Algorithmen & Komplexität

Angelika Steger
Institut für Theoretische Informatik

steger@inf.ethz.ch

Kap. 3: Algorithmische Grundprinzipien

3.1 Divide & Conquer Verfahren „top-down“

- MergeSort
- QuickSort
- Binäre Suche
- Alg. von Strassen

3.2 Dynamische Programmierung „bottom-up“

- Alg. von Floyd-Warshall
- Knapsack Problem (heute)

erlaubt Gaussklammern

Satz 3.1 *(Master-Theorem) Seien $\alpha \geq 1$, $\beta > 1$ und $C \geq 0$ Konstanten und sei $f(n)$ eine positive Funktion. Weiter seien $c_1(n), \ldots, c_\alpha(n)$ Funktionen mit $|c_i(n)| \leq C$ für alle $1 \leq i \leq \alpha$ und $n \in \mathbb{N}$. Ist dann $T(n)$ eine Funktion mit $T(1) = 0$, die für $n \geq 1$ die Rekursionsgleichung*

$$T(n) = T(n/\beta + c_1(n)) + \cdots + T(n/\beta + c_\alpha(n)) + f(n)$$

erfüllt, dann gilt

$$T(n) = \begin{cases} \Theta(n^{\log_\beta \alpha}), & \text{falls } f(n) = O(n^{\log_\beta \alpha - \epsilon}) \text{ für ein } \epsilon > 0, \\ \Theta(f(n)\log n), & \text{falls } f(n) = \Theta(n^{\log_\beta \alpha}(\log n)^\delta) \text{ für ein } \delta \geq 0, \\ \Theta(f(n)), & \text{falls } f(n) = \Omega(n^{\log_\beta \alpha + \epsilon}) \text{ für ein } \epsilon > 0. \end{cases}$$

MergeSort: $C_n = C_{\lceil n/2 \rceil} + C_{\lfloor n/2 \rfloor} + n - 1$

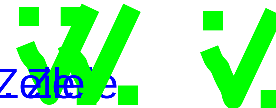
BinäreSuche: $C_n = C_{\lfloor n/2 \rfloor} + 1$

Strassen: $T_n = 7T_{n/2} + 15n^2$

also $\alpha=2$, $\beta=2$, $f(n) = n-1$ 2. Zeile

also $\alpha=1$, $\beta=2$, $f(n) = 1$ 2. Zeile

also $\alpha=7$, $\beta=2$, $f(n) = 15n^2$ 1. Zeile

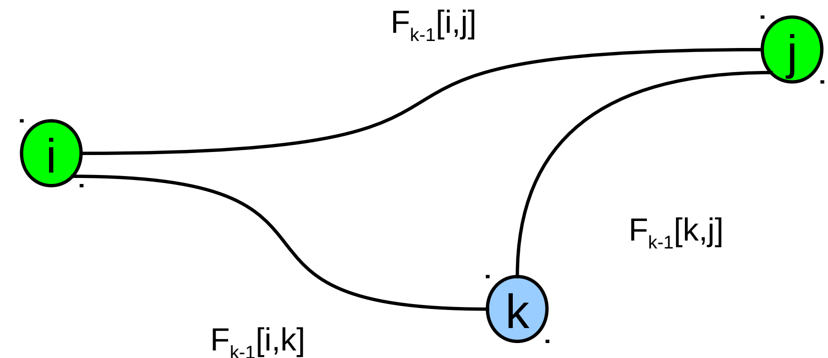


Algorithmus von Floyd-Warshall

Idee:

Berechne eine Folge F_k von Matrizen für k = 1, 2, …, n:

$F_k[i, j]$:= Länge eines kürzesten Pfades zwischen i und j, der nur innere Knoten aus der Menge {1, 2, …, k} hat.



k = 0: trivial

k-1 ➔ k:

$$F_k[i,j] = \min \{ F_{k-1}[i, j], F_{k-1}[i, k] + F_{k-1}[k, j] \}$$

Rucksackproblem

Gegeben:

Eine Kapazität $B \in \mathbf{N}$ des Rucksacks und

n Objekte mit Gewichten $w_1, \ldots, w_n \in \mathbf{N}$ und Profiten $p_1, \ldots, p_n \in \mathbf{N}$.

Gesucht:

Eine optimale Packung des Rucksacks, d.h.

eine Teilmenge $I \subseteq [n]$ mit $\Sigma_{i \in I} w_i \leq B$ und

$$\Sigma_{i \in I} p_i = \max \{ \Sigma_{i \in I'} p_i : I' \subseteq [n] \text{ mit } \Sigma_{i \in I'} w_i \leq B \}$$

Rucksackproblem

Idee:

Berechne zunächst Teillösungen, wobei nur die Objekte 1, 2, …, i berücksichtigt werden:

f [i, t] :=

minimal mögliches Gewicht des Rucksacks, wenn der Profit mindestens t betragen soll und nur die ersten i Objekte zur Verfügung stehen.

i = 1: $f[1, t] = w_1$ für $t \leq p_1$ und $f[1, t] = \infty$ sonst.

i -1 ➔ i :

$$f[i, t] = \min \{ f[i-1, t],\ w_i + f[i-1, t-p_i] \}$$

Rucksackproblem

max. möglicher Profit

Algorithmus 3.1 KNAPSACK PACKING: Berechnung des Wertes

Eingabe: $n, w_1, \ldots, w_n, p_1, \ldots, p_n, B$

Ausgabe: $\max\{\sum_{i \in I} p_i \mid I \subseteq [n], \sum_{i \in I} w_i \leq B\}$

$p \leftarrow \sum_{i=1}^{n} p_i$;

for t **from** 1 **to** p_1 **do** $f[1,t] \leftarrow w_1$;

for t **from** $p_1 + 1$ **to** p **do** $f[1,t] \leftarrow \infty$;

for i **from** 2 **to** n **do begin**

  for t **from** 1 **to** p **do begin**

    if $t \leq p_i$ **then**

      $f[i,t] \leftarrow \min\{f[i-1,t], w_i\}$;

    else

      $f[i,t] \leftarrow \min\{f[i-1,t], w_i + f[i-1, t-p_i]\}$;

  end

end

return $\max\{t \mid f[n,t] \leq B\}$;

max. möglicher Profit mit Gesamtgewicht ≤ B

Kapitel 3:

Algorithmische Grundprinzipien

Kapitel 3.3:

Greedy-Algorithmen

Kap. 3: Algorithmische Grundprinzipien

3.1 Divide & Conquer Verfahren „top-down“

- MergeSort
- QuickSort
- Binäre Suche
- Alg. von Strassen

3.2 Dynamische Programmierung „bottom-up“

- Alg. von Floyd-Warshall
- Knapsack Problem

3.3 Greedy-Algorithmen

- Alg. von Kruskal
- (Alg. von Prim, Djikstra)

Definition: Ein **Matroid** $M=(S,U)$ besteht aus einer endlichen Menge S und einer Familie von unabhängigen Mengen $U \subseteq Power(S)$ von Teilmengen von S, so dass die folgenden drei Bedingungen erfüllt sind:

(M1) $\emptyset \in U$
(M2) $A \in U$ und $B \subseteq A \Rightarrow B \in U$
(M3) $A,B \in U$ und $|B| > |A| \Rightarrow \exists\, b \in B \setminus A$ mit $A \cup \{b\} \in U$

Greedy-Alg. für Matroide

Algorithmus 3.2 Greedy-Algorithmus für Matroide

Eingabe: Matroid $\mathcal{M} = (S, \mathcal{U})$, Gewichtsfunktion $w : S \to \mathbb{R}$.

Ausgabe: Basis A mit minimalem Gewicht: $w(A) = \min\{w(B) \mid B \text{ Basis}\}$.

$A := \emptyset$;

while A ist keine Basis von $\mathcal{M}$ **do begin**

  $X := \{x \in S \setminus A \mid A \cup \{x\} \in \mathcal{U}\}$;

  wähle $x_0 \in X$, so dass $w(x_0) = \min_{x \in X} w(x)$;

  $A := A \cup \{x_0\}$;

end

Kapitel 4:

Datenstrukturen

Datenstrukturen - Motivation

Algorithmus 2.5 Algorithmus von Dijkstra

Eingabe: Ein zusammenhängendes Netzwerk $N = (V, E, \ell)$ mit $\ell \geq 0$ und Knoten $s, t \in V$
Ausgabe: Ein kürzester s–t-Pfad in N

{ *Initialisierung* }
$W := \emptyset$; $\rho[s] := 0$; pred$[s] := nil$;
for all $v \in V \setminus \{s\}$ **do** $\rho[v] := \infty$;

{ *Traversieren und Markieren* }
while $t \notin W$ **do**
  { *Traversieren* }
  Finde ein $x_0 \in V \setminus W$ mit $\rho[x_0] = \min\{\rho[v] \mid v \in V \setminus W\}$;
  $W := W \cup \{x_0\}$;
  { *Markieren* }
  for all $v \in \Gamma(x_0) \cap (V \setminus W)$ mit $\rho[v] > \rho[x_0] + \ell(x_0, v)$ **do**
    $\rho[v] := \rho[x_0] + \ell(x_0, v)$; pred$[v] := x_0$;

Datenstrukturen

Gegeben: Datensätze $x_1,\ldots,x_n$, wobei jeder Datensatz x durch einen Schlüssel key[x] gekennzeichnet ist.

Gesucht: Datenstruktur S, die die folgenden Operationen (oder Teile davon) effizient realisiert:

$Find(k,S)$	liefert $x \in S$ mit $\texttt{key}[x] = k$ bzw. *nil*
$Insert(x,S)$	fügt den Datensatz x ein
$Delete(x,S)$	löscht den Datensatz x
$DecreaseKey(x,\Delta,S)$	ersetzt $\texttt{key}[x]$ durch $\texttt{key}[x] - \Delta$
$FindMin(S)$	liefert $x \in S$ mit $\texttt{key}[x] = \min\{\texttt{key}[y] \mid y \in S\}$
$FindMax(S)$	analog
$ExtractMin(S)$	wie $FindMin(S)$, der ausgegebene Datensatz wird gelöscht
$Union(S_1,S_2)$	vereinigt die beiden Datenstrukturen S_1 und S_2 zu einer neuen Datenstruktur S

Datenstrukturen

Suchbäume:

- Operationen: *Find, Insert, Delete,*

Vorrangwarteschlangen:

- Operationen: *Insert, ExtractMin, DecreaseKey*

Mengen:

- Operationen: *Find, Insert, Union*

Wörterbücher:

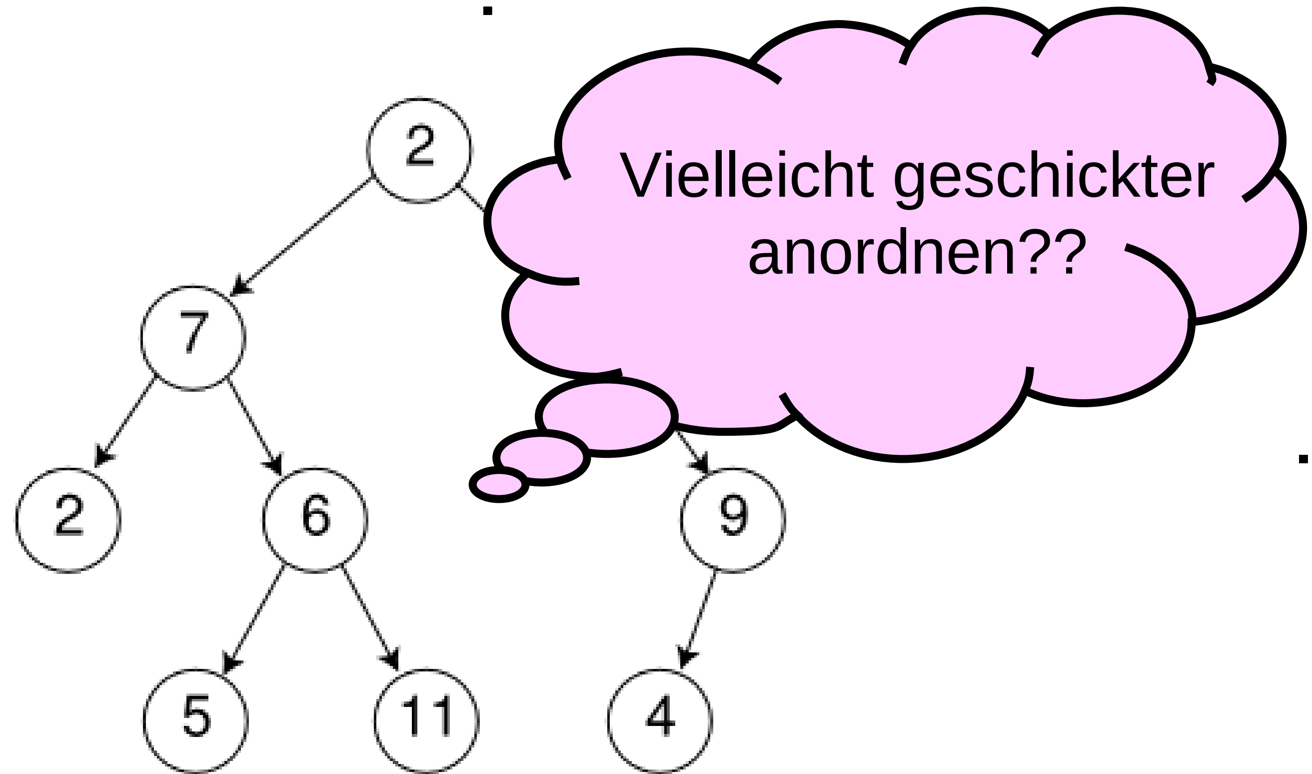
- Operationen: *Find, Insert, Delete*

Kapitel 4:

Datenstrukturen

Kapitel 4.1:

Suchbäume



Operationen: *Find, Insert, Delete*

Annahmen:

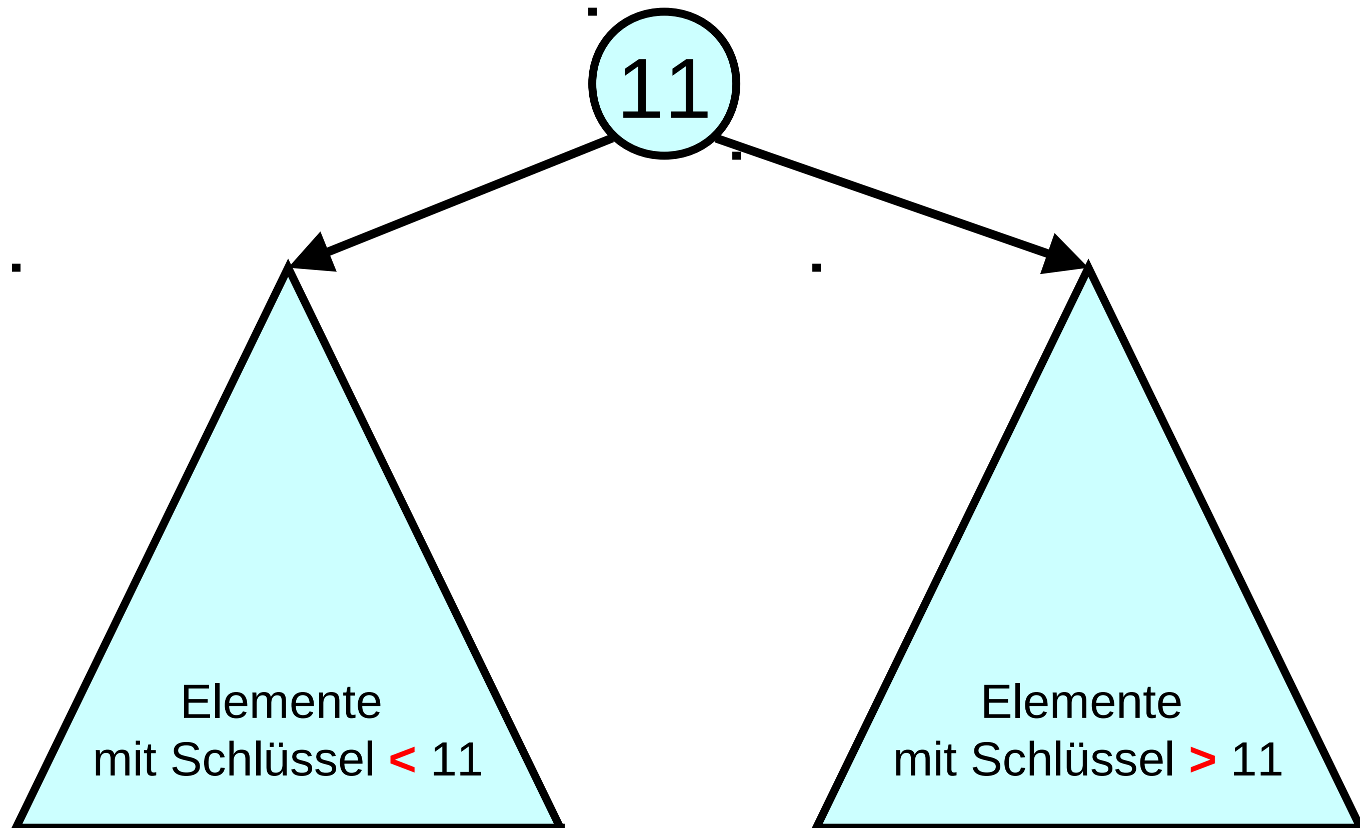
- Die Schlüssel sind aus einem total geordneten Universum

  Bsp.: Zahlen (natürliche, reelle, …),

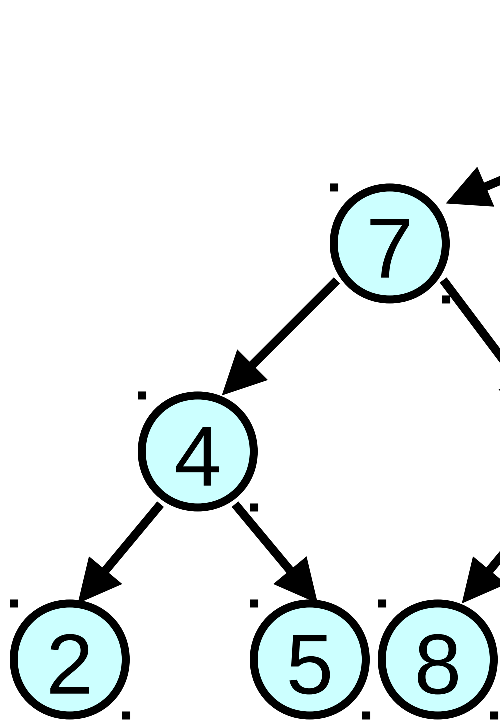
  Zeichenketten mit lexikographischer Ordnung

- Jeder Schlüssel kommt höchstens einmal vor.

  Dies ist keine unbedingt notwendige Annahme; wir treffen sie für die Vorlesung, da sie an einigen Stellen die Darstellung vereinfacht.


11
Elemente
mit Schlüssel < 11
Elemente
mit Schlüssel > 11

Balancierte binäre Suchbäume



Zum Beispiel
- AVL-Bäume
- (a,b)-Bäume,
- Rot-Schwarz-Bäume

- alle Knoten - bis auf die in den beiden untersten Ebenen - haben genau zwei Kinder
- Höhe des Baumes ist ≈ log(n) (n = Anzahl Elemente im Baum)
- Find(x) - Laufzeit O(log(n))
- Delete(x) , Insert(x):

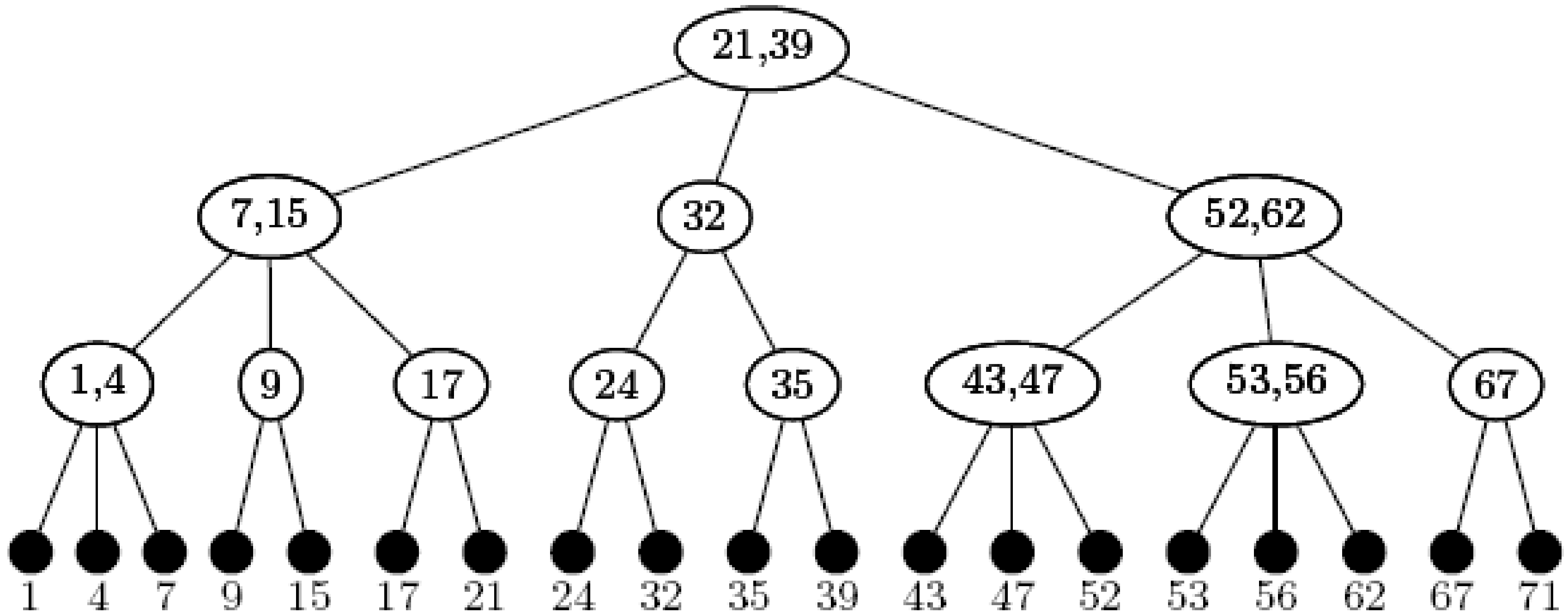
unklar … wie erhält man die Balanciertheit?

(a,b)-Baum

Definition Ein *(a,b)–Baum* ist ein Suchbaum, bei dem alle Blätter dieselbe Tiefe haben, und für den gilt:

- $a \leq \#$ Kinder $\leq b$ an allen Knoten, ausser der Wurzel.
- $2 \leq \#$ Kinder $\leq b$ an der Wurzel.
- $b \geq 2a - 1$, $a \geq 2$.
- $\rho(v) := \#$ Kinder von v.
- An jedem inneren Knoten v wird gespeichert:
  $\rho(v) - 1$ Schlüssel $K_1, \ldots, K_{\rho(v)-1}$, so dass gilt
  $K_{i-1} < \langle$alle Schlüssel im i–ten Unterbaum von $v\rangle \leq K_i$,
  hierbei sei $K_0 = -\infty$ und $K_{\rho(v)} = +\infty$

(2,3)-Baum



Innere Knoten: Verwaltungsinformation

Äussere Knoten: Schlüssel

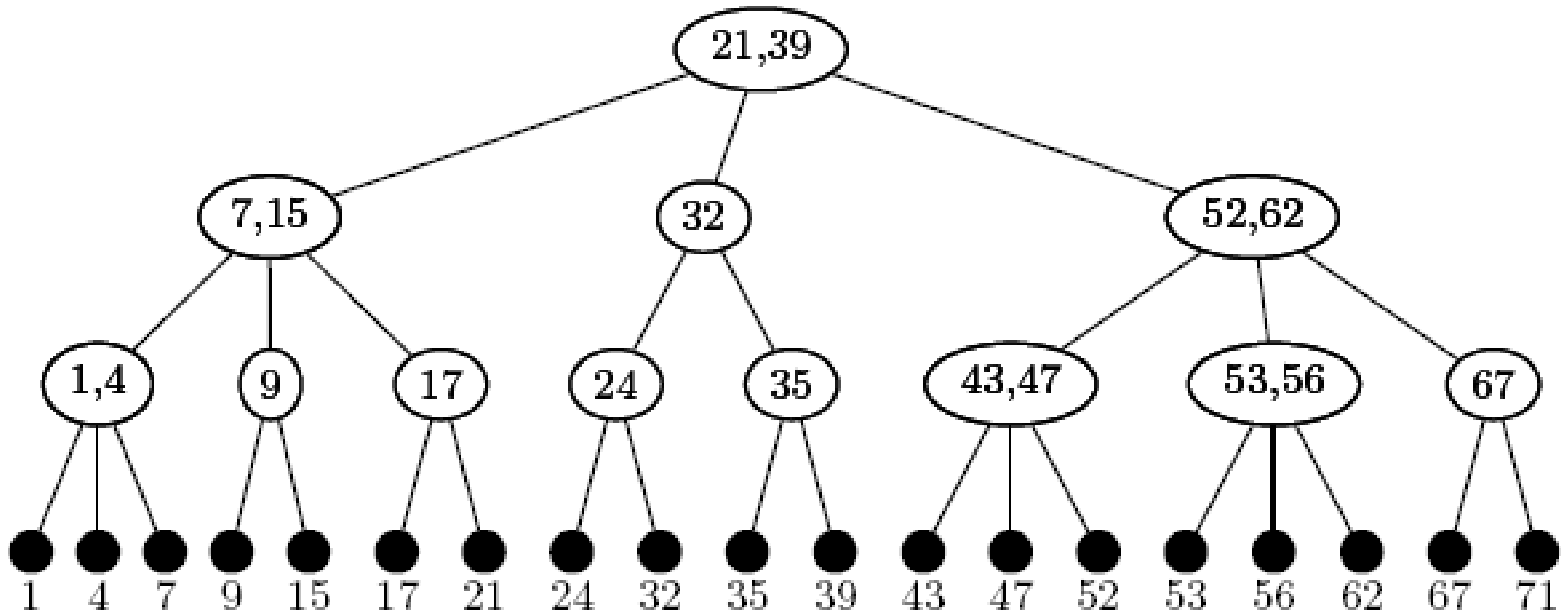
Externer Suchbaum

Lemma:
Sei T ein (a,b)–Baum mit n Blättern und Höhe h. Dann gilt:

$$(i)\ \ 2 \cdot a^{h-1} \leq n \leq b^h,$$

$$(ii)\ \ \log_b(n) \leq h \leq 1 + \log_a\left(\frac{n}{2}\right).$$

(2,3)-Baum – Find(x)

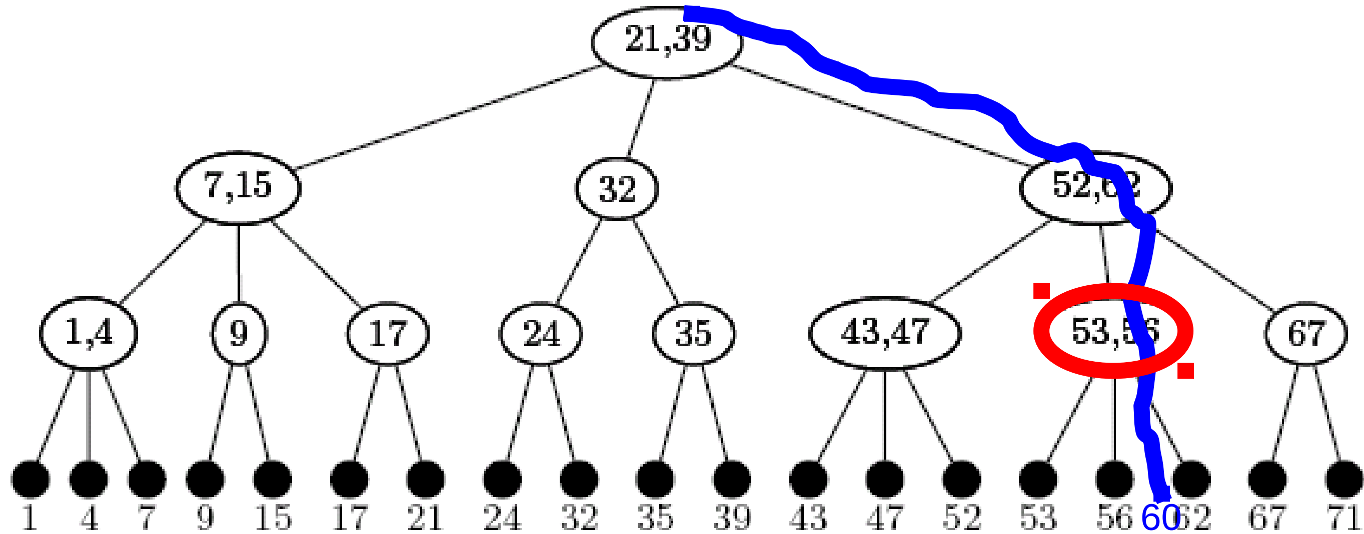


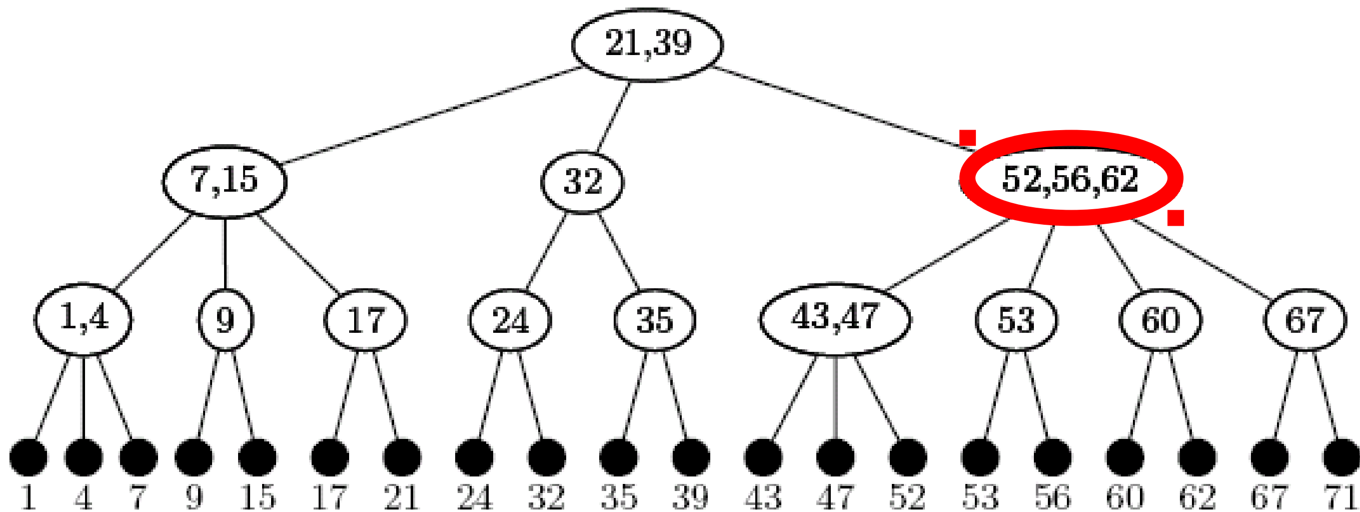
Find(k, T)**:**

$v :=$ Wurzel von T;
while (v nicht Blatt) **do**
    $i := \min\{j \mid 1 \leq j \leq \rho(v)$ und $k \leq K_j\}$;
    $v := i$-tes Kind von v
if $\texttt{key}[v] = \text{k}$
    then return v
    else return nil

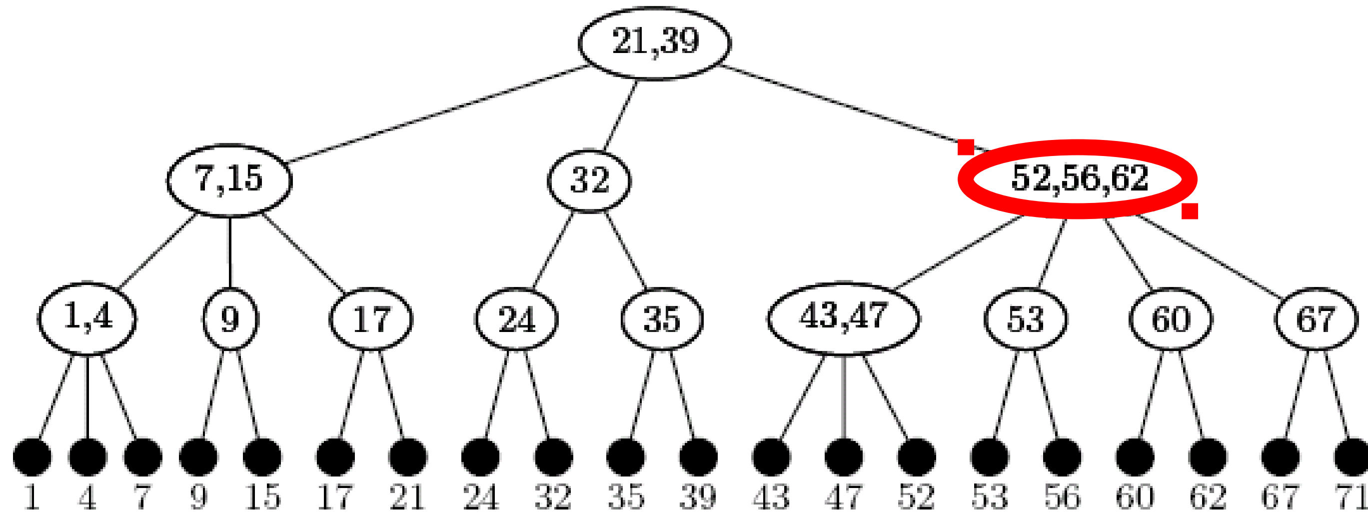
Laufzeit:
O(log(n))

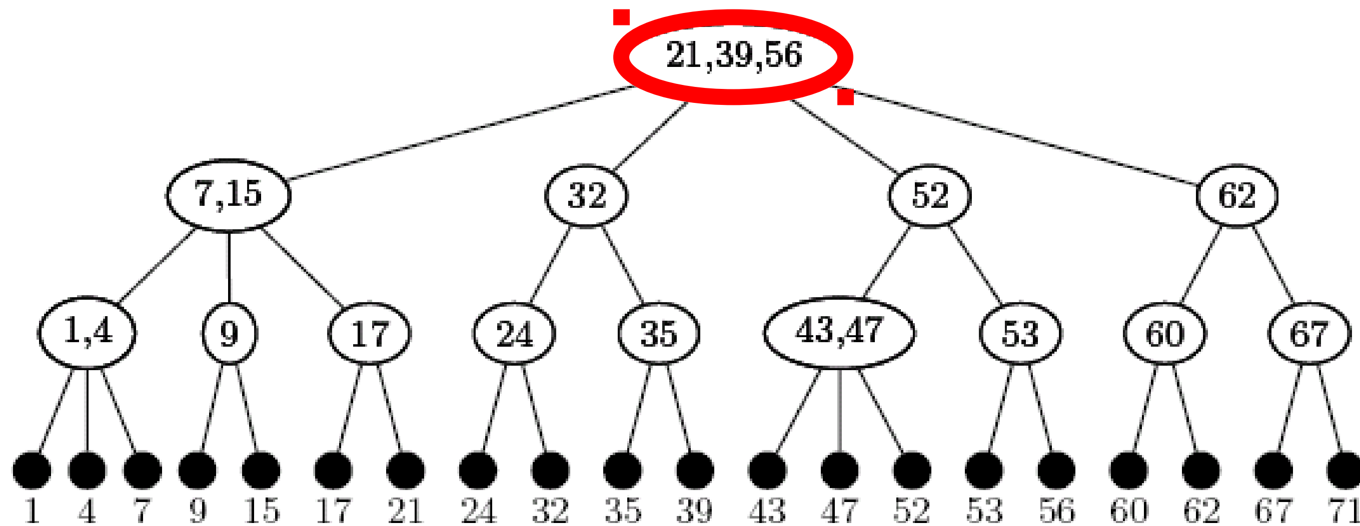
Insert(60) – 1.Schritt



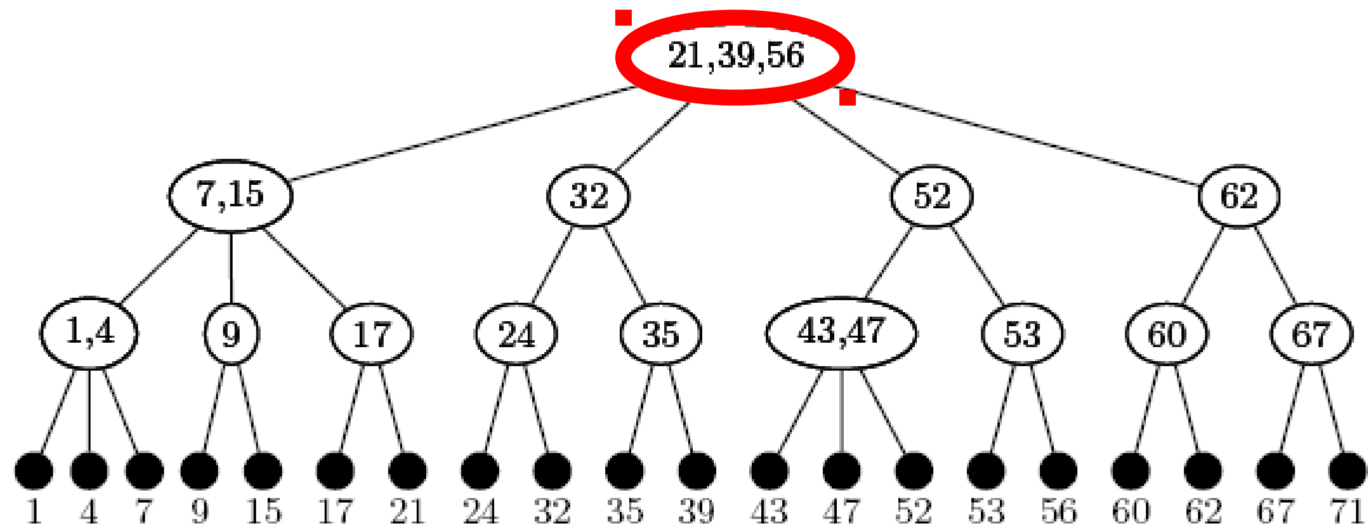


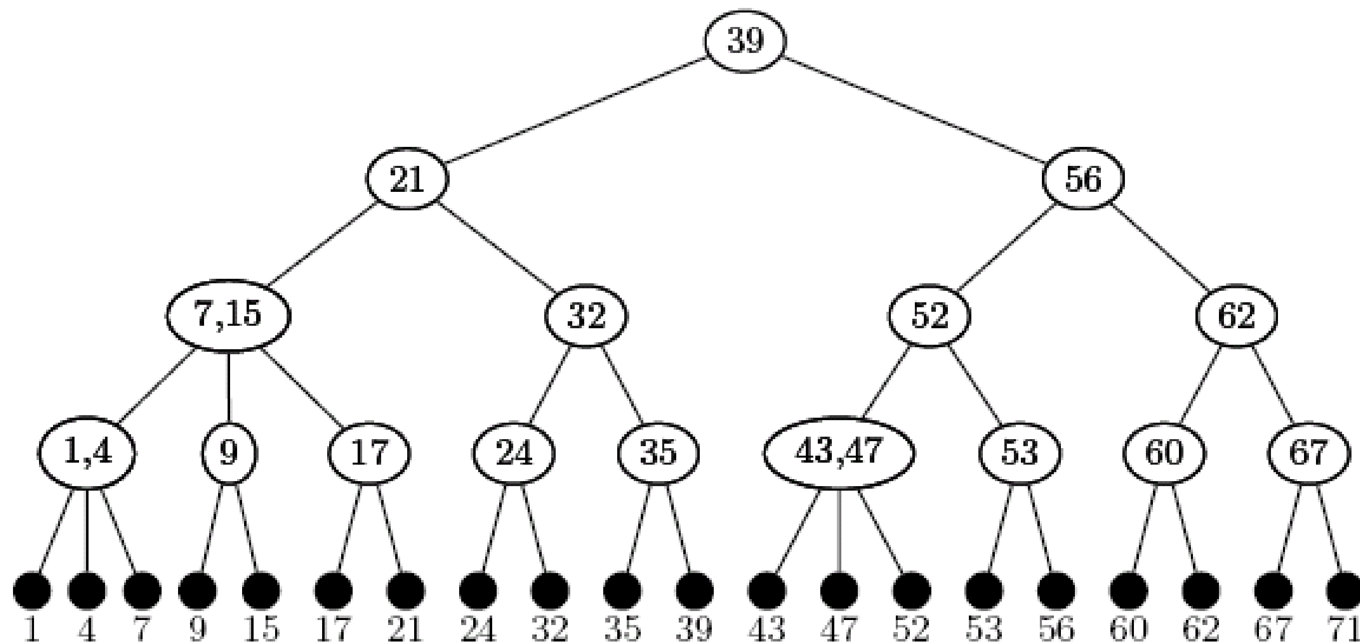
Insert(60) – 2.Schritt





Insert(60) – 3.Schritt





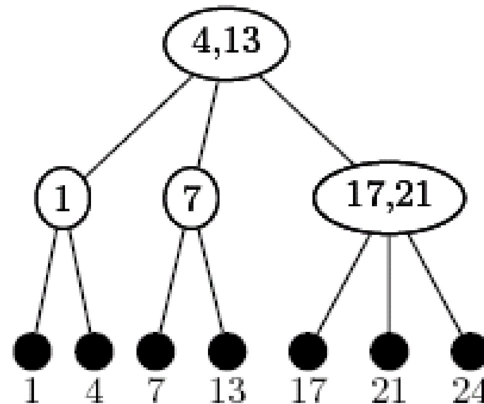
(a,b)-Baum – Insert

Insert(x, T)**:**

1. Führe $Find(x)$ aus.

Laufzeit:
O(log(n))

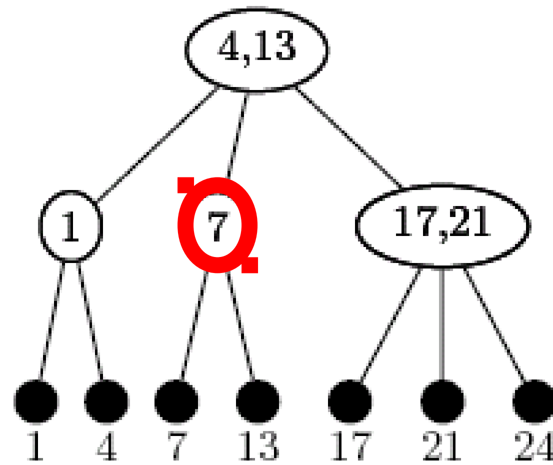
Delete(7)



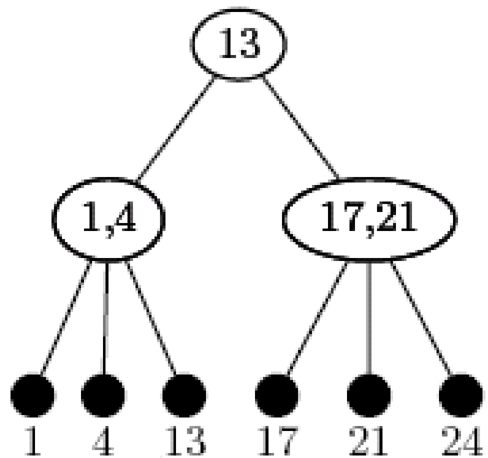
Zwei Möglichkeiten:

- „adoptiere“ Kind von rechten Nachbarn
- verschmelze Knoten mit linkem Nachbarn

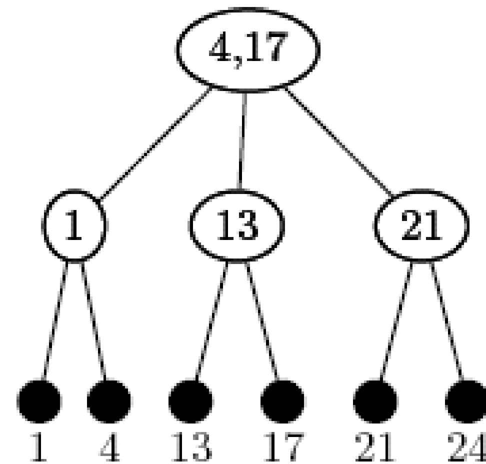
Delete(7)



1. Möglichkeit
[Verschmelze]



2. Möglichkeit
[Adoptiere]



(a,b)-Baum – Delete

Delete(x, T):

1. Führe *Find* aus; dies führt zu einem Blatt w.
2. Falls $w = x$, lösche das Blatt. Dadurch verkleinert sich der Grad $\rho(v)$ des Vaters v.
3. Wird $\rho(v) < a$, so gibt es zwei Fälle:
   (i) Falls der Nachbar von v den Grad a hat, so verschmelze v mit diesem Nachbarn.
   (ii) Falls der Nachbar von v den ... ll
   v ein Kind von seinem Nachbarn

Laufzeit:
$O(\log(n))$

Union-Find-Strukturen

Gegeben:

Datensätze partitioniert in (paarweise disjunkte) Mengen, wobei jede Menge durch einen in ihr enthaltenen Datensatz repräsentiert werde.

Gesucht:

Datenstruktur, die folgende Operationen unterstützt.

MakeNewSet(x) → Füge neuen Datensatz x ein; dieser bildet eine neue (einelementige) Menge.

Find(x) → Gebe Repräsentanten derjenigen Menge aus, die x enthält.

Union(x,y) → Vereinige die beiden Mengen, zu denen x und y gehören.