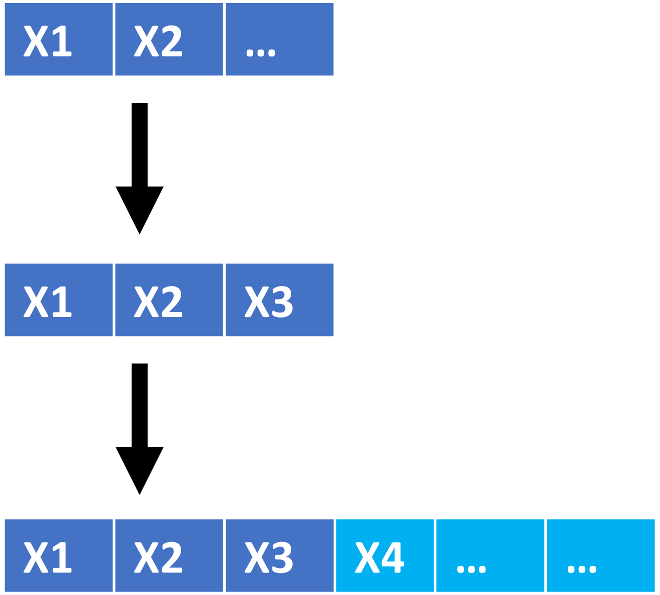
Tutorium Grundlagen: Algorithmen und Datenstrukturen

Übungsblatt Woche 5

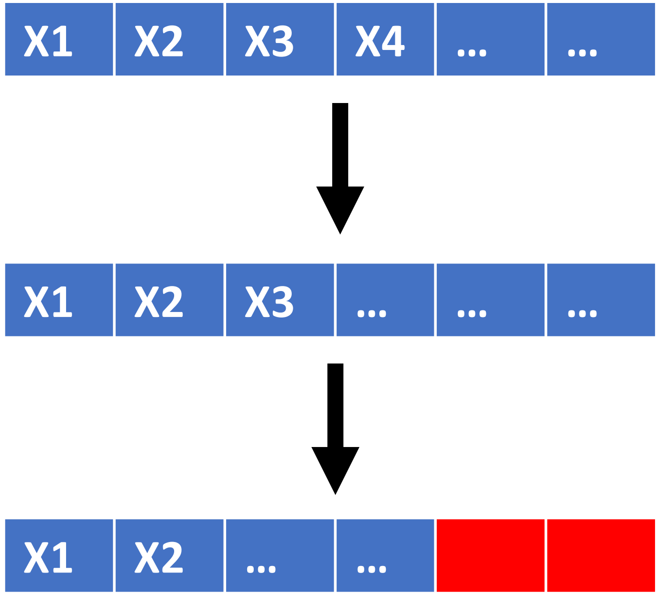
Aufgabe 5.1 – Dynamisches Array

Aus der Übung bekanntes dynamisches Array mit **$\alpha > \beta > 1$** und **α, β** natürliche Zahlen. n ist aktuelle Anzahl.
Reallocate bei pushBack auf **vollem Array**: Vergrößern zu **β*w = β*n**
Reallocate bei popBack mit **α*n kleiner gleich w**: Verkleinern zu **β*n**

Push-Back (α=3)



Pop-Back (β=2)



Aufgabe 5.1 – Dynamisches Array

Es ist nicht schwierig zu sehen, dass die tatsächliche Laufzeit von `pushBack` und `popBack` konstant ist (d.h. durch eine Konstante nach oben beschränkt), und die Laufzeit von `re-allocate` durch $\mathcal{O}(1) + c \cdot n$ beschränkt ist, wobei c eine Konstante ist und n die Zahl der kopierten Elemente. Wie in der Vorlesung ist es legitim, $c = 1$ zu setzen, indem wir annehmen, dass wir die Laufzeit in einer Einheit messen, die gerade der Laufzeit einer einzelnen Kopieroperation entspricht. Wir erhalten damit:

$$
\begin{aligned}
T(\texttt{pushBack}) &\in \mathcal{O}(1) \\
T(\texttt{popBack}) &\in \mathcal{O}(1) \\
T(\texttt{reallocate}) &= \mathcal{O}(1) + n, \text{ wobei } n = \text{Anzahl der kopierten Elemente}
\end{aligned}
$$

Ziel dieser Aufgabe ist der Nachweis mit einer amortisierten Analyse, dass die Laufzeit von Operationen der Länge m auf einem zu Beginn leeren dynamischen Array in $\mathcal{O}(m)$ liegt (zu Beginn hat das Array Größe 1).

Aufgabe 5.1 – Dynamisches Array

$$\begin{aligned}
\Delta(\texttt{pushBack}) &= \beta/(\beta-1)\\
\Delta(\texttt{popBack}) &= \beta/(\alpha-\beta)\\
\Delta(\texttt{reallocate}) &= -n,\ \text{wobei } n = \text{Anzahl der kopierten Elemente}
\end{aligned}$$

(a) Zeigen Sie, dass dieses Amortisationsschema zulässig ist, indem Sie zeigen, dass das Tokenkonto zu jedem Zeitpunkt nichtnegativ ist.

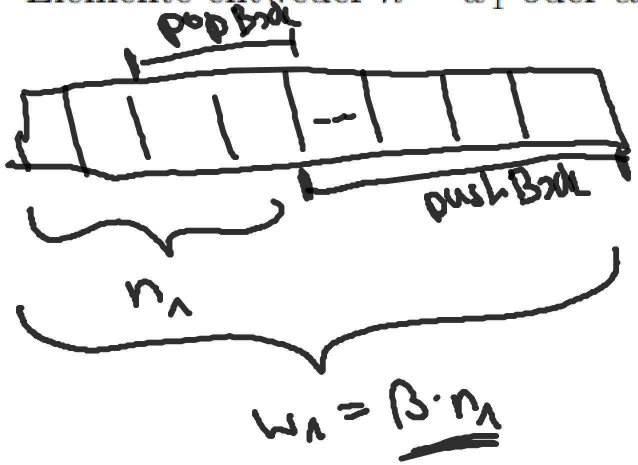
Hinweis: Bezeichne n_1 die Zahl der Elemente *unmittelbar* nach einem `reallocate` (im Falle von `pushBack` also noch vor der Einfügung des neuen Elements). Machen Sie sich klar, dass das Array *zu diesem Zeitpunkt* Größe $w_1 := \beta \cdot n_1$ hat, und $w_1 - n_1$ Positionen frei sind. Das nächste `reallocate` wird erst dann aufgerufen, wenn für die Zahl n der Elemente entweder $n = w_1$ oder $\alpha n \leq w_1$ gilt.

(b) Zeigen Sie, dass unter diesem Amortisationsschema die amortisierte Laufzeit jeder Operation in $\mathcal{O}(1)$ liegt, und folgern Sie, dass die Worst-Case-Laufzeit für Operationsfolgen der Länge m in $\mathcal{O}(m)$ liegt.

Aufgabe 5.1 – Dynamisches Array

(a) Zeigen Sie, dass dieses Amortisationsschema zulässig ist, indem Sie zeigen, dass das Tokenkonto zu jedem Zeitpunkt nichtnegativ ist.

Hinweis: Bezeichne n_1 die Zahl der Elemente *unmittelbar* nach einem `reallocate` (im Falle von `pushBack` also noch vor der Einfügung des neuen Elements). Machen Sie sich klar, dass das Array *zu diesem Zeitpunkt* Größe $w_1 := \beta \cdot n_1$ hat, und $w_1 - n_1$ Positionen frei sind. Das nächste `reallocate` wird erst dann aufgerufen, wenn für die Zahl n der Elemente entweder $n = w_1$ oder $\alpha n \leq w_1$ gilt.



pushBack: $\underbrace{(\beta - 1) n_1}_{\text{Anzahl pushBacks}} \cdot \frac{\beta}{\beta - 1} = \beta \cdot n_1 = w_1$ ⇒ ist Anzahl an abgezogenen Tokens

popBack: $(n_1 - n) \cdot \frac{\beta}{(\alpha - \beta)} = \left(\frac{w_1}{\beta} - \frac{w_1}{\alpha}\right) \cdot \frac{\beta}{(\alpha - \beta)} = \frac{\alpha w_1 - \beta w_1}{\alpha \beta} \cdot \frac{\beta}{(\alpha - \beta)}$

$= \frac{w_1}{\alpha} \geq \frac{\alpha n}{\alpha} = n$ ⇒ mehr Tokens als abgezogen werden

$w_1 - n_1$

Aufgabe 5.1 – Dynamisches Array

(a) Zeigen Sie, dass dieses Amortisationsschema zulässig ist, indem Sie zeigen, dass das Tokenkonto zu jedem Zeitpunkt nichtnegativ ist.

Hinweis: Bezeichne n_1 die Zahl der Elemente *unmittelbar* nach einem `reallocate` (im Falle von `pushBack` also noch vor der Einfügung des neuen Elements). Machen Sie sich klar, dass das Array *zu diesem Zeitpunkt* Größe $w_1 := \beta \cdot n_1$ hat, und $w_1 - n_1$ Positionen frei sind. Das nächste `reallocate` wird erst dann aufgerufen, wenn für die Zahl n der Elemente entweder $n = w_1$ oder $\alpha n \leq w_1$ gilt.

$n = 0 \qquad w = 1$

$\Rightarrow$ ein Push-Back : $\frac{\beta}{\beta - 1} \geq 1$

Aufgabe 5.1 – Dynamisches Array

(b) Zeigen Sie, dass unter diesem Amortisationsschema die amortisierte Laufzeit jeder Operation in $\mathcal{O}(1)$ liegt, und folgern Sie, dass die Worst-Case-Laufzeit für Operationsfolgen der Länge m in $\mathcal{O}(m)$ liegt.

$T(\text{pushback}) \in O(1) \Rightarrow A(\text{pushback}) = O(1) + \underbrace{\frac{\beta}{\beta-1}} \in O(1)$

$T(\text{popback}) \in O(1) \Rightarrow A(\text{popback}) = O(1) + \frac{\beta}{\alpha-\beta} \in O(1)$

$T(\text{reallocate}) \in O(1) + n \Rightarrow A(\text{reallocate}) = O(1) + n - n \in O(1)$

m Operationen: $m \cdot \underbrace{\max_{\sigma}(A(\sigma))}_{\text{Abschätzung nach oben}} = m \cdot O(1) \in O(m)$

Aufgabe 5.2 – Betrunkener Übungsleiter

Wir betrachten einen torkelnden Übungsleiter an einer Kletterwand, der die folgenden beiden Operationen durchführen kann:

- `hoch`, $\Rightarrow T(1)$ $A(hoch) = O(1)$
- `runter(int` k`)`. $\Rightarrow T(\min\{k, h\})$ $A(runter(k)) = O(1)$

Die Starthöhe des Übungsleiters beträgt 0 Meter. Durch die Operation `hoch` steigt der Übungsleiter von seiner aktuellen Position aus genau einen Meter höher. Diese Operation hat die Laufzeit 1. Durch die Operation `runter(int` k`)` fällt der Übungsleiter von seiner aktuellen Position aus exakt $\min\{h, k\}$ Meter nach unten, wobei h die aktuelle Höhe des Übungsleiters ist. (Das bedeutet, dass der Übungsleiter nie tiefer als seine Starthöhe sinken kann). Wir nehmen hierbei an, dass k stets eine natürliche Zahl (nicht-negativ) ist. Die Operation `runter(int` k`)` hat die Laufzeit $\min\{h, k\}$.

Zeigen Sie mithilfe der Bankkonto-Methode, dass die amortisierten Laufzeiten der Operationen `hoch` und `runter(int` k`)` in $\mathcal{O}(1)$ liegen.

Aufgabe 5.2 – Betrunkener Übungsleiter

Zeigen Sie mithilfe der Bankkonto-Methode, dass die amortisierten Laufzeiten der Operationen **hoch** und **runter(int k)** in $\mathcal{O}(1)$ liegen.

$\Delta(\text{hoch}) = 1$

$A(\text{hoch}) = 1 + 1 = 2 \in O(1)$

$\Delta(\text{runter}(k)) = -\min\{h, k\}$

$A(\text{runter}(k)) = \min\{h, k\} + (-\min\{h, k\}) = 0 \in O(1)$

- Tokens gekoppelt an Höhe (nie negativ)

Aufgabe 5.3 – Hashing mit Chaining

Veranschaulichen Sie Hashing mit Chaining. Die Größe m der Hash-Tabelle ist in den folgenden Beispielen jeweils die Primzahl 11. Die folgenden Operationen sollen nacheinander ausgeführt werden.

`insert`	$3, 11, 9, 7, 14, 56, 4, 12, 15, 8, 1$
`delete`	56
`insert`	25

Der Einfachheit halber sollen die Schlüssel der Elemente die Elemente selbst sein.

Aufgabe 5.3 – Hashing mit Chaining

a) Verwenden Sie zunächst die Hashfunktion $m = 11$

$$g(x) = 5x \mod m.$$

$k(e)$	1	3	4	7	8	9	11	12	14	15	25	56
$g(k(e))$	5	4	9	2	7	1	0	5	4	5	4	5
$h(k(e))$	5	4	9	2	7	1	0	5	4	9	4	5

b) Berechnen Sie die Hashwerte unter Verwendung der Hashfunktion

$$h(x) = \mathbf{a} \cdot \mathbf{x} \mod m \qquad x = (0, 1)$$

nach dem aus der Vorlesung bekannten Verfahren für einfache universelle Hashfunktionen, wobei $\mathbf{a} = (7, 5)$ und $\mathbf{x} = (\lfloor \frac{x}{2^w} \rfloor \mod 2^w, x \mod 2^w)$ für $w = \lfloor \log_2 m \rfloor = \lfloor 3.45 \ldots \rfloor = 3$ gilt und der Ausdruck $\mathbf{a} \cdot \mathbf{x}$ ein Skalarprodukt bezeichnet.

Aufgabe 5.3 – Hashing mit Chaining

a)

$k(e)$	1	3	4	7	8	9	11	12	14	15	25	56
$g(k(e))$	5	4	9	2	7	1	0	5	4	9	4	5

b)

$k(e)$	1	3	4	7	8	9	11	12	14	15	25	56
$h(k(e))$	5	4	9	2	7	1	0	5	4	9	4	5

1. Operation: `insert`(3):

0	1	2	3	4	5	6	7	8	9	10
				3						

Aufgabe 5.3 – Hashing mit Chaining

a)

$k(e)$	1	3	4	7	8	9	11	12	14	15	25	56
$g(k(e))$	5	4	9	2	7	1	0	5	4	9	4	5

b)

$k(e)$	1	3	4	7	8	9	11	12	14	15	25	56
$h(k(e))$	5	4	9	2	7	1	0	5	4	9	4	5

2. Operation: `insert(11)`:

0	1	2	3	4	5	6	7	8	9	10
11				3						

Aufgabe 5.3 – Hashing mit Chaining

a)

$k(e)$	1	3	4	7	8	9	11	12	14	15	25	56
$g(k(e))$	5	4	9	2	7	1	0	5	4	9	4	5

b)

$k(e)$	1	3	4	7	8	9	11	12	14	15	25	56
$h(k(e))$	5	4	9	2	7	1	0	5	4	9	4	5

3. Operation: `insert`(9):

0	1	2	3	4	5	6	7	8	9	10
11	9			3						

Aufgabe 5.3 – Hashing mit Chaining

a)

$k(e)$	1	3	4	7	8	9	11	12	14	15	25	56
$g(k(e))$	5	4	9	2	7	1	0	5	4	9	4	5

b)

$k(e)$	1	3	4	7	8	9	11	12	14	15	25	56
$h(k(e))$	5	4	9	2	7	1	0	5	4	9	4	5

4. Operation: `insert`(7):

0	1	2	3	4	5	6	7	8	9	10
11	9	7		3						

Aufgabe 5.3 – Hashing mit Chaining

a)

$k(e)$	1	3	4	7	8	9	11	12	14	15	25	56
$g(k(e))$	5	4	9	2	7	1	0	5	4	9	4	5

b)

$k(e)$	1	3	4	7	8	9	11	12	14	15	25	56
$h(k(e))$	5	4	9	2	7	1	0	5	4	9	4	5

5. Operation: `insert`(14):

0	1	2	3	4	5	6	7	8	9	10
11	9	7		3						
				14						

Aufgabe 5.3 – Hashing mit Chaining

a)

$k(e)$	1	3	4	7	8	9	11	12	14	15	25	56
$g(k(e))$	5	4	9	2	7	1	0	5	4	9	4	5

b)

$k(e)$	1	3	4	7	8	9	11	12	14	15	25	56
$h(k(e))$	5	4	9	2	7	1	0	5	4	9	4	5

6. Operation: `insert`(56):

0	1	2	3	4	5	6	7	8	9	10
11	9	7		3	56					
				14						

Aufgabe 5.3 – Hashing mit Chaining

a)

$k(e)$	1	3	4	7	8	9	11	12	14	15	25	56
$g(k(e))$	5	4	9	2	7	1	0	5	4	9	4	5

b)

$k(e)$	1	3	4	7	8	9	11	12	14	15	25	56
$h(k(e))$	5	4	9	2	7	1	0	5	4	9	4	5

7. Operation: `insert`(4):

0	1	2	3	4	5	6	7	8	9	10
11	9	7		3	56				4	
				14						

Aufgabe 5.3 – Hashing mit Chaining

a)

$k(e)$	1	3	4	7	8	9	11	12	14	15	25	56
$g(k(e))$	5	4	9	2	7	1	0	5	4	9	4	5

b)

$k(e)$	1	3	4	7	8	9	11	12	14	15	25	56
$h(k(e))$	5	4	9	2	7	1	0	5	4	9	4	5

8. Operation: `insert`(12):

0	1	2	3	4	5	6	7	8	9	10
11	9	7		3	56				4	
				14	12					

Aufgabe 5.3 – Hashing mit Chaining

a)

$k(e)$	1	3	4	7	8	9	11	12	14	15	25	56
$g(k(e))$	5	4	9	2	7	1	0	5	4	9	4	5

b)

$k(e)$	1	3	4	7	8	9	11	12	14	15	25	56
$h(k(e))$	5	4	9	2	7	1	0	5	4	9	4	5

9. Operation: `insert`(15):

0	1	2	3	4	5	6	7	8	9	10
11	9	7		3	56				4	
				14	12				15	

Aufgabe 5.3 – Hashing mit Chaining

a)

$k(e)$	1	3	4	7	8	9	11	12	14	15	25	56
$g(k(e))$	5	4	9	2	7	1	0	5	4	9	4	5

b)

$k(e)$	1	3	4	7	8	9	11	12	14	15	25	56
$h(k(e))$	5	4	9	2	7	1	0	5	4	9	4	5

10. Operation: `insert`(8):

0	1	2	3	4	5	6	7	8	9	10
11	9	7		3	56		8		4	
				14	12				15	

Aufgabe 5.3 – Hashing mit Chaining

a)

$k(e)$	1	3	4	7	8	9	11	12	14	15	25	56
$g(k(e))$	5	4	9	2	7	1	0	5	4	9	4	5

b)

$k(e)$	1	3	4	7	8	9	11	12	14	15	25	56
$h(k(e))$	5	4	9	2	7	1	0	5	4	9	4	5

11. Operation: `insert`(1):

0	1	2	3	4	5	6	7	8	9	10
11	9	7		3	56		8		4	
				14	12				15	
					1					

9

Aufgabe 5.3 – Hashing mit Chaining

a)

$k(e)$	1	3	4	7	8	9	11	12	14	15	25	56
$g(k(e))$	5	4	9	2	7	1	0	5	4	9	4	5

b)

$k(e)$	1	3	4	7	8	9	11	12	14	15	25	56
$h(k(e))$	5	4	9	2	7	1	0	5	4	9	4	5

12. Operation: `delete`(56):

0	1	2	3	4	5	6	7	8	9	10
11	9	7		3	12		8		4	
				14	1				15	

Aufgabe 5.3 – Hashing mit Chaining

a)

$k(e)$	1	3	4	7	8	9	11	12	14	15	25	56
$g(k(e))$	5	4	9	2	7	1	0	5	4	9	4	5

b)

$k(e)$	1	3	4	7	8	9	11	12	14	15	25	56
$h(k(e))$	5	4	9	2	7	1	0	5	4	9	4	5

13. Operation: `insert`(25):

0	1	2	3	4	5	6	7	8	9	10
11	9	7		3	12		8		4	
				14	1				15	
				25						

Aufgabe 5.4 – Stapelschlange

In dieser Aufgabe geht es darum, einen Algorithmus, der eine Queue für `Integer`-Zahlen mittels zweier Stacks implementiert, hinsichtlich seiner Laufzeit zu untersuchen.

```java
class Stapelschlange {
  private Stack s1 = new Stack();
  private Stack s2 = new Stack();

  public void enqueue(int v) {
    s1.push(v);
  }

  public int dequeue() {
    if(s2.isEmpty())
      while(!s1.isEmpty())
        s2.push(s1.pop());
    return s2.pop();
  }
}
```

$n \cdot (pop + push + isEmpty)$

$n \cdot (O(1))$

$O(n \cdot \underbrace{c}_{\text{Konstant}}) = \underline{O(n)}$

Aufgabe 5.4 – Stapelschlange

Die Klasse `Stack` hat dabei folgende Methoden:

Methode	Beschreibung	Laufzeitklasse
`void push(int v)`	legt eine Zahl auf den Stack	$\mathcal{O}(1)$
`int pop()`	nimmt die oberste Zahl vom Stack	$\mathcal{O}(1)$
`boolean isEmpty()`	prüft, ob der Stack leer ist	$\mathcal{O}(1)$

a) Geben Sie die Laufzeitklasse der Worst Case-Laufzeit eines Aufrufs der `dequeue()`-Methode in Abhängigkeit der aktuellen Größe der Queue n in Landau-Notation an.

b) Überlegen Sie sich ein Amortisierungsschema, welches die amortisierte Laufzeit der einzelnen Operationen minimiert. Nennen Sie die amortisierten Laufzeitklassen der `enqueue(int v)`- und `dequeue()`-Methode, die sich aus Ihrem Amortisierungsschema ergeben. Zeigen Sie die Richtigkeit Ihres Amortisierungsschemas (das Tokenkonto darf niemals negativ werden) und der resultierenden Laufzeitklassen.

Aufgabe 5.4 – Stapelschlange

a) Geben Sie die Laufzeitklasse der Worst Case-Laufzeit eines Aufrufs der `dequeue()`-Methode in Abhängigkeit der aktuellen Größe der Queue n in Landau-Notation an.

Wenn s2 nicht leer: $O(1)$ (nur pop)

s2 leer : $O(1) + \underline{c} \cdot n$ (pop + für jedes Element in der Queue konstanter Aufwand zum push und pop nach s1)

$\Rightarrow O(1) + O(c \cdot n)$

$\in O(1) + O(n)$

$\in O(n)$

c, konstant

$\Rightarrow O(n)$ worst-case

Aufgabe 5.4 – Stapelschlange

b) Überlegen Sie sich ein Amortisierungsschema, welches die amortisierte Laufzeit der einzelnen Operationen minimiert. Nennen Sie die amortisierten Laufzeitklassen der `enqueue(int v)`- und `dequeue()`-Methode, die sich aus Ihrem Amortisierungsschema ergeben. Zeigen Sie die Richtigkeit Ihres Amortisierungsschemas (das Tokenkonto darf niemals negativ werden) und der resultierenden Laufzeitklassen.

$\Delta(\text{enqueue}): c$

$\Delta(\text{dequeue}) = \begin{cases} -n \cdot c : & S2 \text{ leer} \\ 0 : & \text{sonst} \end{cases}$

$T(\text{enqueue}) \in O(1)$

$T(\text{dequeue}) = \begin{cases} O(1) + c \cdot n : & S2 \text{ leer} \\ O(1) : & \text{sonst} \end{cases}$

$A(\text{enqueue}) = O(1) + c$

$A(\text{dequeue}) = \begin{cases} O(1) + c \cdot n - c \cdot n \in O(1) : & S2 \text{ leer} \\ O(1) + 0 \in O(1) : & \text{sonst} \end{cases}$