Textübung

Einführung in die Datenstrukturen

Wie du vielleicht schon weißt, ist Python eine high-level Programmiersprache. Python-Code ähnelt unserer Sprache und Denkweise. Das macht die Sprache für uns so einfach zu bedienen, und wir können uns darauf konzentrieren, analytische Probleme zu lösen.

Das ist einerseits sehr praktisch. Andererseits verbirgt Python viele Details und Komplexitäten der Hardware. Schreiben wir Python-Code, dann kümmern wir uns beispielsweise in der Regel nicht darum, wie unsere Daten auf der Hardware angeordnet sind. Stattdessen nutzen wir eingebaute Datentypen wie `list`, `dict`, `tuple` oder `set`. Sie stellen uns eine Menge Funktionalitäten zur Verfügung: Mit `my_list.sort()` können wir ein `list`-Objekt sortieren und mit `my_set.union(other_set)` zwei Mengen vereinigen.

Solche Operationen sind nichts anderes als Algorithmen. Erinnern wir uns: Ein Algorithmus ist eine Reihe von klar definierten Anweisungen, die ausgeführt werden, um eine bestimmte Aufgabe zu erfüllen. Sortieren wir mittels `my_list.sort()` die Listen-Elemente aufsteigend oder absteigend, wird im Hintergrund eine Reihe von Vergleichs- und Austauschschritten durchgeführt. Welche Laufzeitkomplexitäten haben `my_list.sort()` oder `my_set.union()`?

Diese Frage ist nicht ohne Weiteres zu beantworten. Uns fehlt eine wichtige Information: Wir wissen eben nicht, wie unsere Daten auf der Hardware angeordnet sind. Anders ausgedrückt: Wir kennen die **Datenstrukturen** von `list`, `tuple`, `dict` und `set` nicht. So können wir auch nichts darüber sagen, wie effizient ihre Methoden implementiert werden können.

Datenstrukturen sind grundlegende Konzepte in der Informatik, die die Organisation, Verwaltung und Speicherung von Daten beschreiben. Sie definieren, wie auf die Daten zugegriffen werden kann und wie sie manipuliert werden können. Je nach Anwendungsfall können bestimmte Datenstrukturen effizienter sein als andere.

Mit Datenstrukturen werden wir uns in diesem zweiten Teil des Moduls beschäftigen. In dieser Textlektion werden wir uns zunächst anschauen, wie wir Datenstrukturen kategorisieren können. Außerdem werden wir sehen, dass verschiedene Datenstrukturen unterschiedliche gut für verschiedene Anwendungsgebiete geeignet sind, und dass wir sie unseren Bedürfnissen entsprechend anpassen können.

Kategorisierung von Datenstrukturen

Schauen wir uns zunächst einmal an, wie sich Datenstrukturen kategorisieren lassen:

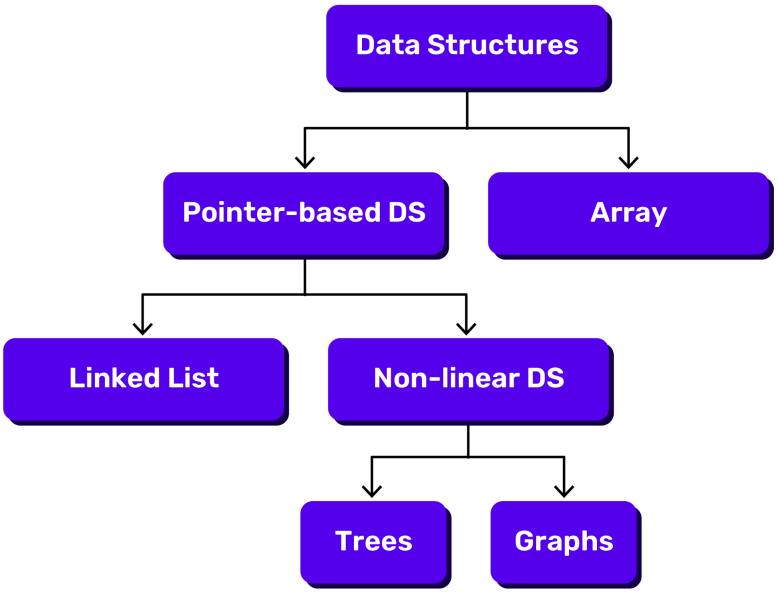

Abbildung 1: Kategorisierung von Datenstrukturen

Wir unterscheiden zwischen Arrays und *Pointer*-basierten Datenstrukturen.

Arrays sind als zusammenhängende Blöcke auf dem Speicher angeordnet. Das bedeutet, sie sind direkt hintereinander ohne Unterbrechungen oder Lücken dort angeordnet. Sie ähneln also auch physisch unserem Bild einer Liste. In Python ist der Datentyp `list` beispielsweise als Array implementiert. Seine Elemente sind als zusammenhängende Blöcke im Speicher angelegt. Das macht den Zugriff auf Elemente in einer Python-Liste ziemlich schnell, da der Speicherort jedes Elements vorhersehbar ist, basierend auf dem Speicherort des ersten Elements und der Position des gewünschten Elements in der Liste.

In Pointer-basierten Strukturen dagegen erhält jedes Element seinen eigenen Platz auf dem Speicher, und so genannte Pointer zeigen auf das jeweils nachfolgende Element oder auch auf die nachfolgenden Elemente.

Ein gängiges Beispiel ist die *Linked List*. Jedes Element besteht hier aus einem Datenwert und einem Pointer, der auf das nächste Element zeigt. Wir können sie uns wie eine Schnitzeljagd vorstellen: an jedem Ort (Element) liegt neben einem Hinweis zur Lösung des Rätsels (dem Datenwert) auch ein Hinweis zum nächsten Ort (Pointer). Pointer-basierte Datenstrukturen können auch nicht-linear sein. Ein Beispiel ist der Binärbaum. Jedes Element kann hier zwei Pointer haben, zum jeweils rechten und linken Kind.

Übrigens: es gibt auch Mischformen. In Python sind beispielsweise `dict` und `set` als *Hash Tables* implementiert. Der *Hash Table* ist in seiner Grundstruktur als Block angelegt, also als Array. Die einzelnen Positionen im Array enthalten dann wiederum die Daten, manchmal sogar mehrere. Diese können dann in Pointer-basierten Datenstrukturen organisiert werden. Wenn du mehr über den Hash Table erfahren möchtest, kannst du auf die aufklappbare Box klicken.

Vertiefung: Hash Table.

Den Hash Table können wir uns als Liste mit einer festgesetzten Länge vorstellen. Im Vergleich zur Python-Liste legen Nutzende hier nicht die Anordnung der Elemente fest. Sie werden mittels einer *Hash-Funktion* den Indexpositionen zugeordnet. Die Hash-Funktion wird in der Datenstruktur bereits mit angelegt. Beispielsweise könnte eine Hashfunktion alle Strings, die mit dem Buchstaben A beginnen, Indexposition 0 zuordnen, die mit B beginnen Indexposition 1 und so weiter. Wenn dann mehrere Elemente an ein und derselben Indexposition landen, sprechen wir von einer *Kollision*. Ein Hash Table muss damit umgehen können. Eine Möglichkeit ist, die Daten in einer linearen Datenstruktur anzulegen.

Die beste Datenstruktur existiert nicht

Es gibt nicht **die** beste Datenstruktur. Verschiedene Strukturen sind unterschiedlich gut für bestimmte Operationen geeignet. Schauen wir uns ein paar Beispiele an:

Der Zugriff auf Elemente per Index ist im Array sehr effizient, nicht aber in der Linked List oder im Hash Table. Wollen wir hingegen nach einem Element anhand seines Wertes suchen - und nicht anhand seines Indizes – ist das im Hash Table in der Regel sehr gut zu bewältigen. Ist uns wiederum die Anordnung der Elemente wichtig, ist der Hash Table recht unbrauchbar. Das ist auch der Grund, warum in Python `set` und `dict` als Hash-Table angelegt sind, `list` und `tuple` hingegen als Array.

Beim Python-Datentyp *`set`* spielt die Anordnung der Elemente keine Rolle. Wir können `set` nicht sortieren. Allerdings können wir Elemente anhand des Wertes suchen und entfernen und zusätzliche einfügen. In `list` hingegen ist die Anordnung der Elemente essenziell. Wir können schnell Elemente per Index finden, und neue an die Liste anhängen.

Verwaltet eine Datenstruktur die Elemente in einer von uns festgelegten Anordnung, sprechen wir von einer **sequenziellen Datenstruktur**. Das Array ist eine solche sequenzielle Datenstruktur. Werden die Elemente wie im Hash Table anders angeordnet, sprechen wir von einer **Mengenstruktur**.

Auch die Linked List ist eine sequenzielle Datenstruktur. Sie hat in Python mit `collections.deque` nur einen recht unbekannten Vertreter, denn für die allermeisten Anwendungen leisten die eingebauten Datentypen wie `list` und `tuple` sehr gute Dienste. Dennoch hat sie in manchen Situationen Vorteile. Wir werden uns in den nachfolgenden Übungen ein Szenario anschauen, in dem sie dem Array hinsichtlich der Laufzeitkomplexitäten ihrer wichtigsten Methoden überlegen ist.

Wir werden auch eine nicht-lineare Pointer-basierte Datenstruktur implementieren, nämlich den Binärbaum. Binärbäume sind hierarchisch aufgebaut und haben daher erstmal keine festgelegte Ordnung. Wenn wir sie durchlaufen, halten wir allerdings immer eine bestimmte Reihenfolge ein. Daher kann der Binärbaum Daten im Prinzip sequenziell verwalten. Ihre Vorteile spielen Binärbäume allerdings auf typischen Mengenoperationen, wie Suchen nach Wert oder die Vereinigung von Mengen, aus. Wir werden uns dazu ein Szenario anschauen, das einer Mengenstruktur bedarf.

Datenstrukturen sind ein abstraktes Konzept

Es gibt also nicht die beste Datenstruktur. Um genau zu sein, gibt es noch nicht einmal **das eine** Array, **die eine** Linked List oder **den einen** Hash Table. Diese Begriffe bezeichnen Konzepte, wie Daten organisiert, gespeichert und manipuliert werden können. Wir sollten sie als Baupläne oder Skizzen verstehen und nicht als fertige, konkrete Bauwerke.

So sind zwar `list` und `tuple` in Python beide als Array implementiert, dennoch haben sie sehr unterschiedliche Funktionalitäten. Das Python-Tupel ist, anders als die Python-Liste nicht veränderlich, daher können wir keine Elemente hinzufügen, entfernen oder überschreiben.

Datenstrukturen können sehr verschiedene Ausprägungen haben. Daher können wir auch nicht pauschal sagen, wie gut oder effizient eine Datenstruktur für eine Operation geeignet ist. Wir können jedoch die grundsätzlichen Merkmale und Eigenschaften der Datenstrukturen verstehen, und uns fragen: Wie sollte sie am besten implementiert werden, um optimal mit unseren Anforderungen umgehen zu können?

In der folgenden Praxisübung werden wir daher auch nicht **die** Linked List implementieren, sondern eine Form, die gut zu unserem Szenario passt. Wir werden uns Gedanken darüber machen, welche Methoden wir benötigen, und wie wir sie möglichst effizient umsetzen können. Wir werden sehen: Manchmal kann ein zusätzliches Attribut in unserer Datenstruktur-Klasse unsere Operationen viel effizienter machen.

Merke:

- Datenstrukturen sind ein Konzept, wie Daten organisiert, gespeichert und manipuliert werden können.
- Das Array speichert Datenelemente als Block. Pointer-basierte Datenstrukturen verweisen mittels Pointer auf das folgende Element oder die folgenden Elemente.
- Die Wahl der Datenstruktur sollte den Bedürfnissen der konkreten Anwendung entsprechen. Insbesondere unterscheiden wir sequenzielle Datenstrukturen und Mengenstrukturen.
- Die konkrete Implementierung ein und derselben Datenstruktur kann sehr unterschiedlich gestaltet sein.