



Einführung in die Programmierung

Prof. Dr. Bertrand Meyer

Lecture 13: (Container-)Datenstrukturen



Container und Generik

Container-Operationen

Listen

Arrays

Die Performance eines Algorithmus beurteilen: Die Landau-Symbole (Big-O-Notation)

Hashtabellen

Stapelspeicher (fortan Stacks) und Warteschlangen (fortan queues)



Beinhalten andere Objekte (Elemente, engl. "*items*").

Fundamentale Operationen auf einen Container:

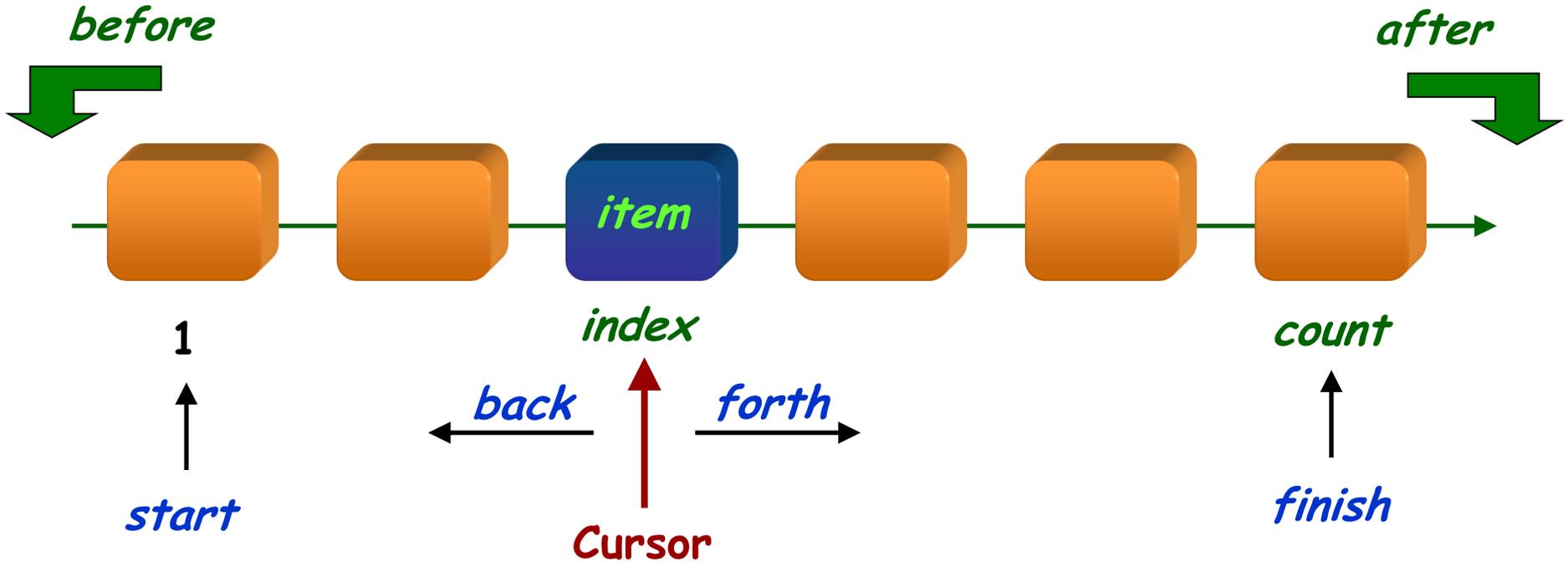
- **Einfügen**: Ein Element hinzufügen
- **Löschen**: Ein Vorkommen eines Elements (falls vorhanden) löschen
- **Ausradieren (Wipeout)**: Alle Vorkommen eines Elements löschen
- **Suchen**: Ist ein bestimmtes Element vorhanden?
- **Traversierung/Iteration**: Eine gegebene Operation auf jedes Element anwenden

Die Implementation eines Containers bestimmt, wie wir in dieser Stunde sehen werden:

- Welche dieser Operationen verfügbar sind
- Ihre Geschwindigkeiten
- Die Speicheranforderungen

Diese Lektion ist nur eine kurze Einführung. Mehr dazu im zweiten Semester in der Vorlesung "Datenstrukturen und Algorithmen"

Ein bekannter Container: Die Liste



Um die Iteration und andere Operationen zu vereinfachen haben unsere Listen **Cursors**. (hier intern, können aber auch extern sein.)

Abfragen
Befehle



Containerklassen in EiffelBase benutzen standardisierte Namen für grundlegende Containeroperationen:

```
is_empty: BOOLEAN  
has(v: G): BOOLEAN  
count: INTEGER  
item: G
```

```
make  
put(v: G)  
remove(v: G)  
wipe_out  
start, finish  
forth, back
```

Wenn angebracht, benutzen auch Sie diese Namen in Ihren eigenen Klassen

Beim Entwurf von Containerstrukturen sollte man festgelegte Grenzen vermeiden!

“Eng mich nicht ein!": EiffelBase mag harte Grenzen nicht.

- Die meisten Strukturen sind unbegrenzt
- Auch die Grösse von Arrays (zu jeder Zeit begrenzt) kann verändert werden

Wenn eine Struktur begrenzt ist, nennt man die maximale Anzahl Elemente *capacity*, und die Invariante lautet

count <= *capacity*



Wie behandeln wir Varianten von Containerklassen, die sich nur in ihrem Elementtyp unterscheiden?

Lösung: **Generik** ermöglicht explizite Typ-Parametrisierung, konsistent mit statischer Typisierung.

Containerstrukturen sind als generische Klassen implementiert:

LINKED_LIST [G]

pl : LINKED_LIST [PERSONE]

sl : LINKED_LIST [STRING]

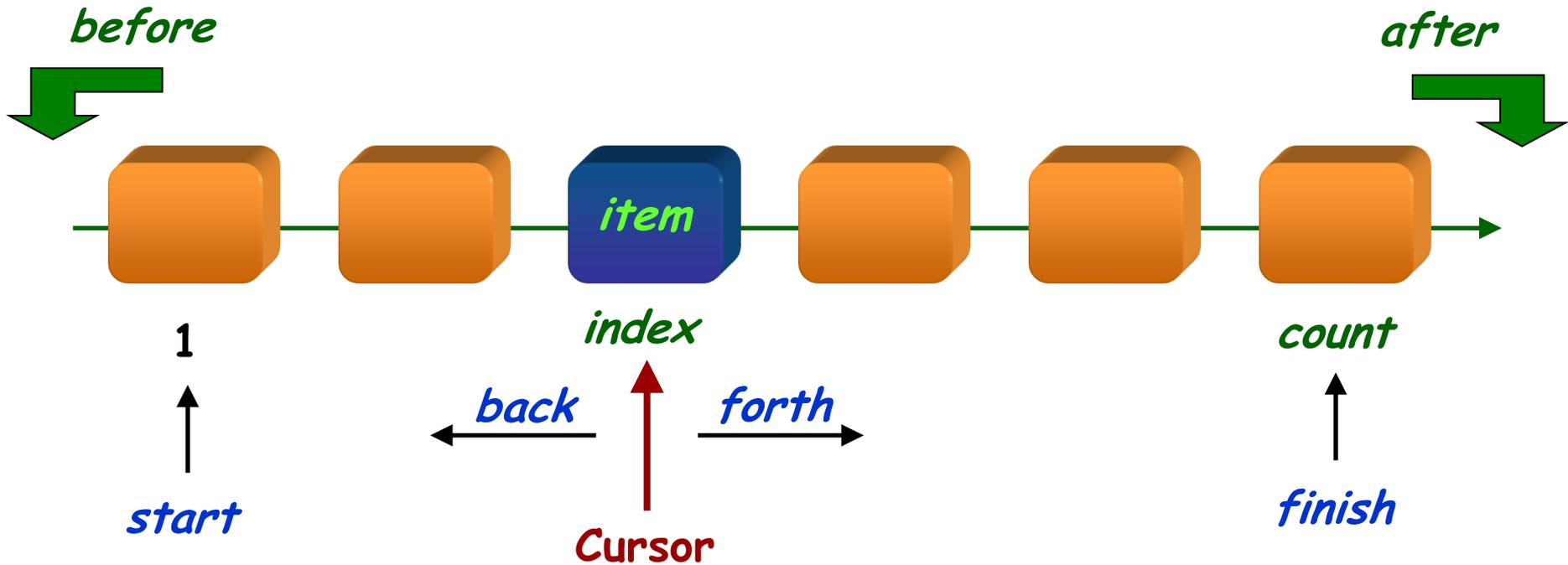
al : LINKED_LIST [ANY]

Listen



Eine Liste ist ein Container, der Elemente in einer bestimmten Ordnung beinhaltet.

Listen in EiffelBase haben Cursors



Cursor-Eigenschaften (Alle in der Klasseninvariante!)



Der Cursor reicht von 0 bis $count + 1$:

$$0 \leq index \leq count + 1$$

Der Cursor ist auf Position 0 gdw *before* wahr ist:

$$before = (index = 0)$$

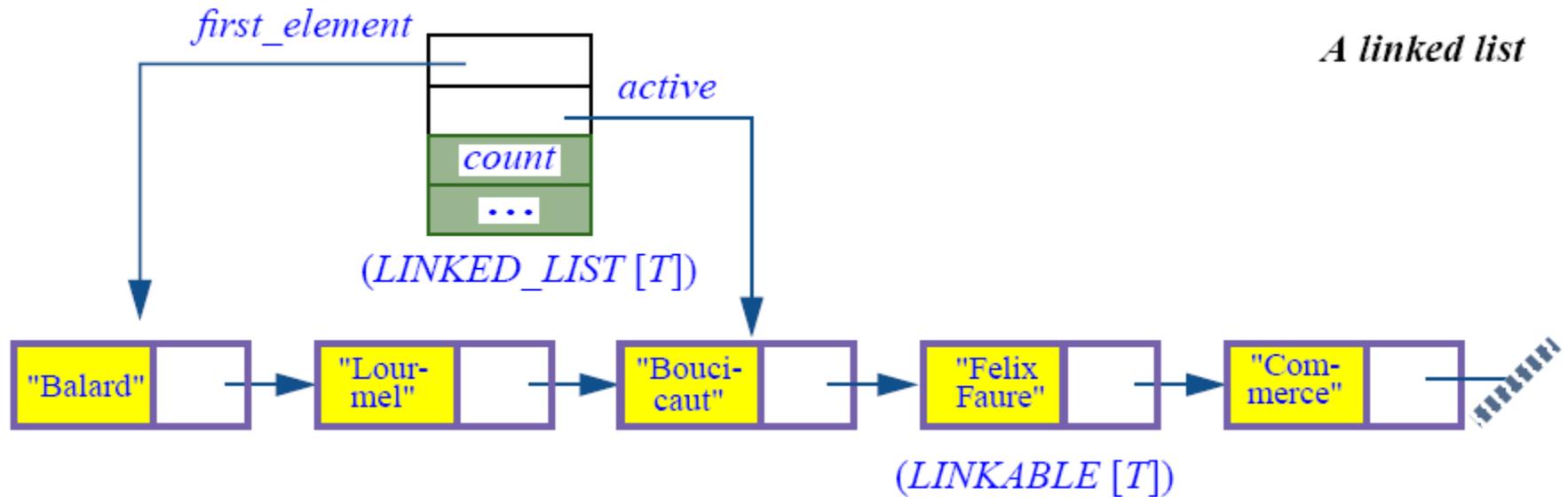
Der Cursor ist auf Position $count + 1$ gdw *after* wahr ist:

$$after = (index = count + 1)$$

In einer **leeren Liste** ist der Cursor auf Position 0 oder 1:

$$is_empty \text{ implies } ((index = 0) \text{ or } (index = 1))$$

Eine spezifische Implementation: (einfach) verkettete Liste

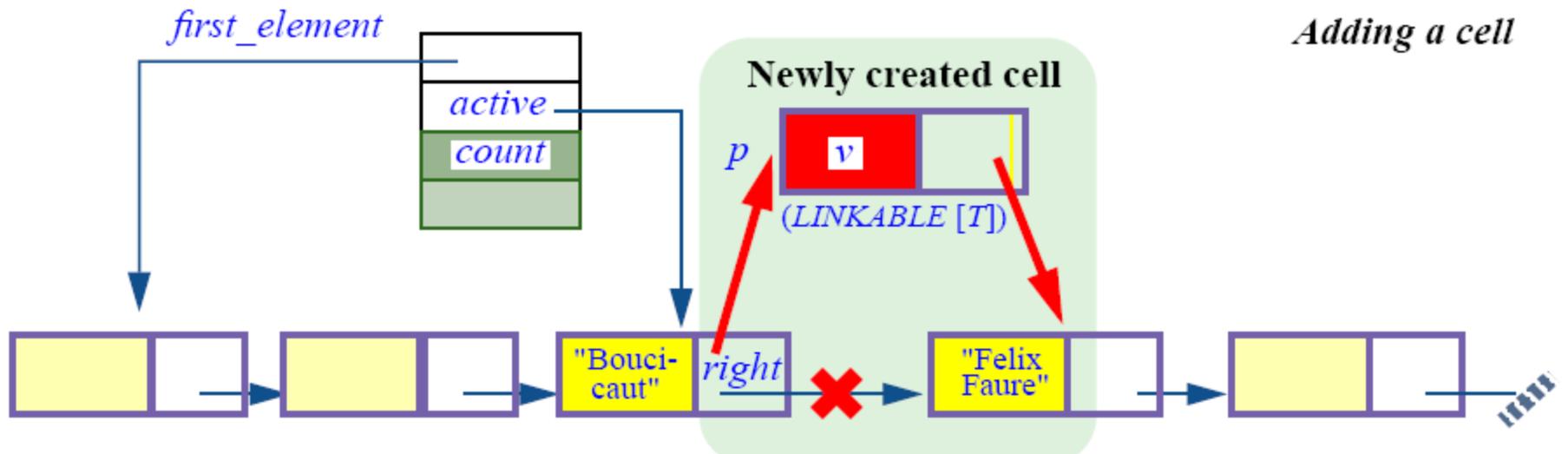




Vergessen sie auf keinen Fall die Grenzfälle, wenn Sie eine Container-Datenstruktur und die zugehörige Klasse definieren :

- Leere Struktur
- Volle Struktur (bei endlicher Kapazität)

Eine Zelle hinzufügen



Der dazugehörige Befehl



put_right(v: G)

-- v rechts der Cursorposition einfügen, Cursor nicht bewegen.

require

not_after: not after

local

p: LINKABLE[G]

do

create *p.make(v)*

if *before* **then**

p.put_right(first_element)

first_element := p

active := p

else

p.put_right(active.right)

active.put_right(p)

end

count := count + 1

ensure

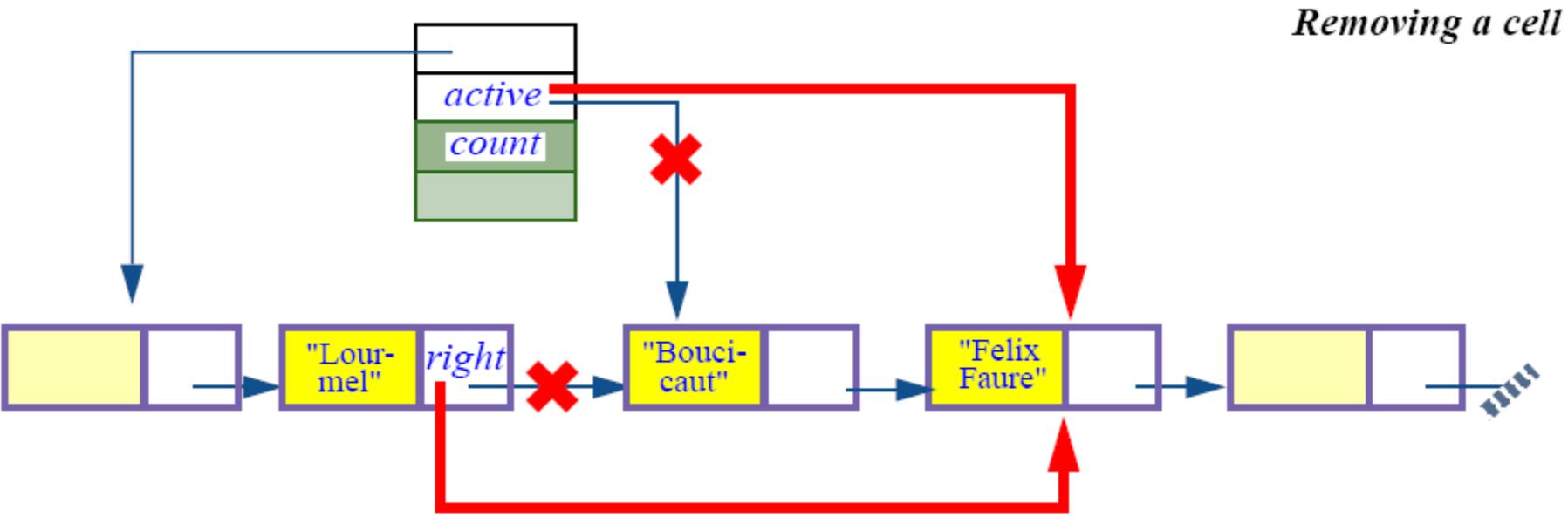
next_exists: active.right != Void

inserted: (not old before) implies active.right.item = v

inserted_before: (old before) implies active.item = v

end

Eine Zelle löschen

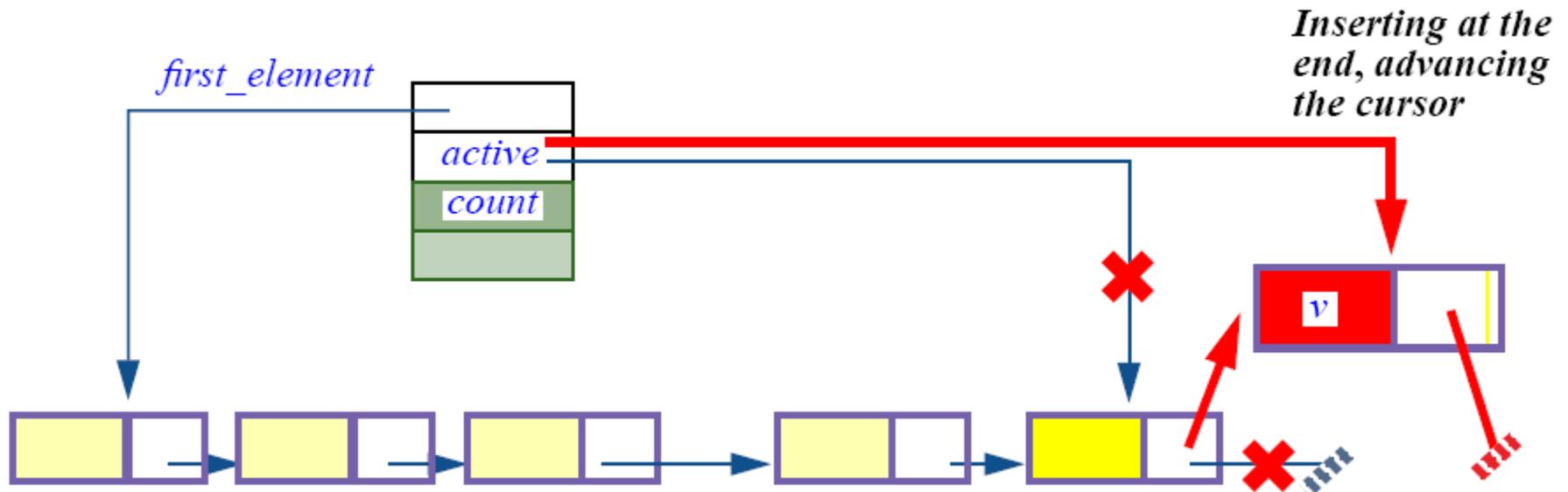


Der zugehörige Befehl

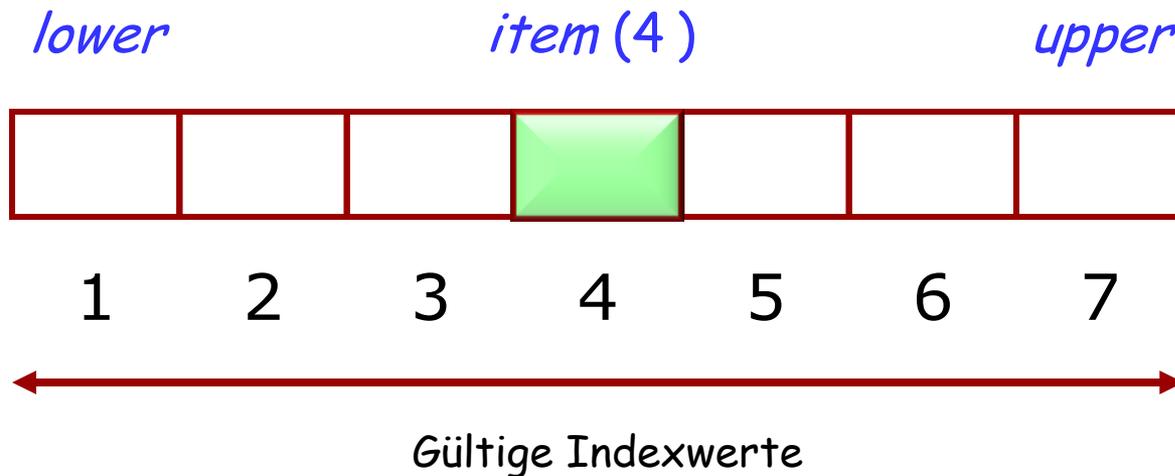


Entwerfen Sie *remove* als Übung!

Am Ende einfügen: *extend*



Ein Array ist ein Container-Speicher, der Elemente in zusammenhängende Speicherstellen speichert, wobei jede solche durch einen Integer-Index identifiziert wird.





Arrays haben Grenzen:

lower: INTEGER

-- Minimaler Index.

upper: INTEGER

-- Maximaler Index.

Die Kapazität eines Arrays ist durch die Grenzen bestimmt:

$$capacity = upper - lower + 1$$



item (i: INTEGER) : G

-- Eintrag an Stelle *i*, sofern im Index-Intervall.

require

valid_key: valid_index (i)

put (v: G; i: INTEGER)

-- Ersetze *i*-ten Eintrag, sofern im Index-Intervall,
-- durch *v*.

require

valid_key: valid_index (i)

ensure

inserted: item (i) = v

$i \geq \text{lower}$ and $i \leq \text{upper}$



Das Feature *item* ist wie folgt deklariert:

```
item (i: INTEGER) alias "[ ]": G assign put
```

Dies ermöglicht folgende synonyme Notationen:

$a[i]$	für	$a.item(i)$
$a.item(i) := x$	für	$a.put(x, i)$
$a[i] := x$	für	$a.put(x, i)$

Diese Vereinfachungen sind prinzipiell für alle Klassen möglich.

Eine Klasse darf maximal ein Feature mit Alias "[]" haben.

Die Grösse eines Arrays ändern



Arrays haben immer eine fixe obere und untere Grenze, und deshalb auch eine fixe Kapazität.

Eiffel erlaubt - anders als die meisten Programmiersprachen - das Verändern der Grösse eines Arrays (*resize*).

Das Feature *force* verändert die Grösse eines Arrays: Im Unterschied zu *put* hat es keine Vorbedingung.

Das Verändern der Grösse bedingt meistens eine Neu-Allokation des Arrays und das Kopieren der alten Werte. Solche Operationen sind teuer!

Einen Array benutzen, um eine Liste zu repräsentieren.



Siehe Klasse *ARRAYED_LIST* in EiffelBase

Einführung des Features *count* (Anzahl der Elemente in der Liste.)

Die Anzahl der Listenelemente reicht von 0 bis *capacity*:

$$0 \leq \textit{count} \leq \textit{capacity}$$

Eine leere Liste hat keine Elemente:

$$\textit{is_empty} = (\textit{count} = 0)$$

LINKED_LIST oder *ARRAYED_LIST*?



Die Wahl der Datenstruktur hängt von der Geschwindigkeit ihrer Containeroperationen ab.

Die Geschwindigkeit einer Containeroperation hängt von ihrer Implementation und dem zugrundeliegenden Algorithmus ab.

Wie schnell ist ein Algorithmus?



Abhängig von der Hardware, dem Betriebssystem, der Auslastung der Maschine...

Aber am meisten hängt die *Geschwindigkeit* vom Algorithmus selbst ab!

Komplexität eines Algorithmus: die O-Notation



Sei n die Grösse einer Datenstruktur ($count$).

" f ist $O(g(n))$ "

heisst, dass es eine Konstante k gibt, so dass:

$$\forall n, |f(n)| \leq k |g(n)|$$

Definiert die Funktion nicht mithilfe einer exakten Formel, sondern durch Grössenordnungen, z.B.

$O(1)$, $O(\log count)$, $O(count)$, $O(count^2)$, $O(2^{count})$.

~~$7count^2 + 20count + 4$~~ ist $O(count^2)$

put_right von *LINKED_LIST*: $O(1)$

Unabhängig von der Anzahl Elementen in einer verketteten Liste: Es braucht nur konstante Zeit, um ein Element bei der Cursorposition einzufügen.

force von *ARRAY*: $O(\text{count})$

Im schlechtesten Fall wächst die Zeit für diese Operation proportional mit der Anzahl Elemente im Array.

Wieso konstante Faktoren ignorieren?



Betrachten wir Algorithmen mit Komplexitäten

$O(n)$

$O(n^2)$

$O(2^n)$

Nehmen Sie an, Ihr neuer PC (Weihnachten steht vor der Tür!) ist 1000 mal schneller als Ihre alte Maschine.

Wie viel grösser kann ein Problem sein, damit Sie es immer noch in einem Tag lösen können?



Wir sind interessiert an

- Der Leistung im schlechtesten Fall (worst case)
- Der Leistung im besten Fall (best case, eher selten)
- Der durchschnittlichen Leistung (benötigt statistische Verteilung)

Solange nicht explizit genannt beziehen wir uns in dieser Diskussion auf die Leistung im schlechtesten Fall.

Notation der unteren Grenze: $\Omega(n)$

Kosten der Operationen einer einfach verketteten Liste

Operation	Feature	Komplexität
Einfügen rechts vom Cursor	<i>put_right</i>	$O(1)$
Einfügen am Ende	<i>extend</i>	$O(count)$ $O(1)$
Rechten Nachbarn löschen	<i>remove_right</i>	$O(1)$
Element beim Cursor löschen	<i>remove</i>	$O(count)$
Index-basierter Zugriff	<i>i_th</i>	$O(count)$
Suchen	<i>has</i>	$O(count)$

Kosten der Operationen einer **doppelt verketteten Liste**



Operation	Feature	Komplexität
Einfügen rechts vom Cursor	<i>put_right</i>	$O(1)$
Einfügen am Ende	<i>extend</i>	$O(1)$
Rechten Nachbarn löschen	<i>remove_right</i>	$O(1)$
Element beim Cursor löschen	<i>remove</i>	$O(1)$
Index-basierter Zugriff	<i>i_th</i>	$O(count)$
Suchen	<i>has</i>	$O(count)$

Kosten der Operationen eines Arrays



Operation	Feature	Complexity
Index-basierter Zugriff	<i>item</i>	$O(1)$
Index-basierte Ersetzung	<i>put</i>	$O(1)$
Index-basierte Ersetzung ausserhalb der Grenzen	<i>force</i>	$O(count)$
Suchen	<i>has</i>	$O(count)$
Suchen in sortiertem Array	-	$O(\log count)$

Können wir die Effizienz von Arrays erreichen:

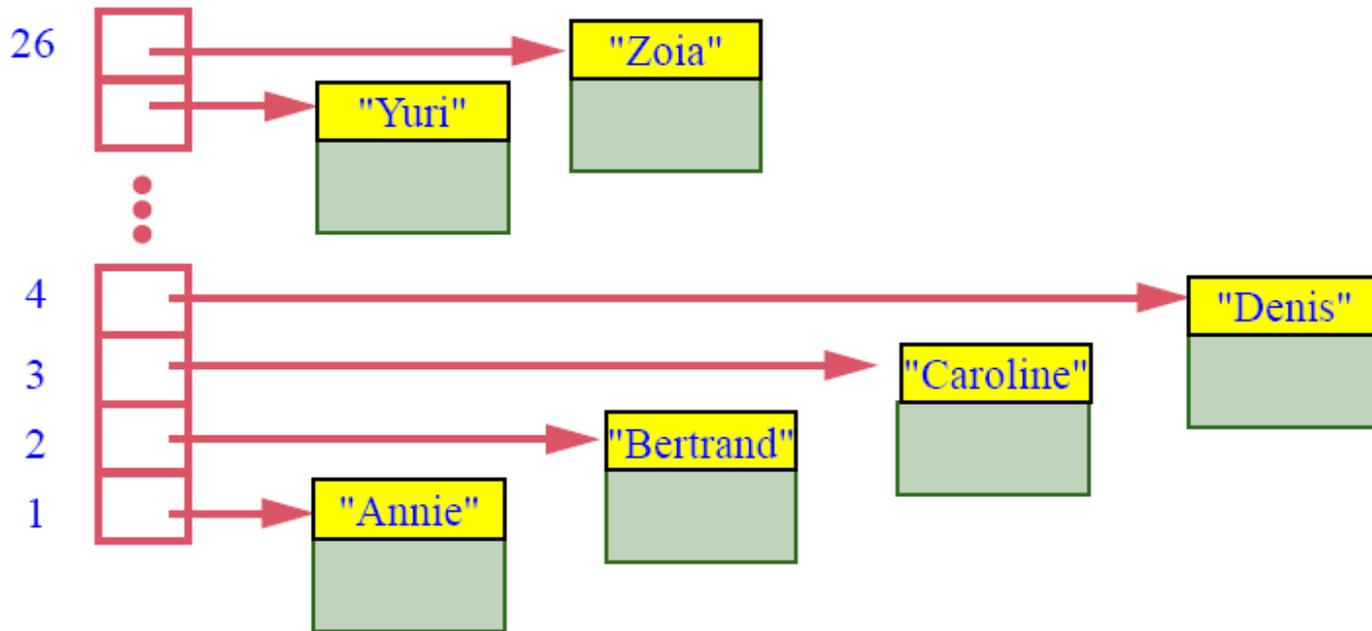
- Zeitlich konstanter Zugriff
- Zeitlich konstante Änderungen

... ohne uns dabei auf ganze Zahlen als Schlüssel zu beschränken?

Die Antwort: Hashtabellen (... jedenfalls beinahe)

Arrays und Hashtabellen sind beide indexierte Strukturen; Das Verändern eines Elements benötigt einen **Index** oder, im Fall von Hashtabellen, einen **Schlüssel** (*key*).

Im Unterschied zu Arrays sind bei Hashtabellen auch nicht-Integer als Schlüssel möglich.





Endlich kein Mundgeruch
So können Sie Ihren Mundgeruch auf natürlichem Weg beseitigen.
www.fangocur.at/Mundgeruch
Ads by Google

TinyURL.com

Making long URLs usable! More than 400 million of them. Over 2 billion hits/month.

[Home](#)

[Example](#)

[Make Toolbar Button](#)

[Redirection](#)

[Hide URLs](#)

[Preview Feature](#)^{cool!}

[Link to Us!](#)

[Terms of use](#)

[Contact Us!](#)



Cool Sites

- [CoolWhois.com](#)
- [Unicyclist Community](#)
- [Gilby.com](#)
- [MagicBounce Party Rentals](#)

TinyURL was created!

The following URL:

```
http://www.amazon.com/Touch-Class-Learning-Program-Contracts  
/dp/3540921443/ref=sr_1_1?ie=UTF8&s=books&qid=1238000471&sr=  
8-1
```

has a length of 123 characters and resulted in the following TinyURL which has a length of 25 characters:

```
http://tinyurl.com/dmtk6s  
\[Open in new window\]
```

Or, give your recipients confidence with a preview TinyURL:

```
http://preview.tinyurl.com/dmtk6s  
\[Open in new window\]
```

This TinyURL may have been copied to your clipboard. (This no longer works for those who have upgraded to Flash 10.) To paste it in a document, press and hold down the ctrl key (command key for Mac users) while pressing the V key, or choose the "paste" option from the edit menu.

Enter another long URL to make tiny:

Custom alias (optional):

May contain letters, numbers, and dashes.

Der Gebrauch von Hashtabellen



```
persone, persone1: PERSONE  
personal_verzeichnis: HASH_TABLE [PERSONE, STRING]
```

```
create personal_verzeichnis.make (100000)
```

Ein Element speichern:

```
create persone1
```

```
personal_verzeichnis.put (persone1, "Annie")
```

Ein Element abfragen:

```
persone := personal_verzeichnis.item ("Annie")
```

Eingeschränkte Generik und die Klassenschnittstelle



class

HASH_TABLE[*G*, *K* -> *HASHABLE*]

Erlaubt *h*["ABC"] für *h.item*("ABC")

Erlaubt *h.item*["ABC"] := *x*
für *h.put*(*x*, "ABC")

feature

item **alias** "[" "]" (*key*: *K*): *G* **assign force**

Zusammen: Erlauben *h*["ABC"] := *x*
für *h.put*(*x*, "ABC")

put(*new*: *G*; *key*: *K*)

- Füge *new* mit *key* ein, sofern kein anderes
- Element mit diesem Schlüssel existiert.

do ... end

force(*new*: *G*; *key*: *K*)

- Tabelle updaten, so dass *new* das Element
- ist, das mit *key* assoziiert wird.

...

end

Das Beispiel, neu geschrieben



```
persone, persone1: PERSONE  
personal_verzeichnis: HASH_TABLE[PERSONE, STRING]
```

```
create personal_verzeichnis.make(100000)
```

Ein Element speichern:

```
create persone1
```

```
personal_verzeichnis["Annie"] := persone1
```

Ein Element abfragen:

```
persone := personal_verzeichnis["Annie"]
```

Kein guter Stil - Wieso?



Die Hashfunktion bildet K , die Menge der möglichen Schlüssel, auf ein ganzzahliges Intervall $a..b$ ab.

Eine **perfekte** Hashfunktion ergibt für jedes Element von K einen anderen Integer.

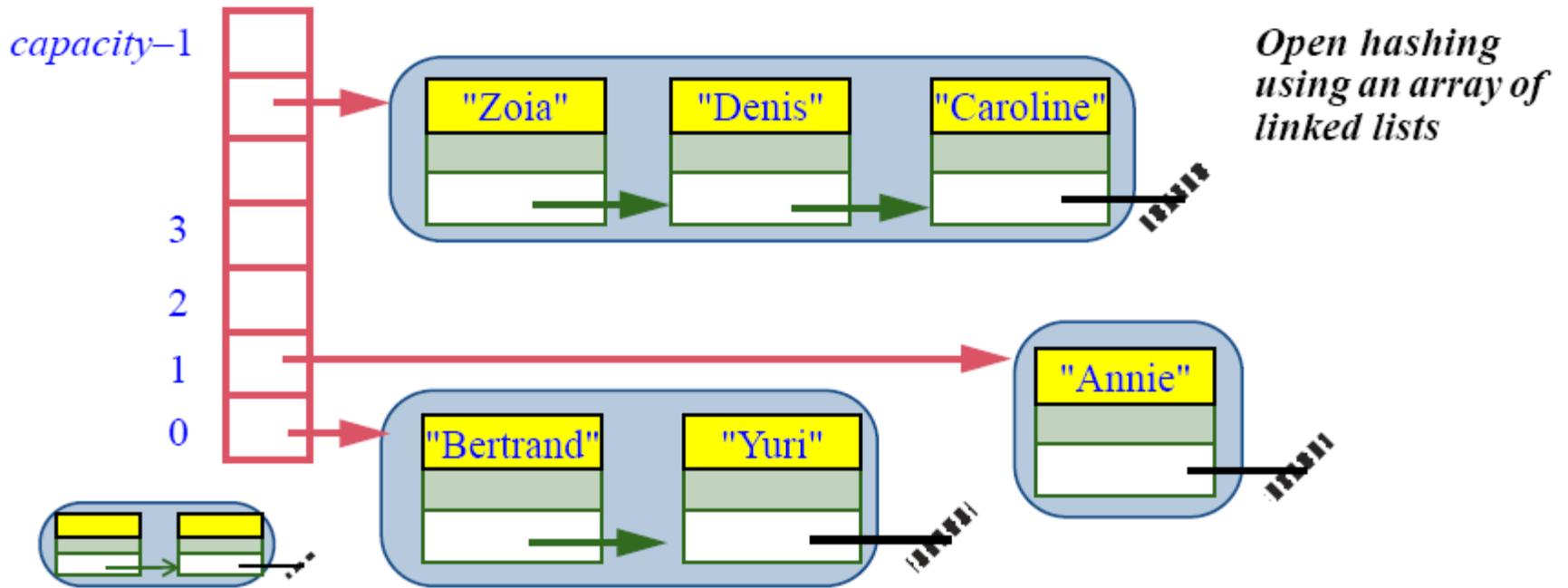
Immer, wenn zwei unterschiedliche Schlüssel denselben Hashwert ergeben, entsteht eine Kollision.

Behandlung von Kollisionen



Offenes Hashing:

ARRAY[LINKED_LIST[G]]

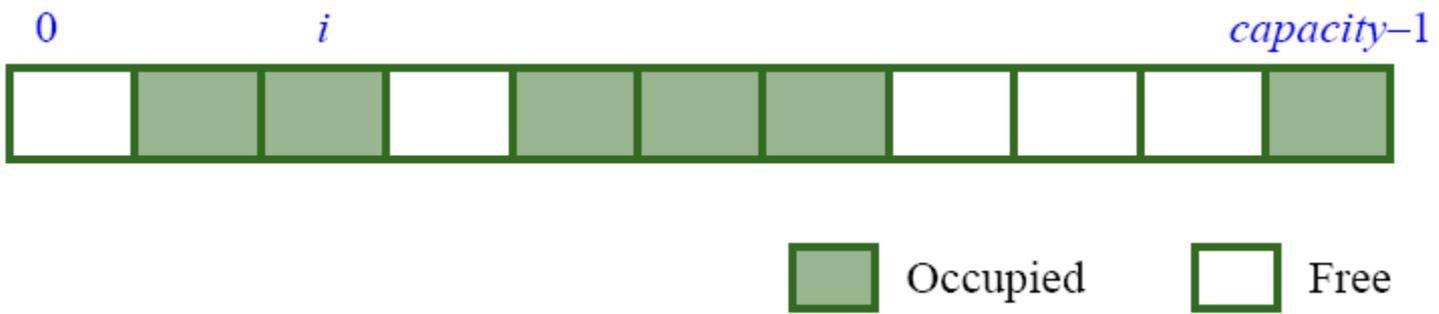


Eine andere Technik: Geschlossenes Hashing



Die Klasse *HASH_TABLE*[*G*, *H*] implementiert geschlossenes Hashing:

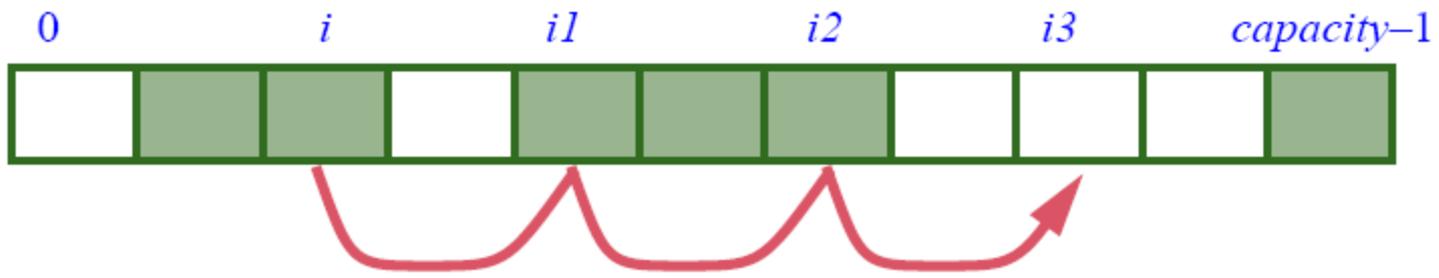
HASH_TABLE[*G*, *H*] benutzt einen einzigen *ARRAY*[*G*], um die Elemente zu speichern. Zu jeder Zeit sind einige Positionen frei und einige besetzt:



Geschlossenes Hashing



Falls die Hashfunktion eine bereits besetzte Position ergibt, wird der Mechanismus probieren, eine Folge von anderen Positionen (i_1, i_2, i_3) zu erreichen, bis er eine freie Stelle findet.



Mit dieser Richtlinie und einer guten Wahl der Hashfunktion ist das Suchen und das Einfügen in Hashtabellen in $O(1)$ erreichbar.

Kosten der Operationen einer Hashtabelle



Operation	Feature	Komplexität
Schlüsselbasierter Zugriff	<i>item</i>	$O(1)$ $O(\text{count})$
Schlüsselbasierte Einfügung	<i>put, extend</i>	$O(1)$ $O(\text{count})$
Löschung	<i>remove</i>	$O(1)$ $O(\text{count})$
Schlüsselbasiertes Ersetzen	<i>replace</i>	$O(1)$ $O(\text{count})$
Suchen	<i>has</i>	$O(1)$ $O(\text{count})$

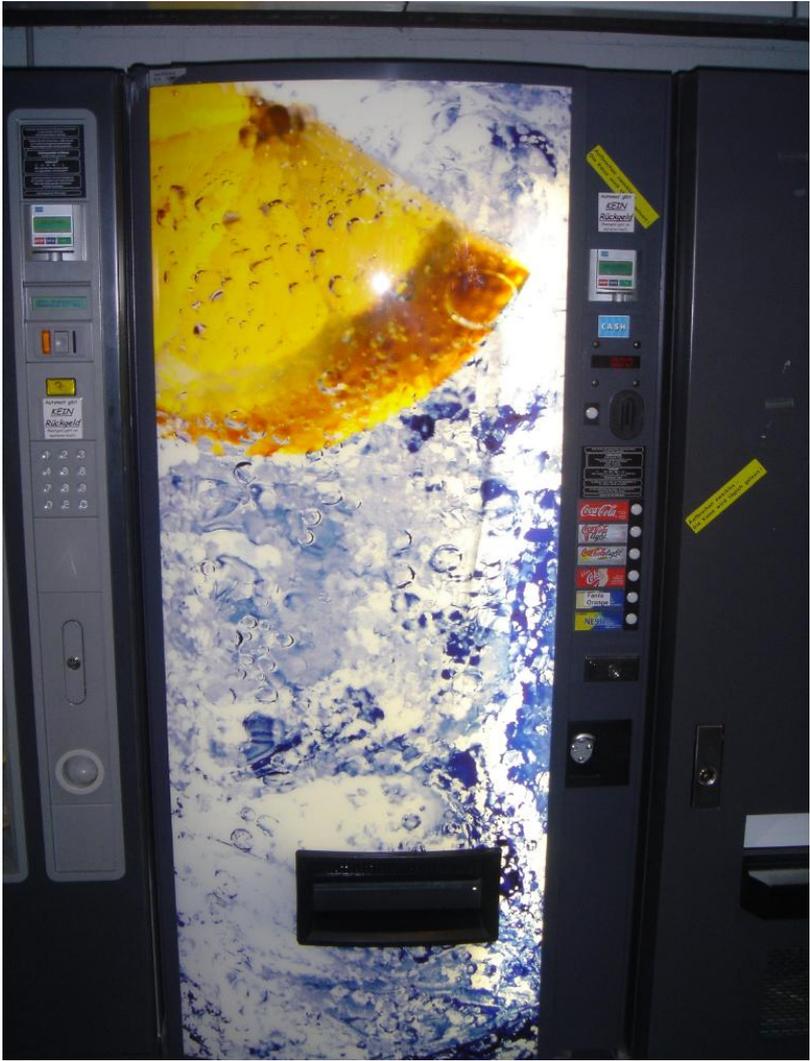


Im Unterschied zu indexierten Strukturen wie Arrays und Hashtabellen gibt es keinen Schlüssel oder andere identifizierende Informationen für Elemente von Dispensern.

Dispenser sind Container-Datenstrukturen, die eine spezifische Ausgaberegeln besitzen:

- **Last In First Out (LIFO)**: Wähle das Element, das zuletzt eingefügt wurde → **Stack**.
- **First In First Out (FIFO)**: Wähle das älteste, noch nicht entfernte Element → **Queue**.
- **Vorrangwarteschlange (priority queue)**: Wähle das Element mit der höchsten Priorität.

Dispenser



Ein Stack ist ein Dispenser, der die LIFO-Regel anwendet.
Die grundlegenden Operationen sind:

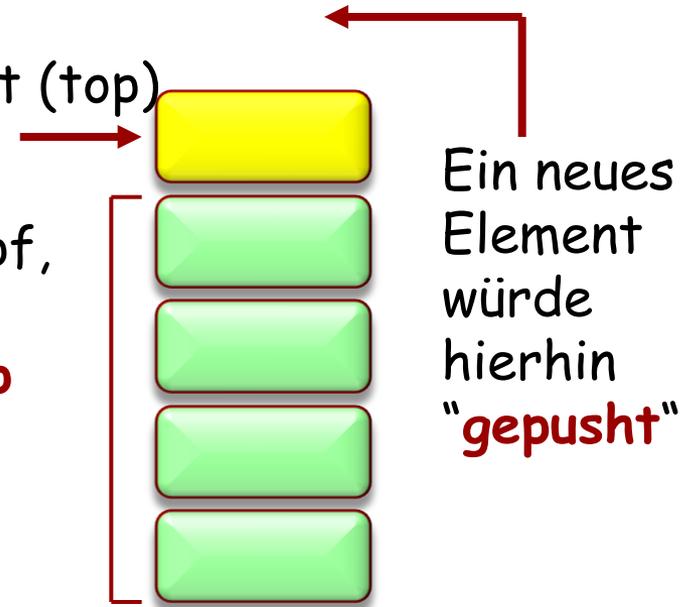
Ein Element auf den Stack

"drücken" (*put*) Oberstes Element (top)

Das oberste Element
wegnehmen (*remove*)

Zugriff auf das oberste
Element (*item*)

Der Rumpf,
der nach
einem **pop**
übrig
bleiben
würde.





Viele!

Allgegenwärtig in der Implementation von Programmiersprachen:

- Parsen von Ausdrücken (bald)
- Ausführung von Routinen managen ("DER Stack")
Spezialfall: **Rekursion** implementieren
- Bäume traversieren
- ...



from
until

"Alle Terme der Polnischen Notation wurden gelesen"

loop

"Lese nächsten Term in Polnischer Notation"

if "x ist ein Operand" then

s.put(x)

else -- x ist ein binärer Operator

-- Erhalte die beiden Operanden:

op1 := s.item; s.remove

op2 := s.item; s.remove

-- Wende Operator auf Operanden an und «drücke»

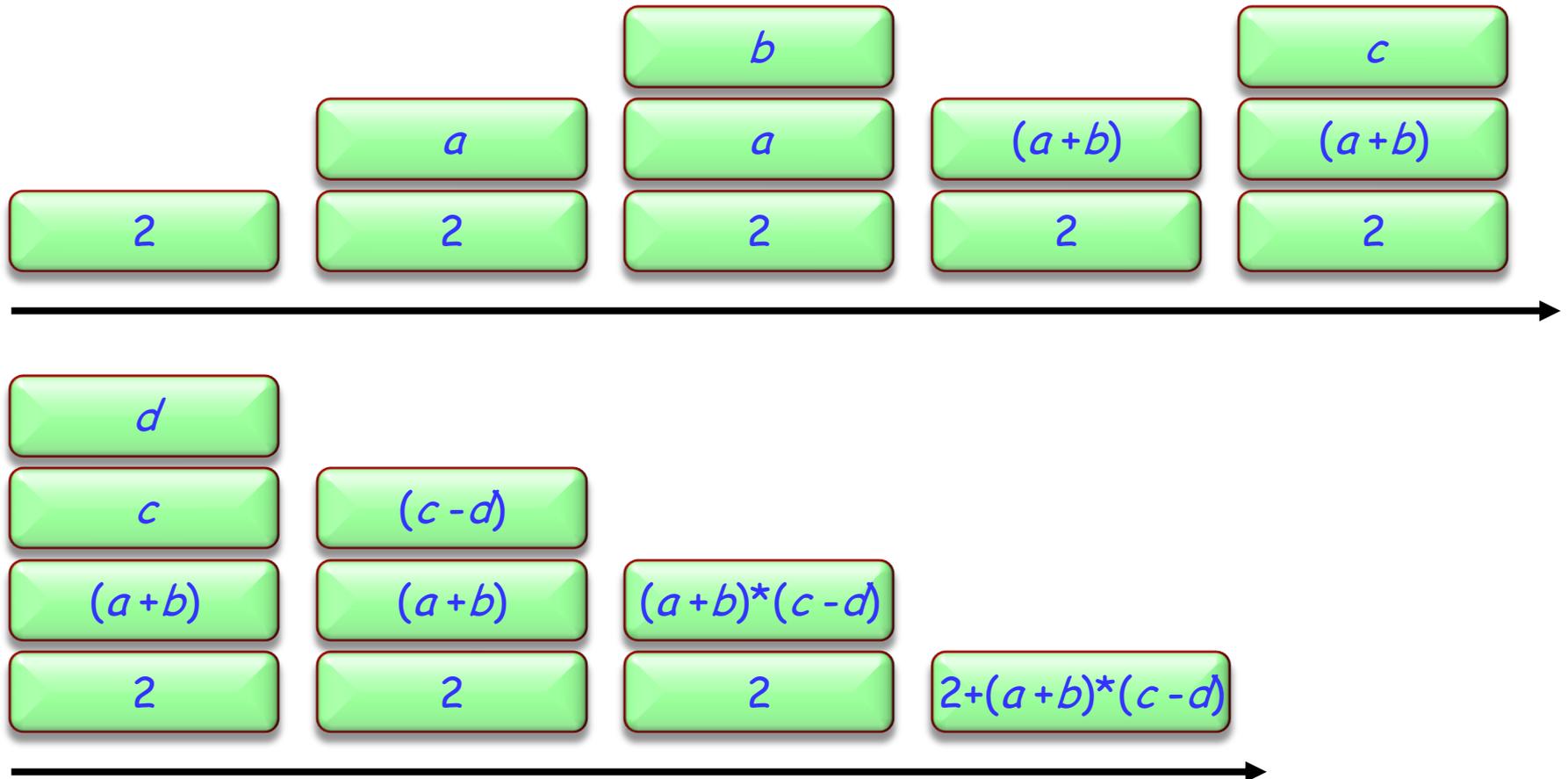
-- das Resultat:

s.put(application(x, op2, op1))

end

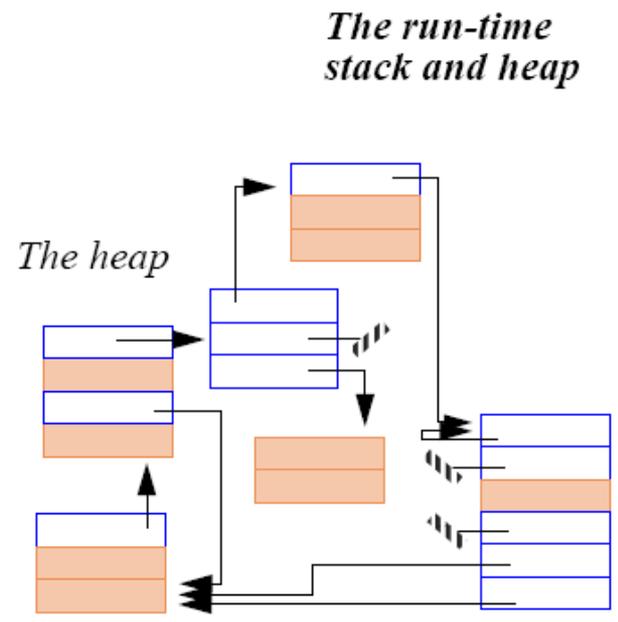
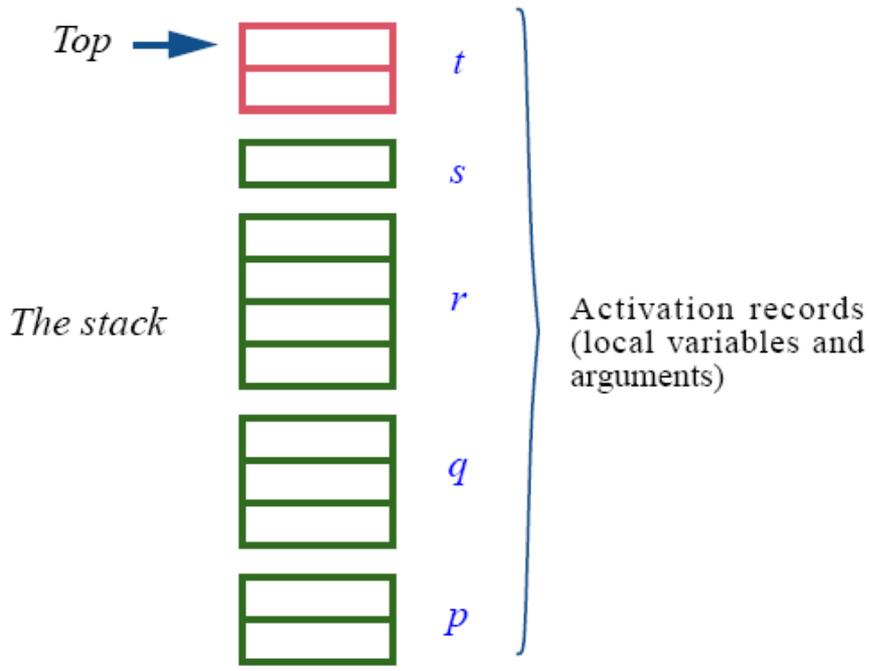
end

Auswertung von $2 a b + c d - * +$



Der Laufzeit-Stack enthält Aktivierungseinträge für alle zur Zeit aktiven Routinen.

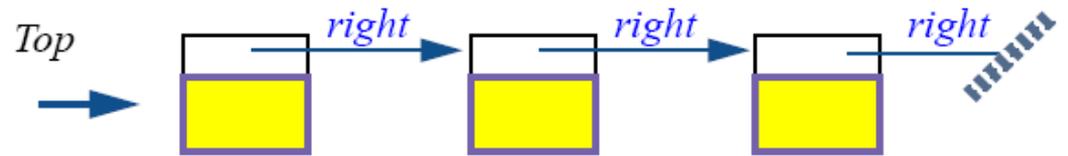
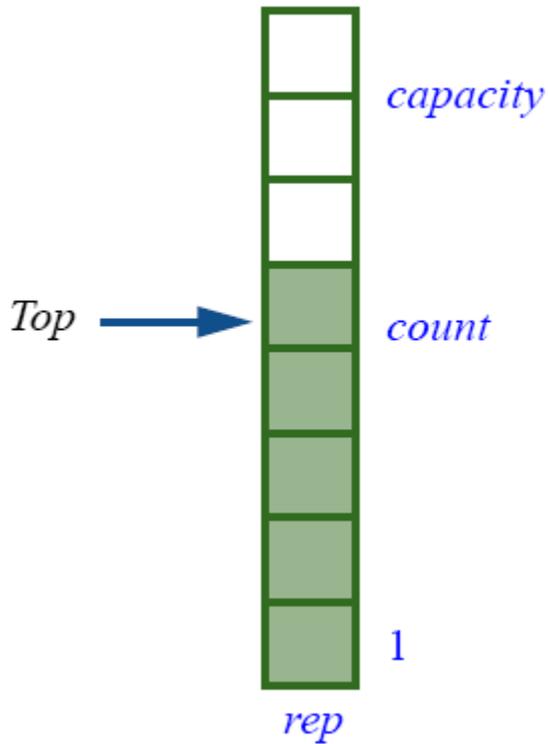
Ein Aktivierungseintrag beinhaltet die lokalen Variablen einer Routine (Argumente und lokale Entitäten).



Stacks implementieren



Die häufigsten Implementierungen eines Stacks sind entweder verkettet oder indiziert (arrayed).





Benutzen Sie eine verkettete Liste, falls:

- Die Ordnung der Elemente wichtig ist
- Die meisten Zugriffe in dieser Ordnung erfolgen
- (Bonusbedingung) Kein festes Grössenlimit

Benutzen Sie einen Array, falls:

- Jedes Element mit einem Integer-Index identifiziert werden kann
- Die meisten Zugriffe über diesen Index erfolgen
- Feste Grössengrenze (zumindest für längere Ausführungszeit)

Benutzen Sie eine Hashtabelle, falls:

- Jedes Item einen entsprechenden Schlüssel hat.
- Die meisten Zugriffe über diese Schlüssel erfolgen.
- Die Struktur beschränkt ist.

Benutzen Sie einen Stack:

- Für eine LIFO-Regel
- Beispiel: Traversieren von verschachtelten Strukturen (z.B. Bäume).

Benutzen Sie eine Queue:

- Für eine FIFO-Regel
- Beispiel: Simulation eines FIFO-Phänomens.



Container-Datenstrukturen: Grundlegende Begriffe,
Schlüsselbeispiele

Ein wenig Komplexitätstheorie ("Big-O")

Wann welcher Container zu wählen ist