Algorithmen und Datenstrukturen

Prof. Dr. Hans-Dietrich Hecker

Wintersemester 2003/04

Inhaltsverzeichnis

Algorithmenverzeichnis

Vorwort

Dieses Dokument wurde als Skript für die auf der Titelseite genannte Vorlesung erstellt und wird jetzt im Rahmen des Projekts „Vorlesungsskripte der Fakultät für Mathematik und Informatik“ weiter betreut. Das Dokument wurde nach bestem Wissen und Gewissen angefertigt. Dennoch garantiert weder der auf der Titelseite genannte Dozent, die Personen, die an dem Dokument mitgewirkt haben, noch die Mitglieder des Projekts für dessen Fehlerfreiheit. Für etwaige Fehler und dessen Folgen wird von keiner der genannten Personen eine Haftung übernommen. Es steht jeder Person frei, dieses Dokument zu lesen, zu verändern oder auf anderen Medien verfügbar zu machen, solange ein Verweis auf die Internetadresse des Projekts `http://uni-skripte.lug-jena.de/` *enthalten ist.*

Diese Ausgabe trägt die Versionsnummer 2975 und ist vom 6. April 2010. Eine neue Ausgabe könnte auf der Webseite des Projekts verfügbar sein.

Jeder ist dazu aufgerufen, Verbesserungen, Erweiterungen und Fehlerkorrekturen für das Skript einzureichen bzw. zu melden oder diese selbst einzupflegen – einfach eine E-Mail an die Mailingliste `<uni-skripte@lug-jena.de>` *senden. Weitere Informationen sind unter der oben genannten Internetadresse verfügbar.*

Hiermit möchten wir allen Personen, die an diesem Skript mitgewirkt haben, vielmals danken:

- *Sebastian Oerding (Erfassung der einzelnen Mitschriften)*
- *Dr. Jana Grajetzki (Fachliche Korrektur)*

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. Dabei ist n die Anzahl der Rechteckkanten. Besteht der

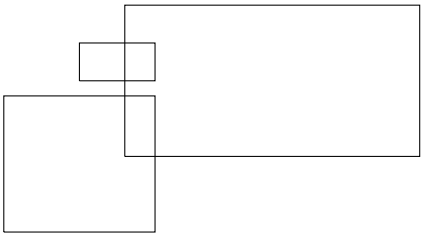


Abbildung 1.1.: Überlappen sich die drei Rechtecke?

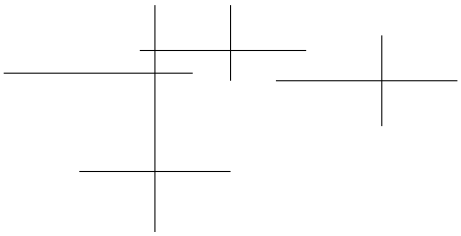


Abbildung 1.2.: Das Segmentschnitt-Problem

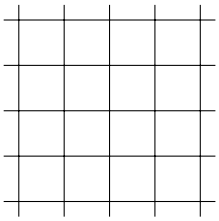


Abbildung 1.3.: Ungünstigste Kantenanordnung, $\frac{n^2}{4}$ Kreuzungspunkte

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.

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 (siehe Abbildung 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, wäre die Zahl der Prüfpunkte überabzählbar 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 so genannten **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 1.1 (Datenstruktur)
Die Art und Weise wie Daten problembezogen verwaltet und organisiert werden, nennt man **Datenstruktur**.

Definition 1.2 (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 1.3 (Abstrakter Datentyp (ADT))
Eine Datenstruktur, zusammen mit den von ihr unterstützten Operationen, heißt **abstrakter Datentyp** (**ADT**).

Definition 1.4 (Dictionary)
Der abstrakte Datentyp, der das Tripel der Operationen `INSERT`, `DELETE`, `SEARCH` unterstützt, heißt **Dictionary**.

Der obige Plane-Sweep-Algorithmus benötigt $\mathcal{O}(n \log(n))$ Schritte für das Sortieren (siehe unten $\mathcal{O}$-Notation). Da es sich um einen output-sensitiven Algorithmus handelt, belaufen sich seine Gesamtkosten auf $\mathcal{O}(n \log(n) + k)$, 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 1.5
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 1.6

$\mathcal{O}$-Notation: Es bedeutet $T_a(n) \in \mathcal{O}(n^2)$, 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* proportional zu n^2 viele Schritte benötigt.

Θ-Notation: Der Algorithmus benötigt höchstens aber auch mindestens so viele Schritte wie angegeben (Verknüfpung von $\mathcal{O}$- 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 $\mathcal{O}(n \log n)$ Schritte sowie zusätzlich $\mathcal{O}(k)$ Schritte für die Ausgabe. Der gesamte Algorithmus ist somit in $\mathcal{O}(n \log n + k)$ zu bewältigen.

1.1. Über schnelle und langsame Algorithmen

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.

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.
$1000n$	10^3	10^5	10^7	10^9	10^{11}	10^{13}
$100n \log n$	$1,4 * 10^2$	$7,7 * 10^3$				$2,6 * 10^{11}$
$100n^2$	10^2	10^3	10^4	10^5	10^6	10^7
$10n^3$	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

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 $\mathcal{O}(n^2)$ und von $\mathcal{O}(n^3)$ 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 exponentielle Laufzeit von $\mathcal{O}(n!)$, 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 $\mathcal{O}$-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 1.7 (Effiziente Algorithmen)
Ein Algorithmus, dessen Laufzeit im schlimmsten Fall (*worst case*) von $\mathcal{O}(n^k)$ für konstantes k nach oben beschränkt wird, d. h. er hat **polynomielle Laufzeit**, heißt **effizient**.

Es gilt also für ein festes k:

$$T_a(n) = \mathcal{O}(n^k)$$

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 1.8 (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:

Algorithmus 1.3.1 : InsertionSort

```
1  begin
2      for j ← 2 to length(A) do
3          key ← A[j];
4          i ← j − 1;
5          while i > 0 and A[i] > key do
6              A[i + 1] ← A[i];
7              i ← i − 1
8          end
9          A[i + 1] ← key
10     end
11 end
```

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 1.9
Es sind c_i die Kosten für die Ausführung der i-ten Zeile des Algorithmus und t_j die 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

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

Als Laufzeiten erhalten wir also

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

Bemerkung 1.1
Für den durchschnittlichen Fall wird angenommen, daß alle Inputreihenfolgen gleichwahrscheinlich sind. Dieser Algorithmus braucht auch im **average case** $\mathcal{O}(n^2)$. Für das Sortieren gibt es bessere Algorithmen.

1.4. Die $\mathcal{O}$-Notation

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

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

f wächst mit zunehmendem n proportional zu g.

1.4.1. Ein Beispiel

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

1. Behauptung: $f \in \mathcal{O}(n^2)$
   Beweis: Gesucht ist ein $c > 0$ und ein $n_0 \in \mathbf{N}$, für das gilt $f(n) \leq cn^2$ 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: $f(n) \geq cn^2 \forall n \geq n_0$. Also $n^2 + 99n \geq cn^2$. 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: $f \in \mathcal{O}(n^2), \quad f(\Omega(n^2)) \Rightarrow f \in \mathcal{O}(n^2) \cap \Omega(n^2) = \Theta(n^2)$

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

1.4.2. Einige Eigenschaften

Transitivität:
$f(n) \in \mathcal{O}(g(n))$ und $g(n) \in \mathcal{O}(h(n)) = f(n) \in \mathcal{O}(h(n))$
$f(n) \in o(g(n))$ 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: $f(n) \in \mathcal{O}(f(n)), \quad f(n) \in \Theta(f(n)), \quad f(n) \in \Omega(f(n))$

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

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

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

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

1.4.3. Konventionen

1. Wir führen die Gaußklammern $\lfloor\rfloor$ und $\lceil\rceil$ ein, wobei $\lfloor x \rfloor$ bzw.$\lceil x \rceil$ die größte bzw. kleinste ganze Zahl kleiner oder gleich bzw. größer oder gleich x bezeichnet. Beispielsweise ist $3 = \lfloor 3,5 \rfloor \leq 3,5 \leq \lceil 3,5 \rceil = 4$.
2. Der Logarithmus log soll immer als $\log_2$, also als dualer Logarithmus interpretiert werden. Im Sinne der $\mathcal{O}$-Notation ist das irrelevant, da Logarithmen mit unterschiedlicher Basis in einem konstanten Verhältnis zueinander stehen. Beispielsweise ist $\log_2 n = 2 \log_4 n$.
3. Es gilt, $\log^{(0)} n \coloneqq n, \log^{(i)} n \coloneqq \log^{(i-1)} \log n$ und
4. $\log^* n \coloneqq \min \left\{ i \;\middle|\; \log^{(i)} n \leq 1 \right\}$. Es gilt, $\lim_{n\to\infty} \log^* n = +\infty$.

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	

Tabelle 1.3.: Cook'sche Linie

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

2. Jetzt geht's los

2.1. Rekurrenzen

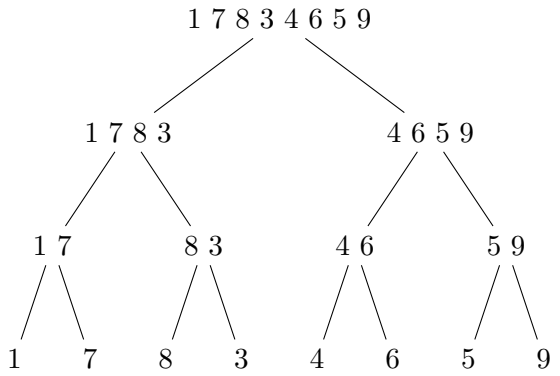
Hier werden ein rekursive Ansätze verwendet. Das Ausgangsproblem wird also in immer kleinere Teilprobleme zerlegt. Irgendwann liegt, analog zu einem induktiven Beweis, ein Trivialfall vor, der sich einfach lösen läßt. Aus den Lösungen der Trivialfälle wird dann sukzessiv eine Lösung des Gesamtproblems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

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

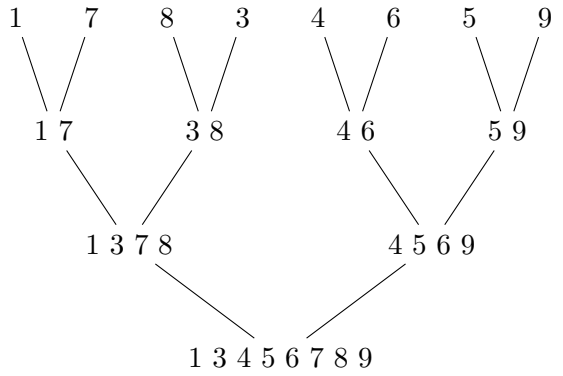
Dazu wird zunächst das Verfahren MergeSort vorgestellt und anhand eines Beispiels verdeutlicht.

Definition 2.1 (MergeSort)
Eine Folge $A = a_l, \ldots, a_r$ 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. Dieses Verfahren nennt man **MergeSort**.

Beispiel 2.1
Gegeben sei die Zahlenfolge $1, 7, 8, 3, 4, 6, 5, 9$. Wir wollen diese mittels MergeSort in die korrekte Reihenfolge bringen. Der Algorithmus verläuft in zwei Schritten. Im ersten Schritt wird die Eingabe aufgesplittet. Das heißt, die Eingabe wird sukzessive in zwei Hälften geteilt.



Im zweiten Schritt des Algorithmus' werden nun die sorierten Teilfolgen wieder gemischt und im Ergebnis erhält man die sortierte Liste.



Das Mischen funktioniert in $\mathcal{O}(n)$ 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 $\mathcal{O}(1)$ Zeit klappt. Also sind zwei Teilfolgen nach $\mathcal{O}(n)$ Schritten zu einer neuen Folge zusammengemischt und man erhält $\mathcal{O}(n)$ als Kosten für das Mischen. Für das gesamte Verfahren MergeSort ergibt sich die Rekurrenz $T(n) = 2T(n/2) + \mathcal{O}(n)$, die zu einer Komplexität von $\mathcal{O}(n \log n)$ führt. Verfahren

für das Lösen von Rekurrenzen werden nach Angabe des Pseudocodes und einer etwas formaleren Laufzeitanalyse für MergeSort eingeführt.

Algorithmus 2.1.1 : MergeSort

```
1 begin
2     if l < r then
3         p ← ⌊(l+r)/2⌋;
4         MergeSort (A,l,p);
5         MergeSort (A, p+1,r);
6         Merge
7     end
8 end
```

Zeile	Asymptotischer Aufwand
2	$\Theta(1)$
3	$\Theta(1)$
4	$T(\frac{n}{2})$
5	$T(\frac{n}{2})$
6	$\Theta(n)$

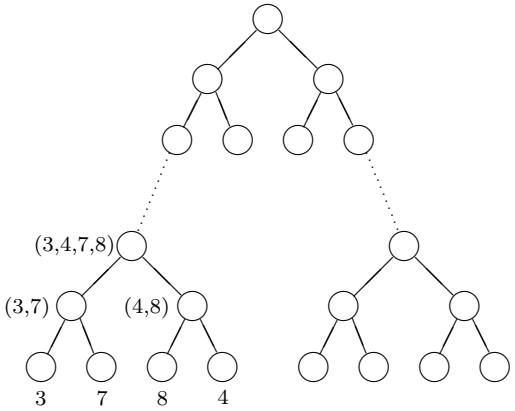
Tabelle 2.1.: Kosten für MergeSort

Zeile 1 und 2 sind elementare Vergleichs- und Zuweisungsoperationen, welche in $\mathcal{O}(1)$ 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.

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 $T(1) = 1 (= \Theta(1))$

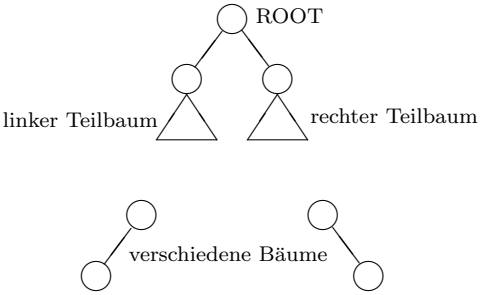
Mit Hilfe des Mastertheorems (siehe Abschnitt 2.3) ergibt sich die Lösung, $T(n) = \Theta(n \log n)$. Dabei wurden die Gaußklammern weggelassen. Da das vertretbar ist, werden wir das auch in Zukunft tun.

Binärbaum-Paradigma:



Definition 2.2
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.



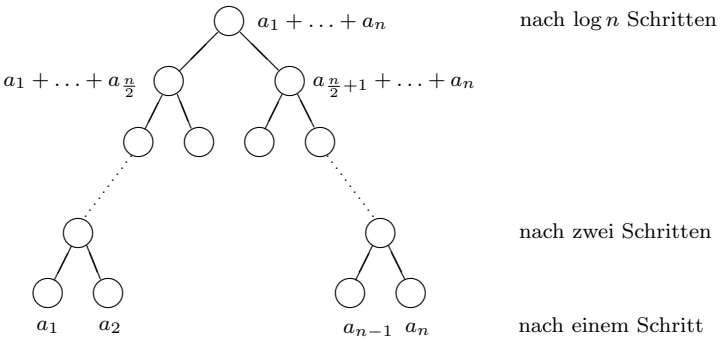
Die Anwendung des Binärbaumparadigmas für parallele Algorithmen wird durch die folgende, grobe Schilderung deutlich. Man stellt sich vor, auf jedem inneren Knoten des Baumes sitzt ein Prozessor, welcher parallel von Level zu Level fortschreitend die Aufgabe löst.

Paralleles und optimales Sortieren benötigt $\mathcal{O}(\log n)$ Zeit. Der Beweis hierfür wurde von Richard Cole erbracht, 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.

Algorithmus 2.1.2 : ParallelSort

```
1 begin
2     Zerlege A in zwei Hälften A_1, A_2;
3     Add (A_1);
4     Add (A_2);
5     Add (A_1) + Add (A_2);
6     Ausgabe
7 end
```



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

2.2. Methoden zur Lösung von Rekurrenzen

Viele Algorithmen enthalten rekursive Aufrufe. Die Zeitkomplexität solcher Algorithmen wird oft durch Rekurrenzen beschrieben. Dieses Kapitel liefert vier Ansätze zum Lösen von Rekurrenzen.

Definition 2.3 (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.

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

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

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

$$\Rightarrow T(n) = \mathcal{O}(n\log n)$$

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

Ansatz: $T(n) = \mathcal{O}(n) \Rightarrow T(n) \leq cn$; c ist konstant

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

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

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

$$\Rightarrow T(n) = \mathcal{O}(n)$$

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

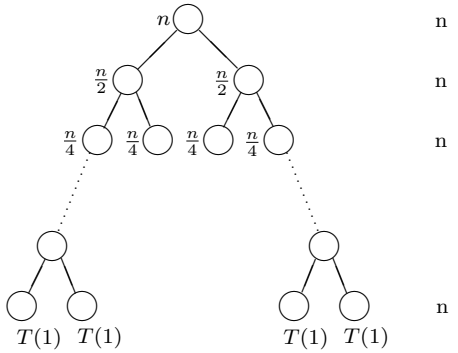
Substitutionsansatz: $k := \log n \Rightarrow 2^k = n$, nach Einsetzen gilt also: $T(2^k) = 2T(2^{\frac{k}{2}}) + k$

Jetzt wird $S(k) := T(2^k)$ gesetzt und somit gilt $S(k) = 2S(\frac{k}{2}) + k$, 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 Abschnitt 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 kleiner als 1 steht. Die Summe aller Knoteneinträge bezeichnet die Lösung der Rekurrenz.

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

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



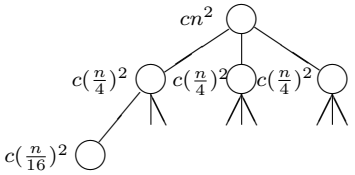
Der Aufwand jeder Zeile beträgt $\mathcal{O}(n)$. Der Baum hat eine Höhe von $\log n$. Damit ergibt sich als Lösung: $\mathcal{O}(n \log n)$

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

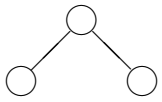
Ansatz: $T(n) = cn^2 + (\frac{3}{16})cn^2 + (\frac{3}{16})^2 cn^2 + \cdots + (\frac{3}{16})^{\log(n-1)} cn^2 + \Theta(n^{\log_4 3})$

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

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



Beispiel: $T(n) = T(\frac{n}{3}) + T(\frac{2 \cdot n}{3}) + \mathcal{O}(n)$, 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 $T(\frac{n}{b})$ 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 $T(n) = aT(\frac{n}{b}) + f(n)$ mit $a \geq 1, b > 1$, dann gilt asymptotisch für große n.

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

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 [1] zu finden.

2.3.1. Beispiele

Beispiel: Binäre Suche

Eingabe ist eine sortierte Folge $a_1 < \ldots < a_n$ und ein Wert b. Ausgegeben werden soll $\iota i \colon a_i \leq b < a_{i+1}$, falls es existiert und ansonsten eine Fehlermeldung.

EXKURS: Binärer Suchbäum

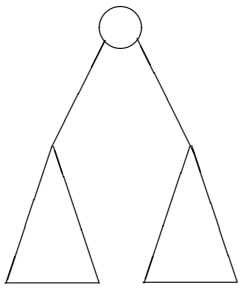


Abbildung 2.1.: Binärbaum

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.

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

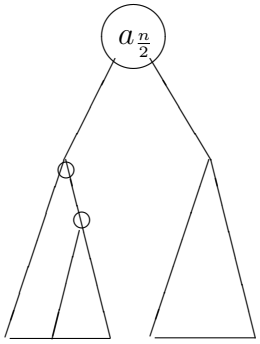


Abbildung 2.2.: Suche im Binärbaum

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: $T(n) = T(\frac{n}{2}) + \mathcal{O}(1)$

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 $a < 500 \Rightarrow 0 \leq a < 500$
2. Ist $a < 250 \Rightarrow 0 \leq a < 250$.
3. Ist $a < 125 \Rightarrow 125 \leq a < 250$.
4. Ist $a < 187 \Rightarrow 125 \leq a < 187$.
5. Ist $a < 156 \Rightarrow 125 \leq a < 156$.
6. Ist $a < 141 \Rightarrow 125 \leq a < 141$.
7. Ist $a < 133 \Rightarrow 125 \leq a < 133$.
8. Ist $a < 129 \Rightarrow 125 \leq a < 129$.
9. Ist $a < 127 \Rightarrow 127 \leq a < 129$.
10. Ist $a < 128 \Rightarrow 128 \leq a < 129 \Rightarrow a = 128$

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 $\Theta(n^{\log_b a}) = \Theta(1)$ und der zweite Fall kommt zur Anwendung. Also ist $T(n) \in \Theta(n^{\log_b a} \log n) = \Theta(\log n)$.

Weitere Beispiele

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

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

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

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

   Die Lösung dieser Übungsaufgabe steht im Abschnitt B.2.

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

   Wie man weiß: $4n \log n \in \Omega(n) \forall \epsilon > 0$ aber $4n \log n \notin \Omega(n^{1+\epsilon})$. Es trifft *kein* Fall zu!

Das Mastertheorem deckt nicht alle Fälle ab!

3. Sortieren und Selektion

Es ist $\mathcal{O}(n \log n)$ die obere Schranke für MergeSort und $\mathcal{O}(n^2)$ für Insertionsort. Lässt sich diese Schranke weiter verbessern? Auf diese Frage gibt es zwei Antworten. Unter bestimmten Bedingungen, wie z. B. mit Parallelrechnern, lassen sich Verbesserungen erzielen. Dies ist jedoch nicht das Thema dieser Vorlesung. Für unseren Fall lautet die Antwort „Nein“. Denn bei allgemeinen Sortierverfahren auf der Basis von Schlüsselvergleichen ist $\mathcal{O}(n \log n)$ die Schranke. Im folgenden wollen wir beweisen, dass für allgemeine Sortierverfahren auf der Basis von Schlüsselvergleichen $\Omega(n \log n)$ eine untere Schranke ist.

Den Beweis wollen wir in mehreren Schritten erarbeiten. Zunächst modellieren wir den Ansatzes „Auf der Basis von Schlüsselvergleichen“. Dazu ist zu bestimmen, was INPUT und OUTPUT besagen.

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

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

  zugegriffen werden kann.

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

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

Definition 3.1
$a(i < j) \coloneqq \{\ (a_1, \ldots, a_n) \mid a_i \in S \wedge a_i < a_j\ \}$

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 $a(i < j) = \{\ (a_1, \ldots, a_n) \in \mathbb{R}^n \mid a_i < a_j$ 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

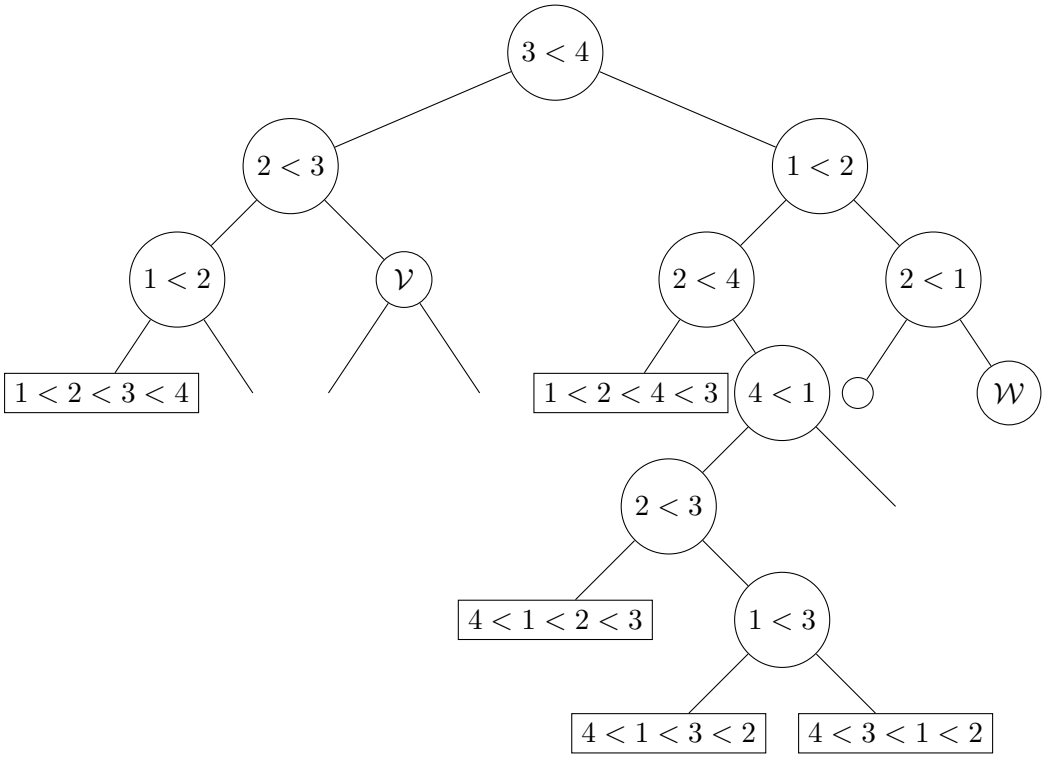


Abbildung 3.1.: Entscheidungsbaum

von Algorithmus A, einem binären Baum, dessen Knoten mit Vergleichen „$a_i < a_j$“ 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 $a(3 < 4) \cap a(3 < 2) = \{ (a_1, \ldots, a_n) \in \mathbb{R}^n \mid a_3 < a_4 \wedge a_3 < a_2 \}$ 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 $a_{\pi(1)} < a_{\pi(2)} < \ldots < a_{\pi(n)}$ zu diesem Blatt führen. Der Entscheidungsbaum hat daher mindestens $n!$ Blätter. Der Beweis dafür stammt fast unverändert aus [5].

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 $\mathcal{W}$).

Beispiel 3.1 (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 3.1
Ein Binärbaum der Höhe h hat höchstens 2^h Blätter.

Beweis:
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

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

Der Beweis ist trivial da

$$n! = 1 \cdot 2 \cdot 3 \cdot \ldots \cdot n = 1 \cdot 2 \cdot 3 \ldots \cdot \underbrace{(\frac{n}{2}+1) \cdot \ldots \cdot n}_{\frac{n}{2}} \geq (\frac{n}{2})^{\frac{n}{2}}$$

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

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 $\mathcal{O}(n \log n)$ Zeit. q. e. d.

Damit ist die Gesamtaussage bewiesen.

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 $g(x_1, \ldots, x_n) = c_1x_1 + \cdots + c_nx_n + d$ 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 3.2
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 $\mathcal{O}(n)$ ist, dann muß die Komplexität des Sortierens $\Omega(n \log n)$ sein.

Beim Elementtest ist eine Menge $\mathbb{M}$ mit $\mathbb{M} \subseteq \mathbb{R}^n$ sowie ein variabler Vektor $(x_1, \ldots, x_n)$ gegeben. Es wird getestet, ob $(x_1, \ldots, x_n) \in \mathbb{M}$, wobei $\mathbb{M}$ 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: $\exists i, j (1 \leq i, j \leq n) \colon |a_i - a_j| < \varepsilon$?

Trivalerweise ist bei einer bereits sortierten Eingabe ε-Closeness in $\mathcal{O}(n)$ entscheidbar. Dazu wird einfach nacheinander geprüft, ob $|a_i - a_{i+1}| < \varepsilon$ für $i < n$.

Satz 3.3
Wenn ε-Closeness die Komplexität $\Omega(n \log n)$ hat, so auch das Sortieren.

Satz 3.4
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:
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 äquivalent zu einer Höhe des Entscheidungsbaumes von $\log(\mathrm{card}(\mathbb{M})) = \log m$ ist. Nun sei b ein Blatt des Baumes.

Definition 3.2
$E(b) \coloneqq \{\, \vec{x} \in \mathbb{R}^n \mid \text{Alg. } A \text{ landet bei der Eingabe von } \vec{x} = (x_1, \ldots, x_n) \text{ in Blatt } b \,\}$

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

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

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

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 3.3
$\mathbb{K}$ ist genau dann **konvex**, wenn $\forall p, q \colon p \in \mathbb{K} \wedge q \in \mathbb{K} \to \overline{pq} \subseteq \mathbb{K}$.

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

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

Satz 3.5
Die Komplexität der ε-closeness ist $\Omega(n \log n)$.

Beweis:
ε-closeness ist ein Elementtest-Problem!

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

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

Satz 3.6
Die Zusammenhangskomponenten von $\mathcal{M}_\varepsilon$ sind die Mengen $\mathcal{M}_\pi$.

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 $\Omega(n \log n)$ $\rightarrow$ dann exisitiert für ε-closeness ein Verfahren von geringerer Komplexität als $\Omega(n \log n)$ $\rightarrow$ Widerspruch zur Voraussetzung $\rightarrow$ 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 $t_{1,1} \ldots t_{1,m_1}$ und $t_{2,1} \ldots t_{2,m_2}$ bezeichnet.

Nach Voraussetzung gilt:

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

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:

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

Hilfssatz:

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

Dabei gelten die selben Bezeichnungen wie oben.
Beweis:

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

Somit folgt, $m \leq 2^{\frac{1}{m}\sum_{i=1}^{m} t_i}$. Also gilt für die Pfadlänge: $\log m \leq \frac{1}{m}\sum_{i=1}^{m} t_i$. q. e. d.

Satz 3.7 (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:
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 $A = (a_0, \ldots, a_n)$, die Werte stehen dabei in den Plätzen $a_1, \ldots, a_n$. Dabei dient $a_0 := -\infty$ als Markenschlüssel, als sogenanntes Sentinel-Element (siehe auch MERGESORT) und v ist das Pivotelement.

Algorithmus 3.2.1 : erste Variante von QUICKSORT

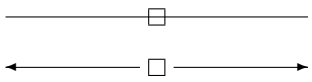
```
  Input : A = (a_0, ..., a_n)
1 begin
2     if r > l then
3         i → Partition (l, r);
4         Quicksort (l, i − 1);
5         Quicksort (i + 1, r)
6     end
7 end
```

Was passiert bei 3.2.2?

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	4 =: Pivotelement
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**	



□=:Pivotelement

Abbildung 3.2.: TODO: Hier muss eine Unterschrift rein!

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

Algorithmus 3.2.2 : Zweite Variante von QUICKSORT

```
   Input : A = (a_0, ..., a_n)
 1 begin
 2     if r > l then
 3         v ← a[r];
 4         i ← l − 1;
 5         j ← r;
 6         repeat
 7             repeat
 8                 i ← i + 1
 9             until a[i] ≥ v ;
10             repeat
11                 j ← j − 1
12             until a[j] ≤ v ;
13             t ← a[i];
14             a[i] ← a[j];
15             a[j] ← t;
16         until j ≤ i ;
17     end
18     Quicksort(l, i − 1);
19     Quicksort(i + 1, r);
20 end
```

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

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

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

Satz 3.8
QUICKSORT benötigt bei gleichwahrscheinlichen Eingabefolgen im Mittel etwa $1,38n \log n$ Vergleiche.

BEWEIS:
$n = 1$:

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

$n \geq 2$:

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

Zur Lösung dieser Rekurrenz wird zuerst die Summe beseitigt, dies sollte jeder selbst nachvollziehen. Die ausführliche Rechnung steht im Abschnitt 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:

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

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

$$T(n) = 2n \ln n \approx 1,38n \log n \qquad \text{q. e. d.}$$

3.3. Auswählen (Sortieren)

Wie in vorangegangenen Vorlesungen besprochen wurde, braucht QUICKSORT bestenfalls $\mathcal{O}(n \log n)$ Zeit und im worst case, wenn beim „Teile und Herrsche“ - Verfahren die Länge der beiden Teilfolgen sehr unterschiedlich ist, $\mathcal{O}(n^2)$ 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 3.4
Sei eine Folge $A = (a_1, \ldots, a_n)$ 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 $\text{Rang}(a_i : A) \coloneqq |\{\, x \mid x \in A \colon x \leq a_i \,\}|$.

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

Sei nun A sortiert, also $A_{sortiert} = (a_{\pi(1)}, \ldots, a_{\pi(n)})$, dann gilt für das Element c mit $\text{Rang}(c : A) = k$ für $1 \leq k \leq n$, daß $c = a_{\pi(k)}$, das heißt: $a_{\pi(1)} \leq \ldots \leq a_{\pi(k)} \leq \ldots \leq a_{\pi(n)}$. 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 3.5
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 $A \coloneqq (a_1, \ldots, a_n)$ 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: $a_{\pi(1)}, a_{\pi(2)}, \ldots, a_{\pi(n)}$ (braucht $\mathcal{O}(n \log n)$ 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)

In Algorithmus ändern.

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: $\mathcal{O}(1)$) Else Zerlege A in $\frac{|A|}{Q}$ Teilfolgen der maximalen Länge Q

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

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

Schritt 4: Definition von drei Mengen:

$$A_1 := \{ x \in A \mid x < m \}$$
$$A_2 := \{ x \in A \mid x = m \}$$
$$A_3 := \{ x \in A \mid x > m \}$$

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: $\max\{\mathcal{O}(1), \mathcal{O}(n)\}$

zu Schritt 2: $\mathcal{O}(1)$ für jedes Sortieren der $\mathcal{O}(n)$ Teilfolgen

zu Schritt 3: $T(\frac{n}{Q})$

zu Schritt 4: $\mathcal{O}(n)$

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.

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

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

3.3.2. Algorithmus S-Quicksort(A)

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
```

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

3.4. Heapsort

Abstrakte Datentypen wurden bereits auf Seite 12 definiert, ebenso der Datentyp 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 3.6 (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 3.7 (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 $\mathcal{O}(n \log n)$ Schritten sortiert werden, unabhängig von den Eingabedaten.

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

Definition 3.8 (Binärer Heap)
Ein **Binärer Max-Heap** ist ein spezieller Binärbaum (wird im Folgenden nochmals definiert) mit den Eigenschaften, daß der Wert jedes Knotens jeweils größer oder gleich 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 $F = k_1, k_2, \ldots, k_n$ von Schlüsseln für alle i mit $2 \leq i \leq n$ die Beziehung $k_i \leq k_{\lfloor i/2 \rfloor}$ 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

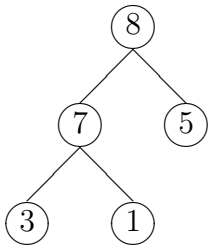


Abbildung 3.3.: Binärer Max-Heap

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 3.9 (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 3.10 (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 3.9
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 3.10
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:
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

$$\frac{K(j)}{K(i)} = \frac{2^l - 1}{3 \cdot 2^{l-1} - 1} \leq \frac{2^l}{3 \cdot 2^{l-1}} = \frac{2}{3} \qquad q.e.d.$$

Satz 3.11
In einem n-elementigen Binärheap gibt es höchstens $\lceil \frac{n}{2^{h+1}} \rceil$ Knoten der Höhe h.

Definition 3.11 (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

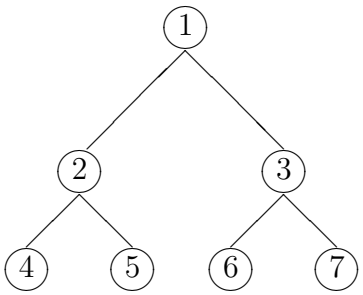


Abbildung 3.4.: Binärer Min-Heap

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:

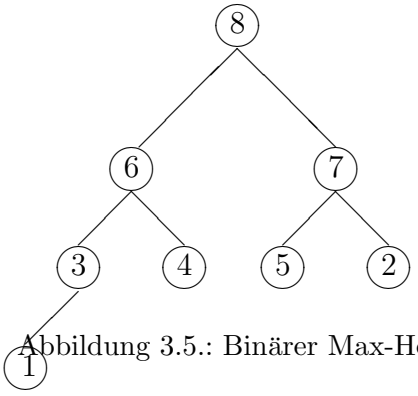


Abbildung 3.5.: Binärer Max-Heap

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:

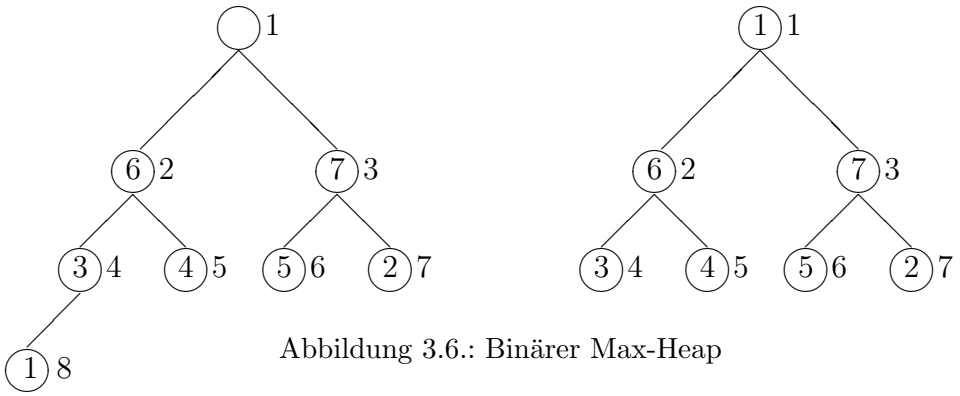


Abbildung 3.6.: Binärer Max-Heap

→ entnehme die Wurzel

→ setze 1 an die Wurzel

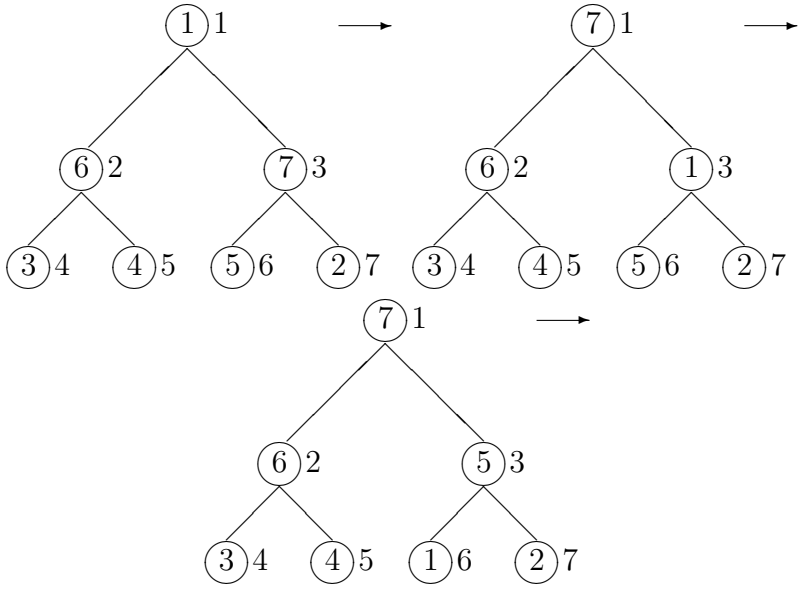


Abbildung 3.7.: Binärer Min-Heap

→ 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 $T(n) = T(2/3n)+\mathcal{O}(1)$, damit gilt $T(n) \in \mathcal{O}(\log n)$.

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)
```

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

Damit kostet Build-Heap nur $\mathcal{O}(n)$.

Damit hat Heapsort die Komplexität $\mathcal{O}(n \log n)$. In jedem der $\mathcal{O}(n)$ Aufrufe von Build-Heap braucht Heapify $\mathcal{O}(\log n)$ 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 3.12
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.

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

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.

IncreaseKey kostet $\mathcal{O}(\log n)$, dazu siehe auch Abbildung 3.8. 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 $\mathcal{O}(\log n)$.

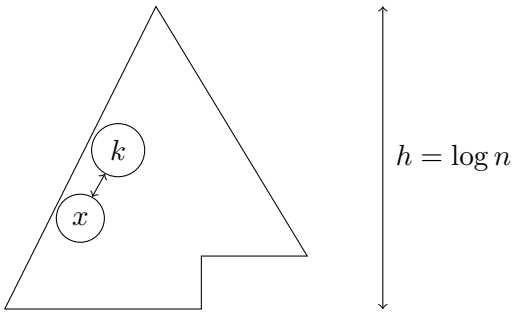


Abbildung 3.8.: Heapify

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

Algorithmus 3.4.1 : INCREASEKEY (A, x, k)

```
1  begin
2      if key[x] > k then
3          Fehler älterer Schlüsselwert ist größer
4      end
5      key[x] ← k;
6      y ← x;
7      z ← p[y];
8      while z ≠ NIL AND key[y] > key[z] do
9          tausche key[y] und key[z];
10         y ← z;
11         z ← p[y]
12     end
13 end
```

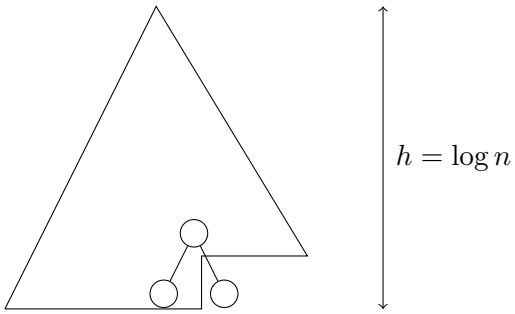


Abbildung 3.9.: Insert

Damit unterstützen binäre Heaps:

- BUILDHEAP $\mathcal{O}(1)$ - (im Sinne von MAKEHEAP = Schaffen der leeren Struktur)
- INSERT $\mathcal{O}(\log n)$
- MAX $\mathcal{O}(1)$
- EXTRACTMAX $\mathcal{O}(\log n)$
- INCREASEKEY $\mathcal{O}(\log n)$

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:

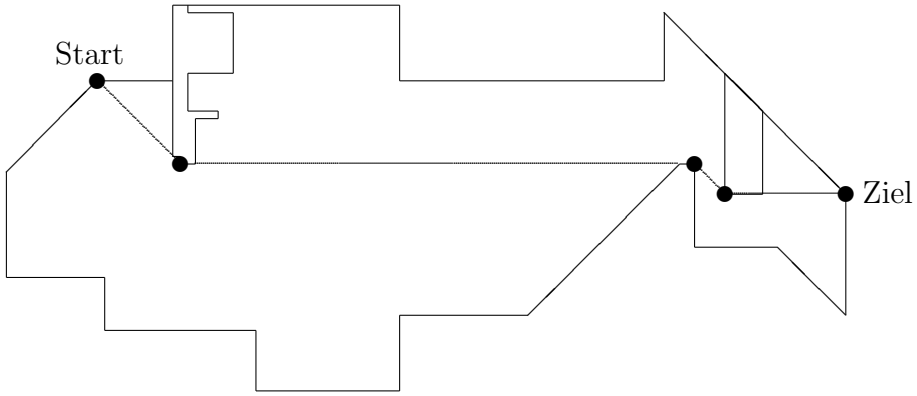


Abbildung 3.10.: Kürzesten Weg finden

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 3.13 (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 3.12
Der kürzeste Pfad verläuft entlang der Kanten des Sichtbarkeitsgraphen (Lorenzo-Parez).

Problem ALL-TO-ONE SHORTEST PATHS

One (In Abbildung 3.11 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.

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

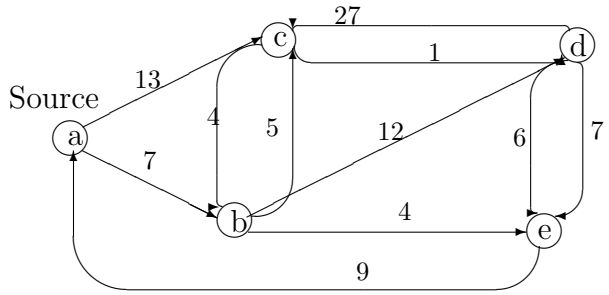


Abbildung 3.11.: Graph mit gerichteten Kanten

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.

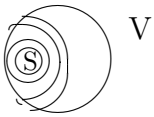


Abbildung 3.12.: Menge W wird aufgeblasen

Nun zum eigentlichen Algorithmus:

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 3.11. 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:

Algorithmus 3.5.1 : DIJKSTRA

```
1  begin
2      for ∀v ∈ V do
3          d(v) ← +∞
4      end
5      d(s) ← 0;
6      W ← ∅;
7      Initialisiere Struktur V;
8      while V \ W ≠ ∅ do
9          v ← min(V \ W);
10         ExtractMin(V \ W);
11         W ← W ∪ {v};
12         for ∀w in Succ (v) \ W do
13             if d(r) + l(vw) < d(w) then
14                 DecreaseKey (w, d(v) + l(vw))
15             end
16         end
17     end
18 end
```

$(v, d(v)) : v \in W$ / $(v, d(v)) : v \in V \cup W$	$v = min(V \setminus W)$ / $SUCC(v)$	$v = min(V \setminus W)$, $w \in SUCC(v) \setminus (W \cup v)$, $(w, l(v\vec{w})$ / $min(d(w), d(v) + l(v\vec{w}))$
$\emptyset$ / $(a, 0), (b, \infty), (c, \infty), (d, \infty), (e, \infty)$	a / c, b	$v = a : (c, 13), (b, 7)$ / $(c, 13), (b, 7)$
$\{(a, 0)\}$ / $(b, 7), (c, 13), (d, \infty), (e, \infty)$	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 die Anzahl der Knoten m ist.

1. $\mathcal{O}(|V|) = \mathcal{O}(n)$

2. $\mathcal{O}(n)$

3. meist in $\mathcal{O}(1)$

4. nicht zu beantworten

5. nicht zu beantworten

6. $\mathcal{O}(1)$ im Regelfall (es kann auch komplexer sein, da es darauf ankommt, wie W verwaltet wird)

7.

8. $\mathcal{O}(1)$

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 $\rightarrow \mathcal{O}(n^2)$

2. Möglichkeit: Binärer Heap für V\W, dann geht EXTRACTMIN in $\mathcal{O}(n \log n)$ und DECREASEKEY in $\mathcal{O}(m \log n)$ $\rightarrow \mathcal{O}((m+n) \log n)$

3. Möglichkeit: V in einen Fibonacci-Heap $\rightarrow \mathcal{O}((n \log n) + m)$

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 $\mathcal{O}(n \log n)$ 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 3.14 (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 $k \in \mathcal{O}(n)$ ist $T(n) \in \mathcal{O}(n) \cup \mathcal{O}(k)$.

Kann man hier überhaupt ein $\cup$ verwenden?

Satz 3.13
Falls $k \in \mathcal{O}(n)$, so funktioniert Counting Sort in $\mathcal{O}(n)$.

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 12). 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 [6], wo ein ADT als Signatur vereinigt mit Axiomen für die Operationen definiert wird.

Definition 4.1 (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 4.2 (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 4.3
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

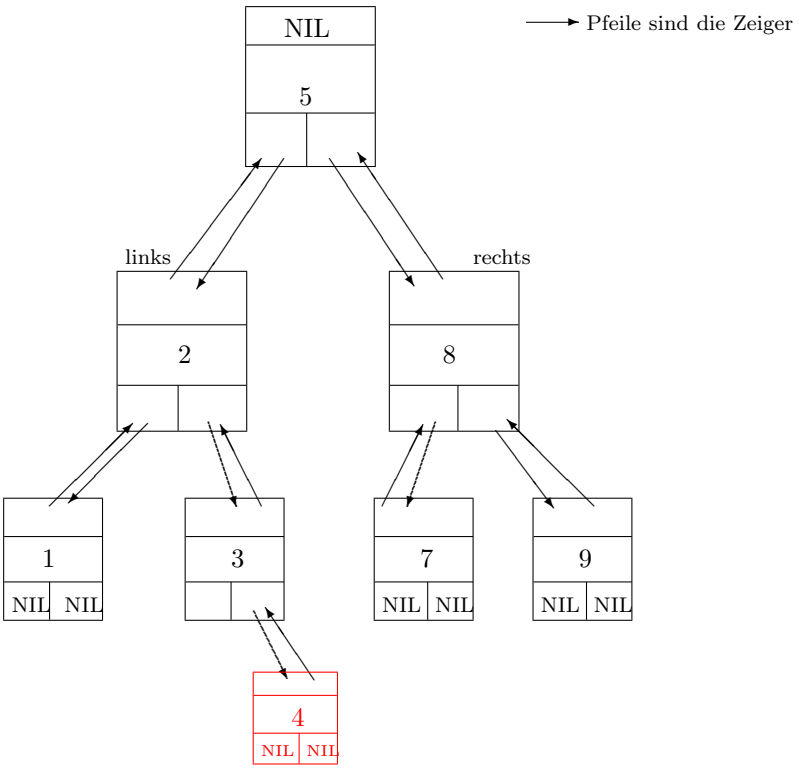


Abbildung 4.1.: Binärer Suchbaum

- 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.

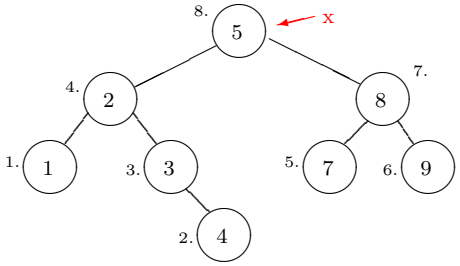


Abbildung 4.2.: TREEPOSTORDER(x)

- 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

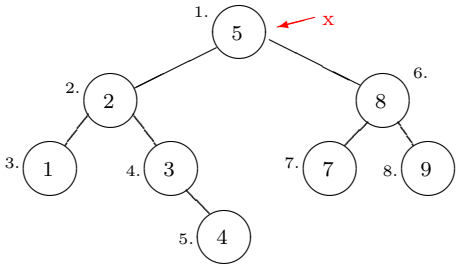


Abbildung 4.3.: TREEPREORDER

- 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

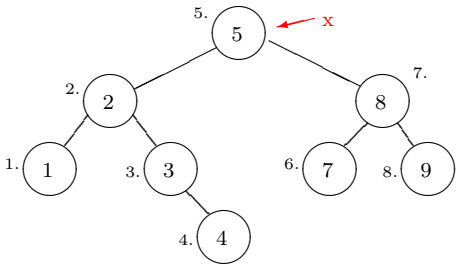


Abbildung 4.4.: TREEINORDER

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

Satz 4.1
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 Einfügens das Aussehen des Baumes bestimmt, deswegen entartet der binäre Suchbaum beim Einfügen einer bereits sortierten Eingabe zu einer linearen Liste.

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
```

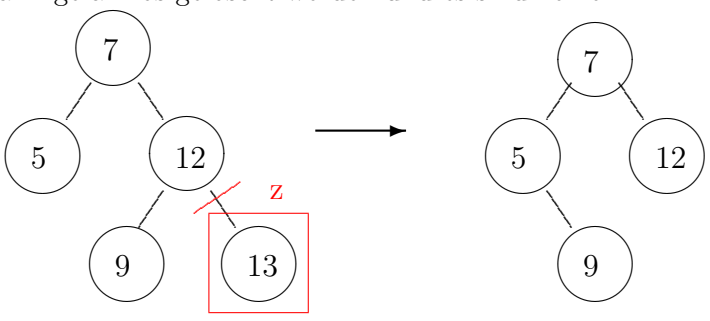
Die Laufzeit liegt in $\mathcal{O}(h)$, wobei h die Höhe von T ist.

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.

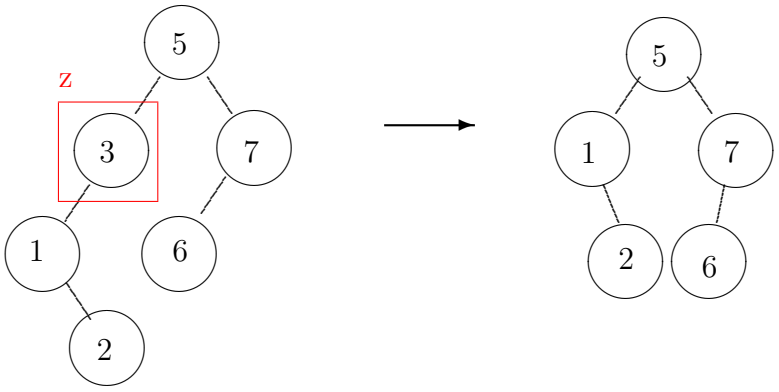
4.1.4. Das Löschen eines Knotens

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.

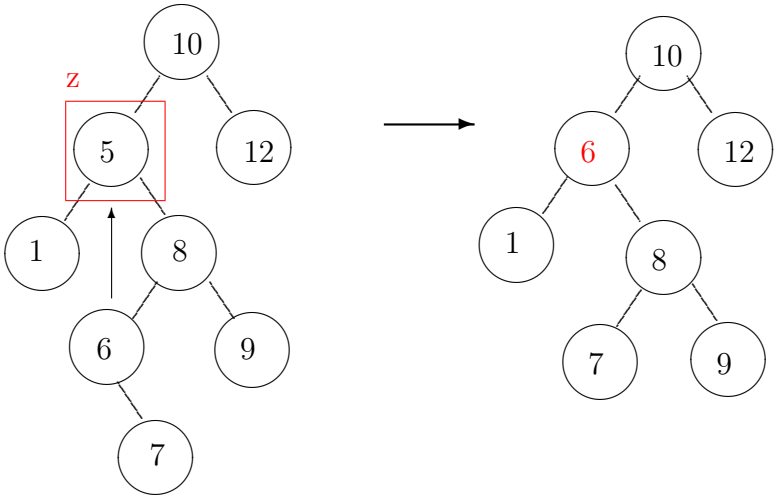


2. Fall z hat genau einen linken Sohn
Der zu löschende Knoten wird entfernt und durch den Wurzelknoten des linken Teilbaums ersetzt.



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)

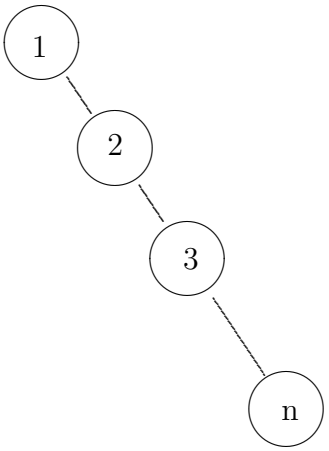
TREE-DELETE

```
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 $\mathcal{O}(h)$, wobei h wieder die Höhe von T bezeichnet.

Im worst case wird TREE-SUCCESSOR mit einer Laufzeit von $\mathcal{O}(h)$ einmal aufgerufen, andere Funktionsaufrufe oder Schleifen gibt es nicht.

Binäre Suchbäume als Implementierung des ADT Wörterbuch



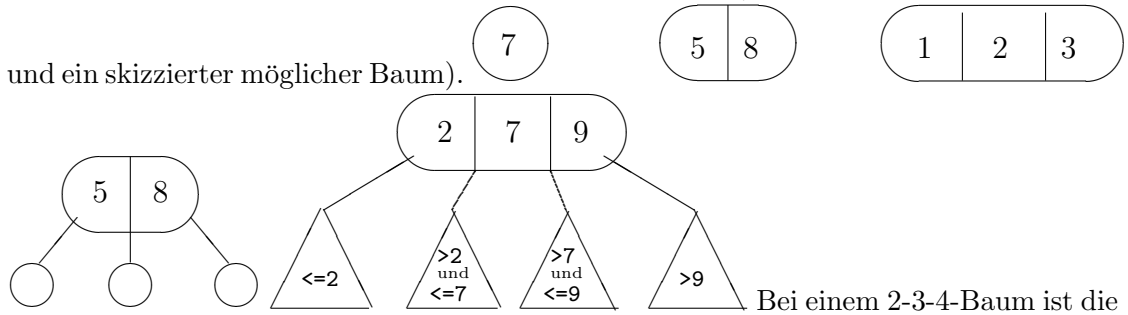
Wie bereits bei der Funktion TREE-INSERT beschrieben, kann eine ungünstige Einfügereihenfolge den Suchbaum zu einer linearen Liste entarten lassen. Deswegen sind allgemeine binäre Suchbäume nicht geeignet, den ADT Wörterbuch zu implementieren.

4.2. 2-3-4-Bäume

Definition 4.4 (2-3-4-Bäume)
2-3-4-Bäume sind Bäume mit folgenden speziellen Eigenschaften:

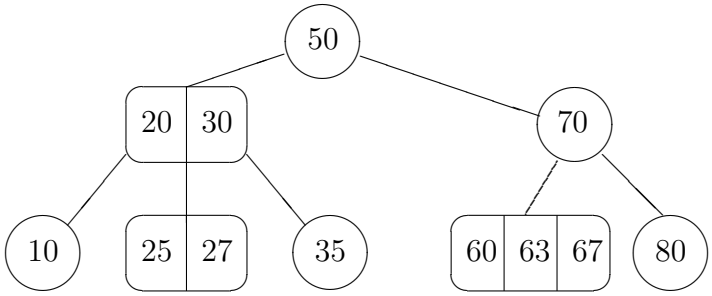
- Jeder Knoten im Baum enthält einen, zwei oder drei Schlüssel, die von links nach rechts aufsteigend sortiert sind.
- Ein Knoten mit k Schlüsseln hat k+1 Söhne (oder er hat überhaupt keine:"Blatt") und wird als (k+1)-Knoten bezeichnet.
- Für Schlüssel im Baum gilt die verallgemeinerte Suchbaumeigenschaft.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

Beispiel für einen 2-3-4-Baum



- Erfolgreiche Suche nach 35
- Erfolglose Suche nach 69

Die Laufzeit für das Suchen liegt wieder in $\mathcal{O}(h)$, mit h als Höhe des Baumes.

4.2.1. Top Down 2-3-4-Bäume für den ADT Dictionary

Wenn die Höhe des Baumes logarithmisch ($h \in \mathcal{O}(\log n)$) ist, eignet er sich gut für den Datentyp Wörterbuch, da dann alle Operationen in $\mathcal{O}(\log n)$ gehen. Insbesondere das in einem Wörterbuch zu erwartende häufige Suchen hat die Komplexität $\mathcal{O}(\log n)$.

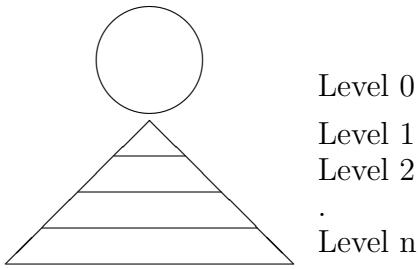


Abbildung 4.5.: $h \in \mathcal{O}(\log n)$

Bereits bei den Binärbäumen muß die Baumstruktur nach dem Löschen eines Knotens manchmal repariert werden. Nun ist klar ersichtlich, daß ein Baum, der immer logarithmische Höhe haben soll, nicht zu einer linearen Liste entarten darf. Falls also das Einfügen eines Elemente in einen Baum die Baumstruktur so ändert, daß die Eigenschaften verletzt sind, muß der Baum repariert werden.

Wie in Abbildung 4.6 zu sehen ist, kann das Einfügen eines einzigen Knotens dazu führen, daß eine neue Ebene einfügt werden muß. Falls dies in der untersten Ebene geschieht (worst-case), kann sich das bis zur Wurzel fortsetzen. Top down 2-3-4 Bäume sollen diese Situation verhindern!

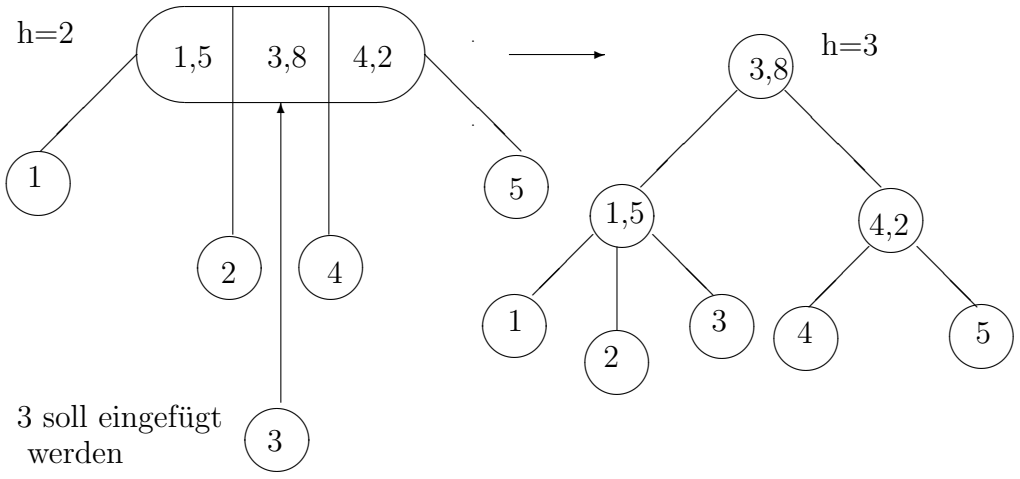


Abbildung 4.6.: $h \in \mathcal{O}(\log n)$

Der worst case wird dadurch verhindert, daß zwischendurch etwas mehr Aufwand betrieben wird. So wird beim dem Einfügen vorausgehenden Suchen jeder erreichte 4-Knoten sofort aufgesplittet. So ist gewährleistet, daß immer Platz für einen neuen Knoten ist.

Allerdings haben Top down 2-3-4-Bäume auch den Nachteil, daß sie schwerer implementierbar sind als die 2-3-4 Bäume. Dafür sind Rot-Schwarz-Bäume besser geeignet.

4.3. Rot-Schwarz-Bäume

Definition 4.5 (Rot-Schwarz-Bäume)
Rot-Schwarz-Bäume sind binäre Suchbäume mit folgenden zusätzlichen Eigenschaften:

1. Jeder Knoten ist rot oder schwarz.
2. Die Wurzel ist schwarz.
3. Jedes Blatt ist schwarz.
4. Ein roter Vater darf keinen roten Sohn haben.
5. Die Schwarzhöhe ($\mathrm{Bh}(j)$) ist die Anzahl der schwarzen Knoten auf einem Weg von einem Knoten j zu einem Blatt. Sie ist für einen Knoten auf allen Wegen gleich.

Definition 4.6 (Die Schwarzhöhe)
$\mathrm{Bh}(x)$ ist die Anzahl von schwarzen Knoten auf einem Weg, ohne den Knoten x selbst von x zu einem Blatt und heißt Schwarzhöhe von x.

Satz 4.2 (Die Höhe von Rot-Schwarz-Bäumen)
Die Höhe eines Rot-Schwarz-Baumes mit n Knoten ist kleinergleich $2\log(n+1)$

Satz 4.3
Sei x die Wurzel eines Rot-Schwarz-Baumes. Dann hat der Baum mindestens $2^{\mathrm{Bh}(x)} - 1$ Knoten. Der Beweis kann induktiv über die Höhe des Baumes erfolgen.

Um Speicherplatz zu sparen, bietet es sich an, nur ein Nil-Blatt abzuspeichern. Dann müssen natürlich alle Zeiger entsprechend gesetzt werden.

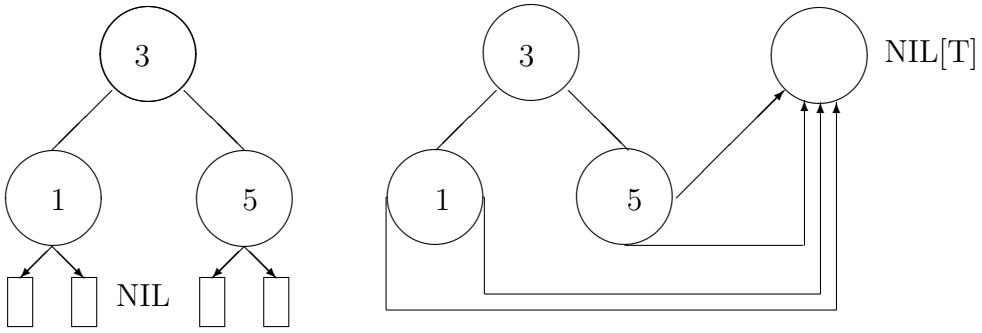


Abbildung 4.7.: Nur ein NIL-Blatt

Beweis:
IA : Sei $h = 0$. Dann handelt es um einen Baum, der nur aus einem NIL-Blatt besteht. Dann ist $\mathrm{Bh}(x) = 0$ und nach Satz 4.3 die Anzahl der inneren Knoten mindestens $2^0 - 1 = 0$. Damit ist der Induktionsanfang für Satz 4.3 gezeigt.

IS : Sei nun h'<h. Dazu betrachten wir die zwei Teilbäume eines Baumes mit der Höhe h und der Wurzel x. Dann gibt es jeweils für den rechten und den linken Teilbaum zwei Fälle. Dabei sind die Fälle für die beiden Teilbäume analog und werden deswegen gleichzeitig abgearbeitet.

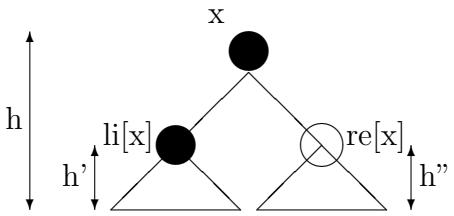


Abbildung 4.8.: Baumhöhe mit $\mathrm{Bh}(li[x]) = \mathrm{Bh}(x) - 1$ und $\mathrm{Bh}(re[x]) = \mathrm{Bh}(x)$

1. Fall Sei $\mathrm{Bh}(li[x]) = \mathrm{Bh}[x]$. Da eine untere Schranke gezeigt werden soll und in diesem Fall der Baum nicht weniger Knoten hat als im zweiten Fall, reicht es den Beweis für den zweiten (kritischeren) Fall zu führen (Der zweite Fall ist kritischer, da der Teilbaum weniger Knoten haben kann als im ersten Fall und damit eher in der Lage ist, die untere Schranke zu durchbrechen).

2. Fall Sei $\mathrm{Bh}(li[x]) = \mathrm{Bh}[x] - 1$, dann gibt mindestens soviele innere Knoten, wie die beiden Teilbäume nach Induktionsvoraussetzung zusammen haben. Damit hat der gesamte Baum mindestens $1+2^{\mathrm{Bh}(x)-1}-1+2^{\mathrm{Bh}(x)-1}-1 = 2\cdot 2^{\mathrm{Bh}(x)-1}+1-2 = 2^{\mathrm{Bh}(x)} - 1$. Damit ist Satz 4.3 bewiesen.

Zum Beweis von Satz 4.2 wird Eigenschaft 4 der Rot-Schwarz-Bäume ausgenutzt. Daraus folgt direkt, daß auf dem Weg von der Wurzel zu den Blättern, mindestens die Hälfte der Knoten schwarz ist. Sei h wieder die Höhe des Baumes, dann gilt damit

Unten steht das Wort root im Mathesatz.

$$\begin{aligned} \mathrm{Bh}(root) \geq \frac{h}{2} &\overset{\text{Satz 4.3}}{\longrightarrow} \\ n \geq 2^{\mathrm{Bh}(root)} - 1 \geq 2^{\frac{h}{2}} - 1 &\Rightarrow 2^{\frac{h}{2}} \leq n+1 \\ \Leftrightarrow \frac{h}{2} \leq \log(n+1) & \qquad \text{q. e. d.} \end{aligned}$$

4.3.1. Operationen in RS-Bäumen

Die Operationen Tree-Insert und -Delete haben bei Anwendung auf einen RS-Baum eine Laufzeit von $\mathcal{O}(\log n)$. Da sie den Baum verändern, kann es vorkommen, daß die Eigenschaften des Rot-Schwarz-Baumes verletzt werden. Um diese wieder herzustellen, müssen die Farben einiger Knoten im Baum sowie die Baumstruktur selbst verändert werden. Dies soll mittels Rotationen realisiert werden, dabei gibt es die Linksrotation und die Rechtsrotation.

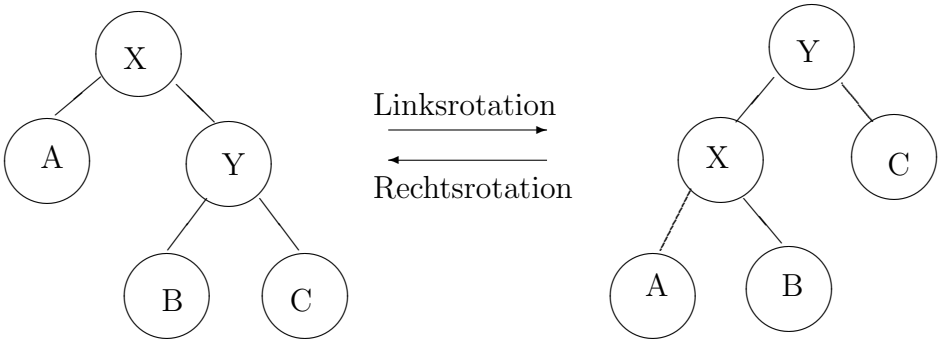


Abbildung 4.9.: Die Rotation schematisch

LINKSROTATION(T, z)

```
1  y := re[x]
2  re[x] := li[y]
3  p[li[y]] := x
4  p[y] := p[x]
5  if p[x] = NIL
6    then root[T] := y
7    else if x = li[p[x]]
8      then li[p[x]] := y
9      else re[p[x]] := y
10 li[y] := x
11 p[x] := y
```

Satz 4.4
Rotationen ändern die Gültigkeit der Suchbaumeigenschaft nicht.

Wie die Skizze vermuten läßt, ist der Code für die RETCHSROTATION symmetrisch zu dem für die LINKSROTATION. Beide Operationen erfordern $\mathcal{O}(1)$ Zeit, da mit jeder Rotation nur eine konstante Anzahl von Zeigern von umgesetzt wird und der Rest unverändert bleibt.

Sei T ein Rot-Schwarz-Baum. Ziel ist, daß T auch nach Einfügen eines Knotens z ein Rot-Schwarz-Baum ist. T soll also nach Anwendung von RS-INSERT(T, z) und einer eventuellen Korrektur die Bedingungen für Rot-Schwarz-Bäume erfüllen. Im folgenden Beispiel wird die „3“ eingefügt.

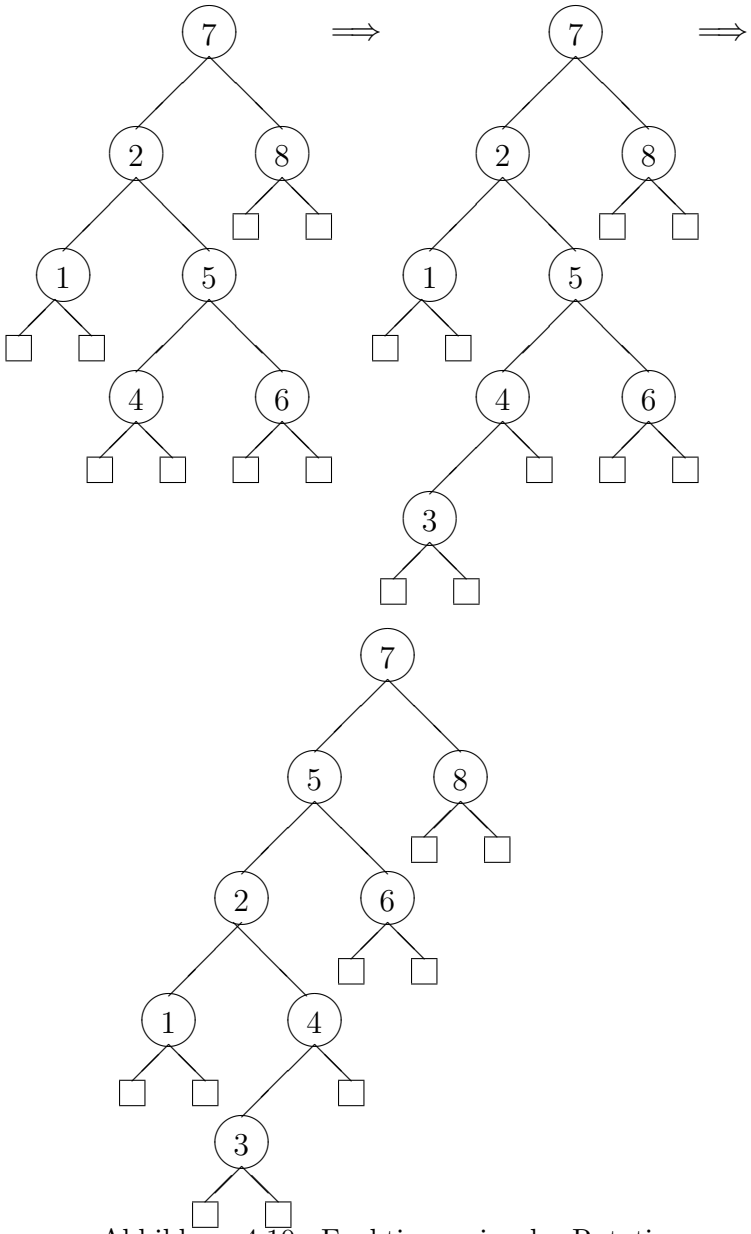


Abbildung 4.10.: Funktionsweise der Rotation

Die letzte Rotation sollte zur übung selbst nachvollzogen werden. Der fertige Baum als Ergebnis dieser letzten Roation steht im Anhang auf Seite 119.

RS-INSERT(T, z)

```
1  y := NIL[T]
2  x := root[T]
3  while x ≠ NIL[T] do
4    y := x
5    if key[z] < key[x]
6      then x := li[x]
7      else x := re[x]
8  p[z] := y
9  if y = NIL[T]
10   then root[T] := z
11   else if key[x] < key[y]
12     then li[y] := z
13     else re[y] := z
14 li[z] := NIL[T]
15 re[z] := NIL[T]
16 Farbe[z] := ROT
17 KORRIGIERE (T, z)
```

Früher wurden die Knoten gefärbt, mittlerweile ist man aber dazu übergegangen, die Kanten zu färben. Dabei gilt, daß eine rote Kante auf einen früher rot gefärbten Knoten zeigt und eine schwarze Kante auf Knoten, die früher schwarz gefärbt wurden. Die Färbungen sind auch auf andere Bäume übertragbar. Die Färbung der Kanten hat den Nachteil, daß sich die Schwarzhöhe so ergibt, daß die roten Kanten auf einem Weg nicht mitgezählt werden. Einfacher und damit sicherer ist es, wenn nur die schwarzen Knoten gezählt werden.

Für alle höhenbalancierten Bäume gilt, daß ihre Höhe in $\mathcal{O}(\log n)$ liegt, allerdings haben Rot-Schwarz-Bäume den Vorteil, daß sie leichter zu implementieren sind. Da die NIL-Blätter schwarz sind, kann ein einzufügender Knoten auch erstmal einmal rot gefärbt sein.

Eine mögliche Anwendung für einen Rot-Schwarz-Baum ist das bereits einleitend erwähnte Segmentschnitt-Problem. Dabei kann ein Rot-Schwarz-Baum für die Verwaltung der Sweepline-Status-Struktur verwendet werden (Menge Y, dazu siehe auch Anhang A).

4.4. Optimale binäre Suchbäume

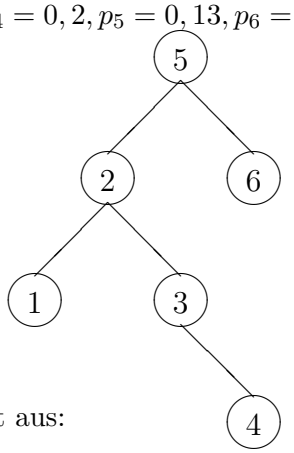
Seien wie im folgenden Beispiel Schlüsselwerte $a_1 < \ldots < a_n$ mit festen bekannten Wahrscheinlichkeiten $p_1, \ldots, p_n$ gegeben, wobei gilt $p_i \geq 0$ und $\sum_{i=1}^{n} p_i = 1$. Dabei bezeichnet

Algorithmus 4.3.1 : Korrigiere(T, z)

```
1  while Farbe von p[z] = ROT do
2      if p[z] = links[p[p[z]]] // Vater ist linker Sohn von Opa von z
3      then
4          y := [p[p[z]]] // y ist Onkel von z
5          if Farbe[y] = ROT // Vater und Onkel rot
6          then
7              Farbe[p[z]] := SCHWARZ // Vater wird schwarz
8              Farbe[y] := SCHWARZ // Onkel wird schwarz
9              Farbe[p[p[z]]] := ROT // Opa wird schwarz
10             z := p[p[z]]
11         end
12         else if z = rechts[p[z]] then
13             z := p[z];
14             Linksrotation(T, z);
15             Farbe[p[z]] := SCHWARZ;
16             Farbe[p[p[z]]] := ROT;
17             Rechtsrotation(T, p[p[z]])
18         end
19     end
20     else
21         wie oben, recht und links vertauscht
22     end
23 end
24 Farbe[Root[T]] := SCHWARZ
```

p_i die Wahrscheinlichkeit mit der auf den Wert a_i zugegriffen wird.

Beispiel: Sei nun $a_1 = 1, a_2 = 2, \ldots, a_6 = 6$,
$p_1 = 0,25, p_2 = 0,28, p_3 = 0,1, p_4 = 0,2, p_5 = 0,13, p_6 = 0,04$ und der binäre Suchbaum sähe wie folgt aus:



An diesem Beispiel wird deutlich, daß es bis zur 4 ein relativ langer Weg ist, d. h. auch die Suche nach 4 dauert im Vergleich zu anderen lange. Nun hat aber die 4 eine hohe Wahrscheinlichkeit und wird deshalb oft abgefragt werden. Also wird die durchschnittliche Rechenzeit relativ lang sein. Es wäre also schöner, wenn die 4 weiter oben im Baum stünde. Diese überlegungen lassen sich fortführen und es fraglich erscheinen, ob sich durch Umplazieren von anderen Schlüsselwerten, die Rechenzeit noch weiter verkürzen läßt (z. B. könnte die 6 weit nach unten, da sie die kleinste Wahrscheinlichkeit hat und somit nur selten abgefragt wird). Aber wo liegen die Grenzen dieses Prozesses? Es ist ja klar, daß er irgendwie begrenzt sein muss. Es bleibt also die Frage:

Gibt es irgendwelche Schranken, durch die die durchschnittliche Rechenzeit beschränkt ist und wenn ja, wie sehen diese aus?

Sicher ist, daß die Rechenzeit eng mit der Höhe des Baumes zusammenhängt. Die Frage ist also äquivalent dazu, ob es Grenzen für die durchschnittliche Knotentiefe gibt und falls ja, wie diese aussehen. Allgemein sollen in diesem Kapitel folgende Fragen beantwortet werden:

1. Wie muss der binäre Suchbaum konstruiert werden, damit er optimal ist?

2. Durch welche Grenzen wird die mittlere Knotentiefe beschränkt?

Dazu muß zuerst **Optimalität** definiert werden. Hier ist damit folgendes gemeint:

Definition 4.7
Ein binärer Suchbaum heisst **optimal**, wenn die Summe $\sum_{i=1}^{n} p_i(t_i + 1)$ minimal ist, wobei t_i die Tiefe des Knotens von a_i angibt. Die Addition von „1“ zur Tiefe erfolgt, damit auch der Wert für die Wurzel in das Ergebnis eingeht.

Fakt ist, daß jeder Teilbaum in einem optimalen Suchbaum wieder optimal ist. Der Sachverhalt, daß eine optimale Lösung des Gesamtproblems auch jedes Teilproblem optimal löst, heißt „Optimalitätskriterium von Bellmann“.

Diese Tatsache kann nun benutzt werden, um mittels dynamischer Programmierung einen optimalen Suchbaum zu konstruieren.

4.4.1. Bottom-Up-Verfahren

Gegeben ist folgendes:

- Gesamtproblem $(a_1, \ldots, a_n)$, d. h. n Schlüsselwerte mit den dazu gehörigen Wahrscheinlichkeiten p_i für $i = 1, \ldots, n$
- Tiefen der Knoten $t_1, \ldots, t_n$
- Die mittlere Suchzeit, gegeben durch $\sum_{i=1}^{n} p_i(t_i + 1)$

Idee Das Gesamtproblem wird in Teilprobleme aufgesplittet, d. h. grössere optimale Suchbäume werden aus kleineren optimalen Suchbäumen berechnet und zwar in einem rekursiven Verfahren.

Angenommen alle möglichen optimalen Suchbäume mit weniger als den n gegebenen Schlüsselwerten sind schon bekannt. Der optimale Suchbaum mit n Schlüsselwerten besteht aus einer Wurzel, einem rechten und einem linken Teilbaum, wobei für die Teilbäume die optimale Darstellung schon gegeben ist. Zu suchen ist noch die Wurzel a_k, für die die Summe der mittleren Suchzeiten der beiden Teilbäume minimal ist.

Dazu definiert man die Teilprobleme $(i, j) = (a_i, \ldots, a_j)$. Der Baum für das optimale Teilproblem wird mit $T(i, j)$ bezeichnet. Weiter ist $p(i, j) \coloneqq \sum_{m=i}^{j} p_m$ die Wahrscheinlichkeit, daß ein Wert zwischen a_i und a_j erfragt wird.
Ziel ist es, die mittlere Teilsuchzeit $t(i, j) \coloneqq \sum_{m=i}^{j} p_m(t_m + 1)$ zu minimieren. Dies beinhaltet das Problem, die optimale Wurzel zu suchen. Man wählt also ein beliebiges $k \in [i, j]$, setzt a_k als Wurzel an und berechnet das dazugehörige $t(i, j)$
Betrachten Teilbaum $T(i, j)$:
Sei a_k die Wurzel und $j > 0$. Die Suchzeit berechnet sich durch
$t(i, j)$ = Anteil der Wurzel + Anteil linker Teilbaum + Anteil rechter Teilbaum $\Rightarrow$ $t(i, j) = p_k \cdot 1 + p(i, k-1) + t(i, k-1) + p(k+1, j) + t(k+1, j)$ wegen $p_k + p(i, k-1) + p(k+1, j) = p(i, j)$ folgt $t(i, j) = \begin{cases} 0, & i > j \\ p(i, j) + \min_{i \le k \le j}[t(i, k-1) + t(k+1, j)], & sonst \end{cases}$

Beispiel: Zahlen wie oben

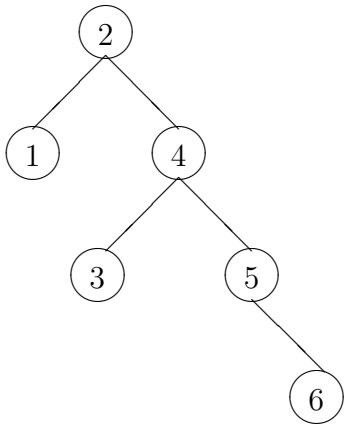
i\ j	1	2	3	4	5	6
1	0,25/1	0,78/2	0,98/2	.../2	.../2	.../2
2		0,28/2	0,48/2	0,98/2	.../2	.../2
3			0,1/3	0,4/4	.../4	.../4
4				0,2/4	.../4	.../4
5					0,13/5	.../5
6						0,04/6

Tabelle 4.1.: Tabelle für $t(i,j)$/ optimale Wurzel k

$$
\begin{aligned}
t(1,2) &= p(1,2) + \min[t(1,0) + t(2,2), t(1,1) + t(3,2)] \\
&= 0,53 + \min[0 + 0,28, 0,25 + 0] \\
&= 0,53 + 0,25 = 0,78 \\
t(2,3) &= p(2,3) + \min[t(2,1) + t(3,3), t(2,2) + t(4,3)] \\
&= 0,38 + \min[0 + 0, 1, 0,28 + 0] \\
&= 0,38 + 0,1 = 0,48 \\
t(2,4) &= p(2,4) + \min[t(2,1) + t(3,4), t(2,2) + t(4,4), t(2,3) + t(5,4)] \\
&= 0,58 + \min[0 + 0,4, 0,28 + 0,2, 0,48 + 0] \\
&= 0,58 + 0,4 = 0,98
\end{aligned}
$$

$\Rightarrow$

optimaler Baum:



4.4.2. Schranken

Wir wollen nun der Frage nachgehen welche Schranken für die mittlere Rechenzeit gelten, und zwar nach oben und nach unten. Dazu ein kurzer Einschub zur Entropie: Gegeben sei ein Zufallsexperiment mit m Ausgängen und den zugehörigen Wahrscheinlichkeiten $p_1, \ldots, p_m$

Entropie $H(p_1, \ldots, p_n)$ = Maß für die Unbestimmtheit eines Versuchsausganges und berechnet sich durch $H(p_1, \ldots, p_n) = -\sum_{i=1}^{m} p_i \log_2 p_i$

Satz 4.5

- $H(p_1, \ldots, p_n) \leq \log_2 m$ (Gleichheit gilt $\Leftrightarrow p_1 = \ldots = p_m = \frac{1}{m}$)
- $H(p_1, \ldots, p_n) = H(p_1, \ldots, p_n, 0)$

- $H(p_1, \ldots, p_n) = H(p_{\pi(1)}, \ldots, p_{\pi(m)})$

- Satz von Gibb: seien $q_1, \ldots, q_m$ Werte größer oder gleich Null und $\sum_{i=1}^{m} q_i \leq 1$, so gilt $H(p_1, \ldots, p_n) \leq -\sum_{i=1}^{m} p_i \log_2 q_i$

Satz 4.6
Wenn die Daten nur in den Blättern stehen, so ist die Entropie eine untere Schranke für die mittlere Tiefe der Blätter $\sum_{i=1}^{m} p_i t_i$ (hier $t_i, i = 1, \ldots, n$ Tiefe der Blätter), es gilt $H(p_1, \ldots, p_n) \leq \sum_{i=1}^{m} p_i t_i$

Beweis:
Sei $(q_1, \ldots, q_m) = (2^{-t_1}, \ldots, 2^{-t_m})$, dann gilt nach obigem Satz:

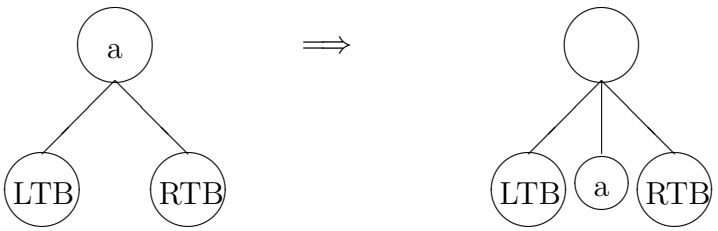
$$\begin{aligned} H(p_1, \ldots, p_n) &\leq -\sum_{i=1}^{m} p_i \log_2 q_i \\ &= -\sum_{i=1}^{m} p_i \log_2 2^{-t_i} \\ &= \sum_{i=1}^{m} p_i t_i \end{aligned}$$

Die Werte für q sind korrekt gewählt, da nach der Ungleichung von Kraft gilt: $\sum_{i=1}^{m} 2^{-t_i} \leq 1$ q. e. d.

Satz 4.7
Die mittlere Knotentiefe eines optimalen binären Suchbaumes (Daten im gesamten Suchbaum) liegt im Intervall $\left[\frac{H(p_1, \ldots, p_n)}{\log_2 3} - 1, H(p_1, \ldots, p_n)\right]$ wobei p_i die Wahrscheinlichkeit ist, mit der der i-te Knoten abgefragt wird.

Beweis:

1. mittlere Knotentiefe $\geq \frac{H(p_1, \ldots, p_n)}{\log_2 3} - 1$.
   Jeder binäre Suchbaum kann in einen ternären Baum transformiert werden, in dem nur die Blätter Daten enthalten. Die neuen Blätter rutschen eine Ebene tiefer:



   Es gilt:

$$\begin{aligned}
\text{mittl. Knotentiefe des bin. Baums} &\geq \text{mittl. Blatttiefe des ternären Baums} - 1 \\
&\geq H_3(p_1, \ldots, p_n) - 1 = -\sum_{i=1}^{n} p_i \cdot \log_3 p_i \\
&= -\frac{1}{\log_2 3} \sum_{i=1}^{n} p_i \cdot \log_2 p_i - 1 \\
&= \frac{1}{\log_2 3} H(p_1, \ldots, p_n) - 1
\end{aligned}$$

2. $H(p_1, \ldots, p_n) \geq$ mittlere Knotentiefe
   Um für $p_i, \ldots, p_j$ einen möglichst guten Suchbaum zu bestimmen, berechnen wir $q = \sum_{k=i}^{j} p_k$ und wählen als (Teilbaum)-Wurzel den Knoten k für den gilt: $\sum_{k=i}^{l-1} p_k \leq \frac{q}{2} \leq \sum_{k=i}^{l} p_k$ Für die Teilfolgen $(p_i, \ldots, p_{l-1})$ und $(p_{l+1}, \ldots, p_j)$ verfahren wir rekursiv. Gestartet wird mit $(p_1, \ldots, p_n)$.
   Nun gilt für Tiefe t_l einer jeden Teilbaumwurzel l (mit $i \leq l \leq j$), daß $\sum_{k=i}^{j} p_k \leq 2^{-t_l}$ (wegen Wahl von Wurzel). Insbesondere folgt: $p_l \leq 2^{-t_l}$. Dies ergibt: mittlere Suchbaumtiefe $\sum_{i=1}^{n} p_i \cdot t_i \leq -\sum_{i=1}^{n} p_i \cdot \log_2 p_i = H(p_1, \ldots, p_n)$ q. e. d.

4.5. Stapel

Stapel haben eine so große Bedeutung in der Informatik, daß sich auch im Deutschen das englische Wort „Stack“ eingebürgert hat. In vielen Algorithmen ist eine Menge zu verwalten, für die sich die einfache Struktur eines Stapels anbietet. Manche Caches arbeiten nach dem LIFO-Prinzip (Last In First Out), dazu reicht ein simpler Stapel, in den die zu verwaltenden Elemente der Reihe nach hereingegeben werden und ein Zeiger auf das zuletzt hereingegebene Element gesetzt wird.

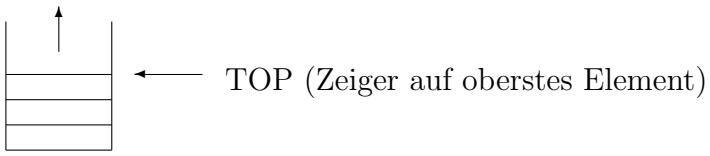


Abbildung 4.11.: Schematische Darstellung eines Stapels

Die Basisoperationen eines Stapels sind natürlich Push, Pop und Stack-Empty (Test ob Stack leer), die Implementierung dieser einfachen Operationen ist simpel. Dabei steht „S“ für den Stapel.

PUSH

```
1 Top[s]:=Top[S]+1
2 S[Top[S]]:=x
```

STACK-EMPTY

```
1 if Top[s]=0
2   Then return true
3   Else return false
```

POP

```
1 if Stack-Empty
2   Then Error
3   Else Top[S]:=Top[S]-1
4     return S[Top[S]+1]
```

Eine These besagt, daß Stapel beim Algorithmenentwurf vor allem für die Verwaltung von Kandidaten verwendet werden. Ein Beispiel dafür ist die Berechnung der konvexen Hülle einer Menge.

4.5.1. Konvexe Hülle

CH von englisch Convex Hull bezeichnet die konvexe Hülle, z. B. CH(P) für ein Polygon oder CH(X) für $X \subseteq \mathbb{R}^2$. Dabei ist diese mathematisch so definiert:

Definition 4.8

$$\mathrm{CH(X)} = \bigcap_{\mathrm{M\ konvex} \wedge X \subseteq M} M$$

Anschaulich gesprochen ist CH(X) damit die kleinste Menge, die X umfaßt. Eine sehr schöne Beschreibung ist auch diese: Man stelle sich ein Brett mit hereingeschlagenen Nägeln vor. Dann vollzieht ein Gummiband, welches um die Nägel gelegt wird, die konvexe Hülle der Nagelmenge nach. Die Punkte der konvexen Hülle sind nur die Nägel, die das Gummiband tangiert.

Satz 4.8
Wenn X endich ist, so ist CH(X) ein Polygon (Jede Punktmenge läßt sich auczh als Polygon auffassen).

Ziel ist die Berechnung der CH(X) für endliche $X \subseteq \mathbb{R}^2$. Erreicht wird dieses Ziel durch den Algorithmus von Graham (1972), der sich grob wie folgt einteilen läßt:

- Sortiere X bezüglich wachsender x-Werte (o. B. d. A. habe X allgemeine Lage, Sonderfälle wie z. B. $x_i = x_j$ für $i \neq j$ treten also nicht auf und müssen nicht behandelt werden)
- Setze $p_1 = p_l \in X$ als den Punkt mit dem kleinsten x-Wert und $p_r \in X$ als den Punkt mit dem größten x-Wert.
- Wende den Algorithmus zur Berechnung von UH(X) an, dabei ist UH(X) die obere konvexe Hülle von X.
- für LH(X) vertausche die Vorzeichen und benutze UH(X) noch einmal, dabei ist LH(X) die untere konvexe Hülle von X.

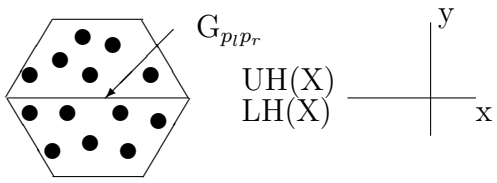


Abbildung 4.12.: TODO: Unterschrift für die Grafik

Doch wie funktioniert das nun genau? Zuerst brauchen wir den Algorithmus für UH(X):

- Eingabe ist die Punktfolge p_l=p_1, p_2, …, p_n=p_r, dabei sind die x-Werte monoton wachsend und alle p_i liegen oberhalb von $G_{p_l p_r}$ (dies läßt sich notfalls in $\mathcal{O}(n \log n)$ Zeit erreichen)
- Der Algorithmus arbeitet dann Punkt für Punkt von links nach rechts und bewahrt folgende Invariante
  1. Der Stapel S speichert Punkte x_0, x_1, …, x_t=x_{Top}, die eine Teilfolge von p_n, p_1, p_2, … sind
  2. $t \geq 2 \wedge x_0 = p_n \wedge x_1 = p_1 \wedge$ im Schritt s gilt: $x_t = p_s$, dabei ist $s \geq 2$
  3. $x_0, x_1, \ldots, x_t$ ist UH($\{p_1, \ldots, p_s\}$)
  4. $x_1, \ldots, x_t$ sind von rechts sortiert.

Algorithmus 4.5.1 : GRAHAM

Output : S

```
1  begin
2      Push (S, p_n);
3      Push (S, p_1);
4      Push (S, p_2);
5      s ← 2;
6      while s ≠ n do
7          α ← top[S];
8          β ← zweites Element aus S;
9          while (p_{s+1}, α, β) keine Linksdrehung do
10             Pop;
11             α ← β;
12             β ← neues zweites Element im Stapel
13         end
14         Push (S, p_{s+1});
15         s ← s + 1
16     end
17     Gib S aus
18 end
```

Der Drehsinn läßt sich dabei mathematisch recht einfach feststellen (siehe Abschnitt B.5), die Details sind an dieser Stelle aber weniger wichtig. Doch wie kann man das für die Berechnung der konvexen Hülle ausnutzen? Abbildung 4.13 macht dies sehr anschaulich klar. Die Punkte sind jeweils die Endpunkte der eingezeichneten Strecken und daher nicht explizit markiert. Im rechten Fall wird der Punkt p_2 für die konvexe Hülle überflüssig, da er durch p_{s+1} „überdeckt“ wird. Im linken Bild ist p_2 hingegen für die konvexe Hülle notwendig (p_2 liege oberhalb der Strecke $\overline{p_1 p_{s+1}}$). Dies verdeutlicht anschaulich, warum ein Punkt der durch POP im Laufe der Berechnung der konvexen Hülle herausfliegt, nicht nochmals angeschaut werden muß.

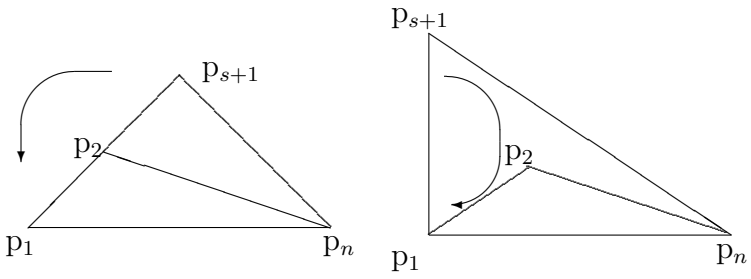


Abbildung 4.13.: Nutzen des Drehsinns für die Berechnung der konvexen Hülle

Nun bleibt noch die Analyse des Algorithmus. Bei der Analyse nach der Guthabenmethode werden Operationen mit Kosten versehen, die auf ein fiktives Bankkonto verbucht werden. Hier kostet nun jedes PUSH 2 €. Davon wird einer verbraucht und einer gespart. Von dem Ersparten werden die POP's bezahlt; dabei kostet jedes POP einen . Da wie bereits oben erwähnt jeder Punkt maximal einmal durch ein POP herausfliegen kann, sind wir fertig. Unser Guthaben reicht aus um alle möglichen POP's zu bezahlen. Mittels der Guthabenmethode ergeben sich so amortisierte Kosten von $\mathcal{O}(n)$ für die Berechnung der konvexen Hülle.

Mithilfe eines Stapels ist also ein sehr effizienter Algorithmus möglich. Allerdings sind natürlich auch Stapel kein Allheilmittel für alle Probleme der algorithmischen Goemetrie. Ein anderes wichtiges Mittel sind die sogenannten Segmentbäume.

4.6. Segmentbäume

Bereits anfangs wurde das Problem der überschneidung von Rechtecken erwähnt. Eingabe sollten Ortho-Rechtecke sein und Ausgabe ein Bericht der überschneidungen. Dazu kann man

1. alle Rechteckseiten bestimmen und dann
2. den Sgmentschnitt-Algorithmus benutzen (siehe Anhang A)

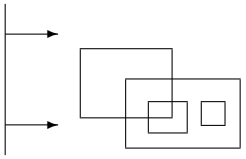
Der dabei verwendete abstrakte Datentyp Dictionary könnte mit den bisher kenngelernten Methoden z. B. RS-, AVL- oder Top-down-2-3-4-Baum realisiert werden. Wichtiger ist jetzt allerdings, daß dieses Problem auch auf das Problem der Punkteinschlüsse zurückgeführt werden kann. Die Eingabe ist dann eine Menge von Punkten und die Ausgabe sind dann alle Punkteinschlüsse (p, R) mit $p \in R$. Dabei sind mit $p_1, p_2, \ldots$ die Punkte und mit $R_1, R_2, \ldots$ die Rechtecke gemeint. Auch hier wird eine Gleitgeradenmethode benutzt; die Ereignispunkte sind die x-Koordinaten der linken und rechten Kanten und der Punkte.

(li, x, (y_1, y_2), R) (o. B. d. A. $y_1 \leq y_2$)

(re, x, (y_1, y_2), R)

(Punkt, x, y, R)

An der folgenden Skizze wird offensichtlich, daß es verschiedene Arten von überschenidungen bzw. Schnitten gibt.



4.6.1. Der Punkteinschluß-Algorithmus

Y wird zu Beginn mit $Y := \emptyset$ initalisiert. Dann wird von Ereignispunkt zu Ereignispunkt gegangen, nachdem diese nach wachsendem x sortiert wurden. Bei m Rechtecken geht dies in $\mathcal{O}(m \log m)$ Schritten.

1. Falls der aktuelle Ereignispunkt die Form (li, x, (y_1, y_2), R) hat, so setze $Y := Y \cup \{(R, [y_1, y_2])\}$ –INSERT
2. Falls der aktuelle Ereignispunkt die Form (re, x, (y_1, y_2), R) hat, so setze $Y := Y \backslash \{(R, [y_1, y_2])\}$ –DELETE
3. Falls der aktuelle Ereignispunkt die Form (Punkt, x, y, R) hat, so finde alle Intervalle in Y, die y enthalten und gib die entsprechenden Schnittpaare aus (d. h. wenn für $(R', [y_1, y_2])$ in Y gilt: $y \in [y_1, y_2]$. Prüfe, ob $R' \neq R$ ist, wenn ja, gib (R', R) aus.) –SEARCH

4.6.2. Der Segment-Baum

Der Segment-Baum dient also der Verwaltung von Intervallen, wobei die üblichen Operationen INSERT, DELETE und SEARCH mögloch sind.

Mit der Eingabe ist die Menge der y-Werte gegeben, welche nach steigendem y geordnet wird. Damit ergibt sich die Folge $y_0, y_1, \ldots, y_n$ bzw. die Folge von Elementarintervallen $[y_0, y_1], [y_1, y_2], \ldots, [y_{n-1}, y_n]$. Durch das folgende Beispiel wird dies vielleicht klarer.

Seien die y-Werte 0, 1, 3, 6, 9 als sogenannte Roatorpunkte , die zu den Elementarintervallen [0, 1], [1, 3], [3, 6] und [6, 9] führen und die Rechtecke wie in Abbildung 4.14 gegeben. Daraus resultiert dann der Baum wie in Abbildung 4.15.

Stimmt das?

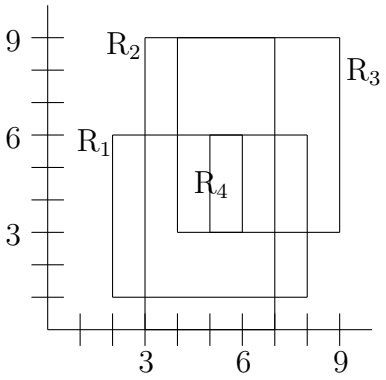


Abbildung 4.14.: Beispiel für überschneidung von Rechtecken

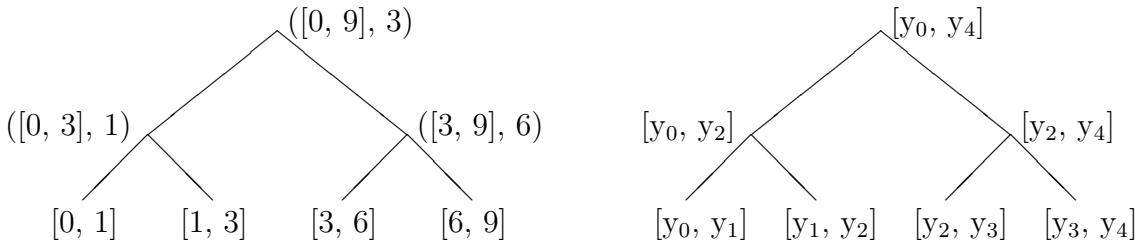


Abbildung 4.15.: Beispiel für einen Segment-Baum

a) $Y = Y \cup \{(R_1, [1,6])\} = \{(R_1, [1,6])\}$ $\quad$ b) $Y = \{(R_1, [1,6]), (R_2, [0,9])\}$

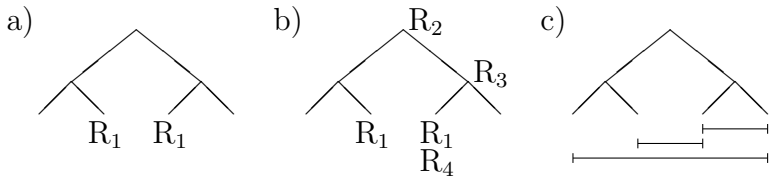


Abbildung 4.16.: Beispiel für einen Segment-Baum im Aufbau

Stimmt das so? Im Skript stand folgendes: (Punkt, x, y, R) → Suche nach y=(4,5), merke die Beschriftungen auf dem Suchpfad von Rechtecken, OUTPUT (R_4, R_1) (R_4, R_2) (R_4, R_3). Wenn aber nach überschendiungen im Intervall y=[4, 5] gesucht wird fehlen doch noch drei überschneidungen, ich werde daraus nicht schlau.

Aus Abbildung 4.14 geht der Baum hervor, der rechts in der Abbildung 4.15 zu sehen ist. Der dritte Wert jedes Tupels in jedem Knoten gibt dabei das Maximum der Tupel im linken Teilbaum dieses Knotens an. Etwas verständlicher, aber formal falsch, ist folgende Formulierung: Der dritte Wert jedes Knotens gibt den maximalen Wert seines linken Teilbaumes an. Der linke Teil von Abbildung 4.15 deutet an, wie die Struktur im allgemeinen Fall bzw. als „Leerstruktur“ aussieht. In der untersten Abbildung zeigen a) und b) wie die Struktur aufgebaut wird. Teil c) verdeutlicht nochmal die überlappung der Intervallgrenzen (im obigen Beispiel!) und macht gleichzeitig folgenden Satz klar.

Satz 4.9
Für jedes Rechteck gibt es pro Level maximal zwei Einträge

Doch wie funktioniert die Suche und wieviel Rechenzeit wird größenordnunsmäßig benötigt, um alle überlappungen herauszufinden und auszugeben?

Falls nach (Punkt, x, y, R) gesucht wird, wird im im Baum getestet, in welchen Intervallen (x,y) liegt.

Für die Suche werden die Beschriftungen auf dem Suchpfad von Rechtecken gemerkt. Für den Baum gilt der Satz über höhenbalancierte Suchbäume. Damit funktioniert das Suchen in $\mathcal{O}(\log m)$ Zeit, und die Ausgabe hat die Größe $\mathcal{O}(k + \log m)$, dabei ist k die

Anzahl der Schnitte. Für das Insert wird eine passende rekursive Prozedur geschrieben, die den Suchpfad abtestet und dei Eintragungen vornimmt; dies geht in ebenfalls in $\mathcal{O}(\log m)$ Zeit. Damit haben wir einen output-sensitiven Algorithmus mit einer Gesamtlaufzeit von $\mathcal{O}(k + m \log m)$, dazu siehe auch den Anfang des Skriptes und Anhang A.

5. Verwaltung von Mengen – kompliziertere Datenstrukturen

Die Operationen Make-Heap(), Insert(H, x), Min(H) und Extract-Min(H) sollten an dieser Stelle hinreichend bekannt sein, vielleicht ist auch Union(H_1, H_2) schon bekannt. Diese für die Verwaltung von Mengen wichtige Operation vereinigt, wie der Name bereits sagt, zwei Heaps H_1 und H_2 zu einem Heap H. H enthält die Knoten von H_1 und H_2, die bei dieser Operation zerstört werden (Stichwort „Mergeable Heaps“). Zusätzlich wird noch Decrease-Key(H, x, k) benutzt.

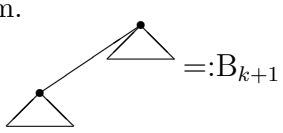
Bei Binär-Heaps funktionieren Make-Heap (Build-Heap) und Min in $\mathcal{O}(1)$ und Insert und Extract-Min in $\mathcal{O}(\log n)$. Allerdings benötigt Union $\mathcal{O}(n)$. Damit unterstützen Binär-Heaps kein Union! Diese wichtige Operation wird aber von **Binomial-Heaps** unterstützt.

5.1. Binomialbäume

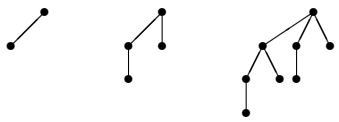
?kleiner Kreis

Definition 5.1 (Binomialbäume)

1. $B_0 :=$ ist Binomialbaum.
2. $B_k \to B_{k+1}$ mit $k \geq 0$.



Beispiel •



Im Beispiel sieht man von links nach rechts B_0, B_1, B_2 und den B_3.

Bei Bäumen gilt es verschiedene Typen zu unterscheiden, es gibt Wurzelbäume, geordnete Bäume und Positionsbäume. Sei nun mit (a, nil, b) der Baum bezeichnet, in dem in den Söhnen der Wurzel die Werte (a, nil, b) in dieser Reihenfolge von rechts nach links gespeichert sind.

Dann gilt für den Wurzelbaum, daß $(a, nil, b) = (nil, a, b)$ ist. In einem geordneten Baum hingegen ist $(a, b, c) \neq (b, a, c)$ weil die Reihenfolge der Söhne relevant ist. In einem Positionsbaum wieder ist $(a, nil, b) \neq (nil, a, b)$ weil danach geguckt wird, welche Platzstellen belegt sind und welche nicht.

nil im Mathesatz. Korrigieren

Definition 5.2 (Binomialbäume)
B_0 ist der Baum mit genau einem Knoten. Für $k \geq 0$ erhält man B_{k+1} aus B_k dadurch, daß man zu einem B_k einen weiteren Sohn an seine Wurzel als zusätzlichen linkesten Sohn hängt, dieser ist ebenfalls Wurzel eines B_k.

Definition 5.3

- Ein freier Baum ist eine ungerichteter zusammenhängender kreisfreier Graph.
- Ein Wurzelbaum (B,x_0) ist ein freier Baum B mit dem Knoten x_0 als „ROOT“.
- Ein Wald ist ein ungerichteter kreisfreier Graph.

5.2. Binomial-Heaps

Satz 5.1
Im Zusammenhang mit Binomial-Heaps wird auch der folgende Satz noch benutzt werden:

$$\binom{n}{i} = \binom{n-1}{i} + \binom{n-1}{i-1}$$

Satz 5.2 (über die eindeutige g-adische Darstellung natürlicher Zahlen)
(für $g \in \mathbb{N} : g \geq 2$)

$n \in \mathbb{N} \rightarrow$ Darstellung eindeutig, $n = \sum_{i=0}^{m} a_i g^i, a_i \in \mathbb{N}$

Mit g=2 erhalten wir die Binärdarstellung natürlicher Zahlen.

Definition 5.4 (Binomialheap)
Ein Binomialheap ist ein Wald von Binomialbäumen, der zusätzlich folgende Bedingungen erfüllt:

1. Heap-Eigenschaft = alle Bäume sind heap-geordnet, d. h. der Schlüsselwert jedes Knotens ist größer oder gleich dem Schlüsselwert des Vaters
2. Einzigkeitseigenschaft = Von jedem Binomialbaum ist maximal ein Exemplar da, d. h. zu einem gegebenen Wurzelgrad gibt es maximal einen Binomialbaum im Heap.

Satz 5.3 (Struktursatz über Binomialbäume)
1. Im Baum gibt es 2^n Knoten.
2. Die Höhe des Baumes B_n ist n

3. Es gibt genau $\binom{n}{i}$ Knoten der Tiefe i im Baum B_n
4. In B_n hat die Wurzel den maximalen Grad (degree) und die Söhne sind von rechts nach links nach wachsendem Grad geordnet.

Definition 5.5
$d(n,i) :=$ Anzahl der Knoten der Tiefe i im B_n

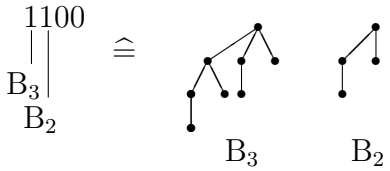
Beweis:
1. I. A. $n = 0$ trivial, $n-1 \Rightarrow n, 2^{n-1} + 2^{n-1} = 2^n$
2. $n = 0$ trivial, $n-1 \Rightarrow n$ trivial
3. IA. trivial $n = 0$

   IS. $n-1 \Rightarrow n$ Setzen die Gültigkeit von 3) für $n-1$ alle i voraus und zeigen sie dann für n und alle i

   $d(n,i) = d(n-1,i) + d(n-1,i-1) = \binom{n-1}{i} + \binom{n-1}{i-1} =_{HS} \binom{n}{i}$ q. e. d.

Es gibt zu $n \in \mathbb{N}$ *genau* (von der Struktur her) einen Binomialheap, der genau die n Knoten speichert.

Beweis: (Eindeutigkeit der Binomialdarstellung von n)
$n = 12$ q. e. d.

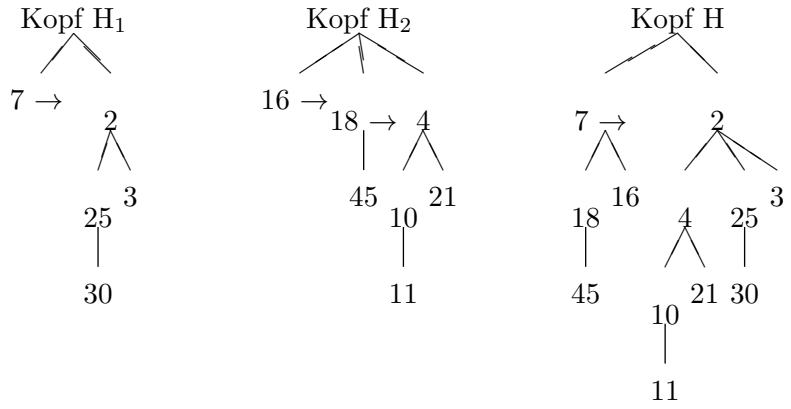


Für n Werte haben wir $\mathcal{O}(\log n)$ (genau $\lfloor \log n \rfloor + 1$) Binomialbäume. Dies ist auch intuitiv verständlich, da jede Zahl $n \in \mathbb{Z}$ zur Darstellung $\log n$ Stellen braucht. Die Eindeutigkeit der Struktur des Heaps wird auch hier wieder klar, da jede Zahl ($\in \mathbb{Z}$) eindeutig als Binärzahl dargestellt werden kann. Was passiert nun aber, wenn zwei Heaps zusammengeführt werden?

5.3. Union

Bei der Zusammenführung (Union) zweier Heaps H_1 und H_2 wird ein neuer Heap H geschaffen, der alle Knoten enthält, H_1 und H_2 werden dabei zerstört.

Leider ist wegen eines technischen Problems die folgende Darstellung nicht ganz korrekt, der Kopf jedes Heaps zeigt nur auf das erste Element der Wurzelliste.



Beim UNION werden zuerst die Wurzellisten gemischt. Dann werden, angefangen bei den kleinsten, Binomialbäume mit gleicher Knotenanzahl zusammengefaßt. Es werden also zwei B_i zu einem B_{i+1} transformiert.

Jeder Knoten im Baum hat dabei folgende Felder:

- Einen Verweis auf den Vater
- Den Schlüsselwert
- Den Grad des Knotens
- Einen Verweis auf seinen linkesten Sohn
- Einen Verweis auf seinen rechten Bruder

UNION(H_1, H_2)

```
1  H:= Make-Heap()
2  Head[H]:=MERGE(H1, H2)
3  Zerstöre die Objekte H1, H2 (nicht die entspr. Listen)
4  if Head[H]=Nil then
5    return H
6  prev-x:=Nil
7  x:=Head[H]
8  next-x:=reB[x]
9  while next-x ≠ Nil do
10   if (degree[x] ≠ degree[next-x] OR (reB[next-x] ≠ Nil AND
11   degree[reB[next-x]]=degree[x])) then
12     prev-x:=x
13     x:=next-x
14   else if (key[x]≤key[next-x]) then
15     reB[x]:=reB[next-x]
16     LINK[next-x, x]
17     else if (prev-x=nil) then
18       Head[H]:=next-x
19       else reB[prev-x]:=next-x
20       LINK[x, next-x]
21       x:=next-x
22   next-x:=reB[x]
23 return H
```

Priority-Queues={MAKE-HEAP, INSERT, UNION, EXTRACT-MAX, DECREASE-KEY, MIN}; MAKE-HEAP erfordert nur $\mathcal{O}(1)$, der Rest geht in $\mathcal{O}(\log n)$.

INSERT(H, x)

```
1  H1:= Make-Heap()
2  p[x]:=Nil
3  Sohn[x]:=Nil
4  reB[x]:=Nil
5  degree[x]:=0
6  Head[H1]:=x
7  UNION(H, H1)
```

Wie funktioniert nun das Entfernen des Minimums? Zuerst wird das Minimum raus-

geworfen, dann werden die Söhne des betroffenen Knotens in umgekehrter Ordnung in eine geordnete Liste H_1 gebracht und anschließend werden mittels UNON H und H_1 zusammengemischt. Dies klappt logischerweise in $\mathcal{O}(\log n)$, da die Wurzellisten durch $\mathcal{O}(\log n)$ in ihrer Länge beschränkt sind (bei n Werten im Heap). Damit arbeitet auch EXTRACT-MIN in $\mathcal{O}(\log n)$.

DECREASE-KEY(H, x, k)

```
1 if k>key[x] then
2   Fehler (→ Abbruch)
3 key[x]:=k
4 y:=x
5 z:=p[y]
6 while (z≠ Nil AND key[y]<key[z]) do
7 vertausche key[y] und key[z]
8 y:=z
9 z:=p[y]
```

Satz 5.4
Die linker-Sohn-rechter-Bruder-Darstellung für Binomial-Heaps ermöglicht in der angegebenen Form für Prioritätswarteschlangen logarithmische Laufzeit.

Merke: Die Länge der Wurzelliste bei einem Binomial-Heap ist logarithmisch.

Bisher war immer leicht verständlich, wie die Schranke für die Komplexität eines Algorithmus zustande kam, doch bei komplizierteren Datenstrukturen ist das nicht mehr immer so einfach. Manche mehrfach ausgeführte Schritte erfordern am Anfang viel Operationen, am Ende aber wenig. Dies erschwert die Angabe einer Komplexität und deswegen werden im Folgenden drei Methoden zur Kostenabschätzung vorgestellt.

5.4. Amortisierte Kosten

Es gibt drei Methoden um amortisierte Kosten abzuschätzen, eine davon wurde bereits im Abschnitt zur konvexen Hülle benutzt.

- die Aggregats-Methode
- die Guthaben-Methode
- die Potential-Methode

Ziel aller drei Methoden ist es, auf die Kosten einer Operation im gesamten Algorithmus zu kommen. Dies bietet sich z. B. an, falls eine Operation im worst-case in einem Schritt des Algorithmus eine Komplexität von $\mathcal{O}(n)$ haben kann, aber auch über den gesamten Algorithmus hinweg bei n Schritten nicht mehr als $\mathcal{O}(n)$ Aufwand erfordert. Dann kommt mittels der Analyse zu den amortisierten Kosten $\mathcal{O}(1)$, zu einer Art Durchschnittskosten im Stile von $\frac{T(n)}{n}$.

Bei den letzten beiden Methoden erhält man Kosten AK(i)=AK$_i$ für die Operationen. Die aktuellen (wirklichen) Kosten werden mit K(i)=K$_i$ bezeichnet und werden können relativ frei gewählt werden. In den meisten Fällen ist $K_i = \mathcal{O}(1)$, dabei muß aber gelten:

$$\sum_{i=1}^{n} AK_i \geq \sum_{i=1}^{n} K_i$$

Damit kann gefolgert werden, daß jede obere Schranke für die amortisierten Kosten AK auch eine obere Schranke für die interessierenden tatsächlichen Kosten ist.

Im Fall des Graham-Scans ergeben sich amortsierte Kosten von $\mathcal{O}(1)$ für das Multi-Pop, obwohl es schon in einem Schritt $\mathcal{O}(n)$ dauern kann. Mittels dieser amortisierten Kosten erhält man $\sum AK_i = \mathcal{O}(n) \Rightarrow \sum K_i = \mathcal{O}(n)$, also letztlich, daß der Algorithmus in $\mathcal{O}(n)$ funktioniert.

5.4.1. Die Potentialmethode

Die Idee ist, daß jede Datenstruktur DS mit einem Potential Φ bezeichnet wird. Es ist DS(0)=DS$_0$ und nach der i-ten Operation DS(i)=DS$_i$. Das Potential wird definiert als $\Phi(DS(i)) =: \Phi_i = \Phi(i)$, wobei es natürlich eine Folge von mindestens i Operationen geben muß.

Die Kosten ergeben sich als Differenz der Potentiale der Datenstruktur von zwei aufeinander folgenden Zeitpunkten. Die Kosten der i-ten Operation ergeben sich als $\Phi_i - \Phi_{i-1}$.

Hierbei muß

$$\sum_{i=1}^{n} AK_i = \sum_{i=1}^{n} (K_i + \Phi_i - \Phi_{i-1})$$

gelten.

Dies (Teleskopsumme, siehe Abschnitt B.6) läßt sich zu

$$\sum_{i=1}^{n} AK_i = \sum_{i=1}^{n} K_i + \Phi_n - \Phi_0$$

umformen.

Als drittes muß ($\Phi_n - \Phi_0 \geq 0$) sein, speziell $\Phi_n \geq 0$, falls $\Phi_0 = 0$. Damit folgt $\sum AK_i \geq \sum K_i$.

Beispiel Graham-Scan

Definition 5.6
$\Phi(\mathrm{DS}(i))$=Anzahl der Elemente im Stapel, $\Phi_i \geq 0, \Phi_0 = 0$

Die amortisierten Kosten für PUSH sind AK_i=K_i+$\Phi_i - \Phi_{i-1} = 1 + 1 = 2$ und für POP AK_i=K_i+$\Phi_i - \Phi_{i-1} = 1 + (-1) = 0$.

Woher ergibt sich das? Nun PUSH heißt ein Element einfügen $\rightarrow \Phi_i$ hat n Elemente, Φ_{i-1} hat n-1 Elemente $\rightarrow \Phi_i - \Phi_{i-1}$=1. Bei POP hat umgekehrt Φ_{i-1} n Elemente und Φ_i n-1 Elemente, damit ist dort $\Phi_i - \Phi_{i-1} = n - 1 - n = -1$.

5.5. Fibonacci-Heaps

Bei den Binomial-Heaps können wegen der Ordnung alle Söhne auf bequeme Weise angesprochen werden, bei den Fibonacci-Heaps fehlt diese Ordnung. Dafür hat jeder Knoten Zeiger auf den linken und den rechten Bruder, die Söhne eines Knotens sind also durch eine doppelt verkette Liste verknüpft. Mit $\min[H]$ kann auf das Minimum der Liste zugegriffen werden, $n[h]$ bezeichnet die Anzahl der Knoten im Heap. Es gilt weiterhin, daß alle Bäume die Eigenschaften eines Heaps erfüllen.

Wie lange dauern Operationen mit Fibonacci-Heaps? Das Schaffen der leeren Struktur geht wieder in $\mathcal{O}(1)$, da zusätzlich zu MAKE-HEAP nur noch $n[H] = 0$ und $\min[H]$ = Nil gesetzt werden muß.

5.5.1. Einfache Operationen

Algorithmus 5.5.1 : Union (H_1, H_2)

```
   Output : H
 1 begin
 2     H ← Make-Heap;
 3     min[H] ← min[H_1, H_2];
 4     Verknüpfe die Wurzellisten von H_2 und H;
 5     if min[H_1] = NIL OR min[H_2] ≠ NIL AND min[H_2] < min[H_1] then
 6         min[H] ← min[H_2]
 7     end
 8     n[H] ← n[H_1] + n[H_2];
 9     Zerstöre Objekte H_1, H_2
10 end
```

Formal korrekt muß es nicht $\min[H]$, sondern $key[\min[H]]$ heißen. Auf diese korrekte Fehl?

Schreibweise wurde zugunsten einer klareren Darstellung verzichtet. Das Aufschneiden der Wurzellisten und Umsetzen der Zeiger (Zeile 3) geht in $\mathcal{O}(1)$. Im Allgemeinen wird der Ansatz verfolgt, Operationen so spät wie möglich auszuführen. Bei UNION wird nicht viel getan, erst bei EXTRACT-MIN.

Algorithmus 5.5.2 : Insert(H, x)

```
1  begin
2      degree[x] := 0;
3      p[x] := Nil;
4      Sohn[x] := Nil;
5      links[x] := x;
6      rechts[x] := x;
7      mark[x] := false // Bis hier neue Wurzelliste nur aus x
8      Verknüpfe mit Wurzelliste H;
9      if min[H] = Nil ODER key[x] < key[min[H]] then
10         min[H] := x
11     end
12     n[H] := n[H] + 1
13 end
```

5.5.2. Anwendung der Potentialmethode

Jetzt wird die vorher erläuterte Potentialmethode benutzt, um die Kosten zu bestimmen, dabei ist

- Φ(H)=t(H)+2m(H) mit
- t[H] als der Anzahl der Knoten in der Wurzelliste und
- m[H] als der Anzahl der markierten Knoten

Wenn nachgewiesen werden soll, daß die Potentialfunktion für Fibonacci-Heaps akzeptabel (zweckmäßig und zulässig) ist, müssen die Bedingungen erfüllt sein. In [2] wird eine andere Potentialmethode benutzt.

Für die Kosten $AK_{i,\text{INSERT}}$ des INSERT gilt:

- $t(H')=t(H)+1$
- $m(H')=m(H)$
- $\Phi_i - \Phi_{i-1} = t(H')+2m(H')-t(H)-2m(H)=1$
- $K_i = 1$
- $AK_i = K_i + \Phi_i - \Phi_{i-1} = 1 + 1 = 2 \in \mathcal{O}(1)$

Satz 5.5
INSERT hat die amortisierten Kosten $\mathcal{O}(1)$

Die Kosten von UNION sind:

$\Phi(\mathrm{H}) - \Phi(\mathrm{H}_1) - \Phi(\mathrm{H}_2) = t(\mathrm{H}) - t(\mathrm{H}_1) - t(\mathrm{H}_2) + 2m(\mathrm{H}) - 2m(\mathrm{H}_1) - 2m(\mathrm{H}_2) = 0 + 0 = 0$

Satz 5.6
UNION hat die amortisierten Kosten $\mathcal{O}(1)$, $AK_{i,\mathrm{UNION}} = K_i + 0 = 1 + 0 \in \mathcal{O}(1)$.

Mit diesen beiden sehr schnellen Operationen sind Fibonacci-Heaps sinnvoll, wenn die genannten Operationen sehr häufig vorkommen, ansonsten ist der Aufwand für die komplizierte Implementierung zu groß.

Nochmal zurück zur amortisierten Kostenanalyse: Hier wird fast nur die Potentialmethode benutzt, andere Methoden sind für uns nicht wichtig. Wesentlicher Bestandteil der Potentialmethode ist die geschickte Definition der Potentialfunktion, dabei hat man viel Freiheit, nur die Bedingung $\Phi(D_n) \geq \Phi(D_0) =_{meist} 0$ muß erfüllt sein.

Die amortisierten Kosten ergeben sich als die Summe aus den aktuellen Kosten und dem Potential der Datenstruktur. Die Bedingung ist dazu da, daß die Kostenabschätzung möglich, aber auch brauchbar ist.

5.5.3. Aufwendige Operationen

EXTRACT-MIN(H)

```
1  z:=min[H]
2  if (z≠Nil)
3    für jeden Sohn x von z do
4      füge x in die Wurzelliste ein
5      p[x]:=Nil
6    entferne z aus der Wurzelliste
7    if z=re[z] then
8      min[H]:=Nil
9    else
10     min[H]:=re[z]
11     Konsolidiere(H)
12   n[H]:=n[H]-1
13 return z
```

Beim Konsolidieren wird eine Struktur ähnlich wie bei den Binär-Heaps geschaffen, dazu gleich mehr.

LINK(H, x, k)

```
1 Entferne y aus der Wurzelliste
2 Mache y zum Sohn von x, degree[x]:=degree[x]+1
3 Mark[y]:=0
```

Der Grad von x wird erhöht, da er die Anzahl der Söhne von x zählt, es ist klar, daß dies immer in $\mathcal{O}(1)$ geht. In [2] ist hier ein kleiner Fehler.

Beim Konsolidieren besteht die Lösung in Verwendung eines Rang-Arrays. Es wird $n = n[H]$ gesetzt und $D(n)$ als der maximale Wurzelgrad definiert, dabei ist $D(n) \in \mathcal{O}(\log n)$. Das Array bekommt die Größe $D(n)$. Dann wird die Wurzelliste durchgegangen und an der entsprechenden Position im Array gespeichert. Falls an dieser Stelle schon ein Eintrg vorhanden ist, wird ein LINK durchgeführt. Dabei wird das Größere an das Kleinere (bzgl. des Wurzelwertes angehängt. Da der Wurzelgrad wächst, muß der Eintrag im Feld dann eins weiter. So wird die gesamte Wurzelliste durchgegangen und es kann am Ende keine zwei Knoten mit gleichem Wurzelgrad geben. Dazu muß immer das Minimum aktualisiert werden.

Was denn genau

Analyse

Wir haben die amortisierten Kosten, die Behauptung ist: $\mathcal{O}(D(n)) = \mathcal{O}(\log n)$. Die aktuellen Kosten sind die Kosten für den Durchlauf duch die Wurzelliste, alles andere ist hier vernachlässigbar. Also sind die aktuellen Kosten $= t(H)+D(n) = \mathcal{O}(t(H)+D(n))$ und die Potentialdifferenz ist $\leq D(n) - t(H)$

Der Abschnitt ist scheiße, verbessern!

Man zeigt schließlich, daß die amortisierten Kosten in $\mathcal{O}(D(n))$ liegen.

5.5.4. Weitere Operationen

DECREASE-KEY(H, x, k)

```
1 if k>key[x] then
2   Fehler!
3 key[x]:=k          ◁ Weiter, falls F-Heap kaputt
4 y:=p[x]
5 if (y≠Nil AND key[x]<key[y] then
6   CUT(H, x, k)
7   CASCADING-CUT(H, y)
8 if key[x]<key[min[H]] then
9   min[H]:=x
```

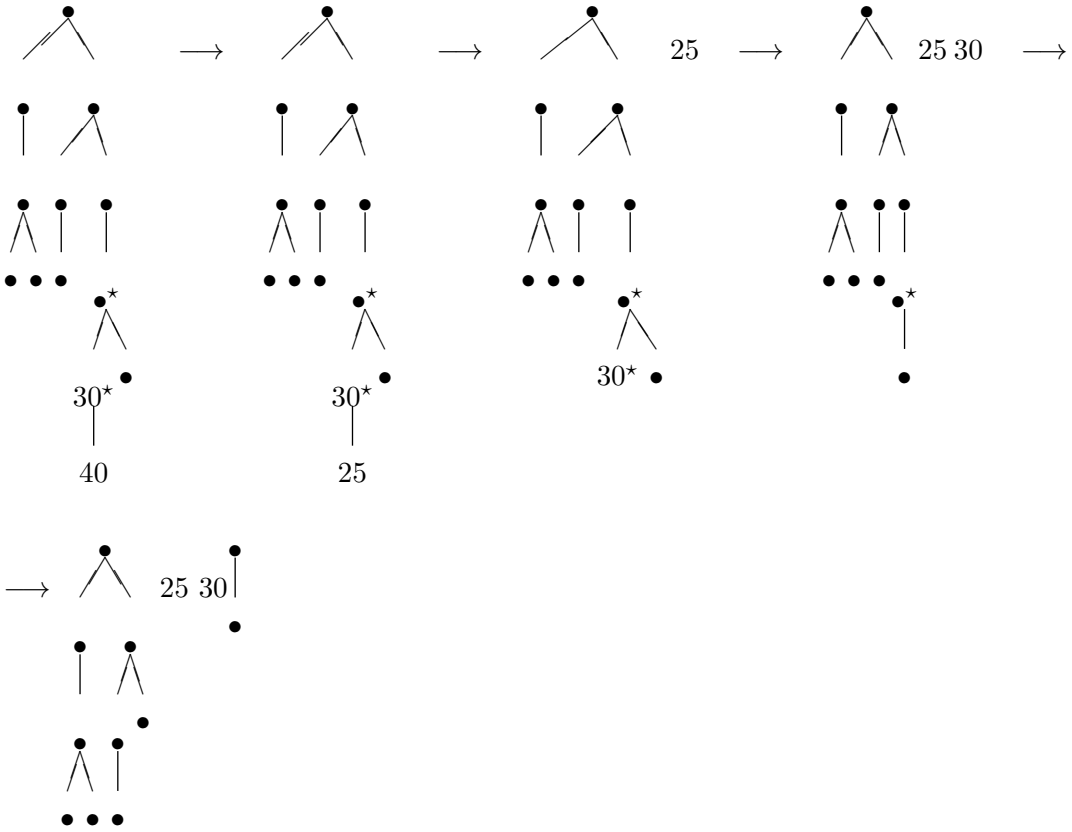
CUT(H, x, k)

```
1 Entferne x aus der Sohnliste von y
2 degree[y]:=degree[y]-1
3 Füge x zur Wurzelliste von H hinzu
4 p[x]:=Nil
5 Mark[x]:=0
```

CASCADING-CUT(H, y)

```
1 z:=p[y]
2 if (z≠Nil then)
3   if Mark[y]=0 then
4     Mark[y]:=1
5   else
6     CUT(H, y, z)
7     CASCADING-CUT(H, z)
```

Im Beispiel markiert $\star$, daß ein Knoten schon einen Sohn verloren hat.



Analyse

Die aktuellen Kosten für das Cut liegen in $\mathcal{O}(1)$, was gilt für das Cascading-Cut?

ein Cut dabei:	− eine Markierung	-2
	+ eine neue Wurzel	$+1$
	$+\mathcal{O}(1)$für das Cut selbst	$+1$
		$= 0$

Sollte das etwas O/Null heißen

Damit hat Decrease-Key $\mathcal{O}(1)$ amortisierte Kosten. Denn das einmalige Abschneiden kostet $\mathcal{O}(1)$ und die Kaskade von Cuts kostet wegen der Markierung nichts.

Satz 5.7 (Lemma 1 über F-Heaps)
Sei v Knoten eines F-Heaps. Ordnet man (in Gedanken) die Söhne von v in der zeitlichen Reihenfolge, in der sie an v angehängt wurden, so gilt: der i-te Sohn von v hat mindestens den Grad $(i-2)$.

Dies ist der Grund für die Leistungsfähigkeit und den Namen der F-Heaps

Beweis:
Damit der i-te Sohn auftaucht, müssen v und dieser Sohn den Rang $(i-1)$ gehabt haben (wenigstens Rang i, falls beide denselben Rang hatten). Wenn das unklar ist, sollte man sich das Konsolidieren noch einmal anschauen. Danach kann dieser Sohn wegen der Markierungsvorschrift maximal einen Sohn verlieren, also ist der Rang mindestens $(i-2)$. q. e. d.

Satz 5.8 (Lemma 2 über F-Heaps)
Jeder Knoten v vom Rang (Grad) k eines F-Heaps ist Wurzel eines Teilbaumes mit mindestens F_{k+2} Knoten.

Definition 5.7
$F_0 := 0, F_1 := 1, F_{k+2} := F_{k+1} + F_k$

Satz 5.9
$\mathrm{F}_{k+2} \geq \phi^k$, $\phi = \frac{1+\sqrt{5}}{2} \approx 1,6$

Satz 5.10 (Hilfssatz)
$\forall k \geq 0$ *gilt* $F_{k+2} = 1 + \sum_{i=0}^{k} F_i$

Beweis: (Beweis vom Hilfssatz durch Induktion über k)
$k = 0 : F_2 = 1 + \sum_{i=0}^{0} F_i = 1 + F_0 = 1$

IV. $F_{k+1} = 1 + \sum_{i=0}^{k-1} F_i$

$F_{k+2} = F_k + F_{k+1} = F_k + 1 + \sum_{i=0}^{k-1} F_i = 1 + \sum_{i=0}^{k} F_i$ q. e. d.

Beweis: (Lemma 2)
Sei S_k die Minimalzahl von Knoten, die Nachfolger eines Knotens v vom Rang k sind (v selbst mitgezählt).

Verweis einbauen

Habe v den Rang 0 (keinen Sohn), dann ist $S_0 = 1$ und $S_1 = 2$. Ab $k \geq 2$ gilt Lemma 1:

$$S_k \geq 2 + \sum_{i=0}^{k-2} S_i \text{ für } k \geq 2$$

Jetzt werden die Söhne von v wieder gedanklich in der Reihenfolge des Anliegens geordnet, und zusammen mit

$$\mathrm{F}_{k+2} = 1 + \sum_{i=0}^{k} F_i = 2 + \sum_{i=2}^{k} F_i$$

gilt

$$\mathrm{S}_k \geq \mathrm{F}_{k+2} \text{ für } k \geq 0 \text{ (Beweis durch Induktion)}$$

Sei v^k der Knoten v mit dem Rang k, dann gilt insgesamt: Anzahl Nachfolger$(v^k) \geq \mathrm{S}_k \geq \mathrm{F}_{k+2} \geq \phi^k \Rightarrow$ Lemma 2 q. e. d.

Daraus folgt, daß der maximale Grad (Rang) eines Knotens in einem F-Heap mit n Knoten $D(n) \in \mathcal{O}(\log n)$ ist.

Beweis:
Sei v beliebig im F-Heap gewählt und $k = \text{Rang}(v)$, dann ist n größer oder gleich der Anzahl der Nachfolger $(v) \geq \phi^k$, also $k \leq \log_\phi n \in \mathcal{O}(\log n)$. Da v beliebig ist, gilt die Ungleichung auch für den maximalen Rang k. q. e. d.

F-Heaps sind Datenstrukturen, die vor allem zur Implementierung von Prioritätswarteschlangen geeignet sind. Extract-Min braucht amortisiert logarithmische Zeit, alle anderen Operationen (Make-Heap, Union, Insert und Decrease-Key) brauchen amortisiert $\mathcal{O}(1)$ Zeit. Das Löschen klappt damit ebenfalls in logarithmischer Zeit, zuerst wird der betroffene Wert mit Decrease-Key auf $-\infty$ gesetzt und dann mit Extract-Min entfernt.

Falls in einer Anwendung oft Werte eingefügt, verändert oder zusammengeführt werden, sind F-Heaps favorisiert. Falls nur selten Werte zusammengeführt werden, bieten sich Binär-Heaps an.

6. Union-Find-Strukturen

Der abstrakte Datentyp wird durch die Menge der drei Operationen {UNION, FIND, MAKE-SET} gebildet. Union-Find-Strukturen dienen zur Verwaltung von Zerlegungen in disjunkte Mengen. Dabei bekommt jede Menge der Zerlegung ein „kanonisches Element“ zugeordnet, dieses dient als Name der Menge. Typisch für eine Union-Find-Struktur ist eine Frage wie „In welcher Menge liegt Element x?“.

Die Bedeutung von UNION und FIND sollte klar sein, doch was macht MAKE-SET(x)? Sei x Element unseres Universums, dann erzeugt MAKE-SET die Einermenge $\{x\}$ mit dem kanonischen Element x.

6.1. Darstellung von Union-Find-Strukturen im Rechner

Bei Listen ist die Zeit für das FIND ungünstig, da die ganze Liste durchlaufen werden muß. UNION dauert ebenfalls lange, da dann durch die erste Liste bis zum Ende gegangen werden muß und dann die zweite Liste an die erste gehängt wird. Hierbei wird auch schon offensichtlich, daß nach dem UNION immer ein neues kanonisches Element bestimmt werden muß. Für Union-Find-Strukturen sind „Disjoint set forests“ das Mittel der Wahl.

Zeichnung

Den Objekten wird eine Größe zugeordnet, dabei werden die Knoten mitgezählt, damit keine lineare Liste entsteht. Das UNION erfolgt zuerst nach der Größe und dann nach der Höhe, dabei wird der kürzere Wald an den längeren drangehängt.

6.2. Der minimale Spannbaum – Algorithmus von Kruskal

Ein Graph G=(V, E) sei wie üblich gegeben. Was ist dann der minimale Spannbaum („Minimum Spanning Tree“)? Dazu müssen die Kanten Gewichte haben, so wie beim Algorithmus von Dijkstra. Dann ist der minimale Spannbaum wie folgt definiert:

1. T=(V, $E^\star$) mit $E^\star \subseteq$ E
2. Alle Knoten, die in G zusammenhängend sind, sind auch in T zusammenhängend
3. Die Summe der Kantengewichte ist minimal

Zeichnung

MST(G, w)

```
1  E* := ∅
2  K:= ∅
3  Bilde Prioritätswarteschlange Q zu E bzgl. w
4  für jedes x ∈ V do
5    Make-Set(x)          ◁ K besteht aus lauter Einermengen
6  while K enthält mehr als eine Menge do
7    (v, w):=Min(Q)
8    Extract-Min
9    if Find(v)≠ Find(w) then
10     Union(v, w)          ◁ Hier reicht ganz simples Union
11     E* :=E* ∪ {(v, w)}
12 return E*
```

Der Algorithmus von Kruskal ist ein gieriger Algorithmus. In [1] steht eine leicht andere Variante als im Beispiel.

aktuelle Kante (v, w)	K:{a}, {b}, {c}, {d}, {e}, {f}, {g}, {h}, {i}, {j}, {k}
1. ⋆(i, k)	{a}, {b}, …, {h}, {i, k}
2. ⋆(h, i)	{a}, {b}, …, {h, i, k}
3. ⋆(a, b)	{a, b}, …, {h, i, k}
4. ⋆(d, e)	{a, b}, {c}, {d, e}, …, {h, i, k}
5. ⋆(e, h)	{a, b}, {c}, {d, e, h, i, k}, {f}, {g}
6. ⋆(e, g)	{a, b}, {c}, {d, e, g, h, i, k}, {f}
7. (g, i)	—
8. (g, h)	—
9. ⋆(f, h)	{a, b}, {c}, {d, e, f, g, h, i, k}
10. (d, f)	—
11. ⋆(c, f)	{a, b}, {c, d, e, f, g, h, i, k}
12. (c, d)	—
13. ⋆(a, c)	{a, b, c, d, e, f, g, h, i, k}
14. (c, b)	—
15. (b, e)	—

Zeichnung!

Das sieht ja alles schon und toll aus, wie wird soetwas aber implementiert? Sehen wir uns dazu weiter das Beispiel an.

v ∈ V	a	b	c	d	e	f	g	h	i	k
p[v]	a	a	c	i	d	f	i	i	i	i

Dies verleitet zu der einfachen Annahme, daß bei UNION(e, f) einfach e Vater und kanoisches Element der Menge wird, zu der f gehört (p[f]\coloneqq e). Da immer der kleinere Baum an den größeren angehängt werden soll (bei uns Größe nach Knotenanzahl, möglich wäre auch die Höhe als Größe zu nehmen), ist dies minimal aufwendiger.

UNION(e, f)

```
1 if size[e]<size[f] then
2   vertausche e und f
3 p[f]:=e
4 size[e]:=size[e]+size[f]
```

MAKE-SET(x)

```
1 p[x]:=x
2 size[x]:=1
```

FIND ist ebenfalls trivial, es wird einfach durch die Menge gegangen, bis das kanonische Element auftaucht, im Beispiel liefert FIND(e) $e \rightsquigarrow d \rightsquigarrow i \rightsquigarrow i$, return i.

Satz 6.1
Für Vereinigung nach Größe gilt: Ein Baum mit Höhe n hat mindestens 2^n Knoten.

Daraus folgt, daß es hier keine dünnen Bäume gibt. Der Beweis des Satzes erfolgt mittels Induktion über die Komplexität der Bäume, doch dazu muß erstmal folgendes definiert werden:

Definition 6.1 (VNG-Baum)
T ist ein VNG-Baum, genau dann wenn

1. T nur aus der Wurzel besteht oder
2. T Vereinigung von T_1 und T_2 mit size[T_2]≤size[T_1] ist.

VNG steht dabei für Vereinigung nach Größe.

BEWEIS:
IA. h=0 (Wurzelbaum) $2^h = 1$

IV. gelte für T_1, T_2 o. B. d. A. size[T_2]≤size[T_1], size[T_2]≥ 2^{h_2}, size[T_1]≥ 2^{h_1}

IB. size[T]≥ 2^h

Ist der Beweis nicht falsch? Beispiel: 2. Fall: h_1=10, h_2=4, h=11

1. Fall $h = \max(h_1, h_2) \Rightarrow$ size[T]=size[T_1]+size[T_2]$\geq 2^{h_1} + 2^{h_2} \geq 2^h$

2. Fall $h = \max(h_1, h_2) + 1 \Rightarrow$ size[T]=size[T_1]+size[T_2]$\geq$ 2·size[T_2]$\geq 2 \cdot 2^{h_2} = 2^{h_2+1} \geq 2^h$ q. e. d.

Was soll die Verarsche, wo ist der Unterschied zum vorherigen Satz? Die Existenz von dünnen Bäume?

Aus dem Satz ergibt sich, daß FIND bei Vereinigung nach Größe $\mathcal{O}(\log n)$ kostet.

Satz 6.2
Für Vereinigung nach Höhe gilt: Ein Baum der Höhe h hat mindestens 2^h Knoten.

Also geht auch hier das FIND in $\mathcal{O}(\log n)$.

Das MAKE-SET(x) funktioniert immer in $\mathcal{O}(1)$, das FIND geht in $\mathcal{O}(\log n)$ und UNION klappt in $\mathcal{O}(1)$ falls nur ein Pointer umgesetzt werden muß. In einer Idealvorstellung, in der alle Elemente auf das kanonische Element zeigen, geht FIND in $\mathcal{O}(1)$, das UNION braucht dann aber $\mathcal{O}(n)$. Solche Idealvorstellungen sind natürlich meist sehr realitätsfern und nicht praktikabel. Doch lassen sich Union-Find-Strukturen noch mit anderen Mitteln verbessern?

Ja, allerdings. Dazu bedient man sich der Pfad-Kompression und der inversen einer Funktion, die starke ähnlichkeit mit der Ackermann-Funktion aufweist. Dies zusammen führt zu dem noch folgenden Satz von Tarjan. Dazu wird die Union-Find-Struktur mit einer amortisierten Analyse auf ihre Kosten untersucht. Wir betrachten eine Folge von insgesamt m UNION-FIND-Operationen mit n MAKE-SET-Operationen dabei. Dazu brauchen wir zuerst die Ackermann-Funktion und ihre Inverse.

Definition 6.2 (Die Ackermann-Funktion und ihre Inverse)
Die Ackermann-Funktion

$$\begin{aligned} A(1,j) &= 2^j && j \geq 1 \\ A(i,1) &= A(i-1,2) && i \geq 2 \\ A(i,j) &= A(i-1, A(i,j-1)) && i,j \geq 2 \end{aligned}$$

und ihre Inverse

$$\alpha(m,n) := \min \left\{ i \geq 1 \;\middle|\; A(i, \lfloor \frac{m}{n} \rfloor) > \log n \right\}$$

Wegen obigem kann n nicht größer als m sein. Für den Grenzwert der Inversen gilt natürlich

$$\lim_{\substack{n \to \infty \\ n \leq m}} = \infty$$

Da die Funktion aber extrem schnell wächst und damit ihre Inverse extrem langsam ist in allen konkreten Anwendungen $\alpha(m,n)$ aber kleiner als 5.

Satz 6.3 (Satz von Tarjan)
Für m UNION-FIND-Operationen mit n≤m MAKE-SET-Operationen benötigt man $\Theta(m \cdot \alpha(m,n))$ Schritte, falls die Operationen in der Datenstruktur "Disjoint-Set-Forest„ mit UNION nach Größe (Höhe) und FIND mit Pfadkompression realisiert werden.

Es wird noch garnicht erläutert, was die Pfadkompression denn nun genau ist.

Noch ein Nachsatz zur Pfadkompression: Wenn wir vom Knoten zur Wurzel laufen, dann laufen wir nochmal zurück und setzen alle Zeiger auf den Vater, den wir nun kennen $\rightarrow \mathcal{O}(2x) = \mathcal{O}(x)$.

Ein weiteres Beispiel ist die Bestimmung der Zusammenhangskomponenten

Beispiel wofür?

7. Hashing

Jeder wird schon einmal vom Hashing gehört haben, daß es irgendwie mit Funktionen zu tun hat, ist wohl auch klar. Was steckt aber genau dahinter?

Für jede Funktion braucht man einen Grundbereich, eine Menge, auf denen diese Funktion operiert. Genau genommen braucht man eine „¸Ursprungsmenge“ für die Argumente und eine „Zielmenge“ für die Ergebnisse. Da aber häufig die „Ursprungsmenge“ (Urbildmenge) und die „Zielmenge“(Bildmenge) gleich sind, betrachtet man ohne weitere Erwähnung eine Menge, die aber (implizit) beides ist.

Im Falle des Hashings ist unser Universum die Urbildmenge U. Dieses enthält alle denkbaren Schlüssel, darin gibt es aber eine Teilmenge K, die nur alle auch wirklich betrachteten Schlüssel enthält.

Sehen wir uns alle Familiennamen mit maximal 20 Buchstaben an, in U sind alle Zeichenketten mit maximal 20 Buchstaben, in K aber nur alle tatsächlich vorkommenden Namen.

7.1. Perfektes Hashing

das sollte noch verbessert werden

Perfektes Hashing ist noch recht einfach, es wird die Tabelle T direkt adressiert und Werte x direkt eingefügt und gelöscht. Dafür brauchen wir eine Hash-Funktion h und einen Schlüssel k (für engl. „key“).

INSERT(T, x)

```
1 T[key[x]]:=x
```

DELETE(T, x)

```
1 T[key[x]]:=Nil
```

SEARCH(T, k)

```
1 return T[k]
```

Alle drei Operationen dauern $\mathcal{O}(1)$ Schritte, aber die Hash-Funktion h bildet aus U auf $m-1$ Werte ab ($h\colon U \to \{0, \cdots, m-1\}$), bei vielen Werten steigt damit die Wahrscheinlichkeit für Kollisionen (Abbildung auf den gleichen Funktionswert) sprunghaft an.

Dies wird durch das folgende Beispiel vielleicht klarer: Bei 23 Personen ist die Wahrscheinlichkeit, daß sie alle an verschiedenen Tagen Geburtstag haben etwas geringer als $0,5$, bei 40 Personen ist die Wahrscheinlichkeit dafür schon auf ca. $10\,\%$ gesunken.

7.2. Drei Methoden zur Behandlung der Kollisionen

1. Hashing mit Verkettung (Chaining):
   - Bei INSERT(T,x) wird x hinter dem Kopf der Liste T[h[key[x]]] eingefügt, jeder Eintrag der Hash-Tabelle kann also eine Liste beinhalten.
   - Beim SEARCH wird in der Liste T[h[k]] mit key[x]=k nach Objekt x gesucht
   - DELETE funktioniert analog zum SEARCH
2. Divisions-Rest-Methode $h(k) \coloneqq k \mod m$ (z. B. m Primzahl, die nicht zu dicht an einer Zweierpotenz liegt)
3. multiplikative Methode $h(k) \coloneqq \lfloor m(k \cdot \phi - \lfloor k \cdot \phi \rfloor) \rfloor$, dabei ist m die konstante Tabellengröße und $0 < \phi < 1$. Donald E. Knuth schlägt z. B. $\phi = \frac{sqrt5-1}{2} \approx 0,618$ vor.

Außer den drei vorgestellten Methoden gibt es noch weitere, die uns hier aber nicht interessieren. Die letzten beiden gezeigten Methoden werfen zusätzlich die Frage auf, was eine „gute“ Hashfunktion ist.

war das so gemeint?

7.3. Analyse des Hashings mit Chaining

Beim Hashing mit Chaining steht in der j-ten Zelle der Hash-Tabelle ein Zeiger auf die Liste mit n_j Elementen. Um die Anzahl der Kollisionen abzuschätzen, gibt es einen sogenannten Ladefaktor α. Jetzt wird einfaches uniformes Hashing angenommen. Dann ist die Wahrscheinlichkeit für ein gegebenes Element in eine bestimmte Zelle zu

kommen, für alle Zellen gleich. Daraus folgt auch die idealisierte Annahme, daß alle Listen gleich lang sind (sonst dauert das SEARCH schlimmstenfalls noch länger). Dann gilt

$$\mathbb{E}[n_j] = \frac{n}{m} = \alpha$$

Satz 7.1 (Theorem 1)
Wenn beim Hashing Kollisionen mit Verkettung gelöst werden, dann braucht eine erfolglose Suche $\Theta(1+\alpha)$ Zeit.

BEWEIS:
k gesucht $\rightarrow h(k)$ berechnet $\rightarrow h(k) = j$ j-te Liste durchlaufen $\Rightarrow \Theta(1+\alpha)$ q. e. d.

Satz 7.2 (Theorem 2)
Für eine erfolgreiche Suche werden ebenfalls $\Theta(1+\alpha)$ Schritte gebraucht.

Was soll die Gülle, das wird doch jetzt nochmal, bloß weniger wischiwaschi bewiesen?

BEWEIS:
Wir stellen uns einfach vor, daß bei INSERT Werte am Ende der Liste eingefügt werden. Dann brauchen wir wieder Zeit zum Berechnen des Tabelleneintrages, daran anschließend muß wieder die Liste durchlaufen werden. q. e. d.

BEWEIS:
Hier nehmen wir an, daß erfolgreich nach einem Element gesucht wird, also muß diese vorher irgendwann auch eingefügt worden sein. Sei nun dieses Element an i-ter Stelle eingefügt worden (vorher $i-1$ Elemente in der Liste), die erwartete Länge der Liste ist dann natürlich $\frac{i-1}{m}$. Jetzt seien n Elemente in der Liste, dann gilt:

$$\frac{1}{n} \cdot \sum_{i=1}^{n} (1 + \frac{i-1}{m}) = 1 + \frac{1}{n-m} \sum_{i=1}^{n} (i-1) = 1 + \frac{1}{n-m} \cdot \frac{(n-1)n}{2} = 1 + \frac{\alpha}{2} - \frac{1}{2m} \in \Theta(1+\alpha)$$

q. e. d.

Bei festem h können aber auch ungünstige Inputs problematisch werden. Dieses Problem führt zum Universal Hashing.

7.4. Universal Hashing

Festgelegte Hashfunktionen können zu ungünstigen Schlüsselmengen führen, bei zufällig gewählten Hashfunktionen werden diese ungünstigen Schlüsselmengen durch die zufällige Wahl kompensiert. Für das Einfügen und Suchen eines Wertes x muß aber immer die gleiche Hashfunktion benutzt werden, dazu wird beim Einfügen die Hashfunktion zufällig ausgewählt und dann abgespeichert.

Definition 7.1
Sei H eine Menge von Hashfunktionen, die das Universum U in $\{0, \ldots, m-1\}$ abbilden. Dann heißt H genau dann universal, wenn für jedes Paar von verschiedenen Schlüsseln $x, y \in U$ gilt:

$$|\{\, h \in H \mid h(x) = h(y) \,\}| = \frac{|H|}{m}$$

oder anders ausgedrückt:

$$\text{H } universal \leftrightarrow Wsk([h(x) = h(y)]) = \frac{1}{m} \text{ für } x, y \in U \wedge x \neq y$$

Sei $K \subseteq U$ eine Menge von Schlüsseln und $|K| = n$. Außerdem sei H eine universale Klasse von Hashfunktionen. Dann gilt der folgende Satz:

Satz 7.3
Falls h zufällig aus H ausgewählt wird, dann ist pro Schlüssel k ∈ K die mittlere Anzahl der Kollisionen (h(x)=h(y)) höchstens α $(= \frac{n}{m})$

der Beweis erfolgte lt. Skript später oder in der übung, kam der noch?

Gibt es soetwas überhaupt?

Sei m eine Primzahl, dann ist die folgende Klasse universal: Die Werte in unserem Universum seien als Bit-Strings der gleichen Länge gegeben, jeder Schlüssel x also in der Gestalt $x =< x_0, x_1, \cdots, x_r >$, wobei die Länge aller x_i gleich ist. Dabei ist $0 \leq x_i \leq m$ Voraussetzung bezüglich m bzw. r.

Definition 7.2

$$H := \{\, h = h_a \mid a \leq a_0, \ldots, a_r >, a_i \in \{0, \ldots, m-1\}, i = 0, \ldots, r \,\}$$

$$h_a(x) := \sum_{i=0}^{r} (a_i \cdot x_i) \mod m$$

Es ist ersichtlich, daß es dabei m^r verschiedene Hashfunktionen geben kann.

Satz 7.4
Die so definierte Klasse ist universal.

Für den Beweis brauchen wir noch einen Hilfssatz:

Satz 7.5 (Hilfssatz)
Falls $x \neq y$ ist, erfüllen von den m^{r+1} veschiedenen Hashfunktionen genau m^r Stück die Bedingung $h_a(x) = h_a(y) \Rightarrow$

$$\text{Wsk}([h(x) = h(y)]) = \frac{m^r}{m^{r+1}} = \frac{1}{m}$$

Beweis: (Beweis des Hilfssatzes und des Satzes)
Sei $x \neq y$ (o. B. d. A. $x_0 \neq y_0$)

$$h_a(x) = h_a(y) \Leftrightarrow \sum_{i=0}^{r}(a_i \cdot x_i) \mod m = \sum_{i=0}^{r}(a_i \cdot y_i) \mod m \leftrightarrow a_0(x_0 - y_0) = \sum_{i=1}^{r} a_i(x_i - y_i)$$

Da m eine Primzahl ist, kann durch $(x_0 - y_0)$ geteilt werden. Damit ist auch die Struktur $(\{0, \ldots, m-1\}, +_{\mod m}, -_{\mod m})$ ein algebraischer Körper. Die Umformungen führen schließlich zu

$$\star \qquad a_0 = \frac{1}{x_0 - y_0} \cdot \sum_{i=1}^{r}(a_i(x_i - y_i))$$

Damit kann für jede Wahl von $< a_1, \ldots, a_r > \in \{0, \ldots, m-1\}$ mit $\star$ das zugehörige a_0 eindeutig bestimmt werden. Bezüglich der ersten Komponente von a besteht also keine freie Wahl, oder anders ausgedrückt: a_0 ist Funktion von $(a_1, \ldots, a_r)$, davon gibt es aber genau m^r Stück. Es folgt aus $h_a(x) = h_a(y)$ insgesamt, daß jedes Tupel $(a_1, \ldots, a_r)$ das a_0 eindeutig bestimmt. Und so haben von m^{r+1} möglichen Tupeln nur m^r dei Eigenschaft $h_a(x) = h_a(y)$. Damit ist der Hilfssatz, aus dem der Satz folgt, bewiesen. q. e. d.

7.5. Open Hashing

Anfangs wurde die direkte Adressierung erwähnt, doch auch dies geht anders. Beim Open Hashing wird eine offene Adressierung verwendet, dies ist sinnvoll, falls praktisch kein Delete vorkommt. Beim Einfügen wird probiert, ob die ausgesuchte Stelle in der Tabelle frei ist. Falls sie frei ist, wird der Wert eingefügt und fertig, falls nicht wird weitergegangen. So ist $\alpha = \frac{n}{m} < 1$ möglich.

Die Proben-Sequenz (auch Probier- oder Sondierfolge) ist als Abbildung ein mathematisches Kreuzprodukt:

$$h : U \times \{0, \ldots, m-1\} \vec{auf} \{0, \ldots, m-1\}$$

Die Algorithmen für die Operationen sind dann natürlich etwas komplexer.

Der Einfachheit halber betrachten wir Schlüsselwerte k mit $k \in U$

INSERT(T, k)

```
1 i:=0
2 repeat j:=h(k,i)
3   if T[j]=Nil then
4     T[j]:=k
5     return j
6   else i:=i+1
7 until i=m
8 Fehler -- Tabelle voll
```

SEARCH(T, k)

```
1 i:=0
2 repeat j:=h(k,i)
3   if T[j]=k then
4     return j
5   else i:=i+1
6 until (T[j]=Nil OR i=m)
7 return Nil
```

Beim DELETE taucht allerdings ein Problem auf. Nil würde gesetzt werden und dann würde SEARCH stoppen und nicht mehr weiter suchen.

Was soll das jetzt, löst das die Porbleme mit dem DELETE?

lineares Sondieren	quadratrisches Sondieren	doppeltes Hashing
$h(k,i) :=$ $(h'(k) + c \cdot i) \mod m$ Sei eine leere Zelle gegeben und davor i volle Dann wird eine Zelle mit Wsk $\frac{i+1}{m}$ belegt $\Rightarrow$ Primärcluster (lange Suchzeiten)	$h(k,i) :=$ $(h'(k) + c_1 \cdot i + c_2 \cdot i^2) \mod m$ $\Rightarrow$ Sekundärcluster	$h(k,i) =$ $(h_1(k) + i \cdot h_2(k)) \mod m$ h_1 und h_2 sind wieder Hashfunktionen

7.6. Nochmal zur Annahme des (einfachen) uniformen Hashings

Beim einfachen uniformen Hashing gilt für jeden Schlüssel k, daß jeder Tabellenplatz für ihn gleichwahrscheinlich ist (unabhängig von den Plätzen anderer Schlüsselwerte). Falls k eine zufällige reelle Zahl ist, die in $[0, 1]$ gleichverteilt ist, gilt dies für $h(k) = \lfloor k \cdot m \rfloor$ Beim uniformen Hashing gilt für jeden Schlüssel k, daß jede der $m!$ Permutationen von {0, ..., m-1} als Sondierungsfolge gleichwahrscheinlich ist. Eigentlich gibt es kein uniformes Hashing, aber man kann sehr nahe an diese Forderung (und ihre Folgen!) herankommen.

A immer darauffolgende Zahl $\rightarrow m!sehrklein$

B $h'(k_1) = h(k_1, 0)$, $h'(k_2) = h(k_2, 0)$, gleiche Funktion $\rightarrow$ gleicher Platz, m verschiedene Sondierungsfolgen

C gute Wahl von h_1 und $h_2 \rightarrow$ nah am uniformen Hashing

Damit ist also Variante "C„ (Doppeltes Hashing) am dichtesten am uniformen Hashing dran, wie bestimmt man nun h_1 und h_2? Dafür gibt es keine festen Regeln, allerdings sollte

- $h_2(k)$ relativ prim zu m oder
- m eine Primzahl sein.

Ersteres wird oft eingehalten. Möglich ist eine Struktur wie

$h_1(k) = k \mod m$
$h_2(k) = 1 + (k \mod m')$ mit m' dicht an m

Beispiel

Sei also

$h_1(k) = k \mod 13$
$h_2(k) = 1 + (k \mod 11)$

Folge	79	69	98	72	50	14
h_1	1	4	7	7	11	1
$h(k, i)$	1	4	7	8	11	4

Bei geschickter Wahl von h_1 und h_2 erfüllt das doppelte Hashing die Annahme des uniformen Hashings ausreichend.

Satz 7.6 (Satz 1 zum Open Hashing)
Gegeben sei eine Open-Hash-Tabelle mit $\alpha = \frac{n}{m} < 1$. Dann ist der Erwartungswert für die Anzahl der Proben in der Sondierungsfolge bei erfolgloser Suche unter Annahme des uniformen Hashings höchstens $\mathcal{O}(\frac{1}{1-\alpha})$

Analoges gilt dann für INSERT. Je voller die Tabelle ist, umso näher ist α an 1 und umso länger dauert das SEARCH.

Satz 7.7 (Satz 2 zum das Open Hashing)
Unter der Voraussetzung von Satz 1 gilt bei erfolgreicher Suche, daß diese weniger als $\mathcal{O}(\frac{1}{\alpha} \cdot \ln \frac{1}{1-\alpha})$ Schritte braucht (die Annahme des uniformen Hashings sei erfüllt).

BEWEIS: (NUR DIE BEWEISSKIZZE)
Sei das k nach dem gesucht wird, in der Tabelle und als $(i+1)$-ter Wert eingefügt worden. Mit der Folgerung vom ersten Satz zum Open Hashing ist $\alpha = \frac{i}{m}$ beim Einfügen von k. Die erwartete Anzahl der Proben beim INSERT von k ist danach

$$\frac{1}{1-\frac{1}{m}} = \frac{m}{m-i}$$

Bei n Schlüsseln ist der Mittelwert zu bilden:

$$\frac{1}{n}\sum_{i=0}^{n-1}\frac{m}{m-i} = \frac{m}{n}\sum_{i=0}^{n-1}\frac{1}{m-i} = \frac{m}{n}(H_m - H_{m-1})$$

dabei steht H_m für die harmonische Reihe und es ist

$$H_m = \sum_{mu}^{m}\frac{1}{\mu}$$

Weiter ist

$$\frac{1}{\alpha}(H_m - H_{m-1}) = \frac{1}{\alpha}\sum_{k=m-n+1}^{m}\frac{1}{k} \leq \frac{1}{\alpha}\int_{m-n}(\frac{1}{x})dx = \frac{1}{\alpha}\ln\frac{m}{m-n} = \frac{1}{\alpha}\ln\frac{1}{1-\alpha}$$

q. e. d.

Sei z. B. die Tabelle halbvoll $\rightarrow\sim 1,387$ oder zu 90 % voll $\rightarrow\sim 2,56$. Diese Werte gelten nicht unbedingt für unsere Beispiele wie doppeltes Hashing sondern für idealisertes uniformes Hashing. Falls aber h_1 und h_2 geschickt gewählt werden (siehe unser Beispiel), liegen die tatsächlichen Werte nah bei den oben angegebenen.

Was bleibt nun als Folgerung? Falls kein DELETE erforderlich ist, wird Open Hashing benutzt, sonst Chaining.

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

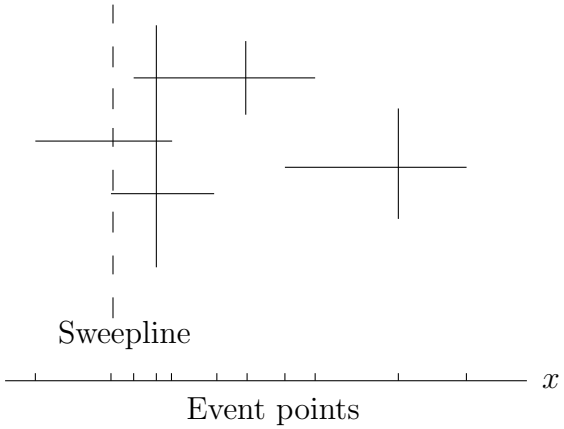


Abbildung A.1.: Liniensegmente mit Gleitgerade

Der Algorithmus nach Güting [6] gestaltet sich dann in dieser Weise:

Algorithmus 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

$$S = \{\ (x_1, (x_1, x_2, y)) \mid (x_1, x_2, y) \in H\ \} \cup \{\ (x_2, (x_1, x_2, y)) \mid (x_1, x_2, y) \in H\ \}$$
$$\cup \{\ (x, (x, y_1, y_2)) \mid (x, y_1, y_2) \in V\ \}$$

   (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 $v = (x, y_1, y_2)$:
      $A := \pi_2(\{\ w \in Y \mid w_0 \in [y_1, y_2]\ \})$; (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.

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

Die Rücksubstitution ist hier recht einfach und ergibt als Gesamtlaufzeit

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

B.2. Lösung Mastertheorem

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

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

$$
\begin{aligned}
3f(\frac{n}{4}) &\leq & cf(n)\\
3[2(\frac{n}{4}) \log(\frac{n}{4})] &\leq & c2n \log n\\
(\frac{3}{4}) \log(\frac{n}{4}) &\leq & c \log n\\
(\frac{3}{4}) \log n - \log 4 &\leq & c \log n\\
(\frac{3}{4}) \log n - 2 &\leq & c \log n\\
(\frac{3}{4}) \frac{\log n}{\log n} - \frac{2}{\log n} &\leq & c\\
(\frac{3}{4}) - \frac{2}{\log n} &\leq & c
\end{aligned}
$$

Da $lim_{n\to\infty}\frac{2}{\log n} = 0$ 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

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

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

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

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

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

B.4. Fertiger RS-Baum nach Rotation

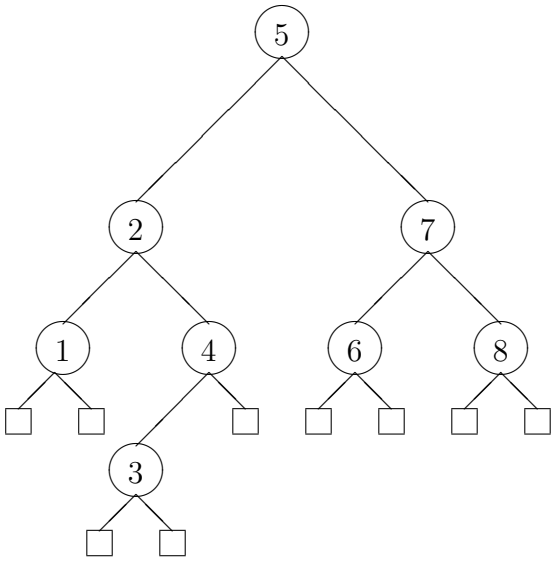


Abbildung B.1.: RS-Baum nach Rotation

B.5. Der Drehsinn

Bei vielen Problemen der algorithmischen Geometrie ist wie bei der konvexen Hülle einer Menge der Drehsinn von Punkten (bzw. Geraden) zueinander wichtig. Dieser läßt sich mittels einer Determinante berechnen, wobei letzten Endes nur wichtig ist, ob die Determinante kleiner oder größer 0 ist. Zur mathematischen Berechnung des Drehsinns. Dafür braucht man natürlich wie in Abbildung B.2 drei Punkte (bzw. zwei Geraden) zwischen denen überhaupt ein Drehsinn berechnet werden kann.

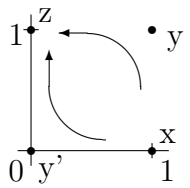


Abbildung B.2.: Beispiel zur Berechnung des Drehsinns

Nun seien $(1,0),(0,1),(1,1)$ und $(0,0)$ die Koordinaten der Punkte x, y, z und y' und wie üblich mit $x_1, x_2, y_1, y_2, z_1, z_2$ und y'_1, y'_2 bezeichnet. Wie bereits angemerkt ist das Vorzeichen, nicht der genaue Wert der Determinante, entscheidend. Diese angenommenen Werte vereinfachen nur die Rechnung:

$$\begin{vmatrix} 1 & x_1 & x_2 \\ 1 & y_1 & y_2 \\ 1 & z_1 & z_2 \end{vmatrix} = \begin{vmatrix} 1 & 1 & 0 \\ 1 & 1 & 1 \\ 1 & 0 & 1 \end{vmatrix} = 1 - 1 + 1 = 1 > 0$$

$$\begin{vmatrix} 1 & x_1 & x_2 \\ 1 & y_1' & y_2' \\ 1 & z_1 & z_2 \end{vmatrix} = \begin{vmatrix} 1 & 1 & 0 \\ 1 & 0 & 0 \\ 1 & 0 & 1 \end{vmatrix} = -1 < 0$$

In dem einen Fall handelt es sich also um eine Linksdrehung, im anderen um eine Rechtsdrehung. Der Aufwand für die Berechnung des Drehsinns liegt in $\mathcal{O}(1)$, da immer nur eine Determinante konstanter Größe berechnet werden muß.

B.6. Teleskopsummen

Ab und zu tauchen Summen ähnlich zu der in der Potentialmethode auf. Folgende Trivialität hilft manchmal sehr beim Vereinfachen:

$$\sum_{i=0}^{n}(\Phi_i - \Phi_{i-1}) = \begin{array}{ll} & \Phi_1 - \Phi_0 \\ + & \Phi_2 - \Phi_1 \\ & \vdots \\ + & \Phi_{n-1} - \Phi_{n-2} \\ + & \Phi_n - \Phi_{n-1} \end{array} = \Phi_n - \Phi_0$$

Literaturverzeichnis

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

[2] Ottmann, Widmeyer; *Algorithmen und Datenstrukturen*, Spektrum Akademischer Verlag GmbH, ISBN 3-8274-1029-0

[3] Schöning; *Algorithmik*, Spektrum Akademischer Verlag GmbH, ISBN 3-8274-1092-4

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

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

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