Algorithmen & Datenstrukturen

Julian Huber & Matthias Panny

Datenstrukturen

(engl. data structures)

Datenstrukturen

> Algorithms + Data Structures = Programs
>
> — *Niklaus Wirth*

> Bad programmers worry about the code. Good programmers worry about data structures and their relationships.
>
> — *Linus Torvalds*

🎯 Lernziele

- Studierende verstehen die Bedeutung von Datenstrukturen
- Studierende kennen grundlegende Datenstrukturen
- Studierende können Datenstrukturen in Python anwenden

Datenstrukturen

- Datenstrukturen organisieren Daten, sodass sie effizient verwendet werden können, indem sie z.B. geschickt mittels Hardware abgelegt werden
- wie die Daten beschreiben und abgelegt werden bedingt wie gut wir z.B. auf Sie zugreifen können, wie schnell wir sie verändern können, wie viel Speicher sie benötigen
- damit wie gut sie sich für bestimmte Algorithmen eignen
- wie gut Computer sie verarbeiten können z.B. Geschwindigkeitsvorteile bei Vektorisierung mit `numpy`-Arrays

Beispiel: Dijkstra's Two Stack Algorithm

- Wird sind es gewohnt arithmetische Ausdrücke in Infixnotation zu schreiben, ein Operator steht dabei zwischen zwei Operanden
  `((5 + 3) * (5 + 1))`
- Computer können solche Ausdrücke nicht direkt auswerten.
- Sie müssen so im Speicher abgelegt werden, dass sie effizient ausgewertet werden können
- Beispielsweise auf zwei Stapeln (Stacks), die die Operanden und Operatoren speichern

```
Operatoren      Operanden
                5
                3
+
                5
                1
+
*
```

Beispiel: Dijkstra's Two Stack Algorithm

Warum sollte man sich mit Datenstrukturen beschäftigen?

- Algorithmen sind nur so gut wie die Datenstrukturen auf denen sie arbeiten
- Speicher- & Rechenkapazitäten mittlerweile sehr groß aber immer noch nicht unbegrenzt
- Für gängige Probleme sind effiziente Datenstrukturen bekannt → man sollte sie kennen (vgl. `numpy`-Arrays für `matmul_solution_numpy.py`)

Datentypen

- Datentypen beschreiben die Art an Werten, die in einer Variablen gespeichert werden können
- Datenstrukturen speichern mehrere Elemente eines oder mehrere Datentypen

C/C++

- `int`, `float`, `double`, `char`, `bool`, ...
- selbst erstellte Sturkturen & Klassen

Python

- `int`, `float`, `str`, `bytes`, `bool`, `datetime`, ...
- selbst erstellte Klassen

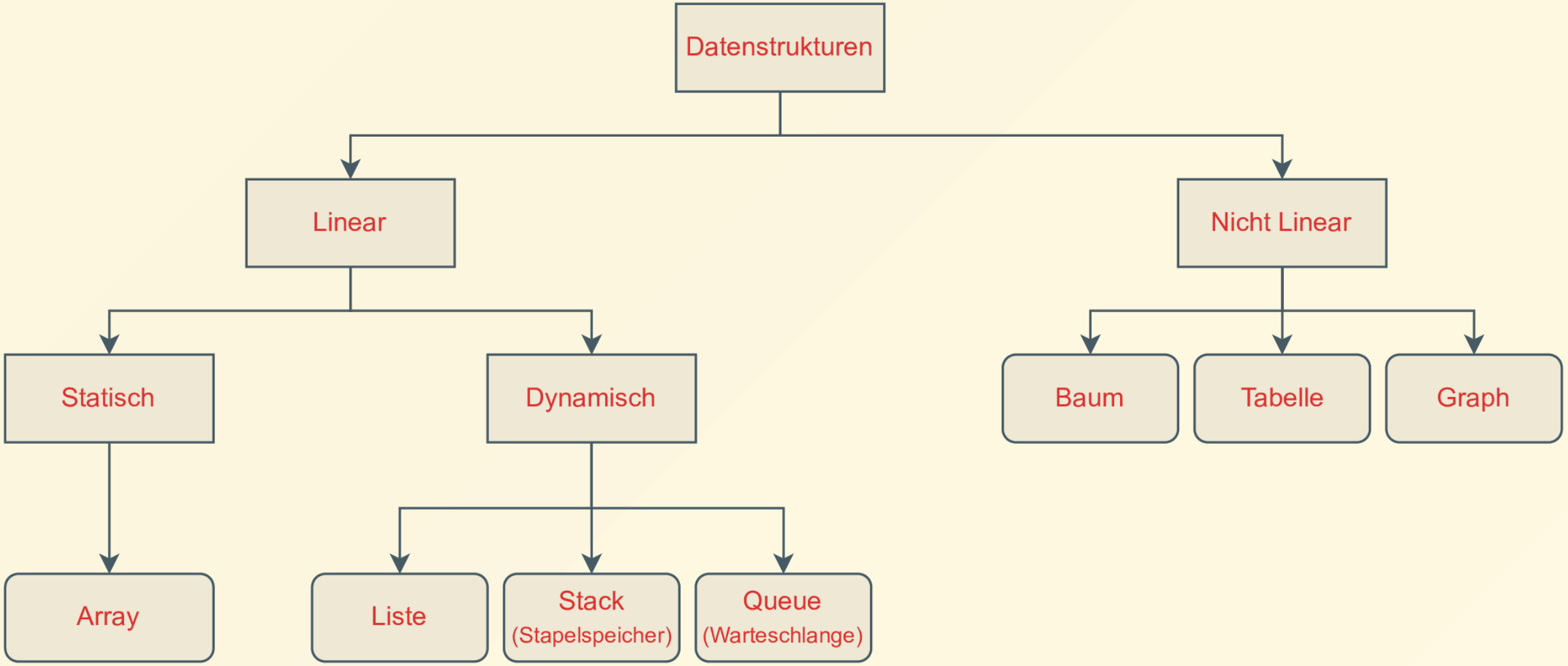
Der (abstrakte) Datentyp (ADT)

- Beschreibt eine Datenstruktur unabhängig von einer Implementierung
- Beschreibt die Operationen die auf der Datenstruktur ausgeführt werden können
- Kann prizipiell auch eine normale Klasse darstellen
- Wird häufig (auch in dieser Lehrveranstaltung) synonym mit Datenstruktur verwendet

Beispiel Stack (stapel):

- Speichert Elemente mit Positionsabhängigkeit zu einander
- Operation um ein Element oben auf den Stack zu legen (push)
- Operation um das oberste Element vom Stack zu entfernen (pop)

Taxonomie der Datenstrukturen



Taxonomie der Datenstrukturen

Lineare Datenstrukturen

- Elemente werden sequentiell geordnet
- Traversieren (Durchlaufen) aller Elemente effizient möglich
- Python `np.array`, `list`, C++ `std::vector`

Nicht Lineare Datenstrukturen

- Elemente werden nicht sequentiell geordnet
- Traversieren (durchlaufen) aller Elemente nicht effizient möglich
- Python `dict`, C++ `std::map`

Taxonomie der Datenstrukturen

Statische Datenstrukturen

- Größe der Datenstruktur ist ab Deklaration fest definiert
- Speicherverbrauch immer bekannt und konstant
- Python *nicht vorhanden*, C++ `std::array` bzw. `int x[10]`

Dynamisch Datenstrukturen

- Größe der Datenstruktur ist nicht fest definiert
- Speicherverbrauch kann sich zur Laufzeit ändern
- Python *alles*, C++ `std::vector`

Taxonomie der Datenstrukturen

Homogene Datenstrukturen

- Kann nur einen (abstrakten) Datentyp speichern
- Python `numpy array`, C++ `std::vector` oder `std::list`

Inhomogene Datenstrukturen

- Kann mehrere (abstrakte) Datentypen speichern
- Python `list`, C++17 `std::variant` oder `std::any` als Wrapper

Einzelen Datenstrukturen im Detail

(Array, Stack, Queue, List)

Array

(dt. manchmal auch Feld)

Arrays in C/C++

- Zusammenhängender Block an Speicher
- Jedes Element mit einzigartigem Index versehen
- Einzige in `C` standardmäßig enthaltene Datenstruktur

Beispiel

```c
int arr[10];

arr[0] = 42;
//arr[11] = 27; // out of bounds!

printf("%p\n", &arr[0]);
printf("%p\n", &arr[1]);
```

Ausgabe

- Aufeinanderfolgende Werte haben aufeinanderfolgende Adressen z.B. bei 16 Bit pro `int`:

```
0x70
0x80
0x90
...
```

Beispiel

- wir definieren ein Array mit 5 Elementen `char arr[5] = {0};`
- jedes Element ist ein 1-Byte großes `char`
- der Speicherbedarf beträgt 5 Byte
- der Computer sucht eine Adresse im Speicher von der an er die 5 Byte reservieren kann z.B. `0x70` und merkt sich, das ab hier nun die indizierten Elemente des Arrays liegen

```
Wert:       0x00  0x00  0x00  0x00  0x00
Addressen:  0x70  0x78  0x80  0x88  0x90
Index:      0     1     2     3     4
```

- `arr` selbst ist ein Zeiger auf das erste Element → hier `0x70`

 Aufgabe

- Bestimmen Sie mit ihrem Vorwissen wo ein C-Array in unserer Taxonomie vorkommt.
- Überlegen Sie auch welche Eigenschaften bzw. Operationen ein Array haben kann.

Array - Lösung der Aufgabe

Klassifikation

- linear
- statisch
- homogen

Operationen

- Weniger klar definiert als bei anderen Datenstrukturen
- Erzeugen eines Arrays: `np.array([1, 2, 3, 4, 5])`
- Auswahl eines Elements über Index: `arr[i] = 42;`
- Überprüfen auf Gleichheit: `arr1 == arr2`

Arrays - Python mit Numpy

- Grundlagen bereits aus den Einheiten zu "Scientific Computing" bekannt
- Wir wollen diese Arrays nun im Kontext der Datenstruktur betrachten

Beispiel `array_example.ipynb`:

- Erzeugen eines `numpy` Arrays
- Verändern des Datentyps
- "Vergrößern" eines Arrays
- Werte zuweisen
- Iterieren über das Array

List

(dt. Liste)

List

- Wird aus verketteten Elementen (Knoten) gebildet

Klassifikation

- linear*
- dynamisch*
- homogen*

Operationen

- Erzeugen einer leeren Liste: `my_list = List()`
- Element an bestimmter Stelle extrahieren:
  `elem = my_list.get(3)`
- Element an bestimmter Stelle einfügen: `my_list.insert(3, 42)`
- Element an bestimmter Stelle entfernen: `my_list.remove(3)`
- ...

* je nach Implementierung

Linked List (dt. verkettete Liste)

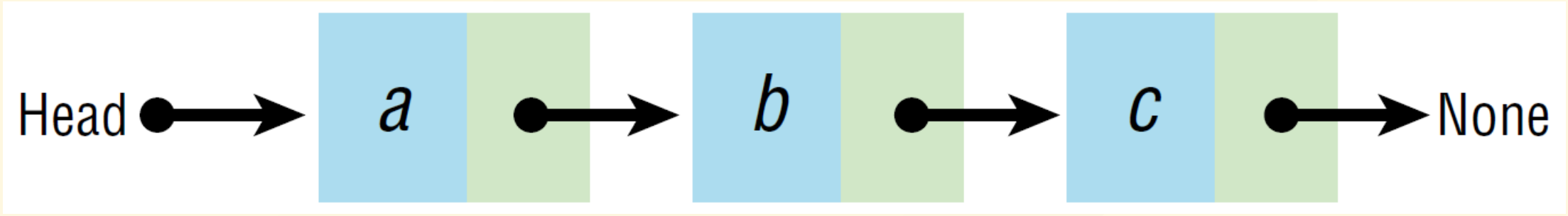


Bild: [Althoff 2021]

Einfach verketteten Liste (singly linked list)

- Besteht aus Knoten (engl. node) die Information (`data`) speichern
- Jeder Knoten verweist auf den nächsten Knoten (`next`) mittels eines Zeiger
- Der letzte Knoten verweist auf `None`
- Der erste Knoten wird `head` genannt

🤓 Doppelt verketteten Liste (doubly linked list)

- Jeder Knoten hat einen Zeiger auf den vorherigen Knoten (`prev`) und den nächsten Knoten (`next`)

Linked List

🤓 Sonderfälle

- Zirkulär verkettet Liste (circular linked list)
  - Der `next`-Zeiger des letzten Knotens verweist auf `head`
  - z.B. Scheduler für wiederkehrende Prozesse
- Doppelt zirkulär verkettete Liste (doubly circular linked list)
  - Kann in beliebiger Richtung durchschritten werden
  - z.B. Playlist in einem Musikplayer
- Mehrfach verkettete Liste (multiply linked list)
  - Jeder Knoten hat mehrere `next`-Zeiger
  - Eher von theoretischem Interesse

Linked List - Python

- Wir verwenden eine Singly Linked List implementieren
- Eine Beispielhafte Implementierung wird gezeigt, es sind aber viele weitere möglich

Beispiel `list_example.ipynb`:

- Definition einer Klasse für eine einfach verketteten Liste
- Abbilden der notwendigen Operationen
- Überladen der Klasse um sie mittels `__iter__()`-Dunder-Methode in Python nativ iterierbar zu machen

Stack

(dt. Stapelspeicher)

Stack

- LIFO (Last In First Out) Datenstruktur

Klassifikation

- linear
- dynamisch*
- homogen*

Operationen

- Erzeugen eines leeren Stapels: `stack = Stack()`
- Element auf den Stapel legen: `stack.push(32)`
- Oberstes Element von Stapel nehmen: `elem = stack.pop()`
- Lesen des obersten Elements: `elem = stack.peek()`
- Größe des Stacks feststellen: `amount = stack.size()`
- ...

* je nach Implementierung

Stack in C/C++

- Mit Hilfe eines Arrays & dynamischer Speicherverwaltung implementiert
- Nur die absolut notwendigen Operationen umgesetzt

Beispiel aus der LV Programmieren II

```c
typedef struct Stack{
  double *items;
  int count;
  int capacity;
} Stack;

void init_stack(Stack *stack){ /*siehe Prog2*/ }
void push_stack(Stack *stack, double item){
  if(stack->count >= stack->capacity){
    stack->capacity *= 2;
    stack->items = realloc(stack->items, sizeof(item) * stack->capacity);
  }
  stack->items[stack->count] = item;
  stack->count++;
}
void pop_stack(Stack *stack){ stack->count--; }
void free_stack(Stack *stack){ /*siehe Prog2*/ }

int main(){
  Stack stack = { NULL, 0, 0 };
  init_stack(&stack);

  for(int i = 0; i < 100; i++) push_stack(&stack, (rand() % 500) / 3.0);
  for(int i = 0; i < stack.count; i++) printf("%f\n", stack.items[i]);
  for(int i = 0; i < 80; i++) pop_stack(&stack);

  free_stack(&stack);
}
```

Stack - Python

- Werden für die Implementierung dieses ADT uns bereits bekannte `Python`-Datentypen (`list`) nutzen
- Eine Beispielhafte Implementierung wird gezeigt, es sind aber viele weitere möglich

Beispiel `stack_example.ipynb`:

- Definieren einer `Stack`-Klasse
- Erzeugen eines Stacks
- Elemente auf den Stack legen
- Elemente vom Stack entfernen
- Nutzen von `type hints` um Homogenität "einzuhalten"

Queue

(dt. Warteschlange)

Queue

- FIFO (First In First Out) Datenstruktur
- z.B. Abarbeiten einer Pipeline z.B. `queue_pipeline.py`

Klassifikation

- linear
- dynamisch*
- homogen*

Operationen

- Erzeugen einer leeren Queue: `queue = Queue()`
- Element hinten and die Queue anfügen: `queue.enqueue(32)`
- Vorderstes Element von Queue entfernen:
  `elem = queue.dequeue()`
- Größe des Queue feststellen: `amount = queue.size()`
- ...

* je nach Implementierung

Queue - Python

- Wir verwenden für die Implementierung dieses ADT wieder eine `Python-list`
- Eine Beispielhafte Implementierung wird gezeigt, es sind aber viele weitere möglich

🤓 Beispiel `queue_example.ipynb`:

- Definieren einer `Queue`-Klasse mit `Python-list`
- Erzeugen einer Queue
- Elemente hinzufügen/entfernen
- Definieren einer `Queue` über den built-in type

🏆 Hausübung

- Berechnen von arithmetischen Ausdrücken in Infixnotation mit Hilfe des `Dijkstra's Two Stack Algorithm`
- Dieser Algorithmus dient z.B. dafür Ausdrücke, wie sie in Taschenrechnern eingegeben werden, auszuwerten:
  `((5 + 3) * (5 + 1))`
- Activity-Diagramm für den Algorithmus auf nächster Folie

Dijkstra's Two Stack Algorithm