Algorithmen und Datenstrukturen

Einführung

Prof. Justus Piater, Ph.D.

6. März 2024

Dieser Kurs folgt in weiten Teilen dem sehr empfehlenswerten Lehrbuch *Data Structures and Algorithms in Java* [Goodrich u. a. 2014].

Inhaltsverzeichnis

Willkommen bei unserer Einführung in Algorithmen und Datenstrukturen!

Sie können bereits ein wenig programmieren. Nun möchten Sie Ihre Programmierfähigkeiten einsetzen, um bestimmte Probleme zu lösen. Nehmen wir als Beispiel an, Sie arbeiten für ein großes Krankenhaus, und es treffen ständig Notfallpatienten ein. Jeder Patient wird mit dem Schweregrad seiner Verletzung oder seiner Krankheit registriert. Ihre Aufgabe ist es, die Patienten zu priorisieren, so dass sich die nächste frei verfügbare Ärztin unter der großen Menge der Wartenden sofort dem dringendsten Notfall widmen kann. Wie lösen Sie dieses Problem?

Die einfachste Möglichkeit wäre, die Patienten in einer Liste in der Reihenfolge ihres Eintreffens zu verwalten. Allerdings muss die Ärztin dann die gesamte Liste durchgehen, um den dringendsten Fall zu finden.

Besser wäre es, die Patienten beim Eintreffen in die Liste gemäß ihres Schweregrades einzusortieren. Dann findet die Ärztin den dringendsten Notfall immer am Anfang der Liste. Allerdings muss für jede eintreffende Patientin im Schnitt die halbe Liste durchgegangen werden, um sie am korrekten Platz einzusortieren.

Wir können hier deutlich Zeit gewinnen, indem wir die Patienten statt in einer Liste in einer sogenannten Baumstruktur verwalten. Jede eintreffende Patientin wird hinten eingefügt und so lange nach vorn durchgereicht, bis sie ihren Platz gefunden hat – ähnlich wie bei der sortierten Liste, aber in viel weniger Schritten. Die Details werden wir uns später in diesem Kurs anschauen.

Die Art und Weise, wie die Patientendaten gespeichert werden, ist hier also entscheidend. Dies ist die Frage nach der ***Datenstruktur***. In unserem Beispiel habe ich angedeutet, dass die *Baumstruktur* eine effizientere Lösung ermöglicht als die *Liste*.

Auf diesen Datenstrukturen haben wir jeweils eine Folge bestimmter Zugriffe durchgeführt, z.B. Vertauschungen, um eine Patientin einzufügen oder die dringendste Patientin herauszuholen. Ein ***Algorithmus*** ist eine systematische Beschreibung solcher Einzelschritte, die automatisch ausgeführt werden können, um einen gewünschten Effekt zu erzielen.

Wie Sie sehen, gehen Datenstrukturen und Algorithmen Hand in Hand. In diesem Kurs werden Sie verschiedene Algorithmen, Datenstrukturen und sogenannte abstrakte Datentypen kenenlernen, mit denen Sie viele verschiedene Anwendungsprobleme effizient lösen können. Darüber hinaus werden Sie lernen, eigene abstrakte Datentypen, Datenstrukturen und Algorithmen zu entwickeln und zu analysieren.

1 Organisatorisches

Ziele dieser Lehrveranstaltung [Slide 1]

Grundbegriffe:

- ***Abstrakter Datentyp*** (ADT): *Welche Operationen* sollen auf den Daten *effizient* ausgeführt werden?
- ***Datenstruktur*** (DS): Welche *Repräsentation* meiner Daten eignet sich hierfür?
- ***Algorithmus***: *Wie* können die gefragten Operationen effizient auf dieser Datenstruktur durchgeführt werden?

Kenntnisse und Kompetenzen:

- Standard-ADT sowie Standard-DS + Algorithmen
- Analysemethoden (Korrektheit, Verhalten, Effizienz) von Algorithmen + DS
- Design und Analyse neuer ADT, DS + Algorithmen

Organisation dieser Lehrveranstaltung [Slide 2]

Alle Informationen und Materialien finden sich auf der Homepage der Lehrveranstaltung, reachable from OLAT: `https://iis.uibk.ac.at/courses/2024s/703010/`

Vorlesung: Einführung neuen Materials

- Kerninhalte (siehe auch die Kurzvideos)
- Ergänzende Inhalte
- Fragen und Antworten

Hausarbeiten: Theoretische Übungen; Programmierübungen

- Ausgabe: zum PS am Tag nach der Vorlesung
- Abgabe: bis 13:00 vor der nächsten Vorlesung
- Besprechung: im Proseminar 8 Tage nach der Vorlesung

Proseminar: Übungen zum Thema der Vorlesung von vor 8 Tagen

- Präsentationen der vergangenen Hausarbeit
- Besprechung der aktuellen Hausarbeit

Interaktion [Slide 3]

Während der Live-Veranstaltungen

- verbal
- Particify

Außerhalb der Live-Veranstaltungen

- OLAT-Forum
  - einziger Kommunikationskanal (damit alle profitieren)

ars.uibk.ac.at
3109 5224

Diese Lehrveranstaltung…

A: ist Teil meines Bachelorstudiums Informatik

B: ist Teil eines anderen Curriculums (Lehramt, Erweiterungsstudium, …)

C: besuche ich als Gast

D: weiß nicht

Inverted Classroom [Slide 4]

- Sie erarbeiten sich die Inhalte ***selbstständig*** und ***im Voraus***.
- Präsenzzeiten:
  - ergänzende Erklärungen
  - Q&A (in beiden Richtungen)
  - ergänzendes Material
  - etwas verkürzt gegenüber dem Stundenplan
- VO-Zeit = Vorbereitungszeit + Präsenzzeit

  Bitte bei der LV-Evaluierung beachten!

Meine Erwartung an dieses Format:

A: Es macht mir Angst.

B: Ich bin völlig emotionslos.

C: Ich freue mich darauf.

D: weiß nicht

Bewertung [Slide 5]

Vorlesung:

Klausur

- am Termin der letzten Sitzung

Proseminar:

- Midterm 2024-05-08 *zur Vorlesungszeit*
- Präsentationen
- Aufgabenzettel

Erwarteter Arbeitsaufwand:

- 7,5 ECTS = 7,5 · 25 h = 12,5 h / Woche

2 Abstraktionsebenen der Programmierung

Programmieren 1945 (ENIAC) [Slide 6]

Rechnerarchitektur: *von Neumann* (CPU, adressierbarer Speicher)

Programmierung: Byte-Sequenzen (*opcodes*), die elementare mathematische und logische Operationen repräsentieren (*Maschinencode*)

Interaktion: = Programmierung

Programmieren heute [Slide 7]

Rechnerarchitektur: in den Grundlagen unverändert

Programmierung:

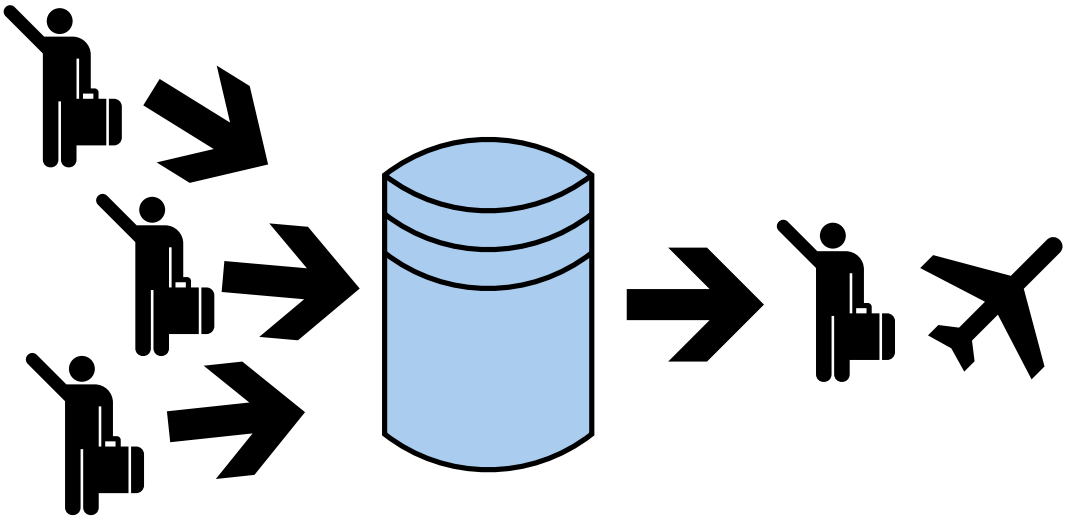
- Höhere Programmiersprachen
  - übersetzt in Maschinencode (C++, C, Rust, Go)
  - übersetzt in Bytecode für eine virtuelle Maschine (Java)
  - interpretiert = *zur Laufzeit* übersetzt in Bytecode für eine virtuelle Maschine oder ("jit") in Maschinencode (Julia, R, Python, Ruby, JavaScript, Lisp)
- Bibliotheken von *Datenstrukturen* und *Algorithmen*, sogar in Programmiersprachen integriert (hier Python):

  ```
  dic['computer'] = 'Rechner'
  ```

- *Domain-Specific Languages*

Interaktion: Graphische Nutzeroberflächen, Touchscreens, gesprochene Sprache

Beispiel: Stand-By-Fluggäste priorisieren [Slide 8]



Viele Abstraktionsebenen [Slide 9]

1. Stand-By-Fluggäste nach Frequent-Flyer-Status einordnen.
2. Zwei voneinander unabhängige Schleifen:
   - Ankommenden Passagier einfügen
   - Passagier mit höchstem Status herausholen
3. Vorrangwarteschlange (***abstrakter Datentyp***)
4. Implementierung auf Basis eines *Heaps* (***Datenstruktur***)
5. Beschreibung der Funktionsweise der Operationen `insert()` und `removeMin()` (***Algorithmen***)
6. Implementation der Datenstruktur und Algorithmen in Java
7. Java Bytecode
8. … Entsprechendes für die JVM …
9. … Entsprechendes für die CPU …
10. … Halbleiter …
11. … Quantenphysik …

Breite Kompetenzen in der Informatik [Slide 10]

- Abstraktion und Formalisierung des Problems
- Design und Analyse möglicher Lösungen auf Basis abstrakter Datentypen
- Design und Analyse möglicher Implementationen auf Basis von Datenstrukturen und Algorithmen

Wichtig

Grundlage: Kenntnis von und kompetenter Umgang mit

- Standardlösungen (abstrakten Datentypen, Datenstrukturen, Algorithmen) und ihren Eigenschaften
- Analysemethoden für Algorithmen und Datenstrukturen

3 Abstrakte Datentypen, Datenstrukturen und Algorithmen

Video 1 beginnt hier.

Erklärung

Die drei zentralen Grundbegriffe dieses Kurses sind der abstrakte Datentyp, die Datenstruktur, und der Algorithmus.

Beispiel: Vorrangwarteschlange *(Priority Queue)* [Slide 11]

Definition des *Abstrakten Datentyps*:

- Datensatz = Schlüssel, Wert
- Totalordnung auf der Menge der Schlüssel
  Informell: Zwei Schlüssel sind entweder identisch, oder einer ist *kleiner* als der andere; diese Relationen zwischen beliebigen Schlüsseln enthalten keine Widersprüche.
- Zwei Operationen:

```
insert(k, v) // fügt Wert v unter Schlüssel k ein
removeMin()  // entfernt v mit min. k und liefert v zurück
```

Implementierung mit verschiedenen *Datenstrukturen* möglich:

- unsortiertes Array
- sortierte verkettete Liste
- *Heap* (Haufen; partiell sortierter Baum)

Effizienz der *Algorithmen*, die die zwei Operationen implementieren?

Beispiel

Um diese Begriffe zu verdeutlichen, betrachten wir das Beispiel der Vorrangwarteschlange, wie sie z.B. dafür verwendet werden kann, Stand-By-Passagiere entsprechend ihrer Priorität zu berücksichtigen, unabhängig von oder nicht nur gemäß der Reihenfolge ihres Auftauchens.

Eine Vorrangwarteschlange behandelt Datensätze in der Form von Schlüssel-Wert-Paaren. Hier betrachten wir den Schlüssel vereinfacht als einen Zahlenwert, der bspw. die Priorität eines Passagiers repräsentiert. Der zugehörige Wert ist der Datensatz des Passagiers in der Datenbank der Fluggesellschaft.

Der abstrakte Datentyp der Vorrangwarteschlange ist durch zwei Operationen definiert: `insert(k, v)` fügt den Wert `v` unter dem Schlüssel `k` in die Vorrangwarteschlange ein, und `removeMin()` entfernt das Schlüssel-Wert-Paar mit dem niedrigsten Schlüsselwert `k`, also der höchsten Priorität, aus der Vorrangwarteschlange, und liefert den zugehörigen Wert `v` zurück.

Welche Datenstrukturen eigenen sich für eine Vorrangwarteschlange? Mit anderen Worten, welche Datenstrukturen erlauben effiziente Algorithmen für die beiden Methoden `insert()` und `removeMin()`? Verschiedene Datenstrukturen haben verschiedene Vor- und Nachteile.

Wir können z.B. eine Vorrangwarteschlange auf der Basis eines unsortierten Arrays implementieren. Damit ist `insert()` einfach und effizient implementierbar: Wir fügen das Schlüssel-Wert-Paar einfach hinter den letzten Eintrag des Arrays ein, was sich in konstanter Laufzeit machen lässt, unabhängig von der Anzahl n der Einträge in der Vorrangwarteschlange. Der Preis dafür ist, dass `removeMin()` das gesamte Array nach dem

Eintrag mit dem kleinsten Schlüssel durchsuchen muss, was eine Laufzeit proportional zu n beansprucht.

Wenn wir dagegen eine sortierte Liste verwenden, dann kann `removeMin()` das höchstpriorisierte Element einfach am Kopf der Liste abgreifen, in konstanter Zeit. Nun allerdings muss `insert()` die Liste im Schnitt zur Hälfte durchlaufen, um seinen Eintrag an der richtigen Stelle einzusortieren. Dies beansprucht wiederum eine Zeit proportional zu n.

Wäre es nicht schön, eine Datenstruktur zu haben, die einen Kompromiss zwischen diesen beiden Laufzeiten ermöglicht, und damit eine kürzere Gesamtlaufzeit über viele Operationen hinweg? Eine solche Datenstruktur ist der sogenannte *heap*, der es erlaubt, sowohl `insert()` als auch `removeMin()` mit einer Laufzeit logarithmisch in n zu implementieren.

Grundbegriffe [Slide 12]

- ***Abstrakter Datentyp*** (ADT): *Welche Operationen* sollen auf den Daten *effizient* ausgeführt werden?

  Definiert eine *Schnittstelle* (*interface*) in der Form von *Methoden*, die auf der Basis anderer ADT und/oder DS implementiert werden können

- ***Datenstruktur*** (DS): Welche *Repräsentation* meiner Daten eignet sich hierfür?

  Definiert mittels Graphentheorie (Bäume), Mathematik (Vektor), …

- ***Algorithmus***: *Wie* können die gefragten Operationen effizient auf dieser Datenstruktur Schritt für Schritt durchgeführt werden?

  Definiert z.B. durch *Pseudocode*

Wichtig

Algorithmen greifen auf *Datenstrukturen* zu; sie gehören untrennbar zusammen.

Erklärung

Nach diesem Beispiel stellen wir noch einmal die drei Grundbegriffe heraus:

Ein abstrakter Datentyp beschreibt, *welche Operationen* auf den Daten ausgeführt werden sollen. Gegebenenfalls ist wichtig, welche dieser Operationen besonders effizient sein sollen. Ein abstrakter Datentyp sagt jedoch nichts darüber aus, *wie* diese Operationen ausgeführt werden, sondern nur, *welche*.

Damit beschreibt ein abstrakter Datentyp eine *Schnittstelle* zu den Daten, auf englisch *interface*. Diese Schnittstelle definiert Methoden, die dann später ggf. durch andere abstrakte Datentypen oder Datenstrukturen implementiert werden können.

In unserem Beispiel der Vorrangwarteschlange bestand die Schnittstelle in den Methoden `insert()` und `removeMin()`. *Nur über diese Schnittstelle*, also nur über diese beiden Methoden, kann auf die Daten in der Vorrangwarteschlange zugegriffen werden.

Eine Datenstruktur beschreibt, wie die Daten so *repräsentiert* werden können, dass für die gewünschten Methoden effiziente Algorithmen existieren. Datenstrukturen werden durch geeignete formale Systeme beschrieben, die aus der Graphentheorie stammen, wie z.B. Bäume oder gerichtete Graphen, oder aus der Mathematik, wie z.B. Vektoren oder Matrizen.

Ein Algorithmus ist eine formale Beschreibung der *einzelnen Schritte* die die Methoden (hier `insert()` und `removeMin()`) *auf den Datenstrukturen* ausgeführen. Es ist hiermit klar, dass Datenstrukturen und Algorithmen untrennbar zusammengehören.

Zur Beschreibung von Algorithmen verwenden wir in diesem Kurs manchmal konkrete Programmiersprachen, aber häufiger sogenannten Pseudocode.

Pseudocode [Slide 13]

- Menschenlesbar, nicht unbedingt maschinenlesbar
- Keine formale Syntax oder Semantik
- Reduziert auf ein Minimum; ohne Spezifika realer Programmiersprachen
- Ausreichend Detail für die Spezifikation der Funktionsweise und für die Analyse von Korrektheit und Effizienz

Beispiel

```
Algorithm arrayMax(A,n):
  Require: An array A storing n ≥ 1 integers.
  Ensure: Return the maximum element in A.
  m ← A[0]
  for i ← 1 to n − 1 do
    if m < A[i] then
      m ← A[i]
  return m
```

Erklärung

Pseudocode dient der kompakten Beschreibung von Algorithmen ohne den Ballast realer Programmiersprachen. Wir verwenden ihn hier ohne formale Definition, sondern verlassen uns darauf, dass seine Notation für angehende Informatik-Fachleute intuitiv verständlich ist. Nichtsdestotrotz ist Pseudocode hinreichend detailliert, um Algorithmen und ihre einzelnen Berechnungsschritte exakt zu beschreiben und ihre Analyse hinsichtlich Korrektheit und Effizienz zu ermöglichen.

Dieses Beispiel beschreibt einen Algorithmus namens `arrayMax()`. Er verlangt, dass ein Array mit mindestens einem Element übergeben wird; die Anzahl der Elemente des Arrays steht im Parameter n. Dann verspricht er, dass er das größte Element des Arrays zurückliefern wird. Impliziert ist das Versprechen, dass der Algorithmus in endlicher Zeit terminieren wird.

Nach dieser Spezifikation folgen die einzelnen Berechnungsschritte. Der linksweisende Pfeil bedeutet, dass der Wert auf der rechten Seite der Variablen auf der linken Seite zugewiesen wird. Die fett gedruckten Schlüsselwörter und die Array-Indizierung mit eckigen Klammern sind realen Programmiersprachen nachempfunden, ohne einer bestimmten Programmiersprache entsprechen zu wollen. Der Rest sind übliche mathematische Operatoren.

Mit solchem Pseudocode werden wir im Laufe dieses Kurses viele unserer Algorithmen beschreiben.

Quiz [Slide 14]

Genau eine der folgenden Aussagen ist falsch. Welche?

A: Ein abstrakter Datentyp definiert, welche Operationen auf Daten ausgeführt werden sollen.

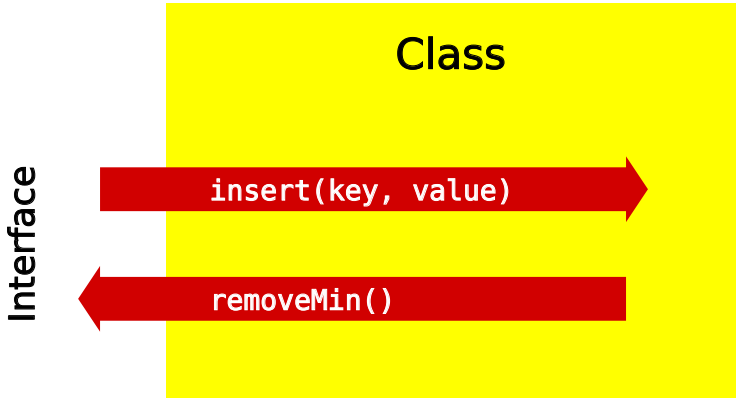
B: Eine Datenstruktur beschreibt, welche Algorithmen auf einem abstrakten Datentyp ausgeführt werden.

C: Ein Algorithmus ist eine Schritt-für-Schritt-Beschreibung einer Methode, um eine bestimmte Berechnung durchzuführen.

D: weiß nicht

ADT und Objekt-orientierte Programmierung (OOP) [Slide 15]

Algorithmik	**OOP**
ADT	Interface
DS + Algorithmen	Klasse
Instanzen/Daten	Objekte



ADT in Java (vereinfacht) [Slide 16]

```
public interface PriorityQueue {
  public void insert(int key, Object value);
  public Object removeMin();
}

public class SortedListPriorityQueue
  implements PriorityQueue {
  public void insert(int key, Object value) { ... }
  public Object removeMin() { ... }
}

public class HeapPriorityQueue
  implements PriorityQueue {
  public void insert(int key, Object value) { ... }
  public Object removeMin() { ... }
}

  PriorityQueue pqOne = new SortedListPriorityQueue();
  PriorityQueue pqTwo = new HeapPriorityQueue();
```

Beispiel: ADT Zuordnungstabelle *(Map, Dictionary)* [Slide 17]

Definition:

- Datensatz = Schlüssel, Wert
- (keine Ordnung auf der Menge der Schlüssel notwendig)
- Drei Operationen:

```
get(k)    // liefert Wert v von Schlüssel k zurück (oder null)
put(k, v) // fügt Wert v unter Schlüssel k ein
remove(k) // entfernt v mit k und liefert v zurück (oder null)
```

Beispiel

In Python:

```
dic = {}                      # instantiate a Dictionary
dic["computer"] = "Rechner"   # put("computer", "Rechner")
print(dic["computer"])        # get("computer")
del dic["computer"]           # remove("computer")
```

DATENSTRUKTUREN (... und Algorithmen) [Slide 18]

Wichtig

Denken in

1. abstrakten Datentypen
2. Datenstrukturen
3. Algorithmen

in dieser Reihenfolge!

> *...git actually has a simple design, with stable and reasonably well-documented data structures. In fact, I'm a huge proponent of* ***designing your code around the data****, rather than the other way around, and I think it's one of the reasons git has been fairly successful [...] I will, in fact, claim that the difference between a bad programmer and a good one is whether he considers his code or his data structures more important. Bad programmers worry about the code. Good programmers worry about* ***data structures and their relationships****.*

[Linus Torvalds 2006-07-27 in einer E-Mail an die Git-Mailingliste; Hervorhebungen J.P.]