

Algorithmen und Datenstrukturen

Peter Rossmanith

Theoretische Informatik, RWTH Aachen

13. April 2015

Organisatorisches

Vorlesung:

Peter Rossmanith (Raum 4104, Informatikzentrum)

Sprechstunde: Mittwochs 11:15–12:00

Übung:

Fernando Sachez, Felix Reidl

sanchez@cs.rwth-aachen.de

reidl@cs.rwth-aachen.de

Organisatorisches

Vorlesung: Mittwochs, 8:30–10:00 Uhr, AH III

Tutorübungen: (ab 13.04.2011)

- Mittwochs 16:15–17:00, E3 9U09
- Mittwochs 17:15–18:00, E3 9U09
- Donnerstags 12:15–13:00, Fo 6
- Donnerstags 16:15–17:00, SG 13
- Freitags 15:15–16:00, SG 23

Anmeldung zur Vorlesung über Campus-Office

Ausgabe des Übungsblatts in den Tutorübungen

Alle Materialien auf Webseite

tcs.rwth-aachen.de/lehre/DA/SS2015/

Organisatorisches

Klausur:

04.08.2015, zwischen 10:30 und 13:00 Uhr (Klausur)

01.09.2015, zwischen 14:00 und 17:00 Uhr (Wiederholung)

Anmeldung: zur "Prüfung Algorithmen und Datenstrukturen"

Abmeldung: über Cambus-Office

Zulassungskriterien:

50% der Punkte aus den Hausaufgaben

Abgabe der Hausaufgaben in Gruppen von bis zu vier Personen

Übersicht

- 1 Einführung
- 2 Suchen und Sortieren
- 3 Graphalgorithmen
- 4 Algorithmische Geometrie
- 5 Textalgorithmen
- 6 Paradigmen

Übersicht

- 1 Einführung
 - Einordnung der Vorlesung
 - Literatur
 - Komplexität von Algorithmen
 - Elementare Datenstrukturen

Einordnung

Etwas Vorwissen kann bei Algorithmen und Datenstrukturen helfen. Diese Vorlesungen sind nützlich:

- Programmierung
- Diskrete Strukturen
- Analysis
- Stochastik

Einordnung

Vorlesungen, die Datenstrukturen und Algorithmen ergänzen:

- Berechenbarkeit und Komplexität
- Formale Systeme, Automaten, Prozesse
- Numerisches Rechnen
- Effiziente Algorithmen

Übersicht

- 1 Einführung
 - Einordnung der Vorlesung
 - **Literatur**
 - Komplexität von Algorithmen
 - Elementare Datenstrukturen

Drei sehr umfassende Bücher



T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein.
Introduction to Algorithms.
The MIT Press, 2d edition, 2001.



T. Ottmann and P. Widmayer.
Algorithmen und Datenstrukturen.
Spektrum Verlag, 2002.



K. Mehlhorn and P. Sanders.
Algorithms and Data Structures: The Basic Toolbox.
Springer, 2008.

Weitere interessante Bücher



A. V. Aho, J. E. Hopcroft, and J. D. Ullman.
The Design and Analysis of Computer Algorithms.
Addison-Wesley, 1974.



S. Skiena.
The Algorithm Design Manual.
Telos, 1997.

Übersicht

- 1 Einführung
 - Einordnung der Vorlesung
 - Literatur
 - **Komplexität von Algorithmen**
 - Elementare Datenstrukturen

Komplexität von Algorithmen

Eine zentrale Frage:

*Zwei verschiedene Algorithmen lösen dasselbe Problem.
Welcher ist schneller?*

Unterscheide:

- 1 Die Laufzeit eines konkreten Algorithmus
- 2 Die Geschwindigkeit mit der sich ein Problem lösen läßt
→ Komplexitätstheorie

Komplexität von Algorithmen

Eine zentrale Frage:

*Zwei verschiedene Algorithmen lösen dasselbe Problem.
Welcher ist schneller?*

Unterscheide:

- 1 Die Laufzeit eines konkreten Algorithmus
- 2 Die Geschwindigkeit mit der sich ein Problem lösen läßt
→ Komplexitätstheorie

Komplexität von Algorithmen

Eine zentrale Frage:

*Zwei verschiedene Algorithmen lösen dasselbe Problem.
Welcher ist schneller?*

Unterscheide:

- 1 Die Laufzeit eines konkreten Algorithmus
- 2 Die Geschwindigkeit mit der sich ein Problem lösen läßt
→ Komplexitätstheorie

Komplexität von Algorithmen

Gegeben: Algorithmus A und Algorithmus B

Wir wählen eine bestimmte Eingabe.

Ergebnis:

- Algorithmus A benötigt 10 Sekunden
- Algorithmus B benötigt 15 Sekunden

Ist Algorithmus A schneller? Er ist auf **dieser Eingabe** schneller.

Es gibt aber viele andere mögliche Eingaben.

Komplexität von Algorithmen

Gegeben: Algorithmus A und Algorithmus B

Wir wählen eine bestimmte Eingabe.

Ergebnis:

- Algorithmus A benötigt 10 Sekunden
- Algorithmus B benötigt 15 Sekunden

Ist Algorithmus A schneller? Er ist auf **dieser Eingabe** schneller.

Es gibt aber viele andere mögliche Eingaben.

Komplexität von Algorithmen

Gegeben: Algorithmus A und Algorithmus B

Wir wählen eine bestimmte Eingabe.

Ergebnis:

- Algorithmus A benötigt 10 Sekunden
- Algorithmus B benötigt 15 Sekunden

Ist Algorithmus A schneller? Er ist auf **dieser Eingabe** schneller.

Es gibt aber viele andere mögliche Eingaben.

Worst-Case-Komplexität von Algorithmen

Lösung:

Die Laufzeit eines Algorithmus ist nicht eine Zahl, sondern eine **Funktion $\mathbf{N} \rightarrow \mathbf{N}$** .

Sie gibt die Laufzeit des Algorithmus für jede **Eingabelänge** an.

Die Laufzeit für Eingabelänge n ist die **schlechteste** Laufzeit aus allen Eingaben mit Länge n .

Wir nennen dies die **Worst-Case-Laufzeit**.

Worst-Case-Komplexität von Algorithmen

Lösung:

Die Laufzeit eines Algorithmus ist nicht eine Zahl, sondern eine **Funktion $\mathbf{N} \rightarrow \mathbf{N}$** .

Sie gibt die Laufzeit des Algorithmus für jede **Eingabelänge** an.

Die Laufzeit für Eingabelänge n ist die **schlechteste** Laufzeit aus allen Eingaben mit Länge n .

Wir nennen dies die **Worst-Case-Laufzeit**.

Sequentielle Berechenbarkeitshypothese

Church'sche These:

Die Menge der algorithmisch lösbaren Probleme ist für alle vernünftigen Berechnungsmodelle identisch.

→ Vorlesung Berechenbarkeit und Komplexität

„Theorem“

Sequentielle Berechenbarkeitshypothese:

Die benötigte Laufzeit für ein Problem auf zwei unterschiedlichen sequentiellen Berechenbarkeitsmodellen unterscheidet sich nur um ein Polynom.

Ein beliebiges Polynom ist für uns zu groß!

Sequentielle Berechenbarkeitshypothese

Church'sche These:

Die Menge der algorithmisch lösbaren Probleme ist für alle vernünftigen Berechnungsmodelle identisch.

→ Vorlesung Berechenbarkeit und Komplexität

„Theorem“

Sequentielle Berechenbarkeitshypothese:

Die benötigte Laufzeit für ein Problem auf zwei unterschiedlichen sequentiellen Berechenbarkeitsmodellen unterscheidet sich nur um ein Polynom.

Ein beliebiges Polynom ist für uns zu groß!

Die RAM (Random Access Machine)

Für die Laufzeitanalyse legen wir uns auf ein Modell fest:
Die **Random Access Machine (RAM)**.

- Einem normalen von Neumann–Computer ähnlich
- Ein sehr einfaches Modell
- Algorithmus auf RAM und Computer: Anzahl der Anweisungen unterscheidet sich nur um konstanten Faktor

Die RAM (Random Access Machine)

Für die Laufzeitanalyse legen wir uns auf ein Modell fest:
Die **Random Access Machine (RAM)**.

- Einem normalen von Neumann–Computer ähnlich
- Ein sehr einfaches Modell
- Algorithmus auf RAM und Computer: Anzahl der Anweisungen unterscheidet sich nur um konstanten Faktor

Die RAM (Random Access Machine)

Für die Laufzeitanalyse legen wir uns auf ein Modell fest:
Die **Random Access Machine (RAM)**.

- Einem normalen von Neumann–Computer ähnlich
- Ein sehr einfaches Modell
- Algorithmus auf RAM und Computer: Anzahl der Anweisungen unterscheidet sich nur um konstanten Faktor

Die RAM (Random Access Machine)

Eine RAM besteht aus:

- 1 Einem Speicher aus Speicherzellen 1, 2, 3, ...
- 2 Jede Speicherzelle kann beliebige Zahlen speichern
- 3 Der Inhalt einer Speicherzelle kann als Anweisung interpretiert werden
- 4 Einem Prozessor der einfache Anweisungen ausführt
- 5 Anweisungen: Logische und Arithmetische Verknüpfungen, (bedingte) Sprünge
- 6 Jede Anweisung benötigt konstante Zeit

Grenzen des RAM-Modells

- Jede Speicherzelle enthält beliebige Zahlen:
Unrealistisch, wenn diese sehr groß werden
- Jede Anweisung benötigt gleiche Zeit:
Unrealistisch, wegen Caches, Sprüngen

Daumenregel: Bei n Speicherzellen, $O(\log n)$ bits verwenden.

```

void quicksort(int N)
{
    int i, j, l, r, k, t;
    l=1; r=N;
    if (N>M)
        while(1) {
            i=l-1; j=r; k=a[j];
            do {
                do { i++; } while(a[i]<k);
                do { j--; } while(k<a[j]);
                t=a[i]; a[i]=a[j]; a[j]=t;
            } while(i<j);
            a[j]=a[i]; a[i]=a[r]; a[r]=t;
            if (r-i ≥ i-l) {
                if (i-l>M) { push(i+1,r); r=i-1; }
                else if (r-i>M) l=i+1;
                else if (stack.is_empty) break;
                else pop(l,r);
            }
            else
                if (r-i>M) { push(l,i-1); l=i+1; }
                else if (i-l>M) r=i-1;
                else if (stack.is_empty) break;
                else pop(l,r);
        }
    for (i=2; i ≤ N; i++)
        if (a[i-1]>a[i]) {
            k=a[i]; j=i;
            do { a[j]=a[j-1]; j--; } while(a[j-1]>k);
            a[j]=k;
        }
}

```

Quicksort – Ein Beispiel

Statt einer **Worst-Case-Analyse** führen wir eine **Average-Case-Analyse** durch.

Die Eingabe besteht aus den Zahlen $1, \dots, n$ in **zufälliger** Reihenfolge.

Wie lange benötigt Quicksort **im Durchschnitt** zum Sortieren?

Es sind $O(n \log n)$ Schritte auf einer RAM.

Für eine genauere Analyse, müssen wir das Modell **genau** festlegen.

Quicksort – Ein Beispiel

Statt einer **Worst-Case-Analyse** führen wir eine **Average-Case-Analyse** durch.

Die Eingabe besteht aus den Zahlen $1, \dots, n$ in **zufälliger** Reihenfolge.

Wie lange benötigt Quicksort **im Durchschnitt** zum Sortieren?

Es sind $O(n \log n)$ Schritte auf einer RAM.

Für eine genauere Analyse, müssen wir das Modell **genau** festlegen.

Einführung

Komplexität von Algorithmen

```

sw -4(r29),r30          addi r2,r2,#-4          addi r8,r3,#4          lw r1,(r4)
add r30,r0,r29          L50:                   j L51                  lw r2,(r3)
sw -8(r29),r31          lw r31,(r2)           addi r3,r8,#-4        sgt r1,r1,r2
subui r29,r29,#464     slt r1,r5,r31         L24:                   beqz r1,L41
sw 0(r29),r2            bnez r1,L50          sgt r1,r5,r9          add r5,r0,r2
sw 4(r29),r3            addi r2,r2,#-4       beqz r1,L32           add r2,r0,r3
sw 8(r29),r4            addi r2,r2,#4        addi r1,r3,#-4        addi r31,r3,#-4
sw 12(r29),r5           lw r6,(r3)           sw (r4),r8            L44:
...                    sw (r3),r31          addi r4,r4,#4        lw r12,(r31)
sw 40(r29),r12          sltu r1,r3,r2        sw (r4),r1            sw (r2),r12
lw r11,(r30)           bnez r1,L15          addi r4,r4,#4        addi r31,r31,#-4
lw r9,4(r30)           sw (r2),r6           j L11                 lw r1,(r31)
slli r1,r11,#0x2       lw r12,(r3)         addi r8,r3,#4        sgt r1,r1,r5
lhi r8,((a+4)>>16)&0xffff sw (r2),r12         L32:                   bnez r1,L44
addui r8,r8,(a+4)&0xffff lw r12,(r7)          sgt r1,r2,r9          addi r2,r2,#-4
lhi r12,((a)>>16)&0xffff sw (r3),r12         bnez r1,L11          sw (r2),r5
addui r12,r12,(a)&0xffff sub r1,r7,r3         addi r7,r3,#-4       L41:
add r7,r1,r12          srai r5,r1,#0x2     seq r1,r4,r10        addi r3,r3,#4
sgt r1,r11,r9          sub r1,r3,r8        bnez r1,L8            sleu r1,r3,r7
beqz r1,L8             srai r2,r1,#0x2    addi r4,r4,#-4       bnez r1,L42
addi r4,r30,#-408     sge r1,r5,r2        lw r7,(r4)           addi r4,r4,#4
add r10,r0,r4          beqz r1,L24         addi r4,r4,#-4       L40:
L11:                   sw (r7),r6          addi r4,r4,#-4       lw r2,0(r29)
addi r3,r8,#-4        sgt r1,r2,r9        j L11                 lw r3,4(r29)
L51:                   beqz r1,L25         lw r8,(r4)           lw r4,8(r29)
add r2,r0,r7          addi r1,r3,#4       L8:                   ...
lw r5,(r7)            sw (r4),r1          slli r1,r11,#0x2     lw r12,40(r29)
L15:                   addi r4,r4,#4       lhi r2,((a)>>16)&0xffff lw r31,-8(r30)
addi r3,r3,#4         sw (r4),r7          addui r2,r2,(a)&0xffff add r29,r0,r30
L49:                   addi r4,r4,#4       add r7,r1,r2         jr r31
lw r1,(r3)            j L11               addi r3,r2,#8        lw r30,-4(r30)
slt r1,r1,r5          addi r7,r3,#-4     sleu r1,r3,r7        ...
bnez r1,L49           L25:                beqz r1,L40          ...
addi r3,r3,#4         sgt r1,r5,r9        addi r4,r2,#4        ...
addi r3,r3,#-4        beqz r1,L34         L42:

```

Quicksort auf dem DLX-Prozessor

Die Laufzeit im Durchschnitt beträgt

$$10A_N + 4B_N + 2C_N + 3D_N + 3E_N + 2S_N + 3N + 6,$$

Schritte, wobei

$$A_N = \frac{2N - M}{M + 2}$$

$$B_N = \frac{1}{6}(N + 1) \left(2H_{N+1} - 2H_{M+2} + 1 - \frac{6}{M + 2} \right) + \frac{1}{2}$$

$$C_N = N + 1 + 2(N + 1)(H_{N+1} - H_{M+2})$$

$$D_N = (N + 1)(1 - 2H_{M+1}/(M + 2))$$

$$E_N = \frac{1}{6}(N + 1)M(M - 1)/(M + 2)$$

$$S_N = (N + 1)/(2M + 3) - 1.$$

und $H_n = 1 + 1/2 + 1/3 + \dots + 1/n$.

O-Notation

Definition

$$O(f) = \{ g: \mathbf{N} \rightarrow \mathbf{R} \mid \exists c \in \mathbf{R}^+ \exists N \in \mathbf{N} \forall n \geq N: |g(n)| \leq c|f(n)| \}$$

$$\Omega(f) = \{ g: \mathbf{N} \rightarrow \mathbf{R} \mid \exists c \in \mathbf{R}^+ \exists N \in \mathbf{N} \forall n \geq N: |g(n)| \geq c|f(n)| \}$$

$$\Theta(f) = \{ g: \mathbf{N} \rightarrow \mathbf{R} \mid \exists c_1, c_2 \in \mathbf{R}^+ \exists N \in \mathbf{N} \forall n \geq N: \\ c_1|f(n)| \leq |g(n)| \leq c_2|f(n)| \}.$$

Informell:

- $g = O(f) \approx g$ wächst langsamer als f
- $g = \Omega(f) \approx g$ wächst schneller als f
- $g = \Theta(f) \approx g$ wächst so schnell wie f

Theorem

Es seien $f, g: \mathbf{N} \rightarrow \mathbf{R}$ zwei Funktionen und $c \in \mathbf{R}$ eine Konstante.

Dann gilt das folgende:

- 1 $O(cf(n)) = O(f(n))$
- 2 $O(f(n)) + O(g(n)) = O(|f(n)| + |g(n)|)$
- 3 $O(f(n) + g(n)) = O(f(n)) + O(g(n))$
- 4 $O(f(n))O(g(n)) = O(f(n)g(n))$
- 5 $O(f(n)g(n)) = f(n)O(g(n))$
- 6 $\sum_{k=1}^n O(f(k)) = O(nf(n))$
- 7 $f(O(1)) = O(1)$

Theorem

Es seien $f, g: \mathbf{N} \rightarrow \mathbf{R}$ zwei Funktionen und $c \in \mathbf{R}$ eine Konstante.

Dann gilt das folgende:

- 1 $O(cf(n)) = O(f(n))$
- 2 $O(f(n)) + O(g(n)) = O(|f(n)| + |g(n)|)$
- 3 $O(f(n) + g(n)) = O(f(n)) + O(g(n))$
- 4 $O(f(n))O(g(n)) = O(f(n)g(n))$
- 5 $O(f(n)g(n)) = f(n)O(g(n))$
- 6 $\sum_{k=1}^n O(f(k)) = O(nf(n))$
- 7 $f(O(1)) = O(1)$

Theorem

Es seien $f, g: \mathbf{N} \rightarrow \mathbf{R}$ zwei Funktionen und $c \in \mathbf{R}$ eine Konstante.

Dann gilt das folgende:

- 1 $O(cf(n)) = O(f(n))$
- 2 $O(f(n)) + O(g(n)) = O(|f(n)| + |g(n)|)$
- 3 $O(f(n) + g(n)) = O(f(n)) + O(g(n))$
- 4 $O(f(n))O(g(n)) = O(f(n)g(n))$
- 5 $O(f(n)g(n)) = f(n)O(g(n))$
- 6 $\sum_{k=1}^n O(f(k)) = O(nf(n))$
- 7 $f(O(1)) = O(1)$

Theorem

Es seien $f, g: \mathbf{N} \rightarrow \mathbf{R}$ zwei Funktionen und $c \in \mathbf{R}$ eine Konstante.

Dann gilt das folgende:

- 1 $O(cf(n)) = O(f(n))$
- 2 $O(f(n)) + O(g(n)) = O(|f(n)| + |g(n)|)$
- 3 $O(f(n) + g(n)) = O(f(n)) + O(g(n))$
- 4 $O(f(n))O(g(n)) = O(f(n)g(n))$
- 5 $O(f(n)g(n)) = f(n)O(g(n))$
- 6 $\sum_{k=1}^n O(f(k)) = O(nf(n))$
- 7 $f(O(1)) = O(1)$

Theorem

Es seien $f, g: \mathbf{N} \rightarrow \mathbf{R}$ zwei Funktionen und $c \in \mathbf{R}$ eine Konstante.

Dann gilt das folgende:

- 1 $O(cf(n)) = O(f(n))$
- 2 $O(f(n)) + O(g(n)) = O(|f(n)| + |g(n)|)$
- 3 $O(f(n) + g(n)) = O(f(n)) + O(g(n))$
- 4 $O(f(n))O(g(n)) = O(f(n)g(n))$
- 5 $O(f(n)g(n)) = f(n)O(g(n))$
- 6 $\sum_{k=1}^n O(f(k)) = O(nf(n))$
- 7 $f(O(1)) = O(1)$

Theorem

Es seien $f, g: \mathbf{N} \rightarrow \mathbf{R}$ zwei Funktionen und $c \in \mathbf{R}$ eine Konstante.

Dann gilt das folgende:

- 1 $O(cf(n)) = O(f(n))$
- 2 $O(f(n)) + O(g(n)) = O(|f(n)| + |g(n)|)$
- 3 $O(f(n) + g(n)) = O(f(n)) + O(g(n))$
- 4 $O(f(n))O(g(n)) = O(f(n)g(n))$
- 5 $O(f(n)g(n)) = f(n)O(g(n))$
- 6 $\sum_{k=1}^n O(f(k)) = O(nf(n))$
- 7 $f(O(1)) = O(1)$

Theorem

Es seien $f, g: \mathbf{N} \rightarrow \mathbf{R}$ zwei Funktionen und $c \in \mathbf{R}$ eine Konstante.

Dann gilt das folgende:

- 1 $O(cf(n)) = O(f(n))$
- 2 $O(f(n)) + O(g(n)) = O(|f(n)| + |g(n)|)$
- 3 $O(f(n) + g(n)) = O(f(n)) + O(g(n))$
- 4 $O(f(n))O(g(n)) = O(f(n)g(n))$
- 5 $O(f(n)g(n)) = f(n)O(g(n))$
- 6 $\sum_{k=1}^n O(f(k)) = O(nf(n))$ falls $|f(n)|$ monoton steigt
- 7 $f(O(1)) = O(1)$

Wie beweisen wir eine Gleichung der Form

$$O(f(n)) = O(g(n))$$

oder allgemein eine Gleichung, mit O-Notation auf der linken *und* rechten Seite?

In Wirklichkeit ist es $O(f(n)) \subseteq O(g(n))$.

Wir beweisen:

Für **jedes** $\hat{f}(n) = O(f(n))$ gilt $\hat{f}(n) = O(g(n))$.

Wie beweisen wir eine Gleichung der Form

$$O(f(n)) = O(g(n))$$

oder allgemein eine Gleichung, mit O-Notation auf der linken *und* rechten Seite?

In Wirklichkeit ist es $O(f(n)) \subseteq O(g(n))$.

Wir beweisen:

Für **jedes** $\hat{f}(n) = O(f(n))$ gilt $\hat{f}(n) = O(g(n))$.

Wir beweisen $O(f(n)) + O(g(n)) = O(|f(n)| + |g(n)|)$:

Beweis.

Sei $\hat{f}(n) = O(f(n))$ und $\hat{g}(n) = O(g(n))$ beliebig.

Dann gilt

$$|\hat{f}(n)| \leq c|f(n)| \text{ und } |\hat{g}(n)| \leq c|g(n)|$$

für ein c und große n .

$$\Rightarrow |\hat{f}(n)| + |\hat{g}(n)| \leq c(|f(n)| + |g(n)|)$$

Daraus folgt $O(|f(n)|) + O(|g(n)|) = O(|f(n) + g(n)|)$.

Es gilt aber $O(f(n)) = O(|f(n)|)$ und $O(g(n)) = O(|g(n)|)$. □

Java

Wir verwenden oft Java für Datenstrukturen und Algorithmen.

Die Vorlesung ist aber von der Programmiersprache unabhängig.

Lernziel sind die einzelnen Algorithmen und Datenstrukturen, nicht ihre Umsetzung in Java.

Für das Verständnis der Vorlesung sind Kenntnisse in einer objektorientierten Sprache notwendig.

Übersicht

- 1 Einführung
 - Einordnung der Vorlesung
 - Literatur
 - Komplexität von Algorithmen
 - **Elementare Datenstrukturen**

Java

```
public class Array < D > {  
    protected D[] a;  
    protected int size;
```

Java

```
public void set(int i, D d) {  
    if(i ≥ size) {  
        size = i + 1;  
        realloc();  
    }  
    a[i] = d;  
}
```

Java

```
private void realloc() {  
    if(size  $\geq$  a.length) {  
        D[] b = (D[]) new Object[2 * size];  
        for(int i = 0; i < a.length; i++) {  
            if(a[i]  $\neq$  null) {  
                b[i] = a[i];  
            }  
        }  
        a = b;  
    }  
}
```

Java

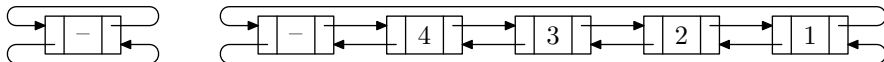
```
public D get(int i) {  
    if(i ≥ size) return null;  
    return a[i];  
}
```

Warum eine eigene Array-Klasse?

- 1 Erweiterbarkeit
- 2 Typsicherheit
- 3 Anfertigen von Statistiken

Doppelt verkettete Listen

- Zusätzlicher Listenkopf
- Daten in Knoten gespeichert
- Zeiger zum Vorgänger und Nachfolger
- Zyklisch geschlossen



Geeignet für kleine assoziative Arrays.

Für Adjazenzlistendarstellung sehr geeignet.