Stacks and Queues

1. Motivation

Es gibt verschiedene Anwendungsfälle für das Sammeln von Elementen in einer Datenstruktur. *Lists*, *Sets* und *Dictionaries* eignen sich, wenn Elemente zusammengefasst werden müssen um sie gemeinsam zur weiteren Bearbeitung an Funktionen weiterzugeben, Information zu aggregieren, etc.

Bei einer anderen Art von Anwendungsfällen geht es darum, Elemente einzeln für spätere Verarbeitung bereitzuhalten, quasi in einer *Warteschlange*. Die wichtigsten Datenstrukturen dafür sind *Stack* und *Queue*. Deren Funktionalität lässt sich unabhängig von der Implementierung verstehen und verwenden als sogenannte *abstrakte Datentypen*.

Python stellt, wie viele andere Programmierumgebungen auch, eine spezielle Datenstruktur *Double-Ended Queue* (*deque*) für verschiedene Warteschlangen-Anwendungen bereit. Diese sind darauf optimiert, an beiden Enden mit konstantem Aufwand Elemente hinzufügen und entfernen zu können.

Implementierungen solcher Datentypen sind auf verschiedene Arten und unterschiedlich effizient möglich.

2. Lernziele

- Ihr könnt erklären, was *abstrakte Datentypen* sind, solche benutzen, implementieren und entwerfen.
- Ihr könnt von den abstrakten Datentypen *Stack* und *Queue* bei Bedarf Gebrauch machen.
- Ihr könnt begründet entscheiden, wann bzw. wofür man *deque* benutzen sollte anstelle von *Listen*.

3. Abstrakte Datenstrukturen

Datenstrukturen kann man aus zwei Perspektiven betrachten: Wer sie in eigenen Programmen benutzt interessiert sich vor allem für die *Spezifikation*, die aussagt *was* diese Datenstrukturen «können». Eine solche Spezifikation abstrahiert von der *Implementierung*, die festlegt, *wie* die spezifizierten Operationen ausgeführt werden können.

Die Spezifikation beschreibt das (zu erwartende) Verhalten – zum Beispiel durch *Vor-* und *Nachbedingungen* der einzelnen Operationen –, während die Implementierung von den einzelnen Schritten der nötigen Algorithmen ausgemacht wird, also von konkreten Programmanweisungen. Ausserdem muss die Implementierung auch die *Information*, die in der Datenstruktur repräsentiert werden soll, auf den Computer-Speicher abbilden, d.h. beschreiben, *wie genau* die Daten repräsentiert werden sollen. Dafür gibt es oft mehr als eine Möglichkeit. Je nach Art der Repräsentation der Daten sind unterschiedliche (oft auch unterschiedlich effiziente) Verfahren für die Operationen darauf möglich.

Ein *abstrakter Datentyp* besteht nur aus einer Spezifikation. Die ermöglicht es, Programme zu formulieren, die eine Datenstruktur dieses Typs benutzen. Um diese Programme dann auszuführen, muss man eine Implementierung dazu binden – also eine *konkrete Datenstruktur*. Die Auswahl der konkreten Implementierung kann jedoch bis zur Programmausführung aufgeschoben werden.

Die Programmiersprachen der Welt bieten individuell verschiedene Mechanismen an, um abstrakte Datentypen sowie die Zusammenhänge mit konkreten Datenstrukturen auszudrücken. *Python* bietet hier weniger explizite Unterstützung als andere, insbesondere als statisch typisierte Sprachen, aber dafür verschiedene Möglichkeiten, wie man sich behelfen kann.

Beispiel Bag – Spezifikation

Betrachten wir als Beispiel einen einfachen abstrakten Datentyp für *Bags* oder *Multisets*. Dabei handelt es sich um eine Verallgemeinerung des mathematischen *Mengen*-Konzepts: Im Unterschied zu einer Menge kann ein Bag einzelne Elemente auch mehrfach enthalten.[1]

In einen Bag können Elemente einzeln eingefügt und daraus entfernt werden, wie das auch bei Mengen der Fall ist. Im Unterschied zu Mengen werden jedoch mehrfach eingefügte gleiche Elemente gezählt. Ein mathematisches Modell dazu ist eine Funktion, die zu jedem Element einer Grundmenge angibt, wie oft es im Bag enthalten ist. Diese Anzahl wird auch als *Multiplizität* bezeichnet. Elemente mit der Multiplizität 0 gelten als im Bag *nicht* enthalten.

[1] *Bags* sind unter anderem eine Grundlage der *Relationen-Algebra*, dem formalen Modell von relationalen Datenbanksystemen.

Ein einfacher *abstrakter Datentyp Bag*, den wir hier als Beispiel betrachten wollen, muss mindestens drei Operationen anbieten:

1. *Ein Element hinzufügen* – `bag.add_one(element)`: Ist `element` noch nicht in `bag` enthalten, ist es nachher darin enthalten; ansonsten ist es nachher einmal mehr enthalten als zuvor. In anderen Worten: Die Multiplizität von `element` in `bag` erhöht sich um 1.
2. *Ein Element entfernen* – `bag.remove_one(element)`: Ist `element` in `bag` mehrfach enthalten, wird es einmal entfernt; ist es einmal enthalten, wird es vollständig entfernt; ist es nicht enthalten, passiert nichts. Kurz: Ist die Multiplizität von `element` in `bag` grösser als 0, wird sie um 1 reduziert.
3. *Die Multiplizität eines Elements feststellen* – `n = bag.number_of(element)`: Die *Multiplizitäts-Funktion* gibt an, wie oft `element` in `bag` enthalten ist. Das Resultat ist eine nicht-negative ganze Zahl.

Beispiel Bag – Technische Umsetzung einer abstrakten Datenstruktur

a. Funktionales Modell:

Die Programmiersprache *Python* erlaubt es, *Funktionen als Werte von Variablen* aufzufassen. Ein *Bag* lässt sich dann als Tupel seiner drei Funktionen (`add_one`, `remove_one` und `number_of`) repräsentieren, die auf einem gemeinsamen (versteckten) Datenspeicher operieren. Wurde ein solches Funktionen-Tripel einer Variablen `bag` zugewiesen, können die einzelnen Operationen mit Aufrufen wie `bag[0](element)` für die `add_one`-Operation aktiviert werden. Mit `namedtuple` aus dem zum Kern gehörenden Modul `collections` kann man das aussagekräftiger formulieren und gleichzeitig für eine einfache Repräsentation des abstrakten Datentyps sorgen:

```
Bag = namedtuple('Bag', 'add_one remove_one number_of')
```

Eine Implementierung von `Bag`, die mit dieser Definition kompatibel ist, kann dann auf die erwartete Art benutzt werden: Mit Aufrufen wie `bag.add_one(element)`.

Das so definierte *named tuple* `Bag` repräsentiert einen *abstrakten Datentyp*, denn es zeigt an, welche Operationen aufgerufen werden können. Zur Beschreibung von Parametern, Resultaten und Auswirkung braucht es allerdings zusätzliche Dokumentation, auf die wir hier verzichten wollen.

Eine mögliche *Implementierung* zu `Bag` auf Basis von Listen könnte so aussehen:

```
def bag_l():                        # generator function for a concrete Bag instance
   lst = []                         # introduce new variable lst to be used
                                    # within the functions defined below
   def add_one(element):
      lst.append(element)

   def remove_one(element):         # these functions are defined separately
      try:                          # upon each call to bag_l and operate on
         lst.remove(element)        # the lst generated during the same call
      except ValueError:
         pass

   def number_of(element):
      return lst.count(element)

   return Bag(add_one, remove_one, number_of)    # bag_l() returns function tripel
```

Diese *konkrete Datenstruktur* könnte so verwendet werden:

```
b = bag_l()                         # generate new Bag b
b.add_one('a')                      # insert an element 'a'
b.add_one('b')
b.add_one('a')
b.remove_one('a')                   # remove an element 'a'
b.add_one('a')
assert b.number_of('a') == 2        # check multiplicity of element 'a' in bag b
assert b.number_of('b') == 1
```

Bei dieser Art der Definition eines abstrakten Datentyps `Bag` und seiner Implementierung besteht die Kopplung nur über *Konventionen*. Das erkennt man u.a. daran, dass der Name `Bag` des abstrakten Datentyps lediglich in der `return`-Anweisung der generierenden Funktion `bag_l` auftaucht.

b. Klassen-Modell:

Eine andere Möglichkeit zur Repräsentation von abstrakten Datentypen und zugehöriger Implementierungen bietet die *objektorientierte Programmierung*. Dabei nutzt man die Klassen-Hierarchie aus, also die Möglichkeit, Spezialisierungen explizit zu deklarieren.

Der *abstrakten Datentyp* `Bag` würde dann als eine *abstrakte Klasse* formuliert. Das ist eine Klasse, welche die Operationen als Funktionen (auch als *Methoden* bezeichnet) definiert, aber nicht implementiert. Letzteres wird in *Python* ausgedrückt, indem man anstelle der Implementierungen die Anweisung nicht zu tun schreibt: `pass`.

```
class Bag:
   def add_one(self, element):
      pass

   def remove_one(self, element):
      pass

   def number_of(self, element):
      pass
```

Abstrakte Klassen eigenen sich also zur Formulierung *abstrakter Datentpyen*. Das auch die Parameter der Operationen angegeben werden, hilft zur Dokumentation. *Konkrete Datenstrukturen* können dazu entsprechend als *konkrete Unterklassen* formuliert werden:

```
class BagL(Bag):                # BagL is a Sub Class of Bag
   def __init__(self):          # generator function for a concrete Bag instance
      self._lst = []            # introduce new variable _lst to be used

   def add_one(self, element):
      self._lst.append(element)

   def remove_one(self, element):
      try:
         self._lst.remove(element)
      except ValueError:
         pass

   def number_of(self, element):
      return self._lst.count(element)
```

Im Gegensatz zur Repräsentation als Funktionen-Tripel ist hier durch die Super-Klassen-Deklaration klar formuliert, dass `BagL` eine Implementierung des abstrakten Datentyps `Bag` sein soll.

Diese *konkrete Datenstruktur* könnte so verwendet werden:

```
b = BagL()                      # generate new Bag b (instantiate class BagL)
b.add_one('a')                  # insert an element ‘a’
b.add_one('b')
b.add_one('a')
b.remove_one('a')               # remove an element ‘a’
b.add_one('a')
assert b.number_of('a') == 2    # check multiplicity of element ‘a’ in bag b
assert b.number_of('b') == 1
```

Interessanterweise sieht das die Datenstruktur verwendende Programm – bis auf das Erzeugen der konkreten Datenstruktur am Anfang – völlig gleich aus, egal ob ein *Bag* als *Funktionen-Tripel* oder als *Instanz einer Klasse* implementiert ist.

Beispiel Bag – Alternative und effizientere Implementierung mittels Dictionaries

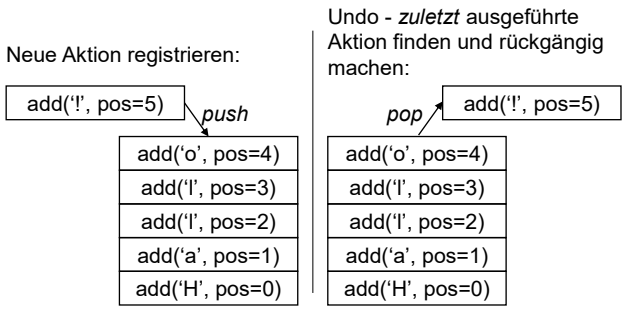
Wir haben zur Illustration jeweils eine einfache Listen-basierte Implementierung des *abstrakten Datentyps* `Bag` angegeben. Mit Ausnahme der Operation `add_one` benötigen aber alle Operationen einen Laufzeitaufwand von $O(n)$. Das lässt sich spürbar verbessern, wenn man *Dictionaries* anstelle von *Listen* verwendet, um die nötigen Informationen im Speicher zu repräsentieren. Entsprechende Implementierungen für die beiden obigen Formulierungs-Modelle seien den Lesenden als Übungsaufgabe überlassen.

4. Warteschlangen: Datentypen «Stack» und «Queue»

In der Programmierung gibt es immer wieder Aufgaben zu lösen, bei denen Elemente entstehen, die später wieder benötigt werden oder mit denen später weitergearbeitet werden muss. Diese Elemente möchte man dann in *Warteschlagen* deponieren, wo sie aufbewahrt werden, bis sie «an der Reihe» sind. Zwei wichtige solche Warteschlangen-Arten sind *Stack* und *Queue*.

4.1. Abstrakter Datentyp «Stack»

Ein Beispiel-Anwendungsfall für einen *Stack* sind die *Undo*- und *Redo-Operationen*, wie sie allgemein von Text-Editoren angeboten werden. Dazu müssen Informationen über die einzelnen ausgeführten Editier-Operationen aufbewahrt werden, damit sie im Falle eines *Undo*-Befehls wieder abgerufen werden können.



Bei einem *Undo*-Befehl muss die zuletzt ausgeführte Aktion rückgängig gemacht werden. Folgen darauf unmittelbar weitere *Undo*s, müssen weitere – frühere – Aktionen zurückgenommen werden. Die Reihenfolge ist dabei umgekehrt zur ursprünglichen Ausführungsreihenfolge der Aktionen.

Das kann man sich so vorstellen, dass man Informationen über die einzelnen Aktionen bei ihrer Ausführung aufstapelt. Für jede neu ausgeführte Aktion legt man einen Zettel mit den Informationen darüber auf einen *Stapel* (engl. *Stack*). Wird nun ein *Undo* verlangt, nimmt dann den obersten Zettel vom Stapel herunter und macht die darauf beschriebene Aktion rückgängig.

Ein solcher *Stack* wird auch als *LIFO*-Datenstruktur (Last-In-First-Out) bezeichnet.

In Analogie zum Bild des Stapels (oder des Aufstapelns) wird die Operation zum Hinzufügen (auf den Stapel legen) meist `push` und die zum wieder herunter nehmen `pop` genannt. Dazu kann es sinnvoll sein noch eine Operation `size` anzubieten, mit der sich die aktuelle Stack-Höhe, also die Anzahl enthaltener Elemente abfragen lässt.

Ein entsprechender abstrakter Datentyp als Funktions-Tripel könnte daher so aussehen:

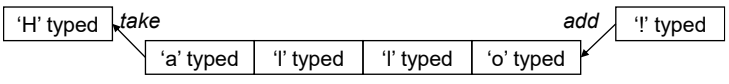
```
Stack = namedtuple('Stack', 'push pop size')
```

4.2. Abstrakter Datentyp «Queue»

Quasi das Gegenstück zur *LIFO*-Datenstruktur ist sind *Warteschlangen* (engl. *Queue*), die *FIFO*-Datenstruktur (First-In-First-Out). Es wird jeweils das Element zurückgegeben, das am längsten in der Schlange «wartet». Eine Queue liefert die Elemente also in genau der Reihenfolge aus, in der sie eingefügt wurden.

Anwendungsfälle für *Queues* sind Situationen, in denen es eine Entkopplung bei der Verarbeitung braucht, z.B. zwischen verschiedenen Prozessen. So kann ein Webserver von mehreren Clients sehr schnell hintereinander Nachrichten (Requests) empfangen, die dann aber von einem Prozess nacheinander verarbeitet werden müssen. Dafür sammelt der Webserver die Nachrichten zunächst in einer *Queue* aus der sie später zur Verarbeitung eine nach der anderen entnommen werden.

Computer-Betriebssysteme enthalten im Kern meist eine *Event Queue* in der alle von aussen eintreffenden Ereignisse registriert werden. Dort laufen die Meldungen einer Vielzahl von Peripheriegeräten (Tastatur, Maus, etc.) zusammen. Nebenbei werden die Ereignisse so auch in eine sequenzielle Reihenfolge gebracht. Sie können dann ohne gegenseitige Konflikte verarbeitet werden.



Für die Namen der *Queue*-Operationen hat sich nicht eine so einheitliche Terminologie durchgesetzt, wie bei den *Stacks*. So sind für die Operation zum Hinzufügen eines Elements die Namen `append` und `add` gebräuchlich und zum Entfernen des ersten Elements `take`, `get`, `first` oder auch `pop`.

Ein entsprechender abstrakter Datentyp als Funktions-Tripel könnte daher so aussehen:

```
Queue = namedtuple('Queue', 'add take size')
```

4.3. Implementierung von «Stack» und «Queue» mit Listen

Stacks und *Queues* lassen sich beide einfach und effizient auf der Basis von Listen (wiederum basierend auf Arrays) implementieren. Man muss allerdings den Aufwand der einzelnen Listenoperationen im Auge haben.

Stacks können naheliegenderweise auf zwei Arten implementiert werden: Ein neues auf den Stapel gelegtes Element kann entweder am Ende oder am Anfang einer Liste eingefügt werden. Das Laufzeitverhalten der beiden Implementierungen ist aber spürbar unterschiedlich.

Welche dieser beiden Implementierungen ist zu bevorzugen? Wie bedeutend ist der Unterschied?

```
def stack_1():
   lst = []

   def push(x):
      lst.append(x)

   def pop():
      return lst.pop()

   def size():
      return len(lst)

   return Stack(push, pop, size)
```

```
def stack_2():
   lst = []

   def push(x):
      lst.insert(0, x)

   def pop():
      return lst.pop(0)

   def size():
      return len(lst)

   return Stack(push, pop, size)
```

Weniger klar stellt sich die Situation bei *FIFO-Queues* dar. Auch hier sind zwei Implementierungen möglich: Einfügen am hinteren Ende der Liste und Entfernen vorne – oder umgekehrt. In absolut gemessenen Zeiten unterscheiden sich diese beiden Implementierungen zwar deutlich, aber nicht in der asymptotischen Komplexität.[2]

```
def queue_1():
   lst = []

   def add(x):
      lst.append(x)

   def take():
      return lst.pop(0)

   def size():
      return len(lst)

   return Queue(add, take, size)
```

```
def queue_2():
   lst = []

   def add(x):
      lst.insert(0, x)

   def take():
      return lst.pop()

   def size():
      return len(lst)

   return Queue(add, take, size)
```

Mindestens von einem theoretischen Standpunkt aus ist es egal, ob man `queue_1` oder `queue_2` verwendet. Für eine sehr kleine Datenmenge kommt es auf den Zeitaufwand nicht so sehr an und für eine grosse Datenmenge möchte man etwas Besseres als beide hier gezeigte Varianten.

Dass dies möglich ist, zeigt der spezielle Datentyp *deque*. Die nebenstehende *Queue*-Implementierung `queue_d` ist klar effizienter als es die beiden auf Listen aufgebauten sind.

```
def queue_d():
   deq = deque()

   def add(x):
      deq.append(x)

   def take():
      return deq.popleft()

   def size():

      return len(deq)

   return Queue(add, take, size)
```

[2] Die Unterschiede der absoluten Zeiten sind wohl nur anhand des internen Speichermanagements der Listen-Implementierung zu erklären, das für uns aber eine Blackbox ist.

4.4. Die Datenstruktur «deque»: Double-Ended Queue

Viele Datenstruktur-Bibliotheken bieten eine spezielle Datenstruktur *deque* (Double Ended Queue – auszusprechen wie «Deck») für Warteschlangen an, die optimiert ist auf das Hinzufügen und Entfernen von Elementen an beiden Enden.

In *Python* findet sich eine `deque`-Implementierung im Modul `collections`. Neben anderen bietet diese Datenstruktur Operationen `append`, `appendleft`, `pop` und `popleft` an, die alle mit konstantem Aufwand *O*(1) ablaufen. Damit ist `deque` der Typ der Wahl, wenn eine FIFO-Warteschlange benötigt wird. Aber auch *Stacks* lassen sich damit effizient implementieren und alle (vorstellbaren) Mischformen dazwischen.[3]

4.5. Effiziente FIFO-Queue mit zyklischen Listen

FIFO-Queues können auch mit *Arrays* (oder in *Python* ersatzhalber mit *Listen*) effizient implementiert werden, wenn man die maximale Anzahl gleichzeitig zu speichernder Elemente im Voraus kennt.[4,5]

Die Effizienz der in Abschnitt 4.3 gezeigten Implementierungen `queue_1` und `queue_2` leidet darunter, dass beim Herausnehmen bzw. Hinzufügen von Elementen an Index-Position 0 alle anderen Elemente in der Liste verschoben werden müssen. Das ist eine (verhältnismässig) teure *O*(n)-Operation.

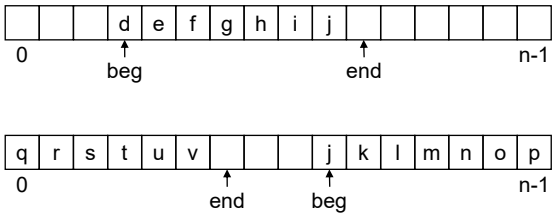
Um das zu vermeiden, muss man den *logischen Anfang* der *Queue* unabhängig vom Index 0 der Liste definieren. Dazu benötigt man eine zusätzliche Variable, z.B. `beg`. Eine zweite Variable, z.B. `end`, markiert das logische Ende der Queue in der Liste. Dabei hat man die Wahl, ob `end` den Index des letzten Elements in der Queue oder den ersten freien Platz dahinter bezeichnet.

Laufzeit-Effizient wäre es, neue Elemente jeweils hinten an die Liste anzuhängen und Elemente von vorne her herauszunehmen, dabei aber die anderen Elemente nicht zu verschieben, sondern in einer Variablen zu speichern, wie viele Plätze am Anfang der Liste *logisch frei* sind, bzw. an welcher Indexposition sich das jeweils erste Element der *Queue* befindet.

Diese Lösung ist aber natürlich nicht Speicher-Effizient, denn der von der Liste real belegte Speicherplatz wird immer grösser. Es werden einfach nach und nach immer mehr Plätze am Anfang der Liste nicht mehr verwendet. Reserviert bleiben sie aber und könne nicht anderweitig genutzt werden.

Man muss also die Speichermenge begrenzen, etwas das sich bei klassischen Arrays von selbst ergibt. Hat man diesen Speicher «bis hinten» aufgefüllt, verlängert man die Liste nicht, sondern beginnt zwischenzeitliche frei gewordene vordere Plätze erneut zu benutzen. Solange nicht mehr Elemente in die Queue eingefügt werden, als Speicherplätze reserviert wurden, funktioniert dieses Schema.

Fügt man in eine solche *Queue* mit maximaler Grösse n 10 Elemente ein und entfernt 3 davon wieder, entsteht die obere der nebenstehend gezeigten Situationen. Neue Elemente werden beim Index `end` eingefügt, der anschliessend um 1 erhöht wird. Herausgenommen wird jeweils das Element beim Index `beg`, der ebenfalls danach um 1 vergrössert wird.



Damit die Werte von `beg` und `end` als Index in die Liste gültig bleiben, muss man sie nach dem Vergrössern um 1 jeweils mit einer Modulo-Operation auf den gültigen Bereich abbilden. So entsteht nach Einfügen von insgesamt 22 und «herausnehmen» von 9 Elementen die untere der gezeigten Situationen.

Eine solche *Queue* in *Python* zu programmieren, ist eine empfehlenswerte Übung. Dabei muss man darauf achten, dass keine illegalen Zuständen dadurch entstehen, dass versucht wird, aus einer leeren *Queue* ein Element zu beziehen oder zu viele Elemente einzufügen.

---

[3] Für weitere Informationen zu `collections.deque` empfiehlt sich ein Blick in die *Python*-Dokumentation (https://docs.python.org/3/library/collections.html#deque-objects)

[4] Man kann natürlich auch solche Datenstrukturen bei Bedarf durch Erzeugen eines neuen Arrays und Umkopieren der Daten vergrössern. Hier gelten dann die gleichen Überlegungen zur Amortisation des Aufwands wie bei halbdynamischen Listen, Hash Tables usw. Dies ist aber eine Aufgabe orthogonal zur Verwendung als zyklische Liste, weswegen wir da hier nicht im Detail betrachten wollen. Es mag aber vielleicht eine gute Übung sein, das einmal zu programmieren.

[5] Zyklische Listen auf Array-Basis sind eine leicht zu realisierende Lösung der Aufgabe und deswegen lohnt die Beschäftigung damit. Der Datentyp *deque* wird jedoch zumeist mit *verlinkten Listen* (linked lists) implementiert.