Felder (Arrays)

$A[i] = a_i$ falls $A = \langle a_0, \ldots, a_{n-1} \rangle$

Beschränkte Felder (Bounded Arrays)

Eingebaute Datenstruktur: Ein Stück Hauptspeicher + Adressrechnung
Größe muss von Anfang an bekannt sein

Unbeschränkte Felder (Unbounded Arrays)

$$\langle e_0, \ldots, e_n \rangle.\text{pushBack}(e) \rightsquigarrow \langle e_0, \ldots, e_n, e \rangle,$$
$$\langle e_0, \ldots, e_n \rangle.\text{popBack} \rightsquigarrow \langle e_0, \ldots, e_{n-1} \rangle,$$
$$\text{size}(\langle e_0, \ldots, e_{n-1} \rangle) = n \ .$$

Unbeschränkte Felder – Anwendungen

wenn man nicht weiß, wie lang das Feld wird.
Beispiele:

- Datei zeilenweise einlesen
- später: Stacks, Queues, Prioritätslisten, ...

Unbeschränkte Felder – Grundidee

wie beschränkte Felder: Ein Stück Hauptspeicher

pushBack: Element anhängen, size$++$
Kein Platz?: umkopieren und (größer) neu anlegen

popBack: size$--$
Zuviel Platz?: umkopieren und (kleiner) neu anlegen

Immer passender Platzverbrauch?

n pushBack Operationen brauchen Zeit

$O(\sum_{i=1}^{n} i) = O(n^2)$

Geht es schneller?

Unbeschränkte Felder
mit teilweise ungenutztem Speicher

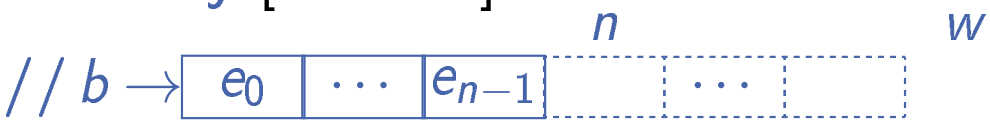
Class UArray **of** Element

w=1 : $\mathbb{N}$ // allocated size

n=0 : $\mathbb{N}$ // current size

invariant $n \leq w < \alpha n$ or $n = 0$ and $w \leq 2$ // e.g., $\alpha = 4$

b : **Array** $[0..w-1]$ **of** Element



Operator $[i : \mathbb{N}]$: Element

  assert $0 \leq i < n$

  return $b[i]$

Function size : $\mathbb{N}$ **return** n

```
Procedure pushBack(e : Element)                    // Example for n = w = 4:
    if n = w then                                                  // b →|0|1|2|3|
        reallocate(2n)                                 // b →|0|1|2|3|  |  |  |  |
    b[n]:= e                                           // b →|0|1|2|3|e|  |  |  |
    n++                                                // b →|0|1|2|3|e|  |  |  |
```

```
Procedure reallocate(w′ : ℕ)                        // Example for w = 4, w′ = 8:
    w:= w′                                                         // b →|0|1|2|3|
    b′:= allocate
        Array [0..w′ − 1] of Element                   // b′ →|  |  |  |  |  |  |  |  |
    (b′[0], . . . , b′[n − 1]):=
        (b[0], . . . , b[n − 1])                       // b′ →|0|1|2|3|  |  |  |  |
    dispose b                                                      // b →|0|1|2|3|
    b:= b′                  // pointer assignment b →|0|1|2|3|  |  |  |  |
```

Kürzen

```
Procedure popBack                                  // Example for n = 5, w = 16:
    assert n > 0                    // b →[0|1|2|3|4]
    n−−                             // b →[0|1|2|3|4]
    if 4n ≤ w ∧ n > 0 then                                  // reduce waste of space
        reallocate(2n)                                  // b →[0|1|2|3]
```

Was geht schief, wenn man auf passende Größe kürzt?

Amortisierte Komplexität unbeschr. Felder

Sei u ein anfangs leeres, unbeschränktes Feld.
Jede Operationenfolge $\sigma = \langle \sigma_1, \ldots, \sigma_m \rangle$
von pushBack oder popBack Operationen auf u
wird in Zeit $O(m)$ ausgeführt.
Sprechweise:
pushBack und popBack haben amortisiert konstante Ausführungszeit —

$$O\left(\overbrace{c \cdot m}^{\text{Gesamtzeit}} \Big/ \underbrace{m}_{\#\text{Ops}} \right) = O(1) \quad .$$

Beweis: Konto-Methode (oder Versicherung)

Operation	Kosten		Typ
pushBack	$\circ\circ$	(2 Token)	einzahlen
popBack	$\circ$	(1 Token)	einzahlen
reallocate($2n$)	$n \times \circ$	(n Token)	abheben

Zu zeigen: keine Überziehungen

Erster Aufruf von reallocate: kein Problem

($n = 2$, $\geq$ 2tes pushBack)

Beweis: Konto-Methode (oder Versicherung)

Operation	Kosten		Typ
pushBack	$\circ\circ$	(2 Token)	einzahlen
popBack	$\circ$	(1 Token)	einzahlen
reallocate($2n$)	$n \times \circ$	(n Token)	abheben

Weitere Aufrufe von reallocate:

rauf: $\text{reallocate}(2n) \underbrace{\overset{\geq n \times \text{pushBack}}{\rightsquigarrow}}_{\geq n \times \circ\circ} \text{reallocate}(4n)$

runter: $\text{reallocate}(2n) \underbrace{\overset{\geq n/2 \times \text{popBack}}{\rightsquigarrow}}_{\geq n/2 \times \circ} \text{reallocate}(n)$

Amortisierte Analyse – allgemeiner

- $\mathscr{O}$: Menge von Operationen, z. B. $\{\text{pushBack}, \text{popBack}\}$
- s (für state): Zustand der Datenstruktur
- $A_{Op}(s)$: amortisierte Kosten von Operation $Op \in \mathscr{O}$ in Zustand s
- $T_{Op}(s)$: tatsächliche Kosten von Operation $Op \in \mathscr{O}$ in Zustand s
- Berechnung: $s_0 \xrightarrow{Op_1} s_1 \xrightarrow{Op_2} s_2 \xrightarrow{Op_3} \cdots \xrightarrow{Op_n} s_n$

Die angenommenen amortisierten Kosten sind korrekt, wenn

$$\underbrace{\sum_{1 \le i \le n} T_{Op_i}(s_{i-1})}_{\text{tatsächliche Gesamtkosten}} \le c + \underbrace{\sum_{1 \le i \le n} A_{Op_i}(s_{i-1})}_{\text{amortisierte Gesamtkosten}}$$

für eine Konstante c

Amortisierte Analyse – Diskussion

- Amortisierte Laufzeiten sind leichter zu garantieren als tatsächliche.
- Der Gesamtlaufzeit tut das keinen Abbruch.
- Deamortisierung oft möglich, aber kompliziert und teuer
  - Wie geht das mit unbeschränkten Feldern?
  - Anwendung: Echtzeitsysteme
  - Anwendung: Parallelverarbeitung