LMU LUDWIG-MAXIMILIANS-UNIVERSITY MUNICH
DEPARTMENT INSTITUTE FOR INFORMATICS
DATABASE SYSTEMS GROUP

Kapitel 12

Datenstrukturen

Skript zur Vorlesung
Einführung in die Programmierung
im Wintersemester 2012/13
Ludwig-Maximilians-Universität München

Übersicht

Übersicht

12.1 Einleitung

- Viele Programme dienen der Verarbeitung von Mengen von Daten.
- Eine Datenmenge muss dazu intern durch eine *Datenstruktur* organisiert und verwaltet werden.
- Als einfache Datenstruktur zur Verwaltung *gleichartiger* Elemente haben wir das Array kennengelernt.
- In der Vorlesung haben wir bei den Wechselgeldalgorithmen *Folgen* als Datenstruktur verwendet.
- Ein Äquivalent zum mathematischen Konzept der *Folge* findet sich in vielen Programmiersprachen als *Liste*.
- Als spezielle Datenstruktur können wir auch die Strings (und verwandte Klassen) betrachten, die für eine Menge von Zeichen stehen.
- Auch eine Klasse dient zunächst der Darstellung von Objekten, die einen Ausschnitt der Wirklichkeit abstrahiert repräsentieren; hier können sogar *verschiedenartige* Elemente verwaltet werden.

12.1 Einleitung

- Bei vielen Anwendungen besteht die wichtigste Entscheidung in Bezug auf die Implementierung darin, die passende Datenstruktur zu wählen.
- Verschiedene Datenstrukturen erfordern für dieselben Daten mehr oder weniger Speicherplatz als andere.
- Für dieselben Operationen auf den Daten führen verschiedene Datenstrukturen zu mehr oder weniger effizienten Algorithmen.
- Manche Datenstrukturen sind dynamisch, andere statisch.
- Die Auswahlmöglichkeiten für Algorithmus und Datenstruktur sind eng miteinander verflochten. Durch eine geeignete Wahl möchte man Zeit und Platz sparen.

12.1 Einleitung

- Eine Datenstruktur können wir auch wieder als Objekt auffassen und entsprechend modellieren.
- Das bedeutet, dass eine Datenstruktur Eigenschaften und Fähigkeiten hat, also z.B. typische Operationen ausführen kann.
- Eine Datenstruktur hat also auch wieder eine *Schnittstelle*, die angibt, wie man sie verwenden kann.
- Für Arrays haben wir z.B. die typischen Operationen:
  - Setze das *i*-te Element von `a` auf Wert `x`: `a[i]=x;`
  - Gib mir das *j*-te Element von `a`: `a[j];`
  - Gib mir die Anzahl der Elemente in `a`: `a.length`
- Bemerkung: das oo Paradigma (insbesondere die Aspekte Abstraktion und Kapselung) eignet sich bestens, um eigene Datenstrukturen (als eigenständige Klassen) zu entwickeln

Übersicht

12.2 Komplexität von Algorithmen

- Ein wichtiger Aspekt von Datenstrukturen ist deren Ressourcenverbrauch
- Algorithmen verbrauchen insbesondere zwei Ressourcen:
  - Rechenzeit
  - Speicherplatz
- Wie können wir z.B. die Rechenzeit messen?

  1. Ansatz: Direktes *Messen* der Laufzeit (z.B. in ms).
  - Abhängig von vielen Parametern (Rechnerkonfig., -last, Compiler, Betr.sys., . . .)
  - Daher: Kaum übertragbar und ungenau.

  2. Ansatz: Zählen der benötigten *Elementaroperationen* des Algorithmus in Abhängigkeit von der Größe *n* der Eingabe.
  - Das Verhalten wird als Funktion der benötigten Elementaroperationen dargestellt.
  - Die Charakterisierung dieser elementaren Operationen ist abhängig von der jeweiligen Problemstellung und dem zugrunde liegenden Algorithmus.
  - Beispiele für Elementaroperationen: Zuweisungen, Vergleiche, arithmetische Operationen, Arrayzugriffe, . . .

12.2 Komplexität von Algorithmen

- Das Maß für die Größe *n* der Eingabe ist abhängig von der Problemstellung, z.B.
  - Suche eines Elementes in einem Array: *n* = Länge des Arrays.
  - Multiplikation zweier Matrizen: *n* = Dimension der Matrizen.
  - Sortierung einer Liste von Zahlen: *n* = Anzahl der Zahlen.
- *Laufzeit*:
  Benötigte Elementaroperationen bei einer bestimmten Eingabelänge *n*.
- *Speicherplatz*:
  Benötigter Speicher bei einer bestimmten Eingabelänge *n*.

12.2 Komplexität von Algorithmen

- Laufzeit einzelner Elementar-Operation ist abhängig von der eingesetzten Rechner-Hardware.
- Frühere Rechner (incl. heutige PDAs, Mobiltelefone, . . .):
  - "billig": Fallunterscheidungen, Wiederholungen
  - "mittel": Rechnen mit ganzen Zahlen (Multiplikation usw.)
  - "teuer": Rechnen mit reellen Zahlen
- Heutige Rechner (incl. z.B. Spiele-Konsolen):
  - "billig": Rechnen mit reellen und ganzen Zahlen
  - "teuer": Fallunterscheidungen

12.2 Komplexität von Algorithmen

- "Kleine" Probleme (z.B. $n = 5$) sind uninteressant:
  Die Laufzeit des Programms ist eher bestimmt durch die Initialisierungskosten (Betriebssystem, Programmiersprache etc.) als durch den Algorithmus selbst.
- Interessanter:
  - Wie verhält sich der Algorithmus bei sehr großen Problemgrößen?
  - Wie verändert sich die Laufzeit, wenn ich die Problemgröße variiere (z.B. Verdopplung der Problemgröße)?
- Das bringt uns zu dem Konzept des *asymptotisches Laufzeitverhalten*.

12.2 Komplexität von Algorithmen

- Mit der *O-Notation* existiert ein Konzept, die asymptotische Komplexität (bzgl. Laufzeit oder Speicherplatzbedarf) eines Algorithmus zu charakterisieren.
- Definition *O*-Notation:
  Seien $f: \mathbb{N} \to \mathbb{N}$ und $s: \mathbb{N} \to \mathbb{N}$ zwei Funktionen (*s* wie *Schranke*).

  Die Funktion *f* ist *von der Größenordnung* $O(s)$, geschrieben $f \in O(s)$, wenn es $k \in \mathbb{N}$ und $m \in \mathbb{N}$ gibt, so dass gilt:

  Für alle $n \in \mathbb{N}$ mit $n \geq m$ ist $f(n) \leq k \cdot s(n)$.

- Man sagt auch: *f* wächst höchstens so schnell wie *s*.

12.2 Komplexität von Algorithmen

- Bemerkungen
  - k ist unabhängig von n:
    k muss dieselbe Konstante sein, die für alle $n \in \mathbb{N}$ garantiert, dass $f(n) \leq k \cdot s(n)$.
  - Existiert keine solche Konstante k, ist f nicht von der Größenordnung $O(s)$.
  - $O(s)$ bezeichnet die Menge aller Funktionen, die bezüglich s die Eigenschaft aus der Definition haben. $O(s)$ heißt auch *Komplexitätsklasse*
  - Man findet in der Literatur häufig $f = O(s)$ statt $f \in O(s)$.

12.2 Komplexität von Algorithmen

- *Rechnen* mit der O-Notation
  - Elimination von Konstanten:
    - $2 \cdot n \in O(n)$
    - $n/2 + 1 \in O(n)$
  - Bei einer Summe zählt nur der am stärksten wachsende Summand (mit dem höchsten Exponenten):
    - $2n^3 + 5n^2 + 10n + 20 \in O(n^3)$
    - $O(1) \subseteq O(\log n) \subseteq O(n) \subseteq O(n \log n) \subseteq O(n^2) \subseteq O(n^3) \subseteq \ldots \subseteq O(2^n)$
  - Beachte:
    $1000 \cdot n^2$ ist nach diesem Leistungsmaß immer "besser" als $0{,}001 \cdot n^3$, auch wenn das m, ab dem die $O(n^2)$-Funktion unter der $O(n^3)$-Funktion verläuft, in diesem Fall sehr groß ist (m=1 Million).

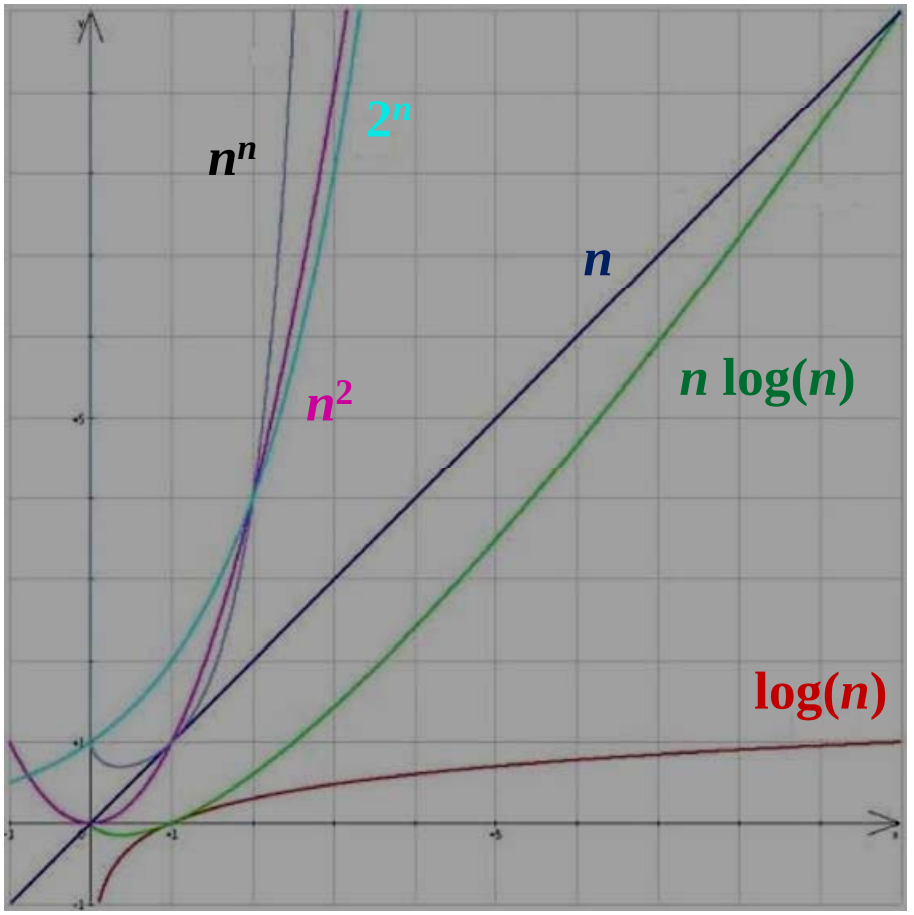
12.2 Komplexität von Algorithmen

- Wichtige Klassen von Funktionen (*Komplexitätsklassen*):

	Sprechweise	**Typische Algorithmen / Operationen**
$O(1)$	konstant	Addition, Vergleichsoperationen, rekursiver Aufruf, . . .
$O(\log n)$	logarithmisch	Suchen auf einer sortierten Menge
$O(n)$	linear	Bearbeiten jedes Elementes einer Menge
$O(n \cdot \log n)$		gute Sortierverfahren
$O(n \cdot \log^2 n)$		
	⋮	
$O(n^2)$	quadratisch	primitive Sortierverfahren
$O(n^k), k \geq 2$	polynomiell	
	⋮	
$O(2^n)$	exponentiell	Ausprobieren von Kombinationen

12.2 Komplexität von Algorithmen

- Die *O*-Notation hilft insbesondere bei der Beurteilung, ob ein Algorithmus für großes *n* noch geeignet ist bzw. erlaubt einen Effizienz-Vergleich zwischen verschiedenen Algorithmen für große *n*.
- Schlechtere als polynomielle Laufzeit gilt als nicht effizient, kann aber für viele Probleme das best-mögliche sein.



Übersicht

12.3 Ein Datenmodell für Listen

- Wie wir gesehen haben, erlauben Arrays effizient den sogenannten "wahlfreien Zugriff", d.h. wir können auf ein beliebiges Element in *O*(1) zugreifen (auf das `i`-te mit `a[i-1]`).
- Bei der Spezifikation von Folgen (und entsprechenden Listen-Implementierungen) gilt das nicht. Der Zugriff auf das *n*-te Element erfordert einen Aufwand in *O*(n).
  Zur Erinnerung: *Definition der Projektion*

$$\pi(x,i) = \begin{cases} first(x), & \text{falls } i = 1, \\ \pi(rest(x), i-1) & \text{sonst.} \end{cases}$$

- Dafür können Folgen (Listen) beliebig wachsen, während wir die Größe eines Arrays von vornherein festlegen müssen.

12.3 Ein Datenmodell für Listen

- Um uns die Implementierung (in Java) von Listen zu überlegen, erinnern wir uns zunächst noch einmal an Folgen:
- Konkatenation einer Folge $x \in M^*$ an ein Element $a \in M$:

$$prefix : M \times M^* \to M^* \qquad \text{mit } prefix(a, x) = (a) \circ x$$

- Induktive Definition von M^*:
  1. $( ) \in M^*$
  2. Ist $a \in M$ und $x \in M^*$, dann ist $prefix(a, x) \in M^*$.

- Zugriff auf das erste Element:

$$first : M^+ \to M \qquad \text{mit } first(prefix(a, x)) = a$$

- Liste nach Entfernen des ersten Elements:

$$rest : M^+ \to M^* \qquad \text{mit } \mathrm{rest}(prefix(a, x)) = x$$

12.3 Ein Datenmodell für Listen

Wir halten fest:

- Eine Liste kann leer sein.
- Ein Element wird vorne an eine Liste angehängt (*prefix*).
- Wir können nur auf das erste Element einer Liste zugreifen und auch nur dieses entfernen (*first/rest*).
- Zusätzlich: Eine Liste kann Auskunft über ihre Länge (= Anzahl der Elemente) geben (*size*).

  Die Länge einer Folge ist rekursiv definiert als:

$$size(x) = \begin{cases} 0, & \text{falls } x = (), \\ 1 + size(rest(x)) & \text{sonst.} \end{cases}$$

- Wie können wir nun Folgen als Objekte modellieren?

12.3 Ein Datenmodell für Listen

- Erster Versuch:
  - Als Eintrag verwenden wir Objekte vom Typ `java.lang.Object`. Mehrerer Einträge halten wir in einem `Object`-Array.

```
public class ListeErsterVersuch {
  private Object[] entries;

  public Object() {
        this.entries = new Object[0];
  }

  public int size() {
        return this.entries.length;
  }

  public Object first() {
        return this.entries[0];
  }
}
```

12.3 Ein Datenmodell für Listen

- Die Methoden `prefix` und `rest` werden leider etwas komplizierter, da Arrays nicht dynamisch sind. Wir müssten immer neue Arrays erzeugen und kopieren …

```
public void prefix(Object obj) {
    Object[] newEntries = new Object[this.entries.length + 1];
    newEntries[0] = obj;
    for(int i=0; i++; i<this.entries.length) {
            newEntries[i+1] = this.entries[i];
    }
    this.entries = newEntries;
}
```

- Welche Komplexität hat `prefix` angewandt auf ein Array mit *n* Objekten?