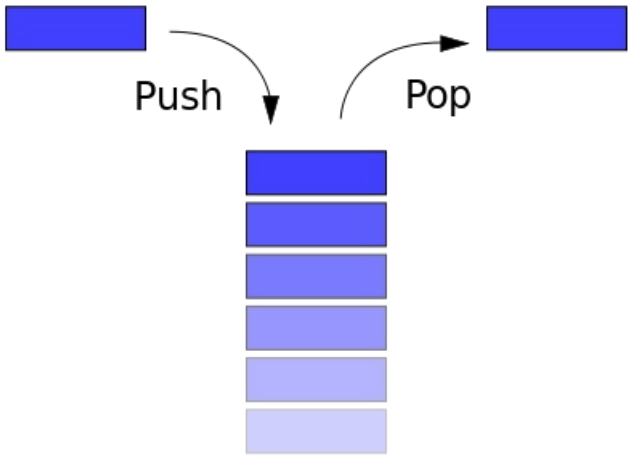
Datenstruktur S T A C K

Man nennt den ***Stack*** auch ***Keller*** oder ***Stapel***. Hierbei handelt es sich um eine Datenstruktur, die tatsächlich einem Stapel Papier oder einem Kartenstapel gleicht. Man legt jedes neue Element oben drauf. Möchte man nun wieder ein Element entfernen, so muss man das oberste nehmen. Es kommt also das zuletzt hinein gelegte Element als erstes wieder heraus. Das Verfahren dieser Datenstruktur nennt man auch ***LIFO*** (**L**ast **I**n **F**irst **O**ut). Ein Stack kann z.B. dazu verwendet werden, um die Reihenfolge von Elementen umzukehren.



Realisierung durch ein Array

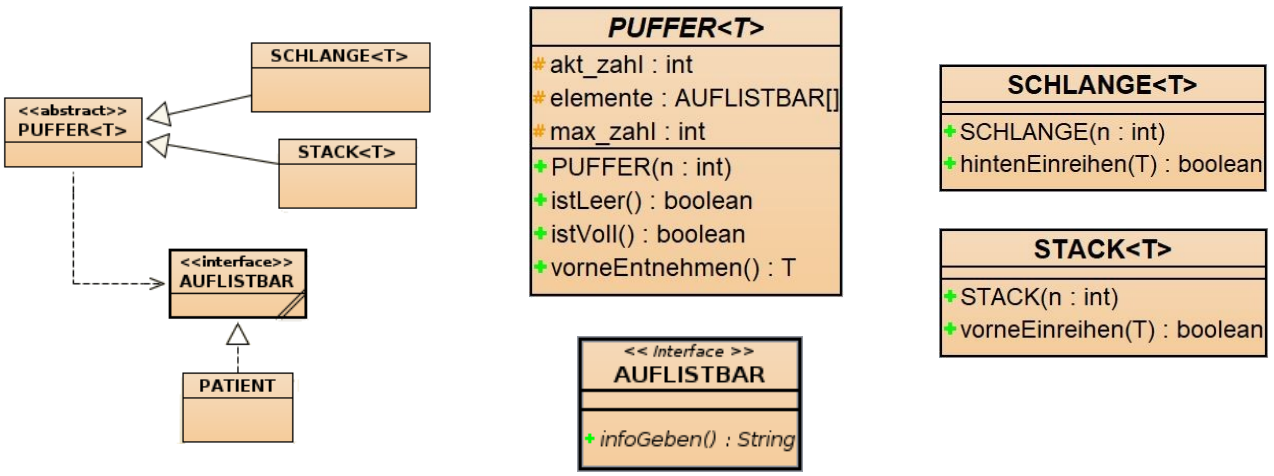
Betrachten wir den Stack zunächst wieder als durch ein Array realisiert, so stellen wir fest, dass der Stack nichts anderes ist als unsere SCHLANGE, nur dass man die Elemente nicht am anderen sondern am selben Ende (vorne) wieder herausnimmt.

Deshalb könnten wir den Stack realisieren, indem wir unsere bisherige Datenstruktur SCHLANGE schlichtweg um eine Methode **vorneEinreihen()** erweitern.

Diese Realisierung widerspricht aber dem Prinzip: **Jede Klasse hat genau eine Aufgabe.**

Aufgabe 1:

Daher erstelle ein neues BlueJ-Projekt „StackArray“ und implementiere – zur Übung – alles von Grund auf neu. Befolge hierbei Struktur wie im folgenden Klassendiagramm. Füge beide TAXIs hinten in den STACK ein und entnehme sie anschließend wieder. Kommen sie in umgekehrter Reihenfolge wieder heraus?



Realisierung als rekursive dynamische Datenstruktur

Bisher ist jede SCHLANGE oder jeder STACK bzgl. seiner Aufnahmefähigkeit begrenzt. Das wollen wir nun 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***.

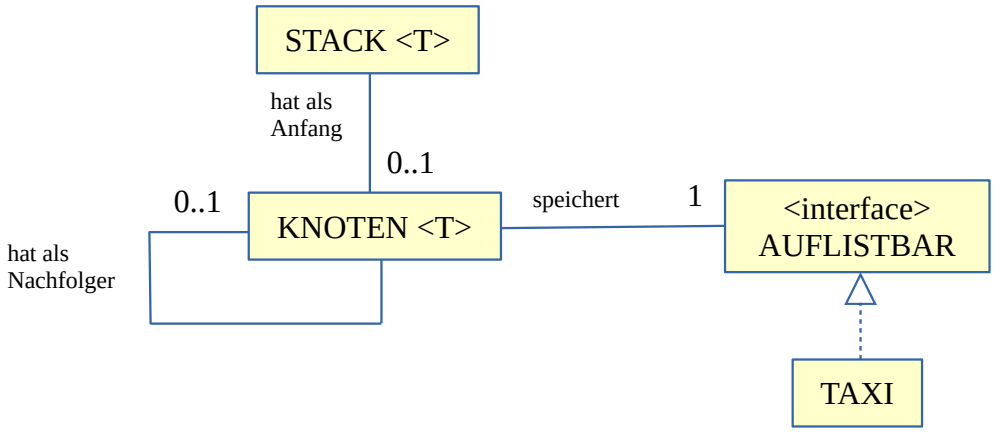
Hierzu müssen wir aber von den Arrays absehen und uns selbst eine entsprechende Datenstruktur zurecht legen. Besondere Beachtung bekommt dabei wieder das

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:



S T A C K <T>
. . .
+ STACK() *+ istLeer() : boolean* *+ push(t: T) : void* *+ pop() : T* *+ laenge() : int*

K N O T E N <T>
. . .
+ KNOTEN(t: T) *+ nachfolger(): KNOTEN* *+ setzeNachfolger(k: KNOTEN): void* *+ inhalt(): T* *+ laenge(): int*

Zunächst implementieren wir nur die Klasse STACK.

(Später auf dieselbe Weise und in Vererbungshierarchie auch die Klasse SCHLANGE)

Nutze hierfür ein neues BlueJ-Projekt „SchlangeStackDynamisch“

Die Methode ***istLeer()***

Ein STACK ist genau dann **leer**, wenn sein Anfangs-KNOTEN (root) `null` ist.

Die Methode ***push(...)***

Prinzipiell gibt es **zwei verschiedene Ausgangssituationen**:

1. Der ***STACK ist noch leer*** (der Anfangs-KNOTEN (root) 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 STACK 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 STACKs.*

b) *Zeichne ein Objektdiagramm dieses STACKs, wenn ein Element eingefügt wurde.*

c) *Zeichne ein Objektdiagramm dieses STACKs, wenn ein weiteres Element eingefügt wurde.*

d) *Programmiere die Methode push(...).*

*Die Methode **pop()** für den STACK*

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

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

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

3. ***Der STACK hat mehrere KNOTEN.***
   Dann muss sich der STACK 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) *Betrachte die Objektdiagramme von Aufgabe 2a-c) in umgekehrter Reihenfolge.*

b) *Programmiere die Methode pop().*

Die Methode ***laenge()***

Diese Methode **erscheint zunächst unmöglich** ohne vorher **eine Zählvariable** im STACK 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**:

- ***Ein STACK*** gibt die Länge 0 zurück, wenn sein Anfangs-KNOTEN `null` ist.
  Wenn der STACK nicht leer ist, dann fragt er einfach seinen Anfangs-KNOTEN und gibt dessen Antwort zurück. (s. nächster Punkt)

- ***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) *Schau dir die Präsentation zu der Methode* `laenge()` *an.*

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

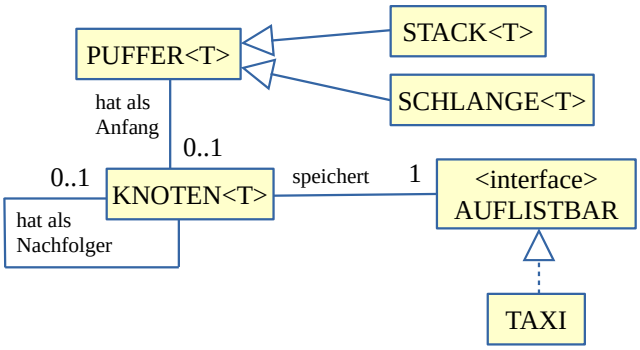
c) *Zeichne ein Sequenzdiagramm zum Methodenaufruf* `laenge()` *eines STACK-Objekts, wie es im Objektdiagramm von Aufgabe 2c) dargestellt ist.*

d) Programmiere die Methode `laenge()`.

Aufgabe 5:

a) *Lagere alles an Code, was auch für eine SCHLANGE benötigt wird, in eine (abstrakte) Superklasse PUFFER aus. Die Klasse STACK erbt dann von PUFFER und muss die abstrakte Methode push(...) des PUFFERs überschreiben.*

b) *Lege nun eine neue Klasse SCHLANGE an, die ebenfalls von PUFFER erbt und implementiere darin ebenfalls eine Methode push(...), welche das einzureihende Element am Ende einfügt.*
Das Prinzip hierfür wird auf der folgend beschrieben …



Datenstruktur STACK und SCHLANGE

*Die Methode **push()** für die SCHLANGE*

Variante 1:

Möchte man die Methode push(…) der Klasse SCHLANGE rekursiv implementieren, so müsste man auch in der Klasse KNOTEN eine entsprechende Methode push(…) implementieren.

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

- In der ***Klasse KNOTEN*** gibt es 2 verschiedene Fälle:
  - Ist der **Nachfolger eines Knotens `null`**, so ist er der letzte KNOTEN in der SCHLANGE 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 6:

Öffne die Version von STACK und SCHLANGE, in der zu den PUFFER durch Vererbungshierarchie „herausgezogen“ hast in BlueJ und implementiere nun in den Klassen SCHLANGE und KNOTEN jeweils die rekursive Variante der Methode push().

Variante 2:

In gewisser Hinsicht mag es als unschön empfunden werden, wenn die Methode push() nachträglich in der Klasse KNOTEN eingeführt wird, weil damit der STACK etwas erben würde, was er nicht braucht.
Vielleicht ist auch die Klasse KNOTEN von einem anderen Programmierer angefertigt worden und liegt nicht im Quelltext vor, so dass sie nicht verändert werden kann.

In diesem Fall implementiert man die Methode pushIter() in der Klasse SCHLANGE nicht rekursiv sondern **iterativ**, ***d.h. mit Hilfe einer Schleife***. Die Klasse KNOTEN wird in diesem Fall nicht angefasst.

- Die ***Klasse SCHLANGE*** kann (wie bei Variante 1) 2 verschiedene Fälle vorfinden:
  - Ist die **SCHLANGE leer**, so wird das Element als root-KNOTEN eingefügt.
  - Hat die **SCHLANGE mehrere Elemente**, so muss sie sich erst (mit einer Hilfs-Referenz) bis zum letzten KNOTEN durchtasten. Als Nachfolger dieses (letzten) KNOTENs wir der neue KNOTEN eingefügt.

    Tipp:
    Die erste Hilfsreferenz verweist zunächst auf den ersten KNOTEN (root). Mithilfe einer Schleife in der fortwährend die Hilfsreferenz auf den entsprechenden Nachfolger-KNOTEN gesetzt wird kann man sich durch die Schlange hangeln.

Aufgabe 7:

a) *Betrachte nochmal die Objektdiagramme von Aufgabe 2a-c) und 6) und versuche das oben beschriebene Verfahren für jedes der Diagramme durchzuspielen.*

b) *Öffne wieder das BlueJ-Projekt und programmiere die Methode pushIterFor() und pushIterWhile(). (Tipp: die Schlange kennt ihre Länge)*

Aufgabe 8 – Effizientes Programmieren:

Wie oft muss die Schlange bei den Varianten push(), pushIterFor() und pushIterWhile() durchlaufen werden?

Betrachte die Objekt-Diagramme von 2a-c), (zeichne bei Bedarf noch mehr KNOTEN ein) und finde eine allgemeine Regel/Zusammenhang, die von laenge abhängt.

Aufgabe 9:

a) *Zeichne ein Sequenzdiagramm zu einer SCHLANGE der Länge 2 und füge einen dritten Knoten mit der rekursiven Methode push().*

b) *Zeichne ein Sequenzdiagramm zu einer SCHLANGE der Länge 2 und füge einen dritten Knoten mit der iterativen Methode pushIter().*

Aufgabe 10:

Rekursion kann nicht nur beim Verwalten von Datenstrukturen hilfreich sein. Auch viele andere Probleme lassen sich auch mittels Rekursion lösen.

Implementiere alle Methoden rekursiv im Projekt RekursionsÜbungen_Vorlage.