Vorlesung 27

19.09.2024

Traits

- Alternative Modellierung
  - Studis und DozentInnen sind Personen
  - Studies sind Personen die Lernen
  - DozentInnen sind Personen die lernen und arbeiten
  - Nicht alle Studis arbeiten
- 1. Idee: Unterklassen workingPerson und studyingPerson und DozentIn erbt von beidem
- Es gibt aber keine Mehrfachvererbung in Scala und java
- Lösung: traits
  - ein bisschen wie abstrakte Klassen, aber andere Sichtweise
  - Traits können als Eigenschaft gesehen werden, die der Klasse hinzugefügt werden
  - kennen schon den Trait Ordering
  - Traits können einige Methoden schon implementieren
  - Wenn von mehreren Traits geerbt wird, sollte es keine bereit implementierten Methoden mit der gleichen Signatur geben
- In Scala 3 benutzt man eigentlich keine abstrakten Klassen mehr, sondern nur noch Traits
- Klassen können Typparameter haben

```
//Typparameter
class Box[A]:
    private var elements: List[A] = Nil

    def add(elem: A): Unit =
        elements = elem :: elements
```

```
    def get():A=
        val rand = new scala.util.Random
        val i:Int = rand.nextInt(elements.size)
        val elem = elements(i)
        elements=elements.take(i) ::: elements.drop(i+1)
        return elem
```

Abstrakte Datentypen: ADT

💡 Ein abstrakter Datentyp ist die Beschreibung einer Menge von Objekten mit: darauf definierten Operationen und nach außen sichtbaren Eigenschaften. Dabei ist die Beschreibung des ADT unabhängig von der Implementierung.

💡 Eine Datenstruktur ist eine konkrete Umsetzung eines ADT’s

→ Den Rest der Vorlesungen beschäftigen wir uns mit 4 unterschiedlichen ADT’s

Vorteile:

- Wir können den DT nutzen über eine Schnittstelle nutzen ohne zu wissen, was genau die Implementierung macht.
    - Das heißt, die Implementierung kann ausgetauscht werden, ohne dass der Code der die Schnittstelle nutzt betroffen ist.

Stack (Stapel)

Last- in-first-out (**LIFO**) Speicher, bei dem Elemente eingefügt (*gepusht*) werden können und das zuletzt eingefügte Element entfernt und zurückgegeben (*pop*) werden kann. Kann man sich vorstellen wie eine Liste in Scala.

Beispiel:

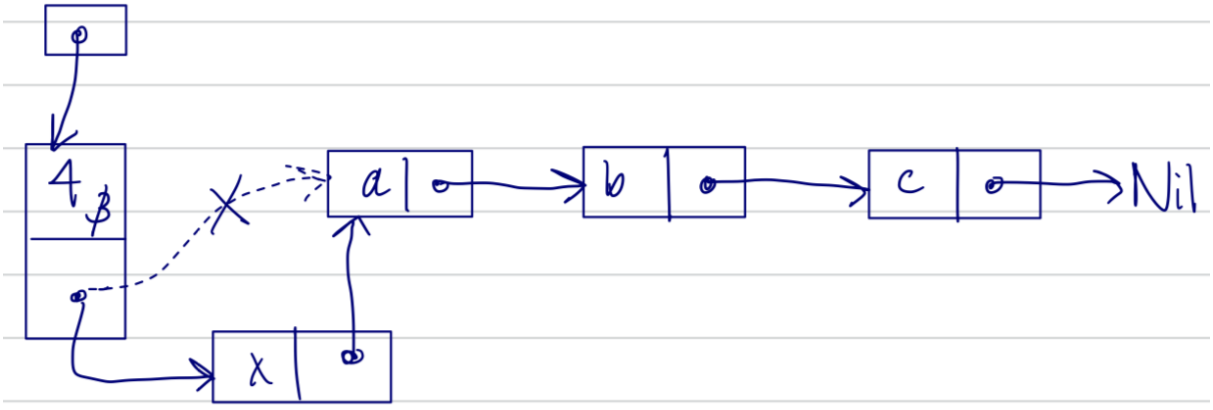
```
push(Teller)
push(Gabel)
push(Huhn)
pop() -> Huhn
push(Möhre)

|Möhre  |
|Gabel  |
|Teller |
|_______|
```

Verwendung z.B. intern für Rekursion, zurück-Knopf im Browser

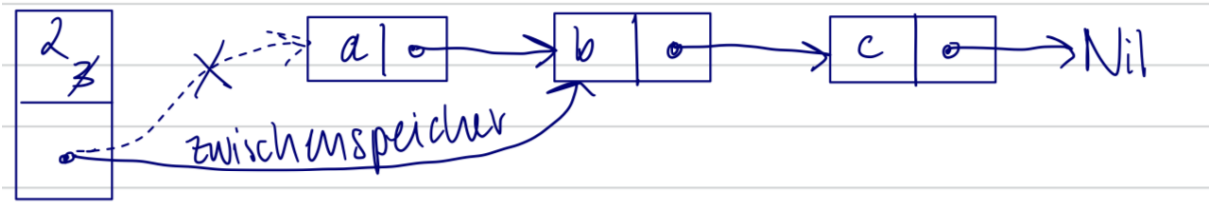
Implementierung mit Verkettete Knoten

- Der Verwaltungskopf speichert die Anzahl der Elemente und den Pointer zum ersten Objekt (der topNode)
- Push(x)



Bei push() wird die Anzahl der Elemente im Verwaltungskopf geupdated (+1) und der Pointer ebenfalls, und zeigt nun zu den neuen gepushten Element. → Laufzeit O(1)

- erstelle neuen Knoten n für x
- Bei pop() wird die Anzahl der Elemente reduziert (-1) und der Pointer muss zum zuvor zweiten Element geupdated werden → Laufzeit O(1)



Implementierung mit einem Array statischer Größe

Unterschied: Gleichberechtigter Zugriff auf alle Elemente des Arrays.

Wir haben wieder einen Verwaltungskopf mit Informationen über Size und Pointer zum Array. Elemente werden nach hinten eingefügt bzw. von hinten entfernt. Bottom Element ist an Index 0 und oberstes Element ist an Stelle `Array(size-1)`

Problem: bei dieser Implementierung gibt es eine von Array Größe beschränkte Anzahl von Elementen. Wenn full, dann full.

Alles sind wieder Elementaroperationen, deswegen ist die Laufzeit von push und pop auch gleich O(1)

```
class ArrayStack[A: ClassTag](private val capacity:Int) exten
    private var array: Array[A] = new Array[A](if capacity <
    private var _size : Int = 0

    def isEmpty: Boolean = _size == 0
    def size: Int = _size

    def pop(): A =
        if isEmpty then throw new Exception("Empty Stack")
        val result = array(_size-1)
        array(_size-1) = null.asInstanceOf[A]
        _size -= _size
        result

    def push(x: A): Unit =
        if _size == array.length then throw new Exception("St
        array(_size) = x
        _size += 1
```

Implementierung mit Dynamischen Arrays

Arrays mit dynamischer Größe passen die Größe des Arrays so an, dass die folgende Invariante gilt:

`I = array.length/4 ≤ size < array.length`

Wenn die Invariante bei push oder pop verletzt wird werden die Elemente kopiert und ein neues Array erstellt, welches entsprechend doppelt(push) oder halb(pop) so groß ist. Das passiert in der privaten Methode `resize()`

```
class DynArrayStack[A:ClassTag] extends MyStack[A]:
    private var array : Array[A] = new Array[A](1)
    private var _size = 0

    def isEmpty : Boolean = _size == 0
    def size:Int = _size
    def top:A = array(_size-1)

    private def resize():Unit =
        if _size < array.length/4 || _size >= array.length th
            val newCapacity:Int = if _size < array.length/4
                                  then array.length/2 else
                                  2*array.length
            var newA : Array[A] = new Array[A](newCapacity)
            for i <- 0 to _size -1 do
                newA(i) = array(i)
            array = newA


    def pop(): A =
        if isEmpty then throw new Exception("Empty Stack")
        val result:A = array(_size-1)
        array(_size-1) = null.asInstanceOf[A]
        _size -= 1
        resize()
        result


    def push(x:A):Unit =
        array(_size) = x
```

```
        _size += 1
        resize()

    override def toString(): String =
        var s : String  = "bottom<"
        var i : Int = 0
        while i<= _size-1  do
            s += " |" + array(i)
            i=i+1
        s+="<top"
        s
```

Analyse und Vergleich: Welche der Implementierungen ist besser ?

	Verkette Liste	Array statischer Größe	Array dynamischer Größe
Laufzeit	Alle Funktionen haben eine Laufzeit von O(1)	Alle Funktionen haben eine Laufzeit von O(1)	`isEmpty` und `size` haben Laufzeiten von O(1). Bei `resize` haben eine Laufzeit von O(n), sonst aber ebenfalls (amortisiert) O(1)
Cache	verteilt im Speicher	eng zusammen im Speicher	eng zusammen
Speicherplatz	Nur Elemente und Verwaltungskopf	Elemente + reservierter Speicher für neue → Speicher kann deutlich größer sein als Anzahl der Elemente	Maximal 4x so viel wie die Anzahl der Elemente
Einschränkungen	keine	begrenzter Speicherkapazitäten	keine

→ Queue in Vorlesungsunterlagen vom 20.09./Vorlesung 28