

Datenstrukturen und Algorithmen

Übung 5

FS 2021

Programm von heute

Amortisierte Analyse: push_back

Strategie: verdoppeln wenn Array voll ist.

Amortisierte Analyse: push_back

Strategie: verdoppeln wenn Array voll ist.

Sei $i \in \mathbb{N}$ die Anzahl eingefügter Elemente und $n_i \in \mathbb{N}$ die Arraygrösse nachdem i eingefügt wurde.

Es gilt

$$n_i = \begin{cases} 1 & \text{falls } i = 1 \text{ [Start]} \\ 2 \cdot n_{i-1} & \text{falls } i - 1 \in \{2^k : k \in \mathbb{N}\} \text{ [Array voll]} \\ n_{i-1} & \text{sonst} \end{cases}$$

$$n_i = 2^{\lceil \log_2 i \rceil}$$

i	n_i
1	1
2	2
3	4
4	4
5	8
6	8
...	...

Amortisierte Analyse: push_back

Strategie: verdoppeln wenn Array voll ist.

¹Nach Aufgabenstellung: $2n$ Initialisierungen + n Kopien + neues Element

Amortisierte Analyse: push_back

Strategie: verdoppeln wenn Array voll ist.

Reale Kosten

$$t_i = \begin{cases} 1 & \text{falls } i = 1 \text{ [Start]} \\ 3n_{i-1} + 1 & \text{falls } i - 1 \in \{2^k : k \in \mathbb{N}\} \text{ [Array voll]}^1 \\ 1 & \text{sonst} \end{cases}$$

¹Nach Aufgabenstellung: $2n$ Initialisierungen + n Kopien + neues Element

Amortisierte Analyse: push_back

Strategie: verdoppeln wenn Array voll ist.

Reale Kosten

$$t_i = \begin{cases} 1 & \text{falls } i = 1 \text{ [Start]} \\ 3n_{i-1} + 1 & \text{falls } i - 1 \in \{2^k : k \in \mathbb{N}\} \text{ [Array voll]}^1 \\ 1 & \text{sonst} \end{cases}$$

Finde Potentialfunktion so dass amortisierte Kosten konstant sind:

$$a_i = t_i + \Phi_i - \Phi_{i-1}$$

¹Nach Aufgabenstellung: $2n$ Initialisierungen + n Kopien + neues Element

Amortisierte Analyse: push_back

Strategie: verdoppeln wenn Array voll ist.

Finde Potentialfunktion so dass amortisierte Kosten konstant sind:

$$a_i = t_i + \Phi_i - \Phi_{i-1}$$

Amortisierte Analyse: push_back

Strategie: verdoppeln wenn Array voll ist.

Finde Potentialfunktion so dass amortisierte Kosten konstant sind:

$$a_i = t_i + \Phi_i - \Phi_{i-1}$$

$$\begin{aligned}\Phi_i &= 6 \cdot \text{Anzahl Elemente in der oberen Hälfte des Arrays} \\ &= 6 \cdot \left(i - \frac{n_i}{2}\right) = 6i - 3n_i\end{aligned}$$

Amortisierte Analyse: push_back

Strategie: verdoppeln wenn Array voll ist.

Finde Potentialfunktion so dass amortisierte Kosten konstant sind:

$$a_i = t_i + \Phi_i - \Phi_{i-1}$$

$\Phi_i = 6 \cdot$ Anzahl Elemente in der oberen Hälfte des Arrays

$$= 6 \cdot \left(i - \frac{n_i}{2}\right) = 6i - 3n_i$$

$$\Phi_i - \Phi_{i-1} = \begin{cases} 6 + 3n_{i-1} - 3 \widehat{n_i}^{2 \cdot n_{i-1}} & \text{falls } i - 1 \in \{2^k : k \in \mathbb{N}\} \text{ [Array voll]} \\ 6 & \text{sonst} \end{cases}$$

Amortisierte Analyse: push_back

Strategie: verdoppeln wenn Array voll ist.

Finde Potentialfunktion so dass amortisierte Kosten konstant sind:

$$\begin{aligned} a_i &= t_i + \Phi_i - \Phi_{i-1} \\ &= \begin{cases} 3n_{i-1} + 1 + 6 - 3n_{i-1} & \text{falls } i - 1 \in \{2^k : k \in \mathbb{N}\} \text{ [Array voll]} \\ 1 + 6 & \text{sonst} \end{cases} \\ &\leq 7 \quad \text{für alle } i \end{aligned}$$

Amortisierte Analyse: pop_back

Strategie: halbieren wenn Array viertel leer ist.

Amortisierte Analyse: pop_back

Strategie: halbieren wenn Array viertel leer ist.

$$t_i = \begin{cases} 1 & \text{wenn Array mehr als viertel voll ist} \\ \frac{n_{i-1}}{2} + \frac{n_{i-1}}{4} = \frac{3}{4}n_{i-1} & \text{sonst, danach } n_i = \frac{n_{i-1}}{2} \end{cases}$$

Amortisierte Analyse: pop_back

Strategie: halbieren wenn Array viertel leer ist.

$$t_i = \begin{cases} 1 & \text{wenn Array mehr als viertel voll ist} \\ \frac{n_{i-1}}{2} + \frac{n_{i-1}}{4} = \frac{3}{4}n_{i-1} & \text{sonst, danach } n_i = \frac{n_{i-1}}{2} \end{cases}$$

Finde Potentialfunktion so dass amortisierte Kosten konstant sind:

$$a_i = t_i + \Phi_i - \Phi_{i-1}$$

Amortisierte Analyse: pop_back

Strategie: halbieren wenn Array viertel leer ist.

$$t_i = \begin{cases} 1 & \text{wenn Array mehr als viertel voll ist} \\ \frac{n_{i-1}}{2} + \frac{n_{i-1}}{4} = \frac{3}{4}n_{i-1} & \text{sonst, danach } n_i = \frac{n_{i-1}}{2} \end{cases}$$

Finde Potentialfunktion so dass amortisierte Kosten konstant sind:

$$a_i = t_i + \Phi_i - \Phi_{i-1}$$

Sei k_i die Anzahl Elemente im Array im Schritt i

$$\begin{aligned} \Phi_i &= 3 \cdot \text{Anzahl leerer Elemente in der unteren Hälfte des Arrays } (1, \dots, \frac{n_i}{2}) \\ &= 3 \cdot \left(\frac{n_i}{2} - k_i \right) \end{aligned}$$

Amortisierte Analyse: pop_back

Strategie: halbieren wenn Array viertel leer ist.

$$t_i = \begin{cases} 1 & \text{wenn Array mehr als viertel voll ist} \\ \frac{n_{i-1}}{2} + \frac{n_{i-1}}{4} = \frac{3}{4}n_{i-1} & \text{sonst, danach } n_i = \frac{n_{i-1}}{2} \end{cases}$$

Finde Potentialfunktion so dass amortisierte Kosten konstant sind:

$$a_i = t_i + \Phi_i - \Phi_{i-1}$$

Sei k_i die Anzahl Elemente im Array im Schritt i

$$\begin{aligned} \Phi_i &= 3 \cdot \text{Anzahl leerer Elemente in der unteren Hälfte des Arrays } (1, \dots, \frac{n_i}{2}) \\ &= 3 \cdot \left(\frac{n_i}{2} - k_i \right) \end{aligned}$$

Amortisierte Analyse: pop_back

Strategie: halbieren wenn Array viertel leer ist. Finde Potentialfunktion so dass amortisierte Kosten konstant sind:

$$a_i = t_i + \Phi_i - \Phi_{i-1}$$

$$\Phi_i = 3 \cdot \left(\frac{n_i}{2} - k_i \right)$$

$$\Phi_i - \Phi_{i-1} = \begin{cases} 3 & \text{wenn Array mehr als viertel voll ist} \\ 3 \cdot \left(1 + \frac{n_{i-1}}{4} - \frac{n_{i-1}}{2} \right) & \text{sonst} \end{cases}$$

Amortisierte Analyse: pop_back

Strategie: halbieren wenn Array viertel leer ist. Finde Potentialfunktion so dass amortisierte Kosten konstant sind:

$$a_i = t_i + \Phi_i - \Phi_{i-1}$$

$$\Phi_i = 3 \cdot \left(\frac{n_i}{2} - k_i \right)$$

$$\Phi_i - \Phi_{i-1} = \begin{cases} 3 & \text{wenn Array mehr als viertel voll ist} \\ 3 \cdot \left(1 + \frac{n_{i-1}}{4} - \frac{n_{i-1}}{2} \right) & \text{sonst} \end{cases}$$

$$\Rightarrow 4 \geq a_i \text{ (in beiden Fällen)}$$

Amortisierte Analyse: pop und push

$$\Phi_i = 6 \cdot \text{Anzahl Elemente obere Hälfte} \\ + 3 \cdot \text{Anzahl leere Elemente untere Hälfte}$$

2. Wiederholung Theorie

Gutes Hashing

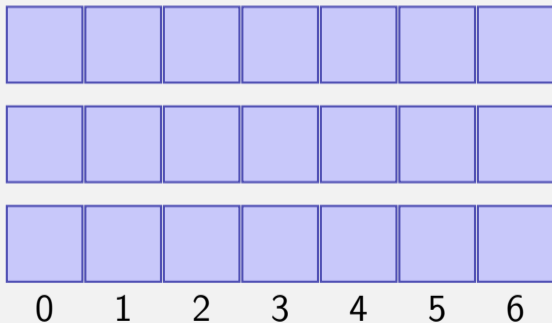
Gutes Hashing. . .

- verteilt die Menge der Schlüssel möglichst gleichmässig auf die Positionen der Hashtabelle.
- vermeidet beim Sondieren möglichst ein Ablaufen langer belegter Bereiche (siehe primäre Häufung).
- vermeidet, Schlüssel mit gleichem Hashwert auch noch gleich zu sondieren (siehe sekundäre Häufung).

Beispiele Hashing

Füge die Schlüssel 25, 4, 17, 45 in die Hashtabelle ein. Verwende dabei $h(k) = k \bmod 7$ und sondiere nach rechts, $h(k) + s(j, k)$:

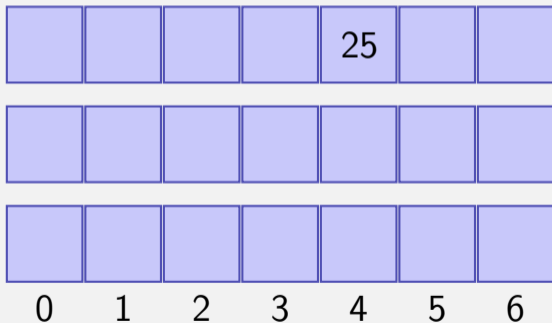
- Lineares Sondieren,
 $s(j, k) = j$.
- Quadratisches Sondieren,
 $s(j, k) = (-1)^{j+1} \lceil j/2 \rceil^2$.
- Double Hashing,
 $s(j, k) = j \cdot (1 + (k \bmod 5))$.



Beispiele Hashing

Füge die Schlüssel 25, 4, 17, 45 in die Hashtabelle ein. Verwende dabei $h(k) = k \bmod 7$ und sondiere nach rechts, $h(k) + s(j, k)$:

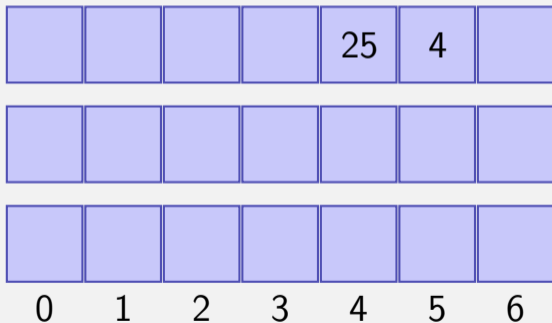
- Lineares Sondieren,
 $s(j, k) = j$.
- Quadratisches Sondieren,
 $s(j, k) = (-1)^{j+1} \lceil j/2 \rceil^2$.
- Double Hashing,
 $s(j, k) = j \cdot (1 + (k \bmod 5))$.



Beispiele Hashing

Füge die Schlüssel 25, 4, 17, 45 in die Hashtabelle ein. Verwende dabei $h(k) = k \bmod 7$ und sondiere nach rechts, $h(k) + s(j, k)$:

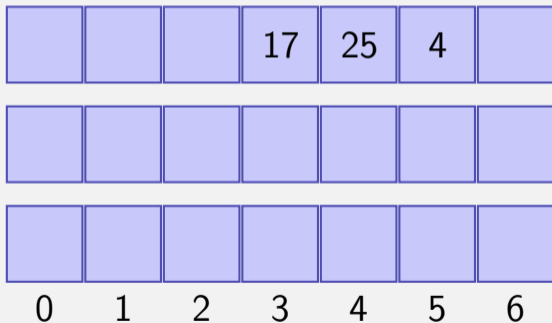
- Lineares Sondieren,
 $s(j, k) = j$.
- Quadratisches Sondieren,
 $s(j, k) = (-1)^{j+1} \lceil j/2 \rceil^2$.
- Double Hashing,
 $s(j, k) = j \cdot (1 + (k \bmod 5))$.



Beispiele Hashing

Füge die Schlüssel 25, 4, 17, 45 in die Hashtabelle ein. Verwende dabei $h(k) = k \bmod 7$ und sondiere nach rechts, $h(k) + s(j, k)$:

- Lineares Sondieren,
 $s(j, k) = j$.
- Quadratisches Sondieren,
 $s(j, k) = (-1)^{j+1} \lceil j/2 \rceil^2$.
- Double Hashing,
 $s(j, k) = j \cdot (1 + (k \bmod 5))$.



Beispiele Hashing

Füge die Schlüssel 25, 4, 17, 45 in die Hashtabelle ein. Verwende dabei $h(k) = k \bmod 7$ und sondiere nach rechts, $h(k) + s(j, k)$:

- Lineares Sondieren,
 $s(j, k) = j$.
- Quadratisches Sondieren,
 $s(j, k) = (-1)^{j+1} \lceil j/2 \rceil^2$.
- Double Hashing,
 $s(j, k) = j \cdot (1 + (k \bmod 5))$.

			17	25	4	45

0 1 2 3 4 5 6

Beispiele Hashing

Füge die Schlüssel 25, 4, 17, 45 in die Hashtabelle ein. Verwende dabei $h(k) = k \bmod 7$ und sondiere nach rechts, $h(k) + s(j, k)$:

- Lineares Sondieren,
 $s(j, k) = j$.
- Quadratisches Sondieren,
 $s(j, k) = (-1)^{j+1} \lceil j/2 \rceil^2$.
- Double Hashing,
 $s(j, k) = j \cdot (1 + (k \bmod 5))$.

			17	25	4	45
				25		

0 1 2 3 4 5 6

Beispiele Hashing

Füge die Schlüssel 25, 4, 17, 45 in die Hashtabelle ein. Verwende dabei $h(k) = k \bmod 7$ und sondiere nach rechts, $h(k) + s(j, k)$:

- Lineares Sondieren,
 $s(j, k) = j$.
- Quadratisches Sondieren,
 $s(j, k) = (-1)^{j+1} \lceil j/2 \rceil^2$.
- Double Hashing,
 $s(j, k) = j \cdot (1 + (k \bmod 5))$.

			17	25	4	45
				25	4	
0	1	2	3	4	5	6

Beispiele Hashing

Füge die Schlüssel 25, 4, 17, 45 in die Hashtabelle ein. Verwende dabei $h(k) = k \bmod 7$ und sondiere nach rechts, $h(k) + s(j, k)$:

- Lineares Sondieren,
 $s(j, k) = j$.
- Quadratisches Sondieren,
 $s(j, k) = (-1)^{j+1} \lceil j/2 \rceil^2$.
- Double Hashing,
 $s(j, k) = j \cdot (1 + (k \bmod 5))$.

			17	25	4	45
			17	25	4	
0	1	2	3	4	5	6

Beispiele Hashing

Füge die Schlüssel 25, 4, 17, 45 in die Hashtabelle ein. Verwende dabei $h(k) = k \bmod 7$ und sondiere nach rechts, $h(k) + s(j, k)$:

- Lineares Sondieren,
 $s(j, k) = j$.
- Quadratisches Sondieren,
 $s(j, k) = (-1)^{j+1} \lceil j/2 \rceil^2$.
- Double Hashing,
 $s(j, k) = j \cdot (1 + (k \bmod 5))$.

			17	25	4	45
		45	17	25	4	
0	1	2	3	4	5	6

Beispiele Hashing

Füge die Schlüssel 25, 4, 17, 45 in die Hashtabelle ein. Verwende dabei $h(k) = k \bmod 7$ und sondiere nach rechts, $h(k) + s(j, k)$:

- Lineares Sondieren,
 $s(j, k) = j$.
- Quadratisches Sondieren,
 $s(j, k) = (-1)^{j+1} \lceil j/2 \rceil^2$.
- Double Hashing,
 $s(j, k) = j \cdot (1 + (k \bmod 5))$.

			17	25	4	45
		45	17	25	4	
				25		
0	1	2	3	4	5	6

Beispiele Hashing

Füge die Schlüssel 25, 4, 17, 45 in die Hashtabelle ein. Verwende dabei $h(k) = k \bmod 7$ und sondiere nach rechts, $h(k) + s(j, k)$:

- Lineares Sondieren,
 $s(j, k) = j$.
- Quadratisches Sondieren,
 $s(j, k) = (-1)^{j+1} \lceil j/2 \rceil^2$.
- Double Hashing,
 $s(j, k) = j \cdot (1 + (k \bmod 5))$.

			17	25	4	45
		45	17	25	4	
		4		25		
0	1	2	3	4	5	6

Beispiele Hashing

Füge die Schlüssel 25, 4, 17, 45 in die Hashtabelle ein. Verwende dabei $h(k) = k \bmod 7$ und sondiere nach rechts, $h(k) + s(j, k)$:

- Lineares Sondieren,
 $s(j, k) = j$.
- Quadratisches Sondieren,
 $s(j, k) = (-1)^{j+1} \lceil j/2 \rceil^2$.
- Double Hashing,
 $s(j, k) = j \cdot (1 + (k \bmod 5))$.

			17	25	4	45
		45	17	25	4	
		4	17	25		
0	1	2	3	4	5	6

Beispiele Hashing

Füge die Schlüssel 25, 4, 17, 45 in die Hashtabelle ein. Verwende dabei $h(k) = k \bmod 7$ und sondiere nach rechts, $h(k) + s(j, k)$:

- Lineares Sondieren,
 $s(j, k) = j$.
- Quadratisches Sondieren,
 $s(j, k) = (-1)^{j+1} \lceil j/2 \rceil^2$.
- Double Hashing,
 $s(j, k) = j \cdot (1 + (k \bmod 5))$.

			17	25	4	45
		45	17	25	4	
		4	17	25	45	
0	1	2	3	4	5	6

Simple Uniform Hashing

Aussage über Gleichverteilung und Unabhängigkeit der *Schlüssel*.

Eigenschaft von geschlossener Addressierung: einfaches gleichmässiges Hashing \Rightarrow Erwartete Länge der Ketten so gut wie nur möglich:

$$\leq \alpha = \frac{n}{m}$$

Uniform Hashing

Aussage über die Gleichverteilung und Unabhängigkeit der *Sondierungssequenzen der Schlüssel*.

Eigenschaft der offenen Addressierung: Gleichmässiges Hashing \Rightarrow
Erwartete Laufzeitkosten $\leq \frac{1}{1-\alpha}$.

Universal Hashing

Aussage über die verfügbaren, zufällig gewählten *Hash-Funktionen*.

$$|\{h \in \mathcal{H} \text{ mit } h(k_1) = h(k_2)\}| \leq \frac{|\mathcal{H}|}{m}$$

Eigenschaft unabhängig von der gewählten Sequenz der Schlüssel:
beim Hashing mit Verketteten erwartete Kettenlänge $\leq \alpha = \frac{n}{m}$

Vorbedingung für das perfekte Hashing

3. Programmieraufgabe

Finden eines Sub-Arrays

- Gegeben: Zwei Felder $A = (a_0, \dots, a_{n-1})$ and $B = (b_0, \dots, b_{k-1})$ mit natürlichen Zahlen
- Aufgabe: Finde Position von B in A .

Finden eines Sub-Arrays

- Gegeben: Zwei Felder $A = (a_0, \dots, a_{n-1})$ and $B = (b_0, \dots, b_{k-1})$ mit natürlichen Zahlen
- Aufgabe: Finde Position von B in A .
- Naiv: Gehe durch A , vergleiche die k folgenden Einträge mit B

Finden eines Sub-Arrays

- Gegeben: Zwei Felder $A = (a_0, \dots, a_{n-1})$ and $B = (b_0, \dots, b_{k-1})$ mit natürlichen Zahlen
- Aufgabe: Finde Position von B in A .
- Naiv: Gehe durch A , vergleiche die k folgenden Einträge mit B
 - $O(nk)$ Vergleiche

Finden eines Sub-Arrays

- Gegeben: Zwei Felder $A = (a_0, \dots, a_{n-1})$ and $B = (b_0, \dots, b_{k-1})$ mit natürlichen Zahlen
- Aufgabe: Finde Position von B in A .
- Naiv: Gehe durch A , vergleiche die k folgenden Einträge mit B
 - $O(nk)$ Vergleiche
- Lösung mit Hashing: Berechne Hash $h(B)$ und vergleiche mit $h((a_i, a_{i+1}, \dots, a_{i+k-1}))$.
- Vermeide die Neuberechnung von $h((a_i, a_{i+1}, \dots, a_{i+k-1}))$ für jedes $i \implies O(n)$ erwartet

Sliding Window Hash

- Mögliche Hash-Funktion: Summe aller Elemente:
 - Kann einfach aktualisiert werden: a_i abziehen und a_{i+k} hinzufügen.
 - Aber: schlechte Hash-Funktion

Sliding Window Hash

- Mögliche Hash-Funktion: Summe aller Elemente:
 - Kann einfach aktualisiert werden: a_i abziehen und a_{i+k} hinzufügen.
 - Aber: schlechte Hash-Funktion
- Besser:

$$H_{c,m}((a_i, \dots, a_{i+k-1})) = \left(\sum_{j=0}^{k-1} a_{i+j} \cdot c^{k-j-1} \right) \bmod m$$

- $c = 1021$ Primzahl
- $m = 2^{15}$ `int`, keine Überläufe bei Berechnungen

Rechenregeln Modulo

$$(a + b) \bmod m = ((a \bmod m) + (b \bmod m)) \bmod m$$

$$(a - b) \bmod m = ((a \bmod m) - (b \bmod m) + m) \bmod m$$

$$(a \cdot b) \bmod m = ((a \bmod m) \cdot (b \bmod m)) \bmod m$$

Aufgabe: Berechnen Sie

$$12746357 \bmod 11$$

Rechenregeln Modulo

Aufgabe: Berechnen Sie

$$12746357 \bmod 11$$

Rechenregeln Modulo

Aufgabe: Berechnen Sie

$$12746357 \bmod 11$$

$$= (7 + 5 \cdot 10 + 3 \cdot 10^2 + 6 \cdot 10^3 + 4 \cdot 10^4 + 7 \cdot 10^5 + 2 \cdot 10^6 + 1 \cdot 10^7) \bmod 11$$

Rechenregeln Modulo

Aufgabe: Berechnen Sie

$$12746357 \bmod 11$$

$$= (7 + 5 \cdot 10 + 3 \cdot 10^2 + 6 \cdot 10^3 + 4 \cdot 10^4 + 7 \cdot 10^5 + 2 \cdot 10^6 + 1 \cdot 10^7) \bmod 11$$

$$= (7 + 50 + 3 + 60 + 4 + 70 + 2 + 10) \bmod 11$$

Für die zweite Gleichheit haben wir verwendet, dass $10^2 \bmod 11 = 1$.

Rechenregeln Modulo

Aufgabe: Berechnen Sie

$$\begin{aligned} & 12746357 \bmod 11 \\ &= (7 + 5 \cdot 10 + 3 \cdot 10^2 + 6 \cdot 10^3 + 4 \cdot 10^4 + 7 \cdot 10^5 + 2 \cdot 10^6 + 1 \cdot 10^7) \bmod 11 \\ &= (7 + 50 + 3 + 60 + 4 + 70 + 2 + 10) \bmod 11 \\ &= (7 + 6 + 3 + 5 + 4 + 4 + 2 + 10) \bmod 11 \end{aligned}$$

Für die zweite Gleichheit haben wir verwendet, dass $10^2 \bmod 11 = 1$.

Rechenregeln Modulo

Aufgabe: Berechnen Sie

$$\begin{aligned} & 12746357 \bmod 11 \\ &= (7 + 5 \cdot 10 + 3 \cdot 10^2 + 6 \cdot 10^3 + 4 \cdot 10^4 + 7 \cdot 10^5 + 2 \cdot 10^6 + 1 \cdot 10^7) \bmod 11 \\ &= (7 + 50 + 3 + 60 + 4 + 70 + 2 + 10) \bmod 11 \\ &= (7 + 6 + 3 + 5 + 4 + 4 + 2 + 10) \bmod 11 \\ &= 8 \bmod 11. \end{aligned}$$

Für die zweite Gleichheit haben wir verwendet, dass $10^2 \bmod 11 = 1$.

Sliding Window Hash

```
template<typename It1, typename It2>
It1 findOccurrence(const It1 from, const It1 to,
                  const It2 begin, const It2 end)
{
    const unsigned k = end - begin;
    const unsigned M = 32768;
    const unsigned C = 1021;

    // your code here
    // ...
}
```

Sliding Window Hash

```
// elements can be compared using std::equal:  
if(std::equal(window_left, window_right, begin, end))  
    return current;  
  
// if no occurrence is found return end of array  
return to;  
}
```

Sliding Window Hash

Gehen Sie sicher, dass

- der Algorithmus c^k nur einmal berechnet,
- alle Werte modulo m berechnet werden, um Überläufe zu vermeiden (verwenden Sie die Rechenregeln für Kongruenzen), und
- alle Werte positiv sind (z.B. durch Addition von Vielfachen von m).

Fragen?