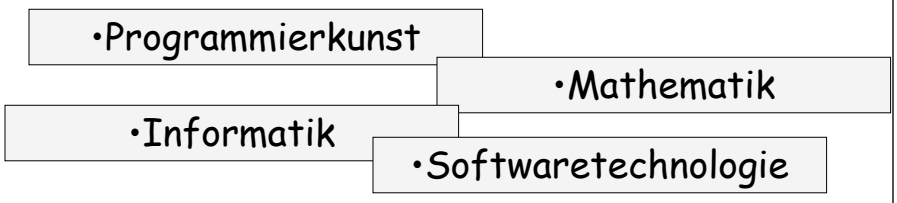
# Abstrakte Datentypen

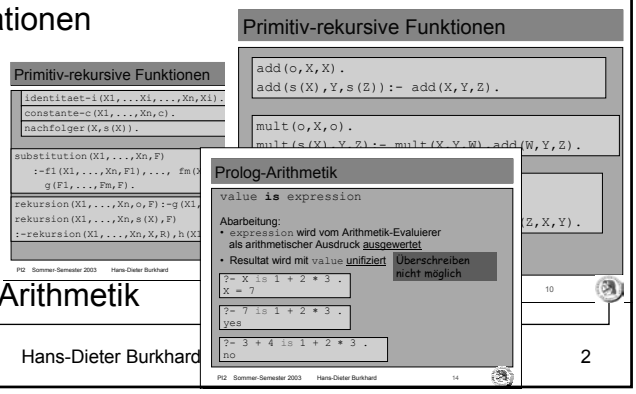
- Implementation
- Prototypen
- Spezifikation
- Formale Beschreibungen
- Abstrakte Modelle






# ABSTRAKTE DATENTYPEN

- Beobachtung: Unterschiedliche Realisierungen
  (Darstellung, Algorithmen, Fehlerbehandlung, ...)
  für gleiche Aufgaben
- Beispiel: Prolog-Arithmetik für natürliche Zahlen
  - Mittels Deklarationen
  - Mittels Prolog-Arithmetik



# Mögliche Implementationen

```
?- nachfolger(a, b).
?- nachfolger(a, Nachfolger).
?- nachfolger(Vorgänger, b).
?- nachfolger(Vorgänger, Nachfolger).
```

```
nachfolger(X,Y):- integer(X), !, Y is X+1.
nachfolger(X,Y):- integer(Y), !, X is Y-1 .
nachfolger(0,1).
nachfolger(X,Y):- nachfolger(U,V), X is U+1, Y is V+1.
```

Fehlermöglichkeiten z.B.:
- Weder Zahl noch Variable
- Negative Zahl



# Mögliche Implementationen

Argumente: Natürliche Zahl, Variable, sonstiges

```
natural(X) :- integer(X), X >= 0 .

nachfolger(X,Y):- natural(X), natural(Y), !, Y is X+1 .
nachfolger(X,Y):- natural(X), var(Y), !, Y is X+1 .
nachfolger(X,Y):- var(X), natural(Y), !, X is Y-1 .
nachfolger(0,1) :- var(X), var(Y) .
nachfolger(X,Y):- var(X), var(Y), nachfolger(U,V), X is U+1, Y is V+1.
nachfolger(X,Y):- write('nicht im Bereich der natürlichen Zahlen') .
```

(es gibt noch weitere Ausnahmen ....)

- „Exceptions": Programmiersprache/-umgebung stellt Mittel für Ausnahmebehandlung bereit

# Weitere Implementationen

*Konvertiere Dezimalzahl n in s(...s(o)...)*

```
nachfolger(X,s(X)).
```

*Konvertiere s(...s(o)...) in Dezimalzahl n*



# Weitere Implementationen

Implementationen in

•JAVA, LISP, C++, PASCAL, …

•

**Abstrakte Datentypen:**

- Daten-Formate und Prozeduren
  in allgemeiner (mathematischer) Form
- Lösung von konkreten Details (Datenkapselung)

# Abstrakte Datentypen

- Mathematische Modelle:
  - Strukturen (Definitionen)
  - Algorithmen
  - Komplexitätsaussagen

•Listen
  •Sortierverfahren
•Relationen
  •Prädikatenlogik

- Datenkapselung:
  - Interface
  - Laufzeiteigenschaften

•Datenbank
•Prolog-Modul
•Java-Klasse

- Implementation in ... von ... Version ...



# Abstrakte Datentypen

- Verallgemeinerung primitiver Datentypen
  - Boolean, integer, char, ...
- Strukturen mit Operationen
  - Mengen
  - Relationen
  - Liste (endliche Folge)
  - Keller
  - Warteschlangen
  - Graph
  - Baum
  - ...

Mathematisches Modell
Algorithmen

**Abstrakter Datentyp**
→ **Funktionalität**

Datenstruktur
→ Programm

# Liste

**Definitionen:** Länge *n*: Anzahl der Elemente

- Liste als Funktion:
  $L: \{1,...,n\} \rightarrow M$ $M$ =*Menge der* (möglichen) *Elemente*
- Liste als Aufzählung: [*L*(1), *L*(2), ..., *L*(*n*)]

*Geordnete Listen, z.B.:*
*- Zahlen in aufsteigender Reihenfolge*
*- Wörter in lexikographische Anordnung*

Geordnete Liste bzgl. reflex. Ordnungsrelation $R \subseteq$ M x M

Liste *L* ist *geordnet bzgl. R* ,

falls gilt: $\forall i, j \in \{1,...,n\}: i < j \rightarrow R(L(i),L(j))$



# Liste

**Operationen:**

- Zugriff auf Elemente :
  - Suchen, Einfügen, Löschen von Elementen
- Bearbeitung von Listen :
  - Verketten
  - Reorganisation, Sortieren, ...

*Sortierverfahren erzeugen eine Reihenfolge der Listenelemente entsprechend einer vorgegebenen Ordnungsrelation R. Unterschiedliche Sortierverfahren haben unterschiedliche Komplexität.*

- *Quick-Sort*
- *Selection-Sort*
- *Merge-Sort*
- *Heap-Sort*
- *Bubble-Sort*
- *...*

# Listen

**Implementation als Datenstruktur:**

Organisation bzgl. Speicherorganisation

- Felder
- Referenzen
  - verkettete Liste: Referenz auf Nachfolger-Element
  - Doppelt verkettete Liste
- Rekursiv (Prolog)

Jeweils mit Methoden für Einfügen, Verketten, ...

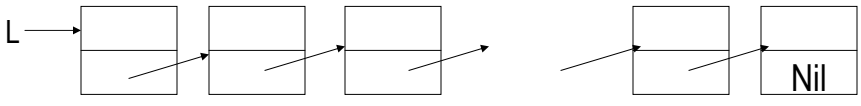
Geschickte Wahl der Datenstruktur



# Listen-Operationen: member

Iterative Implementation für verkettete Liste

```
boolean lookup(List L, Element x)
E= L;
while (true)
{ if (E==NIL) {return false;}
  if (E.Value== x ) {return true;}
  E = E.Next;
}
```



# Listen-Operationen: member

Rekursive Implementation für rekursive Liste

```
boolean lookup(List L, Element x)
 { if ( Head(L) ==  x ) { return true;}
   if ( Tail(L) == NIL)  { return false;}
  return lookup(Tail(L),x);
   }
```

```
member(X, [X | T] ) :- ! .
member(X, [H | T] ) :- member(X, T) .
```



# Listen-Operationen: append

Iterative Implementation für verkettete Liste

```
Liste concatenate(Liste L1, Liste L2)
{ E = L1;
 while (E.Next != NIL) {E=E.Next;}
 E.Next = L2; return L1;
 }
```

Rekursive Implementation für rekursive Liste

```
append([ ], L, L ).
append([X | L1], L2, [X | L3] ) :- append(L1, L2, L3).
```

# Streichen von Elementen

```
delete(X, [ ], [ ] ).
delete(X, [X | L ], Deleted ) :- delete(X, L , Deleted ).
delete(X, [Y | L ], [Y| Deleted] ) :- X \= Y, delete(X, L , Deleted ).
```

Streicht **alle** Vorkommen von X aus L.



# Differenz-Listen

Andere Repräsentation von Listen:

Liste als Anfangsstück $[l_1,...,l_n]$
einer längeren Liste $[l_1,...,l_n, r_1,...,r_m]$ ,
vermindert um den Rest $[r_1,...,r_m]$ .

Schreibweise z.B. $[l_1,...,l_n, r_1,...,r_m] - [r_1,...,r_m]$

Rest $[r_1,...,r_m]$ dabei beliebig,
z.B. auch $[ l_1,...,l_n \mid [ ] ] - [ ]$
allgemeinste Form: $[ l_1,...,l_n \mid R ] - R$

```
conv_DiffList_to_List(A-[ ],A).

conv_List_to_DiffList([ ],L-L).
conv_List_to_DiffList([X|L1], [X|L2]-L)  :- conv_List_to_DiffList(L1, L2-L).
```

# Differenz-Listen

Liste als Anfangsstück $[l_1,...,l_n]$
einer längeren Liste $[l_1,...,l_n, r_1,...,r_m]$ ,
vermindert um den Rest $[r_1,...,r_m]$ .

Verkettung von Differenz-Listen in einem Schritt:

append_dl(A-B, B-C, A-C).

A: $a_1,...,a_n$ , $b_1,...,b_m$, $c_1,...,c_s$

A - B

A - C

B: $b_1,...,b_m$ , $c_1,...,c_s$

B - C

C: $c_1,...,c_s$

```
?- append_dl( [a,b,c|R1] - R1, [1,2|R2] - R2 , L).
L = [a, b, c, 1, 2|_G173]-_G173
```



# Differenz-Listen

Umkehrung einer Liste:

```
naive_reverse([ ],[ ]) .
naive_reverse( [H | T], L) :- naive_reverse(T,L1), append(L1,[H], L).
```

Quadratische Zeit

```
reverse_dl([ ], L - L) .
reverse_dl( [H | T], L - S ) :- reverse_dl(T,L – [H|S] ).

reverse( X,  Y) :- reverse_dl(X,Y – [ ] ).
```

Lineare Zeit

Listen-Subtraktor S
als Akkumulator

```
reverse_acc([ ],R, R).
reverse_acc([ H| T], Acc, R ) :- reverse_acc( T, [H | Acc] , R) .
reverse(L,R) :- reverse_acc(L,[ ],R).
```

# Quicksort

```
quicksort( [H | T], Sorted ) :-
    partition( T,H,Littles,Bigs),
    quicksort(Littles, LSorted),
    quicksort(Bigs,  BSorted ),
    append(LSorted,[H|BSorted],Sorted).
quicksort([ ], [ ]) .


partition( [A | M] ,H, [A | Ls] , Bs)      :- A =< H, partition(M,H,Ls,Bs).
partition( [A | M] ,H,   Ls,     [A | Bs]) :-  A > H, partition(M,H,Ls,Bs).
partition([ ], H, [ ], [ ]).
```



# Quicksort mit Differenz-Liste

```
quicksort_dl( [H | T], Sorted - S ) :-
            partition( T,H,Littles,Bigs),
            quicksort_dl(Littles,   Sorted - [H|B] ),
            quicksort_dl(Bigs,   B - S).
quicksort_dl([ ], L - L) .

quicksort( X,  Y) :- quicksort_dl(X,Y – [ ] ).
```

Verwendet wie vorher:

```
partition( [A | M] ,H, [A | Ls] , Bs)      :- A =< H, partition(M,H,Ls,Bs).
partition( [A | M] ,H,   Ls,     [A | Bs]) :-  A > H,  partition(M,H,Ls,Bs).
partition([ ], H, [ ], [ ]).
```

# Komplexität: **Laufzeit**

abhängig von Eingabewerten
*"worst case"* vs. *"average case"*

- Zeit-Messungen
  - o *Benchmarks*:
    Vergleich von Algorithmen/Programmen
  - o *Profiling*:
    Analyse des Zeitverhaltens im Programm
  - o *Programm-Analyse:*
    Anzahl "elementarer" Anweisungen, Abschätzungen für Schleifen-Durchläufe bzw. Verzweigungen

*90-10-Regel*:
90% der Laufzeit in 10% des Codes verbraucht



# Komplexitäts-Betrachtungen

Programm $\pi$ auf Maschine *M* benötigt
10 234 Operationen und 4323 Byte Speicher
andere Maschine *M'* simuliert *M*:
pro Operation von *M* benötigt *M'* maximal:
12 Operationen und 29 Byte Speicher
$\Rightarrow \pi$ auf *M'* benötigt maximal
10 234 ·12 Operationen und 4323·29 Byte Speicher

$\Rightarrow$Abstraktion von konkreter Maschine
(Unterschied: konstanter Faktor *c*)
Bezug auf universelle Maschine, z.B. TURING-Maschine

# Komplexitäts-Betrachtungen

Allgemeine Aussagen bezogen auf **Problem-Umfang**:
Anzahl der Variablen, Länge eines Ausdrucks, ...

***O*-Notation**

(obere Schranke für Größen-Ordnung von Funktionen):

$T_\pi(n)$ sei Komplexität von $\pi$ bei Problemgröße $n$.
Programm $\pi$ hat Komplexität $O(f(n))$,
falls eine feste (!) Konstante $c$ existiert,
so daß für *fast alle* Problemgrößen $n$ gilt: $T_\pi(n) \leq c \cdot f(n)$

*„fast alle" = alle außer endlich viele (d.h. alle ab einer festen Zahl $n_0$)*



# Zeit-Komplexität

- Erfüllbarkeit eines AK-Ausdrucks mit $n$ Variablen:
  Zeit-Komplexität $O(2^n)$ Wertberechnungen
- *Branch-and-bound*-Verfahren für Suche nach kürzestem Weg in einem Graphen mit $n$ Knoten:
  Zeit-Komplexität $O(n^2)$
  andere Abschätzung mit $m=Max(card(V),card(E))$:
  Zeit-Komplexität $O(m \cdot log(n))$
- Suche in einer geordneten Liste der Länge $n$:
  Zeit-Komplexität $O(log(n))$
- Sortieren einer geordneten Liste der Länge $n$:
  Zeit-Komplexität $O(n\ log(n))$
  Quicksort: $O(n^2)$, im Mittel $O(n\ \log(n))$
- append(L1,L2,L) lineare Zeit bzgl. Länge von L1
- naive_reverse(L,R) quadratrische Zeit bzgl. Länge von L

## Exponentielle vs. polynomiale Komplexität

| n | $n^2$ | $n^3$ | $2^n$ |
|---|---|---|---|
| 10 | 100 | 1000 | 1024 ($\sim 10^3$) |
| 100 | 10000 | 1000000 | $\sim 10^{30}$ |
| 1000 | 1000000 | 1000000000 | $\sim 10^{300}$ |

bei Komplexität $2^n$:

Steigerung der Rechenleistung um Faktor 1000

ermöglicht Steigerung der Problemgröße von *n* auf *n*+10



## Speicher vs. Zeit

Suche in einer geordneten Liste der Länge *n*:
Zeit-Komplexität $O(log(n))$

mit spezieller Datenstruktur („Index“)

für Zugriff auf Liste, z.B. als binärere Baum

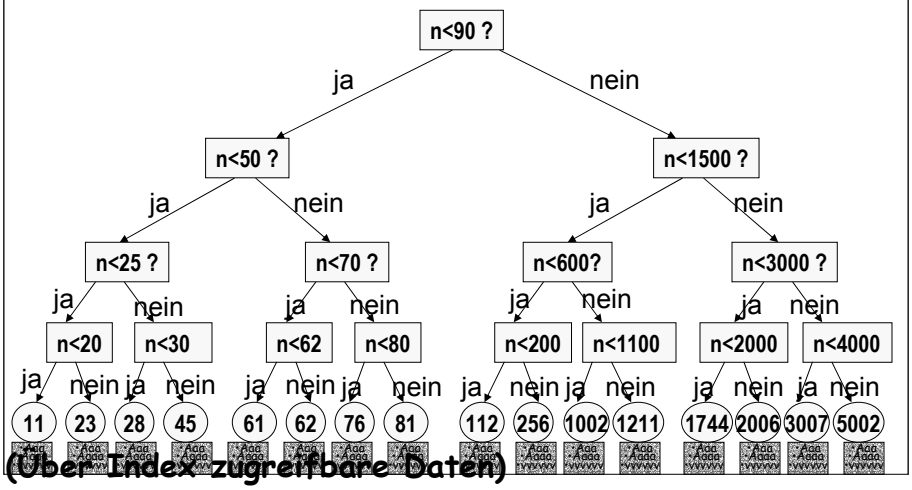
Allgemein:

Speicher-Zeit-“Trade-Off“

[11,23,28,45, 61,62,76,81, 112,256,1002,1211, 1744,2006,3007,5002]

# Indexstruktur „Binärer Suchbaum“

[11,23,28,45, 61,62,76,81, 112,256,1002,1211, 1744,2006,3007,5002]



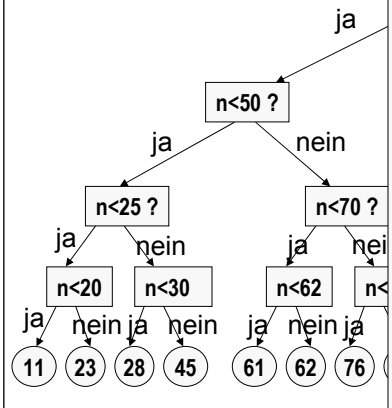

**(Über Index zugreifbare Daten)**



# Indexstr

*log*(*n*) Tests bei günstig gewählter Aufteilung

[11,23,28,45,




```
if n<90 then
    if n <50 then
        if n<25 then
            if n<20 then return 11
            else return 23;
        else
            if n<30 then return 28
            else return 45;
    else
        if n<70 then
            if n<62 then return 61
            else return 62;
        else
            if n<80 then return 76
            else return 81;
else
    if n <1500 then
        if n<600 then
            if n<200 then return 112
            else return 256;
        else
            if n<1100 then return 1002
            else return 1211;
    else
        if n<3000 then
            if n<2000 then return 1744
            else return 2006;
        else
            if n<4000 then return 3007
            else return 5002;
```