Rekursive Datenstrukturen

Bisher ist jede SCHLANGE bezüglich ihrer Aufnahmefähigkeit begrenzt. Das wollen wir ändern. Unsere Datenstrukturen sollen immer genau die richtige Länge haben, also weder leere Plätze aufweisen noch an Platzmangel leiden. Eine solche „mitwachsende/mitschrumpfende“ Datenstruktur nennt man ***dynamische Datenstruktur***.

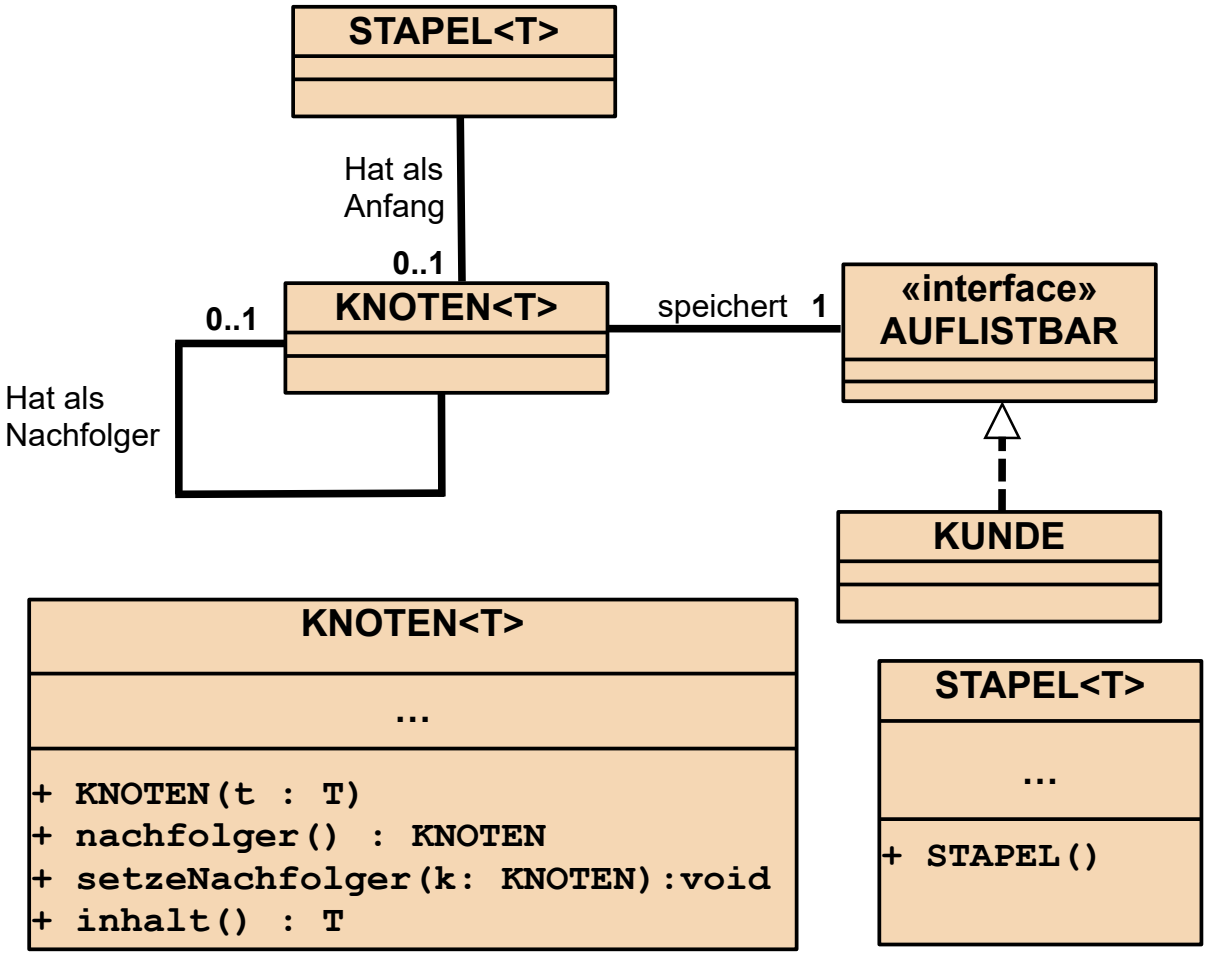
Hierzu müssen wir aber von den Arrays absehen und uns selbst eine entsprechende Datenstruktur zurecht legen. Besondere Beachtung bekommt dabei wieder das

Prinzip der Trennung von Struktur und Inhalt

- Wir bauen also eine Art ***Behälter, der ein Element aufnimmt.***
- Jeder ***Behälter hat einen anderen Behälter als Nachfolger*** oder keinen Nachfolger.

Datenstrukturen, die sich auf diese Art ***selbst referenzieren*** bezeichnet man als ***rekursive Datenstruktur***.

So kommen wir auf folgendes Modell:



Zunächst implementieren wir nur die Klasse STAPEL. ***Nutze hierfür ein neues BlueJ-Projekt „Datenstrukturen mit dynamischer Länge“***

Aufgabe 1:

Erstelle die Klasse KUNDE und das Interface AUFLISTBAR analog zu vorher!

Erstelle die beiden Klassen KNOTEN und STAPEL und ermittle aus dem Klassendiagramm die Referenzattribute, welche STAPEL und KNOTEN besitzen!

Implementiere die angegebenen Methoden!

Die Methode ***istLeer() : boolean***

Ein STAPEL ist genau dann **leer**, wenn sein Anfangs-KNOTEN `null` ist.

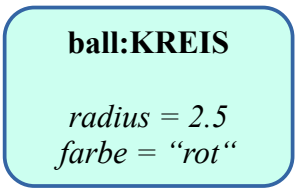
Einschub: Objektdiagramme

Objekte werden – wie bereits bekannt – durch ***Objektkarten*** veranschaulicht. Sie beinhalten neben dem Objektnamen und der dazugehörigen Klasse auch die Attribute sowie deren Werte zu einem bestimmten Zeitpunkt. Wir unterscheiden zwischen einfachen und erweiterten Objektkarten:

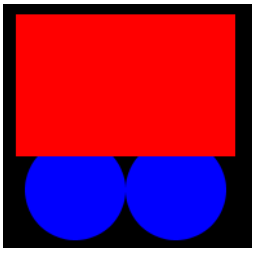
Beispiele:

einfache Objektkarte

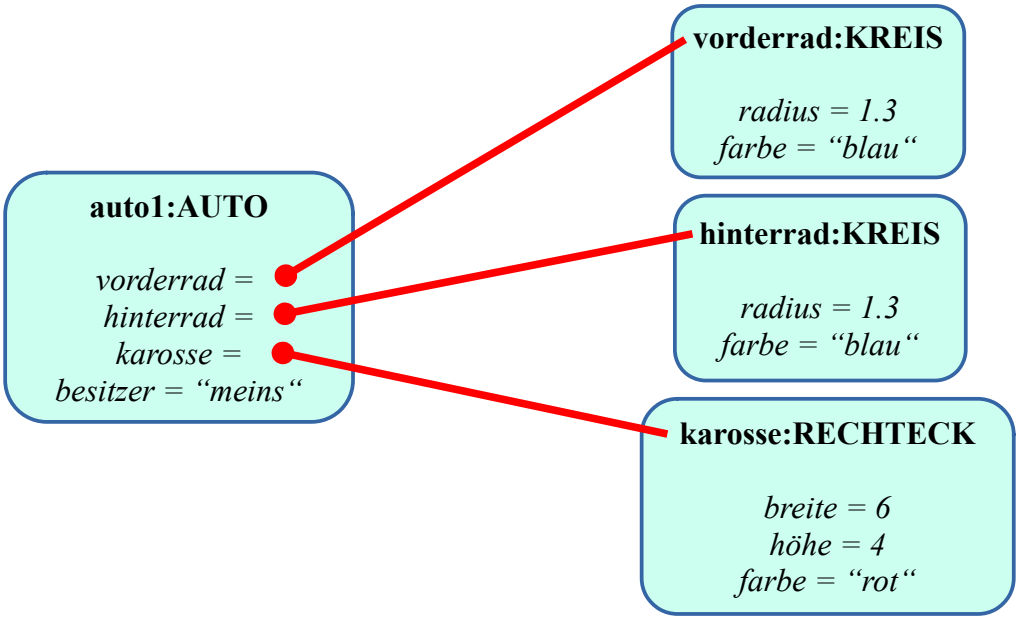
erweiterte Objektkarte



Zwischen Objekten können (Referenz-)***Beziehungen*** bestehen. Ein Objektdiagramm stellt diese Beziehungen grafisch dar. Häufig können die primitiven Attribute der Inhaltsklasse weggelassen werden, sofern es die Aufgabenstellung erlaubt.



Beispiel für ein Objektdiagramm:



*Die Methode **vorneEinfügen(T t) : void***

Prinzipiell gibt es **zwei verschiedene Ausgangssituationen**:

1. Der ***STAPEL ist noch leer*** (der Anfangs-KNOTEN ist also noch `null`).
   In diesem Fall muss für das einzufügende Element ein neuer KNOTEN geschaffen werden.
   Der Anfangs-KNOTEN ist nun nicht mehr `null` sondern dieser neue KNOTEN.

2. ***Der STAPEL ist nicht mehr leer*** (der Anfangs-KNOTEN ist also nicht `null`).
   In diesem Fall muss auch erst ein neuer KNOTEN geschaffen werden. Dieser KNOTEN wird aber als Nachfolger den bisherigen Anfangs-KNOTEN erhalten.
   Der neue Knoten inklusive Nachfolger wird nun als neuer Anfang eingehängt.

Bei genauerer Betrachtung fällt auf, dass das Vorgehen von **Fall 2** auch im **Fall 1** angewendet werden kann. Wenn der alte Anfangs-KNOTEN `null` ist und wir ihn zum Nachfolger-KNOTEN des neuen KNOTENs machen, hat der neue KNOTEN eben gerade null als Nachfolger – genau das, was wir brauchen ***:-)***

Aufgabe 2:

a) Zeichne ein Objektdiagramm eines leeren STAPELs.

b) Zeichne ein Objektdiagramm dieses STAPELs, wenn ein KUNDE „Marc“ eingefügt wurde.

c) Zeichne ein Objektdiagramm dieses STAPELs, wenn ein weiterer KUNDE „Julia“ eingefügt wurde.

d) Programmiere die Methode **`vorneEinfügen(...)`**.

e) Erstelle in BlueJ einen STAPEL und füge die Kunden „Marc“ und „Julia“ ein. Überprüfe anhand der roten Objektkarten in BlueJ, ob es analog zu deinem Diagramm aus der Aufgabe 2c) ist.

*Die Methode **vorneEntfernen() : T** für den STAPEL*

Hier gibt es streng genommen **3 verschiedene Ausgangssituationen**:

1. ***Der STAPEL ist leer***.
   Gib `null` zurück.

2. ***Der STAPEL hat genau einen KNOTEN***, den Anfangs-KNOTEN (dessen Nachfolger `null` ist).
   Dann muss sich der STAPEL das Element dieses Knotens „merken/zwischen-speichern“, den Anfangs-KNOTEN auf `null` setzen und zuletzt das „gemerkte“ Element zurückgeben.

3. ***Der STAPEL hat mehrere KNOTEN.***
   Dann muss sich der STAPEL wieder das Element des Anfangs-KNOTENs „merken/zwischenspeichern“, den Nachfolger des Anfangs-KNOTENs als neuen Anfangs-KNOTEN setzen und anschließend den „gemerkten“ KNOTEN zurückgeben.

Bei genauerer Betrachtung funktioniert die Lösung des Falls 3 auch im Fall 2, sodass hier eine Fallunterscheidung mit nur einer Alternative ausreicht.

Aufgabe 3:

a) Entferne ausgehend vom Objektdiagramm aus der Aufgabe 2c) den ersten Kunden und zeichne das Diagramm. Entferne erneut und zeichne das Diagramm.

b) Programmiere die Methode **`vorneEntfernen()`**.

Die Methode ***gibLänge() : int***

Diese Methode **erscheint zunächst unmöglich** ohne vorher **eine Zählvariable** im STAPEL einzuführen und diese in den bereits geschriebenen Methoden zu berücksichtigen.

Das Besondere an rekursiven Strukturen ist, dass wir uns das sparen können, indem wir die ***rekursive Struktur*** **ausnutzen**:

- Die ***Klasse STAPEL*** kann 2 verschiedene Fälle vorfinden:
  - Der STAPEL gibt die Länge 0 zurück, wenn sein Anfangs-KNOTEN `null` ist.
  - Wenn der STAPEL nicht leer ist, dann fragt er einfach seinen Anfangs-KNOTEN und gibt dessen Antwort zurück. (siehe nächster Punkt)

- In der ***Klasse KNOTEN*** gibt es 2 verschiedene Fälle:
  - Ein KNOTEN gibt 1 zurück, wenn sein Nachfolger `null` ist.
  - Ansonsten fragt er seinen Nachfolger und zählt 1 zu dessen Antwort hinzu und gibt dann diesen Wert zurück.

Aufgabe 4:

a) Programmiere die Methode **`gibLänge()`**.

b) Schau dir die Präsentation zu der Methode **`gibLänge()`** an.

c) Betrachte nochmal die Objektdiagramme von Aufgabe 2a-c) und versuche das oben beschriebene Verfahren für jede der drei Diagramme durch zu spielen.

d) Zeichne selbst ein Sequenzdiagramm zum Methodenaufruf **`gibLänge()`** eines STAPEL-Objekts, wie es im Objektdiagramm von Aufgabe 2c) dargestellt ist. (ohne Vorlage aus der Präsentation)

Datenstruktur LISTE

Die Klasse STAPEL kann keine Methode hintenEinfügen() bekommen, denn das verletzt das Prinzip LIFO. Umgekehrt kann eine Klasse SCHLANGE nicht die Methode vorneEinfügen() besitzen, denn sonst ist das Prinzip FIFO nicht gewährleistet.

Die Datenstruktur ***LISTE*** unterliegt keinem dieser Prinzipien. Eine Liste verwaltet eine beliebige Menge an Elemente, welche an ***beliebiger Stelle eingefügt*** und von ***beliebiger Position entfernt*** werden können.

*Die Methode **hintenEinfügen(T t) : void** für die LISTE*

Benenne deine Klasse STAPEL in LISTE um!

Möchte man die Methode hintenEinfügen(…) der Klasse LISTE rekursiv implementieren, so muss das einzufügende Elemente bis zum Ende der Liste weitergereicht werden. man auch in der Klasse KNOTEN eine entsprechende Methode hintenEinfügen(…) implementieren (analog zur Methode gibLänge()).

- Die ***Klasse LISTE*** kann 2 verschiedene Fälle vorfinden:
  - Ist die **LISTE leer**, so wird das Element als Anfangs-KNOTEN eingefügt.
  - Ist die **LISTE nicht mehr leer**, so gibt sie die Anfrage an ihren ersten KNOTEN weiter und ruft auf diesem die Methode hintenEinfügen(…) auf.

- In der ***Klasse KNOTEN*** gibt es 2 verschiedene Fälle:
  - Ist der **Nachfolger eines Knotens** `null`, so ist er der letzte KNOTEN in der LISTE und fügt als seinen Nachfolger einen neuen KNOTEN ein.
  - Ist der **Nachfolger eines Knotens nicht** `null`, so reicht er das Anliegen einfach an den Nachfolge-KNOTEN weiter.

Aufgabe 5:

a) Implementiere die Methode **`hintenEinfügen()`**.

b) Zeichne ein Objektdiagramm zu einer LISTE, in welcher zuerst der KUNDE „Marc“ und dann der KUNDE „Julia“ eingefügt wurde.

c) Zeichne ein Sequenzdiagramm - basierend auf der LISTE aus der Teilaufgabe b) - zum Methodenaufruf hintenEinfügen(new KUNDE(“Karl“))

Hilfsmethode zum Testen der Liste:

Implementiere die Methode **`public static LISTE init()`**, welche eine vorbereitete Liste zurückgeben soll.

Erzeuge in der Methode eine (lokale) Liste und füge mithilfe der vorherigen Methode Elemente in die Liste ein. Gib anschließend die Liste zurück.

Aufgabe 6: **Entfernen an beliebiger Stelle: `entfernenAn(int i) : T`**

Diese Methode soll ein bestimmtes Element aus der LISTE entfernen. Der Parameter i gibt dabei dessen Position an. Bei unserer Variante soll die 0 das erste Element der Liste (sprich der Anfang sein). Falls i negativ oder größer als die Länge der Liste ist, soll null zurückgegeben werden.

a) Überlege dir, wie das rekursive Entfernen an beliebiger Stelle algorithmisch funktionieren kann. Notiere deinen Algorithmus.

b) Implementiere die Methode **`entfernenAn(int i): T`**, welche den Knoten an der Postion i entfernt und den Inhalt zurückgibt. Falls dies nicht möglich ist, soll null zurückgegeben werden.

Aufgabe 7: **Einfügen an beliebiger Stelle:**
`einfügenAn(T inhalt, int i):boolean`

Diese Methode soll ein Element an einer bestimmten Position in der Liste einfügen. Der Parameter i gibt dabei dessen Position an. Bei unserer Variante soll die 0 „vor“ dem Anfangselement entsprechen, die 1 nach dem Anfangselement usw. Falls i negativ oder größer als die Länge der Liste ist, soll false zurückgegeben werden; ansonsten true.

a) Überlege dir, wie das rekursive Einfügen an beliebiger Stelle algorithmisch funktionieren kann. Notiere deinen Algorithmus.

b) Implementiere die Methode **`einfügenAn(T inhalt, int i) : boolean`**, welche einen neuen Knoten mit dem übergebenen Inhalt an der Position i einfügt. Falls dies nicht möglich ist, gib false zurück; ansonsten true.

Aufgabe 8: **Suchen in der LISTE**

a) Implementiere die Methode **`suchenUndIndexZurückgeben(String info): int`**, welche überprüft, ob die **info()** eines Inhaltsobjekt mit dem übergebenen Info-String übereinstimmt ist und gibt dessen Index zurückgibt. Der Anfang besitzt den Index 0; der Nachfolger von Anfang besitzt den Index 1, usw.. Falls das Element nicht vorhanden ist, soll -1 zurückgegeben werden.
Tipp: Die Signatur dieser Methode in der Klasse KNOTEN sollte einen weiteren Übergabeparameter enthalten, um den Index zählen zu können. → ähnlich wie bei enfernenAn(…) und einfügenAn(…)

String-Vergleich mit compareTo(…) → Link

string1.compareTo(string2) < 0	***string1 ist alphabetisch vor string2***
string1.compareTo(string2) == 0	***string1 ist identisch zu string2***
string1.compareTo(string2) > 0	***string1 ist alphabetisch nach string2***