Einführung in die Programmiertechnik

Klassen und Abstrakte Datentypen

Abstrakte Datentypen (ADT)

- Beschreibung der Datentypen nicht auf Basis ihrer Repräsentation, sondern auf Basis ihrer Operationen und der Zusammenhänge zwischen diesen Operationen
- Algebraische Beschreibung: Typ wird durch Mengen, Konstruktoren, Terme und Axiome beschrieben
  - z.B. ACT ONE
  - Natürliche Zahlen:
    - Operatoren: 0, succ(x), plus(x,y), minus(x,y)
      - Zahlen sind 0, succ(0), succ(succ(0)), ...
      - Terme sind z.B. plus(0, succ(minus(succ(0), 0)))
    - Axiome: Definition von äquivalenten Termen
      - succ(plus(x, y)) ≡ plus(x, succ(y))
      - plus(succ(x), y) ≡ plus(x, succ(y))
      - plus(0, y) ≡ y

Stack

- Stapelspeicher (LIFO)
  - Werte werden auf den Stapel gelegt und können wieder von ihm heruntergenommen werden
- Menge: Elementtype E, Stack S_E
- Operatoren
  - emptyStack: → S_E
  - push: $S_E \times E \rightarrow S_E$
  - pop: $S_E \rightarrow S_E$
  - top: $S_E \rightarrow E$
  - empty: $S_E \rightarrow$ boolean
- Axiome: $\forall e \in E \; \forall s \in$
  - top(push(s, e)) ≡ e
  - pop(push(s, e)) ≡ s

Queue

- Warteschlange (FIFO): Werte werden am Ende an die Warteschlange angefügt; vorderste Elemente können entfernt werden
- Menge: Elemente E, Queue Q_E
- Operationen:
  - emptyQueue: → Q_E
  - enqueue: $Q_E \times E \rightarrow Q_E$
  - dequeue: $Q_E \rightarrow Q_E$
  - head: $Q_E \rightarrow E$
  - empty: $Q\rangle_E \rightarrow$ boolean

Realisierung abstrakter Datentypen durch Klassen

- Klassen: Kapselung der Implementierungsdetails
- Operationen des ADT werden i.d.R. Methoden der Klasse
  - Ausnahme u.a. initiale Erzeugung des ADT

Stack in Python

- Ziel: class Stack
  - Methoden: .push(e), .empty(), .pop(), .top()
- Implementierungsstrategie 1: Repräsentation des Stapels als Liste von Elementen
  - Hilfsklasse Entry für Einträge im Stack

```
class Item:
    def __init__(self, value, next):
        self.value = value
        self.next = next


class Stack:
    def __init__(self):
        self.items = None
    def push(self, e):
        self.items = Item(e, self.items)
    def pop(self):
        self.items = self.items.next
    def top(self):
        return self.items.value
    def empty(self):
        return self.items is None
```

Implementierungsstrategie 2

- Tupel statt Item

```
class Stack:
    def __init__(self):
        self.items = None
    def push(self, e):
        self.items = (e, self.items)
    def pop(self):
        self.items = self.items[1]
    def top(self):
        return self.items[0]
    def empty(self):
        return self.items is None
```

Implementierungsstrategie 3

- Python-Listen anstelle verketteter Listen

```
class Stack:
    def __init__(self):
        self.items = []
    def push(self, e):
        self.items.append(e)
    def pop(self):
        del self.items[-1]
    def top(self):
        return self.items[-1]
    def empty(self):
        return len(self.items)==0
```

Einfach verkettete Listen

- Realisierung des ADT “Stack”:
  - Einfügen am Anfang, Löschen am Anfang
- Realisierung des ADT “Warteschlange”
  - Einfügen am Ende, Löschen am Anfang
  - Tupel sind immutable, deshalb nicht zum Einfügen am Ende geeignet

```
class Item:
    def __init__(self, value, next):
        self.value = value
        self.next = next
```

Queue auf Basis einfach verketteter Listen

- Kopf-Objekt zur Realisierung der Queue-Schnittstelle
- Einfügen (enqueue): 2 Fälle
  - Queue leer: Anlegen eines ersten Eintrags
  - Queue nicht leer: Auffinden des Endes der Liste, dann Eintragen eines neuen Elements am Ende
- Löschen (dequeue): Vorderstes Element ersetzen durch 2. Element
- Zugriff auf vordersten Wert (head): Wert aus vorderstem Element auslesen

```
class Queue:
    def __init__(self):
        self.items = None
    def enqueue(self, e):
        if self.items is None:
            self.items = Item(e, None)
        else:
            item = self.items
            while item.next is not None: item = item.next
            item.next = Item(e, None)
    def dequeue(self):
        self.items = self.items.next
    def head(self):
        return self.items.value
    def empty(self):
        return self.items is None
```