Dynamische Arrays

	a[i]	insert/del am Ende
• normale Arrays	$O(1)$	$O(n)$ / $O(1)$
• verkettete Listen	$O(n)$	$O(1)$
• Bäume	$O(\log n)$	$O(\log n)$

Wollen sowohl referenzierung als auch insert/del am Ende in $O(1)$ erreichen

Lege Array der Länge L an $\quad (L \leftarrow 2L)$

Einfügen: Wenn n bereits $= L$, dann vergrößere Array und kopiere um, schreibe in nächstfreie Stelle rein

Angenommen $L = 2^k \Rightarrow 2^{k-1} < n \leq 2^k$

d.h. wenn wir mit $L=1$ anfangen und sukzessive Einfügen

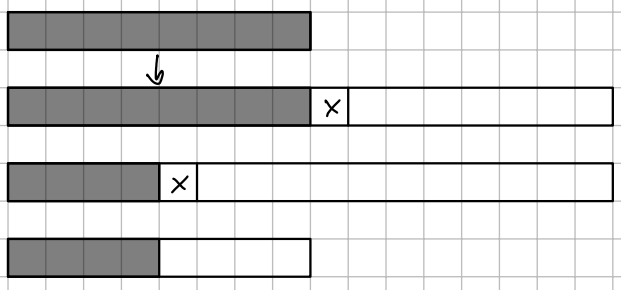
$\underbrace{1 + 2 + \dots + 2^{k-1}}_{n \text{ Einfügen}}$ umkopier OPs $= 2^k - 1$ OPs $\leq 2^k \leq 2n \in O(n)$

$\Rightarrow$ amortisiert eine kopier OP pro Einfügen

Löschen: Inhalt der letzten Zelle löschen

Wenn $n \leq \frac{L}{4}$ (nach der Löschung), dann $L \leftarrow \frac{L}{2}$ und kopiere um

Einfügen: $n = L \Rightarrow L \leftarrow 2L$



Löschen: $n \leq \frac{L}{4} \Rightarrow L \leftarrow \frac{L}{2}$

Invariante: Es wird nie mehr als n Speicher verschwendet ($L \leq 2n$)
± 1

Intuition: Bis zur nächsten teueren OP müssen in beiden fällen mind. n bzw $\frac{n}{2}$ billige OPs* durchgeführt werden, auf die dann die Kosten dieser teuren OP „amortisiert" werden können

* seit der letzten teueren OP

Beim Kuckuckshashing war der Kopieraufwand $O(1)$ erwartet amortisiert pro OP

beim linearen sondieren —— 11 —— $O(1)$ deterministisch —— 11 ——

Allgemein gibt es mehrere formale Verfahren, um amortisierte Laufzeitanalyse zu machen

- Bankkonto-Methode
- potential function method: wähle $\Phi(i)$ mit $\Phi(n) - \Phi(0) \geq 0$

$$T_{am}(i) := T_{echt}(i) + \Phi(i+1) - \Phi(i)$$

$$T_{am} := \sum_{i=0}^{n-1} T_{am}(i) = \sum_{i=0}^{n-1} T_{echt}(i) + \Phi(i+1) - \Phi(i) = T_{echt} + \underbrace{\sum_{i=0}^{n-1} \left(\Phi(i+1) - \Phi(i)\right)}_{\Phi(n) - \Phi(0)}$$

$$\Leftrightarrow T_{am} = T_{echt} + \Phi(n) - \Phi(0)$$

$$\Leftrightarrow T_{echt} = T_{am} - \underbrace{\left(\Phi(n) - \Phi(0)\right)}_{\geq 0} \leq T_{am}$$

Bsp. dyn. Array: $\Phi = |2n - L|$

billiges einfügen ($n \geq \frac{L}{2}$): $1 + \underbrace{|2(n+1) - L| - |2n - L|}_{2} = 3 \in O(1)$

teures Einfügen ($n = L$): $n + 1 + |2(n+1) - 2n| - |2n - n| = n + 1 + 2 - n = 3 \in O(1)$

(n: fürs Kopieren)

Implementierung in Java von der Klasse ArrayList

In python:

- `a[i] = ...` $O(1)$
- `a.append(x)`, `a.pop()` } $O(1)$ amortisiert
- `a+b`, `a+[x]`, `del a[s]` } $O(n)$

Mengen: `s = set()` } Hashtabellen

Wörterbücher: `d = dict` }

- `x in s` ?
- `s/d.add(x)` } erwartetes $O(1)$ Laufzeit
- `d[x]` ?

https://docs.oracle.com/javase/8/docs/technotes/guides/collections/overview.html