Datenstrukturen

Theorie

Version vom 21. Oktober 2018

1 Datentypen

Der Datentyp besagt ...

- welchen Wertebereich die Daten haben,
- welche Operationen auf den Daten zulässig sind.

1.1 Einfache Datentypen

Ein einfacher Datentyp (Synonyme: elementarer oder primitiver Datentyp) besteht aus einem einzelnen Wert.

In Python sind das (unter anderem):

- ganze Zahlen (integer): `+`, `-`, `*`, `//`, ...
- Gleitkommazahlen (float): `+`, `-`, `*`, `/`, ...
- Wahrheitswerte (boolean): `not`, `and`, `or`

1.2 Zusammengesetzte Datentypen

Zusammengesetzte Datentypen sind Zusammenfassungen (Datenstrukturen) aus einfachen Datentypen.

Im folgenden werden einige der in Python bereits vorhandenen zusammengesetzten Datentypen wiederholt.

Listen

Eine Liste ist eine geordnete Menge von null oder mehr Referenzen auf Python-Daten.

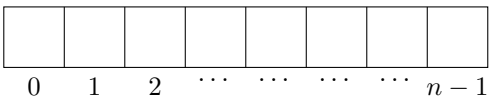


Abbildung 1.1: Modell einer Liste

Operation	Effizienz
`L[i]`	$O(1)$
`L[3]=25`	$O(1)$
`L.append(item)`	$O(1)$
`L.pop()`	$O(1)$
`L.pop(i)`	$O(n)$
`L[i:j]`	$O(k)$
`L1+L2`	$O(k)$
`L.del(i)`	$O(n)$
`L.insert(i,item)`	$O(n)$
`x in L`	$O(n)$
`for item in L`	$O(n)$
`L.reverse`	$O(n)$
`L.sort()`	$O(n \log n)$

Tabelle 1.1: Effizienz einiger Funktionen und Methoden für Listen

Tupel

Tupel sind unveränderliche Listen. Das heisst, dass von den Listenmethoden nur diejenigen anwendbar sind, welche das Tupel und ihre Elemente nicht modifizieren.

Bei der literalen Eingabe werden Tupel durch runde anstelle von eckigen Klammern definiert.

Zeichenketten

Zeichenketten (strings) sind Folgen aus null oder mehr Symbolen (Zeichen).

Bei der literalen Eingabe werden Zeichenketten durch einfache oder doppelte Anführungszeichen gekennzeichnet.

Strings verhalten sich wie Tupel, deren Elemente die einzelnen Symbole sind. Die Methoden zum Sortieren, Löschen oder Einfügen sind daher nicht unmittelbar anwendbar.

Sets

Eine Menge ist eine ungeordnete Sammlung unveränderlicher Python-Objekte ohne Duplikate. Die Mengen werden literal als komma-separierte Listen dargestellt, die von geschweiften Klammern eingeschlossen sind.

`A = {1, 'Hallo', 3.14, True}`

`B = set()` (die leere Menge)

Beispiel	Erklärung
`x in M`	Ist `x` ein Element von `M`?
`len(M)`	Anzahl der Elemente in der Menge `M`
`A | B`	Vereinigungsmenge $A \cup B$
`A & B`	Schnittmenge $A \cap B$
`A - B`	Mengendifferenz $A \setminus B$
`A <= B`	Teilmenge $A \subset B$?

Tabelle 1.2: Operationen für Sets

Beispiel	Erklärung
`A.union(B)`	$A \cup B$
`A.intersection(B)`	$A \cap B$
`A.difference(B)`	$A \setminus B$
`A.issubset(B)`	$A \subset B$
`A.add(item)`	Füge `item` zur Menge `A` hinzu.
`A.remove(item)`	Entferne `item` aus Menge `A`.
`A.pop()`	Entferne beliebiges Element aus Menge `A`.
`A.clear()`	Entferne alle Element aus Menge `A`.

Tabelle 1.3: Methoden für Sets

Dictionaries (Assoziative Listen)

Eine Dictionary ist eine ungeordnete Sammlung von Schlüssel-Wert Paaren (key-value pair).

Dictionaries werden literal als komma-separierte Listen der Schlüssel-Wert-Paare dargestellt, die von geschweiften Klammern eingeschlossen sind. Die Schlüssel-Wert Paare selbst werden durch einen Doppelpunkt getrennt.

Operation	Effizienz
`D[key]`	$O(1)$
`D[key]=25`	$O(1)$
`D.del(key)`	$O(1)$
`key in D`	$O(1)$
`for key in D`	$O(n)$
`D.copy()`	$O(n)$

Tabelle 1.4: Effizienz einiger Funktionen und Methoden für Dictionaries

1.3 Abstrakte Datentypen

Schnittstellen

Autofahrer müssen nicht unbedingt etwas von Motoren, Getriebe, oder Bremsen verstehen, um ein Auto fahren zu können. Gas- und Bremspedal, ein Getriebe und eine Zündung sorgen dafür, dass auch Nicht-Ingenieure ein Auto bedienen können.

Ähnlich verhält es sich beim Informatikanwender. Er muss nichts über Variablen, Schleifen oder Unicode wissen, um Dokumente zu schreiben, Mails zu verschicken oder im Web zu surfen (auch wenn es hilfreich ist).

In beiden Beispielen muss der Benutzer (*Client*) einer Abstraktion nichts von den darunter liegenden Details wissen, so lange er die *Schnittstelle* (Steuerrad, Bremspedal, Browser, Textverarbeitungsprogramm) versteht und bedienen kann.

Im Zusammenhang mit Datentypen wollen wir den Begriff der Schnittstelle noch weiter einschränken:

Unter einer Schnittstelle versteht man eine Beschreibung aller Operationen (Methoden), mit denen auf eine Sammlung von Daten zugegriffen werden kann.

In vielen Fällen wünschen sich Programmierer für eine Aufgabe eine massgeschneiderte Datenstruktur, die das Lösen der Aufgabe möglichst einfach macht.

Daher schafft man sogenannte *Abstrakte Datentypen* (*abstract data type, ADT*), welche eine logische Sicht auf die Daten erlaubt, die nicht unbedingt mit der physikalischen Darstellung der Daten übereinstimmen muss.

Ein abstrakter Datentyp (ADT) ist eine Sammlung von Objekten sowie eine Beschreibung der zulässigen Operationen, die darauf zugreifen.

Kapselung und Information Hiding

Bei der Beschreibung der Operationen eines abstrakten Datentyps sind zwei Dinge wesentlich:

- *Kapselung:* Der Zugriff auf die Operation darf nur über die Abstraktion der Schnittstelle erfolgen.
- *Information Hiding:* Die Details der Implementierung (Variablen) müssen vor dem Client verborgen werden. Bei einer Änderung der Implementierung könnten diesese Details nicht mehr gültig sein und zu Fehlern führen.

Durch stabile Schnittstellen können komplexe Softwaresysteme verbessert werden, ohne das gesamte System verändern zu müssen.

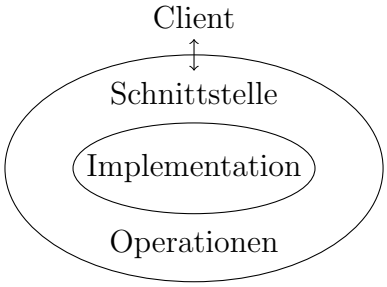


Abbildung 1.2: Abstrakter Datentyp

2 Stacks (Stapel)

- Last In First Out (LIFO)
- Hinzufügen (push) und Entfernen (pop) erfolgen von derselben Seite
- Anwendung: Browser-History; Undo-Funktion von Anwendungsprogammen; Auswertung von Postifix-Ausdrücken und Übersetzung von Infix-Ausdrücken in Postfix-Form; Backtracking-Algorithmen; Verwaltung des Arbeitsspeichers von Computern

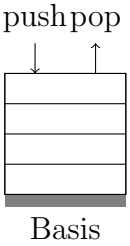


Abbildung 2.1: Modell eines Stacks

Python-Implementation eines Stacks

Die Klasse `Stack` wird als Python-Liste implementiert. Die Methoden sind:

- Der Konstruktor `Stack()` erzeugt einen leeren Stack.
- `push(item)` legt `item` oben auf den Stack.
- `pop()` entfernt das oberste Element vom Stack und liefert es als Wert zurück.
- `peek()` liefert das oberste Element des Stacks als Wert zurück ohne es zu entfernen.
- `isEmpty()` liefert `True` bzw. `False` als Rückgabewert, wenn der Stack leer bzw. nicht-leer ist.
- `s.size()` liefert die Anzahl Elemente des Stacks zurück.

Anwendung

Schreibe eine Funktion `check(string)`, das einen String als Argument entgegen nimmt und überprüft, ob die (runden) Klammern in diesem String korrekt gesetzt sind.

Lösungsidee: Erzeuge einen leeren Stack `s`. Durchlaufe die Zeichenkette zeichenweise. Ist das Symbol eine öffnende Klammer, kommt es auf den Stack. Ist das Symbol eine schliessenden Klammer, so entferne das oberste Stackelement. Überprüfe, ob diese Operation auf einem leeren Stack ausgeführt wird. Wenn ja, gibt es mehr schliessende als öffende Klammern und der Ausdruck ist falsch. Nachdem der gesamte String verarbeitet wurde, ist noch zu prüfen, ob noch Klammern auf dem Stack liegen. Wenn ja, dann gibt es mehr öffnende als schliessende Klammern und der Ausdruck ist falsch.

3 Queues (Warteschlangen)

- First In First Out (FIFO)
- Hinzufügen (enqueue) und Entfernen (dequeue) erfolgen an entgegengesetzten Seiten
- Beispiele: Druckerwarteschlangen; Warteschlange von Computerprozessen, Simulationen (Strassenverkehr, Supermarkt, ...)

Abbildung 3.1: Modell einer Queue

Python-Implementation einer Queue

Die Klasse `Queue` soll als Python-Liste implementiert werden. Die Methoden sind:

- Der Konstruktor `Queue()` erzeugt eine leere Queue.
- `enqueue(item)` fügt `item` am Ende der Queue ein.
- `dequeue()` entfernt das vorderste Element der Queue.
- `isEmpty()` liefert `True` bzw. `False` als Rückgabewert, wenn die Queue leer bzw. nicht-leer ist.
- `s.size()` liefert die Anzahl Elemente der Queue zurück.

Anwendung: Gesellschaftsspiel-Simulation

Bei dem in den USA verbreiteten Kinderspiel *Hot Potato* geht es darum, dass die im Kreis sitzenden Teilnehmer einen Gegenstand (die heisse Kartoffel) weiterreichen. Derjenige, der nach Ablauf einer (zufälligen) Frist, den Gegenstand in der Hand hält, scheidet aus.

Schreibe eine Funktion `hotPotatoe(namelist, k)` mit einer Namensliste und einer ganzen Zahl `k` als Argument.

Die Namen werden in eine Warteschlange eingefüllt. In jeder Runde wird die Warteschlange um k Positionen zyklisch vertauscht und derjenige Name, der danach ganz vorne in der Schlange steht, entfernt. Am Ende gibt das Programm den Namen der Person aus, die alle Runden überstanden hat.

4 Deques (zweiseitige Warteschlangen)

Deque [ausgesprochen „Deck“] steht für Double-ended queue.

Abbildung 4.1: Modell einer Deque

- Hinzufügen (add) und Entfernen (remove) erfolgen jeweils an beiden Seiten
- Anwendungen: Textsuche mittels regulärer Ausdrücke, nichtdeterministische endliche Automaten

Python-Implementation einer Deque

Die Klasse `Deque` soll auf der Basis einer Python-Liste implementiert werden. Die Methoden sind:

- Der Konstruktor `Deque()` erzeugt eine leere Deque.
- `addFront(item)` fügt `item` am vorderen Ende ein.
- `addRear(item)` fügt `item` am hinteren Ende ein.
- `removeFront(item)` entfernt `item` vom vorderen Ende.
- `removeRear(item)` entfernt `item` vom hinteren Ende.
- `isEmpty()` liefert `True` bzw. `False`, wenn die Deque leer bzw. nichtleer ist.
- `s.size()` liefert die Anzahl Elemente der Queue zurück.

Anwendung: Palindrome erkennen

- Ein Palindrom-Kandidat wird zeichenweise in eine Deque „geschoben“.
- So lange die Deque mehr als ein Element besitzt, wird von beiden Seiten jeweils ein Element entfernt. Sind die beiden Elemente verschieden, liefert die Funktion den Wert `False` zurück.
- Hat die Deque nur noch ein Element oder ist sie leer, muss es sich um ein Palindrom handeln und die Funktion liefert `True` zurück.

5 Linked Lists (einfach verkettete Listen)

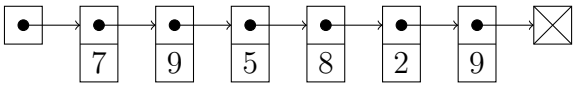


Abbildung 5.1: Modell einer Linked List

- Die einfach verkettete Liste wird durch einen einzelnen Zeiger repräsentiert. (Punkt ganz links)
- Dieser Zeiger zeigt auf den ersten Knoten (Rechteck). Ein Knoten besteht aus einem Zeiger (Punkt) und einem Datenfeld.
- Der Zeiger eines Knotens zeigt entweder auf einen weiteren Knoten oder auf das Ende der verketteten Liste, das durch den leeren Wert (Quadrat mit Kreuz) gekennzeichnet ist.

Die Klasse `Node`

Die Knoten werden als Objekte der Klasse `Node` implementiert.

`Node`
`data: <any>` `next: Node`

`Node(data: <any>)`
`getData(): <any>`
`getNext(): Node`
`setData(newData: <any>)`
`getNext(newNext: Node)`

- Der Konstruktor `Node(data)` hat als Argument einen Datenwert, mit dem die Instanzvariable `data` initalisiert wird. Die zweite Instanzvariable `next` wird standardmässig mit `None` initalisiert; zeigt also vorerst auf keinen anderen Knoten.
- `getData()` und `getNext()` liefern bei ihrem Aufruf die Wert der jeweiligen Instanzvariablen zurück.
- Mit `setData(newData)` und `setNext(newNext)` werden den Instanzvariablen neue Werte zugewiesen.

In der objektorientierten Programmierung werden oft Methoden geschrieben, um auf Instanzvariablen zuzugreifen (*Setter- und Getter-Methoden*). Zweck: Der Client soll ausschliesslich Methoden der Schnittstelle verwenden und nicht in den internen Details der Klasse „herumpfuschen“.

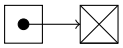
Die Klasse `Linkedlist`

`Linkedlist`
`head: Node`

`add(item: <any>)`
`isEmpty(): bool`
`length(): int`
`search(item: <any>) bool`
`remove(item: <any>)`

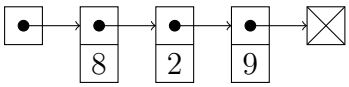
Der Konstruktor

Es wird eine Instanzvariable `head` erzeugt der mit `None` initialisiert wird. Sobald später der erste Knoten hinzugefügt wird, ersetzt man Node durch den Wert (im Grunde die Adresse) des Knotes.

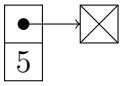


Die Methode `add(item)`

Möchte man der einfach verketteten Liste

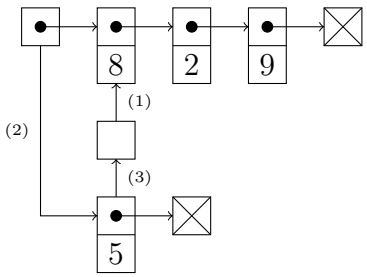


ein Element hinzufügen, so erfolgt dies effizient am Kopf der Liste. Dazu erzeugt man einen Knoten mit dem zu speichernden Wert.



Dann geht man wie folgt vor:

1. Die Referenz auf den bisherigen ersten Knoten in einer Hilfsvariable speichern.
2. Der Variable `head` die Referenz auf den neuen Knoten zuweisen.
3. Der Variable `next` den Wert der Hilfsvariable zuweisen, um den ersten Knoten mit den restlichen zu verbinden.



Die Methode `length()`

Man beginnt bei `head` und geht dann mit einer Schleife von Referenz zu Referenz, bis man auf `None` stösst, also das Ende der Liste erreicht hat.

Erhöht man bei jedem Durchlauf einen Zähler um 1, so erhält man am Ende die Länge der Datenstruktur.

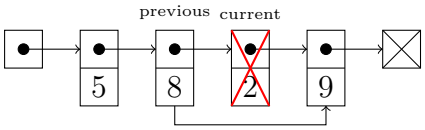
Die Methode `search(item)`

Diese Methode ist ähnlich wie `length` aufgebaut. Zusätzlich überprüft man bei jedem Schleifendurchlauf, ob sich `item` im Knoten befindet. Wenn ja. liefert die Methode frühzeitig `True` als Rückgabewert.

Erricht die Schleife das Ende der einfach verketteten Liste, ohne `item` gefunden zu haben, so liefert die Methode `False` zurück.

Die Methode `remove(item)`

Um den Knoten mit dem Datenwert `item` aus der Linkedlist zu entfernen muss man ihn wie in `search(item)` zuerst finden. Das Problem dabei ist, dass die Referenz auf den zu löschenden Knoten beim Erreichen des gesuchten Knotes bereits übersprungen wurde; also nicht mehr zur Verfügung steht, um auf den (allfälligen) übernächsten Knoten zu zeigen.



Die Lösung besteht darin, beim Traversieren (Durchlaufen) der einfach verketteten Liste jeweils eine Referenz auf den aktuellen *und* den davor liegenden Knoten zu speichern.

Man beachte, dass der Listenkopf `head` keinen Vorgänger hat, weshalb man seinen Vorgängerknoten mit `None` initialisieren sollte.

Weiter gilt es zu beachten, dass das Entfernen des ersten Knotens getrennt von dem Entfernen eines „inneren“ Knotens behandelt werden muss.

Muss man auch das Entfernen des letzten Knotens separat behandeln?

Bemerkung

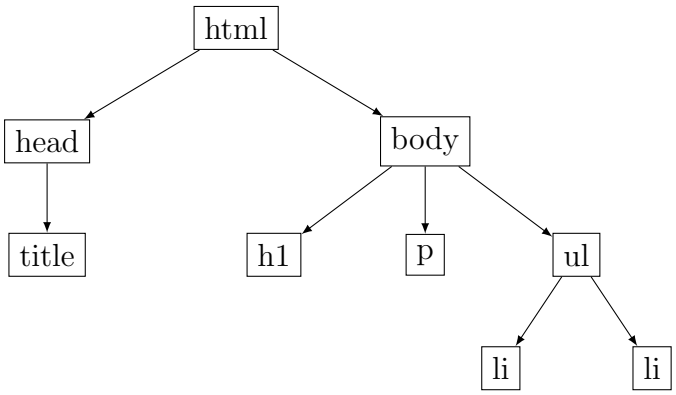
Das Entfernen eines Elements, hinterlässt einen verwaisten Knoten. In vielen Programmiersprachen muss der Programmierer dieses Objekt explizit mit einem sogenannten *Destruktor* löschen, wenn er nicht Speicherplatz verschwenden will.

Auch Python kennt eine `del()` Methode für Objekte. Diese wird in der Regel aber nicht benötigt, da ein spezieller Mechanismus, der *Garbage Collection* genannt wird, von Zeit zu Zeit dafür sorgt, dass nicht mehr verwendete Speicherbereiche wieder für neue Werte freigegeben werden.

6 Bäume (Trees)

Beispiel

```
<html>
  <head>
    <title>Simple HTML-Document</title>
  </head>
  <body>
    <h1>Caption</h1>
    <p>Paragraph</p>
    <ul>
      <li>Item 1</li>
      <li>Item 2</li>
    </ul>
  </body>
</html>
```



Begriffe

Knoten (node): Fundamentaler Baustein eines Baums. Ein Knoten kann eine zusätzliche Nutzdaten enthalten.

Kante (edge): Eine Kante verbindet zwei Knoten.

Wurzel (root): Der einzige Knoten ohne eingehende Kante.

Pfad (path): Eine geordnete Liste von Knoten, die durch Kanten verbunden sind.

Kinder (children): Menge der Knoten, die vom gleichen Knoten eine eingehenden Kante haben.

Eltern (parent): Derjenige Knoten, der alle Knoten

Geschwister (sibling): Menge der Knoten, die Kinder des gleichen Elternknotens sind.

Teilbaum (subtree): Die Menge der Knoten und Kanten, die aus einem Elternknoten und all dessen Kindern und Kindeskindern besteht.

Blatt (leaf node): Ein Knoten, der keine Kindknoten (oder: ausgehende Kanten) hat.

Level eines Knotens (level): Die Anzahl der Kanten auf dem Pfad vom Wurzelknoten zum betreffenden Knoten. Der Wurzelknoten hat den Level 0.

Höhe eines Baums (height): Das Maximum über die Menge der Levels aller möglichen Knoten.

Binärbaum (binary tree): Ein Baum, dessen Knoten maximal zwei Kinder haben.

Wald (forest): Eine Menge von Bäumen.

Bemerkung

In der Informatik haben Bäume normalerweise *gerichtete* Kanten. In der Graphentheorie (einem Teilgebiet der Mathematik) können Bäume aber auch ungerichtet sein.

Implementation eines Binärbaums

- `BinaryTree()` erzeugt eine neue Instanz eines Binärbaums
- `getLeftChild()` liefert den Binärbaum zurück, der sich im linken Kindknoten des aktuellen Knotens befindet.
- `getRightChild()` liefert den Binärbaum zurück, der sich im rechten Kindknoten des aktuellen Knotens befindet.
- `setRootValue(val)` speichert das Objekt in `val` im aktuellen Knoten.
- `getRootValue()` liefert das Objekt im aktuellen Knoten zurück.
- `insertLeft(val)` erzeugt einen neuen Binärbaum und richtet ihn im linken Kindknoten des aktuellen Baums ein.
- `insertRight(val)` erzeugt einen neuen Binärbaum und richtet ihn im rechten Kindknoten des aktuellen Baums ein.

7 Graphen

Graphen stellen eine Verallgemeinerung von Bäumen dar. Graphen können beispielsweise dazu verwendet werden, komplexe Strukturen wie Verkehrswege oder Beziehungen zwischen Personen abstrakt darzustellen.

Begriffe

Ein Graph besteht aus einer Menge von Knoten (*nodes* oder *vertices*) und einer Menge von Kanten (*edges*), die jeweils bestimmte Paare von Knoten miteinander verbinden.

- *Knoten (node):* Fundamentaler Bestandteil eines Graphen, der eine „Entität“ darstellt. Knoten können zusätzlich Nutzdaten enthalten.
- *Kante (vertex):* Fundamentaler Bestandteil eines Graphen, der eine „Beziehung“ zwischen zwei Entitäten“ herstellt. Abhängig davon, ob die Beziehung eine Richtung hat oder nicht spricht man von *gerichteten* oder *ungerichteten* Graphen.
- *Gewicht (weight):* Einer Kante kann ein Gewicht zugeordnet werden, das beispielsweise die Länge eines Weges oder die Kosten für den Transport einer Nachricht darstellt.

Anwendungen

- Travelling Salesman Problem (TSP)
- Kürzeste Wege (Shortest Path)
- Minimaler Spannbaum
- Topologisches Sortieren
- Auffinden starker Zusammenhangskomponenten
- Auffinden maximaler Cliquen
- ...