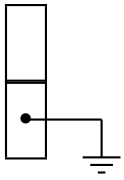
Dynamische Datenstrukturen

Einfach verkettete Listen

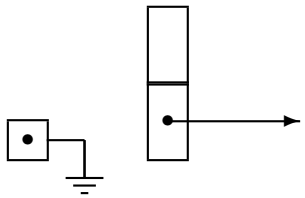
Die Vorgehensweise Pointer (Zeiger) in einer Implementierung zu verwenden, beruht auf der Tatsache, dass statische Datenstrukturen, wie zum Beispiel Felder Speicherplatz `vor` der Laufzeit des Programms zugewiesen bekommen.
Bei einer Datenbank mit wenigen Einträgen würde unter Umständen Speicher (alloziert) angesprochen werden, der unter Umständen gar nicht benötigt wird. Um Ressourcen eines Rechners zu schonen, ist das Konzept dynamischer Datenstrukturen entstanden. Dieses Konzept verwenden Sie bei der Arbeit mit Python fortlaufend.
Es werden Speicherzellen nur dann belegt, wenn Sie auch wirklich gebraucht werden, zur Laufzeit des Programms.
Pointer (Zeiger) dienen dabei als Zeiger auf vorhandene Speicherinhalte. Durch Sie wird das jeweils benötigte Datum referenziert. Da durch diese Logik von vorherein nicht klar ist, wieviele Daten zur Laufzeit tatsächlich benötigt werden, bietet sich ein rekursiver Aufbau an.
Diese beschriebene Prinzip lässt sich mit Hilfe von Klassen in Python veranschaulichen.
Es wird eine Klasse (Knoten) geschaffen, mit der es möglich ist, Elemente für die spätere dynamische Liste durch Instanzen zu erzeugen.
Auf der anderen Seite erfolgt die Schaffung der Klasse Liste. In dieser Klasse wird ein noch ungebundener Zeiger geschaffen. Durch die Methode Insert ist es dann möglich diesen Zeiger auf einen Knoten zu `biegen`.



Nebenstehendes Bild zeigt eine Instanz der Klasse Knoten. Beim Aufruf der Klasse Knoten mit einem entsprechenden Wert wird self.value mit diesem Wert belegt. Die Referenz self.next bleibt dagegen am Anfang ohne Referenzobjekt, der Zeiger `baumelt`.

Listing 1: Startprogramm

```
class Knoten :

  def __init__(self, value):

        self.value = value
        self.next = None

  def get_value (self):

        return self.value

  def set_next (self, node):

        self.next = node
```



Es ist nun erforderlich in einer weiteren Klasse die eigentliche Liste zu intialisieren. Der Pointer self.liste wird auf den Zustand `NONE` gebogen. Noch ist kein Element vorhanden.
Soll durch die Methode Insert ein Element bzw. key eingefügt werden, ist zu unterscheiden. Ist die Liste noch leer, wird der Zeiger auf den Knoten gebogen, ist sie es nicht, muss man erst an das Ende der Liste gelangen. Ein rekursiver Weg bietet sich hier an, ein imperativer ist möglich.
Vergleichen Sie die Implementierung !

Listing 2: Startprogramm

```
class Liste :

    def __init__(self):

        self.liste = None

    def Insert (self, elem):  # Initialisierung

        if self.liste : self._insert (self.liste, elem)
        else          : self.liste = Knoten (elem)

    # rekursiver Listenaufruf

    def _insert (self, liste, key):
        if liste.next : self._insert (liste.next, key)
        else          : liste.next = Knoten (key)

    # imperativer Listenaufruf

    def _insert1 (self, liste, key):
        while liste.next : liste = liste.next
        else             : liste.next = Knoten (key)
```

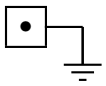
Eine Showfunktion, um die Ergebnisse anzuzeigen, muss ähnlich implementiert werden.

Listing 3: Startprogramm

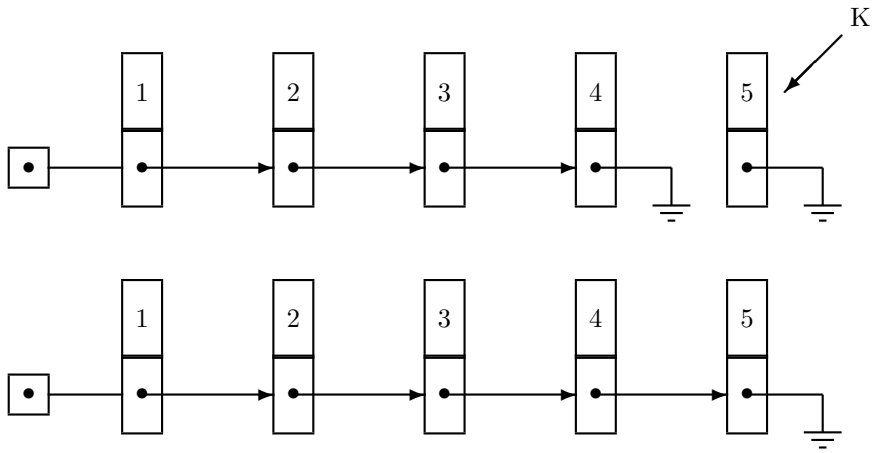
```
def Show (self):
        if self.liste : self._show (self.liste)
        else          : return 0

    def _show (self, liste):
        while liste :
                print liste.get_value () ,
                liste = liste.next
```

Nun ist es möglich eine dynamische Liste zu erzeugen und diese mit show anzuzeigen.

1. Schritt



2. Schritt



Aufgabe 2

Implementieren Sie `del_fst_elem`, `delate` und `search` !