Begleitblatt Algorithmik I

Grundbegriffe:

Signal := Mitteilung eines Senders an einen Empfänger unter Zuhilfenahme der zeitlichen Veränderung einer physikalischen Größe (analoges: kontinuierliche Veränderung; digitales: diskrete Werte (Quantisierung) zu diskreten Zeitpunkten (Abtastung)).
Nachricht := Abstraktion von Signalen sowie deren Speicherung. → Signale dienen der Wiedergabe und Weitergabe von Nachrichten.
Information := einer Nachricht zugeordneten Bedeutung (Interpretationsvorschrift).
Entscheidungsgehalt H eines Zeichenvorrates mit n Zeichen := kleinste Anzahl H von Entscheidungen, mit denen man feststellen kann, welches der n Zeichen vorliegt. [H] = 1 b(it). Für n > 2: $H = \lceil \log_2 n \rceil$
Datum := bedeutungstragende Nachricht mit standardisierter Interpretationsvorschrift.
System := Kollektion von Gegenständen (=Komponenten), die in einem inneren Zusammenhang stehen samt Beziehungen zwischen diesen Gegenständen. Systemzustand: Eigenschaften aller vorhandenen Komponenten und der Beziehungen zwischen diesen. Systemverhalten: zeitliche Abfolge von Zuständen.
Algorithmus := Verfahren zur Lösung einer Problemklasse mit einer präzisen, endlichen Beschreibung unter Verwendung effektiver Verarbeitungsschritte. Effektiv: in endlicher Zeit durchführbar. Effizient: gute Leistung, sparsamer Ressourcenverbrauch.
Strukturierung von Algorithmen: Elementare Operationen, Sequenz, Fallunterscheidung, Schleife, Unterprogrammaufruf, Parallele Ausführung (darstellbar z.B. mittels Flussdiagrammen).
Java (hier werden nur die wesentlichen Unterschiede zu C++ erwähnt)
Syntax von Java
Konstanten belegen: **final** <Typ> <name> = <wert>;
Bei Objektvariablen handelt es sich immer um Referenzen (Vorbelegung mit **null**). Daher immer mit **new** erst zuweisen (kein delete nötig!)
Attribute können bereits bei der Deklaration vorbesetzt werden (Konstruktoraufruf).
Vererbung: class Child **extends** parent mit **super** lässt sich auf Oberklasse zugreifen.
Reihe: <typ>[ ] <name> mehrdimensionale Reihen über Reihungen von Reihungen (da Referenzen)
Jeder primitive Typ besitzt bereits implementierte Grundoperationen
Wichtig: Beim Typ boolean werden folgende Operationen !, &, |, ^ strikt ausgewertet und folgende &&, || nur solange bis es sinn macht, im Sinne des Ergebnisses.
Bei Ganzzahltypen gibt es >>> -Operator der das Vorzeichen nicht mit shiftet.
In Java können keine Variablen überschrieben werden falls sie in einem Block schon einmal deklariert worden sind.
Interfaces dienen in Java lediglich als Schnittstelle für Unterklassen bzw. weiterer Interfaces (unter Interfaces ist eine eingeschränkte Mehrfachvererbung möglich; gibt es z.B. zwei gleiche Bezeichner (Attribute) in zwei geerbten Interfaces so wird beim Zugriff auf diese ein Fehler ausgegeben). Syntax:
Interface <specifier>
{ <list of method declarations> }
class <specifier> implements <interface>
{ <list of interface methods and further methods> }
Abstrakte Klassen können explizit angegeben werden:
abstract class <specifier>
{ abstract <method head> … }
Frühe und späte Bindung (wie c++ nur mit Referenzen) ist möglich.
Labels wie in C++ an Blockanfängen; Ausnahme aus inneren Schleifen kann man falls eine äußere Schleife mit einem Label versehen ist, mit *break <label>;* aus der äußeren herausspringen.
Statische Initialisierer um statische Variablen zu initialisieren besteht die Möglichkeit komplexeren Code zu verwenden. Syntax: class <name> { static <type> <var>
static { <var> = <value>; … }
static wird aufgerufen sobald die Klasse in die Runtimeumgebung geladen wird.

Rekursion

Eine Methode f heißt rekursiv genau dann wenn zur Berechnung von f diese Methode f wieder benutzt wird.
Verschränkte Rekursion: f wird indirekt über g welches wiederum in f aufgerufen wird, aufgerufen.
Rekursion als konstruktive Form der Induktion. Terminierung lässt sich of t mit Induktion beweisen.
Terminierungsbeweis:

Sei $e_0, e_1, e_2, \ldots$ die Argumentfolge der Rekursionsschritte bei Auswertung von f. So finde ganzzahlige Terminierungsfunktion $t(e_i)$ und beweise mit Induktion das: t bei jedem Rekursionsschritt streng monoton fällt und t nach unten beschränkt ist.
Entwurf von Algorithmen mit Induktion:

(1) Bestimme Lösung für einen Basisfall.
(2) Zerlege das Problem in ein, oder mehrere Teilprobleme die direkt oder rekursiv lösbar sind und verknüpfe die Lösung zur Gesamtlösung.

Teile und Herrsche (=Anwendung des Induktionsprinzip zur Algorithmenkonstruktion)

- Zerlegung des Problems in kleinere Teilprobleme (Lösung durch Rekursion oder direkt) meist 2 Problem halber Größe.
- Füge Teillösungen zur Gesamtlösung zusammen (Verschmelzung) mit möglichst wenig Aufwand.

Rekursionsformen:

- Lineare Rekursion: f wird in jedem Zweig einer Fallunterscheidung höchstens einmal aufgerufen.
- Repetitive Rekursion: Spezialfall der linearen, rekursive Aufruf stets als letzte Aktion in einem Zweig. (können leicht durch Schleifen ersetzt werden)
- Kaskadenartige Rekursion: in einem Zweig einer Fallunterscheidung treten zwei oder mehr rekursive Aufrufe von f auf (Baumartige Ablaufstruktur; sehr schnelles anwachsen der rekursiven Aufrufe)
- Verschachtelte Rekursion: rekursive Aufrufe treten zusätzlich bei Parameterausdrücken auf.
- Verschränkte Rekursion: Eine Methode f enthält einen Aufruf g welche wiederum einen Aufruf von f enthält.

Algorithmenherleitung durch Induktion

Nachweis der Gültigkeit einer Aussage A(n) für alle $n \in N$.

- Basisform (-1): A(1) ist wahr; für jedes n > 1: A(n-1) impliziert A(n).
- Basisform (+1):A(1) ist wahr; für jedes $n \geq 1$: A(n) impliziert A(n+1).
- k-Anfangsform der Induktion (-k), analog (+k): A(1)...A(k) sind wahr; für jedes n > k:A(n-k) impliziert A(n)
- strenge Induktion: A(1) ist wahr; für jedes n > 1: für alle m<n:A(m) impliziert A(n)
- Rückwärtsinduktion: A(n) ist wahr für alle n aus einer unendlichen Teilmenge von N; für jedes n > 2: A(n) impliziert A(n-1).

Dynamisches Programmieren (=Anwendung des Induktionsprinzip zur Algorithmenkonstruktion)

(1) Zerlegung des Problems in mehrere kleinere Probleme. Diese sind vorab gelöst und können aus einer Tabelle abgelesen werden.
(2) Füge Teillösungen zu einer Gesamtlösung zusammen.
(3) Tabelle mit optimalen Zwischenergebnissen wird iterativ aufgebaut:
(4) Der Wert einer Zelle ergibt sich durch Kombination aus früher berechneten Werten in früher vervollständigten Zeilen der Tabelle oder in der gleichen Zeile weiter links.

Gierige (greedy)-Algorithmen:

Hierbei handelt es sich um Algorithmen die im Moment der Entscheidung den am besten zu scheinenden Weg wählen.

- wende Bewertungsfunktion an um das zur Zeit am besten passende Element auszuwählen und füge es zur Ergebnismenge hinzu.
- Wiederhole bis Eingabemenge leer.

Abstrakte Datentypen

Definition: Abstrakter Datentyp (ADT) = Die Schnittstelle einer Komponente und ihr Verhalten (ein Typ) wird ohne Angabe einer konkreten Implementierung spezifiziert. (Festlegung der anwendbaren Operationen und deren Wirkung)

Die **Signatur** eines ADT legt fest, welche Operationen erlaubt sind, welche Typen die Parameter der Operationen haben müssen und von welchem Typ das Resultat ist.

Beispielsignatur ADT Boolean:

true:		→ Boolean
false:		→ Boolean
not:	Boolean	→ Boolean
and:	Boolean x Boolean	→ Boolean
or:	Boolean x Boolean	→Boolean

Axiome legen die Wirkung der Operationen fest, indem sie auf andere Funktionen (hauptsächlich auf Konstruktoren) zurückführt.

Axiome ADT Boolean:

A1	not(false) = true
A2	not(true) = false
A3	and(false,false) = false
…	

Operationen die Datenobjekte erzeugen heißen **Konstruktoren**.

Normalform eines Datenobjekts: konstruiert durch eine minimale Zahl von Konstruktoraufrufen. Übrige heißen Hilfskonstruktoren.

Operationen, die Informationen über einen Datentyp oder Datenobjekt liefern, heißen **Projektoren** bzw. **Selektoren**.

Elementare Datenstrukturen

Liste (T) Signatur

- create:		→ Liste
- append:	T x Liste	→ Liste
- head :	Liste	→ T
- tail:	Liste	→ Liste
- length :	Liste	→ N

Axiome:

A1:	head(append(x,l)) = x
A2 :	tail(append(x,l)) = l
A3:	length(create) = 0
A4 :	length(append(x,l)) = 1 + length(l)

weitere Datenstrukturen : siehe Ergänzung auf Seite 10.

Verifikation

Hinweis: In der Praxis muss man abwägen ob es sich lohnt eine aufwändige Verifikation durchzuführen.

Mit der Verifikation wird versucht implementierten Code formal auf Korrektheit zu überprüfen. Dazu müssten alle möglichen Eingabedaten betrachtet werden und überprüft werden dass immer die gewünschte Ausgabe erreicht wird. Dies erreicht man indem man die Eingabe / Ausgabewerte mittels Prädikaten beschreibt. Somit kann die Wirkung eines Codestücks durch Angaben von Prädikaten zu Beginn und Ende des Codes spezifiziert werden. **Zusicherungen** sind prädikatenlogische Aussagen über die Werte der Programmvariablen an den Stellen im Programm, an denen sie stehen. Dabei unterscheidet man Vorbedingungen (*pre-condition*) und Nachbedingungen (*post-condition*) die entsprechend vor oder nach einer Anweisungsfolge stehen. Somit bilden die Vor- und Nachbedingungen des gesamten Programms eine prädikatenlogische Spezifikation (nur für funktionales Verhalten). Dabei unterscheidet man stärkere und schwächere Vorbedingungen.

Verifikation von imperativen Programmen

- Verifikation prüft ob Zusicherungen erfüllt werden. (d.h. ob Nachbedingung Q einer Anweisung A erfüllt ist wenn diese im Falle einer erfüllten Vorbedingung P abgearbeitet worden ist)
- Partielle Korrektheit: Ausführung von A beginnt in einem Zustand der P genügt und die Ausführung nach endlich vielen Schritten terminiert und dann erfüllt der erreichte Zustand Q. Schreibweise: P {A} Q
- Totale Korrektheit: verlangt den Beweis der Terminierung (kann auch unmöglich sein). Schreibweise: {P} A {Q}
- Verifikationskalküle: verwenden Regelwerke zur Ableitung zu verifizierender Aussagen. Können **korrekt** sein: alles, was abgeleitet werden kann, ist wahr und können **vollständig** sein: alles, was wahr ist, kann abgeleitet werden.

Vorgehensweise: vorwärts und rückwärts Verifikation (geg. P und Programm → best. gute Nachbedingung → beweise Nachbedingung; geg. Q und Programm best. gute Vorbedingung und beweise diese).
Eine Zusicherung P heißt **schwächer** als P' wenn $P' \Rightarrow P$

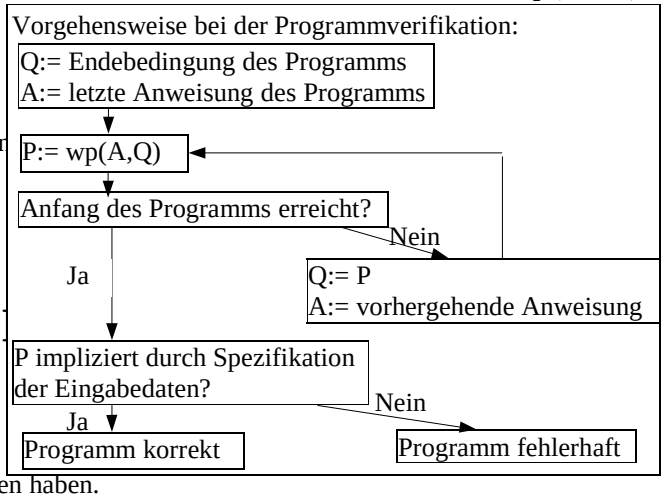
WP-Kalkül als Mittel der Rückwärtsanalyse

Idee: Untersuche statt vieler Vorbedingungen nur die schwächste Vorbedingung (weapest precondition), ausgehend von dieser wird die Nachbedingung gerade noch erfüllt.

Definition: Wenn Q ein Prädikat ist und A eine Anweisungsfolge, dann ist die schwächste Vorbedingung von A in Bezug auf Q, wp(A,Q), ein Prädikat, das die Menge aller Startzustände beschreibt, so dass wenn die Ausführung von A in einem dieser Zustände beginnt, dann endet (terminiert) die Ausführung in einem Zustand der Q genügt.

wp(Programm, Nachbedingung) = schwächste Vorbedingung wp enthält Terminierung!

Wenn also eine Vorbedingung P die Terminierung von A garantiert und wenn gilt {P} A {Q}, dann gilt: $P \Rightarrow \mathrm{wp}(A,Q)$ somit {wp(A,Q)} A {Q}



Regeln des WP-Kalküls:

1. Sequenzregel: sind Nachbedingung einer Anweisung und die Vorbedingung einer zweiten Anweisung gleich so können sie zu einem Programmstück zusammengesetzt werden

$\mathrm{wp}('s_1; s_2', R) = \mathrm{wp}(s_1, \mathrm{wp}(s_2, R))$

Wenn gilt: $Q \Rightarrow Q'$ so gilt auch: $\{P\}\, s_1 \{Q\}$ und $\{Q'\}\, s_2 \{R\}$ folgt: $\{P\}\, s_1; s_2 \{R\}$

2. Zuweisungsregel:

$\mathrm{wp}('x := e', Q) = zulässig(x) \wedge zulässig(e) \wedge Q[x/e]$

Zur Konstruktion der Vorbedingung wird mit der Nachbedingung Q begonnen, darin werden alle Vorkommen von x durch den Ausdruck e substituiert. (Die zulässig() Prädikate werden oft weggelassen).

Definitionen: wp(leer, Q) = Q vor leer muss also mind. Q gegolten haben.

wp(crash, Q) = false Es dürfen keine Fehler passieren. crash ist der Versuch eine Anweisung auszuführen deren Vorbedingung nicht erfüllt ist.

Rechenregeln:

wp(A, false) = false Das false nicht erfüllbar ist muss die Vorbedingung unerfüllbar sein.

Aus $P' \Rightarrow P$, $Q \Rightarrow Q'$ und {P} A {Q} folgt {P'} A {Q'} Verschärfung der Vorbedingung, abschwächung der Nachbedingung.

Montonie: $Q \Rightarrow R$ dann $\mathrm{wp}(A,Q) \Rightarrow \mathrm{wp}(A,R)$

Distributivitäten: $\mathrm{wp}(A,Q) \wedge \mathrm{wp}(A,R) = \mathrm{wp}(A, Q \wedge R)$ $\mathrm{wp}(A,Q) \vee \mathrm{wp}(A,R) \Rightarrow \mathrm{wp}(A, Q \vee R)$

(= nur bei deterministischen Programmen)

3. If-then-else Regel: $\mathrm{wp}('if\ b\ then\ s_1\ else\ s_2', Q) = [b \Rightarrow \mathrm{wp}(s_1, Q)] \wedge [\neg b \Rightarrow \mathrm{wp}(s_2, Q)]$ zus. zulässig(b) gelten und b muss seiteneffektfrei sein. Die Regel kann entsprechend umgeformt werden:

$\mathrm{wp}('if\ b\ then\ s_1\ else\ s_2', Q) = [b \Rightarrow \mathrm{wp}(s_1, Q)] \wedge [\neg b \Rightarrow \mathrm{wp}(s_2, Q)] = [b \wedge \mathrm{wp}(s_1, Q)] \vee [\neg b \wedge \mathrm{wp}(s_2, Q)]$

Fehlt der else-Teil so gilt: $\mathrm{wp}('if\ b\ then\ A', Q) = [b \Rightarrow \mathrm{wp}(A, Q)] \wedge [\neg b \Rightarrow Q]$ was äquivalent zu $\mathrm{wp}('if\ b\ then\ A\ else\ leer', Q) = [b \wedge \mathrm{wp}(A, Q)] \vee [\neg b \wedge Q]$ ist.

case-Regel: Hintereinanderschaltung von if-then-else Regeln.

4. Schleifeninvariante: (jede Schleife kann als while-Schleife geschrieben werden)

- Definition: $H_k(Q)$ schwächste Vorbedingung, die garantiert, dass, wenn die Schleife k-mal $(k \geq 0)$ durchlaufen wird, danach Q gilt. $H_k(Q) := \mathrm{wp}('while\ b\ do\ A', Q)$. Nach Umformungen gilt: $H(Q) = \mathrm{wp}('while\ b\ do\ A', Q) = H_0(Q) \wedge H_1(Q) \wedge ... = (\forall k \geq 0 : H_k(Q))$ H(Q) läss sich als Rekurrenz definieren: $H(Q) = (\forall k \geq 0 : H_k(Q))$; $H_0(Q) = (\neg b \Rightarrow Q)$; $H_{k+1}(Q) = (b \Rightarrow \mathrm{wp}(A, H_k(Q)))$ gesucht ist also ein H das diese Rekurrenz erfüllt, mit so einem H kann die totale Korrektheit (partielle Korrektheit und Terminierung) einer Schleife gezeigt werden.
- Schleifeninvariante (teile schwächste Vorbedingung H auf in eine Bedingung b, so dass bei $\neg b$ die Schleife terminiert und eine Schleifeninvariante I: ein Prädikat, das vor und nach jeder Iteration einer Schleife immer wahr ist)
  Definition: Eine **Invariante** der Schleife *while b do A* ist eine Bedingung I für die gilt: $\{I \wedge b\}\, A \{I\}$ wenn vor dem Schleifenrumpf $I \wedge b$ gilt, dann terminiert die Ausführung von A und danach ist I erfüllt. Es muss also gezeigt werden: $I \wedge b \Rightarrow \mathrm{wp}(A, I)$ daraus folgt das die Schleife terminiert: $\{I\}\, while\ b\ do\ A \{I \wedge \neg b\}$ (kennt man nun eine Invariante des Rumpfes so kann man die schwächste Vorbedingung der Schleife erschliessen. Es muss lediglich die Terminierung gezeigt werden)
  Konstruktives Vorgehen bei Schleifen:
  - Bestimme I und konstruiere Rumpf s. { I } while b do { $I \wedge b$ } s { I } { $I \wedge \neg b$ } { Q }
  - Nachweis der totalen Korrektheit:
    1. Beweise, dass I vor Eintritt der Schleife gilt.
    2. Beweise, dass { $I \wedge b$ } s { I }, dass also I tatsächlich eine Invariante ist.
    3. Beweise, dass $(I \wedge \neg b) \Rightarrow Q$, so dass bei Terminierung das gewünschte Ergebniss erreicht wird.
    4. Beweise, dass die Schleife terminiert.

Ausnahmebehandlung in Java

Sehr ähnlich dem Mechanismus in C++, daher werden nur die Unterschiede erwähnt.
Ausnahmetypen sind in Java Klassen die alle von der Klasse java.lang.Throwable erben. Diese hat genau zwei direkte Unterklassen:

- java.lang.Error zeigt Probleme bei der Programmausführung an, die das Programm nicht selber beheben kann.
- Java.lang.Exception ist die Oberklasse, unter der alle Ausnahmetypen zusammengefasst werden, die das Programm selbst behandeln möchte/sollte.

Wird eine geworfene Ausnahme nicht behandelt so führt sie zum Programmabbruch mit entsprechender Fehlerausgabe.
Neben den üblichen catch-Klauseln gibt es in Java eine **finally**-Klausel welche am Ende einer try-catch Anweisung stehen kann. Der Code in dieser Klausel wird in jedem Fall ausgeführt egal ob eine Ausnahme geworfen wird oder nicht.
Werden in einer Methode Ausnahmen geworfen so müssen diese in der Signatur der Methode bekannt gemacht werden (außer Ausnahmen vom Typ bzw. von abgeleiteten Typen der Klasse java.lang.RuntimeException müssen nicht bekannt gemacht werden). Syntax: <modifierList> <methodName>(<parameterList>) **throws** <ExceptionClass 1>,...,<ExceptionClass n>
Es genügt dabei auch nur einen Obertyp anzugeben.
Wird eine solche Methode überschrieben so kann diese:

- ganz auf die Deklaration des Ausnahmetyps verzichten.
- Ebenfalls eine Ausnahme vom gleichen Typ deklarieren.
- Eine Ausnahme von einem Untertyp des Typs deklarieren. Es kann kein neuer Typ hinzugefügt werden (auch kein Obertyp).

Standardmäßig können Objekte von java.lang.Exception schon eine Fehlernachricht aufnehmen.
Zusicherungen in Java: Syntax AssertStatement: *assert Expression1* oder *assert Expression1 : Expression2*
eine verletzte Zusicherung wirft eine eine Ausnahme vom Typ AssertionError. Bei der Assertion mit : wird Expression2 in das Ausnahmeobjekt geschrieben.
Testen von Programmen:

- **Black-Box Testing:** Testen aufgrund externer Beobachtung (es werden Daten reingesteckt und die Ausgabe mit der Spezifikation verglichen)
- **White-Box Testing:** Testen aufgrund interner Betrachtung (Eingabedaten werden unter Kenntnis des internen Ablaufs so gewählt das bestimmte Testkriterien erfüllt werden.
- **Kontrollflussgraph:** Anweisungen werden zu Knoten und Sprünge, Schleifen und Bedingungen induzieren Kanten.
  - **Anweisungsüberdeckung(statement coverage):** jeder Knoten soll beim Test einmal besucht werden. Grad Überdeckung = #besuchte Knoten / #aller Knoten. Aber niedrige Fehlerentdeckungsrate
  - **Zweigüberdeckung(branch coverage):** jede Kante soll beim Test mindestens einmal besucht werden.
  - **Pfadüberdeckung(all-paths coverage):** alle unterschiedlichen Pfade sollen beim Test mindestens einmal durchlaufen werden.
  - **Bedingungsüberdeckung(condition coverage):**
    - Einfache Bedingungsüberdeckung: Alle atomaren Prädikate in Bedingungen sollen beim Test mindestens einmal wahr oder falsch werden (schwächer als Zweigüberdeckung).
    - Mehrfach-Bedingungsüberdeckung: Bei zusammengesetzten Prädikaten kommen alle Kombinationen von wahr und falsch der atomaren Prädikate vor (exp. Aufwand)
    - Minimale Mehrfach-Bedingungsüberdeckung: jedes Prädikat wird mind. einmal wahr oder falsch.
  - **Definitions-und Nutzungsgraph:** (entspricht Kontrollflussgraph) jede Wertzuweisung an eine Variable wird beim Knoten vermerkt: Definition („definition“, „def“). Jeder Variablenzugriff in einem Ausdruck wird beim Knoten vermerkt c-Nutzung („computational use“, „c-use“). Jeder Variablenzugriff in einem Prädikat wird an Ausgangskanten vermerkt p-Nutzung („predicate-use“, „p-use“)

Charakterisierung von Aufwänden

Umfang n: Anzahl der Eingabewerte
Aufwand T(n): Anzahl der Zeiteinheiten die der Algorithmus für Problem mit Umfang n benötigt.
Weiterhin unterscheidet man: ungünstigster Aufwand („worst case“), mittlerer Aufwand („average case“) und Aufwand im besten Fall („best case“)
Für die Theorie sind lineare Faktoren uninteressant, es interessiert vor allem der Aufwand für sehr große Umfänge.
O-Kalkül (Edmund Landau)
Funktionen die den Aufwand berechnen werden in Funktionsklassen eingeteilt.
Def.: Seien $c, n, n_0 \in \mathbb{N}$ so ist Groß-O: $O(f(n)) = \{ g(n) | \exists c > 0 \ \exists n_0 > 0 \ \forall n \geq n_0 : 0 \leq g(n) \leq cf(n) \}$
(asymptotisch obere Schranke).
$h(n) \in O(f(n))$ Bedeutet: es gibt eine Konstante c, sodass für große n immer h(n) < cf(n) ist.
Beispiel: wird eine Schleife n-mal durchlaufen (oder auch nur n/2-mal) so ist der Zeitaufwand O(n).
Eigentlich interessiert man sich aber für den Aufwand des Problems und nicht für den eines konkreten Algorithmus.
Kleinstmöglicher Aufwand für ein Problem = Komplexität des Problems (bestimme Algorithmus mit diesem Aufwand!)
Rechenregeln:

- seien $f(n) \in O(r(n))$ und $g(n) \in O(s(n)), c > 0$ konstant, so gilt:
  - $f(n) + g(n) \in O(r(n) + s(n))$; $f(n) * g(n) \in O(r(n) * s(n))$; $c * f(n) \in O(r(n))$ $f(n) \pm c \in O(r(n))$
  - Entsprechende Regeln für Subtraktion und Division gelten nicht!
  - Bei $O(\log(n))$ wird auf Basis verzichtet (konstanter Faktor).
  - Für c > 0 und a > 1und f(n) monoton steigend gilt: $f^c(n) \in O(a^{f(n)})$

Oftmals schreibt man statt $f \in O(g)$ folgendes: $f = O(g)$ Aber Vorsicht dies gilt nur in eine Richtung also beim lesen immer denken das = eigentlich $\in$ bzw. $\subseteq$ bedeutet.

Weitere Rechenregeln (artithmetische Operationen):

- Für Funktionen f,g und Konstante c > 0 gilt:

$c * O(f) = O(c * f) = O(f)$; $O(f) * O(g) = O(f * g)$; $O(O(f)) = O(f)$;
$O(f) \pm c = O(f \pm c) = O(f)$; $O(f) + O(f) = O(f)$; $O(f + g) = O(max(f, g))$

o-Notation (obere Schranke)

Def.: $o(f(n)) = \{ g(n) | \forall c > 0 \; \exists n_0 > 0 \; \forall n \geq n_0 : 0 < g(n) < c * f(n) \}$ (Unterschied: für jede Konstante c)

Ω-Notation (asymptotisch untere Schranke)

$h(n) \in \Omega(f(n))$: h(n) wächst mindestens so schnell wie f(n).

Def.: $\Omega(f(n)) = \{ g(n) | \exists c > 0 \; \exists n_0 > 0 \; \forall n \geq n_0 : 0 \leq c * f(n) \leq g(n) \}$

ω-Notation (untere Schranke)

$h(n) \in \omega(f(n))$: h(n) wächst deutlich schneller als f(n)

Def.: $\omega(f(n)) = \{ g(n) | \forall c > 0 \; \exists n_0 > 0 \; \forall n \geq n_0 : 0 \leq c * f(n) < g(n) \}$

Θ-Notation (asymptotisch gebunden)

$h(n) \in \Theta(f(n))$: h(n) wächst ebenso schnell wie f(n)

Def.: $\Theta(f(n)) = \{ g(n) | \exists c_1 > 0 \; \exists c_2 > 0 \; \exists n_0 > 0 \; \forall n \geq n_0 : 0 \leq c_1 * f(n) \leq g(n) \leq c_2 * f(n) \}$

Rechenregel: $f(n) \in \Theta(g(n))$ genau dann, wenn $f(n) \in O(g(n))$ und $f(n) \in \Omega(g(n))$

Symmetrie: $f(n) \in \Theta(g(n))$ genau dann, wenn $g(n) \in \Theta(f(n))$

Rekurrenzen

Eine Rekurrenz ist eine Gleichung oder Ungleichung, bei der der Funktionswert in Form von Funktionswerten für kleinere Eingabewerte beschrieben wird. Einen allgemeinen Algorithmus zur Lösung von Rekurrenzen gibt es nicht.
(Bsp.: Sykline-Problem: T(1) = 1; T(n) = 2T(n/2) + n) Die Laufzeiten rekursiver Algorithmen lassen sich manchmal mit Rekurrenzen beschreiben.

Lösen von Rekurrenzrelationen: Abschätzungsmethode

- Berechne T(n) für einige n.
- Schätze geschlossene Form für T(n) (oder ggf. nur Schranke) ab.
- Beweise dass geschätztes T(n) korrekt ist.

Allgemeine Lösung für Teile-Herrsche Probleme:

Teile-Herrsche Probleme lassen sich i.A. durch eine Rekurrenz der Form: $T(n) = a * T(n/b) + f(n)$ charakterisieren wobei $a \geq 1$, $b \geq 1$ und f(n) eine Funktion über $\mathbb{N}$ ist. Für deartige Rekurrenzen gibt es eine allgemeine Lösung.

Hauptsatz über lineare Rekurrenzen

Sei $a \geq 1, b \geq 1$ sei f(n) eine Funktion und T(n) gegeben durch die Rekurrenz $T(n) = a * T(n/b) + f(n)$ dann ist T(n) asymptotisch beschränkt durch

- wenn $f(n) \in O(n^{\log_b a - \epsilon})$ für $\epsilon > 0$, dann ist $T(n) \in \Theta(n^{\log_b a})$
- wenn $f(n) \in \Theta(n^{\log_b a})$ dann ist $T(n) \in \Theta(n^{\log_b a} * \log_2 n)$
- wenn $f(n) \in \Omega(n^{\log_b a + \epsilon})$ für $\epsilon > 0$ und wenn $a * f(n/b) \leq c * f(n)$ für c < 1 und genügend große n, dann $T(n) \in \Theta(f(n))$

Beweis: siehe Theoretische Informatik

Weitere Umrechnungen:

$1 + 2 + ... + n = 1/2\, n(n+1)$; $1 + 2 + ... + 2^n = 2^{n+1} - 1$; $1 + 4 + ... + n^2 = 1/6\, n(n+1)(2n+1)$

$\sum_{i=1}^{n} \lfloor \log_2 i \rfloor \in \Theta(n \log_2 n)$

Verkettete Liste

Eigenschaften einer verketteten Liste:

- Duplikate sind möglich; kontrollierte Reihenfolge der Elemente über Index.

Listeninterface in Java ist java.util.List (konkrete Implementierung als java.util.LinkedList)

Iteratoren: = Möglichkeit der Navigation durch eine bel. Datenstruktur. Mit einem Iterator wird der Besuch einer Datenstruktur unabhängig von der konkreten Implementierung dieser. Eine Datenstruktur kann somit mehrmals gleichzeitig traversiert werden. (Implementierung als innere Klasse der Listenklasse da so auf die Instanzvariablen der Listenklasse zugegriffen werden kann, innere Klasse = enthaltene Klasse die nicht statisch ist (Stichwort: geschachtelte Klassen)). Syntax anonymer Klassen:

new BaseClassOrInterface() { class-def. }

ADT Liste:

Signatur: Liste(T)		Axiome:
– create:	-> Liste	A1: head(append(x,l)) = x
– append: T x Liste	-> Liste	A2: tail(append(x,l)) = l
– head: Liste	-> T	A3: length(create) = 0
– tail: Liste	-> Liste	A4: length(append(x,l)) = 1 +
– length: Liste	-> $\mathbb{N}$	length(l)

```
java.util.Iterator:
public interface Iterator
{
   boolean hasNext();
   Object next();
   void remove();
}
```

ADT Menge(T) Signatur:

create:	-> Menge	equals: Menge x Menge	-> Bool.
add: T x Menge	-> Menge	containsAll: Menge x Menge	-> Bool.
isIn: T x Menge	-> Bool.	intersect: Menge x Menge	-> Menge
del: T x Menge	-> Menge	diff: Menge x Menge	-> Menge
single: T	-> Menge	empty: Menge	-> Bool.
union: Menge x Menge	-> Menge		

Typ-Parameter in Java. Syntax: *public class/interface<A>* optional *extends/implements Parentclass<A>* wobei A den Typparameter darstellt.
ADT Menge kann zum Beispiel als verkettete Liste implementiert werden.
Verwendet man eine Menge mit Sortierung so können die Mengenoperationen schneller (O(n)) implementiert werden.

Sortieren einer Menge:

Gegeben: Grundmenge U mit totaler Ordnung $.\leq.$ auf dieser, gesucht ist Anordnung von M gemäß $.\leq.$. Eine Sortierung heißt **stabil**, wenn gleiche Werte ihre relative Reihenfolge nicht ändern. Formal: wenn k < j und $e_k = e_j$ dann $i_k = perm(k) < perm(j) = j_k$ und $e_{i_k} = e_{j_k}$.

Man unterscheidet verschiedene Klassen von Sortierverfahren:

- interne Sortierverfahren: sortieren von Datensätzen im Hauptspeicher. Direkter Zugriff auf alle Elemente
- externes Sortierverfahren: sortieren von Massendaten, die auf externen Speichermedien gehalten werden. Zugriff auf Ausschnitt der Datenelemente beschränkt.

Objektvergleiche zur Sortierung in Java: in Java gibt es das Interface java.lang.Comparable das eine einzige Methode *public abstract int compareTo(Object o);* deklariert, diese kann als Basis für vergleichende Objekte genommen werden. Dabei sollten die Methode einen negativen Wert liefern wenn this < o, 0 wenn this = o und einen positiven Wert wenn this > o ist.

Sortierverfahren

1. **Selection sort („Sortieren durch Auswählen“)**
   Idee: lösche nacheinander alle Maxima aus einer Liste M und füge sie vorne an eine anfangs leere Ergebnissliste L an, die Ergebnissliste ist dann aufsteigend sortiert (Minima löschen führt zu einer absteigenden Sortierung).
   Aufwandsabschätzung für selection sort bei Implementierung mit verketter Liste: $T_{selsort} = \Theta(n^2)$
2. **Insertion sort („Sortieren durch Einfügen“)**
   Idee: Sortieren von Spielkarten, starte neuen Stapel mit erster Karte; nimm je erste Karte vom Stapel und füge sie an der richtigen Stelle im neuen Stapel ein. Konkret also: seien n-1 Werte bereits sortiert so kann man den n-ten Wert einsortieren indem wir seinen Platz an der richtigen stelle in der bereits sortierten Liste finden und die restlichen Elemente nach hinten verschieben. Wiederhole dies solange bis Originalliste leer ist. Aufwandsabschätzung für insertion sort: $T_{insort}(n) = O(n^2)$. Im besten Fall ist Gesamtaufwand O(n).
3. **Bucket sort („Fächersortierung“)**
   Diese Sortierung setzt voraus das die zu sortierenden Elemente im Intervall von [0,1) liegen bzw. gelegt werden können. Man erstellt nun mit einem Hilfsarray n Fächer in welche die Elemente einsortiert werden (z.B. mit Index $[\lfloor n * A[i] \rfloor]$). Nun werden die einzelnen Fächer noch mit insertion-sort sortiert. Sind die Werte gleichmäßig verteilt so ergibt sich ein Aufwand von $T(n) = O(n)$! (dies liegt daran das gewisse Vorrausetzung vom Input verlangt werden). Nicht praktibel wenn die zu sortierenden Werte größer sind als die Anzahl der Elemente.
4. **Radix sort (Radixsortierung)**
   Verallgemeinerung der Fächersortierung. Anstatt für jeden infragekommenden Wert ein Fach anzulegen wird dieser als Schlüssel in Segmente aufgeteilt. Pro Schlüsselteil gibt es ein Fach. Die Elemente werden gemäß ihres Schlüsselteils per Fächersortierung sortiert. Man benötigt also keine n Fächer mehr sondern nur noch k, wobei k die Anzahl der Schlüsselteile ist (z.B. 10). Aufwandsabschätzung für radix sort: $T_{radix} = O(n * \log_2 n)$ Setzt allerdings vorraus das Sortierschlüssel in Segmente aufgeteilt werden kann, also eine feste Struktur hat.
5. **Merge sort („Sortieren durch Mischen“)**
   Teile-und Herrsche: Liste der Länge $n = 2^m$ m > 0 in zwei Teillisten der Länge $n/2 = 2^{m-1}$ zerlegen und diese Teile wieder rekursiv mit merge-sort sortieren bis Leere oder einelementige Liste vorhanden ist diese ist dann bereits sortiert. Zusammenfügen der Ergebnisse(merge) zur sortierten Ergebnisliste (eigentliches sortieren).
   Aufwandsabschätzung für merge sort: $T_{merge} = O(n * \log_2 n)$. Mergesort ist ein stabiles Verfahren. Schnellstes Verfahren auf verketteten Listen wegen der geringsten Zahl an Vergleichen. Verbreitestes Verfahren zur externen Sortierung(wegen aufteilung in Teillisten).

Reihungen:

Eignen sich besser als verkettete Listen für das Suchen von Elementen, direkter Zugriff auf Element möglich. Dies beschleunigt erheblich das setzen und löschen von Elementen an direkter Position (O(1)!) im Vergleich zu Mengen die auf verketteten Listen basieren. Einfügen und Löschen von Element aus der Mitte mit Aufwand O(n) da restlichen Elemente nach rechts geschoben werden müssen. Bewährt sind dynamische Reihungen die die Speicherreservierung entsprechend erhöhen so dass immer genügend Platz reserviert ist.

Binärsuche

Vorraussetzung: eine sortierte Reihung und ein Element nach dem in der Reihung gesucht wird. Idee:

- greife Element in der Mitte der sortierten Reihung heraus (oder etwa der Mitte)
- vergleiche dieses mit dem gesuchten Element. Falls Übereinstimmung fertig.
- Keine Übereinstimmung, so suche in der Hälfte weiter in der es nach der Sortierung liegen muss. Führe dies solange rekursiv aus bis Element gefunden oder der zu durchsuchende Teil nur noch Länge 1 hat.

Aufwandsabschätzung im durchschnittlichen Fall: $T_{binary}(n) = O(\log_2 n)$ (sequentielle Suche hat Aufwand von O(n))

Das Prinzip das hinter der Binärsuche steckt wird auch **Bisektion** genannt, man kann jedoch oft schneller zum Ziel kommen wenn man nicht blind halbiert sondern wissen über Werteverteilung des zu durchsuchenden Bereichs mit einbezieht (**Interpolation**). (Bsp.: Nullstellensuche einer Funktion). Bei der Interpolation wird anstatt in zwei gleich große Hälften, der Suchbereich in die Hälften geteilt die den besten Erfolg versprechen (dabei wird Wissen über Daten genutzt).

Es folgen nun weitere Sortierverfahren die Eigenschaften der Reihung ausnutzen (das auf ein Element mit Aufwand O(1) zugegriffen werden kann). Folgende Verfahren arbeiten ohne Kopie auf ein und derselben Reihung.

Sortierverfahren (Fortsetzung, speziell für Reihungen geeignet)

6. **Bubble sort („Blasensortierung“)**
   Reihung wird wiederholend durchlaufen und dabei werden benachbarte Elemente in die richtige Reihenfolge gebracht. Somit überholen größere Elemente die kleineren und drängen ans Ende der Folge vor.
   Aufwandsabschätzung für bubble sort: $T_{bubble} = O(n^2)$
7. **Heap sort („Haldensortierung“)**
   Sortieren durch Auswählen gepaart mit einer Halde als Datenstruktur, da bei einer Halde die Maximumauswahl nur $O(\log_2 n)$ Schritte dauerte. Halden sind ideal zur Implementierung von Prioritätswarteschlangen.
   Haldeneigenschaft: der Wert eines Knotens ist größer oder gleich dem Wert seiner zwei Nachfolger. Dadurch ist der Knoten mit maximalem Wert immer an der Spitz der Halde nach Entnahme dieses Wertes muss Halde wieder umgeordnet werden damit Haldeneigenschaft wieder erfüllt ist. Es ist auch möglich in $O(nlog_2 n)$ eine Reihung in eine Halde mit Haldeneigenschaft einzufügen.
8. **quick sort („Sortieren durch Zerlegen“)**
   Teile-Herrsche Verfahren:
   Zerlege Gesamtaufgabe in zwei Sortierteilaufgaben, wobei linker Teil der Reihung kleine Werte und der rechte Teil große Werte enthält. Eine solche Anordnung wird hergestellt indem im Rahmen der Zerlegung zunächst einige Reihungselemente ihre Plätze tauschen. Zusammenfügen der Teillösungen ist trivial und einfach die konkatenation dieser. Sortierung findet innerhalb der gegebenen Reihung statt (in-situ Verfahren). Das Platztauschen und zerlegen erfolgt folgendermaßen: Auswahl eines **Pivotelements p** (möglichst mittig in der Reihung) welche die Trennstelle darstellt, links davon sind alle Elemente die kleiner gleich p sind und rechts davon alle die größer gleich p sind. Eine solche Partitionierung erfolgt wie folgt: starte jeweils am Anfang und am Ende rücke links wie auch rechts in Richtung des Pivotelements vor bis ein Wert der größer (bzw. kleiner bei rechtem Index) als p ist oder p selbst erreicht ist. Sind je auf einer Seite zwei solche ungültigen Werte gefunden worden tausche diese und fahre fort bis rechter Index < als linker Index ist. Aufwandsabschätzung für quick sort: $T_{quick}(n) = O(n \log_2 n)$ falls das Pivotelements immer genau mittig gewählt wird. Im schlechtesten Fall hat man aber $O(n^2)$. In der Praxis und in der Regel ist aber quick sort extrem schnell. Verbesserungen in der Wahl von p:
   zufälliges p bestimmen oder mehrere „Kandidaten“ bestimmen und von denen die Mitte wählen.
   Bemerkung: bei weniger als 10-20 Elementen ist normale Einfügesortierung schneller als Quicksort daher kann man die Rekursion nicht bis zur einelementigen Liste laufen lassen sondern früher auf die Einfügesortierung umsteigen.

Theoretische untere Schranke des Sortierens

Es gibt kein Sortierverfahren das auf Vergleichen von Schlüsselpaaren beruht und schneller als $O(n \log_2 n)$ ist. Das **Sortierproblem** hat damit die **Komplexität** von $O(n \log_2 n)$.

Streuspeicherung (Hash-Tabelle)

Eine Streuspeicherung ist eine Reihung in die Werte eingesetzt werden und in der nach Werten nachgeschlagen wird.Die Größe der Streutabelle wird wesentlich kleiner gewählt als der mögliche Wertebereich. Einzufügende Elemente werden nun mit einer Hashfunktion über die Tabelle verstreut. Einfügen eines Elements e:

1. Schlüssel k für e berechnen.
2. h(k) = i bestimmen und Element in die Reihung an Stelle i speichern, falls der Platz noch frei ist, sonst Kollisionsbehandlung durchführen.

Die Suche nach einem Element verläuft genauso. Es ist nicht notwendig das eine totale Ordnung auf den zu speichernden Objekten vorhanden ist. Streutabellen haben in der Regel eine feste Größe.
Die Streufunktion h wird i.A. nicht injektiv gewählt, dadurch kann es zu Kollisionen bei den berechneten Schlüsseln geben.
Lastfaktor/Belegungfaktor einer Streutabelle: $BF := |G'|/|I|$ wobei G' die Schlüsselmenge und I die Menge der bereits vergebenen Indizes darstellt.

Kollisionsvermeidung:

- durch Wahl der Streufunktion h, Schlüssel möglichst gleichmässig verteilen, h sollte effizient berechenbar sein.
  z.B. setzte Größe der Streutabelle auf m fest, wobei m eine Primzahl ist. $h(x) = x \bmod m$
  ist die Größe eine Zweierpotenz also $n = 2^k$ so ist gute Streufunktion $h(x) = (x \bmod p) \bmod n$ p ist Primzahl und $n < p \ll |G|$.
  sollten die zu speichernden Werte einen Zusammenhang zur Primzahl p haben, dann wähle zufällig $0 \neq a < p$ und $b < p$: $h(x) = ((ax+b) \bmod p) \bmod n$
- Auflösung durch Verkettung. Elemente mit gleichen Index werden als Elemente einer verketteten Liste an den entsprechenden Tabelleneintrag eingehängt. Dadurch ist Löschen möglich. Aufwandsabschätzung im schlimmsten Fall: $O(n)$ und im besten $O(1)$
- Auflösung durch Sondieren („open addressing“) Bei einer Kollision wird mit zyklisch nach einer Lücke gesucht, z.B. in Richtung des nächsten Elements. Dabei können Sekundärkollisionen auftreten, daher sollte die Fortschreitefunktion c nicht linear sein oder selbst wieder Hashfunktion. Vorteile: keine Listen, sparsamer Speicherverbrauch. Nachteil: Löschen ist schwierig und nicht direkt möglich(mit Löschmarkierung, etc.). langsamer als Kollisionsauflösung durch Verkettung. Um jede Stelle der Tabelle durch c zu erreichen muss die Schrittweite und Tabellengröße teilerfremd sein.

Kombinationen sind möglich zwischen sondieren und Verkettung. Grundsätzlich will man Laufzeit O(1) erreichen bei hohem Lastfaktor aber O(n) -> versch. Verfahren z.B. Reorganisation bei hohem BF (dynamisch durch anlegen zus. Reihung)

Suchbäume

Bäume kann man sich grundsätzlich als strukturelles Abbild des Teile-Herrsche Prinzips vorstellen. Ein Binärer Suchbaum ist ein Baum dessen Knoten maximal zwei Kinder haben. Die **Höhe** h eines Knotens entspricht der Anzahl der Kanten des längsten Pfades zu einem von diesem Knoten aus erreichbaren Blatt. Die Baumhöhe ist die Höhe der Wurzel.

Blattbaum/hohler Baum: Nutzdaten werden nur in den Blattknoten gespeichert.

Natürlicher Baum: Nutzdaten werden an jeden Knoten gespeichert.

Ausgewohgener, **balancierter** Baum: für jeden Knoten unterscheiden sich die Höhen der Unterbäume nur um höchstens 1.

ADT BinBaum(T):

Signatur:

- create: -> BinBaum
- bin: BinBaum x T x BinBaum -> BinBaum
- left: BinBaum -> BinBaum
- right: BinBaum -> BinBaum
- value: BinBaum -> T
- empty: BinBaum -> Boolean

Axiome:

A1: left(bin(x,b,y))=x

A2: right(bin(x,b,y))=y

A3: value(bin(x,b,y))=b

A4: empty(create)=true

A5: empty(bin(x,b,y))=false

SuchBaum(T) T mit Ordnungsrelation

create: -> BinTree

insert: T x BinTree -> BinTree

find: T x BinTree -> Boolean

delete: T x BinTree -> BinTree

Ein Suchbaum ist ein binärer Baum der folgendermaßen organisiert ist. Im linken Teil bzgl. der Wurzel befinden sich Einträge die < sind als die Wurzel und im rechten Einträge die > (oder auch >=) sind als die Wurzel. Diese Organisation findet man bei jeder Wurzel wieder. Somit befindet sich das Minimum eines solchen Baumes immer am linkesten Ast und entsprechend das Maximum am rechtesten. Eine Suche findet somit wie folgt statt. Man beginnt bei der Wurzel (Vergleich des Wertes) und verzweigt in den Teil des Baumes in den der Wert fällt dies wiederhole man so oft bis der entsprechend Wert gefunden worden ist oder man am Ende des Baumes angelangt ist.

- **Einfügen** in den Suchbaum: beginne bei der Wurzel und steige wie bei suche den Baum ab bis ein Blatt erreicht ist hänge neuen Eintrag entsprechend an dieses Blatt.
- **Löschen** eines Elements:
  1. Fall: Löschen eines Blatts. Suche den zu löschenden Knoten im Baum entferne Knoten
  2. Fall: Löschen eines Knotens mit nur einen Nachfolger. Nachfolgerzeiger parent und parent.left/right entsprechend umsetzen.
  3. Fall: Löschen von inneren Knoten mit 2 Nachfolgern. Ersetze zu löschenden Knoten durch größten Knoten im linken Unterbaum oder durch den kleinsten Knoten im rechten Unterbaum(beide haben max. 1 Nachfolger) der einzige Nachfolgeknoten kann leicht wie im 2. Fall behandelt werden.

Suchaufwand bei ausgewogenen Bäumen: $O(Höhe)=O(\log_2 n)$. Einfügen in $O(Höhe)=O(\log_2 n)$ bei ausgewogenen Bäumen, ebenso Löschen in $O(Höhe)=O(\log_2 n)$

Je nach Einfügereihenfolge sehen Suchbäume anders aus.

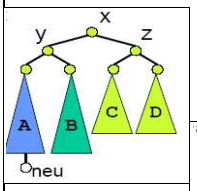
AVL(Adelson, Velskii, Landis)-Bäume

Sind Suchbäume die ausgewogen sind und diese Eigenschaft auch erhalten.

Balance-Faktor eines Knotens v = Differenz zwischen der Höhe des linken Unterbaums und der Höhe des rechten Unterbaums. Dieser Balance-Faktor gibt an ob ein Unterbaum der beim Knoten beginnt ausgewogen ist oder nicht. Ein ausgewogener Baum hat einen Balance-Faktor zwischen -1,1. Der Balance-Faktor muss bei Änderungen am Baum eventuell wieder neu berechnet werden dies geschieht bei einem bereits ausgewogenen Baum mit $O(\log_2 n)$. Die Bedigung für AVL-Bäume lautet wie folgt:

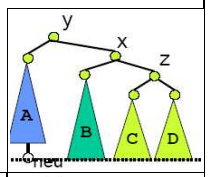
AVL-Bedingung: sei e beliebiger Knoten eines binären Suchbaumes und h(e) die Höhe des Unterbaums mit Wurzel e, dann gilt für die beiden Kinder e.links und e.rechts von e: $|h(e.links)-h(e.rechts)|\leq 1$. Ein AVL-Baum hat im schlechtesten Fall für Einfügen, Löschen und Suchen einen Aufwand von $O(\log_2 n)$

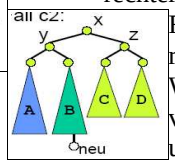
- **Einfügen**:
  1. Einfügen wie bei binären Suchbaum.
  2. Neuberechnung der Balance-Faktoren auf dem Pfad vom neuen Knoten zur Wurzel in $O(\log_2 n)$.
  3. Ausbalancieren, falls ein Knoten mit Balancefaktor $.\geq 2$ vorliegt. Dies ist mit Aufwand O(1) möglich:



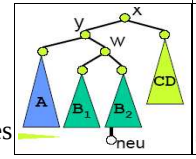
Es gibt prinzipiell zwei unterschiedliche Fälle beim ausbalancieren:

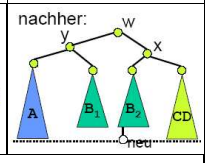
Fall A: (siehe linker Kasten) neu wird hinzugefügt. Die Knoten y und x werden einmal rotiert, d.h. Y wird neue Wurzel und die rechte Hälfte kippt nach unten (siehe rechter Kasten)





Fall B: In diesen Fall wird zunächst eine Vergrößerung betrachtet wie am rechten Rand erkenntlich. Nun wird der Knoten w neue Wurzel und x wird in Richtung des niedrigeren Teilbaums verschoben. Es findet also eine Doppelrotation statt y mit w und dann nochmal w mit x. Die Situation danach ist ganz rechts zu sehen. Das Einfügen erfolgt also in konstanter Zeit. Hinweis das ausbalancieren des Baumes ist nur in den Unterbäumen notwendig die unausgewogen sind.





Nachteile eines AVL-Baums: zusätzlicher Platzbedarf für Balancefaktoren. Komplizierte Implementierung

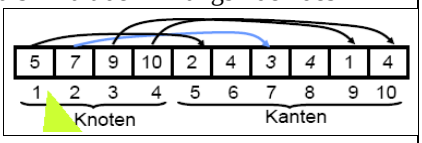
Aber Vorteil: alle Operationen erfolgen in Zeit $O(\log_2 n)$!

Graph-Grundlagen (siehe Theoretische Informatik)

Def.: **Spannbaum:** Sei G ein Graph und R ein zyklenfreier Teilgraph von G, der alle Knoten von G enthält und sind G und R beide zusammenhängend, dann heißt R **Spannbaum** von G.

Eine Familie von Teilgraphen G_i heißt **Partitionierung** eines Graphen G genau dann wenn: jeder Graph G_i zusammenhängend ist und $\forall i, j \in \{1,\dots,n\}, i \neq j \Rightarrow V_i \cap V_j = \emptyset$, $\cup_{i=1\dots n} V_i = V$ jeder Graph G_i wird dabei **Komponente** von G genannt.
Eine **Zusammenhangskomponente** Z eines Graphen ist ein zusammenhängender Teilgraph von G, der in keinem anderen zusammenhängenden Teilgraph von G enthalten ist. Z ist maximaler zusammenhängender Teilgraph von G.
Jeder Graph kann somit in eindeutige Weise in die Menge seiner Zusammenhangskomponenten partitioniert werden.
Hat ein gerichteter azyklischer Graph (DAG) nur eine Wurzel so heißt dieser **Wurzelgraph**.
Der Grad eines Knoten beim Baum ist die Anzahl der Nachfolgeknoten. Grad des Baumes ist der maximale Grad seiner Knoten. Länge des Pfades von der Wurzel bis zu einen Knoten k bestimmt die Höhe von k im Baum.
Darstellung von Graphen im Rechner:

- Adjazenzmatrix: Matrixdarstellung wobei Spalten und Zeilen Knoten sind und die Einträge angeben ob eine Verbindung vorliegt oder nicht (oder es können auch Kantengewichte gespeichert werden). Allerdings Speicherverschwendung bei dünnen Graphen $O(n^2)$.
- Adjazenzlisten: für jeden Knoten wird eine Liste angelegt wo die Kanten gespeichert werden die von diesem Knoten zu einem anderen Knoten gehen. Sehr wenig Speicherverschwendung aber kostspielige suche nach Kanten. $O(|E|)$.
- Reihung: am Anfang der Reihe werden die Knoten nacheinander gespeichert, wobei als Inhalt der Anfangsindex des Eintrags gespeichert wird der einen Knoten als direkten Nachfolger hat. Liegen mehrere Nachfolger vor so kann man solange weitergehen bis der Index des nächsten Knoteneintrags erreicht ist. Dies ist eine noch kompaktere Darstellung aber mit aufwändigen Änderungen (Einfügen/Löschen) verbunden.



Eulerscher Pfad: Pfad der alle Kanten umfasst und jede genau einmal durchläuft.
Eulerscher Zyklus: Eulerscher Pfad, bei dem Start- und Endknoten identisch sind. Es existiert ein Euler-Zyklus gdw. der Grad jedes Knoten durch 2 teilbar ist und der Graph zusammenhängend ist.

Graph-Traversierung

Ziel ist es in einem Graph jeden Knoten einmal zu besuchen.

- **Tiefensuche (Depth first search DFS):** besuche zuerst die Kinder jedes Knotens, absteigen bis zum Blatt dann zum nächsten Blatt,..., sind alle Blätter besucht zurück zum Elternknoten und eventuell noch unerforschte Zweige besuchen. Dabei muss markiert werden was schon alles besucht wurde. Es wird also nur die Zusammenhangskomponente besucht. Eine Implementierung einer DFS kann iterativ mittels eines Kellers oder auch rekursiv erfolgen. Oftmals wird ein Graph benutzt um eine bestimmte Aufgabe zu erledigen. Dabei unterscheidet man was beim ersten betreten eines Knotens zu tun, was ist zwischen den Besuchen der Nachfolger zu tun, was ist nach Besuch aller Nachfolger zu tun und was ist beim Abstieg zu tun. (Bsp.: Zählen der Knotenzahl in einem Baum). Ein Inorder-Besuch eines binären Suchbaum liefert eine sortierte Liste. Eine solche Inorder-Traversierung kann wie folgt erfolgen:
  1. beginne am linkesten Knoten, und setze aktuelle Position auf nächsten Knoten (hier unterscheidet man 2. Fälle:)
  - 1. Fall: alte Position hat genau einen rechten Nachfolger: gehe zu diesem Knoten und wandere sofort wieder zum linkesten Knoten ab.
  - 2. Fall: alte Position hat keinen rechten (wie auch linken) Nachfolger. Falls man nun von links kommt so ist die nächste Position die unmittelbare Wurzel der alten Position. Falls man von rechts kommt wird solange zurückgegegangen bis schliesslich ein Knoten gefunden wurde von man nicht mehr rechter Nachfolger ist. (Beachte den Fall falls man die Wurzel erreicht hat)

  DFS-Nummerierung: Graph bzw. Knoten werden mit Nummern versehen so wie sie bei der Tiefensuche durchlaufen werden. Weitere Anwendungen DFS-Algorithmus zur Erkennung von Zyklen.
- **Breitensuche (Breadth first search BFS):** besuche die Knoten ebenenweise, also erst alle in Höhe 0 dann in Höhe 1 usw. Eine Breitensuche kann mittels einer Schlange implementiert werden.

Kürzeste Pfade

Gesucht sei der kürzeste Pfad zwischen zwei gegebenen Knoten in einem Graphen. Die verallgemeinerte Pfadlänge wird nicht durch die Kantenzahl bestimmt sondern durch die Summe der positiven Kantengewichte.
Satz: Für jeden kürzesten Pfad $p = (v_0, v_1, \dots, v_k)$ von v_k ist jeder Teilpfad $p' = (v_i, \dots, v_j), 0 \leq i < j \leq k$ ein kürzester Pfad von v_i nach v_j .
Algorithmus von Dijkstra (auf positiv gewichteten gerichteten Graphen)

- markiere alle Knoten als unbesucht.
- Setze v.sp = 0 und $w.sp = \infty$ für alle $w \neq v$
- solange nicht besuchte Knoten existieren
  - 1. Minimumauswahl: wähle/besuche einen Knoten w, mit w.sp minimal (bei 1. Iteration wird wg. v.sp = 0 v gewählt)
  - 2. Aktualisierung: für alle Kanten (w,z) zu unbesuchten z:
    wenn (w.sp + c(w, z) < z.sp)
    dann setze z.sp := w.sp + c(w, z)

Aufwand für Minimalauswahl mittels einer Halde ist $O(|V| * \log_2 |V|)$
Gesamtaufwand für den Algorithmus: $O((|E| + |V|) * \log_2 |V|)$

Minimaler Spannbaum

Gegeben sei ein ungerichteter Graph mit Kantengewichten. Gesucht sei ein zusammenhängender Teilgraph der alle Knoten enthält, wobei die Summe der Kantengewichte minimal ist. (Dieser Teilgraph muss ein Spannbaum sein).
Der Algorithmus für die kürzesten Pfade lieferte einen Spannbaum kürzester Pfade für ein gegebenen Knoten. Infolgedessen kann man für alle Knoten die Spannbäume berechnen lasse und den minimalen auswählen.

Algorithmus von Prim:
Beginne mit einem Knoten der sicher zum minimalen Spannbaum gehört (dies kann jeder sein!)
suche nun aus den mit Kanten verbundenen benachbarten Knoten denjenigen der den kürzesten Abstand zum Anfangsknoten hat. Diese Kante muss zum MSB gehören (Wiederspruchsbeweis!) und wird hinzugenommen. Wiederhole dies mit dem neu hinzugenommenen Knoten wobei Kanten zu bereits besuchten Knoten nicht mehr beachtet werden. Wiederhole dies solange bis alle Knoten zum MSB gehören. $O(|E|+|V|*\log_2|V|)$

Algorithmus von Kruskal:
Man beginnt nun mit einem Wald von einzelnen Knoten und fügt diese nach und nach zum MSB zusammen. Beginne mit sortierter Kantenliste(nach Gewichten) in aufsteigener Reihenfolge. Eine Kante gehört zur Lösung wenn sie:

- einen vorhandenen Baum um einen noch nicht betrachteten Knoten erweitert.
- Zwei noch nicht betrachtete Knoten verbindet (zu einem neuen Baum),
- zwei verschiedene Bäume verbindet.

Nun kann man die Kanten nacheinander betrachten und sofort entscheiden ob die Kante zur Lösung gehört oder nicht.
Für die Implementierung eignet sich z.B. eine Halde. Aufwandsabschätzung: $O(|E|*\log_2|V|)$

<u>Ergänzung zu elementaren Datenstrukturen</u>
Besondere Listen:

- <u>Orthogonale Listen:</u> Einsatz bei sehr dünn besetzten Matrizen; die Listenelemente werden dabei gleichzeitig in mehreren Listen geführt. In der Regel verwendet man eine verkettete Liste pro Zeile und eine pro Spalte, zudem speichert jedes Listenelement noch seinen Zeilenindex und Spaltenindex in der Matrix sowie den Zeilennächsten bzw. Spaltennächsten

<u>Keller (Stack)</u> LIFO-Strategie
Ein Keller ist gedanklich ein Stapel auf den Elemente draufgelegt sowie wieder von oben abgenommen werden können. Somit funktioniert er nach dem LIFO (Last In First Out)-Prinzip. Implementiert werden kann ein Keller z.B. mit einer Liste.

<u>Schlange (queue)</u> FIFO-Strategie
Ähnlich wie beim Keller nur das hier das wegnehmen von Elementen am anderen Ende geschieht sodass die FIFO-Eigenschaft erfüllt ist. Auch eine Schlange kann mit einer Liste implementiert werden, dabei wird meistens zusätzlich ein Zeiger auf das letzte Element verwendet.
Eine Sonderform der Schlange ist die <u>zirkuläre Reihung</u>. Hier muss man sich eine Reihung als geschlossenen Kreis vorstellen wobei das erste Element auf das letzte folgt. Für Index i heißt das $i = Index \% maxSizeOfQueue$
<u>Prioritätswarteschlange (priority queue)</u>
Fast dieselbe Struktur wie Schlange nur das die Elemente nun eine Priorität haben und somit zuerst diejenigen Elementen herausgenommen werden welche die höchste Priorität haben. Auch hier kann eine Implementierung mittels einer Liste erfolgen (bei der Implementierung ist teilweise das verwenden von Schleppzeigern nötig), diese kann auch doppelt verkettet sein was die Implementierung etwas erleichtern kann.

<u>Binärbaum</u>
Ein Binärbaum ist ein Baum mit max. Verzweigungsgrad 2. Dabei können die Elementen einen solchen Binärbaumes nach bestimmten Regeln angeordnet sein (siehe Suchbaum, Halde). Die Implementierung eines Binärbaumes kann wie folgt erfolgen:

- Verweis von Knoten zu Nachfolgern oder Verweis von Nachfolger auf Vorgänger oder beides.
- Indexreihungen: jedem Knoten wird ein Index zugeordnet wobei Index -1 für keinen Nachfolger steht. Die Struktur ist dann in Tabellenform aufgebaut wobei die Spalten die einzelnen Knoten sind und in den zwei Zeilen die Indizes der linken und rechten Kinder stehen. (bei Bäumen mit höheren Verzweigungsgrad müssen entsprechend mehr Zeilen gespeichert werden)
- Impliziete Darstellung mit einer Reihung: Knoten werden wie folgt in einer Reihung gespeichert. Wurzel sei an erster Position. Der linke Nachfolger von Knoten i steht in A[2*i], der rechte in A[2*i+1] somit findet man den Vorgänger eines Knotens k bei A[k/2].

<u>Binäre Halde</u>
Eine Halde zeichnet sich durch folgende Eigenschaft aus.
Haldeneigenschaft: Der Wert eines Knotens ist größer oder gleich dem Wert seiner zwei Nachfolger (analog für $\leq$).
Somit steht an der Spitze der Halde das größte (kleinste) Element. Eine Halde ist daher ideal zur Implementierung einer Prioritätswarteschlange geeignet. Für eine Halde wird gern die impliziete Darstellung (letzter Punkt beim Binärbaum) mit einer Reihung verwendet. Grundlegende Operationen auf einer Halde laufen demnach wie folgt ab:

- Entnahme eines Elements: (es genügt den Fall zu betrachten wenn das größte Element entfernt wird) Nach Entfernung des größten Elements wird der letzte Knoten an die Spitze der Halde kopiert. Nun wird die neue Spitze mit seinen beiden Nachfolgern verglichen und mit dem größeren vertauscht, wiederhole dies bis die Haldeneigenschaft wieder erfüllt ist. Erfolgt im schlimmsten Fall in $O(2*\log_2 n)$
- Einfügen in eine Halde: Neues Element wird als letztes Element an die Reihung angehängt. Falls der Wert größer als der des Elternknotens ist wird getauscht, wiederhole bis Haldeneigenschaft wieder erfüllt ist. Schlimmster Fall in $O(\log_2 n)$