# Grundlagen der Informatik (II)

## Abstrakte Datentypen & Algorithmen

**Dr. Stefan Bosse**

**Universität Bremen**

**2.7.2007**

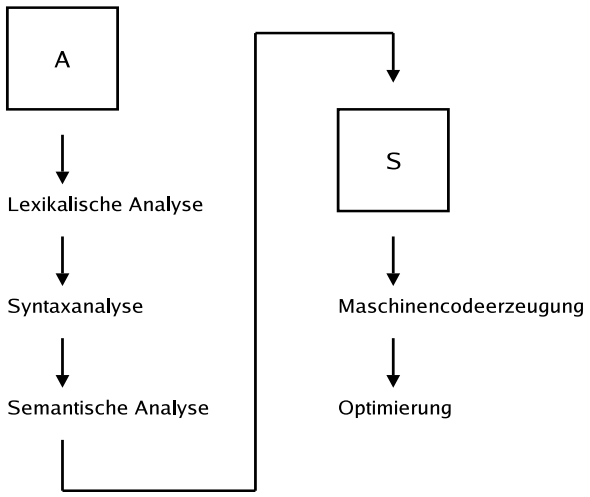
# Compiler: Analyse und Synthese von Programmen (I)

➤ Ein Übersetzer (Compiler) erzeugt aus einem Quellprogramm ein Maschinenprogramm. Teilaufgaben:

**Analyse A**
Prüfung der Syntax und (statischen) Semantik des Quellprogramms

**Synthese S**
Erzeugung des Maschinenprogramms

**Die Analyse- und Syntheseteile sind in einzelne Phasen unterteilt:**






# Compiler: Analyse und Synthese von Programmen (II)

➤ Ein Quellprogramm ist ein zusammenhängender Text. Zunächst ohne Strukturierung aus der Sicht des Compilers.

## Lexikalische Analyse

- Zerlegung des Quellprogramms in eine Folge von Symbolen: Namen, Zahlen, Schlüsselwörter der Programmiersprache, Sonderzeichen usw. Wird von einem sog. Lexer durchgeführt.
- ➥ Lexikalische Symbole sind Zeichenfolgen des Quellprogramms. Diese können aus syntaktischer Sicht nicht weiter zerlegt werden. Ein Symbol ist gekennzeichnet durch:
```
{
Symbolcode (Name→Code),
Symbolwert,
Symboltextposition(für Fehlermeldungen)
}
```
- Überlesen von bedeutungslosen Zeichen und Kommentaren,
- Behandlung von Steueranweisungen (Compileroptionen und Pragmas)
- Prüfung auf lexikalische Fehler

# Compiler: Analyse und Synthese von Programmen (III)

## Lexikalische Analyse

- Man unterscheidet:

  **Einfache Symbole**
  kommen nur in einer einzigen Bedeutung vor, z.B. Schlüsselwörter {if, while, else, ...}

  **Symbolklassen**
  beschreiben Mengen gleichartiger Symbole, z.B. Funktionsnamen, die sich in ihrer Bedeutung unterscheiden.

- Beispiel einer lexikalischen Analyse eines C-Programmfragments:

```
while(x<10)
{
  x=x+1;
  y=x*x;
};
```

⇓ **Lexikalische Analyse in eine Symbolfolge**

| Code | Wert |
|---|---|
| while (1) | |
| name (21) | ”x” |
| less (11) | |
| integer (22) | 10 |
| block_start (31) | |
| name (21) | ”x” |
| assign (24) | |
| name (21) | ”x” |
| add (41) | |
| integer (22) | 1 |
| ••• | |



# Compiler: Analyse und Synthese von Programmen (IV)

## Syntaktische Analyse

- Prüfung der syntaktische Korrektheit des Quellprogramms, z.B. 2xa wäre kein gültiger Symbolname.

- Zerlegung und Neugruppierung der vom lexikalischen Analysator gelieferten Symbole in syntaktische Einheiten → Strukturierung:
  - Definitionen und Deklarationen von Daten- und Funktionsobjekten,
  - Programmanweisungen und Programmkonstrukte (if, while, ...),
  - Ausdrücke (arithmetische, relationale, boolesche)

- Funktion des Parsers unter Verwendung von Suchmustern:
  ➥ Zerlegung durch Verwendung von Suchmustern und Suchmasken, z.B.

| Eingabe: C-Code | | | | |
|---|---|---|---|---|
| if | (x<0) | { | x=x+1; | }; |
| ↓ | | ↓ | | ↓ |
| **Ausgabe: Suchmuster des Parsers** | | | | |
| IF | <expr> | BE-GIN | <expr> | END; |

# Compiler: Analyse und Synthese von Programmen (V)

## Semantische Analyse

- Nach der syntaktischen Prüfung des Quellprogramms werden semantische Regeln (sog. Kontextbindungen) geprüft, z.B. wenn eine Variable in einem Ausdruck erscheint, ob diese definiert wurde, Typüberprüfung von Funktionsargumenten usw.
- Es werden compilerinterne Datenstrukturen aufgebaut:
  1. Symbolliste bzw. Symboltabelle
  2. Zwischenkode (Intermediate Representation IR)
- Diese Datenstrukturen und die Symboltabelle werden für die Codeerzeugung benötigt.

## Symboltabelle

Die Symboltabelle enthält Informationen über alle im Quellprogramm deklarierten Namen (Konstanten, Variablen, Typen, Funktionen). Der Übersetzer benötigt diese Informationen

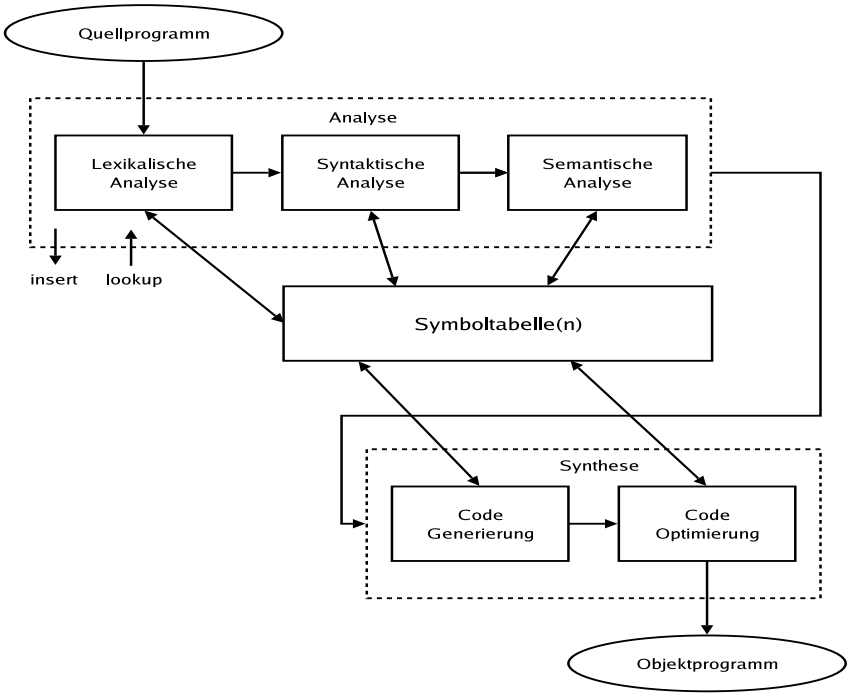
1. zur Typprüfung,
2. zur Erzeugung des Zwischen- und Maschinencodes.



# Compiler: Analyse und Synthese von Programmen (VI)

## Symboltabelle

Die Symboltabelle wird i.A. von allen Phasen der Analyse und Synthese verwendet.



# Compiler: Analyse und Synthese von Programmen (VII)

## Symboltabelle

Ein Eintrag in der Symboltabelle kann folgende Informationen enthalten:

| Variable Name | Adresse | Typ | Dimension | Quelle Zeile | Quelle Referenzen |
|---|---|---|---|---|---|
| x | 1000 | 1 | 0 | Deklaration<br>1 | Verwendung = Referenzierung<br>2, 3 |

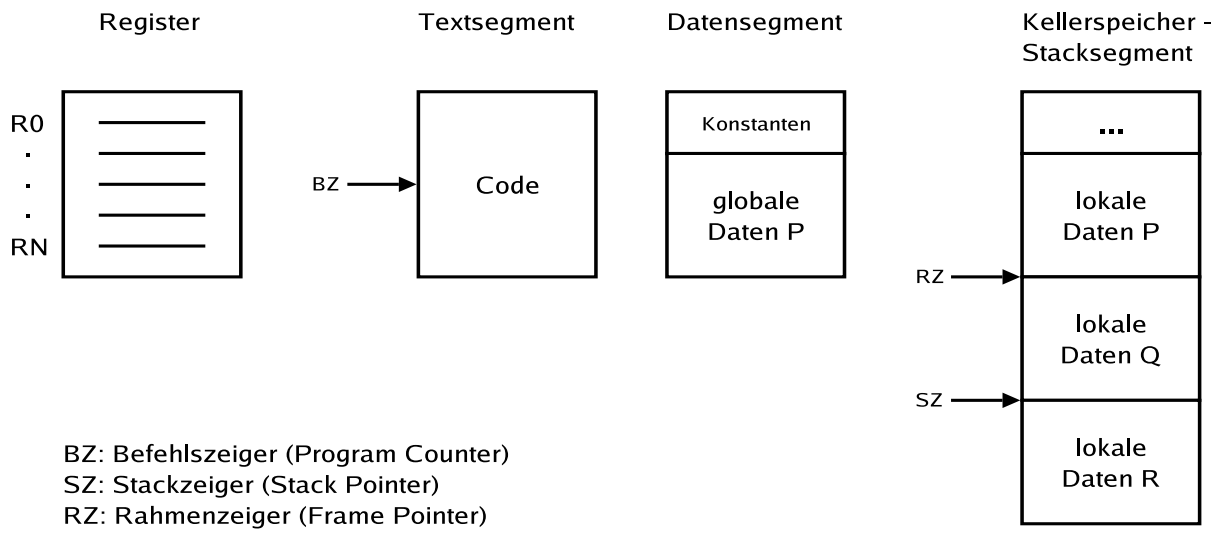
Beispiel:
1: int x; int a;
2: x=4;
3: a=x*2;



# Compiler: Analyse und Synthese von Programmen (VIII)

## Codeerzeugung und Optimierung: die Synthese

Codeerzeugung ist abhängig von Eigenschaften der Zielmaschine, d.h. des Mikroprozessorsystems, auf dem der Objektcode ausgeführt werden soll.
Typisches Laufzeitmodell einer Zielmaschine:




BZ: Befehlszeiger (Program Counter)
SZ: Stackzeiger (Stack Pointer)
RZ: Rahmenzeiger (Frame Pointer)

# Compiler: Analyse und Synthese von Programmen (IX)

## Codeerzeugung und Optimierung: die Synthese

Bei der Codeerzeugung muß auf

1. minimierten Speicherbedarf und
2. schnelle und effiziente Programmausführung (Laufzeitminimierung)

geachtet werden, zwei i.A. gegenläufige Ziele.

- Alle globalen Datenobjekte (unbegrenzte Lebensdauer) muß eine Speicheradresse und eine Speichergrösse im Datensegment zugewiesen werden.
- Lokale Daten werden dynamisch und temporär im Stackspeicher abgelegt → begrenzte Lebensdauer von Datenobjekten.
- Bei einem Funktionsaufruf muß ein. sog. Aufrufrahmen (Frame) im Stackspeicher angelegt werden. Dieser enthält Informationen, die zur Programmfortführung nach Beendigung des Funktionsaufrufs benötigt werden.
- Temporäre Ergebnisse von Ausdrücken sollen aus Gründen der Effizienz und Geschwindigkeit in lokalen Prozessorregistern abgelegt werden.
- Bei der Übersetzung von bedingten Verzweigungen und Schleifen müssen Sprünge im Maschinencode erzeugt werden.



# Compiler: Analyse und Synthese von Programmen (X)

## Codeerzeugung und Optimierung: die Synthese

### Beispiel Codesynthese

```
Quellprogramm      →      Maschinenprogramm
IF-Anweisung
IF E THEN                  E
  S1                       falsejump L1
ELSE                       S1
  S2                       jump L2
END IF                  L1:S2
                        L2:•••
WHILE-Schleife
WHILE E                L1:E
DO                        falsejump L2
  S                       S
DONE                      jump L1
                       L2:•••
```

# Compiler: Analyse und Synthese von Programmen (XI)

## Optimierung

Optimierung kann entweder während oder am Ende des Analyseteils, d.h. auf Programmebene, oder nach der Codeerzeugung, d.h. auf Maschinenebene, erfolgen.
Optimierung Ø dient der Verbesserung des Laufzeit- und Speicherplatzverhaltens eines Programms.

### I. Einfache Optimierungen

**Konstantenfaltung**: Konstantenausdrücke werden bereits zur Übersetzungszeit ausgewertet.
Beispiel:
Ø: `x=100*2` → `x=200`

**Algebraische Vereinfachung**: eine teure Operation (Laufzeit) wird durch eine billigere (geringere Laufzeit) ersetzt.
Beispiel:
Ø: `x=y*2` → `x = y <<1` mit `<<: logical shift left`



# Compiler: Analyse und Synthese von Programmen (XII)

## Optimierung

### I. Einfache Optimierungen

**Unterdrückung von Laufzeitprüfungen**, z.B. bei Arrays (wird von C nicht unterstützt).
Beispiel:
Pascal:

```
var a: array[1..10] of integer;
for i := 1 to 10
do
a[i] := •••
done;
```

Ø: Der Array-Index ist in diesem Beispiel aufgrund der statischen For-Schleife bekannt, und liegt im zulässigen Wertebereich $i \in [1..10]$. Eine Indexprüfung (wie in Pascal-Programmen sonst üblich) bei dem Array-Zugriff a[i] kann daher entfallen. Eine Indexüberprüfung zur Laufzeit bedeutet ansonsten zusätzliche Maschinenbefehle!

# Compiler: Analyse und Synthese von Programmen (XIII)

## Optimierung

### II. Fortpflanzung von Zuweisungen

→
Substitution von Ausdrücken.
Beispiel:
```
∅: x=y; a=x+z;
→
x=y; a=y+z;
```

### III. Entfernung von toten Code

→
Eine Zuweisung an eine Variable, die im folgenden nicht mehr verwendet (gelesen) wird, also auf der rechten Seite eines Ausdrucks erscheint, kann entfernt werden.
Beispiel (von II.):
```
∅: x=y; →⊗
```



# Compiler: Analyse und Synthese von Programmen (XIV)

## Optimierung

### IV. Schleifeninvarianter Code

→
Wenn eine Schleife eine Berechnung (einen Ausdruck) enthält, die unabhängig vom Schleifendurchlauf ist, d.h. immer den gleichen Wert als Ergebnis ergibt, kann diese vor der Ausführung der Schleife berechnet werden → einmalige Auswertung des Ausdrucks vor der Schleife.
Beispiel:
```
int y,x,i;
y=5;
∅: for(i=0;i<10;i++){x=y+2;•••};
→
x=y+2;
for(i=0;i<10;i++){•••};
```

# Compiler: Analyse und Synthese von Programmen (XV)

## Optimierung

### V. Befehlsanordnung

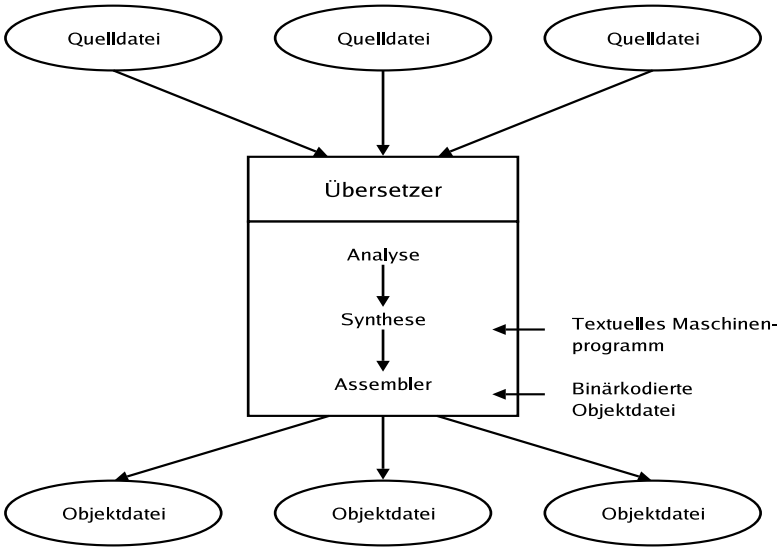
→

Das Laufzeitverhalten moderner Rechner (=Geschwindigkeit) kann von der Reihenfolge der Maschinenbefehele im Maschinenprogramm abhängen. Stichwort: Fließbandbetrieb und teilparallele Ausführung mehrerer Maschinenbefehle im Prozessor (Instruction-Pipeline).



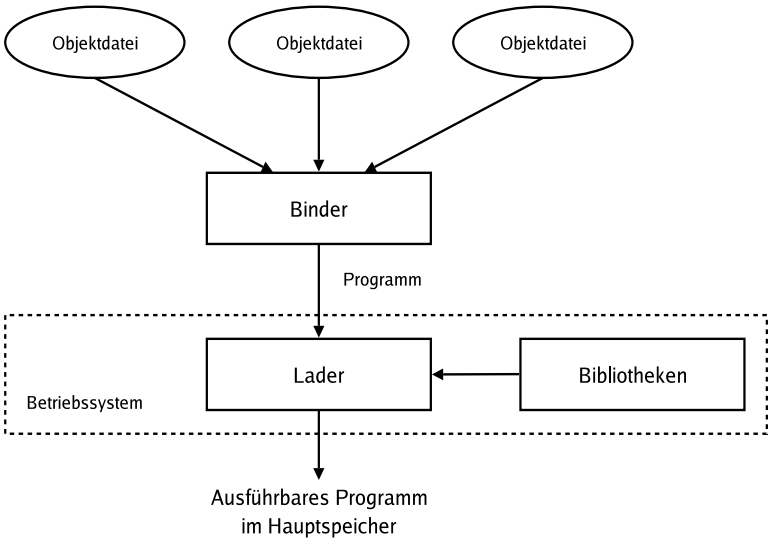
# Binder, Lader und Bibliotheken (I)

- Programme bestehen i.A. aus mehreren Einheiten, sog. Modulen.
- Der Übersetzer erzeugt i.A. aus einer Quelldatei eine Objektdatei. Diese Objektdatei bildet noch kein vollständiges Programm, welches als Prozeß ausgeführt werden kann.
- Ein- und Ausgabedateien beim Übersetzungsvorgang:



# Binder, Lader und Bibliotheken (II)

- Ein sog. Binder (linker) fügt die einzelnen Objektdateien zu einem ausführbaren Programm zusammen.
- Statischer Bindevorgang von einzelnen Objektdateien zu einem Programm und dynamische Bindung bei der Ausführung durch den Lader:



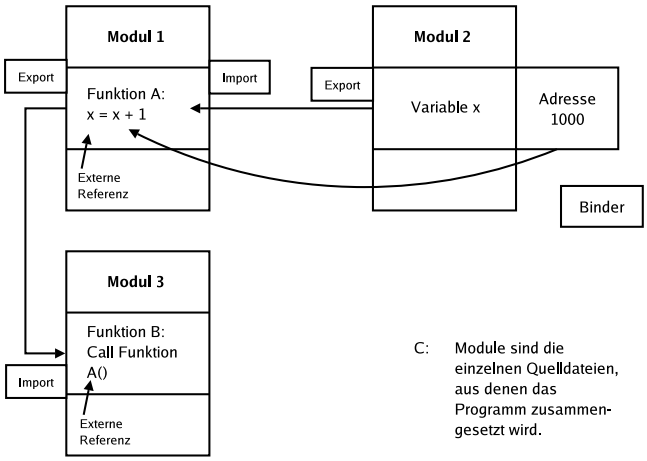



# Binder, Lader und Bibliotheken (III)

- Ein sog. Lader (Loader) bringt eine Programmdatei in ein ausführbares Programm, d.h. ein Prozeß wird erzeugt. Hier beginnt die Betriebssystemebene. Der Lader kann zur Laufzeit weiteren Programmcode aus Bibliotheken zum Programm hinzufügen.
- Aufgaben des Laders:
  - Anlegen von Speicherbereichen für Programmcode und Daten; füllen dieser Speicherbereiche mit dem Inhalt aus der Programmdatei,
  - Auflösen externen Programmreferenzen (externe Funktionen und Variablen in Bibliotheken),
  - Initialisierung.
- Inhalt einer Objektdatei:
  - Vorspann: Name, Version des Moduls, Strukturierung der nachfolgenden Informationen,
  - Maschinencode, z.B. Instruktion MOVE AX,7
  - Globale Daten mit Initialisierung, z.B. C: `int a=14;`
  - Lokale Daten mit Initialisierung, z.B. C: `static int z=10;`

# Binder, Lader und Bibliotheken (IV)

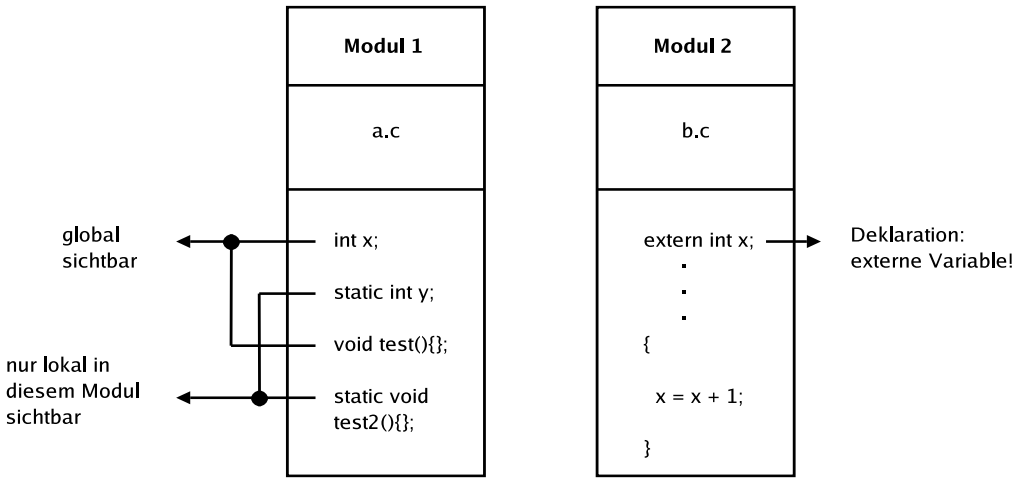
- Inhalt einer Objektdatei (Forts.):
  - Binär-Informationen:
    1. Exportliste: Bekanntgabe von exportierten Funktionen und Variablen (globale Daten)
    2. Importliste: nicht in diesem Modul definierte aber referenzierte Funktionen und Variablen
    3. Referenzliste: Liste aller Stellen im Maschinenkode, an denen externe Objekte verwednete werden.
- Im- und Exportlisten einzelner Module:






# Binder, Lader und Bibliotheken (V)

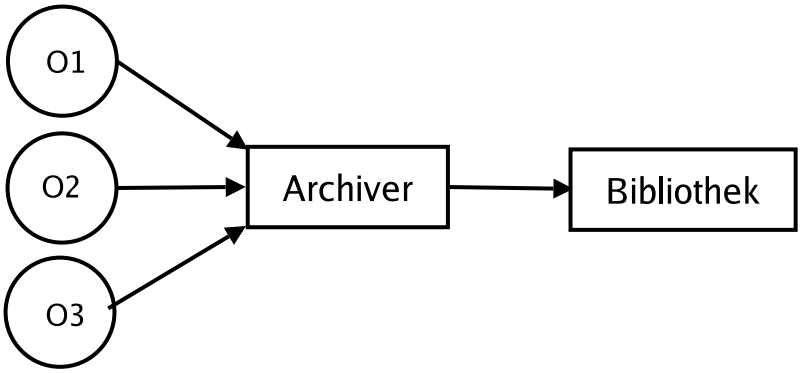
- Bei der Erzeugung der einzelnen Objektdateien müssen für externe Referenzen Platzhalter (dummies) in dem Maschinencode eingefügt werden, bis der Binder (bzw. der Lader) alle Module zusammenfügt.
- C: Globale Variablen und Funktionen sind alle diejenigen, die nicht mit dem Schlüsselwort `static` vorangestellt definiert werden!
- C: Bei der Verwendung von externen Variablen und Funktionen müssen diese mit dem Schlüsselwort `extern` vorangestellt deklariert werden!
- Globale und lokale Variablen in C:



# Binder, Lader und Bibliotheken (VI)

- Eine beliebige Zahl von Objektdateien können zu einem Archiv oder einer Bibliothek für späteres Binden zusammengefasst werden.
- Man unterscheidet:
  1. Statische Bibliotheken → Binder
  2. Dynamische Bibliotheken → Binder ⊕ Lader
- Im letzten Fall wird der endgültigen Programmdatei nur Informationen über die Bibliothek hinzugefügt, aber keine Code-Inhalte.
- Der Lader muß bei jedem Programmstart die fehlenden Bibliothekeninhalte (Funktionen und Variablen) dem Programm hinzufügen.

Archivierung von Objektdateien in einer Bibliothek.



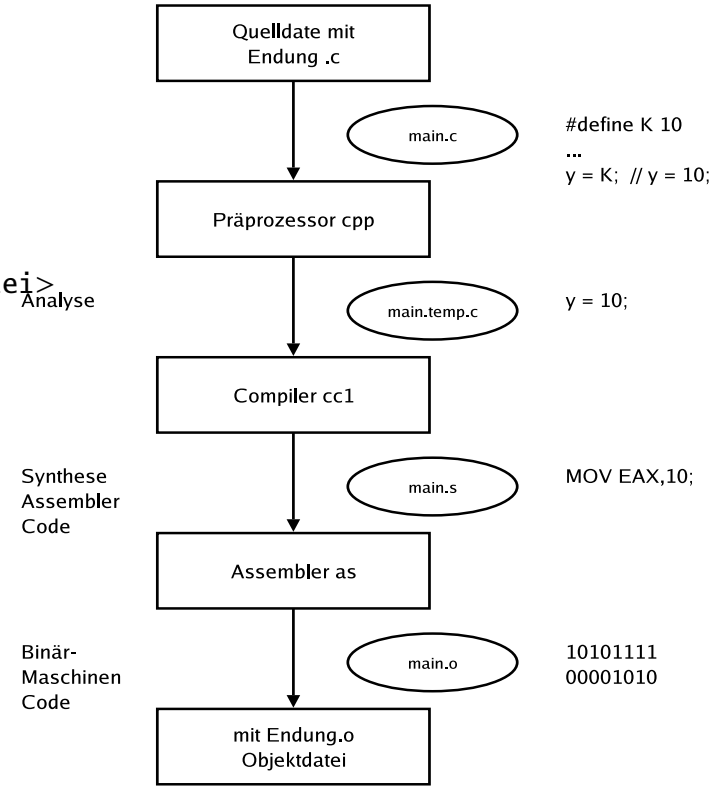



# Der GCC-Compiler (I)

- GCC steht für: **G**NU (Open Source Lizenz) **CC**ompiler
- Der GCC-Compiler arbeitet kommandozeilenorientiert, d.h. alle Dateinamen und Parameter müssen beim Aufruf dem Programmnamen angehängt werden.
  GCC-Aufruf über die Kommandozeile:

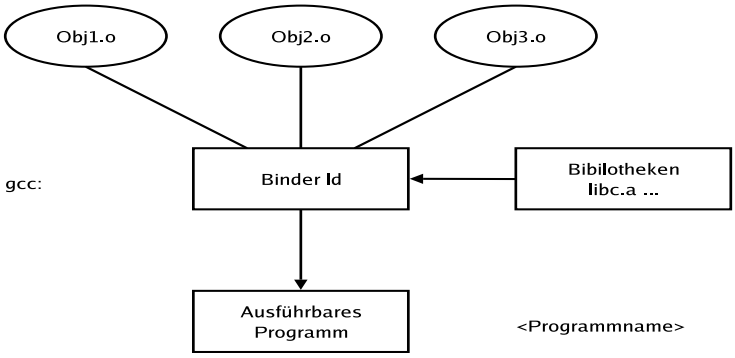
```
gcc [Parameter/Optionen] <Datei>
```

- Der GCC-Compiler verarbeitet C-Quellcode Dateien in mehreren Schritten. Gcc selbst ist nur ein sog. Frontend- Programm, welches während der einzelnen Phasen der Übersetzung weitere Programme aufruft und Daten weiterleitet und Ergebnisse aufnimmt.



# Der GCC-Compiler (II)

- Damit aus einer Quell- eine Objektdatei, aber keine ausführbare Programmdatei entsteht, muß die `-c` Option gesetzt werden:
  `gcc -c <Datei>`
- Die Ausgabedatei kann explizit angegeben werden mittels der `-o` Option:
  `gcc -c <Quell.c> -o <Obj.o>`
- Besteht ein Programm aus mehreren Quelldateien, wird dieser Vorgang für jede Quelldatei wiederholt.
  Schließlich muß ein ausführbares Programm erzeugt werden:
  `gcc -o <Programmname> <obj1.o> <obj2.o> •••`
- Erzeugung eines ausführbaren Programms mit gcc:

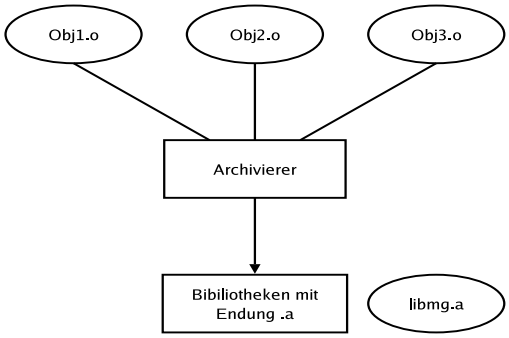





# Der GCC-Compiler (III)

- Archivierung von Objektdateien mit dem `ar` Programm:
  `ar ca <libx.a> <obj1.o> <obj2.o> •••`
- Mehrere Objektdateien könne zu einer Bibliothek (Archiv) zusammengefaßt werden. Hier muß ein eigenes Archivierungsprogramm `ar` verwendet werden.
- Anschließender Bindeprozeß:
  `gcc -o <Programmname>`
  `-l<x> <obj1.o> <obj2.o> •••`
- Beispiel einer Multimodul-Compilierung mit Archivierung und Bindung.

```
gcc –c eval.c
gcc –c delete.c
gcc –c search.c
ar ca libtree.a eval.o delete.o
                search.o
gcc –c main.c
gcc –o myprog –ltree main.o
```



# Standardbibliotheken (I)

- Vom Betriebssystem oder Grafiksystem bereitgestellte Objekt-Bibliotheken, die standardisierte Funktionen und Variablen zur Verfügung stellen.
  - Std-C Bibliothek `libc.a`→`-lc`
  - Ein- und Ausgabefunktionen wie z.B. `printf`
  - Stringfunktionen wie z.B. `strcmp`
  - Prozeßkontrolle wie z.B. `exit`
  - Systemfunktionen wie z.B. `time`
  - X11-Grafikbibliotheken `libx11.a`→`-lx11`
  - Basis Grafikfunktionen wie `XCreateWindow`
  - Zeichenfunktionen wie z.B. `XPolyLine`
  - Fontfunktionen wie z.B. `XQueryFont`
- Um Funktionen aus (Standard-) Bibliotheken nutzen zu können, benötigt man die entsprechenden Funktionsdeklarationen, die Rückgabetyp, Namen und Argumenttypen angeben. Zusätzlich werden Deklarationen von (abstrakten) Datentypen benötigt, wie z.B. für die `time`-Funktion.



# Standardbibliotheken (II)

- Beispiel:

```
time_t time(time_t *t);
typedef long time_t;
int gettimeofday(struct time_val *tv,
     struct timzone *tz);
struct timeval{
     long tv_sec;
     long tv_usec;
};
```

- Alle Funktionen, Datentypen und Variablen, die aus Bibliotheken exportiert werden, werden in sog. Header-Dateien mit der Dateieindung .h bekanntgegeben. Diese Header-Dateien müssen in den Quelltext eingebunden werden:

```
#include <header.h>
```

**Beispiele:**

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <x11.h>
```

# Abstrakte Datentypen (I)

- Zentraler Bestandteil der Algorithmik sind sogenannte abstrakte Datentypen (ADT). Diese sind gekennzeichnet durch:

  1. eine Datenstruktur, die sich mit Mitteln einer Programmiersprache implementieren läßt,
  2. und Operationen auf diesem Datentyp, die überhaupt erst einen ADT definieren.

- Charakteristisch von ADTs ist, daß sie nicht direkt von einer speziellen Programmiersprache implementiert bzw. unterstützt werden.
- Z.B. bieten verkettete Listen ein Möglichkeit, Datensätze gleichen Datentyps zu verknüpfen, ähnlich wie bei Arrays. Die Programmiersprache C unterstützt Arrays, aber keine Listen (hier ADT), so daß diese mit Mitteln der Programmiersprache implementiert werden müssen. Die funktionale Programmiersprache ML bietet hingegen Unterstützung für Arrays und Listen (hier kein eigentlicher ADT).
- Man unterscheidet in der Informatik:

  **Datentyp**
  die in der Programmiersprache vorhandenen Grunddatentypen, z.B. INTEGER oder CHAR, bei denen alle Grundoperationen wie Komposition (Addition) oder Wertzuweisung und Werterhalt bereits vorhanden/definiert sind.



# Abstrakte Datentypen (II)

- Man unterscheidet in der Informatik (Forts.):

  **Abstrakter Datentyp ADT**
  betseht aus einer Menge von Datenobjekten und Operationen, die auf diesen Datentyp angewendet werden können, die sog. Algorithmen.
  Beispiel: Liste →
  Objekte: Listenelemente
  Operationen: Listenelemente hinzufügen, entfernen, suchen, •••

  **Datenstruktur**
  Realisierung der Objektmengen eines ADT's mit Mitteln der Programmiersprache.

- Die Implementierung eines ADT wie Listen setzt daher Datenstrukturen und Funktionen, die mit diesen Datenstrukturen operieren, voraus.
- Elementare und häufig verwendete ADT:

  - Lineare Listen
  - Stapelspeicher (Stack)
  - Schlangenspeicher (Queue)
  - Bäume (Trees)
  - Graphen als Verallgemeinerung von Bäumen

# Abstrakte Pseudo-Notation (I)

- Die Beschreibung der Operationen auf ADTs nimmt daher eine zentrale Stellung ein.
- Die Algorithmik ist aber i.A. unabhängig von einer konkreten Programmiersprache wie C oder Pascal. Es ist daher zweckmäßig, die Operationen nicht mit einer konkreten Programmiersprache, sondern mit einer abstrakten Pseudo-Notation (APN) vereinfacht darzustellen, um die wesentlichen Schritte des Algorithmus zu verdeutlichen, und eine Duplizierung der Erklärung für verschiedene Programmiersprachen zu vermeiden.
- Alternative symbolische Darstellungsmethoden wie Flußdiagramme sind in vielen Fällen zu unübersichtlich. Eine Pseudo- oder abstrakte Notation wird vielfach in der Fachliteratur für Algorithmik und wissenschaftlichen Publikationen eingesetzt.
- Diese Notation ist nicht standardisiert, sondern wird vielmehr von Autoren nach Belieben definiert. Ausgangspunkt ist i.A. die Programmiersprache Pascal oder mittlerweile Java.
- Die APN enthält nur zum Verständnis wesentliche Anweisungen und Ausdrücke, so wird z.B. im Gegensatz zu C auf explizite Variablendefinition verzichtet.
- Stattdessen werden Bezeichner direkt in Ausdrücken eingeführt, wenn sie benötigt werden.



# Abstrakte Pseudo-Notation (II)

- Objektdefinition und Verwendung in Ausdrücken:

```
Allgemein für veränderliche Objekte:
  DEF obj == type⟨:= init⟩;
    obj←expr;
    obj∈expr
Variablen:
  DEF v == VAR value⟨:= init⟩;
```

- Die Typendefinition (Produktbildungstyp) ist ähnlich der Strukturdefinition in der Programmiersprache C, Typdefinition eines abstrakten Datentyps und Objektdefinition (Instanziierung eines Datentyps, z.B. dieses ADTs):

```
TYPE tpname == Constr(
  el1: eltype1;
  el2: eltype2;
  •••
);
Erzeugung eines Datenobjekts
  DEF obj == Constr⟨(init)⟩;
Zugriff auf ein Typelement
  obj.el
```

# Abstrakte Pseudo-Notation (III)

- Beispiel Typdefinition und Objekterzeugung:

```
TYPE datum == Datum(
  tag: INT;
  monat: INT;
  jahr: INT;
);
DEF dienstag == VAR 0;
dienstag ← 27;
DEF heute == Datum();
heute.tag ← dienstag;
heute.monat ← 6;
heute.jahr ← 2006;
⇔
```

```
C:
struct datum {
  int tag;
  int monat;
  int jahr;
};
struct datum heute;
heute.tag=27;
heute.monat=6;
heute.jahr=2006;
```

- Die Typerzeugungsfunktion Datum() enthält hier zur Vereinfachung keine explizite Initialisierung, es wird stattdessen implizite Initialisierung angenommen!



# Abstrakte Pseudo-Notation (IV)

- Im Gegensatz zu konkreten Programmiersprachen findet bei der abstrakten Notation kein Speichermanagement statt (implizit). Weiterhin sind parametrisierte Typedefinitionen möglich, so z.B. wenn die Größe eines Arrays erst bei der Instanziierung eines Typs festgelegt wird.
- Parametrisierte Typedefinition und entsprechende Umsetzung in C:

```
TYPE skipelement == SEL (
  next: ARRAY INT[1..höhe];
  höhe: INT;
);
⇔
C:
struct skipelement {
  int *next;
  int hoehe;
};
•••
struct skipelement skipel1;
skipel1.hoehe=4;
skipel1.next=(int *)malloc(sizeof(int)*skipel1.hoehe);
```

# Abstrakte Pseudo-Notation (V)

- Zusätzlich zur Produktbildung gibt es die Summenbildung als Typdefinition, um eine Vielzahl verschienartiger Typen zu gruppieren.
- Typdefinition Summenbildung.

```
TYPE tpname == (
    Constr1 (type1)
  | Constr2 (type2)
  •••
);
◆
Erzeugung eines Datenobjekts:
  DEF obj == Constr (⟨init⟩);
Zugriff auf Summentype nur mit match-with:
  MATCH obj WITH
  | Constr1 (x) ⇒...
  | Constr2 (x) ⇒...
  END;
```



# Abstrakte Pseudo-Notation (VI)

- Neben Datenobjekten gibt es wie in C auch Referenzen auf Datenobjekte. **Zwischen Referenzen und den eigentlichen Datenobjekten wird aber im folgenden nicht näher unterschieden**, da sich die eigentlichen Operationen auf Datenobjekten immer auf den Datenkontainer bezieht. Referenzarithmetik wird nicht verwendet.
- Referenz (Zeigertyp)

```
REF type
Beispiel:
  DEF x == 0
  DEF r == REF x
Zuweisung:
  r := !r + 1;
```

- Die elementaren Datentypen entsprechen den gängigen von imperativen Programmiersprachen: {INTEGER,FLOAT,CHAR,STRING,BOOL}
- Die einsortige Produktbildung wird auch hier mit Arrays ausgedrückt.
- Defintion von einem Array und Elementzugriff:

```
DEF x == ARRAY type[range];
range: A..B mit A < B
◆
Arrayelement in einem Ausdruck selektieren:
obj.[index]
```

# Abstrakte Pseudo-Notation (VII)

- Beispiel Arrays.

```
(* Array Objekt definieren *)
DEF tage == ARRAY datum[1..365];
tage[1].tag ← 1;
⇔ C:
struct datum tage[365];
tage[0].tag=1;
```

- Neben Datenobjekten und Typen, die der Datenstrukturierung dienen, gibt es Funktionen und Prozeduren für die namentliche Komposition und Strukturierung von Programmanweisungen. Prozeduren besitzen im Gegensatz zu Funktionen keinen Rückgabewert. Man unterscheidet die Deklaration (Bekanntgabe der Schnittstelle) und Definition (Bekanntgabe der Programmanweisungen, d.h. Beschreibung des Verhaltens) von Funktionen und Prozeduren.
- Funktions- und Prozeduredeklaration. Bei der Funktion gibt der letzte Typ den Rückgabewert an.

```
FUN fname:  type1×
            type2...→
            typeres;
PRO pname: type1 ×
           type2;
```



# Abstrakte Pseudo-Notation (VIII)

- Funktions- und Prozeduredefinition.

```
DEF fname == λ arg1,arg2... •
  ••• (* Implementierung der Funtion *)
  → rückgabewert
DEF pname == λ arg1,arg2... •
  •••
  Programmanweisungen
```

- Eine Funktion muß wenigstens an einer Stelle im Programmfluß (Ende) explizit einen Funktionswert (Ergebnis) zurückgeben. Bei der Prozedur entfällt dieser Schritt.
- Funktionen und Prozeduren - Vergleich mit C:

```
FUN sq: INT×INT → INT;
DEF sq == λ a,b •
  res ← a*b;
  → res;
⇔ C:
int sq(int a,int b)
{ int res;
  res=a*b;
  return res;};
```

# Abstrakte Pseudo-Notation (IX)

- Neben der namentlichen Komposition von Anweisungsblöcken durch Funktionen gibt es Anweisungen zur sequenziellen Steuerung des Programmablaufs, wie bedingte Verzweigungen und Schleifen.
- Bedingte Verzweigungen:

```
IF expr THEN
  ••• Anweisungen
ELSE
  ••• Anweisungen
END;
MATH expr IWITH
  | val1⇒...
  | val2⇒...
  | _ ⇒...
END;
```

- Beispiele:

```
DEF fac == λ n •
  IF n = 1 THEN →1 ELSE → n*fac(n−1) END;
⇔ C:
int fac(int n) { if (n==1) return 1 else
                           return (n*fac(n−1));};
```



# Abstrakte Pseudo-Notation (X)

- Schleifen:

```
WHILE <expr>
DO
  ••• Anweisungen
DONE;
◆
FORindex = a TO b  | a DOWNTO b
DO
  ••• Anweisungen
DONE;
```

- Beispiele:

```
DEF fac == λ n •
  LET r == VAR 1 IN
  FOR i == 2 TO n DO r ← r * i; DONE;
  →r;
⇔ C:
int fac(n) {int r =1; int i;
  for(i=2;i<n;i++) r=r*i;
  return r;};
```

# ADT: Lineare Listen (I)

- Eine lineare Liste ist eine endliche Folge von Elementen. Das Element kann von einem beliebigen Typ sein, auch zusammengesetzt als Datenstruktur.
- Eine lineare Liste ist eine endliche Folge von Elementen. Das Element kann von einem beliebigen Typ sein, auch zusammengesetzt als Datenstruktur:

```
TYPE listenelement == Listenelement(
  key: INTEGER;  oder allg. keytype
  data: datatype;
);
```

- Der Schlüssel *key* dient als Identifizierungselement, hier ein ganzzahliger Index.
- Der eigentliche Dateninhalt ist *data*, der durch die Liste verknüpft werden soll.
- Man unterscheidet:

  **Geordnete Liste**
  Der Schlüssel bestimmt die Verknüpfung eines Listenelements innerhalb einer Liste.

  **Ungeordnete Liste**
  Der Schlüssel kann entfallen oder nicht beachtet werden.

- Alle Listenelemente werden bzw. können mit einer gemeinsamen Datenstruktur `TYPE` *liste* zusammengefasst werden.



# ADT: Lineare Listen (II)

Operationen auf lineare Listen

**Einfügen**
Ein neues Element x vom Grundtyp `listenelement` soll in der Liste L an der Position p eingefügt werden:

```
PRO Einfügen :
  x: listenelement ×
  p: INTEGER ×
  L: liste;
```

Die Liste ändert sich daher:

$$f : L \rightarrow L' \tag{1}$$

$$L = \{a_1, a_2, .., a_N\} \tag{2}$$

$$L' = \left\{a_1, a_2, ..., a_{p-1}, x, a_p, .., a_N\right\} \tag{3}$$

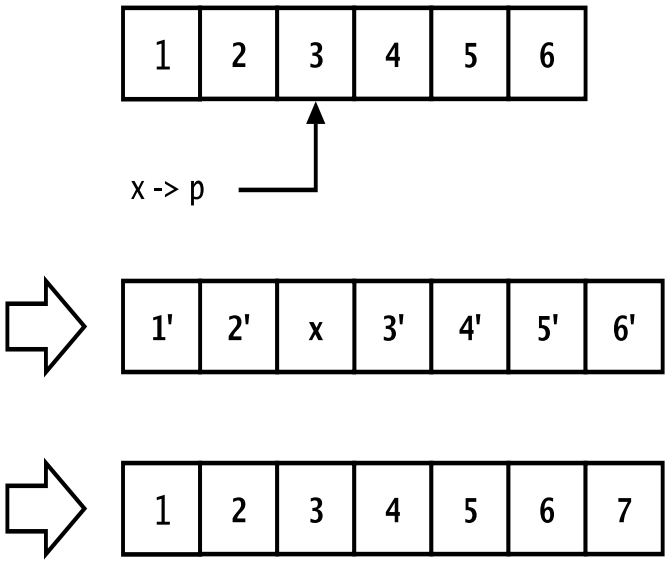
Sonderfälle:
`Einfügen(x,1,L)` → HEAD
`Einfügen(x,N,L)` → TAIL

# ADT: Lineare Listen (III)

Operationen auf lineare Listen

**Einfügen**

Beispiel: Element x soll an der Position p=3 eingefügt werden.






# ADT: Lineare Listen (IV)

Operationen auf lineare Listen

**Entfernen**

Entfernen eines Listenelements.

```
PRO Enfernen :
  p: INTEGER →
  L: liste;
```

Das Entfernen eines Elements $x_p$ an der Position p ändert die Liste, alle Elemente ab Position p+1 müssen nach links verschoben werden:

$$f : L \rightarrow L' \tag{4}$$

$$L' = \left\{a_1, a_2, ..., a_{p-1}, a_{p+1}, .., a_N\right\} \tag{5}$$

# ADT: Lineare Listen (V)

Operationen auf lineare Listen

**Zugriff**

Zugriff auf ein Listenelement.

```
FUN Zugriff :
  p: INTEGER ×
  L: liste →
  x: listenelement;
```

Diese Operation liefert das Element $x_p$ an der p-ten Position der Liste.

**Suche**

Suche nach einem Listenelement.

```
FUN Suchen :
  x: listenelement ×
  L: liste →
  p: INTEGER;
∨
FUN Suchen :
  key: INTEGER ×
  L: liste →
  x: listenelement;
```



# ADT: Lineare Listen (VI)

Operationen auf lineare Listen

**Suche**

- Diese Operation liefert I.) die Position des Elements x in der Liste, falls x∈L, sonst NONE.
- Operation nicht eindeutig, wenn x mehrmals in der Liste vorkommt.
- Suche ist als Möglichkeit II.) praktisch sinnvoller mit einem **Schlüssel** *key* durchführbar und liefert als Rückgabewert das Element $x_{key}$ oder NONE.

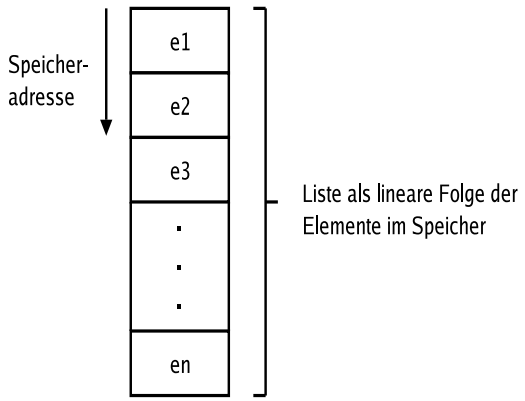
◆◆◆

**Listen können auf zwei Arten implementiert werden**:

1. Sequenziell gespeicherte Listen mit Arrays,
2. verkettet gespeicherte Listen.

# Sequenzielle Speicherung von Listen (I)

- Speicherung von Listen mit Feldern/Arrays. Auf das i-te Element der Liste kann durch direkten Feld≡Adreßzugriff zugegriffen werden.
- Speichermodell einer linearen Liste.




- Sequenziell gespeicherte lineare Liste.

```
TYPE liste == Listseq (
  elements: ARRAY listenelement[1..N];
  elzahl: INTEGER;
);
```



# Sequenzielle Speicherung von Listen (II)

- Das zweite Typelement *elzahl* vermerkt die aktuelle Größe der Liste (Anzahl der gespeicherten Listenelemente < N).
- Elementare Operation auf linearen Listen:

**Einfügen**
Ein neues Element an eine bestimmten Position in der Kette einfügen.

**Entfernen**
Ein vorhandenes Element aus der Liste entfernen.

**Suchen**
Nach einem Element anhand des Schlüssels in der Liste suchen, oder alternativ die Position eines gegebenen Listenelements ermitteln.

- Beispiel seq. linearen Liste mit statischer Speicherreservierung, d.h. die Größe des Arrays ≡ maximale Länge der Liste muß zur Kompilierungszeit bereits bekannt sein.

```
C:
struct listenelement {
  int key;
  char name[80];
};
#define N 100
struct liste {
  struct listenelement elements[N];
  int elzahl;
};
```

# Sequenzielle Speicherung von Listen (III)

- Beispiel seq. lineare Liste, mit dynamischer Speicherreservierung zur Laufzeit.

```
struct listedyn {
  struct listenelement *elements;
  int elzahl;
  int elmax;
};
struct liste mylist;
struct listedyn mylistdyn;
...
{
  mylist.elzahl=0;
  mylistdyn.elements=(struct listenelement *)
              malloc(sizeof(struct listenelement)*N);
  mylistdyn.elmax=N;
  mylistdyn.elzahl=0;
};
```



# Sequenzielle Speicherung von Listen (IV)

- Operationen Lineare Liste bei sequenzieller Speicherung.

**Einfügen**

Fügt ein neues Element x an der Position p in die Liste ein.

```
PRO Einfügen :
  x: listenelement×
  p: INT ×
  L: liste;
◆
DEF Einfügen == λ x,p,L •
  IF L.elzahl = MAXZAHL THEN FEHLER Liste voll;
  IF p > L.elzahl + 1 OR p < 1 THEN
    FEHELR Index;
  FOR pos = L.elzahl DOWNTO p
  DO
    L.elements[pos+1] ← L.elements[pos];
  DONE;
  L.elements[p] ← x;
  INCR L.elzahl;
```

# Sequenzielle Speicherung von Listen (V)

➤ Operationen Lineare Liste bei sequenzieller Speicherung.

**Entfernen**

Entfernt das Element an Stelle p (Index des Arrays) aus der Liste.

```
PRO Entfernen :
  p: INT ×
  L: liste;
◆
DEF Entfernen == λ p,L •
  IF L.elzahl = MAXZAHL THEN FEHLER Liste voll;
  IF p > L.elzahl + 1 OR p < 1 THEN
    FEHELR Index;
  FOR pos = p TO L.elzahl
  DO
    L.elements[pos] ← L.elements[pos+1];
  DONE;
```



# Sequenzielle Speicherung von Listen (VI)

➤ Operationen Lineare Liste bei sequenzieller Speicherung.

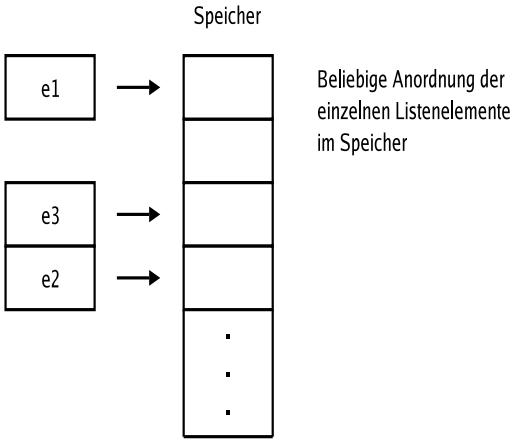
**Suchen**

Liefert das Element mit dem Schlüssel key oder NONE falls kein solches Element gefunden wurde..

```
FUN Suchen :
  key: INT ×
  L: liste →
  p: INT | el: listenelement;
◆
DEF Suchen == λ key L •
  pos ← 0;
  WHILE pos ≤ L.elzahl AND L.lements[pos].key ≠ key
  DO
    INCR pos;
  DONE;
  IF pos > L.elzahl THEN
    → NONE;
  ELSE
    → L.elements[pos]; (* Listenelement oder pos *)
```

# Verkettete Listen (I)

- Bei der sequenziellen Speicherung von linearen Listen kann jedes Listenelement durch eine Adreßrechnung direkt bestimmt werden, man sagt dieser Zugriff benötigt eine Laufzeit der Größenordnung $\Theta(1)$.
- Nachteil: Größe (maximale) der Liste muß bekannt sein, und ein Teil des für die Liste reservierten Speicherplatzes ist i.A. ungenutzt, und Einfügen und Entfernen von Listenelementen ist eine aufwendige Operation mit einer Laufzeit der Größenordnung $\Theta(N)$.
- Für dynamische Listen, deren Größe im voraus nicht bekannt ist, wird jedes Listenelement einem eigenen Speicherbereich zugeordnet.
- Speichermodell einer dynamischen Liste:





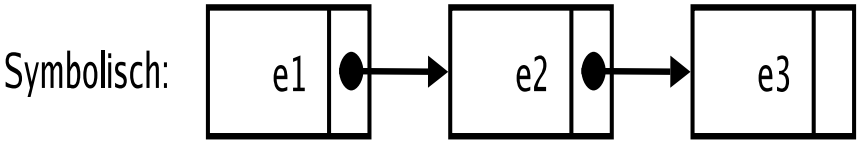

# Verkettete Listen (II)

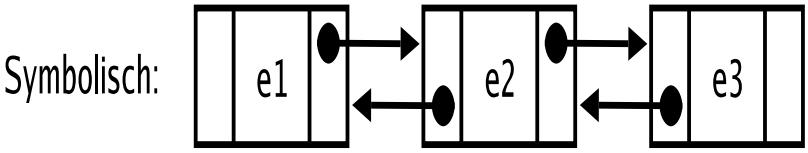
- Verkettete Speicherung einer Liste ist eine dynamische Struktur. Man unterscheidet:

  **Einfach verkettete Listen**
  Jedes Listenelement hat nur einen Zeiger auf das nächste Element.

  **Zweifach verkettete Listen**
  Jedes Listenelement hat zwei Zeiger, jeweils einen auf das nächste und das vorherige Listenelement.

- Einfach verkettete Liste (links)
  Doppelt verkettete Liste (rechts)



# Verkettete Listen (III)

- Listenelemente von einfach- und doppelt verketteten Listen:

```
TYPE listenelement1 == Liste1(
  key: keytype;
  data: datatype;
  next: ↑listenelement1;
);
TYPE listenelement2 == Liste2(
  key: keytype;
  data: datatype;
  prev: ↑listenelement;
  next: ↑listenelement;
);
```

- Nachteile von Verkettung von Listen:
  1. Zeiger benötigen zusätzlichen Speicherplatz,
  2. einfach verkettete Listen ermöglichen das Suchen von Elementen immer nur in einer Richtung, beginned vom Kopf der Liste (oder Ende).
- Vorteile von (doppelter) Verkettung von Listen:
  1. Das Einfügen von Listenelementen benötigt nur max. vier Operationen:
     1. Listenelement $x_p$: Zeiger ↑prev und ↑next setzen
     2. Listenelement $x_{p-1}$: Zeiger ↑next auf $x_p$
     3. Listenelement $x_{p+1}$: Zeiger ↑prev auf $x_p$



# Verkettete Listen (IV)

- Vorteile von (doppelter) Verkettung von Listen (Forts.):
  2. Entfernen von Listenelementen benötigt äquivalent zum Einfügen nur max. 2 Operationen.
  3. Speicherbedarf orientiert sich an aktueller Größe der Liste und ist dynamisch angepasst.
- Suchen in verketteten Listen benötigt bis zu $\Theta(N)$ Operationen. Suche kann Voraussetzung für Einfüge- und Entfernungsoperationen sein!
- Bei dynamisch gespeicherten Listen gibt es keine globale Datenstruktur, sondern nur eine auf lokaler Basis eines jeden Listenelements.
- Der Zugang zur Liste findet mit Referenzen auf das erste und letzte Listenelement statt:
  `head: ↑`*`listenelement`*
  `tail: ↑`*`listenelement`*
- Das erste und letzte Element einer Liste muß kenntlich gemacht werden:
  1. Einfach verkettete Liste:
     $e_{head}: -$
     $e_{tail}: \ next = Ende = NONE$

# Verkettete Listen (V)

- Das erste und letzte Element einer Liste muß kenntlich gemacht werden (Forts.):

  2. Doppelt verkettete Liste:
     $e_{head} : prev = ANFANG = NONE$
     $e_{tail} : next = Ende = NONE$
  3. Ringstruktur:
     $1 + 2VK : e_{tail} : next = \uparrow e_{head}$
     $2VK : e_{head} : prev = \uparrow e_{tail}$

- Beispiele für verkettete Listenstrukturen in C:

```
struct teleintrag {
  char name[80];
  char strasse[80];
  char telefon[80];
  struct teleintrag *prev;
  struct teleintrag *next;
};
struct person {
  char name[80];
  int jahrgang;
};
struct personen_liste {
  struct person wichtig;
  struct personen_list *next;
};
```



# Verkettete Listen (VI)

- Bei der folgenden Beschreibung der Algorithmen wird nur noch auf zweifach verkettete Listen eingegangen. Die elementaren Operationen sind die gleichen wie bei sequenziell gespeicherten linearen Listen.

```
PRO Einfügen :
  x: listenelement2 ×
  p: listenelement2 ×
  head: listenelement2 ×
  tail: listenelement2;
◆
PRO Entfernen :
  x: listenelement2 ×
  head: listenelement2 ×
  tail: listenelement2;
◆
FUN Suchen :
  key: INT ×
  head:listenelement2 →
  x:listenelement2;
```

# Verkettete Listen (VII)

- Implementierung

```
DEF Einfügen == λ x,p,head,tail •
  (*
  ** Füge Element x vor Element p ein
  *)
  IF p = head THEN prev ← NONE;
    ELSE prev ← p.prev;
  x.prev ← prev;
  x.next ← p;
  p.prev ← x;
  IF prev ≠ NONE THEN prev.next ← x;
◆
DEF Entfernen == λ x,head,tail •
  IF x = tail THEN next ← NONE;
    ELSE next ← x.next;
  IF x = head THEN prev ← NONE;
    ELSE prev ← x.prev;
  IF next ≠ NONE THEN next.prev ← prev;
  IF prev ≠ NONE THEN prev.next ← next;
```



# Verkettete Listen (VIII)

- Implementierung (Forts.)

```
DEF Suchen == λ key,head •
  found ← NONE;
  el ← head;
  WHILE found = NONE AND el ≠ NONE
  DO
    IF el.key = key THEN found ← el;
    el ← el.next;
  DONE;
  → found;
```

# Verkettete Listen (IX)

➤ Zusammenfassung und Effizienz der beiden Kettenarten:

**L1**
einfach verkettet, Zugang zur Liste mit ↑head

**L2**
einfach verkettet, Zugang zur Liste mit ↑head und ↑tail

**L3**
doppelt verkettet, Zugang mit ↑head und ↑tail

Tabelle: Effizienz verschiedener Verkettungstechniken von Listen.
➥ A: Anfang, E: Ende, P: Listenelement.

| | L1 | L2 | L3 |
|---|---|---|---|
| Einfügen A | $\Theta(1)$ | $\Theta(1)$ | $\Theta(1)$ |
| Einfügen E | $\Theta(N)$ | $\Theta(N)$ | $\Theta(1)$ |
| Einfügen P | $\Theta(N)$ | $\Theta(N)$ | $\Theta(1)$ |
| Entfernen P | $\Theta(N)$ | $\Theta(N)$ | $\Theta(1)$ |
| Suchen P | $\Theta(N)$ | $\Theta(N)$ | $\Theta(N)$ |
| Hintereinander-hängen L1⊕L2 | $\Theta(N)$ | $\Theta(1)$ | $\Theta(1)$ |



# Stapel- und Schlangenspeicher (I)

➤ Listen erlauben das Entfernen und Einfügen von Datensätzen an beliebiger Stelle. Viele Anwendungen benötigen Operationen nur für den Anfang und das Ende der Liste:

**PushHead**
Fügt Element x am Anfang der Liste L ein (neues Kopfelement).

**PushTail**
Fügt Element x am Ende der Liste L ein (neues Endelement).

**PopHead**
Entfernt Kopfelement x von der Liste L.

**PopTail**
Entfernt Endelement x von der Liste L.

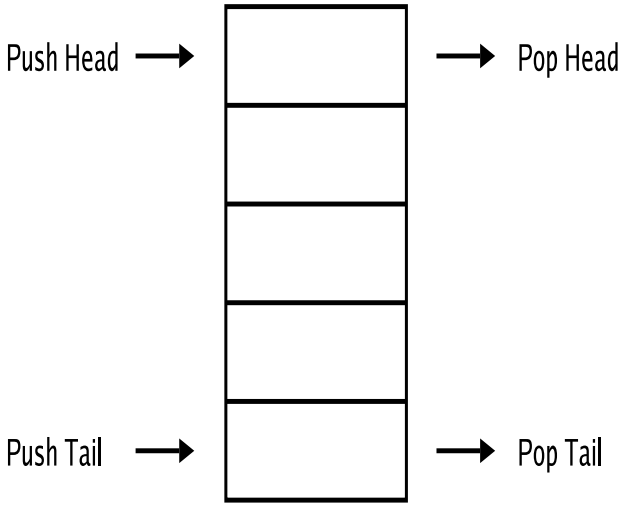
**Top**
Liefert Kopfelement ohne dieses zu entfernen.

**Bottom**
Liefert Endelement ohne dieses zu entfernen.

# Stapel- und Schlangenspeicher (II)

- Spezielle Operationen für den ADT Liste:

```
PRO  PushHead :
  x: listelement →
  L: liste;
PRO  PushTail :
  x: listelement →
  L: liste;
FUN  PopHead :
  L: liste →
  x: listelement;
FUN  PopTail :
  L: liste> →
  x: listelement;
```

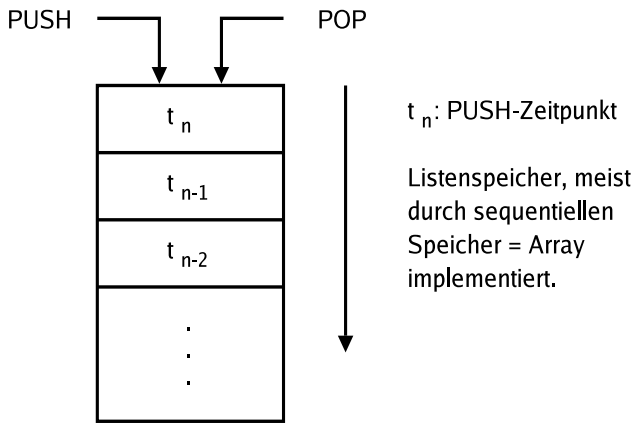





# Stapel- und Schlangenspeicher (III)

## A. Stapelspeicher (Stack)

- Spezialfall einer Liste mit den Operationen: {Push≡PushHead,Pop≡PopHead}.
- Element, das zuletzt in den Stapel eingefügt wurde (PushHead), wird auch als erstes wieder entfernt oder gelesen (PopHead). Man spricht von einem LastIn-FirstOut-Speicher (LIFO).
- Bekanntes Beispiel: Programmstack. Implementierung häufig mit Arrays.
- Zugriff auf einen LastIn-FirstOut-Speicher (LIFO).



# Stapel- und Schlangenspeicher (IV)

## A. Stapelspeicher (Stack)

➤ ADT Stapelspeicher:

```
TYPE stack == Stack(
  elements: ARRAY elementtype[1..N];
  tail: INTEGER;
);
◆
DEF Push == λ S,x •
  IF S.tail > N THEN FEHLER Full;
  S.elements[S.tail] ← x;
  INCR S.tail;
◆
DEF Pop == λ S •
  IF S.tail = 1 THEN FEHELR Empty;
  x ← S.elements[S.tail−1];
  DECR S.tail;
  → x
◆
DEF Init == S •
  S.tail ← 1;
```



# Stapel- und Schlangenspeicher (V)

## A. Stapelspeicher (Stack)

➤ Beispiel eines Stackspeichers für den Datentyp integer.

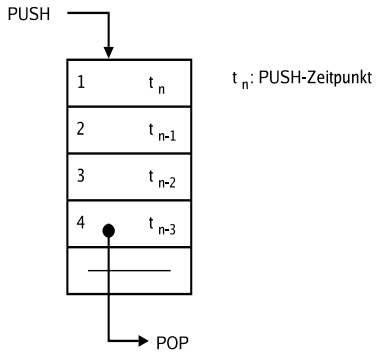
```
C:
#define N 100
struct stack {
  int s_elems[N];
  int s_tail;
};
◆
struct stack intstack;
void init()
{
  stack.s_tail=0;
};
◆
void push(struct stack *s,int x)
{
  if (s->s_tail == N) error("full");
  s->s_elems[s->s_tail]=x;
  s->s_tail=s->s_tail+1;
  return;
};
```

```
void push(struct stack *s,int x)
{
  if (s->s_tail == N) error("full");
  s->s_elems[s->s_tail]=x;
  s->s_tail=s->s_tail+1;
  return;
};
◆
int pop(struct stack *s)
{
  int x;
  if (s->s_tail == 0) error("empty");
  x=s->s_elems[s->s_tail−1];
  s->s_tail=s->s_tail−1;
  return x;
};
```

# Stapel- und Schlangenspeicher (VI)

## B. Schlangenspeicher (Queue)

- Spezialfall einer Liste mit den Operationen: {Push≡PushHead,Pop≡PopTail}.
- Element, das zuerst in die Schlange eingefügt wurde (PushHead) (das älteste), wird auch als erstes wieder entfernt oder gelesen (PopHead). Man spricht von einem FirstIn-FirstOut-Speicher (FIFO).
- Zugriff auf FirstIn-FirstOut-Speicher (FIFO):




- Bekanntes Beispiel: Pufferspeicher bei der Datenübertragung zwischen zwei Geräten mit unterschiedlicher Datenverarbeitungsgeschwindigkeit. Implementierung häufig mit Arrays.



# Stapel- und Schlangenspeicher (VII)

## B. Schlangenspeicher (Queue)

- Beispiel: Der Sender überträgt 1000 Byte innerhalb einer Zeitdauer von 1ms → Datenrate beträgt 1MByte/s. Der Empfänger kann aber nur eine Datenrate von 0.5 MBytes/s verarbeiten, d.h. er bnötigt 2ms, um die Daten aus dem FIFO zu lesen, oder der Empfänger ist kurzzeitig mit anderen Operationen beschäftigt.
- ADT Schlangenspeicher

```
TYPE queue == Queue(
  elements: ARRAY elementtype[1..N];
  head: INTEGER;
  tail: INTEGER;
);
◆
PRO Push == λ S,x •
  next ← (Q.tail MOD N)+1; (* für Bereich 1..N *)
  IF next = Q.head AND NOT Q.head=Q.tail THEN FEHLER Full;
  Q.elements[Q.tail] ← x;
  Q.tail ← next;
```

# Stapel- und Schlangenspeicher (VIII)

## B. Schlangenspeicher (Queue)

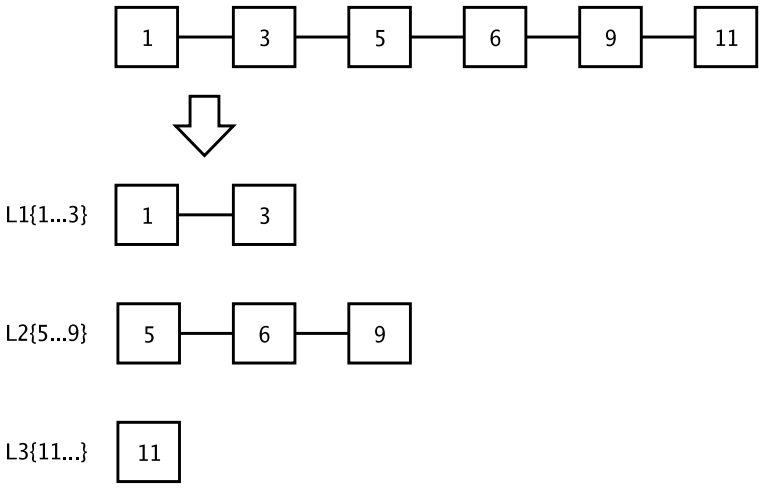
- ADT Schlangenspeicher (Forts.)

```
DEF Pop == λ S •
  IF Q.tail = Q.head THEN FEHELR Empty;
  x ← Q.elements[Q.head];
  Q.head ← (Q.head MOD N) + 1;
  → x
◆
DEF Init == λ S •
  Q.head ← 1;
  Q.tail ← 1;
```



# Skip-Listen (I)

- Lineare Listen benötigen Θ(N) Operationen bzw. Iterationen für die Suche von Listenelementen. Der Wert Θ() gibt eine Größenordnung an, die im ungünstigsten Fall benötigt wird. Sie dient der Vergleichbarkeit der Effizienz zwischen Algorithmen. Eine Suchoperation kann Voraussetzung für Einfüge- und Löschoperationen sein.
- Sind die Listenelemente so angeordnet, daß der Suchschlüssel, z.B. eine Integer-Zahl, auf- oder absteigend sortiert ist, ist eine Bereichsstrukturierung der Listenelemente anhand ihres Schlüssels sinnvoll.
- Unterteilung einer Liste in (sortierte) Teillisten:



# Skip-Listen (II)

- Bereichsstrukturierung einer Liste mit Skip-Struktur. Die Arrays der Typenstrukturen sind parametrisiert. Der Parameter ist M=höhe.

```
TYPE skipel == Skipel(
  data: datatype;
  key: keytype;
  next: ARRAY ↑skipel[1..M];
  höhe: INTEGER;
);
TYPE skipliste == Skipliste (
  kopf: ARRAY ↑skipel[1..M];
);
```

- Die Listenelemente werden so angeordnet, daß Bereiche des Schlüssels <key> bei der Suche übersprungen werden können.
- Bereichsstrukturierung ist ein nicht vollständig bestimmtes Problem, d.h. es gibt eine Vielzahl von Anordnungsmöglichkeiten in Abhängigkeit von der Skip-Tiefe (bzw. Höhe).
- Bereichsstrukturierung ist ein nicht vollständig bestimmtes Problem, d.h. es gibt eine Vielzahl von Anordnungsmöglichkeiten in Abhängigkeit von der Skip-Tiefe (bzw. Höhe).



# Skip-Listen (III)

- Jedes Listenelement ist durch einen Zeiger auf Niveau 1 (Erste Zelle des Arrays `next`) mit dem nächstfolgenden Listenelement verbunden.
- Jedes Listenelement der höheren Stufen überspringt eine Anzahl S von Listenelementen, und besitzt in Abhängigkeit der Stufe U Zeiger auf die nächsten Elemente der Stufen 1..U.
- Als Kopf der Liste kann alternativ ein leeres Listenelement mit maximaler Höhe h verwendet werden.

## Perfekte Skip-Listen

- Vollständig sortierte Liste: Jedes $2^i$-te Element hat einen Zeiger auf das $2^i$-Positionen weiter rechts entfernte Element für jedes $i = 0..log_2 N$. N ist dabei die Anzahl der Listenelemente.
- Die Anzahl der Suchoperationen einer Skip-Liste reduziert sich auf die Größenordnung $\Theta(log_2 N)$.

# Skip-Listen :: Perfekte Skip-Listen (I)

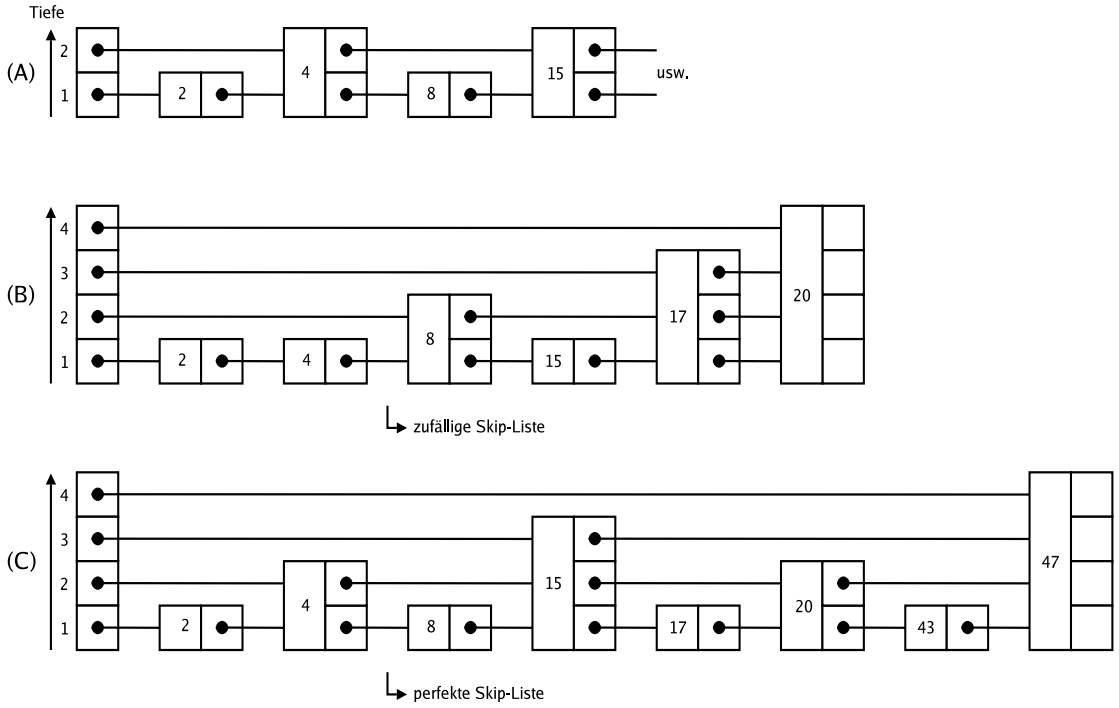
- Teil I: Operationen einer Skip-Liste.

```
FUN  Suchen :  k:keytype ×
               L:skipliste →
               x:skipel;
DEF  Suchen  == λ k,L •
  p ← L.kopf;
  FOR i = L.höhe DOWNTO 2
  DO
    WHILE p.next[i].key < k
    DO
      p ← p.next[i];
    DONE;
  DONE;
  p ← p.next[1];
  IF p.key = k THEN
    → p
  ELSE
    → NONE;
```



# Skip-Listen :: Perfekte Skip-Listen (II)

- Beispiele von Skip-Listen. (A): Skip-Liste der Höhe 2, (B): zufällige Skip-Liste der Höhe 4, (C): wie (B), aber als perfekte Skip-Liste der Höhe 4.



# Skip-Listen :: Randomisierte Skip-Listen (I)

- Bei perfekten Skip-Listen bedeutet das Einfügen oder Entfernen von Listenelementen eine vollständige Rekonfiguration der Liste.
- Abhilfe schaffen hier sog. randomisierte Skip-Listen. Bei denen findet eine Aufweichung des starren Schemas der perfekten Skip-Liste hin zu einer statistisch verteilten Gewichtung der Höhen der Skip-Elemente statt, so daß gilt:

$$\bar{n}_1 \approx n_1^{perfekt} \tag{6}$$

$$\bar{n}_2 \approx n_2^{perfekt} \tag{7}$$

$$\omega(p.hoehe = i) \approx \frac{1}{2^i} \tag{8}$$

- Die Einfüge-Operation einer randomisierten Liste beginnt mit der Suche eines Elements, daß sich in der Ordnung der Liste vor dem einzufügenden Element befindet.
- Die Höhe für das neue Element wird mittels eines Zufallszahlengenrators im Bereich 1..MAX-HÖHE bestimmt. Das neue Element wird dann in die Skip-Liste eingefügt.



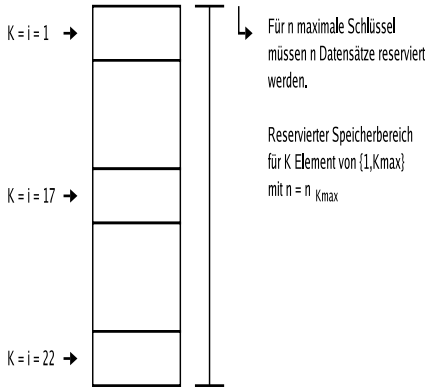
# Skip-Listen :: Randomisierte Skip-Listen (II)

- Teil II: Operationen einer Skip-Liste.

```
PRO Einfügen : x:skipel→
               L:skipliste;
◆
DEF Einfügen == λx,L •
  p ← L.kopf;

  FOR i = L.höhe DOWNTO 2
  DO
    WHILE p.next[i].key < k
    DO
      p ← p.next[i];
    DONE;
  DONE;

  p ← p.next[1];
  IF p.key = x.key THEN
    FEHLER <exists>;

  neuehöhe ← randomhöhe();

  IF neuehöhe > L.höhe THEN
    FOR i = L.höhe + 1 TO neuehöhe
    DO
      update[i] ← L.kopf;
    DONE;
    L.höhe ← neuehöhe;
  END IF;

  x.höhe ← neuehöhe;

  FOR i = 1 to neuehöhe
  DO
    x.next[i] ← update[i].next[i];
    update[i].next[i] ← x;
  DONE;
```

# Hash-Verfahren (I)

- Listen erlauben eine Menge von Datensätzen so zu speichern, daß die Operationen Suchen, Einfügen und Entfernen unterstützt werden.
- Jeder Datensatz ist gekennzeichnet durch einen eindeutigen Schlüssel, z.B. eine Integer-Zahl oder eine Zeichenkette.
- Zu jedem Zeitpunkt ist lediglich eine (kleine) Teilmenge $\sigma$ aller möglichen Schlüssel $\Sigma$ gespeichert.
- Das bedeutet bei einer linearen Speicherung der Datensätze mit einer Tabelle, wo der Index der Tabelle/des Arrays gleich (bzw. die Speicheradresse proportional) dem Schlüssel ist, eine große Tabelle mit geringer Zeilenausnutzung.
- Lineare Speicherung von Datensätzen mit einer Array-Tabelle.






# Hash-Verfahren (II)

- Vorteil einer direkten Zuordnungsfunktion

$$F(k) = \text{Speicheradresse } A \sim k \tag{9}$$

  liegt in der Suchzeit $\Theta(1)$!
- Wünschenswert wäre eine indirekte Zuordnungsfunktion, die eine bessere Ausnutzung der Datentabelle ermöglicht:

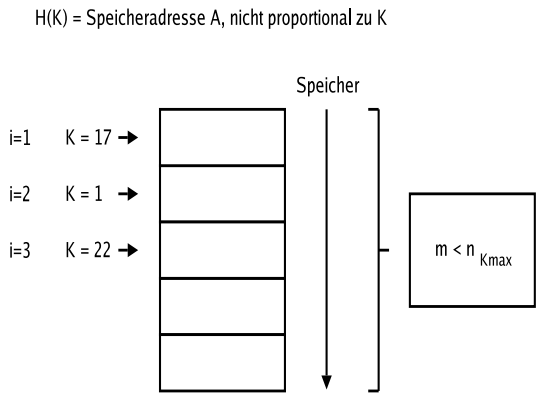
$$H(k) = \text{Speicheradresse } A \neq k \tag{10}$$

$$H : k \rightarrow A \tag{11}$$

- Die Funktion H nennt man Hash-Funktion. Die Datensätze werden in einer linearen Tabelle (Array) abgelegt, die man Hash-Tabelle nennt.
- Die Größe der Hash-Tabelle ist zweckmäßigerweise $m<n=n_{kmax}$. Die Hash-Funktion ordnet jedem Schlüssel k einen Index der Tabelle (oder eine Speicheradresse) h(k) mit $1 \leq h(k) \leq m$ im Falle eines Indexes zu.
- I.a. ist $\sigma$ eine sehr kleine Teilmenge von $\Sigma$. Z.B. gibt es $26 \bullet 36^{79}$ mögliche Schlüssel für eine Zeichenkette mit 80 Zeichen, die mit einem Buchstaben beginnt, und mit uchstaben oder Ziffern fortgesetzt wird. Beispiel: Symboltabelle eines Compilers.

# Hash-Verfahren (III)

- Speicherung der Datensätze mit einer Hash-Tabelle.




- Die Hash-Funktion besitzt die Eigenschaft, daß zwei verschiedene Schlüssel $k_1$ und $k_2$ die gleiche Hash-Adresse besitzen können:

$$h(k_1) = h(k_2) \tag{12}$$

$k_1$ und $k_2$ heißen dann Synonyme. Ergibt sich direkt aus der Tatsache m<n und wird als Mehrdeutigkeit bzw. Adreßkollision der Hash-Funktion bezeichnet. Die Adresskollision muß als Sonderfall getrennt behandelt werden.



# Hash-Verfahren (IV)

- Hash-ADT:
  Das Array enthält entweder die Datensätze oder Zeiger auf Datensätze. Im ersteren Fall muß der Datensatz eine zusätzliche Statusvariable <flag> enthalten, die angibt, ob ein Hash-Eintrag verwendet wird, und im zweiten Fall wird ein NONE-Zeiger für den nicht genutzten Fall verwendet.

```
TYPE hash == Hash(
  hashaddr: HashAddr;
  hashtab: ARRAY dataentry[HashAddr];
           | ARRAY ↑dataentry[HashAddr];
);
TYPE HashAddr == {1,...,M};
TYPE dataentry == Dataentry(
  key: keytype;
  ⟨flag: {USED,FREE}⟩
  data: datatype;
);
◆
FUN hashfun: key → HashAddr;
```

# Hash-Verfahren (V)

- Hash-ADT: Beispiel in C

```
struct entry {
  int key;
  int used;  /* Flag markiert ob Eintrag verwendet wird */
  char name[81];
  char street[81];
  char phone[81];
};
struct hash {
  int size; /* TabellengröSSe */
  struct entry *hashtable;
};
◆
void hash_init()
{
  int size=100;
  int i;
  hash.hashtable=(struct entry *)malloc(size *
                                  sizeof(struct entry));
  if (hash.hashtable == 0) error("no memory");
  hash.size=size;
  for(i=0;i<0;i++) hash.hashtable[i].used=0;
  return;
};
```



# Hash-Verfahren (VI)

- Eine gute Hash-Funktion sollte möglichst schnell berechenbar sein, und die zu speichernden Datensätze möglichst gleichmäßig, aber ohne erkennbare Struktur oder Ordnung auf den Speicherbereich verteilen, um Kollisionen zu minimieren.
- Problem: Die Schlüsselmenge $\sigma$={$k_1$, ..., $k_1$} ist meistens nicht gleichmäßig verteilt. Z.B. bestehen Variablenamen in Programmen häufig aus 1-2 Zeichen mit den Buchstaben {x,y,z} und den Zahlen {1,2,..}.
- Wahrscheinlichkeit der Adreßkollision kann nach [W.Feller, 1968] abgeschätzt werden:

$$\gamma \approx \frac{\mathrm{n}}{\sqrt{\pi m/2}} \tag{13}$$

  mit n als Anzahl der Schlüssel und m Größe der Hash-Tabelle.
- **Divisions-Rest-Methode**
  Die einfachste Hash-Funktion läßt sich mittels der mathematischen Modulo-Funktion (Rest einer Ganzzahldivision) realisieren:

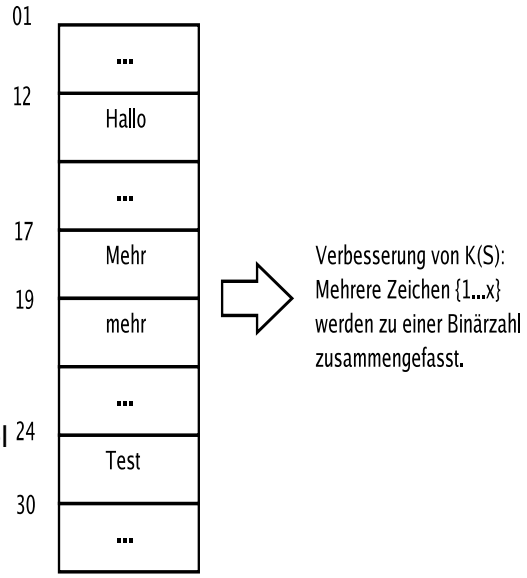
$$\mathrm{h(k)} = \mathrm{k \bmod m} \tag{14}$$

- m sollte nicht als Potenz der Basis B des Zahlensystems der Schlüssel gewählt werden, d.h. m $\neq B^i$, m sollte eine Primzahl sein.

# Hash-Verfahren (VII)

- Beispiel eines Hash-Verfahrens mit Zeichenketten und modulo-Funktion.

```
Wahl HashtabellengröSSe: m=30
Datensatz →
  Zeichenkette S; s={'A',..,'Z','a',..,'z'};
Schlüssel →
  erster Buchstabe der Zeichenkette
  k(S) = INT(S[1])
  ⇒ k={65,..,90,..,97,..,122}

Datensätze:
S1="Test"   → k=84  → h(k)=24
S2="Hallo"  → k=72  → h(k)=12
S3="Mehr"   → k=77  → h(k)=17
S4="mehr"   → k=109 → h(k)=19
```




- Anordnung der Hash-Eninträge aus dem Beispiel (rechts).

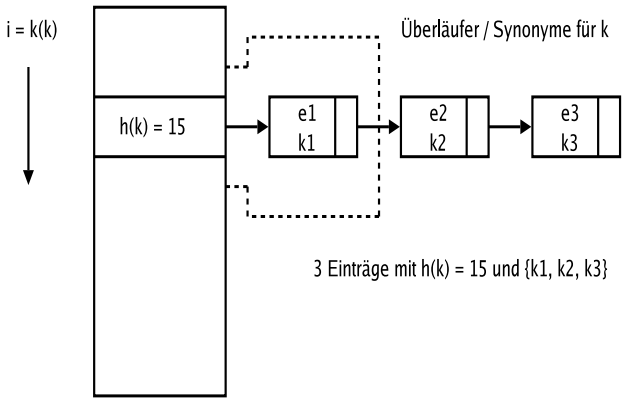


# Hash-Verfahren :: mit Verkettung der Überläufer (I)

- Soll in einer Hash-Tabelle T, die bereits den Schlüssel k enthält, ein Synonym k' von k eingefügt werden, tritt Adreßkollision auf. Der Platz h(k)=h(k') ist bereits besetzt.
- Diese sog. Überläufer müssen gesondert gespeichert werden.

## Methode I: Listen- Verkettung

- Alle Überläufer werden in einer linearen Liste in dem entsprechenden Tabelleneintrag mit dem initialen Eintrag verknüpft.
- Jedes Element der Hash-Tabelle T ist Anfangselement einer Überlaufkette.
- Verkettung der Überläufer am Beispiel für h(k)=15 mit drei Elementen $k_1, k_2, k_3$.



# Hash-Verfahren :: mit Verkettung der Überläufer (II)

➤ Hash-Operationen mit einfach verketteter Speicherung von Synonymen.

```
TYPE dataentry == Dataentry(
  key: keytype;
  data: datatype;
  next: ↑dataentry;
);
◆
FUN Suchen :
  k:keytype →
  T:hash →
  x:dataentry;
PRO Einfügen :
  x:dataentry →
  T:hash;
◆
PRO Entfernen :
  x:dataentry →
  T:hash;
◆
DEF h == λ k • → k mod M;
```



# Hash-Verfahren :: mit Verkettung der Überläufer (III)

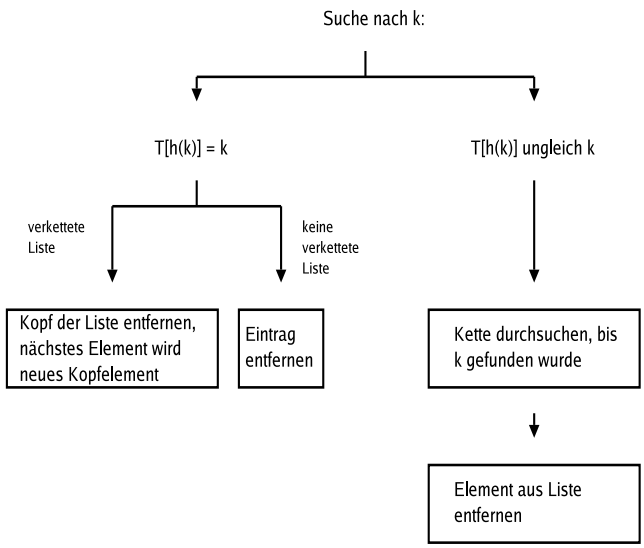
➤ Hash-Operationen (Forts.).

```
DEF Suche == λ k,H •
  T ← H.hashtab;
  x ← T[h(k)];
  WHILE x <> NONE AND x.key <> k
  DO
    x ← x.next;
  DONE;
  → x;
◆
DEF Einfügen == λ x,H •
  T ← H.hashtab;
  temp ← T[h(k)];
  IF temp = NONE THEN
    T[h(k)] ← x;
  ELSE
   WHILE temp.next ≠ NONE
   DO
     temp ← temp.next;
   DONE;
   temp.next ← x;
  END IF;
```

```
DEF Entfernen == λ k,H •
  T ← H.hashtab;
  temp ← T[h(k)];
  prev ← NONE;
  WHILE temp ≠ NONE AND temp.key ≠ k
  DO
    prev ← temp;
    temp ← temp.next;
  DONE;
  IF temp = NONE THEN
    FEHELR <not found>;
  IF prev ≠ NONE THEN
    prev.next ← temp.next;
  ELSE
    T[h(k)] ← temp.next;
```

# Hash-Verfahren :: mit Verkettung der Überläufer (IV)

- Entfernen eines Elements mit dem Schlüssel k aus einer Hashtabelle mit verketteten Überläufern.




- Die Effizienz der erfloglosen Suche läßt sich erhöhen, wenn man die Überlaufkette nach den Schlüsseln k sortiert. Im Mittel kann dann die erfolglose Suche schon in der Mitte der Überlaufkette abgebrochen werden.



# Hash-Verfahren :: mit Verkettung der Überläufer (V)

## Methode II: Offene Hash-Verfahren

- Im Unterschied zur Verkettung der Überläufer außerhalb der Tabelle versucht man bei offenen Hash-Verfahren Überläufer in der Tabelle unterzubringen..
- Soll ein Datensatz mit Schlüssel k an der Position h(k) eingetragen werden, dieser Eintrag aber schon besetzt ist, muß nach einer festen Regel ein anderer nicht belegter Platz h'(k) gefunden werden.
- Da nicht bekannt ist, ob ein anderer Tabelleneintrag schon besetzt ist, muß es eine Sondierungsfolge geben (Reihenfolge).
- Z.B. Regel:
  `wenn h(k) besetzt,`
  `untersuche h(k)+2, dann`
  `untersuche h(k)+6 usw.`
- **Systematische Sondierung**
  Sei s(j,k) eine Funktion von j und Schlüssel k so, daß

$$S\forall j = 1, ..., m \Leftarrow (h(k) - s(j, k)) \bmod m \tag{15}$$

eine Sondierungsfolge bildet, d.h. eine Permutation aller Hash-Adressen h.

# Hash-Verfahren :: mit Verkettung der Überläufer (VI)

- Ein Eintrag der Hash-Tabelle benötigt hier die zusätzliche Information, ob der Eintrag frei oder belegt ist. Entweder erweitert man die Datenstruktur um eine Markierungsvariable, oder benutzt eine weitere Tabelle F gleicher Größe wie die Hash-Tabelle T, die nur diese Information beinhaltet.
- Operationen mit offenen Hash-Tabellen.

```
FUN Suchen :
  k:keytype →
  H:hash →
  x:dataentry;
DEF Suchen == λ k,H •
  T ← H.hashtab;
  F ← H.hashflags;
  j ← 0;
  (* Suchen *)
  DO
    i ← (h(k)−s(j,k)) MOD m;
    INCR j;
  DONE
  WHILE T[i].key ≠ k AND F[i] ≠ Frei;
  IF F[i] = Belegt THEN
    → T[i];
  ELSE
    → NONE; (* nicht gefunden *)
```



# Hash-Verfahren :: mit Verkettung der Überläufer (VII)

- Operationen mit offenen Hash-Tabellen (Forts.).

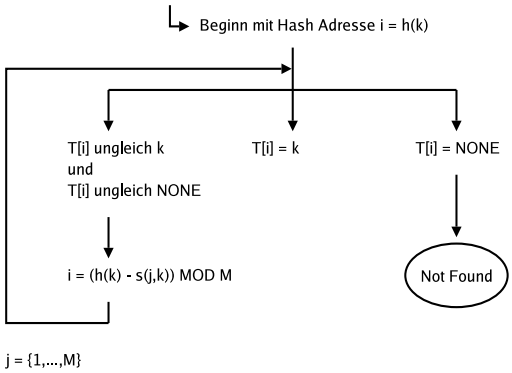
```
PRO Einfügen :
  x:dataentry →
  H:hash;

DEF Einfügen == λ x,H •
  T ← H.hashtab;
  F ← H.hashflags;
  j ← 0;
  (* Suchen *)
  DO
    i ← (h(k)−s(j,k)) MOD m;
    INCR j;
  DONE
  WHILE F[i] = Belegt;
  T[i] ← x;
  F[i] ← Belegt;
```

◆

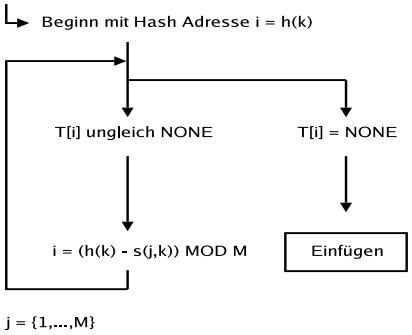
```
PRO Entfernen :
  k:keytype →
  H:hash;

DEF Entfernen == λ k,H •
  T ← H.hashtab;
  F ← H.hashflags;
  j ← 0;
  (* Suchen *)
  DO
    i ← (h(k)−s(j,k)) MOD m;
    INCR j;
  DONE
  WHILE T[i] ≠ k AND F[i] ≠ Frei;
  IF F[i] = Belegt THEN
    F[i] ← Entfernt;
  ELSE
    FEHLER <not found>;
```

# Hash-Verfahren :: mit Verkettung der Überläufer (VIII)

- Suche eines Elements mit dem Schlüssel k in offenen Hashtabellen.




- Entfernen eines Elements mit dem Schlüssel k in offenen Hashtabellen.






# Hash-Verfahren :: mit Verkettung der Überläufer (IX)

## Lineares Sondieren

- Sondierungsfunktion:

$$S(j,k) = j \tag{16}$$

- Sondierungsfolge:

$$h(k), h(k) - 1, ..., 0, ..., m - 1, .., h(k) + 1 \tag{17}$$

- Durchschnittliche Anzahl von untersuchten Einträgen in der Hash-Tabelle bei der Suche hängt vom Belegungsfaktor $\eta$ ab:

$$C_n \approx 1/2\left(1 + \frac{1}{(1-\eta)}\right) \tag{18}$$

$$C_n \approx 1/2\left(1 + \frac{1}{(1-\eta)}\right) \tag{19}$$

mit $C_n^{'}$ als Anzahl bei erfolgloser, und $C_n$ bei erfolgreicher Suche [nach D.Knuth ”The Art of Computer Programming”].

# Hash-Verfahren :: mit Verkettung der Überläufer (X)

## Lineares Sondieren

- Lineares Sondieren und mittlere Anzahl von erfolglosen und erfolgreichen Suchen.

| $\eta$ | $C_n$ | $C_n$ |
|---|---|---|
| 0.5 | 1.5 | 2.5 |
| 0.9 | 5.5 | 50.5 |
| 0.95 | 10.5 | 200.5 |
| 1.00 | $\infty$ | $\infty$ |

## Quadratisches Sondieren

- Sondierungsfunktion:

$$S(j,k) = (j/2)^2 - (-1)^2 \tag{20}$$

- Sondierungsfolge:

$$h(k), h(k)+1, h(k)-1, h(k)+4, h(k)-4, \ldots \tag{21}$$



# Hash-Verfahren :: mit Verkettung der Überläufer (XI)

## Quadratisches Sondieren

- Durchschnittliche Anzahl von untersuchten Einträgen in der Hash-Tabelle bei der Suche hängt vom Belegungsfaktor $\eta$ ab:

$$C_n' \approx \frac{1}{1-\eta} - \eta + \ln\left(\frac{1}{(1-\eta)}\right) \tag{22}$$

$$C_n \approx 1 + \ln\left(\frac{1}{(1-\eta)}\right) - \frac{\eta}{2} \tag{23}$$

mit $C_n'$ als Anzahl bei erfolgloser, und $C_n$ bei erfolgreicher Suche [nach D.Knuth "The Art of Computer Programming"].

- Quadratisches Sondieren und mittlere Anzahl von erfolglosen und erfolgreichen Suchen.

| $\eta$ | $C_n$ | $C_n$ |
|---|---|---|
| 0.5 | 1.44 | 2.19 |
| 0.9 | 2.85 | 11.40 |
| 0.95 | 3.52 | 22.05 |
| 1.00 | $\infty$ | $\infty$ |

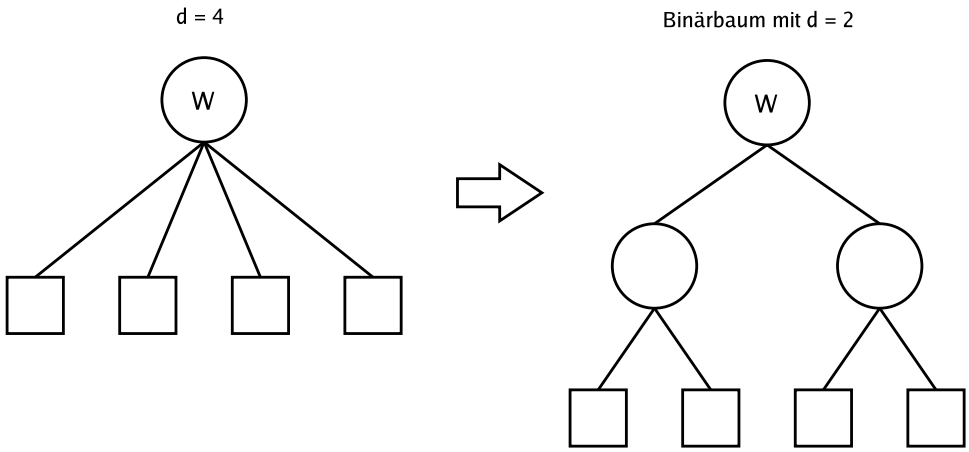
# Bäume

- Bäume gehören zu den wichtigsten Datenstrukturen (ADT):
  - Entscheidungsbäume
  - Syntaxbäume
  - Ableitungsbäume
  - Suchbäume
  - Kodebäume usw.
- Bäume sind verallgemeinerte Listenstrukturen:
  - Ein Element wird als **Knoten** bezeichnet,
  - ein Knoten {kann/hat} im Gegensatz zu Listen mehr als einen Nachfolger, eine endlich begrenzte Anzahl von sog. Söhnen (Childs).
  - Ein Knoten eins Baumes, ein Element, ist als **Wurzel** des Baumes ausgezeichnet, der einzige Knoten ohne Vorgänger.
  - Jeder andere Knoten hat einen unmittelbaren **Vorgänger**, den Vater (Parent).
  - Knoten, die nicht weiter verzweigen, werden **Blätter** genannt (haben keine Söhne).
  - Eine Folge von Knoten $\{p_0,...,p_k\}$ eines Baumes, die die Bedingung $p_{i+1} = Sohn(p_i)$ erfüllt, heißt **Pfad** mit der Länge k und $0 \leq i < k$, der $p_0$ mit $p_k$ verbindet.
  - Die **Ordnung** oder der Rang eines Baumes gibt die (maximale) Zahl von Söhnen eines Knotens an.
  - **Binärbäume** sind Bäume der Ordnung=2 (es gibt nur linken und rechten Sohn).



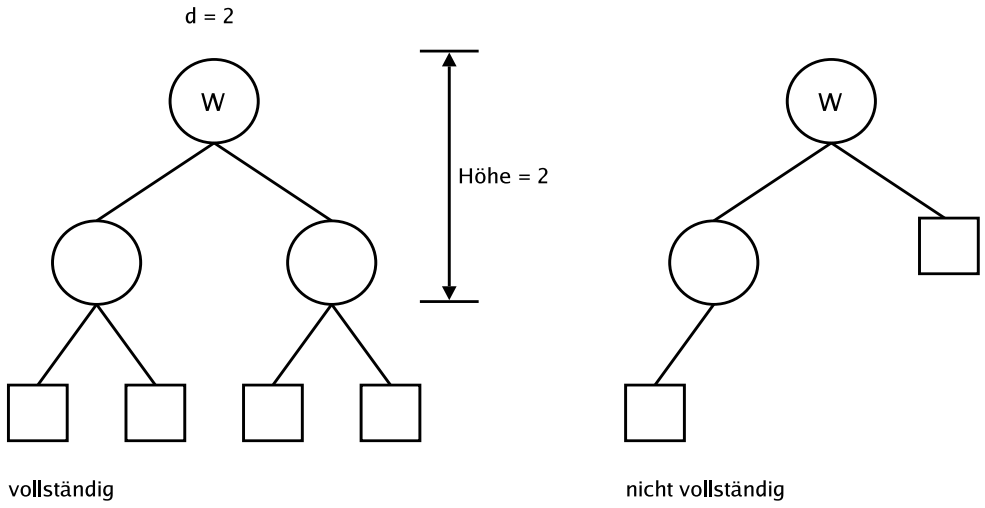
# Bäume :: Aufbau von Bäumen (I)

- Bäume können rekursiv aufgebaut werden, da jeder Knoten als Baum der Ordnung d aufgefasst werden kann, symbolisch:
  ♡ Sind $\{t_1,...,t_d\}$ beliebige Bäume der Ordnung d, so erhält man einen neuen Baum der Ordnung d, indem man die Wurzelknoten von $\{t_1,...,t_d\}$ zu Söhnen eines neu geschaffenen Wurzelknotens w macht:
  ♡ Zerlegung eines Baumes der Odnung d in einem Baum der Ordnung d'<d: Z.B. ein Baum der Ordnung 4 kann man in einen Binärbaum zerlegen (aufspalten).
- Transformation von Bäumen:



# Bäume :: Aufbau von Bäumen (II)

- Bäume der Odrnung d nennt man **vollständig**, wenn jeder Knoten d Söhne besitzt, ausgenommen die leeren Teilbäume ohne weitere Verzweigung, die Blätter - alle Blätter befinden sich auf gleicher Ebene.
- Vollständige und unvollständige Bäume.






# Bäume :: Aufbau von Bäumen (III)

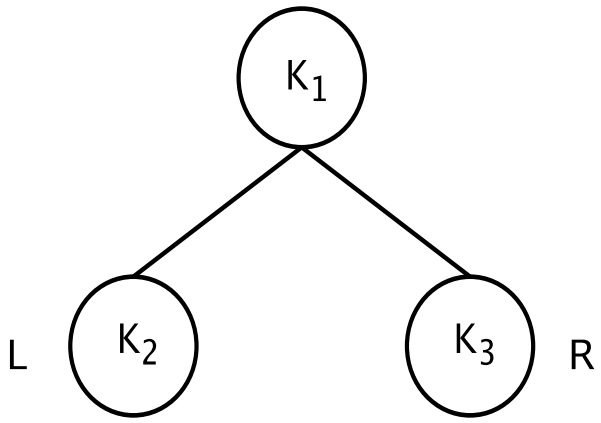
- Bäume der Ordnung d>2 nennt man Vielwegbäume. Sog. B-Bäume, eine wichtige Unterklasse von Bäumen, fordern, daß ein Knoten eine Anzhal n von Söhnen zwischen einer Unter- und Obergrenze besitzt:

$$a \leq n \leq b \tag{24}$$

- Jeder Knoten muß durch einen eindeutigen **Schlüssel** bezeichnet sein, zweckmäßigerweise eine ganze Zahl. Der Schlüssel bestimmt den Aufbau und die Struktur des Baumes!
- **Operationen auf Bäumen**
  1. Suchen mittels Schlüssel
  2. Einfügen nach Schlüsselposition
  3. Entfernen eines Knotens (oder eines Blattes)
  4. Aufspalten von Bäumen in Teilbäume
  5. Zusammenfügen von Teilbäumen
  6. Restrukturierung und Balanzierung eines nicht vollständigen Baumes.

# Bäume :: Natürliche Bäume - Binärbäume (I)

- Jeder Knoten hat maximal (minimal?) zwei Söhne, d.h. Verzweigungen. Die Ordnung ist d=2.
- Aufbau eines binären Baums.




L,R: linker, rechter Sohn
Ki: Schlüssel



# Bäume :: Natürliche Bäume - Binärbäume (II)

- Es gibt verschiedene Strukturierungen eines binären Baumes:

  **Suchbäume**
  jeder Knoten enthält den Schlüssel und einen Datensatz → Blätter enthalten keine Datensätze, bzw. ein Blatt ist der letzte Knoten (mit Datensatz) in einem Pfad.

  **Blattsuchbäume**
  nur die Blätter enthalten die eigentlichen Schlüssel, die mit den Datensätzen verknüpft sind - die Knoten stellen nur einen Suchweg dar. Beispiel: Binary-Decision-Diagrams (BDD).

- Knoten (und Blätter) werden nach einer bestimmten Regel angeordnet in Abhängigkeit der Schlüssel:

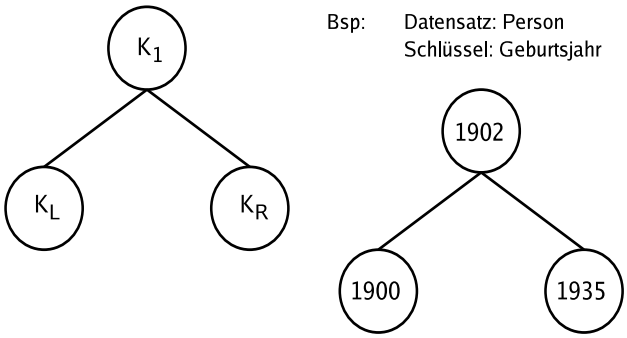
$$\text{Linker Sohn L}: \; k_L < k \tag{25}$$

$$\text{Rechter Sohn R}: \; k_R > k \tag{26}$$

$$\text{Elternknoten} \; : \; k \tag{27}$$

# Bäume :: Natürliche Bäume - Binärbäume (III)

- Aufbau und Beispiel eines Knotens.




- ADT binärer Baum. Jeder Knoten ist hier mit einem Datensatz verknüpft. Ein Blatt ist ein Knoten ohne Nachfolger (Nullzeiger).

```
TYPE knoten == Knoten(
  leftson: ↑knoten;
  rightson: ↑knoten;
  key: keytype;
  data: datatype;
);
```



# Bäume :: Natürliche Bäume - Binärbäume (IV)

## Suchen eines Datensatzes

- ADT binärer Baum, Suchoperation.

```
FUN Suchen :
  p:↑knoten →  (* Wurzelknoten *)
  k:keytype →
  x:↑knoten;

DEF REC Suchen == λ p,K •
  IF p = NONE THEN →NONE;
  ELSE IF k < p.key THEN
    Suchen (p.left,k);
  ELSE IF k > p.key THEN
    Suchen (p.right,k);
  ELSE
    →P;
  END IF;
```

# Bäume :: Natürliche Bäume - Binärbäume (V)

## Einfügen eines neuen Datensatzes

- Um einen neuen Schlüssel in einen Suchbaum einzufügen, wird zuerst nach diesem Schlüssel gesucht.
- Falls dieser nicht schon bereits vorhanden ist, endet die Suche erfolglos bei einem Blatt (oder dem letzen Knoten).
- Der neue Datensatz wird dann an den letzen Knoten der Suche angehängt.
- Einfügen eines neuen Elements (Knoten) x in einem Binärbaum. Der aktuelle Knoten ist der Zeiger p.

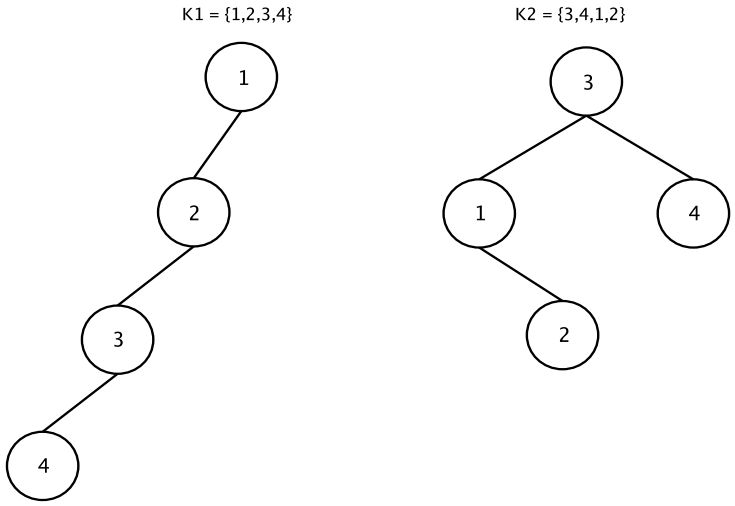
```
PRO Einfügen :
  p:↑knoten →
  x:↑knoten;
```

```
DEF REC Einfügen == λp,x •
  IF p = NONE THEN
    {
      • neue Datenstruktur erzeugen
      LET p == Knoten() IN
      p.leftson ← NONE;
      p.rightson ← NONE;
      p.key ← x.key;
      p.data ← x.data;
    }
    ∨ • alternativ mit Referenz
    {
      p ← x;
    }
  ELSE
    IF x.key < p .key THEN
      Einfügen p.leftson x;
    ELSE IF x.key > p.key THEN
      Einfügen p.rightson x;
    ELSE
      FEHLER Schlüssel vorhanden;
    END IF;
  END IF;
```



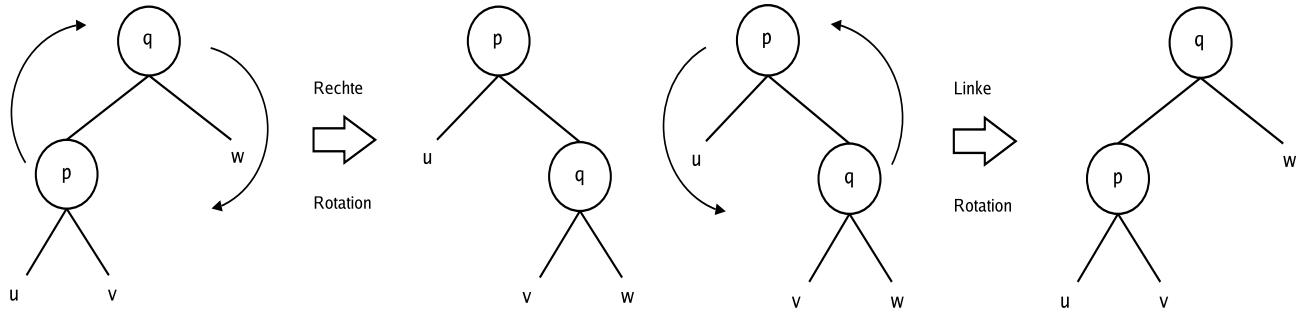
# Bäume :: Natürliche Bäume - Binärbäume (VI)

- Die Struktur eines Baumes hängt von der Reihenfolge der Schlüssel beim Einfügen ab. Beispiel:
  $K_1$={1,2,3,4}
  $K_2$={3,4,1,2}
- Abhängigkeit der Baumstruktur von der Einfügereihenfolge.



# Bäume :: Natürliche Bäume - Binärbäume (VII)

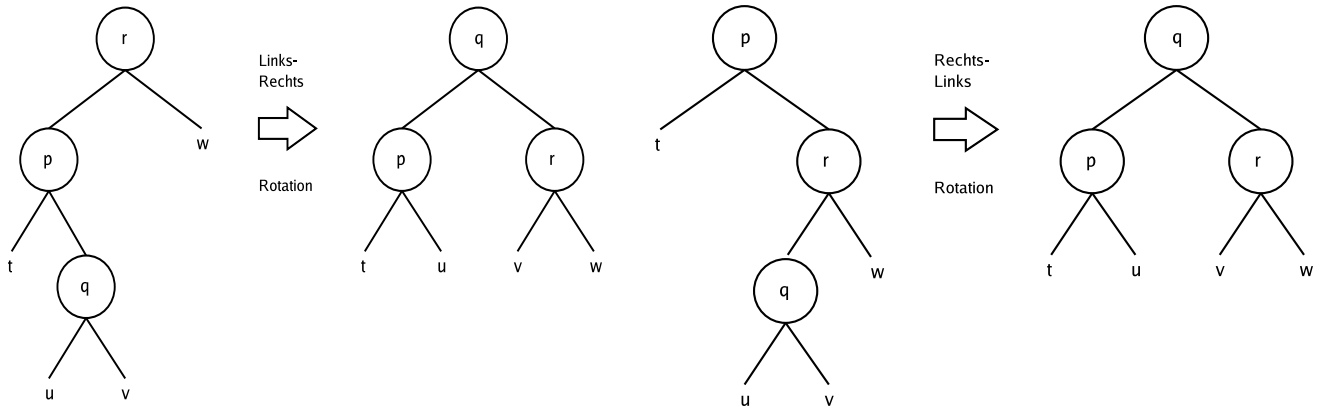
- Für eine möglichst schnelle Suche, optimal $\Theta(log_2 N)$, sollte der Baum balanziert sein, d.h. jeder Knoten hat zwei Nachfolger (außer Endknoten).
- Die Höhe eines Baumes ist dann minimal $H=log_2 N$:
- Für die **Balanzierung** eines Baumes benötigt man zwei Rotationsoperationen:
  1. Rechte Rotation im Uhrzeigersinn von zwei benachbarten Knoten,
  2. und linke Rotation gegen den Uhrzeigersinn.
- Rechte Rotation zweier Knoten q und p. Die weiterführenden Verzweigungen werden entsprechend der Ordnungsrelation umgeordnet (linkes Bild).
  Linke Rotation zweier Knoten q und p. Die weiterführenden Verzweigungen werden entsprechend der Ordnungsrelation umgeordnet (rechtes Bild).






# Bäume :: Natürliche Bäume - Binärbäume (VIII)

- Kombination aus Links- und Rechtsrotation führt zur Balanzierung eines linkslastigen asymmetrischen Baumes (linkes Bild).
- Kombination aus Rechts- und Linksrotation führt zur Balanzierung eines rechtslastigen asymmetrischen Baumes (rechtes Bild).



# Bäume :: Natürliche Bäume - Binärbäume (IX)

## Entfernen eines Datensatzes

1. Ist k(x) der Schlüssel des Knotens x, und hat x keine Nachfolger, kann x einfach entfernt werden.
2. Hat x einen Sohn, wird dieser an den Vater-Knoten von x anstelle von x angehängt.
3. Wenn x zwei Söhne hat, muß der rechte Sohn-Knoten den Knoten x ersetzen, und der linke wird der Sohn vom neuen Knoten. Wenn der alte rechte Sohn von x wieder zwei Söhne hat, muß dieser Vorgang solange wiederholt werden, bis Fall 1-2 eintritt.



# Systematische Reduktion logischer Funktionen :: BDD (I)

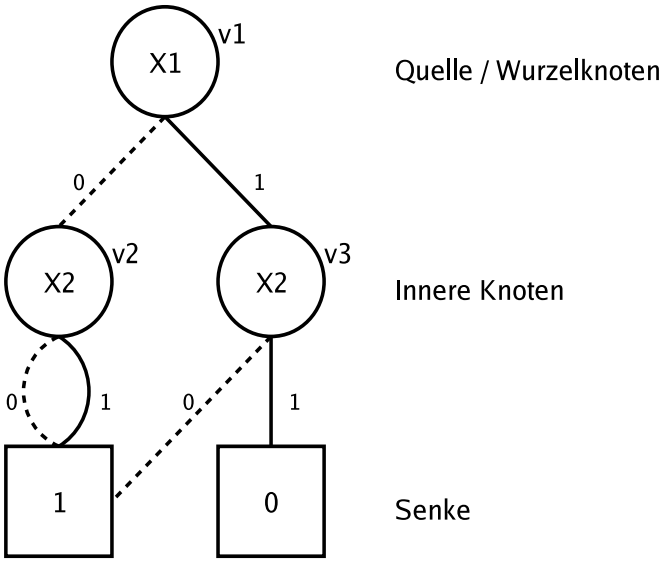
## Binary-Decision-Diagram (BDD)-Methode

- Ein Binary-Decision-Diagram (BDD) BDDs ist ein azyklischer und gerichteter Graph, der einen Algorithmus zur Berechnung einer booleschen Funktion (BF) beschreibt.
- Jede boolesche Funktion kann als BDD dargestellt werden.
- Berechnung der dargestellten Funktion für einen gegebenen Eingangsvektor $\{x_1,...,x_n\}$ beginnt an der Quelle (Wurzelknoten).
- Ein BDD besteht aus einem Startknoten (Quelle) und inneren Knoten mit dem Ausgangsgrad 2.
- Die inneren Knoten sind Variablen der BF zugeordnet.
- Die beiden ausgehenden Kanten der inneren Knoten entsprechen der booleschen Wertemenge {0,1}.
- Am Ende eines Pfades im BDD befindet sich eine Senke mit dem Ausgangsgrad 0.
- Ein Pfad (Quelle → Senke) beschreibt die Evaluierung eines Funktionswertes der BF.
- Die Größe eines BDD's ist maximal $O(2^n/n)$ bei einer BF mit n Variablen.
- Nachteilige Eigenschaft von BDDs (wenn n groß ist):
  Die Tatsächliche Größe und Struktur eines BDD's hängt von der Variablenreihenfolge bei der Erzeugung ab! (Ausnahme: symmetrische BF).

# Systematische Reduktion logischer Funktionen :: BDD (II)

## Binary-Decision-Diagram (BDD)-Methode

Beispiel: $f(x_1,x_2)=\neg x_1+\neg x_2$






# Systematische Reduktion logischer Funktionen :: BDD (III)

## Shanon-Zerlegung

Erzeugung eines BDD's aus BF schrittweise durch Evaluierung der einzelnen Variablen $\{x_1,..,x_2\}$ mit den Werten {0,1}:

$$f = \neg x_1 f_{|x1=0} \vee x_1 f_{|x1=1} \tag{28}$$

Jeder innere Knoten stellt eine neue (Sub-) BF dar:

$$f_{v1} = \neg x_1 + \neg x_2 \tag{29}$$

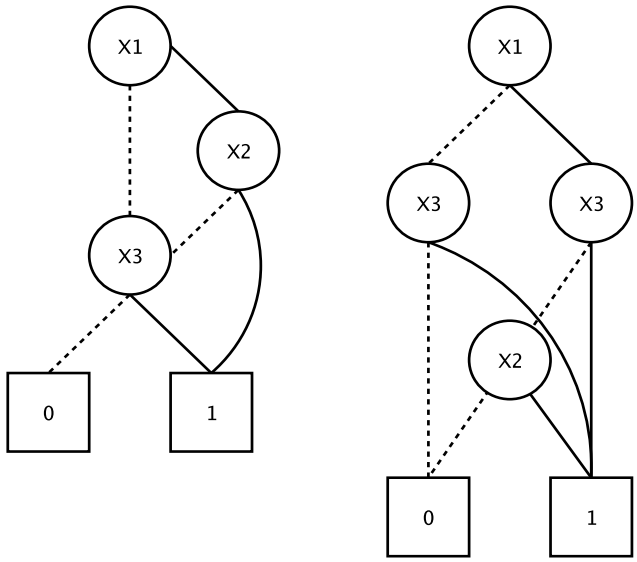
$$f_{v2} = 1 \tag{30}$$

$$f_{v3} = \neg x_2 \tag{31}$$

# Systematische Reduktion logischer Funktionen :: BDD (IV)

## Shanon-Zerlegung

Die Tatsächliche Größe und Struktur eines BDD's hängt von der Variablenreihenfolge bei der Erzeugung ab! (Ausnahme: symmetrische BF).
Beispiel zweier BDDs erzeugt aus $f(x_1,x_2,x_3)=x_1 \bullet x_2+x_3$




Reihenfolge {x1, x2, x3} {x1, x3, x2}



# Systematische Reduktion logischer Funktionen :: BDD (V)

## Reduktion von BDDs

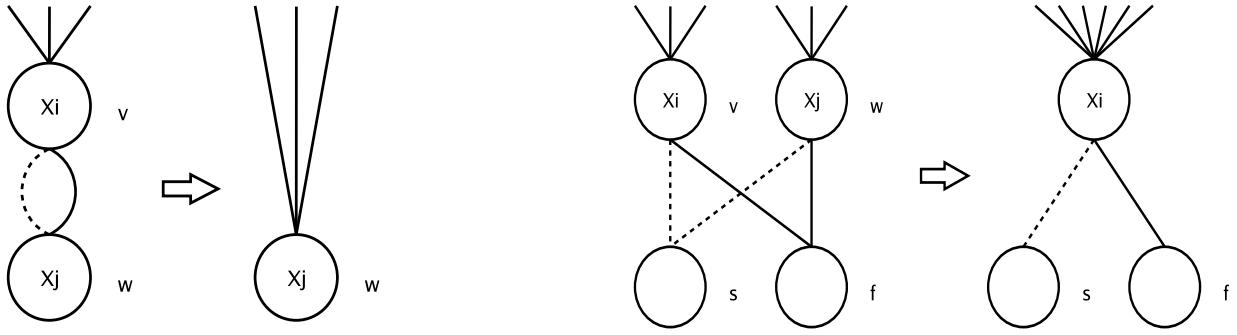
Die Reduktion eines (O)BDD's hat das Ziel, die Anzahl der Pfade und Variablen zu minimieren, was in einer reduzierten und minimierten BF resultiert.

## Deletion-Rule

Die beiden Kanten eines Knotens $v(x_i)$ verzweigen beide auf den gleichen Nachfolgerknoten $w(x_j)$. Der Knoten v kann entfernt werden, und alle eingehenden Kanten werden auf w umgeleitet.

## Merging-Rule

Zwei Knoten $v(x_i)$ und $w(x_i)$ der gleichen Variable besitzen gleiche 0- und 1-Nachfolger. Die Knoten v und w können zu einem neuen Knoten v' zusammengefaßt werden.



# Systematische Reduktion logischer Funktionen :: BDD (VI)

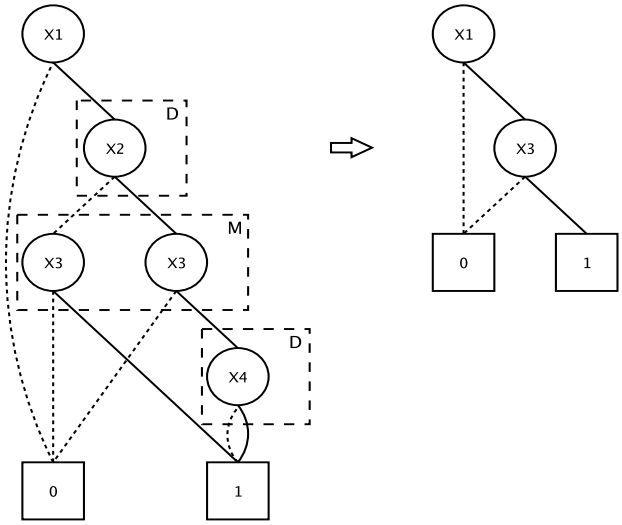
## Reduktion von BDDs (Forts.)

Das folgende Beispiel zeigt eine Reduktion der BF

$$f(x_1, x_2, x_3, x_4) = x_1 \bullet x_2 \bullet x_3 \bullet x_4 + x_1 \bullet \neg x_2 \bullet x_3 + x_1 \bullet x_2 \bullet x_3 \bullet \neg x_4 \quad (32)$$

zu

$$f(x_1, x_2, x_3, x_4) = x_1 \bullet x_3 \quad (33)$$






# Sortierverfahren

- Liste von Datensätzen bzw. Elementen $\vec{a} = \{a_1, a_2, ..., a_N\}$ sollen wie folgt nach ihrem Schlüssel geordnet sein:

$$a_1.\text{key} \leq a_2.\text{key} \leq ... \leq a_N.\text{key} \quad (34)$$

$$a[1].\text{key} \leq a[2].\text{key} \leq ... \leq a[N].\text{key} \text{ mit Array} \quad (35)$$

- Wenn die Liste ungeordnet ist, kann mittels eines Sortierverfahrens eine Ordnung durch Verschieben oder Tausch von Elementen in der Liste bzw. einem Array erreicht werden.

- **Wie bei allen Listen basierten Algorithmen ist die Laufzeit des Sortierverfahrens wichtigste Eigenschaft, zusammen mit dem Speicherbedarf des Verfahrens.**

  **Verschiedene Sortierverfahren:**

  1. Sortieren durch Auswahl
  2. Sortieren durch Einfügen
  3. Shell-Sort
  4. Bubble-Sort
  5. Quick-Sort

# Sortierverfahren :: Sortieren durch Auswahl (I)

- Verfahren:
  - Position $j_1$ bestimmen, an der das Element mit kleinsten Schlüssel aus $\vec{a} = \{a_1, a_2, ..., a_N\}$,
  - Vertausche $a_1 mit a_{j1}$
  - Wiederhole Suche und Vertauschung für $a_2$ mit $a_{j2}$ usw.
  - Der Reihe nach wird das i-kleinste Element $a_{ji}$ mit i={1,...,N-1} bestimmt.

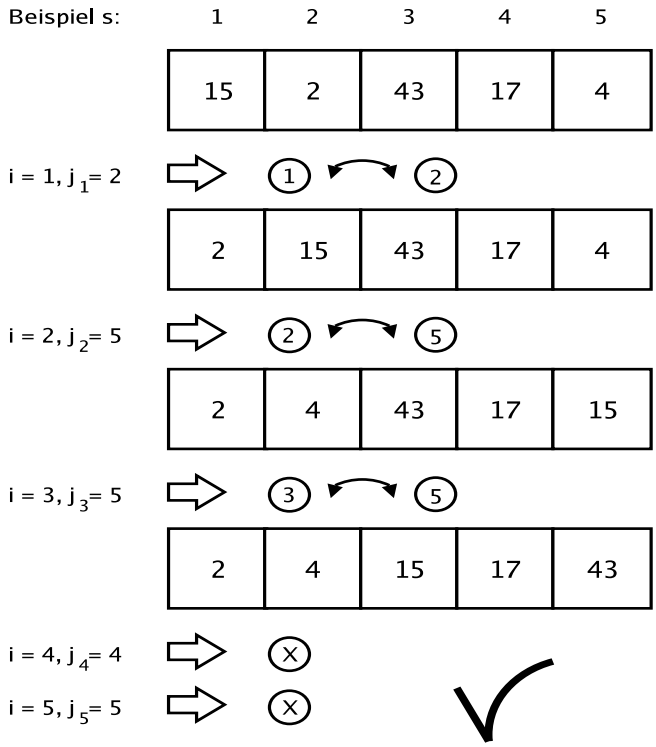
- Algorithmus Auswahlsortierung.

```
PRO AuswahlSort :
  a:dataarray;
◆
DEF AuswahlSort == λ a •
  FOR i = 1 TO n−1
  DO
    min ← i;
    FOR j = i+1 TO n
    DO
      IF a[j].key <
          a[min].key THEN
        min ← j;
    DONE;
    (* Vertauschung *)
    t ← a[min];
    a[min] ← a[i];
    a[i] ← t;
  DONE;
```



# Sortierverfahren :: Sortieren durch Auswahl (II)

- Beispiel Auswahlsorierung.



# Sortierverfahren :: Sortieren durch Auswahl (III)

## Analyse des Laufzeitverhaltens

- Anzahl der Suchoperationen für gegebenes i:

$$N_i = N - i \tag{36}$$

- Gesamtzahl der Suchoperationen:

$$S_{min} = S_{max} = N_{tot} = \sum_{i=1}^{N-1} N - i = \sum_{i=1}^{N-1} i = \frac{N(N-1)}{2} \Rightarrow \Theta(N^2) \tag{37}$$

- Anzahl der Verschiebeoperationen:

$$M_{min} = M_{max} = 3(N-1) \Rightarrow \Theta(N) \tag{38}$$

- Die Laufzeit bei der Suche überwiegt die Laufzeit der Verschiebungsoperationen.
- Bei der Laufzeitanalyse wird immer der minimale und maximale Aufwand von Operationen (in normierten Einheiten) angegeben.
- Die Größenordnung der Laufzeit $\Theta$ ist immer der ungünstigste Fall einer Operation.



# Sortierverfahren :: Sortieren durch Einfügen (I)

- Die N zu sortierenden Elemente werden nacheinander betrachtet und in die jeweils, bereits sortierte, anfangs leere Teilfolge an der richtigen Stelle eingefügt.
- Annahme: Teilfolge $\{a_1, ..., a_{i-1}\}$ ist bereits sortiert mit

$$a_1.key \leq a_2.key \leq ... \leq a_N.key \tag{39}$$

- Verfahren:
  - Einfügen eines neuen Elements $a_i$ durch absteigenden Vergleich von $a_j$ und $a_i$ mit j=(i-1),...,1
  - Element $a_j$ wird um eine Position nach rechts verschoben, wenn $a_j > a_i$.
  - Ist $a_j \leq a_i$, dann wird $a_i$ an der Stelle (j+1) eingefügt.

- Algorithmus Einfügesortierung.

```
PRO EinfügeSort :
  a:<dataarray>;
◆
DEF EinfügeSort == λ a •
  FOR i = 2 TO n
  DO
    j ← i;
    t ← a[i];
    k ← t.key;
    WHILE j > 0 AND
          a[j−1].key > k
    DO
      (* Verschieben *)
      a[j] ← a[j−1];
      DECR j;
    DONE;
    a[j] ← t;
  DONE;
```

# Sortierverfahren :: Sortieren durch Einfügen (II)

- Beispiel Einfügesortierung.

```
a={15,2,43,17,4}
I. a1=15 → bereits sortiert a′={15}
II. a2=2 → j=1,i=2
    a1 > a2 → a1 nach rechts verschieben ℜ
    a′={2,15}
III. a3=43 → j=2, i=3
     a2 < a3 → a3 hinten anhängen ℘
     a′={2,15,43}
IV. a4=17 → j=3, i=4
    a3 > a4 → a3 nach rechts verschieben ℜ
    a′={2,15,17,43}
usw.
```

## Analyse des Laufzeitverhaltens

- Gesamtzahl der Suchoperationen:

$$S_{min} = N - 1 \tag{40}$$

$$S_{max} = \sum_{i=2}^{N-1} i \Rightarrow \Theta(N^2) \tag{41}$$



# Sortierverfahren :: Sortieren durch Einfügen (III)

## Analyse des Laufzeitverhaltens (Forts.)

- Anzahl der Verschiebeoperationen:

$$M_{min} = 2(N - 1) \tag{42}$$

$$M_{max} = \sum_{i=2}^{N-1} i + 1 \Rightarrow \Theta(N^2) \tag{43}$$

- Die Laufzeit bei der Suche ist von gleicher Größenordnung wie die Laufzeit der Verschiebungsoperationen.
  ➠ ➠ ➠ nicht effizienter als Auswahlsortierung.

# Sortierverfahren :: Shell-Sort (I)

- Beim Sorieren durch Einfügen wird ein Element immer nur um eine Stelle nach rechts verschoben.
- Ziel: Verschiebung in größeren Schritten durchführen, so daß ein Element schenller an seine endgültige Position gebracht werden kann.
- Es wird eine abnehmende Folge von Schrittweiten gewählt:

$$S = \{h_t, h_{t-1}, ..., h_1\} \quad (44)$$

- Sortieren mit abnehmenden Inkrementen ist ein Verfahren, was von D. L. Shell (ca. 1959) eingeführt wurde.
- Die Wahl der Schrittweiten entscheidet über die Performance des Sortiervefahrens (Laufzeit). Beispiel für eine Schrittweitenfolge:

$$\{5, 3, 1\} \quad (45)$$

- Algorithmus Shell-Sortverfahren.

```
DEF ShellSort == λ a •
  FOR EACH h ∈ S
  DO
   j ← i;
   t ← a[i];
   k ← t.key;
   WHILE J > h AND
         a[j−h].key > k
   DO
     (* Verschieben *)
     a[j] ← a[j−h];
     j ← j−h;
   DONE;
   a[j] ← t;
  DONE
```



# Sortierverfahren :: Shell-Sort (II)

## Analyse des Laufzeitverhaltens

- Laufzeit: $\Theta(N\, log_2^2\, N)$ bei einer h-Folge mit den möglichen Inkrementen $2^p 3^q$ mit p,q als Laufindizes {p,q=0,1,2,...}.
- Leicht verbessertes Laufzeitverhalten im Vergleich mit Sortieren durch Einfügen und Auswahl.

# Sortierverfahren :: Bubble-Sort (I)

- Das Bubble-Sort-Verfahren ist ebenfalls ein einfaches Sortierverfahren, welches als Mischung aus Einfüge- und Auswahlsortiervefahren betrachtet werden kann.
- Eine Liste $\{a_1, a_2, ..., a_N\}$ wird iterativ durchlaufen. Wenn $a_i.key > a_{i+1}.key$ erfüllt ist, werden die (benachbarten) Elemente $a_i$und $a_{i+1}$ vertauscht.
- Der Listendurchlauf wird solange wiederholt, bis bei einem vollständigen Durchlauf keine Vertauschungen mehr auftreten.
- Algorithmus Bubble-Sortverfahren.

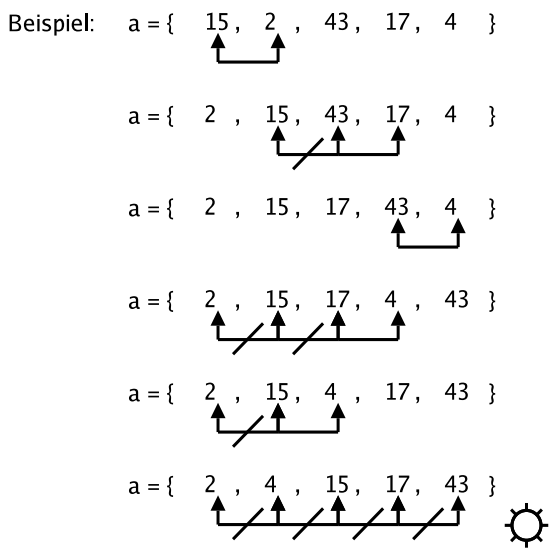
```
DEF BubbleSort == λ a •
  swap ← false;
  DO
    FOR i = 1 to (N−1) DO
      IF a[i].key >
         a[i+1].key THEN
        (* Vertauschung *)
        t ← a[i];
        a[i] ← a[i+1];
        a[i+1] ← t;
        swap ← true;
      END IF;
    DONE;
  DONE
  WHILE swap;
```



# Sortierverfahren :: Bubble-Sort (II)

## Analyse des Laufzeitverhaltens

- Laufzeit: $\Theta(N^2)$
- Bubble-Sort ist nur effizient bei bereits vorsortierten Listen, bei denen also die Inversionszahl der Schlüssel (Anzahl der Vertauschungen) klein ist.
- Beispiel Bubble-Sort



# Sortierverfahren :: Quick-Sort (I)

- Schnelles Sortierverfahren mit einer kürzeren Laufzeit als $\Theta(N^2)$ wird angestrebt.
- Verfahren: Divide-and-Conquer (Teilen und "Erobern"), eine häufig in der Algorithmik verwendete Methodik.
- Laufzeit ist im schlechtesten Fall auch wieder von quadratischer Größenordnung, aber im Mittel nur noch von linear-logarithmischer Ordnung

$$\Theta(N \log_2 N) \tag{46}$$

- Bei dem Divide-and-Conquer-Verfahren (DaC) handelt sich um ein rekursives Vefharen, welches die DaC-Strategie rekursiv auf Teilfolgen anwendet.
- Quick-Sort ist ein In-Situ-Verfahren, d.h. die Sortierung findet innerhalb des Datenfeldes statt. Der Algorithmus wird dadurch komplexer, da DaC-Verfahren auf Teilung basieren, und Datenfelder nur über ihre Indexgrenzen "geteilt" werden können.

### Sortieren durch rekursives Teilen

- Methode: Um eine Folge $F = \{k_1, k_2, ..., k_N\}$ von N Schlüssen und zugehörigen Datensätzen aufsteigend zu sortieren wählt man ein beliebiges Element $k_i \epsilon F$ aus und benutzt dieses als Teilungspunkt, das sog. **Pivot-Element**.



# Sortierverfahren :: Quick-Sort (II)

- Anschließende Aufteilung der Folge F in zwei Teilfogen $F_1$ und $F_2$ ohne $k_i$.
- $F_1$ besteht nur aus Elementen von F, für die gilt:

$$k_j \leq k_i \text{ mit } j < i \tag{47}$$

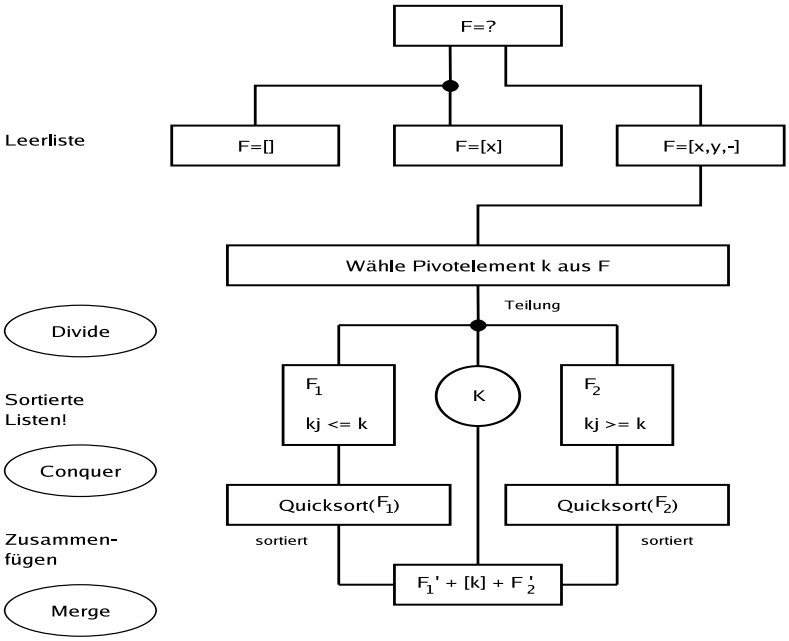
- $F_2$ besteht nur aus Elementen von F, für die gilt:

$$k_j \geq k_i \text{ mit } j > i \tag{48}$$

- $F_1$ ist eine Folge aus i-1 Elementen, und $F_2$ besitzt N-i Elemente.

# Sortierverfahren :: Quick-Sort (II)

- Flußdiagramm des Quick-Sort Algorithmus.

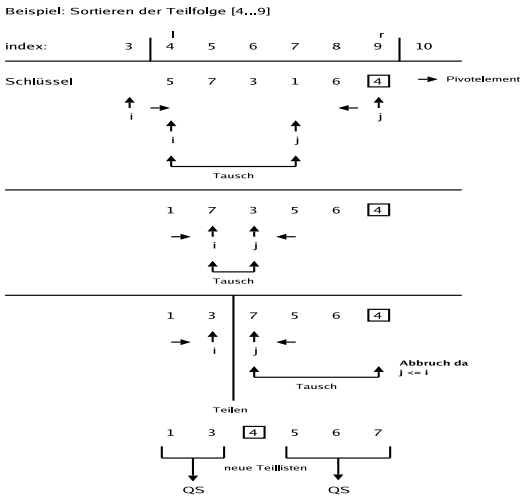





# Sortierverfahren :: Quick-Sort (III)

- Algorithmus Quick-Sort mit Divide-and-Conquer-Methode.Das Pivot-Element ist jeweils das ganz rechte Element r eines Teilarrays mit den Indexgrenzen l und r.

```
PRO QuickSort :
  f: ARRAY →
  l: INT →
  r: INT;
DEF REC QuickSort == λf,l,r •
  ◆Abbruchkriterium der Rekursion
  ◆ abfragen! *)
  IF r > l THEN
    (*
    ** Divide
    *)
    i ← l−1;
    j ← r;
    ◆ Pivot−Element ≡ r
    k ← f[r].key;
    do_search ← true;
    WHILE do_search=true
    DO
      DO INCR i; WHILE f[i].key < k;
      DO INCR j; WHILE f[j].key > k;
      IF i ≥ j THEN
        do_search ← false;
      ELSE
        ◆ Vertauschung
        t ← f[i];
        f[i] ← f[j];
        f[j] ← t;
      END IF;
    DONE;
    ◆ Letzte Vertauschung/
    ◆ Pivot−Element plazieren
    t ← f[i];
    f[i] ← f[r];
    f[r] ← t;

    ◆ Conquer

    QuickSort (f,l,i−1);
    QuickSort (f,i+1,r);
  END IF;
```

# Sortierverfahren :: Quick-Sort (IV)

- Verfahren:
  - Pivot-Element: Schlüssel des Elements k=f[r].key am rechten Ende der zu sortierenden Teilfolge.
  - Aufteilung des Bereichs f[l] ... f[r]
  - Element $k_j$ mit j∈{r,...,l} und $k_j \leq k$ finden
  - Elemente f[i] und f[j] tauschen, so daß sich beide Elementen in den richtigen Teilfolgen befinden.
  - Solange wiederholen bis die gesamte Folge f[l]...f[r] untersucht wurde, und die beiden Teilfolgen sortiert sind.
- Beispiel Quick-Sort.