7.4 Transformation des Definitions- bzw. Wertebereichs

Beispiel 9

$$f_0 = 1$$
$$f_1 = 2$$
$$f_n = f_{n-1} \cdot f_{n-2} \text{ für } n \geq 2 \, .$$

Setze

$$g_n := \log f_n \, .$$

Dann gilt

$$g_n = g_{n-1} + g_{n-2} \text{ für } n \geq 2$$
$$g_1 = \log 2 = 1, \; g_0 = 0 \text{ (für } \log = \log_2 \text{)}$$
$$g_n = F_n \text{ (n-te Fibonacci-Zahl)}$$
$$f_n = 2^{F_n}$$

Beispiel 10

$$f_1 = 1$$
$$f_n = 3f_{\frac{n}{2}} + n; \text{ für } n = 2^k \ ;$$

Setze

$$g_k := f_{2^k} \ .$$

Beispiel 10

Dann gilt:

$$g_0 = 1$$
$$g_k = 3g_{k-1} + 2^k,\ k \geq 1$$

Damit ergibt sich:

$$g_k = 3^{k+1} - 2^{k+1},\ \text{also}$$
$$f_n = 3 \cdot 3^k - 2 \cdot 2^k$$
$$= 3(2^{\log 3})^k - 2 \cdot 2^k$$
$$= 3(2^k)^{\log 3} - 2 \cdot 2^k$$
$$= 3n^{\log 3} - 2n\ .$$

Kapitel II Höhere Datenstrukturen

1. Grundlegende Operationen

Es sei U das Universum von Schlüsseln mit einer (totalen) Ordnung $\leq$. $S \subseteq U$ sei eine Teilmenge der Schlüssel. Gegeben seien eine Menge von Datensätzen $x_1, \cdots, x_n$, wobei jeder Datensatz x durch einen Schlüssel $k(x) \in S$ gekennzeichnet ist. Jeder Datensatz x besteht aus seinem Schlüssel $k(x)$ und seinem eigentlichen Wert $v(x)$.

IsElement(k, S):	ist $k \in S$, wenn ja, return $v(k)$
Insert(k, S):	$S := S \cup \{k\}$
Delete$(k; S)$:	$S := S \setminus \{k\}$
FindMin(S):	return $\min S$
FindMax(S):	return $\max S$
DeleteMin(S):	$S := S \setminus \min S$
ExtractMin(S):	return $\min S$, $S := S \setminus \min S$
DecreaseKey(k, Δ, S):	ersetze k durch $k - \Delta$
Union(S_1, S_2):	$S_1 := S_1 \cup S_2$
Find(k):	falls $k \in S$, so finde x mit $k = k(x)$
Merge(S_1, S_2):	$S_1 := S_1 \cup S_2$, falls $S_1 \cap S_2 = \emptyset$
Split(S_1, k, S_2):	$S_2 := \{k' \in S_1 \mid k' \geq k\}$ $S_1 = \{k' \in S_1 \mid k' < k\}$
Concatenate(S_1, S_2):	$S_1 := S_1 \cup S_2$; Vorauss.: *FindMax*$(S_1) \leq$ *FindMin*(S_2)

Datenstrukturklasse	mindestens angebotene Funktionen	realisiert in
Wörterbuch (Dictionary)	*IsElement()*, *Insert()*, *Delete()*	Hashtable, Suchbäume
Vorrangwarteschlange (Priority Queue)	*FindMin()*, *Insert()*, *Delete()*, *[IsElement()]*	balancierte, leftist Bäume, Binomial Queues
Mergeable heaps	*FindMin()*, *Insert()*, *Delete()*, *Merge()*	2-3-Bäume, Binomial Queues, Leftist-Bäume
Concatenable queues	*FindMin()*, *Insert()*, *Delete()*, *Concatenate()*	2-3-Bäume

Definition 11

- Bei einem externen Suchbaum werden die Schlüssel nur an den Blättern gespeichert, die inneren Knoten enthalten Verwaltungsinformationen.
- Bei internen Suchbäumen liegen die Schlüssel an den internen Knoten, die Blätter sind leere Knoten. Zeiger auf Blätter sind daher NIL-Pointer und werden gewöhnlich nicht angegeben.

2. (Balancierte) Suchbäume

Wir betrachten zunächst zwei Familien höhenbalancierter externer Suchbäume.

2.1 (a,b)-Bäume

Definition 12
Ein (a, b)-Baum ist ein externer Suchbaum mit folgenden Eigenschaften:

1. alle Blätter haben die gleiche Tiefe;
2. die Anzahl der Kinder eines jeden internen Knoten ist $\leq b$ und $\geq$ a (für die Wurzel: ≥ 2);
3. es gilt $b \geq 2a - 1$.
4. (a, b)-Bäume mit $b = 2a - 1$ heißen auch B-Bäume.

Bei jedem internen Knoten v mit $d = d(v)$ Kindern werden $d-1$ Schlüssel $k_1, \ldots, k_{d-1}$ gespeichert, so dass (mit $k_0 := -\infty$, $k_d = +\infty$) gilt:

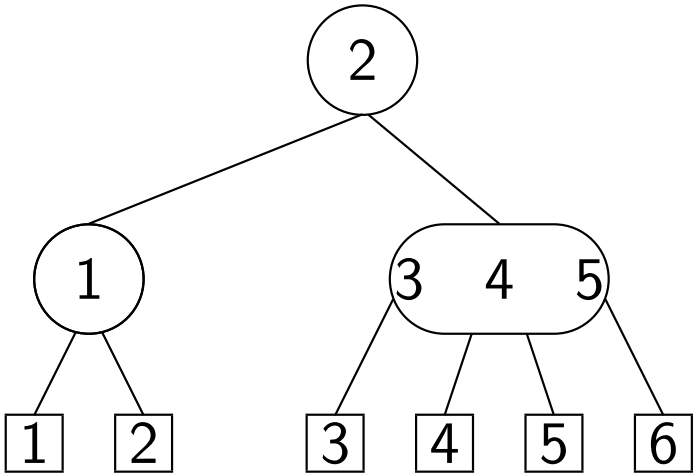
$$k_{i-1} < \text{ alle Schlüssel im } i\text{-ten Unterbaum von } v \leq k_i \ .$$

Also z.B.: $k_i :=$ maximaler Schlüssel im i-ten Unterbaum von v.

Weiter gilt (wie in jedem Suchbaum)

$$\max\{\text{Schlüssel im } i\text{-ten UB}\} \leq \min\{\text{Schlüssel im } i+1\text{-ten UB}\}\,.$$

Beispiel 13



$(2,4)$-Baum

Lemma 14

Sei T ein (a,b)-Baum mit n Blättern und der Höhe h (Höhe definiert als Anzahl der Kanten auf einem Pfad von der Wurzel zu einem Blatt). Dann gilt:

1. $2a^{h-1} \leq n \leq b^h$;
2. $\log_b n = \frac{\log n}{\log b} \leq h \leq 1 + \log_a \frac{n}{2}$.

Beweis:

1. n ist maximal für vollständigen Baum mit Verzweigungsgrad b und minimal für einen Baum, wo die Wurzel Grad 2 und alle anderen internen Knoten Grad a haben.
2. folgt durch Umformung aus (1).

□

Operationen:

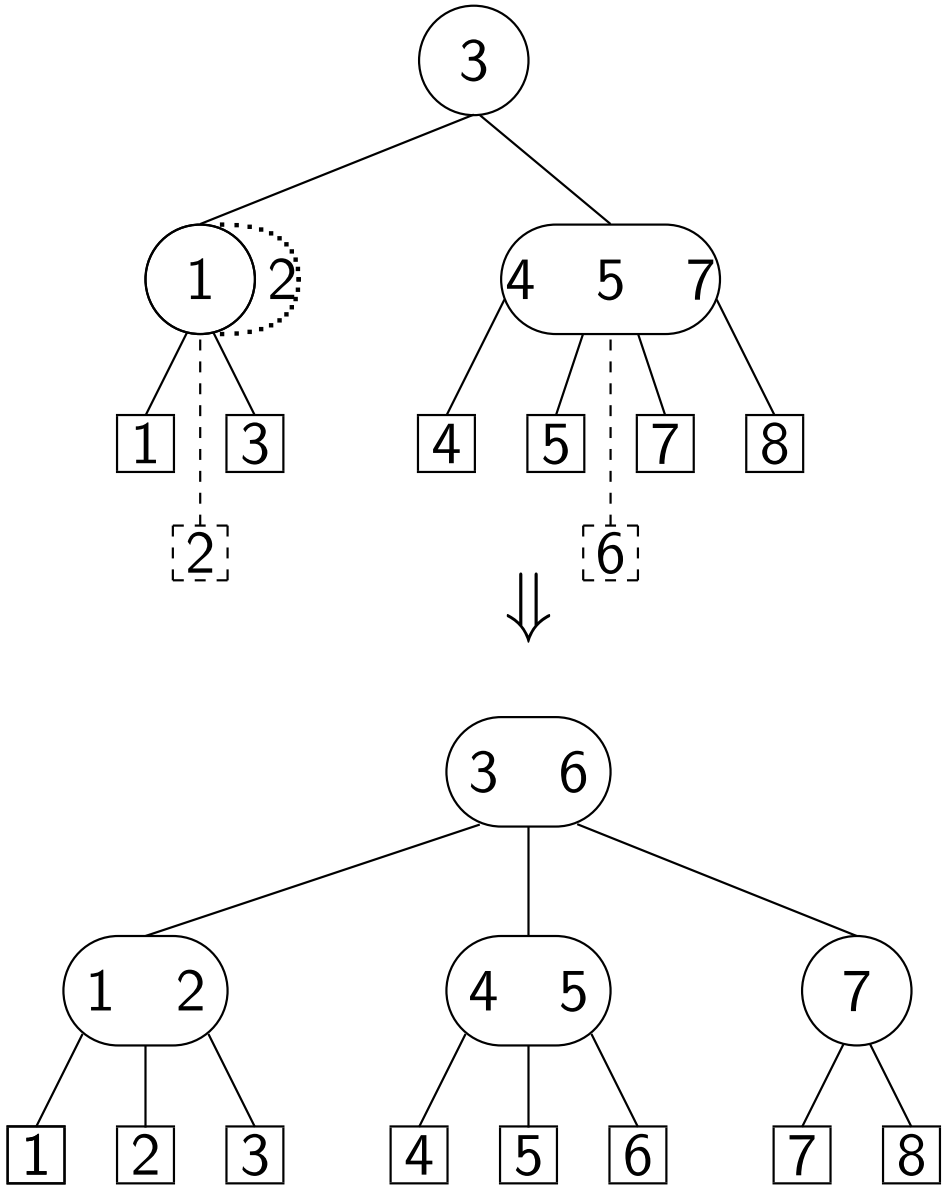
1. *IsElement*(k, S)

$v :=$ `Wurzel von` T`;`
while $v \neq$ `Blatt` **do**
  $i = \min\{s;\ 1 \leq s \leq$ (`#Kinder von` v) `und` $k \leq k_s\}$`;`
  $v :=$ i`-tes Kind von` v`;`
if $k = k(v)$ **then return** $v(k)$ **else return false fi** ;

Zeitbedarf: $\theta(h) = \theta(\log n)$

2. *Insert*(k, S)

```
Führe IsElement(k, S) aus; ⇝ Blatt w;
if k ≠ k(w) then
  co falls k < max S, enthält w den kleinsten
  Schlüssel ∈ S, der > k ist oc
  füge k if k < max S then links else rechts fi von w
  ein;
  v := Vater von w;
  if v hat nun mehr als b Kinder then
    Rebalancierung(v)
  fi
fi
```


3
1 2
4 5 7
1
3
4
5
7
8
2
6
3 6
1 2
4 5
7
1
2
3
4
5
6
7
8

Rebalancierung(v):

- spalte v in zwei Knoten v_1 und v_2, v_1 übernimmt die linke „Hälfte“ der Kinder von v, v_2 den Rest; da

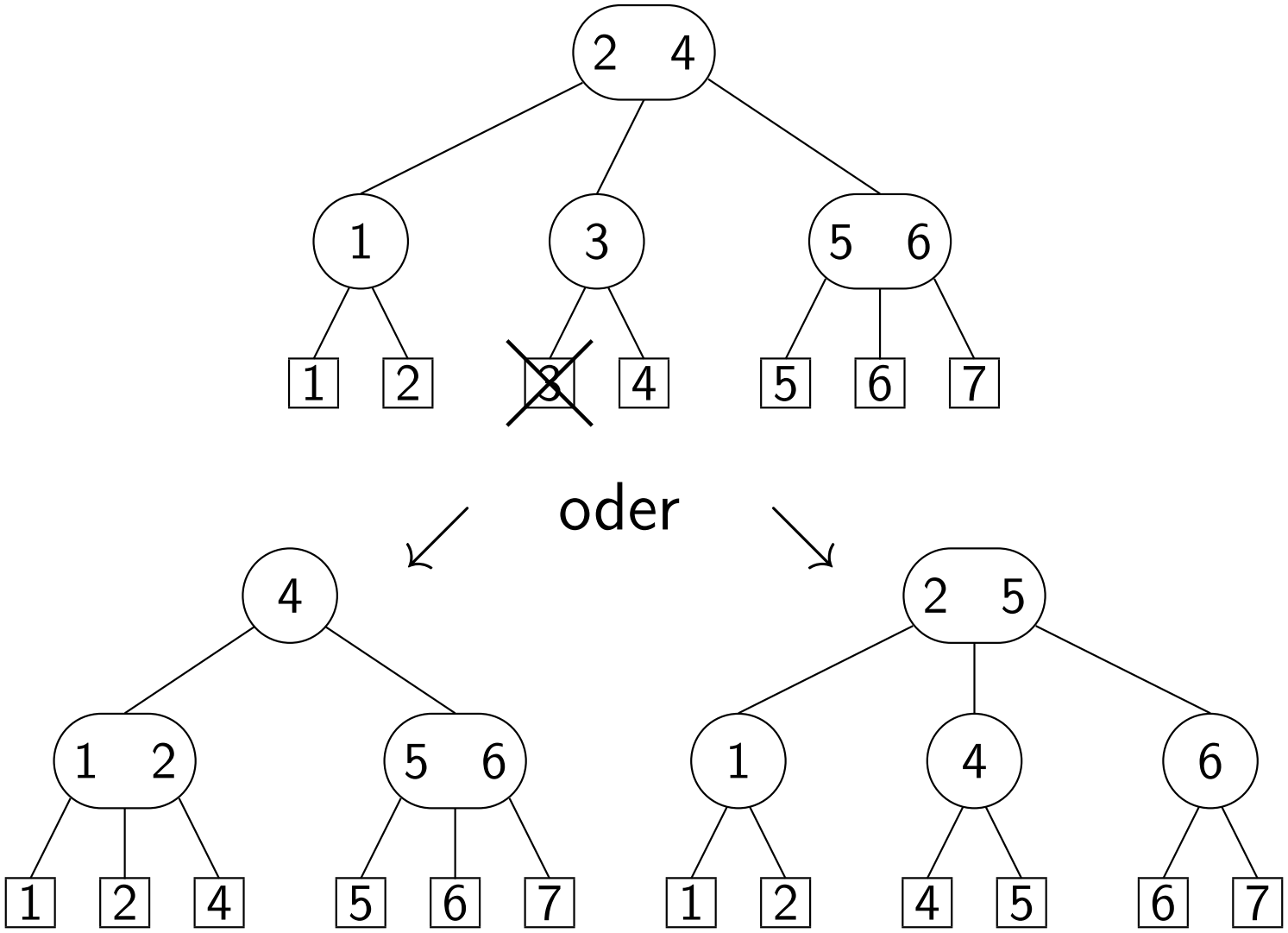
$$a \leq \lfloor \frac{b+1}{2} \rfloor \leq \lceil \frac{b+1}{2} \rceil \leq b \text{ und } b \geq 2a-1\,,$$

  erfüllen v_1 und v_2 die Gradbedingung;
- Vereinfachung: Falls ein unmittelbarer Nachbar v' von v Grad $< b$ hat, übernimmt v' das äußerste linke/rechte Kind von v.
- Falls nun der Vater u von v mehr als b Kinder hat, führe Rebalancierung(u) aus;
- Falls u die Wurzel des Baums ist, kann dadurch eine neue Wurzel geschaffen werden und die Höhe des (a,b)-Baums wachsen.

Kosten $\mathcal{O}(\log n)$

3. *Delete*(k, S):

Führe *IsElement*(k, S) aus. $\rightsquigarrow$ Blatt w. Sei $k(w) = k$;
$v :=$ Vater von w;
Lösche w;
if v hat nunmehr $< a$ Kinder **then**
  führe rekursiv aufsteigend Rebalancierung$'(v)$ durch
fi


2 4
1
3
5 6
1
2
3
4
5
6
7
oder
4
1 2
5 6
1
2
4
5
6
7
2 5
1
4
6
1
2
4
5
6
7

Rebalancierung$'(v)$:

- Falls ein unmittelbarer Nachbar v' von v Grad $> a$ hat, adoptiert v das nächste Kind von v';
- Ansonsten wird v mit einem unmittelbaren Nachbarn verschmolzen; die Gradbedingung für den dadurch entstehenden Knoten ist erfüllt; die Rebalancierung wird rekursiv/iterativ für den Vaterknoten fortgesetzt.

Zeitbedarf: $\mathcal{O}(\log n)$

Spezialfälle von (a, b)-Bäumen:

- $(2, 3)$-Bäume;
- $(2, 4)$-Bäume;
- $(a, 2a - 1)$-Bäume: B-Bäume

Bemerkungen:

1. zur Wahl von a:
   - Daten in RAM $\Rightarrow$ a klein, z.B. $a = 2$ oder $a = 3$.
   - Daten auf Platte $\Rightarrow$ a groß, z.B. $a = 100$.
2. Zur Wahl von b:
   $b \geq 2a$ liefert wesentlich bessere amortisierte Komplexität (ohne Beweis, siehe Mehlhorn).

Top-Down-Rebalancierung: $b \geq 2a$.

Bei der Restrukturierung nach der Top-Down-Strategie folgen wir wie gewohnt dem Pfad von der Wurzel zum gesuchten Blatt. Beim Einfügen stellen wir jetzt jedoch für jeden besuchten Knoten sicher, dass der Knoten weniger als b Kinder hat.
Wenn der gerade betrachtete Knoten ein b-Knoten ist, dann spalten wir ihn sofort auf. Weil der Vater kein b-Knoten ist (das haben wir ja bereits sichergestellt), pflanzt sich der Aufspaltungsprozess nicht nach oben hin fort. Insbesondere kann das neue Element ohne Probleme eingefügt werden, wenn die Suche das Blattniveau erreicht hat.
Damit diese Strategie möglich ist, muss b mindestens $2a$ sein (und nicht nur $2a - 1$), da sonst nach der Spaltung nicht genügend Elemente für die beiden Teile vorhanden sind.

Beim Löschen verfahren wir analog. Für jeden besuchten Knoten, außer der Wurzel des (a, b)-Baumes, stellen wir sicher, dass er mindestens $a + 1$ Kinder hat. Wenn der gerade betrachtete Knoten nur a Kinder hat, so versuchen wir zuerst, ein Element des rechten oder linken Nachbarknoten zu stehlen.
Haben beide Nachbarknoten nur jeweils a Kinder, so verschmelzen wir unseren Knoten mit einem der beiden Nachbarn. Ist der Vater nicht die Wurzel, so hatte er aber vorher mindestens $a + 1$ Kinder, und dieser Verschmelzungsprozess kann sich nicht nach oben fortsetzen. Andernfalls erniedrigt sich der Grad der Wurzel um 1, wenn die Wurzel Grad > 2 hat, oder die alte Wurzel wird gelöscht und der soeben verschmolzene Knoten wird zur neuen Wurzel.

Restrukturierung nach der Top-Down-Strategie sorgt nicht für eine bessere Laufzeit, sondern erleichtert die Synchronisation, wenn mehrere Prozesse gleichzeitig auf einen (a, b)-Baum zugreifen wollen.
Bei herkömmlichen (a, b)-Bäumen schreiten die Such- und die Restrukturierungsoperationen in entgegengesetzter Richtung fort, was dazu führt, dass sich oft Prozesse gegenseitig behindern (der eine will im Baum absteigen, der andere muss auf dem gleichen Pfad aufwärts restrukturieren).
Bei der Top-Down-Strategie gibt es nur Operationsfolgen, die den Baum hinabsteigen. Mehrere Prozesse können so in einer Art Pipeline gleichzeitig einen Pfad mit kurzem Abstand begehen.