INFORMATIK

Algorithmen und Künstliche Intelligenz

Umbrüche und Laufweiten wie auf Cover – auf Muster warten

Vorabmaterial – nicht schlusskorrigiert
Stand 14.08.2023

Grundlagen der Informatik für Schweizer Maturitätsschulen

Dennis Komm, Juraj Hromkovič,
Regula Lacher, Harald Pierhöfer

Klett und Balmer Verlag

Dieses Material / Vorabdruck darf nur bis zum 31.10.2023 verwendet werden. Hernach steht das vollständige Lehrbuch zum käuflichen Erwerb bereit.»

Einleitung

Das zentrale Thema der Informatik ist die Automatisierung der Informationsverarbeitung. Die Automatisierung selbst ist so alt wie die ersten menschlichen Kulturen: Menschen haben Wissen erworben und es angewendet, um Vorgehensweisen zu entwickeln, mit denen sie die gewünschten Ziele erreichen konnte (z. B. ein Instrument, Transportmittel oder einen Gebrauchsgegenstand erstellen). Diese Vorgehensweisen konnten dann alle Menschen erfolgreich anwenden, ohne zu verstehen, warum sie funktionieren, also ohne das Wissen der Erfinder besitzen zu müssen. Nach ihrer Geburtsstunde setzt die Automatisierung ihren Erfolgszug fort und findet heute überall statt. Sie verleiht der Menschheit ihre hohe Effizienz.

Die ersten Algorithmen als mathematisch exakte Beschreibungen der Datenverarbeitung finden wir schon auf rund 4000 Jahre alten babylonischen Tontafeln. Die ältesten Algorithmen bezogen sich meist auf das Rechnen. Die bekanntesten historischen Beispiele sind die «ägyptische Multiplikation» (auch «äthiopische Multiplikation» genannt), die für eine effiziente Ausführung der Multiplikation ohne Stellenwertdarstellung der Zahlen sorgte, und der Euklidische Algorithmus, der zur Berechnung des grössten gemeinsamen Teilers zweier Zahlen verwendet wurde. Die ältesten Bücher, die Algorithmen präsentiert haben, sind das Buch «Elemente» von Euklid (ungefähr 300 Jahre v. Chr.) und das umfangreiche Buch zu mathematischen Lösungsmethoden von al-Chwarizmi aus dem neunten Jahrhundert, das das Rechnen und dadurch auch die Wirtschaft im Mittelalter in Europa geprägt hat. Weil das Buch das Mass des Rechenwesens vieler Jahrhunderte war, leitete man aus dem Namen des Autors den Begriff Algorithmus für die Bezeichnung von Methoden zur Lösung mathematischer Aufgaben.

Heute verwenden wir Algorithmen nicht nur für das Rechnen. Wir entwickeln z. B. Algorithmen zur übersichtlichen Datenorganisation für Suchalgorithmen, um schnell etwas Gesuchtes im Internet oder in Datenbanken finden zu können, auch Algorithmen zum Schachspielen oder – wie im Band «Data Science und Sicherheit» – zur Kodierung von Daten mit gewünschten Eigenschaften (Chiffrierungsalgorithmen, Komprimierungsalgorithmen usw.). Algorithmen können auch Autos oder ganze Produktionsanlagen steuern. Solche Algorithmen treffen anhand der aktuellen Daten Entscheidungen und steuern die Technik bei der Ausführung.

Vorabmaterial – nicht schlusskorrigiert
Stand 14.08.2023

Das Lösen von Problemen kann man allgemein als Vorgehensweise zum Extrahieren (Berechnen) gewünschter Informationen aus vorhandenen Daten ansehen: Eine solche Problemstellung bestünde beispielsweise darin, die kürzeste Strassenverbindung zwischen A und B zu bestimmen, wenn ein Strassennetz mit Entfernungsangaben zwischen allen Paaren direkt durch Strassen verbundener Ortschaften gegeben ist. Andere, komplexere Algorithmen verwendet man im Bereich des maschinellen Lernens bei der künstlichen Intelligenz. Diese Algorithmen simulieren den Lernprozess durch das Trainieren an vorhandenen Daten mit dem Ziel, Expertise in einem Gebiet (Diagnostik, Schachspiel, Mustererkennung, Übersetzung usw.) zu erwerben, um danach die Tätigkeit eines Experten zu übernehmen. Die Vielfalt der Automatisierung mit Algorithmen kennt keine Grenzen.

Die höchste Kunst der Informatik ist, Algorithmen zu entwickeln, die gewünschte Ziele zuverlässig und effizient erreichen. Dieser kreativen Tätigkeit ist das vorliegende Lehrmittel gewidmet. Dabei geht es nicht nur darum, durch Probieren Lösungswege zu finden, sondern allgemeine Strategien für den Algorithmenentwurf kennenzulernen, die bei einer grossen Vielfalt von unterschiedlichen Problemstellungen zur erfolgreichen Entwicklung von Lösungsmethoden führen.

Dieses Material / Vorabdruck darf nur bis zum 31.10.2023 verwendet werden. Hernach steht das vollständige Lehrbuch zum käuflichen Erwerb bereit.»

Inhalt

Das erste Kapitel ist dem ewigen Thema «Ordnung und Suche» gewidmet, das an der Grenze zwischen den zwei Grundbereichen der Informatik «Data Science» und «Algorithmik» steht. Die ursprüngliche Fragestellung klingt ganz einfach: «Wie organisiere ich meine Datensammlung so, dass ich immer alles schnell finde?». Diese Frage in der Datenverwaltung zu beantworten ist alles, nur nicht einfach, weil es um die Effizienz (den Rechenaufwand) geht. Jede Erstellung einer systematischen Ordnung in Daten kostet einen gewissen Rechenaufwand, dasselbe gilt auch für jede Suche darin. Die Investition in die Ordnung muss sich bei der Suche auszahlen, also viele Suchaktivitäten müssen den Aufwand für die Ordnungserstellung kompensieren (amortisieren). Zusätzlich kommen die Kosten für die Ordnungshaltung hinzu, falls sich die Datensammlung dynamisch verändert. Die Ordnungshaltungskosten wachsen mit der Häufigkeit der Updates der Datensammlung. Deswegen gibt es auch keine universelle Antwort auf die Frage nach einer optimalen Datenorganisation und dem besten Suchalgorithmus. In diesem Kapitel lernen Sie drei unterschiedliche Konzepte zur Suche und entsprechenden Datenorganisation kennen, nämlich die binäre Suche, die eine vollständige lineare Ordnung verlangt, die Suchbäume als partielle Ordnung sowie das Hashing. Welche Vorgehensweise für eine konkrete Datenverwaltung vorteilhaft ist, hängt hauptsächlich davon ab, wie oft sich die vorhandene Datensammlung ändert.

Kapitel 2 ist den mathematischen Grundlagen gewidmet, die man als Basisinstrumente braucht, wenn man Algorithmen entwickeln, überprüfen und analysieren will. In der Kombinatorik lernen Sie, wie man alle Objekte mit gegebenen Eigenschaften aufzählen oder systematisch auflisten kann. Dies dient unter anderem auch zur Auflistung aller möglichen Lösungen eines algorithmischen Problems mit dem Ziel, unter den Lösungen die für uns geeignetste auszusuchen. Zusätzlich werden Sie die über 2000 Jahre alte Methode der «konstruktiven Induktion» als Forschungsinstrument innerhalb der Mathematik kennenlernen. Sie lernen dann auch deren «jüngere Schwester» die «vollständige Induktion» erfolgreich für das Lösen von unterschiedlichen mathematischen Herausforderungen anzuwenden.

Kapitel 3 ist der kreativen Tätigkeit des Algorithmenentwurfs gewidmet. Sie entdecken hier die unglaubliche Stärke der Induktion beim Lösen algorithmischer Problemen. Die Vielfalt der erfolgreichen Anwendungen der Induktion als Instrument für die Forschung und Gestaltung ist faszinierend und Sie lernen die Induktion zur Entwicklung von Lösungsmethoden für unterschiedliche Problemstellungen und Rätselaufgaben kennen. Dazu gehören unter anderem Sortieralgorithmen, optimale Rechenverfahren, die Suche nach Gewinnstrategien für endliche Spiele, die Färbung von Landkarten und unterschiedliche Optimierungsprobleme wie die Suche nach kürzesten Wegen.

Die künstliche Intelligenz (KI) erstaunt heute die ganze Menschheit mit ihren Anwendungen. Unter künstlicher Intelligenz betrachten wir alle IT-Systeme, die Tätigkeiten ausüben, die wir eher nur den Menschen zugetraut hätten. In Kapitel 4 fokussieren wir uns auf das maschinelle Lernen, das der künstlichen Intelligenz den Durchbruch vor ein paar Jahren ermöglicht hat. Beim maschinellen Lernen geht es nicht direkt um das Lösen von Problemen, sondern um die Programmierung von Lernalgorithmen, die mittels eines meist umfangreichen Trainings lernen. Das Produkt ihres Lernprozesses ist eine hohe Expertise in einem Bereich, die dem Vergleich mit führenden menschlichen Experten standhält und deren Expertise nicht selten sogar übertrifft.

Vorabmaterial – nicht schlusskorrigiert
Stand 14.08.2023

Dieses Material / Vorabdruck darf nur bis zum 31.10.2023 verwendet werden. Hernach steht das vollständige Lehrbuch zum käuflichen Erwerb bereit.»

Sie lernen hier anhand der endlichen Spiele Lernalgorithmen anzuwenden, die durch ein Spieltraining die Fähigkeit erwerben, optimal zu spielen. Sie entzaubern damit das scheinbare Wunder der künstlichen Intelligenz, indem Sie begreifen, wie Sie selbst mit Lernalgorithmen Expertensysteme mit künstlicher Intelligenz bauen können.

Lernen und Lehren mit diesem Buch

Dieses Lehrmittel unterscheidet sich wesentlich von anderen Unterrichtsmaterialien und Lernumgebungen. Es ist nicht nur mit den neuesten Erkenntnissen der Lehr- und Lernforschung abgestimmt, sondern auch darauf ausgerichtet, dass man sich selbstständig mit hoher Nachhaltigkeit das ganze zu vermittelnde Wissen aneignen kann. Wir erreichen dies, weil wir nicht die fertigen Produkte der Wissenschaft wie Fakten, Modelle und Methoden sowie deren Handhabung unterrichten, sondern Ihnen die Gelegenheit bieten, so weit wie möglich die kreativen Entdeckungsprozesse zu erleben, die zur Erzeugung des angestrebten Wissens geführt und die Gestaltung der neuen Technologien und die damit verbundenen Produkte ermöglicht haben. Ein solcher Lernprozess ist motivierender und attraktiver, vor allem aber auch viel verständlicher und fantasievoller und gewährleistet eine hohe Erfolgsgarantie. Derartiges Lehren und Lernen gibt Ihnen im grossen Mass das, was man von der Bildung erwartet: eine Chance für alle, sich vielseitig zu entwickeln und eigenes intellektuelles Potenzial und Kreativität zu entfalten sowie zu lernen, vorhandene Konzepte als Werkzeug zur Erforschung und zur Gestaltung der Welt anzuwenden. Werden Sie also aktiv – je mehr, desto interessanter wird das Lernen.

Wie kann man dieses Ziel erreichen? Wir müssen das Konzept der falschen Effizienz aufgeben, bei dem um viele Tatsachen zu lernen, der Unterricht überwiegend auf die Bekanntmachung und den Umgang mit Produkten der Wissenschaft und Technologie reduziert wird (z. B. Fakten, Modelle, Methoden). Diese Oberflächlichkeit überwindet man mit der historischen Methode, die aus den Versuchen der vergangenen Zeit ihre Kreativität schöpft: Hier geht es nicht nur darum, komplexe Inhalte durch eine Folge von kleinen und verständlichen Schritten eigenständig zu erforschen, sondern durchaus auch darum, Fehler (die die Erfinder auch gemacht haben) zu machen und aus ihnen zu lernen. Die Wissenschaft ist eine Ausdauerübung mit vielen gescheiterten Versuchen, die uns Stück für Stück klüger machen, bis wir daraus so viel erkennen, dass wir unser Ziel erreichen. Dies ist auch das Einzige, was man als «natürlichen» Lernprozess bezeichnen darf, in dem es nicht nur um das Auswendiglernen geht.

Vorabmaterial – nicht schlusskorrigiert
Stand 14.08.2023

Elemente des Lehrmittels

Alle Elemente des Lehrmittels unterstützen didaktisch unseren Weg in die Welt der Algorithmen. Alle Themen werden durch eine «Knobelaufgabe» eröffnet (gekennzeichnet als Aufgabe mit einer Glühbirne ). Man muss nicht unbedingt für jede Knobelaufgabe eine pfiffige Idee finden, um sie zu lösen. Die Zielsetzung dieser Knobelaufgaben ist, die bevorstehende Herausforderung zu verstehen und das Interesse an der Lösung zu wecken. «Beispiele» werden genutzt, um kleine Vorwärtsschritte zu erklären oder die verwendeten Lösungsansätze zu einer Lösungsstrategie zu verallgemeinern. Dem Element «Neue Konzepte und Begriffe» kommt eine Schlüsselrolle zu. Hier wird die genaue Bedeutung der neuen Konzepte festgehalten, ihr Potenzial und ihre Grenzen angesprochen und die Fachsprache schrittweise eingeführt. Verallgemeinernd und zusammenfassend ist dieses Element jeweils Voraussetzung für die weitere Entwicklung. Die «Lern- und Projektaufgaben»

Dieses Material / Vorabdruck darf nur bis zum 31.10.2023 verwendet werden. Hernach steht das vollständige Lehrbuch zum käuflichen Erwerb bereit.»

(gekennzeichnet mit einem ⚗) vertiefen entweder das Thema für Interessierte oder ermöglichen, selbstständig im Thema einen Schritt weiterzukommen, der unausweichlich für die Fortsetzung des Lernens im nachfolgenden Teil ist. Das Wichtigste ist aber, dass alle Lösungen sowie nützliche Gedanken und unterschiedliche Lösungswege zu den Aufgaben im Buch als PDF via digiMedia auf meinklett.ch zur Verfügung stehen. Scheitern oder Ratlosigkeit beim Lösen von Aufgaben sind also ausgeschlossen. Als weiteres Element werden auch «klassische» Aufgaben» angeboten (gekennzeichnet als ✎), die zum Üben und Festigen des Stoffes in kleinen Schritten dienen. Das letzte zu erwähnende Element verdankt sich der oben erwähnten historischen Methode. Mit eigenständigen Beiträgen als «Geschichtlicher und gesellschaftlicher Kontext» wird die Geschichte der entsprechenden Teilgebiete der Informatik skizziert und damit werden Motivationen und Erforschungsbemühungen im historischen Kontext dargestellt. Dadurch wird auch der gesellschaftlichen Relevanz der Themen Raum gegeben ebenso wie Konsequenzen für unsere Haltung zu zentralen Themen wie Datenverwaltung und Einsatz von Systemen mit KI, künstliche Intelligenz usw. angesprochen.

Alle Kapitel sind in zielgerichtete Abschnitte unterteilt. Die einzelnen Abschnitte enden mit dem Element «Was Sie gelernt haben», in welchem das konzeptuell Wichtigste des Abschnitts zusammengefasst ist. Drei weitere Elemente unterstützen Sie zusätzlich am Ende jedes Kapitels auf Ihrer Reise. In der «Zusammenfassung» wird im allgemeinen Kontext das erworbene Wissen resümiert. Hier werden nochmals die wichtigen Begriffe und Konzepte angesprochen, deren Stärken und Schwächen thematisiert und, nach Möglichkeit, ein Ausblick für eine Vertiefung vermittelt. Damit werden die letzten beiden Elemente zum Kapitelende unter der Überschrift «Testen Sie sich selbst» vorbereitet: die selbstständige Überprüfung des eigenen Wissens und der erreichten Kompetenzen. Mit den «Fragen» überprüfen Sie Ihr Verständnis der eingeführten Konzepte, Methoden und Begriffe ebenso wie die Beziehungen zwischen diesen. Die Antworten sind absichtlich nicht im Online-Lösungsbuch formuliert, weil Sie aufgefordert sind, Ihre Antworten in den Elementen «Neue Konzepte und Begriffe» und «Was Sie gelernt haben» zu suchen und so zu repetieren. Unter den «Kontrollaufgaben» bearbeiten Sie zwei Aufgabentypen: einerseits solche, die Sie mit den bearbeiteten Konzepten bereits lösen können sollten, und andererseits kontextuelle Aufgaben, welche die Einbindung mehrerer Konzepte oder Methoden aus dem Kapitel für die Lösungssuche erfordern.

Vorabmaterial – nicht schlusskorrigiert
Stand 14.08.2023

Zusätzlich zu den Lösungen als PDF werden Ihnen für unterschiedliche Themenbereiche interaktive Online-Lernumgebungen angeboten, die umfangreicheres Übungsmaterial zur Verfügung stellen. Alle Lösungsversuche erhalten sofort ein automatisches Feedback, das häufig über «richtig» oder «falsch» hinausgeht. Die Links zu den interaktiven Lernumgebungen finden Sie unter klett-online.ch.

Wir wünschen Ihnen viel Erfolg auf der «Entdeckungsreise Informatik» und insbesondere, dass es Ihnen gelingt, informatische Konzepte als Instrumente zur Gestaltung der Welt und zur Generierung von neuem Wissen zu nutzen.

Die Autorinnen und Autoren

Dieses Material / Vorabdruck darf nur bis zum 31.10.2023 verwendet werden. Hernach steht das vollständige Lehrbuch zum käuflichen Erwerb bereit.»

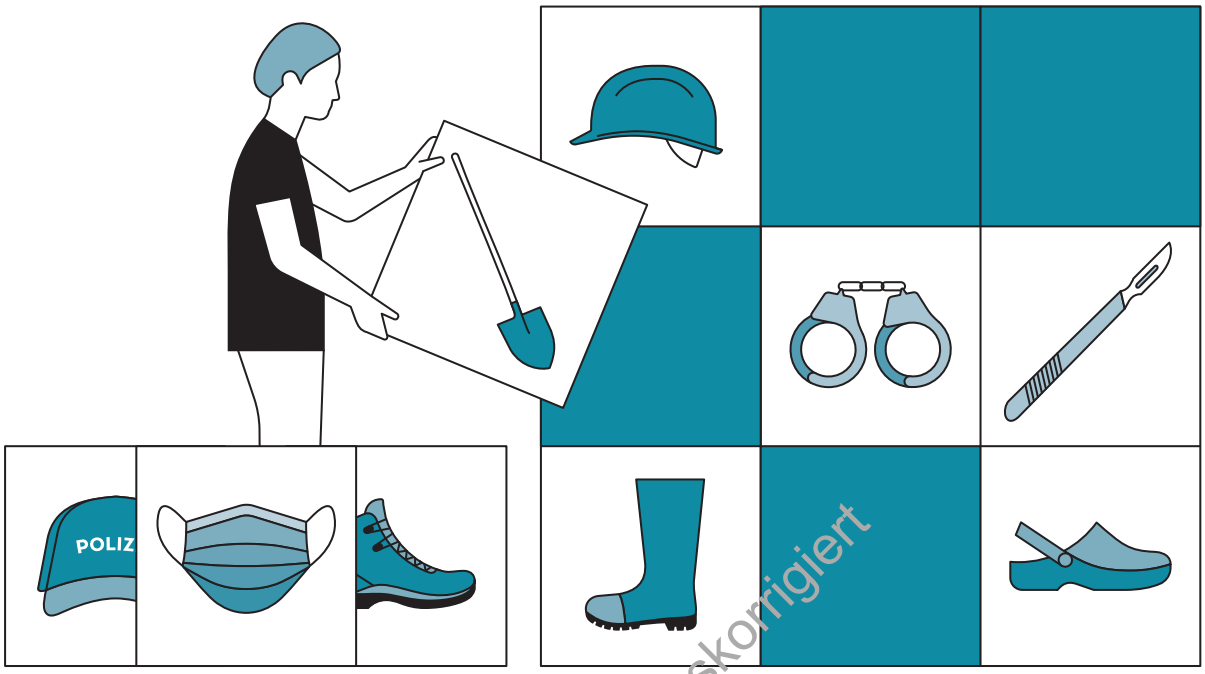
Vorabmaterial – nicht schlusskorrigiert
Stand 14.08.2023

Dieses Material / Vorabdruck darf nur bis zum 31.10.2023 verwendet werden. Hernach steht das vollständige Lehrbuch zum käuflichen Erwerb bereit.»

Inhaltsverzeichnis

Vorabmaterial – nicht schlusskorrigiert
Stand 14.08.2023

Dieses Material / Vorabdruck darf nur bis zum 31.10.2023 verwendet werden. Hernach steht das vollständige Lehrbuch zum käuflichen Erwerb bereit.»



Vorabmaterial – nicht schlusskorrigiert
Stand 14.08.2023

1 Ordnung und Suche

Die Suche nach Informationen und somit nach Daten in Informationssystemen ist eine Aktivität, die unsere Computer und Computernetzwerke mindestens so sehr beschäftigt wie das Rechnen selbst. Deswegen gehört das schnelle Finden des Gesuchten zu den zentralen Aufgaben der Informatik. Etwas in riesigen Datensammlungen schnell zu finden, ist nicht ausschliesslich eine Frage der Entwicklung von effizienten Suchalgorithmen. Die Schlüsselrolle spielt hier die Organisation von Daten (d. h. die von uns eingeführte Ordnung von Daten), die massgebend bestimmt, ob überhaupt und in welchem Mass eine effiziente Suche möglich ist. Deswegen liegt dieses Thema an der Grenze zwischen den zwei Basiskonzepten der Informatik – Daten und Algorithmen. Die besten (effizientesten) Algorithmen funktionieren nur in Bezug auf konkrete Datenorganisationen.

Eine Ordnung für Daten einzuführen und zu halten ist aber auch mit einem gewissen Aufwand verbunden. Deswegen bedeutet das Minimieren des Gesamtaufwands, dass die Summe der Aufwände für Ordnungsherstellung, Ordnungshaltung und Suchaufwand minimiert werden muss. Zu dieser komplexen Problemstellung gibt es keine eindeutige, klare Vorgehensweise. Sie lernen, dass die Wahl einer geeigneten Lösungsstrategie davon abhängt, wie oft gesucht wird (ob sich der Aufwand für die Ordnungsherstellung durch mehrfache Suche amortisiert) und wie oft sich die Datensammlung verändert (wie gross der Aufwand für die Haltung der Ordnung ist). Betrachten wir folgendes Beispiel: In einem Regal stehen 100 Bücher in zufälliger Reihenfolge. Wenn Sie genau wissen, dass Sie ein einziges Mal ein Buch darin suchen werden, dann verzichten Sie aufs Ordnen der Bücher, und bei der Suche ist es sinnvoll, von einer Seite her durch die Reihe von Büchern zu gehen, bis Sie das eine gefunden haben. Wenn Sie aber damit rechnen, öfter unterschiedliche Bücher zu suchen, nehmen Sie sicher gerne die einmalige Arbeit auf sich, die Bücher beispielsweise alphabetisch nach ihrem Titel zu ordnen. In dieser Ordnung können Sie dann fast direkt auf das gesuchte Buch zugreifen. Wenn Sie ein neues Buch hinzufügen wollen, müssen Sie das Buch aber an der richtigen Stelle einfügen, um die Ordnung zu halten.

In diesem Kapitel lernen Sie auch, wie man dank Objektorientierung in Python die zur Umsetzung der Ordnung verwendeten Datenstrukturen bauen

Dieses Material / Vorabdruck darf nur bis zum 31.10.2023 verwendet werden. Hernach steht das vollständige Lehrbuch zum käuflichen Erwerb bereit.»

und somit die dynamische Datenverwaltung automatisieren kann.
Zudem lernen Sie, abhängig von der Dynamik der Veränderung der Datensammlung, welche Möglichkeiten für die Wahl einer geeigneten Ordnung bestehen und wie dann eine effiziente Suche in dieser Datenorganisation aussehen kann.

Lineare Ordnung und binäre Suche

Stellen Sie sich vor, Sie haben eine Datensammlung, die sich in naher Zukunft nicht ändern wird oder sehr selten modifiziert wird (eine alte Datei wird gelöscht oder eine neue hinzugefügt). In so einer stabilen Situation ziehen es Informatikerinnen und Informatiker vor, ein Kriterium zu finden, nach dem sie die Daten vollständig **sortieren** können.

Neue Konzepte und Begriffe

Ein Kriterium für eine Menge A von Objekten ermöglicht eine **lineare Ordnung** (auch **totale Ordnung** genannt), wenn jeweils zwei Elemente der Menge A nach diesem Kriterium mit der Relation (Beziehung) «kleiner gleich» vergleichbar sind. Zum Beispiel hat eine Menge von Zahlen mit der Relation «≤» eine lineare Ordnung, d. h., man kann alle Zahlen in eine nicht absteigende Reihenfolge bringen, so dass jede Zahl a kleiner gleich ist als alle Zahlen rechts von a in der Reihenfolge. Für eine Datensammlung kann man die Dateinamen nehmen und diese lexikographisch wie in einem Wörterbuch ordnen. Graphen kann man z. B. nach der Anzahl der Knoten oder nach der Anzahl der Kanten ordnen. Wenn ein Kriterium zwei Objekten denselben Wert zuordnet (d. h., beide Objekte sind gleich bezüglich des Kriteriums), spielt es keine Rolle, in welcher Reihenfolge die zwei Objekte aufgelistet werden.
Wenn wir eine Menge A mit einer linearen Ordnung haben und alle Elemente von A paarweise unterschiedliche Werte durch das Kriterium der Ordung besitzen, können wir die Elemente von A in einer aufsteigenden Reihenfolge auflisten oder anders formuliert für jedes Element seine Ordnung (erstes, zweites, drittes, …) in dieser Reihenfolge bestimmen. Wenn a «kleiner gleich» b ist, dann sagen wir, dass b «grösser gleich» a ist.
Als ein **Minimum** einer Menge bezüglich einer Relation «kleiner gleich» bezeichnet man jedes Element, das kleiner gleich aller anderen Elemente der Menge ist. Analog ist ein **Maximum** einer Menge jedes Element, das grösser gleich alle anderen Elemente der Menge ist.

Vorabmaterial – nicht schlusskorrigiert
Stand 14.08.2023

1.1 Finden Sie für die folgenden zwei Mengen jeweils drei unterschiedliche Kriterien, die eine lineare Ordnung definieren und zu drei unterschiedlichen nicht absteigenden Folgen führen.

a {5, −6, 3, −3, 0, 2, −1, 7}

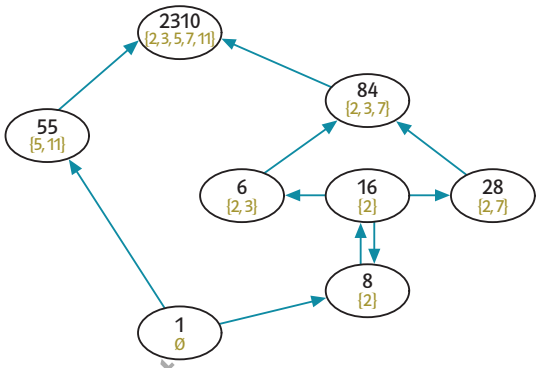
b {Zoo, Bergrestaurant, Naturmuseum, freie Wölfe und Bären, Spielplatz}

Nichtlineare Ordnungen

Beispiel 1.1 Nicht alle Kriterien führen zu einer linearen Ordnung, also sind lineare Ordnungen keine Selbstverständlichkeit. Wir illustrieren dies hier für natürliche Zahlen. Wir sagen, a ist kleiner gleich b bezüglich der Faktorisierung, wenn alle Primfaktoren von a auch Primfaktoren von b sind. Somit ist 84 kleiner gleich 210 bezüglich der Faktorisierung, weil $84 = 2 \cdot 2 \cdot 3 \cdot 7$ und $210 = 2 \cdot 3 \cdot 5 \cdot 7$ und die Menge {2, 3, 7} von Primfaktoren von 84 somit eine Teilmenge der Menge {2, 3, 5, 7} der Primfaktoren von 210 ist. Andererseits sind die Zahlen $66 = 2 \cdot 3 \cdot 11$ und $350 = 2 \cdot 5 \cdot 5 \cdot 7$ nicht vergleichbar, weil keine der Mengen {2, 3, 11} und {2, 5, 7} eine Teilmenge der anderen Menge ist.
Wie können wir dann die Menge {1, 6, 8, 16, 28, 84, 55, 2310} ordnen und insbesondere diese Ordnung anschaulich darstellen?

Dieses Material / Vorabdruck darf nur bis zum 31.10.2023 verwendet werden. Hernach steht das vollständige Lehrbuch zum käuflichen Erwerb bereit.»

Wir verwenden dazu einen gerichteten Graphen, weil Graphen ein geeignetes Instrument für die transparente Darstellung der Beziehungen zwischen Paaren von Objekten sind. Die Knoten sind die jeweiligen Zahlen (schwarz) und ihre Menge von Primfaktoren (olivgrün). Ein gerichteter Weg führt von einem Knoten mit der Zahl a zu einem Knoten mit der Zahl b, wenn a kleiner gleich b bezüglich der Faktorisierung ist. Der gerichtete Graph für unsere Zahlenmenge {1, 6, 8, 16, 28, 84, 55, 2310} ist rechts abgebildet. Wenn zwischen zwei Knoten kein gerichteter Weg existiert, dann sind die entsprechenden Zahlen bezüglich unseres Kriteriums nicht vergleichbar. Zum Beispiel sind 55 und 28 nicht vergleichbar. Wir legen eine gerichtete Kante von a nach b, wenn a «kleiner gleich» b ist und es keine andere Zahl c gibt, sodass a kleiner gleich c ist und c kleiner gleich b ist.



1.2 Bestimmen Sie alle Paare von Zahlen aus der Abbildung in Beispiel 1.1, die nach der Relation «kleiner gleich bezüglich der Faktorisierung» nicht vergleichbar sind.

1.3 Zeichnen Sie einen gerichteten Graphen (wie in Beispiel 1.1) für die Zahlenmenge {7, 49, 70, 91, 98, 175}, um die Relation «kleiner gleich bezüglich der Faktorisierung» darzustellen.

1.4 Ändern Sie die Relation «kleiner gleich bezüglich der Faktorisierung» aus Beispiel 1.1 wie folgt. Eine natürliche Zahl a ist kleiner gleich b, wenn jeder Primfaktor von a in der Primfaktorzerlegung von b mindestens so oft vorkommt wie in der Primfaktorzerlegung von a; zum Beispiel ist

$$20 = 2 \cdot 2 \cdot 5 \text{ «kleiner gleich» } 60 = 2 \cdot 2 \cdot 3 \cdot 5.$$

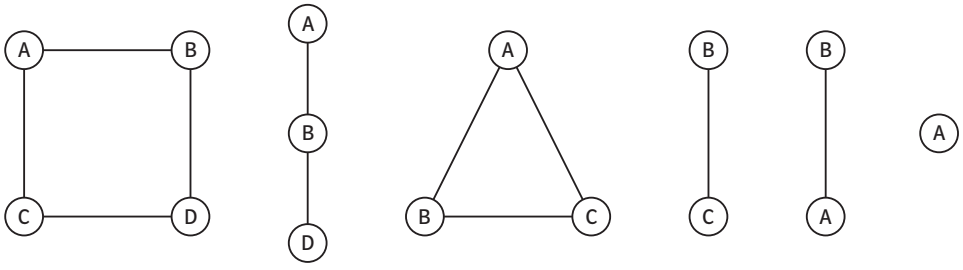
Jedoch ist 20 nicht vergleichbar mit

$$70 = 2 \cdot 5 \cdot 7,$$

weil der Faktor 2 in der Zerlegung von 20 zweimal vorkommt und in der Zerlegung von 70 nur einmal. Auf der anderen Seite ist 7 ein Primfaktor von 70, den 20 nicht besitzt. Zeichnen Sie für die Menge aus Beispiel 1.1 den zugeordneten gerichteten Graphen für diese modifizierte «kleiner gleich»-Relation.

1.5 Welchen Typ von Relation würden wir erhalten, wenn wir die Relation «kleiner gleich bezüglich der Faktorisierung» wie folgt ändern? Eine natürliche Zahl a ist kleiner gleich b, wenn die Anzahl der Primfaktoren von a kleiner gleich der Anzahl von Primfaktoren von b ist. Hier zählt jeder Primfaktor nur einmal, auch wenn er in der Zerlegung mehrfach vorkommt. Zum Beispiel hat $100 = 2 \cdot 2 \cdot 5 \cdot 5$ zwei Primfaktoren, 2 und 5. Wie würde der entsprechende gerichtete Graph für die Menge aus Beispiel 1.1 aussehen?

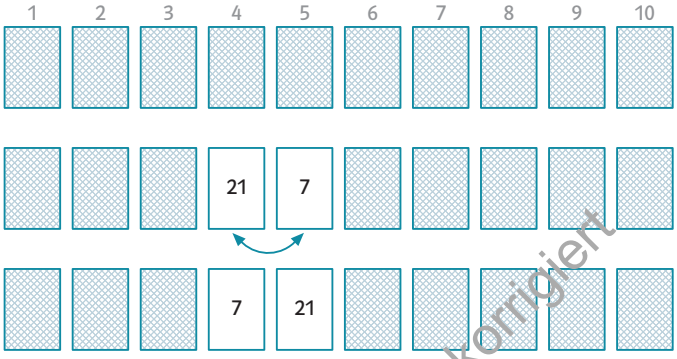
1.6 Man definiert die Relation «kleiner gleich» auf Graphen wie folgt. Ein Graph G_1 ist kleiner gleich G_2, wenn G_1 ein Teilgraph (Kapitel 1, Band «Data Science und Sicherheit») von G_2 ist. Stellen Sie die Beziehungen zwischen den folgenden sechs zusammenhängenden Graphen mit einem gerichteten Graphen (wie in Beispiel 1.1) dar. Die Knoten des gerichteten Graphen sollen diese sechs Graphen sein.



Vorabmaterial – nicht schlusskorrigiert
Stand 14.08.2023

Dieses Material / Vorabdruck darf nur bis zum 31.10.2023 verwendet werden. Hernach steht das vollständige Lehrbuch zum käuflichen Erwerb bereit.»

1.7 Man hat zehn Karten. Auf der Vorderseite stehen unterschiedliche natürliche Zahlen. Die Rückseiten aller zehn Karten ist gleich, also nicht unterscheidbar. Die Karten liegen in einer Reihe verdeckt und unsortiert in einer beliebigen Reihenfolge. Die einzige erlaubte Operation ist, zwei nebeneinanderliegende Karten aufzudecken und ihre Zahlen zu vergleichen. Falls sie der aufsteigenden Ordnung entsprechen, dreht man sie einfach wieder um (die Karten werden nach der Ausführung der Operation wieder verdeckt). Falls die Reihenfolge nicht stimmt, tauscht man die Karten auf ihren Positionen aus und dreht beide wieder um.



a Wie viele solcher Operationen reichen Ihnen, um die Karte mit der grössten Zahl ganz rechts zu platzieren und somit das Maximum zu bestimmen?

b Wie viele solcher Operationen reichen Ihnen, um die verdeckten Karten nach ihren Zahlen zu sortieren? Ihre Strategie muss funktionieren, egal welche Zahlen auf den Karten stehen.

Vorabmaterial – nicht schlusskorrigiert
Stand 14.08.2023

Beispiel 1.2

Bubblesort

Wenn man mit dem Computer eine unsortierte Folge von Zahlen sortieren will, steht man vor einer ähnlichen Herausforderung wie in Knobelaufgabe 1.7. Die einzelnen Zahlen sind in einzelnen Zellen des Computerspeichers gespeichert. Der Computer «sieht» die Speicherinhalte genauso wenig, wie wir die Zahlen von verdeckten Karten sehen. Die einzelnen Zellen des Computerspeichers sind nummeriert, 0, 1, 2, 3, … und diese Nummern heissen **Adressen** der Speicherzellen – analog zu den verdeckten Karten in Knobelaufgabe 1.7 auf den Positionen 1 bis 10.
Dank dieser Positionierung der Karten sind wir fähig, unsere Sortierstrategie zu beschreiben und konkrete Karten «anzusprechen». Genauso kann der Computer mittels der Adressen auch den Inhalt jeder Speicherzelle anschauen und bearbeiten. Der einzige Unterschied zu unseren Regeln in Knobelaufgabe 1.7 ist, dass der Computer nicht eingeschränkt ist, nur Inhalte von zwei benachbarten Zellen zu vergleichen. So wie wir auch Karten, die nicht nebeneinander liegen, aufdecken können, kann der Computer zwei beliebige Speicherstellen vergleichen.

Wenn der Computer wie vorgestellt nur anhand von Vergleichen von Inhalten von zwei benachbarten Zellen sortiert, spricht man von **Bubblesort** (Kapitel 5, Band «Programmieren und Robotik»). Das Wort «bubble» kommt aus dem Englischen, bedeutet Blase und erinnert an Luftblasen, die in einer Flüssigkeit aufsteigen. Hier bewegen sich die grösseren Werte Schritt für Schritt nach oben (was in einer Zahlenfolge nach rechts entspricht), bis sie ihre definitive Position erreichen.
Unsere Strategie aus der Lösung von Knobelaufgabe 1.7 kann man für eine beliebige Anzahl von n Karten (Speicherzellen) verallgemeinern. Man bestimmt den grössten Wert und platziert ihn in der Speicherzelle mit der höchsten Adresse mit $n - 1$ Vergleichen. Die Voraussetzung ist dabei natürlich, dass die n Zahlen in n nebeneinanderstehenden Speicherzellen abgelegt sind. Für den zweitgrössten Wert brauchen wir dann $n - 2$ Vergleiche und so weiter. Insgesamt sortiert Bubblesort mit
$(n - 1) + (n - 2) + (n - 3) + \ldots + 3 + 2 + 1$
Vergleichen. Wie wir in Kapitel 2 analysieren werden, ist diese Summe gleich $n \cdot (n - 1) / 2$.

Dieses Material / Vorabdruck darf nur bis zum 31.10.2023 verwendet werden. Hernach steht das vollständige Lehrbuch zum käuflichen Erwerb bereit.»

1.8 Es kann vorkommen, dass Bubblesort immer noch arbeitet, obwohl die Zahlenfolge bereits aufsteigend sortiert ist. Im Extremfall liegt bereits ganz am Anfang eine aufsteigend sortierte Folge vor und Bubblesort führt die ganze Zeit Vergleiche ohne eine einzige Vertauschung von Nachbarn aus.

a Wie kann man bemerken, dass die Zahlenfolge bereits sortiert ist, und wie kann man die Ausführung von Bubblesort vorzeitig stoppen?

b Finden Sie eine Eingabe (Zahlenfolge) für Bubblesort, so dass der Algorithmus unvermeidlich alle $1 + 2 + 3 + \ldots + (n - 1)$ Vergleiche ausführen muss, um eine aufsteigend sortierte Folge zu erhalten.

Neue Konzepte und Begriffe

Wenn wir Daten verwalten, können die einzelnen Elemente unserer Datensammlung unterschiedliche Objekte unterschiedlicher Grösse sein. Es können ganze Dateien, Datensätze (Kapitel 5, Band «Data Science und Sicherheit») oder einfach nur Zahlen sein. Um unsere Datensammlung zu verwalten (z. B. sortieren, suchen, updaten), brauchen wir zuerst eine lineare Ordnung auf ihren Elementen. Deswegen ordnen wir jedem Element unserer Datensammlung einen Identitätsausweis genannt **Schlüssel** zu. Schlüssel können Zahlen oder Wörter sein, einfach Objekte, die für natürliche Kriterien eine lineare Ordnung ermöglichen. Ein Schlüssel muss eine echte Identifikation ermöglichen, d. h., zwei unterschiedlichen Objekten einer Datensammlung sollen unterschiedliche Schlüssel zugeordnet sein. Als Schlüssel kann man z. B. den Wert eines Attributs eines Datensatzes, den Namen der Datei oder eine Zahl wählen, die man leicht aus den Informationen im Datenelement ausrechnen kann.
Die Verwendung von Schlüsseln ermöglicht uns einerseits eine lineare Ordnung auf den Elementen unserer Datensammlung und andererseits eine Abstraktion, dank der die Algorithmen der Datenverwaltung sich auf die Schlüsselverwaltung reduzieren.
Wenn wir hier mit Zahlen oder Namen (Wörtern) arbeiten (sortieren, suchen), dann stehen diese Zahlen und Namen als Schlüssel (Repräsentanten) für beliebig komplexe Elemente von Datensammlungen.

Vorabmaterial – nicht schlusskorrigiert
Stand 14.08.2023

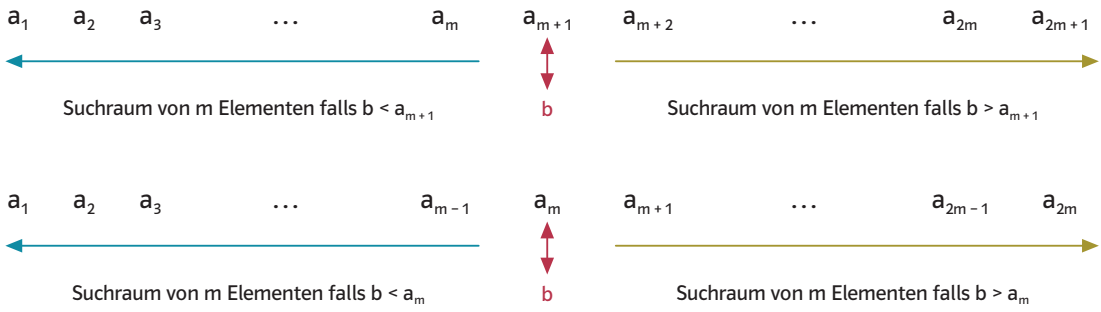
1.9 Ähnlich wie in Knobelaufgabe 1.7 haben wir eine Folge von diesmal 15 verdeckten Karten. Wir wissen zwar, dass diese Karten bereits nach ihren Zahlen aufsteigend sortiert sind, aber nicht, welche Zahlen auf den Karten stehen. Von einer Karte ist jedoch bekannt, dass sie die Zahl 113 enthalten muss. Die einzige erlaubte Operation ist nun, eine Karte umzudrehen und sich den dort stehenden Wert anzuschauen und mit 113 zu vergleichen. Finden Sie eine Strategie, mit der Sie so wenige Karten wie möglich umdrehen müssen, um die Karte mit der Zahl 113 zu finden. Wie viele Umdrehoperationen reichen immer, egal wo die Karte mit der Zahl 113 liegt?

Binäre Suche

Beispiel 1.3 Die **binäre Suche** ist die effizienteste Suchstrategie in sortierten Folgen (Kapitel 5, Band «Programmieren und Robotik»). Man verwendete die binäre Suche schon lange bevor die ersten Computer gebaut wurden, z. B. für die Suche in Wörterbüchern oder Enzyklopädien, die lexikographisch sortiert sind. Das Prinzip ist, durch eine Anfrage (oder einen Vergleich) den Suchraum mindestens zu halbieren. Somit reichen bei einer sortierten Folge von n Elementen $\text{dislog}_2(n)$ Anfragen, bis der Suchraum aus einem einzigen Element besteht. (Schauen Sie sich für die Definition des diskreten Logarithmus das Kapitel 1, Band «Data Science und Sicherheit», an.)
Einen allgemeinen Schritt der Suche kann man wie folgt beschreiben: Wenn man einen aufsteigend sortierten Suchraum von n Elementen $a_1, a_2, \ldots, a_n$ ($a_i \leq a_{i+1}$ für $i = 1, \ldots, n - 1$) hat, vergleicht man das gesuchte b mit einem der mittleren Elemente des Suchraums.

Dieses Material / Vorabdruck darf nur bis zum 31.10.2023 verwendet werden. Hernach steht das vollständige Lehrbuch zum käuflichen Erwerb bereit.»

Falls n = 2m + 1 (falls n ungerade ist), ist das mittlere Element a_{m+1}.
Falls n = 2m (falls n gerade ist), hat man zwei mittlere Elemente a_m und a_{m+1}, und es spielt keine Rolle, welches der beiden man für den Vergleich mit dem gesuchten Element b nimmt.



Die obere Abbildung zeigt, dass sich für ungerade n = 2m + 1 der Suchraum auf die Grösse m < n/2 reduziert, falls das mittlere Element a_{m+1} nicht das gesuchte Element b ist. Die untere Abbildung zeigt, dass sich der Suchraum für gerade n = 2m entweder auf die Grösse m − 1 < n/2 oder auf die Grösse m = n/2 reduziert, falls a_m (welches wir in diesem Beispiel als «Mitte» gewählt haben) nicht das gesuchte Element b ist. Man wiederholt diesen Reduktionsschritt so lange, bis der Suchraum nur aus einem Element besteht. Falls man im Vorhinein weiss, dass sich b im ursprünglich gegebenen Suchraum befinden muss, ist die binäre Suche damit abgeschlossen. Falls es nicht bekannt ist, ob b sich im gegebenen Suchraum befindet, muss man sich das verbliebene Element im reduzierten Suchraum der Grösse 1 anschauen (mit b vergleichen). In diesem Fall bedeutet die binäre Suche $dislog_2(n) + 1$ Vergleiche.

Vorabmaterial – nicht schlussKorrigiert
Stand 14.08.2023

1.10 Gehen Sie in die interaktive Lernumgebung unter klett-online.ch und führen Sie rund 10 binäre Suchen für unterschiedlich grosse Suchräume aus.

1.11 Wie viele Vergleiche braucht die binäre Suche maximal, um ein darin vorkommendes Element in einem Suchraum der folgenden Grössen zu finden?

a 1000
b 10^6
c 10^9

Was Sie gelernt haben

Wenn man eine Menge von Elementen hat, kann man unterschiedliche Kriterien betrachten, um zwei Elemente zu vergleichen. Wenn ein Kriterium es ermöglicht, zwei beliebige Elemente der Menge zu vergleichen (in die Beziehung «kleiner gleich» zu setzen), dann erhalten wir eine lineare Ordnung auf dieser Menge. Wenn eine lineare Ordnung vorliegt, kann man die Elemente in eine aufsteigende oder absteigende Folge sortieren.
Elemente einer Menge oder einer unsortierten Folge zu sortieren, bedeutet eine totale Ordnung zu schaffen. Jedes Element erhält eine Zahl, die seiner Ordnung in der sortierten Folge entspricht.

Der Vorteil einer totalen Ordnung ist, dass man mit der binären Suche durch $dislog_2(n)$ Vergleiche im Suchraum der Grösse n (in einer sortierten Folge der Länge n) ein gesuchtes Element finden kann, wenn es sicher vorkommt. Wenn man häufig sucht, lohnt sich der Aufwand für die Sortierung. Ein sehr einfacher Sortieralgorithmus ist Bubblesort, der n Elemente mit $n \cdot (n - 1) / 2$ Vergleichen sortiert. Dabei verwendet Bubblesort nur eine einzige lokale Operation – einen Vergleich von zwei benachbarten Elementen der Folge und nach Bedarf das Vertauschen dieser beiden Elemente.

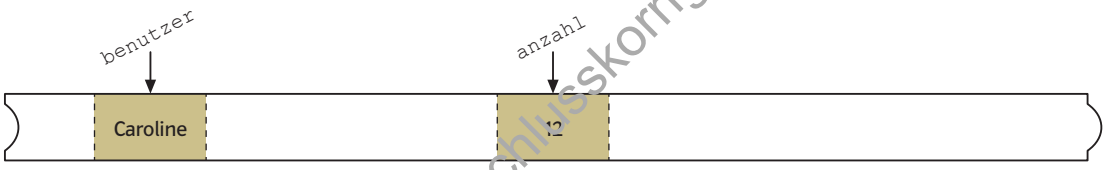
Dieses Material / Vorabdruck darf nur bis zum 31.10.2023 verwendet werden. Hernach steht das vollständige Lehrbuch zum käuflichen Erwerb bereit.»

Datenablage im Speicher mit Datenstrukturen – Arrays und Listen

Im vorherigen Unterkapitel haben wir die binäre Suche auf einer abstrakten Ebene vorgestellt. Um die binäre Suche so effizient wie gezeigt umzusetzen, ist es erforderlich, dass die sortierte Folge (Daten des Suchraums) in einer speziellen Struktur, genannt Array, gespeichert wird. In diesem Unterkapitel lernen Sie, dass Daten in Strukturen wie Arrays und verketteten Listen im Computer organisiert werden können und welche Vorteile und Nachteile diese beiden Datenstrukturen zur Organisation und Speicherung von Daten haben.
In den meisten Programmen sollen Werte während des Programmablaufs gespeichert werden, beispielsweise der Vorname der Benutzerin oder die Anzahl von kleinen Quadraten, die gezeichnet werden sollen. Dazu werden beim Programmieren Variablen eingeführt (Kapitel 3, Band «Programmieren und Robotik»), die einen Namen haben und einen Inhalt aufnehmen können, beispielsweise:

Variablenname	Inhalt
`benutzer`	Caroline
`anzahl`	12

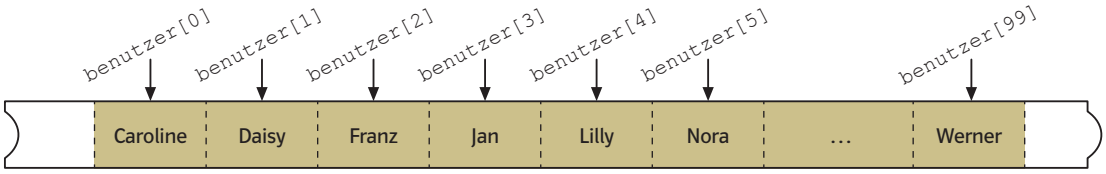
Beim Programmablauf weist die Speicherverwaltung jedem Variablennamen einen Zeiger zu, der an die Stelle im Speicher zeigt, an welcher das Datum (der Wert der Variablen) steht:



Vorabmaterial – nicht schlusskorrigiert
Stand 14.08.2023

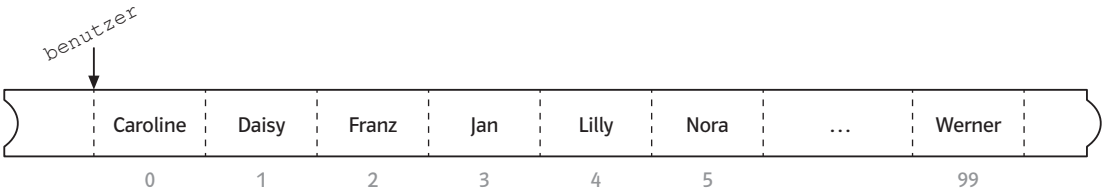
Die Zeiger als Pfeile in der Abbildung gibt es im Computer nicht wirklich, sie dienen nur der anschaulichen Darstellung des Zugriffs auf die Elemente der Datenstruktur. Ein Zeiger entspricht einer Adresse im Speicher, an der sich der Wert der Variablen befindet. Der Computer hat eine Tabelle mit allen verwendeten Variablen, in der für jede Variable die Adresse der Speicherzelle steht, in der sich der Wert der Variablen befindet. Wenn der Computer den Wert der Variablen braucht, schaut er sich in der Tabelle die Adresse an und geht in die entsprechende Speicherzelle, um den Wert der Variablen zu lesen oder diesen mit einem neuen Wert zu überschreiben.
Das ermöglicht dem Teil des Betriebssystems des Computers, der für die Speicherverwaltung zuständig ist, eine hohe Flexibilität. Wenn eine neue Variable initialisiert und verwendet wird, kann die Speicherverwaltung flexibel eine Adresse wählen, an deren Speicherzelle sich keine verwendeten Daten befinden (keine andere Variable zeigt auf diese Speicherzelle). Das befreit auch die Programmierenden vom Mikromanagement der Speicherverwaltung. Dank dem Konzept der Variablen müssen sie nicht wissen, wo welcher Wert im Speicher abgelegt ist.

Sind es nun aber sehr viele Daten, auf die zugegriffen werden soll, oder ist deren Anzahl beim Schreiben des Programms noch gar nicht klar, so funktioniert der Ansatz nicht mehr, jedes Datum über einen eigenen Variablennamen anzusprechen. Wir brauchen Datenstrukturen, die über einen Zeiger viele Daten ansprechen können (Kapitel 5, Band «Programmieren und Robotik»). Eine erste solche Datenstruktur ist das Array. Wird ein Array erzeugt, so muss neben dem Namen auch angegeben werden, wie viele Einträge das Array höchstens aufnehmen soll.
So kann beispielsweise ein Array mit dem Namen `benutzer` erzeugt werden, das 100 Einträge aufnimmt. Die einzelnen Einträge werden über den Namen und einen Index angesprochen: `benutzer[0]` ist der erste Eintrag, `benutzer[1]` der zweite, `benutzer[12]` der dreizehnte, `benutzer[99]` der hundertste. Aus unserer Sicht hat man den Eindruck, dass das Array wie in der Abbildung unten aussieht. Wir können direkt auf den Wert jeder Variablen `benutzer[i]` für i = 0, 1, …, 99 zugreifen.



Dieses Material / Vorabdruck darf nur bis zum 31.10.2023 verwendet werden. Hernach steht das vollständige Lehrbuch zum käuflichen Erwerb bereit.»

Tatsächlich gibt es für ein Array nur einen einzigen Zeiger (ein einziger Eintrag mit einer Adresse in der Tabelle der verwendeten Variablen) und dieser Zeiger zeigt auf den Anfang der Datenstruktur `benutzer`, nämlich auf die Speicherzelle, an der der Wert der Variablen `benutzer[0]` abgelegt ist.



Neue Konzepte und Begriffe

Eine **Datenstruktur** ist eine Struktur, die das Organisieren von Daten unterstützt. Diese Organisation von Daten (eine gewisse Ordnung) dient nicht nur der Speicherung von Daten (Ablegen der Daten im Computerspeicher), sondern auch dem effizienten Zugriff und der Verwaltung der Daten. Die gewählte Datenstruktur ermöglicht gewisse Operationen auf den Daten und ist dann bestimmend für die Effizienz des Zugriffs auf die Daten und der Ausführung der ermöglichten Operationen.

Ein **Array** ist eine Datenstruktur die aus n Variablen (n wählbar) nummeriert von 0 bis n – 1 besteht. In unserem Beispiel heisst das Array `benutzer` und besteht aus 100 Variablen `benutzer[0]`, `benutzer[1]`, ..., `benutzer[99]`. Der Sinn hinter der Einführung von Arrays ist, dass man in der Tabelle der verwendeten Variablen nur einen Eintrag mit der Adresse der Speicherzelle hat, an der die Speicherung des Arrays anfängt. Die Zielsetzung dabei ist, dass der Computer **direkt** auf jedes Element des Arrays **zugreifen** kann. Direkt bedeutet, dass der Computer gezielt an die richtige Speicherzelle (ohne Inhalte anderer Speicherzellen anzuschauen) gehen kann, weil er sich die Adresse der Speicherzelle selbst ausrechnen kann.

Um dem Computer zu ermöglichen, sich selbst die Adresse jedes i-ten Elements eines Arrays auszurechnen, muss man das Speichern von Daten wie folgt managen:

1. In der Tabelle der Variablen gibt es einen Zeiger (eine Adresse) für den Anfang des Arrays.
2. Das Array ist kompakt wie in der Abbildung oben gespeichert, d.h., es gibt einen Teil des Speichers, in dem nur das Array lückenlos gespeichert wird.
3. Es ist im Voraus eine feste Grösse der Speichereinheiten für jedes Element des Arrays (z.B. 64 Bits) festgelegt und es ist auch im Voraus bestimmt, wie viele Elemente das Array höchstens haben wird. Somit kann das Speicherverwaltungssystem einen kompakten Teil des Speichers der Grösse «die Anzahl der Elemente» × «die Speichergrösse eines Elements» für das Array zur Verfügung stellen.

Wenn ein Computer beispielsweise Speichereinheiten der Grösse von 32 Bits adressieren kann und das Array `benutzer` 100 Elemente hat und 128 = 4 · 32 Bits pro Element braucht, reserviert die Speicherverwaltung des Betriebssystems des Computers einen kompakten Speicherabschnitt der Grösse 100 · 128 = 12 800 Bits.

Wenn die Speicherung des Arrays `benutzer` bei der Adresse 500 anfängt, dann ist `benutzer[0]` an der Adresse 500, `benutzer[1]` an 504, `benutzer[2]` an 508 usw. Allgemein ist die Variable `benutzer[i]` an Adresse 500 + 4 · i. Somit kann die Speicherverwaltung des Betriebssystems für jedes `i` selbständig die Adresse von `benutzer[i]` aus dem Zeiger 500 auf `benutzer` und der Ordnung `i` ausrechnen und direkt auf den Wert von `benutzer[i]` zugreifen.

Vorabmaterial – nicht schlusskorrigiert
Stand 14.08.2023

Dieses Material / Vorabdruck darf nur bis zum 31.10.2023 verwendet werden. Hernach steht das vollständige Lehrbuch zum käuflichen Erwerb bereit.»

1.12 Für eine Liste von 700 rationalen Zahlen will man ein Array `x` erstellen. Um die Zahlen genauer abzuspeichern, verwendet man 64 Bits pro Zahl (Kapitel 1, Band «Data Science und Sicherheit»). Die adressierbaren Speichereinheiten des Computers sind 32 Bits.
Wie gross ist der reservierte Speicher für Array `x` und wie bestimmt man die Adresse von `x[i]`, wenn die Adresse von `x` in der Tabelle 313 ist?

1.13 In der Informatik arbeiten wir oft mit speziellen Tabellen, auch Matrizen genannt (wie z. B. die Nachbarschaftsmatrix zur Darstellung eines Graphen, s. Kapitel 1, Band «Data Science und Sicherheit»). Deswegen wollen Sie auch mit zweidimensionalen Arrays `m[i, j]` arbeiten. Dies bedeutet, dass Sie `m[i, j]` mit einem «eindimensionalen» Speicher so abspeichern müssen, dass Sie für alle `i` und `j` die Adresse von `m[i, j]` leicht ausrechnen können, um direkten Zugriff auf `m[i, j]` zu erhalten.
Nehmen wir an, jedes Element der Matrix braucht nur eine Speichereinheit des Computers und die Matrix ist 10 × 10 gross. Nehmen wir weiter an, wir haben die ersten 100 Speicherzellen mit Adressen 0, 1, 2, …, 99 reserviert. Wie speichern Sie die Matrix eindimensional in diesem Speicher und wie berechnen Sie für jedes `i` und `j` die Adresse von `m[i, j]`? Sie können die Aufgabe auch anders auffassen. Wie implementieren Sie ein zweidimensionales Array `m[i, j]`, wenn Sie nur ein eindimensionales Array `x[k]` zur Verfügung haben?

Vorabmaterial – nicht schlusskorrigiert
Stand 14.08.2023

Geschichtlicher und gesellschaftlicher Kontext

Die ersten Computer hatten eine feste Grösse von adressierbaren Speichereinheiten, die jeweils die Grösse der Register der Prozessoren hatten. Deswegen nannte man die Speichereinheiten ebenfalls **Register**. Das Betriebssystem des Computers war auch sehr schlank und offerierte den Nutzerinnen und den Programmierern viel weniger Unterstützung als heute. Beim Programmieren standen zum Beispiel auch keine Variablen zur Verfügung. Die «Variablen» damals waren die konkreten Speichereinheiten. Die Programmiererin musste einfach wissen, unter welcher Adresse sie was abgespeichert hatte.
Was bedeutete es, zu programmieren? Die Programmierenden mussten sich die Variablentabellen selbst notieren: zum Beispiel, x ist an Adresse 113, y ist an Adresse 71 und z ist an Adresse 3 gespeichert. Wenn man x und y addieren wollte und das Resultat in z speichern wollte (z = x + y), dann sah der Befehl auf der Mikroebene wie folgt aus:
Register(3) ← Inhalt(Register(113)) + Inhalt(Register(71))
Man addierte die Inhalte der Register 113 und 71 und speicherte das Resultat in Register 3.
Auf der Mikroebene beim Programmieren sah die Umsetzung wie folgt aus:
REG(1) ← Inhalt(Register(113))
Der Inhalt des Registers 113 wurde in Register REG(1) des Prozessors kopiert.
REG(2) ← Inhalt(Register(71))
Der Inhalt des Registers 71 wurde in Register REG(2) des Prozessors kopiert.
ADD
Die Inhalte der Register REG(1) und REG(2) wurden aufaddiert und das Resultat wurde in REG(1) gelegt.
Register(3) ← Inhalt(REG(1))
Der Inhalt von Register REG(1) des Prozessors wurde in Register(3) des Speichers kopiert.
Wir sehen hier zwei Typen von Befehlen. Der erste Typ kopiert Inhalte eines Registers in ein anderes Register und sorgt somit auch für die Kommunikation zwischen dem Prozessor und dem Speicher. Der zweite Typ von Befehlen, wie hier beispielsweise die arithmetischen Operationen, werden nur mit den Inhalten der Register des Prozessors ausgeführt. Eine Programmiersprache mit derartigen Befehlen heisst **Assembler**.
Ganz am Anfang der Computerentwicklung offerierte das Betriebssystem eines Computers nicht einmal einen Assembler. Man musste in sogenanntem Maschinencode programmieren. Die einzelnen Anweisungen (Befehle) waren ausschliesslich Folgen von Nullen und Einsen in einer festen Länge. Ein fester Teil der Folge kodierte die auszuführende Operation und die restlichen Teile die Argumente (Parameterwerte) oder sogar nur die Adressen der Argumente. Heutige Betriebssysteme haben den Programmiererinnen und Programmierern viel von dieser Arbeit auf der Mikroebene mit dem physischen Teil des Computers (Prozessor, Speicher) abgenommen. Wir müssen uns heute beim Programmieren gar nicht mehr mit der physischen

Dieses Material / Vorabdruck darf nur bis zum 31.10.2023 verwendet werden. Hernach steht das vollständige Lehrbuch zum käuflichen Erwerb bereit.»

Speicherung von Variablenwerten und der Kommunikation zwischen unterschiedlichen Computerkomponenten beschäftigen. Wo unsere Daten im Computer gespeichert werden und wann der Prozessor zur Ausführung unserer Programme zur Verfügung gestellt wird, entscheidet das Betriebssystem selbst. Der **Compiler** einer Programmiersprache übersetzt unsere Programme automatisch in den Maschinencode, der für die Hardware des Computers verständlich ist und somit die Ausführung unserer Programme ermöglicht. Der **Interpreter** kompiliert, d. h. übersetzt ein Programm stückweise in seinen Maschinencode und führt die übersetzten Befehle sofort aus. Dieser «Luxus» beim Programmieren ermöglicht uns viel effizienter und bequemer neue Anwendungsprogramme zu entwickeln. Dieser Komfort birgt jedoch auch Nachteile. Weil wir die Übersicht über die Ausführung sowie die Steuerungsmöglichkeiten bei der Ausführung unserer Programme verlieren (sie sind beim System schon vorprogrammiert), können wir nicht die schnellstmögliche Ausführung unserer Programme anstreben. Bei der sogenannten **kritischen Software** (z. B. Raketen- oder Flugzeugsteuerung) gehen die Entwickelnden oft so weit, dass sie in Assembler programmieren. Dabei geht es nicht nur darum, die maximal mögliche Effizienz zu erreichen, sondern auch um die Möglichkeit, die Ausführung der Programme genau zu analysieren (beobachten), sodass man garantieren kann, dass das Programm in einer vorgegebenen Zeit die gewünschten Aktionen der technischen Systeme ausführt.

Vorabmaterial – nicht schlusskorrigiert
Stand 14.08.2023

Beispiel 1.4 **Binäre Suche in einem Makro-Assembler**

Nehmen wir an, wir haben eine sortierte Zahlenfolge in den ersten 1000 Registern mit den Adressen 0, 1, 2, …, 999 gespeichert. Wir suchen die Registeradresse (von 0 bis 999) einer Zahl, die in Register(1000) gespeichert ist.
Allgemein müssen wir uns immer den Suchraum zwischen zwei Adressen i und j notieren, in dem der gesuchte Wert Inhalt(Register(1000)) sich befinden kann. Wir «reservieren» uns Register 1001 für i und Register 1002 für j. Am Anfang sind i = 0 und j = 999. In Register 1003 soll am Ende das Resultat stehen.
Um die Implementierung anschaulich und kurz zu halten, führen wir dies auf einer Makroebene aus, in der wir direkt mit den Werten der Speicherregister arbeiten, ohne sie vorher in die Register des Prozessors zu übertragen. Register(1004) nutzen wir, um die mittlere Adresse des Suchraums zu berechnen. In Register(1005) speichern wir den Wert 2, um die ganzzahlige Division durch 2 ausüben zu können.
Im Programm nutzen wir mehrfach den Befehl GOTO. Der Befehl GOTO i sagt, dass die Ausführung des Programms in Zeile i fortgesetzt werden soll. Die Umsetzung im Rechner sieht so aus, dass es ein spezielles Register REG(0) gibt, in dem die Nummer der Programmzeile gespeichert ist, die der Rechner als nächste ausführen soll. Der Prozessor liest nach jedem Schritt den Inhalt dieses Registers und führt die entsprechende Programmzeile aus. Wenn man diese Nummer mit GOTO ändert, springt man zur Ausführung der angegebenen Zeile mit dieser Nummer. Wurde kein GOTO-Befehl ausgeführt, wird automatisch nach der Ausführung einer Zeile des Programms der Wert von REG(0) um 1 erhöht. Damit setzt der Computer die Ausführung des Programms mit der nächsten Zeile fortgesetzt.

```
0  Register(1005) ← 2
1  Register(1003) ← 0
2  Register(1001) ← 0
3  Register(1002) ← 999
4  if Inhalt(Register(1001)) = Inhalt(Register(1002)) GOTO 9
5  Register(1004) ← Inhalt(Register(1001)) +
   Inhalt(Register(1002))
6  Register(1004) ← Inhalt(Register(1004)) /
   Inhalt(Register(1005))
7  if Inhalt(Register(1000)) ≠ Inhalt(Register(Inhalt
   (Register(1004)))) GOTO 10
8  Register(1003) ← Inhalt(Register(1004))
9  HALT
```


Dieses Material / Vorabdruck darf nur bis zum 31.10.2023 verwendet werden. Hernach steht das vollständige Lehrbuch zum käuflichen Erwerb bereit.»

```
10  if Inhalt(Register(1000)) > Inhalt(Register(Inhalt
    (Register (1004)))) GOTO 13
11  Register(1002) ← Inhalt(Register(1004)) - 1
12  GOTO 4
13  Register(1001) ← Inhalt(Register(1004)) + 1
14  GOTO 4
```

Obwohl wir einen Makro-Assembler verwendet haben, sehen wir, wie unübersichtlich und lang das Programm ist. Bei der Umsetzung in einer echten Assemblersprache würde dies noch deutlicher werden.
In den Zeilen 7 und 10 verwenden wir etwas ganz Neues, nämlich die **indirekte Adressierung**. Dies ist unvermeidbar, wenn man das Array als Datenstruktur verwenden möchte.
Inhalt(Register(Inhalt(Register(1004)))) bedeutet, dass man mit Inhalt(Register(1004)) die Adresse bestimmt (z. B. den Index des Arrays), an die man dann geht, um den dort gespeicherten Wert zu lesen. Wenn beispielsweise Inhalt(Register(1004)) = 499 ist, dann erhalten wir den Wert, der in Register(499) gespeichert ist. Um die lange Bezeichnung «Inhalt(Register(Inhalt(Register(k))))» zu vermeiden, verwendet man für die indirekte Adressierung oftmals die kurze Bezeichnung Inhalt*(Register(k)).

1.14 Simulieren Sie die Arbeit des Programms in Makro-Assembler aus Beispiel 1.4 so lange, bis Zeile 4 des Programms fünfmal ausgeführt worden ist. Zeichnen Sie eine Tabelle mit dem Inhalt der Register mit den Adressen 1000, 1001, 1002, 1003, 1004, 1005 und des Registers REG(0), das den Inhalt der als Nächstes auszuführenden Zeile beinhaltet. In der Tabelle sollen jeweils die Werte der Register nach der Ausführung der Zeilen 4 und 7 festgehalten werden.
Bevor das Programm startet, beinhaltet Register(1000) die gesuchte Zahl 222 und die tausend Register Register(0), Register(1), Register(2), …, Register(999) beinhalten die ersten tausend geraden Zahlen 0, 2, 4, 6, …, 1998.

1.15 Die folgende Tabelle ist ein Auszug der Speicherinhalte gewisser Register. Bestimmen Sie die Resultate folgender indirekter Adressierungen.

a Inhalt*(Register(1))
b Inhalt*(Register(2))
c Inhalt*(Register(3))
d Inhalt*(Register(5))

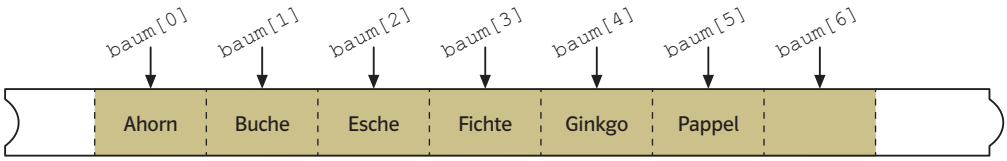
Register-adresse	Register-inhalt
1	4
2	72
3	1
4	1
5	5
72	13

1.16 Den Sortieralgorithmus Bubblesort kann man als eine mehrfache Wiederholung desselben Moduls ansehen. Man «läuft» einmal von links nach rechts über eine Folge $a_1, a_2, \ldots, a_n$, vergleicht immer die Nachbarn a_i und a_{i+1} für $i = 1, 2, \ldots, n - 1$ und vertauscht sie nach Bedarf. Programmieren Sie dieses Modul in Makro-Assembler für die folgende Situation. In den Registern mit den Adressen 51, 52, …, 62 steht eine Zahlenfolge der Länge 12. Das Programm soll das Modul auf dieser Folge ausführen.

Vorabmaterial – nicht schlusskorrigiert
Stand 14.08.2023

Dieses Material / Vorabdruck darf nur bis zum 31.10.2023 verwendet werden. Hernach steht das vollständige Lehrbuch zum käuflichen Erwerb bereit.»

1.17 Ein Array `baum` der Grösse 7 hat (wie unten gezeigt) bereits sechs lexikographisch sortierte Einträge, der letzte Platz ist noch leer. Nun soll an dritter Stelle (mit Index 2) der Eintrag «Eiche» eingefügt werden, ohne dass der dort stehende Name überschrieben wird. Am Ende sollen alle 7 Einträge sortiert im Array stehen. Welche Zuweisungen müssen in welcher Reihenfolge ausgeübt werden?



Neue Konzepte und Begriffe

Wir halten fest, dass Arrays als Datenstruktur ermöglichen, beliebig viele Elemente in einer Reihenfolge zu speichern und dabei in der Tabelle der Variablen nur einen Namen des Arrays mit einem Zeiger zum Anfang des für das Array reservierten Speicherplatzes zu haben. Der Hauptvorteil von Arrays ist, dass man einen direkten Zugriff auf alle Einträge (Elemente) des Arrays hat, weil der Computer selbständig für jedes i die Adresse des i-ten Elements des Arrays ausrechnen kann. Der Nachteil von Arrays ist, dass das Einfügen eines neuen Eintrags in einer bestimmten Stelle (ausser der letzten Stelle) aufwändig sein kann. Deswegen ist das Halten der vorhandenen Ordnung (der sortierten Folge) beim Einfügen und Löschen von Elementen auch aufwändig. Für die Verwaltung von sortierten Folgen mit häufigen Updates, also für eine sogenannte **dynamische Datenverwaltung**, eignen sich Arrays dementsprechend nicht. Ein zusätzlicher Nachteil der Verwendung von Arrays ist, dass man im Voraus wissen sollte, wie viele Elemente das Array höchstens haben wird und wie viel Platz man für die einzelnen Elemente braucht, um einen entsprechend grossen kompakten Teil des Speichers für das Array zu reservieren.

Vorabmaterial - nicht schlusskorrigiert
Stand 14.08.2023

1.18 Im Array `baum` soll der Eintrag «Fichte» in `baum[3]` aus dem Array in der Abbildung in Knobelaufgabe 1.17 entfernt werden. Dabei müssen die Einträge rechts von «Fichte» um eine Position nach links verschoben werden. Welche Zuweisungsbefehle verwenden Sie dabei?

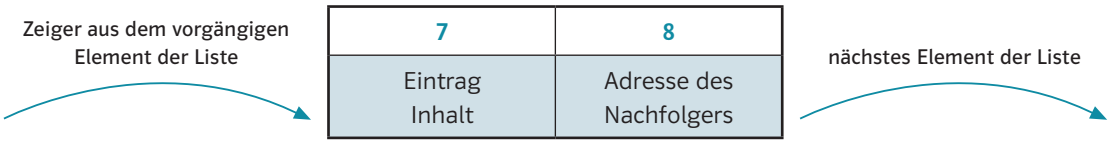
Konstruktion von verketteten Listen

Beispiel 1.5

Eine Alternative zu Arrays als Datenstruktur stellen **Listen** dar. Um sie von Python-Listen zu unterscheiden, werden sie auch **verkettete Listen** genannt. Hier verzichten wir auf den Vorteil des direkten Zugriffs auf jedes Element ausser dem nullten Element (Anfang der Liste). Dafür bleiben wir frei bezüglich der Anzahl der Elemente der Liste (wir müssen die Grösse der Liste nicht im Voraus angeben), weil wir die Liste nicht in einem kompakten vorreservierten Teil des Speichers speichern müssen. Auch die Grösse des Speichers für die einzelnen Elemente muss nicht normiert sein. Jedes Element kann individuell eine passende Speichergrösse beanspruchen. Wir können eine Liste jederzeit erweitern und die neuen Elemente auf einem beliebigen freien Speicherplatz (einer unbenutzten Lücke) speichern. Zusätzlich soll das Einfügen und das Entfernen eines neuen Elements höchstens 2–3 Zuweisungsoperationen kosten.

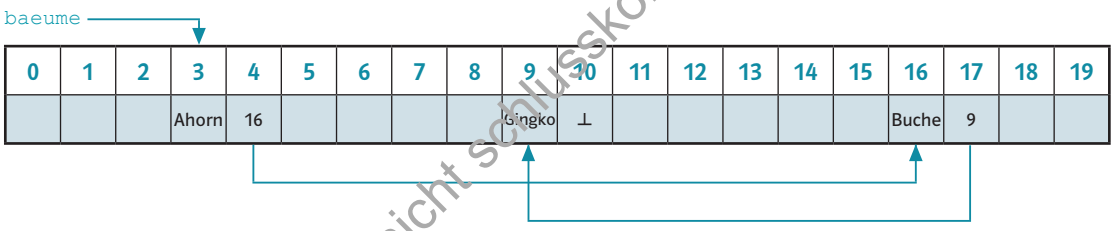
Dazu verwendet man in der Liste strukturierte Elemente, die aus zwei Komponenten bestehen. Die erste Komponente beinhaltet das Element (den Eintrag oder eine ganze Datei) und die zweite Komponente ist die Adresse, an der sich das nachfolgende Element der Liste befindet. In der folgenden Abbildung steht das strukturierte Element an den zwei Adressen 7 und 8.

Dieses Material / Vorabdruck darf nur bis zum 31.10.2023 verwendet werden. Hernach steht das vollständige Lehrbuch zum käuflichen Erwerb bereit.»



Genau wie bei Arrays steht bei einer verketteten Liste in der Variablentabelle nur der Name der Liste mit einem Zeiger (einer Adresse) auf den Anfang der Liste, also auf das nullte Element der Liste. Man verwendet das Zeichen Nil («⊥») für einen leeren Inhalt einer Speicherzelle. Dieses spezielle Zeichen brauchen wir tatsächlich. Keine Speicherzelle ist tatsächlich leer. Falls wir eine Speicherzelle noch nicht verwendet haben, liegt dort eine Null. Aber eine Null kann man nicht nur als «leer» interpretieren, sondern auch als eine Zahl oder eine Adresse. Das Zeichen ⊥ hat eine eindeutige Interpretation. Wenn wir z. B. bei einem Listenelement in der zweiten Komponente ⊥ haben, wissen wir, dass dieses Element das letzte Element in der Liste ist, weil keine Nachfolgeadresse vorhanden ist.

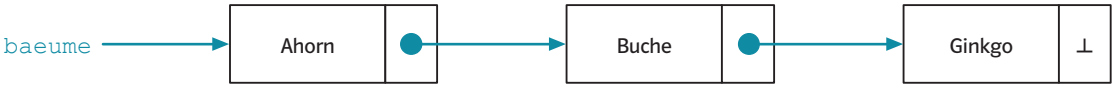
In der Abbildung unten sehen wir eine Liste `baeume` mit drei Elementen «Ahorn», «Buche» und «Ginkgo» in lexikographischer Reihenfolge. Die Liste ist nicht kompakt gespeichert, d. h., die Elemente liegen an unterschiedlichen Adressen im Speicher. Die Reihenfolge der Elemente der Liste ist unabhängig von der Reihenfolge der verwendeten Adressen der Speicherzellen.



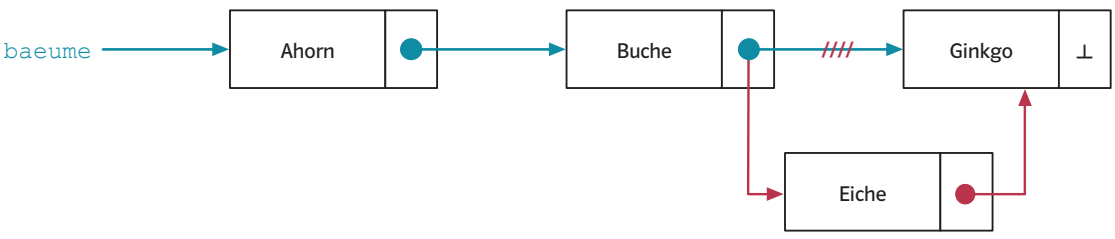
Vorabmaterial – nicht schlusskorrigiert
Stand 14.08.2023

Das erste Element der Liste steht an den Adressen 3 und 4. An die Adresse 3 zeigt der Zeiger der Liste `baeume`. Der Inhalt «Ahorn» (die erste Komponente) steht an Adresse 3 und Adresse 4 beinhaltet die Adresse 16 des zweiten Elements der Liste. An Adresse 16 steht tatsächlich die erste Komponente «Buche» des zweiten Listenelements und an der nachfolgenden Adresse 17 befindet sich die Adresse (der Zeiger) 9 des dritten Elements «Ginkgo» der Liste. Das dritte Element der Liste ist auch das letzte Element, weil in der Adressenkomponente das Symbol ⊥ steht.

Heute müssen wir uns beim Programmieren nicht um die physischen (konkreten) Adressen kümmern, an denen einzelne Listenelemente gespeichert werden. Das übernimmt für uns die Speicherverwaltung des Betriebssystems abhängig von den verfügbaren freien Speicherzellen. Wir haben keine Ahnung, an welchen Adressen des Speichers die einzelnen Listenelemente gespeichert sind. Deswegen nutzen wir oft eine vereinfachte Darstellung von Listen. Wir abstrahieren von den konkreten Adressen und stellen Zeiger nur als Pfeile dar.

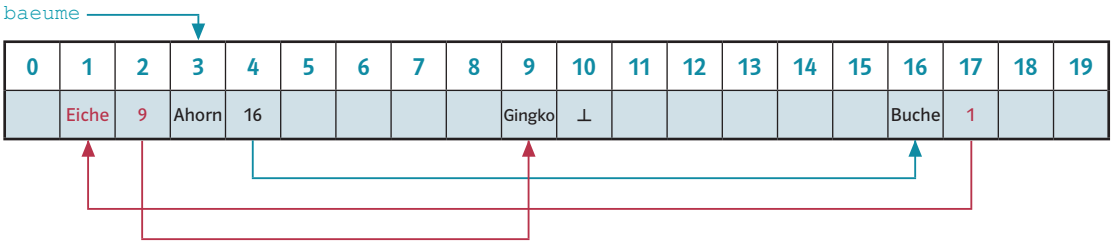


Wir wollen diese Liste um einen Eintrag «Eiche» erweitern und dabei sollte die Liste lexikographisch geordnet bleiben. Das bedeutet, dass «Eiche» nach «Buche» und vor «Ginkgo» eingefügt werden muss. In der vereinfachten Darstellung geht es nur um die Änderung von zwei Zeigern (rot).



Dieses Material / Vorabdruck darf nur bis zum 31.10.2023 verwendet werden. Hernach steht das vollständige Lehrbuch zum käuflichen Erwerb bereit.»

Dies entspricht der Änderung bzw. Neuzuweisung von zwei zweiten Komponenten mit Nachfolgeadressen. Wenn man das neue Element «Eiche» an den Adressen 1 und 2 platziert, sieht das Ergebnis wie in der Abbildung unten aus. Die neuen Einträge in den Speicherzellen sowie die Zeiger sind rot gezeichnet.



1.19 **a** Nehmen Sie die Liste `baeume` aus Beispiel 1.5 mit den vier Elementen «Ahorn», «Buche», «Eiche» und «Ginkgo» und erweitern Sie die Liste um einen neuen Eintrag «Fichte» so, dass die Elemente der Liste lexikographisch geordnet bleiben. Das Element «Fichte» sollte an den Adressen 13 und 14 gespeichert werden. Zeigen Sie das Resultat der Einfügung im Speicher wie in der Abbildung von Beispiel 1.5. Die geänderten Inhalte sollten rot markiert werden.

b Entfernen Sie aus der Liste `baeume`, die in Teilaufgabe a modifiziert wurde, das Element «Buche». Zeigen Sie in einer Abbildung, wie in Teilaufgabe a, das Resultat mit rot markierten Änderungen. Danach entfernen Sie auch das Element «Ginkgo» und zeichnen das neue Resultat (die entstandene Liste mit drei Elementen).

c Fügen Sie zu der in Teilaufgabe b konstruierten Liste ein neues Element «Kiefer» an den Adressen 8 und 9 hinzu und zeichnen Sie die Implementierung der Liste im Speicher.

d Jetzt will man aus der Liste, die in Teilaufgabe c entstanden ist, das nullte Element «Ahorn» entfernen. Wie kann man dies umsetzen?

1.20 Die Listen, die wir bisher betrachtet und in Beispiel 1.5 eingeführt haben, heissen auch **einfach verkettete Listen**. Mit diesem Namen will man zum Ausdruck bringen, dass man sie bei der Suche nach Daten nur in eine Richtung von Anfang bis Ende durchlaufen kann. Somit muss man unvermeidlich so viele Elemente der Liste anschauen, wie die Ordnung des gesuchten Elements vorgibt. Zusätzlich kann man nicht umdrehen und zu einem schon besuchten Element zurückgehen, sondern muss zwingend erneut am Anfang der Liste beginnen und in der vorgegebenen Richtung neu suchen.
Eine Idee, diesen Nachteil der einfach verketteten Liste zu überwinden, wäre, die Liste dank mehrerer Zeiger so zu konstruieren, dass man die Liste in beiden Richtungen durchlaufen könnte.

a Entwickeln Sie ein Listenmodell, in dem man mittels Zeiger (Adressen) in der Liste in beide Richtungen laufen kann. Wenden Sie Ihr Modell auf die Liste `baeume` aus Beispiel 1.5 mit drei Elementen «Ahorn», «Buche» und «Ginkgo» an und zeigen Sie die Organisation der neuen Liste `baeume2` im Speicher.

b Fügen Sie zu Ihrer Liste `baeume2` ein neues Element «Eiche» an der Adresse 6 hinzu. Zeichnen Sie mit Rot die Änderungen des Zeigers ein.

c Ihre neue Liste sollte eine effizientere Implementierung von Bubblesort ermöglichen. Zeichnen Sie die Änderungen mit Rot, mit denen man das Vertauschen der Reihenfolge der «benachbarten» Elemente «Buche» und «Eiche» bewirken kann. Wie setzen Sie die Vertauschung auf Ebene der Speichereinheiten um?

d Entfernen Sie das Element «Buche» aus `baeume2` nach der Ausführung von Teilaufgabe b. Zeichnen Sie wieder mit Rot die Änderungen in der Speicherdarstellung ein.

Vorabmaterial – nicht schlusskorrigiert
Stand 14.08.2023

Dieses Material / Vorabdruck darf nur bis zum 31.10.2023 verwendet werden. Hernach steht das vollständige Lehrbuch zum käuflichen Erwerb bereit.»

Neue Konzepte und Begriffe

Verkettete **Listen** sind Datenstrukturen, die es ermöglichen, eine beliebige Anzahl von Elementen zu speichern. Die Anzahl der Elemente sowie den Speicherbedarf der einzelnen Elemente muss man nicht im Voraus kennen und man kann jederzeit ein neues Element zu einer Liste hinzufügen. Listen erfordern im Gegensatz zu Arrays keine Reservierung eines kompakten Speicherbereichs, weil die Listenelemente an frei verfügbaren Plätzen über den gesamten Speicher verstreut werden können.
Ein Element aus einer Liste hat eine Struktur. Bei **einfach verketteten Listen** hat jedes Element zwei Komponenten. Die erste Komponente dient zur Speicherung von Daten und die zweite Komponente zur Speicherung der Adresse des nachfolgenden Elements der Liste. Bei **doppelt verketteten Listen** haben die Elemente drei Komponenten. Die zusätzliche Komponente dient zur Speicherung des Vorgängers in der Liste. Damit besteht die Möglichkeit, die Liste in beiden Richtungen zu durchlaufen.
Im Unterschied zu Arrays dürfen die Elemente einer Liste beliebig grosse Daten beinhalten, deren Grössen sich paarweise stark unterscheiden dürfen. Dies kann man auf zwei unterschiedliche Arten verwalten. Die erste Möglichkeit ist, dass die erste Komponente einer Liste auch nur eine Adresse ist und an dieser Adresse der entsprechende Dateninhalt abgelegt ist. Die zweite Möglichkeit ist, jedem Element eine zusätzliche Komponente am Anfang hinzuzufügen. Der Inhalt dieser Komponente besagt, wie viele Speichereinheiten zur Speicherung des Dateninhalts des Elements verwendet werden und somit an welcher Adresse die Zeiger des Elements vorliegen.

Vorabmaterial – nicht schlusskorrigiert
Stand 14.08.2023

1.21 Carla träumt. Sie ist auf der Suche nach der Stelle 7, denn dort ist er versteckt, der Schatz. Worum es sich dabei genau handelt, weiss sie nicht – Träume sind da recht unpräzise – vielleicht um einen saftigen Doughnut, ein Ticket für das nächste Konzert der Lieblingsband, den Heiligen Gral, …
Die Stelle 7 zu finden ist abenteuerlich. Der Weg im dichten Wald führt geradeaus und man kann ihn nicht verlassen. Alle 100 Meter wird Carla auf eine Lichtung stossen, die eine Nummer trägt. Welche Zahlen dort aber erscheinen werden und in welcher Reihenfolge, ist ihr völlig unbekannt – es können negative Zahlen sein oder positive. Jede Lichtung hat genau einen Eingang und einen Ausgang und sie darf nicht zurückgehen. Sie läuft los, in den Nebel hinein, und bald erinnert sie sich nur noch daran, dass sie die 7 sucht, es kein Zurück gibt und sie immer wieder an Lichtungen kommen wird, die vielleicht die 7 sind, vielleicht aber auch nicht. Die erste Lichtung kann die 34 sein oder die −2016 oder – mit viel Glück – die gesuchte 7.
Wüsste Carla wenigstens, dass es insgesamt genau 82 Lichtungen sind, alle mit unterschiedlichen Zahlen, und wüsste sie auch, dass eine davon mit Sicherheit die 7 ist, so wäre das sicher eine Erleichterung.

a Die Suche von Carla entspricht der Suche in einer Datenstruktur. Um welche Datenstruktur handelt es sich?

b Wie viele Lichtungen wird Carla mindestens besuchen, bis sie die 7 findet, und wie viele höchstens?

c Würde sich etwas ändern, wenn die Stellen aufsteigend sortiert wären, also eine nächste Stelle immer eine grössere Zahl aufweist als die vorhergehende?

d Nehmen wir an, Carla macht die Suche nach 7 in einer Folge von 82 Lichtungen wiederholend viele Male mit immer anderen Zahlen und anderen Reihenfolgen. Die 7 ist aber immer genau in einer Lichtung dabei und erscheint gleich häufig in jeder der 82 Lichtungen. Wie viele Lichtungen muss Carla im Durchschnitt besuchen, um die Lichtung mit 7 zu finden?

Dieses Material / Vorabdruck darf nur bis zum 31.10.2023 verwendet werden. Hernach steht das vollständige Lehrbuch zum käuflichen Erwerb bereit.»

Geschichtlicher und gesellschaftlicher Kontext

In den neunziger Jahren des zwanzigsten Jahrhunderts hat Guido van Rossum im Centrum Wiskunde & Informatica (CWI) in Amsterdam eine neue Programmiersprache «Python» entwickelt. Der Name bezog sich weniger auf die Gattung der Schlangen aus der Familie Python, sondern auf die britische Komikertruppe Monty Python. Das Ziel von Guido van Rossum war, eine Programmiersprache zu entwickeln, in der Programmiererinnen und Programmierer viel mehr Freude am Programmieren haben würden. Dazu gehörte, sich mehr auf das Algorithmische konzentrieren zu können und die Speicherverwaltung so weit wie möglich vom Programmieren zu entfernen. Die meisten Programmiersprachen erforderten im Vorfeld, dass man am Anfang eines Programms die verwendeten Variablen mit ihren Typen deklariert. Python hingegen ordnet die Typen den Variablen automatisch während der Ausführung der Programme zu. Die meisten Programmiersprachen offerierten den Programmierenden von Anfang an die Datenstruktur des Arrays, wobei zu Programmbeginn die Grössen der verwendeten Arrays anzugeben waren, um der Speicherverwaltung die Reservierung eines geeigneten kompakten Speicherbereichs zu ermöglichen. Deswegen bezeichnet man Arrays als **statische** (nicht dynamisch veränderbare) **Datenstruktur**, deren Grösse man während der Ausführung des Programms nicht durch das Hinzufügen neuer Elemente ändern kann. Im Kontrast dazu sind verkettete Listen eine **dynamische Datenstruktur**, weil man sie unbeschränkt jederzeit um neue Elemente erweitern kann, ohne vorher eine Platzreservierung fester Grösse machen zu müssen. Python delegiert das Problem der Speicherreservierung vollständig an die Speicherverwaltung, indem Python als Datenstruktur **dynamische Arrays** zur Verfügung stellt. Dynamische Arrays verbinden die Vorteile von Arrays und Listen. Man hat direkten Zugriff auf jedes Element wie bei Arrays und trotzdem kann man sie dynamisch verändern und ihre Grösse unbeschränkt wachsen lassen. Wie diese Dynamik bei der Speicherverwaltung umgesetzt wird, ist nicht mehr Aufgabe der Programmierenden, sondern der Speicherverwaltung. Das einzige Unglück passierte in der Benennung der dynamischen Arrays. Man nennt sie (so wie auch im Band «Programmieren und Robotik») in Python ebenfalls «Listen», was aus Sicht der Datenstruktur für Verwirrung sorgen kann. Man muss wissen, dass Listen in Python keine der hier vorgestellten verketteten Listen sind, sondern dynamische Arrays mit direktem Zugriff auf jedes Element. Um Missverständnisse zu vermeiden, werden wir sie im Folgenden in diesem Zusammenhang als Python-Listen bezeichnen. Zusätzlich spricht man in Python nicht von Zeigern, sondern von Referenzen. Wir bleiben aber hier bei dem eingeführten Begriff des Zeigers.

Vorabmaterial – nicht schlusskorrigiert
Stand 14.08.2023

Implementieren von einfach verketteten Listen in Python

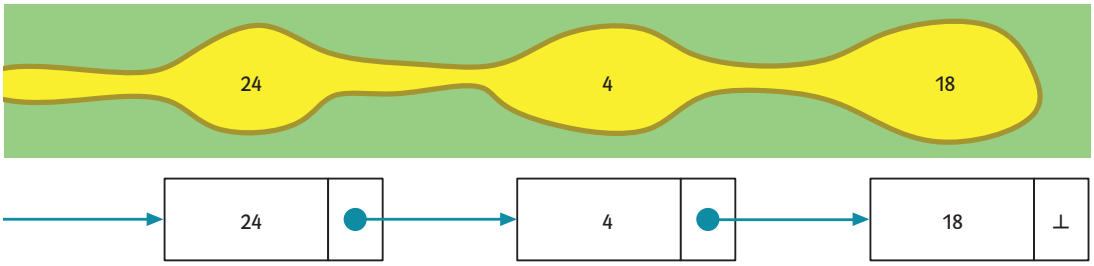
Beispiel 1.6

Dynamische Arrays stehen uns in Python unter dem Namen «Liste» zur Verfügung. Wenn wir aber die vorgestellte einfach verkettete Liste in Python verwenden wollen, müssen wir sie selbst implementieren. Dazu stellt Python das Werkzeug der **objektorientierten Programmierung** zur Verfügung.

Wir können in Python neue Objekte definieren, die eine Struktur (zusammengesetzt aus mehreren Objekten) haben dürfen. Das brauchen wir z. B. für diejenigen Elemente (die wir hier Knoten nennen) der Liste, die zwei Komponenten besitzen. Aus Knoten können wir dann eine Liste zusammensetzen. Das geht deswegen, weil wir auf den von uns definierten Objekten neue Operationen (Funktionen) definieren können.

Der Weg mit den Lichtungen aus Carlas Traum (Lernaufgabe 1.20) wird in der Informatik, wie wir bereits wissen, als **(einfach verkettete) Liste** bezeichnet. Eine solche Liste besteht aus **Knoten**, die Daten und einen **Zeiger** zum Nachfolgeknoten beinhalten. Zuvorderst steht der **Anker**, der Zeiger auf den ersten Knoten der Liste, der uns den Zugriff auf die Liste ermöglicht.

Dieses Material / Vorabdruck darf nur bis zum 31.10.2023 verwendet werden. Hernach steht das vollständige Lehrbuch zum käuflichen Erwerb bereit.»



Der letzte Knoten hat einen leeren Zeiger (`None`, was in Python dem zuvor verwendeten «Nil» entspricht), der damit das Ende der Liste anzeigt.
Für eine einfach verkettete Liste verwenden wir zwei Bausteine:

- Der Baustein `Knoten` besteht aus einem Dateninhalt und einem Zeiger, der den nächsten Knoten der Liste aufnehmen kann. Somit hat der «Knoten» schon die bezeichnete Struktur eines Listenelements mit zwei Komponenten.
- Der Baustein `Liste` beinhaltet den Anker, also den Zeiger auf den ersten Knoten.

Beide Bausteine werden in einer objektorientierten Programmiersprache wie Python als sogenannte **Objekte** umgesetzt. Was genau Objekte in objektorientierten Programmiersprachen sind, ist nicht einfach mit dem vorhandenen Vorwissen zu erklären, und deswegen werden wir das Vorstellen in mehrere Schritte aufteilen. Hier beginnen wir nur mit einem Beispiel. Wenn man neue Objekte wie Knoten einführen will, muss man sie zuerst beschreiben, also bestimmen, was ein Knoten ist und was kein Knoten ist. Dazu dient der Begriff der Klasse. Der erste Teil der Definition einer Klasse beinhaltet eine Funktion zur Generierung der Objekte (Instanzen) der Klasse. Alles, was man mit dieser Funktion bei einem Aufruf generieren kann, ist ein Objekt der Klasse.

Im folgenden Programm wird zunächst eine Klasse Knoten definiert: Das Schlüsselwort `class` teilt Python mit, dass eine Klasse definiert wird und der nachfolgende Name `Knoten` die Bezeichnung der Klasse ist. Jedes Objekt der Klasse `Knoten` besteht aus zwei Komponenten und nimmt als Daten `inhalt` und `naechster` auf. In `inhalt` wird das Datum (der Datenschlüssel oder die vollständigen Informationen) abgelegt, das gespeichert werden soll, `naechster` ist ein Zeiger auf den nächsten Knoten.
Die Objekte der Klasse `liste` haben nur eine Komponente (ein Attribut), hier `anker` genannt. Um Objekte zu verstehen, denken Sie an die folgende Analogie zu den Datensätzen (Kapitel 5, «Data Science und Sicherheit»). Datensätze können mittels Objekten umgesetzt werden. Datensätze haben auch mehrere Komponenten und jede Komponente entspricht einem Attribut, das den Wertebereich für diese Komponente bestimmt. Genauso besteht ein Objekt aus einer oder mehreren Komponenten und für jede Komponente wird dessen Wertebereich durch einen Datentyp bestimmt. Die Klasse `liste` enthält das Attribut `anker`.
Beide Klassen verfügen über spezielle Funktionen, die man Methoden der Klasse nennt.

Vorabmaterial – nicht schlusskorrigiert
Stand 14.08.2023

```
1  class Knoten:
2      def __init__(self, inhalt=0, naechster=None):
3          self.inhalt = inhalt
4          self.naechster = naechster
5
6      def __str__(self):
7          return str(self.inhalt)
8
9  class Liste:
10     def __init__(self, anker=None):
11         self.anker = anker
12
13 K3 = Knoten(27, None)
14 K2 = Knoten(13, K3)
15 K1 = Knoten(7, K2)
16 folge = Liste(K1)
17 K3.naechster = K1
```

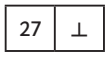

Dieses Material / Vorabdruck darf nur bis zum 31.10.2023 verwendet werden. Hernach steht das vollständige Lehrbuch zum käuflichen Erwerb bereit.»

Die Funktion `__init__()` wird ausgeführt, um ein Objekt (auch bezeichnet als eine Instanz) der jeweiligen Klasse zu erstellen. Schauen wir uns genau an, was beim Aufruf `Knoten()` passiert und halten wir die Unterschiede zur bisherigen Verwendung von Funktionen fest.

1. Wir schreiben nicht explizit `__init__()`. Die Funktion `__init__()` wird immer dann ausgeführt, wenn ein Objekt der Klasse erstellt wird. In Zeile 13 wird Knoten K3 erzeugt:
`K3 = Knoten(27, None)`
Durch die Ausführung werden die beiden Parameterwerte `27` und `None` an die Parameter `inhalt` und `naechster` von `__init__()` übergeben. Dabei ist `inhalt` das Datum, das gespeichert werden soll, und `naechster` ein Zeiger auf den nächsten Knoten in der Liste. Da `K3` der letzte Knoten der erstellten Liste sein soll, ist dieser Wert `None`.

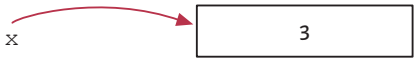
2. Wir sehen, dass die Funktion `__init__()` in ihrer Definition drei Parameter hat, aber beim Aufruf nur zwei Werte übergeben werden. Dem ersten Parameter `self` kommt eine besondere Rolle zu: Er steht an erster Stelle und gibt den Namen an, über den wir auf das gerade erzeugte Objekt der Klasse `Knoten` zugreifen können. Somit entspricht `self.inhalt` bei der Ausführung von
`K3 = Knoten(27, None)`
`K3.inhalt` und ermöglicht somit, den Wert `27` in `K3.inhalt` zu speichern. Man kann sich die Übertragung der Parameterwerte so vorstellen, dass `self` im Körper der Funktion `__init__()` überall durch `K3` ersetzt wird. Analog bezeichnet `self.naechster` beim Aufruf
`K2 = Knoten(13, K3)`
die Variable `K2.naechster` und ordnet ihr den Zeiger auf `K3` zu.
In Zeile 3 setzen wir also das Attribut `inhalt` des gerade erstellten Knotens (`self.inhalt`) auf den Wert des Parameters `inhalt`, der beim impliziten Aufruf der Methode `__init__()` übergeben wurde. Beachten Sie, dass `self.inhalt` und `inhalt` zwei verschiedene Variablen sind, die auch unterschiedliche Namen haben dürfen. `self` ist somit der Platzhalter für den Namen des Knotens, der durch den Aufruf von `__init__()` erzeugt wird.

3. Bisher mussten wir beim Aufruf einer Funktion Werte für alle Parameter der Funktion angeben. Hier müssen wir dies nicht notwendigerweise. Bei der Definition von `__init__()` können wir sogenannte **Standardwerte** festlegen. In Zeile 2 erhält `inhalt` den Standardwert `0` und `naechster` den Standardwert `None`. Diese Werte werden verwendet, wenn bei der Erstellung eines Objekts durch den Aufruf `Knoten()` keine Werte für die Parameter übergeben werden. Betrachten wir obiges Programm jetzt im Detail. Mit dem Befehl in Zeile 13 wird der Knoten `K3` mit den Komponenten `K3.inhalt = 27` und `K3.naechster = None` definiert, den wir wie folgt darstellen können:



Wenn man in Zeile 13 nur
`K3 = Knoten(27)`
schreiben würde, würde genau der gleiche Knoten erstellt werden, weil durch
`def __init__(self, inhalt=0, naechster=None)`
in Zeile 2 der Standardwert von `naechster` auf `None` gesetzt wird. Dabei werden die Werte der Reihe nach zugeordnet, die hinteren Parameter erhalten also ihre Standardwerte, wenn weniger Werte übergeben werden, als es Parameter gibt.

Die Objekte kann man nur verstehen, wenn man sich ihre Umsetzung und den Umgang mit ihnen auf der Ebene des Computerspeichers anschaut. Es gibt hier Ähnlichkeit zur Behandlung von Variablen, aber auch einen wesentlichen Unterschied zu Variablen.
Wenn man in einem Programm
`x = 3`
als Anweisung gibt, erzeugt der Computer eine neue Variable `x` mit Wert `3`. Wie Sie schon wissen, bedeutet dies, der Variable `x` einen Zeiger (eine Adresse) zuzuordnen, der auf eine Speicherzelle zeigt, in der der Wert `3` gespeichert ist.

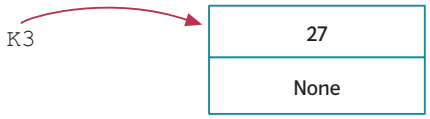


Mit der Anweisung
`K3 = Knoten(27, None)`
wird ein Objekt `K3` erzeugt. Die Umsetzung der Erzeugung ist analog zur Erzeugung einer Variablen. Dem Objekt `K3` ordnet man einen Zeiger auf einen Bereich des Speichers, der entsprechend

Vorabmaterial - nicht schlusskorrigiert
Stand 14.08.2023

Dieses Material / Vorabdruck darf nur bis zum 31.10.2023 verwendet werden. Hernach steht das vollständige Lehrbuch zum käuflichen Erwerb bereit.»

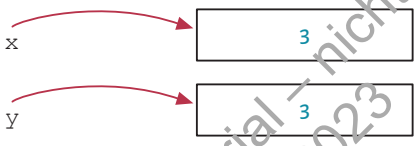
der Struktur des Objekts mehrere Komponenten haben kann.



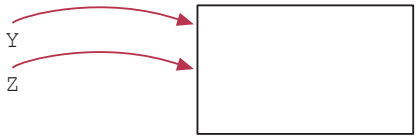
Der Aufruf

```
K3 = Knoten()
```

erzeugt entsprechend einen Knoten `K3` mit `K3.inhalt = 0` und `K3.naechster = None`.
Mit der Ausführung der Zeilen 13, 14 und 15 im ursprünglichen Programm entsteht bereits eine «Liste» ohne Verankerung mit drei Knoten `K1`, `K2` und `K3`.
In diesem Prozess der Erzeugung der Liste beobachten wir einen wesentlichen Unterschied zwischen Variablen und Objekten. Wenn wir für die Variablen `x` und `y` die Anweisung `y = x` ausführen, entsteht eine neue Variable `y` mit Zeiger auf eine Speicherzelle, in die man den Wert 3 der Variable `x` hineinschreibt.



Danach kann man mit den Variablen in dem Sinne unabhängig arbeiten, dass die Änderung des Variablenwerts von `x` keinen Einfluss auf den Wert von `y` hat und umgekehrt.
Bei den Objekten ist es anders. Mit `Y = Z` für ein schon existierendes Objekt `Z` erzeugt man `Y` so, dass man einen Zeiger von `Y` auf den Speicherbereich von `X` führt.



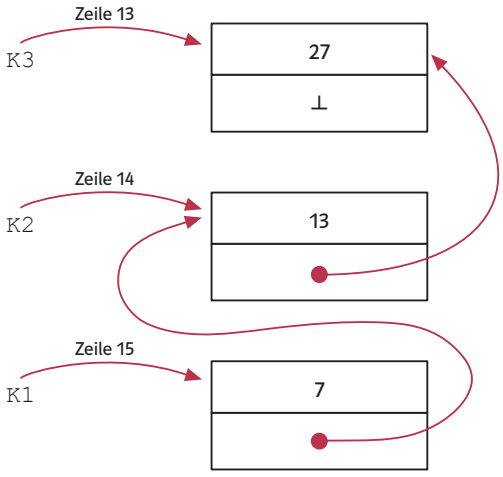
Damit bekommt `Y` die gleichen Werte für alle Komponenten wie `Z`. Die Werte werden aber nicht in einen anderen Speicherbereich kopiert. Das hat zur Folge, dass die Änderung eines Werts von `Z` automatisch die gleiche Änderung für `Y` nach sich zieht (und umgekehrt). So lange wir einen der Zeiger von `Y` und `Z` nicht ändern, wird jede Änderung eines der Objekte automatisch auf das andere Objekt übertragen.
Schauen wir uns jetzt die Ausführung der Anweisung (Zeile 14 im Programm)

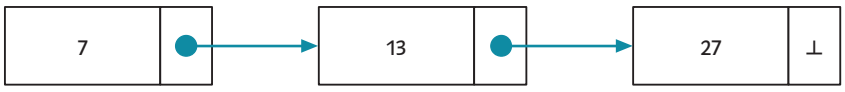
```
K2 = Knoten(13, K3)
```

Somit wird `K2.inhalt` den Wert 13 erhalten. Das Objekt (der Knoten) `K3` existiert schon, wodurch der Zeiger `K2.zeiger` an den gleichen Bereich des Speichers zeigen wird wie `K3`. Schauen Sie sich dazu die Abbildung unten an.
Analog führt die Ausführung der Anweisung in Zeile 15

```
K1 = Knoten(7, K2)
```

dazu, dass `K1.zeiger` ein Zeiger auf die gleiche Adresse ist wie `K2`, wie in der folgenden Abbildung.





Vorabmaterial – nicht schlusskorrigiert
Stand 14.09.2023

Dieses Material / Vorabdruck darf nur bis zum 31.10.2023 verwendet werden. Hernach steht das vollständige Lehrbuch zum käuflichen Erwerb bereit.»

Die vereinfachte Darstellung in der vorangehenden Abbildung können wir uns nur vorstellen, weil wir aus dem vorherigen Teil wissen, dass man die Komponente `self.naechster` als einen Zeiger interpretieren kann. Die Arbeit mit Zeigern nimmt uns Python aber ab. Die Definition der Klasse `Knoten` besagt nur, dass die Knoten über eine Komponente `self.naechster` verfügen und wir übergeben als Wert jeweils wiederum einen Knoten (oder `None`).

Rufen Sie

```
print(K1.naechster.
naechster.inhalt)

print(K3.inhalt)
```

auf und vergleichen Sie die ausgegebenen Werte.

4. Mit der Funktion `__str__(self)` in Zeile 6 sagen wir, was wir sehen werden, wenn wir uns mit dem Befehl `print()` einen Knoten anschauen. Wir definieren die Funktion so, dass nur die erste Komponente (der Inhalt `self.inhalt`) eines Knotens ausgegeben wird. Mit dem Befehl `str()` wandeln wir diese Zahl in eine Zeichenkette um.
Wenn wir also hinter Zeile 15 schreiben

```
print(K1)
print(K2)
print(K3)
```

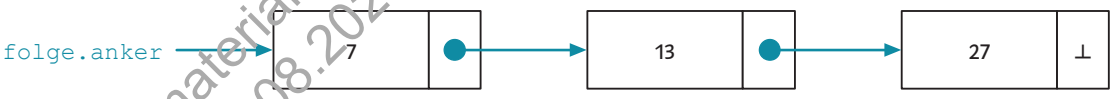
erhalten wir die drei Zahlen 7, 13 und 27 in dieser Reihenfolge.
Oder wir schreiben

```
print(K1.naechster)
```

und erhalten dieselbe Ausgabe wie bei `print(K2)`, nämlich 13.
Wir können auch

```
print(K1.naechster.
naechster)
```

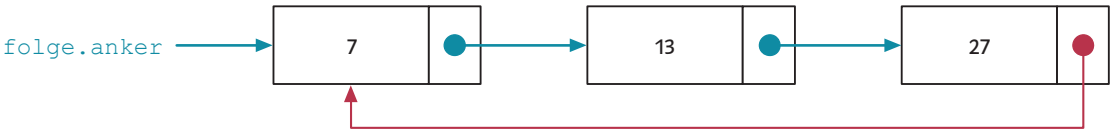
schreiben und erhalten 27 als den Inhalt von `K3 = K1.naechster.naechster`.
Schreiben Sie das Programm ab und führen Sie alle angegebenen `print()`-Befehle aus, um erste Erfahrungen zu sammeln.
In den Zeilen 9, 10 und 11 wird die Klasse `Liste` definiert. Ihre Generierungsfunktion `__init__()` besitzt, neben dem obligatorischen `self`, nur einen weiteren Parameter `anker`, welcher den Standardwert `None` besitzt. Das Attribut `self.anker` ist die einzige Komponente der Liste. Mit der Ausführung von Zeile 16 wird `folge.anker` gleich `K1`.
In unserer Darstellung mit Zeigern sieht die fertige Liste dann wie folgt aus:



Wir können diese Darstellung überprüfen, indem wir

```
print(folge.anker)
```

ausführen und 7 erhalten, also den Inhalt von `K1`, was `folge.anker.inhalt` entspricht.
Durch die Ausführung von Zeile 17 erhält man die folgende, spezielle (zyklische) Struktur:



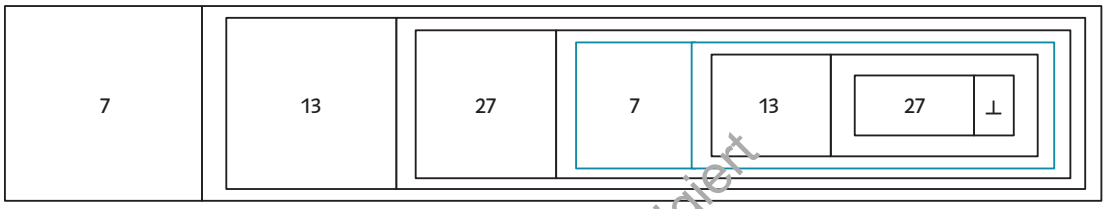
Vorabmaterial – nicht schlusskorrigiert
Stand 14.08.2023

Dieses Material / Vorabdruck darf nur bis zum 31.10.2023 verwendet werden. Hernach steht das vollständige Lehrbuch zum käuflichen Erwerb bereit.»

Überprüfen Sie dies, indem Sie die folgenden Befehle ausführen.

```
print(K3.naechster)
print(K3.naechster.naechster)
print(K3.naechster.naechster.naechster)
print(folge.anker.naechster.naechster.naechster)
```

Sie sollten die Ausgaben 7, 13, 27, 7 in dieser Reihenfolge erhalten.



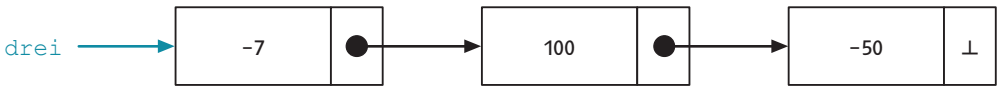
1.22 Betrachten Sie die ersten 11 Zeilen des Programms in Beispiel 1.6 mit den Definitionen der Klassen `Knoten` und `Liste`. Zeichnen Sie in der Zeigerdarstellung die Liste, die entsteht, wenn die folgenden Befehle ausgeführt werden.

```
13  X4 = Knoten(77)
14  X3 = Knoten(10, X4)
15  X2 = Knoten(5, X3)
16  X1 = Knoten(22, X2)
17  X0 = Knoten(77, X1)
18  neueListe = Liste(X0)
```

Was erhalten Sie bei der Ausführung folgender Befehle? Schätzen Sie zunächst und überprüfen Ihre Schätzung dann durch die Ausführung der Befehlsfolge.

```
print(X4)
print(X0.naechster)
print(X1.naechster)
print(neueListe.anker)
y = X0.naechster.naechster
print(y)
```

1.23 Die Klassen `Knoten` und `Liste` sind wie in Beispiel 1.6 definiert. Welche Anweisungen müssen Sie jetzt schreiben und ausführen lassen, so dass die Liste in der folgenden Abbildung entsteht?



Vorabmaterial – nicht schlusskorrigiert
Stand 14.08.2023

Dieses Material / Vorabdruck darf nur bis zum 31.10.2023 verwendet werden. Hernach steht das vollständige Lehrbuch zum käuflichen Erwerb bereit.»

Neue Konzepte und Begriffe

Objektorientierte Programmiersprachen wie auch Python ermöglichen es, **Klassen** zu definieren. Die Beschreibung der Klasse beinhaltet «Baupläne», mit denen man **Objekte** (auch Instanzen genannt) der Klasse erzeugen und Funktionen, mit den man Operationen auf den Objekten ausführen kann. Mit

```
class KlasseX:
```

beginnt man die Definition einer Klasse mit dem Namen `KlasseX`.
Mit den Zeilen

```
def __init__ (self, X1, X2, …, Xk)
    self.X1 = X1
    self.X2 = X2
    …
    self.Xk = Xk
```

definiert man eine Funktion zur Erzeugung der Instanzen von `KlasseX`. Die Instanzen bestehen aus k Komponenten `self.X1, self.X2, …, self.Xk`. Die Werte dieser Komponenten werden über die k Parameter `X1, X2, …, Xk` beim Aufruf

```
neue = KlasseX("Fuchs", None, -7, …, 13)
```

festgelegt. Dadurch entsteht ein neues Objekt (eine neue Instanz) der Klasse mit den gegebenen Werten der Komponenten. Mit der Funktion `__str__(self)` wird angegeben, was man erhält, wenn man sich eine Instanz von `KlasseX` «anschaut». Alles, was man mit `__init__()` der `KlasseX` generieren (erstellen) kann, bezeichnen wir als Objekte der `KlasseX`.

Objekte sind eine Verallgemeinerung des Konzeptes der Variablen. Objekte der `KlasseX` sind nicht Datensätze (Tuples, Vektoren) mit k Werten wie `("Fuchs", None, -7, …, 13)`. Objekte sehen wie komplexe Variablen aus, die mehrere Komponenten mit unterschiedlichen Attributen besitzen können. Mit dem Aufruf

```
T3 = (56, None)
```

wird Python ein neues Objekt `T3` einführen mit dem Zeiger auf einen Speicherbereich, in dem die Werte beider Komponenten von `T3` gespeichert werden. Das Paar `(56, None)` als der «Gesamtwert» von Variable `T3` nennt man den **Zustand** von `T3`.
Wenn man noch den Aufruf

```
T6 = (56, None)
```

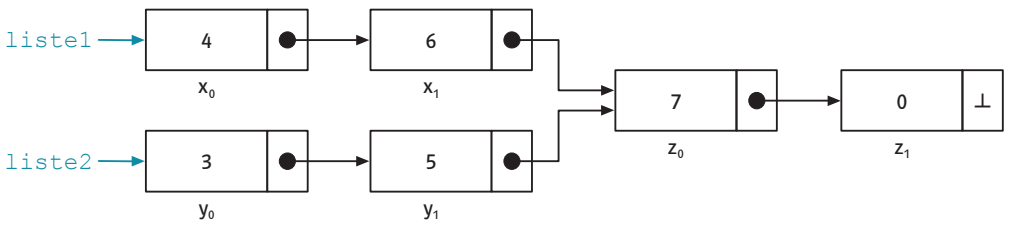
ausführt, haben wir nicht ein Objekt `(56, None)`, sondern zwei Objekte `T3` und `T6` im gleichen Zustand `(56, None)`. Im Speicher gibt es auch zwei separate Speicherbereiche, in denen die Werte (Zustände) von `T3` und `T6` gespeichert werden. Wenn wir aber statt `T6 = (56, None)` den Befehl

```
T6 = T3
```

ausgeführt hätten, würde das Objekt `T6` durch einen Zeiger auf dem Speicherbereich von `T3` erzeugt. Somit wären die Zustände von `T3` und `T6` auch gleich, aber die Zustandswerte nur in einem Speicherbereich gespeichert, zu dem beide Zeiger von `T3` und `T6` führen.

Vorabmaterial - nicht schlusskorrigiert
Stand 14.08.2023

1.24 Die Klassen `Knoten` und `Liste` sind erneut wie in Beispiel 1.6 definiert. Welche Anweisungen müssen Sie jetzt ausführen, damit die zwei verflochtenen Listen aus der folgenden Abbildung entstehen?



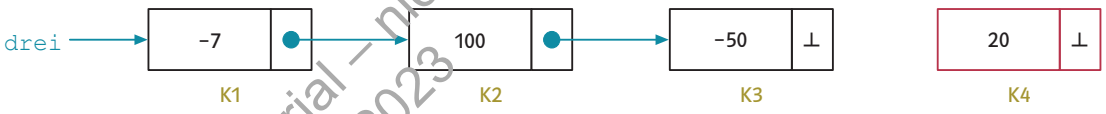
Dieses Material / Vorabdruck darf nur bis zum 31.10.2023 verwendet werden. Hernach steht das vollständige Lehrbuch zum käuflichen Erwerb bereit.»

1.25 Nutzen Sie das Konzept der objektorientierten Programmierung, um eine neue Klasse `Konto` zu definieren. Diese Klasse soll drei Komponenten (Attribute) haben: Die Kontonummer, den Namen der Kontoinhaberin oder des Kontoinhabers und den Kontostand (Geldbetrag). Beim «Anschauen» des Zustands eines Objekts (einer Instanz) der Klasse `Konto` mit `print()` soll der Kontostand ausgegeben werden.
Erzeugen Sie danach ein Konto mit der Nummer «77777», Inhaber «Lucky Luke» und Kontostand «1000000».

Operationen auf verketteten Listen

Beispiel 1.7

Bisher haben wir gelernt, wie man Klassen definiert und wie man Instanzen einer Klasse generieren kann. Wir wollen aber viel mehr. Wir wollen Operationen auf den Klasseninstanzen (Objekten) definieren, die uns die Arbeit mit ihnen (in unserem Fall Listen) dadurch erleichtern, dass sie solche Basisoperationen wie das Einfügen eines neuen Elements zur Liste automatisieren.
Diese Operationen definiert man als Funktionen, die im Rahmen der objektorientierten Programmierung **Methoden** genannt werden. Alle Funktionen innerhalb der Definition einer Klasse, auch `__init__()` und `__str__()`, bezeichnet man als Methoden der Klasse und wir werden im Weiteren diese Terminologie verwenden. Meistens dienen Methoden dem Umgang mit Objekten wie der Generierung von Objekten und der Veränderung der Zustände von schon vorhandenen Objekten.
Eine der charakteristischsten Eigenschaften von Listen ist, dass man mit sehr kleinem Aufwand (unabhängig von der Listenlänge) einen neuen Knoten an das Ende der Liste anhängen kann. Das können wir natürlich mit bereits bekannten Anweisungen in konkreten Fällen leicht umsetzen. Nehmen wir an, wir haben die Liste `drei` aus Aufgabe 1.23 und wollen am Ende einen Knoten mit Inhalt 20 einfügen.

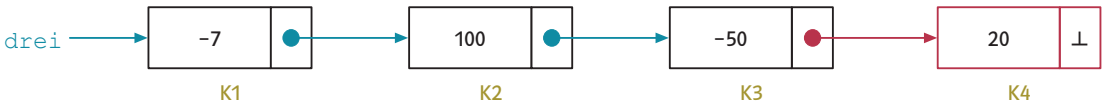


Dazu reichen zwei Anweisungen aus. Mit der ersten generiert man den neuen Knoten `K4` und mit der zweiten setzt man den Zeiger von `K3` auf `K4`.

```
K4 = Knoten(20, None)
K3.naechster = K4
```

oder

```
drei.anker.naechster.naechster.naechster = K4
```



Auf diese Art und Weise ein neues Element zur Liste hinzuzufügen, erfordert eine bekannte Benennung des aktuell letzten Knotens der Liste oder die Listenlänge. Wir wollen aber eine Methode erstellen, die für eine beliebige Liste (angegeben nur durch ihren Anker) mit unbekannter Länge und einen Wert z einen Knoten mit Inhalt z erzeugt und am Ende der Liste hinzufügt. Wir setzen dieses Vorhaben modular in zwei Schritten um. Zuerst definieren wir eine Methode, die den aktuell letzten Knoten der Liste ausgibt, und dann die gewünschte Methode für das Einfügen eines neuen Knotens. Zusätzlich definieren wir die Methode `__str()__` für Listen so, dass sie die Folge der Inhalte aller Knoten ausgibt.
Wichtig ist, dass alle Methoden einer Klasse definiert werden müssen, bevor wir die Klasse beim Programmieren verwenden. Die definierten Methoden funktionieren nur für Klassen, für die sie eingeführt worden sind. Welche Methode zu welcher Klasse gehört, erkennen wir an dem für Python typischen Einrücken nach rechts.

Vorabmaterial – nicht schlusskorrigiert
Stand 14.08.2023

Dieses Material / Vorabdruck darf nur bis zum 31.10.2023 verwendet werden. Hernach steht das vollständige Lehrbuch zum käuflichen Erwerb bereit.»

Bei Definitionen von Methoden verwenden wir wieder `self` als Platzhalter. Mit welcher Liste wir konkret arbeiten, spezifizieren wir erst beim Aufrufen mit `NamederListe.NamederMethode (Parameterwerte)`, wie zum Beispiel `drei.findeLetzten()` oder `drei.einfuegen(-7)` für die Liste `drei`.

```
1  class Knoten:
2      def __init__(self, inhalt=0, naechster=None):
3          self.inhalt = inhalt
4          self.naechster = naechster
5
6      def __str__(self):
7          return str(self.inhalt)
8
9  class Liste:
10     def __init__(self, anker=None):
11         self.anker = anker
12
13     def __str__(self):
14         anzeige = " - "
15         knoten = self.anker
16         while knoten != None:
17             anzeige = anzeige + str(knoten.inhalt) + " - "
18             knoten = knoten.naechster
19         return anzeige
20
21     def findeLetzten(self):
22         if self.anker == None:
23             return None
24         knoten = self.anker
25         while knoten.naechster != None:
26             knoten = knoten.naechster
27         return knoten
28
29     def einfuegen(self, inhalt):
30         aktuellletzter = self.findeLetzten()
31         knoten = Knoten(inhalt)
32         if aktuellletzter == None:
33             self.anker = knoten
34         else:
35             aktuellletzter.naechster = knoten
36
37 drei = Liste(None)
38 drei.einfuegen(-7)
39 print(drei)
40 drei.einfuegen(100)
41 drei.einfuegen(-50)
42 print(drei)
43 drei.einfuegen(20)
44 print(drei)
```

Vorabmaterial – nicht schlusskorrigiert
Stand 14.08.2023

Dieses Material / Vorabdruck darf nur bis zum 31.10.2023 verwendet werden. Hernach steht das vollständige Lehrbuch zum käuflichen Erwerb bereit.»

In den Zeilen 37 bis 44 erzeugt das Programm die Liste `drei` mittels Einfügens aller vier Elemente eins nach dem anderen mit der Funktion `einfuegen()`. Mit `print(drei)` überprüfen Sie, wie die Liste tatsächlich schrittweise aufgebaut wird. Der Vorteil ist, dass Sie sich gar nicht um die Benennung der Knoten sowie das «Setzen» der Zeiger kümmern müssen.

1.26 Nutzen Sie das in Beispiel 1.7 definierte Programm mit der Definition der Klassen `Knoten` und `Liste`, um mittels Einfügens eine Liste `liste26` mit den Inhalten -4, 3, -7, 6, 12 zu generieren.

1.27 Ergänzen Sie das in Beispiel 1.7 definierte Programm für die Klassen `Knoten` und `Liste` um eine Methode `suchen(self, inhalt)`, welche die Knoten einer Liste der Reihe nach durchgeht und überprüft, ob das Datum (der Inhalt) des Knotens mit dem Argument der Methode übereinstimmt. Wenn dies der Fall ist, soll der Knoten zurückgegeben werden, andernfalls soll die Ausgabe `None` sein.

Vorabmaterial – nicht schlusskorrigiert
Stand 14.08.2023

Neue Konzepte und Begriffe

Eine **Klasse** definiert einen Bauplan für **Objekte** (auch **Klasseninstanzen** genannt), die eine gemeinsame Struktur haben, und Operationen, die man mit diesen Objekten ausführen kann. Diese Operationen werden als Funktionen umgesetzt, welche man **Methoden** nennt.
Die Definition einer Klasse besteht aus:

1. Einer Methode, die Objekte der Klasse generiert und somit genau bestimmt, welche Objekte zur Klasse gehören.
2. Einer Methode, die bestimmt, was man erhält, wenn man sich den Zustand eines Objektes der Klasse mit `print()` «anschaut».
3. Einer Menge von weiteren Methoden, die als Funktionen die Operationen auf den Objekten definieren. Eine Methode kann z. B. den Zustand eines Objektes ändern, eine Information über den Zustand des angegebenen Objektes ausgeben oder ein Merkmal (z. B. eine Zahl) eines Objektes suchen und ausgeben.

Objektorientierte Programmierung ist sehr gut zur Datenverwaltung geeignet. Einen Datensatz (Kapitel 5, Band «Data Science und Sicherheit») kann man als einen Vektor (ein Tupel) bestehend aus einer endlichen Folge von Attributen bezeichnen. Eine solche Struktur kann man leicht als eine Klasse beschreiben, wobei die Attribute der Klasse mit denen des Datensatzes übereinstimmen. Die Operationen auf Datensätzen kann man dann wiederum als Methoden umsetzen.
Klassen ermöglichen aber viel komplexere Strukturen als Datensätze durch die Modularität bei der Generierung. Ein Objekt einer Klasse kann eine Komponente eines Objektes einer anderen Klasse sein (wie bei Knoten und Listen). Mit der Definition einer Klasse baut man sich ein Gerüst, mit dem man sehr effizient mit den betrachteten Objekten arbeiten und modular komplexe Objekte konstruieren kann.

Dieses Material / Vorabdruck darf nur bis zum 31.10.2023 verwendet werden. Hernach steht das vollständige Lehrbuch zum käuflichen Erwerb bereit.»

Geschichtlicher und gesellschaftlicher Kontext

In den sechziger Jahren des zwanzigsten Jahrhunderts haben Ole-Johan Dahl und Kristen Nygaard an der Universität Oslo die erste objektorientierte Sprache **Simula-67** entwickelt, um Simulationen von Prozessen in naturwissenschaftlichen Fächern zu fördern. Die in Simula-67 eingeführte Objektorientierung wurde zu einem weit verbreiteten Ansatz für das modulare Lösen von Problemen und für die modulare Softwareentwicklung. Die Objektorientierung ermöglicht, neue Datentypen zu definieren und effizient mit diesen zu arbeiten. Unter einem Datentyp versteht man eine Menge von Objekten und eine Menge von Operationen, die man auf diesen Objekten ausführen kann. Einfachste Beispiele von Datentypen sind unterschiedliche Zahlenmengen mit geeigneten arithmetischen Operationen.

Alan Kay, Dan Ingalls und Adele Goldberg brachten mit ihrem Team am Xerox PARC in den 1970er-Jahren die Objektorientierung mit der Erfindung der Programmiersprache **Smalltalk** an die Grenze dieses Einsatzes. Hier sind sogar die klassischen elementaren Datentypen wie Integer als Objekte einer Klasse repräsentiert und alles, was vorkommt, wird als ein Objekt betrachtet.

Moderne Programmiersprachen wie beispielsweise Python ermöglichen die Objektorientierung, verzichten aber dabei nicht auf die klassischen prozeduralen Ansätze der Algorithmik und der Programmierung.

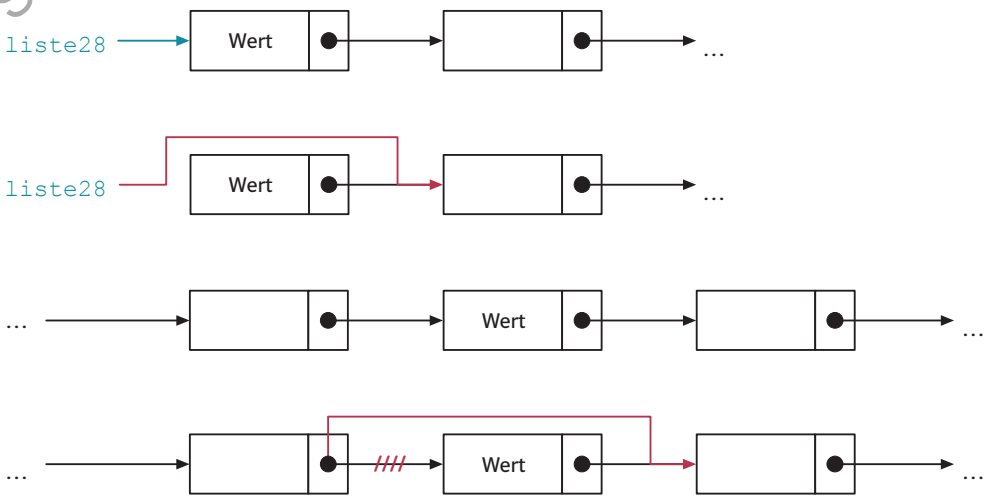
Vorabmaterial – nicht schlusskorrigiert
Stand 14.08.2023

1.28 Verkettete Listen sind eine Datenstruktur, die zu einer dynamischen Datenverwaltung verwendet wird. Somit ist das Löschen eines Elements einer Liste eine genauso wichtige Operation wie das Einfügen eines neuen Elements.

Entwickeln Sie eine Methode für die Klasse `Liste`, die Folgendes bewirkt:

1. Die Methode erhält als Parameter einen Wert und sucht von links nach rechts nach einem Knoten mit diesem Wert.
2. Wenn es keinen Knoten mit demselben Inhalt wie dem gegebenen Wert gibt, dann bleibt die Liste unverändert.
3. Wenn ein Knoten mit dem gesuchten Inhalt (Wert) gefunden wird, wird dieser Knoten aus der Liste entfernt (somit entfernt man den am weitesten links stehenden Knoten mit dem gegebenen Inhalt).

Dabei sollten Sie beide Fälle in der Abbildung beachten: Der entfernte Knoten ist entweder der erste in der Liste (man muss die Verankerung ändern) oder er ist ein Knoten mit einem Vorgänger und einem Nachfolger in der Liste.



Die Werte des entfernten Knotens muss man nicht löschen, der Knoten ist einfach nicht mehr in `liste28` erreichbar. Aus Sicht der Datenstruktur `Liste` geht es nur um die Änderung eines Zeigers.

Dieses Material / Vorabdruck darf nur bis zum 31.10.2023 verwendet werden. Hernach steht das vollständige Lehrbuch zum käuflichen Erwerb bereit.»

1.29 In den Beispielen 1.6 und 1.7 und in den nachfolgenden Aufgaben haben wir ein Modul entwickelt, das die Definitionen und einige Methoden der Klassen `Knoten` und `Liste` beinhaltet. Ihre Aufgabe ist jetzt, die Methode `einfuegen()`, die einen neuen Knoten mit einem gegebenen Inhalt am Ende der Liste anfügt, durch eine andere Methode zum Einfügen des Knotens zu ersetzen. Diese neue Variante des Einfügens soll den Knoten vor dem ersten Knoten der Liste einfügen, deren Inhalt grösser gleich dem Inhalt des neuen Knotens ist. Somit bleiben die Inhalte der Liste nach dem Einfügen des neuen Knotens sortiert, falls die Liste vorher sortiert war.
Anders formuliert: Wenn man mit dem neuen Einfügen schrittweise Knoten um Knoten eine Liste erzeugt, sind die Inhalte der Liste aufsteigend sortiert.

1.30 Entwickeln Sie ein neues Modul mit den Klassen `Knoten` und `Liste` für doppelt verkettete Listen (Lernaufgabe 1.20). Somit sollen die Knoten drei Komponenten (Attribute) haben, nämlich `inhalt`, `vorgaenger` und `nachfolger` (`naechster`). Ausser den Definitionen zur Erzeugung der Klassen mittels `__init__()` sollte das Modul auch alle Methoden beinhalten, die man in der Lösung von Aufgabe 1.29 für die einfach verkettete Liste hat.

Vorabmaterial – nicht schlusskorrigiert
Stand 14.08.2023

Was Sie gelernt haben

Arrays sind eine statische Datenstruktur, die einen direkten Zugang zu jedem Element des Arrays ermöglicht. Das kommt dadurch, dass die Grösse der Speichereinheiten einheitlich für alle Elemente im Voraus gesetzt wird, sowie auch die maximal mögliche Anzahl der Elemente. Der Computer hat nur einen Zeiger auf den Anfang des Arrays, das kompakt (lückenlos) gespeichert ist, und kann die Adresse jedes Elements einfach berechnen. Arrays eignen sich sehr gut für das Sortieren sowie für die Suche nach einem Wert (Datum). Die Datenstruktur «Liste» in Python entspricht dynamischen Arrays, da ihre Grösse im Ablauf der Berechnungen modifiziert werden kann.
Verkettete Listen sind eine dynamische Datenstruktur, die für Datensammlungen mit ständigen Updates erstellt wurde. Für die Umsetzung benötigen Listen keine Reservierung eines kompakten Speicherbereichs. Dank Zeigern (Speicheradressen) als eine Komponente aller Elemente können einzelne Elemente der Liste beliebig im Speicher platziert werden. Der Nachteil von Listen ist, dass die Suche nach einem Wert im Durchschnitt das «Anschauen» der Hälfte aller Elemente der Liste erfordert. Das ist exponentiell in Relation zum Aufwand der binären Suche in sortierten Arrays.
Verkettete Listen kann man dank Objektorientierung in Python selbst erzeugen. Dazu verwendet man das Konzept von Klassen, die als Baupläne aller Objekte einer Klasse dienen und somit eindeutig die Objekte der Klasse beschreiben.
Zur Definition einer Klasse gehören ausser der Erzeugungsfunktion `__init__()` noch weitere Funktionen, die man Methoden nennt. Methoden definieren Operationen auf den Objekten der Klasse und ermöglichen eine effiziente Arbeit mit ihnen.

Dieses Material / Vorabdruck darf nur bis zum 31.10.2023 verwendet werden. Hernach steht das vollständige Lehrbuch zum käuflichen Erwerb bereit.»