Algorithmen und Datenstrukturen

Prof. Dr. Hans-Dietrich Hecker

Wintersemester 2003/04

Inhaltsverzeichnis

Algorithmenverzeichnis

Literatur

Ergänzend zur Vorlesung wird die nachfolgende Literatur empfohlen:

1. Cormen, Rivest, Leiserson, Stein; Introduction to algorithms; MIT Press; 0-262-03293-7
2. Ottmann, Widmeyer; Algorithmen und Datenstrukturen; Spektrum Akademischer Verlag GmbH; 3-8274-1029-0
3. Schöning; Algorithmik; Spektrum Akademischer Verlag GmbH; 3-8274-1092-4
4. Sedgewick; Algorithmen (mehrere unterschiedliche Fassungen verfügbar)

1. Einführung

Liest man heutzutage eine nahezu beliebige Einführung in die theoretische Informatik, so werden zwei Begriffe immer in einem Atemzug genannt: *Algorithmen* und *Datenstrukturen*. Sie gehören zusammen wie die Marktlücke zum Unternehmensgründer. Und tatsächlich kann der wirtschaftliche Erfolg einer EDV-basierten Idee existentiell von der Wahl der passenden Algorithmen und Datenstrukturen abhängen.

Während ein **Algorithmus** die *notwendigen Schritte zum Lösen eines Problems* beschreibt, dienen **Datenstrukturen** *zum Speichern und Verwalten* der verwendeten Daten.

Wie so oft beginnt ein Vorhaben mit den zwei Fragen „Was *habe* ich?“ und „Was *möchte* ich erhalten?“. In Abbildung 1.1 sind drei einander schneidende Rechtecke zu sehen. Sie bilden die Ausgangssituation – das „Was habe ich?“. Man spricht im Allgemeinen vom **Input**. Gesucht ist ein Algorithmus, der die zugehörigen Überlappungen dieser drei Rechtecke ermittelt und ausgibt. Das Ergebnis (Menge der Überlappungen) heißt **Output**.

Zum besseren Verständnis ist es jedoch sinnvoll, das allgemeinere Segmentschnitt-Problem zu betrachten. Eine gegebene Menge von horizontal und vertikal verlaufenden Liniensegmenten in der Ebene soll auf sich schneidende Segmente untersucht werden.

Die in Abbildung 1.2 dargestellten n Linien beschreiben ein solches Segmentschnitt-Problem. Die triviale Idee, jede Strecke mit jeder anderen zu vergleichen und somit alle möglichen Schnittpunkte zu ermitteln, charakterisiert einen sogenannten **Brute-Force-Algorithmus**, der mittels erschöpfenden Probierens zu einer Lösung gelangt. Solche Algorithmen sind im Allgemeinen sehr ineffizient und tatsächlich benötigt dieses Verfahren n^2 Vergleiche (n: Anzahl der Rechteckkanten). Besteht der Input nämlich nicht wie hier aus einigen wenigen Linien, sondern sollen statt dessen eine Million Strecken untersucht werden, dann sind bereits 10^{12} Vergleiche erforderlich.

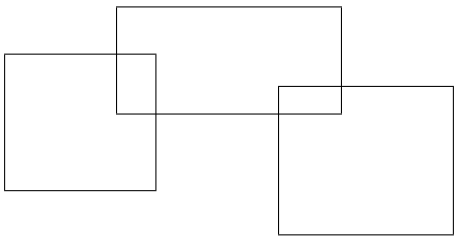


Abbildung 1.1.: Überlappen sich die drei Rechtecke?

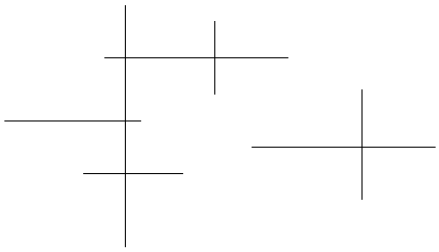


Abbildung 1.2.: Das Segmentschnitt-Problem

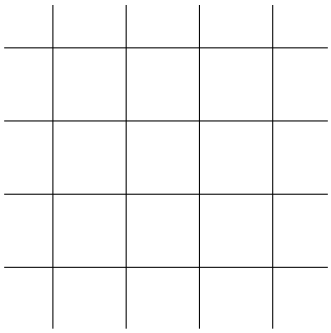


Abbildung 1.3.: Ungünstigste Kantenanordnung, $\frac{n^2}{4}$ Kreuzungspunkte

Im ungünstigsten Fall können alle horizontal verlaufenden Linien alle vertikalen schneiden, und die Hälfte der Kanten verläuft horizontal, die andere Hälfte vertikal, die Strecken bilden dann eine Art Schachbrett 1.3. Jede Linie (Dimension) hat $\frac{n}{2}$ Segmente, die sich mit den anderen $\frac{n}{2}$ Linien schneiden. Die Ausgabe (Output) besteht folglich aus $\frac{n^2}{4}$ Paaren.

Es ist somit eine berechtigte Frage, ob es überhaupt einen besseren Algorithmus als den bereits vorgestellten geben kann.

In praktischen Anwendungen, zum Beispiel dem Chip-Design, ist so ein Kreuzungsbild wie in Abbildung 1.3 eher unwahrscheinlich. Die Anzahl der Kreuzungspunkte wächst üblicherweise in linearer Abhängigkeit zu n oder noch geringer. Die Suche nach einem neuen, im Allgemeinen besseren, Algorithmus ist also sinnvoll.

Wenn wir k als die Anzahl der resultierenden Schnitte für das Segmentschnitt-Problem auffassen, können wir das Problem wie folgt formulieren: Existiert ein Algorithmus mit besserer Laufzeit als n^2 in Abhängigkeit von n und k?

Mit dieser neuen Problemstellung wird die Zeitkomplexität von der Länge der Eingabe und der Länge der Ausgabe abhängig, man spricht von einem **output-sensitiven** Algorithmus.

Eine grundlegende Idee ist dabei die Reduktion der Komplexität durch Verminderung der Dimension. Wie kann man also den Test in der zweidimensionalen Ebene vermeiden und mit einem Test über eine Dimension, z. B. entlang einer Horizontalen, alle Kantenschnitte finden?

Dazu bedient man sich einer Gleitgeraden, die von links nach rechts über die gegebenen Liniensegmente streicht und nur an ihrer aktuellen Position prüft. Dieses Verfahren heißt Plane-Sweep-Algorithmus, eine detaillierte Beschreibung findet sich in Anhang A.

Würde der Strahl stetig über die Segmente streichen, gäbe es überabzählbar viele Prüfpunkte und das Verfahren wäre nicht maschinell durchführbar. Daher ist es auch hier wie überall sonst in der elektronischen Datenverarbeitung notwendig, mit diskreten Werten zu arbeiten.

Zu diesem Zweck definiert man eine Menge mit endlich vielen **event points** (Ereignispunkte), an denen Schnitte gesucht werden sollen. Im vorliegenden Fall kommen dafür nur die x-Werte in Frage, an denen ein horizontales Segment beginnt oder endet bzw. die x-Koordinaten der vertikal verlaufenden Linien. Sie bilden die event points, an denen das Überstreichen simuliert wird. Eine solche Simulation ist einfach zu beschreiben:

Speichere die horizontalen Linien und prüfe beim Auftreten einer senkrechten Linie, ob sie sich mit einer aktuell gemerkten horizontalen schneidet.

Prinzipiell geht man nach folgendem Algorithmus vor:

1. Ordne die event points nach wachsender x-Koordinate (dies sei hier als möglich vorausgesetzt, eine Diskussion über Sortierverfahren erfolgt später)
2. Menge $Y := \emptyset$
   - *INSERT (Aufnahme)* der horizontalen Strecken bzw.
   - *DELETE (Entfernen)*

   der zugehörigen y-Werte
3. bei den event-points (vertikale Strecken):
   - *SEARCH (Suche)* im Vertikalintervall nach y-Werten aus der Menge Y
   - Ausgabe der Schnitte

Die Implementierung des obigen Plane-Sweep-Algorithmus' in einer konkreten Programmiersprache sei als zusätzliche Übungsaufgabe überlassen. Dazu ist jedoch ein dynamischer Datentyp für die Menge Y erforderlich, der die Operationen INSERT, DELETE und SEARCH unterstützt. Dieser wird in einem späteren Kapitel noch behandelt.

Definition 1 (Datenstruktur)
Die Art und Weise wie Daten problembezogen verwaltet und organisiert werden, nennt man Datenstruktur.

Definition 2 (Unterstützte Operationen)
Eine Operation heißt unterstützt, wenn sie für eine Eingabe der Länge n proportional nicht mehr als $\log(n)$ Teilschritte und somit proportional $\log(n)$ Zeiteinheiten benötigt.

Definition 3 (Abstrakter Datentyp (ADT))
Eine Datenstruktur, zusammen mit den von ihr unterstützten Operationen, heißt abstrakter Datentyp (ADT).

Definition 4 (dictionary)
Der abstrakte Datentyp, der das Tripel der Operationen INSERT, DELETE, SEARCH unterstützt, heißt ***dictionary****.*

Der obige PLANE-SWEEP-Algorithmus benötigt $O(n\,log(n))$ Schritte für das Sortieren (siehe unten O-Notation). Da es sich um einen **output-sensitiven** Algorithmus handelt, belaufen sich seine Gesamtkosten auf $O(n\,log(n) + k)$, was meist deutlich besser ist als proportionales Verhalten zu n^2.

Dieses Verfahren erweist sich jedoch mit Hilfe von Arrays als nicht realisierbar, ein guter Algorithmus nützt folglich nichts ohne eine geeignete Datenstruktur.

Um die Effizienz von Algorithmen genauer betrachten zu können, ist es erforderlich, sich im Vorfeld über einige Dinge zu verständigen:

Maschinenmodell: RAM (Random access machine)

- Speicherzugriffszeiten werden ignoriert
- arbeitet mit Maschinenwörtern fester Länge für Zahlen und Zeichen
- Speicher ist unendlich groß
- Operationen wie Addition, Subtraktion, Multiplikation und Division sind in einem Schritt durchführbar

Zeitkomplexität: Mißt die Rechenzeit des Algorithmus

- abhängig von der Größe der Eingabe und der Art
- immer für einen konkreten Algorithmus

Definition 5
best-case: *minimale Rechenzeit für einen Input der Länge n*

average-case: *mittlere Rechenzeit für einen Input der Länge n*

worst-case: *maximale Rechenzeit für einen Input der Länge n*

Für die Analyse von Algorithmen ist der **worst-case** von ganz besonderer Bedeutung. Über ihn sind viele Dinge bekannt, er wird damit mathematisch fassbar und gut berechenbar. Ein Algorithmus, der für den worst-case gut ist, ist auch in allen anderen Fällen gut.

Die worst-case-Zeit T für den Algorithmus a für Eingaben der Länge n ist das Maximum der Laufzeiten für alle möglichen Eingaben dieser Länge:

$$T_a(n) = \max_{w:|w|=n} \{T_a(w)\}$$

Es sei an dieser Stelle angemerkt, daß die worst-case-Betrachtung mitunter einen verzerrten Eindruck liefert und average-case-Betrachtungen aus praktischen Gründen die bessere Wahl darstellen.

Zur Bewertung von Algorithmen haben sich verschiedene Notationen etabliert:

Definition 6

O-Notation: *Es bedeutet $T_a(n) \in O(n^2)$, dass der Algorithmus a* ***höchstens*** *proportional zu n^2 viele Schritte benötigt.*

Ω-Notation: *$T(n) \in \Omega(n^2)$ bedeutet, dass der Algorithmus* ***mindestens*** *prop. zu n^2 viele Schritte benötigt.*

Θ-Notation: *Der Algorithmus benötigt höchstens aber auch mindestens so viele Schritte wie angegeben (Verknüfpung von O- und Ω-Notation)*

Die genaue mathematische Definition erfolgt später.

Zusammen mit dem vereinbarten Maschinenmodell läßt sich nun die Frage untersuchen, wieviel Schritte der obige Plane-Sweep-Algorithmus benötigt. Das Sortieren der event points zu Beginn erfordert $\Theta(n\,log\ n)$ Operationen (der Beweis hierfür erfolgt später). Damit ist $n\,log(n)$ auch für den Gesamtalgorithmus eine untere Schranke.

Wenn die Menge Y in einer Datenstruktur verwaltet wird, die *INSERT*, *DELETE* und *SEARCH* unterstützt, so reichen $O(n \log\ n)$ Schritte sowie zusätzlich $O(k)$ Schritte für die Ausgabe. Der gesamte Algorithmus ist somit in $O(n\,log\,n + k)$ zu bewältigen.

1.1. Über schnelle und langsame Algorithmen

Zeit/ Komplexität	1 sec —	10^2 sec ≈ 1,7 min	10^4 sec ≈ 2,7 Std	10^6 12 Tage	10^8 3 Jahre	10^{10} 3 Jhd.
$1000\,n$	10^3	10^5	10^7	10^9	10^{11}	10^{13}
$100\,n\,log\,n$	$1,4 * 10^2$	$7,7 * 10^3$				$2,6 * 10^{11}$
$100\,n^2$	10^2	10^3	10^4	10^5	10^6	10^7
$10\,n^3$	46	$2,1 * 10^2$	10^3	$4,6 * 10^3$	$2,1 * 10^4$	10^5
.....						
2^n	19	26	33	39	46	53
3^n	12	16	20	25	29	33

Tabelle 1.1.: Zeitkomplexität im Verhältnis zur Eingabegröße

Die Tabelle verdeutlicht die Wichtigkeit schneller Algorithmen. In der linken Spalte steht die Rechenzeit, bezogen auf die Eingabegröße, von Algorithmen. Die Tabelleneinträge geben an, wie groß eine Eingabe sein darf, damit ihr Ergebnis in der angegebenen Zeit berechnet werden kann.

Der konkrete Algorithmus ist hier irrelevant; z.B. werden mit der ersten Zeile alle Algorithmen mit der Laufzeit 1000 n erfaßt.

Bei Algorithmen mit einer schnell wachsenden Laufzeit kann auch in wesentlich mehr Zeit bzw. mit wesentlich schnelleren Rechnern nur ein minimal größeres Problem gelöst werden. Deswegen ist es wichtig, Algorithmen zu finden, deren Rechenzeit bei einer wachsenden Eingabegröße möglichst langsam wächst.

Die punktierte Linie ist nach Cook „die Trennung zwischen Gut und Böse“. Die Laufzeit der ersten Zeile ist linear und somit sehr gut. Für kleine Eingaben reichen auch Laufzeiten von $O(n^2)$ und von $O(n^3)$ aus.

1.2. Die Klassen P und NP

Das Problem des Handlungsreisenden, manchmal auch mit **TSP** abgekürzt, sieht wie folgt aus: Es gibt n Städte, die alle einmal besucht werden sollen. Dabei soll die Rundreise so kurz wie möglich sein.

Dieses Problem hat eine Laufzeit von $O(n!)$ (exponentiell), da alle Reihenfolgen durchprobiert werden müssen. Es dauert zu lange, die beste Route herauszufinden. Eine Möglichkeit wäre es, nur eine Route zu testen.

Das TSP ist ein typisches Beispiel für Probleme der Klasse NP. Nichtdeterministisch kann es in polynomialer Zeit gelöst werden, deterministisch nur in exponentieller Zeit (es sei denn P=NP). Dies führt zur Definition der Klassen P und NP.

P: Die Klasse der Probleme, die für eine Eingabe der Größe n in $Pol(n)$ (polynomialer) Zeit gelöst werden können.

NP: Die Klasse der Probleme, die für eine Eingabe der Größe n **nichtdeterministisch** in $Pol(n)$ Zeit gelöst werden können (nur überprüfen, ob die Lösung richtig ist, typisch sind rate-und-prüfe-Verfahren).

Die große Frage lautet: $P = NP$? Kann man nichtdeterministische Algorithmen durch deterministische ersetzen, die im Sinne der O-Notation ebenfalls polynomiale Laufzeit haben? Dieses offene Problem besteht seit 1970; bisher gibt es keine Ansätze zu einer Lösung. Ein dazu äquivalentes Problem wurde aber bereits 1953 von G. Asser formuliert.

Mehr dazu gibt es in der Informatik IV, hier beschäftigen wir uns ausschließlich mit effizienten Algorithmen.

1.3. Effiziente Algorithmen

Definition 7 (Effiziente Algorithmen)
*Ein Algorithmus, dessen Laufzeit im schlimmsten Fall (*worst case*) von $O(n^k)$ (k konst.) nach oben beschränkt wird, d. h. er hat* polynomielle Laufzeit*, heißt effizient.*

Es gilt also für ein festes k:

$$T_a(n) = O(n^k)$$

1.3.1. Die Zeitanalyse an einem Beispiel

Allgemein stellt sich ein Sortierproblem wie folgt dar:

INPUT Folge $< a_1, \ldots, a_n >$, $n \in \mathbb{N}$, a_i Elemente einer linear geordneten Menge (also eine Menge mit einer totalen, transitiven, reflexiven und antisymmetrische Relation)

OUTPUT: umgeordnete Folge $< a_{\Pi(1)}, \ldots, a_{\Pi(n)} >$: $a_{\Pi(1)} \leq a_{\Pi(2)} \leq \ldots \leq a_{\Pi(n)}$, Π: Permutation

Die zu sortierende Folge liegt meist in einem Feld `A[1...n]` vor.

Definition 8 (InsertionSort)
Dieser Algorithmus funktioniert so, wie viele Menschen Karten sortieren.

Man nimmt eine Karte auf und vergleicht diese nacheinander mit den Karten, die man bereits in der Hand hält. Sobald die Karte wertmäßig zwischen zwei aufeinaderfolgenden Karten liegt, wird sie dazwischen gesteckt.

Die Karten vor uns auf dem Boden entsprechen der nicht sortierten Folge in dem Feld A. Der Algorithmus dafür sind nun wie folgt aus:

InsertionSort

```
1 for j := 2 to length(A) do
2   key := A[j];
3   i := j-1;
4   while (i > 0 and A[i] > key) do
5     A[i+1] := A[i];
6     i := i-1;
7   A[i+1] := key;
```

1.3.2. Beispiel für InsertionSort

INPUT: $A =< 5, 1, 8, 0 >$

OUPUT: $< 0, 1, 5, 8 >$

Problemgröße: (= Größe der Eingabe) n

Ablauf:

`for`-Zähler	`key`	`i`	Feld A (Am Ende der `while`)
Anfang			$< 5, 1, 8, 0 >$
$j = 2$	1	0	$< 1, 5, 8, 0 >$
$j = 3$	8	1	$< 1, 5, 8, 0 >$
$j = 4$	0	3	$< 1, 5, 0, 8 >$
$j = 4$	0	2	$< 1, 0, 5, 8 >$
$j = 4$	0	1	$< 0, 1, 5, 8 >$

Nun stellt sich natürlich die Frage nach der Laufzeit des Algorithmus für eine Eingabe der Größe n. Dazu treffen wir erstmal folgende Festlegungen. Die Laufzeit wird im weiteren als Komplexität des Algorithmus bezeichnet.

Definition: c_i sind die Kosten für die Ausführung der i-ten Zeile des Algorithmus.
Definition: t_j ist Anzahl der Ausführungen des Tests der while-Bedingung für A[j].

In der Tabelle wird die Anzahl der Ausführungen jeder Zeile des Pseudocodes angegeben. Daraus errechnen wir dann die Komplexität des Algorithmus, so erhalten wir eine Aussage über die Güte des Verfahrens.

Befehl	c_i	Anzahl der Aufrufe
1	c_1	n
2	c_2	$n-1$
3	c_3	$n-1$
4	c_4	$\sum_{j=2}^{n} t_j$
5	c_5	$\sum_{j=2}^{n} t_j - 1$
6	c_6	$\sum_{j=2}^{n} t_j - 1$
7	c_7	$n-1$

Tabelle 1.2.: Kosten für InsertionSort

$$
\begin{aligned}
T(n) &= c_1\, n + c_2(n-1) + c_3(n-1) + c_4 \sum_{j=2}^{n} t_j + (c_5 + c_6) \sum_{j=2}^{n} (t_j - 1) \\
&= \underbrace{(c_1 + c_2 + c_3 + c_4 + c_7)}_{a}\, n - \underbrace{(c_2 + c_3 + c_4 + c_7)}_{b} + \underbrace{(c_4 + c_5 + c_6)}_{d} \sum_{j=2}^{n} (t_j - 1) \\
&= a\, n - b + d \sum_{j=2}^{n} (t_j - 1)
\end{aligned}
$$

Als Laufzeiten erhalten wir also

bester Fall (schon sortiert): $\forall j\ t_j = 1 \Rightarrow T(n) = a * n - b \qquad = \Theta(n)$
schlechtester Fall: $\forall j\ t_j = j \Rightarrow T(n) = \sum_{j=2}^{n}(j-1) = \frac{n^2-n}{2} \quad = O(n^2)$
durchschnittlicher Fall: $T(n) = \frac{d}{4}n^2 + (a - \frac{d}{4})n - b \quad = O(n^2)$

Anmerkungen dazu:

Für den durchschnittlichen Fall wird angenommen, daß alle Inputreihenfolgen gleichwahrscheinlich sind. Dieser Algorithmus braucht auch im **average case** $O(n^2)$. Für das Sortieren gibt es bessere Algorithmen.

1.4. Die O-Notation

$\forall_n^\infty$ bedeutet für alle n bis auf endlich viele Ausnahmen, gleichbedeutend mit $\exists n_0 \in \mathbb{N} : \forall n \geq n_0$

$$
\begin{aligned}
O(g(n)) &:= \{f(n) | \exists c_2 > 0, n_0 \in \mathbb{N} :\ \forall_n^\infty : 0 \leq f(n) \leq c_2\, g(n)\} \\
\Omega(g(n)) &:= \{f(n) | \exists c_1 > 0, n_0 \in \mathbb{N} :\ \forall_n^\infty : 0 \leq c_1\, g(n) \leq f(n)\} \\
\Theta(g(n)) &:= \{f(n) | \exists c_1, c_2 > 0, n_0 \in \mathbb{N} :\ \forall_n^\infty : c_1\, g(n) \leq f(n) \leq c_2\, g(n)\} = O(g(n)) \cap \Omega(g(n)) \\
o(g(n)) &:= \{f(n) | \exists c_2 > 0, n_0 \in \mathbb{N} :\ \forall_n^\infty : 0 \leq f(n) < c_2\, g(n)\} \\
\omega(g(n)) &:= \{f(n) | \exists c_1 > 0, n_0 \in \mathbb{N} :\ \forall_n^\infty : 0 \leq c_1\, g(n) < f(n)\}
\end{aligned}
$$

f wächst mit zunehmendem n proportional zu g.

1.4.1. Ein Beispiel

Es sei $f(n) = n^2 + 99n$

1. Behauptung: $f \in O(n^2)$
   Beweis: Gesucht ist ein $c > 0$ und ein $n_0 \in \mathbf{N}$, für das gilt $f(n) \leq c\,n^2$ für alle $n \geq n_0$
   Das bedeutet konkret für unsere Behauptung:
   $f(n) = n^2 + 99n \leq n^2 + 99n^2 = 100n^2$.
   Mit den Werten $c = 100$ und $n_0 = 1$ ist unsere Behauptung erfüllt.

2. Behauptung: $f \in \Omega(n^2)$
   Hier werden Werte $c > 0$, $n_0 \in \mathbf{N}$ gesucht für die gilt: $f(n) \geq c\,n^2\ \forall n \geq n_0$. Also $n^2 + 99n \geq c\,n^2$. Das läßt sich umformen zu $99n \geq (c-1)n^2$ und weiter zu $99 \geq (c-1)n$, also ist jedes $c : 0 < c \leq 1$ eine Lösung.

3. Behauptung: $f \in \Theta(n^2)$
   Beweis: $f \in O(n^2), \quad f(\Omega(n^2)) \Rightarrow f \in O(n^2) \cap \Omega(n^2) = \Theta(n^2)$

4. Behauptung: $f \in O(n^2 \log\log n)$
   Beweis: Übung

1.4.2. Einige Eigenschaften

Transitivität: $f(n) \in O(g(n))$ und $g(n) \in O(h(n)) = f(n) \in O(h(n))$
$f(n) \in o(g(n))$ und $g(n) \in o(h(n)) \Rightarrow f(n) \in o(h(n))$
$f(n) \in \Omega(g(n))$ und $g(n) \in \Omega(h(n)) = f(n) \in \Omega(h(n))$
$f(n) \in \omega(g(n))$ und $g(n) \in \omega(h(n)) \Rightarrow f(n) \in \omega(h(n))$
$f(n) \in \Theta(g(n))$ und $g(n) \in \Theta(h(n)) = f(n) \in \Theta(h(n))$

Reflexivität: $f(n) \in O(f(n)), \quad f(n) \in \Theta(f(n)), \quad f(n) \in \Omega(f(n))$

Symmetrie: $f(n) \in \Theta(g(n)) \Leftrightarrow g(n) \in \Theta(f(n))$

Schiefsymmetrie: $f(n) \in O(g(n)) \Leftrightarrow g(n) \in \Omega(f(n))$

Θ ist eine Äquivalenzrelation auf der Menge der schließlich positiven Funktionen. O, o, Ω, ω sind nichtlineare (totale) Ordnungen.

Beispiel: $f(n) = n$ und $g(n) = n^{1+sin(n\Pi)}$ sind nicht vergleichbar mittels der O-Notation.

1.4.3. Konventionen

1. Wir führen die Zeichen *floor* $\lfloor\rfloor$ und *ceil* $\lceil\rceil$ ein, wobei $\lfloor x \rfloor$ ($\lceil x \rceil$) die größte (kleinste) ganze Zahl kleiner oder gleich (größer oder gleich) x bezeichnet. Z.B. $3 = \lfloor 3,5 \rfloor \leq 3,5 \leq \lceil 3,5 \rceil = 4$

2. Der Logarithmus log soll immer als $\log_2$, also als dualer Logarithmus interpretiert werden. Im Sinne der O-Notation ist das irrelevant, da Logarithmen mit unterschiedlicher Basis in einem konstanten VerhÄltnis zueinander stehen. Z.B. $\log_2 n = 2 \log_4 n$

3. $\log^{(0)} n := n; \quad \log^{(i)} n := \log^{(i-1)} \log n$

4. $\log^* n := \min \{i \,|\, log^{(i)} n \leq 1\}$ Es gilt $\lim\limits_{n\to\infty} log^* n = +\infty$

1.4.4. Eine Beispieltabelle

Die folgende Tabelle enthält, aufsteigend nach dem Wachstum geordnet, Beispielfunktionen. Dabei soll gelten: $f(n) = o(g(n))$; $0 < \alpha < \beta$, $0 < a < b$, $1 < A < B$, $\alpha,\ \beta,\ a,\ b,\ A,\ B \in \mathbb{R}$.

Die Linie zwischen Formel Nummer neun und zehn repräsentiert die bereits erwähnte Cook'sche Linie.

Nummer	Funktion	
1	$\alpha(n)$	
2	$\log^* n$	
3	$\log \log n$	
4	$(log\, n)^\alpha$	
5	$(log\, n)^\beta$	
6	n^a	
7	$n\,(log\, n)^\alpha$	
8	$n^\alpha (\log n)^\beta$	
9	n^b	noch polynomial
10	A^n	exponentiell
11	$A^n n^a$	
12	$A^n n^b$	
13	B^n	

Desweiteren gilt die folgende Regel:
$(f_1(n) + \cdots + f_m(n)) \in O(\max\{f_1(n), \ldots, f_m(n)\})$, mengentheoretisch ausgedrückt gilt also: $O(f_1(n)) \cup \cdots \cup O(f_m(n)) = O(\max\{f_1(n), \ldots, f_m(n)\})$

2. Jetzt geht's los

2.1. Rekurrenzen

Hier werden ein rekursive Ansätze verwendet. Das Ausgangsproblem wird also in immer kleinere Teilprobleme zerlegt. Irgendwann liegt, analog zu einem induktiven Beweis, ein Trivialfall vor, der sich einfach lösen läßt. Aus den Lösungen der Trivialfälle wird dann sukzessiv eine Lösung des Gesamtproblems konstruiert.

2.1.1. Sortieren über Divide and Conquer (Teile und Herrsche)

Dazu wird zunächst das Verfahren MergeSort vorgestellt und anhand eines Beispiels verdeutlicht

Definition 9 (MergeSort)
Eine Folge $A = a_l \dots a_r$ von n=r-l+1 Schlüsseln wird sortiert, indem sie zunächst rekursiv immer weiter in möglichst gleich lange Teilfolgen gesplittet wird. Haben die Teilfolgen die Länge 1 können jeweils zwei durch einen direkten Vergleich sortiert werden. Dann werden die Teilfolgen wieder schrittweise zusammengemischt, bis schließlich die sortierte Eingabe vorliegt.

2.1.2. Ein Beispiel für MergeSort

1 7 8 3 4 6 5 9

1 7 8 3 4 6 5 9

1 7 8 3 4 6 5 9

$\underbrace{1\ 7\ 8\ 3\ 4\ 6\ 5\ 9}_{\text{Aufsplitten der Eingabe}}$

1 7 3 8 4 6 5 9

1 3 7 8 4 5 6 9

$\underbrace{1\ 3\ 4\ 5\ 6\ 7\ 8\ 9}_{\text{Sukzessives Zusammenmischen der sortierten Teilfolgen}}$

Das Mischen funktioniert in O(n) Zeit, zur Verdeutlichung wird es nochmal exemplarisch skizziert, dazu werden zwei Folgen mit m Elementen zusammengemischt.

$$\left.\begin{array}{l} a_1 < \ldots < a_m \\ \\ b_1 < \ldots < b_m \end{array}\right\} c_1 < \ldots < c_{2m}$$

Die beiden Folgen werden also zu einer in sich sortierten Folge der doppelten Länge gemischt. Wie im folgenden zu sehen, werden immer genau zwei Elemente miteinander verglichen. Der Fall, daß zwei Teilfolgen unterschiedliche Länge haben, kann o.B.d.A. ignoriert werden.

$$b_1 < a_1 \rightsquigarrow \underset{b_1}{\downarrow} \rightsquigarrow b_2 < a_1 \rightsquigarrow \underset{b_2}{\downarrow} \rightsquigarrow a_1 < b_3 \rightsquigarrow \underset{a_1}{\downarrow} \rightsquigarrow \ldots \rightsquigarrow b_1,\, b_2,\, a_1, \ldots$$

An konkreten Zahlen läßt sich das vielleicht noch besser verfolgen, das Mischen im letzten Schritt aus dem Beispiel sähe wie folgt aus.

$$1 < 4 \rightsquigarrow \underset{①}{\downarrow} \rightsquigarrow 3 < 4 \rightsquigarrow \underset{③}{\downarrow} \rightsquigarrow 7 > 4 \rightsquigarrow \underset{④}{\downarrow} \rightsquigarrow 7 > 5 \rightsquigarrow \underset{⑤}{\downarrow} \rightsquigarrow 7 > 6 \rightsquigarrow \underset{⑥}{\downarrow} \rightsquigarrow$$
$$7 < 9 \rightsquigarrow \underset{⑦}{\downarrow} \rightsquigarrow 8 < 9 \rightsquigarrow \underset{⑧}{\downarrow} \rightsquigarrow 9 < +\infty \rightsquigarrow \underset{⑨}{\downarrow} \rightsquigarrow < 1, 3, 4, 5, 6, 7, 8, 9 >$$

Der Vergleich mit ∞ vereinfacht das Mischen, da sich damit eine kompliziertere Fallunterscheidung für den Fall erübrigt, daß alle Elemente einer Teilfolge beim Mischen bereits „verbraucht“ wurden.

Jeder Vergleich von zwei Elementen ergibt ein Element der neuen Folge und es werden immer nur zwei Werte verglichen, was in O(1) Zeit klappt. Also sind zwei Teilfolgen nach $O(n)$ Schritten zu einer neuen Folge zusammengemischt und man erhält $O(n)$ als Kosten für das Mischen. Für das gesamte Verfahren MergeSort ergibt sich die *Rekurrenz* T(n)=2T(n/2)+O(n), die zu einer Komplexität von O(n log n) führt. Verfahren für das Lösen von Rekurrenzen werden nach Angabe des Pseudocodes und einer etwas formaleren Laufzeitanalyse für MergeSort eingeführt.

MergeSort

```
1 if (l < r) then
2   p :=⌊(l+r)/2⌋
3   MergeSort(A, l, p)
4   MergeSort(A, p+1, r)
5   Merge
```

Zeile 1 und 2 sind elementare Vergleichs- und Zuweisungsoperationen, welche in O(1) Zeit möglich sind. Dem rekursiven Aufruf von Mergesort in Zeile 3 und 4 wird jeweils nur eine Hälfte der Schlüssel übergeben, daher ist der Zeitaufwand je $T(\frac{n}{2})$. Für das Zusammenführen der beiden Teilmengen in Zeile 5 gilt: Für zwei Teilmengen der Länge n_1 und n_2 sind mindestens $min(n_1, n_2)$ und höchstens $n_1 + n_2 - 1$ Schlüsselvergleiche notwendig. Zum Verschmelzen zweier etwa gleich langer Teilfolgen der Gesamtlänge n, werden also im ungünstigsten Fall $\Theta(n)$ Schlüsselvergleiche benötigt.

Zeile	Asymptotischer Aufwand
1	$\Theta(1)$
2	$\Theta(1)$
3	$T(\frac{n}{2})$
4	$T(\frac{n}{2})$
5	$\Theta(n)$

Tabelle 2.1.: Kosten für MergeSort

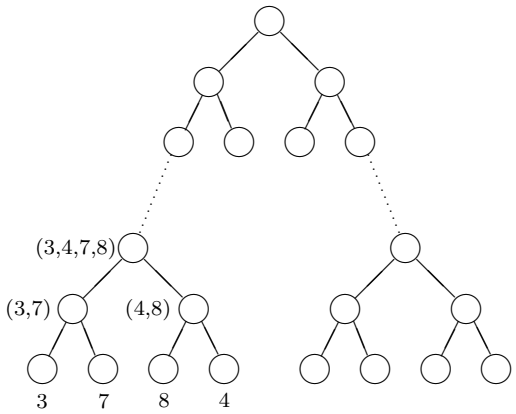
Wie bereits gezeigt, gilt für die Gesamtlaufzeit die folgende Rekurrenz $T(n) = 2 \cdot T(\frac{n}{2}) + \Theta(n)$, zum Lösen einer Rekurrenz muß aber auch immer eine sog. boundary condition, zu deutsch Randbedingung, bekannt sein, analog zur Lösung des Trivialfalles einer rekursiven Funktion. Bei MergeSort gilt für T(1) die Randbedingung $T(1) = 1 \ (= \Theta(1))$

Mit Hilfe des Mastertheorem - siehe Kapitel *Methoden zur Lösung von Rekurrenzen* - ergibt sich folgende Lösung:

$T(n) = \Theta(n \log n)$

Anmerkung: Floors und ceilings werden jetzt und meist in Zukunft weggelassen - das ist vertretbar.

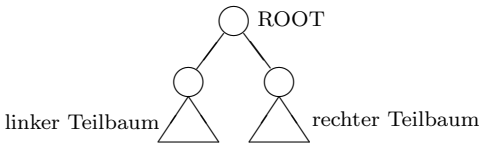
Binärbaum-Paradigma:

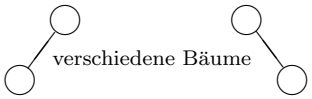


Definition 10

Ein Binärbaum ist eine Menge von drei Mengen von Knoten. Zwei davon sind wieder Binärbaume sind und heissen linker bzw. rechter Teilbaum, die dritte Menge ist die Einermenge $\{ROOT\}$. Andernfalls ist die Menge leer.

Bei einem Baum, der nur aus der Wurzel besteht, sind also die Mengen linker Teilbaum und rechter Teilbaum jeweils die leere Menge.





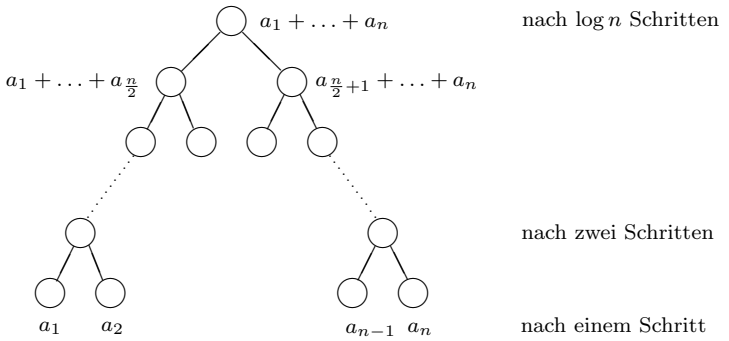
Die Anwendung des Binärbaumparadigmas für parallele Algorithmen wird durch die folgende, grobe Schilderung deutlich. Man stellt sich vor, auf jedem inneren Knoten des Baumes sitzt ein Prozessor, welcher parallel von Level zu Level fortschreitend die Aufgabe löst.

Paralleles und optimales Sortieren benötigt $O(log\, n)$ Zeit. Der Beweis hierfür (Richard Cole) ist extrem schwer und wird an dieser Stelle nicht aufgeführt.

Dafür folgt hier ein einfacheres Beispiel: Addiere n Zahlen, die Eingabe liege wieder als Liste A ($= (a_1, \ldots, a_n)$) vor.

ParallelSort

```
1 Zerlege A in A1, A2 (zwei Hälften)
2  ADD(A1)
3  ADD(A2)
4      ADD(A1)+ADD(A2)
5      Ausgabe
```



Für die Laufzeit gilt nun die Rekurrenz $\mathrm{T}_{\mathrm{parallel}}(n) = \mathrm{T}_{\mathrm{parallel}}(\frac{n}{2}) + O(1)$, also $\mathrm{T}(n) = \mathrm{T}(\frac{n}{2}) + 1$ und nach Auflösen der Rekurrenz T(n) = O(log n)

2.2. Methoden zur Lösung von Rekurrenzen

Viele Algorithmen enthalten rekursive Aufrufe. Die Zeitkomplexität solcher Algorithmen wird oft durch Rekurrenzen beschrieben.

Definition 11 (Rekurrenzen)
Rekurrenzen sind Funktionen, welche durch ihren eigenen Wert für kleinere Argumente beschrieben werden. Für den Trivialfall muß wie bei jeder rekursiven Funktion eine Lösung angegeben werden.

Dieses Kapitel liefert vier Ansätze zum Lösen von Rekurrenzen. Begonnen wird mit der Methode der vollständigen Induktion (Substitutionsmethode), bei der zur Lösung von Rekurrenzgleichungen im ersten Schritt eine mögliche Lösung der Gleichung erraten wird, die im zweiten Schritt mittels vollständiger Induktion bestätigt werden muß.

Beispiel: $T(n) = 2\,T\left(\frac{n}{2}\right) + n$, Randbedingung: $T(1) = 1$

Ansatz: $T(n) \leq c\,n\,log\,n$; c ist konstant, $c \geq 1$

$$\begin{aligned} T(n) &= 2\,T\left(\tfrac{n}{2}\right) + n \leq 2\,c\,\left(\tfrac{n}{2}\right)\,log\left(\tfrac{n}{2}\right) + n \\ &= c\,n\,log\,n - c\,n\,log\,2 + n \\ &= c\,n\,log\,n - c\,n + n \leq c\,n\,log\,n \end{aligned}$$

$$\Rightarrow T(n) = O(n\,log\,n)$$

Beispiel: $T(n) = 2\,T\left(\frac{n}{2}\right) + b$, Randbedingung: $T(1) = 1$

Ansatz: $T(n) = O(n) \Rightarrow T(n) \leq c\,n$; c ist konstant

$$\begin{aligned} T(n) &= 2\,T(\tfrac{n}{2}) + b \\ &= 2c\,\tfrac{n}{2} + b \\ &= c\,n + b; \text{ Annahme nicht erfüllt, kein ausreichender Beweis} \end{aligned}$$

neuer Ansatz: $T(n) \leq c\,n - b$

$$\begin{aligned} T(n) &= 2\,T\left(\tfrac{n}{2}\right) + b \\ &\leq 2\left(c\,\tfrac{n}{2} - b\right) + b \\ &= c\,n - 2\,b + b \\ &\leq c\,n - b \end{aligned}$$

$$\Rightarrow T(n) = O(n)$$

Als nächstes wird die Methode der Variablensubstitution gezeigt, bei der ein nicht elementarer Teilausdruck durch eine neue Variable substituiert wird. Dabei ist es wesentlich, daß sich die dadurch neu entstandene Funktion gegenüber der Ausgangsfunktion vereinfacht. Die vereinfachte Rekurrenzgleichung wird mittels anderer Verfahren gelöst. Das Ergebnis wird anschließend rücksubstituiert.

Beispiel: $T(n) = 2\,T(\lceil\sqrt{n}\rceil) + log\,n$, Randbedingung: $T(1) = 1$

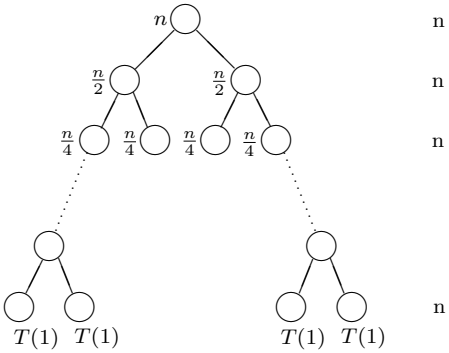
Substitutionsansatz: $k := log\,n \Rightarrow 2^k = n$, nach Einsetzen gilt also: $T(2^k) = 2\,T(2^{\frac{k}{2}}) + k$

Jetzt wird $S(k) := T(2^k)$ gesetzt und somit gilt $S(k) = 2\,S\left(\frac{k}{2}\right) + k$, die Auflösung dieser Rekurrenz z.B. mittels Induktion sei als Übungsaufgabe überlassen und ist deswegen im Anhang zu finden. Für die Lösung der Ausgangsrekurrenz muß dann aber noch die Substitution rückgängig gemacht werden, dabei sei ebenfalls auf Anhang B.1 verwiesen.

Als dritttes wird die Methode der Rekursionsbäume vorgestellt. Hierbei wird die Funktion mittels eines Rekursionsbaumes dargestellt. Dabei wird der nicht-rekursive Funktionsanteil in jeden Knoten geschrieben. Für jeden rekursiven Aufruf pro Funktionsaufruf erhält jeder Knoten einen Sohnknoten. Dies wird solange fortgeführt bis in den Blättern ein Wert < 1 steht. Die Summe aller Knoteneinträge bezeichnet die Lösung der Rekurrenz.

Beispiel: $T(n) = 2\,T\left(\frac{n}{2}\right) + n$, Randbedingung: $T(1) = 1$

Ansatz: $T(n) = n + \frac{n}{2} + \frac{n}{2} + \frac{n}{4} + \frac{n}{4} + \frac{n}{4} + \frac{n}{4} + 8\,\frac{n}{8} + \ldots + 2^k\,\frac{n}{2^{k+1}}\,T\left(\frac{n}{2^{k+1}}\right)$



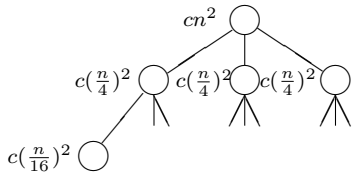
Der Aufwand jeder Zeile beträgt O(n). Der Baum hat eine Höhe von $log\,n$. Damit ergibt sich als Lösung: $O(n\,log\,n)$

Beispiel: $T(n) = 3\,T(\frac{n}{4}) + c\,n^2$

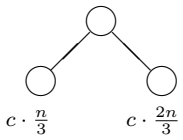
Ansatz: $T(n) = c\,n^2 + \left(\frac{3}{16}\right)\,c\,n^2 + \left(\frac{3}{16}\right)^2\,c\,n^2 + \ldots + \left(\frac{3}{16}\right)^{log\,(n-1)}\,c\,n^2 + \Theta\left(n^{\log_4 3}\right)$

[es gilt: $n^{\log_4 3} = 3^{\log_4 n}$]

$T(n) = \sum_{i=0}^{\log_4(n-1)} \left(\frac{3}{16}\right)^i\,c\,n^2 + \Theta\left(n^{\log_4 3}\right) < \sum_{i=0}^{\infty} \left(\frac{3}{16}\right)^i\,c\,n^2 + \Theta(n^{\log_4 3})$
$= \frac{1}{1-\frac{3}{16}}\,c\,n^2 + \Theta(n^{\log_4 3}) = O(n^2)$



Beispiel: $T(n) = T\left(\frac{n}{3}\right) + T\left(\frac{2 \cdot n}{3}\right) + O(n)$, Randbedingung: $T(1) = 1$



Die vierte Methode, das Lösen mittels des sogenannten Mastertheorems erhält wegen ihres gänzlich anderen Charakters einen eigenen Abschnitt.

2.3. Das Mastertheorem

Hier rückt das Wachstum des nicht-rekursiven Summanden im Vergleich zum Wachstum des rekursiven Anteils in den Mittelpunkt. Die Frage lautet also, wie f(n) im Vergleich zu $\mathrm{T}\left(\frac{n}{b}\right)$ wächst, die Vergleichsgröße ist dabei $n^{\log_b a}$. Im ersten Fall wächst f(n) langsamer, im zweiten gleich schnell und im dritten polynomial schneller.

Sei $\mathrm{T}(n) = a\,\mathrm{T}\left(\frac{n}{b}\right) + \mathrm{f}(n)$ mit $a \geq 1,\ b > 1$, dann gilt asymptotisch für große n.

1. $\mathrm{f}(n) \in \mathrm{O}(n^{\log_b a - \varepsilon})$ (mit $\varepsilon > 0$ fest) $\rightarrow \mathrm{T}(n) \in \Theta(n^{\log_b a})$
2. $\mathrm{f}(n) \in \Theta(n^{\log_b a}) \rightarrow \mathrm{T}(n) \in \Theta(n^{\log_b a} \log n)$
3. $(\mathrm{f}(n) \in \Omega(n^{\log_b a + \varepsilon})$ (mit $\varepsilon > 0$ fest) $\wedge\ \exists c < 1 : \forall_n^\infty a\,\mathrm{f}\left(\frac{n}{b}\right) \leq c\,\mathrm{f}(n)) \rightarrow \mathrm{T}(n) \in \Theta(\mathrm{f}(n))$

Der Beweis ist etwas schwieriger und für diese Vorlesung auch nicht von allzu großer Bedeutung. R. Seidel hat auf seiner Homepage von 1995 den Kernsatz bewiesen, der ganze aber etwas kompliziertere Beweis ist in [2] zu finden.

2.3.1. Beispiele

Beispiel: Binäre Suche

Eingabe ist eine sortierte Folge $\mathrm{a}_1 < \cdots < \mathrm{a}_\mathrm{n}$ und ein Wert b. Ausgegeben werden soll $\iota\,\mathrm{i}{:}\mathrm{a}_\mathrm{i} \leq b < \mathrm{a}_{\mathrm{i}+1}$, falls es existiert und ansonsten eine Fehlermeldung.

EXKURS: Binärer Suchbäum

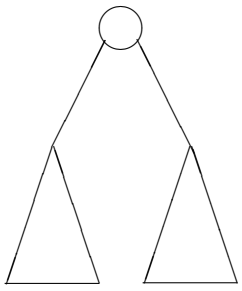


Abbildung 2.1.: Binärbaum

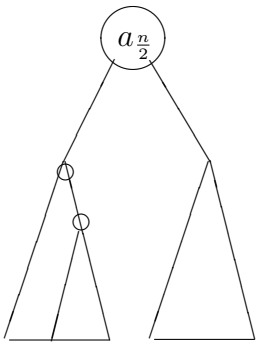


Abbildung 2.2.: Suche im Binärbaum

Eigenschaften:

- Die Werte sind in Bezug auf die Größe vergleichbar
- Die rechten Söhne eines Knotens enthalten größere Werte als die linken Söhne.

Bei einer Anfrage an einen höhenbalancierten binären Suchbaum werden in jedem Schritt die in Frage kommenden Werte halbiert (siehe Abbildung 2.2), es werden praktisch Fragen der folgenden Form gestellt.

$b < a_{\frac{n}{2}}$ oder $b \geq a_{\frac{n}{2}}$?
$b < a_{\frac{n}{4}}$ oder $b \geq a_{\frac{n}{4}}$? bzw. $b < a_{\frac{3n}{4}}$ oder $b \geq a_{\frac{3n}{4}}$?
usw.

Die Suche läuft bis zu der Stelle, an der der gesuchte Wert sein müsste. Wenn er nicht dort ist, ist er nicht in der sortierten Folge vorhanden. Der Einschränkung des Suchraumes durch eine Intervallhalbierung entspricht jeweils ein Abstieg innerhalb des Baumes um einen Höhenlevel. D.h. die Anzahl der Rechenschritte ist so groß, wie der Baum hoch ist und es gilt die Rekurrenz: $\mathrm{T(n)} = \mathrm{T}(\frac{\mathrm{n}}{2}) + O(1)$

Zur Veranschaulichung einer alternativen Vorstellung, bei der in einem Feld gesucht wird, gehen wir von folgender Wette aus:

Denke dir eine natürliche Zahl a zwischen 0 und 1000. Wetten, daß ich mit 10 Fragen herausbekomme, welche Zahl du dir gedacht hast!

Nun sei 128 die gedachte Zahl, die Fragen sähen dann so aus:
1. Ist a<500? $\Rightarrow$ 0≤a<500
2. Ist a<250? $\Rightarrow$ 0≤a<250
3. Ist a<125? $\Rightarrow$ 125≤a<250
4. Ist a<187? $\Rightarrow$ 125≤a<187
5. Ist a<156? $\Rightarrow$ 125≤a<156
6. Ist a<141? $\Rightarrow$ 125≤a<141
7. Ist a<133? $\Rightarrow$ 125≤a<133
8. Ist a<129? $\Rightarrow$ 125≤a<129
9. Ist a<127? $\Rightarrow$ 127≤a<129
10. Ist a<128? $\Rightarrow$ 128≤a<129 $\Rightarrow$ a=128

Bei einer Million Zahlen reichen übrigens 20 Fragen aus!

Alternative Vorstellung schematisch:

a_1	a_2	$\dots$	$a_{\frac{n}{2}}$	$\vert a_{\frac{n}{2}+1}$	$\dots$	a_n
	?	$\geq$	oder	$<$	?	
$\vert$			$\leftarrow$	$\vert \rightarrow$		$\vert$
$\vert$	$\leftarrow$	$\vert \rightarrow$		$\vert$		
		$\vert \leftarrow$	$\vert \rightarrow$	$\vert$		

Wie sieht nun die Einordung der binären Suche in das Mastertheorem aus? Mit a = 1 und b = 2 gilt $\log_b a = 0$, also $\Theta\left(\mathrm{n}^{\log_b a}\right) = \Theta(1)$ und der zweite Fall kommt zur Anwendung. Also ist $\mathrm{T}(n) \in \Theta(n^{\log_b a} \log n) = \Theta(\log n)$.

Weitere Beispiele

1. $\mathrm{T}(n) = 9\,\mathrm{T}\left(\frac{n}{3}\right) + 3\,n \log n$, also $a = 9,\ b = 3$ und $n^{\log_b a} = n^{\log_3 9} = n^2$.

   $\mathrm{f}(n) = 3n \log n \in \mathrm{O}(n^{\log_b a - \varepsilon})$ z.B. $\mathrm{O}(n^{\frac{3}{2}})$ mit $\varepsilon = \frac{1}{2}$

   $\Rightarrow$ Erster Fall $\mathrm{T}(n) \in \Theta(n^{\log_b a}) = \Theta(n^2)$.

2. $\mathrm{T}(n) = 3\,\mathrm{T}\left(\frac{n}{4}\right) + 2n \log n$.

   Die Lösung dieser Übungsaufgabe steht im Anhang B.2.

3. $\mathrm{T}(n) = 2\,T(\frac{n}{2}) + 4n \log n$ also $a = 2,\ b = 2$ und $n^{\log_b a} = n$.

   Wie man weiß: $4n \log n \in \Omega(n)\ \forall \epsilon > 0$ aber $4n \log n \notin \Omega(n^{1+\epsilon})$

   Es trifft <u>kein</u> Fall zu!

Das Mastertheorem deckt nicht alle Fälle ab!

3. Sortieren und Selektion

$O(n \log n)$ ist die obere Schranke für MergeSort ($O(n^2)$ für Insertionsort).
Frage: Geht es besser?

- **Ja** 1. Mit Parallelrechnern, aber das ist nicht Thema dieser Vorlesung.
  2. Unter bestimmten Bedingungen.

- **Nein** bei allgemeinen Sortierverfahren auf der Basis von Schlüsselvergleichen. Unser Ziel ist der Beweis, daß für allgemeine Sortierverfahren auf der Basis von Schlüsselvergleichen $\Omega(n \log n)$ eine untere Schranke ist.

Beweis 1
Modellieren des Ansatzes: „Auf der Basis von Schlüsselvergleichen".

- INPUT ist ein Array mit (o. B. d. A.) paarweise verschiedenen Werten $(a_1, \ldots, a_n)$, $a_i \in S$, $i = (1, \ldots, n)$ auf das nur mit der Vergleichsfunktion

$$V(i,j) := \begin{cases} 1, & a_i < a_j \\ 0, & a_i > a_j \end{cases}$$

  zugegriffen werden kann.

- OUTPUT ist eine Permutation π für die $a_{\pi(1)} < a_{\pi(2)} < \cdots < a_{\pi(n)}$ ist

Sei A ein beliebiges o. B. d. A. deterministisches Sortierverfahren dieser Art. Die erste Anfrage ist dann nicht $(a_i < a_j)$ sondern $(i < j)$

Definition 12
$a(i < j) := \{(a_1, \ldots, a_n | \, a_i \in S \wedge a_i < a_j\}$

Der erste Test $V(i,j)$ der Vergleichsfunktion wird immer für dasselbe Indexpaar (i,j) der Eingabe $(a_1, \ldots, a_n)$ ausgeführt, über die der Algorithmus A zu diesem Zeitpunkt noch keinerlei Informationen besitzt.

Falls nun $V(i,j) = 1$ ist, d. h. für alle Eingaben, die der Menge a(i<j)= $\{(a_1, \ldots, a_n) \in \mathbb{R}^n : a_i < a_j\}$ angehören, wird der zweite Funktionsaufruf immer dasselbe Indexpaar (k,l) als Parameter enthalten, da A deterministisch ist und zu diesem Zeitpunkt nur weiß, daß $a_i < a_j$ ist. Analog wird für alle Folgen a(j<i) derselbe Funktionsaufrauf als zweiter ausgeführt. Die Fortführung dieser Überlegung führt zu dem vergleichsbasierten Entscheidungsbaum von Algorithmus A, einem binären Baum, dessen Knoten mit

Vergleichen „$a_\mathrm{i} < a_\mathrm{j}$“ beschriftet sind. An den Kanten steht entweder „j“ oder „n“. Ein kleines Beispiel ist in Abbildung 3.1 zu sehen.

Genau die Eingabetupel aus der Menge $\mathrm{a}(3 < 4) \cap \mathrm{a}(3 < 2) = \{(a_1, \ldots, a_\mathrm{n}) \in \mathbb{R}^n : a_3 < a_4 \wedge a_3 < a_2\}$ führen zum Knoten $\mathcal{V}$.

Weil A das Sortierproblem löst, gibt es zu jedem Blatt des Entscheidungsbaumes eine Permutation π, so das nur die Eingaben mit $\mathrm{a}_{\pi(1)} < \mathrm{a}_{\pi(2)} < \ldots < \mathrm{a}_{\pi(\mathrm{n})}$ zu diesem Blatt führen. Der Entscheidungsbaum hat daher mindestens n! Blätter. Der Beweis dafür stammt fast unverändert aus [3].

Im Regelfall hat ein Entscheidungsbaum allerdings mehr als $n!$ Blätter. Es gibt auch Knoten, die die leere Menge enthalten, oder Knoten, die nie erreicht werden können (z. B. Knoten $\mathcal{W}$).

Beispiele für den Baum aus Abbildung 3.1:
Bei der Eingabe (3, 4, 17, 25) wäre man nach Abarbeitung der Vergleiche 17<25, 4<17 und 3<4 im linkesten Knoten.
Bei der Eingabe (17, 4, 3, 25) wäre man nach Abarbeitung der Vergleiche 3<25 und 4<3 im Knoten $\mathcal{V}$.
Wir gehen über $\mathcal{V}$, wenn $a_3 < a_4$ und $a_3 < a_2$, also für alle Tupel aus $a(3 < 4) \cap a(3 < 2)$.

Satz 1
Ein Binärbaum der Höhe h hat höchstens 2^h Blätter.

Die Höhe eines Entscheidungsbaumes ist die maximale Weglänge von der Wurzel zu einem Blatt, sie entspricht der Rechenzeit. Wie bereits vorhin angedeutet, muß ein solcher Baum mindestens n! Blätter haben, wenn er alle Permuationen der Eingabe berücksichtigen können soll (z.B. für das Sortieren), damit muß gelten

$$2^h \geq n! \leftrightarrow h \geq \log n! \geq \log \left(\frac{n}{2}\right)^{\frac{n}{2}}$$

Der Beweis ist trivial da $n! = 1 \cdot 2 \cdot 3 \ldots \cdot n = 1 \cdot 2 \cdot 3 \ldots \cdot \underbrace{\left(\frac{n}{2} + 1\right) \cdot \ldots \cdot n}_{\frac{n}{2}} \geq \left(\frac{n}{2}\right)^{\frac{n}{2}}$

$\log \left(\frac{n}{2}\right)^{\frac{n}{2}} = \frac{n}{2} \cdot \log \frac{n}{2} = \frac{n}{2} \log n - \frac{n}{2} \underbrace{\log 2}_{1} = \frac{n}{3} \log n + [\frac{n}{6} \log n - \frac{n}{2}] \geq \frac{n}{3} \log n$

für $n \geq 8$ ist $\log n \geq 3 \leftrightarrow \frac{n}{6} \log n \geq \frac{n}{2} \leftrightarrow \frac{n}{6} \log n - \frac{n}{2} \geq 0$, also $h \geq n \log n$.

Worst-case-Fall im Sortieren hier ist ein Ablaufen des Baumes von der Wurzel bis zu einem Blatt und dies geht bei einem Baum der Höhe $n \log n$ in $\mathrm{O}(n \log n)$ Zeit.

q. e. d.

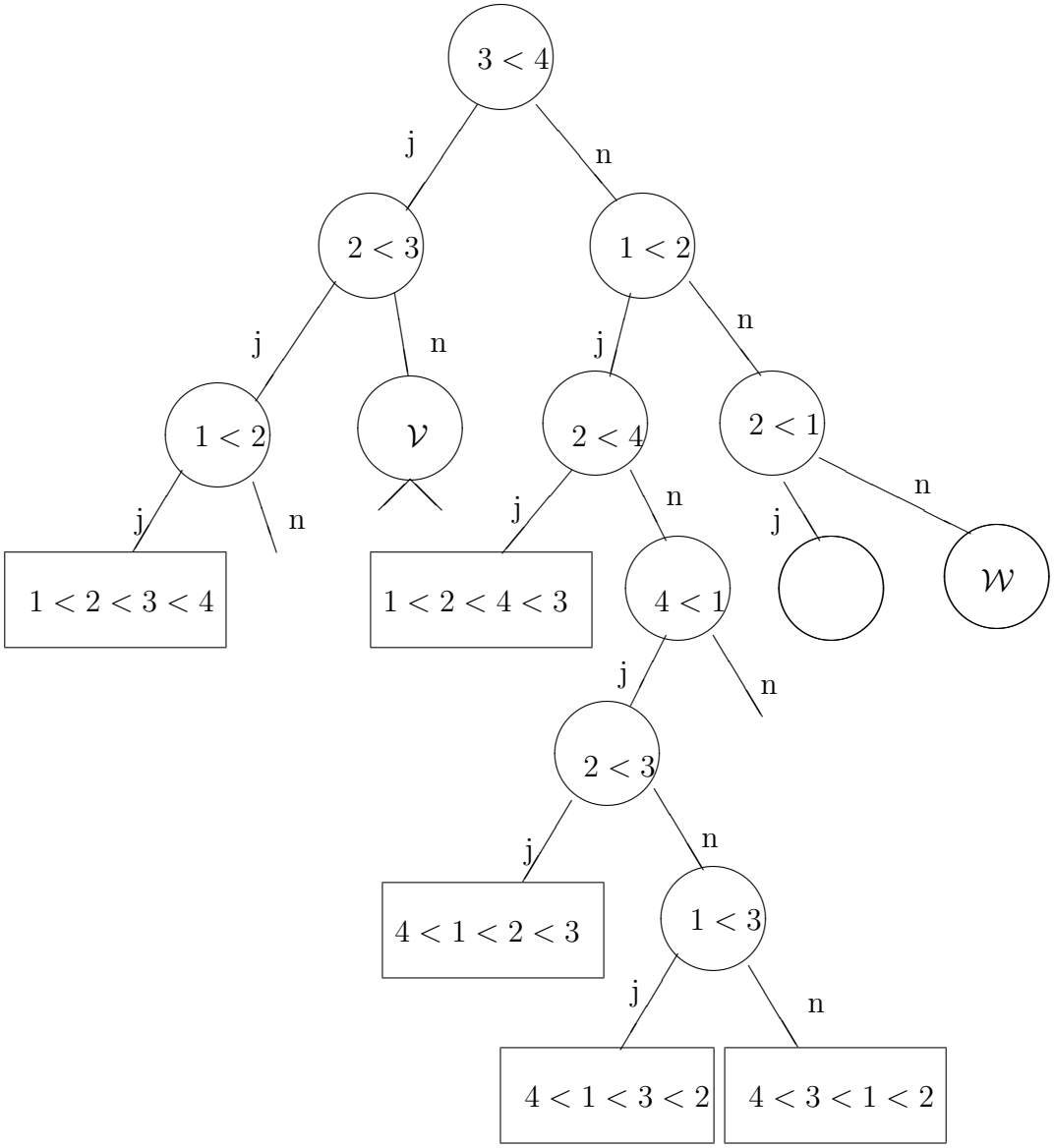


Abbildung 3.1.: Entscheidungsbaum

3.1. Verschärfung des Hauptsatzes 1. „Lineares Modell“

$a_i < a_j \leftrightarrow a_j - a_i > 0 \leftrightarrow \exists\, d > 0 : a_j - a_i = d \leftrightarrow \exists\, d > 0 : a_j - a_i - d = 0$

Von Interesse ist, ob $g(x_1, \ldots, x_n) < 0$, wobei $g(x_1, \ldots, x_n) = c_1 x_1 + \cdots + c_n x_n + d$ mit $c_1, \ldots, c_n, d$ als Konstanten und $x_1, \ldots, x_n$ als Variablen. Da Variablen nur in linearer Form vorkommen, nennt man dies „Lineares Modell“.

Satz 2
Im linearen Modell gilt für das Sortieren eine untere Zeitschranke von $\Omega(n \log n)$.

Der Beweis erfolgt dabei über die Aufgabenstellung „ε-closeness“. Denn, wenn die Komplexität der ε-closeness in einer unsortierten Folge $\Omega(n \log n)$ und in einer sortierten Folge $\mathrm{O}(n)$ ist, dann muß die Komplexität des Sortierens $\Omega(n \log n)$ sein.

Beim Elementtest ist eine Menge $\mathbb{M}$, $\mathbb{M} \subseteq \mathbb{R}^n$ gegeben, sowie ein variabler Vektor $(x_1, \ldots, x_n)$. Es wird getestet, ob $(x_1, \ldots, x_n) \in \mathbb{M}$, wobei $\mathbb{M}$ natürlich fest ist.

Bei der ε-closeness sieht die Fragestellung etwas anders aus. Als Eingabe sind wieder n reelle Zahlen $a_1, \ldots, a_n$ und $\varepsilon > 0$ gegeben. Von Interesse ist aber, ob es zwei Zahlen in der Folge gibt, deren Abstand kleiner oder gleich ε ist.

Viel kürzer ist die mathematische Schreibweise: $\exists\, i, j\ (1 \leq i, j \leq n) : |a_i - a_j| < \varepsilon$?

Trivalerweise ist bei einer bereits sortierten Eingabe ε-closeness in $\mathrm{O}(n)$ entscheidbar. Dazu wird einfach nacheinander geprüft, ob $|a_1 - a_2| < \varepsilon$ oder $|a_2 - a_3| < \varepsilon$ oder $\ldots |a_{n-1} - a_n| < \varepsilon$.

Satz 3
Wenn ε-closeness die Komplexität $\Omega(n \log n)$ hat, so auch das Sortieren.

Satz 4
Die Menge $\mathbb{M}$ habe m Zusammenhangskomponenten. Dann gilt, dass jeder Algorithmus im linearen Modell im worst case mindestens $\log m$ *Schritte braucht, wenn er den Elementtest löst.*

Beweis 2
Jeder Algorithmus A im linearen Modell liefert einen Entscheidungsbaum mit Knoten, in denen für alle möglichen Rechenabläufe gefragt wird, ob $g(x_1, \ldots, x_n) < 0$ ist. Jetzt genügt es zu zeigen, daß der Entscheidungsbaum mindestens soviele Blätter hat, wie die Menge $\mathbb{M}$ Zusammenhangskomponenten. Mit dem eben bewiesenen folgt, daß dies äqivalent zu einer Höhe des Entscheidungsbaumes von $\log(card(\mathbb{M})) = \log m$ ist. Nun sei b ein Blatt des Baumes.

Definition 13
$E(b) := \{\vec{x} \in \mathbb{R}^n : \textit{Alg. A landet bei der Eingabe von } \vec{x} = (x_1, \ldots, x_n) \textit{ in Blatt } b\}$

$$
\begin{array}{rcl}
 & g(x_1, \quad \ldots \quad , x_n) < 0\ ? & \\
 & \text{ja } / \qquad \backslash \text{ nein} & \\
g(x_1, \ldots, x_n) < 0 & & g(x_1, \ldots x_n) \geq 0
\end{array}
$$

Nach Definition des linearen Modells sind diese Mengen $E(b)$ jeweils Durchschnitt von offenen und abgeschlossenen affinen Teilräumen

$$\{(x_1, \ldots, x_n) \in \mathbb{R}^n : g(x_1, \ldots, x_n) < 0\}$$
$$\{(x_1, \ldots, x_n) \in \mathbb{R}^n : g(x_1, \ldots, x_n) \geq 0\}$$

Die Mengen $E(b)$ sind konvex und insbesondere zusammenhängend. Für alle Punkte a in $E(b)$ trifft der Algorithmus dieselbe Entscheidung; folglich gilt entweder $E(b) \subset \mathbb{M}$ oder $E(b) \cap \mathbb{M} = \emptyset$.

Sei $\mathcal{V}$ ein beliebiger Knoten. Genau die Inputs einer konvexen und zusammenhängenden Menge führen dorthin (= der Durchschnitt von Halbräumen).

Definition 14
$\mathbb{K}$ *ist konvex* $\leftrightarrow \forall\, p,\, q : p \in \mathbb{K} \wedge q \in \mathbb{K} \rightarrow \overline{pq} \subseteq \mathbb{K}$

Nun gilt $\mathbb{R}^n = \bigcup_{b\ ist\ Blatt} E(b)$ also $\mathbb{M} = \mathbb{R}^n \cap \mathbb{M} = \bigcup_{b\, Blatt} E(b) \cap \mathbb{M} = \cdots = \bigcup_{b \in \mathbb{B}} E(b)$ für eine bestimmte Teilmenge $\mathbb{B}$ aller Blätter des Entscheidungsbaumes. Weil jede Menge $E(b)$ zusammenhängend ist, kann diese Vereinigung höchstens $|\mathbb{B}|$ viele Zusammenhangskomponenten haben. Die Anzahl aller Blätter des Entscheidungsbaumes kann nicht kleiner sein als $|\mathbb{B}|$, sie beträgt also mindestens m.

q. e. d.

Die Komplexität des Elementtests ist also $\Omega(\log m)$

Satz 5
Die Komplexität der ε-closeness ist $\Omega(n \log n)$.

Beweis 3
ε-closeness ist ein Elementtest-Problem!

$\mathbb{M}_{\varepsilon^i} := \{(a_1, \ldots, a_n) \in \mathbb{R}^n | \forall\, i \neq j : |a_i - a_j| \geq \varepsilon\}$ ist ein spezielles Elementtestproblem.

π sei eine Permutation $\pi(1, \ldots, n)$, dann ist $\mathbb{M}(\pi) := \{(a_1, \ldots, a_n) \in \mathbb{M} | a_{\pi(1)} < a_{\pi(2)} < \ldots a_{\pi(n)}\}$

Satz 6
*Die Zusammenhangskomponenten von $\mathcal{M}_\varepsilon$ sind die Mengen $\mathcal{M}_\pi$ (*Beweis Übung*)*

Folgerung: Jeder Entscheidungsbaum im linearen Modell erfordert $\log(n!)$ *Schritte im worst case. (*$\Rightarrow \Omega(n \log n)$*)*

Folgerung: Sortieren hat im linearen Modell die Komplexität $\Omega(n \log n)$

Angenommen das stimmt nicht, dann existiert ein schnelleres Sortierverfahren. Dann benutze das für den Input von ε-closeness und löse ε-closeness schneller als in $\Omega(n \log n)$ → dann exisitiert für ε-closeness ein Verfahren von geringerer Komplexität als $\Omega(n \log n)$ → Widerspruch zur Voraussetzung → Behauptung

(relativer Komplexitätsbeweis)

q. e. d.

Hilfssatz zur Berechnung der Pfadlängen (Ungleichung von Kraft):
Sei t_i die Pfadlänge für alle m Pfade eines Binärbaumes von der Wurzel zu den m Blättern, dann gilt:

$$\sum_{i=1}^{m} 2^{-t_i} \leq 1$$

Beweis induktiv über m
$m = 1$: trivial
$m \geq 2$: Dann spaltet sich der Baum in maximal zwei Bäume auf, deren Pfadlängen m_1 und m_2 um eins geringer sind, als die von m. Die neuen Längen werden mit $\mathrm{t}_{1,1} \ldots \mathrm{t}_{1,m_1}$ und $\mathrm{t}_{2,1} \ldots \mathrm{t}_{2,m_2}$ bezeichnet.

Nach Voraussetzung gilt:

$$\sum_{i=1}^{m_1} 2^{-t_{1i}} \leq 1 \text{ und} \sum_{i=1}^{m_1} 2^{-t_{2i}} \leq 1$$

Jetzt werden Pfadlängen um 1 größer, dann gilt für t_{1i} (und analog für t_{2i}):

$$2^{(-t_{1i}+1)} = 2^{-t_{1i}-1} = 2^{-1}\, 2^{-t_{1i}}$$

Für T folgt also:

$$\sum_{i=1}^{m} 2^{-t_j} = 2^{-1} \left(\sum_{i=1}^{m_1} 2^{-t_{1i}} + \sum_{i=1}^{m_2} 2^{-t_{2i}} \right) \leq 2^{-1}(1+1) = 1$$

q.e.d

Hilfssatz:

$$\frac{1}{m} \sum_{i=1}^{m} t_i \geq \log m$$

Dabei gelten die selben Bezeichnungen wie oben.
Beweis:

$$\frac{1}{m} \sum_{i=1}^{m} 2^{-t_i} \geq \sqrt[m]{\pi_{i=1}^{m}\, 2^{-t_i}} = \sqrt[m]{2^{-t_i}\, 2^{-t_2} \cdot \ldots \cdot 2^{-t_m}} = \sqrt[m]{2^{-t_1 - \ldots - t_m}} = 2^{-\frac{1}{m} \sum_{i=1}^{m} t_i}$$

$$\Rightarrow m \leq 2^{\frac{1}{m} \sum_{i=1}^{m} t_i}$$

$$\text{Also gilt für die Pfadlänge: } \log m \leq \frac{1}{m} \sum_{i=1}^{m} t_i$$

q.e.d.

Satz 7
Hauptsatz über das Sortieren

Das Sortieren auf der Basis von Schlüsselvergleichen kostet bei Gleichwahrscheinlichkeit aller Permutationen der Eingabe $\theta(n \log n)$ (mit den schnellstmöglichen Algorithmen).

Beweis 4
Annahme $m > n!$

$$\Omega(n \log n) \ni \log n! \leq \frac{1}{n!} \sum_{i=1}^{m} t_i \leq \frac{1}{m} \sum_{i=1}^{m} t_i$$

Da wir bereits die untere Schranke bewiesen haben, muß $\frac{1}{m} \sum_{i=1}^{m} t_i \geq \frac{1}{n!} \sum_{i=1}^{m} t_i$ gelten, also $\frac{1}{m} \geq \frac{1}{n!}$ und damit $m \geq n!$ sein.

Falls $m > n!$, dann ist aber $\frac{1}{m} < \frac{1}{n!}$.

Widerspruch zur Voraussetzung (untere Schranke) $\Rightarrow (m \geq n! \wedge \neg(m > n!)) \rightarrow m = n!$

q.e.d.

3.2. Quicksort

Bei QUICKSORT handelt es sich ebenfalls um ein Teile-und-Hersche-Verfahren.

Eingabe ist wieder ein Feld A=$(a_0, \ldots, a_\mathrm{n})$, die Werte stehen dabei in den Plätzen $a_1, \ldots, a_\mathrm{n}$. Dabei dient $a_0 := -\infty$ als Markenschlüssel, als sogenanntes Sentinel-Element (siehe auch MERGESORT) und v ist das Pivotelement.

Quicksort

```
1)Quicksort(l,r)
  if r>l then
    i:=Partition(l,r)
    Quicksort(l,i-1)
    Quicksort(i+1,r)

2)Quicksort(l,r)
  if r>l then
  v:= a[r]
  i:= l-1
  j:= r
  repeat
    repeat
      i:=i+1
    until a[i]>=v
    repeat
      j:=j-1
    until a[j]<=v
    t:=a[i]
    a[i]:=a[j]
    a[j]:=t
  until j<=i
Quicksort(l,i-1)
Quicksort(i+1,r)
```

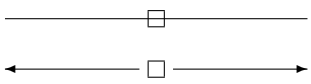
Was passiert bei 2)?

1. Es wird von links nach rechts gesucht, bis ein Element größer v ist
2. Es wird von rechts nach links gesucht, bis ein Element kleiner v ist
3. Dann werden die beiden Elemente vertauscht, falls sie sich treffen, so kommt v an diese Stelle

Beispiel (getauscht werden die **Fetten**):

2	**7**	8	9	0	1	5	**3**	6	4
2	3	**8**	9	0	**1**	5	7	6	4
2	3	1	**9**	**0**	8	5	7	6	4
2	3	1	0	**9**	8	5	7	6	**4**

4 =: Pivotelement



□=:Pivotelement

Am Ende sind alle Zahlen, die kleiner bzw. größer sind als 4, davor bzw. dahinter angeordnet. Jetzt wird der Algorithmus rekursiv für die jeweiligen Teilfolgen aufgerufen.

Nach einer anderen Methode von Schöning (nachzulesen in [4]) sieht die Eingabe 2 7 8 9 0 1 5 7 6 4 nach dem ersten Durchlauf so aus: 2 0 1 4 7 8 9 5 7 6

3.2.1. Komplexität des QUICKSORT-Algorithmus'

T(n) sei die Anzahl der Vergleiche, im besten Fall „zerlegt" das Pivotelement die Sequenz in zwei gleich große Teile und es gilt die bekannte Rekurrenz

$$T(n) = 2T\left(\frac{n}{2}\right) + n \Rightarrow O(n \log n)$$

Im schlechtesten Fall, nämlich bei bereits sortierten Folgen, ist aber $T(n) \in \Omega(n^2)$

Satz 8
QUICKSORT *benötigt bei gleichwahrscheinlichen Eingabefolgen im Mittel etwa* $1,38n \log n$ *Vergleiche.*

Beweis 5
$n = 1$:

$$T(1) = T(0) = 0$$

$n \geq 2$:

$$T(n) = (n+1) + \frac{1}{n} \sum_{1 \leq k \leq n} [T(k-1) + T(n-k)] = (n+1) + \frac{2}{n} \sum_{1 \leq k \leq n} T(k-1)$$

Zur Lösung dieser Rekurrenz wird zuerst die Summe beseitigt, dies sollte jeder selbst nachvollziehen. Die ausführliche Rechnung steht im Anhang B.3. Es ergibt sich $\frac{T(n)}{n+1} = \frac{T(n-1)}{n} + \frac{2}{n+1}$ und Einsetzen der Rekurrenz führt zu:

$$\frac{T(n)}{n+1} = \frac{T(n-1)}{n} + \frac{2}{n+1} = \frac{T(n-2)}{n-1} + \frac{2}{n} + \frac{2}{n+1} = \dots$$

$$\dots = \frac{T(2)}{3} + \sum_{3 \leq k \leq n} \frac{2}{k+1} \approx 2 \sum_{k=1}^{n} \frac{1}{k} \approx 2 \int_1^n \frac{1}{x} dx = 2 \ln n$$

$$T(n) = 2n \ln n \approx 1,38n \log n$$

q.e.d.

3.3. Auswählen (Sortieren)

Wie in vorangegangenen Vorlesungen besprochen wurde, braucht QUICKSORT bestenfalls $O(n \log n)$ Zeit und im worst case, wenn beim „Teile und Herrsche“ - Verfahren die Länge der beiden Teilfolgen sehr unterschiedlich ist, $O(n^2)$ Zeit.

Um diese benötigte Zeit zu verringern, versuchen wir nun, einen Algorithmus zu finden, mit dem wir den worst case ausschließen können.

Die Idee hierbei ist, ein Element zu finden, daß in der sortierten Folge ungefähr in der Mitte stehen wird und diesen sogenannten „Median“ als Pivot-Element für das „Teile und Herrsche“ - Verfahren bei QUICKSORT zu verwenden.

Wie kompliziert ist es nun, diesen Meridian zu ermitteln? Dazu ist zuerst zu sagen, daß bei einer geraden Anzahl von Elementen zwei Elemente als Meridian in Frage kommen. Hierbei ist es allerdings egal, ob man sich für das kleinere oder für das größere Element entscheidet.

Definition 15
Sei eine Folge $A = (a_1, ..., a_n)$ gegeben, wobei alle a_i die Elemente einer linear geordneten Menge sind. Dann ist der Rang eines Elements a_i im Feld A definiert als $Rang(a_i : A) := |\{x | x \in A : x \leq a_i\}|$.

Sei $A = (9, -5, 2, -7, 6, 0, 1)$, dann ist $Rang(1 : A) := 4$ (4 Elemente von A sind ≤ 1)

Sei nun A sortiert, also $A_{sortiert} = (a_{\pi(1)}, ..., a_{\pi(n)})$, dann gilt für das Element c mit $Rang(c : A) = k$ für $1 \leq k \leq n$, daß $c = a_{\pi(k)}$, das heißt: $a_{\pi(1)} \leq ... \leq a_{\pi(k)} \leq ... \leq a_{\pi(n)}$. Die ursprüngliche Reihenfolge paarweise gleicher Elemente wird hierbei beibehalten. Im weiteren wird eine Kurzschreibweise für den Rang verwendet, $a_{(k)}$ ist das Element mit dem Rang k. Ein Feld A mit n Elementen wird kurz mit A^n bezeichnet.

Definition 16
Der Median von A ist demzufolge:
$a_{(\lfloor \frac{n}{2} \rfloor)}$ *(Element vom Rang* $\lfloor \frac{n}{2} \rfloor$ *bei n Elementen)*

Hierbei kann allerdings, wie oben schon erwähnt, auch $a_{(\lceil \frac{n}{2} \rceil)}$, also die nächstgrößere ganze Zahl, als Rang festgelegt werden.

Unter Selektion verstehen wir eine Vorbehandlung, die als Eingabe $A := (a_1, ..., a_n)$ erhält und unter der Bedingung $1 \leq k \leq n$ als Ausgabe das Pivot-Element $a_{(k)}$ liefert.

Dieser Algorithmus zur Selektion (brute force) besteht nun aus folgenden zwei Schritten:

1. SORT A: $a_{\pi(1)}, a_{\pi(2)}, ..., a_{\pi(n)}$ (braucht $O(n \log n)$ Zeit)
2. Ausgabe des k-ten Elementes

Für die Selektion wird die „Median-der-Mediane-Technik“ verwendet.

3.3.1. Algorithmus $SELECT(A^n, k)$:

Wähle beliebige feste Ganzzahl Q (z.B. 5)

Schritt 1: If $|A| \leq Q$ Then sortiere A (z.B. mit Bubblesort)

Ausgabe des k-ten Elementes (da Anzahl konstant: $O(1)$)

Else Zerlege A in $\frac{|A|}{Q}$ Teilfolgen der maximalen Länge Q

Schritt 2: Sortiere jede Teilfolge und bestimme den Median m_i (dauert $O(n)$ Zeit)

Schritt 3: $SELECT(\{m_1, m_2, ..., m_{\frac{|A|}{Q}}\}, \frac{|A|}{2Q})$, Ausgabe m

Schritt 4: Definition von drei Mengen:

$A_1 := \{x \in A | x < m\}$

$A_2 := \{x \in A | x = m\}$

$A_3 := \{x \in A | x > m\}$

Schritt 5: If $|A_1| \geq k$ Then $SELECT(A_1, k)$

Elseif $|A_1| + |A_2| \geq k$ Then Output m

Else $SELECT(A_3, k - (|A_1| + |A_2|))$

Zeitanalyse:

zu Schritt 1: $max\{O(1), O(n)\}$
zu Schritt 2: $O(1)$ für jedes Sortieren der $O(n)$ Teilfolgen
zu Schritt 3: $T(\frac{n}{Q})$
zu Schritt 4: $O(n)$
zu Schritt 5: $T(\frac{n}{Q})$

Seien die Mediane m_j aller Teilfolgen sortiert. Dann ist m, der Median der Mediane, der Median dieser Folge. Wieviele Elemente aller Teilfolgen sind größer oder gleich m?

$\frac{|A|}{2Q}$ Mediane der Teilfolgen sind größer oder gleich m und für jeden dieser $\frac{|A|}{2Q}$ Mediane sind $\frac{Q}{2}$ Elemente „seiner“ Teilfolge größer oder gleich m. Damit sind mindestens $\frac{|A|}{2Q} \cdot \frac{Q}{2} = \frac{|A|}{4}$ Elemente größergleich m.

$\Rightarrow |A_1| \leq \frac{3}{4}|A| \Rightarrow T(n) = O(n) + T(\frac{n}{Q}) + T(\frac{3}{4}n)$ und $T(n) = O(n) \Longleftrightarrow \frac{n}{Q} + \frac{3}{4}n < n$

Dies trifft für $Q \geq 5$ zu. Damit hat $\textsc{Selection}(A^n, k)$ die Komplexität O(n).

3.3.2. Algorithmus S-Quicksort(A)

Der worst case wird durch die Bestimmung des Medians ausgeschlossen, die die Komplexität O(n) hat. Damit gilt die Rekurrenz $T(n) = 2T(\frac{n}{2}) + O(n)$ und der Algorithmus funktioniert immer in $O(n \log n)$ Zeit.

S-Quicksort

```
1 If |A| = 2 und a_1 < a_2 Then Tausche a_1 und a_2
2 Elseif |A| > 2 Then
3   Select (A, |A|/2) ⟶ m
4   A_1 := {x ≤ m, x ∈ A, sodaß |A_1| = ⌈|A|/2⌉}
5   A_2 := {x ≥ m, x ∈ A, sodaß |A_2| = ⌈|A|/2⌉}
6   S-Quicksort(A_1)
7   S-Quicksort(A_2)
8 End
```

3.4. Heapsort

Abstrakte Datentypen wurden bereits auf Seite 8 definiert, ebenso der ADT **Dictionary**. Nun wird ein weiterer Datentyp, der **Heap** vorgestellt. Der **Heap** wird in der Literatur oft auch mit **Priority Queue** bezeichnet. Er findet z.B. bei der Verwaltung von Druckaufträgen Anwendung.

Definition 17 (Heap)
Der abstrakte Datentyp, der das Quintupel der Operationen MakeHeap, Insert, Max *und* ExtractMax *unterstützt heißt* ***Heap*** *(oft auch* ***Priority Queue*** *genannt).*

Hierbei handelt es sich um einen sogenannten Max-Heap. Analog kann ein Min-Heap gebildet werden, indem statt des Maximums immer das Minimum genommen wird (bzw. statt $\geq$ immer $\leq$ im Algorithmus).

Definition 18 (In place)
Ein Verfahren heißt ***In place****, wenn es zur Bearbeitung der Eingabe unabhängig von der Eingabegröße nur zusätzlichen Speicherplatz konstanter Größe benötigt.*

Auf der Basis der genannten und später erläuterten Operationen mit Heaps kann ein effizientes Sortierverfahren namens Heapsort definiert werden, das in place funktioniert.

Das wird so erreicht, daß ein direkter Zugriff auf das Feld besteht und das Sortieren an Ort und Stelle und ohne Verwendung weiteren Speicherplatzes vorgenommen werden kann. Des weiteren wird garantiert, dass n Elemente in $O(n \log n)$ Schritten sortiert werden, unabhängig von den Eingabedaten.

Die Vorteile von MergeSort ($O(n \log n)$) und InsertionSort (In place) werden also vereint.

Definition 19 (Binärer Heap)
Ein Binärer Max-Heap ist ein spezieller Binärbaum (wird im Folgenden nochmals definiert) mit den Eigenschaften, daß der Wert jedes Knotens jeweils größergleich den Werten seiner Söhne ist und daß außer dem Level mit der Höhe 0 alle Level voll besetzt sein müssen. Jeder Level wird von links beginnend aufgefüllt. Hat also ein Blatt eines

Heaps die Höhe 1 im gesamten Heap, so haben auch alle rechts davon stehenden Blätter genau dieselbe Höhe.

Dies ist äquivalent dazu, daß für eine Folge $F = k_1, k_2, ..., k_n$ von Schlüsseln für alle i mit $2 \leq i \leq n$ die Beziehung $k_i \leq k_{\lfloor \frac{i}{2} \rfloor}$ gilt (Heap-Eigenschaft), wobei kein Eintrag im Feld undefiniert sein darf.

Wegen der Speicherung in einem Array werden die Söhne auch als Nachfolger bezeichnet. Wird ein binärer Heap in einem Array gespeichert, so wird die Wurzel des Baums an Position 1 im Array gespeichert. Die beiden Söhne eines Knotens an der Arrayposition i werden an den Positionen 2i und 2i+1 gespeichert. Und mit der Heap-Eigenschaft gilt $k_i \geq k_{2i}$ und $k_i \geq k_{2i+1}$ für alle i mit $2i < n$

Anschaulicher ist vielleicht die Vorstellung als Binärbaum

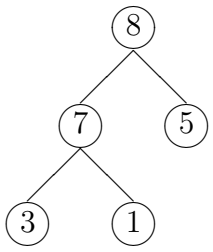


Abbildung 3.2.: Binärer Max-Heap

Das korrespondierende Array wäre A=(8,7,5,3,1).

Um auf den Vater, den linken oder den rechten Sohn eines Knotens i zuzugreifen, genügen die folgenden einfachen Berechnungen:

Ziel	Berechnung
Vater(i)	$\lfloor \frac{i}{2} \rfloor$
LSohn(i)	$2i$
RSohn(i)	$2i + 1$

Für die folgenden Überlegungen sind noch weitere Definitionen nützlich.

Definition 20 (Graphentheoretische Definition eines Binärbaumes)
*Ein **Binärbaum** ist ein Baum, bei dem jeder Knoten vom Grad höchstens 3 ist. Ein Knoten mit höchstens Grad 2 kann dabei als Wurzel fungieren. Ein solcher Knoten existiert immer (im Extremfall ist er ein Blatt, hat also den Grad 1).*

Definition 21 (Ein vollständiger Binärbaum)
*Ein **vollständiger Binärbaum** hat zusätzlich die Eigenschaften, daß genau ein Knoten den Grad 2 besitzt und alle Blätter die gleiche Höhe haben. In diesem Fall wird immer der Knoten vom Grad 2 als Wurzel bezeichnet.*

Satz 9
In einem vollständigen (im strengen Sinne) Binärbaum der Höhe h gibt es 2^h Blätter und $2^h - 1$ innere Knoten.

Der Beweis ist mittels vollständiger Induktion über h möglich.

Satz 10
Der linke Teilbaum eines Binärheaps mit n Knoten hat maximal $\frac{2n}{3}$ Knoten.

Beweisidee: Berechne erst wieviele Knoten der rechte Teilbaum hat. Dann benutze dies um die Knotenanzahl des linken Teilbaumes zu berechnen. Rechne dann das Verhältnis der Knotenanzahlen zueinander aus.

Beweis 6
Da der Grenzfall von Interesse ist, wird von einem möglichst asymmetrischen Heap ausgegangen. Sei also der linke Teilbaum voll besetzt und habe der rechte genau einen kompletten Höhenlevel weniger. Noch mehr Disbalance ist aufgrund der Heap-Eigenschaft nicht möglich. Dann ist der rechte Teilbaum ebenfalls voll besetzt, hat allerdings einen Level weniger als der linke.

Sei also i die Wurzel eines solchen Baumes mit der Höhe l und j der linke Sohn von i. Dann ist j Wurzel des linken Teilbaumes. Nun bezeichne $K(v)$ die Anzahl der Knoten im Baum mit der Wurzel v. Dann soll also gelten $\frac{K(j)}{K(i)} \leq \frac{2}{3}$. Nach Voraussetzung gilt $K(j) = 2^l - 1$ und $K(i) = 2^l - 1 + 2^{l-1} - 1 + 1 = 3 \cdot 2^{l-1} - 1$, es folgt

$$\frac{K(j)}{K(i)} = \frac{2^l - 1}{3 \cdot 2^{l-1} - 1} \leq \frac{2^l}{3 \cdot 2^{l-1}} = \frac{2}{3}$$

q.e.d.

Satz 11
In einem n-elementigen Binärheap gibt es höchstens $\left\lceil \frac{n}{2^{h+1}} \right\rceil$ Knoten der Höhe h.

Definition 22 (Höhe eines Baumes)
Die Höhe eines Knoten ϑ ist die maximale Länge des Abwärtsweges von ϑ zu einem beliebigen Blatt (also die Anzahl der Kanten auf dem Weg).

Beweisidee: Die Knoten der Höhe 0 sind die Blätter. Dann wird von unten beginnend zur Wurzel hochgelaufen und dabei der Exponent des Nenners wird immer um eins erhöht

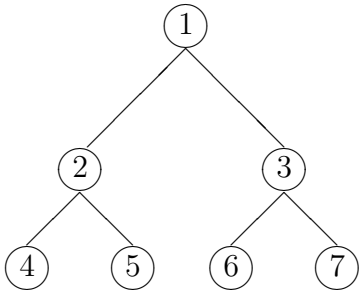
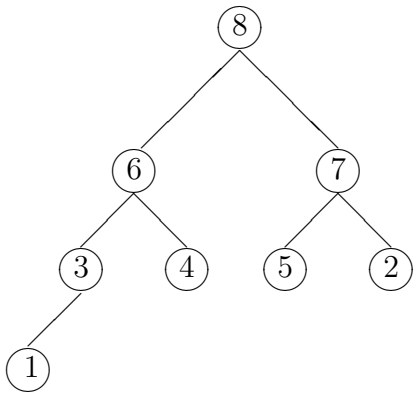


Abbildung 3.3.: Binärer Min-Heap

Aus der Heap-Eigenschaft folgt, daß das Maximum in der Wurzel steht. Sei nun also F=(8, 6, 7, 3, 4, 5, 2, 1) gegeben. Handelt es sich dabei um einen Heap?

Ja, da $F_i \geq F_{2i}$ und $F_i \geq F_{2i+1}$, für alle i mit $5 \geq i \geq 1$, da $8 \geq 6 \wedge 8 \geq 7 \wedge 6 \geq 3 \wedge 6 \geq 4 \wedge 7 \geq 5 \wedge 7 \geq 2 \wedge 3 \geq 1$.

Dieser Max-Heap sieht dann grafisch wie folgt aus:



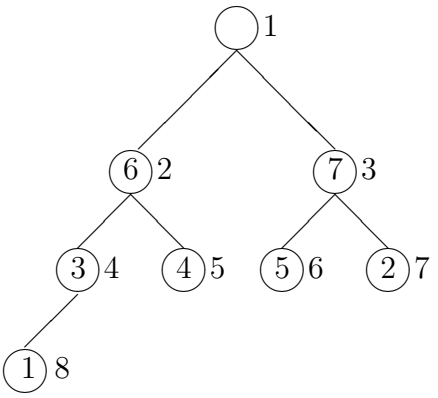
Sei nun eine Folge von Schlüsseln als Max-Heap gegeben und die Ausgabe sortiert in absteigender Reihenfolge gewünscht; für einen Min-Heap müssen die Relative „kleiner“ und „größer“ ausgetauscht werden. Um die Erläuterung einfacher zu halten, wird von dem größeren Sohn gesprochen, nicht von dem Knoten mit dem größeren gespeicherten Wert. Genauso werden Knoten und nicht Werte vertauscht. Dies ist allerdings formal falsch!

Für den ersten Wert ist dies einfach, da das Maximum bereits in der Wurzel steht. Dies läßt sich ausnutzen, indem der erste Wert in die Ausgabe aufgenommen wird und aus dem Heap entfernt wird, danach wird aus den verbleibenden Elementen wieder ein Heap erzeugt. Dies wird solange wiederholt, bis der Heap leer ist. Da in der Wurzel immer das Maximum des aktuellen Heaps gespeichert ist, tauchen dort die Werte der Größe nach geordnet auf.

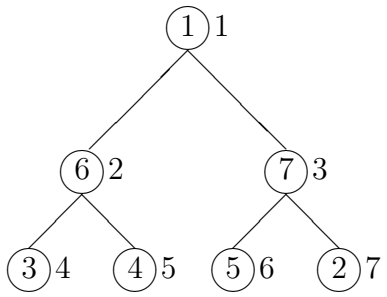
Der neue Heap wird durch Pflücken und Versickern des Elements mit dem größten Index erreicht. Dazu wird die Wurzel des anfänglichen Heaps geleert. Das Element mit dem größten Index wird aus dem Heap gelöscht (gepflückt) und in die Wurzel eingesetzt. Nun werden immer die beiden Söhne des Knotens verglichen, in dem der gepflückte Wert steht. Der größere der beiden Söhne wird mit dem Knoten, in dem der gepflückte Wert steht, vertauscht.

Der gepflückte Wert sickert allmählich durch alle Level des Heaps, bis die Heap-Eigenschaft wieder hergestellt ist und wir einen Heap mit n-1 Elementen erhalten haben.

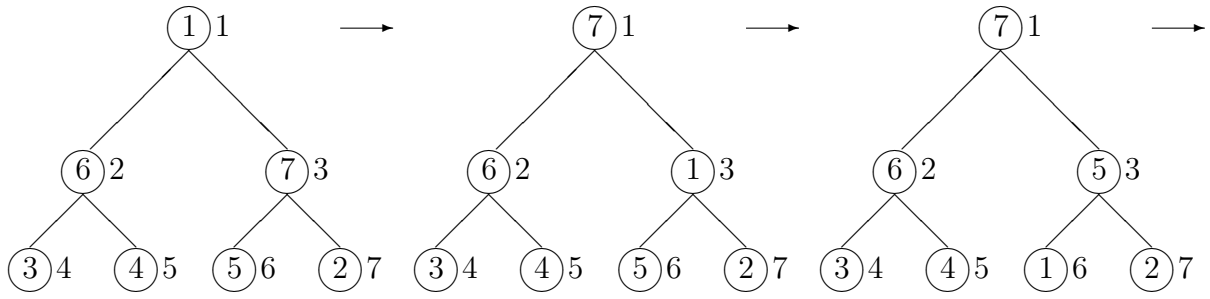
Im obigen Beispiel heißt das also: Wenn man nun die Wurzel (hier: 8) wieder entfernt, wandert die 1 nach oben und in diesem Moment ist es kein Binärheap. D.h. es muß ein neuer Heap erzeugt werden und dies geschieht unter zu Zuhilfenahme von Heapify (Versickern). In der grafischen Darstellung wurde die Position im Array rechts neben die Knoten geschrieben:



→ entnehme die Wurzel



→ setze 1 an die Wurzel



→ Heapify (Versickern) der „1“
Als Algorithmus:

HEAPIFY(A)

```
1  l:= LSohn(i)
2  r:= RSohn(i)
3  if l <= Heapsize[A] und A[l]=Succ(A[i])
4    then Max:= l
5    else Max:= i
6  if r <= Heapsize[A] und A[r]=Succ(A[Max])
7    then Max:= r
8  if Max != i
9    then tausche A[i] und A[Max]
10     HEAPIFY(A,Max)
```

INPUT: F bzw. A in einen Heap überführen
Komplexität der einzelnen Algorithmen
Für HEAPIFY gilt die Rekurrenz T(n)=T(2/3n)+O(1), damit gilt T(n) $\in$ O($\log n$)
Für BUILD-HEAP ist dies etwas komplizierter Sei h die Höhe eines Knotens und c eine Konstante größer 0, dann gilt:

BUILD-HEAP(A)

```
1 Heapsize[A]:= Länge[A]
2 for i:= ⌊Länge(A/2)⌋ down to 1
3    HEAPIFY(A,i)
```

HEAPSORT(A)

```
1 BUILD-HEAP(A)
2 for i:= Laenge[A] down to 2
3   do tausche A[1] und A[i]
4     Heapsize[A]:= Heapsize[A]-1
5     HEAPIFY(A,1)
```

$$\sum_{h=0}^{\lfloor(\log n)\rfloor} \left\lceil \frac{n}{2^{h+1}} \right\rceil \cdot ch \in O\left(n \cdot \sum_{h=0}^{\lfloor\log n\rfloor} \frac{n}{2^{h+1}}\right) \in O\left(n \cdot \underbrace{\sum_{h=0}^{\infty} \frac{h}{2^{h+1}}}_{=2}\right) = O(n)$$

Damit kostet BUILD-HEAP nur $O(n)$.

Damit hat HEAPSORT die Komplexität $O(n \log n)$. In jedem der $O(n)$ Aufrufe von BUILD-HEAP braucht HEAPIFY $O(\log n)$ Zeit.

3.4.1. Priority Queues

In der Literatur wird die Begriffe Heap und Priority Queue (Prioritätswarteschlange) oftmals synonym benutzt. Hier wird begrifflich etwas unterschieden und Heaps werden für die Implementierung von solchen Warteschlangen benutzt. Auch die Bezeichnungen für BUILDHEAP ist nicht einheitlich, in einigen Büchern wird stattdessen MAKEHEAP oder MAKE verwendet.

Definition 23
Der abstrakte Datentyp, der die Operationen MAKE-HEAP, INSERT, MAX, EXTRACT-MAX *und* INCREASE KEY *unterstützt wird* ***Priority Queue*** *genannt.*

Wie bei einem Heap kann natürlich auch hier immer mit dem Minimum gearbeitet werden, die Operationen wären dann BUILDHEAP, INSERT, MIN, EXTRACTMIN und DECREASE KEY

Behauptung: Binary Heaps unterstützen Warteschlangen.

Increasekey sorgt dafür, das nach Änderungen die Heapbedingung wieder gilt. Es vertauscht solange ein Element mit dem Vater, bis das Element kleiner ist.

INCREASEKEY kostet $O(\log n)$, dazu siehe auch Skizze 3.4. Damit wird INCREASEKEY unterstützt und wir wenden uns der Operation INSERT zu. Dabei wandert key solange nach oben, bis die Heap-Eigenschaft wieder gilt. Damit kostet auch INSERT $O(\log n)$.

ExtractMax(A)

```
1 if heap-size[A] < 1
2     then Fehler
3 Max := A[1]
4 A[1] := A[heap-size[A]]
5 heap-size[A] := heap-size[A]-1
6 Heapify(A, 1)
7 Ausgabe Max
```

Der Heap wird mittels A an Extractmax übergeben. Diese Funktion merkt sich die Wurzel (das Element mit dem größten Schlüssel). Dann nimmt es das letzte im Heap gespeicherte Blatt und setzt es als die Wurzel ein. Mit dem Aufruf von Heapify() wird die Heap-Eigenschaft wieder hergestellt. Das gemerkte Wurzel wird nun ausgegeben. Falls die Anzahl der Elemente in A (heap-size) kleiner als 1 ist, wird eine Fehlermeldung ausgelöst.

IncreaseKey(A, x, k)

```
1 if key[x] > k
2     then Fehler "Alter Schlüsselwert ist größer"
3 key[x] := k
4 y := x
5 z := p[y]
6 while z ≠ NIL und key[y] > key[z]
7     do tausche key[y] mit key[z]
8     y := z
9     z := p[y]
```

Die übergebene Variable x ist der Index im Array und k ist der neue Schlüsselwert. Der Vater des Knotens x in der Baumstruktur wird mit p[x] bezeichnet.

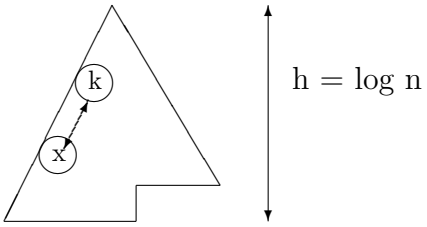


Abbildung 3.4.: Heapify

Ist der im Vaterknoten gespeicherte Wert größer als der im Sohn gespeicherte, vertauscht Increasekey die beiden Werte.

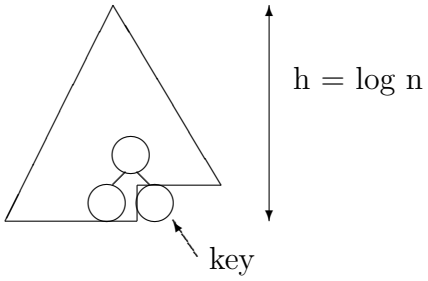


Abbildung 3.5.: Insert

Damit unterstützen binäre Heaps:

- BuildHeap $O(1)$ - (im Sinne von MakeHeap = Schaffen der leeren Struktur)
- Insert $O(log\ n)$
- Max $O(1)$
- ExtractMax $O(log\ n)$
- IncreaseKey $O(log\ n)$

Somit ist die Behauptung erfüllt.

Ein interesantes Anwendungsbeispiel ist, alle $\leq$ durch $\geq$ zu ersetzen. Also Max durch Min, ExtractMax durch ExtractMin und IncreaseKey durch Decreasekey zu ersetzen und nur Insert zu belassen.

3.5. Dijkstra

Problem:

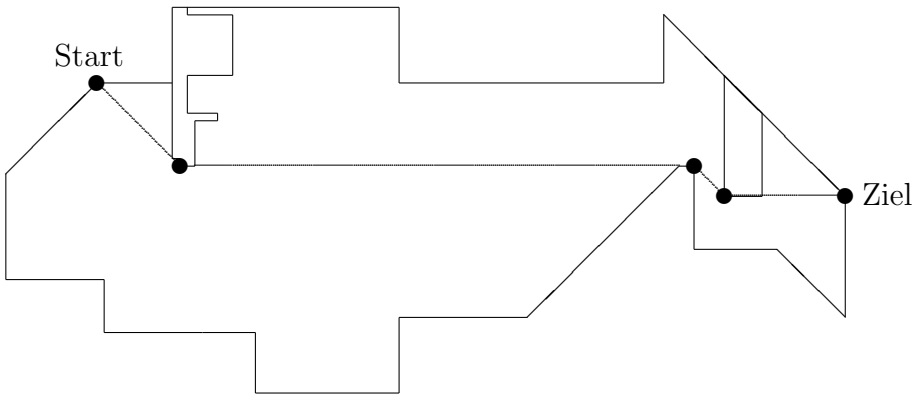


Abbildung 3.6.: Kürzesten Weg finden

Sehr wichtig für die Informatik sind Graphen und darauf basierende Algorithmen. Viele anscheinend einfache Fragestellungen erfordern recht komplexe Algorithmen.

So sei z.B. ein unwegsames Gelände mit Hindernissen gegeben und der kürzeste Weg dadurch herauszufinden. Für die algorithmische Fragestellung ist es völlig egal, ob es sich um ein Gelände oder eine andere Fläche handelt. Deswegen wird soweit abstrahiert, daß aus der Fläche und den Hindernissen Polygone werden. Doch unverändert lautet die Fragestellung, wie man hier den kürzesten Weg finden kann. Es ist zu erkennen, das dies nur über die Eckpunkte (von Eckpunkt zu Eckpunkt) zu bewerkstelligen ist. Dabei darf natürlich nicht der zugrunde liegende Graph verlassen werden. Ist eine aus Strecken zusammengesetzte Linie der kürzeste Weg?

Definition 24 (Visibility Graph)
M = Menge der Ecken = Menge der Polygonecken. $a, b \in M \rightarrow \overline{ab}$ ist Kante des Graphen $\leftrightarrow \overline{ab}$ ganz innerhalb des Polygons liegt.

Satz 12
Der kürzeste Pfad verläuft entlang der Kanten des Sichtbarkeitsgraphen (Lorenzo-Parez).

Problem ALL-TO-ONE SHORTEST PATHS

One (In Abbildung 3.7 ist das der Punkt a) ist der Startpunkt und von allen anderen Punkten wird der kürzeste Weg dahin berechnet. Die Gewichte an den Kanten sind dabei immmer ≥ 0.

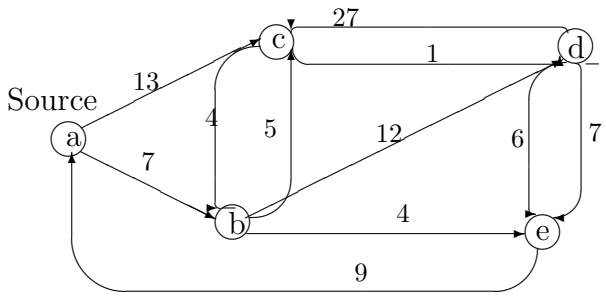


Abbildung 3.7.: Graph mit gerichteten Kanten

Zur Lösung des Problems verwenden wir den Algorithmus von Dijkstra (1959).

Paradigma: Es ist ein Greedy (gieriger) Algorithmus. (einfach formuliert: ich denk nicht groß nach, ich nehm einfach den kürzesten Weg um von Punkt a weg zu kommen)

Start: V ist die Menge der Knoten, W ist die Menge der erreichten Knoten.

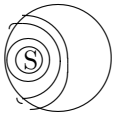


Abbildung 3.8.: Menge W wird aufgeblasen

Nun zum eigentlichen Algorithmus:

DIJKSTRA

```
1 for all v in V do d(v) := + ∞
2 d(s) := 0; W := ∅
3 Initalisiere Struktur V (mit d(v), v in V)
4 while V \ W ≠ ∅ do
5   v := Min(V \ W); ExtractMin(V \ W);
6   W := W∪{v}
7   for all w in Succ(v) \ W do
8     if d(r) + l(vw) < d(w)
9       then DecreaseKey(w, d(v) + l(vw))
```

Der Nachweis der Korrektheit dieses Algorithmus ist sehr schwer und soll nicht Gegenstand dieses Skriptes sein.

Ein einfaches Beispiel soll seine Funktion veranschaulichen. Die verwendeten Knoten und Kanten entsprechen denen aus Abbildung 3.7. Jede Zeile der Tabelle entspricht einer Ausführung des Rumpfes der while Schleife (ab Zeile 5) und zeigt die Werte, welche die verschiedenen Variablen annehmen. Aus Gründen der Übersicht werden nur 3 Spalten augeführt, der / tritt deshalb nochmal als Trennhilfe auf:

$(v, d(v)) : v \in W$ / $(v, d(v)) : v \in V \cup W$	$v = min(V - W)$ / $SUCC(v)$	$v = min(V - W)$, $w \in SUCC(v) \setminus (W \cup v)$, $(w, l(v\vec{w})$ / $min(d(w), d(v) + l(v\vec{w}))$
$\emptyset$ / $(a, 0), (b, \infty), (c, \infty), (d, \infty), (e, \infty)$	a / c, b	$v = a : (c, 13), (6, 7)$ / $(c, 13), (b, 7)$
$\{(a, 0)\}$ / $(b, 7), (c, 13), (d, \infty), (e, \infty)$	b / c, d, e	$v = b : w \in \{c, d, e\} \setminus \{a, b\}$, $(c, 5), (d, 12), (e, 4)$ / $(c, 12), (d, 19), (e, 11)$
$(a, 0), (b, 7)$ / $(e, 11), (c, 12), (d, 19)$	e / a, d	$v = e, w \in \{a, d\} \setminus \{a, b, e\}$ $= \{d\}, (d, 7)$ / $min(19, 18) : (d, 18)$
$(a, 0), (b, 7), (e, 11)$ / $(c, 12), (d, 18)$ nächste Zeile $(d, 13)$	c / b, d	

Nun interessiert uns natürlich die Komplexität des Algorithmus. Dazu wird die jeweilige Rechenzeit der einzelnen Zeilen betrachtet, wobei $|V| = n$ und $|Knoten| = m$.

1. $O(|V|) = O(n)$
2. $O(n)$
3. meist in $O(1)$
4. nicht zu beantworten
5. nicht zu beantworten
6. $O(1)$ im Regelfall (es kann auch komplexer sein, da es darauf ankommt, wie W verwaltet wird)
7.
8. $O(1)$
9. die Komplexität von DECREASEKEY ist auch nicht zu beantworten

Die Laufzeit hängt wesentlich davon ab, wie die Priority Queue verwaltet wird.

1. Möglichkeit: V wird in einem Array organisiert $\rightarrow O(n^2)$
2. Möglichkeit: Binärer Heap für V\W, dann geht EXTRACTMIN in $O(n\ log\ n)$ und DECREASEKEY in $O(m\ log\ n)$ $\rightarrow O((m+n)\ log\ n)$
3. Möglichkeit: V in einen Fibonacci-Heap $\rightarrow O((n \log n) + m)$

3.6. Counting Sort

Einen ganz anderen Weg zum Sortieren von Zahlen beschreitet Counting Sort. Es funktioniert, unter den richtigen Bedingungen angewendet, schneller als in $O(n \log n)$ und basiert nicht auf Schlüsselvergleichen.

COUNTING SORT

```
1 for i := 0 to k do
2  C[i]:= 0;
3  for j := 1 to Laenge[A] do
4    C[A[j]] := C[A[j]]+1;
5  for i := 1 to k do
6    C[i] := C[i] + C[i-1];
7  for j := Laenge[A] downto 1 do
8    B[C[A[j]]] := A[j];
9    C[A[j]] := C[A[j]]-1
```

Das Sortierverfahren COUNTING SORT belegt eventuell auf Grund der Verwendung von zwei zusätzlichen Feldern B und C sehr viel Speicherplatz. Es funktioniert ohne die Verwendung von Schlüsselvergleichen. Es wird zu Begin ein Zähler (Feld C) erzeugt dessen Größe abhängig ist von der Anzahl der möglichen in A enthaltenen Zahlen (Zeile 1 und 2). Für jeden möglichen Wert in A, also für jeden Wert des Zahlenraumes, wird eine Zelle des Feldes reserviert. Seien die Werte in A z.B. vom Typ Integer mit 16 Bit. Dann gibt es 2^{16} mögliche Werte und das Feld C würde für jeden der 65536 möglichen Werte eine Zelle erhalten; dabei wird C[0] dem ersten Wert des Zahlenraums zugeordnet, C[1] dem zweiten Wert des Zahlenraumes usw. Anschließend wird die Anzahl der in A enthaltenen Elemente schrittweise in C geschrieben. Bei mehrfach vorhandenen Elementen wird der entsprechende Wert erhöht, daher kommt auch der Name Counting Sort (Zeile 3 und 4). Nun werden die Adressen im Feld berechnet. Dazu wird die Anzahl eines Elemente mit der Anzahl eines Vorgängerelements addiert um die entsprechende Anzahl im Ausgabefeld frei zu halten (Zeile 5 und 6). Zum Schluss wird die richtige Reihenfolge durch zurücklaufen des Arrays A und der Bestimmung der richtigen Stelle, mit Hilfe von C, in B geschrieben. Bei COUNTING SORT handelt es sich um ein stabiles Sortierverfahren.

Definition 25 (Stabile Sortierverfahren)
*Ein Sortierverfahren heißt **stabil**, falls mehrfach vorhandene Elemente in der Ausgabe in der Reihenfolge auftauchen, in der sie auch in der Eingabe stehen.*

3.6.1. Counting Sort an einem Beispiel

Input: $A = (1_a, 3, 2_a, 1_b, 2_b, 2_c, 1_c)$ und $k = 3$

Output: $B = (1_a, 1_b, 1_c, 2_a, 2_b, 2_c, 3)$, $C = (0, 0, 3, 6)$

Ablauf:

Zeile	Feld C	Erläuterung
nach 1 und 2	$< 0, 0, 0, 0 >$	Zähler wird erzeugt und 0 gesetzt
nach 3 und 4	$< 0, 3, 3, 1 >$	Anzahl der Elemente wird “gezählt”
nach 5 und 6	$< 0, 3, 6, 7 >$	Enthält Elementzahl kleiner gleich i
nach 9	$< 0, 0, 3, 6 >$	$A[i]$ werden in B richtig positioniert

3.6.2. Komplexität von Counting Sort

Die Zeitkomplexität von Counting Sort für einen Input von A^n mit $k \in$ O(n) ist $T(n) \in O(n) \cup O(k)$.

Satz 13
Falls $k \in O(n)$, so funktioniert Counting Sort *in O(n).*

Die Stärke von Counting Sort ist gleichzeitig auch Schwäche. So ist aus dem obigen bereits ersichtlich, daß dieses Verfahren z.B. zum Sortieren von Fließkommazahlen sehr ungeeignet ist, da dann im Regelfall riesige Zählfelder erzeugt werden, die mit vielen Nullen besetzt sind, aber trotzdem Bearbeitungszeit (und Speicherplatz!) verschlingen.

3.7. Weitere Sortieralgorithmen

Außer den hier aufgeführten Sortieralgorithmen sind für uns noch Bucket Sort und Radix Sort von Interesse.

4. Einfache Datenstrukturen: Stapel, Bäume, Suchbäume

Bevor mit den einfachen Datenstrukturen begonnen wird, noch eine Bemerkung zum Begriff des abtrakten Datentyps (siehe Seite 8). Auch dieser Begriff wird leider nicht immer einheitlich verwendet. Mal wird er wie eingangs definiert oder als Menge von Operationen, dann wieder wie in der folgenden Definition oder noch abstrakter, wie z.B. in [1], wo ein ADT als Signatur vereinigt mit Axiomen für die Operationen definiert wird.

Definition 26 (ADT)
Ein Abstrakter Datentyp ist eine (oder mehrere) Menge(n) von Objekten und darauf definierten Operationen

Einige Operationen auf ADT's

Q sei eine dynamische Menge

Operation	Erläuterung
INIT(Q)	Erzeugt eine leere Menge Q
STACK-EMPTY	Prüft ob der Stack leer ist
PUSH(Q,x),INSERT(Q,x)	Fügt Element x in Q ein (am Ende)
POP(Q,x),DELETE(Q,x)	Entfernt Element x aus Q (das Erste x was auftritt)
POP	Entfernt letztes Element aus Q
TOP	Zeigt oberstes Element an
SEARCH(Q,x)	Sucht El. x in Q (gibt erstes Vorkommende aus)
MIN(Q)	Gibt Zeiger auf den kleinsten Schlüsselwert zurück
MAXQ)	Gibt Zeiger auf den größten Schlüsselwert zurück
SUCC(Q,x)	Gibt Zeiger zurück auf das nächst gr. El. nach x
PRED(Q,x)	Gibt Zeiger zurück auf das nächst kl. El. nach x

4.1. Binäre Suchbäume

Definition 27 (Binärer Suchbäume)
Ein binärer Suchbaum ist ein Binärbaum folgender Suchbaumeigenschaft: Sei x Knoten des binären Suchbaumes und Ahn vom Knoten y. Falls der Weg von x nach y über den linken Sohn von x erfolgt, ist $key[\,y\,] \leq key[\,x\,]$. *Andernfalls istt* $key[\,y\,] > key[\,x\,]$.

Definition 28
Ein Suchbaum heißt ***Blattsuchbaum****, falls die Elemente der dynamischen Menge im Gegensatz zum normalen Suchbaum nur in den Blättern gespeichert werden.*

4.1.1. Beispiel für einen binären Suchbaum

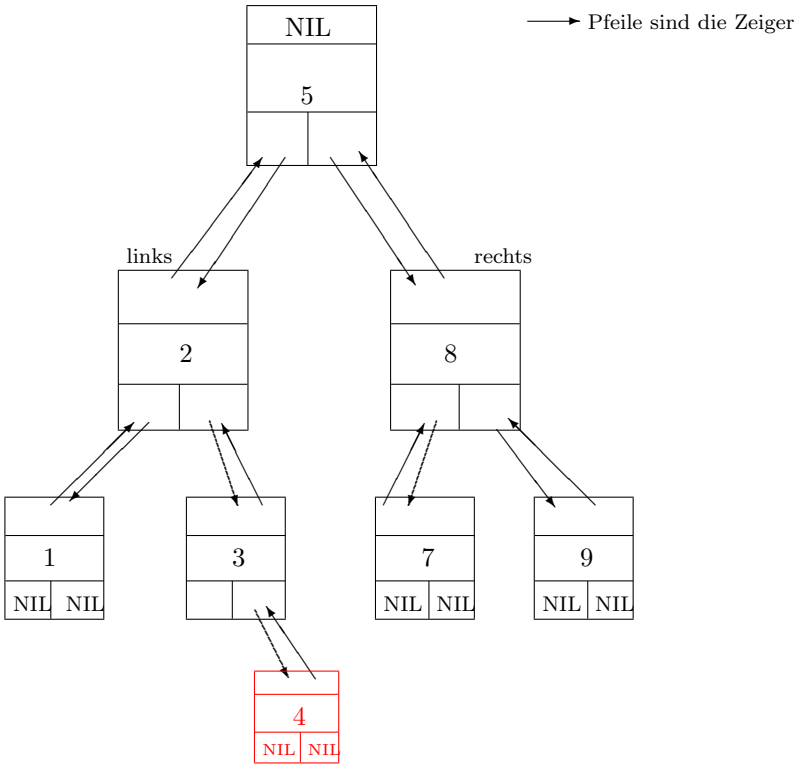


Abbildung 4.1.: Binärer Suchbaum

- Rotes Blatt wird erst durch INSERT hinzugefügt
- $Q = \{5, 2, 1, 3, 8, 7, 9\}$
- SEARCH(Q,4):
  Beim Suchen wird der Baum von der Wurzel an durchlaufen und der zu suchende Wert mit dem Wert des Knoten verglichen. Ist der zu suchende Wert kleiner als der Knotenwert wird im linken Teilbaum weitergesucht. Ist er größer wie der Knotenwert wird im rechten Teilbaum weitergesucht. Zurückgegeben wird der zuerst gefundene Wert. Ist das Element nicht im Suchbaum enthalten, wird NIL bei Erreichen eines Blattes zurückgegeben.
  Im Beispiel wird der Suchbaum in der Reihenfole 5, 2, 3 durchlaufen und dann auf Grund des Fehlens weiterer Knoten mit der Rückgabe von NIL verlassen.

- INSERT(Q,4):
  Beim Einfügen wird das einzufügende Element mit dem jeweiligen Element des aktuellen Knotens verglichen. Begonnen wird dabei in der Wurzel. Ist das einzufügende Element größer, wird im Baum nach rechts gegangen, ist es kleiner, nach links. Ist in ein Blatt erreicht, wird dann, die Suchbaumeigenschaft erhaltend, entweder rechts oder links vom Blatt aus eingefügt.
  Im Beispiel wird der Baum in der Reihenfolge 5, 2, 3 durchlaufen und die 4 dann rechts von der 3 als neues Blatt mit dem Wert 4 eingefügt.

- Nach Einfügen: $Q = \{\, 5, 2, 1, 3, 4, 9, 7, 8 \,\}$

4.1.2. Operationen in binären Suchbäumen

TREE-SEARCH

```
1 if x=NIL or k=key[x]
2    then Ausgabe x
3 if k<key[x]
4    then Ausgabe TREE-SEARCH(li[x],k)
5    else Ausgabe TREE-SEARCH(re[x],k)
```

Bei TREE-SEARCH wird der Baum von der Wurzel aus durchlaufen. Gesucht wird dabei nach dem Wert k. Dabei ist x der Zeiger, der auf den Wert des aktuellen Knotens zeigt. In den ersten beiden Zeilen wird der Zeiger zurückgegeben wenn ein Blatt erreicht oder der zu suchende Wert gefunden ist (Abbruchbedingung für Rekursion). In Zeile 3 wird der zu suchende Wert mit dem aktuellen Knotenwert verglichen und anschlïeßend in den Zeilen 4 und 5 entsprechend im Baum weitergegangen. Es erfolgt jeweils ein rekursiver Aufruf.
Die Funktion wird beendet wenn der Algorithmus in einem Blatt angekommen ist oder der Suchwert gefunden wurde.

Traversierung von Bäumen

TREEPOSTORDER(x)

```
1 if x ≠ NIL
2   then TREEPOSTORDER(li[x])
3        TREEPOSTORDER(re[x])
4        Print key[x]
```

Bei TREEPOSTORDER handelt es sich um einen rekursiven Algorithmus. Es wird zuerst der linke, dann der rechte Teilbaum und erst zum Schluß die Wurzel durchlaufen.

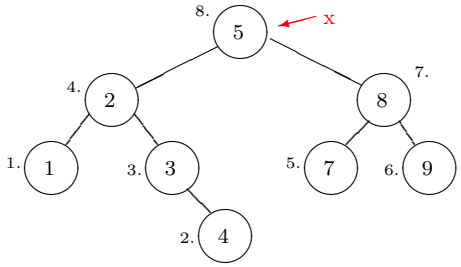


Abbildung 4.2.: TREEPOSTORDER(x)

- x ist der Zeiger auf dem Knoten
- Die Ausgabereihenfolge ist {1,4,3,2,7,9,8,5}

Treepreorder(x)

TREEPREORDER(x)

```
1  if x ≠ NIL
2     then Print key[x]
3          TREEPREORDER(li[x])
4          TREEPREORDER(re[x])
```

Beim ebenfalls rekursiven TREEPREORDER wird bei der Wurzel begonnen, dann wird der linke Teilbaum und anschließend der rechte Teilbaum durchlaufen.

Beispiel für TREEPREORDER

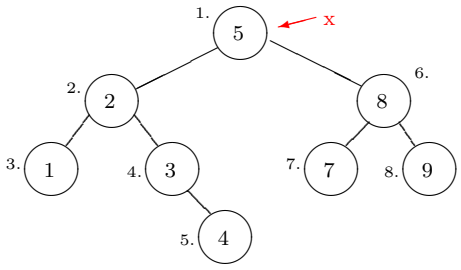


Abbildung 4.3.: TREEPREORDER

- x ist der Zeiger auf dem Knoten
- Die Ausgabenreihenfolge ist {5,2,1,3,4,8,7,9}

Treeinorder(x)

TREEINORDER(x)

```
1  if x ≠ NIL
2     then TREEINORDER(li[x])
3          Print key[x]
4           TREEINORDER(re[x])
```

Bei TREEINORDER wird zuerst der linke Teilbaum, dann die Wurzel und anschließend der rechte Teilbaum durchlaufen.

Beispiel für TREEINORDER

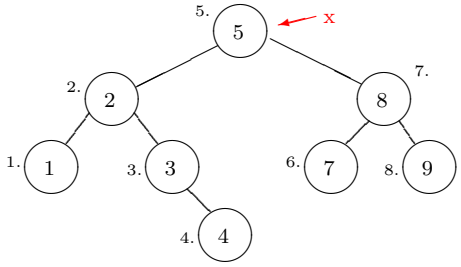


Abbildung 4.4.: TREEINORDER

- x ist der Zeiger auf dem Knoten
- Die Ausgabenreihenfolge ist {1,2,3,4,5,7,8,9}

Satz 14
*Bei gegebenem binären Suchbaum ist die Ausgabe mit allen drei Verfahren (*INORDER, PREORDER *und* POSTORDER*) in* $\Theta(n)$ *möglich.*

Folgerung: Der Aufbau eines binären Suchbaumes kostet $\Omega(n \log n)$ Zeit.

Tree-Successor(x)

MIN(x)

```
1  while li[x] ≠ NIL do
2    x:= li[x]
3  return x
```

MIN(x) liefert das Minimum des Teilbaumes, dessen Wurzel x ist.

TREE-SUCCESSOR(x)

```
1 if re[x] ≠ NIL
2    then return MIN(re[x])
3 y:=p[x]
4 while y ≠ NIL and x=re[y]
5    do x:=y
6       y:=p[y]
7 return y
```

Beim TREE-SUCCESSOR werden zwei Fälle unterschieden. Falls x einen rechten Teilbaum besitzt, dann ist der Nachfolger das Blatt, das im rechten Teilbaum am weitesten links liegt (MIN(x)). Besitzt x keinen rechten Teilbaum, so ist der successor y der Knoten dessen linker Sohn am nächsten mit x verwandt ist. Zu beachten ist dabei, daß sich der Begriff Nachfolger auf einen Knoten bezieht, der Algorithmus aber den Knoten liefert, dessen gespeicherter Wert im Baum Nachfolger des im ersten Knoten gespeicherten Wertes ist.

Die Operationen zum Löschen und Einfügen von Knoten sind etwas komplizierter, da sie die Baumstruktur stark verändern können und erhalten deswegen jeweils einen eigenen Abschnitt.

4.1.3. Das Einfügen

Beim TREE-INSERT werden zwei Parameter übergeben, wobei

- T der Baum ist, in dem eingefügt werden soll und
- z der Knoten, so daß
  - key[z] = v (einzufügender Schlüssel),
  - left[z] = NIL und
  - right[z] = NIL

  ist.

Erklärung: Bei diesem Einfügealgorithmus werden die neuen Knoten immer als Blätter in den binären Suchbaum T eingefügt. Der einzufügende Knoten z hat keine Söhne. Die genaue Position des Blattes wird durch den Schlüssel des neuen Knotens bestimmt. Wenn ein neuer Baum aufgebaut wird, dann ergibt der erste eingefügte Knoten die Wurzel. Der zweite Knoten wird linker Nachfolger der Wurzel, wenn sein Schlüssel kleiner ist als der Schlüssel der Wurzel und rechter Nachfolger, wenn sein Schlüssel größer ist als der Schlüssel der Wurzel. Dieses Verfahren wird fortgesetzt, bis die Einfügeposition bestimmt ist.

Anmerkungen dazu: Dieser Algorithmus zum Einfügen ist sehr einfach. Es finden keine Ausgleichs- oder Reorganisationsoperationen statt, so daß die Reihenfolge des

TREE-INSERT

```
1  y := NIL
2  x := root[T]
3  while ( x ≠ NIL ) do
4    y := x
5    if ( key[z] = key[x] )
6      then x := left[x]
7      else x := right[x]
8  p[z] := y
9  if ( y = NIL )
10   then root[T] := z
11   elseif ( key[z] < key[y] )
12     then left[y] := z
13     else right[y] := z
```

Die Laufzeit liegt in O(h), wobei h die Höhe von T ist.

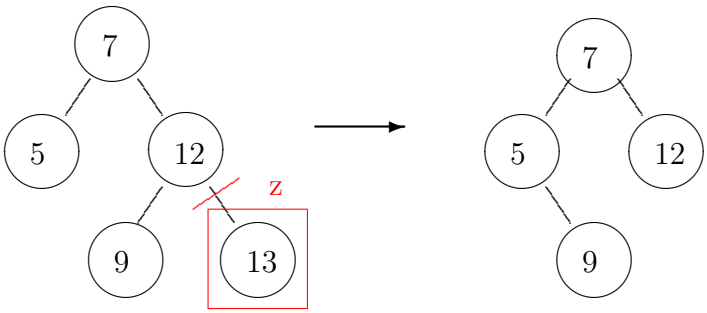
Einfügens das Aussehen des Baumes bestimmt, deswegen entartet der binäre Suchbaum beim Einfügen einer bereits sortierten Eingabe zu einer linearen Liste.

Da der Knoten immer in einem Blatt eingefügt wird, ist damit zu rechnen, daß im worst case das Blatt mit der größten Entfernung von der Wurzel genommen wird. Da dieses die Höhe h hat sind folglich auch h Schritte notwendig, um zu diesem Blatt zu gelangen.

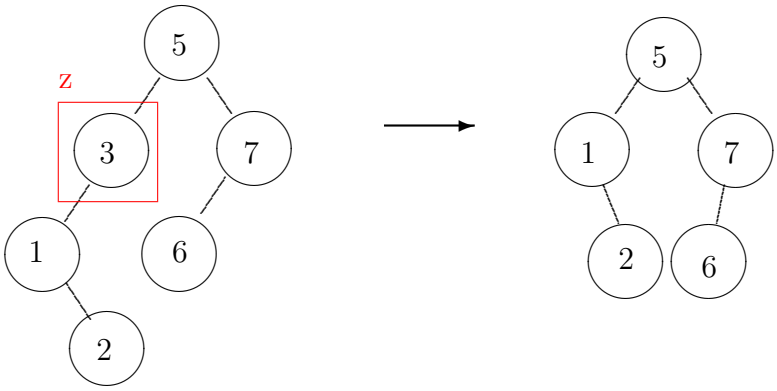
4.1.4. Das Löschen eines Knotens

Beim Löschen eine Knotens z in eiem binären Suchbaum müssen drei Fälle unterschieden werden:

- **1. Fall:** z hat keine Söhne
  Der Knoten kann gefahrlos gelöscht werden und es sind keine weiteren Operationen notwendig.

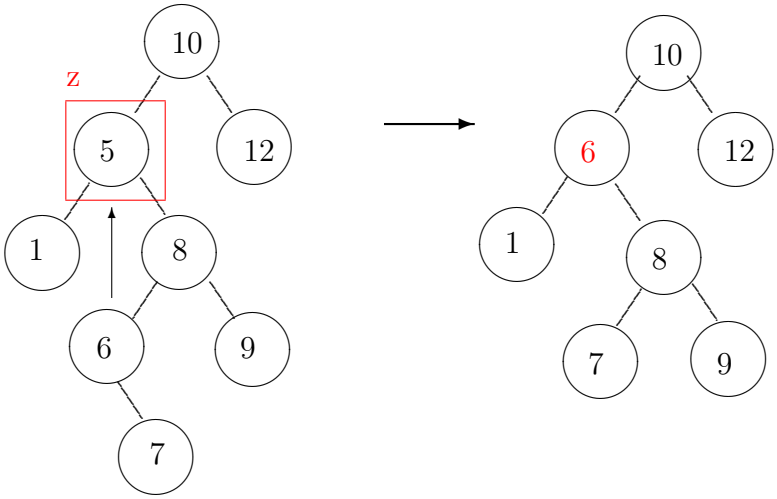


- **2. Fall:** z hat genau einen linken Sohn
  Der zu löschende Knoten wird entfernt und durch den Wurzelknoten des linken Teilbaums ersetzt.



- **3. Fall:** z hat genau einen rechten Sohn
  Analog dem 2. Fall.

- **4. Fall:** z hat zwei Söhne
  Problem: Wo werden die beiden Unterbäume nach dem Löschen von z angehängt?
  Lösung: Wir suchen den Knoten mit dem kleinsten Schlüssel im rechten Teilbaum von z. Dieser hat keinen linken Sohn, denn sonst gäbe es einen Knoten mit einem kleineren Schlüssel. Der gefunden Knoten wird mit dem zu löschenden Knoten z vertauscht und der aktuelle Knoten entfernt.



Auch beim Löschen (Tree-Delete) werden wieder zwei Parameter übergeben, dabei ist

- T der Baum und
- z der zu löschende Knoten

Rückgabewert ist der (tatsächlich) aus dem Baum entfernte Knoten, dies muss nicht z sein, (siehe 4. Fall)

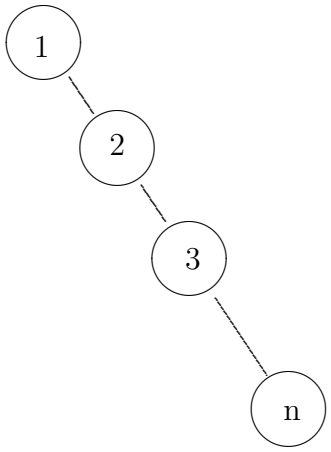
Im worst case wird Tree-Successor mit einer Laufzeit von O(h) einmal aufgerufen, andere Funktionsaufrufe oder Schleifen gibt es nicht.

TREE-DELETE

```
1                               if ( left[z] = NIL or right[z] = NIL )
2                                      then y .= z
3                                      else y := TREE-SUCCESSOR(z)
4                               if ( left[y] ≠ NIL )
5                                      then x := left[y]
6                                      else x := right[y]
7                               if ( x ≠ NIL )
8                                      then root[T] := x
9                                      else if ( y = left[p[y]] )
10                                            then left[p[y]] := x
11                                            else right[p[y]] := x
12                              if ( y ≠ z )
13                                     then key[z] := key[y]
14                              return y
```

Laufzeit liegt wieder in O(h), wobei h wieder die Höhe von T bezeichnet.

4.2. Binäre Suchbäume als Implementierung des ADT Wörterbuch



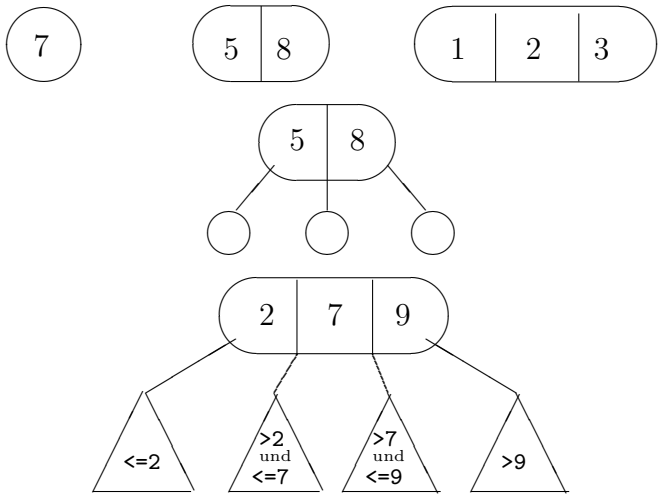
Wie bereits bei der Funktion TREE-INSERT beschrieben, kann eine ungünstige Einfügereihenfolge den Suchbaum zu einer linearen Liste entarten lassen. Deswegen sind allgemeine binäre Suchbäume nicht geeignet, den ADT Wörterbuch zu implementieren.

4.3. 2-3-4-Bäume

Definition 29 (2-3-4-Bäume)
2-3-4-Bäume *sind Bäume mit folgenden speziellen Eigenschaften:*

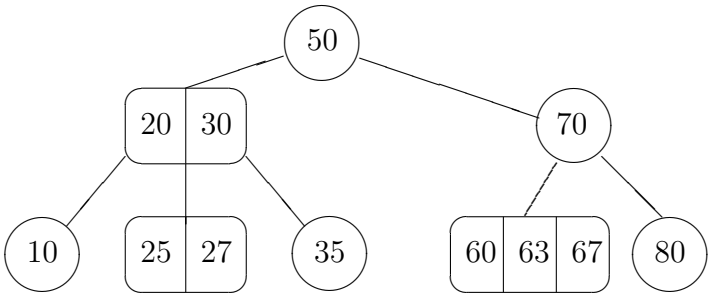
- *Jeder Knoten im Baum enthält einen, zwei oder drei Schlüssel, die von links nach rechts aufsteigend sortiert sind.*
- *Ein Knoten mit k Schlüsseln hat k+1 Söhne (oder er hat überhaupt keine:"Blatt") und wird als (k+1)-Knoten bezeichnet.*
- *Für Schlüssel im Baum gilt die verallgemeinerte Suchbaumeigenschaft.*
- *Alle Blätter haben den gleichen Abstand zur Wurzel.*

Zur Veranschaulichung dienen die folgenden Abbildungen (2-, 3- und 4-Knoten, ein Blatt und ein skizzierter möglicher Baum).



Bei einem 2-3-4-Baum ist die Anzahl der Knoten deutlich geringer als bei einem vergleichbaren binären Suchbaum. Damit ist die Zahl der besuchten Knoten bei einer Suche geringer. Daraus folgt, daß das Suchen nach einem Schlüssel in einem 2-3-4-Baum effizenter ist, als in einem vergleichbaren binären Suchbaum. Allerdings ist der Aufwand beim Einfügen und beim Löschen von Schlüsseln höher.

4.3.1. Vollständiges Beispiel und Suchen



- Erfolgreiche Suche nach 35
- Erfolglose Suche nach 69

Die Laufzeit für das Suchen liegt wieder in O(h), mit h als Höhe des Baumes.

A. Der Plane-Sweep-Algorithmus im Detail

Der Plane-Sweep-Algorithmus ist eine äußerst bekannte Methode zur Lösung von Mengenproblemen. Der primäre Gedanke besteht darin, eine vertikale Gerade, die *Sweepline*, von links nach rechts durch die Ebene zu schieben und dabei den Schnitt der Geraden mit der Objektmenge zu beobachten. Es ist ebenfalls möglich, statt einer vertikalen Sweepline eine horizontale zu verwenden und diese von oben nach unten über die Objekte zu führen. Einige Algorithmen benötigen mehrere Sweeps, durchaus in unterschiedliche Richtungen, um gewonnene Daten aus vorangegangenen Überstreichungen zu verarbeiten.

Dazu „merkt“ sich die Sweepline entsprechend ihrer aktuellen Position die horizontal schneidenden Segmente anhand deren y-Koordinaten. Bei Erreichen eines vertikalen Elements werden alle in der Sweepline vorhandenen y-Koordinaten ermittelt, die im y-Intervall des vertikalen Segments liegen.

Die Gleitgerade wird nicht stetig über die Objektmenge geführt, überabzählbar viele Teststellen wären die Folge. Lediglich x-Koordinaten, an denen ein Ereignis eintreten kann, werden in der Simulation berücksichtigt.

Im vorliegenden Fall sind das Anfangs- und End-x-Koordinaten von horizontalen Elementen sowie die x-Koordinate vertikal verlaufender Strecken. Diese Werte bilden die Menge der *event points*, sie kann häufig statisch implementiert werden.

Die Sweepline-Status-Struktur hingegen muß dynamisch sein, da sich deren Umfang und Belegung an jedem event point ändern kann.

Zur Beschreibung des Verfahrens ist es erforderlich, die nachfolgende Terminologie zu vereinbaren. Sie orientiert sich an der mathematischen Schreibweise:

$h = (x_1, x_2, y)$**:** horizontal verlaufendes Liniensegment mit dem Anfangspunkt (x_1, y) und dem Endpunkt (x_2, y)

$v = (x, y_1, y_2)$**:** vertikal verlaufendes Liniensegment mit dem Anfangspunkt (x, y_1) und dem Endpunkt (x, y_2)

t_i **für** $t = (x_1, \ldots, x_n)$**:** Zugriff auf die i-te Komponente des Tupels t, also $t_i = x_i$

$\pi_i(S)$**:** Für eine Tupelmenge S ist $\pi_i(S)$ die Projektion auf die i-te Komponente

Der Algorithmus nach Güting [1] gestaltet sich dann in dieser Weise:

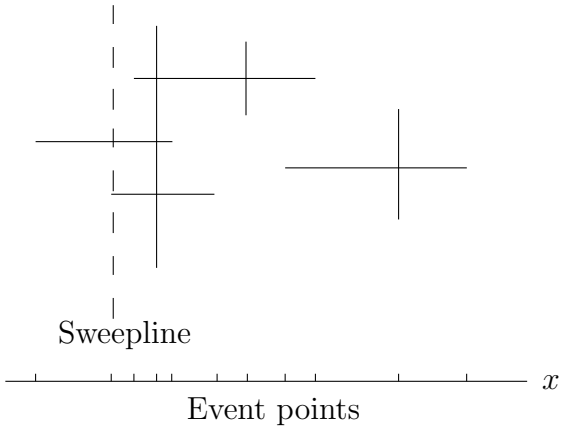


Abbildung A.1.: Liniensegmente mit Gleitgerade

algorithm SegmentIntersectionPS (H,V)
{Eingabe ist eine Menge horizontaler Segmente H und eine Menge vertikaler Segmente V, berechne mit Plane-Sweep die Menge aller Paare (h, v) mit $h \in H$ und $v \in V$ und h schneidet v}

1. Sei
$$\begin{aligned} S = \quad & \{(x_1, (x_1, x_2; y)) | (x_1, x_2, y) \in H\} \\ \cup \quad & \{(x_2, (x_1, x_2, y)) | (x_1, x_2, y) \in H\} \\ \cup \quad & \{(x, (x, y_1, y_2)) | (x, y_1, y_2) \in V\}; \end{aligned}$$
(S ist also eine gemischte Menge von horizontalen und vertikalen Segmenten, in der jedes horizontale Segment einmal anhand des linken und einmal anhand des rechten Endpunkts dargestellt ist. Diese Menge beschreibt die Sweep-Event-Struktur.)
Sortiere S nach der ersten Komponente, also nach x-Koordinaten.

2. Sei Y die Menge horizontaler Segmente, deren y-Koordinate als Schlüssel verwendet wird (Sweepline-Status-Struktur);
$Y := \emptyset$;
durchlaufe S: das gerade erreichte Objekt ist

   a) linker Endpunkt eines horizontalen Segments $h = (x_1, x_2, y)$:
   $Y := Y \cup \{(y, (x_1, x_2, y))\}$ (füge h in Y ein)

   b) rechter Endpunkt von $h = (x_1, x_2, y)$:
   $Y := Y \backslash \{(y, (x_1, x_2, y))\}$ (entferne h aus Y)

   c) ein vertikales Segment $v = (x, y_1, y_2)$:
   $A := \pi_2(\{w \in Y | w_0 \in [y_1, y_2]\})$; (finde alle Segmente in Y, deren y-Koordinate im y-Intervall von v liegt)
   gibt alle Paare in $A \times \{v\}$ aus

end SegmentIntersectionPS

B. Beispiele

B.1. Lösung Substitutionsansatz

Die Rekurrenz läßt sich natürlich auch anders als mit vollständiger Induktion lösen, aber da dies mit der entsprechenden Abschätzung ein gute Rechenübung ist, wird diese Methode benutzt.

Als Verdacht wird S(k)$\leq (k \log k)$ genommen.

$S(k) = 2\,S\left(\frac{k}{2}\right) + k \leq 2\,\left(\frac{k}{2} log\, \frac{k}{2}\right) + k = k\, log\, \frac{k}{2} + k = k\,(log\, k - log\, 2) + k = (k \log k)$

Die Rücksubstitution ist hier recht einfach und ergibt als Gesamtlaufzeit

$T(n) = T(2^k) = S(k) = O(k \log k) = O(log\, n\, log^{(2)}\, n)$.

B.2. Lösung Mastertheorem

$T(n) = 3\,T\left(\frac{n}{4}\right) + 2n \log n \Rightarrow a = 3,\ b = 4$ und $\log_b a < 1 \rightarrow f(n) \in \Omega(n^{log_4 3+\varepsilon})$ da $\forall n \in \mathbb{N} : n > 0 : n^{log_4 3} < 2n \log n$.

Es könnte sich also um den dritten Fall handeln, dazu muß aber noch ein $c < 1 : \forall_n^\infty : a\,f\left(\frac{n}{b}\right) \leq c\,f(n)$ existieren. Gibt es also ein $c < 1 : \forall_n^\infty : 3f\left(\frac{n}{4}\right) \leq cf(n)$? Ja, denn

$$\begin{aligned}
3f\left(\frac{n}{4}\right) &\leq c\,f(n) \\
3\left[2\left(\frac{n}{4}\right) log\left(\frac{n}{4}\right)\right] &\leq c\,2n \log n \\
\left(\frac{3}{4}\right) log\left(\frac{n}{4}\right) &\leq c \log n \\
\left(\frac{3}{4}\right) \log n - \log 4 &\leq c \log n \\
\left(\frac{3}{4}\right) \log n - 2 &\leq c \log n \\
\left(\frac{3}{4}\right) \frac{\log n}{\log n} - \frac{2}{\log n} &\leq c \\
\left(\frac{3}{4}\right) - \frac{2}{\log n} &\leq c
\end{aligned}$$

Da $lim_{n\rightarrow\infty} \frac{2}{\log n} = 0$ gilt die Ungleichung für jedes $c \in [\frac{3}{4}, \ldots, 1]$. Also handelt es sich um den dritten Fall des Mastertheorems und $\Rightarrow T(n) \in \Theta(f(n)) = \Theta(n \log n)$.

B.3. Aufwandsabschätzung Quicksort

$$T(n) = (n+1) + \frac{2}{n}\sum_{k=1}^{n} T(k-1) \rightarrow \tag{B.1}$$

$$T(n-1) = n + \frac{2}{n-1}\sum_{k=1}^{n-1} T(k-1) \tag{B.2}$$

$$(n-1)T(n-1) - (n-1)n = 2\sum_{k=1}^{n-1} T(k-1) \tag{B.3}$$

$$\begin{aligned}
n\,T(n) &= (n+1) + 2\sum_{k=1}^{n} T(k-1) \\
&= n(n+1) + 2\,T(n-1) + 2\sum_{k=1}^{n-1} T(k-1) && \text{B.3 einsetzen} \\
&= (n+1)n + 2\,T(n-1) + (n-1)T(n-1) - (n-1)n \\
&= [(n+1) - (n-1)]n + (2+n-1)\,T(n-1) \\
&= 2n + (n+1)T(n-1) \rightarrow
\end{aligned}$$

$$\frac{T(n)}{n+1} = \frac{T(n-1)}{n} + \frac{2}{n+1}$$

Literaturverzeichnis

[1] R. Güting; *Algorithmen und Datenstrukturen*, Teubner, 1992

[2] T.H. Cormen, C.E. Leierson, R.L. Rivest, C. Stein; *Introduction to Algorithms*, MIT Press, 2^{nd} edition, 2001

[3] R. Klein; *Algorithmische Geometrie*, Addison–Wesley, 1997

[4] Sedgewick; Algorithmen (mehrere unterschiedliche Fassungen verfügbar)