

Algorithmen und Datenstrukturen

B2. Arrays & Verkettete Listen

Marcel Lüthi and Gabriele Röger

Universität Basel

20. März 2019

Algorithmen und Datenstrukturen

20. März 2019 — B2. Arrays & Verkettete Listen

B2.1 Arrays

B2.2 Verkettete Listen

B2.1 Arrays

Die Datenstruktur Array (Feld)

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

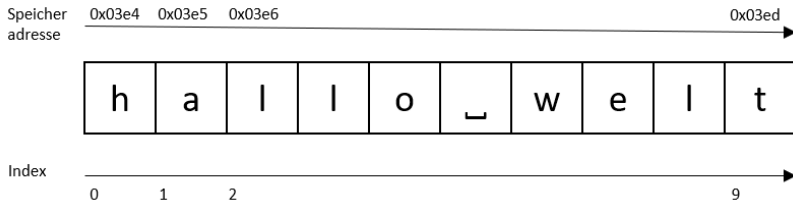
In Java:

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

Die Datenstruktur Array (Feld)

Array

Sequenz von Elementen die in gleichmässigen Abständen im Speicher angeordnet 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

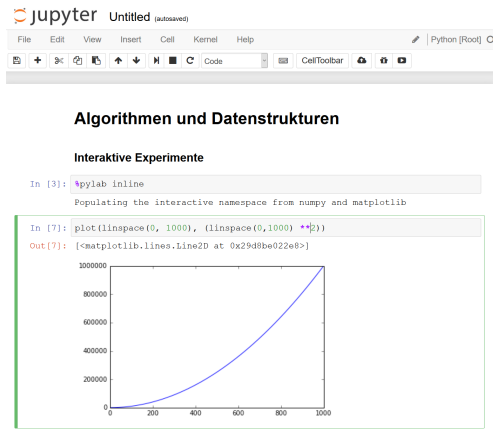
- ▶ Brauchen Arrays, die dynamisch wachsen können.
- ▶ Laufzeit Eigenschaften bestehender Methoden sollen gleich bleiben.

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?

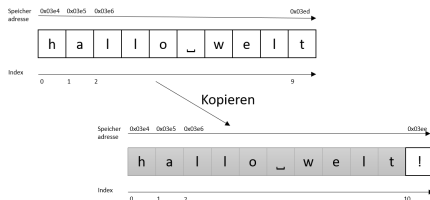
Empirische Laufzeitanalyse, Python Arrays



IPython Notebook: Arrays-und-linked-lists.ipynb

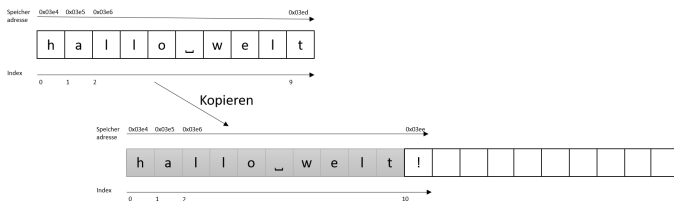
Arrays vergrössern / verkleinern : Naive Methode

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



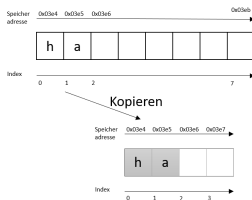
Arrays vergrössern : Schlauere Methode

- ▶ 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
    _lastIdx = 0

    def append(self, elem):
        if len(self._data) == self._lastIdx:
            self._resize(len(self._data) * 2)
        self._data[self._lastIdx] = elem
        self._lastIdx += 1

    def pop(self, elem):
        self._lastIdx -= 1
        item = self._data[self._lastIdx];
        if self._lastIdx > 0
            and self._lastIdx == 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
    _lastIdx = 0

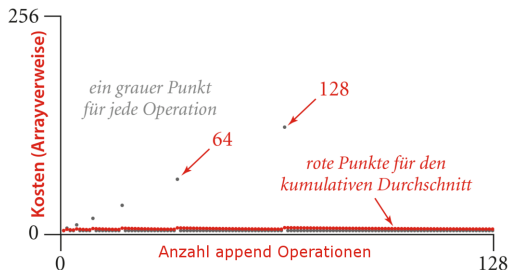
    def append(self, elem):
        ...

    def pop(self, elem):
        ...

    def _resize(self, numElements ):
        newArray = [None] * numElements
        for i in range(0, self._lastIdx):
            newArray[i] = self._data[i]
        self._data = newArray
```

Theoretische Analyse der append Operation

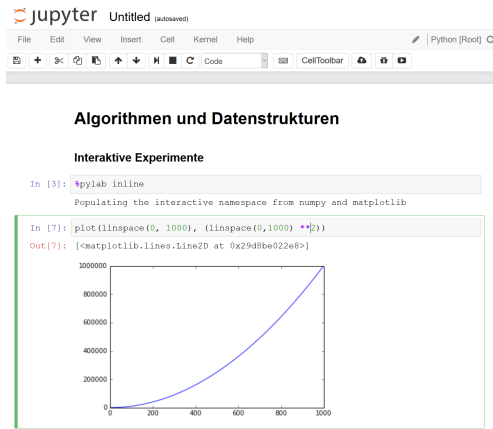
Die append Operation hat (amortisierte) Laufzeit $O(1)$



Quelle: Abbildung 1.28 - Algorithms, Sedgewick & Wayne

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

Amortisierte Analyse



IPython Notebook: Arrays-und-linked-lists.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

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

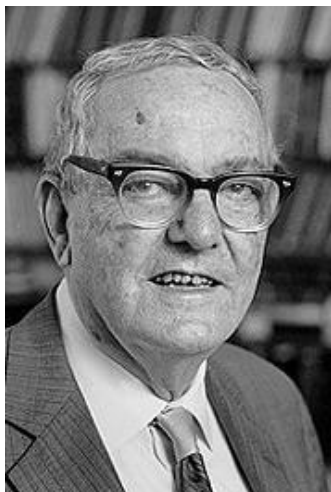
Wir nutzen, dass $\sum_{i=0}^n 2^i = 2^{n+1} - 1$

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

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

B2.2 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?

to

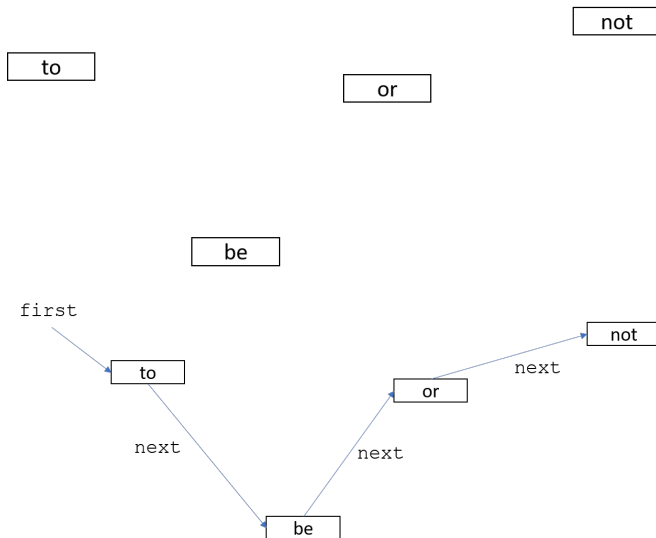
or

not

be

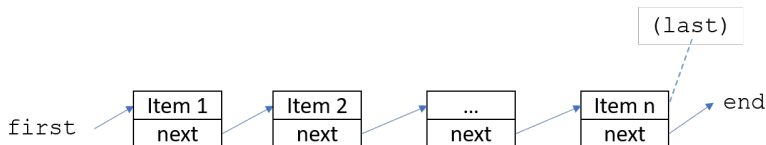
Frage?

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



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

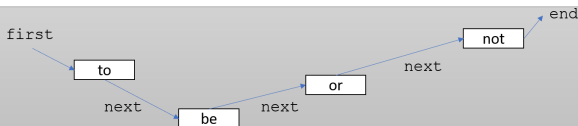
Operation	Array	Verkettete Liste
Zugriff auf beliebiges Element	$O(1)$	$O(n)$
Einfügen, Löschen am Anfang	$O(n)$	$O(1)$
Einfügen am Ende	$O(1)$ (ammortisiert)	$O(1)$
Löschen am Ende	$O(1)$ (ammortisiert)	$O(n)$
Einfügen, Löschen in Mitte	$O(n)$	$O(n)$
Verschwendeter Speicher	$O(1)$	$O(n)$

Take-home Message

- ▶ Verschiedene Datenstrukturen machen verschiedene Trade-offs

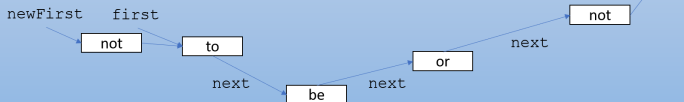
Einfügen am Anfang

Ausgangssituation



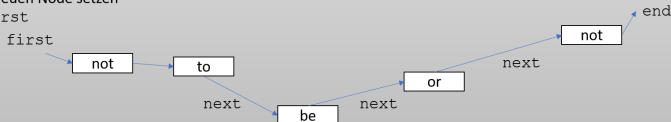
Neuer Node mit Referenz auf Anfang

```
newFirst = Node("not", first)
```



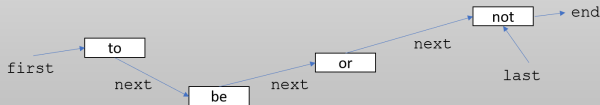
First Referenz auf neuen Node setzen

```
first = newFirst
```



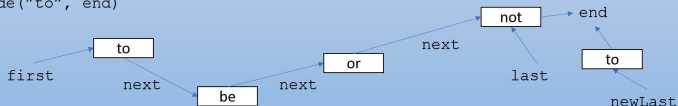
Einfügen am Ende

Ausgangssituation



Neuer Node mit Referenz auf Ende

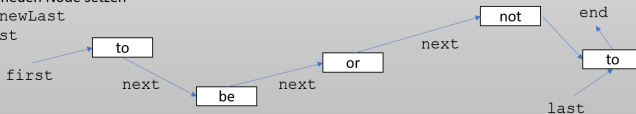
`newLast = Node("to", end)`



First Referenz auf neuen Node setzen

`last.next = newLast`

`last = newLast`



Weitere Operationen

Einfach:

- ▶ Vom Anfang entfernen
- ▶ Traversieren

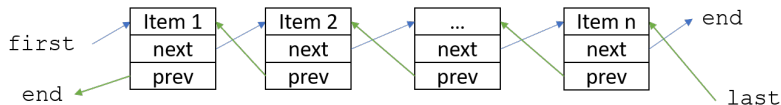
Schwierig:

- ▶ Vom Ende entfernen
- ▶ An beliebiger Position einfügen
- ▶ An beliebiger Position entfernen
- ▶ Element an beliebiger Position lesen/schreiben

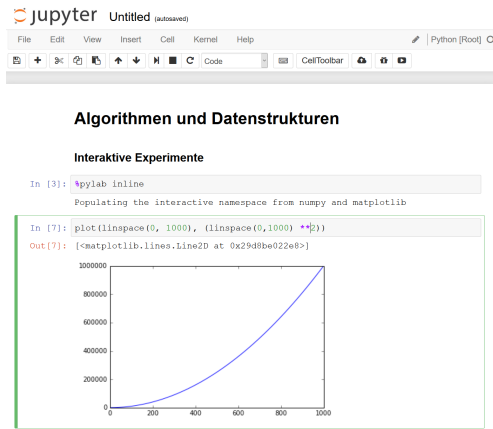
Einfach/Schwierig bezieht sich auf Aufwand und nicht Implementation.

Doppelt verkettete Liste

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



Implementation in Python

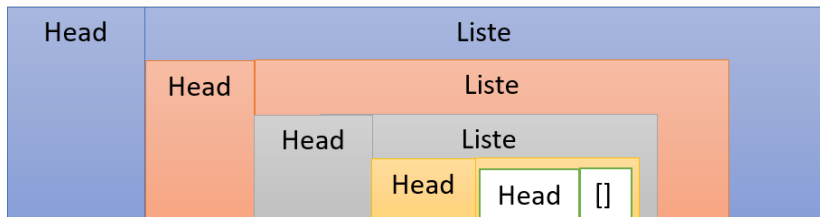


IPython Notebook: Arrays-und-linked-lists.ipynb

Rekursive Definition

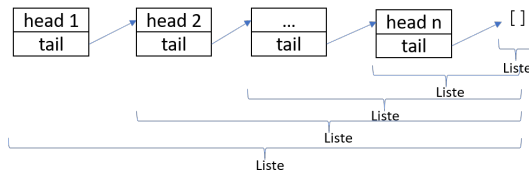
Eine Liste L ist

- ▶ 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:

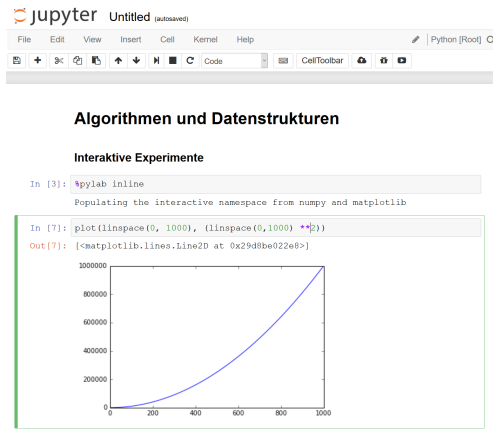
```
class Node[Item]:  
  item : Item  
  next : Node  
  Node(head : Item, tail : Node[Item]) # Konstruktor
```

Verkettete Listen (rekursiv)

- ▶ Natürliche, rekursive Implementation vieler Operationen
- ▶ Implementation folgt Datenstruktur

```
def printList(list):  
    if (list == emptyList):  
        return ""  
    else:  
        return str(list.head) + printList(list.tail)
```


Implementation in Python



IPython Notebook: Arrays-und-linked-lists.ipynb