätzlicher Speicher für Häufigkeitszähler). Experimentell zeigt sich, dass T schlechter als MF ist. FC und M sind ähnlich, wobei MF manchmal besser abschneidet.

Allgemeine Aussagen zur Güte? Wir wollen mit einem beliebigen Algorithmus A vergleichen:

3.9 Definition
Sei S eine Zugriffsfolge. Wir definieren

$C_A(S)$	*Kosten für Zugriffe S.*
$F_A(S)$	*kostenfreie Vertauschungen von benachbarten Elementen an den jeweils durchsuchten Positionen.*
$X_A(S)$	*kostenpflichtige Vertauschungen.*



Insbesondere gilt:

$X_{MF}(S) = 0$	für alle S
$F_A(S) \leq C_A(S) - \|S\|$	für alle A, S da man bei Kosten i für einen Zugriff mit maximal $i-1$ Elementen kostenfrei tauschen kann.

3.10 Satz

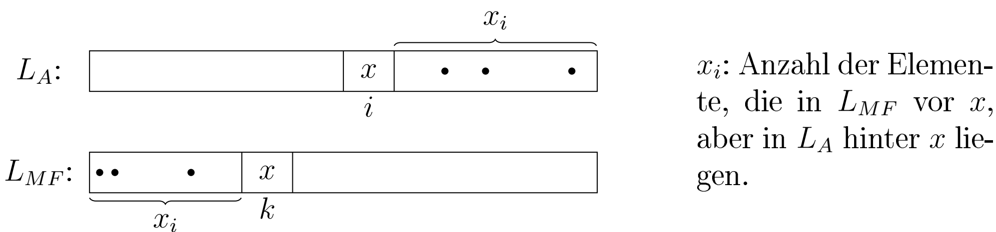
Für jeden Algorithmus A zur Selbstanordnung von Listen gilt für jede Folge S von Zugriffen

$$C_{MF}(S) \;\leq\; 2 \cdot C_A(S) + X_A(S) - F_A(S) - |S|$$

Beweis. [Amortisierte worst-case Analyse]

Um die unterschiedlichen Kosten von MF und A beurteilen zu können (ohne A zu kennen) führen wir Buch über den Zustand der Liste, indem wir die Anzahl der Inversionen (Paare von Elementen, die nicht in der gleichen Reihenfolge auftreten) abschätzen.

Beide beginnen mit derselben Ordnung $L_{MF}^0 = L_A^0$, die Zahl der Inversionen ist $inv(L_A^0, L_{MF}^0) = 0$. Was ändert sich durch einen Zugriff auf das i-te Element von L_A?



x_i: Anzahl der Elemente, die in L_{MF} vor x, aber in L_A hinter x liegen.

Da MF x nach vorne bringt		Vertauschungen von A bewirken	
$-x_i$	Inversionen	$-F_A(i)$	Inversionen
$+(k-1-x_i)$	Inversionen	$+X_A(i)$	Inversionen

Also ist die Anzahl der Inversionen bei Zugriff t

$$inv(L_A^t, L_{MF}^t) \;=\; inv(L_A^{t-1}, L_{MF}^{t-1}) - x_i + (k-1-x_i) - F_A(i) + X_A(i)$$

für $S = (s_1, \ldots, s_m)$ mit $s_t = i$ und $t \in \{1, \ldots, m\}$.

Statt der echten Kosten c_t für MF betrachten wir die „amortisierten Kosten“ a_t aus echten Kosten und Investitionen in die Listenordnung:

$$\begin{aligned} a_t &= c_t + inv(L_A^t, L_{MF}^t) - inv(L_A^{t-1}, L_{MF}^{t-1}) \\ &= k - x_i + (k - 1 - x_i) - F_A(i) + X_A(i) \\ &= 2 \underbrace{(k - x_i)}_{\substack{\leq\ i \\ \text{\# Elemente, die in} \\ \text{beiden Listen vor } x: \\ k-1-x_i \leq i-1 \\ \Leftrightarrow k - x_i \leq i}} -1 - F_A(i) + X_A(i) \end{aligned}$$

Da die amortisierten Kosten

$$\begin{aligned} \sum_{t=1}^{|S|} a_t &= \sum_{t=1}^{|S|} \left(c_t + inv(L_A^t, L_{MF}^t) - inv(L_A^{t-1}, L_{MF}^{t-1})\right) \\ &= \underbrace{-inv(L_A^0, L_{MF}^0)}_{=0} + \sum_{t=1}^{|S|} c_t + inv(L_A^{|S|}, L_{MF}^{|S|}) \\ &\geq \sum c_t = C_{MF}(S) \end{aligned}$$

folgt

$$\begin{aligned} C_{MF}(S) \leq \sum_{t=1}^{|S|} a_t &\leq \sum_{t=1}^{|S|} 2i - 1 - F_A(i) + X_A(i) \\ &= 2\left(\sum_{t=1}^{|S|} i\right) - |S| - F_A(S) + X_A(S) \\ &= 2C_A(S) - |S| - F_A(S) + X_A(S) \end{aligned}$$

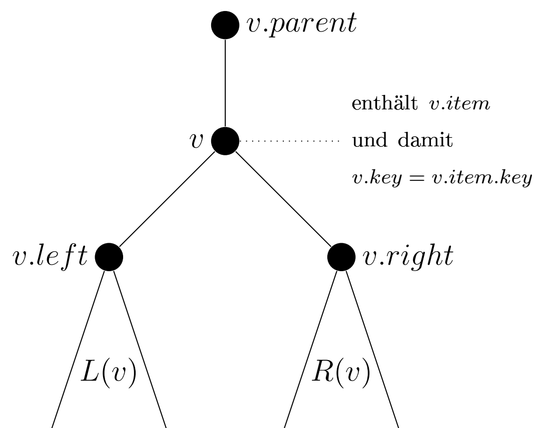
□

MF ist also im Wesentlichen mindestens halb so gut wie ein beliebiger Algorithmus A, der sogar speziell für eine schon vorher bekannte Zugriffsfolge S antworten wird.

3.2 Suchbäume

3.2.1 Binäre Suchbäume

Node	
`node`	parent()
`node`	left(), right()
`item`	item()



Suchbaumeigenschaft:

$$w.key < v.key \quad \forall w \in L(v)$$
$$w.key > v.key \quad \forall w \in R(v)$$

(vgl. Heap, wegen ADT `Dictionary` jetzt aber nur noch eindeutige Schlüssel)

Wegen der Suchbaumeigenschaft können die Elemente eines Suchbaums T durch inorder-Durchlauf in Linearzeit sortiert ausgegeben werden (Aufruf: `inordertraversal`$(T.root)$).

Algorithmus 20: Inorder-Traversal

```
inordertraversal(v) begin
    if v ≠ nil then
        inordertraversal(v.left)
        print v.key
        inordertraversal(v.right)
end
```

Im Folgenden werden die Methoden der ADT `Dictionary` mit binären Suchbäumen implementiert:

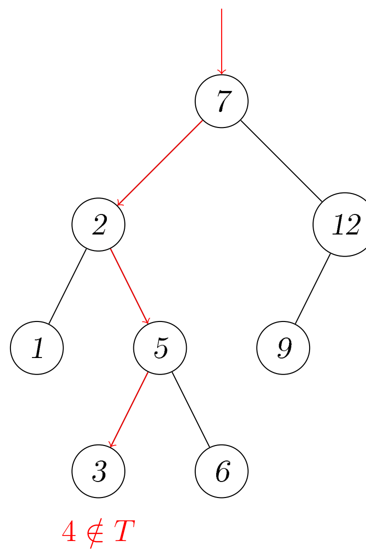
Algorithmus 21: find (x)

```
v ← search(T, x)
if v ≠ nil then
    return v.item
else                          // kein Element in T hat Schlüssel x
    return nil

search(T, x)  begin
    v ← T.root
    while (v ≠ nil) and (v.key ≠ x) do
        if x < v.key then
            v ← v.left
        else
            v ← v.right
    return v
end
```

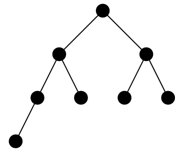
3.11 Beispiel

`find`(4)

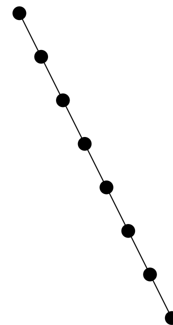


Die Laufzeit ist offensichtlich linear in der Höhe des Baumes. Wie hoch kann ein binärer Baum sein, wenn er die Suchbaumeigenschaft erfüllt?

→ Ähnlich der Aufteilung bei QuickSort ist die Höhe (z.B. für $n = 8$)



mindestens $\lfloor \log n \rfloor$



höchstens $n - 1$

Algorithmus 22: insert (i)

```
v ← T.root
if v = nil then
    T.root ← newnode(i)
else
    while v ≠ nil do
        u ← v
        if i.key < v.key then
            v ← v.left
        else
            v ← v.right
    if i.key = u.key then
        print(Fehler)                    // Schlüssel bereits vergeben
    else
        v ← newnode(i)
        if i.key < u.key then
            u.left ← v
        else
            u.right ← v
        v.parent ← u
```

Algorithmus 23: remove (x)

```
v ← search(x)
if v ≠ nil then
    if v.left ≠ nil then
        w ← v.left
        while w.right ≠ nil do
            w ← w.right
        if w = v.left then
            v.left ← w.left
        else
            (w.parent).right ← v.left
        v.item ← w.item
        deletenode(w)
    else
        if v = T.root then
            T.root ← v.right
        else
            u ← v.parent
            if x < u.key then
                u.left ← v.right
            else
                u.right ← v.right
```

3.12 Bemerkung
Falls der Baum in einem Array realisiert ist, kann $parent$ weggelassen und während des Abstiegs identifiziert werden.

3.2.2 AVL-Bäume

Ziel geringe Baumhöhe, da diese di Laufzeiten der Wörterbuch-operationen bestimmt.

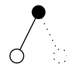
Balanciertheitseigenschaft: Für jeden (inneren) Knoten eines Binärbaums gilt, dass die Höhe der Teilbäume der beiden Kinder sich um höchstens 1 unterscheidet.

Ein binärer Suchbaum, der die Balanciertheitseigenschaft erfüllt, heisst AVL-Baum (Adelson-Velskij, Landis).

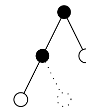
3.13 Satz
Ein AVL-Baum mit n Knoten hat Höhe $\Theta(\log n)$.

Beweis. Jeder Binärbaum mit n Knoten hat Höhe $\Omega(\log n)$. Für die Abschätzung nach oben betrachte das umgekehrte Problem: Wieviele innere Knoten hat ein AVL-Baum der Höhe h mindestens? Wir nennen diese Zahl $n(h)$.

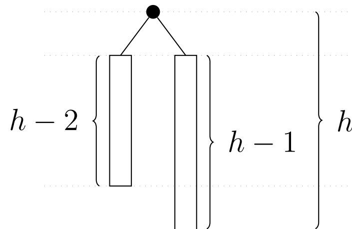
$n(1) = 1$



$n(2) = 2$



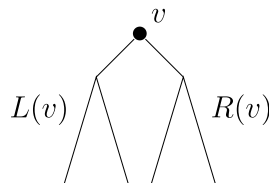
$k \geq 3$: Die beiden Unterbäume der Wurzel müssen ebenfalls AVL-Bäume sein und haben mindestens Höhe $h-1$ und $h-2$.



$$\begin{aligned} \curvearrowright \quad n(h) &\geq n(h-1) + n(h-2) + 1 \\ &> 2 \cdot n(h-2) \\ &\quad \text{da } n(h) \text{ offensichtlich streng} \\ &\quad \text{monoton wachsend} \\ &\geq 2^{\lceil \frac{h}{2} \rceil - 1} \cdot n(\underbrace{h - 2 \cdot (\lceil h/2 \rceil - 1)}_{\in\{1,2\}}) \\ &\geq 2^{\lceil \frac{h}{2} \rceil - 1} \\ \Leftrightarrow \quad \log n(h) &\geq \frac{h}{2} - 1 \\ \Leftrightarrow \quad h &\leq 2 \log n(h) + 2 \end{aligned}$$

Also hat ein AVL-Baum mit n Knoten Höhe $\mathcal{O}(\log n)$. □

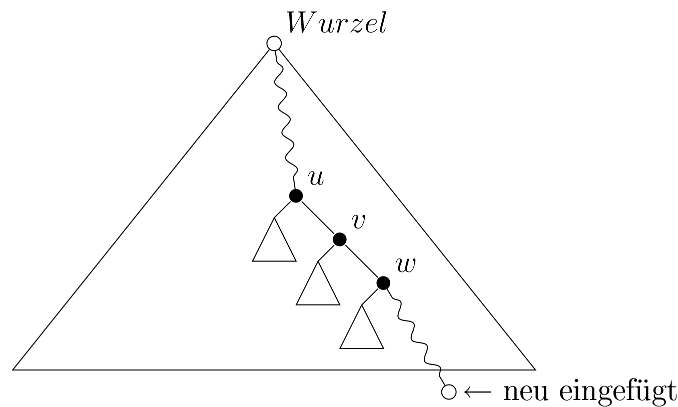
`find` kann unverändert implementiert werden, bei `insert` und `remove` muss jedoch die Balanciertheit erhalten werden. Dazu speichert man in jedem Knoten v die Balance an diesem Knoten:



$$\mathit{balance}(v) = h(R(v)) - h(L(v)) \in \{-1, 0, 1\}$$

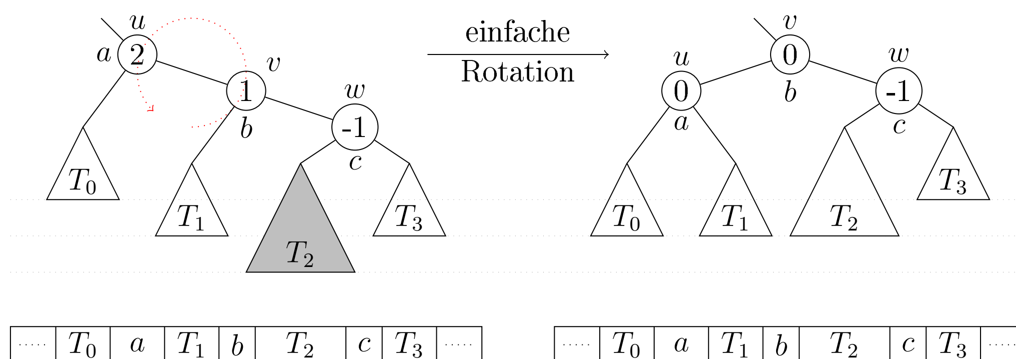
Fügt man wie gehabt ein neues Blatt ein, wächst die Höhe der Teilbäume von Vorfahren um höchstens 1. $\curvearrowright$ evtl. ist die Balanciertheitseigenschaft verletzt.

Sei u der erste Knoten auf dem Weg vom Eingefügten zur Wurzel, der nicht balanciert ist (d.h. $|balance(u)| > 1$); v und w sind die beiden Nachfahren auf dem Weg zum eingefügten Knoten (Balanciertheitseigenschaft ist frühestens beim dritten Knoten verletzt!)

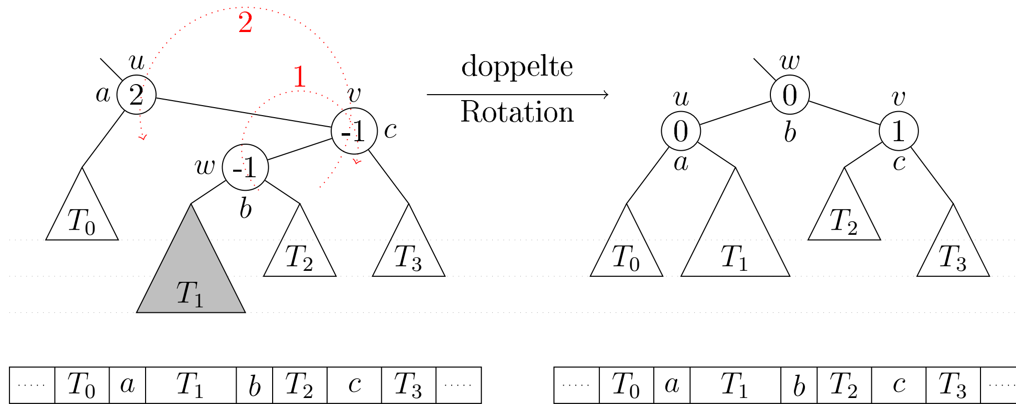


Wir bezeichenen mit a, b, c die Knoten u, v, w in inorder-Reihenfolge und mit T_0, T_1, T_2, T_3 deren Unterbäume in inorder-Reihenfolge.

4 Fälle



Symmetrisch für $balance(u) = -2$.

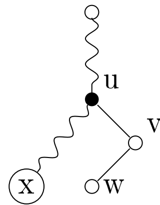


Symmetrisch für $balance(u) = -2$.

Höhe des gesamten Teilbaums ist anschließend wie vor der Einfügeoperation.
$\curvearrowright$ alle Vorfahren balanciert.

Entfernen beginnt ebenfalls wie beim gewöhnlichen binären Suchbaum. Dabei wird ein Knoten x gelöscht, sodass auf dem Weg von x zur Wurzel die Balanciertheitseigenschaft evtl. an <u>einem</u> Knoten u verletzt ist (da ein Teiblaum fehlende Höhe für Vorfahren <u>keine</u> Auswirkung hat). Wir bezeichnen mit

- v Kind von u mit größerer Höhe,
- w Kind von v mit größerer Höhe (bei Gleichheit beliebig)



→ Situation wie vorher, einfache oder doppelte Rotation.

Achtung: Unterbaum mit Wurzel b möglicherweise um 1 weniger hoch. $\curvearrowright$ Fortsetzen bis zur Wurzel. ?? was ist b

3.2.3 Rot-Schwarz-Bäume

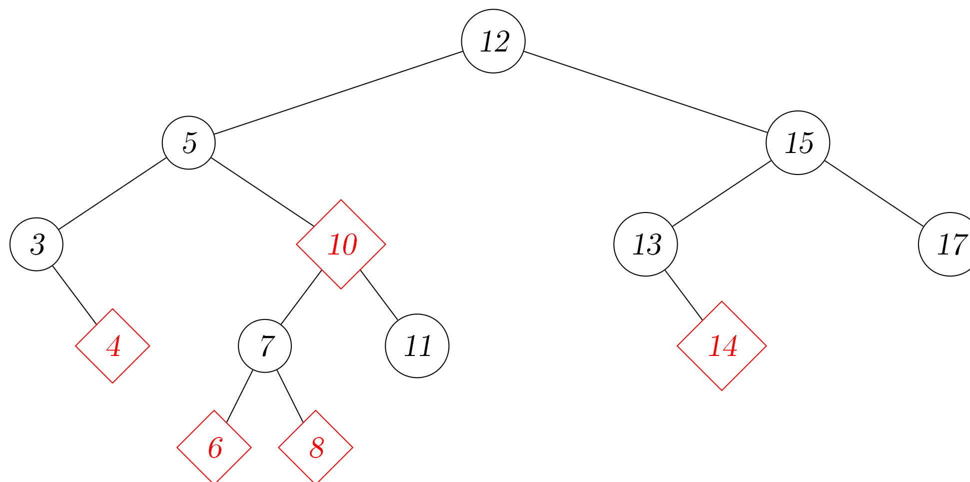
Idee Statt der Balanciertheit speichere an jedem Baumknoten eine Farbmarkierung:

schwarz:	normaler Knoten
rot:	Ausgleichsknoten (für Höhenunterschiede)

Da im Idealfall alle Blätter von der gleichen Tiefe sein sollen, fordere für die Färbung folgende schwächere Invariante:

Wurzelbedingung:	Die Wurzel ist schwarz.
Farbbedingung:	Die Kinder von roten Knoten sind schwarz (oder **nil**).
Tiefenbedingung:	Alle Blätter haben die gleiche schwarze Tiefe (definiert als die Anzahl schwarzer Vorfahren -1).

3.14 Beispiel



○ *schwarz* *schwarze Tiefe:* 2

◇ *rot*

3.15 Satz

Die Höhe eines rot-schwarzen Baumes mit n Knoten ist $\Theta(\log n)$.

Beweis. Höhe $\Omega(\log n)$ ist wieder klar, weil es sich um einen Binärbaum handelt.

Da ein roter Knoten keinen roten Elternknoten haben kann, ist die Höhe nicht größer als zweimal die maximale schwarze Tiefe eines Blattes. Da alle

Blätter die gleiche schwarze Tiefe haben, kann diese aber nicht größer als $\lfloor \log n \rfloor$ sein. □

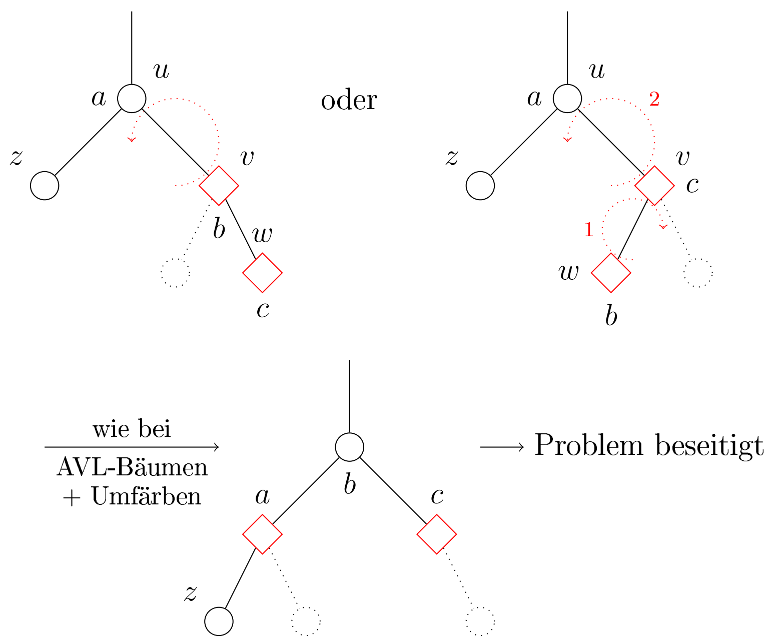
`insert` Beginnen wie bei binärem Suchbaum. ↷ Element wird in ein neues Blatt w eingefügt.

$$\text{Färbe } w \begin{cases} \text{schwarz,} & \text{falls Wurzel} \\ \text{rot} & \text{sonst} \end{cases}$$

Wurzel- und Tiefenbedingung bleben so erfüllt, aber die Farbbedingung ist verletzt, falls der Elternknoten v von w auch rot ist. Wegen der Wurzelbedingung ist v dann nicht die Wurzel und hat einen schwarzen (sonst schon vorher Konflikt) Elternknoten u. Diese Situation heißt <u>doppelrot bei w</u>.

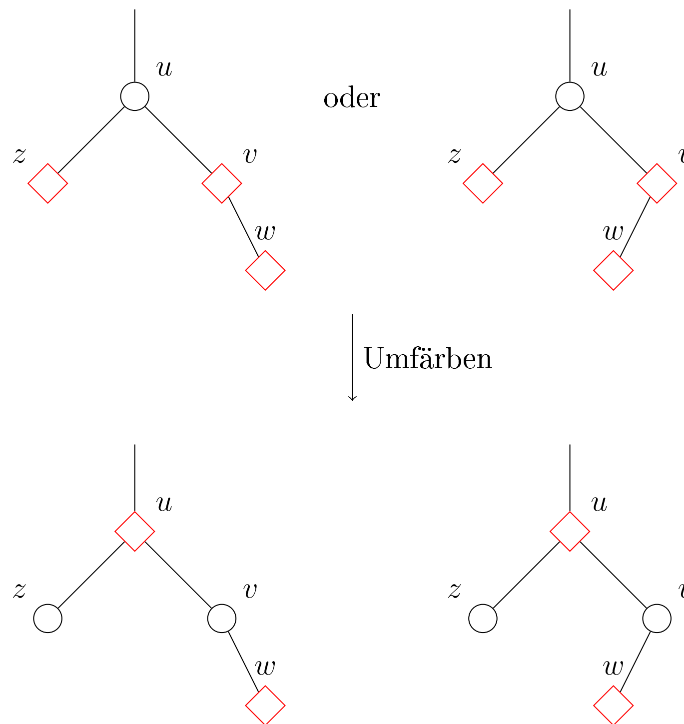
habe b und c bei rechtem Baum vertauscht

<u>Fall 1:</u> (der Geschwisterknoten z von v ist schwarz)



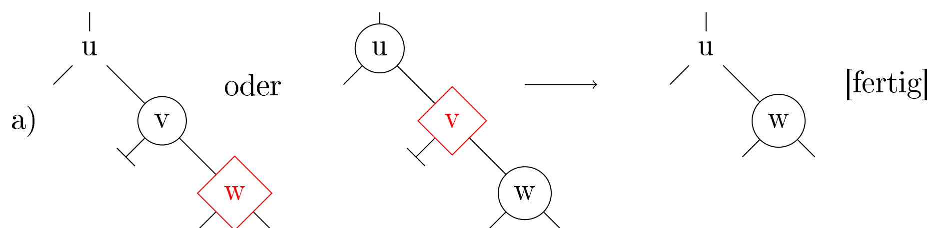
Die beiden symmetrischen Fälle (wie bei AVL) werden analog behandelt.

<u>Fall 2:</u> (der Geschwisterknoten z von v ist rot)

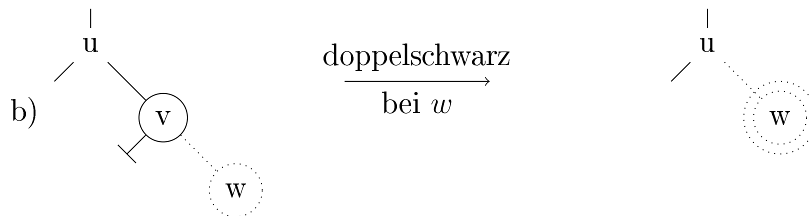


Die beiden symmetrischen Fälle (wie bei AVL) werden analog behandelt. Falls u die Wurzel ist, dann wird auch u schwarz gefärbt. Falls doppelrot jetzt bei u auftritt, werden Fall 1 und 2 iteriert, bis die Wurzel erreicht oder das Problem wegrotiert ist.

`remove` Beginnen wie bei binärem Suchbaum. $\curvearrowright$ Es wird ein Knoten v mit höchstens einem Kind gelöscht.

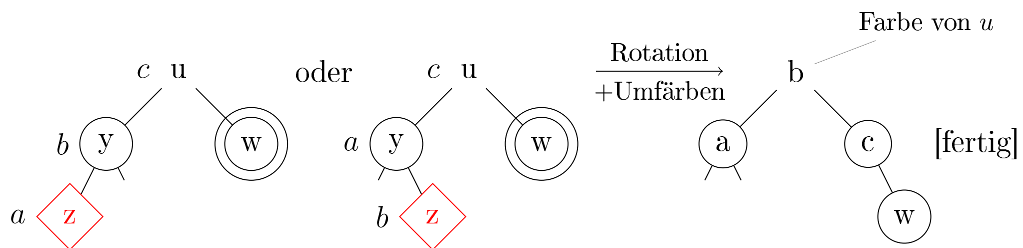


b)



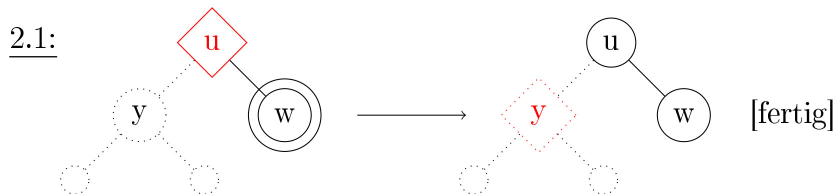
(zeigt an, dass Tiefenbedingung verletzt) Falls w vorhanden, betrachte Geschwisterknoten y von w.

Fall 1: (y schwarz mit mindestens einem roten Kind z)

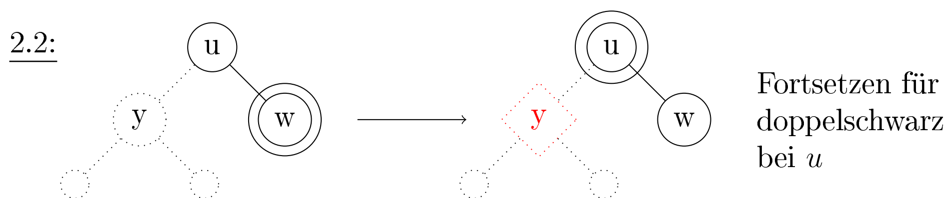


Fall 2: (y schwarz ohne rotes Kind, oder y gibt es nicht)

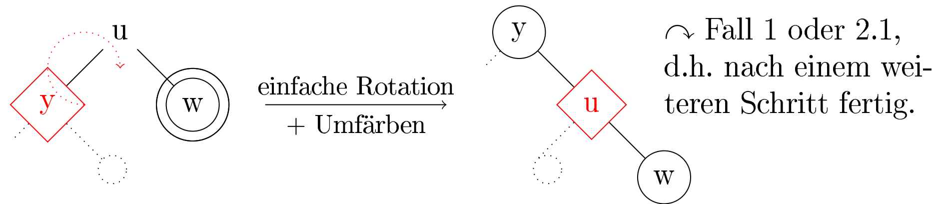
2.1:



2.2:



Fall 3: (y rot)



stimmt das? doppelrot bei insert,Fall2 bzw. doppelschwarz bei remove, Fall2.2

3.16 Satz
`find`, `insert` *und* `remove` *sind in rot-schwarzen Bäumen in Zeit* $\mathcal{O}(\log n)$ *realisierbar.* `insert` *und* `remove` *benötigen maximal eine bzw. zwei (einfache oder doppelte) Rotationen.*

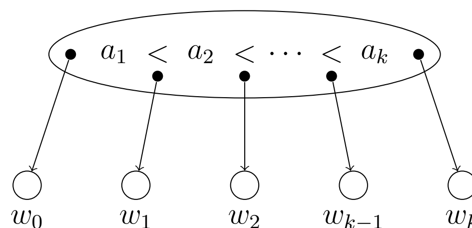
[Erinnerung: `remove` aus AVL-Baum benötigt evtl. $\mathcal{O}(\log n)$ Rotationen]

3.2.4 B-Bäume

Suchbäume können als Indexstruktur für extern gespeicherte Daten verwendet werden. Ist der Index allerdings selbst zu groß für den Hauptspeicher, dann werden viele Zugriffe auf möglicherweise weit verstreut liegende Werte nötig (abhängig davon, wie der Baumdurchlauf und die Speicherung der Knoteninformationen zusammenpassen).

Idee Teile Index in Seiten auf (etwa in Größe von Externspeicherblöcken) und sorge dafür, dass Suche nur wenige Seiten benötigt.

Ein Vielwegbaum speichert pro Knoten mehrere Schlï¿œssel in aufsteigender Reihenfolge und zwischen je zwei Schlüsseln (und vor dem Ersten und nach dem Letzten) einen Zeiger auf ein Kind. Die Zahl der Kinder ist damit immer 1 größer als die der Schlüssel.



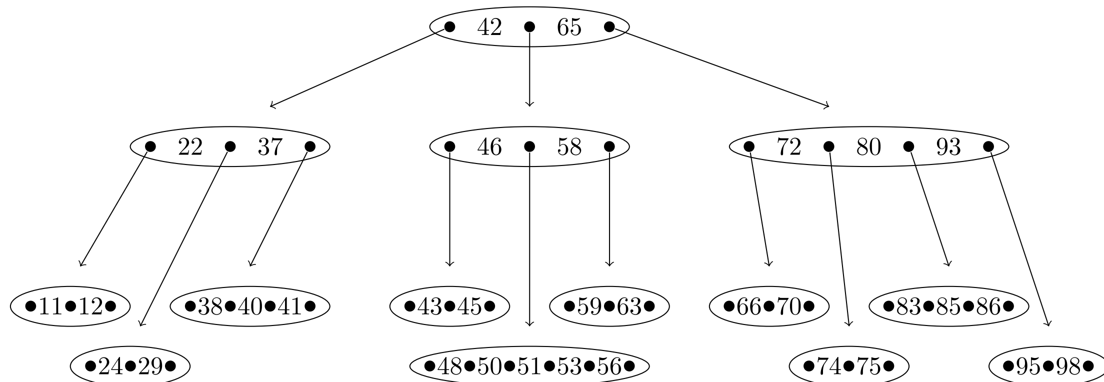
Suchbaum, falls für alle Knoten v mit Schlüssel $a_1, \dots, a_k$ und Kindern $w_0, \dots, w_k$ gilt:

$$\text{Alle Schlüssel in } T(w_{i-1}) < a_i < \text{ alle Schlüssel in } T(w_i) \quad i = 1, \dots, k$$

B-Baum der Ordnung $d, d \geq 2$: Vielwegsuchbaum mit

Wurzelbedingung:	Die Wurzel hat mindestens 2 und höchstens d Kinder.
Knotenbedingung:	Jeder andere Knoten hat mindestens $\lceil d/2 \rceil$ und höchstens d Kinder.
Schlüsselbedingung:	Jeder Knoten mit i Kindern hat $i-1$ Schlüssel.
Tiefenbedingung:	Alle Blätter haben dieselbe Tiefe.

3.17 Beispiel
(B-Baum der Ordnung 6)



`find` Suche in Knoten; falls Schlüssel nicht vorhanden, liegt er zwischen zwei enthaltenen Schlüsseln (bzw. vor dem Ersten oder hinter dem Letzten) $\curvearrowright$ Suche an entsprechendem Kind fortsetzen.

3.18 Satz
Ein B-Baum der Ordnung d mit n Knoten hat Höhe $\Theta(\log_d n)$.

Beweis. Zeige zunächst: B-Baum mit n Schlüsseln hat $n+1$ externe Knoten („**nil**-Zeiger“).

Induktion über Höhe h:

$h = 0$ Wurzel hat $1 \leq i \leq d-1$ Schlüssel und $i+1$ Kinder, die alle externe Knoten sind.

$h \to h+1$ Wurzel hat $1 \leq i \leq d-1$ Schlüssel und $i+1$ Kinder, die alle Wurzeln von Teilbäumen der Höhe h sind. Diese haben $n_0, \ldots, n_i$ viele Schlüssel und $(n_0+1) + \ldots + (n_i+1)$ externe Knoten. Es gibt also insgesamt

$n_0 + \ldots + n_i + i$ viele Schlüssel und
$(n_0+1) + \ldots + (n_i+1) = n_0 + \ldots + n_i + i + 1$ viele externe Knoten.

Wegen der Tiefen- und Knotenbedingung für die Höhe $h(n)$ eines B-Baumes mit $n+1$ externen Knoten dann aber

$$2 \cdot \left\lceil \frac{d}{2} \right\rceil^h \leq n \leq d^{h+1}$$

□

konnte Sinn des Bsp. nicht erkennen

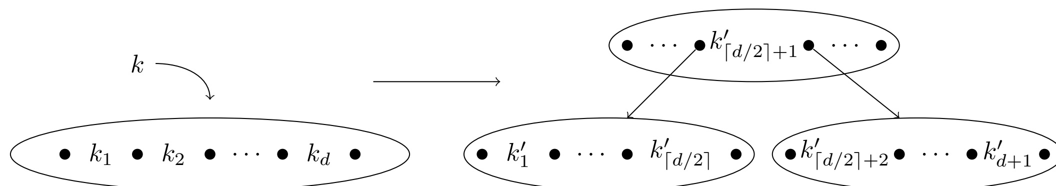
3.19 Beispiel

`insert` Suche analog zu vorher das Blatt, in das eingefügt werden kann. Wieviele Schlüssel hat dieses Blatt?

$< d-1$ einfügen, fertig.

$= d-1$ Überlauf $\to$ teile die d Schlüssel auf in die ersten und letzten jeweils $\lceil d/2 \rceil$ Viele, für die zwei neue Knoten erzeugt werden, und das $\lceil d/2 \rceil$-te, das im Elternknoten eingefügt wird. $\to$ evtl. Überlauf im Elternknoten, dann iterieren [evtl. neue Wurzel].

Indizes in Grafik richtig?

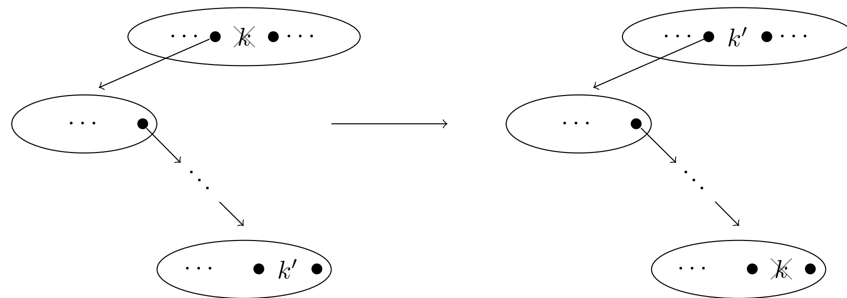


`remove` Suche den Knoten aus dem ein Schlüssel gelöscht werden soll. Wieviele Schlüssel enthält er?

$< \lceil \frac{d}{2} \rceil - 1$ löschen, fertig.

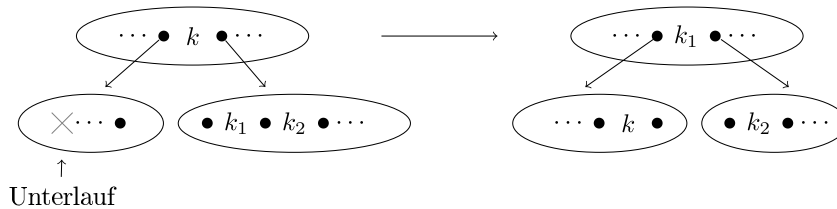
$= \lceil \frac{d}{2} \rceil - 1$ Unterlauf

a) Knoten ist innerer Knoten (d.h. kein Blatt)
   → ersetze Schlüssel durch inorder-Vorgänger (oder Nachfolger), d.h. letzten Schlüssel im rechtesten Blatt des vorangehenden Teilbaums.

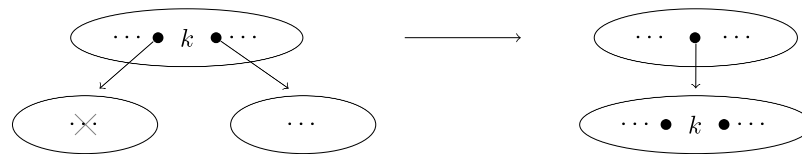


   → fahre fort mit b).

b) Knoten ist Blatt
   → falls ein Geschwisterknoten mehr als $\lceil \frac{d}{2} \rceil - 1$ Schlüssel enthält, verschiebe den Letzten bzw. Ersten von dort in den gemeinsamen Vorgänger und den trennenden Schlüssel von vorher in den Knoten mit Unterlauf.



   → sonst verschmilz den Knoten mit einem seiner Geschwister und dem trennenden Schlüssel aus dem Vorgänger.



Falls Unterlauf in Vorgänger: iteriere b) für innere Knoten.
[Bemerkung: anders als in a) gibt es jetzt kein überzähliges Kind.]