Array und Linked List

In der vorigen Textlektion haben wir gesehen, dass sich Datenstrukturen in sequenzielle Strukturenund Mengenstrukturen kategorisieren lassen. In dieser Lektion werden wir uns ein Szenario anschauen, welches eine sequenzielle Datenstruktur erfordert. Das bedeutet, dass die Datenstruktur unsere Daten in einer festen Reihenfolge verwalten soll. Wir werden uns hierzu zwei Beispiele näher anschauen, und uns Gedanken darum machen, wie geeignet beide Strukturen für unser Szenario sind: das Array und die Linked List.

Nach dieser Lektion kannst du:

* benennen, wie Array und Linked List auf dem Speicher angelegt sind,
* die Laufzeitkomplexitäten für wichtige Methoden in beiden Datenstrukturen ermitteln,
* *data structure augmentation einsetzen,* um die Laufzeitkomplexitäten von Methoden zu verbessern.

Szenario: Bei einem kleinen Online-Versandhändler werden eingehende Aufträge sukzessive abgearbeitet. Geht ein Auftrag ein, wird er in eine Warteschleife gestellt. Wechselt der Auftrag in den Bearbeitungsstatus, so wird er aus der Warteschleife entfernt und in ein Bearbeitungssystem verschoben. Die Geschäftsführung bittet dich, Code für die Warteschleife zur Verfügung zu stellen.

Bevor wir unser Problem in Code übersetzen können, sollten wir uns Gedanken um die passende Datenstruktur machen. Wie würdest du die Warteschleife implementieren? Sie sollte im Wesentlichen zwei Sachen können:

1. Geht ein Auftrag ein, wird ein Element am Ende eingefügt.
2. Wechselt ein Auftrag in den Bearbeitungsstatus, wird er vom Anfang entfernt.

Die Güte unserer Datenstruktur bemisst sich an den Laufzeitkomplexitäten dieser beiden Aufgaben. Die zugehörigen Methoden werden wir, unabhängig von der gewählten Datenstruktur, `insert\_last()` und `delete\_first()` nennen.

## Die Warteschleife als Array

Eine naheliegende Lösung ist, die Warteschleife als Python-Liste anzulegen. Mittels `append()` können wir Aufträge am Ende einfügen, mittels `pop(0)` vom Anfang entfernen. Die Aufträge könnten wir als Python-Dictionary speichern, der neben der Auftragsnummer auch Datum, Auftragsvolumen, Zahlungsmethode und andere Informationen enthalten kann.

Unsere Datenstruktur mit den Methoden `insert\_last()` und `delete\_first()` könnte so aussehen:

```python

class Queue:

  def \_\_init\_\_(self):

    self.queue = []

  def insert\_last(self, data):

    assert isinstance(data, dict)

    self.queue.append(data)

  def delete\_first(self):

    if len(self.queue) >= 1:

      self.queue.pop(0)

    else:

      print('Empty queue')

```

Ist `Queue` eine gute Implementierung für eine Warteschleife? Wie geeignet eine Datenstruktur ist, bestimmt sich daran, wie hoch die Laufzeitkomplexitäten der häufig genutzten Methoden sind. Wir betrachten im Folgenden immer den schlechtesten Fall, der durch die Big O-Notation beschrieben wird.

Was sind die Laufzeitkomplexitäten von `insert\_last()` und `delete\_first()` in `Queue`? Um diese Fragen beantworten zu können, müssen wir wissen, wie die `list` hinter den Kulissen arbeitet. Was passiert, wenn wir `append()` und `pop(0)` nutzen?

In Python ist der Datentyp `list` alsArray angelegt. Für das Array wird ein bestimmter Platz im Speicher reserviert. Seine Elemente sind als zusammenhängender Block ohne Unterbrechungen oder Lücken angelegt. Dem ersten Element wird eine so genannte *Basisadresse* zugeordnet. Dort befindet sich die Indexposition 0. Für jedes Element ist ein bestimmter Platz vorgehalten. Zählen wir zur Basisadresse die Länge dieses Platzes nach rechts, so landen wir bei Indexposition 1, zählen wir dann wieder die Länge eines Platzes nach rechts, landen wir bei Indexposition 2 und so weiter. Weil der Speicherort jedes Elements vorhersehbar ist, ist der Zugriff auf Elemente in einer Liste anhand der Indexposition in $O(1)$ möglich. Das einer der größten Vorteile des Arrays.

Das Array können wir uns für unser Warteschleifenbeispiel in etwa so vorstellen:

Ein Bild, das Text, Reihe, Screenshot, Diagramm enthält.

Automatisch generierte Beschreibung

Abbildung 1: Das Array

Die Aufträge hatten wir in `Queue` als `dict`definiert. Hier stellen wir sie der Einfachheit halber nur als Auftragsnummern dar.

**Vertiefung**: dynamische Anpassungen im Array.

Vielleicht fragst du dich, was passiert, wenn der reservierte Block voll ist? In Python kann eine Liste schließlich immer erweitert werden. Dann wird ein neuer Block mit mehr Platz freigeräumt, in den alle Elemente kopiert werden. Wird absehbar, dass der reservierte Platz nicht benötigt wird, werden analog hierzu die Elemente in einen kleineren Block verschoben.

## Das erste Element aus dem Array entfernen

Was ist die Laufzeiteffizienz von `delete\_first()` in `Queue`? Wechselt ein Auftrag in den Bearbeitungsstatus, entfernt die Methode ihn mittels der Listenmethode `pop(0)` aus unserer Warteschleife. Die Laufzeitkomplexitäten von `delete\_first()` und `pop(0)` sind damit identisch.

Kannst du dir bereits denken, welche Operationen die Listenmethode ` pop(0)` im Hintergrund ausführt? Versuche doch mal, die folgende Frage zu beantworten, bevor du weiterliest.

**Frage 1 (als Quiz mit Feedback): Was ist die Laufzeitkomplexität von `my\_list.pop(0)´?**

1. **$O(1)$**
2. **$O(n)$\***
3. **$O(n^2)$**

Wird ein Element aus einem Array entfernt, müssen alle folgenden Elemente jeweils um einen Platz nach links verschoben werden. Wird das erste Element entfernt, werden **alle** übrigenElemente verschoben. In unserer Warteschleife können wir uns das in etwa so vorstellen:

Ein Bild, das Text, Screenshot, Zahl, Rechteck enthält.

Automatisch generierte Beschreibung

Abbildung 2: Der erste Auftrag wird aus dem Array entfernt (gif)

Zunächst wird also der erste Auftrag entfernt. Anschließend rutscht der zweite Auftrag auf die nun frei gewordene Indexposition 0, danach rutscht der dritte Auftrag auf Indexposition 1, und so weiter. Weil diese Operation für sämtliche Aufträge in der Warteschleife durchgeführt wird, hat `pop(0)` und damit auch `delete\_first()` in `Queue` eine lineare Laufzeitkomplexität, $O(n)$.

Das ist nicht sonderlich schnell. Kann die Linked List das besser lösen? Wir schauen uns zunächst wieder an, wie die Linked List im Speicher angelegt ist.

## Die *Linked List*

Die Linked List funktioniert etwas anders als das Array: Sie ist eine Pointer-basierte Datenstruktur. Statt Speicherplatz für die gesamte Warteschleife freizuhalten, erhält jeder Auftrag nun seinen eigenen Platz im Speicher. Zusätzlich zu den eigentlichen Informationen zum Auftrag speichern wir in jedem Element der Linked List einen Pointer zum nächsten Auftrag mit ab. Mittels dieser Pointer wird die Datenstruktur zusammengehalten, und ihre Elemente erhalten eine Reihenfolge.

Wie im Array erhält auch hier die Datenstruktur eine Basisadresse. Sie ist die Speicheradresse vom ersten Element der Linked List, auch als *head* bezeichnet*.* Alternativ können wir auch sagen, dass unsere Linked List-Struktur einen *head*-Pointer zum ersten Element hat. Sprechen wir in der Linked List also von Pointer, unterscheiden wir zwischen dem ***head*-Pointer** der Linked List und den ***next*-Pointern** der einzelnen Elemente. Die folgende Abbildung verdeutlicht die Struktur der Linked List.

Ein Bild, das Text, Screenshot, Diagramm, Schrift enthält.

Automatisch generierte Beschreibung

Abbildung 3: Die Linked List

In dieser Textlektion schauen wir uns noch keinen Code für die Linked List an. Stattdessen wollen wir uns hier ein paar konzeptionelle Gedanken um diese Datenstruktur machen. In der nächsten Übung werden wir sie dann implementieren.

Möglicherweise ist es hilfreich, sich die Grundstruktur bereits vor Augen zu halten. Die Datenstruktur selbst wird durch eine Klasse `LLQueue` repräsentiert. In ihr legen wir den *head*-Pointer als Attribut an. Ein Auftrag ist ein Objekt der Klasse `LinkedNode`. Der *next*-Pointer ist Attribut dieser Klasse. Unser Hauptziel ist, zwei Methoden in `LLQueue` anzulegen: eingehende Aufträge werden mittels `insert\_last()` angehängt, in den Bearbeitungsstatus wechselnde Aufträge werden mittels `delete\_first()` aus der Warteschleife entfernt. Die Laufzeitkomplexität beider Methoden soll möglichst gering sein.

## Das erste Element aus der Linked List entfernen

Wechselt ein Auftrag in den Bearbeitungsstatus, war das in unserer *Array*-Struktur nicht sehr schnell in linearer Laufzeit möglich. In diesem Abschnitt werden wir erkunden, ob die *Linked List* besser für diese Aufgabe geeignet ist.

Der erste Auftrag in unserer Warteschleife ist der, zu dem der *head*-Pointer hindeutet. Was müssen wir tun, um den ersten Auftrag zu entfernen? Versuche wieder, die Frage zu beantworten, bevor du weiterliest.

**Frage 2: Welches ist die bestmögliche Laufzeitkomplexität für das Entfernen des ersten Auftrags in der *Linked List*?**

1. $O(1)$\*
2. $O(n)$
3. $O(n^2)$

Die Lösung ist einfach. Wir müssen den head-Pointer neu setzen. Deutet er auf den ursprünglich zweiten Auftrag, rückt dieser automatisch an die erste Stelle. Der ursprünglich erste Auftrag ist dann nicht mehr mit unserer Datenstruktur verlinkt. Die Abbildung stellt den Prozess dar.

Ein Bild, das Diagramm, Plan, Rechteck, Quadrat enthält.

Automatisch generierte Beschreibung

Ein Bild, das Diagramm, Plan, Rechteck, Text enthält.

Automatisch generierte Beschreibung

Abbildung : das erste Element wird aus der Linked List entfernt (gif)

Die Anzahl an Operationen, die notwendig ist, um den *head*-Pointer neu zu setzen, ist vollkommen unabhängig von der Anzahl der Aufträge in der Warteschleife. Damit ist das Entfernen des ersten Auftrags in der *Linked List* in konstanter Laufzeit möglich. Besser geht es nicht! Die *Linked List* ist für diese Aufgabe eindeutig besser geeignet als das Array.

In unserer Warteschleife müssen allerdings auch neu eingegangene Aufträge ans Ende angehängt werden. Wie schnell kann die *Linked List* diese Aufgabe erledigen?

## Einen Auftrag ans Ende der *Linked List* anhängen

Wollen wir einen neu eingehenden Auftrag in unsere Warteschleife einstellen, müssen wir zwei Sachen tun: Wir müssen den bisher letzten Auftrag finden und ihn mittels eines *next-Pointer* mit dem neuen Auftrag verlinken. In der nächsten Frage sollst du beantworten, wie schnell das Auffinden des letzten Auftrags erledigt werden kann. Nimm an, dass die *Linked List* genauso strukturiert ist, wie in der Abbildung oben dargestellt: Die Struktur selbst hat einen head-Pointer zum ersten Element und jedes Element hat einen *next*-Pointer zum folgenden Element.

**Frage 3: Was ist die Laufzeitkomplexität für das Auffinden des letzten Auftrags (ohne Entfernen)?**

1. **$O(1)$**
2. **$O(n)$\***
3. **$O(n^2)$**

Hier stoßen wir auf einen der fundamentalen Nachteile der *Linked List*: um das letzte Element zu **finden**, müssen wir durch unsere gesamte Warteschleife hindurch iterieren, beginnend beim ersten Auftrag. Andernfalls wissen wir nicht, an welchem Platz im Speicher sich dieser Auftrag befindet. Ist unsere Warteschleife sehr lang, kann das durchaus einige Zeit in Anspruch nehmen!

Wie schnell ist das Anhängen eines neu eingehenden Auftrags dann? Nimm für die nächste Frage an, dass unsere Methode `insert\_last()` beides beinhaltet, das Auffinden und das Entfernen.

**Frage 4: Welches ist die bestmögliche Laufzeitkomplexität von `my\_llqueue.insert\_last()`?**

* 1. **O(1)**
  2. **O(n)\***
  3. **O(n^2)**

*Lösungserklärung (aufklappbar): Das Auffinden hat eine lineare Laufzeitkomplexität, wie oben gesehen. Das Anhängen hat eine konstante Laufzeit, weil nur der Pointer des vorletzten Elements angepasst werden muss. Insgesamt ergibt sich damit O(n).*

Auch wenn das Entfernen an sich nur eine konstante Laufzeit benötigt, so dominiert das Auffinden den gesamten Prozess. So richtig zufriedenstellend ist die Laufzeit von `my\_llqueue.insert\_last()` also nicht.Allerdings: es ist unsere Datenstruktur, und niemand schreibt uns vor, dass wir einem vorgegebenen Schema folgen müssen. Wir dürfen unsere *Linked List* modifizieren. Hast du eine Idee, wie wir unsere Methode schneller gestalten können?

**Frage 4: Wie kann `insert\_last()` in *O(1)* implementiert werden? Welche kleine Änderung musst du an deiner Datenstruktur vornehmen?**

*Lösung (ausklappbare Box):*

*Bisher hat die Datenstruktur einen head-Pointer zum ersten Element. Analog hierzu könnte ein tail-Pointer zum letzten Element zeigen. Hierdurch ist es möglich, direkt auf das letzte Element zuzugreifen. Die Methode `insert\_last()` verlinkt dann den bisher letzten Auftrag mit dem neuen Auftrag und passt den tail-Pointer entsprechend an.*

Bist du auf die Lösung gekommen? Dann hast du eine Technik verwendet, die in Datenstrukturen eine große Rolle spielt: die *data structure augmentation.* Sie wird oft dazu genutzt, Code effizienter zu gestalten. Hier führen wir den *tail*-Pointer als zusätzliches Attribut in unsere Datenstruktur `LLQueue` ein. Dadurch müssen wir nun nicht mehr durch die gesamte Datenstruktur hindurch iterieren. Der Preis: Wir benötigen etwas mehr Speicherplatz und müssen den *tail-*Pointer aktualisieren, wenn wir einen neuen Auftrag in unsere Warteschleife einstellen.

**Merke**: Bei der *data structure augmentation* legen wir ein oder mehrere Attribute mit in der Klasse an, die wir in den Methoden gegebenenfalls anpassen. Hierdurch erreicht eine Methode unter Umständen eine bessere Laufzeitkomplexität.

Ob sich *data structure augmentation* lohnt, ist immer eine Abwägung: Erfordert das zusätzliche Attribut einen enorm hohen Aktualisierungsaufwand? Dann wird die verbesserte Laufzeitkomplexität durch die Hintertür wieder wettgemacht. Möglicherweise steigt auch der Speicherplatzbedarf, und macht so die Vorteile der *data structure augmentation* zunichte. Für den vorgesehenen *tail*-Pointer müssen wir uns weder um Aktualisierungsaufwand noch um einen gesteigerten Speicherplatzbedarf Sorgen machen: die Aktualisierung muss nur für den jeweils letzten Auftrag vorgenommen werden, und auch der zusätzlich benötigte Speicherplatz ist sehr gering. Die Vorteile, die der *tail*-Pointer mit sich bringt, werden keinesfalls aufgewogen. Wir halten fest: führen wir den *tail*-Pointer als zusätzliches Attribut in unsere Datenstruktur ein, ist auch das Anhängen von Aufträgen ans Ende unserer Warteschleife in konstanter Laufzeit möglich.

Vertiefung: **Big-O für Speicherplatz.** Wir hatten bereits gesehen, dass der Effizienzbegriff neben der Laufzeit auch den Speicherplatzbedarf umfasst. Analog zur Big O-Notation für die Laufzeitkomplexität gibt es auch eine Big O-Notation für Speicherplatzkomplexität. $O(1)$ bedeutet, dass der benötigte Speicherplatz eines Programms/einer Methode unabhängig von der Eingabegröße $n$ ist. Steigt der Speicherplatz linear, ist die Komplexität $*O(n)$.* Das Einstellen eines Auftrags in die Warteschleife hat sowohl im Array als auch in der Linked List eine Speicherplatzkomplexität von $O(1)$. Stellen wir $n$ Aufträge in die Warteschleife ein, so ist die Speicherplatzkomplexität hierfür $O(n)$.

## Einen Auftrag ans Ende des Arrays anhängen

Wir wissen bereits, dass die Linked List dem Array in unserem Warteschleifenbeispiel überlegen ist, denn sie kann sowohl beides in konstanter Laufzeitkomplexität: Aufträge vom Anfang entfernen und ans Ende anhängen. Das Array hingegen benötigt eine lineare Laufzeit, um Aufträge vom Anfang zu entfernen. Der Vollständigkeit halber überlegen wir uns hier auch noch, wie effizient das Array Aufträge ans Ende anhängen kann.

Erinnere dich noch einmal kurz an die Funktionsweise der Python-Liste bzw. des Arrays: der reservierte Speicherplatz im Array wird in der Regel nicht vollständig ausgenutzt. Wollen wir ein Element ans Ende einer Python-Liste anhängen, müssen wir wissen, wie viele Plätze bereits besetzt sind. Wir könnten hierfür durch das Array hindurchiterieren. Das ist allerdings wieder nur in linearer Laufzeit möglich. Deshalb bietet sich eine data structure augmentation an: ein Attribut `*length`* macht das Anhängen eines Auftrags ans Ende der Warteschleife schneller.

Überhaupt ist in Python die Listenmethode `len()` sehr zentral. Sie wird häufig und für viele andere Methoden genutzt. Das zusätzliche Attribut lohnt sich also.

Du solltest nun die folgende Frage beantworten können:

**Frage 5: Wie effizient ist `list.append()`?**

1. **O(1)\***
2. **O(n)**
3. **O(n^2)**

Fassen wir also zusammen: eine modifizierte Version der Linked List ist für unser Warteschleifenbeispiel ideal, denn beide zentralen Methoden können in konstanter Laufzeit implementiert werden. Eine Array-Struktur kann zwar gut mit dem Anhängen von neu eingehenden Aufträgen ans Ende der Warteschleife umgehen. Das Entfernen von Aufträgen vom Anfang ist allerdings nur in linearer Laufzeit möglich.

**Vertiefung***:* FIFO, LIFO, Queue und Stack. Die Warteschleife ist eine gängige Illustration für die Vorteile der Linked List. Weil sie durch den *tail*-Pointer erweitert wird, wird sie bisweilen selbst als Datenstruktur bezeichnet. Gängige Ausdrücke hierfür sind \*FIFO (First-In-First-Out)\* oder \*Queue\*. Das Gegenstück hierzu ist die \*LIFO (Last-In-First-Out)\*-Struktur, bei der die zuletzt eingegangenen Daten zuerst wieder entfernt werden. \*LIFO\* wird auch als \*Stack\* bezeichnet. Die Begriffe \*LIFO\* und \*FIFO\* sind sowohl in der Informatik als auch in der Lagerhaltung oder im Rechnungswesen gängig.

Du entscheidest dich, die Warteschleife für den Online-Versandhändler als *Linked List* zu implementieren. Das wirst du in der nächsten Übung tun.

Merke:

* Für die Anforderungen an eine Warteschleife eignet sich die *Linked List* hervorragend.
* Sie sollte dann zusätzlich zum *head-*Pointer einen *tail-*Pointer haben.
* *Mit data structure augmentation* lassen sich Datenstrukturen zuweilen effizienter gestalten.