O-Notation (O-Kalkül, Big-Oh, Landau'sche Symbole), asymptotisches Maß, Sortieren und Algorithmen

1 Allgemeines

1.1 Ursprung

An Anfang: Die Zahlentheoretiker Bachmann und Landau

Die O-Notation wird grundsätzlich auf die Veröffentlichung *Analytische Zahlentheorie* aus dem Jahr 1892 von Peter Bachmann (1837 - 1920) zurückgeführt. Einige Zeit später hat der Zahlentheoretiker Edmund Landau (1877 - 1938) davon Gebrauch gemacht und neben ‚O' auch ‚o' betrachtet. Aufgrund dessen werden wird Notationen auch als Gebrauch Landau'scher Symbole bezeichnet. Auf Landau geht auch das Symbol $\mathbb{Z}$ für die Menge der ganzen Zahlen zurück.

1.2 Zweck

Was wir wollen: eigentlich nur π-mal-Daumen-Aussagen

Die O-Notation wird genutzt um Abschätzungen über Aufwände von Algorithmen vorzunehmen. Hierbei sticht zunächst das Bedürfnis hervor obere Schranken zu erkennen, da Probleme vorwiegend durch die hohen Aufwände für Speicher und Laufzeit entstehen. Daneben gibt es jedoch auch Maße zur Ermittlung unterer Schranken und arithmetischer Mittel. Untere Schranken können von Bedeutung sein, um zu zeigen, dass es keinen Algorithmus gibt, der schneller läuft als $g(n)$.

1.3 Mengen

1.3.1 Was Mengen sind

*Sortieren, Suchen, Vergleichen etc. fängt in der Theorie mit einer **Menge** an.*

Die Definition von Mengen geht im Allgemeinen auf Georg Cantor zurück. Dieser hat 1883 veröffentlicht:

Menge

„Eine Menge ist eine Zusammenfassung von wohlbestimmten und wohlunterschiedenen Objekten zu einem Ganzen.“

Das Zusammenfassen von mehreren Objekten, die Elemente genannt werden, führt also auf ein neues Objekt, dass dann Menge genannt wird. Wichtig für das Sortieren ist das Kriterium der Ordnung. Die oben angegebene Menge M_a genügt sehr wohl die Definition von Cantor. Dennoch liegt zunächst keine Ordnung vor. Sollten wir Elemente daraus in einer Folge sortieren, wäre unklar,

wann wir dies richtig gemacht hätten oder was überhaupt falsch wäre – in diesem Zustand ist zunächst noch alles richtig, weil es keine Einschränkungen gibt.

Wir wollen jedoch Folgen der Größe nach sortieren, dazu müssen wir den Mengen also noch etwas Beschränkendes oder Beschreibendes hinzufügen, hierbei handelt es sich um *Relationen*. Für unseren Zweck genügen zweistellige Relationen auf einer Menge M.

Um mit Mengen arbeiten, zu können, braucht man ***Relationen****.*

Exkurs – Eigenschaften von Relationen

Relationen können bestimmte Eigenschaften haben, die sich danach richten, wie eine Relationen Elemente einer Menge in Verbindung setzt. Diese Beurteilung von Relationen stellt einen recht weit gefassten Hintergrund für das Problem des Sortierens dar, allerdings tauchen diese Betrachtungen in der Literatur häufig auf. Demnach nennt man eine Relation R

Relationen können nach bestimmten Eigenschaften unterschieden werden.

Eigenschaft		
reflexiv	$\Leftrightarrow \forall a \in M$	$: aRa$
irreflexiv	$\Leftrightarrow \neg\exists a \in M$	$: aRa$
symmetrisch	$\Leftrightarrow \forall a, b \in M$	$: aRb \leftrightarrow bRa$
antisymmetrisch	$\Leftrightarrow \forall a, b \in M$	$: aRb \wedge bRa \Rightarrow a = b$
asymmetrisch	$\Leftrightarrow \forall a, b \in M$	$: aRb \rightarrow \neg(bRa)$
transitiv	$\Leftrightarrow \forall a, b, c \in M$	$: aRb \wedge bRc \rightarrow aRc$
irreflexiv	$\Leftrightarrow \neg\exists a \in M$	$: aRa$
alternativ	$\Leftrightarrow \forall a, b \in M$	$: aRb \vee bRa$
total	$\Leftrightarrow \forall a, b \in M$	$: a \neq b \Rightarrow (a, b) \in R \vee (b, a) \in R$

Das sieht möglicherweise kompliziert aus, ist jedoch nichts anderes als eine Systematik um Relationen wie ‚$=$’, ‚$<$’, ‚$\geq$’ oder ‚$\Rightarrow$’ unterscheiden zu können. Die Gleichheit ist beispielsweise reflexiv: Wenn man das Symbol ‚$=$’ so wie oben angegeben zwischen zwei $a \in M$ setzt, also $a = a$ schreibt, entsteht eine wahre Aussage. Eine andere Relation, die Ungleichheit, verhält sich hier genau anders: Zwar lässt sich $a \neq a$ schreiben, dies stellt jedoch keine wahre Aussage dar, da ein Element sehr wohl mit sich selbst identisch ist.

Neben der hier genutzten Schreibweise gibt es in der Mathematik auch hierfür Alternativen. Der Duden Mathematik schreibt etwa für die Reflexivität:

> Eine Relation heißt reflexiv, wenn $(a, a) \in$ R für alle $a \in$M.

Dies entspricht der oben angegebenen Definition für die Eigenschaft der Reflexivität, drückt sich aber formal etwas anders aus.

Zwar können die Eigenschaften von Relationen separat betrachtet werden, wie in dem obigen Beispiel, allerdings ergeben die Relationen die gemeinsam bestimmte Eigenschaften erfüllen erst bestimmte Gruppierungen unter den Relationen.

Ein bekanntes Beispiel sind etwa *Äquivalenzrelationen*. Eine Relation R ist eine Äquivalenzrelation über der Menge M, wenn sie die Eigenschaften Reflexivität. Symmetrie und Transitivität aufweist. Dies trifft z. B. auf die Relation der Gleichheit zu. Um dies zu zeigen, ersetzen wir einfach bei den obigen Definitionen jeweils das R gegen die zu untersuchende Relation:

Eine Gruppierung mit gleichen Relations-eigenschaften sind ***Äquivalenzrelationen***

Reflexivität: Für jedes $a \in \mathbb{N}$ gilt, dass $a = a$ ist. Es gibt also keine Zahl, die es schafft, nicht mit sich selbst identisch zu sein.

Symmetrie: Für zwei Elemente a, $b \in \mathbb{N}$ gilt: $a = b \leftrightarrow b = a$. Für die Gleichheit ist es irrelevant, was auf welcher Seite des Gleichheitszeichens steht, solange es sich links und rechts um das Gleiche handelt.
Transitivität: Für zwei Elemente a, b, $c \in \mathbb{N}$ gilt: $a = b \wedge b = c \rightarrow a = c$. Die Gleichheit kann paarweise für mehrere Elemente ermittelt werden.

Mit dem Ausdruck Äquivalenzrelation verbinden wir also Relationen, die die drei genannten Eigenschaften erfüllen. Wenn wir von einer Relation wissen, dass sie einer solchen Gruppierung angehört, wissen wir in der Regel auch wie wir damit umgehen dürfen.

Neben den Äquivalenzrelationen sind Ordnungsrelationen von großer Bedeutung, auch hierfür gibt es eine Reihe von Eigenschaften, die eine Relation zu erfüllen hat, um zu diesen Relationen zu gehören: Wenn eine Relation reflexiv, antisymmetrisch, und transitiv ist und auf Elementen einer Menge operiert, dann handelt es sich um eine *Ordnungsrelation*.

__Ordnungsrelation__: Operation auf Elementen einer Menge, die
- *reflexiv*
- *antisymmetrisch und*
- *transitiv*

ist.

Reflexivität: Für jedes $a \in \mathbb{N}$ gilt, dass $a \leq a$ ist. Eine Element ist kleiner *oder* gleich sich selbst (das ist wahr, weil ein Element gleich sich selbst ist – man kann sich hierbei die passende Möglichkeit aussuchen).
Antissymmetrie: Für zwei Elemente a, $b \in \mathbb{N}$ gilt: $a \leq b \leftrightarrow b \leq a \Rightarrow a = b$. Wenn ein Element kleiner *oder* gleich einem anderen ist und diese Aussage umgekehrt auch gilt, kann nur der Fall der Gleichheit vorliegen.
Transitivität: Für zwei Elemente a, b, $c \in \mathbb{N}$ gilt: $a \leq b \wedge b \leq c \rightarrow a \leq c$. Wenn ein Element kleiner oder gleich einen anderen ist und dieses andere wiederum kleiner oder gleich einem dritten ist, so ist dieses auch kleiner oder gleich dem ersten Element.

Hier kann weiter zwischen einfachen und *strengen* bzw. *totalen Ordnungsrelationen* unterschieden werden. Eine totale Ordnungsrelation ist eine Ordnungsrelation für die $\forall a, b \in M$ zusätzlich die $aRb \vee bRa$ gefordert wird (auch als weitere Eigenschaft namens *Konnexität* zu finden). Die Menge zusammen mit der Relation trägt dann den Begriff *geordnete Menge*.

Strenge Ordnungsrelation = Ordnungsrelation + $\forall a, b \in M$: $a\mathrm{R}b \vee b\mathrm{R}a$

Nun müssen wir noch die Mengen und die Relationen in Verbindung bringen. Für die Äquivalenzrelationen gilt beispielweise, dass bestimmte Termumformungen mit Elementen bestimmter Mengen möglich sind aber die Äquivalenz der Ausdrücke erhalten bleibt. Um sich dies vorstellen zu können, nehme man beispielsweise den Begriff der Datenstruktur in der Informatik: Darunter wird hier ein Datentyp mit den zugehörigen Operationen verstanden. Kombinieren wir bestimmte Mengen mit Relationen entstehen Mengen mit besonderen Eigenschaften.

Für das Sortieren und die Beurteilung der Aufwände dabei, darauf wollen wir hier letztendlich hinaus, benötigen wir eine solche *'spezialisierte'* Menge, die in der Mathematik *geordnete Menge* heißt. Da die obige Darstellung einigermaßen abstrakt ist, wollen wir hier noch eine weitere Sicht auf geordnete Mengen eröffnen. Eine geordnete Menge entsteht aus einer Menge, die lediglich dem einfachen Cantor'schen Mengenbegriff zu genügen braucht, und zwei Relationen, die auf Elemente dieser Menge anwendbar sein müssen.

1. Mit der Relation '=' können wir schon viel erreichen: Es lässt sich z.B. sagen, ob ein Element in einer Menge enthalten ist oder nicht. Es mag banal klingen, kennt man aber die Relation der Gleichheit nicht, geht nicht einmal das.

Eine elementare Relation ist also die __Gleichheit__.

2. Damit aber nicht genug: Wenn wir nur entscheiden können, ob ein Element einem anderen gleicht oder nicht, können wir immer noch nichts sortieren. Um Ordnen zu können bedarf es offensichtlich einer weiteren Relation. Die Mathematik spricht von *geordneten Mengen*, wenn neben der Relation *gleich* bzw. ‚=' eine weitere Relation *größer als* bzw. ‚>' definiert ist.

Ebenfalls wichtig: die ***Kleiner als****-Relation*

Ganz kurz ausgedrückt also: Geordnete Mengen sind Mengen, auf denen die zwei Relationen ‚gleich' und ‚kleiner als' definiert sind. Da dies von großer Relevanz ist und wir damit gelegentlich formal zu argumentieren haben, wird nun noch gezeigt, wie dies formal ausgedrückt werden kann:

Geordnete Menge

Eine Menge *M* heißt *geordnete Menge*, wenn auf ihr die Operationen ‚=' und ‚>' definiert sind, so dass gilt:

a) $a, b \in M \quad \rightarrow \quad a > b \vee b > a \vee a = b$

Das heißt: Das Ordnen von Elementen einer Größe nach geht über paarweises Vergleichen, wobei ein Paar durch die zwei Variablen *a* und *b* dargestellt ist. Hierbei kann bei einer Vergleichsoperation entweder das eine Element (eben *a*, wenn $a > b$) oder das andere (also *b*, wenn $b > a$) größer sein, oder es sind beide gleich groß (also $a = b$).

b) $a, b, c \in M \quad \rightarrow \quad a < b \wedge b < c \rightarrow a < c$

Der Gedanke aus a) lässt sich mit drei Elementen fortsetzen: Wenn ein Element kleiner als ein anderes ist, bleibt es gegenüber einem noch größeren Element (natürlich) das kleinere. Treffen beide beschriebenen Eigenschaften auf Elemente einer Menge zu, bezeichnet man sie als *geordnete Menge*. Auf geordneten Mengen lassen sich Sortierungen sinnvoll durchführen.

Geordnete Menge*: Menge +* Ordnungsrelation
Vorsicht! *Das hat noch nichts mit Sortieren zu tun*

Dies ähnelt den Eigenschaften von Relationen, was wenig erstaunlich ist, da dies lediglich diese Aspekte wiedergibt. Leider widerspricht dieser Terminus der umgangssprachlichen Belegung der Begriffe. Das Attribut *‚geordnet'* suggeriert, dass eine Tätigkeit des Ordnens bereits irgendwie ausgeführt wurde. Tatsächlich ist damit lediglich etwas gemeint, was man umgangssprachlich als *‚ordenbar'* bezeichnen würde.

Achtung: Wenn wir sortieren, erhalten wir *keine* geordnete Menge, sondern eine *sortierte Folge*.

Die Folgen, die wir sortieren können, entstammen Mengen, die Mengen selbst hingegen können nicht sortiert werden. Mengen sind Konstrukte zur Definition von Zusammengehörigen bestimmter Elemente. Dies ist entfernt vergleichbar mit Datentypen – diese können auch nicht sortiert werden. Es können aber Elemente von einem bestimmten Datentyp sortiert werden.

1.3.2 Wie man Mengen „macht"

Schreibweise von Mengen

Explizit gegebene Menge

Um die folgenden Definitionen zu verstehen, muss man mit der mathematischen Schreibweise von Mengen umgehen können. Grundsätzlich stehen Mengen in der Mathematik in geschweiften Klammern:

$M_1 = \{1, 2, ..., 8\}$

Mit dieser Schreibweise kommt man jedoch nicht weit – man muss immer alles aufzählen, was man beinhaltet haben möchte. Einerseits ist das mühselig und ineffizient, anderseits geht dies bei unendlichen Mengen nicht. Daher gibt es folgende Schreibweise, die anstelle der Elemente einer Menge zunächst die beteiligten Komponenten benennt und danach angibt, was dafür gelten soll:

$M_E = \{x \mid x \text{ hat die Eigenschaft } E\}$

Also zum Beispiel:

Deskriptiv gegebene Menge

$M_2 = \{x \mid x \in \mathbb{R} \wedge x > 0\}$

Hieraus entsteht dann die *Erfüllungsmenge*, die hier $\mathbb{R}^+$ wäre, also die Menge der positiven rationalen Zahlen ohne die 0. Man spricht auch von einer deskriptiv gegebenen Menge, im Gegensatz zu explizit gegebenen Mengen (wie oben). Wörtlich kann man das so lesen: „*x steht für ein Element der Menge* $\mathbb{R}$ *das größer als 0 sein soll.*“ Diese Vorschrift macht also nichts anderes als die Menge der rationalen Zahlen auf deren positiven Teil zu beschränken. Auch Mengen, die auf keine Elemente führen, sind möglich:

$M_3 = \{x \mid x \neq x\}$

Hieraus folgt

$M_3 = \emptyset$ (M_3 ist eine leere Menge, hierfür gibt es das Symbol ‚$\emptyset$’)

Ein weiteres Beispiel:

$M_4 = \{x \mid x \in \mathbb{N} \wedge x \leq 8\}$

Das heißt

$M_4 = \{1, 2, 3, 4, 5, 6, 7, 8\}$

Wichtige Beschreibungsmittel sind hierbei häufig *Quantoren*. Dies hängt mit dem Begriff der Quantität zusammen. Quantitativ möchte man zwei Dinge ausdrücken: „*Für alle n Elemente gilt*“ und „*es gibt (mindestens) ein Element n, für das gilt*“. Den ersten Sachverhalt drückt man mit „$\forall n$“ aus, den zweiten mit „$\exists n$“. Die folgende Menge

$\exists n$ ... := es gibt ein n ...
$\forall n$ ... := für alle n gilt ...

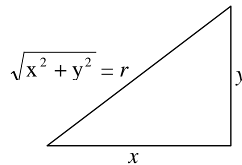
$M_5 = \{n \mid \forall k \in \mathbb{N}: k^2 = n\}$

besagt also: Es gibt eine Variable k, wird diese quadriert, ist dies ein Element der Menge M. Macht man dies also mit allen k, ist dies die Menge der quadratischen Zahlen.

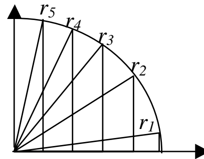
Nicht immer ist der Umfang der Elemente einer Menge gleich erkennbar, folgende Menge bedarf möglicherweise erst einiger Überlegung, wie bei M_6:

$M_6 = \{r \mid \forall x, y \in \mathbb{R}: \sqrt{x^2 + y^2} = r\}$

Der Mengenausdruck fordert, dass r aus der Wurzel der Quadrate aus $x^2 + y^2$ gebildet wird. Dies entspricht der Definition des Satzes von Pythagoras, im Falle rechtwinkliger Dreiecke kann damit aus zwei Seitenlängen die Länge der dritten Seite errechnet werden.



Nehmen wir mehrere r_i vom Ursprung eines Koordinantenkreuzes ausgehend und variieren immer jeweils die Anteile von x und y nach der Rechenvorschrift in der Menge, bildet jeder Endpunkt eines r_i einen Punkt auf einem Kreis. Damit enthält die Menge M_6 also die Menge aller Punkte auf einem Kreis mit dek Radius r.



Exkurs – Spaß unter Mathematikern

Ein besonderer Spaß unter Mathematikern scheint das Paradoxon von Bertrand Russel zu sein. In diesem Kontext taucht es immer auf, daher der Vollständigkeit halber auch hier.

Hierbei handelt es sich um eine deskriptive Mengenangabe, die eine Schwäche der Cantor'schen Definition ausnutzt und lediglich auf Widersprüche hinsichtlich der enthaltenen Elemente führt. Nehmen wir M als eine Menge aller Mengen, dann kann man auch Folgendes definieren:

$$M_7 = \{S \in M \mid S \notin S\}$$

Was könnte das heißen? M_7 ist zunächst die Menge aller Mengen, die sich *nicht* selbst als Element enthalten. Will man beantworten, ob S nun ein Element aus S ist, stößt man auf den Kern des Paradoxons: Angenommen ‚ja' – es sei $S \in S$, dann folgt jedoch $S \notin S$, was soviel wie ‚nein' bedeutet. Andersherum geht es uns auch nicht besser: Wir nehmen an, dass $S \notin S$ ist, dann müsste aber $S \in S$ sein, was nach der Annahme aber eben nicht der Fall wäre. Es lässt sich also nichts genaues Aussagen, wenn meine seine Mengen mit solchen Widersprüchen definiert. Russel hat dies seinerzeit mit einer bestimmten penetranten Offensichtlichkeit getan, um Lücken in Cantors Mengenbegriff aufzuzeigen. Für uns ist dies lediglich insofern von Bedeutung, als wir auch bei der Programmierung Mengen entwerfen können. Hier sollten wir solche Widersprüche vermeiden. Ansonsten müssen wir diese Gedanken nicht weiter vertiefen.

Die bisherigen Beispiele sind Mengen, deren Bestandteile nicht weiter evaluierbare Elemente darstellen; bislang waren die Elemente immer Zahlen verschiedener Mengen, teilweise waren die Mengen auch unendlich.

Die Mengen, die mit dem O-Kalkül beschrieben werden, sind Mengen von Funktionen. Je nach Definition der Funktionen, die eine Menge umfasst, bein-

halten solche Mengen weitere Informationen. Analysieren wir einmal den folgenden Ausdruck:

a_1	a_2	a_3	a_4	a_5
$O(f(n)) =$	$\{g: \mathbb{N} \rightarrow \mathbb{N} \mid$	$\exists n_0 > 0 \wedge \exists c > 0,$	so dass für $\forall n \geq n_0$ gilt:	$g(n) \leq c \cdot f(n)\}$

Hierin stecken folgende Aussagen:

- Es geht um die Menge $O(f(n)$ [Aussage a_1],
- diese besteht aus Funktionen g, die auf den natürlichen Zahlen operieren und auch natürliche Zahlen zurückliefern [Aussage a_2].
- Ferner gibt es ein Element n_0 und ein Element c [Aussage a_3].
- Das Element n_0 soll als eine Untergrenze gelten [Aussage a_4], von der ab Aussage a_5 erfüllt sein soll.
- Aussage 5 (im Kern die Vorschrift, aus der sich diese Menge zusammensetzt) lautet: Für jeden Wert einer Funktion $g(n)$ gibt es einen Wert einer Funktion $f(n)$, der in Verbindung mit dem erwähnten Faktor c größer oder gleich als der der Funktion $g(n)$ ist.

Zusammengesetzt: $O(f(n))$ ist eine Menge. Diese besteht aus Funktionen ($g(n)$), von denen man immer weiß, dass es bestimmte Funktionen gibt ($f(n)$), deren Werte versehen mit einer Konstanten (c) immer größer sind als die von $g(n)$. Solchen Mengen bilden die asymptotischen Maße. Dies wird im noch eingehender betrachtet.

1.3.3 Mengen von Funktionen – verschiedene Definitionen des *O*-Kalküls

Bei der Formulierung dieser Mengen gibt es gewisse Freiheitsgrade. Im Kern besagt die Definition von $O(f(n))$ zwar immer dasselbe, dennoch gibt es im Detail bestimmte Abweichungen. Die hier genutzten Definitionen folgen alle diesem Muster:

$$O(f(n)) = \{g: \mathbb{N} \rightarrow \mathbb{N} \mid \exists n_0 > 0 \wedge \exists c > 0, \text{ so dass für } \forall n \geq n_0 \text{ gilt: } g(n) \leq c \cdot f(n)\}$$

In anderen Büchern können natürlich abweichende Darstellungen genutzt werden. Relativ ähnlich ist beispielsweise die Definiton des O-Kalküls nach Ottmann und Widmayer:

$$O(f) = \{g \mid \exists c_1 > 0: \exists c_2 > 0: \forall N \in Z^+: g(N) \leq c_1 \cdot f(n) + c_2\}$$

Ottmann und Widmayer geben also den Definitonsbereich von g an, indem sie die Variable N als Element der positiven ganzen Zahlen ohne die 0 (also Z^+) festlegen., anstatt $g: N \rightarrow N$ zu schreiben. Ferner definieren sie eine zweite additiv zu benutzende Konstante c_2, nutzen dafür aber keine Formulierung wie $n \geq n_0$ in der ersten Definition. Die erste Definition fordert, dass die Funktion $f(n)$, versehen mit einem Faktor c ab einem bestimmten n_0 größer als $g(n)$ ist. Die zweite Definition fordert hingegen, das $g(N)$ für alle ganzen positiven Zahlen N ohne die 0 von einem $f(n)$ versehen mit einem konstanten Faktor c_1 und einem konstanten Summanden c_2 beschränkt werden muss. Die zweite Konstante wird nötig, da es hier keine Schwelle n_0 mehr gibt. Anstatt eine Schwelle n_0 zu erlauben, für die die Bedingung der Definition eben nicht erfüllt ist, wird dies auf

der rechten Seite mittels c_2 draufgeschlagen, dann ist die rechte Seite der Definitionsbedingung eben für alle N größer als $g(N)$.

Eine andere Variante beschreibt die Menge $O(n)$ in Form einer Äquivalenzgleichung:

Seien f, g: $\mathbb{N}_0 \rightarrow \mathbb{R}_0^+$ Funktionen. Dann ergibt sich die Menge $O(f(n))$ wie folgt: $g(n) \in O(n) \Leftrightarrow \exists n_0 \in \mathbb{N}_0, \exists c > 0, \forall n \geq n_0 : g(n) \leq c \cdot f(n)$

Dies hat abgesehen von der umgangssprachlicheren Schreibweise relativ viel Ähnlichkeit mit der ersten Definition. Erheblich anders wirkt dagegen wiederum die Definition von Sedgewick und Flajolet für das asymptotische Maß:

Für eine Funktion $f(n)$ wird $g(n) \in O(f(n))$ geschrieben, wenn gilt, dass $\lim_{n\to\infty} \frac{g(n)}{f(n)}$ beschränkt ist.

Solche und ähnliche Abweichungen wird es in vielen Quellen geben, im Kern besagen die Definitionen jedoch alle dasselbe. Alle Varianten können auf die folgenden asymptotischen Maße angewandt werden.

1.4 Definition des asymptotischen Maßes

Grundsätzlich handelt es sich bei den Aufwandsmaßen immer um *Mengen*. Wenn gesagt wird, der zeitlich Aufwand für einen Algorithmus sei quadratisch, also $T(n) = O(n^2)$, ist immer gemeint, dass $T(n)$ Element einer Menge von Algorithmen ist, deren Aufwand quadratisch ist. Keineswegs heißt das – was aber das Gleichheitszeichen impliziert – der zeitliche Aufwand ist exakt $O(n^2)$. Korrekt für diesen Sachverhalt wäre eigentlich der Ausdruck $T(n) \in O(n^2)$.

Wir wollen feststellen: der Algorithmus A gehört zu Menge der Algorithmen O(N). N steht hierbei für eine Menge von Größenordnungen wie {1, log n, n, n log n, n^2, ...)

Das wichtigste Maß bezeichnet der *Worst-Case*, also die maximale Laufzeit bei der unangenehmsten Konstellation der Elemente, die der Algorithmus bearbeiten kann. Bei Sortierproblemen ist die negativste Konstellation beispielsweise der Fall, in dem die zu sortierenden Elemente invers zu der gewünschten Konstellation vorliegen also 5, 4, 3, 2, 1 anstatt 1, 2, 3, 4, 5. Als minimal aufwändige Konstellation kann eine bereits sortierte Folge betrachtet werden. In beiden Fällen können verschiedene Algorithmen für die Lösung des Problems unterschiedliche Aufwände haben. Möglich ist auch, dass Aufwandskomponenten gegeneinander eingesetzt werden. Eine Beschleunigung der Rechenzeit kann in bestimmten Fällen durch das vermehrte Nutzen von Speicher erreicht werde. Ebenso kann häufig der Bedarf an Speicher verringert werden, indem eine Erhöhung der Laufzeit in Kauf genommen wird. Hier nun die Größen, die bei der asymptotischen Analyse betrachtet werden:

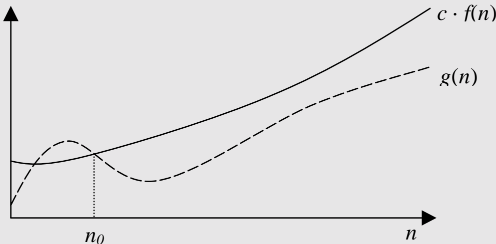
$O(n)$ – Worst-Case/obere Schranke

Elementar: Die obere Schranke

$O(f(n)) = \{g: \mathbb{N} \rightarrow \mathbb{N} \mid \exists n_0 > 0 \wedge \exists c > 0$, so dass für $\forall n \geq n_0$ gilt:

$$g(n) \leq c \cdot f(n)\}$$

Das heißt $O(f(n))$ ist eine Menge aller Funktionen $g(n)$, für die ab einem Schwellenelement n_0, dass kleiner als alle weiteren n sein soll mit Berücksichtigung einer Konstante c gilt, $g(n)$ ist immer kleiner oder gleich $c \cdot f(n)$.



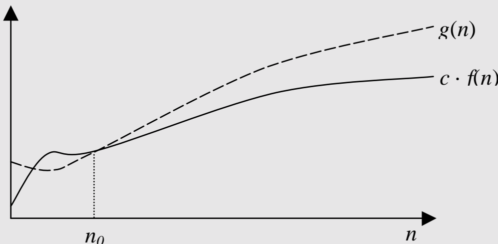
Beispiel:
Mit $g \in O(f(n))$ wird ausgedrückt, dass $g(n)$ höchstens so schnell wächst wie $f(n)$.

$\Omega(n)$ – Best-Case/untere Schranke

Manchmal von Bedeutung: Die untere Schranke

$$\Omega(f(n)) = \{g: \mathbb{N} \rightarrow \mathbb{N} \mid \exists n_0 > 0 \wedge \exists c > 0, \text{ so dass } \forall n \geq n_0 \text{ gilt: } c \cdot f(n) \leq g(n)\}$$

Das heißt $\Omega(f(n))$ ist eine Menge aller Funktionen $g(n)$, für die ab einem Schwellenelement n_0, dass kleiner als alle weiteren n sein soll mit Berücksichtigung einer Konstante c gilt, $g(n)$ ist immer größer oder gleich $c \cdot f(n)$.



Beispiel:

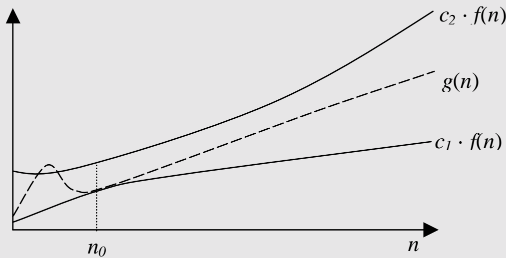
Mit $g \in \Omega(f(n))$ wird ausgedrückt, dass $g(n)$ mindestens so schnell wächst wie $f(n)$.

Θ (*n*) – Average-Case/asymptotisch enges Maß

Eher theoretisch: Das asymptotisch enge Maß

$\Theta(f(n)) = \{g: \mathbb{N} \rightarrow \mathbb{N} \mid \exists n_0 > 0 \wedge \exists c_1, c_2 > 0$, so dass $\forall n \geq n_0$:
$c_1 f(n) \leq g(n) \leq c_2 f(n)\}$

Das heißt $\Theta(f(n))$ ist eine Menge aller Funktionen $g(n)$, für die ab einem Schwellenelement n_0, dass kleiner als alle weiteren n sein soll mit Berücksichtigung zweier Konstante c_1 *und* c_2 gilt, $g(n)$ ist immer größer als $c_1 \cdot f(n)$ und kleiner als $c_2 \cdot f(n)$.



Beispiel:

Mit $g \in \Omega\ (f(n))$ wird ausgedrückt, dass $g(n)$ ebenso so schnell wächst wie $f(n)$.

O(*n*) – Real Worst-Case/echte obere Schranke

Eher theoretisch: die echte obere Schranke

$o(f(n)) = \{g: \mathbb{N} \rightarrow \mathbb{N} \mid \exists n_0 > 0$, so dass $\forall c > 0 \wedge \forall n \geq n_0$ gilt: $g(n) \leq c \cdot f(n)\}$

Das heißt $o(f(n))$ ist die Menge aller Funktionen $g(n)$, für die ab einem Schwellenelement n_0, dass kleiner als alle weiteren n sein soll für alle Konstanten c gilt, $g(n)$ ist immer kleiner oder gleich $c \cdot f(n)$.

Beispiel:

Mit $g \in o(f(n))$ wird ausgedrückt, dass $g(n)$ höchstens so schnell wächst wie $f(n)$.

o(*n*) – Real Worst-Case/echte obere Schranke

Eher theoretisch: die echte untere Schranke

$\omega(f(n)) = \{g: \mathbb{N} \rightarrow \mathbb{N} \mid \exists n_0 > 0$, so dass $\forall c > 0 \wedge \forall n \geq n_0$ gilt: $f(n) \leq c \cdot g(n)\}$

Das heißt $\omega(f(n))$ ist die Menge aller Funktionen $g(n)$, für die ab einem Schwellenelement n_0, dass kleiner als alle weiteren n sein soll, unter Berücksichtigung einer Konstante c gilt, $g(n)$ ist immer größer oder gleich $c \cdot f(n)$

Beispiel:

Mit $g \in \Omega\,(f(n))$ wird ausgedrückt, dass $g(n)$ mindestens so schnell wächst wie $f(n)$.

Wie erwähnt, ist in der Regel lediglich das erste Maß von Bedeutung. Die anderen Größen werden lediglich bei spezielleren Aspekten der Analyse von Algorithmen herangezogen.

1.5 Eigenschaften und Besonderheiten der asymptotischen Notation

Die O-Notation dient der *Abschätzung* von Aufwänden, es interessieren hierbei nur Größenordnungen und keineswegs exakte Größen, selbst wenn diese von Fall zu Fall bestimmbar sind. Der Term aus einer möglichen exakten Aufwandsbestimmung wie

$$3{,}456n^6 + 2\pi n^4 + 816n^2 + 2$$

beinhaltet hierfür lediglich die Information, dass er Element von $O(n^6)$ ist, ebenso so wie $1\,000\,000\, n^6$. Es gilt somit

$$\begin{array}{ll} 3{,}456n^6 + 2\pi n^4 + 816n^2 + 2 & \in O(n^6) \\ 1\,000\,000\, n^6 & \in O(n^6) \end{array}$$

Vorsicht – hierbei handelt es sich um eine Schreibkonvention, das Gleichheitszeichen ist eigentlich kein Gleichheitszeichen sondern ein ist-Element-von Symbol ($\in$). Aus den beiden Gleichungen kann *nicht* gefolgert werden, dass

$$3{,}456n^6 + 2\pi n^4 + 816n^2 + 2 \quad = 1\,000\,000\, n^6$$

ist. Sollte dies zutreffen, dann höchstens zufällig. Die asymptotische Notation wird immer von links nach rechts gelesen, die Seiten sind nicht vertauschbar. Die korrekte Darstellung der obigen Gleichungen ist

$$\begin{array}{ll} 3{,}456n^6 + 2\pi n^4 + 816n^2 + 2 & \in O(n^6) \\ 1\,000\,000\, n^6 & \in O(n^6) \end{array}$$

Der O-Kalkül funktioniert nur von links nach rechts

Der höhere (wenn in der Sache auch meist irrelevante) Informationsgehalt befindet sich auf der linken Seite von O-Notationen. Aus diesem Grunde wird, wenn man die $\in$-Schreibweise schon ignoriert, immer geschrieben

$n^2 = O(n^6)$ (was eigentlich auch $n^2 \in O(n^6)$ bedeutet)

und niemals

$O(n^6) = n^2$ ***[SO HERUM NICHT!!!]***

Der O-Kalkül beschreibt lediglich Obergrenzen!

Der O-Kalkül beschreibt lediglich Obergrenzen Der Sachverhalt

$$n^2 \in O(n^6)$$

ist im Sinne der asymptotischen Notation korrekt. Auch eine quadratische Funktion steigt weniger schnell als eine Funktion sechsten Grades. Ebenso korrekt ist auch

$$1 \in O(n^n)$$

Allerdings ist die Feststellung dass auch Operationen mit minimalen Aufwänden in der Menge der Algorithmen, die Aufwände von n^n verursachen, enthalten sind, trivial. Allgemein sucht man die kleinsten Schranken.

Vergleiche von Algorithmen.

Vergleicht man die Laufzeit T von Algorithmen, kann einmal das direkte Zeitverhalten sowie das asymptotische Zeitverhalten betrachtet werden. Hierbei ergeben sich folgende Unterschiede:

Ist ein Algorithmus A_1 grundsätzlich schneller als ein zweiter Algorithmus A_2, so liegt die Laufzeit $T_{A_1}(n)$ unter $T_{A_2}(n)$, also

*A_1 ist **schneller** als A_2*

$$T_{A_1}(n) \leq T_{A_2}(n)$$

Grundsätzlich sind niedrige Laufzeiten erwünscht, die Suche nach Wegen, die Laufzeiten verringern, nimmt in der Informatik entsprechend viel Raum ein. Etwas anders sieht es aus, wenn ein Algorithmus A_1 asymptotisch schneller ist als ein Algorithmus A_2: hierbei kann für bestimmte Größen n zunächst A_2 sogar kürzere Laufzeiten $T_{A_2}(n)$ haben, erst mit Wachstum von n, wird der Vorteil von A_1 sichtbar. Um dies mathematisch zu zeigen, setzt man beide Größen in ein Verhältnis und lässt n $\rightarrow \infty$ streben:

*A_1 ist **asymptotisch schneller** als A_2*

$$\lim_{n \to \infty} \frac{T_{A_1}(n)}{T_{A_2}(n)} = 0$$

Also: Für große n wächst die Laufzeit $T_{A_2}(n)$ des zweiten Algorithmus erheblich, während die des ersten tendenziell immer weiter zurückfällt. Mit immer größerem Nenner und immer kleinem Zähler wird der Wert des Bruchs tendenziell 0 (d.h. der Grenzwert liegt bei 0).

Achtung: Hiermit ist nicht gesagt, ab wann dies passiert und in welchem Ausmaß es dann passiert. Angenommen $T_{A_1}(n)$ liegt bei 10 000 n und $T_{A_2}(n)$ bei 2 n^2, dann für den asymptotischen Aufwand

Manchmal können diese Schwellen von Bedeutung sein

$$T_{A_1}(n) = 10\,000\, n \qquad = O(n) \text{ und}$$
$$T_{A_2}(n) = 2\, n_2 \qquad = O(n^2)$$

Aus der Sicht der asymptotischen Analyse gilt klar $O(n) \leq O(n^2)$. Würden reale Ergebnisse diesen konstruierten Zahlen nahe kommen, müsste sicherlich untersucht werden, in welchem Bereich n regelmäßig liegen kann. Vorteile brächte der lineare Algorithmus hier erst ab Größen von 5000 für n. Darunter wäre hier der quadratische Zeit benötigende Algorithmus zunächst vorzuziehen. Eine andere Variante in einer solchen Situation, wäre ab einer Schwelle n den Algorithmus A_1 einzusetzen, darunter jedoch A_2 zu nutzen.

1.6 Aufwandsklassen

Zwar spielen viele Elemente von Termen keine Rolle und werden vernachlässigt, bestimmte Aspekte bleiben jedoch und bilden den Kern von Aussagen über das asymptotische Verhalten. Um mit der asymptotischen Notation arbeiten zu können, muss man wissen, welche Elemente von Bedeutung sind und welche man vernachlässigen kann.

Klasse	*Lesart*
$O(1)$	Der Aufwand ist konstant und unabhängig von der Problemgröße.
$O(\log n)$	Der Aufwand wächst logarithmisch (zur Basis 2) zur Problemgröße.
$O(n)$	Der Aufwand wächst linear zur Problemgröße.
$O(n \cdot \log n)$	Der Aufwand wächst linear logarithmisch zur Problemgrö-

	ße.
$O(n^2)$	Der Aufwand wächst quadratisch zur Problemgröße.
$O(n^3)$	Der Aufwand wächst kubisch zur Problemgröße.
$O(n^k)$	Der Aufwand wächst polynomial.
$O(2^n)$	Der Aufwand wächst exponentiell (zur Basis 2) zur Problemgröße.
$O(3^n)$	Der Aufwand wächst exponentiell (zur Basis 3) zur Problemgröße.
$O(n!)$	Der Aufwand wächst gemäß Fakultätsfunktion zur Problemgröße.
$O(n^n)$	Der Aufwand wächst superexponentiell zur Problemgröße.
$O(n^{k_1^{k_2^{\cdot^{\cdot^{\cdot^{k_i}}}}}})$	Der Aufwand wächst hyperexponentiell.

Beim heutigen Stand der Technik kann als Faustregel gelten: Eine Komplexität bis hin zu $O(n^2)$ kann toleriert werden. Aufwände, die darüber liegen, sind bereits bei mittleren Eingaben problematisch und Aufwände mit exponentiellem Wachstum oder höher sind in Regel nicht zu verarbeiten (sondern allenfalls in Ausnahmenfällen).

1.7 Rechenregeln

Es gibt fünf Größen, die bei der asymptotischen Notation betrachtet werden können. Um die folgenden Rechenregeln nicht fünffach notieren zu müssen, wird eine Variable ϕ genutzt, deren Inhalt eines der fünf Symbole $\{O, \Omega, \Theta, o, \omega\}$ sein kann. D.h., für $\phi = \{O, \Omega, \Theta, o, \omega\}$ gilt

1. $c = \phi(1)$
2. $\phi(f(n)) + \phi(f(n)) = \phi(f(n))$
3. $c \cdot \phi(f(n)) = \phi(f(n))$
4. $f(n) = \phi(f(n))$
5. $\phi(f(n)) \cdot \phi(g(n)) = \phi(f(n) \cdot g(n))$
6. $\phi(\phi(f(n))) = \phi(f(n))$
7. $\phi(log_b n) = \phi(log\ n)$

Weitere Regeln:

8. $O(f(n) + g(n)) = O(max\{f(n), g(n)\})$

Für Polynome der Form $P(n) = a_0 + a_1 n + ... + a_m n^m$ gilt

$$P(n) = O(n^m)$$

Zwischen den asymptotischen Maßen bestehen weitere Zusammenhänge:

$$g(n) \in \Theta(f(N)) \quad \Leftrightarrow \quad g(n) \in O(f(n)) \wedge g(n) \in \Omega(f(n))$$

Um Element der Menge der Funktion zu sein, für die es ein asymptotisch enges Maß gibt, muss es sowohl eine obere als auch eine untere Schranke geben.

$$g(n) \in O(f(n)) \quad \Leftrightarrow \quad f(n) \in O(g(n))$$

Wenn die Funktion $f(n)$ die Funktion $g(n)$ asymptotisch nach oben beschränkt, dann beschränkt umgekehrt die Funktion $g(n)$ auch die Funktion $f(n)$ nach unten. (Nichts anderes als binsenweise: wenn f über g liegt, liegt g natürlich unter f.)

2 Aufgabentypen

2.1 Regeln des O-Kalküls zeigen

Zeigen Sie, dass gilt: $O(f(n)) + O(f(n)) \in O(f(n))$

Inhaltlich: Die Menge der Funktionen $g(n)$, die ab einem n_0 immer kleiner oder gleich einer Funktion $f(n)$, versehen mit einer Konstante $c > 0$, bleiben, verändert sich nicht, indem man diese Menge mit sich selbst zusammenführt. Elemente sind in einer Menge oder nicht. Beispiel: Die Menge der Wochentage ändert sich nicht, auch wenn man diese mehrfach aufführt, bleiben es lediglich sieben unterschiedliche und damit für eine Menge relevante Elemente.

Lösungsweg: Wir stellen uns eine beliebige Funktion $g_1(n)$ vor, die jedoch durch eine Funktion $f(n)$ mit einer Konstanten c_1 ab einem Element n_0 beschränkt ist, also auch Element der Ordnung $O(f(n))$ ist. Diese steht stellvertretend für die Menge der Funktionen, die den linken Summanden bilden. Für den rechten Summanden nehmen wir eine beliebige Funktion $g_2(n)$, der ebenfalls durch eine Funktion $f(n)$ beschränkt ist, allerdings versehen mit einer anderen Konstante c_2. Ein kurzes Brainstorming, wir haben nun folgenden Sachverhalt:

$$g_1(n) \leq c_1 \cdot f(n)$$
$$g_2(n) \leq c_2 \cdot f(n)$$

Unter diesen Umständen fassen wir das ganze per Addition zusammen:

$$g_1(n) + g_2(n) \leq c_1 \cdot f(n) + c_2 \cdot f(n)$$

Jetzt wird langsam erkennbar, dass sich nach einigen weiteren Umformungen die ursprüngliche Definition wieder herleiten lässt:

$$g_1(n) + g_2(n) \leq (c_1 + c_2) \cdot f(n)$$

Die Summe von Funktionen $g_1(n) + g_2(n)$ kann als eine umfassende Funktion $g(n)$ betrachtet werden:

$$g(n) = g_1(n) + g_2(n)$$

Die Summe zweier Konstanten $c_1 + c_2$ kann ebenso also eine umfassende Kostante c betrachtet werden:

$$c = c_1 + c_2$$

Mit den letzten beiden Umformungen gilt also

$$g(n) \leq c \cdot f(n)$$

Dies entspricht der Definition von $O(f(n))$. ■

Zeigen Sie, dass gilt: $c \cdot O(f(n)) \in O(f(n))$

Inhaltlich: Die Menge der Funktionen $O(f(n))$ verändert sich nicht durch das Zufügen eines konstanten Faktors c.

Lösungsweg: Grundsätzlich muss für eine Funktion $g(n)$ eine Beschränkung durch $c \cdot f(n)$ vorliegen, um Element der Menge $O(f(n))$ zu sein. Der folgende Zusammenhang muss gelten:

$$g(n) \leq c \cdot f(n)$$

Eine Ergänzung um eine weiter Konstante c_1 führt auf Folgendes

$$c_1 \cdot g(n) \leq c_1 \cdot c \cdot f(n)$$

Die Konstanten c_1 und c lassen sich einfach zu eine weiteren Konstanten c_2 zusammenfassen:

$$c_1 \cdot c = c_2$$

Es ergibt sich

$$c_1 \cdot g(n) \leq c_2 \cdot f(n)$$

Die Ungleichung bleibt bestehen, die Funktion $f(n)$ versehen mit einer Konstanten c (rechte Seite) beschränkt nach wie vor die Funktion $g(n)$ auf die linke Seite, auch wenn diese nun ebenfalls mit einem Faktor versehen ist. ■

Zeigen Sie, dass gilt: $O(f(n)) \cdot O(g(n)) \in O(f(n) \cdot g(n))$

Inhaltlich: Es liegen zwei Mengen von Funktionen vor, eine Menge von Funktionen $f_1(n)$, deren Aufwand in der Ordnung $O(f_1(n))$ liegt und eine Menge von Funktionen $f_2(n)$, deren Aufwand in der Ordnung $O(f_2(n))$ liegt. Eine Multiplikation der Aufwände führt auf eine neue Ordnung O, die sich aus dem Produkt der Aufwände, der beteiligten Funktionen $f_1(n)$ und $f_2(n)$ ergeben und als $O(f_1(n) \cdot f_2(n))$ betrachtet werden.

Lösungsweg: Es gilt, dass $O(f(n))$ die Menge aller Funktionen $h_1(n)$ darstellt, die durch einen konstanten Faktor c_1 und eine Funktion $f(n)$ beschränkt werden. Ebenso gilt, dass $O(g(n))$ die Menge aller Funktionen $h_2(n)$ darstellt, die durch einen konstanten Faktor c_2 und eine Funktion $g(n)$ beschränkt werden.

$$h_1(n) \leq c_1 \cdot f(n)$$
$$h_2(n) \leq c_2 \cdot g(n)$$

Das Produkt aus den Funktionen $h_1(n)$ und $h_2(n)$ kann wiederum mit dem Produkt der beschränkenden Funktionen beschränkt werden:

$$h_1(n) \cdot h_2(n) \leq c_1 \cdot f(n) \cdot c_2 \cdot g(n)$$

Durch Substitution von $h_1(n) \cdot h_2(n)$ durch $h(n)$ und Zusammenfassen der Konstanten $c_1 + c_2$ zu c ergibt sich:

$$h(n) \leq c \cdot f(n) \cdot g(n)$$

Dies entspricht einer Ordnung $O(f(n) \cdot g(n))$ ■

Zeigen Sie, dass für Polynome der Form $P(n) = a_0 + a_1 n + ... + a_m n^m$ gilt: $P(n) \in O(n^m)$

Gesagt wird also:

$$g(n) = a_0 + a_1 n + ... + a_{m-1} n^{m-1} + a_m n^m \in O(n^m)$$

Zur Erinnerung - das bedeutet:

$$g(n) \qquad \leq c \cdot f(n)$$

bzw.

$$a_0 + a_1 n + \ldots + a_m n^m \quad \leq c \cdot f(n)$$

Die Beweisidee versucht die Koeffizienten a_i zu einer Konstanten c zusammenzufassen. Hierbei wird über Ausklammern und Abschätzen versucht, die Potenzen auszusondern.

Wir beginnen mit dem Polynom:

$$g(n) = a_0 + a_1 n + \ldots + a_m n^m \qquad / \, n^m \text{ ausklammern}$$

$$= \left(\frac{a_0}{n^m} + \frac{a_1}{n^{m-1}} + \cdots + \frac{a_{m-1}}{n^1} + a_m \right) n^m$$

Nach Abschätzung unter der Vorraussetzung $n \geq 1$ ergibt sich:

$$\leq \left(a_0 + a_1 + \cdots + a_{m-1} + a_m \right) n^m$$

Hiermit liegt eine Menge von Konstanten a_i vor, die zu einer Konstanten c zusammengefasst werden können:

$$c = \sum_{i=0}^{m} a_i$$

Es lässt sich immer eine Konstante c finden, mit der $n_0 = 1$ ist. ■

2.2 Zugehörigkeit zu Aufwandsklassen bestimmen

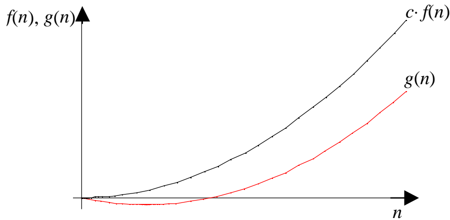
Zeigen Sie: $g(n) = 2n^2 - 8n \in O(n^2)$

Lösung: Bei diesem Aufgabentyp muss immer die Beziehung genutzt werden, die die Menge bildet. Bei $O(f(n))$ ist dies: $g(n) \leq c \cdot f(n)$. Dies ist erledigt, wenn man (1.) eine geeignete Konstante c liefern kann – es ist keineswegs nötig das minimale c zu liefern, und (2.) die Stelle n_0, von der ab die geforderte Beziehung immer erfüllt ist. Also:

$$g(n) = 2n^2 - 8n \qquad \leq c \cdot n^2 \qquad / : n^2$$

$$= 2 - \frac{8}{n} \qquad \leq c$$

Für n → ∞ strebt die linke Seite gegen den Wert 2. Damit gilt bereits mit $c = 2$, dass die rechte Seite immer größer oder gleich der linken ist. Dies ist ab $n_0 = 1$ erfüllt (theoretisch auch schon bei $n_0 = 0$, aber Vorsicht: die Definition fordert, dass $n_0 > 0$ sein muss, daher ist dies keine optimale Antwort). ■



Zeigen Sie: $g(n) = n^2 + 1000\, n \in O(n^2)$

Das $g(n)$ in $O(n^2)$ liegt, kann gezeigt werden, indem ein minimales Element n_0 ermittelt wird, ab dem $c \cdot n^2$ immer größer als $g(n)$ ist, also

$$g(n) \leq c \cdot n^2$$

gilt. Das heißt:

$$n^2 + 1000\, n \leq c \cdot n^2$$

Nehmen wir zunächst den linken Teil der Ungleichung, erhöhen n zu n^2 und setzen dies auf die rechte Seite, erhalten wir

$$n^2 + 1000\, n^2 \leq c \cdot n^2$$

Das sich die Konstante c bei dieser Abschätzung erhöht, ist kein Problem, eher im Gegenteil: Finden wir eine begrenzende Darstellung für die linke Seite, die nun aufgrund der Abschätzung leicht erhöhte Werte liefert, begrenzt diese die ursprüngliche Darstellung natürlich erst recht. Mit dieser Abschätzung ist nun jedoch ein Zusammenziehen der Elemente der linken Seite möglich:

$$1001\, n^2 \leq c \cdot n^2$$

Mit $c = 1001$ ergibt sich:

$$n^2 + 1000\, n \leq 1001\, n^2$$

Der rechte Term entspricht der Form $c \cdot n^2$ mit $c = 1001$, soweit so gut. Es bleibt die Frage, ab welchem n die abgeschätzte (eigentlich *auf*geschätzte) rechte Seite immer größere Ergebnisse als die linke Seite liefert. Wir sehen durch Versuchen, dass die geforderte Ungleichung bereits ab $n_0 = 1$ erfüllt ist.

n	$n^2 + 1000$	$1001\, n^2$	
1	1001	1001	✓
2	1004	4004	✓
3	1009	9009	✓
4	1016	16016	✓
5	1032	25025	✓

Damit ist nicht gesagt, dass c mit 1001 nun den einzigen oder auch nur den kleinsten Faktor darstellt, sondern lediglich irgendeinen, mit dem die geforderte Bedingung erfüllt ist. Versuchen wir willkürlich einmal einen anderen Faktor, z.B. 500, dann ergibt sich:

N	$n^2 + 1000$	$500\ n^2$	
1	1001	500	✗
2	1004	2000	✓
3	1009	4500	✓
4	1016	8000	✓
5	1025	12500	✓

Dies klappt offensichtlich genauso, lediglich die Schwelle n_0 liegt nun um eins höher, d.h. bei 2. ■

Ist folgende Aussage wahr? $n - 7n^4 \in O(n^4)$

Lösung:

Damit die Aussage $n - 7n^4 \in O(n^4)$ wahr ist, muss für $g(n) = n - 7n^4$ gelten:

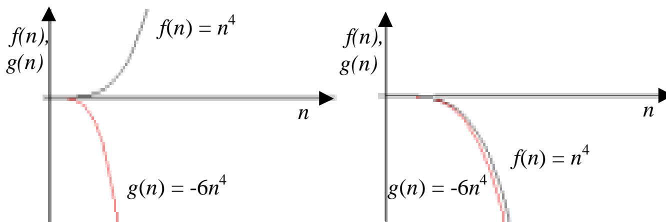
$$g(n) \leq cn^4$$

Also:

$$n - 7n^4 \leq cn^4$$
$$n^4 - 7n^4 \leq cn^4 \quad \text{/ Erhöhung der linken Seite per Abschätzung}$$
$$-6n^4 \leq cn^4 \quad \text{/ dies ermöglicht das Zusammenfassen}$$

Ab hier kann eingesetzt werden, außer Hinsehen ist nichts mehr möglich. Mit $c = 1$ und $n_0 = 1$ ist die Aussage wahr. Die theoretisch mögliche Wahl von $c = -5$ ist intuitiv zwar offensichtlich, wird aber durch die Definition nicht gedeckt – darin steht: $\exists n_0 > 0 \wedge \exists c > 0$, daher kann mit der Konstanten $c \leq 0$ nicht argumentiert werden.

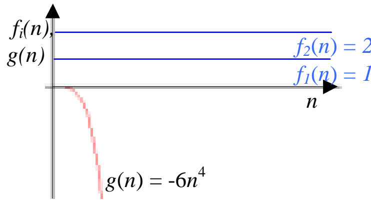
Die folgende Abbildung (links) zeigt deutlich, dass bereits n^4 die Funktion $g(n)$ beschränkt. Da $g(n)$ für den gesamten Definitionsbereich negative Werte liefert, genügen auch Funktionen wesentlich geringer Grade um $g(n)$ zu beschränken. Auch die eigentlich nicht definierte Funktion $-5n^4$ würde diesen Zweck praktisch erfüllen können (aber eben theoretisch keinesfalls).



Natürlich gilt auch

$$-6n^4 \leq 1$$

und damit $g(n) \in O(1)$. Eine Funktion $g(n)$, die negative Werte liefert, kann natürlich auch von einer konstanten Funktion $f(n)$ beschränkt werden, sofern diese über dem absoluten Maximum von $g(n)$ angesetzt ist. Die folgende Abbildung veranschaulicht dies:



In der Realität ist ein solcher Aufwand mit negativen Vorzeichen nicht vorstellbar. Dies hieße, dass ein Algorithmus Rechenschritte nicht benötigt sondern irgendwie produziert bzw. mehr Rechenleistung hinterließe, als der Rechner vor Aufruf hatte, o.ä. Solche Aufgaben existieren vereinzelt, die entsprechenden Ergebnisse sind jedoch höchstens theoretisch von Bedeutung.
Formulieren ließe sich allenfalls eine Differenz, die ein negatives Vorzeichen haben könnte. Zum Beispiel: Man hat den Aufwand eines Algorithmus A_1 von $O(n^4)$ hin zu einem Algorithmus A_2 $O(n^2)$ senken können, es bedarf also im schlimmsten Fall $-n^2$ Rechenschritte im Verhältnis zu A_1. Dies ist denkbar, aber eigentlich unschön. ■

Ist folgende Aussage wahr? $g(n) = \sin(n^2) \in O(n)$

Lösung:

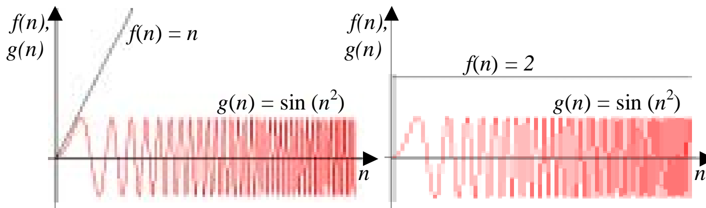
Damit die Aussage $\sin(n^2) \in O(n^4)$ wahr ist, muss für $g(n) = \sin(n^2)$ gelten:

$$g(n) \quad \leq cn$$

Also:

$$\sin(n^2) \leq cn$$

Per Definition ergibt sich der Sinus aus dem Verhältnis vom Gegenkathete zu Hypotenuse. Da die Hypotenuse die längste Seite eines Dreiecks ist, ist der maximale Wert, den der Sinus annehmen kann 1. Beim Abbilden dieses Sachverhalts auf den Einheitskreis sind auch unbeschränkt große Eingangswerte (z.B. größer als der maximale Winkel eines Kreises, also 360°) möglich und werden dann als mehrfach Kreisumdrehungen betrachtet. Die Länge des Sinus kann hingegen den Wert 1 nicht überschreiten, der Sinus ist damit beschränkt auf das Intervall [-1, 1] und ist damit $\in O(n)$, da $\forall n \in \mathbb{N} > 0$ und $\forall c \in \mathbb{N} > 0$ gilt $cn > 1$ (linke Abbildung).



Die Aufwandsklasse lässt sich allerdings nach weiter nach unten abschätzen: Da auch für alle $\forall c \in \mathbb{N} > 0$ gilt, dass $c \geq 1$ ist, gilt $g(n) \in O(1)$, siehe rechte Abbildung mit $f(n) = 2$. ■

Ist folgende Aussage wahr? $g(n) = n^2/\ln(n) \in O(n^2)$

Lösung:
Es muss also gezeigt werden, dass

$$g(n) \quad \leq cn^2$$

ist, also, dass

$$\frac{n^2}{\ln(n)} \quad \leq \mathrm{c}n^2$$

gilt. Da $\forall n > 3$ gilt:

$$\frac{1}{\ln(n)} < 1$$

ist $\forall c > 0$ und $n_0 > 3$ wahr:

$$\frac{n^2}{\ln(n)} \quad < cn^2$$

3 Sortieren

3.1 Grundsätzliches

Sortieren lassen, erfordert Vorüberlegungen

Intuitiv ist Sortieren eine klare Tätigkeit, der häufig unbewusst bestimmte Vorannahmen zugrunde liegen. Formal, um also Software zu entwickeln, die dieses Tätigkeit übernimmt, muss man diese bestimmten Voraussetzungen etwas genauer fassen. Die Menge

M_a = {a, A, **a**, *a*, ASCII(65), A, a, ~~A~~}

Elemente müssen unterschieden werden können. Wir brauchen Kriterien.

enthält zwar Elemente, die alle eindeutig und voneinander unterscheidbar sind, sich aber auf den ersten Blick nicht vernünftig sortieren lassen, schließlich sind alles Darstellungen des Buchstaben ‚a'. Der erste intuitive Ansatz Buchstaben dem Alphabet nach zu ordnen entfällt. Dies ginge erst, wenn weitere Kriterien gefunden wären, nach denen sortiert werden soll - etwa die Schönheit eines Ausdrucks, die benötigten Kurven, der Aufwand für die Repräsentation, die Größe der Darstellung, also Höhe, Breite bzw. Fläche oder ein willkürlich vereinbarter Index, der jedem Element eine eindeutige Position im Falle einer Aufreihung zuordnet.

Wie unter 1.3 besprochen, benötigt man zum Sortieren geordnete Mengen, dass heißt Mengen, für die Relationen *gleich* (=) und *größer als* (<)definiert sind.

3.2 Sortierproblem

Ein Sortierproblem liegt vor, wenn eine Folge von Elementen einer geordneten Menge gegeben ist, und eine bestimmte Permutation (d. h. eine bestimmte Anordnung der Elemente einer Menge in einer Folge) der Elemente gewünscht ist, beispielsweise *'aufsteigend der Größe nach'*. Die geordnete Menge M wird in diesem Zusammenhang auch als Universum bezeichnet.

Formal beschrieben: Es gibt eine Folge S mit Elementen $a_1,...a_n$ einer geordneten Menge M. Gesucht ist hierbei eine Permutation $P = (p_1, ..., p_n)$ von $(1, 2, ..., n)$, so dass $a_{p_1} \leq ... a_{p_n}$ gilt, p_i steht hierbei für die Position des Elements a_p in der Folge S. Sei S eine Folge mit $n = 8$ Elementen

$$S = \{49, 67, 31_1, 96, 21, 33_2, 58, 74\}$$

dann gilt für die Positionen zunächst

$$a_1 = 49, a_2 = 67, a_3 = 31_1, a_4 = 96, a_5 = 21, a_6 = 33_2, a_7 = 58 , a_8 = 74$$

Das Sortierproblem: Wir suchen bestimmte Permutationen der Indizes

Um dies der Größe nach zu sortieren, müsste also beispielsweise das kleinste Element, die 21, die das Element a_5 darstellt, an die erste Position gerückt werden, p_1 wäre also 5, da dies der Index des kleinsten Elements ist. Danach käme die 31, die Inhalt der Elemente mit den Positionen a_3 und a_6 ist. Da über das Vorkommen mehrere gleicher Zahlen noch nichts festgelegt ist, gibt es hiermit zwei Permutationen, die korrekte Lösungen darstellen:

$$P_1 = (5, 3, 6, 1, 7, 2, 8, 4)$$

und

$$P_2 = (5, 6, 3, 1, 7, 2, 8, 4)$$

Denn es gilt für P_1

$$a_{p_5} \leq a_{p_3} \leq a_{p_6} \leq a_{p_1} \leq a_{p_7} \leq a_{p_2} \leq a_{p_8} \leq a_{p_4}$$

$$\Leftrightarrow 21 \leq 31_1 \leq 31_2 \leq 49 \leq 58 \leq 67 \leq 74 \leq 96$$

sowie für P_2

$$a_{p_5} \leq a_{p_6} \leq a_{p_3} \leq a_{p_1} \leq a_{p_7} \leq a_{p_2} \leq a_{p_8} \leq a_{p_4}$$

$$\Leftrightarrow 21 \leq 31_2 \leq 31_1 \leq 49 \leq 58 \leq 67 \leq 74 \leq 96$$

3.2.1 Inversionen

Inversion: Zwei Elemente, die vertauscht werden müssten, um in der richtigen Reihenfolge zu sein.

Eine Inversion zwischen Elementen einer Folge liegt vor, wenn zwei Elemente innerhalb dieser Folge der gewünschten Ordnung entgegen einsortiert sind. In der Folge

$$S = \{2, 1, 3, 4, 5, 6, 7, 8, 9\}$$

liegt genau eine Inversion zwischen den Elementen a_1 und a_2 vor, alle anderen Elemente befinden sich der Größe nach in der richtigen Reihenfolge.

3.2.2 Grad der Sortiertheit einer Folge

Grad der Sortiertheit = Anzahl der Inversionen. ***Achtung:*** *Das ist genau genommen der Grad der Unsortiertheit, diesen Begriff gibt es aber nicht.*

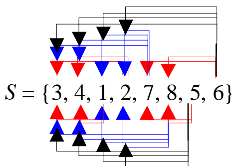
Ein Maß zur Einschätzung des Zustands einer Folge bezüglich seiner Sortiertheit ist die Anzahl der Inversionen. Die Folge

$$S = \{1, 2, 3, 4, 5, 6, 7, 8\}$$

weist keine Inversion auf, der Grad der Sortiertheit ist somit 0, die Folge ist 'sortiert'. Bei der Folge

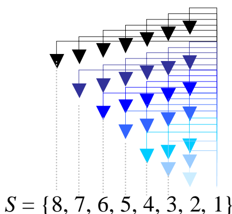
$$S = \{3, 4, 1, 2, 7, 8, 5, 6\}$$

liegen zahlreiche Inversionen (also Vertauschungen entgegen der Ordnung von kleinen hin zu großen Zahlen) vor. Zeichnen wir einfach einmal alle Vertauschungen ein – also einen Pfeil für je zwei Zahlen, die sich in der falschen Reihenfolge befinden:



$$S = \{3, 4, 1, 2, 7, 8, 5, 6\}$$

In diesem Beispiel stecken also 12 Vertauschungen (Pfeilspitzen zählen bzw. selbst ermitteln). Eine Fragestellung, der hierbei oft nachgegangen wird, ist die, wie viele Vertauschungen überhaupt möglich sind. Nehmen wir hierzu eine Folge S, deren Elemente invers sortiert sind:



$$S = \{8, 7, 6, 5, 4, 3, 2, 1\}$$

Das kleinste Elemente der Folge kann also maximal mit allen Elementen außer sich selbst vertauscht sein, also n - 1 Inversionen aufweisen. Diese eine Inversion sei bei der Betrachtung des zweiten Elements berücksichtigt (*eine* Inversion liegt zwischen *zwei* Elementen vor, wird daher also nur einmal gezählt, auch wenn beide Elemente nacheinander betrachtet werden), dann kann dieses nur noch n - 2 Vertauschungen aufweisen. Es ergibt für die maximale Anzahl der Inversionen V offensichtlich folgende Form:

$$V = (n - 1) + (n - 2) + \ldots + 1$$

also

$$V = \sum_{i=1}^{n-1} i$$

Für die Summe der Zahlen von 1 bis n gibt es bekanntermaßen eine hilfreiche Darstellung:

$$\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$$

Diese muss angepasst werden, da unsere Folge nicht von 1 bis *n* sondern nur bis *n* - 1 reicht, wir reduzieren daher im rechten Term alle *n* um 1.

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

Damit können wir die maximale Anzahl möglicher Vertauschungen *V* alleine aus der Länge der Folge *n* berechnen:

$$V = \frac{n-1(n)}{2} = \frac{n^2 - n}{2}$$

Die minimale Anzahl möglicher Inversionen einer Folge ist recht trivial 0, dies liegt dann vor, wenn eine Folge bereits sortiert ist.

3.2.3 Stabilität der Sortierung

Ein *stabile Sortierung* erhält untergeordnete Sortierungen, die in einer Folge bereits vorliegen können. Nehmen wir an, eine Folge *S* liegt vor mit

$S =$

49	67	33_1	96	21	33_2	58	74

Hierin ist die Zahl 33 zweimal vorhanden, um beide Elemente unterscheiden zu können, wurden sie mit einem Index versehen, an dem wir erkennen können, welche die erste und welche die zweite 33 ist. Nehmen wir weiter an, die Reihenfolge ist gewollt und die beiden 33-Elemente sind aufgrund einer Vorsortierung bereits in einer richtigen Anordnung. Dann würde ein stabiles Verfahren das Element 33_1 immer vor das Element 33_2 setzen, als Ergebnis einer Sortierung als zuverlässig folgendes Ergebnis liefern:

$S_{stabil} =$

21	33_1	33_2	49	58	67	74	96

Ein instabiles Verfahren *kann* zwar auch auf das Ergebnis S_{stabil} führen, jedoch lediglich durch Zufall. Es ist ebenso möglich, dass die Vorsortierung ignoriert wird und dieses Ergebnis am Ende einer Sortierung steht:

$S_{instabil} =$

21	33_2	33_1	49	58	67	74	96

3.3 Algorithmen

3.3.1 InsertSort

Kochrezept:
1. Nehme das zweite Element einer Liste und packe es an die erste oder zweite Stelle einer Liste, je nachdem ob es größer oder kleiner als das erste Element ist.
2. Nehme das folgende und plaziere es seiner Größe nach in die bereits sortierten Elemente ein.

- Falls das Element kleiner ist als eines der bislang sortierten Elemente, verschiebe so viele Elemente nach rechts verschoben werden, bis das momentane Element eingefügt werden kann.

Tracing mit Musterdatensatz:

49	67	33_1	96	21	33_2	58	74		49	67						
49	67	33_1	96	21	33_2	58	74		49		67					
49	67	33_1	96	21	33_2	58	74			49	67					
49	67	33_1	96	21	33_2	58	74		33_1	49	67					
49	67	33_1	96	21	33_2	58	74		33_1	49	67	96				
49	67	33_1	96	21	33_2	58	74		33_1	49	67		96			
49	67	33_1	96	21	33_2	58	74		33_1	49		67	96			
49	67	33_1	96	21	33_2	58	74		33_1		49	67	96			
49	67	33_1	96	21	33_2	58	74			33_1	49	67	96			
49	67	33_1	96	21	33_2	58	74		21	33_1	49	67	96			
49	67	33_1	96	21	33_2	58	74		21	33_1	49	67		96		
49	67	33_1	96	21	33_2	58	74		21	33_1	49		67	96		
49	67	33_1	96	21	33_2	58	74		21	33_1		49	67	96		
49	67	33_1	96	21	33_2	58	74		21	33_1	33_2	49	67	96		
49	67	33_1	96	21	33_2	58	74		21	33_1	33_2	49	67		96	
49	67	33_1	96	21	33_2	58	74		21	33_1	33_2	49		67	96	
49	67	33_1	96	21	33_2	58	74		21	33_1	33_2	49	58	67	96	
49	67	33_1	96	21	33_2	58	74		21	33_1	33_2	49	58	67		96
49	67	33_1	96	21	33_2	58	74		21	33_1	33_2	49	58	67	74	96

Algorithmus in Pascal-Notation:

Gegeben ist ein Feld $a[N]$ der Länge N, also ein Feld, dass die Elemente von $[1..N]$ enthält)

```
for j := 2 to length do          // von 2 bis length wiederhole:
begin
  key := a[j];                   // wähle eine Elem. key := a[j]
  i := j - 1;                    // wähle eine zweite Marke i
  // Teilaufgabe: füge key in a[1..j - 1] ein
  while i > 0 and a[i] > key do // solange key nicht einfügbar
```

```
  begin
    a[i + 1] := a[i];          // a[i] eine Pos. nach rechts
    i := i - 1;                // i incr.
  end;
  a[i + 1] := key;             // Pos gefunden -> Key einfügen
end;
```

Analyse von Insertsort

Für die Analyse wird der Algorithmus mit Kostenindizes c_i versehen, das heißt, jede Zeile, in der Code abgearbeitet wird, bekommt einen Kostenindex:

```
[ 1] for j := 2 to length do                  c1
[ 2] begin                                    -
[ 3]   key := a[j];                           c2
[ 4]   i := j - 1;                            c3
[ 5]   { füge a[j] in a[1..j - 1] ein }       -
[ 6]   while i > 0 and a[i] > key do          c4
[ 7]   begin                                  -
[ 8]     a[i + 1] := a[i];                    c5
[ 9]     i := i - 1;                          c6
[10]   end;                                   -
[11]   a[i + 1] := key;                       c7
[12] end;                                     -
```

Zeilen, in denen kein ausführbarer Code steht, sind für die Berechnung der Laufzeit in der Regel nicht von Bedeutung.[1] Diese Indizes werden zur Berechnung der Laufzeit $T(n)$ eines Algorithmus zusammengezogen. Dies geht jedoch nicht durch einfaches Aufsummieren der c_i, da verschiedene Statements verschiedene Laufzeiten haben (bzw. „*kosten*"). Eine Zuweisung erfordert beispielsweise einen Rechenschritt, eine Schleife kann hingegen unendliche viele Rechenschritte nach sich ziehen, zu den individuellen Zeitaufwänden, die in den Indizes wiedergegeben sind, kommt nun noch die Anzahl der Ausführungen. Diese folgen aus dem Code des Algorithmus und aus den Daten, die er verarbeitet. Für eine bereits sortierte Folge kann ein Algorithmus das häufig schon sehr befriedigende Laufzeitverhalten $O(n)$ haben, im Worst-Case, z.B. einer zu sortierenden Folge, die genau entgegengesetzt sortiert ist, wie gewünscht, ein wesentlich schlechteres Verhalten wie $O(n^2)$ zeigen. In der Praxis wird ein Sortieralgorithmus kaum für den Fall optimiert werden, in dem er eigentlich nicht benötigt würde. Unser Hauptaugenmerk liegt daher auf dem Worst-Case O, dennoch betrachten wir bei InsertSort einmal beide Fälle. Ein Feld

$a =$

10	20	30	40	50
a_1	a_2	a_3	a_4	a_5

soll sortiert werden. Um zu sehen, was passiert, merken wir uns jeweils die Belegungen aller Variablen. Diese werden in der Reihenfolge protokolliert, in

[1] Dies gilt zumindest allgemein. Ausnahmen sind jedoch denkbar, etwa wenn Systeme wie bestimmte Versionen von Visual Basic in die Kompilate (die hier zudem nicht immer echte Kompilate sind) auch die Kommentare von Quelltexten aufnehmen. Zwar werden diese auch hier nicht ausgeführt, jedoch besteht theoretisch die Möglichkeit, dass große Textmengen die Laufzeit verlängern können. Wenn Kommentare mehrere Megabyte lang sind, ist eine erhöhte Suchzeit anzunehmen, bis die Speicheradresse für den folgenden Befehl oder die nächsten zu verarbeitenden Daten angesprungen werden. Dies sind jedoch theoretische Erwägungen.

der sie ein Debugger anzeigen würde. In einem solchen Tracing kann man auch sehen, wie oft eine Variable belegt wird.

$j = 2$:
$key := a[j] = 20$ $i := 1$ ***while*** $i > 0 =$ *true*, $a[i] > \text{key} =$ *false*   (denn $10 < 20$)   **→ insgesamt *false*, übergangen** $a[2] := 20$ **Durchlauf beendet → *j* erhöhen** $a = [10, 20, \ , \ , \ ]$
$j = 3$
$key := a[j] = 30$ $i := 2$ ***while*** $i > 0 =$ *true*, $a[i] > \text{key} =$ *false*,   (denn $20 < 30$)   **→ insgesamt *false*, übergangen** $a[3] := 30$ **Durchlauf beendet → *j* erhöhen** $a = [10, 20, 30, \ , \ ]$
$j = 4$
$key := a[j] = 40$ $i := 3$ ***while*** $i > 0 =$ *true*, $a[i] > \text{key} =$ *false*,   (denn $30 < 40$)   **→ insgesamt *false*, übergangen** $a[4] := 40$ **Durchlauf beendet → *j* erhöhen** $a = [10, 20, 30, 40, \ ]$
$j = 5$
$key := a[j] = 50$ $i := 4$ ***while*** $i > 0 =$ *true*, $a[i] > \text{key} =$ *false*,   (denn $40 < 50$)   **→ insgesamt *false*, übergangen** $a[5] := 50$ **Durchlauf beendet → *j* erhöhen** $a = [10, 20, 30, 40, 50]$
length < 6, *for*-Grenze überschritten
→ *for*-Schleife wird verlassen

Ist die Folge sortiert, ergibt sich für die *while*-Schleife immer *false*, die Zeilen 8 und 9 werden nicht durchlaufen. Damit läßt sich unserer Darstellung nun die Anzahl der Durchläufe jeder Zeile hinzufügen:

Zeile	*Kosten*	*Anzahl*
`[ 1] for j := 2 to length do`	c_1	n
`[ 2] begin`	-	

```
[ 3]    key := a[j];                           c2    n - 1
[ 4]    i := j - 1;                            c3    n - 1
[ 5]    { füge a[j] in a[1..j - 1] ein }       -
[ 6]    while i > 0 and a[i] > key do          c4    n - 1
[ 7]    begin                                  -
[ 8]      a[i + 1] := a[i];                    c5    0
[ 9]      i := i - 1;                          c6    0
[10]    end                                    -
[11]    a[i + 1] := key;                       c7    n - 1
[11]  end;                                     -
```

Die Anweisung, die für die zeitlichen Kosten c_1 steht, wird n mal ausgeführt, wobei n der Länge des zu sortierenden Feldes a entspricht. Zwar steht in der *for*-Anweisung lediglich, dass von 2 bis zu *length* mal durchlaufen werden soll, was $n - 1$ entspräche, es ergibt sich jedoch nach dem Erreichen von *length* durch j ein weiterer Vergleich zwischen j und *length*, der ebenfalls als eine Anweisung zu zählen ist. Daher führt die erste Zeile zwar nur n - 1 mal zur Ausführung des Schleifenkörpers, der Schleifenrumpf wird aber einmal mehr – und damit genau n mal – angesprungen. Die Kosten c_2, c_3 und c_4 werden für jedes j einmal durchlaufen, also n - 1 mal. Es ergibt sich in diesem einfachsten Fall für die Laufzeit $T(n)$:

$$T(n) = c_1 n + c_2(n-1) + c_3(n-1) + c_4(n-1) + c_7(n-1)$$

Dieser Sachverhalt wird anschaulicher, wenn ausgeklammert und dannach sortiert wird:

$$\begin{aligned} T(n) &= c_1 n + c_2 n - c_2 + c_3 n - c_3 + c_4 n - c_4) + c_7 n - c_7 \\ &= (c_1 + c_2 + c_3 + c_4 + c_7)\, n - (c_2 - c_3 - c_4 - c_7) \end{aligned}$$

Nimmt man $a = c_1 + c_2 + c_3 + c_4 + c_7$ und $b = c_2 - c_3 - c_4 - c_7$, dann entspricht dies nun die Form $an + b$, einer linearen Funktion von n. Der Aufwand für InsertSort liegt bei $O(n)$, da $an + b$ mit den Konstanten a und $b \in O(n)$ ist.

Für einen Sortieralgorithmus wäre dies ein äußert gutes Ergebnis, dass Insertsort leider nur im Ausnahmefall der bereits sortierten Eingabefolge erreicht. Einen weitaus realistischeren Eindruck vermittelt das Ergebnis der Worst-Case-Analyse, dazu nehmen wir als zu sortierende Folge eine genau entgegengesetzt zur gewünschten Ordnung sortierte Folge a:

$a =$

50	40	30	20	10
a_1	a_2	a_3	a_4	A_5

Es ergibt sich folgender Ablauf (jeweils von links nach rechts zu lesen):

$j = 2$: *key* $:= a[j] = 40$	$j = 3$ *key* $:= a[j] = 30$
i := 1 *while* $i > 0$: *true*, $a[i = 1] >$ key: *true* (denn 50 > 40) → **insgesamt *true*** $a[i + 1 = 2] := 50$ $i := 0$ $a = [\ \ , 50, \ \ , \ \ , \ \ ]$	i := 2 *while* $i > 0$: *true* $a[i = 2] >$ key: *true* (denn 50 > 30) → **insgesamt *true*** $a[i + 1 = 3] = 50$ $i := 1$ $a = [40, 50, 50, \ \ , \ \ ]$

while
$i > 0 =$ *false*,
$a[i] > \text{key} =$ *false*
→ **insgesamt *false***
$a[i + 1] := 40$
Durchlauf beendet → *j* erhöhen
$a = [40, 50, \ , \ , \ ]$

$j = 4$	*key*	$:= a[j] = 20$

$i := 3$
while
$i > 0$: *true*
$a[i = 3] > \text{key}$: *true*
(denn $50 > 20$)
→ **insgesamt *true***
$a[i + 1 = 4] := 50$
$i := 2$
$a = [30, 40, 50, 50, \ ]$
while
$i > 0$: *true*
$a[i = 2] > \text{key}$: *true*
(denn $40 > 20$)
→ **insgesamt *true***
$a[i + 1 = 3] := 40$
$i := 1$
$a = [30, 40, 40, 50, \ ]$
while
$i > 0$: *true*
$a[i = 1] > \text{key}$: *true*
(denn $30 > 20$)
→ **insgesamt *true***
$a[i + 1 = 2] := 30$
$i := 0$
$a = [30, 30, 40, 50, \ ]$
while
$i > 0 =$ *false*,
$a[i] > \text{key} =$ *false*
→ **insgesamt *false***
$a[i + 1 = 1] := 20$
Durchlauf beendet → *j* erhöhen
$a = [20, 30, 40, 50, \ ]$

while
$i > 0$: *true*
$a[i = 1] > \text{key}$: *true*
(denn $40 > 30$)
→ **insgesamt *true***
$a[i + 1 = 2] := 40$
$i := 0$
$a = [40, 40, 50, \ , \ ]$
while
$i > 0 =$ *false*,
$a[i] > \text{key} =$ *false*
→ **insgesamt *false***
$a[i + 1 = 1] := 30$
Durchlauf beendet → *j* erhöhen
$a = [30, 40, 50, \ , \ ]$

$j = 5$	*key*	$:= a[j] = 10$

$i := 4$
while
$i > 0$: *true*
$a[i = 4] > \text{key}$: *true*
(denn $50 > 10$)
→ **insgesamt *true***
$a[i + 1 = 5] := 50$
$i := 3$
$a = [20, 30, 40, 50, 50]$
while
$i > 0$: *true*
$a[i = 3] > \text{key}$: *true*
(denn $40 > 10$)
→ **insgesamt *true***
$a[i + 1 = 4] := 40$
$i := 2$
$a = [20, 30, 40, 40, 50]$
while
$i > 0$: *true*
$a[i = 2] > \text{key}$: *true*
(denn $30 > 10$)
→ **insgesamt *true***
$a[i + 1 = 3] := 30$
$i := 1$
$a = [20, 30, 30, 40, 50]$
while
$i > 0$: *true*
$a[i = 1] > \text{key}$: *true*
(denn $20 > 10$)
→ **insgesamt *true***
$a[i + 1 = 2] := 20$
$i := 0$
$a = [20, 20, 30, 40, 50]$
while
$i > 0 =$ *false*,
$a[i] > \text{key} =$ *false*
→ **insgesamt *false***
$a[i + 1 = 1] := 10$
Durchlauf beendet → *j* erhöhen
$a = [10, 20, 30, 40, 50]$

length < 6, *for*-Grenze überschritten

→ ***for*-Schleife wird verlassen**

Es ist deutlich zu erkennen, dass die Anzahl der zu verarbeitenden Anweisungen erheblich zugenommen hat. Dies ist zurückzuführen auf die zweite Schleife, die *while*-Schleife. Beim ersten Beispiel wurde diese ähnlich einer Anweisung einfach passiert. Das Worst-Case-Szenario erfordert nun für alle j die Ausführung des Schleifenkörpers. So viele Elemente, wie bereits sortiert sind, müssen nun für jedes j umsortiert werden. Für $j = 2$ macht dies 2 Aufrufe, für $j = 3$ sind es 3Aufrufe, für $j = 4$ werden 4 Aufrufe getätigt und für für $j = 5$ sind es 5. Jede Ausführung des Schleifenkörpers bedeutet die Ausführung der Zeilen 8 und 9 mit den Kosten c_5 und c_6. Die Anzahl der Aufrufe der *while*-Schleife liegt auch hier um eins höher als die Anzahl der Ausführungen des Körpers, da ein Aufruf immer zum Verlassen der Schleife benötigt wird. Für c_4 gilt nun:

$$c_4 = \sum_{j=2}^{n} j \text{, d.h}$$

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

Für c_4 und c_5 gilt ähnlich – nur um ein Element verringert:

$$c_{4,5} = \sum_{j=2}^{n} j - 1 \text{, d.h}$$

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

Im Worst-Case ergeben sich damit diese Häufigkeiten des Durchlaufs bei InsertSort:

Zeile		*Kosten*	*Anzahl*
[1]	**for** j := 2 **to** length **do**	c_1	n
[2]	**begin**	-	
[3]	key := a[j];	c_2	n - 1
[4]	i := j - 1;	c_3	n - 1
[5]	{ füge a[j] in a[1..j - 1] ein }	-	
[6]	**while** i > 0 **and** a[i] > key **do**	c_4	$\sum_{j=2}^{n} j$
[7]	**begin**	-	
[8]	a[i + 1] := a[i];	c_5	$\sum_{j=2}^{n} j-1$
[9]	i := i - 1;	c_6	$\sum_{j=2}^{n} j-1$
[10]	**end**	-	
[11]	a[i + 1] := key;	c_7	n - 1
[11]	**end;**	-	

Damit setzt sich die Laufzeit $T(n)$ in diesem Fall wie folgt zusammen:

$$T(n) = c_1 n + c_2(n-1) + c_3(n-1) + c_4\left(\frac{n(n+1)}{2} - 1\right) + c5\frac{n(n-1)}{2} + c_6\frac{n(n-1)}{2}$$

$$+ c_7 (n-1)$$

$$= c_1 n + c_2 n - c_2 + c_3 n - c_3 + c_4\frac{n^2+n}{2} - c_4 + c_5\frac{n^2-n}{2} + c_6\frac{n^2-n}{2} + c_7 n$$

$$- c_7$$

$$= c_1n + c_2n - c_2 + c_3n - c_3 + c_4\frac{n^2}{2} + c_4\frac{n}{2} - c_4 + c_5\frac{n^2}{2} - c_5\frac{n}{2} + c_6\frac{n^2}{2} - c_6\frac{n}{2} + c_7n - c_7$$

$$= \left(\frac{c_4}{2} + \frac{c_5}{2} + \frac{c_6}{2}\right)n^2 + (c_1 + c_2 + c_3 + \frac{c_4}{2} - \frac{c_5}{2} - \frac{c_6}{2} + c_7)\ n - (c_2 + c_3 + c_4 + c_7)$$

Damit läßt sich zeigen, aus welcher Aufwandsklasse $T(n)$ in diesem Fall ist, mit

$$a = \frac{c_4}{2} + \frac{c_5}{2} + \frac{c_6}{2}$$

$$b = c_1 + c_2 + c_3 + \frac{c_4}{2} - \frac{c_5}{2} - \frac{c_6}{2} + c_7 \quad \text{und}$$

$$c = c_2 + c_3 + c_4 + c_7$$

entspricht dies der Form $a^2 + bn + c$, also einer quadratischen Funktion von n.

3.3.2 Mergesort

Kochrezept

1. Zerlege eine Folge in zwei etwa gleich große Teilfolgen
2. Zerlege die Teilfolgen wie in 1. bis die Teilfolgen lediglich ein Element enthalten.
3. Verbinde die Teilfolgen schrittweise:
    1. Vergleiche jeweils das erste Element zweier benachbarter Teilfolgen und füge das minimale Element an die jeweils erste freie Position von links der zusammengesetzten Folge.
    2. Streiche übertragene Elemente aus den Teilfolgen.
    3. Liegt eine leere Teilfolge vor, übertrage die restlichen Elemente der anderen Folge der Reihe nach in die leeren Elemente der Verbindungsfolge.
    4. Wiederhole ab Punkt 4 bis nur noch eine Folge übrig ist.

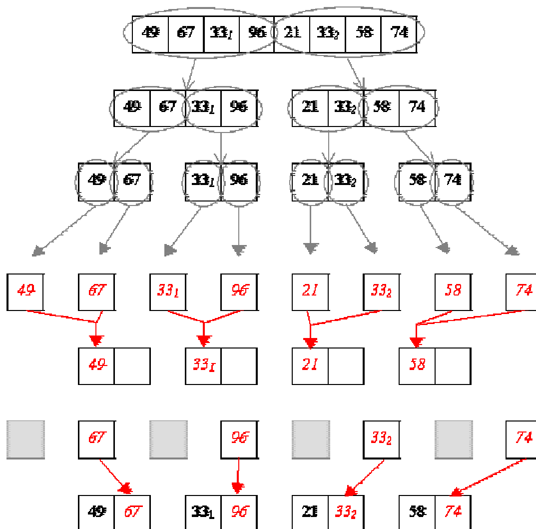
Ähnlich wie in Heapsort wird also auch hier in zwei Schritten verfahren (Teile- und Herrsche-Strategie):

- Zunächst wird eine bestimmte Struktur erzeugt
- Dann wird aus dieser Struktur heraus sortiert.

Tracing mit Musterdatensatz:

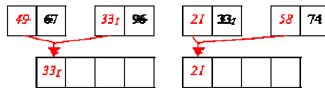
Erzeugen von Teilfolgen durch jeweils mittiges Teilen von Folgen, und verbinden der minimalen Folgen zu sortierten Folgen nächstgrößerer Ordnung durch Einfügen des kleinsten Elements von links (da die nächstgrößere Ordnung zwei-elementige Folgen darstellen, sind zwei Schritte notwendig, im ersten Schritt wird das größere Element in die Verbindungsfolge übertrage, im zweiten Schritt werden die übrigen Elemente übertragen).

1:

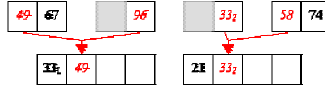


Aus den zwei-elementigen Teilfolgen werden nun sortierte vier-elementige Teilfolgen aufgebaut.

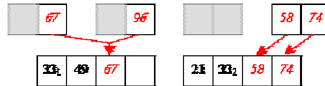
2:



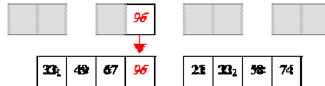
3:



4:

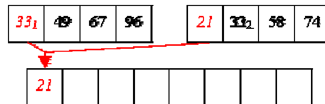


5:

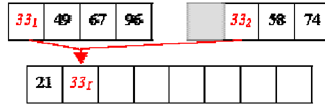


Und aus zwei vier-elementigen Folge wird nun eine sortierte acht-elementige Folge erzeugt:

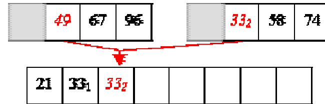
6:



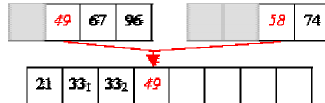
7:



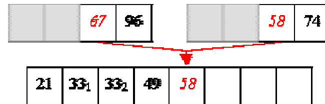
8:



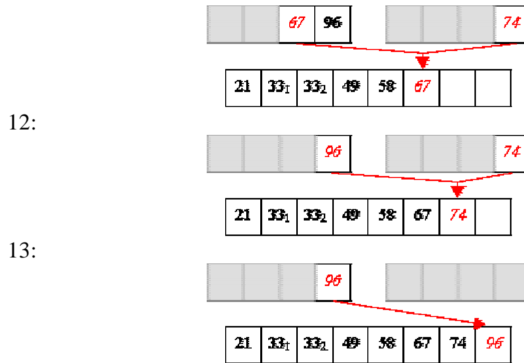
9:



10:



11:



Algorithmus in Pascal-Notation:

Gegeben ist ein Feld *a*[*N*] der Länge *N*, also ein Feld, dass die Elemente von [1..*N*] enthält)

Zeile		*Kosten*
[1]	`if l < r`	c_1
[2]	`then begin`	-
[3]	`m := (l + r) div 2;`	c_2
[4]	`MergeSort(a, l, m);`	c_3
[5]	`MergeSort(a, m + 1 , r);`	c_4
[6]	`Merge(a, l, m, r);`	c_5
[7]	`end;`	-

Wie in Zeile 6 zu sehen, ist eine Routine *Merge(a, l, m, r)* notwendig, deren Aufgabe das Zusammenfügen der jeweils sortierten Teilfolgen ist.

```
          procedure Merge(var a : Feld; l, m, r : integer);
          //  1. Zusammenführen von a[l..m] und a[m + 1 .. r]
          //  2. Speichern von a[l..m] und a[m + 1 .. r] in a[l ..r]
          var b            : feld;
          h, i, j, k : integer;
          begin
[ 1]        i := l;
[ 2]        j := m + 1;
[ 3]        k := l;
[ 4]        while (i <= m) and (j <= r) do
[ 5]        begin
[ 6]          if a[i] <= a[j]             // vgl. li. Feldelem m. re.
[ 7]          then begin                  // wenn a[i] kleiner dann...
[ 8]                 b[k] := a[i];        // an 1. freie Pos. v. b
[ 9]                 i    := i + 1;
[10]               end
[11]          else begin
[12]                 b[k] := a[j];        // sonst a[j] n. b kopieren
[13]                 j := j + 1;
[14]               end;
[15]          k := k + 1;                  // b[k] besetzt, daher k inc.
[16]        end;
```

```
[17]    if i > m
[18]    then for h := j to r do       // wenn a[i..m] fertig ->
[19]            b[k + h - j] := a[h] // b[k++] :=a [(m + 1)]++]
[20]    else for h := i to m do       // wenn a[m+1..r] fert. ->
[21]            b[k + h - i] := a[h];// b[k++] := a[(m + 1)]++]
[22]    for h := l to r do            // und alles zurück nach a
[23]            a[h] := b[h];
[24] end;
```

Analyse

Protokollieren wir auch bei MergeSort die Folge

$a =$	10	20	30	40	50
	a_1	a_2	a_3	a_4	a_5

mit dem entsprechenden Aufruf *MergeSort(a, 1, 5)*.

Der aktuell benutzte Teil der Folge ist schwarz, der Rest hingegen grau kursiv gesetzt. Das Symbol für die Vereinigung von Mengen ∪ wird hier genutzt, um die Vereinigung der Teilfolgen anzudeuten. Die Schreibweise

MergeSort(*a*, l = 1, m = 3)
→ MergeSort(*a*, l = 1, r = 3)

zeigt oben den Aufruf der Routine MergeSort (sozusagen von aussen) und unten die Ausführung, also die Parameter, wie sie von innen erscheinen. Es ergibt sich für MergeSort (ohne detaillierte Betrachtung von *Merge(a, l, m, r)*) folgender Ablauf:

l: 1
m: -
r: 5
MergeSort(*a*, l=1, r = 5) → [10][20][30][40][50]
l: 1;
m: 3
r: 5
MergeSort(*a*, l = 1, m = 3)
→ MergeSort(*a*, l = 1, r = 3) → [10][20][30]*[40][50]*
l: 1
r: 3
m: 2
MergeSort(*a*, l = 1, m = 2)
→ MergeSort(*a*, l = 1, r = 2) → [10][20]*[30][40][50]*
l: 1
r: 2
m: 1
MergeSort(*a*, l = 1, m = 1)
→ MergeSort(*a*, l = 1, r = 1) → [10]*[20][30][40][50]*
⊥
MergeSort(*a*, m + 1 = 2, r = 2)
→ MergeSort(*a*, l = 2, r = 2) → *[10]*[20]*[30][40][50]*
⊥
Merge(*a*, l = 1, m = 1, r = 2)
= [10] ∪ [20] ⇒ [10][20] [30][40][50]
MergeSort(*a*, m + 1 = 3, r = 3)
→ MergeSort(*a*, l = 3, r = 3) → *[10][20]*[30]*[40][50]*
⊥
Merge(*a*, l = 1, m = 2, r = 3)

= [10][20] ∪ [30] ⇒ [10][20][30] [40][50]
MergeSort(a, m + 1 = 4, r = 5)
→ MergeSort(a, l = 4, r = 4) → [10][20][30][40][50]
⊥
→ MergeSort(a, l = 5, r = 5) → [10][20][30][40][50]
⊥
Merge(a, l = 4, m = 4, r = 5)
= [40] ∪ [50] ⇒ [10][20][30] [40][50]
Merge(a, l = 1, m = 3, r = 5)
= [10][20][30] ∪ [40][50] ⇒ [10][20][30][40][50]

Fakt 1: Es liegt Rekursion vor

Da MergeSort sich selbst aufruft und somit Rekursion vorliegt, spiegelt sich dies auch in der Analyse der Laufzeit $T(n)$ von MergeSort wieder. Es ist charakteristisch für Rekursion, dass es einen Zweig gibt, der den Term mit dem Selbstaufruf enthält und einen Zweig, der einen Term für den Fall der Terminierung der Rekursion enthält. Für ein bestimmtes $n \leq c$ ergibt sich für den Terminierungszweig eine Laufzeit, die als konstant betrachtet werden kann und somit einen Aufwand von $O(1)$ hat. Bei MergeSort ist $c = 1$, in diesem Fall wird lediglich der Vergleich in Zeile 1 und terminiert.

Fakt 2: MergeSort arbeitet nach dem Divide-and-Conquer-Prinzip

Bei MergeSort handelt es sich um einen Algorithmus, der nach dem Divide-and-Conquer-Prinzip vorgeht. *Teilen* und *Herrschen* meint eine Strategie, die ein Problem in immer kleinere Teilprobleme aufteilt, separat löst und zu einer Gesamtlösung zusammenfügt. Der Quellcode von MergeSort besteht aus nichts anderem als Anweisungen zum Teilen und zum Zusammenfügen von Eingangsfolgen. Gehen wir daher davon aus, dass der Rekursionsteil ein Problem mit der Gesamtlaufzeit b löst und dieses in a Unterprobleme aufteilt. Dann ergibt sich für die Laufzeit $T(b)$:

$$T(b) = aT\left(\frac{1}{b}\right)$$

Also:
$T(n)$ = Divide
+Conquer
+ Zusammenfügen

Gehen wir ferner davon aus, dass $D(n)$ die Zeit zum Zerlegen der Folge und $C(n)$ die Zeit zum Zusammenfügen der Folge ist, dann läßt sich für $T(n)$ zusammenfassen:

$$T(n) = \begin{cases} O(1) & \textit{wenn } n = 1, \\ a(T\left(\frac{n}{b}\right) + D(n) + C(n) & \textit{sonst} \end{cases}$$

Folgende Kosten lassen sich zunächst notieren:

Zeile		*Kosten*
[1]	`if l < r`	D
[2]	`then begin`	-
[3]	`m := (l + r) div 2;`	D
[4]	`MergeSort(a, l, m);`	$T(\lfloor n/2 \rfloor)$
[5]	`MergeSort(a, m + 1 , r);`	$T(\lceil n/2 \rceil)$
[6]	`Merge(a, l, m, r);`	C
[7]	`end;`	-

Der Aufwand für den Schritt des Teilens setzt sich also aus dem Vergleich der oberen und unteren Grenze sowie der Berechnung der Feldmitte m zusammen.

Diese Operationen sind von n unabhängig und haben somit eine konstante Laufzeit, es ergibt sich $D \in O(1)$. Der Conquer-Schritt findet zweimal statt, einmal wird die untere Hälfte des Feldes übergeben, einmal die obere. Der Algorithmus teilt die Aufgabe immer in etwa gleichgrosse Teilaufgaben, etwaige Abweichungen treten höchstens bei ungerade Anzahlen von Elementen auf und bewirken eine minimale Abweichung. So läßt sich also sagen, dass die beiden Aufrufe *MergeSort*(a, l, m) und *MergeSort*(a, m + 1, r) die Laufzeit $a\,T(^n/_b)$ in etwa auf zwei Subroutinnen verteilen, a und b sind somit 2, es ergibt sich:[2]

$$T(b) = T(\lfloor ^n/_2 \rfloor) + T(\lceil ^n/_2 \rceil)$$

Analyse der Merge-Routine

Merge-Routine: In Zeile 4 der Merge-Routine werden Schlüssel verglichen. Nach jedem Vergelich werden die Indizes der linken bzw. rechten Teilfolge inkrementiert. Wurde eine der beiden Teilfolge durchlaufen, terminiert die Schleife. Im Best-Case wird diese Schleife also $^n/_2$ mal durchlaufen, im Worst-Case $\lfloor ^n/_2 \rfloor + \lceil ^n/_2 \rceil$ mal, das ergibt für die Merge-Routine eine Laufzeit von $O(n)$.

$$T(n) = \begin{cases} O(1) & \textit{wenn } n = 1, \\ 2(T\left(\frac{1}{2}\right) + O(1) + O(n) & \textit{sonst} \end{cases}$$

Hier kann etwa zusammengefasst werden:

$$\begin{aligned} O(1) + O(n) &= O(\max\{O(1), O(n)\}) \\ &= O(n) \end{aligned}$$

Es ergibt sich:

$$T(n) = \begin{cases} O(1) & \textit{wenn } n = 1, \\ 2T\left(\frac{1}{2}\right) + O(n) & \textit{sonst} \end{cases}$$

Damit läßt sich bis hier nur sagen, dass im Worst-Case ein Aufwand von $O(n)$ auftreten kann. In welcher Größenordnung der Summand $2T(½)$ liegt, bleibt zunächst unklar. Neben zwei möglichen formalen Wegen (die in [CL+ 96] gezeigt werden und einige mathematische Beziehungen geschickt ausnutzen), gibt es eine inhaltliche Argumentation von Ottmann und Widmayer [OW 90]. Da MergeSort die Folge a in immer gleichgrosse Subfolgen zerlegt, bis die Länge der Teilfolgen 1 ist, ist die Rekursionstiefe logarithmisch beschränkt. Daher ergibt sich

$$T(n) = \begin{cases} O(1) & \textit{wenn } n = 1, \\ O(n \textit{ ld } n) + O(n) & \textit{sonst} \end{cases}$$

Auch hier kann die Summe zusammengefasst werden:

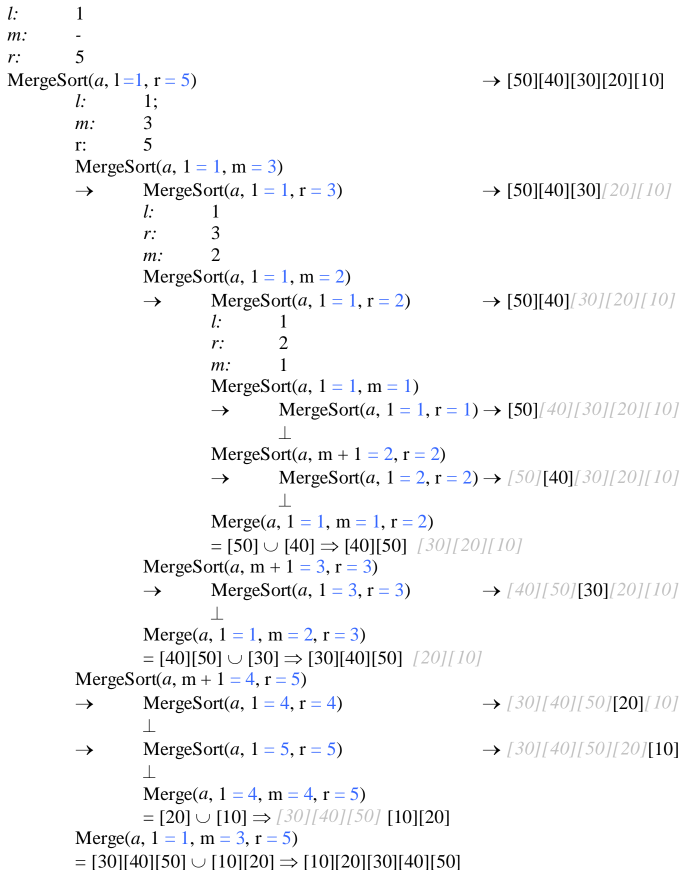
$$\begin{aligned} O(n \text{ ld } n) + O(n) &= O(\max\{O(n \text{ ld } n), O(n)\}) \\ &= O(n \text{ ld } n) \end{aligned}$$

Eingesetzt in unser Zwischenergebnis für $T(n)$ läßt sich nun schreiben:

[2] $\lfloor x \rfloor$ steht für die Rundung von x nach unten, $\lceil x \rceil$ für die Aufrundung.

$$T(n) = \begin{cases} O(1) & wenn\ n = 1, \\ O(n\ ld\ n) & sonst \end{cases}$$

Soweit die Analyse der Laufzeit mit einer optimal vorsortierten Folge. Betrachten wir nun den Worst-Case einer genau invers zur gewünschten Ordnung vorliegenden Folge, hierbei ergibt sich folgender Ablauf:



l: 1
m: -
r: 5
MergeSort(*a*, l = 1, r = 5) → [50][40][30][20][10]
  l: 1;
  m: 3
  r: 5
  MergeSort(*a*, 1 = 1, m = 3)
  → MergeSort(*a*, 1 = 1, r = 3) → [50][40][30][20][10]
    l: 1
    r: 3
    m: 2
    MergeSort(*a*, 1 = 1, m = 2)
    → MergeSort(*a*, 1 = 1, r = 2) → [50][40][30][20][10]
      l: 1
      r: 2
      m: 1
      MergeSort(*a*, 1 = 1, m = 1)
      → MergeSort(*a*, 1 = 1, r = 1) → [50][40][30][20][10]
        ⊥
      MergeSort(*a*, m + 1 = 2, r = 2)
      → MergeSort(*a*, 1 = 2, r = 2) → [50][40][30][20][10]
        ⊥
      Merge(*a*, 1 = 1, m = 1, r = 2)
      = [50] ∪ [40] ⇒ [40][50] [30][20][10]
    MergeSort(*a*, m + 1 = 3, r = 3)
    → MergeSort(*a*, 1 = 3, r = 3) → [40][50][30][20][10]
      ⊥
    Merge(*a*, 1 = 1, m = 2, r = 3)
    = [40][50] ∪ [30] ⇒ [30][40][50] [20][10]
  MergeSort(*a*, m + 1 = 4, r = 5)
  → MergeSort(*a*, 1 = 4, r = 4) → [30][40][50][20][10]
    ⊥
  → MergeSort(*a*, 1 = 5, r = 5) → [30][40][50][20][10]
    ⊥
    Merge(*a*, 1 = 4, m = 4, r = 5)
    = [20] ∪ [10] ⇒ [30][40][50] [10][20]
  Merge(*a*, 1 = 1, m = 3, r = 5)
  = [30][40][50] ∪ [10][20] ⇒ [10][20][30][40][50]

Dieser Ablauf entspricht genau dem der optimal vorsortierten Folge. Das heißt, dass der Aufwand bei MergeSort allein von der Länge der Folge abhängt, nicht jedoch vom Grad der Invertierungen. In diesem Fall ist mit $O(n \text{ ld } n)$ nicht nur der maximale Aufwand bestimmt sondern auch der minimale $\Omega(n \text{ ld } n)$ und das asymptotisch enge Maß $\Theta(n \text{ ld } n)$. Daher notieren wir abschliessend

Bei MergeSort ist $O(n \text{ ld } n) = \Omega(n \text{ ld } n)$

$$T(n) = \begin{cases} \Theta(1) & wenn\ n = 1, \\ \Theta(n\ ld\ n) & sonst \end{cases}$$

3.3.3 Countsort

Kochrezept

1. Schaffe eine (Hilfs-) Folge, die genauso viele Felder hat, wie die zu sortierende Folge. Setzte den Inhalt aller Felder der Hilfsfolge auf 0.
2. Vergleiche die zu sortierende Folge - von rechts beginnend - paarweise, bis jedes Element mit jedem verglichen wurde. Das jeweils größere Element wird an der ihm entsprechenden Stelle in der Hilfsfolge um 1 erhöht. Sind zwei Elemente gleich groß, wird das Hilfselement der rechten Folge um 1 erhöht.
3. Nachdem die Vergleiche abgeschlossen sind, werden die Elemente gemäß der ihnen zugeordneten Zahlen der Hilfsfolge entsprechend angeordnet.

Tracing mit Musterdatensatz:

Vgl. von a_n mit $[a_{n-1},..., a_1]$

49	67	33_1	96	21	33_2	58	74
0	0	0	0	0	0	0	1

49	67	33_1	96	21	33_2	58	74
0	0	0	0	0	0	0	2

49	67	33_1	96	21	33_2	58	74
0	0	0	0	0	0	0	3

49	67	33_1	96	21	33_2	58	74
0	0	0	1	0	0	0	3

49	67	33_1	96	21	33_2	58	74
0	0	0	1	0	0	0	4

49	67	33_1	96	21	33_2	58	74
0	0	0	1	0	0	0	5

49	67	33_1	96	21	33_2	58	74
0	0	0	1	0	0	0	6

Vgl. von a_{n-1} mit $[a_{n-2}, ..., a_1]$

49	67	33_1	96	21	33_2	58	74
0	0	0	1	0	0	1	6

49	67	33_1	96	21	33_2	58	74
0	0	0	1	0	0	2	6

49	67	33_1	96	21	33_2	58	74
0	0	0	2	0	0	2	6

49	67	33_1	96	21	33_2	58	74
0	0	0	2	0	0	3	6

49	67	33_1	96	21	33_2	58	74
0	1	0	2	0	0	3	6

49	67	33_1	96	21	33_2	58	74
0	1	0	2	0	0	4	6

Vgl. von a_{n-2} mit $[a_{n-3}, ..., a_1]$

49	67	33_1	96	21	33_2	58	74
0	1	0	2	0	1	4	6

49	67	33_1	96	21	33_2	58	74
0	1	0	2	0	1	4	6

49	67	33_1	96	21	33_2	58	74
0	1	1	3	0	2	4	6

49	67	33_1	96	21	33_2	58	74
0	2	1	3	0	2	4	6

49	67	33_1	96	21	33_2	58	74
1	2	0	3	0	2	4	6

Vgl. von a_{n-3} mit $[a_{n-4}, ..., a_1]$

49	67	33_1	96	21	33_2	58	74
1	2	0	4	0	2	4	6
49	67	33_1	96	21	33_2	58	74
1	2	1	4	0	2	4	6
49	67	33_1	96	21	33_2	58	74
1	3	1	4	0	2	4	6
49	67	33_1	96	21	33_2	58	74
2	3	1	4	0	2	4	6

Vgl. von a_{n-4} mit $[a_{n-5}, ..., a_1]$

49	67	33_1	96	21	33_2	58	74
2	3	1	5	0	2	4	6
49	67	33_1	96	21	33_2	58	74
2	3	1	6	0	2	4	6
49	67	33_1	96	21	33_2	58	74
2	3	1	7	0	2	4	6

Vgl. von a_{n-5} mit $[a_{n-6}, ..., a_1]$

49	67	33_1	96	21	33_2	58	74
2	4	1	7	0	2	4	6
49	67	33_1	96	21	33_2	58	74
3	4	1	7	0	2	4	6

Vgl. von a_{n-6} mit $[a_{n-7}, ..., a_1]$

49	67	33_1	96	21	33_2	58	74
3	5	1	7	0	2	4	6

Übertragen der Zahlen in Reihenfolge der Werte der Hilfsfolge:

49	67	33_1	96	21	33_2	58	74
3	5	1	7	0	2	4	6
21	33_1	33_2	49	58	67	74	67

Algorithmus in Pascal-Notation:

Gegeben sind ein Feld $a[1, ..., N]$ der Länge N (also ein Feld, dass die Elemente von $[1..N]$ enthält), ein Feld $c[1, ..., N]$, in dem für jede Zahl notiert wird, wie groß sie im Verhältnis zu den anderen Zahlen von a ist, sowie ein Feld $b[1, ..., N]$ der Länge N, welches zum Ende des Ablaufs das nach den Angaben des Feldes c umzusortierende Feld a aufnimmt.

Zeile

```
[ 1]    for i := 1 to N do            // v. c[vorne] b. c[hinten]
[ 2]      c[i] := 0;                  // initialisiere c[i]
[ 3]    for i := N downto 2 do        // von hinten bis (vorne-1)
[ 4]      for j := i – 1 downto 1 do  // ...von (hinten-1) bis 1
[ 5]        if a[j] < a[i]            // ...wenn a[j] < a[j]
[ 6]        then c[i] := c[i] + 1     // ...dann erhöhe a[i]
[ 7]        else c[j] := c[j] + 1;    // ...sonst erhöhe a[j]
[ 8]    for i := 1 to N do            // wenn fertig, kopiere
[ 9]      b[c[i] + 1] := a[i];        //   a nach b gemäß (c+1)
```

Hier läßt sich für den Speicher sagen: Zu einem Feld kommen immer zwei weitere Felder hinzu, eines, dass genau dieselben Informationen aufnehmen muss, die das Datenfeld beinhaltet und eines, dass die Position der Daten aufnehmen kann. Es ergibt sich im schlimmsten Fall ein Speicherbedarf von $3n \in O(n)$. Für die Rechenzeit ergibt sich im Worst Case folgende Betrachtung:

Zeile		*Kosten*	*Anzahl*
[1]	`for i := 1 to n do`	c_1	$n+1$
[2]	`  c[i] := 0;`	c_2	n
[3]	`for i := N downto 2 do`	c_3	n
[4]	`  for j := i - 1 downto 1 do`	c_4	$\sum_{j=1}^{n-1} j$
[5]	`    if a[j] < a[i]`	c_5	$\sum_{j=1}^{n-1} j-1$
[6]	`    then c[i] := c[i] + 1`	c_6	$\sum_{j=1}^{n-1} j-1$
[7]	`    else c[i] := c[i] + 1;`	c_7	$\sum_{j=1}^{n-1} j-1$
[8]	`for i := 1 to n do`	c_8	$n+1$
[9]	`  a[c[i] + 1] := a[i];`	c_9	n

Damit gilt für

$$T(n) = c_1(n + 1) + c_2n + c_3n + c_4\frac{n(n-1)}{2} + c_5\left(\frac{n(n-1)}{2}-1\right) + c_6\left(\frac{n(n-1)}{2}-1\right) + c_7\left(\frac{n(n-1)}{2}-1\right) + c_8(n+1) + c_9n$$

Eigentlich sich bereits hiermit Terme der Ordnung $O(n^2)$ offensichtlich. Um sicher zu gehen, dass sich keine Terme wegheben, versuchen wir jedoch auch hier nach Graden zu sortieren:

$$= c_1n + c_1 + c_2n + c_3n + c_4\frac{n^2-n}{2} + c_5\frac{n^2-n}{2} - c_5 + c_6\frac{n^2-n}{2} - c_6 + c_7\frac{n^2-n}{2} - c_7 + c_8n + c_8 + c_9n$$

$$= c_1n + c_1 + c_2n + c_3n + c_4\frac{n^2}{2} - c_4\frac{n}{2} + c_5\frac{n^2}{2} - c_5\frac{n}{2} + c_6\frac{n^2}{2} - c_6\frac{n}{2} + c_7\frac{n^2}{2} - c_7\frac{n}{2} - c_7 + c_8n + c_8 + c_9n$$

Zusammenfassen der Terme mit gleichen Graden von n:

$$= \left(\frac{c_4}{2} + \frac{c_5}{2} + \frac{c_6}{2} + \frac{c_7}{2}\right)n^2 + \left(c_1 + c_2 + c_3 + c_8 + c_9 - \frac{c_4}{2} - \frac{c_5}{2} - \frac{c_6}{2} - \frac{c_7}{2}\right)n + \left(c_1 - c_7 + c_8\right)$$

Mit den Substitutionen

$$a = \frac{c_4}{2} + \frac{c_5}{2} + \frac{c_6}{2} + \frac{c_7}{2}$$

$$b = c_1 + c_2 + c_3 + c_8 + c_9 - \frac{c_4}{2} - \frac{c_5}{2} - \frac{c_6}{2} - \frac{c_7}{2} \text{ und}$$

$$c = c_1 - c_7 + c_8$$

entspricht dies der Form $a^2 + bn + c$, also einer quadratischen Funktion von n. Die Laufzeit liegt damit für den Worst-Case im Bereich $O(n^2)$. Anders als etwa bei InsertSort wird diese hier nicht durch andere Eigenschaften der Folge als ausschliesslich der Länge beeinflusst. Bei InsertSort kann im Falle einer vorsortierten Folge das Laufzeitverhalten $O(n)$ erreicht werden, im Wort-Case hingegen $O(n^2)$. Im Best-Case kopiert InsertSort die Elemente in der Reihenfolge, in der sie kommen lediglich an die Position in ein Feld, an der sie ohnehin schon stehen. Im Worst-Case müssen für jedes Element jedoch alle bereits eingefügt Elemente jeweils um eine Position nach rechts gerückt werden. Anders bei dieser Countsort-Variante: Es werden unabhängig von der Vorsortiertheit der Folge alle enthaltenen Elemente verglichen. Damit ist der minimale Aufwand hier (wie auch bei Mergesort) gleich dem maximalen Aufwand.

3.3.4 Countsort II

Die Theorie geht im Allgemeinen davon aus, dass n Elemente einer Folge im Worst Case mit nicht geringerem zeitlichen Aufwand als $n \log_2 n$ sortiert werden können, Schöning führt in [Sch 97] einen Beweis dieser sogenannten *informationstheoretischen Schranke* vor. Dies gilt für den Vorgang des Sortierens im Allgemeinen, nicht jedoch für jeden Spezialfall.

Verfügt man über zusätzliche Informationen, die jedoch immer im Kontext des Problems zu sehen sind und nicht immer in nutzbarer Form vorzufinden sind, kann die der Aufwand weiter gesenkt werden. Dies hebt die informationstheoretische Schranke natürlich nicht auf sondern setzt eine andere. Eine wichtige Voraussetzung, die erfüllt sein muß, um Countsort II anwenden zu können ist, dass die Elemente der Folge innerhalb eines bestimmten Intervalls liegen müssen, dessen Ober- und Untergrenze zudem bekannt sein muß. Dies kann erfüllt werden, wenn nach dem Alter von Menschen oder der Kilowattleistung eines PKW-Motors zu sortieren ist. Sind solche Zusatz-Informationen nicht gegeben, kann das Verfahren seine Vorteile nicht ausspielen.

Nehmen wir also eine Folge, die von der bisher genutzten Folge abweicht. Alle Elemente der Folge S_{In} liegen im Intervall [49,..., 55], wobei 49 die Untergrenze und 55 die Obergrenz darstellt:

S_{In} =	49	54	53	50	51	54	55	49	54	50

Dann werden zunächst zwei Felder erstellt. Der Index in der obersten Zeile dient der Übersichtlichkeit wobei die Elemente S_i angeben wie oft das Element in der Folge S_{In} bislang gezählt wurde. Da die Elemente im Intervall von 49 bis 55 liegen, werden zwei Felder mit sieben Elementen in diesen Grenzen angelegt (S_1, S_2: array[49..50] of anything):

Index	S_{49}	S_{50}	S_{51}	S_{52}	S_{53}	S_{54}	S_{55}
S_1	0	0	0	0	0	0	0

S_2	0	0	0	0	0	0	0

Das gegebene Feld wird einmal durchlaufen, wobei für jedes Element der Folge S in der Folge S_1 eine Notiz gemacht wird.

1:

49	54	53	50	51	54	55	49	54	50

Index	S_{49}	S_{50}	S_{51}	S_{52}	S_{53}	S_{54}	S_{55}
S_1	1	0	0	0	0	0	0
S_2	0	0	0	0	0	0	0

2:

49	**54**	53	50	51	54	55	49	54	50

Index	S_{49}	S_{50}	S_{51}	S_{52}	S_{53}	S_{54}	S_{55}
S_1	1	0	0	0	0	**1**	0
S_2	0	0	0	0	0	0	0

3:

49	54	**53**	50	51	54	55	49	54	50

Index	S_{49}	S_{50}	S_{51}	S_{52}	S_{53}	S_{54}	S_{55}
S_1	1	0	0	0	**1**	1	0
S_2	0	0	0	0	0	0	0

4:

49	54	53	**50**	51	54	55	49	54	50

Index	S_{49}	S_{50}	S_{51}	S_{52}	S_{53}	S_{54}	S_{55}
S_1	1	1	0	0	1	1	0
S_2	0	0	0	0	0	0	0

5:

49	54	53	50	**51**	54	55	49	54	50

Index	S_{49}	S_{50}	S_{51}	S_{52}	S_{53}	S_{54}	S_{55}
S_1	1	1	**1**	0	1	1	0
S_2	0	0	0	0	0	0	0

6:

49	54	53	50	51	**54**	55	49	54	50

Index	S_{49}	S_{50}	S_{51}	S_{52}	S_{53}	S_{54}	S_{55}
S_1	1	1	1	0	1	**2**	0
S_2	0	0	0	0	0	0	0

7:

49	54	53	50	51	54	**55**	49	54	50

Index	S_{49}	S_{50}	S_{51}	S_{52}	S_{53}	S_{54}	S_{55}
S_1	1	1	1	0	1	2	**1**
S_2	0	0	0	0	0	0	0

8:

49	54	53	50	51	54	55	**49**	54	50

Index	S_{49}	S_{50}	S_{51}	S_{52}	S_{53}	S_{54}	S_{55}
S_1	**2**	1	1	0	1	2	1
S_2	0	0	0	0	0	0	0

9:

49	54	53	50	51	54	55	49	**54**	50

Index	S_{49}	S_{50}	S_{51}	S_{52}	S_{53}	S_{54}	S_{55}
S_1	2	1	1	0	1	**3**	1
S_2	0	0	0	0	0	0	0

10:

49	54	53	50	51	54	55	49	54	**50**

Index	S_{49}	S_{50}	S_{51}	S_{52}	S_{53}	S_{54}	S_{55}
S_1	2	**2**	1	0	1	3	1
S_2	0	0	0	0	0	0	0

Danach werden einmalig die Felder von S_1 aufsummiert, dies ergibt den Wert 9. Dieser wird erhöht um 1 in das letzte Feld von S_1 eingetragen.

49	54	53	50	51	54	55	49	54	50

Index	S_{49}	S_{50}	S_{51}	S_{52}	S_{53}	S_{54}	S_{55}
S_1	2	2	1	0	1	3	**10**
S_2	0	0	0	0	0	0	0

Ausgehend von diesem Wert wird die Folge nun von hinten nach vorne durchlaufen wobei alle Felder nun mit einer akkumulierten Variante des Indexes versehen werden. Es wird jeweils der Wert eines linken Feldes errechnet, indem dessen Wert vom Wert des rechten Nachbarfeldes abgezogen wird. Diese Differenz überschreibt dann den ursprünglichen Wert.

1:

Index	S_{49}	S_{50}	S_{51}	S_{52}	S_{53}	S_{54}	S_{55}
S_1	2	2	1	0	1	**7**	10
S_2	0	0	0	0	0	0	0

2:

Index	S_{49}	S_{50}	S_{51}	S_{52}	S_{53}	S_{54}	S_{55}
S_1	2	2	1	0	**6**	7	10
S_2	0	0	0	0	0	0	0

3:

Index	S_{49}	S_{50}	S_{51}	S_{52}	S_{53}	S_{54}	S_{55}
S_1	2	2	1	**6**	6	7	10
S_2	0	0	0	0	0	0	0

4:

Index	S_{49}	S_{50}	S_{51}	S_{52}	S_{53}	S_{54}	S_{55}
S_1	2	2	**5**	6	6	7	10
S_2	0	0	0	0	0	0	0

5:

Index	S_{49}	S_{50}	S_{51}	S_{52}	S_{53}	S_{54}	S_{55}
S_1	2	**3**	5	6	6	7	10
S_2	0	0	0	0	0	0	0

6:

Index	S_{49}	S_{50}	S_{51}	S_{52}	S_{53}	S_{54}	S_{55}
S_1	**1**	3	5	6	6	7	10
S_2	0	0	0	0	0	0	0

Nun wird die Folge S_{In} nach S_{Out} übertragen. Hierbei wird das Feld S_2 zur Hilfe genommen um abzuzählen, wie oft ein mehrfach vorhandenes Element bereits *‚abgetragen'* wurde. Gut sichtbar im folgenden Tracing ist die Inkrementierung eines Elements von S_2, wenn das betreffende Element in S_{In} behandelt wird. In jedem Durchlauf wird die Position eines Wertes von S_{Out} aus $S_1[i] + S_2[i]$ errechnet.

1:

S_{In} =	**49**	54	53	50	51	54	55	49	54	50

Index	S_{49}	S_{50}	S_{51}	S_{52}	S_{53}	S_{54}	S_{55}
S_1	**1**	3	5	6	6	7	10
S_2	**1**	0	0	0	0	0	0

S_{Out} =	**49**	-	-	-	-	-	-	-	-	-

2:

S_{In} =	49	**54**	53	50	51	54	55	49	54	50

Index	S_{49}	S_{50}	S_{51}	S_{52}	S_{53}	S_{54}	S_{55}
S_1	1	3	5	6	6	**7**	10
S_2	1	0	0	0	0	**1**	0

S_{Out} =	49	-	-	-	-	-	**54**	-	-	-

3:

S_{In} =	49	54	**53**	50	51	54	55	49	54	50

Index	S_{49}	S_{50}	S_{51}	S_{52}	S_{53}	S_{54}	S_{55}
S_1	1	3	5	6	**6**	7	10
S_2	1	0	0	0	**1**	1	0

S_{Out} =	49	-	-	-	-	**53**	54	-	-	-

4:

S_{In} =	49	54	53	**50**	51	54	55	49	54	50

Index	S_{49}	S_{50}	S_{51}	S_{52}	S_{53}	S_{54}	S_{55}
S_1	1	**3**	5	6	6	7	10
S_2	1	**1**	0	0	1	1	0

S_{Out} =	49	-	**50**	-	-	53	54	-	-	-

5:

S_{In} =	49	54	53	50	**51**	54	55	49	54	50

Index	S_{49}	S_{50}	S_{51}	S_{52}	S_{53}	S_{54}	S_{55}
S_1	1	3	**5**	6	6	7	10
S_2	1	1	**1**	0	1	1	0

S_{Out} =	49	-	50	-	**51**	53	54	-	-	-

6:

S_{In} =	49	54	53	50	51	**54**	55	49	54	50

Index	S_{49}	S_{50}	S_{51}	S_{52}	S_{53}	S_{54}	S_{55}
S_1	1	3	5	6	6	**7**	10
S_2	1	1	1	0	1	**2**	0

S_{Out} =	49	-	50	-	51	53	54	**54**	-	-

7:

S_{In} =	49	54	53	50	51	54	**55**	49	54	50

Index	S_{49}	S_{50}	S_{51}	S_{52}	S_{53}	S_{54}	S_{55}
S_1	1	3	5	6	6	7	**10**
S_2	1	1	1	0	1	2	**1**

S_{Out} =	49	-	50	-	51	53	54	54	-	**55**

8:

S_{In} =	49	54	53	50	51	54	55	**49**	54	50

Index	S_{49}	S_{50}	S_{51}	S_{52}	S_{53}	S_{54}	S_{55}
S_1	**1**	3	5	6	6	7	10
S_2	**2**	1	1	0	1	2	1

S_{Out} =	49	**49**	50	-	51	53	54	54	-	55

9:

S_{In} =	49	54	53	50	51	54	55	49	**54**	50

Index	S_{49}	S_{50}	S_{51}	S_{52}	S_{53}	S_{54}	S_{55}
S_1	1	3	5	6	6	**7**	10
S_2	2	1	1	0	1	**3**	1

S_{Out} =									
49	49	50	-	51	53	54	54	**54**	55

10:

S_{In} =									
49	54	53	50	51	54	55	49	54	**50**

Index	S_{49}	S_{50}	S_{51}	S_{52}	S_{53}	S_{54}	S_{55}
S_1	1	**3**	5	6	6	7	10
S_2	2	**2**	1	0	1	3	1

S_{Out} =									
49	49	50	**50**	51	53	54	54	54	55

Am Ende steht dann die sortierte Folge:

S_{Out} =									
49	49	50	50	51	53	54	54	54	55

Algorithmus in Pascal-Notation:

Gegeben sind ein Feld $S_{In}[1, ..., N]$ der Länge N (also ein Feld, dass die Elemente von $[1..N]$ enthält), ein Feld $S_{Out}[1, ..., N]$ der Länge N, in das die Folge S_{In} einsortiert wird, ein Feld $S_1[min, ..., max]$, in dem für jedes Element, dass sich in S_{In} befinden kann, zunächst notiert wird, wie oft dieses auftritt und ein Feld $S_2[min, ..., max]$, in dem für jedes Element der Folge S_1 notiert wird, wie oft es bereits an S_{Out} übertragen wurde. Die Wahl des Wertes 1 für die Untergrenze dient hier der einfachen Ermittlung des Aufwandes, in der Praxis kann das Feld auch von $[x, ..., N + x]$ gewählt werden.

Die Grenzen *min* und *max* stellen im Allgemeinen Teilbereiche der Mengen dar, die die Elemente für S_{In} bilden können. Während also die $S_{In}[i]$ z.B. einen Bereich von 0 bis 150 umfassen können (etwa um das Alter von Kunden problemlos zu speichern), ergibt sich bei der Anwendung die Konstellation, dass lediglich Objekt deren Alter von 40 – 60 Jahren liegt, sortiert werden müssen. Dies würde hier also bedeuten $min = 40$ und $max = 60$ zu wählen.

Zeile

```
        // Initialisierung von S1[i] un S2[i]
[ 1]    for i := min to max do          // von relat. min
[ 2]    begin                           // bis relat. max
[ 3]      S1[i] := 0;                   // ...initialisiere S1[i]
[ 4]      S2[i] := 0;                   // ...initialisiere S2[i]
[ 5]    end;

        // Notieren, wie viele Elemente wie oft da sind
[ 6]    for i := 1 to n do              // von absol. unten bis
[ 7]    begin                           // absol. oben
[ 8]      temp  := SIn[i]               // ...sum. nach s1, welche
[ 9]      s1[i] := s1[i] + 1;           // ...Elem. wie oft da sind
[10]    end;

[11]    temp  := 0;                     // temp leer machen

        // Akkumulation von S1 bilden und nach S1[max] notieren
[12]    for i := min to max - 1 do      // von relat. min bis max
[13]      temp   := temp  + S1[i];      // ...akkumul. S1 nach temp
[14]    S1[max] := temp  + 1;           // S1[max] := Akkumul(S1)

        // Von S1[max] aus, akkumulierte Werte für alle S1 bilden
[15]    for i := max - 1 downto min do// von rel. max-1 bis min
[16]      S1[i] := S1[i + 1] – S1[i];   // ... bilde akkum S1[i]
```

```
       // Von SOut füllen + S2 updaten
[17]   for i := 1 to n do                // von abs. unten bis oben
[18]   begin
[19]     temp      := SIn[i];            // notiere zu bearb. Elem
[20]     k         := S1[i] + S2[i];     // errechne Pos. für SOut
[21]     SOut[k]   := SIn[i];            // schreibe Elem nach SOut
[23]     S2[temp]  := S2[temp] + 1;      // incr. Wert f. Elem in S2
[24]   end;
```

Analyse

Versehen wir auch dieses Listing mit Kosten und den Häufigkeiten für deren Auftreten:

Zeile		*Kosten*	*Anzahl*
	`// Initialisierung von S1[i] un S2[i]`		
[1]	`for i := min to max do`	c_1	$(max - min) + 1$
[2]	`begin`		
[3]	`  S1[i] := 0;`	c_2	$max - min$
[4]	`  S2[i] := 0;`	c_3	$max - min$
[5]	`end;`		
	`// Notieren, wie viele Elemente wie oft da sind`		
[6]	`for i := 1 to n do`	c_4	$n + 1$
[7]	`begin`		
[8]	`  temp := Sin[i]`	c_5	n
[9]	`  s1[temp] := s1[temp] + 1;`	c_6	n
[10]	`end;`		
	`// Akkumulation von S1 bilden und nach S1[max] notieren`		
[11]	`temp := 0;`	c_7	1
[12]	`for i := min to max - 1 do`	c_8	$(max - min) + 1$
[13]	`  temp := temp + S1[i];`	c_9	$max - min$
[14]	`S1[max] := temp + 1;`	c_{10}	1
	`// Von S1[max] aus, akkumulierte Werte für alle S1 bilden`		
[15]	`for i := max - 1 downto min do`	c_{11}	$max - min$
[16]	`  S1[i] := S1[i + 1] - S1[i];`	c_{12}	$max - min$
	`// Von SOut füllen + S2 updaten`		
[17]	`for i := 1 to n do`	c_{13}	$n + 1$
[18]	`begin`		
[19]	`  temp := SIn[i];`	c_{14}	n
[20]	`  k := S1[temp] + S2[temp];`	c_{15}	n
[21]	`  SOut[k] := SIn[i];`	c_{16}	n
[22]	`  S2[temp]:= S2[temp] + 1;`	c_{17}	n
[23]	`end;`		

Damit ergibt sich $T(n)$ wie folgt:

$$
\begin{aligned}
T(n) \quad = \; & c_1\,(max - min + 1) + c_2\,(max - min) + c_3\,(max - min) + c_4\,(n + 1) \\
& + c_5 n + c_6 n + c_7 + c_8\,(max - min + 1) + c_9\,(max - min) + c_{10} \\
& + c_{11}\,(max - min) + c_{12}\,(max - min) + c_{13}\,(n + 1) + c_{14} n \\
& + c_{15} n + c_{16} n + c_{17} n
\end{aligned}
$$

Die Variablen *max* und *min* lassen sich formal nicht auf *n* zurückführen, sie stehen allerdings inhaltlich in einem direkten Zusammenhang:

$$[min,\ldots, max] \subseteq [1,\ldots, n]$$

Wenn kein Element des Feldes mehrfach vorkommt und wenn jedes mögliche Element des definierten Intervalls im Feld enthalten ist, sind $min = 1$ und $max = n$ (Worst-Case). Dies angenommen, vereinfacht sich $T(n)$:

$$T(n) = c_1 (n+1) + c_2 n + c_3 n + c_4 (n+1) + c_5 n + c_6 n + c_7 + c_8 (n+1) + c_9 n + c_{10} + c_{11} n + c_{12} n + c_{13} (n+1) + c_{14} n + c_{15} n + c_{16} n + c_{17} n$$

Dies lässt sich ausklammen:

$$T(n) = c_1 n + c_1 + c_2 n + c_3 n + c_4 n + c_4 + c_5 n + c_6 n + c_7 + c_8 n + c_8 + c_9 n + c_{10} + c_{11} n + c_{12} n + c_{13} n + c_{13} + c_{14} n + c_{15} n + c_{16} n + c_{17} n$$

Eine Zusammenfassung der Terme mit Graden von n (hier lediglich n^1 und n^0) ergibt:

$$T(n) = (c_1 + c_2 + c_3 + c_4 + c_5 + c_6 + c_9 + c_{11} + c_{12} + c_{13} + c_{14} + c_{15} + c_{16} + c_{17})\, n + c_1 + c_4 + c_7 + c_8 n + c_8 + c_{10} + c_{13}$$

Mit den Substitutionen

$$a = c_1 + c_2 + c_3 + c_4 + c_5 + c_6 + c_9 + c_{11} + c_{12} + c_{13} + c_{14} + c_{15} + c_{16} + c_{17})$$

und

$$b = c_1 + c_4 + c_7 + c_8 n + c_8 + c_{10} + c_{13}$$

erhält man eine Gleichung der Form $an + b$, eine Geradengleichung, deren Verlauf linear ist. Damit liegt der Rechenaufwand für Countsort II bei $O(n)$.

Literatur - kommentiert für Lernende

[AU 92] Aho, A.V. und Ullmann, Foundations of Computer Science. Computer Science Press, 1992.

> *Das Kapitel zum O-Kalkül The Running Time of Programs ist eine hervorragende Erläuterung zu diesem Thema. Würden zu den vielen Aufgabenanregungen Musterlösungen vorliegen, könnte dies ein unschlagbares Werk sein. Leider verlassen sich auch hier die Autoren zu sehr darauf, dass der Leser alles ‚leicht sieht', was sie mit ihrer Erfahrung sicherlich leicht sehen.*

[BB 96] Brassard, G. und Bratley, P., *Fundamentals of Algorithmics*. Prentice Hall, 1996.

> *Ein gutes Kapitel, dass in die asymptotische Notation einführt. Ansonsten viel Material über Algorithmen, dass nicht immer übersichtlich wirkt.*

[CL+ 96] Cormen, Thomas, Leisserson, Charles E. und Rivest, Ronald L., *Introduction to Algorithms*. MIT-Press und MacGraw-Hill, 1996.

> *Eines der besten mir bekannten Bücher zu diesem Thema. Auch diese Einführung wurde in Teilen davon angeregt. Allerdings lassen sich die Autoren das wirklich reichliche Wissen manchmal nicht ganz einfach entlocken. Immerhin: Im Gegensatz zu den meisten anderen Büchern, die sich mit diesem Thema auseinandersetzen, kann man es mit diesem*

überhaupt lernen. Eine Vielzahl anderer Bücher ermöglicht lediglich ein Nachvollziehen, wenn man ein Thema sowieso schon gut verstanden hat.

[Fac 85] Fachredaktionen des Bibliographischen Instituts [Hrsg.], *Duden Rechnen und Mathematik*. Mayers Lexikonverlag 1885.

Wurde nur zum Nachschlagen bei einigen Definitionen benutzt – ohne Bedeutung für das eigentliche Thema.

[Eng 00] Engelmann, Lutz [Hrsg.], *Kleiner Leitfaden, Informatik und ihre Anwendugen*. paetec Gesellschaft für Bildung und Technik GmbH, 2000.

Gute Übersicht zum Thema (vom Konzept her auch als solche gedacht), zum wirklichen Lernen jedoch eindeutig zu wenig.

[OW 90] Ottmann, T. und Widmayer P., *Algorithmen und Datenstrukturen*. Bibliographisches Institut, Reihe Informatik, Band 70, 1990.

Zwar eines der besten Bücher zum Thema, leider wird jedoch ausgerechnet die asymptotische Notation im Galopp verloren – zu kurz, keine Lösungsmuster, lediglich die Definitionen. Die Analyse der Algorithmen ist hingegen gut und verständlich. Wäre das O-Kalkül nicht so knapp dargestellt, handelte es sich hierbei wohl um das herausragende Lehrbuch schlechthin. So braucht der Lernende leider mindestens ein Weiteres neben diesem.

[Raw 92] Rawlins, G. J.E., *Compared to what? An introduction to the analysis of algorithms*. Computer Science Press, 1992.

Dies Werk macht insgesamt den Eindruck eines hervorragenden Lehrbuchs zum Thema. Leider gilt auch hier: Zu den durchweg vielfältigen und exzellenten Anregungen zum Üben fehlen Musterlösungen.

[Sch 97] Schöning, U., *Algorithmen – kurz gefasst*. Spektrum Akademischer Verlag, 1997.

In vielerlei Hinsicht ein gutes Buch, etwa wenn ein Lehrveranstalter es mit den Lernenden durchgeht. Zum Lernen alleine – ohne eine auf das Buch gestützte Lehrveranstaltung – eher ungeeignet. Das Motto „kurz gefasst" dürfte es dem Lernenden doch relativ schwer machen, sich das durchaus enthaltene Wissen alleine zugänglich zu machen.

[Sch 80] Schülkes Tafeln. B. G. Teubner, Stuttgart, 1980.

Wurde nur zum Nachschlagen bei einigen Definitionen benutzt – ohne Bedeutung für das eigentliche Thema.

[SF 96] Sedgewick, R. und Flajolet, P., *An Introduction to the Analysis of Algorithms*. Addison-Wesley, Reading, 1996.

In diesem Werk steht schon einiges zur Sache, viel Freude kommt beim Lesen jedoch nicht auf. Die Autoren haben sich um die Verständlichkeit gegenüber dem Lernenden offensichtlich nicht allzu viele Sorgen gemacht.

Beispiele zur Erstellung von einfachen Java-Programmen

Der Quellcode eines einfachen Java-Programms besteht aus einer Datei mit dem Suffix `.java`. In einer solchen Datei wird eine Klasse gleichen Namens definiert. Als Standardbeispiel dient die folgende Datei HalloWelt1.java

```
/* Langer Kommentar ueber mehrere Zeilen:
   HalloWelt1.java
   Ole Schulz-Trieglaff
   Dies ist ein ziemlich sinnfreies Programm.
*/
public class HalloWelt1 { // Kurzkommentar: Beginn der Klassendefinition

        public static void main(String[] args) { // Methode main
                System.out.print("Ich: Hallo ");
                System.out.println("Welt !!");
                System.out.println();
                System.out.println("Welt: Hallo Du!!");
                System.out.println();
                hallo();
        }

        public static void hallo() { // Methode hallo
          System.out.println("Ich: Wie geht es Dir?");
        }

}       //Ende der Klassendefinition
```

Mit dem Befehl `javac HalloWelt1.java` wird die Datei HalloWelt1.class erzeugt, welche mit dem virtuellen Prozessor JVM auf jeder Plattform ausgeführt (genauer interpretiert) werden kann. Das geschieht durch den Aufruf `java HalloWelt1`. Dabei ist wichtig, dass die Datei eine Methode `main` mit der oben beschriebenen Signatur besitzt. In unserem Beispiel wird in der Methode `main` die Methode `hallo` aufgerufen. Dieser prozedurale Aufbau ist ein äußerst wichtiger Aspekt bei der Erstellung von komplexen Programmen, hier hätte man an Stelle der Anweisung `hallo();` in `main` auch die Codezeile aus der Definition von `hallo()` schreiben können.

Das folgende Beispiel zeigt, wie man Eingaben in der Kommandozeile übergeben kann, nämlich als Zeichenfolgen (Strings), die als Argumente im Aufruf nach dem Klassennamen folgen. Die Eingabeparameter sind durch (ein oder mehrere) Leerzeichen voneinander getrennt. Die Anzahl dieser Parameter kann man durch `args.length` abfragen. In unserem Beispiel werden sie als Strings übernommen (mit einer for-Schleife), konkateniert (mit jeweils einem Leerzeichen als Zwischenraum) und am Ende wieder ausgegeben. Der Aufruf `java Echo Bla bla bla` würde also die Ausgabe `Bla bla bla ` erzeugen.

```
public class Echo {
        public static void main(String[] args) {
                String echo;  // Deklaration der Variablen echo
                echo = "";    // Initialisierung der Variablen echo
                for (int i=0;i<args.length;i++) {
                        echo = echo +  args[i] + " ";
                }
                System.out.println(echo);
        }
}
```

Für die Übernahme von Zahlen als Eingabeparameter verwendet man spezielle Methoden aus einer Bibliothek. Natürlich muss man dabei den Typ des Eingabeparameters angeben. Das Beispielprogramm berechnet $n!$, d.h. der Eingabeparameter n ist von Typ `int`. Hier wird in der Methode `main` nur die Ein-Ausgabe vorgenommen und die Methode `fakultaet` aufgerufen, in der die Berechnung von $n!$ durch rekursiven Aufruf erfolgt. Man könnte $n!$ natürlich auch mit einer Schleife berechnen.

```
public class Fakultaet {

        public static void main(String[] args) {

                int n;
                n = Integer.parseInt(args[0]);

                System.out.println("Berechne " + n + "!");
                int fakult = fakultaet(n);    // Berechne die Fakultaet
                System.out.println(fakult);

        }

        // Rekursive Funktionen in Java
        public static int fakultaet(int n) {

                if (n==1) {
                        return 1;
                } else {
                        return n * fakultaet(n-1);
                }

        }

}
```

Das folgende Beispiel zeigt, dass man auch mehrere Parameter von verschiedenen Typen übergeben kann. Darüber hinaus wird demonstriert, wie eine for-Schleife durch eine while-Schleife ersetzt werden kann.

```
public class Exponential {

        public static void main(String[] args) {

                float basis = Float.parseFloat(args[0]);
                int exponent  = Integer.parseInt(args[1]);

                if (exponent < 0) {
                        System.out.println("Ungueltige Eingabe !");
                        System.exit(0);
                }
                float ergebnis = 1;
                System.out.print(basis + " hoch " + exponent + " ergibt: ");

                /*
                for (int i=0; i<exponent; i++) {
                        ergebnis = ergebnis * basis;
                }*/

                while(exponent > 0) {
                        ergebnis *= basis;
                        exponent--;
                }
                System.out.println(ergebnis);
        }
}
```

Das letzte Beispiel zeigt die Verwendung einer switch-Anweisung. Ziel ist es, jeder Zahl den richtigen Wochentag zuzuordnen, wenn man die Zählung der Tage an einem Montag mit Null beginnt. Switch-Anweisungen dienen zur Fallunterscheidung bezüglich der Werte, die eine bestimmte Variable (hier `rest`) annehmen kann. Die einzelnen Fälle weden mit `case` aufgeführt. Mit `default` werden alle noch nicht behandelten Fälle bezeichnet (ähnlich wie otherwise in Haskell). Die `break;`-Anweisungen am Ende jedes Falls dienen dazu, die Fallunterscheidung vorzeitig zu beenden.

```
public class Wochentag {

        public static void main(String[] args) {

                int nummer = Integer.parseInt(args[0]);
                int rest = nummer % 7;  // '%' ist die Modulo-Funktion
                String wochentag = "";

                switch(rest) {

                        case 0:
                          wochentag = "Montag";
                          break;

                        case 1:
                          wochentag = "Dienstag";
                          break;

                        case 2:
                          wochentag = "Mittwoch";
                          break;

                        case 3:
                          wochentag = "Donnerstag";
                          break;

                        case 4:
                          wochentag = "Freitag. Informatik B Zettel abgeben !";
                          break;

                        case 5:
                          wochentag = "Samstag => Wochenende !!";
                          break;

                        default :
                          wochentag = "Sonntag => Wochenende !!";
                          break;
                }
                System.out.println("Heute ist " + wochentag);

        }

}
```

Zusammenfassung

Einige Begriffe und Zusammenhänge, die an Hand der Beispielprogramme eingeführt und demostriert wurden, verdienen es, noch einmal hervorgehoben zu werden:

Variable

Jede Variable hat einen Typ, der durch eine Deklaration festgelgt werden muss. Eine Wertzuweisung kann in der Deklaration oder auch später erfolgen.
Variable sind innerhalb des kleinsten Blocks, in dem sie deklariert wurden, sichtbar. Ein Block ist eine von { } eingeschlossene Gruppe von Anweisungen. Sichtbarkeit in ihrem Block bedeutet auch Sichtbarkeit in jedem Unterblock dieses Blocks.

Bedingungen

Eine Bedingung ist ein Ausdruck, dessen Auswertung ein Boolescher Wert ist.
Die Booleschen Werte sind in Java mit `false` und `true` bezeichnet. Durch Anwendung Boolescher Operationen können zusammengesetzte Ausdrücke gebildet werden.

Kontrollstrukturen

Dieser Begriff ist eine Zusammenfassung für

Bedingungsanweisungen (if/else),
Auswahlanweisungen (switch),
Schleifenanweisungen (for, while, do while) und
Sprunganweisungen (break, continue, return).

Bedingungsanweisungen

Eine `if`-Anweisung hat die Syntax

```
if (bedingung) anweisung1;
```

und bewirkt, dass `anweisung1;` genau dann ausgeführt wird, wenn die Auswertung von `bedingung` den Wert `true` ergibt (bei `false` geht es gleich mit der nächsten Anweisung weiter). Man beachte, dass `anweisung1;` auch für einen Block von Anweisungen stehen kann (Klammern nicht vergessen!).
Eine `if/else`-Anweisung hat die Syntax

```
if (bedingung) anweisung1; else anweisung2;
```

und bewirkt, dass `anweisung1` genau dann ausgeführt wird,, wenn die Auswertung von `bedingung` den Wert `true` ergibt, bei `false` wird `anweisung2` ausgeführt. Der Unterschied wird durch die folgenden zwei Codezeilen deutlich, von denen die erste die Signumfunktion $y = sign(x)$ berechnet und die zweite die Variable x in $|x|$ verwandelt:

```
if ( x == 0 ) y = 0; else if ( x < 0 ) y = -1; else y = 1;
if ( x < 0 ) x = -x;
```

Schleifenanweisungen

Ursprünglich wurden `for`-Schleifen als reine Zählschleifen verwendet, die Syntax in Java ermöglicht eine wesentlich größere Flexibilität:

```
for (initialisierung; bedingung; modifizierung) anweisung;
```

Hier repräsentiert `anweisung;` die Schleife selbst (ist deshalb meistens ein Block). Die Anzahl der Durchläufe wird in der Regel durch eine Schleifenvariable gesteuert. Vor jedem Durchlauf wird geprüft, ob `bedingung` noch erfüllt ist, wenn nicht, erfolgt der Abbruch, sonst der nächste Durchlauf und danach die Modifizierung der Variablen.
Das Gleiche kann man mit einer `while`-Schleife realisieren. Dazu muss die Initialisierung vor der `while`-Anweisung erfolgen. Die Modifizierung ist Bestandteil der Schleife (`anweisung`) selbst. Wir haben folgende Syntax:

```
while (bedingung) anweisung;
```

In `while`-Schleifen erfolgt die Überprüng der Bedingung immer vor dem Schleifendurchlauf. Die Umkehrung dieser Reihenfolge kann durch `do-while`-Schleifen realisiert werden. Insbesondere ist bei `do-while`-Schleifen gesichert, dass mindestens ein Schleifendurchlauf gestartet wird.

Sprunganweisungen

Die Anweisung `break;` kann in einer Schleifen- oder Auswahlanweisung verwendet werden und bewirkt die sofortige Beendung dieser Anweisung. Bei geschachtelten Schleifen- und Auswahlanweisungen betrifft das immer die innerste Anweisung, zu der das `break;` gehört. Darüber hinaus kann man Blöcke mit einem Namen (Label) bezeichnen und eine `break`-Anweisung mit diesem Namen verwenden, um den Block zu beenden.

Die Anweisung `continue;` kann nur in Schleifen verwendet werden. Sie bewirkt einen Abbruch des aktuellen Durchlaufs und einen Sprung zum nächsten Durchlauf.

Mit der `return`-Anweisung erfolgt die Rückgabe des Ergebnisses einer Methode und die Beendigung des Methodenaufrufs.

Vererbung (Inheritance)

Vererbung ist ein zentraler Bestandteil der Objektorientierung. Man beschreibt damit die Möglichkeit, Eigenschaften und Methoden vorhandener Klassen auf andere (neue) Klassen zu übertragen. Leider ist die Terminologie auf den ersten Blick etwas verwirrend:
Die erbende Klasse wird **Ableitung** oder **Spezialisierung** der vererbenden Klasse genannt, aber auch **Erweiterung** oder **Unterklasse**. Für die vererbende Klasse verwendet man den Begriff **Oberklasse**. Wenn es sich um eine direkte Vererbung (ohne Zwischenklassen) handelt, sprechen wir auch von der **Superklasse** der entsprechenden Unterklasse.
Da jedes Objekt einer Unterklasse alle Eigenschaften der Oberklasse besitzt (es hat sie geerbt), kann es auch gleichzeitig als Objekt der Oberklasse betrachtet werden. Diese Vielgestaltigkeit von Objekten wird als **Polymorphie** bezeichnet und ist eine weiterer zentraler Begriff der Objektorientierung.

Um die Begriffe besser zu verstehen, kann man das folgende Standardbeispiel betrachten. Eine Klasse `Person` wird durch die Atribute Namen und Geburtsjahr charakterisiert.
Beschreibt man `Student` als eine Klasse, die alle Eigenschaften von Person erbt und dazu noch durch eine Matrikelnummer, Hauptfach und Immatrikulationsjahr charakterisiert ist, dann wird die Klasse Student aus der Klasse Person abgeleitet. Ein Student hat spezielle Eigenschaften, die nicht jede beliebige Person hat – deshalb sprechen wir von einer Spezialisierung. Gleichzeitig ist Student (im Sinne der charakterisierenden Eigenschaften) eine Erweiterung von Person, denn zu den Personeneigenschaften kommen weitere Attribute (und möglicherweise auch neue Methoden). Da jeder Student auch eine Person ist, kann man im Sinne der Mengenlehre von einer Unterklasse (Untermenge) sprechen.

Vererbung ist ein transitiver Begriff. Wenn Informatikstudent eine Unterklasse von Student ist und Student eine Unterklasse von Person , dann ist Informatikstudent auch eine Unterklasse von Person (mit Namen, Geburtsjahr und allen anderen sichtbaren Attributen und Methoden von Person). Im Gegensatz zu einigen anderen objektorientierten Programmiersprachen kennt Java keine Mehrfachvererbung, d.h. eine Klasse A kann nur eine direkte Oberklasse B haben (man nennt B die Superklasse von A), alle anderen Oberklassen von A müssen auch Oberklassen von B sein. Die Zuordnung zur Superklasse B erfolgt schon bei der Defintion von A durch `public class A extends B{....`
Wie wir aus den ersten Beispielen wissen, kann bei einer Klassendefinition auch auf den Zusatz `extends B` verzichtet werden. In diesem Fall wird die Klasse `Object` implizit als Superklasse festgelegt. Diese Klasse ist in Java die einzige, die keine Superklasse besitzt. Alle anderen Klassen bilden unter `Object` eine baumförmige Klassenhierarchie, d.h. jede Klasse A ist Unterklasse von `Object` und der aufsteigende Weg von A zu `Object` ist eindeutig.

Das folgende Beispiel demonstriert einige Aspekte, die bei der Vererbung zu beachten sind. Die Oberklasse `Rectangle` gibt eine relativ abstrakte Beschreibung eines Rechtecks nur durch seine Seitenlängen. Spezielle Rechtecke, bei denen die Lage in der Ebene eine Rolle spielt, werden durch die Unterklasse `RectangleInPlane` beschrieben.

```
public class Rectangle{
   // *********************************************************
   // * "abstraktes" Rechteck ohne konkrete Lage in der Ebene   *
   // *********************************************************

   // *********************************************************
   // * Attribute (Variablen)                                   *
   // *********************************************************
   // Zwei Seitenlaengen
   public double a,b;

   // *********************************************************
   // * Konstruktor(en)                                         *
   // *********************************************************
   public Rectangle(double a, double b){
      this.a = a;
      this.b = b;
   }

   // *********************************************************
   // * Methoden                                                *
   // *********************************************************
   // Flaeche berechnen
   public double area(){
      return a * b;
   }

   // weitere Methoden wie z.B. Umfang berechnen ...
   //Gleichheitstest
   public boolean equals(Rectangle r){
         return((a = r.a && b = r.b) || (a = r.b && b == r.a));
   }
}
```

Bei der Erweiterung ist neben dem Zusatz `extends Rectangle` zu beachten, dass jeder Konstruktor zuerst einen Konstruktor der Superklasse aufrufen muss. Der Aufruf erfolgt aber nicht durch `Rectangle(a,b)` sondern durch `super(a,b)`. In diesem Beispiel wurde auf den einfachsten Konstruktor verzichtet, dem man einfach die 4 Eckpunkte übergibt und der daraus die Seitenlängen bestimmt (die Klasse Point wird hier in der einfachen Version verwendet, bei der die Koordinaten public sind). An Stelle dessen findet man die Beschreibung eines Konstruktors, der neben den Seitenlängen einen Eckpunkt pA und den Anstiegswinkel der Seite (pA,pB) verwendet.

```java
public class RectangleInPlane extends Rectangle{
   // *******************************************************
   // * Rechteck mit konkreter Lage in der Ebene            *
   // *******************************************************

   // *******************************************************
   // * Attribute (Variablen)                               *
   // *******************************************************
   // Seitenlaengen schon in Rectangle
   // 4 Eckpunkte (entgegen Uhrzeigerrichtung)
    Point pA, pB, pC, pD;
   // Winkel zwischen x-Achse und Grundseite im Bogenmass
   // Konvention: Seite mit Laenge a verlaeuft von pA nach pB
   double alpha;

   // *******************************************************
   // * Konstruktor(en)                                     *
   // *******************************************************
   // Konstruktor mit Punkt pA, Winkel und beide Seitenl\"angen gegeben
   public RectangleInPlane(Point p, double alpha, double a, double b){
      super(a,b);  // Konstruktor fuer Rectangle
      pA = p;
      pB = new Point((p.x)+Math.cos(alpha)*a, (p.y)+ Math.sin(alpha)*a);
      pD = new Point((p.x)-Math.sin(alpha)*b, (p.y)+ Math.cos(alpha)*b);
      pC = new Point((p.x) + Math.cos(alpha)*a - Math.sin(alpha)*b,
                     (p.y) +  Math.sin(alpha)*a + Math.cos(alpha)*b);
      this.alpha = alpha;
   }

   // weitere Konstruktoren denkbar, z.B. wenn pA, pB und Seite b gegeben
   // dann wird a als pA.dist(pB) berechnet

   // *******************************************************
   // * Methoden                                            *
   // *******************************************************
   // Mittelpunkt
   public Point center(){
        Point p = new Point(0.5*((pA.x) + (pC.x)),
                            0.5*((pA.y) + (pC.y)));
        return p;
   }
}
```

Man beachte, dass zur Gleichheit von zwei Objekten von `Rectangle` nur die Seitenlängen gleich sein müssen, bei zwei Objekten von `RectangleInPlane` auch die Lage übereinstimmen (also die Mengen der Eckpunkte). Wir verzichten hier auf die konkrete Beschreibung der Methode `equals` für `RectangleInPlane` (umfangreiche Fallbetrachtung auf Grund der möglichen Symmetrien) und kommen zum entscheidenden Punkt: In beiden Klassen gibt es eine Methode gleichen Namens und auf Grund der Polymorphie ist nicht mehr klar, welche von beiden gemeint ist, wenn sie für ein `RectangleInPlane`-Objekt aufgerufen wird. Man könnte an dieser Stelle noch argumentieren, dass die Parameterlisten verschieden sind, aber auch dort ist durch die Polymorphie keine Eindeutigkeit gegeben. Hier ist ein anderes Beispiel, bei dem selbst die Parameterlisten keine Unterscheidung erlauben: Definiert man eine Klasse `Triangle` für Dreiecke durch 3 Seitenlängen, so ist die Flächenberechnung relativ kompliziert (erfordert Winkelfunktionen oder die Wurzelfunktion). Dagegen gibt es für `TriangleInPlane` eine einfache Formel, die Fläche aus den Eckpunktkoordinaten zu berechnen. Man würde also die Methode `area()` **überschreiben**. Bleibt zu klären, wann welche Methode verwendet wird und wie man das beeinflussen kann.
Bei der Polymorphie ist es wichtig, zwischen den Typ einer Referenz (Verweis) und dem Typ des Objekts (Exemplar, Instanz) zu unterscheiden, auf das verwiesen wird. Wir erklären das an einem schematischen Beispiel von zwei Klasen A und B:

```
public class A{
    public int a;
    public A() {a = 1;} // Konstruktor
    public int meth() {return a;}
}
public class B extends A{
    public int a; // Attribut a aus A wird "beschattet", es gibt a doppelt
    public A() {
        super();
        a = 100;
    }
    public int meth() {return a;} // das ist das a aus B
}
```

Wird ein Attribut aus der Oberklasse in der Unterklasse noch einmal neu deklariert, spricht man von Beschattung, bei Methoden von Überschreibung. Diese begriffliche Trennung wird sich als wichtig erweisen. Mit den Anweisungen

```
B bref = new B();
A aref = bref; // Namen betonen, dass es um Referenzen geht
```

wird ein Objekt von Typ B angelegt, auf das zwei Referenzen verweisen: Die Referenz bref vom Typ B und die Referenz aref vom Typ A. Für die zweite Zuweisung muss der Typ von bref nicht explizit umgewandelt werden. Bei aref sind Typinformationen verloren gegangen, d.h. es ist nicht mehr ersichtlich, dass der Verweis auf ein Objekt vom Typ B zeigt. Man kann das aber mit der Bedingung `(aref instanceof B)` abfragen, die hier den

Wert true bekommt. In diesem Fall ist dann eine explizite Typumwandlung möglich:
`B cref = (B)aref; // ohne Typumwandlung --> Fehler`
Den Unterschied zwischen Beschatten und Überschreiben kann man nun in einen kurzen Merksatz fassen: Beim Beschatten (Attribute) entscheidet der Typ der Referenz, beim Überschreiben der Typ des Objekts (der Instanz). Am Beispiel bedeutet das:

```
int i;
i = bref.a;   // i hat den Wert 100
i = aref.a;   // i hat den Wert 1
i = ((A)bref).a; // explizite Typunwandlung der Referenz, i hat den Wert 1
i = ((B)aref).a; // explizite Typunwandlung der Referenz, i hat den Wert 100
i = bref.super.a;   // Syntaxfehler
i = bref.meth(); // i hat den Wert 100
i = aref.meth(); // Typ des Objekts entscheidet, i hat den Wert 100
i = ((B)aref).meth(); // nur Umwandlung des Typs der Referenz, i bleibt 100
i = ((A)bref).meth(); // nur Umwandlung des Typs der Referenz, i bleibt 100
i = bref.super.meth(); // Syntaxfehler
```

Der Grund für die Syntaxfehler liegt darin, dass die Referenz `super` auf das Objekt der Superklasse kein public-Attribut der Unterklasse ist, sondern nur wie ein private-Attribut verwendet werden kann, also nur in der Klassendefinition der Unterklasse. Aus dem gleichen Grund ist es nicht möglich `super.super` zu verwenden. Es gibt deshalb nur eine abgeschwächte Möglichkeit, auf meth() der Superklasse zuzugreifen, nämlich indem man in der Definition eine Methode
`public int super_a() {return super.meth();}`
unterbringt. Dann würde der Aufruf `i = bref.super_a();` die Methode `meth` aus A verwenden und folglich `i` mit dem Wert 1 belegen.

Das Prinzip, Methoden nicht nach dem (deklarierten) Typ der Referenz, sondern jeweils nach dem Typ des referenzierten Objekts auszuwählen, führt dazu, dass diese Entscheidung noch nicht bei der Übersetzung sondern erst zur Laufzeit getroffen werden kann. Wir sprechen deshalb von einer dynamischen Methodenauswahl (dynamic method lookup), die natürlich mehr Rechenzeit als eine statische Auswahl erfordert. Deshalb kann es sinnvoll sein, Methoden - für den Compiler eindeutig erkennbar - vor Überschreiben zu schützen. Dafür gibt es verschiedene Möglichkeiten:

- Verwendung des Modifikators `final`,
- Verwendung des Modifikators `private`, dann wird die Methode nicht vererbt und ist damit implizit final,
- Verwendung des Modifikators `static`, damit kann die Methode beschattet, aber nicht überschrieben werden,
- Verwendung des Modifikators `final` für die ganze Klasse.

Klassen und Objektorientierung

Die zentrale Idee der Objektorientierung besteht darin, die Trennung zwischen Daten und Operationen auf diesen Daten aufzuheben und beides in einem Objekt zusammenzufassen. Ein **Objekt** zeichnet sich durch die folgenden drei Bestandteile aus:

1. Eine **Identität** (repräsentiert durch einen Namen) zur Unterscheidung von andern Objekten;
2. Eine Menge von **Attributen** (Eigenschaften) zur Beschreibung des inneren Zustands;
3. Eine Menge von **Methoden** (dynamische Eigenschaften) zur Beschreibung des Verhaltens;

Unter einer **Klasse** verstehen wir die Zusammenfassung von Objekten mit gleichartigen Attributen und Methoden. Mit anderen Worten, eine Klassendefinition beschreibt die Attribute durch ihren Typ und die Methoden als Funktionen auf den Objekten. Zur Beschreibung eines Objekts müssen die Attribute mit Werten (bzw. Referenzen) belegt werden.
Ein Objekt einer Klasse nennt man alternativ Exemplar oder Instanz (schlechte Übersetzung von instance) der Klasse. Attribute werden auch Variablen, Datenfelder oder Member genannt. Methoden werden oft als Funktionen bezeichnet.
Während Klassen **statisch** sind, da sie schon zur Übersetzungszeit als Programmtext existieren, werden Objekte erst zur Laufzeit angelegt und sind deshalb **dynamisch**.
Das folgende Beispiel zeigt den prinzipiellen Aufbau einer Klassendefinition.

```
public class Point{

   // *********************************************************
   // * Attribute (Variablen)                                 *
   // *********************************************************
   // Punkt-Koordinaten
   public double x,y;

   // *********************************************************
   // * Konstruktor(en)                                       *
   // *********************************************************
   public Point(double a, double b){
      x = a;
      y = b;
   }

   // *********************************************************
   // Methoden                                                *
   // *********************************************************
   // Distanz zum Nullpunkt berechnen (keine Parameter benoetigt)
```

```
   public double distFromOrigin(){
      return Math.sqrt(x*x + y*y);
   }

   // Distanz zwischen aktuellem "Klassenpunkt" und p berechnen
   public double dist(Point p){
      return Math.sqrt((p.x-x)*(p.x-x) + (p.y-y)*(p.y-y));
   }
}
```

Wie bei allen Referenztypen wird mit der Deklaration `Point p;` nur eine Referenz auf `null` und nicht das Objekt `p` selbst angelegt. Eine Instanziierung erfolgt dann z.B. durch `p = new Point(1.2, 5);`
Jede Klasse hat spezielle Methoden zur Instaziierung von Objekten, sogenannte **Konstruktoren**. Folgendes ist dabei zu beachten:

1. Alle Konstruktoren haben den gleichen Namen wie die Klasse. Für eine Klasse kann man verschiedene Konstruktoren definieren, diese müssen sich aber durch verschiedene Parameterlisten unterscheiden.

2. Konstruktoren haben keinen Rückgabetyp.

3. Gibt es keinen definierten Konstruktor, so ist implizit ein Default-Konstruktor definiert (mit leerer Parameterliste), der alle Attribute mit den Default-Werten belegt.

4. Wenn ein explizit definierter Konstruktor existiert, so gibt es keinen Default-Konstruktor.

5. Man kann für Parameter von Konstruktoren (und von anderen Methoden) auch Namen von Attributen verwenden (Beschattung). In diesem Fall muss man, wie das folgende Beispiel zeigt, die Referenz `this` auf das aktuelle Objekt verwenden, um auf die beschatteten Attribute zuzugreifen.

```
public Point(double x, double y){
       this.x = x; // linke Seite Attribut, rechts uebergebener Parameter
       this.y = y;
}
```

Für alle Attribute und alle Methoden, die kein Konstruktor sind, muss ein **Rückgabetyp** deklariert werden. Möglich sind primitive Typen, Klassen und Felder. Ist ein Objekt definiert, erfolgt der Zugriff auf seine Eigenschaften, indem man vor den Namen der Eigenschaft den Objektnamen mit einem Punkt setzt, wie z.B. `p.x` oder `p.distFromOrigin()`. Wir verwenden hier und im Folgenden Eigenschaften als Sammelbegriff für Attribute und Methoden. Durch sogenannte **Modifikatoren** kann man Einfluss auf die Sichtbarkeit (den Zugang) zu den Eigenschaften nehmen. Mit `public` deklarierte Eigenschaften sind überall sichtbar und damit verwendbar, mit `private` deklarierte Eigenschaften dagegen nur in der eigenen Klassendefinition. Damit kann man Attribute nach außen verbergen oder lesbar

machen, bei gleichzeitigem Schreibschutz. Das folgende Beispiel zeigt, wie man die Klasse `Point` ändern muss, um die Koordinaten vor Änderungen zu schützen und gleichzeitig ihre Lesbarkeit zu sichern:

```
private double x, y; // ausserhalb der Klassendefinition nicht mehr sichtbar
public double get_x(){ // Lesefunktion ist nach aussen sichtbar
        return x;
}
// get_y() analog
```

Den hier skizzierten Mechanismus nennt man **Datenkapselung**. Damit ist gemeint, dass eine Klasse aus einem Kern von privaten (also nach außen abgeschirmten) Eigenschaften und einer Hülle von öffentlichen Eigenschaften besteht, mit denen ein potenzieller Anwender auf die eigentliche Funktionalität der Klasse zugreifen kann, wärend die für ihn unwichtigen Details der Implementierung verborgen bleiben. Dieses Entwurfsprinzip für weiter- und wiederverwendbare Software nennt man **Datenabstraktion**, die Schnittstelle wird **API** (Application Programming Interface) genannt.

Zur Verfeinerung dieses Konzepts stehen weitere Modifikatoren zur Verfügung. Um sie zu verstehen, muss man wissen, dass größere Javprojekte in Paketen organisiert werden. Eine Gruppe von Klassen mit starken inhaltlichen Bezügen kann zu einem **Paket** (`package`) zusammenfassen. Mit dem Modifikator `protected` wird eine Eigenschaft im eigenen Paket und in allen Unterklassen sichtbar. Verwendet man keinen der Modifikatoren `public, protected, private`, so ist die Eigenschaft nur im eigenen Paket sichtbar.

Mit dem Modifikator `static` werden **Klassenvariablen** und **Klassenmethoden** deklariert. Auf solche Eigenschaften kann man schon zugreifen, bevor ein entsprechendes Objekt der Klasse instanziiert ist. Man kann auf solche Eigenschaften mit dem Klassennamen und einem Punkt zugreifen.

Neben den Modifikatoren muss für alle Attribute ein Typ festgelegt werden und auch für alle Methoden, die kein Konstruktor sind, muss ein Rückgabetyp deklariert werden. Bei Methoden, die nur intern etwas ändern, aber nichts zurückgeben wird der Rückgabetyp `void` verwendet.

Schnittsstellen (Interfaces)

Der Begriff Schnittstelle fiel bereits beim Thema Datenabstraktion. Allgemein bezeichnet man damit die Daten und Funktionen, die ein Programm nach aussen zur Verfügung stellt, damit ein Anwender die Software nutzen kann, ohne das Programm selbst zu ändern. Zu diesem Zweck muss der Anwender auch nicht wissen, wie eine Methode implementiert ist - er muss nur die Signatur kennen, um die Methode syntaktisch korrekt verwenden zu können. In einer Schnittstelle kann man sich also auf die abstrakte Beschreibung von Funktionen (Deklaration) beschränken und auf die konkrete Implementierung (Definition) verzichten.

In Java ist der Begriff Schnittstelle eng mit sogenannten **abstrakten Klassen** verbunden. Damit bezeichnet man Klassen, deren Implementierung ganz oder teilweise offen ist:

- Eine **abstrakte Methode** ist eine nur deklarierte, aber nicht definierte (implementierte) Methode. Sie wird mit dem Modifikator `abstract` gekennzeichnet und erhält ein Semikolon nach der Deklaration.
- Eine Klasse, die eine abstrakte Methode enthält, muss selbst mit `abstract` gekennzeichnet werden. Eine solche Klasse kann **nicht** instanziiert werden (kein new).
- Jede Unterklasse einer abstrakten Klasse, die nicht alle abstrakten Methoden implementiert (durch Überschreiben), muss selbst abstrakt sein.
- Abstrakte Methoden können nicht static, private oder final sein.
- Auch eine Klasse ohne abstrakte Methoden kann mit `abstract` deklariert werden (kann dann nicht instanziiert werden!).

Als Beispiel kann man eine abstrakte Klasse `Shape` für ebene Figuren einführen. Diese Klasse nennt nur zwei Eigenschaften von ebenen Figuren, nämlich, dass sie eine Fläche und einen Umfang haben:

```
public abstract class Shape{
   public abstract double area();
   public abstract double circumf();
}
```

In den Unterklassen `Rectangle, Triangle, Circle` kann man diese Methoden dann implementieren. Wenn man aber z.B. für Dreiecke keine Formel kennt, die die Flächen nur aus den Seitenlängen bestimmt, ist es auch möglich, nur den Umfang zu implementieren und dafür die Klasse `Triangle` mit `abstract` zu deklarieren.
In der Klasse `Shape` sind alle Methoden abstrakt und damit ist sie auch ein Beispiel für ein **Interface**. Es gibt einen besondern Grund dafür, dass man zwischen gewöhnlichen abstrakten Klassen und Interfaces unterscheidet: Das Konzept der Einfachvererbung wird durch die Verwendung von Interfaces etwas aufgeweicht, denn eine Klasse kann neben der Vererbung durch seine Superklasse noch ein oder mehrere Interfaces implementieren, also

zusätzlich die Methoden des Interfaces erben. Konflikte, die normalerweise bei Mehrfachvererbung entstehen können (Erben von zwei gleichnamigen Methoden mit verschiedenen Definitionen), werden dabei vermieden, da die Methoden im Interface gar nicht definiert sind. Der Hauptnutzen eines Interfaces besteht also nicht in der Vererbung von konkreten Inhalten, sondern in der Beschreibung einer abstrakten Funktionalität. Jede Klasse, die das Interface implementiert, muss diese abstrakte Funktionalität durch konkrete Inhalte ausfüllen.
Folgendes sollte man beim Umgang mit Interfaces beachten:

- Das Schlüsselwort `interface` muss (an Stelle von `class`) vor der Definition stehen.
- Alle Methoden eines Interfaces sind (implizit) `public`, d.h. `private` und `protected` sind verboten.
- Alle Methoden eines Interfaces sind (implizit) `abstract`
- Ein Interface darf nur Attribute haben, die static und final sind.
- Ein Interface ist nicht instanziierbar und darf keinen Konstruktor haben.
- Wenn eine Klasse ein Interface implementiert, wird das durch das Schlüsselwort `implements` angezeigt. Die Klasse muss alle Methoden implementieren oder abstrakt sein.

Das folgende Beispiel zeigt ein Interface für Figuren, die in der Ebene liegen. Es fordert, dass man testen kann, ob ein Punkt in der Figur liegt und dass man die Figur in einer beliebigen Richtung verschieben kann:

```
public interface ShapeInPlane{
    public boolean inside(Point p); // man kann public auch weglassen
    public void shift(double x, double y);
}
```

Die Definition der Klasse `RectangleInPlane` würde dann wie folgt beginnen:

```
public class RectangleInPlane extend Rectangle implements ShapeInPlane{
```

Zur Implementierung von `shift` muss man nur die 4 Eckpunkte verschieben. Die Implementierung von `inside` ist etwas komplizierter. Eine Variante wäre der Test, ob der Punkt p im Dreieck $\Delta(pA, pB, pC)$ oder im Dreieck $\Delta(pC, pD, pA)$ liegt und diese Fälle durch Berechnung der sogenannten baryzentrischen Koordinaten (lineares Gleichungssystem) zu entscheiden. Als zweite Variante könnte man testen, ob p jeweils auf der linken Seite der vier gerichteten Geraden von pA nach pB, von pB nach pC, von pC nach pD und von pD nach pA liegt. Dazu muss man aber voraussetzen, dass das Rechteck gegen den Uhrzeiger orientiert ist.

Abstrakte Datentypen (ADTs)

Mathematische Strukturen und ihre Modellierung als ADTs

Ein **abstrakter Datentyp (ADT)** dient zur Beschreibung der (äußeren) Funktionalität und der (von außen) sichtbaren Eigenschaften von bestimmten Objekten und abstrahiert von der konkreten Art der Implementierung dieser Funktionalität. Wenn wir einen ADT zusammen mit einer Implementierung betrachten, sprechen wir von einer **Datenstruktur**. Unter spezieller Berücksichtigung der Java-Syntax kann ein ADT also durch ein Interface beschrieben werden, eine Datenstruktur ist in diesem Kontext eine Klasse, die das Interface implementiert. In der Regel sprechen wir bei einem Interface nur dann von einem ADT, wenn ein gewisser Bezug zu realen oder mathematischen Objekten gegeben ist.
Eine wichtige Motivation zum Entwurf von ADTs besteht darin, bestimmte Strukturen mit einem (manchmal minimalistischen) Konzept so zu beschreiben, dass man sie nutzen kann, ohne all ihre Details und ihren inneren Aufbau zu kennen. Auch hier wird wieder der Vorteil von Schnittstellen deutlich: Der Anwender kann die für ihn wichtige Funktionalität nutzen und gleichzeitig hat der Bereitsteller relativ große Freiheiten bei der konkreten Implementierung.
Auf der anderen Seite ist mit der fortschreitenden Entwicklung der Informatik eine Reihe von Datenstrukturen entwickelt worden, die sich als sehr nützlich und effizient erwiesen haben. Die abstrakte Beschreibung dieser Strukturen bildet ein reiches Reservoir an Standard-ADTs, die ein Programmierer möglichst gut kennen sollte, um seine Arbeit effektiv zu gestalten. Zu diesen ADTs gehören Stapel, Warteschlangen, Listen und Wörterbücher. Bevor wir uns damit näher beschäftigen, wollen wir einige vor allem in der Diskreten Mathematik häufig verwendete Strukturen kennenlernen und dafür geeignete ADTs diskutieren.
Die grundlegendste und gleichzeitig einfachste mathematische Struktur ist die **Menge**. Leider ist es relativ schwierig, gute Datenstrukturen für Mengen zu entwerfen. Ein wichtiger Grund dafür ist die Tatsache, dass eine Menge eine ungeordnete Struktur ist, in der Elemente nur einfach vorkommen. Ein ADT für Mengen sollte die Größe der Menge M angeben, sowie Elemente einfügen und streichen können. Bei der Implementierung ist zu beachten, dass bei der Anweisung ein bereits vorhandenes Element einzufügen oder ein nicht zur Menge gehörendes Element zu löschen, die Größe unverändert bleibt. Wir sprechen hier nur über endliche Megen, die Größe ist also eine natürliche Zahl. Darüber hinaus sollte der ADT die Abfrage beinhalten, ob ein bestimmtes Element zur Menge gehört. Wünschenswert wäre auch eine Aufzählung der Elemente, wobei das in beliebiger Reihenfolge beschehen kann.
Bei einer **geordneten Menge** spielt die Reihenfolge eine wichtige Rolle. Um neue Elemente richtig einzuordnen und die Elemente in der richtigen Reihenfolge aufzählen, muss entweder der Einfügeplatz angegeben werden oder der ADT muss über ein vorgegebenes Vergleichskriterium verfügen. Obwohl das nach zus"atzlichen Aufwand aussieht, werden Mengen oft als geordnete Mengen verwaltet.

Eine sehr verwandte Struktur ist die **Folge**, bei der die Reihenfolge eine Rolle spielt und bei der außerdem Elemente mehrfach auftreten können. Beim Einfügen und Streichen muss man auch die Stelle angeben (es ergibt sich nicht mehr aus einem Ordnungskriterium). Wichtig sind das erste und das letzte Element. Zum Durchlaufen der Folge gibt es zwei Varianten; Man kann fordern, auf eine bestimmte Position in der Folge zugreifen zu wollen, oder sich darauf beschränken, für jedes Element das Nachfolgeelement zu bekommen.
Eine weitere Familie von untereinander sehr verwandten mathematischen Strukturen sind **Relationen, Matrizen** und **Graphen**. All diese Strukturen beschreiben die Beziehungen zwischen Paaren von Elementen. Eine Relation gibt für jedes Paar (a, b) Auskunft, ob a in Relation (Beziehung) zu b steht, ein Graph sagt, ob der Knoten a mit dem Knoten b durch eine Kante verbunden ist und eine Matrix hat einen Wert in der a-ten Zeile und b-ten Spalte. Ist dieser Wert immer ein Boolescher, ist man zurück bei den Relationen. Ein ADT muss also die entsprechenden Anfragen beantworten und (eventuell) Werte auf Anweisung aktualisieren.
Die folgenden ADTs Stapel (Stack) und Warteschlange (Queue) sind nur eingeschränkt zur Verwaltung von Folgen geeignet, dennoch spielen sie gerade durch ihre Einfachheit eine sehr wichtige Rolle.

Stapel (Stacks)

Ein Stack verwaltet eine Folge von Objekten, wobei man immer ein Element anfügen oder das zuletzt angefügte Element wieder löschen kann. Beispiele für einen Stack (auf Deutsch Stapel oder Kellerspeicher) sind:

Kartenstapel: Bei einigen Patience-Spielen ist es erlaubt Karten auf einen Hilfsstapel abzulegen, wobei die Regel besagt, dass man mit einem Ass beginnen muss und dann die $2, 3, 4, \ldots$ der gleichen Farbe ablegen kann. Man darf keine Karte aus der Mitte des Stapels herausnehmen, sondern immer nur die oberste Karte abbauen.

Back-Knopf bei Internet-Browsern: Beim Eingeben einer URL oder dem Anklicken eines Links wird ein Element auf den Stack gelegt. Die jeweils oben liegende URL wird angezeigt. Durch Back wird sie wieder heruntergenommen und die davor liegende URL wird angezeigt.

Rekursive Programmierung: Die Auswertung von rekursiven Funktionen und allgemein von verschachtelten Funktionsaufrufen erfolgt auf einem Stapel (Execution-Stack). Dazu wird die ursprünglich aufgerufene Funktion in die unterste Zelle des Stapels gelegt. Die Auswertung erfolt immer in der obersten (belegten) Zelle des Stapels. Ist die Auswertung noch nicht möglich, weil eine weiterer Funtionsaufruf erfolgt, wird dieser Aufruf (oben) auf den Stapel gelegt. Nur wenn die Auswertung ohne weitere Funktionsaufrufe möglich ist, wird der entsprechende Wert nach

unten übergeben und die oberste Zelle gelöscht. Wir zeigen das am Beispiel der Fibonacci-Rekursion.

$$f(n) = \begin{cases} 0 & \text{falls } n = 0 \\ 1 & \text{falls } n = 1 \\ f(n-1) + f(n-2) & \text{sonst} \end{cases}$$

Der Aufruf $f(3)$ führt zu den folgenden Operationen auf dem Stack

				$f(1) = 1$	
	$\rightarrow$	$f(2) = f(1) + f(0)$	$\rightarrow$	$f(2) = f(1) + f(0)$	$\rightarrow$
$f(3) = f(2) + f(1)$		$f(3) = f(2) + f(1)$		$f(3) = f(2) + f(1)$	

		$f(0) = 0$			
$f(2) = 1 + f(0)$	$\rightarrow$	$f(2) = 1 + f(0)$	$\rightarrow$	$f(2) = 1 + 0 = 1$	$\rightarrow$
$f(3) = f(2) + f(1)$		$f(3) = f(2) + f(1)$		$f(3) = f(2) + f(1)$	

	$\rightarrow$	$f(1) = 1$	$\rightarrow$	
$f(3) = 1 + f(1)$		$f(3) = 1 + f(1)$		$f(3) = 1 + 1 = 2$

Ähnlich wie im letzten Fall sind die Anwendungen zur Auswertung arithmetischer Ausdrücke oder zur Überprüfung der Korrektheit von Klammerausdrücken. Bei der Auswertung von arithmetischen Ausdrücken mit einem Stack benutzt man die umgekehrte polnische Notation (RPN). Die Bezeichnung polnische Notation führt man darauf zurück, dass sie von dem polnischen Mathematiker Lukasiewicz eingefürt wurde, nachdem er beobachtet hatte, dass die Verwendung von Präfix-Operatoren (d.h. erst der Operator dann die Argumente) das Klammersetzen überflüssig macht. So bedeutet der Ausdruck $+ - 8\ 3\ 2$ in üblicher Notation $(8 - 3) + 2$, während man $8 - (3 + 2)$ durch $-8 + 3\ 2$ dartsellen müsste. Umgekehrte polnische Notation bedeutet einfach die Operatoren hinter die Operanden zu stellen, also $8\ 3 - 2+$ im ersten und $8\ 3\ 2 + -$ im zweiten Fall. Zur Auswertung wird der Ausdruck von links nach rechts gelesen. Wird ein Operand gelesen, legt man ihn auf den Stack. Wird ein Operationssymbol gelesen, so werden die beiden obersten Operanden von Stack genommen, die Operation ausgeführt, und das Ergebnis auf den Stack gelegt. Der Stapel beinhaltet also nur Zahlen und am Ende steht das Ergebnis in der untersten Zelle.

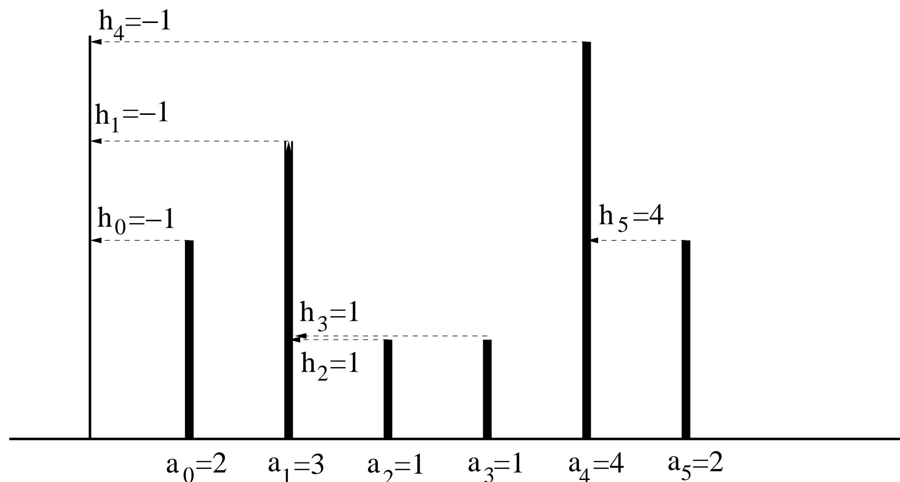
Eine weniger offensichtliche Anwendung der Stapel-Datenstruktur ergibt sich aus der folgenden Aufgabe. Gegeben ist eine Folge von Tageskursen (einer Aktie oder einer Währung) $a_0, a_1, a_2, \ldots, a_n$. Gefragt sind die Spannen s_i der Tageskurse a_i, d.h. für wieviele Tage rückwirkend, der Tageskurs a_i mindestens so gut wie an den Vortagen war. Da der Tag selbst immer mitzählen soll, gilt $s_i \geq 1$. Etwas formaler definieren wir die Menge der besseren Vogängertage $B_i = \{j | 0 \leq j < i \text{ und } a_j > a_i\}$ und definieren

$$h_i = \begin{cases} max(B_i) & fallsB_i \neq \emptyset \\ -1 & \text{sonst} \end{cases}$$

Dann kann man die Spannen durch $s_i = i - h_i$ berechnen.

Dieser Ansatz liefert auch gleich eine algorithmische Idee. Für jeden Tag i geht man soweit zurück, bis man ein $a_j > a_i$ gefunden hat und setzt $h_i = j$. Wenn es ein solches j nicht gibt, wird $h_i = -1$ gesetzt. Mit $s_i = i - h_i$ erhält man die Spannen. Die Implementierung erfolgt in zwei Schleifen: die äußere für i von 0 bis n und die innere im schlechtesten Fall für j von $i-1$ bis 0. Insgesamt ergibt sich eine quadratische Laufzeit.

Mit einer besseren Idee und der Nutzung eines Stacks kommt man auf lineare Laufzeit: Die Kurse werden (zusammen mit ihrem Index, also als Paare (a_i, i)) auf einen Stack gelegt. Die Invariante ist, dass die Kursfolge im Stack von unten nach oben streng monoton fallend sein muss. Um das zu sichern, werden vor dem Einfügen von (a_i, i) alle kleineren oder gleichen Kurse (die müssen ganz oben liegen) gelöscht. Damit ist klar, dass unter (a_i, i) nun (a_{h_i}, h_i) liegt, bzw. wenn (a_i, i) alle anderen Einträge herausgeworfen hat und ganz unten steht, ist $h_i = -1$. Man kann also beim Eintragen von (a_i, i) den Index des darunter stehenden Eintrags lesen (das ist h_i), bzw. bei leerem Stack $h_i = -1$ setzen und $s_i = i - h_i$ ausrechnen.



Nachdem klar geworden ist, dass Stapel eine äußerst nützliche Datenstruktur sind, kommen wir zur Beschreibung des ADTs Stack. Ein Stack hat die Hauptmethoden:

```
void push(Object o); // Einfuegen des Objekts o auf oberste Position
Object pop();        // Entfernen des obersten Objekts plus Rueckgabe
                     // Fehlermeldung, wenn Stack schon leer ist
```

Die folgenden Methoden stellen eine sehr nützliche Ergänzung dar:

```
int size();         // Anzahl der Objekte im Stack zurueckgeben
boolean isEmpty();  // Stack leer?
Object top();       // oberstes Objekt zur\"uckgeben, ohne es zu loeschen
```

Wir werden drei verschiedene Implementierungen dieses ADTs besprechen und ihre Vor- und Nachteile vergleichen.

1. Implementierung mit einem Array fester Größe
Man definiert eine Klasse `ArrayStack`, die das Interface `Stack` implementiert und legt sich zuerst auf eine Default-Feldgröße fest, z.B. durch:
`public staitc final int CAPACITY = 1000;`
Daneben kann man noch eine Variable `public int cpacity;` einführen, um auch mit dem Konstruktor die Größe festlegen zu können. Die wichtigsten Attribute sind

```
private Object[] S;  // haelt die Daten des Stacks
private int top;     // am Anfang -1; es gilt size = top + 1
```

Die Konstruktoren legen ein Feld der gegebenen Größe an und setzen top auf -1. Die Zugriffsmethoden müssen natürlich den Wert von `top` aktualisieren. Die Methode `top()` greift auf `S[top]` zu.
Neben ihrer Einfachheit hat diese Implementierung den Vorteil, dass jede Operation in konstanter Zeit ausgeführt wird. Der Hauptnachteil besteht darin, dass ein weiterer Fehler beim Stack-Überlauf auftreten kann. Darüber hinaus kann der Default-Wert oder eine zu groß gewälter Konstruktorparameter zur Speicherverschwendung führen.

2. Implementierung mit verkettenen Listen
Vor der Definition der Klasse `ListStack` muss man eine Klasse `ListElem` für Listenelemente definieren. Ein Listenelement besteht aus zwei Referenzen
`private Object data;` und `private ListElem next;`.
Die erste Refernz verweist auf das Objekt, das eigentlich in der Liste gehalten werden soll, die zweite auf das Vorgängerelement in der Liste. Für beide Refernzen werden get- und set-Methoden implementiert.
Ein ListStack hält nur ein Listenelement `topElem` für den obersten Eintrag im Stack, alle anderen Inhalte sind über eine Kette von next-Zeigern erreichbar, den untersten Eintrag erkennt man daran, dass next auf null zeigt. Der Konstruktor setzt `topElem = null;` und folglich muss bei `isEmpty()` nur `topElem == null` getestet werden. Die push- und pop-Methoden arbeiten wie folgt:

```
public void push(Object o){
    ListElem newtop = new ListElem();
    newtop.setData(o);
    newtop.setNext(topElem);
    topElem = newtop;
}
public Object pop(){
    Object o = topElem.getData();
    topElem = topElem.getNext(); // Fehler abfangen!
    return o;
}
```

Auch diese Implementierung realisiert jede Operation in konstanter Zeit. Ein Überlauf des Stacks ist ausgeschlossen. Einziger Nachteil ist der Speicherverbrauch, da man

pro gespeichertem Objekt ein zusätzliche Referenz für `next` benötigt. Dafür ist der Speicherverbrauch aber jederzeit den dynamischen Erfordernissen angepasst.

3. Implementierung mit dynamischen Feldern
In diesem Fall wird wie im ersten ein Array, verwendet, mit dem Unterschied, dass die Arraygröße dynamisch an den tatsächlichen Bedaarf angepasst wird. Dazu wird immer, wenn das aktuelle Array entweder zu klein oder viel zu gro ist, ein neues Array passender Größe erzeugt, die Daten kopiert und das alte Array gelöscht. Dazu gibt es drei Regeln:

- Beginne mit einem Array der Größe N, wobei N vernünftig gewählt werden sollte.
- Wenn ein Objekt gepusht werden soll, und das Array voll ist, verdoppele die Größe des Arrays.
- Wenn im Array nur noch ein Viertel der Plätze belegt sind, und das Array noch mehr als N Elemente hat, halbiere die Größe des Arrays

Im Gegensatz zu den ersten beiden Lösungen kann hier eine Push- oder Pop-Operation unter Umständen sehr viel Zeit für das Umkopieren des Array-Inhalts verbrauchen. Wir beobachten aber, dass nach einer Größenänderung des Stacks eine gewisse Zeit garantiert keine zweite Größenänderung auftreten kann. Angenommen, der Stack ist gerade auf die Größe M verändert worden. Dann ist er halb voll, hat also $\frac{M}{2}$ Elemente. Die nächste Änderung tritt ein, wenn er entweder nur noch $\frac{M}{4}$ Elemente hat, oder auf M Elemente gefüllt wurde. Dazu werden im ersten Fall mindestens $\frac{M}{4}$ Pops benötigt, im zweiten Fall $\frac{M}{2}$ Pushes. Mit einer **amortisierten Analyse** schätzt man die durchschnittliche Laufzeit für eine Operation ab: Nach jeder Größenänderung des Stack-Arrays (auf die Größe M) kann man die dazu notwendigen $\frac{M}{2}$ Referenzkopien auf mindestens $\frac{M}{4}$ nachfolgende Stackoperationen umlegen, d.h. die durchschnittlichen Kopierkosten einer Push- oder Pop-Operation sind konstant. Diese Variante ist durch ihre dynamische Größenanpassung sehr speichereffizient. Sie wurde in `java.util.Stack` implementiert.

Anmerkungen:
1) Dynamische Arrays mit der oben beschriebenen Funktionsweise werden häufig verwendet. Man nennt diese Datenstruktur Vector. Sie ist in `java.util.Vector` implementiert und spielt auch in der STL (Standard Template Library) von C++ eine wichtige Rolle.
2) Ein gemeinsamer Vorteil der drei hier beschreibenen Stack-Implementierungen liegt darin, dass man sie für beliebige Klassenobjekte verwenden kann (Stack von Personen, Punkten, Strings, ...). Dabei wird einfach das Vererbungskonzept von Java zur Anwendung gebracht: Jede Klasse ist Unterklasse von Object und damit ist auch jedes Exemplar einer Klasse auch Exemplar von Object.
Schwierigkeiten gibt es nur mit primitiven Typen wie `int` oder `double`. Hier lässt sich aber auch ein einfacher Ausweg finden, nämlich die Benutzung der Wrapper-Klassen `Integer` oder `Double`.

Warteschlangen (Queues)

Eine Warteschlange (Queue) verwaltet eine Folge von Objekten, wobei man immer Objekte (an das Ende) anfügen kann und zu jedem Zeitpunkt das Objekt löschen kann, das (von den vorhandenen) als erstes eingefügt wurde.
Stack und Queue unterscheiden sich also nach dem Auswahlprinzip für das zu löschende Element:

Stapel (Stack)	Warteschlange (Queue)
last in first out	first in first out
LIFO	**FIFO**

Typische Anwendungen für Warteschlangen (in der Informatik):

- Druckerschlange
- Tastatureingabe
- Multitask-Systeme

Bei der Beschreibung einer Queue als ADT ergeben sich die folgenden Hauptmethoden:

```
void enqueue(Object o); // Einfuegen des Objekts o an letzte Position
Object dequeue();       // Entfernen des vordertsten Objekts plus Rueckgabe
                        // Fehlermeldung, wenn Queue schon leer ist
```

sowie als Ergänzung die Methoden:

```
int size();         // Anzahl der Objekte in der Queue zurueckgeben
boolean isEmpty();  // Queue leer?
Object front();     // vorderstes Objekt zur\"uckgeben, ohne es zu loeschen
```

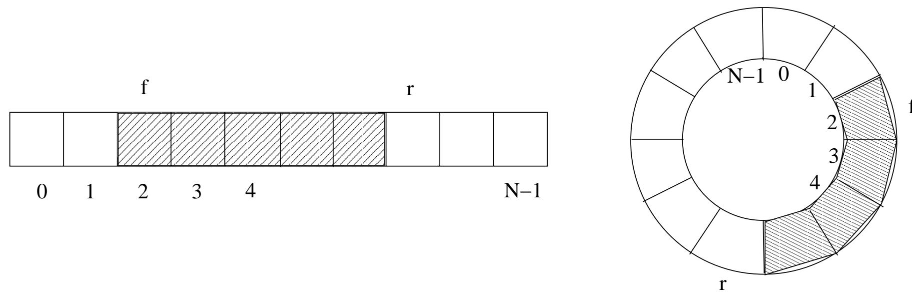
Ähnlich wie im Fall von Stacks kann man sich bei Queues prinzipiell für Implementierungen mit Arrays und mit verketteten Listen entscheiden.

1. Implementierungen von Queues mit Arrays

a) In der einfachsten Version wird ein Array Q der Default-Größe N oder einer als Parameter übergebenen Größe n angelegt. Zwei Zahlen f und r repräsentieren die erste belegte Zelle und die erste freie Zelle hinter dem belegten Teil. Der Spezialfall einer leeren Queue wird durch $f = r$ repräsentiert, d.h. die Konstruktoren setzen f und r auf 0. Zum Einfügen wird in $Q[r]$ die Referenz auf das einzufügende Objekt eingetragen und r um 1 erhöht. Zum Entfernen wird $Q[f]$ (das ist das zuerst eingefügte Element) zurückgegeben und f um 1 erhöht. Die aktuelle Größe ergibt sich einfach durch $r - f$. Bei dieser Queue-Implementierung können (im Gegensatz zur Stack-Implementierung mit Feldern fester Größe) auch dann ein Überlauf-Problem auftreten, wenn nur sehr wenige Objekte in der Warteschlange stehen. Der Grund für diesen Unterschied erklärt sich daraus, dass beim Stack freigegebnener Speicherplatz für neue Einträge genutzt werden kann (die Zahl *top* kann steigen und fallen, während für die Queue die Zahl r mit jedem neuen Eintrag steigt. Die Implementierung mit dynamischen Arrays (Vektoren)

könnte in diesem Fall den Überlauf verhindern, aber keine dynamische Anpassung des Speicherbedarfs sicher stellen.

b) Die Wiederverwendung von freigewordenen Speicherplatz wird durch sogenannte zyklische Arrays möglich. Man kann sich das als ein Feld von Daten vorstellen, das zu einem Ring geschlossen wurde. Die Idee ist in der folgenden Abbildung illustriert.



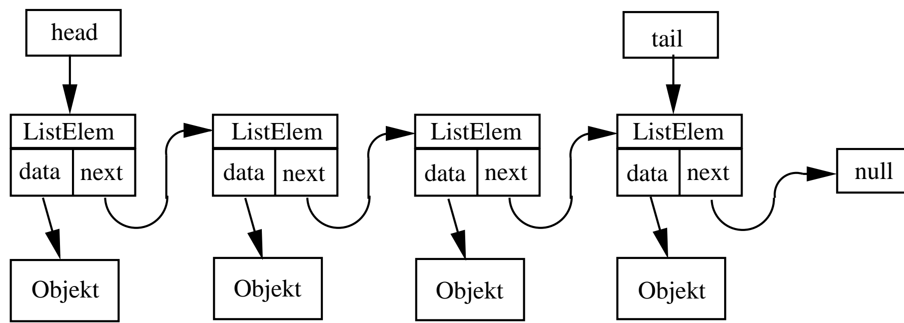
Natürlich gibt es im Rechner keinen Speicher dieser Art, man muss den Ringschluss durch Indexarithmetik modulo N simulieren. Die Zahlen f und r repräsentieren nun in Uhrzeigerrichtung die erste belegte Zelle und die erste freie Zelle hinter dem belegten Teil. Beim Einfügen muss r und beim Entfernen muss f um 1 modulo N erhöht werden. Ein Überlauffehler tritt immer dann auf, wenn in einer Einfügeoperation die Situation $f == r$ eintritt. Wenn die Situation $f == r$ durch eine Löschoperation entsteht, ist das zulässig, die Anwendung einer Löschoperation bei $f == r$ führt aber zu einem Fehler. Man kann beobachten, dass bei dieser Implementierung immer mindestens eine Zelle unbelegt bleiben muss, da man nur durch die Werte von f und r nicht zwischen einem voll belegtem und einem leeren zyklischen Array unterscheiden kann. Durch Kombination von zyklischen Arrays mit dynamischer Größenanpassung werden Überlauffehler verhindert.

2. Implementierung mit verkettenen Listen

Wie schon bei Stacks wird hier mit der Klasse `ListElem` gearbeitet. Der wesentliche Unterschied zur Stack-Implementierung besteht darin, dass eine Klasse `ListQueue` zwei Referenzen auf Listenelemente benötigt, das erste und das letzte in der verketten Liste. Wir nennen sie `head` und `tail` (zur Erinnerung, bei Stacks brauchte man nur das erste Element).

Bleibt die Frage, wie man einfügen und löschen kann. Es ist leicht zu sehen, dass man auf beiden Seiten einfügen könnte. Zuerst wird ein neues Listenelement `le` erzeugt und die Referenz `data` auf das einzufügende Objekt gesetzt. Will man `le` auf der rechten Seite (tail) einfügen, so muss `tail.next` auf `le` und `le.next` auf `null` gesetzt werden und zum Abschluss setzt man `tail=le;`. Will man `le` auf der linken Seite (head) einfügen, so muss `le.next` auf `head` und `head=le;` gesetzt werden. Dagegen ist das Löschen nur auf der Seite von `head` möglich. Deahlb wird bei einer Queue-Implementierung auf der `tail`-Seite eingefügt und auf der `head`-Seite gelöscht.

Aus dieser Beschreibung folgt, dass alle Queue-Operationen auf verketteten Listen in



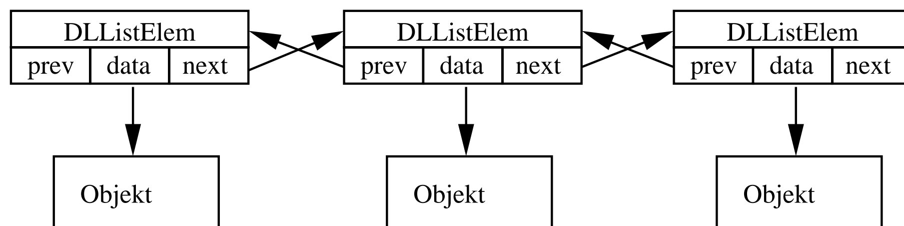
konstanter Zeit ausgeführt werden.

Double-Ended Queues

Eine Double-Ended Queue (Deque, gesprochen Deck) ist eine geordnete Datenstruktur, bei der man am Anfang und am Ende der Ordnung jeweils einfügen und löschen kann. Ein solcher ADT ist durch die folgenden Methoden gekennzeichnet:

```
void insertFirst(Object o);
void insertLast(Object o);
Object removeFirst();
Object removeLast();
Object first();    \\erstes Objekt zurueckgeben ohne zu loeschen
Object last();     \\letztes Objekt zurueckgeben ohne zu loeschen
```

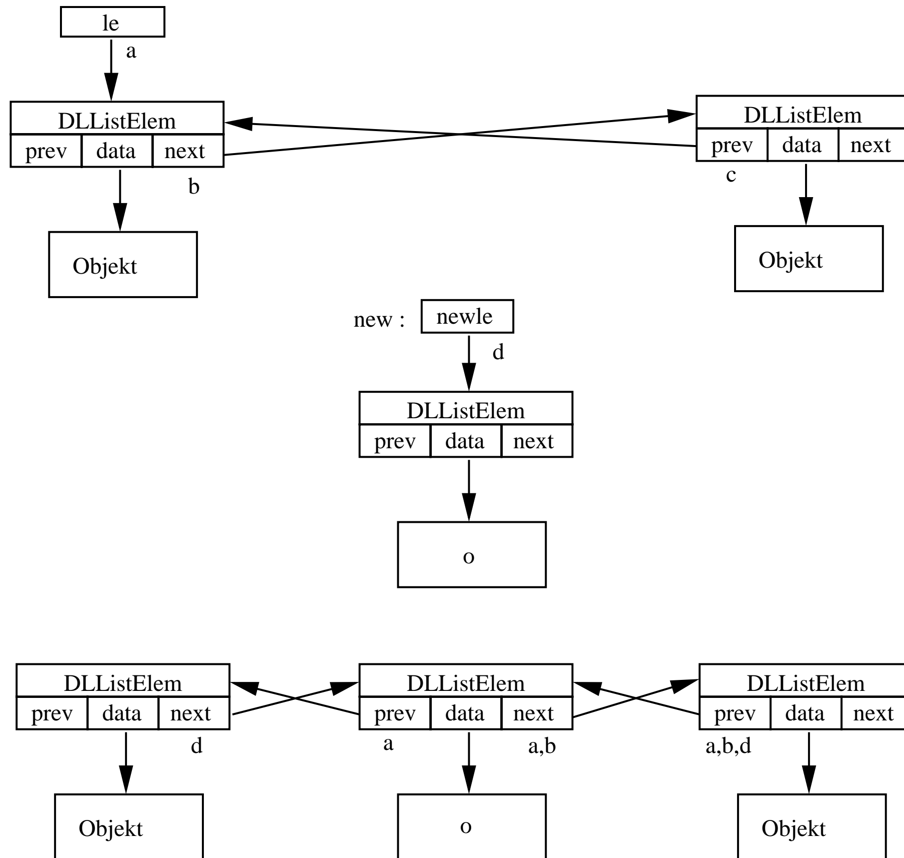
Damit ist klar, dass man mit einer Deque-Implementierung sowohl einen Stack als auch eine Queue simulieren kann. Die Implementierung kann wieder mit zyklischen, dynamischen Arrays erfolgen oder mit doppelt verketteten Listen. Die Idee zu dieser Struktur besteht darin, zu jeder `next`-Referenz von einem Listenelement auf das nächste auch die Rückreferenz `prev` zu speichern. Die folgende Abbildung illustriert den Aufbau eines Listenelements in einer doppelt verketteten Liste:



Ein Objekt von `DLListElem` hat also die Attribute

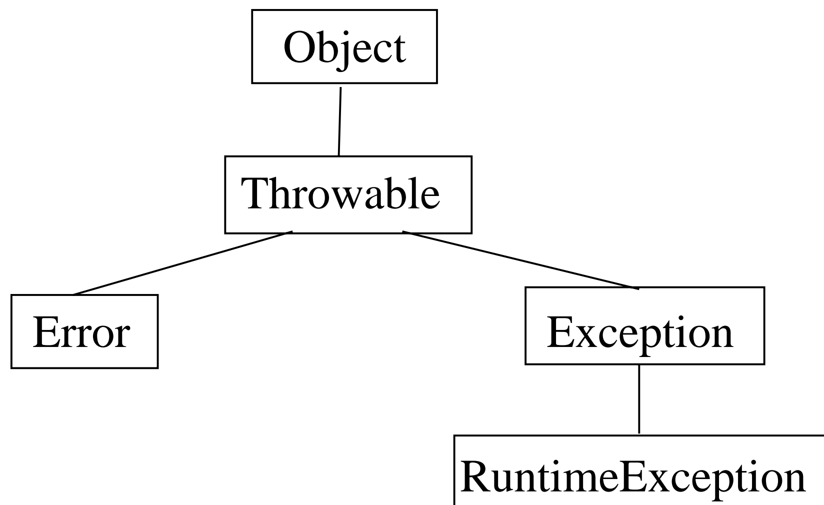
```
Object data;
DLListElem next;
DLListElem prev;
```

Die Einfüge- und Löschoperationen funktioniern, wie schon bei den Queues beschrieben, in konstanter Zeit. Darüber hinaus kann man bei doppelt verketteten Listen auch an beliebigen Stellen in konstanter Zeit Objekte einfügen und wieder löschen. Man braucht zum Einfügen nur eine Referenz auf das Listenelement, hinter dem eingefügt werden soll, und zum Löschen eine Referenz auf das zu streichende Listenelement.
Die folgende Abbildung beschreibt die Schritte zur Ausführung einer Einfügeoperation eines Objekts `o` hinter ein gegebenes `DLListElem le`. Zuerst muss ein neues `DLListElem newle` angelegt werden, dessen `data`-Referenz auf `o` gesetzt wird. Danach müssen die `prev`- und `next`-Referenzen von newle, die `prev`-Referenz des alten Nachfolgers von `le` und zuletzt die `next`-Referenz von `le` neu gesetzt werden. Die Buchstaben unter den entsprechenden Feldern zeigen an, welche von den alten Referenzen gebraucht werden, um die neue Referenz festzulegen.



Fehlerbehandlung

Bei den verschiedenen Implementierungsvarianten für Stacks und Queues wurde mehrfach auf die möglichen Fehler verwiesen. Das Auftreten eines Fehlers führt normalerweise zum Abbruch des Programms. Zum Sprachkonzept von Java gehört aber ein Mechanismus, mit dem man auftretende Fehler, genauer sollte man hier von Ausnahmesituationen sprechen, abfangen und behandeln kann. Wir werden die Funktionsweise dieser Ausnahmebehandlung am Beispiel der Queue-Implementierung mit zyklischen Arrays besprechen.
Der Mechanismus zur Ausnahmebehandlung ist (wie alle anderen wichtigen Sprachbestandteile) in das Klassenkonzept eingebettet. Wir unterscheiden zwischen Fehlern (Klasse `Error`) und Ausnahmen (Klasse `Exception`). Beide Klassen sind Erweiterungen von `Throwable` und besitzen eine Reihe von vordefinierten Unterklassen. Methoden, in denen ein Fehler oder eine Ausnahmesituation auftritt, "werfen" ein Objekt der entsprechenden Klasse. Objekte der Klasse `Error` beschreiben schwere Fehler, die zum Abbruch des Programms führen. Beispiele dafür sind `IllegalAccessError`, `NoSuchMethodError` und `OutOfMemoryError`. Dagegen können Objekte der Klasse `Exception` "abgefangen" und behandelt werden, so dass das Programm weiter ausgeführt werden kann. Vordefinierte Beispiele dafür sind `NullPointerException` und `ArrayIndexOutOfBoundException`. Der folgende Ausschnitt des Klassendiagramms illustriert die Situation:



Bei Ausnahmen unterscheidet man überprüfte (checked) und nicht überprüfte (unchecked) Exceptions. Nicht überprüft werden nur Unterklassen der Klasse `RuntimeException` (vordefiniert), alle anderen Ausnahmen werden vom Compiler überprüft. Das bedeutet insbesondere, dass in der Deklaration einer Methode `f(...))` vermerkt werden muss, ob bei der Ausführung eine bestimmte Ausnahme `MyException` geworfen werden könnte (ob das wirklich passiert hängt in der Regel von den übergebenen Parametern ab). Eine solche Deklaration könnte z.B. so aussehen:

```
public void f(int i) throws MyException
```

Da eine nicht abgefangene Exception an die aufrufende Methode weitergericht wird, muss auch jede Methode, die `f()` aufruft, `MyException` abfangen (Details später) oder `throws MyException` in der Deklaration verwenden. Nicht überprüfte Ausnahmen, also RuntimeExceptions werden nicht deklariert. Eine Ausnahme, die bis in die Methode `main` gereicht und von dort geworfen wird, führt zum Programmabbruch.
Zum Abfangen von Ausnahmen verwendet man die `try/catch/finally`-Anweisung. Dabei wird mit `try` ein Block von Anweisungen gebildet, der durch eine eventuell geworfene Exception vorzeitig verlassen wird. An den `try`-Block können sich ein oder mehrere `catch`-Blöcke anschließen, wobei jeder eine Exception-Klasse abfangen kann. Die abzufangende Exception-Objekt wird als Parameter übergeben. Im `catch`-Block wird beschrieben, wie man sich verhalten will, z.B. Ausgabe einer Warnung und/oder Festlegung eines Default-Rückgabewerts. Die Anweisung kann (optional) durch einen `finally`-Block abgeschlossen werden, der in jedem Fall ausgefürt wird, egal ob eine Exception abgefangen oder nicht abgefangen wurde oder gar keine Exception geworfen wurde. Diese Blöcke werden oft zum Schließen von geöffneten Dateien verwendet.
Bei selbstdefinierten Ausnahmeklassen muss man sich entscheiden, ob alle Exceptions dieser Art in der gleichen Weise behandelt werden sollen (in diesem Fall genügt ein Default-Konstruktor) oder ob man konkrete Informationen über die Umstände benötigt, unter denen die Exception auftrat. Diese Informationen müssen dem Konstruktor als Parameterübergeben und duch Attribute der Klasse festgehalten werden.
In unserem Beispiel geht es um die Ausnahmesituation `FullQueueException`, in der ein zyklisches Array vollgeschrieben wird (Eine Zelle müsste frei bleiben). Da unsere Fehlerbehandlung nur darin bestehen wird, den Eintrag zu verweigern und eine Warnung auszugeben, können wir uns auf den Default-Konstruktor beschränken:

```
public class FullQueueException extends Exception{
        public FullQueueException(){super();}
}
```

Folgendes Interface muss implementiert werden:

```
public interface Queue{
    public void     enqueue(Object o);  //Einfuegen
    public Object   dequeue();          //Rueckgabe und Loeschen
    public int      size();
    public boolean  isEmpty();
    public Object   front();            //Rueckgabe ohne Loeschen
}
```

Die einfachste Queue-Implementierung mit einem (nichtzyklischen) Array hat die folgende Gestalt:

```
public class ArrayQueue implements Queue{
    public int N;        //frei waehlbare  Groesse des Arrays
    private Object[] Q;  //Array fuer Queue-Inhalt
```

```
    private int f;       //erste belegte Zelle
    private int r;       //erste freie Zelle hinter belegtem Teil
    public ArrayQueue(int n){
        N = n;  f=0;  r=0;
        Q = new Object[n];
   } //Aus Platzgruenden kein Default-Konstruktor
// Methoden ohne  Fehlerbehandlung bei Ueberlauf bzw. bei leerer Queue:
    public void     enqueue(Object o){ Q[r++] = o;}
    public Object   dequeue(){return Q[f++];}
    public int      size(){return( N+r-f );}
    public boolean  isEmpty() {return( r==f );} ;
    public Object   front(){return Q[f];}
}
```

Diese Implementierung vernachlässigt die Fehlererkennung und Behandlung, aber bei Verwendung der folgenden Test-Klasse sieht man, dass durch den Versuch außerhalb der Arraygrenzen zu schreiben bereits eine `java.lang.ArrayIndexOutOfBoundsException` geworfen wird, die zum Programmabbruch führt.

```
public class Test{
      public static void main(String[] args){
          ArrayQueue q = new ArrayQueue(4);
          String s = "erster";
          String t = "zweiter";
          String u = "spaeterer";
          q.enqueue(s);
          q.enqueue(t);
          System.out.println(q.dequeue());
          for(int i = 0; i < 8; i++){q.enqueue(u);}
      }
}
```

Bei der Implementierung mit zyklischen Arrays sollte man unbedingt eine Fehlerbehandlung vornehmen, denn anderenfalls werden Inhalte der Queue ohne Warnung überschrieben. Das folgende Beispiel, in dem die Array-Methoden nur dadurch modifiziert werden, dass man alle Indexoperationen modulo N ausgeführt, demonstriert diesen Effekt:

```
// Methoden ohne  Fehlerbehandlung bei Ueberlauf bzw. bei leerer Queue:
    public void     enqueue(Object o){
        Q[r] = o;
        r= (r+1)%N;
    }
    public Object   dequeue(){
        Object o= Q[f];
        f= (f+1)%N;
```

```
        return(o);
    }
    public int      size(){return((r-f)%N);}
    public boolean  isEmpty() {return( r==f );} ;
    public Object   front(){return Q[f];}
```

Nach Austausch der Methoden und Aufruf der Testklasse, erfolgt weder eine Fehlermeldung noch ein Programmabbruch, aber der String `"zweiter"`, der eigentlich an zweiter Stelle ausgegeben werden müsste, wurde zum Ausgabezeitpunkt schon überschrieben. Dieser Fehler in der Methode `enqueue` wird mit der folgenden Variante geworfen. Man beachte, dass der Zusatz `throws FullQueueException` auch im Interface einzufügen ist.

```
    public void enqueue(Object o) throws FullQueueException{
        if( (r+1-f)%N == 0) throw new FullQueueException();
        Q[r] = o;
        r= (r+1)%N;
    }
```

Diese Fehler können dann in der `main`-Methode abgefangen werden, indem man jeden Aufruf von `enqueue` in einen `try`-Block setzt.

```
   try{q.enqueue(s);}
   catch(FullQueueException e){
         System.out.println("Queue voll, Eintrag nicht realisiert");}
    }
```

Damit hat man nicht die beabsichtigte Funktionalität gerettet, denn der geforderte Eintrag in die Queue wurde einfach ignoriert, aber das Überschreiben von Objekten, die bereits in der Queue sind, wird verhindert und die Ausnahme wird für den Anwender sichbar gemacht.
Hinweis: Bei der hier beschriebenen Fehlerbehandlung ging es vor allem darum, die notwendigen Schritte an einem einfachen Beispiel zum Vorlesungsstoff zu demonstrieren, und weniger um die Beschreibung einer optimalen Lösung. Diese erhält man mit dynamischen, zyklischen Arrays.

Bäume

Motivation: Der Sequence-ADT

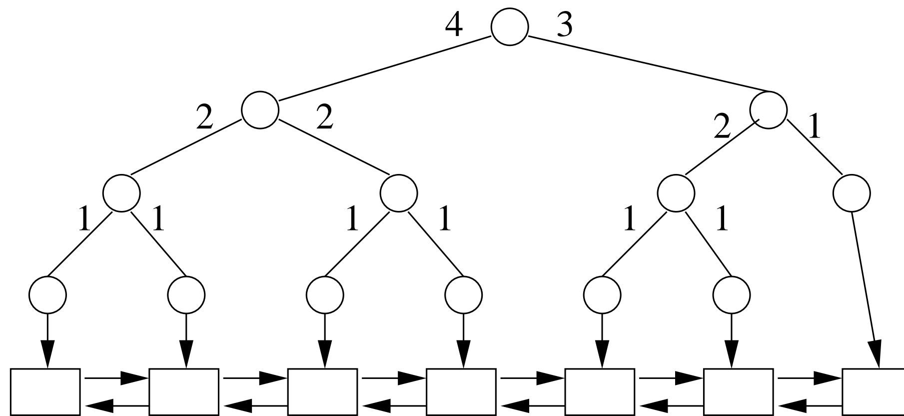
Der Begriff **Sequence** (**Folge**) bezeichnet einen ADT, der eine Verallgmeinerung von Stack, Queue und Deque ist. Auch hier werden Daten in einer linearen Ordnung gehalten, aber der Zugriff auf die Daten kann an beliebger Stelle erfolgen. Dabei unterscheidet man zwischen der Position und dem Rang eines Elements. Unter der Position des Elements verstehen wir eine Referenz auf das Element. Dagegen ist der Rang die nummerierte Stelle des Elements in der Folge. Neben dem Zugriff auf ein Element über seine Position oder über seinen Rang sind Einfügeoperationen vor oder hinter einer gegebenen Position bzw. auf einem bestimmten Rang und Löschoperationen an einer Position oder auf einem bestimmten Rang gefordert.
Wie bereits bei Deques besprochen, kann dieser Datentyp durch dynamische Arrays oder durch doppelt verkettete Listen implementiert werden. Die Vor- und Nachteile dieser Implementierungen werden in der folgenden Tabelle deutlich. Die Größe der Folge wird mit n bezeichnet.

Operation	Laufzeit bei Array-Implementierung	Laufzeit bei Implementierung mit doppelt verk. Listen
Zugriff auf Position	$\Theta(1)$	$\Theta(1)$
Zugriff auf Rang	$\Theta(1)$	$\Theta(n)$
Einfügen mit Position	$\Theta(n)$	$\Theta(1)$
Einfügen mit Rang	$\Theta(n)$	$\Theta(n)$
Löschen mit Position	$\Theta(n)$	$\Theta(1)$
Löschen mit Rang	$\Theta(n)$	$\Theta(n)$

Der einzige wesentliche Vorteil der Array-Implementierung liegt also im Zugriff auf das Element mit Rang k, für den man bei der Listenimplementierung k next-Referenzen vom Anfangselement verfolgen muss. Beim Einfügen und Löschen mit Rang hat die Array-Implementierung zwar auch den Vorteil, das man unmittelbar auf die Stelle zugreifen kann, aber dafür mus der gesamte Array-Inhalt hinter dieser Stelle um eins nach rechts (Einfügen) oder um eins nach links (Löschen) verschoben werden. Keine dieser Implementierungen ist also wirlich befriedigend.

Die Idee für eine effizientere Implementierung gründet sich zuerst auf die Beobachtung, dass sich alle Nachteile der Listenimplementierung auf einen Punkt fokussieren lassen: Das Finden der Position (Referenz) bei gegebenen Rang. Ist das geschehen, kann auch in konstanter Zeit eingefügt und gelöscht werden. Um den Zusammenhang zwischen Rang und Position algorithmisch schneller bearbeiten zu können, wird ein binärer Baum über die Liste gelegt, so dass jedes Blatt eine Referenz auf ein darunter liegendes Listenelement hält. Wenn man zusätzlich in jedem inneren Knoten vermerkt, wieviele Blätter der linke und der rechte Teilbaum enthalten, kann man leicht einen Algorithmus entwerfen, der bei gegebenem k von der Wurzel beginnend das k-te Blatt erreicht. Die Zeit ist proportional zur Tiefe des Baums, bei einem balancierten Baum $O(\log_2 n)$. Beim

Einfügen und Löschen muss natürlich auch der Baum aktualisiert werden, aber das geht wieder in logarithmischer Zeit.



Das Hauptproblem besteht darin, dass bei einer ungünstigen Folge von Einfügeoperationen der Baum die Balanceeigenschaft verlieren und damit die Zeit auf $\Omega(n)$ anwachsen kann. Es gibt verschiedene Ansätze zur Lösung dieses Problems mit denen wir uns in den nächsten Vorlesungen beschäftigen werden.

Bäume mit Wurzeln (rooted Trees)

Definition: Ein **(gewurzelter) Baum** besteht aus einer Menge T von **Knoten**, die wie folgt strukturiert ist:

1. Es gibt genau einen hervorgehobenen Knoten $r \in T$, die **Wurzel** des Baums
2. Jeder Knoten außer r hat genau einen **Vaterknoten** (pc: **Elternknoten**)
3. Die Wurzel r ist **Vorfahre** jedes Knotens.

Dabei wird der Begriff $v \in T$ ist **Vorfahre** $u \in T$ rekursiv definiert, durch
i) $u = v$ oder
ii) v ist Vorfahre des Vaterknotens von u

Weiter Begriffe, die mit dieser Definition in Zusammenhang stehen, sind:
$u \in T$ ist **Kind** von $v \in T$ genau dann, wenn v Vaterknoten von u ist.
$u \in T$ ist **Nachfahre** von $v \in T$ genau dann, wenn v Vorfahre von u ist.
$u \in T$ und $v \in T$ sind **Geschwister** genau dann, wenn sie den selben Vaterknoten haben.
Knoten ohne Kinder heißen **Blätter** oder **äußere Knoten**.
Knoten mit (mindestens) einem Kind heißen **innere Knoten**.
Wenn wir in diesem Teil der Vorlesung von einem Baum sprechen, meinen wir immer einen gewurzelten Baum. Wir werden später auch ungewurzelte Bäume kennenlernen.

Definition: Ein Baum ist ein **geordneter Baum**, wenn für jeden Knoten die Menge seiner Kinder geordenet ist (erstes Kind, zweites Kind,...).

Der Baum-ADT

Die Beschreibung eines ADTs für gewurzelte Bäume weist große Ähnlichkeit zum Listen-ADT auf. Auch hier konzentriert sich die Beschreibung auf die Knoten. Zur Reprŝentation des Baums reicht dann ein einzelner Knoten, nämlich die Wurzel aus. Ein Baumknoten muss die Referenzen auf den Elternknoten und die Kinder sowie auf ein Objekt (falls der Knoten Daten speichern soll) halten. Der Typ `TreeNode` wird durch die folgenden Methoden beschrieben:

```
TreeNode  parent();    //der Elternkonten
Object  element();     //die Daten
TreeNode[] children(); //Liste der Kinder
boolean isRoot();
boolean isLeaf();
void  setElem(Object e);
void  setParent(TreeNode v);
void  addChild(TreeNode v);   //dazu muss v setParent(this) ausf\"uhren
void  addSubtree(TreeNode v); //alternative Variante
```

Definition: Ist T ein Baum und $T' \subseteq T$ eine Untermenge der Knotenmenge, so nennen wir T' einen Unterbaum von T, wenn T' selbst ein Baum ist und für jeden Knoten $v \in T'$ auch alle Kinder von v zu T' gehören.

Wie man leicht sieht gibt es eine Bijektion (d.h. eine umkehrbare Abbildung) zwischen der Menge der Knoten und der Menge der Unterbäume von T. Einerseits kann man jedem Knoten v, den Unterbaum aller Nachfolger von v (inklusive v als Wurzel) zuordnen. Für die Umkehrabbildung ordnen wir jedem Unterbaum seine Wurzel zu.
Definition: Sei T ein Baum und $v \in T$ ein Knoten. Der Abstand von v zur Wurzel nennen wir die Tiefe von v und den Abstand von v zu seinem weitesten Nachfahren die Höhe von v. Die Höhe der Wurzel definiert die Höhe und gleichzeitig die Tiefe des Baums.
Beide Definitionen kann man rekursiv formulieren und auch entsprechend implementieren:

$$\text{Tiefe}(v) = \begin{cases} 0 & \text{falls } v \text{ die Wurzel ist} \\ 1 + \text{Tiefe}(\text{Vater von } v) & \text{sonst} \end{cases}$$

$$\text{Höhe}(v) = \begin{cases} 0 & \text{falls } v \text{ ein Blatt ist} \\ 1 + max\{\text{Höhe}(u) | u \text{ ist Kind von } v\} & \text{sonst} \end{cases}$$

```
int depth(){   // als Methode von TreeNode
          int d=0;
          if(! isRoot()) d = 1 + parent().depth();
          return d;
}
```

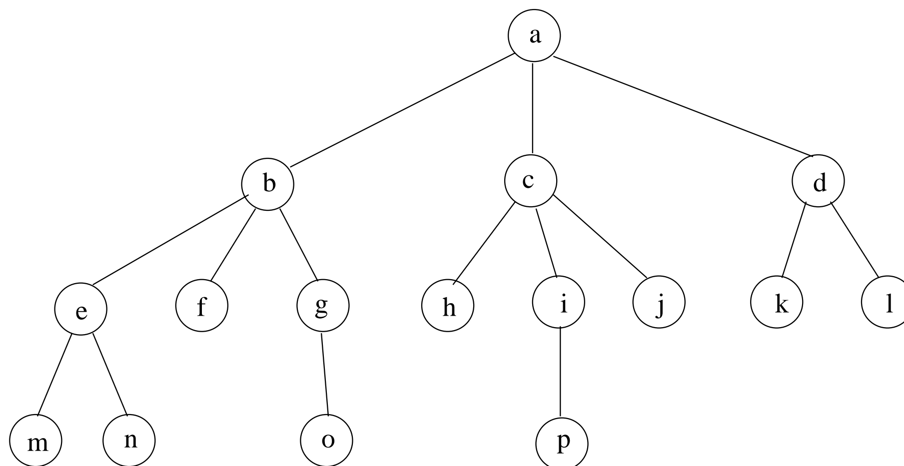
```
int height(){   // als Methode von TreeNode
          int h=0;
          if(! isLeaf()){
                 for(int i=0; i< children().length; i++)
                          if(h < (children()[i]).height()){
                              h= (children()[i]).height();
                          }
          }
          return h;
}
```

Die Laufzeit der Tiefenberechnung ist proportional zur Tiefe, die Laufzeit der Höhenberechnung ist proportional zur Größe des Unterbaums.

Ähnlich wie bei verketteten Listen wird ein Baum nur durch seine Wurzel repräsentiert. Die vollständige Struktur ergibt sich erst durch die Verkettung von Referenzen. Da diese Struktur komplexer als bei verketteten Listen ist, spielen Methoden zum Durchlaufen von Bäumen (tree traversals) eine wichtige Rolle. Zwei dieser Durchlaufmethoden kann man für beliebige Bäume anwenden: **Preorder-** und **Postorder-Traversierungen**. Sie laufen nach den folgenden Regeln ab:

- **Preorder:** Für jeden zu durchlaufenden Teilbaum besuche zuerst die Wurzel und durchlaufe dann nacheinander die Teilbäume aller Kinder.
- **Postorder:** Für jeden zu durchlaufenden Teilbaum durchlaufe nacheinander die Teilbäume aller Kinder und zum Schluss die Wurzel.

Für den abgebildeten Baum ergibt sich beim Preorder-Durchlauf die Knotenfolge
$a, b, e, m, n, f, g, c, o, h, i, p, j, d, k, l$
und beim Preorder-Durchlauf die Knotenfolge
$m, n, e.f.o, g, b, h, p, i, j, c, k, l, d, a.$



Binäre Bäume

Die Begriffsdefinition ist in der Literatur leider nicht ganz eindeutig. Die allgemeinere Definition (wie auch in Informatik A verwendet) nennt einen Baum binär, wenn jeder Knoten höchstens zwei Kinder besitzt und jedes Kind als rechtes bzw. linkes Kind gekennzeichnet ist. Im Spezialfall, wenn jeder innere Knoten zwei Kinder hat, spricht man dann von einem echten, vollen oder vollständigen binären Baum. Wir werden uns in diesem Semester nur mit dem Spezialfall beschäftigen und den Begriff des vollständigen binären Baums noch einmal neu (anders als in Informatik A) definieren.

Definition: Ein **binärer Baum** ist ein geordneter Baum, bei dem jeder innere Knoten genau zwei Kinder hat. Das erste Kind nennt man auch **linkes Kind** und das zweite **rechtes Kind**.

Mit den folgenden Sätzen werden die wichtigsten Fakten über die Anzahl von Blättern und inneren Knoten in einem binären Baum zusammengestellt. Man beachte, dass sich diese Aussagen auf die hier verwendete Definition beziehen und bei der Verwendung der allgemeinen Definition zum Teil falsch sind. Wir werden im Folgenden mit $i(T)$ und $b(T)$ die Anzahlen der inneren Knoten und Blätter des Baums T bezeichnen. Für die Höhe eines Knotens v bzw. des Baums T werden wir die Bezeichnungen $h(v)$ bzw. $h(T)$ verwenden.

Satz 1: Für jeden binären Baum T gilt $b(T) = i(T) + 1$.
Der Beweis erfolgt mit verallgemeinerter vollständiger Induktion nach $h(T)$. Der Induktionsanfang für $h(T) = 0$ ist einfach, denn in diesem Fall ist die Wurzel von T ein Blatt und folglich $b(T) = 1 = 0 + 1 = i(T) + 1$.
Sei der Satz für alle binären Bäume der Höhe $\leq n$ bewiesen und sei T ein Baum der Höhe $n + 1$. Dann liegen unter der Wurzel r von T zwei Teilbäume T_l und T_r deren Höhe $\leq n$ ist. Folglich kann man für beide die Induktionsvorausetzung anwenden. Offensichtlich gilt auch $b(T) = b(T_1) + b(T_2)$ sowie $i(T) = i(T_1) + i(T_2) + 1$, da die Wurzel ein zusätzlicher innerer Knoten von T ist. Daraus kann die Induktionsbehauptung wie folgt abgeleitet werden:

$$b(T) = b(T_1) + b(T_2) = (i(T_1) + 1) + (i(T_2) + 1) = (i(T_1) + i(T_2) + 1) + 1 = i(T) + 1.$$

Satz 2: Sei T ein binären Baum mit der Höhe h mit n Knoten. Dann gelten die folgenden vier Bedingungen:

1) $h + 1 \leq b(T) \leq 2^h$

2) $h \leq i(T) \leq 2^h - 1$

3) $2h + 1 \leq n \leq 2^{h+1} - 1$

4) $\log_2 (n + 1) - 1 \leq h \leq \frac{n-1}{2}$

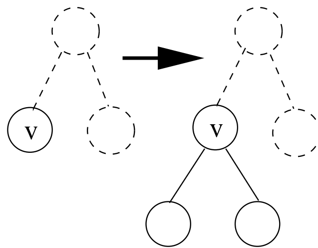
Beweisidee: Die zwei Ungleichungen in der ersten Aussage beweist man wie oben mit verallgemeinerter vollständiger Induktion. Die zweite Aussage folgt unmittelbar aus der ersten durch Anwendung von Satz 1. Die dritte Aussage ist nur die Zusammenfassung der ersten beiden Aussagen, denn $n = b(T) + i(T)$. Die vierte Aussage folgt aus der dritten durch einfache Umformungen, wobei die linke Ungleichung aus 3) die rechte Ungleichung aus 4) impliziert, während aus der rechten Ungleichung aus 3) die linke Ungleichung aus 4) folgt:

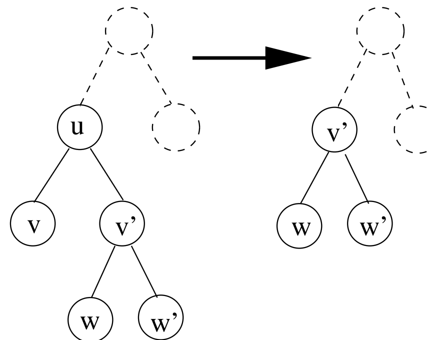
$2h + 1 \leq n \Longleftrightarrow 2h \leq n - 1 \Longleftrightarrow h \leq \frac{n-1}{2}$

$n \leq 2^{h+1} - 1 \Longleftrightarrow n + 1 \leq 2^{h+1} \Longleftrightarrow \log_2(n+1) \leq h + 1 \Longleftrightarrow \log_2(n+1) - 1 \leq h$

Man teilt die Knoten eines binären Baums oft bezüglich ihrer Tiefe in Niveaus (Level) ein. Level 0 enthält nur die Wurzel, Level 1 die Kinder der Wurzel und allgemein Level k alle Knoten der Tiefe k.

Das Interface für Knoten von binären Bäumen kann man nicht als einfache Erweiterung von `TreeNode` beschreiben, denn insbesondere beim Einfügen und Löschen von Knoten muss beachtet werden, dass man weiterhin einen binäre Baum (in Sinne unserer Definition) behält.
1) Es ist günstig, die Methode `children()` durch zwei Methoden `leftChild()` und `rightChild()` zur Rückgabe des linken und rechten Kindes zu ersetzen.
2) Die neue Einfügemethode `expandExternal()` setzt voraus, dass der aktuelle Knoten this ein Blatt ist und wandelt ihn durch Anfügen von zwei neuen Blättern in einen inneren Knoten um.
3) Die neue Streichmethode `removeAboveExternal` kann man auch nur für Blätter v verwenden. Um weiterhin einen binären Baum zu erhalten wird mit dem Blatt v auch der Elternknoten u von v gestrichen und an die Stelle von u rückt der Geschwisterknoten v' von v. Diese Operationen sind in der folgenden Abbildung demonstriert:





Die Preorder- und und Postorder-Traversierungen kann man für binäre Bäume durch eine dritte Durchlaufmethode ergänzen, die **Inorder-Traversierungen**. Bei dieser Methode wird immer zuerst der linke Teilbaum, dann die Wurzel und dann der rechte

Teilbaum besucht. Wie schon bei den anderen Traversierungen ergibt sich eine sehr einfache Implementierung, hier der Pseudocode mit v als Parameter:

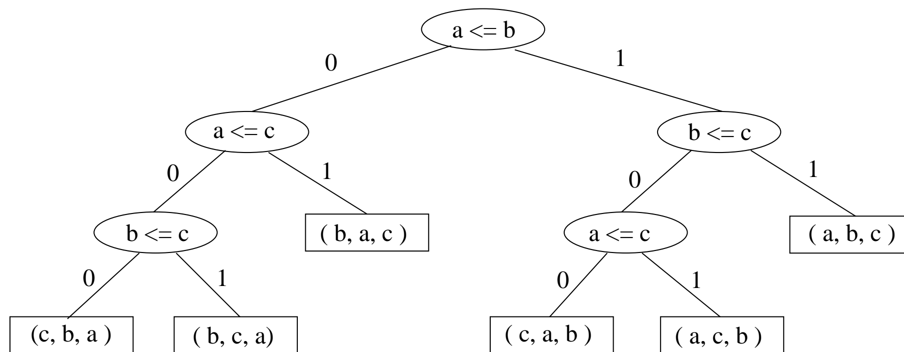
```
inorder(v)
    if(! v.isLeaf()) inorder(v.leftChild());
    visit(v);
    if(! v.isLeaf()) inorder(v.rightChild());
```

Preorder-Traversierungen sind besonders geeignet, um Methoden zur Evaluierung der Knoten eines Baums zu implementieren, bei denen die Werte der Kinderknoten von Werten der Elternknoten abhängen. Das Stadardbeispiel dafür ist die Bewertung der Tiefe aller Knoten eines Baums. Im Gegensatz dazu sind Postorder-Traversierungen besonders für Methoden geeignet, bei denen der Wert des Elternknotens von Werten der Kinderknoten abhängt. Stadardbeispiele dafür sind die Bewertung der Höhe aller Knoten eines Baums oder die Bewertung aller Knoten mit der Größe des von ihnen aufgespannten Unterbaums. Alle genannten Methoden haben lineare Laufzeit.
Die wichtigsten Anwendungen von Inorder-Traversierungen sind mit dem Begriff des binären Suchbaums verbunden:
Ein **binärer Suchbaum** speichert in seinen Knoten Zahlen (oder allgemeiner Objekte aus einer Klasse mit Vergleichsoperation) und hat die Eigenschaft, dass für jeden inneren Knoten v alle Zahlen aus dem linken Unterbaum von v kleiner oder gleich der Zahl in v sind und diese wieder kleiner oder gleich allen Zahlen im rechten Unterbaum von v ist. Für einen binären Suchbaum gibt der Inorder-Durchlauf die geordnete Folge der Zahlen wieder.

Entscheidungsbäume
Entscheidungsbäume dienen zur Klassifizierung von Objekten, wobei die zu unterscheideden Objektklassen (diese können auch aus einem einzelnen Objekt bestehen) in den Blättern liegen. In den inneren Knoten v werden Fragen über das zu klassifizierende Objekt gestellt und in Abhängigkeit von der Antwort wird man zu einem bestimmten Kind von v weitergeleitet. Wenn alle Fragen nur zwei mögliche Antworten 0 oder 1 haben, sprechen wir von einem binären Entscheidungsbaum.
In versteckter Form trifft man Entscheidungsbäume sehr häufig an:

- Unterhaltungsspiele in denen Begriffe oder Personen erraten werden müssen, wobei man nur mit ja oder nein zu beantwortende Fragen stellen darf;
- Bestimmung von Mineralien aus ihren physikalischen Eigenschaften (nicht binär);
- Checkliste zur Fehlerbestimmung bei technischen Geräten;

In der Informatik werden binäre Entscheidungsbäume zur Beschreibung von booleschen Funktionen verwendet (Komplexitätstheorie). Wir beschäftigen uns mit einer anderen Anwendung, der Herleitung von unteren Schranken für vergleichbasierte Sortieralgorithmen. Zum besseren Verständnis der Vorgehensweise beginnen wir mit einem Entscheidungbaum zum Sortieren einer 3-elementigen Folge a, b, c:



Wir betrachten nun einen beliebigen vergleichsbasierten Sortieralgorithmus $\mathcal{A}$. Die Laufzeit $T(n)$ ist definiert als maximale Anzahl der Elementarschritte, die der Algorithmus bei einer Eingabefolge von n Zahlen ausführt. Da das Ziel darin besteht, eine untere Schranke herzuleiten, reicht es aus, die Anzahl der Vergleichsoperationen im schlechtesten Fall abzuschätzen. Es zeigt sich, dass man die Eingabemenge sogar auf alle Folgen $(a_1, a_2, \ldots, a_n)$ einschränken kann, die eine Permutation von $(1, 2, \ldots, n)$ sind. Eine Permutation ist eine umkehrbare Abbildung $\pi : \{1, 2, \ldots, n\} \longrightarrow \{1, 2, \ldots, n\}$ und sie repräsentiert die Eingabe:

$$(a_1 = \pi(1), a_2 = \pi(2), \ldots, a_n = \pi(n)).$$

Da der Algorithmus die sortierte Folge ausgeben soll, muss er in jedem Fall die Umkehrpermutation ausrechnen. Die Ausgabe ist in jedem Fall die sortierte Folge $(1, 2, \ldots, n)$, aber an dieser Stelle ist es wichtig zu verstehen, dass der Algoritmus nicht "wissen" kann, dass er nur mit Testeingaben gefüttert wird, die alle zur gleichen Ausgabe führen. Er muss durch eine Folge von Vergleichen entscheiden, in welcher Reihenfolge die Zahlen aus der jeweiligen Eingabe umzuordnen sind. Mit anderen Worten muss der Algorithmus $\mathcal{A}$ zu jeder Eingabepermutation π die Umkehrpermutation π^{-1} bestimmen, in diesem Sinne gibt es $n!$ verschiedene Ergebnisse.
Wir simulieren die Arbeit von $\mathcal{A}$ durch einen binären Entscheidungsbaum. Wie im Beispiel erzeugt jede Vergleichsoperation (if-Anweisung) eine Verzweigung der Berechnung und in den Blättern befindet sich die jeweils gesuchte Umkehrpermutation. Das tiefste Blatt repräsentiert den schlechtesten Fall einer Eingabe der Länge n. Diese maximale Tiefe (und gleichzeig die Höhe h des Baums) gibt die Anzahl der Vergleichsoperationen bei dieser Eingabe an.
Der Baum hat $n!$ Blätter und aus Satz 2 folgt $n! \leq 2^h$. Das ist äquivalent zu $h \geq \log_2 n!$. Diese Ungleichung kann man von links durch $T(n) \geq h$ ergänzen und von rechts durch $\log_2 n! \in \Theta(n \log_2 n)$. Damit ist die folgende Aussage bewiesen.

Satz: Jeder vergleichsbasierte Sortieralgorithmus hat eine (worst case) Laufzeit von $\Omega(n \log_2 n)$.

Heap-Sort und andere Sortieralgorithmen

Im Zentrum diese Abschnitts steht eine neue Datenstruktur, die Halde (Heap), mit deren Hilfe man einen optimalen Sortieralgorithmus implementieren kann. Zur besseren Einordnung dieses Verfahrens beginnen wir mit einer kurzen Übersicht zu Sortieralgorithmen. Die allgemeine Problemstellung kann man wie folgt beschreiben:
Eingabe: Eine Folge $A = (a_1, a_2, \ldots a_n)$ der Länge n
Ausgabe: Die sortierte Folge B
Ob die Eingabe in Form eines Arrays oder einer verketteten Liste gegben wird, ist eine zweitrangige Frage, denn man kann jede dieser Strukturen in linearer Zeit in die jeweils andere umwandeln. Folglich hat man für jeden Algorithmus die Wahl, ob er mit Arrays oder mit verketteten Listen implementiert werden soll.
Die Elemente der Folge sind in der Regel Zahlen, die Algorithmen sollten aber auch auf Folgen von Objekten anwendbar sein, für die eine totale Ordnungsrelation definiert ist (eine formale Beschreibung folgt am Ende des Abschnitts). Elemente dürfen in der Eingabefolge mehrfach auftreten, sie müssen in der sortierten Folge auch in der entsprechenden Vielfachheit erscheinen. Die Laufzeit $T(n)$ wird in Abhängigkeit von der Länge der Eingabefolge analysiert und bezieht sich immer auf den schlechtesten Fall.

Insertion-Sort
Man implementiert eine Methode `insert(x)`, die das Element x in eine bereits sortierte Folge an der richtigen Stelle einfügt. Beginnend mit einer leeren Liste B werden die Elemente a_i aus A nacheinander (in der gegebnenen Reihenfolge) mit $B.insert(a_i)$ in B eingefügt. Da die einzelnen Einfügeoperationen nur $O(i)$ Zeit erfordern (egal, ob Array- oder Listenimplementierung), ergibt sich insgesamt die quadratische Laufzeit $T(n) \in O(n^2)$. Für eine bereits sortierte Eingabefolge A, werden auch $0+1+2+\ldots+n-1$ Vergleichoperationen ausgeführt und folglich ist $T(n) \in \Theta(n^2)$.

Selection-Sort
In der Standardimplementierung sehr ähnlich zum Insertion-Sort, mit dem Unterschied, dass der aufwendige Teil in der Auswahl und Löschung des kleinsten Elements aus A (bzw. aus der Restliste von A) besteht und dafür das Einfügen in B in konstanter Zeit erfolgt (einfach anhängen). Auch hier ist $T(n) \in \Theta(n^2)$.

Quick-Sort
Diese Verfahren ist rekursiv definiert. Ein Rekursionschritt besteht darin, die Liste A bezüglich ihres ersten Elements a_1 in die Teillisten $A_l = (a_j | j > 1 \text{ und } a_j < a_1)$ und $A_r = (a_j | j > 1 \text{ und } a_j \geq a_1)$ zu zerlegen und die (rekursiv) sortierten Teillisten durch $B_l + a_1 + B_r$ zusammenzufügen. Die Ausführung eines Rekursionsschritts für eine Liste mit k Elementen erfordert $k - 1$ Vergleiche zur Listenaufteilung und $O(k)$ Elementarschritte sind auch immer ausreichend (Konstante abhängig von Array- oder Listenimplementierung). Da man die Rekursionstiefe durch n beschränken kann, ergibt sich $T(n) \in O(n^2)$. Schlechteste Fälle sind sortierte und umgekehrt sortierte Listen, die auch quadratische Laufzeit erzwingen: $T(n) \in \Theta(n^2)$.

Man kann aber zeigen, dass die erwartete Laufzeit für zufällige Eingabelisten $\Theta(n \log n)$ ist. Diese Verbesserung beruht auf der Tatsache, dass bei zufälligen Eingabefolgen beide Teilfolgen mit sehr hoher Wahrscheinlichkeit nicht zu klein sind (mindestens ein Viertel) und dadurch logarithmische Rekursionstiefe erreicht wird. Das folgende Sortierverfahren erzwingt sogar eine perfekte Teilung in zwei fast gleich große Teilfolgen, dafür muss man beim Zusammensetzen der Teilfolgen einen höheren Aufwand betreiben.

Merge-Sort
Ein Rekursionschritt besteht darin, ein Liste der Länge n in die Teillisten der ersten $\lceil n/2 \rceil$ Elemente und der letzten $\lfloor n/2 \rfloor$ Elemente zu unterteilen und die rekusiv sortierten Teillisten durch eine Merge-Methode zusammenzufügen. Dabei geht man beide Teillisten parallel durch und nimmt das jeweils kleinste noch verfügbare Element in die Ergebnisliste auf.
Die Rekursionstiefe ist $\lceil \log_2 n \rceil$ und die Kosten eines Rekursionsschritts für eine Folge der Länge k sind $c \cdot k \in O(k)$. Aus diesen Fakten kann man $T(n) \in O(n \log n)$ ableiten. Genauer zeigt man induktiv für alle Zweierpotenzen $n = 2^i$, dass $T(n) \leq (c+1) n \log_2 n$ ist. Für den Induktionsanfang reicht die Feststellung, dass man für Folgen der Länge 1 nichts tun muss.
Ist die Behauptung für $n/2$ bereits bewiesen, erhalten wir die folgende Abschätzung:

$$T(n) \leq \; cn + T\left(\frac{n}{2}\right) + T\left(\frac{n}{2}\right) \leq cn + 2 \cdot (c+1) \cdot \frac{n}{2} \cdot \log_2 \frac{n}{2} \leq$$

$$(c+1) \cdot n + (c+1) \cdot n \cdot (\log_2 n - 1) = (c+1) \cdot n \log_2 n$$

Zahlen n, die keine Zweierpotenz sind kann man einfach durch die nächsthöhere Zweierpotenz $n^{\lceil \log_2 n \rceil} < 2n$ abschätzen: $T(n) \leq 2 \cdot (c+1) \cdot n \cdot \lceil \log_2 n \rceil$. Auf Grund der im letzten Abschnitt abgeleiteten allgemeinen unteren Schranke für Sortieralgorithmen ist $T(n) \in \Theta(n \log n)$.

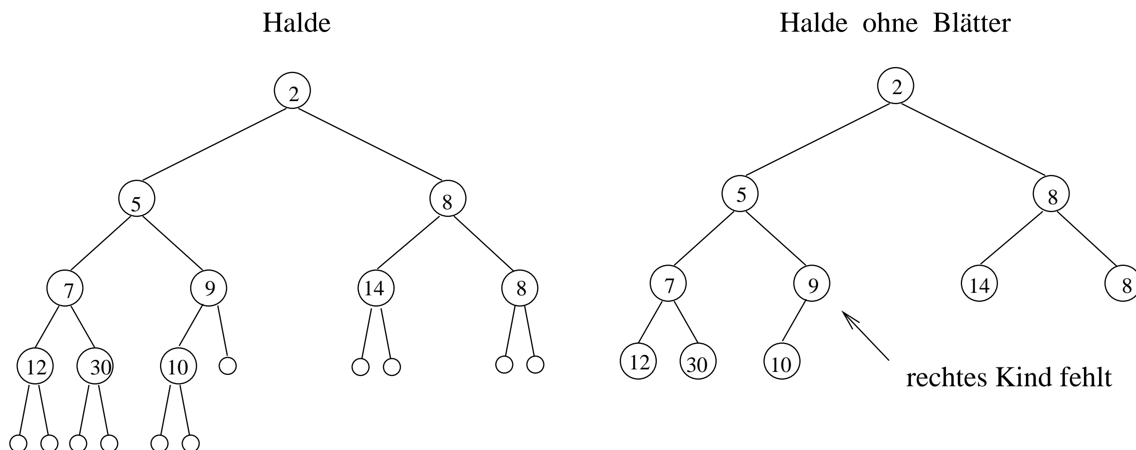
Counting-Sort
Dieser Algorithmus ist nicht vergleichsbasiert und kann deshalb die allgemeine untere Schranke durchbrechen. Dafür ist der Algorithmus nur unter der Einschr änkung verwendbar, dass alle Elemente der Folge aus einem Bereich $1, 2, 3, \ldots, K$ kommen. Man richtet ein mit Nullen initialisiertes Array C der Länge K ein, durchläuft dann die Eingabefolge $A = (a_1, a_2, \ldots a_n)$ und erhöht für jedes a_i den Eintrag $C[a_i]$ um 1. Danach durchläuft man das Feld C und wenn man auf einen Eintrag $C[j] > 0$ stößt, wird die Zahl j genau $C[j]$ mal in die Ausgabefolge B eingetragen.
Es gibt nur zwei Schleifendurchläufe, folglich ist die Laufzeit $\Theta(n+K)$. Da es wenig sinnvoll ist, bei Eingabegrẞen n, die beliebig anwachsen können, K als konstant anzunehmen, wird K oft als von n abhängige Größe betrachtet. So erreicht man unter der zusätzlichen Voraussetzung $K \in O(n)$ lineare Laufzeit.

Radix-Sort
Auch bei diesem Algorithmus ist die Laufzeitanalyse von mehreren Parametern abhängig. Wir erklären die Idee für das Sortieren von ganzen Zahlen in Dezimalschreibweise. Das Sortieren erfolgt in Runden. In der ersten Runde wird die Folge A durchlaufen und

die Zahlen bezüglich Ihrer letzten Stelle in 10 Folgen $A_0, A_1, \ldots A_9$ eingefügt. Mit der konkatenierten Folge $A_0 + A_1 + \ldots + A_9$ geht man in die zweite Runde in der nach der vorletzten Stelle aufgeteilt und wieder konkateniert wird. In jeder weiteren Runde geht man eine Stelle weiter nach links, also bestimmt die Anzahl d der Dezimalstellen der größten Zahl die Anzahl der Runden. Offensichtlich funktioniert dieses Verfahren, denn nach der i-ten Runde ist die Folge der Zahlen modulo 10^i korrekt sortiert.
Die Laufzeit ist $\Theta(d \cdot n)$. Man kann die Laufzeit reduzieren, indem man in jeder Runde nach zwei, drei oder mehr Dezimalstellen unterteilt, also jeweils in 100, 1000 oder mehr Folgen unterteilt, weil das die Anzahl der Runden halbiert, drittelt, usw. Mit anderen Worten wird die Zahldarstellung von der Basis 10 auf eine andere Basis $k = 100, 1000, \ldots$ umgestellt. Man muss aber dabei beachten, dass dann auch die Konkatenation k Schritte erfordert., was zu einer Laufzeit von $\Theta(d_k \cdot (n + k))$ führt, wobei $d_k = \lceil \log_k max \rceil$ die Stelligkeit der größten Zahl aus der Folge bezüglich der Basis k ist. Durch geschickte Parameteranpassung kann man viele Folgen in linearer Zeit sortieren, die bei Counting-Sort quadratische Laufzeit oder mehr erfordern würden.
Beispiel: Für eine Folge von n Zahlen aus dem Bereich $1, 2, \ldots, n^3$ benötigt Counting-Sort $\Theta(n^3)$ Zeit. Mit $k = 10^{\lceil \log_{10} n \rceil}$ erreicht Radix-Sort lineare Laufzeit $\Theta(n)$.

Heap-Sort

Grundlage für dieses Sortierverfahren ist eine spezielle Datenstruktur, durch deren Verwendung man optimale Laufzeit erreicht.
Definition: Eine **Halde (Heap)** ist ein binärer Baum in dessen inneren Knoten Zahlen gespeichert sind und der die fogenden zwei Eigenschaften hat:
Heap-Ordnungseigenschaft: Für jeden inneren Knoten v ist seine Zahl kleiner oder gleich den Zahlen in den Kindern von v (sofern diese Kinder keine Blätter sind).
Heap-Vollständigkeitseigenschaft: Ist h die Höhe der Halde, so sind alle Level von 0 bis $k-1$ vollständig gefüllt und die Knoten im letzten Level sind linksbündig angeordnet.

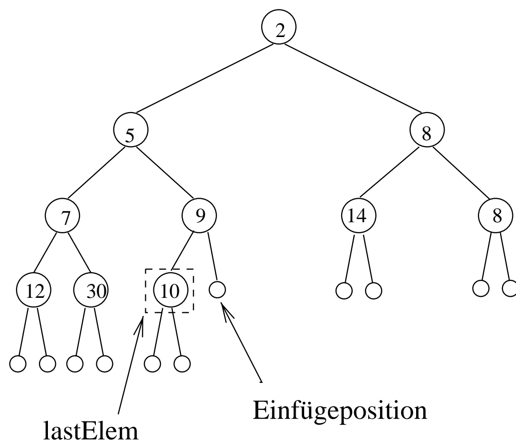


Die Vollständigkeitseigenschaft beschreibt also die äussere Gestalt einer Halde. Da die Blätter keine Informationen tragen, sind sie nur Platzhalter, um Halden in unser

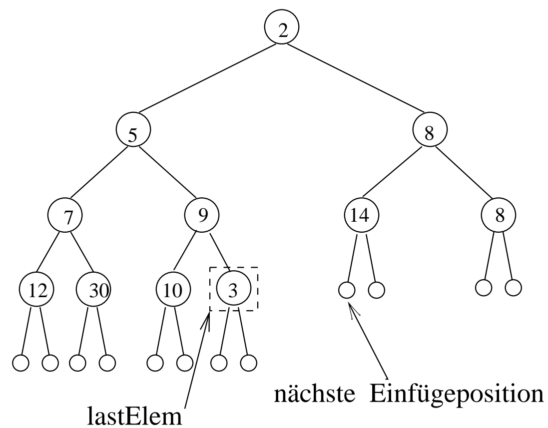
Konzept von binären Bäumen zu integrieren und zur Unterstützung einiger Algorithmen. Zur Vereinfachung von Beispielzeichnungen kann man auch auf die Blätter verzichten.
Wegen der Ordnungseigenschaft muss die kleinste Zahl immer in der Wurzel gespeichert sein und daraus resultiert die Idee zu einem zweistufigen Sortieralgorithmus:
1) Bilde eine Halde in der alle Zahlen aus der zu sortierenden Folge gespeichert sind.
2) Wiederhole die folgende Prozedur bis die Halde leer ist:
    Lösche die Zahl aus der Wurzel, füge sie in die sortierte Folge ein und
    rekonstruiere eine Halde aus den restlichen Zahlen.
Zur Umsetzung benötigt man Einfüge- und Löschoperationen, für die neben der üblichen Baumstruktur der am weitesten rechts stehende innere Knoten aus dem vorletzten Level wichtig ist. Wir bezeichnen ihn mit `lastElem`.

Einfügen einer Zahl x: Man bestimmt zuerst die Einfügeposition, indem man von `lastElem` den nach oben startenden Weg der Euler-Tour bis zum nächsten Blatt folgt. Das ist gleichzeitig das erste Blatt, dass bei der (zyklischen) Inorder-Traversierung auf `lastElem.rightChild()` folgt. Durch Anhängen von zwei Blättern wird dieses Blatt in einen inneren Knoten verwandelt und als neues `lastElem` gesetzt. Damit ist die Vollständigkeitseigenschaft erfüllt. Dann wird x in die Einfügeposition eingetragen und bis zur Herstellung der Ordnungseigenschaft schrittweise nach oben vertauscht. Man nennt diese Prozedur Verhalden (Up-Heap Bubbling).

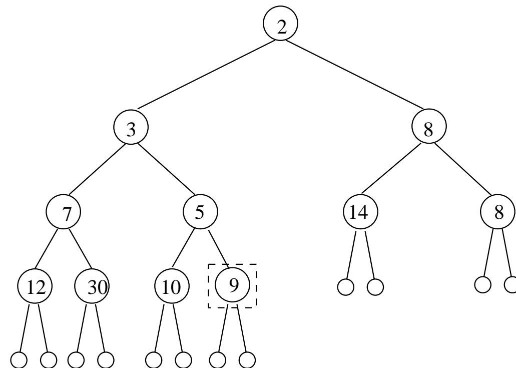
Einfügen von x = 3:



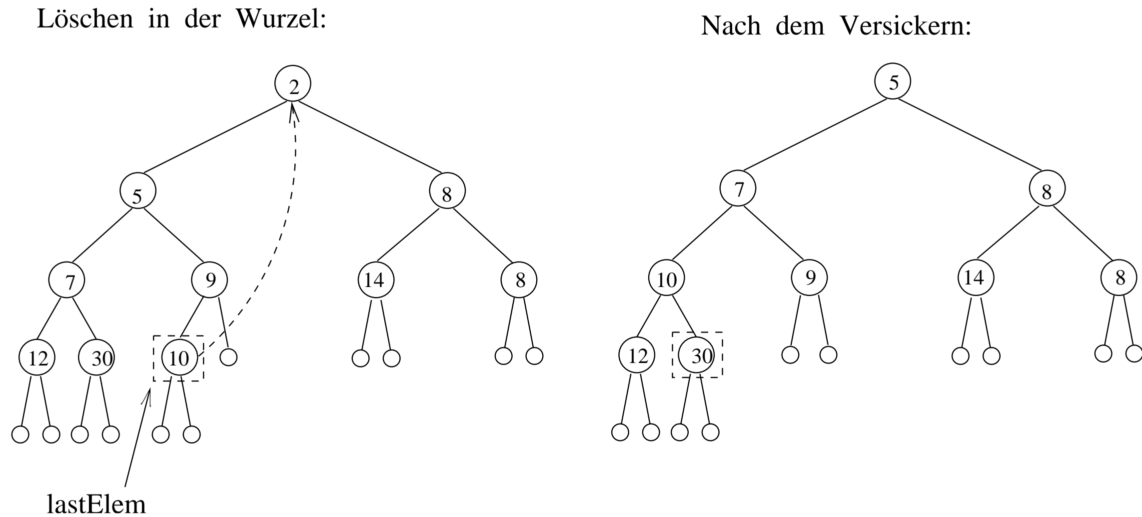
Herstellung der Vollständigkeit:



nach dem Verhalden:



Löschen der Zahl aus der Wurzel: Das Löschen ist trivial, aber zur Rekonstruktion einer Halde ist noch einiges zu tun. Die Zahl y aus `lastElem` wird in die Wurzel verschoben, die Blätter von `lastElem` gestrichen und `lastElem` zurückgesetzt (umgekehrt wie beim Einfügen). Zur Wiederherstellung wird y so lange nach unten getauscht, wie ein Kind unter der aktuellen Position von y eine kleinere Zahl speichert. Halten beide Kinder eine kleinere Zahl als y, entscheidet man sich für die kleinere von beiden. Man nennt diese Prozedur Versickern (Down-Heap Bubbling).



Laufzeitanalyse: Offensichtlich kann jede Einfüge- und Löschoperation in $O(h)$ Zeit ausgeführt werden, wobei h die Höhe (Tiefe) der aktuellen Halde ist. Da man zum Sortieren einer Folge der Länge n genau n Einfüge- und n Löschoperationen ausgeführt werden und nach folgenden Satz $h \in O(\log n)$ gilt, ist $T(n) \in O(n \log n)$.

Satz: Die Höhe einer Halde, die n Zahlen speichert, ist $h = \lceil \log_2 (n+1) \rceil$.
Beweis: Aus der Vollständigkeitseigenschaft folgt, dass die Level i von 0 bis $h-2$ jeweils 2^i und Level $h-1$ mindestens einen und höchstens 2^{h-1} innere Knoten haben:

$$\begin{array}{rcccl} 1+2+4+\ldots+2^{h-2}+1 & \leq & n & \leq & 1+2+4+\ldots+2^{h-2}+2^{h-1} \\ 2^{h-1} & \leq & n & \leq & 2^h - 1 \\ 2^{h-1}+1 & \leq & n+1 & \leq & 2^h \\ h-1 < \log_2 (2^{h-1}+1) & \leq & \log_2 (n+1) & \leq & \log_2 2^h = h \end{array}$$

Damit ist $\lceil \log_2 (n+1) \rceil = h$. □

Die erste Stufe von Heap-Sort kann durch eine sogenannte Bottom-Up-Heapkonstruktion weiter verbessert werden. Zur Vereinfachung wird diese Idee nur für Zahlen der Form $n = 2^h - 1$ erläutert. Man nimmt 2^{h-1} dieser Zahlen und baut mit jeder eine Halde der Höhe 1 mit der Zahl in der Wurzel. Man bildet aus diesen Halden 2^{h-2} Paare. Aus jedem

Paar wird eine neue Halde gebildet und in die Wurzel wird eine von den verbliebenen Zahlen gesetzt, die versickert werden muss. Das wird rekursiv wiederholt, bis im letzten Schritt eine Halde mit allen $n = 2^h - 1$ Zahlen entstanden ist.

Satz: Die Bottom-Up-Heapkonstruktion erfordert nur lineare Zeit.
Beweis: Es genügt die Anzahl der Vertauschungen abzuschätzen. Der erste Schritt kommt ohne Vertauschungen aus, für die nächsten 2^{h-2} Zahlen reicht eine Vertauschung, es folgen 2^{h-3} Zahlen mit zwei Vertauschungen, im letzten Schritt braucht man maximal $h - 1$ Tauschoperationen. Durch geeignetes Zusammenfassen, umformen und Verwendung der Summenformel für geometrische Reihen erhält man:

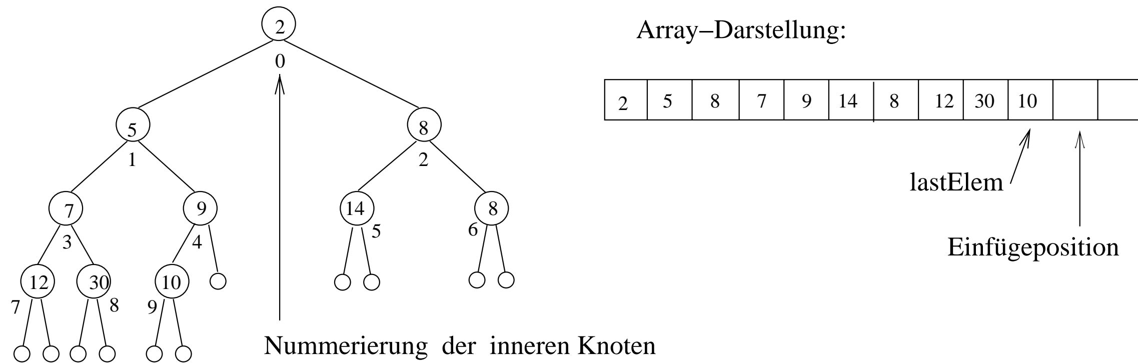
$$1 \cdot 2^{h-2} + 2 \cdot 2^{h-3} + \ldots + (h-1) \cdot 2^0 = \sum_{i=1}^{h-1} \sum_{j=i}^{h-1} 2^{h-1-j} = 2^h \sum_{i=1}^{h-1} (2^{-i} - 2^{-h}) \leq n \cdot \sum_{i=1}^{h-1} 2^{-i} \leq n$$

Es gibt einen zweiten Beweis in Form einer Gewinn-Verlustrechnung. Die Rechnung wird wesentlich einfacher, aber formal muss man vollständige Induktion anwenden. Wir gehen davon aus, das für $n = 2^h - 1$ alle Zahlen in den Levels 0 bis $h - 2$ ein Guthaben von zwei Euro bekommen. Für jede Austauschoperation muss ein Euro bezahlt werden. Wir zeigen, dass am Ende immer mindestens $h - 1$ Euro übrig bleiben. Damit kann man die Anzahl der Tauschoperationen mit $2 \cdot (2^{h-1} - 1)) - (h - 1) = n - h$ abschätzen.
Induktionsanfang:
Das ist offensichtlich richtig für $h = 1$ (kein Guthaben, aber wenn nur in der Wurzel eine Zahl steht muss nicht getauscht werden und das Restguthaben ist $0 = 1 - 1$) und für $h = 2$ (Anfangsguthaben 2 Euro und maximal ein Austausch).
Induktionsschritt von h zu $h + 1$:
Wir betrachten die Halde, die nach Bottom-Up-Konstruktion für eine Folge aus $2^{h+1} - 1$ Zahlen entsteht. Nach Induktionsvoraussetzung gibt es im linken und im rechten Teilbaum jeweils ein Restguthaben von mindestens $h - 1$ Euro. Zusammen mit den 2 Euro der Wurzel sind das $2h$ Euro. Da man die Zahl aus der Wurzel im schlechtesten Fall bis zum Level h versickern muss, bleiben am Ende mindestens $h = (h + 1) - 1$ Euro übrig. □

Es bleibt anzumerken, dass es keine Chance gibt, auch die zweite Phase des Heap-Sorts in linearen Zeit zu realisieren, denn das stünde im Widerspruch zur allgemeinen unteren Schranke für Sortieralgorithmen.

Array-Implementierung für Halden
Bisher sind wir, auch um die anschauliche Intuition zu unterstützen, davon ausgegeangen, dass man Halden als binäre Bäume implementiert. Man kann Halden aber auch sehr einfach mit Feldern implementieren:
Die Level werden nacheinander (von 0 beginnend) und jedes einzelne Level von links nach rechts in das Array übertragen. Also steht in $A[0]$ der Wert aus der Wurzel in $A[1]$ und $A[2]$ die Werte des linken und rechten Kindes der Wurzel u.s.w.
Das hat den Vorteil, dass lastElem immer am Ende steht und die erste freie Zelle die Einfügeposition darstellt. Zum Einfügen und Löschen ist aber auch die Baumstruktur sehr wichtig. Sie ist nicht ganz offensichtlich, aber bei genauerer Betrachtung findet

man heraus, dass die Kinder des von $A[i]$ repräsentierten Knotens von $A[2i+1]$ und $A[2a+2]$ repräsentiert werden. Damit ist es auch leicht den Vaterknoten zu bestimmen: Für ungerade j geht man nach $\frac{j-1}{2}$ und für gerade j nach $\frac{j-2}{2}$.



Die Halden-Datenstruktur ist nicht nur zum Sortieren geeignet, man kann sie auch sehr gut zur Implementierung von **Prioritäts-Warteschlangen (Priority-Queue)** verwenden. Solche Warteschlangen verwalten Objekte mit einem Schüssel, wobei die Schlüssel von einem Datentyp sind für den ein Vergleichsoperator $\leq$ definiert ist (im einfachsten Fall sind es Zahlen). Der Vergleichsoperator muss die folgenden vier Eigenschaften haben:

1. **total**, d.h. für beliebige Schlüssel a, b muss $a \leq b$ oder $b \leq a$ gelten;

2. **reflexiv**, d.h. für jeden Schlüssel a gilt $a \leq a$;

3. **transitiv**, d.h. für beliebige Schlüssel a, b, c gilt: aus $a \leq b$ und $b \leq c$ folgt $a \leq c$;

4. **antisymmetrisch**, d.h. für beliebeige Schlüssel a, b gilt: aus $a \leq b$ und $b \leq a$ folgt $a = b$;

Eine Prioritäts-Warteschlange zeichnet sich dadurch aus, dass man jederzeit beliebige Objekte mit Schlüssel einfügen kann und dass sie jederzeit das Objekt mit dem kleinsten Schlüssel zur Verfügung stellen und löschen kann. Das FIFO-Prinzip wird also durch die höchste Priorität (gleich kleinster Schlüssel) ersetzt.
Prioritäts-Warteschlangen können auch mit Arrays oder verketteten Listen implementiert werden, aber dann benötigt eine Opertationsart lineare Zeit und die andere konstante Zeit. Die Heap-Implementierung mit logarithmischer Zeit für beide Operationen ist wesentlich effizienter.

Wörterbücher (Dictonaries) und ihre Implementierungen

Auch in diesem Abschnitt geht es um die Verwaltung von Objekten mit einem Schlüssel. Wir nennen die aus einem Objekt (Element) und einem Schüssel bestehende Einheit einen **Eintrag** (`Item`). Ziel ist die Konstruktion einer Datenstruktur zur Verwaltung von Items, die über Methoden zum Einfügen und Löschen von Items verfügt und in der man Items nach ihrem Schlüssel suchen kann. Solche Datenstrukturen spielen in verschiedensten Anwendungen eine wichtige Rolle:

Lexikon (Schüssel = Suchbegriff, Objekt = Erklärung)
Telefonbuch (Schlüssel = Teilnehmer, Objekt = Telefonnummer)
Kundendatei (Schlüssel = Kundennummer, Objekt = Kundendaten)

Man kann hier auch auf die Forderung verzichten, dass die Schlüsselmenge total geordnet (siehe Priority Queues) sein soll, und spricht dann von ungeordneten Wörterbüchern. Wir werden uns aber nur mit dem geordneten Fall beschäftigen.

Im Einzelnen sollte ein Wörterbücher die folgenden Methoden implementieren:

```
int       size()
Object    findElement(key k)
Object[]  findAllElements(key k)
Object    insertItem(key k,Object e)
Object    removeElement(key k)
Object[]  removeAllElements(key k)
```

Auch für Wörterbücher bieten sich die bereits mehrfach besprochenen Implementierungen mit Arrays oder mit doppelt verketteten Listen an. Da bei Listenimplementierungen (geornet und ungeordnet) sowie bei der Implementierung mit ungeordneten Feldern schon für das Suchen lineare Zeit erforderlich ist (bei Listen kann man dafür danach in konstanter Zeit einfügen und löschen), sollte man sie nur für kleine Datenmengen verwenden.
Dagegen haben geordnete Felder den Vorteil, das man Schlüssel mit **binärer Suche** in logarithmischer Zeit finden kann. Zur Beschreibung dieser Methode bezeichnen wir mit *low* und *high* zwei Indizes, die ein Intervall in einem Feld A eingrenzen, in dem der Schlüssel k gefunden werden soll. Zur Vereinfachung bezeichnen wir mit $key(i)$ den Schlüssel des Eintrags an der i-ten Stelle im Feld. Die binäre Suche nach k ist rekursiv definiert. Der Pseudocode ist so zu verstehen, dass jede return-Anweisung die Methode sofort beendet:

`while` $(low \leq high)$
  $mid = \lfloor \frac{low+high}{2} \rfloor$
  `if` $(k == key(mid))$ `return` $elem(mid)$
    `else if` $(k < key(mid))$ $high = mid - 1$
      `else` $low = mid + 1$
`return NO_SUCH_KEY`

Wenn man also eine (weitgehend) statische Datenstruktur entwerfen will, in der nach einer längeren Vorverarbeitungsphase nur gesucht werden soll (ohne sie zu aktualisieren), ist ein geordnetes Feld die geeignetste Datenstruktur. Leider wird dieser Vorteil bei dynamischen Strukturen durch lineare Einfüge- und Löschzeiten wieder eingebüßt.

Binäre Suchbäume als Wörterbücher

Zur Implementierung von Wörterbüchern verwenden wir binäre Suchbäume, bei denen nur die inneren Knoten Einträge (items) tragen. Zur Suche eines Eintrags mit Schlüssel k wird die eine rekursive Methode `TreeSearch(k,v)` implementiert, die entweder einen inneren Knoten mit Schlüssel k oder ein Blatt zurückgibt, das für den Fall `NO_SUCH_KEY` steht.

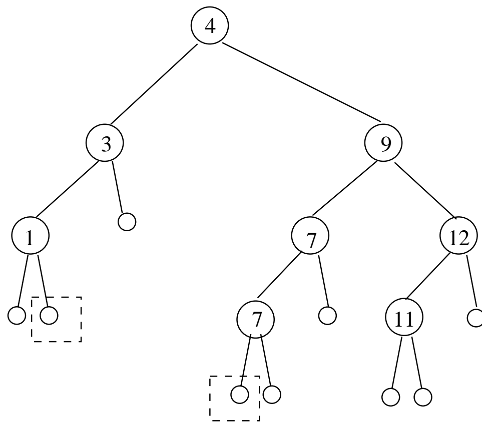
```
TreeSearch (k, v) {
        if (v Blatt) return v
        else if (k == key(v)) return v
                else if (k < key(v)) return TreeSearch (k, leftChild(v))
                        else return TreeSearch (k, rightChild(v))
}
```

Die Laufzeit dieser Suchmethode ist $O(h)$ wobei h die Höhe von v ist, bei Aufruf mit der Wurzel ist es die Höhe des Baums. Auch das Einfügen und Löschen lässt sich nach folgenden Regeln in dieser Zeit realisieren.

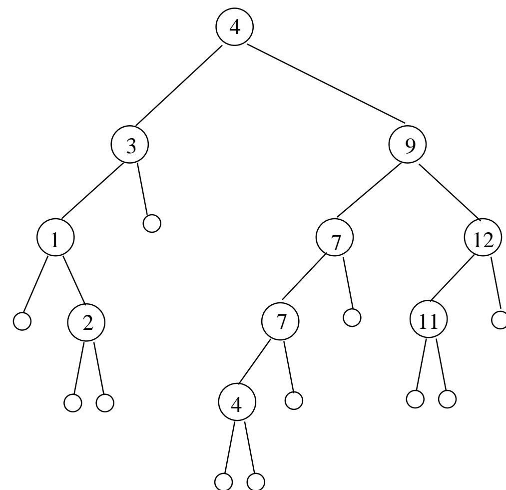
Einfügen eines Eintrags $item(k, e)$:

1) $v =$ `TreeSearch` $(k, root)$
2.a) Ist v ein Blatt, so werden 2 Blätter angefügt und (k, e) in v eingetragen
2.b) Ist v ein innerer Knoten, sei w der erste Knoten bei der Inorder-Traversierung von `rightChild`(v). Achtung: w ist ein Blatt! Dann wird (k, e) in w eingetragen und zwei Blätter angefügt.

Suchbaum mit Einfügepositionen für die Schlüssel 2 und 4



Nach dem Einfügen



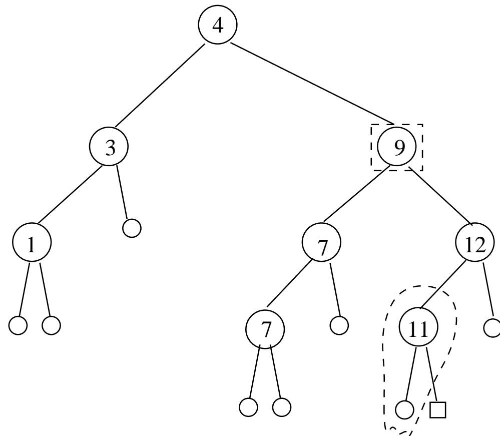
Löschen eines Eintrags mit Schlüssel k:

1) $v =$ `TreeSearch` $(k, root)$

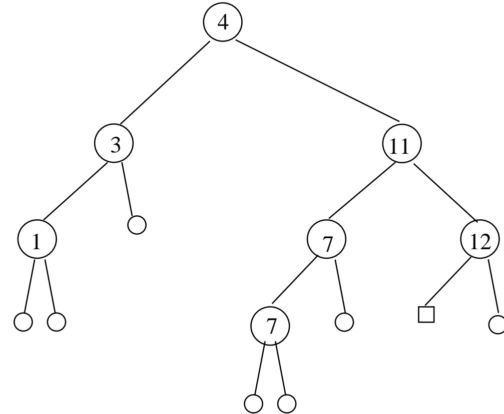
2.a) Ist v ein Blatt, gibt es keinen Eintrag mit Schlüssel $k \mapsto$ `NO_SUCH_KEY`

2.b) Ist v ein innerer Knoten, sei w der erste Knoten bei der Inorder-Traversierung von `rightChild`(v) und u der Vaterknoten von w. Die Einträge von v und u werden vertauscht. Dann löscht man die Knoten u und w und der Geschwisterknoten von w wird an die Stelle von u gesetzt.

Suchposition und Löschpositionen den Schlüssel 9



Nach Austausch der Einträge und Löschung



Das nächste Ziel besteht darin, spezielle Suchbaumfamilien zu entwerfen, die logarithmische Höhe garantieren und diese Eigenschaft auch bei Einfüge- und Löschoperationen behalten. Es gibt dazu eine Reihe von Lösungen, aus der wir nur die sogenannten AVL-Bäume näher betrachten werden.

AVL-Bäume

Definition: Ein binärer Baum hat die **Höhen-Balance-Eigenschaft**, wenn für jeden inneren Knoten v die Höhendifferenz des linken und des rechten Kindes von v höchstens 1 ist. Wir verwenden für die Höhen-Balance-Eigenschaft die Abkürzung **HBE**. Wenn der von dem Knoten v bestimmte Unterbaum $T(v)$ die HBE hat, sprechen wir auch davon, dass v die HBE hat.

Definition: Ein **AVL-Baum** ist ein binärer Suchbaum (mit Einträgen in den inneren Knoten), der die HBE besitzt.

Satz: Ein AVL-Baum mit n Einträgen hat die Höhe $O(\log n)$.

Beweis: Wir betrachten das Problem zuerst von der anderen Seite und bezeichnen mit $n(h)$ die minimale Anzahl von inneren Knoten, die ein AVL-Baum der Höhe h haben muss. Offensichtlich ist $n(1) = 1$ und $n(2) = 2$. Allgemein muss in jedem AVL-Baum der Höhe h mindestens ein Kind der Wurzel die Höhe $h-1$ haben. Wegen der Höhen-Balance muss dann das andere Kind mindestens die Höhe $h-2$ haben. Berücksichtigt

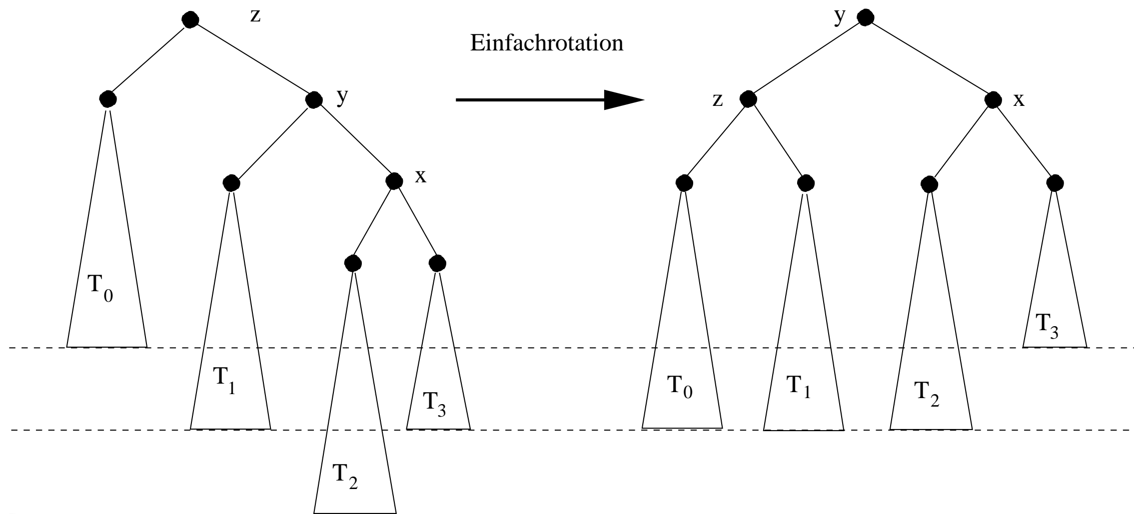
man die Wurzel als weiteren inneren Knoten, so erhält man:

$$n(h) \geq 1 + n(h-1) + n(h-2) > 2n(h-2) \geq 2^{\lfloor \frac{n}{2} \rfloor} n(1) = 2^{\lfloor \frac{h}{2} \rfloor}$$

Durch Logarithmieren ergibt sich $\lfloor \frac{h}{2} \rfloor \leq \log_2 n(h)$, woraus die Behauptung folgt.

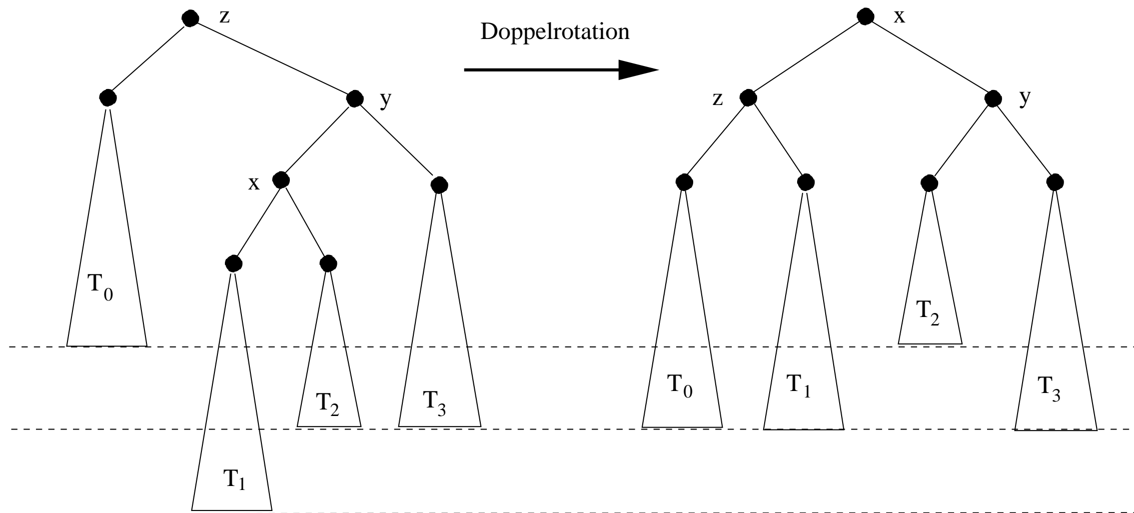
Aus diesem Satz folgt, dass Einfüge- und Löschoperationen in AVL-Bäumen in logarithmischer Zeit arbeiten. Es kann aber sein, dass eine solche Operation die HBE zerstört und deshalb müssen auch die Kosten für die Wiederherstellung dieser Eigenschaft auf die Operationen angerechnet werden. Zur Rekonstruktion von AVL-Bäumen nutzt man sogenannte Rotationen. Das sind Operationen, die lokal die Struktur eines Baums (insbesondere die Höhenverhältnisse) verändert. Es gibt zwei Grundtypen. Sie werden **einfache Rotation** und **doppelte Rotation** genannt und sind in den folgenden Abbildungen illustriert.
In beiden Fällen geht man davon aus, dass z ein tiefster Knoten ohne HBE ist. Wir bezeichnen die Höhe von z mit $h = h(z)$, das Kind und das Enkelkind auf dem längsten Weg von z zu einem Blatt mit y und x. Durch die Auswahl von z müssen x und y die HBE haben und es gilt $h(x) = h(y) - 1 = h - 2$. Wir setzen außerdem voraus, dass der Geschwisterknoten von y (also das zweite Kind von z) die Höhe $h - 2$ hat.

Einfache Rotation Um diese Rotationen anwenden zu können, müssen y und x jeweils linke oder jeweils rechte Kinder sein. In diesem Fall kann man $T(z)$ so verändern, dass y zur Wurzel wird mit den Kindern z und x. Das ursprüngliche Geschwisterkind von x wechselt als Kind zu z. Im Ergebnis hat der neue Baum $T'(y)$ die HBE und seine Höhe hat sich auf $h - 1$ reduziert. Wenn man die Reihenfolge der Teilbäume $T_0, \ldots T_3$ von links nach rechts (siehe Abbildung) nicht ändert, bleibt auch die Suchbaum-Ordnung erhalten.



Doppelte Rotation Um diese Rotationen anwenden zu können, muss y ein rechtes und x ein linkes Kind sein oder umgekehrt. In diesem Fall kann man $T(z)$ so verändern, dass

x zur Wurzel wird mit den Kindern y und z. Ein Kind von x wechselt zu y, das andere zu z. Damit wird die HBE hergestellt und die Höhe des Baums auf $h-1$ reduziert. Die Zuordnung der Kinder von x auf y und z muss wieder so erfolgen, dass durch Einhaltung der Reihenfolge von $T_0, \ldots T_3$ die Suchbaum-Ordnung erhalten bleibt.



Da eine Rotation (egal, ob einfach oder doppelt) nur eine lokale Operation ist, die man durch eine konstante Anzahl von Referenzverschiebungen realisieren kann, verbraucht sie auch nur $O(1)$ Zeit. Um diese Operationen aber effizient zur Wiederherstellung der HBE nach einer Einfüge- oder Löschoperation einsetzen zu können, müssen alle Knoten des AVL-Baums ihrer Höhe kennen. Da beide Operationen nur die Höhe der Knoten auf dem Weg p von der Einfge- bzw. Löschposition zur Wurzel verändern können, kostet die Aktualisierung nur $O(\log n)$ Zeit (wenn man auf dem Weg p zum ersten Mal einen Knoten betritt, dessen Höhe sich nicht ändert, kann man auch abbrechen). Bei dieser Aktualisierung wird gleichzeitig die HBE der Knoten überprüft (einfach Höhen der Kinder vergleichen). Wenn man auf einen Knoten ohne HBE stößt, wird die passende Rotation angewendet. Dabei ist auf einen grundlegenden Unterschied zwischen Einfüge- oder Löschoperationen zu achten:

Durch eine **Einfügeoperation** kann sich die Höhe nur vergrößern, d.h. die zu rotierenden Knoten x, y, z liegen auf dem Weg p. Da sich nach der ersten Rotation die (vergrößerte) Höhe wieder um 1 reduziert, ist man fertig.

Durch eine **Löschoperation** kann sich die Höhe nur verkleinern, d.h. man trifft aus dem Teilbaum mit der geringeren Höhe kommend auf z, und muss y und x in dem anderen Teilbaum suchen. Da sich die Höhe des rotierten Teilbaums verringert, kann es sein, dass auf dem weitern Weg zur Wurzel neue Rotationen notwendig werden. In jeden Fall reichen aber $O(\log n)$ Rotationen aus.

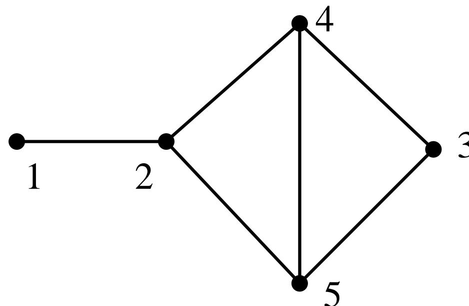
Satz: Bei der Implementierung des Wörterbuch-ADTs mit AVL-Bäumen haben Such-, Einfüge- und Löschoperationen eine Laufzeit von $O(\log n)$.

Graphen und ihre Darstellungen

Ein **Graph** beschreibt Beziehungen zwischen den Elementen einer Menge von Objekten. Die Objekte werden als Knoten des Graphen bezeichnet; besteht zwischen zwei Knoten eine Beziehung, so sagen wir, dass es zwischen ihnen eine Kante gibt.
Definition: Für eine Menge V bezeichne $\binom{V}{2}$ die Menge aller zweielementigen Untermengen von V. Ein **einfacher, ungerichteter Graph** $G = (V, E)$ (kurz **Graph** genannt) besteht aus einer endlichen Menge V von **Knoten** , auch **Ecken (Vertex)** genannt, und einer Menge $E \subseteq \binom{V}{2}$ von **Kanten (Edge)**. Hier sind Kanten ungeornete Paare von Knoten, d.h. $\{u, v\}$ und $\{v, u\}$ sind zwei verschiedene Schreibweisen für ein und dieselbe Kante. Im Gegensatz dazu ist endlicher **gerichteter Graph** G ein Paar (V, E) bestehend aus einer endlichen Knotenmenge V und einer Kantenmenge E von geordneten Knotenpaaren $e = (u, v)$, mit $u, v \in V$.

Ist $e = \{u, v\}$ eine Kante von G, dann nennt man die Knotenen u und v zueinander **adjazent** oder **benachbart** und man nennt sie **inzident** zu e. Die Menge $N(v) = \{u \in V \mid \{u, v\} \in E\}$ der zu einem Knoten v benachbarten Knoten wird die **Nachbarschaft** von v genannt. Der **Grad** eines Knotens v wird durch $deg(v) = |N(v)|$ definiert.
Die Anzahl der Knoten $|V|$ bestimmt die **Ordnung** und die Anzahl der Kanten $|E|$ die **Größe** eines Graphen.

Im Folgenden werden verschiedene Darstellungen von Graphen an einem Beispiel demonstriert.
1) Darstellung als Zeichnung:



2) Darstellung als **Adjazenzmatrix**. Jedem Knoten wird eine Zeile und eine Spalte zugeordnet und der Eintrag in der Zeile von u und der Spalte von v wird 1 gesetzt, wenn $\{u, v\}$ eine Kante des Graphen ist (sonst 0):

$$\begin{pmatrix} 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 & 1 \\ 0 & 1 & 1 & 0 & 1 \\ 0 & 1 & 1 & 1 & 0 \end{pmatrix}$$

Während die Adjazenzmatrix eines ungerichteten Graphen symmetrisch ist (Diagonale als Symmetrieachse) sind Adjazenzmatrizen von gerichteten Graphen im allgemeinen nicht symmetrisch. Durch Verwendung beliebiger Zahlen für die einzelnen Einträge kann man auch Graphen mit Kantenbewertungen darstellen.

3) Darstellung als **Adjazenzliste**. Für jeden Knoten wird die Liste seiner Nachbarn angegeben (Liste von Listen):
$1:2;\ \ 2:1,4,5;\ \ 3:4,5;\ \ 4:2,3,5;\ \ 5:2,3,4;$ oder mit anderer Syntax
$(2),(1,4,5),(4,5),(2,3,5),(2,3,4)$

4) Darstellung als **Inzidenzmatrix**. Jedem Knoten wird eine Zeile und jeder Kante eine Spalte zugeordnet und der Eintrag in der Zeile von u und der Spalte von e wird 1 gesetzt, wenn u eine Ecke der Kante e ist (sonst 0):

$$\begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 & 0 \\ 0 & 1 & 0 & 0 & 1 & 1 \\ 9 & 0 & 1 & 1 & 0 & 1 \end{pmatrix}$$

Die Darstellung mit Inzidenzmatrizen hat sich zwar für verschiedene theoretische Betrachtungen als nützlich erwiesen, ist aber für algorithmische Anwendungen eher ungeeignet. Ob man sich bei der Beschreibung und Implementierung von Graphalgorithmen für Adjazenzlisten oder für Adjazenzmatrizen entscheiden sollte, hängt zum einen von den Eigenschaften der zu bearbeitenden Graphen ab (oft werden Algorithmen für spezielle Graphklassen entwickelt), zum anderen von der Art der Zugriffe, die der Algorithmus vornehmen muss. Folgende Aspekte sind dabei zu beachten:

1) Speicherverbrauch Die Adjazenzmatrix verbraucht für Graphen mit n Knoten $\Theta(n^2)$ Speicher, unabhängig von der Größe $m = |E|$ des Graphen. Dagegen ist der Speicherbedarf einer Adjazenzliste nur $\Theta(n+m)$ oder sogar $\Theta(m)$, wenn man leere Nachbarschaftslisten einfach ignoriert. Da für planare Graphen oder Graphen mit beschränktem Grad $m = O(n)$ gilt, reduziert sich der Speicherbedarf auf $O(n)$.

2) Anfragezeit Häufig wird in Graphalgorithmen die Frage, ob zwei bestimmte Knoten adjazent sind, als Entscheidungsgrundlage für das weitere Vorgehen verwendet. Eine solche Abfrage kann mit der Adjazenzmatrix in konstanter Zeit beantwortet werden, bei Adjazenzlisten erhöht sich der Aufwand je nach Implementierung auf $\Theta(n)$ (Listen der Nachbarn ungeordnet) oder $\Theta(\log n)$ (geordnete Arrays mit Binärsuche oder AVL-Bäume).

3) Updates Das Einfügen und Streichen von Kanten erfolgt bei Adjazenzmatrizen in linearer Zeit, bei Adazenzlisten in linearer Zeit (oder in logarithmischer Zeit bei Verwendung vvon AVL-Bäumen oder ähnlichen Strukturen).

4) Nachbarschaftsaufzählung Um einen Nachbarn eines gegebenen Knoten zu erhalten sind die Adjazenzlisten mit konstanter Zeit (bzw. Zeit $O(l)$ für die Liste aller l Nachbarn) im Vorteil gegen $\Theta(n)$ bei Adjazenzmatrizen.

Da Graphen über eine sehr einfache Struktur verfügen, finden sie bei der Modellierung und algorithmischen Lösung vieler praktischer Probleme Anwendung, wie z.B.

- Modellierung von Straßen-, Flug- und Telefonnetzen
- Darstellung von Molekülen
- Interpretation von Relationen (Beziehungsgeflechten)
- Gerüste von Polyedern (lineare Optimierung)
- Entwurf von Mikrochips (VLSI-Design)

Die folgenden Beispiele für graphentheoretische Aufgabenstellungen haben die die Entwicklung der Graphentheorie stark beeinflusst und unterstreichen die praktische Relevanz dieser Struktur:

1) 4-Farben-Problem: Man stelle sich die Welt mit einer beliebigen politischen Landkarte vor. Wir definieren einen Graphen, indem wir jedem Land einen Knoten zuordnen und zwei Knoten mit einer Kante verbinden, wenn sie einen gemeinsamen Grenzabschnitt haben. Wie viele Farben braucht man, um die Länder so einzufärben, dass benachbarte Länder verschiedene Farben haben. Man hat (mit Computerhilfe) bewiesen, dass vier Farben immer ausreichen! Einen Beweis ohne Computer gibt es bis heute nicht.

2) Eulersche Graphen: Man charakterisiere jene Graphen, bei denen man die Kanten so durchlaufen kann, dass man jede Kante einmal benutzt und man am Schluss wieder am Ausgangspunkt steht. ("Haus vom Nikolaus"-Prizip). Der Ausgangspunkt für diese Frage war das von Euler gelöste sogenannte Königsberger Brückenproblem.

3) Hamiltonsche Graphen: Dies sind solche Graphen, die man so durchlaufen kann, dass man jeden Knoten genau einmal besucht bis man zum Ausgangsknoten zurückkehrt. Während man für das vorherige Problem effiziente algorithmische Lösungen kennt, ist dieses algorithmisch schwer (NP-vollständig).

4) Travelling Salesman Problem (TSP): Oft hat man es mit bewerteten Graphen zu tun, das heißt Kanten und/oder Knoten haben zusätzliche Informationen wie Gewichte, Längen, Farben etc.
Ein Beispiel ist das TSP. Wir haben n Städte. Für jedes Paar $\{u, v\}$ von Städten kennt man die Kosten, um von u nach v zu kommen. Man entwerfe für den Handelsreisenden eine geschlossene Tour, die alle Städte besucht und minimale Gesamtkosten hat. Auch dies ist ein algorithmisch schweres Problem.

5) Planare Graphen: Welche Graphen lassen sich so in der Ebene zeichnen, dass sich Kanten nicht schneiden, also sich höchstens in Knoten berühren? Wie kann man sie charakterisieren und algorithmisch schnell erkennen?

6) Flussprobleme: Angenommen ein (Informations)-Netzwerk wird durch einen Graphen mit Kantenbewertung beschrieben, wobei diese Werte eine Obergrenze für die Übertragungskapazitäten der Kanten angeben. Wie groß ist dann der maximale (Informations)-Fluss zwischen zwei gegebenen Knoten s und t?

Satz: Für jeden Graph $G = (V, E)$ gilt $\sum_{v \in V} deg(v) = 2|E|$, , d.h. $\sum_{v \in V} deg(v)$ ist eine gerade Zahl.
Beweis: Bei Betrachtung der Inzidenzstruktur zwische Knoten und Kanten ergibt sich die Aussage durch doppeltes Abzählen: Für jeden Knoten v ist die Anzahl inzidenter Kanten $deg(v)$ und jede Kante ist zu ihren zwei Eckknoten inzident. Damit erhalten wir $\sum_{v \in V} deg(v) = \sum_{e \in E} 2 = 2|E|$.
Folgerung: Die Anzahl der Knoten mit ungeraden Grad ist in jedem Graphen gerade.

Definition: Seien $G = (V, E)$ und $G' = (V', E')$ zwei Graphen. Eine Abbildung $\varphi : V \longrightarrow V'$ wird **Graphhomomorphismus** genannt, falls für alle Kanten $\{u, v\} \in E$ auch $\{\varphi(u), \varphi(v)\} \in E'$ gilt. Ist darüber hinaus φ eine bijektive Abbildung und φ^{-1} auch ein Graphhomomorphismus, so nennt man φ einen **Graphisomorphismus** (und G, G' zueinander **isomorph**).

Die folgenden Standardbeispiele beschrieben formal gesehen nicht einzelne Graphen, sondern Isomorphieklassen.

1) Mit K_n $(n \geq 1)$ bezeichnet man den **vollständigen Graphen** der Ordnung n, d.h. eine Knotenmenge V mit $|V| = n$ und der vollen Kantenmenge $\binom{V}{2}$.
2) Mit C_n $(n \geq 3)$ bezeichnet man den **Kreis** der Länge n, d.h. eine Knotenmenge $\{v_1, v_2, \ldots, v_n\}$ mit der Kantenmenge $\{\{v_1, v_2\}, \{v_2, v_3\}, \ldots, \{v_{n-1}, v_n\}, \{v_n, v_1\}\}$. **3)** Mit Q_n $(n \geq 1)$ bezeichnet man den **n-dimensionalen Würfel** mit Knotenmenge $\{0, 1\}^n$ (Menge aller n-Tupel über $\{0, 1\}$) wobei zwei Tupel dann und nur dann adjazent sind, wenn sie sich an genau einer Stelle unterscheiden.
4) Mit $K_{n,m}$ $(n, m \geq 0)$ bezeichnet man den **vollständigen, bipartiten Graphen**, dessen Eckenmenge V die disjunkte Vereinigung von zwei Mengen A und B mit $|A| = n,\ |B| = m$ ist und dessen Kantenmenge aus allen Paaren $\{a, b\}$ mit $a \in A,\ b \in B$ besteht.
Definition: Man nennt $G' = (V', E')$ einen **Untergraph** von $G = (V, E)$, wenn $V' \subseteq V$ und $E' \subseteq E$ gilt. Ist außerdem $E' = E \cap \binom{V'}{2}$, so wird G' als **induzierter Untergraph** von G bezeichnet.
Definition: Ein Graph $G = (V, E)$ wird **bipartit** genannt, wenn er Untergraph eines vollständigen, bipartiten Graphen ist. Etwas anschaulicher kann man formulieren, daß ein Graph genau dann bipartit ist, wenn man die Knoten so mit zwei Farben einfärben kann, daß keine gleichfarbigen Ecken benachbart sind.

Definition: Das **Komplement** eines Graphen $G = (V, E)$ ist der Graph
$\overline{G} = (V, \binom{V}{2} \setminus E)$.

Definition: Eine Folge von paarweise verschiedenen Ecken $v_1, v_2, \ldots, v_k$ eines Graphen $G = (V, E)$ repräsentiert einen **Weg der Länge** $k - 1$, falls $\{v_i, v_{i+1}\} \in E$ für alle $1 \leq i < k$. Ist außerdem $\{v_k, v_1\} \in E$, so repräsentiert die Folge auch einen **Kreis der Länge** k.
Man sagt, daß v von u erreichbar ist, falls ein Weg $v_1, v_2, \ldots, v_k$ mit $v_1 = u$ und $v_k = v$ in G existiert. Die Länge eines kürzesten Weges zwischen zwei Knoten nennt man ihren Abstand in G.

Lemma: Die Relation der Erreichbarkeit in der Knotenmenge V eines Graphen ist eine Äquivalenzrelation, d.h. sie ist reflexiv, symmetrisch und transitiv.

Definition: Die Knotenmenge V wird durch die Erreichbarkeitsrelation in Äquivalenzklassen zerlegt, wobei zwei Knoten u, v genau dann in derselben Klasse liegen wenn v von u erreichbar ist. Diese Klassen nennt man die **Zusammenhangskomponenten** (kurz **Komponenten**) des Graphen. G wird **zusammenhängend** genannt, wenn er genau eine Komponente hat.

Stellt man G durch eine Zeichnung dar, so kann man diesen Begriff anschaulich erklären: Zwei Knoten gehören zur selben Komponente, wenn man sie durch einen geschlossenen Kantenzug verbinden kann, wobei die Kanten nur in Knoten und nicht auf Kantenschnitten in der Zeichnung gewechselt werden dürfen.

Satz: Sind zwei Graphen isomorph so sind jeweils die Ordnung, die Größe, die sortierten Gradfolgen und die Anzahl der Komponenten der beiden Graphen gleich.

Definition: Seien u, v Knoten in einem ungerichteten Graphen $G = (V, E)$. Sind u und v in einer gemeinsamen Zusammenhangskomponente von G, so definieren wir ihren **Abstand** $d(u, v)$ als Länge eines kürzesten Weges (Anzahl der Kanten des Wegs) von u nach v. Gehören sie zu verschiedenen Komponenten, so setzen wir $d(u, v) = \infty$.

Definition: Der **Durchmesser** $D(G)$ des Graphen ist definiert als das Maximum über alle paarweisen Abstände zwischen Knoten.

Satz: Ein Graph ist genau dann bipartit, wenn alle in ihm als Untergraph enthaltenen Kreise gerade Länge haben.

Beweis: Zunächst überlegt man sich, dass wir den Graphen als zusammenhängend voraussetzen können, ansonsten führt man den folgenden Beweis für jede Zusammenhangskomponente.

Sei $G = (V, E)$ bipartit, das heißt, $V = A \cup B$ mit $A \cap B = \emptyset$ und Kanten verlaufen nur zwischen Knoten aus A und Knoten aus B. Sei des weiteren C ein Kreis in G. C benutzt abwechselnd Knoten aus A und B und hat somit gerade Länge.

Wir zeigen die andere Richtung. Wir fixieren einen beliebigen Knoten $u \in V$. Wir definieren: $A = \{v \in V | \ d(u, v) \text{ gerade }\}$, $B = V \setminus A$ Zu zeigen, es gibt keine Kanten zwischen Knoten aus A (bzw. aus B). Wir führen einen indirekten Beweis:

Wir nehmen an, es gibt eine Kante $\{v, w\}, v, w \in B$ (für A analog) und finden einen Widerspruch zur Annahme, dass alle Kreise gerade Länge haben. Wir betrachten kürzeste Wege von u zu v und zu w. Diese Wege haben gleiche Länge! (wegen der Kante zwischen v und w) Sei x der letzte gemeinsame Knoten auf beiden Wegen. Dann bilden die beiden Wegabschnitte von x nach v bzw. nach w zusammen mit der Kante $\{v, w\}$ einen Kreis ungerader Länge.

Streuspeicherverfahren (Hash-Tabellen)

In diesem Abschnitt lernen wir eine weitere Datenstruktur zur Implementierung von Wörterbüchern kennen. Sie ist anwendbar, wenn die Schlüssel bereits natürliche Zahlen sind oder als solche interpretiert werden können. Diese Struktur ist sehr einfach, hat aber den Nachteil, dass alle Operationen (im schlechtesten Fall) $O(n)$ Zeit kosten. Dafür ist die erwartete Laufzeit konstant und deshalb ist diese Datenstruktur in der Praxis sehr wichtig.
Die Idee besteht darin, alle Einträge (Items) anhand ihres Schlüssels auf N Fächer (engl. bucket - Eimer) zu verteilen. Die Fächer werden in einem Feld A der Größe N (**Bucket Array**) verwaltet. Jedes Fach muss den Zugriff auf alle darin enthaltenen Einträge sicherstellen, z.B. durch eine verkettete Liste.

Die eigentliche Verteilung geschieht durch eine **Hash-Funktion** h, die jedem Schlüssel k eine Zahl $h(k) \in \{0, 1, \ldots, N-1\}$ zuordnet. Werden zwei Einträge durch h in das gleiche Fach gelegt, spricht man von einer **Kollision**. Gute Hash-Funktionen zeichnen sich durch weitgehende Kollisionsvermeidung aus.
Häufig sind die Schlüssel noch nicht als ganze Zahlen gegeben. In solchen Fällen bietet es sich an, eine Hash-Funktion in zwei Stufen zu definieren. Zuerst verwendet man einen **Hash-Code**, der jedem Schüssel eine ganze Zahl zuordnet und im zweiten Schritt wir diese ganze Zahl durch eine **Kompressionsabbildung** in das Intervall $[0, N-1]$ abgebildet. Die Klasse `Object` in Java hat eine Methode `hashCode()`, die einen `int`-Wert (32 Bit) zurückgibt. Da sich dieser Wert in der Regel auf die Lage des Objekts im Speicher bezieht, kann es im Einzelfall zu ungewollten Effekten kommen (z.B. wenn gleiche Strings in verschiedene Fächer gelegt werden).

Hash-Codes
Für Strings (und andere Objekte, die man in einen String umwandeln kann) gibt es zwei Standardmethoden zur Codierung. Zuerst werden bei beiden Methoden alle Zeichen des Strings in `int`-Werte $x_0, x_1, \ldots x_{k-1}$ umgewandelt.
1) Die **Summenmethode** berechnet $x_0 + x_1 + \ldots + x_{k-1}$ als Hash-Code des Strings (unter Verwendung der `int`-Addition). Eine Änderung des Strings an einer Stelle bewirkt damit auch eine Veränderung des Codes, aber diese Methode hat den wesentlichen Nachteil, dass der Code sich bei Buchstabenvertauschung nicht ändert, z.B. haben die Strings `stop` und `post` die gleiche Codierung.
2) Dieser Effekt kann durch die **Polynommethode** umgangen werden. Man fixiert dazu einen `int`-Wert $a \notin \{0, 1\}$ und codiert den String durch die Auswertung des Polynoms $x_0 a^{k-1} + x_1 a^{k-2} + \ldots + x_{k-2} a + x_{k-1}$.

Kompressionsabbildungen
Sei n die Anzahl der Objekte, die in einer Hash-Tabelle mit N Buckets gespeichert sind. Der Wert $\lceil \frac{n}{N} \rceil$ wird als **Ladefaktor** der Tabelle bezeichnet. Mindestens ein Bucket muss diese Anzahl von Einträgen enthalten. Um Kollisionen zu vermeiden, muss der Ladefaktor 1, also $N \geq n$ sein. Der erste Schritt zur Kollisionsvermeidung besteht in der Auswahl einer geeigneten Kompressionsfunktion. Wir gehen jetzt davon aus, dass

der Schlüssel k bereits ein `int`-Wert ist.
Die einfachste Variante einer Kompressionsfunktion ist die sogenannte **Divsionsmethode**, definiert durch $h(k) = k \; mod \; N$. Bei einer zufälligen Schlüsselmenge kann man N beliebig festlegen. Da reale Daten aber oft gewisse Regularitäten aufweisen, ist es besser, für N eine Primzahl zu verwenden.
In vielen Anwendungen hat sich gezeigt, dass die folgende **MAD-Methode** (multiply, add and divide) noch bessere Ergebnisse liefert: $h(k) = (a \cdot k + b) \; mod \; N$, wobei N wieder eine Primzahl und a eine zu N teilerfremde Zahl ist.

Kollisionsbehandlung
Da man Kollisionen nie vollkommen vermeiden kann, muss man Strategien für diesen Fall entwickeln. Man kann zwei Grundstrategien unterscheiden:

1) Die Fächer (Buckets) werden so eingerichtet, dass sie mehrere Objekte halten können, z.B. als dopelt verkettete Liste. Im schlechtesten (aber sehr unwahrscheinlichen) Fall hat man dann eine einzige Liste mit allen Einträgen.

2) Man versucht beim Auftreten einer Kollision das neue Element in ein anderes Fach zu legen und sondiert dazu freie Fächer. Der größte Nachteil von Sondierungsstrategien liegt im hohen Aufwand für Updates nach Löschoperationen.

2.a) **Lineares Sondieren:** Ist $h(k)$ schon belegt, sondiert man die Buckets $h(k) + 1, h(k) + 2, h(k) + 3$ u.s.w. und wählt das erste freie aus. Diese Methode hat den zusätzlichen Nachteil, dass sich sehr schnell voll belegte Abschnitte (Cluster) bilden, die schneller wachsen als die Belegung der unterbesetzten Regionen. Man kann das etwas ausgleichen, wenn man für ein festes $c \neq 1$ die Buckets $h(k) + c, h(k) + 2c, h(k) + 3c$ u.s.w. sondiert.

2.b) **Quadratisches Sondieren:** Diese Methode ist wesentlich besser geeignet, die Bildung großer Cluster zu verhindern oder zumindest zu verzögern. Ist $h(k)$ schon belegt, sondiert man die Buckets $h(k) + 1^2, h(k) + 2^2, h(k) + 3^2$ u.s.w. und wählt das erste freie aus.

Skipt zur Vl Informatik B, 14./16.07.2004

1.1 Spezielle Graphklassen

Wir lernen zunčhst im Folgenden einige wichtige, weil zumindestens in der Theorie häufig auftretende Graphen kennen.

1. Der *vollständige Graph* K_n, $n \geq 1$ eine natürliche Zahl, besteht aus n Knoten und allen möglichen $\binom{n}{2}$ Kanten.
   Jeder Graph ist natürlich Untergraph eines vollständigen Graphen.

2. Der *vollständige bipartite Graph* $K_{n,m}$, mit $n, m \geq 1$, besteht aus $n + m$ Knoten und allen $n \cdot m$ Kanten, die jeweils einen der n Knoten mit einem der m Knoten verbinden.
   Untergraphen eines vollständigen bipartiten Graphen heißen *bipartit*.

3. Der *Hyperwürfel* Q_n hat als Knoten alle 0-1-Folgen der Länge n, zwei Folgen werden durch eine Kante verbunden, wenn sie sich genau an einer Stelle unterscheiden, also Hamming–Abstand 1 haben.
   Der Q_n hat 2^n Knoten und $n \cdot 2^{n-1}$ Kanten, was sofort aus dem Handschlag–Lemma folgt.

4. Der *Weg* P_n, $n \geq 0$ besteht aus $n + 1$ Knoten $v_1, \ldots, v_{n+1}$ und den n Kanten $\{v_i, v_{i+1}\}$ für $1 \leq i \leq n$.

5. Der *Kreis* C_n, $n \geq 3$, entsteht aus dem Weg P_{n-1} durch das Hinzufügen der Kante $\{v_1, v_n\}$.

6. Ein Graph heißt *k-regulär*, wenn alle Knoten den Grad k haben.
   Alle Kreise sind also 2–regulär, der K_n ist $(n - 1)$–regulär und der Q_n ist n–regulär.

7. *Bäume*, wie wir sie schon in Inf A zumindestens als gewurzelte Bäume zur Genüge kennen gelernt haben, sind die in der Informatik am häufigsten auftretenden Graphen.

1.2 Charakterisierung bipartiter Graphen

Es erweist sich in vielen Fällen als nützlich, mehrere äquivalente Charakterisierungen ein und derselben Graphklasse zu haben. Im Falle der bipartiten Graphen, die wir als Untergraphen der vollständigen bipartiten Graphen eingeführt hatten, liefert dies der folgende Satz.

Satz: Ein Graph ist genau dann bipartit, wenn alle in ihm als Untergraph enthaltenen Kreise gerade Länge haben.

Beweis: Zunächst überlegt man sich, dass wir den Graphen als zusammenhängend voraussetzen können, ansonsten führt man den folgenden Beweis für jede Zusammenhangskomponente.
Sei $G = (V, E)$ bipartit, das heißt, $V = A \cup B$ mit $A \cap B = \emptyset$ und Kanten verlaufen nur zwischen Knoten aus A und Knoten aus B. Sei des weiteren C ein Kreis in G. C benutzt abwechselnd Knoten aus A und B und hat somit gerade Länge.

Wir zeigen die andere Richtung. Wir fixieren einen beliebigen Knoten $u \in V$. Wir definieren:

$$A = \{v \in V | \ d(u,v) \ \text{ gerade } \}, B = V \setminus A$$

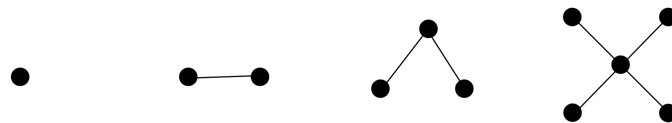
Zu zeigen, es gibt keine Kanten zwischen Knoten aus A (bzw. aus B). Wir führen einen indirekten Beweis.
Wir nehmen an, es gibt eine Kante $\{v, w\}, v, w \in B$ (für A analog) und finden einen Widerspruch zur Annahme, dass alle Kreise gerade Länge haben.
Wir betrachten kürzeste Wege von u zu v und zu w. Diese Wege haben gleiche Länge! (wegen der Kante zwischen v und w) Sei x der letzte gemeinsame Knoten auf beiden Wegen. Dann bilden die beiden Wegabschnitte von x nach v bzw. nach w zusammen mit der Kante $\{v, w\}$ einen Kreis ungerader Länge.

Man beachte, dass der Satz insbesondere gilt für Graphen, die gar keine Kreise besitzen, also Bäume und Wälder.

1.3 Bäume und ihre Charakterisierung

Definition: Ein zusammenhängender ungerichteter Graph ist ein *Baum*, wenn er keinen Kreis enthält.
Ein ungerichteter Graph, dessen Zusammenhangskomponenten Bäume sind, heißt *Wald*.

Beispiel:



Baumbeispiele

Definition: Sei $G = (V, E)$ ungerichtet und zusammenhängend mit $|V| = n$. Ein Untergraph T auf allen n Knoten, der ein Baum ist, heißt *aufspannender Baum*

von G.

Analog definiert man *aufspannende Wälder* für nicht zusammenhängende Graphen.

Beobachtung: Jeder zusammenhängende Graph hat einen aufspannenden Baum, dieser ist aber nur eindeutig, wenn der Graph selbst ein Baum ist. Dann ist T der Graph selbst.

Satz: Folgende Aussagen sind äquivalent für $G = (V, E)$.

(1) $G = (V, E)$ ist ein Baum.

(2) Je zwei Knoten sind durch genau einen Weg verbunden.

(3) G ist zusammenhängend und es gilt: $|E| = |V| - 1$.

Beweis:

(1) $\Rightarrow$ (2) und (2) $\Rightarrow$ (1) sind einfache indirekte Schlüsse.

Wir zeigen (1) $\Rightarrow$ (3):
Zunächst hat jeder Baum Knoten vom Grad 1, diese nennt man *Blätter*. Das sieht man wie folgt. Seien $u_1, u_2, \ldots, u_i$ die Knoten eines längsten Weges in G. Alle Nachbarn von u_1 liegen auf diesem Weg, sonst wäre er nicht längster Weg. Nur u_2 kann Nachbar sein, sonst gäbe es einen Kreis.
Wir entfernen u_1 und die Kante $\{u_1, u_2\}$ aus G und erhalten einen zusammenhängenden Restgraphen G'. Dieser hat genau einen Knoten und eine Kante weniger als G. Wenn wir dies iterieren, bleibt zum Schluss genau ein Knoten ohne Kanten übrig. Also $|E| = |V| - 1$.

(3) $\Rightarrow$ (1):
Sei $T = (V, E')$ aufspannender Baum von G, damit $|V| - |E'| = 1$. Aber nach Vorraussetzung gilt auch $|V| - |E| = 1$. Allerdings ist $E' \subseteq E$ und die einzige Möglichkeit hierfür ist $E = E'$.

2. Grundlegende graphentheoretische Algorithmen

Neben der Untersuchung struktureller Eigenschaften von Graphklassen ist es für praktische Anwendungen eine zentrale Aufgabe, möglichst effiziente algorithmische Lösungen zu finden zur Bestimmung graphentheoretischer Parameter und Eigenschaften.

Wie bestimmt man den Abstand zwischen Knoten eines Graphen, wie testet man, ob er zusammenhängend ist und welche Datenstrukturen eignen sich dafür?

2.1 Graphdurchmustern: Breitensuche und Tiefensuche

Im Folgenden wollen wir Graphen systematisch von einem Startknoten aus durchmustern, das heißt, alle Knoten und Kanten 'anschauen'.
Sei $G = (V, E)$ ein ungerichteter (oder gerichteter) Graph gegeben durch seine Adjazenzlistendarstellung.

2.1.1 Breitensuche BFS

Die Breitensuche (breadth first search) startet in $s \in V$. Wir schauen uns zuerst alle Nachbarn von s an, danach die Nachbarn der Nachbarn usw. bis wir alle Knoten und Kanten erreicht haben.

Wir geben den Knoten 'Farben', diese symbolisieren ihren aktuellen Zustand:

- weiß: Knoten wurde noch nicht gesehen; zu Beginn sind alle Knoten weiß
- grau: Knoten wurde schon gesehen, wir müssen aber noch überprüfen, ob er noch weiße Nachbarn hat
- schwarz: Knoten ist erledigt, Knoten selbst und alle seine Nachbarn wurden gesehen.

Die noch zu untersuchenden Knoten werden in einer Warteschlange Q verwaltet. Knoten, deren Farbe von weiß nach grau wechselt, werden ans Ende der Schlange eingefügt. Die Schlange wird vom Kopf her abgearbeitet. Ist die Schlange leer, sind alle von s erreichbaren Knoten erledigt.

Mit BFS kann der Abstand $d[u] = d(s, u)$ eines erreichbaren Knotens u von s berechnet werden (Beweis später) und gleichzeitig wird ein Baum von kürzesten Wegen von s zu allen erreichbaren Knoten aufgebaut. Dieser ist dadurch beschrieben, dass man für jeden Knoten einen Zeiger $\pi[u]$ auf den Vorgängerknoten auf einem kürzesten Weg von s nach u aufrechterhält. Ist dieser noch nicht bekannt oder existiert gar nicht, so ist der Zeiger auf NIL gesetzt.

Breitensuche BFS(G, s):

```
01 for jede Ecke u ∈V(G)\{s}
02      do Farbe[u] ← weiß
03         d[u] ← ∞
04         π[u] ← NIL
05 Farbe[s] ← grau
06 d[s] ← 0
07 π[s] ← nil
08 Q ← {s}
09 while Q≠ ∅
10         do u ← Kopf[Q]
11            for jeden Nachbarn v ∈ Adj[u]
12                do if Farbe[v] = weiß
13                      then Farbe[v] ← grau
14                           d[v] ← d[u]+1
15                           π[v] ← u
16                           Setze v ans Ende von Q
17            Entferne Kopf aus Q
18            Farbe[u] = schwarz
```

Die Komplexität des BFS–Algorithmus ist offensichtlich $O(|V|+|E|)$. Man beachte, dass der konstruierte Baum kürzester Wege von der Reihenfolge der Knoten in den Adjazenzlisten abhängt, der Abstand der Knoten natürlich nicht.

2.1.2. Tiefensuche DFS

Die Idee der *Tiefensuche* (depth first search) ist einfach. Hat ein Knoten, den man besucht, mehrere unentdeckte Nachbarn, so geht man zum ersten Nachbarn und von dort in die 'Tiefe' zu einem noch unentdeckten Nachbarn des Nachbarn, falls es ihn gibt. Das macht man rekursiv, bis man nicht mehr in die Tiefe gehen kann. Dann geht man solange zurück, bis wieder eine Kante in die Tiefe geht, bzw. alles Erreichbare besucht wurde.

Die Farben haben wieder dieselbe Bedeutung wie beim BFS, ebenso die π–Zeiger, die die entstehende Baumstruktur implizit speichern. Der DFS berechnet nicht die Abstände vom Startknoten.

Man kann aber jedem Knoten ein Zeitintervall zuordnen, indem er 'aktiv' ist, was für verschiedene Anwendungen interessant ist. Dazu läßt man eine globale Uhr mitlaufen, und merkt sich für jeden Knoten u den Zeitpunkt $d[u]$, bei dem u

entdeckt wird und den Zeitpunkt $f[u]$, bei dem u erledigt ist, da der Algorithmus alles in der 'Tiefe' unter u gesehen hat. Daher gilt für beliebige zwei Knoten im Graphen, dass entweder eines der Zeitintervalle voll im anderen enthalten ist, oder beide disjunkt sind.

Die geeignete Datenstruktur, um DFS zu implementieren, ist ein *stack (Kellerspeicher)*.
Ist der Graph in Adjazenzlistenform gegeben, so läuft DFS ebenfalls in Zeit $O(|V| + |E|)$.

Wie beim BFS hängen die entstehenden DFS–Bäume von der Reihenfolge in den Adjazenzlisten ab.

DFS(G)

01 **for** *jede Ecke u* $\in$ *V(G)*
02 **do** *Farbe[u]* $\leftarrow$ *weiß*
03 π*[u]* $\leftarrow$ *NIL*
04 Zeit $\leftarrow$ *0*
05 **for** *jede Ecke u*$\in$*V(G)*
06 **do if** *Farbe[u]* $=$ *weiß*
07 *DFS-visit(u)*

DFS-visit(u)

01 Farbe[u] $\leftarrow$ *grau*
02 d[u] $\leftarrow$ *Zeit* $\leftarrow$ *Zeit+1*
03 **for** *jede Ecke v* $\in$ *Adj[u]*
04 **do if** *Farbe[v]* $=$ *weiß*
05 **then** π*[v]* $=$ *u*
06 *DFS-visit(v)*
07 Farbe[u] $=$ *schwarz*
08 f[u] $\leftarrow$ *Zeit* $\leftarrow$ *Zeit+1*

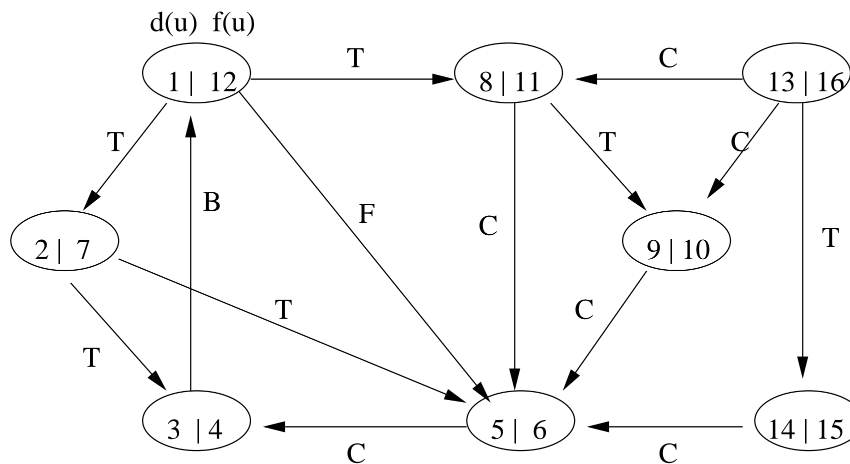
Der Stack ist eine Datenstruktur, die das LIFO–Prinzip (*last–in–first–out* umsetzt. Das heißt, die zu speichernden Daten (hier die grau werdenden Knoten) sind linear angeordnet und man kann ein neuen Eintrag nur als "oberstes" Element in den Stack einfügen (mit einer *push*–Operation) bzw. das oberste Element entfernen (mit einer *pop*–Operation, also beim DFS wenn der Knoten schwarz wird). Als weitere Funktionalität bietet ein Stack die Abfrage nach seiner Größe (*size*–Operation), die Boolesche Anfrage *isEmpty* und die Ausgabe des obersten Eintrages (*top*–Operation, ohne den Eintrag zu entfernen).

Wir werden verschiedene Realisierungen einer Stack–Datenstruktur noch kennen lernen und in Java implementieren.

Man kann die Kanten eines Graphen nach ihrer Rolle bei einer DFS–Durchmusterung klassifizieren. Wir unterscheiden:

- Tree-Kanten (T–Kanten): von grauen nach weißen Knoten
- Back–Kanten (B–Kanten): von grau nach grau
- Forward–Kanten (F–Kanten): von grau nach schwarz und zwar von Vorfahren zu Nachkommen im DFS–Baum
- Cross–Kanten (C–Kanten): alle restlichen Graphkanten

Im folgenden Beispiel sind die Zeitintervalle und die Kantenklassifikation illustriert:



DFS–Suche. Zeitintervalle der Knoten und Kantenklassifikation

Gerichtete azyklische Graphen:
Gerichtete azyklische Graphen (dag's) sind gerichtete Graphen ohne gerichtete Kreise. Diese spielen in der Informatik an verschiedenster Stelle eine Rolle, zum Beispiel bei Vererbungshierarchien in Java. Oder man stelle sich eine Menge von Jobs vor die linear zu ordnen sind. Dabei gibt es Einschränkungen derart, dass ein Job a vor einem anderen Job b bearbeitet werden muß. Gesucht ist eine lineare Ordnung (eine *topologische Sortierung*), die alle diese Constraints berücksichtigt.
Definition: Eine topologische Sortierung eines gerichteten Graphen ist eine Nummerierung seiner Knoten derart, dass aus $(u, v) \in E$ folgt $u \leq v$ in der Nummerierung.

Offensichtlich muss der gerichtete Graph ein dag sein, um eine topologische Sortierung zuzulassen. Das es dann aber immer geht, überlegt man sich wie folgt.

Fakt: Ein gerichteter Graph ist ein dag genau dann, wenn ein DFS–Durchmustern keine Back–Kanten produziert.

Basierend auf dieser Einsicht kann man topologisches Sortieren wie folgt realisieren: Führe ein DFS für den dag durch; wenn immer ein Knoten schwarz wird, gib ihn aus.

Behauptung: Dies ergibt ein topologisch invers sortierte Knotenfolge.

Beweis: Zu zeigen, wenn $(u, v) \in E$, dann ist $f(v) \leq f(u)$. Wir betrachten den Moment, wenn die Kante (u, v) vom DFS untersucht wird. Zu diesem Zeitpunkt ist u grau. v kann nicht grau sein wegen obigen Fakts. Also bestehen nur die beiden Möglichkeiten, v ist weiß oder schwarz. Aber in beiden Fällen ist offensichtlich $f(v)$ kleiner als $f(u)$, denn Knoten die unter u liegen, sind "eher" fertig als u.

2.2 Minimal aufspannende Bäume

Sei $G = (V, E)$ ein ungerichteter zusammenhängender Graph und w eine Gewichtsfunktion, die jeder Kante eine reelle Zahl zuordnet, dies könnte zum Beispiel deren Länge sein.
Wir wissen, G hat einen aufspannenden Baum. Sei dies $T = (V, E')$. Wir definieren

$$w(T) = \sum_{e \in E'} w(e)$$

.

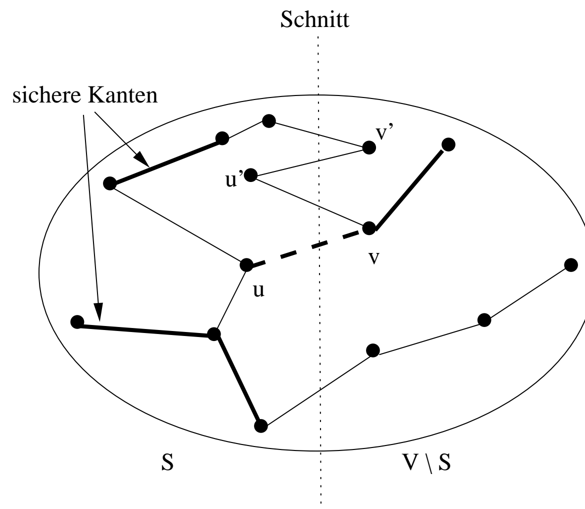
Aufgabe: Finde einen aufspannenden Baum mit minimalem Gesamtgewicht!

Offensichtlich hat jeder bewichtete zusammenhängende Graph einen minimal aufspannenden Baum (MST) und im Allgemeinen muss der auch nicht eindeutig sein. Wir werden zuerst einen generischen MST–Algorithmus kennenlernen und danach zwei konkrete Umsetzungen.

Definition: Ein *Schnitt* von G ist eine Zerlegung $(S, V \setminus S)$ seiner Knotenmenge. Eine Kantenmenge A *respektiert* einen Schnitt $(S, V \setminus S)$, falls keine Kante aus A einen Knoten aus S mit einem aus $V \setminus S$ verbindet.
Eine Kantenmenge A heißt *sicher*, wenn es einen MST T von G gibt, so dass A in der Kantenmenge von T enthalten ist.

Satz: (Generischer MST–Algorithmus)
Sei G ein gewichteter ungerichteter zusammenhängender Graph und sei A eine sichere Menge von Kanten, die einen Schnitt $(S, V \setminus S)$ respektiert. Wir betrachten eine leichteste Kante uv, mit $u \in S$, $v \in V \setminus S$. Dann ist $A \cup \{uv\}$ sicher.

Beweis: Sei T ein MST, der A enthält. Wir nehmen an, dass uv nicht zu T gehört, ansonsten ist nichts zu beweisen.
Wenn wir die Kante uv zu T hinzunehmen, entsteht genau ein Kreis C. Wir betrachten in C alle Kanten xy mit $x \in S, y \in V \setminus S$. Außer uv muss es wenigstens noch eine weitere solche Kante geben. Sei dies $u'v'$.
Streichen wir $u'v'$ aus $T \cup \{uv\}$, so entsteht ein aufspannender Baum T'. Da nach Annahme $w(uv) \leq w(u'v')$ und T ein MST ist, folgt sofort, T' ist MST und mithin ist $A \cup \{uv\}$ sicher.



Der MST T und die Kante uv definieren Kreis!

Der Satz kann algorithmisch umgesetzt werden. Man startet mit A leer, sucht sich einen Schnitt, der von A respektiert wird (dies sind am Anfang alle), nimmt die leichteste Kante über den Schnitt hinzu usw.

Die beiden MST–Algorithmen von Prim und Kruskal sind konkrete Umsetzungen davon.

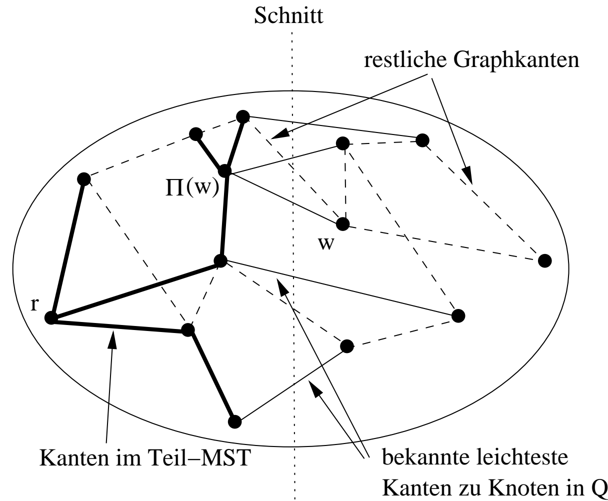
2.2.1 Der MST–Algorithmus von Prim

Dieser ist konzeptionell sehr einfach. Wir lassen den MST–Baum von einem beliebigen Startknoten r aus 'wachsen'. Die Frage ist, um welche Kante die Teillösung in einem Schritt erweitert wird. Wir schauen uns den Schnitt an, der die Teilösung vom Rest trennt. In einer Prioritätsschlange verwalten wir alle noch nicht erreichten Knoten w zusammen mit einem Schlüssel, der angibt, was im Moment das Gewicht einer leichtesten Kante ist, mittels derer w von einem der bereits in den Teilbaum aufgenommenen Knoten aus erreichbar ist.
Diese Schlüssel benötigen ggf. entsprechende Updates. Die π–Zeiger speichern wieder den aktuell gefundenen Teil–MST mit Wurzel r sowie die bekannten kürzesten Kanten zu Knoten in Q.

MST-Prim(G, w, r)

01 $Q \leftarrow$ *V(G)*
02 **for** *jedes* $u \in Q$
03   *key[u]* $\leftarrow \infty$
04 *key[r]* $\leftarrow$ *0*
05 π*[r]* $\leftarrow$ *NIL*
06 **while** $Q \neq \emptyset$
07   **do** $u \leftarrow$*Extract-Min(Q)*
08   **for** *jedes* $v \in Adj$*[u]*
09   **do if** $v \in Q$ *und w(u,v)* $<$ *key[v]*
10   **then** π*[v]* $\leftarrow$ *u*
11   *key[v]* $\leftarrow$ *w(u,v)*

Die Komplexität des Algorithmus hängt ab von der konkreten Realisierung des abstrakten Datentyps Prioritätsschlange ab, wir werden einige kennen lernen. Benutzt man für die Implementierung der Prioritätswarteschlange einen binären Heap , so ist die Komplexität $O(|V| \log |V| + |E| \log |V|) = O(|E| \log |V|)$.



2.2.2 Der MST–Algorithmus von Kruskal

Zuerst werden die Kanten nach aufsteigenden Gewichten sortiert. Danach wird in dieser Reihenfolge, mit der leichtesten Kante beginnend, getestet, ob eine Kante, einen Kreis im bisher konstruierten Wald schließt (falls ja wird die Kante verworfen). Falls nein wird die sichere Menge um diese Kante erweitert. Anders gesagt, man prüft, ob es für die Kanten jeweils einen Schnitt gibt, für den die Kante die leichteste ist. Wenn die Kante einen Kreis schließt, so gibt es einen solchen Schnitt nicht, denn die Zusammenhangskomponenten der sicheren Kanten liegen auf einer

Seite des Schnittes (respektieren ihn).

Das Interessanteste dabei ist die verwendete Datenstruktur, eine sogenannte Union-Find–Struktur zur Verwaltung von Partitionen einer Menge, hier der Knotenmenge V.

Die von der Datenstruktur unterstützten Operationen sind:

- MakeSet(v): Aus dem Element $v \in V$ wird eine Menge gemacht.
- Union(u,v): Die beiden Mengen, zu denen u bzw. v gehören, werden vereinigt.
- FindSet(v): Bestimme die Menge, zu der v gehört.

Dazu stelle man sich vor, dass jede Menge einen Repräsentanten hat, auf den alle Elemente der Menge zeigen. Bei der Union–Operation müssen also Zeiger umgehangen werden. Eine einfache Realisierung besteht darin, bei Union zweier Mengen, die Zeiger der kleineren Menge auf den Repräsentanten der größeren umzuleiten.
Man überlegt sich, dass damit insgesamt nur höchstens $|V| \log |V|$ mal Zeiger umgeleitet werden.
Hier ist der Pseudocode für Kruskals Algorithmus.

MST-Kruskal(G, w)

```
01 A ← ∅
02 for jede Ecke v ∈ V(G)
03       do Make-Set(v)
04 Sortiere die Kanten aus E in nichtfallender Reihenfolge
   entsprechend ihres Gewichts
05 for jede Kante (u,v) in sortierter Reihenfolge
06       do if Find-Set(u) ≠ Find-Set(v)
07             then A ← A∪ {(u,v)}
08                   Union(u,v)
09 return A
```