

# Notizen: Algorithmen und Datenstrukturen

## Klaus Kusche

### 1 Die O-Notation

Beschreibt den **Berechnungsaufwand**  
(Anzahl Maschinenbefehle, Durchläufe innerster Schleife,  
Anzahl der „wesentlichen“ Operation ...)

als Formel / Funktion in Abhängigkeit von der **Größe  $n$  der Eingabe**  
(Anzahl der Elemente, Eingabezahl, ...)

Achtung bei Zahlen-Eingaben:  $n$  = Zahlen**wert** oder  $n$  = Zahlen**länge** (Bits bzw. Ziffern) ???

Eventuell auch zwei oder mehr Eingabegrößen  $m, n$  in der Formel

$O(\dots)$  beschreibt das **asymptotische Wachstum** für  $n$  gegen **unendlich**  
unter **Vernachlässigung konstanter Faktoren**  
und aller für wachsendes  $n$  **schwächer wachsender Formel-Anteile**

1.) Nicht aussagekräftig für kleines  $n$  !

Ein laut  $O(\dots)$  schlechterer Algorithmus kann für kleine  $n$  deutlich besser sein,  
eventuell sogar für alle praktisch relevanten  $n$  besser als einer mit „besserem“  $O(\dots)$ :  
- Der „bessere“ Algorithmus kann eine viel schlechtere Konstante haben!  
- Bei kleinem  $n$  können die schwach wachsenden Anteile bei weitem dominieren!

2.) Liefert keine Aussage über die absolute Geschwindigkeit bei bestimmtem  $n$ !

Zwei Algorithmen können bei gleichem  $n$  praktisch sehr unterschiedlich schnell sein,  
auch wenn sie gleiches  $O(\dots)$  haben:  
- Unterschiedliche Konstanten und schwach wachsende Anteile,  
die man im  $O(\dots)$  nicht sieht!

Unterscheide, worauf sich das  $O(\dots)$  genau bezieht:

- **Rechenschritte** (Zeit, meistens) oder **Speichereinheiten** (Platz, seltener)?
- **Minimum, Mittel oder Maximum (Worst Case)?**  
Mittel = Mittelwert über alle möglichen Eingaben der Größe  $n$   
Das Minimum ist meistens uninteressant, Maximum und Mittel sind wichtig
- **Nur obere Schranke** (ev. zu groß) oder **genaue Größenordnung** der Komplexität  
(asymptotisch bis auf Faktoren ident)?

**Berechnung:**

1. Schritte genau ausrechnen, wenn möglich ausmultipliziert  
==> Formel abhängig von  $n$
2. Nur den bei  $n$  gegen unendlich am stärksten wachsenden Summanden behalten,  
alle langsamer wachsenden Summanden weglassen  
(wenn die Formel ein Polynom ist: Nur die höchste Potenz behalten)

3. Alle konstanten Faktoren in diesem größten Summanden weglassen  
(daher auch: Beim log ist die Basis egal)

#### Hinweise:

- Man sieht einem Problem nicht immer gleich sein  $O(\dots)$  an:  
Z.B. Test auf Primzahl: Erst seit 2002 ein polynomialer Algorithmus nachgewiesen
- Manchmal kommen auch sehr „komische“ Formeln vor:  
Z.B. Faktorisierung ganzer Zahlen: Überpolynomial, aber subexponentiell  
Zahl mit L Ziffern:  $O(\exp(\sqrt{L \cdot \log(L)}))$
- Es gibt Funktionen (u.a. die „Ackermannfunktion“), die rekursiv berechenbar sind, aber überexponentielle Laufzeit haben (keine geschlossene Formel für das  $O(\dots)$ )

Vereinfachter Ackermann:

Rekursiver Aufruf mit weiterem rekursiven Aufruf in den Argumenten:

$$a(0,m) = m + 1$$

$$a(n,0) = a(n - 1, 1)$$

$$a(n,m) = a(n - 1, a(n, m - 1))$$

Die Umkehrfunktion des Ackermann (wächst extrem langsam, ist aber nicht konstant), tritt in manchen Laufzeit-Analysen auf!

#### Übliche $O(\dots)$ :

$O(1)$  Konstante Laufzeit, unabhängig von der Eingabe.

$O(\log(n))$  Wächst sehr langsam:  
Beispielsweise bei wiederholter Reduktion auf ein Problem halber Größe, wie beim binären Suchen und vielen Baum-Algorithmen.  
Problemgröße  $\cdot 2 \Rightarrow$  Laufzeit  $+1$ , Problemgröße  $\cdot 1024 \Rightarrow$  Laufzeit  $+10$

$O(\sqrt{n})$  Langsam wachsend. Kommt selten vor.

$O(n)$  Wächst linear mit  $n$ : Doppeltes  $n \Rightarrow$  doppelte Laufzeit. Bei einer Schleife mit  $n$  (oder z.B.  $n/2$ ) Durchläufen oder einer Rekursion der Größe  $n-1$ .

$O(n \cdot \log(n))$  Meist die letzte „angenehm wachsende“ Komplexitätsklasse (z.B. Sortieren), wächst bei sehr großem  $n$  kaum stärker als  $O(n)$ .  
Beispielsweise bei wiederholter Reduktion auf zwei Probleme halber Größe, wie bei Quicksort, Mergesort usw..

$O(n \cdot \log(n) \cdot \log(\log(n)))$  sowie  $O(n \cdot \log(n)^2)$  Nur unwesentlich schlechter.  
Bei einigen praktischen Algorithmen.

$O(n^{\log_2(3)}) = O(n^{1,585})$  Bei wiederholter Reduktion auf drei Probleme halber Größe (z.B. Karazuba-Multiplikation).

$O(n^2)$  Quadratisches Wachstum: Doppeltes  $n \Rightarrow$  vierfache Laufzeit  
Einerseits zwei geschachtelte Schleifen mit je  $n$  Durchläufen (oder z.B. äußere Schleife  $n-1$ , innere Schleife im Mittel  $n/2$ : Primitive Sort's)  
Andererseits bei wiederholter Reduktion auf vier Probleme halber Größe.

$O(n^3)$  Kubisches Wachstum: Doppeltes  $n \Rightarrow$  achtfache Laufzeit  
Einerseits drei geschachtelte Schleifen mit je  $n$  Durchläufen, beispielsweise naive Matrix-Multiplikation.  
Andererseits bei wiederholter Reduktion auf acht Probleme halber Größe.

$O(2^n)$       Exponentielles Wachstum.  $n$  wird um 1 größer ==> Laufzeit verdoppelt sich!  
 Bei wiederholter Reduktion auf zwei Probleme der Größe  $n-1$ :  
 Naive rekursive Berechnung der Fibonacci-Zahlen & Binomialkoeffizienten.  
 Backtracking mit  $n$  Schritten und 2 Möglichkeiten für jeden Schritt.

$O(m^n)$       Exponentielles Wachstum in  $n$  für gegebenes (fixes)  $m$ .  
 $n$  wird um 1 größer ==> Laufzeit ver- $m$ -facht sich!  
 Bei wiederholter Reduktion auf  $m$  Probleme der Größe  $n-1$ :  
 Backtracking mit  $n$  Schritten und  $m$  Möglichkeiten für jeden Schritt.

$O(n!)$  und  $O(n^n)$  wachsen noch stärker als rein exponentielles Wachstum.  
 Es gilt  $n! = O(n^n) = O(2^{n \cdot \log(n)})$  (aber nicht umgekehrt!).  
 Weiters gilt  $O(\log(n!)) = O(\log(n^n)) = O(n \cdot \log(n))$ .

n	log(n)	n*log(n)	n <sup>2</sup>	n <sup>3</sup>	2 <sup>n</sup>	n!
2	1	2	4	8	4	2
3	1,585	4,755	9	27	8	6
4	2	8	16	64	16	24
10	3,322	33,22	100	1000	1024	3628800
11	3,459	38,05	121	1331	2048	39916800
12	3,585	43,02	144	1728	4096	479001600
20	4,322	86,44	400	8000	1048576	2,433*10 <sup>18</sup>
100	6,644	664,4	10000	1000000	1,268*10 <sup>30</sup>	9,333*10 <sup>157</sup>
1000	9,966	9966	1000000	1000000000	1,072*10 <sup>301</sup>	4,024*10 <sup>2567</sup>

## 2 Problem-Klassen

Quantencomputer ignorieren wir jetzt einmal ...!

Was ist eine nichtdeterministische Turingmaschine?

Bei einer solchen Maschine sind in manchen oder allen Schritten die Aktion und/oder der Folgezustand nicht eindeutig festgelegt, sondern es gibt für bestimmte Situationen (Zustand und Bandsymbol) mehrere mögliche Aktionen und Folgezustände zur Auswahl.

Die nichtdeterministische Turingmaschine besitzt ein „Orakel“ (einen Wahrsager), das in diesem Fall **immer die „richtige“ (zur Lösung führende) Aktion** samt Folgezustand auswählt!

Sinngemäß erweiterbar auf fast jedes Berechnungsmodell:

*Wenn die Lösung eines Problems aus mehreren Schritten besteht und man für jeden Schritt mehrere Möglichkeiten / Alternativen probieren müsste, dann entscheide dich „mit Wahrsagerei“ immer gleich für die richtige Alternative!*

Daraus ergeben sich zusammen mit  $O(\dots)$  folgende **Problem-Klassen**:

$P \subseteq NP \subseteq PSPACE = NPSpace \subseteq EXPTIME \subseteq NEXPTIME \subseteq EXPSPACE = NEXPSPACE$

$\subseteq$  ... „ist Teilmenge von oder gleich“

$\subset$  ... „ist echte Teilmenge von“ (echt kleiner, nicht gleich)

P Alle Probleme, die eine deterministische Turingmaschine in polynomialer Zeit lösen kann, d.h.  $O(\dots)$  ist ein Polynom von  $n$

NP Alle Probleme, die eine nichtdeterministische Turingmaschine in polynomialer Zeit lösen kann

PSPACE Alle Probleme, die eine deterministische Turingmaschine mit polynomial viel Speicher (und beliebig viel Zeit) lösen kann

NPSpace analog mit nichtdeterministischer Turingmaschine

EXPTIME Alle Probleme, die eine deterministische Turingmaschine in exponentieller Zeit lösen kann, d.h.  $O(\dots)$  ist  $2^n$  o.ä.

NEXPTIME analog mit nichtdeterministischer Turingmaschine

EXPSPACE Alle Probleme, die eine deterministische Turingmaschine mit exponentiell viel Speicher lösen kann

NEXPSPACE analog mit nichtdeterministischer Turingmaschine

Die Abbildung Turingmaschine  $\Leftrightarrow$  realer Rechner erhält die Komplexitäts-Klasse!

Weiters gibt es noch einige „einfachere“ Klassen als P

**Bewiesen:**  $P \subset EXPTIME$  und  $PSPACE \subset EXPSPACE$

$\Rightarrow$  Es gibt EXPTIME-Probleme, die sich beweisbar nicht in polynomialer Zeit lösen lassen (zumindest nicht mit einem deterministischen Rechner), und dasselbe gilt für exponentiellen und polynomialen Speicher.

$\Rightarrow$  Mindestens eines der ersten drei  $\subseteq$  oben muss echt kleiner sein!

$\Rightarrow$  Mindestens eines der letzten drei  $\subseteq$  oben muss echt kleiner sein!

Aber welches???

Unbewiesene Vermutung:

***Alle  $\subseteq$  oben sind echt kleiner,  
d.h. die Problemklassen sind alle verschieden und aufsteigend mächtig!***

Interessant vor allem für  $P \subseteq NP$  !!

## 2.1 P versus NP

### **$P \subseteq NP$ oder $P \subset NP$ ???**

Eines der ältesten und berühmtesten *ungelösten Probleme* der theoretischen Informatik ... und das einzige der Fragestellungen oben von *praktischer Relevanz!*

P ... Deterministische Maschine, polynomiale Zeit (worst case, nicht average)  
***==> „Praktisch lösbar“!***

NP ... Nichtdeterministische Maschine, polynomiale Zeit

Derzeitige praktische Umsetzung auf deterministischer Maschine:

- Entweder exponentielle Zeit (alle Möglichkeiten nacheinander probieren, und das sind bei praktisch allen interessanten Problemen exponentiell viele)
- oder polynomiale Zeit, exponentiell viele Prozessoren parallel (alle Möglichkeiten gleichzeitig verfolgen, denn jede einzelne Möglichkeit lässt sich in polynomialer Zeit lösen)

***==> Derzeit nur für (zu) kleines N praktisch lösbar!***

***==> In der Praxis stattdessen manchmal: Ziemlich gute Näherungsverfahren, probabilistische und heuristische Verfahren***

Wäre  $P = NP$  (was keiner glaubt, aber was noch nicht formal widerlegt ist), dann wären sehr viele praktisch relevante Probleme, die derzeit nicht für relevante Problemgrößen lösbar sind,

**plötzlich praktisch lösbar!**

(aber für die Kryptografie wäre das eventuell ziemlich schlimm!)

„Positive“ Erkenntnisse:

- Auch NP-Probleme brauchen nur polynomialen Platz, egal welche Maschine! (alle NP-Probleme sind PSPACE, sogar alle NPSPACE-Probleme sind auch PSPACE)
- Die oben definierte nichtdeterministische Maschine (bzw. die Klasse NP) ist auch nur gleichmächtig wie eine deterministische Maschine: Sie löst dieselben Probleme, alle NP-Probleme sind auch deterministisch lösbar! Aber sie ist für viele Probleme nach derzeitiger Wissenschaft eine Klasse schneller!
- Die Prüfung der Korrektheit einer Lösung eines NP-Problems ist in P (die Lösungen sind „einfach“ prüfbar)
- Es gibt einige wenige praktisch relevante Probleme, die nicht in P sind, aber trotzdem praktisch lösbar, weil für P / NP der Worst Case entscheidet, aber nicht die mittlere Komplexität.

Beispiel Simplex-Algorithmus:

- Worst Case ist exponentiell
- Mittel ist polynomial
- Best Case ist de facto linear
- Ist praktisch für reale Problemgrößen gut berechenbar!

## 2.2 NP-vollständige Probleme

NP-vollständige Probleme sind Probleme, auf die ***alle anderen NP-Probleme in polynomialer Zeit abgebildet*** (zurückgeführt) werden können.

Beispiele:

- Rucksackproblem (n ... Anzahl Gegenstände)
- Erfüllbarkeit aussagenlogischer Formeln („SAT“) (n ... Anzahl Variablen)  
Wahrheitstafeln haben  $2^n$  viele Zeilen!
- Viele Graphen-Probleme: (n ... Anzahl Knoten oder Kanten)
  - Hamilton-Kreis (jeden Knoten genau 1 x besuchen)  
und damit Travelling Salesman,
  - Graph-Cliquen-Problem
  - Knotenfärbung
  - ...

Würde man für ein einziges NP-vollständiges Problem einen Lösungsalgorithmus der Klasse P finden (also mit polynomialer Zeit) so wären damit automatisch alle NP-Probleme in P lösbar,  
d.h.  $P = NP!$

Aber: Der „naheliegende“ Lösungsansatz für sehr viele NP-Probleme und insbesondere fast alle NP-vollständigen Probleme beruht auf ***Backtracking***, und die worst-case-Komplexität von Backtracking ist offensichtlich  $O(m^n)$ .

## 2.3 (Un-) Berechenbarkeit

Die wichtigsten Berechenbarkeitsmodelle sind ***gleich fähig / gleich mächtig***:

- **Theoretische Modelle:**  
Turing-Maschine, Register-Maschine, Lambda-Calculus, Term-Rewrite-Systeme, ...
- **Normale Programmiersprachen und Rechner:**
  - Entweder Schleifen & unbeschränkte Arrays (kann Rekursion emulieren)
  - oder Rekursion & unbeschränkter Stack (kann Schleifen emulieren)
- „Intuitiver“ Berechenbarkeitsbegriff

Es gibt Probleme, von denen ***bewiesen*** ist, dass sie ***nicht berechenbar / nicht algorithmisch lösbar*** sind (aber manche dieser Probleme sind semi-entscheidbar):

- Viele Probleme, die ***ein oder mehrere Programme als Input*** haben (z.B. Source-Code oder formale Beschreibung einer konkreten Turing-Maschine) und inhaltliche ***Aussagen über das Verhalten dieser Programme*** betreffen:

- **Das Halteproblem:**  
**Input:** Ein beliebiges Programm  
**Output ja / nein:** Hält dieses Programm für alle möglichen Eingaben,  
oder gibt es Fälle, in denen es endlos läuft?  
Nicht einmal für eine bestimmte, vorher festgelegte Eingabe entscheidbar!
- **Programm-Korrektheit:**  
**Input:** Ein beliebiges Programm und eine formale Problem-Spezifikation  
**Output ja / nein:** Erfüllt das Programm diese Problem-Spezifikation,  
d.h. ist das Programm eine korrekte Lösung des Problems?
- **Programm-Äquivalenz:**  
**Input:** Zwei beliebige Programme  
**Output ja / nein:** Liefern beide Programme für jeden Input denselben Output?
- Einige (teilweise äquivalente) Probleme über **Grammatiken** und **formale Sprachen:**  
**Input:** Formale Grammatiken  
- Ist die resultierende Sprache leer?  
- Ist die resultierende Sprache endlich?  
- Liefern zwei Grammatiken dieselbe Sprache?  
Für allgemeine Grammatiken (Chomsky Typ 0 und Typ 1) unentscheidbar,  
für eingeschränkte Grammatiken (ab Typ 2) entscheidbar.
- **Automatisches Beweisen:**  
**Input:** Eine beliebige Formel der Prädikatenlogik 1. Stufe  
(das umfasst viel mehr als Aussagenlogik!)  
**Output ja / nein:** Ist das eine gültige (wahre) mathematische Aussage?
- **Lösbarkeit polynomialer diophantischer Gleichungen:**  
**Input:** Eine beliebige solche Gleichung  
(= Gleichung mit Potenzen und ganzzahligen Koeffizienten)  
**Output ja / nein:** Hat diese Gleichung eine ganzzahlige Lösung?

#### Außerdem:

- Es gibt Funktionen / Mengen (z.B. „busy beaver“ = Rado-Funktion),  
die für jedes konkrete  $n$  wohldefiniert und eindeutig sind,  
aber nicht mit einem allgemeinen Algorithmus (für jedes  $n$ ) berechenbar sind:  
- Es kann keinen allgemeinen Algorithmus geben, der für beliebiges  $n$  als Eingabe  
 $f(n)$  berechnet, oder für alle  $n$  der Reihe nach  $f(n)$  aufzählt.  
- Die Größe des Ergebnis-Wertes wächst stärker als jede berechenbare Funktion!  
(und damit wächst auch die Komplexität von Berechnungs-Versuchen so)  
**Gesucht:** Maximum des Band-Outputs aller möglichen haltenden Turing-Maschinen  
bestimmter Konstruktionsvorschrift mit  $n$  Zuständen.  
**Problem:** Die Konstruktionsvorschrift ermöglicht auch nicht haltende Maschinen!  
 $\implies$  Aufzählung aller Outputs ist daher wegen des Halteproblem nicht möglich!  
(auch: Beweis ohne Halteproblem wegen Wachstum des Outputs)
- Es gibt schon sehr einfache Programme, für die das Halte-Verhalten unbekannt ist  
(z.B. die **Collatz-Vermutung**: Bisher ist weder die Unlösbarkeit bewiesen  
noch ein Beweis in die eine oder andere Richtung geglückt).

### 3 Die Rekursion & Backtracking

Siehe separate Folien ...

### 4 Suchen

Im Array:

- Unsortiertes Array ==> lineares Suchen  
Ganz schlecht, **Komplexität  $O(n)$** : Im Schnitt  $n/2$  wenn gefunden,  $n$  wenn nicht  
==> im Mittel 500000 Vergleiche für 1 Mio. Elemente.
- Sortiertes Array ==> binäres Suchen  
Viel besser, **Komplexität  $O(\log(n))$**   
==> Nur max. 20 Vergleiche für 1 Mio. Elemente!

Problem: Einfügen / Löschen in sortiertem Array ist ganz schlecht!

**(Komplexität  $O(n)$ ):** Im Schnitt die Hälfte aller Elemente verschieben!

==> Arrays für Daten, in denen gesucht werden soll,  
sind nur dann sinnvoll, wenn sich die Daten nicht (oder sehr selten) ändern!

Daher meist: Suchen in besser geeigneter Datenstruktur (siehe später):

- Baum:  
Ermöglicht effizientes **Suchen in  $O(\log(n))$**  und sortiertes Durchlaufen in  $O(n)$ .
- Hashtable:  
Noch viel effizienter beim Suchen:  **$O(1)$**  (konst. Zeit!) solange sie nicht überfüllt ist!  
Aber: Nur Suchen möglich, kein sortiertes Durchlaufen!

## 5 Sortieren

### Grundlegende Begriffe:

- (Primärer) Sortierschlüssel:  
Jener Teil der Daten, nach dem sortiert wird (z.B. Name, Alter, ...).
- Sekundärer Sortierschlüssel:  
Jener Teil der Daten, nach dem Elemente sortiert werden, die denselben Primär-Schlüsselwert haben (z.B. Vorname, Dienstalter, ...).
- Stabiles Sortieren: Sortierverfahren, die die relative Reihenfolge von Elementen mit gleichem Sortierschlüssel im Input erhalten.

### Sonderfälle:

- Externes Sortieren (auf Band / Platte)  
Meist Merge-Sort, siehe unten.
- Sortieren ohne Vergleichen (z.B. Radix Sort / Distribution Sort, siehe unten)

### Theoretische Erkenntnisse:

Sortieren von  $n$  Elementen durch Vergleichen geht grundsätzlich (unabhängig vom Sortieralgorithmus!) nicht besser als mit

**im Mittel  $O(n * \log(n))$  Vergleichen.**

Radix Sort ist besser:  $O(n * m)$  für fixes  $m$  (weil er nicht vergleicht)

### 5.1 Simple Sortier-Verfahren:

Brauchen alle **im Mittel  $O(n^2)$  Schritte**

==> Sinnvoll nur bis rund 20 Elemente.

3 Varianten:

- **Sortieren durch Einfügen:**  
Schiebe das 2., 3., ...  $n$ . Element an die richtige Stelle nach vor (und alle Elemente auf dem Weg nach vor um eins nach hinten).  
Vorteil: Erkennt schon sortiertes Array schnell ( $O(n)$ ).  
Kommt mit dem häufigen Fall "fertig sortierter Input mit wenigen neuen, unsortierten Elementen am hinteren Ende" relativ gut zurecht.  
Nachteil: Im allgemeinen Fall eher langsam (sehr viele Zuweisungen...).
- **Sortieren durch Auswählen:**  
Suche das kleinste Element im verbliebenen Rest des Arrays und vertausche es mit der ersten, zweiten, dritten, ... Stelle im Array.  
Nachteil: Braucht immer  $O(n^2)$ , auch wenn der Input schon fertig sortiert ist.  
Vorteil: Bei großer Unordnung relativ am schnellsten (weil die wenigsten Zuweisungen / Vertauschungen gemacht werden und falsche Elemente in einem Schritt über weite Distanzen wandern)!
- **Sortieren durch Vertauschen:**  
Bubble Sort: So lange immer wieder alle durchgehen und falsche benachbarte Elemente vertauschen, bis kein Austausch mehr stattfindet (max.  $n$  Durchläufe).

Vorteil: Erkennt schon sortiertes Array schnell (nur 1 Durchlauf,  $O(n)$ )!

Nachteil: Ist im Mittel der Langsamste (macht die meisten Zuweisungen).

Problem: Verhält sich asymmetrisch: Zu große Elemente vorne wandern in einem einzigen Durchlauf ganz nach hinten, aber ganz kleine Elemente hinten wandern nur einen einzigen Platz pro Durchlauf nach vor ==>  $n$  Durchläufe nötig!

Leider ist "vorne sortiert, hinten ein paar neue, unsortierte Elemente" häufig!

Trick dagegen: *Abwechselnd* aufsteigend und absteigend durchlaufen!

## 5.2 Verbesserte simple Verfahren:

Grundidee: Beim Sortieren durch Einfügen oder Vertauschen wandern Elemente immer nur einen Platz, Umordnungen über weite Strecken sind daher aufwändig

==> Beseitige zuerst die grobe Unordnung über große Distanzen, dann lokale Unordnung!

- **Shell Sort:**

Einfüge-Sort mit absteigenden Vergleichsdistanzen, zuletzt normaler Einfüge-Sort.

Wissenschaft: Was ist die beste Folge von Distanzen? (z.B. Fibonacci-Folge, nicht  $2^n$ !)

Theoretisch:  $O(n^{1+\epsilon})$  für sehr große  $n$ , praktisch:  $O(n^{1.33})$  oder  $O(n * (\log(n))^2)$

- **Gap Sort:**

Bubble Sort mit absteigenden Vergleichsdistanzen.

## 5.3 Höhere Sortierverfahren:

Alle  $O(n * \log(n))$  mittlere (nicht maximale!) Laufzeit.

- **Quick Sort:**

Nimm irgendein Element (das mittlere; Trick: das wertmäßig mittlere von 3).

**Partitioniere:** Schiebe alle größeren Elemente nach rechts, alle kleineren nach links.

Sortiere die beiden Teile rekursiv (Trick: Wenn klein genug: Mit simplem Sort).

Im Mittel  $O(n * \log(n))$ , in der Praxis das

schnellste halbwegs kurz & einfach implementierbare Verfahren,

ändert schon sortierte Arrays bei klugem Partitions-Algorithmus nicht!

Aber:

**Worst Case  $O(n^2)$**  (bei sehr ungleichen Hälften,

wenn das teilende Element sehr groß oder sehr klein war).

Kein stabiles Sortierverfahren.

- **Merge Sort:**

Array in zwei Hälften teilen, jede rekursiv sortieren, die beiden sortierten Hälften danach elementweise zu einem sortierten Array vereinigen („Merge“).

Problem: **Braucht ein zweites Arrays zum Vereinigen der beiden Hälften.**

==> Einziger Sort, der zusätzlich zum zu sortierenden Array  $O(n)$  Speicher braucht (statt  $O(1)$  wie die meisten Verfahren und  $O(\log(n))$  wie Quicksort).

In der Praxis das schlechteste Verfahren, weil es die meisten Zuweisungen macht (gleiches  $O(n * \log(n))$ , aber deutlich höhere Konstante),

aber **Worst Case nur  $O(n * \log(n))$ !**

Vorteil: Liest und schreibt die beiden Hälften und das zusammengefügte Array nur sequentiell, nie mit direktem Zugriff ==> **optimal geeignet für externes Sortieren**

(Files und Bänder lassen sich nur sequentiell effizient schreiben / lesen!).

Stabiles Sortierverfahren.

Verbesserung: Natural Merge Sort:

Teilt nicht mehr „ohne Hirn“ wiederholt in zwei gleichgroße Hälften bis Größe 1, sondern sucht und nutzt vorhandene vorsortierte Teilstücke unterschiedlicher Länge.

In-Place-Varianten: Reduzieren Speicherbedarf, meist auf Kosten der Laufzeit.

- **Heap Sort:**

Verwendet intern einen linearisierten Binärbaum (d.h. in einem Array gespeichert) mit größtem Element oben (“**Heap**” oder “**Priority Queue**”, hat nichts mit dem Heap der Speicherverwaltung zu tun), arbeitet in zwei Schritten:

- Eingabe-Array elementweise in einen Heap verwandeln.
- Heap in sortiertes Array verwandeln: Immer das größte Element aus dem Heap entnehmen und nach hinten stellen.

Schwer zu verstehen und “unintuitiv”, im Mittel deutlich schlechter als Quicksort (gleiches  $O(n * \log(n))$ , aber deutlich höhere Konstante)

(macht viel mehr Zuweisungen, auch bei schon fertig sortiertem Input), aber **Worst Case nur  $O(n * \log(n))!$**

Kein stabiles Sortierverfahren.

- **TimSort:**

In der Praxis schnellstes Sortierverfahren:

Best Case  $O(n)$  (nutzt Vorsortierung gut aus), average und worst Case  $O(n * \log(n))$ .

Stabiles Sortierverfahren.

Sehr umfangreicher Algorithmus: Nutzt für kurze Array-Abschnitte Insertion Sort, für längere Abschnitte einen verfeinerten Natural Merge-Sort mit zahlreichen Speicherplatz-Optimierungen.

Standard-Verfahren u.a. in Python und Java.

## 5.4 Radix Sort / Distribution Sort:

Voraussetzung:

- Der Sortierschlüssel besteht aus einzelnen Ziffern oder Zeichen aus einem endlichen, nicht zu großen Zeichensatz (max. k Zeichen).
- Der Sortierschlüssel hat eine fixe (maximale) Länge m, je kürzer umso besser.

Beispiele: Sozialversicherungsnummer, Autokennzeichen, ...

Verfahren:

- Wiederhole für jedes einzelne Zeichen des Schlüssels, von hinten nach vorne:
  - Mach k Hilfs-Datenstrukturen (meist: verkettete Listen), eine pro möglichem Zeichenwert.
  - Verteile die Elemente der Reihe nach gemäß ihrem i-ten Zeichen im Schlüssel auf die Listen (jeweils hinten anhängen, d.h. erhalte dabei ihre relative Reihenfolge!)
  - Hänge die Listen in aufsteigender Zeichenreihenfolge zu einer einzigen Liste zusammen.

Auch der Radix-Sort eignet sich sehr gut für **externes Sortieren** sehr großer Datenmengen (auf Band oder Platte), weil er nur sequentiell liest und schreibt. Stabiles Sortierverfahren.

Komplexität:

Radix-Sort braucht keine Vergleiche, aber  $O(n * m)$  Schlüsselzugriffe und Zuweisungen, hat also für fixes  $m$  nur  $O(n)$ .

Problem: Speicherverbrauch, viele Arrays und/oder viele Kopier-Operationen (geht am besten mit Listen statt Arrays).

## 5.5 Praktische Hinweise:

- Verwende (außer in Spezialfällen) vorhandene Sortier-Funktionen! (z.B. **qsort** in C)
- Bei Strings, Strukturen, ...:  
**Nur die Pointer darauf sortieren**, nicht die Daten selbst herunkopieren!
- Alternative zum Sortieren: Daten in **Baum** einfügen, Baum durchlaufen:  
Im Mittel  $O(n * \log(n))$  auch bei unbalanzierem Baum,  
im Worst Case  $O(n^2)$  wenn unbalanziert,  $O(n * \log(n))$  wenn balanziert.

## 6 Verkettete Listen

### Unterscheide:

- **Einfach verkettete Listen** (jedes Element zeigt auf das nächste)
- **Doppelt verkettete Listen** (jedes Element zeigt auf das nächste und das vorige)

Immer: Pointer auf das **erste** Element (heißt "**Head**")

### Meist:

Pointer auf das **letzte** Element (heißt "**Tail**") zum schnellen Anhängen (sonst Durchlaufen notwendig!), erfordert aber einen Sonderfall beim Löschen / Einfügen!

### Typische Operationen:

- **Durchlaufen:** Typische C-**for**-Schleife:  
**for (elem \*p = head; p; p = p->next) ...**
- **Suchen:** Nur Sequentiell ( $O(n)$ )!
- **Einfügen:**
  - Vorne: Immer schnell!
  - Hinten: Nur mit Tail-Pointer schnell, sonst Durchlaufen nötig!
  - Irgendwo:  
Bei Einfach-Verkettung: Nur hinter dem aktuellen Element möglich (bei Doppelt-Verkettung: Auf beiden Seiten).  
==> Beim sortiert Einfügen in einfachverketteter Liste:  
Beim Durchsuchen Pointer auf das vorige Element merken!  
(Durchlauf mit 2 Pointern auf 2 nacheinanderfolgende Elemente!)
- **Löschen:**
  - Vorne: Immer schnell!
  - Hinten: Geht bei einfach verketteter Liste trotz Tail nicht ohne Durchlauf, da man ja das vorletzte Element braucht!
  - Irgendwo:  
In einfach verketteter Liste: Ebenfalls Pointer davor notwendig!  
Pointer auf zu löschendes Element reicht nicht!!!  
(nur bei doppelt verketteter Liste)
- **Prüfen ob leer**

### Vorteile Doppelverkettung:

- Durchlaufen in beide Richtungen möglich
- Löschen nur mit Element-Pointer allein möglich (bei Einfach-Verkettung braucht man dazu zusätzlich einen Davor-Pointer!)
- Einfügen davor / dahinter nur mit Element-Pointer allein möglich (bei Einfach-Verkettung nur dahinter möglich, nicht davor außer mit Davor-Pointer)

**Sortierung bringt bei Listen nichts beim Suchen (kein binäres Suchen möglich!),** (außer bei Skip-Listen: Listen mit zusätzlicher Verkettung jedes 2., 4., 8., ... Elementes) aber beim Abarbeiten der Reihe nach (kleinstes / größtes vorne).

### Sonderformen:

- **Stack: LIFO** (*Last In First Out*)  
Einfach verkettet, nur Head (kein Tail), Einfügen und Entnehmen nur **vorne**:  
    *“Top of Stack” = zuletzt eingefügtes Element = Head (vorne)*  
    *Verkettung von jüngeren zu älteren (jüngstes Element vorne)*  
Einfügen heißt *“Push”*, Entnehmen / Löschen heißt *“Pop”*
- **Queue: FIFO** (*First In First Out*)  
Einfach verkettet, Head & Tail, **Einfügen** nur am **Ende**, **Entnehmen** nur am **Anfang**:  
    *=> Ältestes Element vorne, Verkettung von älteren zu jüngeren*  
    *(umgekehrt hätte man ein Problem mit dem Entnehmen!)*  
Einfügen heißt *“Enqueue”*, Entnehmen / Löschen heißt *“Dequeue”*
- **Zyklische Liste:** Im Kreis verkettet (einfach oder doppelt)  
Oft mit *Dummy-Element* für leere Liste:  
Erlaubt Programmierung mit weniger **if**'s und Sonderfällen als bei NULL-Pointer, ist bzw. enthält zugleich Head und Tail!  
Beispiel Scheduler: Zyklische Liste, immer mindestens mit Idle-Prozess.
- **Selbstorganisierende Liste:**  
Bei jedem Suchen bzw. Zugriff: *Gesuchtes Element ganz nach vor verschieben*  
*=> Häufig benutzte Elemente werden schneller gefunden, weil sie vorne stehen, selten gesuchte Elemente wandern ganz nach hinten.*

### Umsetzung in C:

- Elementtyp als **struct** deklarieren: Eigentliche Nutzdaten + 1 oder 2 Pointer.
- Anlegen mit **malloc** (und **sizeof**), **free** nicht vergessen
- Next-Pointer am Ende sowie Head und Tail bei leerer Liste: **NULL**
- **Häufiger Trick** („\*\*-Trick“, d.h. Pointer auf Pointer) für den *“Davor”*-Pointer zum Einfügen / Löschen in einfachverketteter Liste:  
Zeigt nicht auf das vorige Element als Ganzes, sondern direkt auf dessen Next-Pointer (denn nur diesen braucht man zum Einfügen / Löschen).

#### Vorteile dieses Tricks:

- *Man spart sich die Fallunterscheidung* zwischen erstem / innerem Element, d.h. ob man ein Davor-Element hat, dessen Next man ändern muss, oder ob man den Head ändern muss:  
Ist man beim *ersten* Element, *zeigt der Davor-Pointer auf den Head-Pointer* statt auf den Next-Pointer des vorigen Elementes  
*=> Kein eigener Fall zum Ändern von Head beim Einfügen / Löschen ganz vorne, der Head wird genauso wie der Next-Pointer des Vorgängers über **\*davor** geändert.*
- *Man spart sich zwei getrennte Pointer* (voriges & aktuelles) beim Durchlauf, der Davor-Pointer allein genügt zum Durchlaufen:  
**\*\*davor** ist das aktuelle Element, kein eigener Pointer darauf notwendig!

### Umsetzung in C++:

Das vordefinierte STL-Template **list** implementiert doppelt verkettete Listen.  
Das vordefinierte STL-Template **forward\_list** (erst ab C++ 11) implementiert einfach verkettete Listen.

Stack (**stack**) und Fifo (**queue**) sind ebenfalls bereits als vordefinierte Templates verfügbar.

### Unterschiede zum Array:

- Vorteil: *Länge muss nicht vorher bekannt sein*
- Vorteil: *Belegt nur so viel Platz, wie wirklich Elemente da sind*
- Vorteil: *Einfügen / Löschen mittendrin in  $O(1)$  statt  $O(n)$*
- Nachteil: *Mehr Platzbedarf (für Pointer & **malloc**-Overhead)*
- Nachteil: *Mehr Zeitbedarf*
- Nachteil: *Kein Direktzugriff auf Element Nummer **i** ( $O(n)$  statt  $O(1)$ )*
- Nachteil: *Kein binäres Suchen (Suche dauert  $O(n)$  statt  $O(\log n)$ )*

### Wichtige Entscheidung vor allem bei Stack und Queue:

- Theoretisch sind Implementierungen mittels Array und mittels Liste in derselben Performance-Klasse:  $O(1)$ , d.h. fixe Zeit für Anhängen, Entnehmen usw. unabhängig vom Füllstand.
- Praktisch braucht die Liste um einen konstanten Faktor mehr Zeit und mehr Platz.
- Dafür muss beim Array schon beim Anlegen die maximale Größe festgelegt werden (vergrößern sehr aufwändig!), und dieser Speicher ist ständig belegt. Die Liste kann unbegrenzt wachsen und belegt nur soviel Speicher wie gerade nötig.

## 7 Hashing

Idee:

Aufteilen einer großen Datenmenge in viele kleine Datenmengen,  
weil kleine Datenmengen effizienter sind  
(vor allem: Schnell durchsucht werden können).

Name: "to hash": Zerhacken, zerkleinern (gleicher Wortstamm: Haschee = Hackbraten)

Deutscher Name: "Streuspeicherung" (veraltet): Daten auf viele Plätze „verstreuen“

In der Praxis:

Jede "kleine" Datenmenge enthält im Normalfall kein oder nur ein Element,  
in Ausnahmefällen einige wenige Elemente (sinnvollerweise  $\ll 10$ ).

==> Die kleinen Datenmengen können problemlos linear durchsucht werden,  
in im Normalfall konstanter Zeit.

==> Wenn das Ermitteln der "richtigen" Datenmenge auch konstante Zeit braucht,  
ist die Gesamtkomplexität für das Suchen (und das Einfügen, Löschen, ...)  
im Mittel gleich **O(1)**.

==> Worst case: Wenn alle Daten in einer einzigen Datenmenge (mit n Elementen) landen,  
ist die Komplexität **O(n)**!

Real-World-Äquivalent: Lagerhaltung mit erfahrener „Lagermeister“:

1. Schritt: Frage den Lagermeister: „In welchem Fach liegt Ersatzteil xyz?“  
==> „Teil xyz muss in Regal a, Fach b liegen“
2. Schritt: Durchsuche nur Regal a, Fach b  
(enthält maximal 3 verschiedene Teile ==> „Mit einem Blick überschaubar“)

### 7.1 Hashfunktion

Die "Hashfunktion" ist das "Orakel", das einem zu jedem Datenelement sagt,  
in welcher der Datenmengen das Element eingefügt / gesucht werden muss  
(im Beispiel oben: Der „Lagermeister“, der einem zu jeder Teilenummer  
sofort die richtige Fachnummer sagen kann.).

Sie ist eine Zuordnung aller möglichen Schlüsselwerte (meist String, selten Zahl, ...)  
auf den entsprechenden Index der Teil-Datenmenge (eine viel kleinere Zahl),  
wobei die Zielmenge (Anzahl der Datenmengen) viel kleiner als die Quellmenge  
(Anzahl aller möglichen Schlüsselwerte) ist (daher: Mehrere Schlüssel ==> selber Index).

**Qualitätskriterium der Hashfunktion:**

Gleichverteilung der Schlüssel auf die Datenmengen (Zielwerte)  
(==> alle Datenmengen möglichst gleich groß bzw. gleich klein!),

auch bei sehr ungleich verteilten Schlüsselwerten!

(deutsche Familiennamen oder Variablenamen eines großen Programms  
sind z.B. statistisch alles andere als gleichverteilt)

Eine Hashfunktion besteht normalerweise aus zwei Schritten:

1. **Umwandeln** des Schlüssels (meist Text) in eine Zahl (meist voller 32-bit-int-Wert)
2. „**Zusammenstauchen**“ der Zahl aus 1 auf den Index-Bereich des Arrays.

Zu 1.: Schlüssel ==> Zahl

Wichtig ist, dass **alle Zeichen** des Schlüssels (und auch alle Bits eines jeden Zeichens) in die Zahl eingehen. Problematisch dabei ist,

- dass die Schlüsselwörter normalerweise verschieden lang sind (beim Auffüllen auf gleiche Maximal-Länge ==> hinten Zwischenraum sehr häufig!) (bei Schlüssel mit nur 1-2 Zeichen ==> Übergewicht bei sehr kleinen Zahlen?!),
- und dass die ASCII-Werte der einzelnen Zeichen sehr ungleich verteilt sind, sowohl bei Betrachtung als Zahl (65-90, 97-122) als auch bei bitweiser Betrachtung (Kleinbuchstabe ==> 011..., Großbuchstabe ==> 010..., Zahl ==> 0011...)

Es gibt folgende Varianten:

- Schlüssel-Buchstaben mit Bit-Rotate und Xor verknüpfen (d.h. pro Zeichen zuerst die bisherige Zahl rotieren und dann das neue Zeichen mit Xor dazurechnen)

Wichtig:

So rotieren, dass sich die Zeichen wirklich bitweise überlappen, nicht nur byteweise (d.h. es soll nicht in jedem Byte der Zahl jedes vierte Zeichen des Inputs landen).  
=> Nicht um 8 oder 4 Bits rotieren, sondern z.B. um 5 oder 7 Bits!

Schlechter: Shift statt Rotate (weil bei langen Schlüsseln die Bits der vorderen Zeichen vorne aus der Zahl herausgeschoben werden, d.h. wegfallen!)  
Auch meist minimal schlechter: Add statt Xor

Ganz schlecht: nur Xor oder nur Add, ohne Rotate oder Shift (zu geringer Zahlenbereich, Bits extrem ungleich verteilt, Worte mit vertauschten Buchstaben haben gleichen Hashwert!)

- Alternative, vor allem für Schlüssel, die Zahlen sind: Quadrieren & die mittleren Bits nehmen (wenn doppelt lange Arithmetik möglich) oder 1-2 Schritte eines Pseudo-Zufalls-Generators (Multiplikation, Modulo) rechnen. Auch als Schritt pro Zeichen sinnvoll, wenn keine Bit-Operationen verfügbar sind.
- Kryptographische Hashfunktionen wären betreffend Verteilung optimal, sind aber viel zu rechen-aufwändig (= zu langsam)!

Zu 2.: Zahl ==> Index-Bereich

Praktisch immer: Zahl aus 1. mod Hashtable-Größe

Aber dabei ganz wesentlich: Wahl der Hashtable-Größe (Anzahl der Datenmengen)

Am besten: **Primzahl**

==> Verteilt ungleich verteilte Zweierpotenzen (Bits) und Zehnerpotenzen gleichmäßig.

Schnell aber ganz schlecht: Zweierpotenz (einfach hintere Bits der Zahl aus 1. nehmen) Denn: Vordere Bits fallen komplett weg (nur die letzten Schlüssel-Buchstaben „zählen“) und Ungleich-Verteilungen in den hinteren Bits bleiben erhalten!

## 7.2 Datendarstellung

Die Hashfunktion liefert die Nummer des „Topfes“ (der Datenmenge), in den der Wert gehört bzw. gesucht wird. In der Praxis immer: Hashwert = Array-Index.

Problem: Die Hash-Funktion kann für verschiedene Werte denselben Index liefern.

Das heißt „**Kollision**“ ==> Man bräuchte ein Array mit Platz für mehrere Werte pro Index.

Frage: Wie werden die Daten gespeichert, um auch kollidierende Werte unterzubringen?

- Alte Variante: Daten direkt im **Array**: Ein Array-Element = ein Datenwert  
Bei Kollision (Platz schon belegt): Weitersuchen bis zu einem freien Element im Array:  
Entweder der Reihe nach (ganz schlecht, führt zu lokalen Zusammenballungen!)  
oder mit zweiter (weiter verteilter) Funktion, meist Quadrat mod Größe.

Problem: **Kein Löschen möglich!!!**

Weil man nicht mehr sinnvoll weitersuchen kann,

sobald „Löcher“ vorhanden sind: Wann hört man mit der Suche auf?

==> Kollisionsopfer hinter dem Loch sind verloren...

- Neue Variante: **Array + n einfach verkettete Listen**

(eine pro Array-Element bzw. Hash-Wert).

Jedes Array-Element enthält den Head-Pointer „seiner“ Liste

Die eigentlichen Daten werden einzeln dynamisch angelegt und stehen außerhalb des Arrays in den (meist unsortierten) Listen.

Vorteil: Löschen möglich, unempfindlicher gegen Kollisionen / zu kleine Hashtable

Nachteil: Mehr Platz und Aufwand nötig (**malloc** / **free** für Listenelemente)

Die Hashtable muss auf „alle Elemente leer“ initialisiert werden (z.B. lauter **NULL**-Heads)!

Optimale Größe:

- Bei Kollisions-Arrays: **Füllstand** < 65 %  
(d.h. die Hashtable muss deutlich größer als die max. Anzahl der Elemente sein)  
Denn: Such- und Einfüge-Aufwand wächst exponentiell mit dem Füllstand!
- Bei Hashtable + Listen: Mittlere Listenlänge am besten < 3, max. < 10  
(d.h. die Hashtable darf etwas kleiner als die max. Anzahl der Elemente sein)

Bei zu hohem Füllgrad ev. **reorganisieren**

(umkopieren auf größere Hashtable durch Neuberechnen aller Hash-Werte)!

## 7.3 Vor- und Nachteile

Vorteile:

- Viel einfacher zu implementieren als Bäume: Suchen und Einfügen sind trivial.  
Bei Hashtables mit Listen: Reihenfolge innerhalb der Listen ist egal  
==> neue Elemente einfach immer vorne oder immer hinten anhängen.
- Zugriffszeit im Wesentlichen größen-unabhängig,  
deutlich schneller als Bäume bei großen Datenmengen  
(bei 1 Mio. Elemente: Baum >= 20 Vergleiche, Hash im Mittel 1-3 Vergleiche)
- Bei guter Hashfunktion völlig unempfindlich gegen schon sortierten Input  
 („hängt nicht schief“ wie bei Bäumen).

Nachteile:

- Die Daten sind nicht sortiert und haben keinen Vorgänger / Nachfolger  
**==> Sortiertes Durchlaufen oder Ausgeben ist unmöglich!**
- Die Performance sinkt linear (Listen) bzw. exponentiell (In-Place Array) bei zu vielen Datenelementen (mittlere Anzahl pro Hashwert  $> 2$  bzw.  $> 0.65$ ).
- Bei alter Variante (In-Place Array): Kein Löschen möglich!
- Mit viel Pech (schlechte Hashwert-Verteilung) sehr langsam (in der Praxis bei guter Hashfunktion nie...).

## 8 Bäume

= Spezialfall eines

**zusammenhängenden, zyklenfreien, gerichteten Graphen:**

- Genau ein Knoten ohne eingehende Kante = **Wurzel**
- Alle anderen Knoten mit genau einer eingehenden Kante

(oder umgekehrt)

Übliche Darstellung: Von oben nach unten (Wurzel oben!) oder von links nach rechts.

**Begriffe:**

- Wurzel / innerer Knoten / Blatt (Endknoten)
- Sohn (engl. auch Child), Vater (Parent)  
=> Wurzel ist einziger Knoten ohne Vater
- (linker / rechter) Teil- oder Unter-Baum (Subtree)
- Knotengrad, bei Knotengrad max. 2 => "Binärbaum"
- Tiefe (manchmal auch: Höhe) eines Knotens  
= Anzahl der Kanten von der Wurzel (je nach Definition: + 1 ?)

Maximale Tiefe (manchmal auch: Höhe) des Baumes  
= Tiefe des tiefsten Blattes (im Mittel / im Worst Case)

Mittlere Tiefe (manchmal auch: Höhe) des Baumes  
= ??? Mittelwert der Tiefe aller Knoten oder aller Blätter

- Balancierter Baum:  
Streng: Alle Blätter haben selbe Tiefe  
In der Praxis: Tiefe und Knotengrad erfüllen bestimmte Regeln  
(z.B. Tiefe +1 oder max. \*2, Knotengrad 0 oder 2 außer in vorletzter Ebene, ...)
- Vollständiger Baum:  
Balancierter Baum, bei dem alle nicht-Blätter Grad n (bzw. 2) haben

**Rekursive Definition:**

**Ein Baum ist entweder leer  
oder besteht aus einem Knoten  
mit genau n (ev. leeren) Bäumen als Söhnen.**

**n = 2: Binärbaum**

Die lineare Liste ist ein "unärer Baum" (Knotengrad 1)  
bzw. der Extremfall eines unbalanzierten (Binär-)Baumes.

**Sortierung bei binären Suchbäumen:** Für alle Knoten x im Baum gilt:

- Alle Knoten im linken Teilbaum von x sind kleiner als x
- Alle Knoten im rechten Teilbaum von x sind größer als x

=> offensichtlicher Such-Algorithmus!

==> Die Tiefe entspricht der Anzahl der Such-Schritte (Vergleiche):

**Ein Binärbaum der Tiefe n  
hat max.  $2^n-1$  Knoten  
( $2^{i-1}$  Knoten auf Ebene i)**

==> Für **n Knoten** braucht man im Mittel mindestens  
und bei balancierten Bäumen im Mittel und im Maximum auch höchstens

**$O(\log_2(n))$  Schritte**

Andere Sonderfälle der Sortierung (Heap, Priority Queue ==> später):  
Jeder Knoten ist kleiner bzw. größer als alle seine Söhne.

Weitere wichtige Operation: **Durchwandern**

- **Breadth first** = Ebene für Ebene  
Schwierig! (benötigt auf den ersten Blick eine dynamische Hilfs-Datenstruktur mit  $O(n)$  Elementen im worst Case, siehe Breadth-first Durchlauf von Graphen)
- **Depth first** = Unterbaum für Unterbaum, zuerst in die Tiefe  
Typischerweise rekursiv!
  - **inorder**: Linker Unterbaum – Knoten – Rechter Unterbaum  
Ergebnis: **Sortierte** Aufzählung aller Knoten
  - **preorder**: Knoten – Linker Unterbaum – Rechter Unterbaum
  - **postorder**: Linker Unterbaum – Rechter Unterbaum – Knoten  
Beispiel: Taschenrechner, Compilerbau

**Anwendungen von Bäumen:**

- Suchen und Sortieren:  
Im Normalfall: Baum möglichst flach halten!  
Bei bekannten Such-Wahrscheinlichkeiten:  
Knoten nach Wahrscheinlichkeit anordnen  
(häufig gesuchter Wert ==> kurzer Pfad zum Knoten)!  
Siehe balancierte Bäume (demnächst), Datenbanken (= B-Bäumen, demnächst).
- Darstellung von arithmetischen Ausdrücken und Programmen, Compilerbau:  
RPN entspricht dem Rechnungs-Baum in Postorder-Notation  
Programmiersprache Lisp!
- Spielbaum (Aufzählung aller Möglichkeiten / Kombinationen)
- Spanning Tree (Verbindung aller Knoten, meist optimiert nach Kantengewicht)
- Hierarchisch strukturierte Daten (z.B. Verzeichnisbaum)

**Speicherung:**

- Verpointerte Knoten (Left- und Right-Pointer, mit / ohne Vater-Pointer)
- Bei Knotengrad  $> 2$  ev. auch als Bruder-Liste  
(Vorteil: Konstante Größe eines Knotens / Nachteil: Langsamer, \*  $O(k)$  )
- Linearisiert im Array: Die Söhne des Knotens i liegen in den Elementen  $2^i$  und  $2^{i+1}$

## **Terminologie beim Kopieren / beim Vergleichen:**

- **Pointer Copy / Compare:**

Der Pointer auf den obersten Knoten wird kopiert.

Die Pointer auf die obersten Knoten werden verglichen (d.h. es muss de facto derselbe Baum sein).

- **Shallow Copy / Compare:**

Der oberste Knoten wird frisch angelegt und der Inhalt kopiert, incl. der Pointer auf die Söhne (= die Söhne sind für beide Bäume gemeinsam).

Die beiden obersten Knoten werden elementweise verglichen, incl. der Pointer auf die Söhne (= beide Wurzeln müssen auf dieselben Söhne zeigen).

- **Deep Copy / Compare:**

Alle Knoten werden rekursiv kopiert,

d.h. die Kopie ist ein eigenständiger, unabhängiger Baum.

Alle Knoten werden rekursiv wertmäßig verglichen.

## **Grundoperationen (unbalanciert):**

- **Einfügen:**

Suchen und dort, wo der Wert sein müsste, aber ein NULL-Pointer ist, anhängen.  
=> Neue Elemente werden immer ein Blatt!

Worst case: In bereits sortierter Reihenfolge einfügen

=> Baum wird zur Liste, Tiefe  $n$  bei  $n$  Werten!

Average Case über alle möglichen Verteilungen der Insert-Reihenfolge:

rund  $1,4 * Best Case$  (also nicht gar so schlecht, immer noch  $O(\log(n))$ )

Notwendig zum Einfügen eines Wertes (wie bei einer Liste):

Pointer auf den Vater-Knoten und Flag, ob links oder rechts angehängt wird.

Trick zum einfacheren Einfügen (spart Fallunterscheidung "als Root / als linker Sohn / als rechter Sohn"), analog zum \*\*-Trick bei Listen: Pointer auf den Pointer, in dem das neue Element einzutragen ist (der Pointer kann auf root oder auf die left- oder right-Verkettung im Vaterknoten zeigen).

- **Löschen:**

- **Trivialfälle:**

Blatt => einfach weglöschen

Nur 1 Sohn => Sohn direkt an den Vater anhängen

- **Nichttrivialer Fall:**

2 Söhne => Knoten ersetzen durch größtes Element des linken Unterbaumes (oder kleinstes Element des rechten Unterbaumes)

Der "heraufkopierte" Knoten hatte ursprünglich immer 0 oder 1 Söhne (nie 2!) und kann daher trivial gelöscht werden.

Finden des heraufzukopierenden Elementes (größtes Element links):

Ein Mal links und dann immer rechts bis kein rechter Sohn mehr da ist (oder ein Mal rechts, dann immer links).

In beiden Fällen: “free” bzw. “delete” des gelöschten Knotens nicht vergessen!

### **Vordefinierte Bäume in C++:**

Auch wenn nicht formal gefordert ist, dass die dahinterstehende Datenstruktur ein Baum ist, sind die STL-Datenstrukturen **set**, **multiset**, **map** und **multimap** intern normalerweise als balancierte Binärbäume implementiert.

## **8.1 Die Balancierung**

Ziel:

Möglichst flacher Baum (balanciert und möglichst voll!)  
zum schnelleren Suchen.

Abfangen des Worst Case  $O(n)$  (“Baum hängt schief”):

Balanciert ==> auch Worst Case  $O(\log(n))$ .

Aber: Balancieren ist sehr aufwändig (kann jedesmal sehr viele Elemente betreffen!)

==> möglichst selten machen, nicht nach jedem Einfügen / Löschen

==> eine gewisse Abweichung vom optimalen Baum wird toleriert!

Im Fall zu starker Unbalance wird immer nur lokal korrigiert,  
nicht der gesamte Baum reorganisiert!

Heute gebräuchlich:

- **AVL-Baum:**

In jedem Knoten des Baumes gilt:

Der **Unterschied** zwischen der **Höhe** des linken und des rechten Subbaumes  
ist **maximal 1**

==> die Höhe ist kleiner  $1.44 * \log_2(n+2) - 1$

(worst Case des AVL-Baumes ist in etwa average Case des unbalancierten Baumes)

==> Average Case ist annähernd optimal! (typischerweise  $\leq 1$  höher als Optimum)

Jeder Knoten hat “Balance“-Feld: +1, 0, -1

Bei Insert:

Auf dem Weg aus der Rekursion zurück: Returnwert “Höhe hat zugenommen”

==> Balance entsprechend aktualisieren.

Wenn die Balance dabei +-2 wird:

**Drei Knoten samt Sub-Bäumen umhängen (“Rotation”, siehe Internet!)**

Max. eine Rotation pro Insert reicht!

**2 Rotations-Grundmuster**, jeweils in 2 spiegelbildlichen Ausprägungen

==> 4 verschiedene Rotationstypen:

- Eine Seite der Wurzel ist um 2 zu hoch,  
der höhere Teilbaum ist entweder balanciert oder außen um 1 höher  
==> “Einfachrotation”, die Wurzel des höheren Teilbaumes wird neue Wurzel
- Eine Seite der Wurzel ist um 2 zu hoch,  
der höhere Teilbaum ist innen um 1 höher  
==> “Doppelrotation”, die Wurzel des zu hohen mittleren Teil-Teilbaumes  
wird neue Wurzel (wandert 2 Ebenen hinauf!)

### Bei Delete:

Analog, Returnwert "Höhe hat abgenommen",  
aber schlimmstenfalls  $\log(n)$  Rotationen pro Delete notwendig  
(eine auf jeder Ebene auf dem Weg nach oben).

### • **Red-Black-Tree:**

- Knoten sind entweder "rot" oder "schwarz".
- Die nicht vorhandenen NULL-Knoten zählen als "schwarz".
- Die Wurzel ist "schwarz".
- Beide Söhne eines "roten" Knoten sind "schwarz".  
( $\Rightarrow$  es folgen nie zwei "rote" Knoten aufeinander)
- In jedem Knoten gilt: Alle Pfade von diesem Knoten zu untergeordneten NULL-Knoten haben gleichviele "schwarze" Knoten.

Daraus folgt:

Der längste Pfad von der Wurzel zu einem Blatt ist  
maximal doppelt so lange wie der kürzeste Pfad  
(weil alle Pfade  $n$  schwarze und zwischen 0 und  $n$  rote Knoten haben)

$\Rightarrow$  die Höhe ist kleiner  $2 * \log_2(n+1)$

$\Rightarrow$  Der Red-Black-Tree ist beim Suchen etwas schlechter als der AVL-Baum  
(weil er im Schnitt etwas höher ist), aber beim Einfügen und Löschen deutlich besser  
als der AVL-Baum (weil er mehr Ungleichgewicht toleriert und daher viel  
seltener rotiert werden muss).

$\Rightarrow$  Red-Black-Tree ist besser für „änderungsintensive“ Bäume

$\Rightarrow$  AVL-Tree ist besser für „leseintensive“ Bäume

Insert und Delete mit Rotationen im Prinzip ähnlich wie beim AVL-Tree,  
aber mehr verschiedene und kompliziertere Fälle ( $\Rightarrow$  Code ist wesentlich  
umfangreicher).

Red-Black-Trees sind mit Vater-Pointern deutlich leichter zu implementieren.

### • **2-3-Bäume:**

Sind keine Binärbäume, sondern eine Sonderform von B-Bäumen:

- Jeder Knoten enthält 1 oder 2 Werte und hat demnach 2 oder 3 Söhne  
(oder gar keine, wenn es ein Blatt ist, aber nie nur einen Sohn!).
- Alle Blätter sind gleich hoch.

### Insert & Delete:

Knoten werden bei Insert bei Bedarf gespalten und bei Delete zusammengelegt

$\Rightarrow$  Baum wächst / schrumpft an der Wurzel! (siehe unten: B-Baum)

Speicherung oft als binäre Knoten (nur 1 Wert, 2 Pointer) mit "Bruder"-Flag

$\Rightarrow$  Entspricht normalem Binärbaum mit max. Höhenunterschied um Faktor 2.

Normale Baumoperationen (Suchen usw.) sind komplizierter als bei Binärbäumen,  
aber das Einfügen / Löschen / Rotieren ist etwas einfacher als bei Red-Black-Trees.

2-3-Bäume kommen heute nur mehr als theoretische Beispiele vor.

Ihr Verhalten entspricht in etwa den Red-Black-Trees.

## 9 Sonderformen von Bäumen

### 9.1 Der Trie

= Zeichenweiser / ziffernweiser Suchbaum für Daten,  
deren Suchschlüssel eine Zeichenkette bzw. Ziffernfolge ist:

- Jede Ebene des Baumes entspricht einer Stelle des Schlüssels:  
Die Wurzel entspricht dem leeren Wort und verzweigt nach dem ersten Zeichen des Schlüssels, die erste Ebene darunter hat einen Knoten für jedes an erster Stelle vorkommende Zeichen und verzweigt nach dem zweiten Zeichen usw..  
Die Tiefe eines Wortes im Baum entspricht daher seiner Wortlänge.
- Jeder Knoten enthält 26 (oder 10 oder wie groß der Zeichensatz eben ist) Sohn-Pointer (als Array of Pointers):  
Es wird daher nie wirklich gesucht und nie der gesamte Schlüssel verglichen, sondern auf jeder Ebene einfach nur das entsprechende Zeichen des Schlüssels als Index in das Array der Söhne verwendet ('a': 1. Sohnpointer, 'b': 2. Sohnpointer, ...).  
Gibt es kein Wort im Baum, dessen nächster Buchstabe das i-te Zeichen im Zeichensatz ist, ist der i-te Sohnpointer NULL.
- Der Suchschlüssel, zu dem ein Knoten gehört, braucht nicht explizit im Knoten gespeichert zu werden: Er ergibt sich aus dem Weg von der Wurzel zum Knoten.  
Üblicherweise ist jedoch noch in jedem Knoten ein Pointer auf die Nutzdaten jenes Schlüssels vorhanden, der zum Knoten gehört (d.h. die Daten jenes Wortes, das in diesem Knoten endet). Endet kein Wort in diesem Knoten, sind die Nutzdaten bzw. der Pointer darauf leer, der Knoten enthält nur Sohnpointer.

#### Vorteile:

- Sehr schnell beim Suchen und Einfügen, unabhängig von der Zahl der Elemente, nur abhängig von der Wortlänge.
- Keine String- oder Vergleichsoperationen, nur Zugriff auf einzelne Zeichen und Index-Operationen.

#### Nachteil:

- Verschwendet sehr viel Platz:
  - Die Knoten sind sehr groß  
Z.B. bei Zeichensatz Groß- & Kleinschreibung + Ziffern:  
Rund 70 Pointer pro Knoten!
  - Statistisch über den ganzen Trie gemittelt ist pro Knoten genau ein einziger dieser Pointer ungleich NULL, d.h. im Beispiel oben sind 98.7 % aller Pointer NULL!
  - In allen Knoten, die Endknoten (Blätter) sind, sind alle Pointer NULL ==> Optimierung: Eigener Knotentyp für Blätter (ohne Sohnpointer).
  - Es gibt viele Zwischenknoten, die keinem eigenen Wort entsprechen (bei typischem Text: 3-7 Mal mehr Knoten als echte Wörter).

### Sortiertes Durchlaufen:

- Sind die Schlüssel *alphabetisch sortiert*, ist das Durchlaufen in Schlüsselreihenfolge (rekursiv) problemlos und effizient möglich.
- Bei numerisch sortierten, ziffernweisen Schlüsseln ist das Durchlaufen in Schlüsselreihenfolge nur dann möglich, wenn alle Schlüssel gleich viele Ziffern haben (sonst muss man vorne mit 0 auf fixe Länge auffüllen).

## 9.2 Der Heap (Priority Queue)

*Nicht verwechseln mit dem Heap in der Speicherverwaltung!*

- Vollständiger, balancierter Binärbaum.
- Fast immer *als Array gespeichert*:  
Index beginnt bei 1 (nicht 0!), Söhne von  $n$  sind  $2*n$  und  $2*n + 1$ , Vater ist  $n/2$ .
- Sortierkriterium:

**Jeder Knoten ist kleiner (oder größer)  
als alle Knoten in seinen Unterbäumen  
==> Kleinstes (größtes) Element ist an der Wurzel!**

### Vorteile:

- Kleinstes (oder größtes) Element suchen ( $O(1)$ ) und entnehmen ( $O(\log(n))$ ) ist schnell, neues Element einfügen ist schnell ( $O(\log(n))$ ).  
Auch andere Einzel-Element-Operationen sind alle  $O(\log(n))$ .
- Keine dynamische Speicherverwaltung notwendig.
- Sehr geringer (optimaler) Platzbedarf: Keine Pointer, kein `malloc`-Overhead, ...

Anwendungen: Immer dann, wenn man **nur das kleinste / größte Element** sucht!

- Scheduler (Jobs nach Priorität: Wichtigster bzw. nächster ist vorne)
- Timelist (Events bzw. Timer nach Zeitpunkt: Nächster ist vorne)
- Heapsort:
  - Erster Schritt: Input-Array in einen Heap verwandeln (mit größtem Element vorne).
  - Zweiter Schritt: Immer wieder größtes Element entnehmen und von hinten nach vorne ins Ergebnis-Array stellen.

Operationen: (angegeben für einen Heap mit größtem Element vorne)

- Einfügen:
  - Element *hinten anhängen* (in den ersten freien Array-Platz stellen ==> Array wird eins länger).
  - Mit einer Schleife über  $i/2$  an die richtige Stelle *nach vorne wandern lassen*: Solange es größer ist als der Vater: Mit dem Vater vertauschen.
- Größtes Element entnehmen:
  - Erstes Element entnehmen.

- Letztes Element an die erste Stellen setzen (==> Array wird eins kürzer).
- ... und an die richtige Stelle nach hinten wandern lassen:  
Solange es kleiner als einer der beiden Söhne ist:  
Mit dem größeren der beiden Söhne vertauschen.
- Beliebiges Element entnehmen:  
Wie beim Entnehmen des ersten Elementes:  
Entnommenes Element durch letztes Element ersetzen (Heap wird kürzer)  
und dieses mit Vater oder Söhnen tauschen, bis es an der richtigen Stelle steht.  
Achtung: Das Element kann in beide Richtungen wandern, vor oder zurück!
- Aus einem unsortierten Array einen Heap erzeugen:
  - Die hintere Hälfte ist schon ein Heap (weil die Elemente keine Väter / Söhne voneinander sind und daher keine Ordnung zueinander haben).
  - Das Array von der Mitte bis zum Anfang elementweise durchgehen:  
Jedes Element einzeln an die richtige Stelle wandern lassen  
(nach hinten durch Vergleich mit den beiden Söhnen, wie beim Entnehmen).
 Die Laufzeit ist besser als auf den ersten Blick vermutet:  $O(n)$ , nicht  $O(n \cdot \log(n))$ .
- Position eines Elementes im Heap korrigieren:  
Entweder entnehmen und geändert wieder einfügen  
oder mit Vater und Söhnen vergleichen und bei Bedarf vertauschen (effizienter!).

In C++ bietet die STL bereits ein vordefiniertes Template für Priority Queues, leider kann es nicht die Position eines Elementes korrigieren und auch nicht mittendrin Elemente löschen und ist daher nicht für alle Heap-Anwendungen (z.B. Graph-Algorithmen von Prim oder von Dijkstra) brauchbar.

Es gibt auch Heaps, denen andere Datenstrukturen als binäre Bäume zu Grunde liegen. Das Ziel ist, neben den oben genannten Operationen vor allem ein effizientes Vereinigen zweier Heaps zu unterstützen (im Idealfall in  $O(1)$ , also konstanter Zeit).

### 9.3 B-Bäume

- Alle Blätter sind gleich hoch  
==> der Baum ist daher balanciert.
- Jeder Knoten enthält zwischen  $n$  und  $2 \cdot n$  (jeweils einschließlich) Werte  
in sortierter Reihenfolge, nur die Wurzel darf auch weniger haben  
==> die Speicherausnutzung und der Baum-Füllgrad ist mindestens 50 %.
- Jeder Knoten außer der Wurzel hat daher entweder keine (Blatt)  
oder zwischen  $n + 1$  und  $2 \cdot n + 1$  Söhne.
- Der erste Unterbaum enthält die Werte kleiner dem ersten Wert, der letzte Unterbaum enthält die Werte größer dem letzten Wert, der  $i$ -te Unterbaum enthält die Werte zwischen dem  $(i-1)$ -ten und dem  $i$ -ten Wert.

Anwendung meist nicht im Speicher, sondern bei **Datenbanken** auf Disk:

**1 Knoten = 1 Disk-Block** fixer Größe (häufig 4 KB oder 32 KB)

Suchen ist trivial (bei großen Knoten: Innerhalb eines Knotens binär suchen!)

und schnell:  $O(\log_n(\text{size}))$  bei minimalem Knotengrad  $n$  und  $\text{size}$  Elementen insgesamt.

Beispiel  $n = 100$  (zwischen 101 und 201 Söhnen), Baum mit 4 Ebenen:  
Baum enthält minimal 2.060.601 und maximal 1.632.240.800 Werte,  
diese können mit 4 Disk-Zugriffen (4 Knoten) durchsucht werden!

#### Einfügen:

- Richtige Blatt-Stelle suchen und dort einfügen.
- Knoten hat immer noch  $\leq 2 \cdot n$  Werte:  $\Rightarrow$  Ok, fertig!
- Knoten hat  $2 \cdot n + 1$  Werte ("zu viel"):
  - Aufteilen in 2 Knoten mit je  $n$  Werten.
  - Mittlerer Wert wandert hinauf in den Vater  $\Rightarrow$  Vater wird 1 Element größer.
  - Rekursiv hinauf wiederholen:  
Wenn auch der Vaterknoten übergeht: Teilen, 1 Element weiter hinauf.
  - Eventuell wächst der Baum an der Wurzel in die Höhe:  
Wurzel zweiteilen, eine neue Wurzel darüber mit einem Wert anlegen.

#### Löschen:

- Löschen eines Blatt-Wertes: Einfach.
- Löschen eines inneren Wertes:  
Durch nächstkleineren / nächstgrößeren Blatt-Wert ersetzen und diesen löschen  
(Blatt-Wert "heraufholen" wie beim Löschen im Binärbaum).
- Wenn das Blatt dadurch nur mehr  $n - 1$  Werte ("zu wenig") enthält:
  - Mit einem der Nachbar-Blätter ausgleichen,  
anderen Teilungswert zwischen den beiden Blättern im Vater speichern.
  - Wenn auch die Nachbar-Blätter beide nur  $n$  Knoten haben:  
Mit einem der beiden Nachbarn zusammenlegen,  
Teilungswert aus dem Vater herunterziehen (Vater wird 1 Element kleiner)  
 $\Rightarrow$  Zusammengelegter Knoten hat  $(n - 1) + n + 1 = 2 \cdot n$  Werte
  - Wenn der kleiner gewordene Vater dadurch auch unter  $n$  Werte sinkt:  
Rekursiv nach oben wiederholen.
  - Eventuell fällt dadurch die Wurzel weg (wenn sie nur mehr 1 Wert hatte)  
 $\Rightarrow$  Baum schrumpft oben.

#### Zusätzliche Operation „Bulk Load“:

Rascher Aufbau eines B-Baums aus geordnetem Input mit allen Datensätzen:  
Daten sequentiell in Blätter füllen, dabei jeden  $(2 \cdot n + 1)$ -ten Datensatz zur Seite legen.  
Dann aus den zur Seite gelegten Datensätzen inneren Baum aufbauen.

#### Viele Varianten:

Zusätzliche „sparse Indices“, zusätzliche Verlinkung,  
spalten & vereinigen „auf Verdacht“ schon beim Abstieg, ...

#### B+-Baum:

- Wurzel, innere Knoten und Blätter enthalten nur Schlüssel und Pointer,

die eigentlichen Nutzdaten sind in eigenen Blöcken unter den Blättern gespeichert

- Jeder Schlüssel steht in einem Blatt, Wurzel und innere Knoten enthalten Duplikate der zum Suchen notwendigen Schlüssel
- Blätter sind zusätzlich in Sortierreihenfolge als Liste verlinkt (zum geordneten Durchlaufen aller Daten)

==> Weniger Platzbedarf pro Datensatz in den Baumknoten (nur Schlüssel)

==> Sehr hoher Knotengrad (typisch  $\geq 100$ ) in den inneren Knoten

==> Sehr flacher Baum (= wenig Zugriffe) auch für große Datenmengen

Häufigste Datenstruktur für Filesysteme und Datenbanken!

### **B\*-Baum:**

Datenverschiebung mit beiden Nachbar-Knoten bei zu vollem / zu leerem Knoten  
Split/merge macht drei 2/3-volle Knoten aus zwei vollen Knoten und umgekehrt

==> Minimaler Füllstand eines Knotens  $2/3$  statt  $1/2$

==> Platz- und Tiefenersparnis

## 10 Graphen

### Terminologie:

- Knoten und Kanten  
englisch: “Vertex” (bei Mathematikern, ev. “Node” bei Informatikern) und “Edge”
- Grad eines Knotens: Anzahl der Kanten eines Knotens  
(bei gerichteten Graphen: eingehender / ausgehender Knotengrad)
- Benachbarte Knoten: Knoten, die direkt durch eine Kante verbunden sind
- gerichtet / ungerichtet  
(haben die Kanten eine Richtung?)
- zusammenhängend / nicht zusammenhängend
- k-fach kantenzusammenhängend / k-fach knotenzusammenhängend:  
Man kann (k-1) beliebige Kanten / (k-1) beliebige Knoten weglöschen,  
ohne dass der Graph auseinanderfällt.
- zyklenfrei / nicht zyklenfrei
- planar / nicht planar  
(lässt sich der Graph in einer Ebene ohne Kantenüberkreuzungen zeichnen?)
- gewichtet (jeder Kante ist ein Wert zugeordnet) / ungewichtet
- Vollständiger Graph: Graph mit allen möglichen  $n \cdot (n-1) / 2$  Verbindungen

### Darstellung:

- Adjazenzmatrix:
  - Zweidimensionales quadratisches Array
  - Anzahl der Zeilen und Anzahl der Spalten = Anzahl der Knoten,  
Zeilen und Spalten sind mit den Knoten als Index beschriftet
  - Werte im Array-Element [x] [y]:  
0...keine Kante / 1...Kante zwischen Knoten x und y
  - Bei gewichteten Graphen: Elemente enthalten Gewicht statt 0 und 1
  - Bei ungerichteten Graphen symmetrisch zur Diagonale ==> Dreieck genügt!

“Sind x und y verbunden?”: Direkter Zugriff, Laufzeit  $O(1)$  (also sehr effizient)

“Zähle alle Nachbarn von x auf”: Laufzeit  $O(n)$  mit  $n$  = Gesamtknotenzahl  
==> effizient nur bei ziemlich dichten Graphen!

Speicherbedarf:  $O(n^2)$  ==> effizient nur bei ziemlich dichten Graphen!

- Adjazenzlisten:
  - Eindimensionales Array von Listen, Anzahl der Listen = Anzahl der Knoten
  - Pro Knoten eine lineare, verkettete Liste der Nachbarn.

- Bei gerichteten Graphen: Überlegen: Was braucht der Algorithmus?!  
Liste nur der ausgehenden Kanten / Nachbarn? Nur der eingehenden?  
Zwei getrennte Listen für beides nötig?
- Bei gewichteten Graphen: Jeder Listenknoten enthält auch das Gewicht.

“Sind x und y verbunden?": Liste durchlaufen, Laufzeit  $O(m)$  mit  $m = \text{Knotengrad}$   
 $\implies$  effizient nur bei kleinem  $m$ , d.h. dünnen Graphen!

“Zähle alle Nachbarn von x auf": Analog: Liste durchgehen,  $O(m)$   
 $\implies$  Bei kleinem  $m$  besser als Adjazenzmatrix, bei großem  $m$  schlecht!

Bei gerichteten Graphen in der „falschen“ Richtung  
(d.h. z.B. „Wer sind alle meine Vorgänger“ bei Liste der Nachfolger):  
Alle Kanten aller Knoten durchsuchen:  $O(m * n) = O(\text{Gesamtzahl Kanten})$

Speicherbedarf:  $O(n * m)$ , aber mit Listen-Overhead  
 $\implies$  Analog: Gut bei dünnen Graphen!

### Algorithmen (Auswahl):

- **Topologisches Sortieren:**  
Ziel: Eine lineare Anordnung der Knoten eines gerichteten Graphen finden, sodass alle Kanten von links nach rechts zeigen.  
Ist zugleich ein Test für Zyklenfreiheit gerichteter Graphen.
- **Durchlaufen aller Knoten, Breadth-First oder Depth-First.**
- **Minimum Cost Spanning Tree** (z.B. Kruskal-Algorithmus):  
Ziel: Die billigste Kantenmenge finden, die alle Knoten verbindet.  
Ist zugleich ein Test für zusammenhängend.
- **Single-Source Shortest Path** (z.B. Dijkstra-Algorithmus),  
**All-Pair Shortest Path** (z.B. Algorithmus von Warshall bzw. Floyd):  
Ziel: Finden der kürzesten Verbindungen von einem Startknoten aus.
- **Hamiltonian-Path-Problem:**  
Ziel: Rundreise durch alle Knoten finden (jeden Knoten nur ein Mal besuchen).  
NP-vollständig!  
Erweiterung: **Travelling Salesman** (Rundreise mit minimalen Kosten)
- **Euler'sches Brückenproblem:**  
Ziel: Rundreise über alle Kanten.
- Tests auf **graphentheoretische Eigenschaften:**  
Planar, k-fach zusammenhängend, ...
- **Färbe-Problem:** (NP!)  
Ziel: Minimale Anzahl von Farben für die Knoten finden, sodass keine benachbarten Knoten gleich gefärbt sind.
- **Maxflow:**  
Kantengewichte sind Leitungskapazitäten.  
Ziel: Den maximalen Durchsatz zwischen zwei Knoten finden.
- Algorithmen auf **Netzplan-Graphen:** (Darstellung von Projekten / Teilschritten):  
Minimale Projektdauer, kritischer (Gesamtzeit-bestimmender) Pfad,...

## 10.1 Kurzbeschreibung einzelner Algorithmen

### Durchlaufen, Depth First:

*Rekursiv*, wie beim Baum, aber zusätzlich in jedem Knoten mit einem Flag "visited":  
In jedem Knoten wird die Durchlauf-Funktion rekursiv für all jene Söhne aufgerufen, die noch nicht "visited" sind.

*Iterative Variante* (ohne Rekursion) siehe nächster Punkt.

### Durchlaufen, Breadth First:

Benötigt eine Queue (FIFO) und ein "visited"-Feld in jedem Knoten:  
"Visited" wird gesetzt, sobald ein Knoten in die Queue gestellt wird.

*Ausgangsknoten in die Queue stellen.*

*Solange die Queue nicht leer ist:*

*Ersten Knoten aus der Queue entnehmen.*

*Alle Nachbarn, die nicht "visited" sind, auf „visited“ setzen + hinten in die Queue stellen.*

Wenn man statt der Queue einen Stack verwendet, ergibt sich ein Depth-First-Durchlauf.

### Topologisches Sortieren: (Ziel: Gib jedem Knoten eine Reihungs-Zahl)

*Berechne für jeden Knoten die Anzahl der Vorgänger.*

*Solange noch Knoten ohne Reihungs-Nummer existieren:*

*Suche einen Knoten mit Vorgänger-Anzahl 0 (wenn keiner existiert: Graph ist zyklisch!)*

*Gib ihm die nächste Reihungs-Nummer.*

*Erniedrige die Vorgänger-Anzahl aller seiner Nachfolger um 1.*

### Floyd / Warshall:

*Schleife über alle Knoten für den mittleren Knoten M*

*Schleife über alle Knoten für den Anfangsknoten A*

*Schleife über alle Knoten für den Endknoten B*

*Wenn der Weg  $A \rightarrow M \rightarrow B$  kürzer ist als die direkte Entfernung  $A \rightarrow B$ :*

*Trage die Länge von  $A \rightarrow M \rightarrow B$  als neue Entfernung  $A \rightarrow B$  ein*

*Achtung: Die Reihenfolge der Schleifen ist essentiell für das korrekte Funktionieren:*

*Die Schleife für den mittleren Knoten muss die äußerste sein!*

### Dijkstra:

Für jeden Knoten wird zusätzlich ein Distanz-Wert (vom Ausgangs-Knoten) und eine Markierung benötigt.

*Initialisiere die Distanz des Ausgangsknotens auf 0, alle anderen auf unendlich.*

*Solange es noch unmarkierte Knoten gibt:*

*Suche den unmarkierten Knoten mit der kleinsten Distanz.*

*Markiere ihn.*

*Für alle seine unmarkierten Nachbarn:*

*Wenn (Distanz-Wert des aktuellen Knotens + Kantenlänge zum Nachbarn)*

*kleiner als die im Nachbarn gespeicherte Distanz ist,*

*speichere das als neue Distanz im Nachbarn.*

Die Herausforderung bei der Implementierung ist die effiziente Suche nach dem unmarkierten Knoten mit der kleinsten Distanz. Üblich ist es, alle unmarkierten Knoten in einem Heap (Priority Queue) mit der Distanz als Reihungskriterium zu organisieren.

## Kruskal:

*Beginne mit leerem Ergebnis (keine Kanten).*

*Wiederhole, bis das Ergebnis  $n-1$  Kanten enthält:*

*Wähle die (sortiert nach aufsteigendem Gewicht) nächste Kante des Graphen.*

*Wenn sie keinen Zyklus im Ergebnis verursacht: Nimm sie zum Ergebnis.*

Mit anderen Worten:

Das Zwischenergebnis besteht zu jedem Zeitpunkt aus mehreren Teilbäumen (am Anfang aus  $n$  Teilbäumen der Größe 1, vor dem letzten Schritt aus 2 Teilbäumen), und man nimmt nur solche Kanten dazu, die 2 verschiedene Teilbäume zu einem verbinden.

Die Herausforderung bei der Implementierung ist einerseits das effiziente Sortieren der Kanten nach Gewicht, andererseits der Test, ob die nächste Kante innerhalb eines Teilbaumes liegt oder zwei verschiedene Teilbäume verbindet. Dafür gibt es eine eigene Datenstruktur (*Union-Find-Struktur*).

Eine *Alternative* ist der Algorithmus von Prim, der ähnlich dem Dijkstra-Algorithmus arbeitet: Ausgehend von einem Startknoten wird in jedem Schritt die billigste Kante dazugenommen, die einen noch nicht erreichbaren mit einem schon erreichbaren Knoten verbindet.

Bei „dünnen“ Graphen ist der Kruskal-Algorithmus besser, bei „dichten“ Graphen der von Prim.

## 11 Compilerbau

Man steht gelegentlich vor dem Problem, **Eingaben mit komplexerer Struktur** zu verarbeiten, beispielsweise eine kleine *Skript- oder Makro-Sprache* oder schlicht *arithmetische Ausdrücke* (Rechnungen) unter Beachtung der Vorrangregeln, Klammern usw. oder *strukturierte Daten- oder Konfigurationsfiles* in XML, JSON oder YAML.

Für solche Aufgabenstellungen kommen Techniken des Compilerbaus zum Einsatz.

Der *erste Schritt* ist daher, die Struktur der Eingabe in **Syntaxregeln** (EBNF = Extended Backus Naur Form) oder in **Syntaxgraphen** zu fassen. Im Detail wird dabei festgelegt:

- Die terminalen Symbole (**Terminals, Tokens**): Schlüsselwörter, Identifier, Operatoren, Trennzeichen, Konstanten, ...
- Die nichtterminalen (zusammengesetzten) Symbole (**Nonterminals**): Ausdrücke, Befehle, Deklarationen, Programm, ...
- Die **Syntaxregeln** oder -Graphen: Wie ist jedes Nonterminal zusammengesetzt?
- Das **Start-Symbol**: Welches Nonterminal repräsentiert die ganze Eingabe?

Für die hier betrachteten *praktikablen Fälle* setzen wir voraus, dass die Syntax so beschaffen ist, dass sie **“leicht”** zu erkennen ist (siehe “Grundlagen der Informatik“): Mit *nur einem Terminal Vorausschau*, *eindeutig* und *ohne Backtracking* (Rückgängigmachen schon angewandter Regeln, wenn sie in eine Sackgasse geführt haben).

Typischerweise wird die Eingabe in *zwei Schritten* verarbeitet:

- Der **Lexer** (*lexikalischer Analysator*) verwandelt einen *Strom von Eingabezeichen* in einen *Strom von Token (Terminal-Symbolen)*, d.h. er fasst Ziffern zu Zahlen und Buchstaben zu Schlüsselwörtern oder Namen zusammen und erkennt Doppelzeichen-Operatoren, String- und Character-Konstanten usw.. Weiters verwirft der Lexer Zwischenräume und Kommentare.

Lexer können mit einem *Tool* (z.B. **“lex”** bzw. **“flex”**) erstellt werden, meistens sind sie aber problemlos *“ad hoc”* zu programmieren (großer **switch**-Befehl).

Typischerweise braucht man dazu *ein Zeichen Vorausschau* im Input.

C (**ungetc**) und C++ (**peek, putback**) stellen dafür I/O-Funktionen zur Verfügung, mit denen man ein Zeichen wieder in den Input *zurückstellen* (*“ungelesen machen”*) kann bzw. ein Zeichen vorausschauen kann, ohne es zu lesen.

- Der **Parser** (*Syntaxanalysator*) erkennt die *syntaktische Struktur*, d.h. findet heraus, welche *Syntaxregeln* wie auf die sequentielle Folge von Terminals *anzuwenden* sind, um Terminals *zu Nonterminals zusammenzufassen*, bis nur mehr das Start-Nonterminal übrig bleibt.

Er erkennt auch *Syntaxfehler*, wenn der Input nicht den Syntaxregel entspricht.

Je nach Anwendung ist der Output des Parsers entweder ein **Syntaxbaum** (d.h. eine Darstellung der syntaktischen Struktur des Inputs ausgehend vom Start-Nonterminal als Wurzel), oder es wird gleich **“on the Fly”** beim Erkennen eines jeden Nonterminals ein Ergebnis *berechnet* (bei Ausdrücken), eine Operation *ausgeführt* (bei Skript-Interpretern) oder Code *erzeugt* (bei Compilern).

Im Wesentlichen kommen heute 2 Arten von Parsern zum Einsatz:

- Parser nach der Methode des **rekursiven Abstiegs (Top-Down-Parser)**: Diese werden eher für einfachere Sprachen eingesetzt und normalerweise *handcodiert* (was bei gegebenen Syntaxregeln oder -Graphen "mechanisch" ohne Denkaufwand möglich ist, solange die Sprache "leicht" erkennbar ist, Details siehe unten).
- **Tabellengesteuerte Parser** (Bottom-Up-Parser, **Shift-Reduce-Parser**): Das ist die heute übliche Methode für komplexe Sprachen. Derartige Parser werden üblicherweise von Tools (z.B. "**yacc**" bzw. "**bison**") *automatisch generiert*, wobei der Code im Prinzip immer derselbe ist und nur die umfangreichen Tabellen zur Steuerung des Ablaufs von der jeweiligen Sprache bzw. Syntax abhängen.

Für einfache Fälle (z.B. arithmetische Ausdrücke, „Operator Grammar Parser“) ist es auch möglich, Code und Tabellen händisch zu erstellen, es erfordert aber *mehr Denkaufwand* als bei rekursiven Parsern (dafür arbeiten Shift-Reduce-Parser normalerweise deutlich *effizienter* als rekursive Parser).

Die Grundidee ist folgende:

- Man hat einen *Stack* von (terminalen und nonterminalen) Symbolen, der am Anfang leer ist.
- In jedem Schritt wird
  - entweder das nächste Token *vom Input gelesen und oben auf den Stack gelegt* ("Shift"),
  - oder die *oben auf dem Stack* befindliche komplette rechte Seite einer Syntaxregel *durch das Nonterminal auf der linken Seite der Syntaxregel ersetzt* ("Reduce") (bzw. der Code ausgeführt, der mit der Anwendung dieser Regel verbunden ist, oder die zu dieser Regel gehörenden Token am Stack in einen Syntaxbaum verwandelt oder was immer).
- Ob geschoben oder reduziert wird, entscheidet man in jedem Schritt auf Grund irgendwelcher Tabellenwerte, die vom nächsten Token im Input (1 Token Vorausschau) und den Symbolen oben am Stack abhängen.

Konkret z.B. bei Operator Grammar Parser:

- Es gibt eine Tabelle, in der für jede Art von Token ein „*Linksvorrang*“ und ein „*Rechtsvorrang*“ (Zahlenwerte) gespeichert ist.
- Der „*Rechtsvorrang*“ des *obersten noch nicht reduzierten Tokens* am Stack wird in jedem Schritt mit dem „*Linksvorrang*“ des *nächsten Tokens am Input* verglichen.
- Hat das *Input-Token* den *höheren* Vorrang-Wert, wird es auf den Stack *geschoben*, ist der Wert des unreduzierten Tokens höher, wird reduziert.
- Ist der Input leer und am Stack nur mehr das Startsymbol, ist man fertig.

**Parser mit rekursivem Abstieg** wollen wir uns etwas näher ansehen:

- Voraussetzung ist, dass man sich bei jeder Alternative in einer Syntaxregel bzw. an jeder Verzweigung eines Syntax-Graphen mit **nur einem Token Vorausschau** entscheiden kann. Daher müssen sich
  - bei jeder Alternative alle Möglichkeiten durch ihr Anfangs-Token unterscheiden,
  - und bei allen Nonterminals, die leer, optional bzw. wiederholt sein können, müssen sich alle Anfangs-Token des jeweiligen Nonterminals von allen Folge-Token nach diesem Nonterminal unterscheiden.
- **Pro Nonterminal wird eine Funktion implementiert.**  
Diese hat normalerweise keine Argumente und als Returnwert das Ergebnis der Auswertung des erkannten Input-Konstruktes bzw. einen Pointer auf die Wurzel des Syntaxbaumes dieses Nonterminals.
- Diese Funktion wird aufgerufen, sobald sicher ist, dass am Input ein Vorkommen dieses Nonterminal kommt (d.h. mit dem ersten Token des Nonterminals als nächstem Input), und kehrt zurück, wenn sie den gesamten Input, der zu diesem Vorkommen des Nonterminals gehört, gelesen und verarbeitet hat (d.h. mit dem ersten Token nach dem Nonterminal als nächstem Input).
- Am Anfang wird die Funktion des Startsymbols aufgerufen. Kehrt sie zurück, ist man fertig.
- Intern ruft jede Nonterminal-Funktion
  - in **if's** (für Alternativen und optionale Elemente)
  - und Schleifen (für wiederholte Elemente),  
deren Struktur der Syntaxregel des Nonterminals entspricht, rekursiv die Funktionen jener Terminals und Nonterminals auf,  
aus denen sich das Nonterminal laut Syntax zusammensetzt.

In den Bedingungen der **if's** und Schleifen wird dabei das nächste Token am Input (1 Token Vorausschau!) geprüft, um festzustellen, welcher Alternative in der Syntaxregel bzw. welchem Pfad im Syntaxgraphen der Input entspricht und welche Funktionen daher aufgerufen werden müssen.

## 12 Speicherverwaltung

1.) Aus **Betriebssystem-Sicht** ist die Speicherverwaltung prinzipiell relativ einfach:

- Speicher wird auf der Ebene von einzelnen Seiten (meist 4 KB) der virtuellen Speicherverwaltung verwaltet, d.h. alle Speicherblöcke sind gleich groß. Damit gibt es nur eine einzige Frei-Liste und keinen Verschnitt.<sup>1</sup>
- Alle Speicheranforderungen von Programmen werden auf Seitengrößen aufgerundet und an Seitenanfängen ausgerichtet.
- Speicheranforderungen für mehr als eine Seite können bedient werden, indem einzelne real verstreut liegende freie Seiten mittels virtueller Speicherverwaltung in einen logisch fortlaufenden Adressbereich abgebildet werden.

2.) Auch an der **Schnittstelle zwischen dem Betriebssystem und einer Anwendung** ist die Speicherverwaltung noch einfach: Für jedes Programm gibt es zwei Grenzen zwischen gültigem und ungültigem Adressbereich:

- Eine zwischen oberem Ende der dynamischen Daten (Heap) und unterem Ende des ungültigen Bereiches: Speicheranforderungen des Programms verschieben diese Grenze mittels Betriebssystem-Aufrufen (Linux: **brk** / **sbrk**) explizit nach oben (um ganze Seiten), d.h. der Platz für den Heap wächst.
- Eine zwischen unterem Ende des Stacks und oberem Ende des ungültigen Bereiches: Sie wird von der virtuellen Speicherverwaltung implizit verschoben, wenn der Stack wächst.

Unabhängig davon werden **mmap**-Bereiche in den Adressraum eingeblendet.

3.) **Innerhalb einer Anwendung** gibt es zwei grundsätzliche Möglichkeiten der Freispeicher-Verwaltung:

- **Explizite Speicherverwaltung** (C / C++: **malloc** / **free** bzw. **new** / **delete**)
- **Garbage Collection** (Java, C#, ...)

### 12.1 Explizite Speicherverwaltung

Für die Implementierung von **malloc** und **free** (bzw. **new** und **delete**) ist die C-Library zuständig. Nur sehr große **malloc**-Anforderungen (einige MB) werden direkt an das Betriebssystem durchgereicht (als **sbrk** oder **mmap**), diese Blöcke werden beim **free** auch wieder direkt an das System zurückgegeben.

Kleinere **malloc**-Anforderungen werden von der C-Library aus einem von ihr verwalteten Speicherbereich (dem "**Heap**") bedient und resultieren daher nicht in Systemaufrufen, sondern werden ohne Zutun des Betriebssystems innerhalb der Anwendung abgehandelt. Nur wenn im Heap kein ausreichend großer freier Block mehr vorhanden ist, wird der Platz für den Heap durch einen Systemaufruf vergrößert, und zwar nicht um den angeforderten Block, sondern gleich auf Vorrat um ein paar MB, um für die nächsten **malloc** wieder eine Reserve zu haben.

<sup>1</sup> Die Wahrheit ist bei heutigen Systemen meist deutlich komplexer: Erstens können moderne Prozessoren meist 2 verschiedene Seitengrößen gleichzeitig verwenden, und zweitens werden im Betriebssystem intern sehr wohl Freilisten für verschiedene Größen real benachbarter freier Seiten benötigt.

Einmal vom System für den Heap angeforderter Speicher wird üblicherweise bis zum Programmende *nicht* mehr an das System zurückgegeben, auch wenn die darin angelegten **malloc**-Blöcke wieder freigegeben wurden.

Die C-Library verwaltet den Heap meistens als (einfach verkettete) **Liste freier Bereiche** (manchmal sind die freien Bereiche auch als Baum oder dergleichen organisiert):

- Beim **malloc** wird ein ausreichend großer freier Block in der Liste gesucht (“**first fit**” oder “**best fit**”).

Passt er genau oder ist knapp zu groß, wird er als Ganzes aus der Freiliste entfernt und dem Benutzer zurückgegeben. Ist er deutlich zu groß, wird er *geteilt*: Hinten wird ein Teil abgetrennt und dem Benutzer gegeben, der vordere Teil bleibt verkleinert in der Freiliste.

Damit sich vorne in der Liste nicht lauter ganz kleine freie Blöcke sammeln, die bei jedem **malloc** eines größeren Blocks zuerst überlesen werden müssen, ist die Liste normalerweise *zyklisch*, und jedes neue **malloc** sucht ab der Stelle, an der der vorige Block entnommen wurde.

Findet sich in der ganzen Liste kein ausreichend großer freier Block, wird Speicher vom System nachgefordert und in die Freiliste eingehängt.

- Bei **free** wird zuerst geprüft, ob die Speicherbereiche unmittelbar davor oder unmittelbar dahinter auch frei sind. Wenn ja, wird der freigegebene Block mit diesen *verschmolzen* (das wirkt der Fragmentierung des Speichers entgegen), wenn nein, wird der freigegebene Block in die Freiliste eingehängt.

Eine Alternative zum Freilisten-Verfahren ist das **Buddy-Verfahren**: Hier ist die Größe aller freien und belegten Blöcke eine *Zweierpotenz*, alle **malloc**-Anforderungen werden auf die nächste Zweierpotenz aufgerundet. Weiters gibt es *eine Freiliste pro Zweierpotenz*.

Jeder Block hat einen (eventuell weiter unterteilten) “Bruder” gleicher Größe unmittelbar davor oder danach, beide sind durch Halbieren eines doppelt so großen Blockes entstanden.

- Beim **malloc** wird der *erste* Block aus der der Größe entsprechenden Freiliste entnommen (es *entfällt* daher der Aufwand, die Freiliste zu durchsuchen!). Ist diese Freiliste leer, wird ein größerer freier Block so lange *halbiert*, bis er passt (die jeweils anderen Hälften kommen in die entsprechenden Freilisten).
- Beim **free** wird geprüft, ob der *Bruder* auch frei ist. Wenn nein, wird der Block gleich in die entsprechende Freiliste gehängt. Wenn ja, wird er mit seinem Bruder verschmolzen. Für diesen neuen, doppelt so großen freien Block wird wieder der Bruder geprüft usw.. Erst wenn der Bruder belegt ist, kommt der Block in die jeweilige Freiliste.

Die **Qualität** einer Speicherverwaltung wird primär an folgenden Faktoren gemessen:

- *Geschwindigkeit* (im Mittel und im schlechtesten Fall)
- *Alterungsbeständigkeit* (auch über lange Zeit möglichst *geringe Fragmentierung*)
- Bei modernen Systemen: Verhalten bei *Multithreading* / auf Multicore-Systemen.

Besondere Herausforderungen für die Speicherverwaltung gelten in zwei Anwendungsfällen:

- *Echtzeitsysteme*: Hier ist nicht die durchschnittliche Performance von **malloc** und **free** das zentrale Thema, sondern der *Worst Case*, da es in einem Echtzeitsystem möglich sein muss, für jede Operation eine *fixe obere zeitliche Schranke* anzugeben. Die Laufzeit eines **malloc** (und / oder **free**) ist in üblichen Speicherverwaltungen aber nicht limitiert, sondern stark Daten- bzw. Verlaufs-abhängig.
- *Multithreaded-Systeme*: Da der Heap von allen Threads *gemeinsam* benutzt wird, muss die Speicherverwaltung *parallelitätsfest* sein. Weiters sollte es nicht passieren, dass ein aufwändiges **malloc** oder **free** in einem Thread alle Speicherverwaltungs-Operationen in den anderen Threads für längere Zeit blockiert: *Gleichzeitige Allokationen und Freigaben* sollten möglich sein.

## 12.2 Garbage Collection

Hier geht es darum, *automatisch zu erkennen, wenn ein dynamisch angeforderter Speicherblock nicht mehr benutzt wird* und daher freigegeben bzw. wiederverwendet werden kann.

Das ist genau dann der Fall, wenn *kein Pointer mehr auf ihn zeigt*: Dann ist er vom Programm aus nicht mehr ansprechbar.

Dafür gibt es zwei grundlegende Verfahren:

- **Reference Counting**
- **Mark & Sweep**

Die folgende Darstellung ist stark vereinfacht, die tatsächlichen Implementierungen sind *sehr komplex*.

Beide Methoden können *nur mit Pointern auf den Block-Anfang* umgehen: Für Pointer mitten in einen Block (wie sie zum Beispiel in C oder C++ durch Pointer-Arithmetik entstehen) sind sie nicht oder nur mit großem zusätzlichem Aufwand geeignet. Garbage Collection kommt daher primär in „sauberen“ Programmiersprachen ohne Pointer-Arithmetik etc. zum Einsatz (Java, C#, ...).

### **Reference Counting**

Hier enthält jeder Speicherblock einen *Zähler, der angibt, wie viele Pointer auf ihn zeigen*. Jedesmal, wenn ein *neuer Pointer* erzeugt wird (Zuweisung, Parameter-Übergabe, Kopie einer Struktur mit Pointern, ...), wird der *Zähler erhöht*, und jedesmal, wenn ein *Pointer überschrieben oder freigegeben* wird (Zuweisung, Freigabe einer lokalen Pointer-Variablen oder eines Parameters, Freigabe einer Struktur mit Pointern), wird er *erniedrigt*.

Sinkt der *Zählerstand auf 0*, sind 3 Dinge zu tun:

- Destruktor aufrufen.
- Counter aller im Block enthaltenen Pointer dekrementieren.
- Block freigeben.

Die freien Blöcke werden *wie bei einer expliziten Speicherverwaltung* verwaltet (*Freiliste*).

### Vorteile von Reference Counting:

- Die Speicherverwaltung wird im normalen Programm-Ablauf miterledigt, es entstehen keine längeren Pausen in der Programm-Ausführung.
- Freiwerdende Daten werden sofort freigegeben, der Destruktor sofort ausgeführt.

### Nachteile von Reference Counting:

- Viel zusätzlicher Code & viel zusätzliche Zeit bei jeder einzelnen Pointer-Operation!
- Zyklische Datenstrukturen werden nie mehr freigegeben, da deren Reference Count nie auf 0 sinkt.
- Zusätzlicher Platzbedarf für den Zähler in jedem Block.
- Der Speicher fragmentiert, die Rekombination benachbarter freier Blöcke kostet zusätzlichen Aufwand.

### **Mark and Sweep**

Hier läuft das Programm im Normalbetrieb im Wesentlichen unverändert, die Garbage Collection erfordert keine besonderen Schritte: Es wird nur allokiert, nicht freigegeben.

Ist der freie Speicher zu Ende, wird das Programm angehalten und der Speicher in 2 Phasen aufgeräumt:

- Die erste Phase "**mark**" sucht und markiert alle belegten Speicherblöcke, indem sie ausgehend von allen Pointern außerhalb des Heaps (globale Variablen, lokale Variablen und Parameter, interne temporäre Variablen, ...) alle von einem Pointer getroffenen Blöcke markiert und rekursiv alle in einem markierten Block befindlichen Pointer weiterverfolgt.
- Die zweite Phase "**sweep**" ("Aufwischen") geht einmal linear über den gesamten Heap und hängt alle nicht markierten Blöcke in die Freiliste. Blöcke, die zu Objekten mit Destruktoren gehören, werden separat gesammelt: Für diese muss nach der Garbage Collection zuerst einmal der Destruktor ausgeführt werden, erst dann werden sie wirklich freigegeben.
- Eine alternative Sweep-Methode ("**Compacting Garbage Collector**") besteht darin, alle markierten Blöcke "dicht auf dicht" an ein Ende des Speichers umzukopieren und in einem weiteren Durchgang alle Pointer im System (sowohl die in den Blöcken als auch die außerhalb des Heaps), die darauf gezeigt haben, entsprechend zu aktualisieren (auf die neue Adresse des Blockes zu ändern).

Auf diese Weise entsteht ein **großer zusammenhängender freier Speicherbereich**: Es gibt kein Fragmentierungsproblem, und das Allokieren wird viel einfacher (man braucht keine Freiliste, sondern nimmt einfach die nächsten freien Bytes), auf Kosten eines viel höheren Aufwandes bei der Garbage Collection.

### Vorteile von Mark and Sweep:

- Kein zusätzlicher Aufwand im normalen Programmablauf, weder Platz noch Zeit noch Code.
- Alle freien Objekte werden erkannt, auch zyklische.
- Die Sweep-Phase rekombiniert automatisch benachbarte freie Blöcke oder erzeugt überhaupt einen großen freien Bereich.

## Nachteile von Mark and Sweep:

- Vollständige Unterbrechung des Programmablaufes, ev. für mehrere Sekunden.
- Der Collector muß alle Pointer in den Daten eines Programmes (globale Daten, Stack, Heap, Register und temporäre Daten) finden können.

Dazu muss er in allen diesen Bereichen Pointer von nicht-Pointern (**char**, **int** oder **double**) unterscheiden können. Das erfordert entweder getrennte Bereiche für Pointer und nicht-Pointer oder zusätzliche Markierungen jedes einzelnen Daten-Wertes im Speicher.

Eine Alternative sind "konservative Garbage-Collektoren": Sie betrachten jeden Wert am Stack, in globalen Variablen, in Objekten usw. als potentiellen Pointer und prüfen, ob der Wert auf den Anfang eines dynamischen Speicherblockes zeigt, wenn man ihn als Pointer interpretiert.

Auf diese Weise werden sicher alle belegten Blöcke erkannt, aber eventuell zu viele Blöcke (nämlich die, deren Adresse zufällig dem Wert z.B. einer **int**- oder **double**-Variablen entspricht) als belegt markiert.

- Pulsierender Speicherbedarf, meist höher als bei Reference Counting.  
Im Besonderen muss für die Garbage Collection der gesamte Speicherbereich des Programmes auf einmal in den realen Hauptspeicher geladen (d.h. falls ausgelagert aus dem Swap-space geholt) werden.
- Ev. stark verzögertes Freigeben (verzögerter Destruktor-Aufruf) von Objekten.

## **Generationsbasierte Garbage Collection**

Wie im vorigen Absatz klar wird, ist eine Mark and Sweep Garbage Collection über den gesamten Speicher ein hoher Aufwand.

Zur Reduzierung dieses Aufwandes hilft in vielen Fällen folgende Beobachtung:

- In typischen Programmen lassen sich die dynamischen Daten größtenteils in zwei Klassen einteilen:
  - Temporäre oder kurzlebige Daten  
(z.B. Hilfsvariablen, „by Value“-Parameter einer Funktion, ...)
  - Langlebige Daten  
(globale, während der gesamten Programmlaufzeit benötigte Daten)

„Mittellebige“ Daten sind normalerweise eher die Ausnahme.

- Anders formuliert: Bei sehr „jungen“ Daten ist die Wahrscheinlichkeit, dass sie bei einer Garbage Collection freigegeben werden, ziemlich hoch, und bei sehr alten Daten ist sie sehr gering.

Der Aufwand der Garbage Collection rechnet sich daher eher, wenn man die Garbage Collection vor allem über junge Daten laufen lässt, weil sie dort in Vergleich zur Datenmenge viel Platz freigeben wird, während eine Garbage Collection über sehr alte Daten meist vergebliche Mühe ist, weil dort die meisten Daten weiterleben werden.

- Als „Altersmaßstab“ eignet sich die Anzahl, wie viele Garbage Collections ein bestimmter Datenblock schon überlebt hat, ohne freigegeben zu werden.

Die Realisierung dieser Beobachtungen führt zu folgendem Konzept:

- Man teilt die Daten in „Generationen“ ein:  
Die Generation, zu der ein bestimmter Datenblock gehört, hängt von der Anzahl der Garbage Collections ab, die er schon überlebt hat: Neu angelegte Daten landen in Generation 0, und nach jeder (oder jeder n-ten) Garbage Collection wandern die Daten einer Generation in die nächste Generation.
- Man organisiert die Generationen im Speicher so, dass der Garbage Collector jede Generation einzeln und unabhängig von den anderen bereinigen kann.
- Man lässt den Garbage Collector jedesmal über Generation 0 laufen, etwas seltener (bzw. nur, wenn Generation 0 nicht genug freien Platz liefert) über Generation 1, noch seltener über Generation 2 usw..

Mit der Zeit sammelt sich dadurch die große Menge der „ewigen“ Stammdaten eines Programmes in einer „Generation uralt“, über die der Garbage Collector äußerst selten läuft, weil es bei diesen Daten ohnehin fast nichts bringt.