Was versteht man unter Algorithmen und Datenstrukturen, und wie hängen diese beiden Begriffe zusammen?

Unter Algorithmus versteht man eine eindeutige Handlungsvorschrift, deren Befolgung auf die Lösung eines Problems abzielt. Datenstrukturen beschreiben wie die Daten relativ zueinander angeordnet sind und wie auf die einzelnen Datenelemente zugegriffen werden kann. Algorithmen setzen bestimmte Datenstrukturen voraus, sodass Algorithmen nur zusammen mit Datenstrukturen entwickelt werden können.

Unter welchen Bedingungen sind zwei Algorithmen bzw. Datenstrukturen gleich? Wann sind sie es nicht?

Algorithmen sind gleich, wenn sie dieselbe Handlungsvorschrift beschreiben. Ihre Implementierung spielt keine Rolle. Beispielsweise kann ein Algorithmus, das Zahlen aufsummiert rekursiv oder iterativ implementiert werden. Algorithmen sind verschieden, wenn sie ungleiche Handlungsvorschriften enthalten. Die gauß'sche Summenformel ist zum Beispiel ein anderer Algorithmus als das Aufsummieren der Zahlen in einer for-Schleife, auch wenn beide Algorithmen dasselbe Problem lösen. Analoges gilt für Datenstrukturen. Die Implementierung spielt keine Rolle. Beispielsweise könnte ein Stack durch eine verkettete Liste oder durch einen Array implementiert werden.

Nennen Sie fünf unterschiedliche Datenstrukturen.

Arrays
Hashtabelle
Verkettete Listen
Binäre Bäume
Stacks

Wozu dienen Lösungsstrategien?

Allgemein dienen Lösungsstrategien zum Erreichen erfolgsentscheidender Ziele. In der Programmkonstruktion ist vor allem ein Ziel von überragender Bedeutung: *Einfachheit*. Lösungsstrategien in der Programmkonstruktion haben daher vor allem eine Vereinfachung von Algorithmen und Datenstrukturen zum Ziel.

Warum sind so viele Datenstrukturen rekursiv?

Rekursive Datenstrukturen lassen sich nur sehr schwer oder kaum durch nicht-rekursive Datenstrukturen ersetzen und sind sehr nützlich, da sie zum Beispiel ihre Größe dynamisch zur Laufzeit ändern und somit nicht nur effizientere Programme gegenüber andere Lösungen ermöglichen, sondern vor allem auch einfachere und übersichtlichere Programme.

Welche Zugriffoperationen haben Stacks, verkettete Listen und binäre Bäume üblicherweise?

Stacks: Push, Pop
Verkettete Listen: add, remove, contains
Binäre Bäume: add, remove, contains

Welche charakteristischen Merkmale zeichnen eine Liste und einen binären Baum aus?

Verkettete Listen: Eine verkettete Liste ist ein zusammenhängender Graph, bei dem jeder Knoten höchstens einen Nachfolger hat. Die Elemente in den Knoten besitzen keinen Index. Die Suche gestaltet sich daher aufwendig. Die Anzahl der Knoten wird – wie auch bei Binärbaume – theoretisch nur vom Speicherplatz des Computers begrenzt und kann sich dynamisch zur Laufzeit erhöhen oder reduzieren.

Binärbäume: Ein nicht-leerer Binärbaum ist ein zusammenhängender gerichteter Graph, in dem jeder Knoten mit einem Label versehen ist und genau einen Vorgänger und höchstens zwei Nachfolger hat. Die Wurzel eines binären Baumes ist eindeutig bestimmt. Jeder Knoten im linken Teilbaum hat ein Label das kleiner dem Label der Wurzel ist und jeder Knoten im rechten Teilbaum ein Label das größer dem Label der Wurzel ist. Die Suche in einem Binärbaum ist gegenüber verketten Listen effizienter, das Hinzufügen neuer Knoten aufwändiger.

Wie hängen Datenstrukturen mit gerichteten Graphen zusammen?

Praktisch alle rekursiven Datenstrukturen lassen sich durch gerichtete Graphen veranschaulichen. Eigenschaften dieser Graphen entsprechen auch Eigenschaften der Datenstrukturen.

Wodurch unterschieden sich rekursive Methoden von entsprechenden iterativen?

Prinzipiell lässt sich jede rekursive Methode durch eine iterative ersetzen und umgekehrt. Rekursive Methoden implementieren dieselben Algorithmen meist kürzer und einfacher. Ein Grund dafür sind beispielsweise die lokalen Variablen, die iterative Methoden für Schleifen brauchen, während rekursive Methoden mit this auskommen. Ein Nachteil rekursiver Methoden besteht in der hohen Anzahl an Methodenaufrufen, was Zeit und Speicherplatz kostet. Trotzdem haben iterative Methoden aufgrund ihres komplizierteren Codes in der Regel einen höheren Ressourcenbedarf.

Was haben rekursive Datenstrukturen und rekursive Methoden mit vollständiger Induktion gemeinsam?

Mit vollständiger Induktion kann bewiesen werden, dass nach Beendigung eines Aufrufs für alle betrachteten Datenelemente bestimmte Eigenschaften erfüllt sind. Ein gutes Verständnis der vollständigen Induktion ist daher sehr hilfreich im Umgang mit rekursiven (und iterativen) Methoden.

Wie kann man den Aufwand eines Algorithmus abschätzen?

Bei der Aufwandsabschätzung eines Algorithmus werden Details, wie Implementierung, Hardware oder Programmiersprache vernachlässigt. Man konzentriert sich auf die sogenannte Ordnung. Steigt zum Beispiel der Aufwand eines Algorithmus linear mit der Anzahl an Elementen (wie zum Beispiel in einer verketteten Liste) spricht man von einem Aufwand von O(n). Steigt der Aufwand mit jedem neuen Element quadratisch an, so ist der Aufwand $O(n^2)$. Steigt er exponentiell an, so liegt der Aufwand bei $O(2^n)$ usw. Konstanten werden bei derartigen Abschätzungen vernachlässigt.

Wofür stehen O(1), O(log(n)), O(n), O(n * log(n)), $O(n^2)$, $O(2^n)$?
Wie wirkt sich eine Verdoppelung oder Verhundertfachung von n aus?

O(1): Aufwand bleibt konstant.
O(n): Aufwand wächst linear.
O(log(n)): Aufwand wächst logarithmisch.
$O(n^2)$: Aufwand wächst quadratisch.
$O(2^n)$: Aufwand wächst exponentiell.
O(n * log(n)): Aufwand wächst super-linear.
Eine Verdoppelung oder Verhundertfachung ist bei derartigen Abschätzungen vernachlässigbar klein.

Wieso kann man konstante Faktoren bei der Aufwandsabschätzung einfach ignorieren?

Konstante Faktoren fallen bei wachsendem n kaum ins Gewicht. Es spielt keine Rolle ob eine Aufgabe mit 10, 100 oder 1000 Anweisungen gelöst wird. Diese Unterschiede sind vernachlässigbar klein und werden von jeder anderen Art von Kosten dominiert.

Wie hoch ist der Aufwand für das Suchen bzw. Einfügen in verketteten Listen, in binären Bäumen und in Hashtabellen im Durchschnitt und im schlechtesten Fall. Was ist der jeweils schlechteste Fall und wann tritt er ein?

Verkettete Liste: Suche: durchschnitt: O(n)/maximal: O(n); Einfügen: O(1)
schlechtester Fall: gesamte Liste muss durchgegangen werden
Binärer Baum: Suche: durchschnitt: O(log(n))/maximal :O(n)
schlechtester Fall: entarteter Baum (Baum wird zur Liste)
Hashtabelle: Suche :durchschnitt: fast konstant/maximal: O(n); Einfügen im besten fall
O(1), im schlechtesten Fall O(n), schlechtester Fall: alle Hashwerte gleich sind.

Wie funktionieren Bubblesort, Mergesort und Quicksort? Wie hoch ist der Aufwand dafür im Durchschnitt und im schlechtesten Fall?

Bubblesort: Man durchläuft alle Elemente der Reihe nach und vergleicht jedes Element mit seinem Nachfolger. Ist der Nachfolger größer werden die Elemente vertauscht. Dieses Verfahren wird solange wiederholt bis sich die Reihenfolge nicht mehr ändert. Aufwand: $O(n^2)$

Mergesort: Feld wird solange geteilt, bis nur einzelne Elemente übrig sind, dann werden die Elemente Ebene für Ebene zusammengesetzt. Aufwand: O(n * log(n))

Quicksort: Element wird willkürlich gewählt (Pivot-Element). Links davon kommen die kleineren und rechts davon die größeren Elemente. Das Verfahren wird rekursiv wiederholt.
Aufwand: $O(n * \log(n))$ / $O(n^2)$

Was ist eine binäre Suche?

Zuerst wird das mittlere Element des Felds überprüft. Es kann kleiner, größer oder gleich dem gesuchten Element sein. Ist es kleiner wird in der unteren Hälfte gesucht, ist es größer wird in der oberen Hälfte gesucht. Dieses Verfahren wird rekursiv angewendet, bis das gesuchte Element gefunden wurde. Vorrausetzung für eine binäre Suche ist ein sortiertes Feld.

Was unterscheidet generische von nicht-generischen Klassen?

Nicht-Generische Klassen haben einen „vordefinierten" Typ, auf dem der Programmcode aufbaut. Generische Klassen können beliebige Typen enthalten, die in spitzen Klammern als Typparameter angegeben werden. Dies ermöglicht beispielsweise das Erstellen einer Klasse List, die beliebige Typen als Elemente enthalten kann. Ohne eine generische Klasse müsste man List für jeden Typ einzeln schreiben. Eine (veraltete und schlechte) Alternative zu generischen Klassen, sind Klassen mit dem Typ Object, womit man aber nicht verhindern kann, dass unerwünschte Objekte in der Datenstruktur landen.

Was unterschiedet einen Typ von einem Typparameter? Kann man Typen und Typparameter gleich verwenden?

Typparameter sind im Prinzip Variablen von Typen. So hat zum Beispiel in einer Klasse List <A> die Variable „A word" noch keinen Typ. In einer Instanz von List <String> hat word hingegen das den Typ String, in einer Instanz von List <Integer> den Typ Integer usw. Typparameter lassen sich nicht uneingeschränkt wie Typen verwenden. Beispielsweise können keine Instanzen von Typparameter erzeugt werden und kein Array das Instanzen eines Typparameters hat.

Wozu dienen Schranken bei gebundener Generizität?

Eine generische Klasse mit einer Schranke erlaubt nur Typen als Typparameter, die Untertypen des Typs der Schranke (bzw. Untertypen der durch die Schranke gebundene Typparameter) sind. Damit wird garantiert, dass alle Typen „kompatibel" zur generischen Klasse sind. Beispielsweise kann man als Schranke ein Interface wählen, das eine Methode int value() enthält. Dann kann der Typparameter nur Typen erhalten in deren Klasse die Methode in value() enthalten ist.

Welchen speziellen Zweck hat rekursiv gebundene Generizität?

Rekursiv gebundene Generizität erlaubt Typen von formalen Parametern so einzuschränken, dass sie gleich den Klassen sind, in denen die Methoden stehen.

Inwiefern ähneln sich Untertypenbeziehungen und Generizität? Wodurch unterschieden sie sich in ihrer Anwendbarkeit?

Untertypenbeziehungen und Generizität sind beide eine Form der Abstraktion. Bei der Generizität kann ein einziger Code für Datenstrukturen für unterschiedliche Elementtypen verwendet werden. Die Datenstrukturen selbst bleiben dabei gleich. Beispielsweise ist eine verkettete Liste genauso auf String anwendbar wie auf Integer. Bei Untertypenbeziehungen bleiben die Elementtypen gleich, aber die Datenstrukturen variieren. Beispielsweise spielt es keine Rolle, ob die Daten in einer verketteten Liste oder in einem Array abgelegt werden. Die Datenstruktur kann leicht durch eine andere geändert werden. Generizität und Untertypenbeziehungen sind unterschiedliche Abstraktionsformen. Untertypenbeziehungen dienen dazu, Implementierungsdetails lokal gekapselt zu halten und durch Ersetzbarkeit den Austausch bzw. die Wiederverwendung von Programmteilen zu erleichtern. Generizität unterstützt die Ersetzbarkeit nicht. Stattdessen erspart Generizität durch direkte Wiederverwendung von Programmcode Arbeit beim Erstellen von Containerklassen.

Was sind und warum verwendet man Iteratoren?

Iteratoren lesen Elemente eines Containers nacheinander aus. Es ist möglich, mehrere Iteratoren gleichzeitig auf demselben Container zu verwenden. Zugriffsoperationen, direkt im Container und ohne Iterator, verändern den Container, mehrere gleichzeitige Durchläufe würden sich gegenseitig beeinflussen. Darin liegt auch der Grund für die Verwendung von Iteratoren: Iteratoren verändern den Container *nicht*.

Welche Schwierigkeiten treten bei der Verwendung von Iteratoren in Zusammenhang mit Rekursion häufig auf? Wie löst man sie?

Rekursion ist bei Iteratoren nicht möglich, weil das Durchwandern nach jedem gefundenen Element abgebrochen werden muss und erst wieder beim nächsten Aufruf fortgesetzt werden kann. Eine Lösung des Problems bieten Stacks. Programmiersprachen verwenden einen für Programmierer nicht sichtbaren Stack, um darin die Variablen geschachtelter Methodenaufrufe zu speichern, auch die von rekursiven Aufrufen. Daher ist ein Stack oft eine hilfreiche Datenstruktur, um eine rekursive Methode in eine nicht-rekursive umzuwandeln.

Durch welches spezielle Sprachkonstrukt unterstützt Java die Verwendung von Iteratoren?

Innere Klassen (Klassen die zu bestimmten Objekten gehören und uneingeschränkten Zugriff auf diese Objekte ermöglichen).

Wodurch wird die Verwendung fertiger Programmteile erschwert? Wie kann man den Ursachen dafür begegnen?

Unkenntnis: Man weiß nichts über deren Existenz oder kann sie nicht anwenden.
Unterschiedliche Modelle: Die fertige Lösung passt nicht gut zum restlichen Programm.
Mangelndes Vertrauen
„Ich kann es besser“: Man glaubt eine bessere oder zumindest eine angepasste Lösung gefunden zu haben.

Mangelndem Vertrauen und Wissen kann man vorbeugen, indem man sich genauere Informationen über die Programmteile verschafft und man gegebenenfalls die Teile auch selbst testet.

Welche Vor- und Nachteile hat die Top-Down-Strategie gegenüber der Bottom-Up-Strategie? Wie lassen sich diese beiden Strategien miteinander kombinieren?

Bei der Top-Down-Strategie fällt es leichter den Überblick zu bewahren. Diese Strategie führt allerdings oft zu einem imperativen Programmierstil. Bei der Bottom-Up-Strategie kann man hingegen das Ziel aus den Augen verlieren. Dafür führt die Bottom-Up-Strategie eher zu einem objektorientierten Programmierstil. Die beiden Strategien lassen sich kombinieren, indem Klassen die sicher gebraucht werden Bottom-Up erstellt werden, während die grobe Programmstruktur Top-Down erstellt wird.

Für welche Aufgaben bietet sich die schrittweise Verfeinerung an?

Große Aufgaben, die in ihrer Gesamtheit nicht durchschaubar sind.

Mit welchen Teilaufgaben sollte man bei der schrittweisen Verfeinerung beginnen? Warum ist das so?

Beginnen sollte man mit den schwierigsten Teilaufgaben auf die es am ehesten ankommt. Die Schwerpunktsetzung wird später im Programm sichtbar. Der Teil mit dem angefangen wurde, wird vermutlich ausgereifter sein, als die später erstellten Teile.

Welche Vorteile und Schwierigkeiten können sich aus der schrittweisen Verfeinerung ergeben?

Vorteile: gute Rückmeldung über die bisherige Qualität des Programms, was wiederum nützlich für die Weiterentwicklung ist (Erfahrungen gehen in weitere Entwicklung ein); Flexibilität bei Anforderungsänderungen
Nachteile: Bei einer Erweiterung des Programms kann es zu Problemen kommen, da man feststellt, dass Datenstrukturen oder die gewählte Faktorisierung für den neuen Programmteil nicht oder nur unzureichend geeignet sind.