

Einführung in die Programmieretechnik

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 , $\text{succ}(x)$, $\text{plus}(x,y)$, $\text{minus}(x,y)$
 - Zahlen sind 0 , $\text{succ}(0)$, $\text{succ}(\text{succ}(0))$, ...
 - Terme sind z.B. $\text{plus}(0, \text{succ}(\text{minus}(\text{succ}(0), 0)))$
 - Axiome: Definition von äquivalenten Termen
 - $\text{succ}(\text{plus}(x, y)) \equiv \text{plus}(x, \text{succ}(y))$
 - $\text{plus}(\text{succ}(x), y) \equiv \text{plus}(x, \text{succ}(y))$
 - $\text{plus}(0, y) \equiv y$

Stack

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

Queue

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

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):
```

```
        del self.items = self.items.next
```

```
    def head(self):
```

```
        return self.items.value
```

```
    def empty(self):
```

```
        return self.items is None
```