Kapitel V Algorithmen und Datenstrukturen

1. Analyse von Algorithmen

Wir wollen die Ressourcen bestimmen, die, in Abhängigkeit von der Eingabe, ein Algorithmus benötigt, z.B.

1. Laufzeit
2. Speicherplatz
3. Anzahl Prozessoren
4. Programmlänge
5. Tinte
6. ...

Beispiel 133

Prozedur für Fakultätsfunktion

func $fak(n)$
  $m := 1$
  for $i := 2$ **to** n **do**
    $m := m * i$
  do
  return (m)

Diese Prozedur benötigt $O(n)$ Schritte bzw. arithmetische Operationen.

Jedoch: die Länge der Ausgabe ist etwa

$$\lceil \log_2 n! \rceil = \Omega(n \log n) \text{ Bits.}$$

Bemerkung:

Um die Zahl $n \in \mathbb{N}_0$ in Binärdarstellung hinzuschreiben, benötigt man

$$\ell(n) := \begin{cases} 1 & \text{für } n = 0 \\ 1 + \lfloor \log_2(n) \rfloor & \text{sonst} \end{cases}$$
$$= \begin{cases} 1 & \text{für } n = 0 \\ \lceil \log_2(n+1) \rceil & \text{sonst} \end{cases}$$

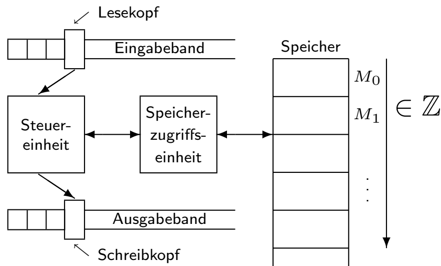
Um die Notation zu vereinfachen, vereinbaren wir im Zusammenhang mit Komplexitätsabschätzungen

$$\log(0) := 0\,.$$

1.1 Referenzmaschine

Wir wählen als Referenzmaschine die Registermaschine (engl. random access machine, RAM) oder auch WHILE-Maschine, also eine Maschine, die WHILE-Programme verarbeiten kann, erweitert durch

- IF . . . THEN . . . ELSE . . . FI
- Multiplikation und Division
- indirekte Adressierung
- arithmetische Operationen wie $\sqrt{n}$, $\sin n$,. . .



Registermaschine

1.2 Zeit- und Platzkomplexität

Beim Zeitbedarf zählt das uniforme Kostenmodell die Anzahl der von der Registermaschine durchgeführten Elementarschritte, beim Platzbedarf die Anzahl der benutzten Speicherzellen.

Das logarithmische Kostenmodell zählt für den Zeitbedarf eines jeden Elementarschrittes

$$\ell(\text{größter beteiligter Operand}),$$

beim Platzbedarf

$$\sum_x \ell(\text{größter in } x \text{ gespeicherter Wert})\,,$$

also die maximale Anzahl der von allen Variablen x benötigten Speicherbits.

Beispiel 134

Wir betrachten die Prozedur

func $dbexp(n)$
  $m := 2$
  for $i := 1$ **to** n **do**
    $m := m^2$
  do
  return (m)

Die Komplexität von $dbexp$, die $n \mapsto 2^{2^n}$ berechnet, ergibt sich bei Eingabe n zu

	Zeit	Platz
uniform	$\Theta(n)$	$\Theta(1)$
logarithmisch	$\Theta(2^n)$	$\Theta(2^n)$

Bemerkung:
Das (einfachere) uniforme Kostenmodell sollte also nur verwendet werden, wenn alle vom Algorithmus berechneten Werte gegenüber den Werten in der Eingabe nicht zu sehr wachsen, also z.B. nur polynomiell.

1.3 Worst Case-Analyse

Sei A ein Algorithmus. Dann sei

$$T_A(x) := \text{Laufzeit von } A \text{ bei Eingabe } x\,.$$

Diese Funktion ist i.A. zu aufwändig und zu detailliert. Stattdessen:

$$T_A(n) := \max_{|x|=n} T_A(x) \quad (= \text{maximale Laufzeit bei Eingabelänge } n)$$

1.4 Average Case-Analyse

Oft erscheint die Worst Case-Analyse als zu pessimistisch. Dann:

$$T_A^{\mathsf{avg}}(n) = \frac{\sum_{x;\ |x|=n} T_A(x)}{|\{x;\ |x| = n\}|}$$

oder allgemeiner

$$\begin{aligned} T_A^{\mathsf{avg}}(n) &= \sum T_A(x) \cdot \Pr\{x \mid |x| = n\} \\ &= \mathbf{E}_{|x|=n}\left[T_A(x)\right] \ , \end{aligned}$$

wobei eine (im Allgemeinen beliebige) Wahrscheinlichkeitsverteilung zugrunde liegt.

Bemerkung:

Wir werden Laufzeiten $T_A(n)$ meist nur bis auf einen multiplikativen Faktor genau berechnen, d.h. das genaue Referenzmodell, Fragen der Implementierung, usw. spielen dabei eine eher untergeordnete Rolle.

2. Sortierverfahren

Unter einem Sortierverfahren versteht man ein algorithmisches Verfahren, das als Eingabe eine Folge $a_1, \ldots, a_n$ von n Schlüsseln $\in \Sigma^*$ erhält und als Ausgabe eine auf- oder absteigend sortierte Folge dieser Elemente liefert. Im Folgenden werden wir im Normalfall davon ausgehen, dass die Elemente aufsteigend sortiert werden sollen. Zur Vereinfachung nehmen wir im Normalfall auch an, dass alle Schlüssel paarweise verschieden sind.

Für die betrachteten Sortierverfahren ist natürlich die Anzahl der Schlüsselvergleiche eine untere Schranke für die Laufzeit, und oft ist letztere von der gleichen Größenordnung, d.h.

$$\text{Laufzeit} = O(\text{Anzahl der Schlüsselvergleiche})\,.$$

2.1 Selection-Sort

Sei eine Folge $a_1, \ldots, a_n$ von Schlüsseln gegeben, indem a_i im Element i eines Feldes $A[1..n]$ abgespeichert ist.

Algorithmus $\textsc{SelectionSort}$

for $i := n$ **downto** 2 **do**
  $m :=$ Index des maximalen Schlüssels in $A[1..i]$
  vertausche $A[i]$ und $A[m]$
od

Nach Beendigung von $\textsc{SelectionSort}$ ist das Feld A aufsteigend sortiert.

Satz 135

SELECTIONSORT *benötigt zum Sortieren von n Elementen genau $\binom{n}{2}$ Vergleiche.*

$\binom{n}{i} = \frac{n!}{i!(n-i)!}$ Binomial-koeffizient

Beweis:

Die Anzahl der Vergleiche (zwischen Schlüsseln bzw. Elementen des Feldes A) zur Bestimmung des maximalen Schlüssels in $A[1..i]$ ist $i-1$.

Damit ergibt sich die Laufzeit von SELECTIONSORT zu

$$T_{\textsc{SelectionSort}} = \sum_{i=2}^{n}(i-1) = \sum_{i=1}^{n-1} i = \frac{n(n-1)}{2} = \binom{n}{2}.$$

2.2 Insertion-Sort

Sei eine Folge $a_1, \ldots, a_n$ von Schlüsseln gegeben, indem a_i im Element i eines Feldes $A[1..n]$ abgespeichert ist.

Algorithmus InsertionSort

for $i := 2$ **to** n **do**
  $m :=$ Rang ($\in [1..i]$) von $A[i]$ in $\{A[1], \ldots, A[i-1]\}$
  $a := A[i]$
  schiebe $A[m..i-1]$ um eine Position nach rechts
  $A[m] := a$
od

Nach Beendigung von InsertionSort ist das Feld A aufsteigend sortiert.

Der Rang von $A[i]$ in $\{A[1], \ldots, A[i-1]\}$ kann trivial mit $i-1$ Vergleichen bestimmt werden. Damit ergibt sich

Satz 136

InsertionSort *benötigt zum Sortieren von* n *Elementen maximal* $\binom{n}{2}$ *Vergleiche.*

Beweis:

Übungsaufgabe!

Die Rangbestimmung kann durch binäre Suche verbessert werden. Dabei benötigen wir, um den Rang eines Elementes in einer k-elementigen Menge zu bestimmen, höchstens

$$\lceil \log_2(k+1) \rceil$$

Vergleiche, wie man durch Induktion leicht sieht.

Satz 137

InsertionSort *mit binärer Suche für das Einsortieren benötigt zum Sortieren von* n *Elementen maximal*

$$n \lceil \operatorname{ld} n \rceil$$

Vergleiche.

Beweis:

Die Abschätzung ergibt sich durch einfaches Einsetzen.

Achtung: Die Laufzeit von InsertionSort ist dennoch auch bei Verwendung von binärer Suche beim Einsortieren im schlechtesten Fall $\Omega(n^2)$, wegen der notwendigen Verschiebung der Feldelemente.

Verwendet man statt des Feldes eine doppelt verkettete Liste, so wird zwar das Einsortieren vereinfacht, es kann jedoch die binäre Suche nicht mehr effizient implementiert werden.

2.3 Merge-Sort

Sei wiederum eine Folge $a_1, \ldots, a_n$ von Schlüsseln im Feld $A[1..n]$ abgespeichert.

Algorithmus $\textsc{MergeSort}$

proc merge (r, s) **co** sortiere $A[r..s]$**oc**
  if $s \leq r$ **return fi**
  merge($r, \left\lfloor \frac{r+s}{2} \right\rfloor$); merge($\left\lfloor \frac{r+s}{2} \right\rfloor + 1, s$)
  verschmelze die beiden sortierten Teilfolgen
end

merge$(1, n)$

Das Verschmelzen sortierter Teilfolgen $A[r..m]$ und $A[m+1,s]$ funktioniert wie folgt unter Benutzung eines Hilfsfeldes $B[r,s]$:

$i := r$; $j := m+1$; $k := r$
while $i \leq m$ **and** $j \leq s$ **do**
  if $A[i] < A[j]$ **then** $B[k] := A[i]$; $i := i+1$
  else $B[k] := A[j]$; $j := j+1$ **fi**
  $k := k+1$
od
if $i \leq m$ **then** kopiere $A[i,m]$ nach $B[k,s]$
else kopiere $A[j,s]$ nach $B[k,s]$ **fi**
kopiere $B[r,s]$ nach $A[r,s]$ zurück

Nach Beendigung von MergeSort ist das Feld A aufsteigend sortiert.

Satz 138

$\textsc{MergeSort}$ *sortiert ein Feld der Länge* n *mit maximal* $n \cdot \lceil \mathrm{ld}(n) \rceil$ *Vergleichen.*

Beweis:

In jeder Rekursionstiefe werde der Vergleich dem kleineren Element zugeschlagen. Dann erhält jedes Element pro Rekursionstiefe höchstens einen Vergleich zugeschlagen.

2.4 Quick-Sort

Beim Quick-Sort-Verfahren wird in jeder Phase ein Element p der zu sortierenden Folge als Pivot-Element ausgewählt (wie dies geschehen kann, wird noch diskutiert). Dann wird in situ und mit einer linearen Anzahl von Vergleichen die zu sortierende Folge so umgeordnet, dass zuerst alle Elemente $< p$, dann p selbst und schließlich alle Elemente $> p$ kommen. Die beiden Teilfolgen links und rechts von p werden dann mit Quick-Sort rekursiv sortiert (Quick-Sort ist also ein Divide-and-Conquer-Verfahren).

Quick-Sort benötigt im schlechtesten Fall, nämlich wenn als Pivot-Element stets das kleinste oder größte der verbleibenden Elemente ausgewählt wird,

$$\sum_{i=1}^{n-1}(n-i) = \binom{n}{2}$$

Vergleiche.

Satz 139

QuickSort *benötigt zum Sortieren eines Feldes der Länge* n *durchschnittlich nur*

$$2\ln(2) \cdot n\,\mathrm{ld}(n) + O(n)$$

viele Vergleiche.

Beweis:

Siehe Vorlesung Diskrete Wahrscheinlichkeitstheorie (DWT).

Entscheidend für die Laufzeit von Quick-Sort ist eine „gute“ Wahl des Pivotelements. U.a. werden folgende Varianten verwendet:

1. Nimm stets das letzte Element der (Teil-)Folge als Pivotelement
   Nachteil: sehr schlecht bei vorsortierten Arrays!
2. Median-of-3 Verfahren: Wähle den Median (das mittlere Element) des ersten, mittleren und letzten Elements des Arrays
   Analyse Übungsaufgabe!
3. Wähle ein zufälliges Element als Pivotelement
   liefert die o.a. durchschnittliche Laufzeit, benötigt aber einen Zufallsgenerator.

2.5 Heap-Sort

Definition 140

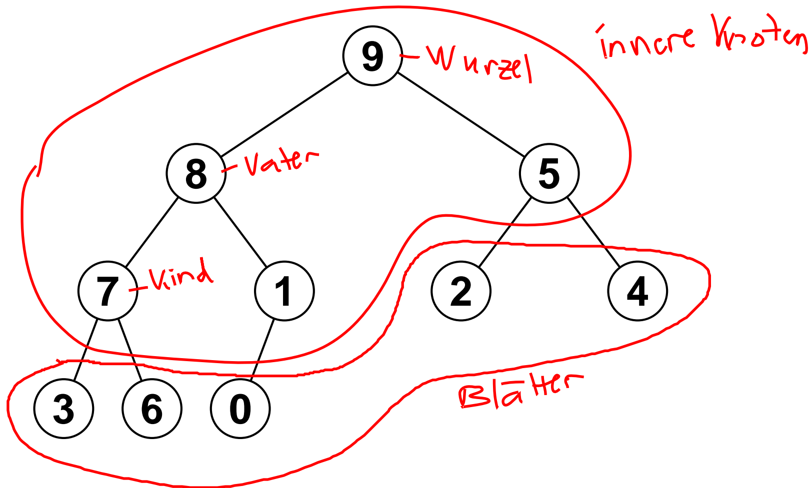
Ein Heap ist ein Binärbaum, an dessen Knoten Schlüssel gespeichert sind, so dass gilt:

1. alle inneren Knoten bis auf maximal einen haben genau zwei Kinder;
2. alle Knoten mit weniger als zwei Kindern (also insbesondere die Blätter) befinden sich auf der untersten oder der zweituntersten Schicht;
3. die unterste Schicht ist von links nach rechts aufgefüllt;
4. für jeden Knoten (mit Ausnahme der Wurzel) gilt, dass sein Schlüssel kleiner ist als der seines Vaters (Heap-Bedingung).

Bemerkungen:

1. Die hier definierte Variante ist ein max-Heap.
2. Die Bezeichnung heap wird in der Algorithmentheorie auch allgemeiner für Prioritätswarteschlangen benutzt!

Beispiel 141



Der Algorithmus HEAPSORT besteht aus zwei Phasen.

1. In der ersten Phase wird aus der unsortierten Folge von n Elementen ein Heap gemäß Definition aufgebaut.
2. In der zweiten Phase wird dieser Heap ausgegeben, d.h. ihm wird n-mal jeweils das größte Element entnommen (das ja an der Wurzel steht), dieses Element wird in die zu sortierende Folge aufgenommen und die Heap-Eigenschaften wird wieder hergestellt.

Betrachten wir nun zunächst den Algorithmus Reheap zur Korrektur der Datenstruktur, falls die Heap-Bedingung höchstens an der Wurzel verletzt ist.

Algorithmus Reheap

sei v die Wurzel des Heaps;
while Heap-Eigenschaft in v nicht erfüllt **do**
  sei v' das Kind von v mit dem größeren Schlüssel
  vertausche die Schlüssel in v und v'
  $v := v'$
od

Zweite Phase von HEAPSORT

for $i := n$ **downto** 1 do
  sei r die Wurzel des Heaps
  sei k der in r gespeicherte Schlüssel
  $A[i] := k$
  sei b das rechteste Blatt in der untersten Schicht des Heaps
  kopiere den Schlüssel von b in die Wurzel r
  entferne das Blatt b
  Reheap
od

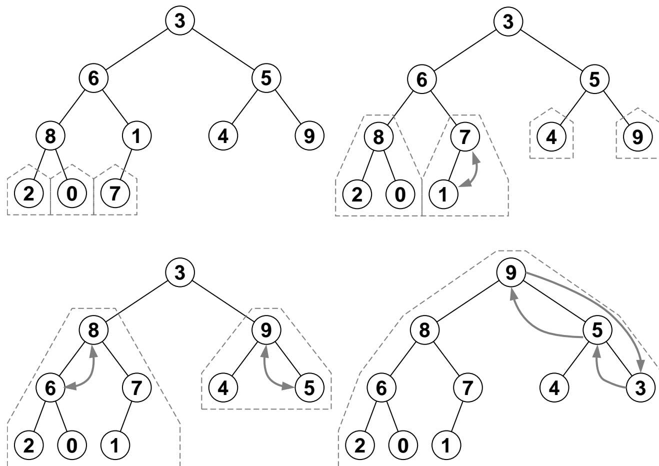
Nach Beendigung der zweiten Phase von HEAPSORT ist das Feld $A[1..n]$ aufsteigend sortiert.

Mit geeigneten (kleinen) Modifikationen kann HEAPSORT in situ implementiert werden. Die Schichten des Heaps werden dabei von oben nach unten und von links nach rechts im Feld A abgespeichert.

In der ersten Phase von HEAPSORT müssen wir mit den gegebenen n Schlüsseln einen Heap erzeugen.

Wir tun dies iterativ, indem wir aus zwei bereits erzeugten Heaps und einem weiteren Schlüssel einen neuen Heap formen, indem wir einen neuen Knoten erzeugen, der die Wurzel des neuen Heaps wird. Diesem neuen Knoten ordnen wir zunächst den zusätzlichen Schlüssel zu und und machen die beiden alten Heaps zu seinen Unterbäumen. Damit ist die Heap-Bedingung höchstens an der Wurzel des neuen Baums verletzt, was wir durch Ausführung der Reheap-Operation korrigieren können.

Beispiel 142



Lemma 143

Die Reheap-Operation erfordert höchstens $O($*Tiefe des Heaps*$)$ *Schritte.*

Beweis:

Reheap führt pro Schicht des Heaps nur konstant viele Schritte aus.

Lemma 144

Die Initialisierung des Heaps in der ersten Phase von HEAPSORT *benötigt nur $O(n)$ Schritte.*

Beweis:

Sei d die Tiefe (Anzahl der Schichten) des (n-elementigen) Heaps. Die Anzahl der Knoten in Tiefe i ist $\leq 2^i$ (die Wurzel habe Tiefe 0). Wenn ein solcher Knoten beim inkrementellen Aufbau des Heaps als Wurzel hinzugefügt wird, erfordert die Reheap-Operation $\leq d - i$ Schritte, insgesamt werden also

$$\leq \sum_{i=0}^{d-1} (d-i) 2^i = O(n)$$

log n

$2^{\log n} = n$

Schritte benötigt.

Satz 145

HEAPSORT *benötigt maximal* $O(n \log n)$ *Schritte.*

Beweis:

In der zweiten Phase werden $< n$ Reheap-Operationen auf Heaps der Tiefe $\leq \log n$ durchgeführt.

Bemerkung:

1. Eine genauere Analyse ergibt eine Schranke von $2n\,\mathrm{ld}(n) + o(n)$.
2. Carlsson hat eine Variante von HEAPSORT beschrieben, die mit $n\,\mathrm{ld}\,n + O(n \log\log n)$ Vergleichen auskommt:

Svante Carlsson:
A variant of heapsort with almost optimal number of comparisons.
Inf. Process. Lett., **24**(4):247–250, 1987

2.6 Vergleichsbasierte Sortierverfahren

Alle bisher betrachteten Sortierverfahren sind vergleichsbasiert, d.h. sie greifen auf Schlüssel k, k' (außer in Zuweisungen) nur in Vergleichsoperationen der Form $k < k'$ zu, verwenden aber nicht, dass etwa $k \in \mathbb{N}_0$ oder dass $k \in \Sigma^*$.

Satz 146

nur dann!

Jedes vergleichsbasierte Sortierverfahren benötigt im worst-case mindestens

$$n \operatorname{ld} n + O(n)$$

Vergleiche und hat damit Laufzeit $\Omega(n \log n)$.

mindestens

Beweis:

Wir benutzen ein so genanntes Gegenspielerargument (engl. adversary argument). Soll der Algorithmus n Schlüssel sortieren, legt der Gegenspieler den Wert eines jeden Schlüssels immer erst dann fest, wenn der Algorithmus das erste Mal auf ihn in einem Vergleich zugreift. Er bestimmt den Wert des Schlüssels so, dass der Algorithmus möglichst viele Vergleiche durchführen muss.

Am Anfang (vor der ersten Vergleichsoperation des Algorithmus) sind alle $n!$ Sortierungen der Schlüssel möglich, da der Gegenspieler jedem Schlüssel noch einen beliebigen Wert zuweisen kann.

Beweis:

Seien nun induktiv vor einer Vergleichsoperation $A[i] < A[j]$ des Algorithmus noch r Sortierungen der Schlüssel möglich.

Falls der Gegenspieler die Werte der in $A[i]$ bzw. $A[j]$ gespeicherten Schlüssel bereits früher festgelegt hat, ändert sich die Anzahl der möglichen Sortierungen durch den Vergleich nicht, dieser ist redundant.

Beweis:

Andernfalls kann der Gegenspieler einen oder beide Schlüssel so festlegen, dass immer noch mindestens $r/2$ Sortierungen möglich sind (wir verwenden hier, dass die Schlüssel stets paarweise verschieden sind).

Nach k Vergleichen des Algorithmus sind also immer noch $n!/2^k$ Sortierungen möglich. Der Algorithmus muss jedoch Vergleiche ausführen, bis nur noch eine Sortierung möglich ist (die dann die Ausgabe des Sortieralgorithmus darstellt).

Damit

$$\#\text{Vergleiche} \geq \lceil \operatorname{ld}(n!) \rceil = n \operatorname{ld} n + O(n)$$

mit Hilfe der Stirlingschen Approximation für $n!$. □

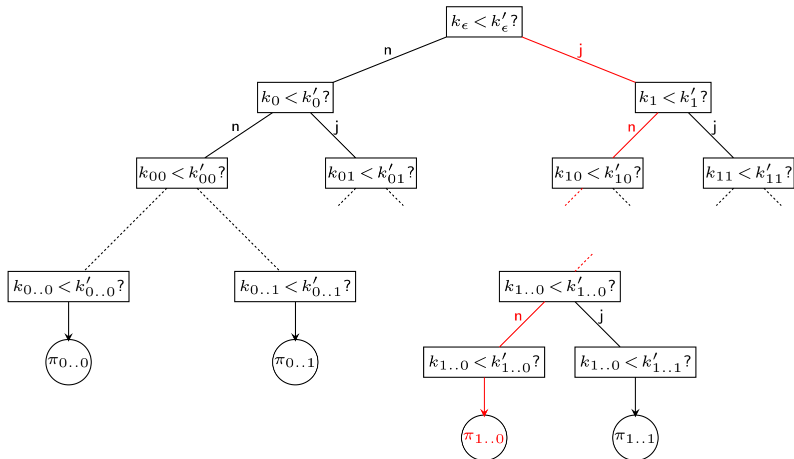
Alternativer Beweis mit Hilfe des Entscheidungsbaums

Ein vergleichsbasierter Algorithmus kann auf die Werte der Schlüssel nur durch Vergleiche $k < k'$ zugreifen. Wenn wir annehmen, dass alle Schlüssel paarweise verschieden sind, ergeben zu jedem Zeitpunkt die vom Algorithmus erhaltenen binären Antworten auf seine Anfragen der obigen Art an die Schlüsselmenge seine gesamte Information über die (tatsächliche) Ordnung der Schlüssel.

Am Ende eines jeden Ablaufs des Algorithmus muss diese Information so sein, dass die tatsächliche Ordnung der Schlüssel eindeutig festliegt, dass also nur eine Permutation der n zu sortierenden Schlüssel mit der erhaltenen Folge von Binärantworten konsistent ist.

Wir stellen alle möglichen Abläufe (d.h., Folgen von Vergleichen), die sich bei der Eingabe von n Schlüsseln ergeben können, in einem so genannten Entscheidungsbaum dar.

Entscheidungsbaum:



Damit muss es für jede der $n!$ Permutationen mindestens ein Blatt in diesem Entscheidungsbaum geben. Da dieser ein Binärbaum ist, folgt daraus:

- Die Tiefe des Entscheidungsbaums (und damit die Anzahl der vom Sortieralgorithmus benötigten Vergleiche im worst-case) ist

$$\geq \lceil \mathrm{ld}(n!) \rceil = n \, \mathrm{ld} \, n + O(n) \, .$$

- Diese untere Schranke gilt sogar im Durchschnitt (über alle Permutationen).

$\approx \log \left(\frac{n}{e}\right)^n \approx n \log n$

2.7 Bucket-Sort

Bucket-Sort ist ein nicht-vergleichsbasiertes Sortierverfahren. Hier können z.B. n Schlüssel aus

$$\{0, 1, \ldots, B-1\}^d$$

in Zeit $O(d(n+B))$ sortiert werden, indem sie zuerst gemäß dem letzten Zeichen auf B Behälter verteilt werden, die so entstandene Teilsortierung dann stabil gemäß dem vorletzten Zeichen auf B Behälter verteilt wird, usw. bis zur ersten Position. Das letzte Zeichen eines jeden Schlüssels wird also als das niedrigwertigste aufgefasst.

Bucket-Sort sortiert damit n Schlüssel aus $\{0, 1, \ldots, B-1\}^d$ in Zeit $O(nd)$, also linear in der Länge der Eingabe (gemessen als Anzahl der Zeichen).

3. Suchverfahren

Es ist eine Menge von Datensätzen gegeben, wobei jeder Datensatz D_i durch einen eindeutigen Schlüssel k_i gekennzeichnet ist. Der Zugriff zu den Datensätzen erfolgt per Zeiger über die zugeordneten Schlüssel, so dass wir uns nur mit der Verwaltung der Schlüssel beschäftigen.

I. A. ist die Menge der Datensätze dynamisch, d.h. es können Datensätze neu hinzukommen oder gelöscht werden, oder Datensätze bzw. Schlüssel können geändert werden.

Definition 147

Ein Wörterbuch (engl. dictionary) ist eine Datenstruktur, die folgende Operationen auf einer Menge von Schlüsseln effizient unterstützt:

1. is_member(k): teste, ob der Schlüssel k in der Schlüsselmenge enthalten ist;
2. insert(k): füge den Schlüssel k zur Schlüsselmenge hinzu, falls er noch nicht vorhanden ist;
3. delete(k): entferne den Schlüssel k aus der Schlüsselmenge, falls er dort vorhanden ist.

Es gibt zwei grundsätzlich verschiedene Ansätze, um Wörterbücher zu implementieren:

- Suchbäume
- Hashing (Streuspeicherverfahren)

Wir betrachten zuerst Suchbäume. Wir nehmen an, dass die Schlüssel aus einer total geordneten Menge, dem Universum $\mathcal{U}$ stammen.

Sind die Schlüssel nur in den Blättern des Suchbaums gespeichert, sprechen wir von einem externen Suchbaum, ansonsten von einem internen Suchbaum (wo dann der Einfachheit halber Schlüssel *nur* in den internen Knoten gespeichert werden).

3.1 Binäre/natürliche Suchbäume

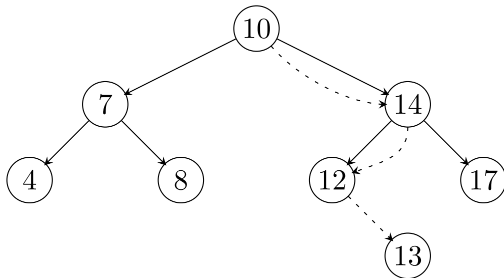
In binären Suchbäumen gilt für alle Knoten x

- $key(x)$ ist größer als der größte Schlüssel im linken Unterbaum von x;
- $key(x)$ ist kleiner als der kleinste Schlüssel im rechten Unterbaum von x.

Die Wörterbuch-Operationen werden wie folgt realisiert:

1. is_member(k): beginnend an der Wurzel des Suchbaums, wird der gesuchte Schlüssel k mit dem am Knoten gespeicherten Schlüssel k' verglichen. Falls $k < k'$ ($k > k'$), wird im linken (rechten) Unterbaum fortgefahren (falls der Unterbaum leer ist, ist der Schlüssel nicht vorhanden), ansonsten ist der Schlüssel gefunden.
2. insert(k): es wird zuerst geprüft, ob k bereits im Suchbaum gespeichert ist; falls ja, ist die Operation beendet, falls nein, liefert die Suche die Position (den leeren Unterbaum), wo k hinzugefügt wird.
3. delete(k): es wird zuerst geprüft, ob k im Suchbaum gespeichert ist; falls nein, ist die Operation beendet, falls ja, sei x der Knoten, in dem k gespeichert ist, und es sei x' der linkste Knoten im rechten Unterbaum von x (x' ist nicht unbedingt ein Blatt!); dann wird $key(x')$ im Knoten x gespeichert und x' durch seinen rechten Unterbaum ersetzt (falls vorhanden) bzw. gelöscht. Spezialfälle, wie z.B., dass x' nicht existiert, sind kanonisch zu behandeln.

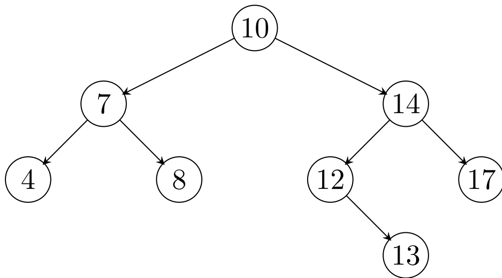
Beispiel 148



Der Schlüssel 13 wird hinzugefügt.

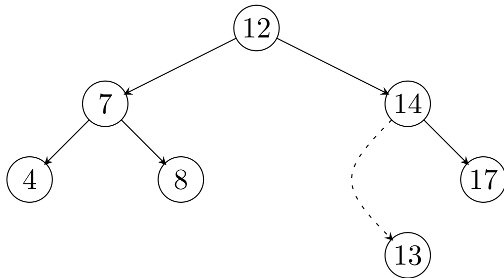
Beispiel

Der Schlüssel 10 (also die Wurzel) wird gelöscht:



Beispiel

Der Schlüssel 10 (also die Wurzel) wird gelöscht:



Satz 149

In einem natürlichen Suchbaum der Höhe h benötigen die Wörterbuch-Operationen jeweils Zeit $O(h)$.

Beweis:

Folgt aus der obigen Konstruktion!

Bemerkung:

Das Problem bei natürlichen Suchbäumen ist, dass sie sehr entartet sein können, z.B. bei n Schlüsseln eine Tiefe von $n-1$ haben. Dies ergibt sich z.B., wenn die Schlüssel in aufsteigender Reihenfolge eingefügt werden.

3.2 AVL-Bäume

AVL-Bäume sind interne binäre Suchbäume, bei denen für jeden Knoten gilt, dass sich die Höhe seiner beiden Unterbäume um höchstens 1 unterscheidet. AVL-Bäume sind nach ihren Erfindern G. Adelson-Velskii und Y. Landis (1962) benannt.

Satz 150

Ein AVL-Baum der Höhe h enthält mindestens $F_{h+3} - 1$ und höchstens $2^{h+1} - 1$ Knoten, wobei F_n die n-te Fibonacci-Zahl ($F_0 = 0, F_1 = 1$) und die Höhe die maximale Anzahl von Kanten auf einem Pfad von der Wurzel zu einem Blatt ist.

Beweis:

Die obere Schranke ist klar, da ein Binärbaum der Höhe h höchstens

$$\sum_{j=0}^{h} 2^j = 2^{h+1} - 1$$

Knoten enthalten kann.

Beweis:

Induktionsanfang:

1. ein AVL-Baum der Höhe $h = 0$ enthält mindestens einen Knoten, $1 \geq F_3 - 1 = 2 - 1 = 1$
2. ein AVL-Baum der Höhe $h = 1$ enthält mindestens zwei Knoten, $2 \geq F_4 - 1 = 3 - 1 = 2$

Beweis:

Induktionsschluss: Ein AVL-Baum der Höhe $h \geq 2$ mit minimaler Knotenzahl hat als Unterbäume der Wurzel einen AVL-Baum der Höhe $h-1$ und einen der Höhe $h-2$, jeweils mit minimaler Knotenzahl. Sei

$$f_h := 1 + \text{minimale Knotenanzahl eines AVL-Baums der Höhe } h\,.$$

Dann gilt demgemäß

$$\begin{aligned} f_0 &= 2 & &= F_3 \\ f_1 &= 3 & &= F_4 \\ f_h - 1 &= 1 + f_{h-1} - 1 + f_{h-2} - 1\,, & &\text{also} \\ f_h &= f_{h-1} + f_{h-2} & &= F_{h+3} \end{aligned}$$

Korollar 151

Die Höhe eines AVL-Baums mit n Knoten ist $\Theta(\log n)$.

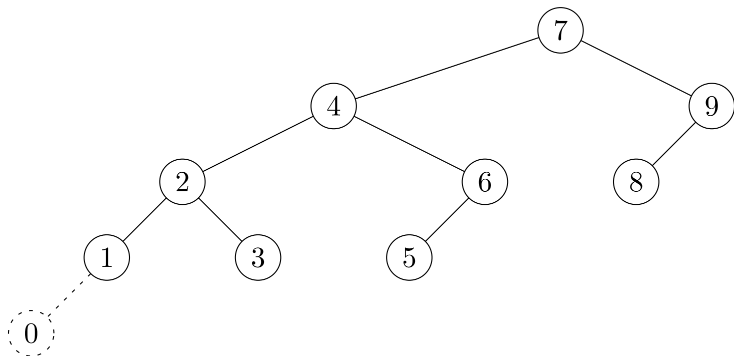
Satz 152

In einem AVL-Baum mit n Schlüsseln kann in Zeit $O(\log n)$ festgestellt werden, ob sich ein gegebener Schlüssel in der Schlüsselmenge befindet oder nicht.

Beweis:

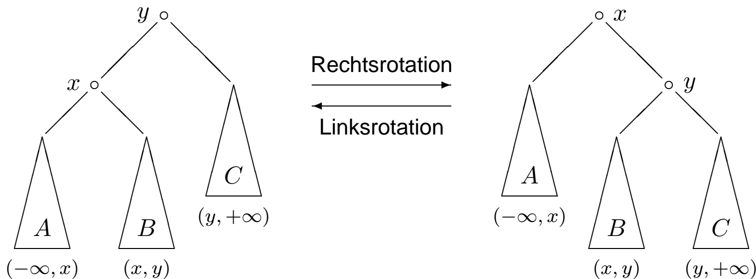
Klar!

Beispiel 153 (Einfügen eines Knotens in AVL-Baum)

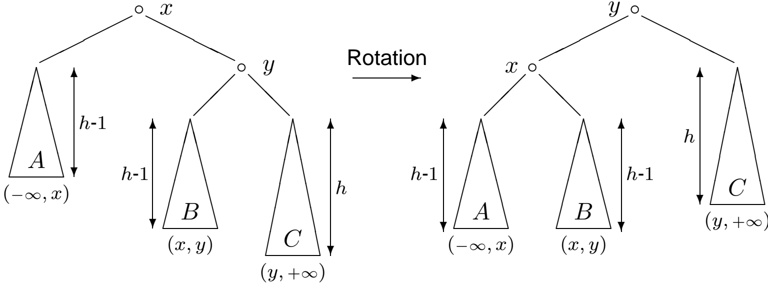


Zur Wiederherstellung der Höhenbedingung benutzen wir so genannte Rotationen und Doppelrotationen.

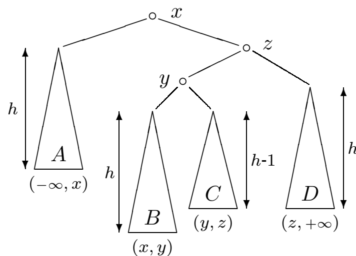
Beispiel 154 (Rotation um (x, y))

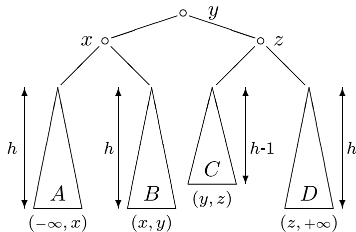


Beispiel 155 (Wiederherstellung der Höhenbedingung)



Beispiel 156 (Doppelrotation zur Rebalancierung)





Zur Rebalancierung des AVL-Baums sind Rotationen und Doppelrotationen nur entlang des Pfades zum eingefügten Knoten erforderlich. Damit ergibt sich

Lemma 157

In einen AVL-Baum mit n Knoten kann ein neuer Schlüssel in Zeit $O(\log n)$ eingefügt werden.

Ebenso kann man zeigen

Lemma 158

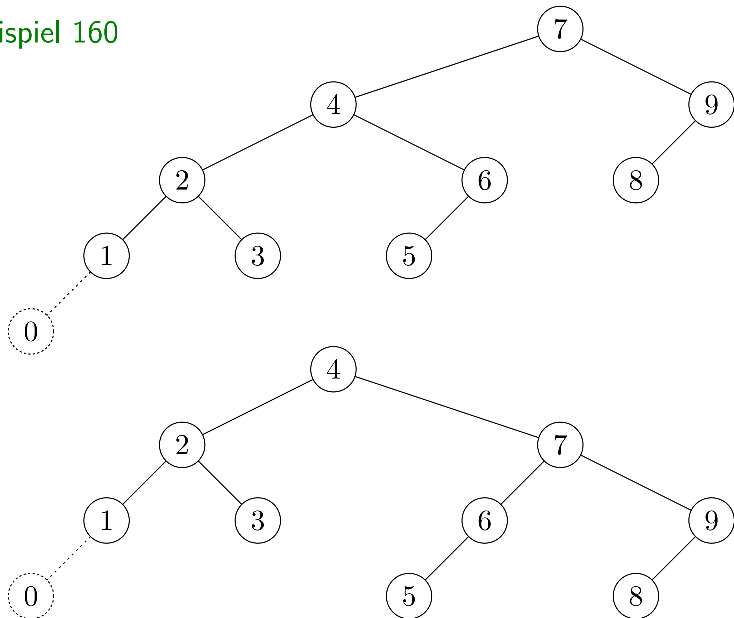
In einen AVL-Baum mit n Knoten kann ein im Baum vorhandener Schlüssel in Zeit $O(\log n)$ gelöscht werden.

Damit

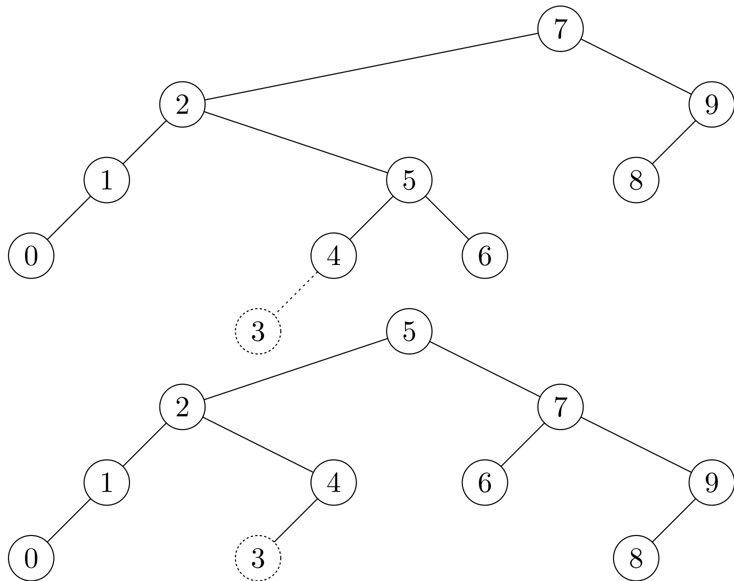
Satz 159

In einem AVL-Baum mit n Knoten kann jede Wörterbuch-Operation in Zeit $O(\log n)$ ausgeführt werden.

Beispiel 160



Beispiel 161 (Rebalancierung mit Doppelrotation)



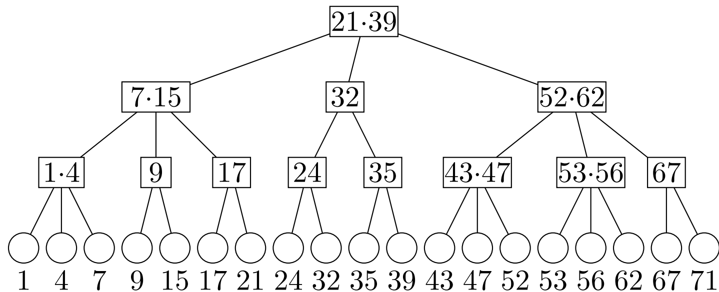
3.3 (a, b)-Bäume

Definition 162

Ein (a, b)-Baum ist ein externer Suchbaum, für den gilt:

1. alle Blätter haben die gleiche Tiefe
2. alle internen Knoten haben $\leq b$ Kinder
3. alle internen Knoten außer der Wurzel haben $\geq a$, die Wurzel hat ≥ 2 Kinder
4. $b \geq 2a - 1$
5. in jedem internen Knoten sind jeweils die größten Schlüssel seiner Unterbäume mit Ausnahme des letzten gespeichert

Beispiel 163

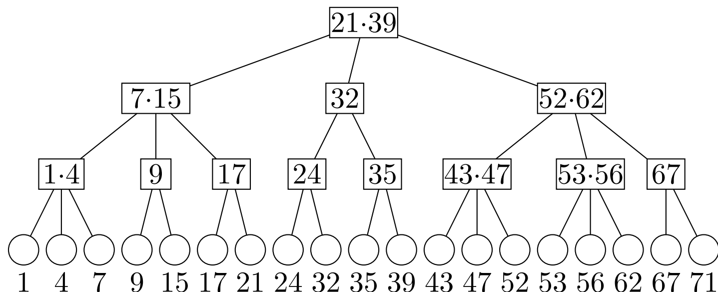


Bemerkung:
(a, b)-Bäume mit $b = 2a - 1$ heißen auch B-Bäume. Diese wurden erstmals in einer Arbeit von R. Bayer und E.M. McCreight im Jahr 1970 beschrieben. (2,3)-Bäume wurden von J. Hopcroft ebenfalls 1970 eingeführt.

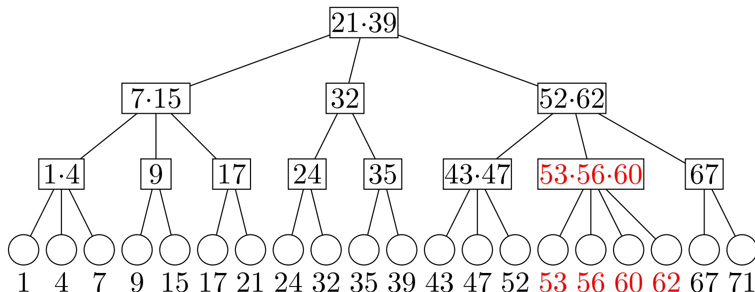
Insert-Operation: Übersteigt durch eine Insert-Operation ein Knoten die Anzahl der zulässigen Kinder, so wird er in zwei Knoten geteilt.

Delete-Operation: Fällt durch eine Delete-Operation die Anzahl der Kinder eines Knoten unter a, so wird ein Kind vom linken oder rechten Geschwister des Knoten adoptiert.

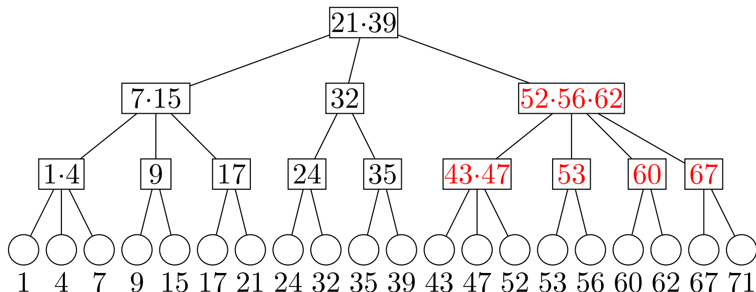
Beispiel 164 (Füge „60“ in (2,3)-Baum ein)



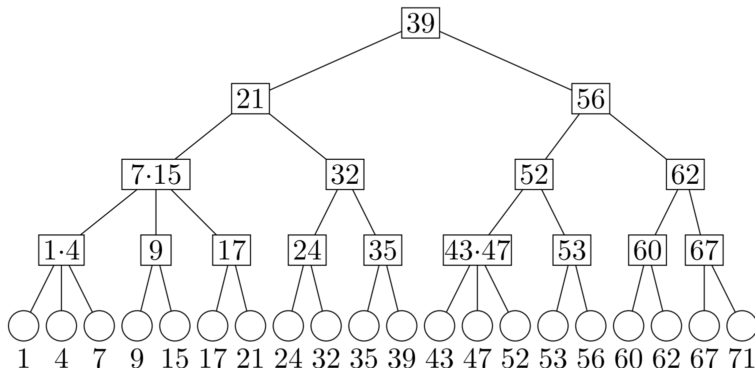
Beispiel 164 (Füge „60“ in (2,3)-Baum ein)



Beispiel 164 (Füge „60“ in (2,3)-Baum ein)



Beispiel 164 (Füge „60“ in (2,3)-Baum ein)



Korollar 165

In einem (a,b)-Baum mit n gespeicherten Schlüsseln können die Wörterbuchoperationen in Zeit $O(\log_a n)$ durchgeführt werden.

Bemerkung: Die Wahl von a und b hängt wesentlich von der Anwendung und der Größe des (a,b)-Baums ab.

- Liegt der (a,b)-Baum im RAM, dann wählt man b klein, um in jedem inneren Knoten den richtigen Unterbaum schnell zu finden.
- Andernfalls wählt man a groß. Der Baum hat dadurch nur geringe Tiefe, seine oberen Schichten können im RAM gehalten werden, und die Anzahl der Zugriffe auf den Sekundärspeicher (Festplatte) ist klein.