Algorithmen und Datenstrukturen

B1. Arrays und verkettete Listen

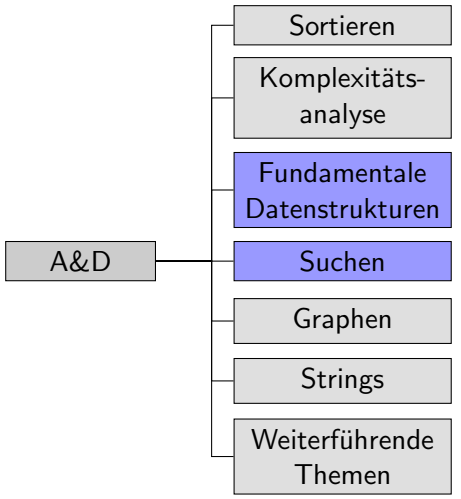
Marcel Lüthi and Gabriele Röger

Universität Basel

23. März 2023

B1.1 Übersicht

Übersicht



Ausblick : Fundamentale Datenstrukturen

- Datenabstraktion (Abstrakte Datentypen)
  - Multimengen, Stapel, (Prioritäts-) Warteschlangen
- Datenstrukturen
  - Arrays, Verkettete Listen, Bäume, Heaps

Höhepunkt: Heapsort

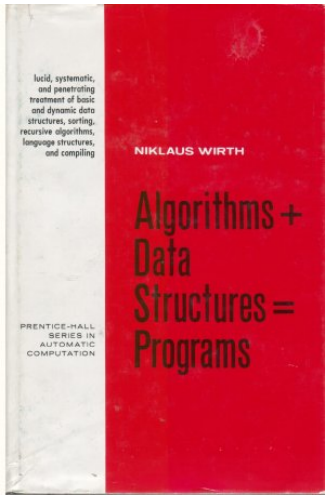
Elegantes Zusammenspiel Algorithmus und Datenstruktur.

- Clevere Datenstruktur - Simpler Algorithmus
- Garantiertes Laufzeitverhalten $O(n \log n)$

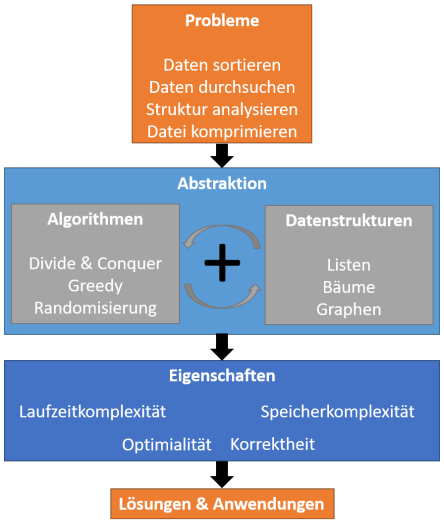
B1.2 Datenstrukturen

Datenstrukturen

- Programmieren ist mehr als Algorithmen schreiben
  - Datenorganisation ist zentral
- Elegante Datenstrukturen führen zu elegantem Code
- Programmierer
  - braucht Katalog von Datenstrukturen
  - muss Eigenschaften kennen



Übersicht



Datenstrukturen

> Bad programmers worry about the code. Good programmers worry about data structures and their relationships.
>
> Linus Torwalds

Datenstrukturen

> Show me your algorithm and conceal your datastructures, and I shall continue to be mystified. Show me your datastructures, and I won't usually need your algorithm; it will be obvious.
>
> Fred Brooks (paraphrased)

B1.3 Arrays

Die Datenstruktur Array (Feld)

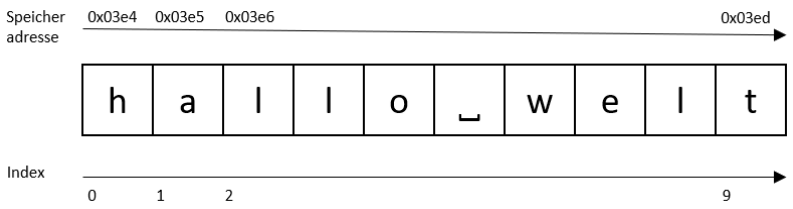
- Eine der grundlegenden Datenstrukturen, die sich in jeder Programmiersprache findet.
- Beschreibt eine Kollektion von fixer Grösse.

In Java:

```
Byte[] ia = new Byte[100];
String[] sa = new String[100];
```

Die Datenstruktur Array (Feld)

Array
Sequenz von Elementen die in gleichmässigen Abständen im Speicher angeordent sind.



Laufzeit grundlegender Operationen

- Was ist die Laufzeitkomplexität von folgenden Operationen (als Funktion der Arraygrösse n)?
  - `get(i)` Element an beliebiger Stelle i lesen
  - `set(i)` - Element an beliebiger Stelle i schreiben
  - `length()` - Länge von Array bestimmen.
  - `find(x)` - Element x finden und Index zurückliefern.
- Was ist die Speicherkomplexität?

Beobachtung
Komplexität direkte Konsequenz aus der Datenrepräsentation

Dynamische Arrays

Fixe Grösse ist für viele Anwendungen einschränkend

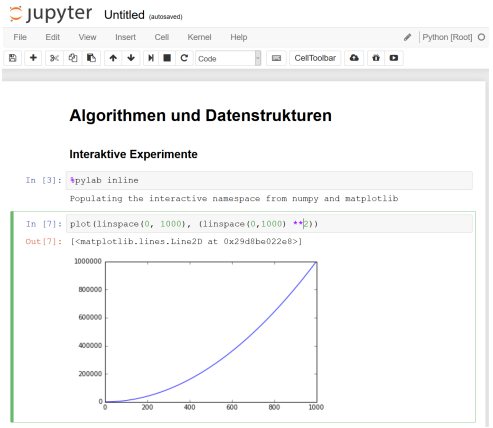
- Brauchen Arrays, die dynamisch wachsen können.
- Laufzeit Eigenschaften bestehender Methoden sollen gleich bleiben.

Zusätzliche Funktionen

- `append(x)` (manchmal push) - Element x ans Ende anfügen
- `insert(i, x)` - Element x an Stelle i einfügen
- `pop()` - letztes Element entfernen
- `remove(i)` - Element an position i löschen

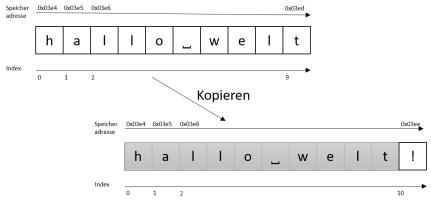
Was ist die Laufzeitkomplexität dieser Funktionen?

Experiment in Python



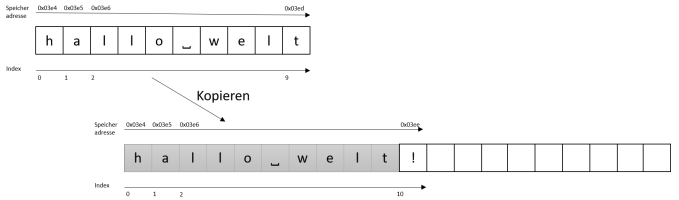
Arrays vergrössern / verkleinern : Naive Methode

- append (und insert) müssen Array vergrössern.
- pop muss Array verkleinern
- Naive Methode: Jeweils um 1 grösses/kleineres Array anlegen
  - Element in neues Array kopieren



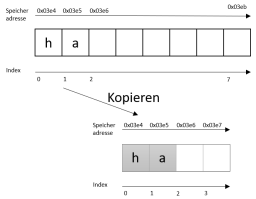
Arrays vergrössern : Schlauere Methode

- append (und insert) müssen Array vergrössern.
- Grösseres Array (von $2n$ Elementen) anlegen.
  - Array muss nur bei jeden n-ten Aufruf von append kopiert werden.



Arrays verkleinern : Schlauere Methode

- pop muss Array verkleinern
- Kleineres Array anlegen nur wenn Array zu $n/4$ gefüllt.
- In neues Array der Grösse $n/2$ kopieren.
  - Array muss nur bei jeden $n/4$-ten Aufruf von pop kopiert werden.



Implementation: Arrays vergrössern / verkleinern (1)

- Implementation der append und pop Methode.

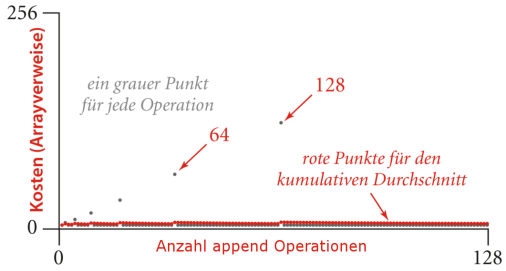
```
class Array:
  data = [None] # list simulates block of memory
  size = 0

  def append(self, elem):
      if len(self.data) == self.size:
          self.resize(len(self.data) * 2)
      self.data[self.size] = elem
      self.size  += 1

  def pop(self, elem):
    self.size -= 1
    item = self.data[self.size];
    if self.size > 0
       and self.size == len(self.data) / 4:
      self.resize(int(len(self.data) / 2));

    return item;
```

Implementation: Arrays vergrössern /verkleinern (2)

```
class Array:
  data = [None] # list simulates block of memory
  size = 0

  def append(self, elem):
    ...

  def pop(self, elem):
    ...

  def resize(self, numElements ):
    newArray = [None] * numElements
    for i in range(0, self.size):
        newArray[i] = self.data[i]
    self.data = newArray
```


Theoretische Analyse der append Operation

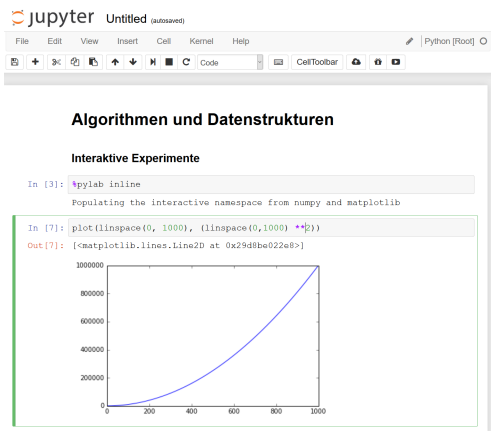
Die append Operation hat (amortisierte) Laufzeit $O(1)$



Quelle: Abbildung 1.28 - Algorithms, Sedgewick & Wayne

- Amortisierte Analyse: Mittlere Laufzeit pro Operation wird über Seqenz von N Operationen (im worst case) ermittelt.

Amortisierte Analyse



Jupyter-Notebook: arrays.ipynb

Analyse der append Operation: Beweisskizze

Annahmen:

- N ist Zweierpotenz.
- Wir starten mit Array der Grösse 1

Betrachte N aufeinanderfolgende Aufrufe von append. Wir haben folgende Anzahl Arrayzugriffe

$$N + 4 + 8 + 16 + \ldots + N + 2N$$

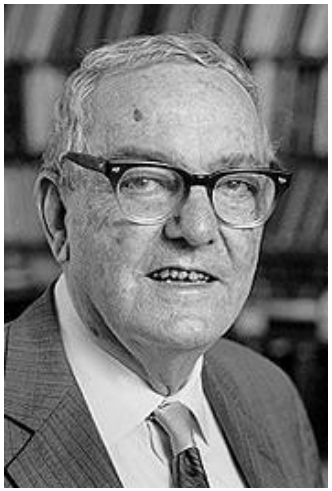
Wir nutzen, dass $\sum_{i=0}^{n} 2^i = 2^{n+1} - 1$

$N + 4 + 8 + 16 + \ldots + N + 2N \leq 3N + \sum_{i=0}^{\log_2 N} 2^i =$
$3N + 2^{(\log_2 N)+1} - 1 = 3N + 2 \cdot 2^{\log_2 N} - 1 \leq 5N$

Beobachtung: Kosten pro Aufruf von append sind konstant (< 5N Operationen für N Aufrufe)

B1.4 Verkettete Listen

Informatiker des Tages



Herbert Simon (Ökonom)

- Nobelpreisträger und Gewinner des Turing Awards
- Pionier in künstlicher Intelligenz
- „Erfinder “der verketteten Liste (im Rahmen der IPL Sprache).

Newell, Allen, and Fred M. Tonge. An introduction to information processing language V. Communications of the ACM (1960).

Motivation

- Arrays sind nicht flexibel genug
- Brauchen immer grossen, kontinuierlichen Block an Speicher
- Einfügen von Elementen an beliebiger Position ist teuer

Lösung muss uns erlauben Elemente im Speicher zu verteilen.

Frage?

- Wie kann man Elemente ordnen die verteilt im Speicher sind?

not

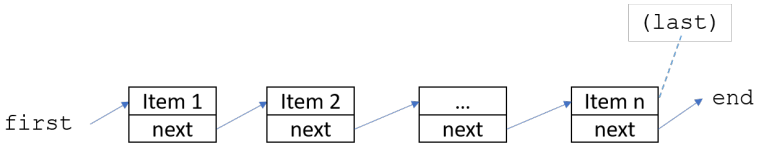
to

or

be

Verkettete Listen

- Wichtige, flexible Datenstruktur
- Jeder Knoten speichert sein Datum, sowie eine Referenz (Zeiger) auf Nachfolger
- Ende muss speziell gekennzeichnet werden (häufig null/None).
- ... oder wir brauchen Referenz auf letztes Element



Quiz: Komplexität Array / Verkettete Liste

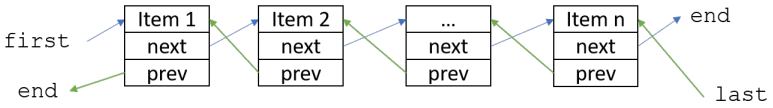
Operation	Array	Verkettete Liste
Zugriff auf beliebiges Element		
Einfügen, Löschen am Anfang		
Einfügen am Ende		
Löschen am Ende		
Einfügen, Löschen in Mitte		
Verschwendeter Speicher		

Take-home Message

- Verschiedene Datenstrukturen machen verschiedene Trade-offs

Doppelt verkettete Liste

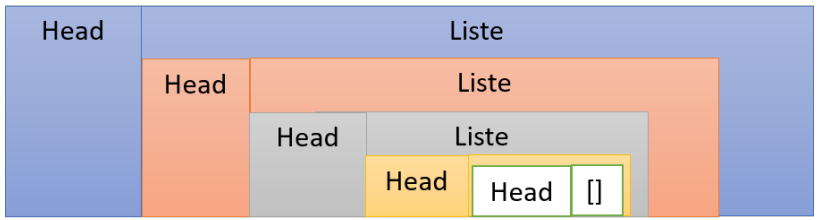
- Referenz nicht nur auf Nachfolger, sondern auch vorhergehendes Element
- Macht Entfernen vom Ende günstig.



Rekursive Definition

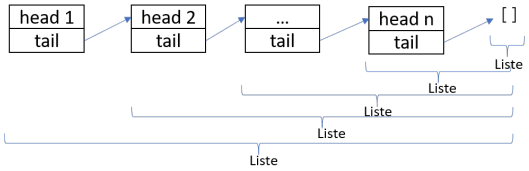
Eine Liste L ist

- die leere Liste
- oder ein Element H (Head) gefolgt von einer Liste: H, L



Verkettete Listen: Datenstruktur (rekursiv)

```
class List[Item]:
  head : Item
  tail : List[Item]
  List(head : Item, tail : List[Item]) # Konstruktor

emptyList = List(None, None)
```



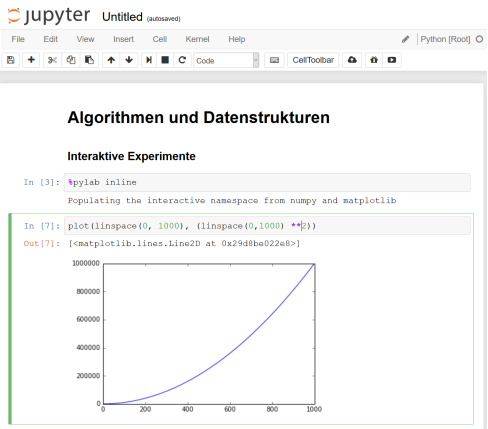
Verkettete Listen: Datenstruktur (rekursiv)

```
class List[Item]:
  head : Item
  tail : List[Item]
  List(head : Item, tail : List[Item]) # Konstruktor

emptyList = List(None, None)
```

Vergleiche:

```
class Node[Item]:
  item : Item
  next : Node
  Node(head : Item, tail : Node[Item])  # Konstruktor
```


Implementation in Python



Jupyter-Notebook: linked-lists.ipynb