ETH Zürich
Chair of Software Engineering

Einführung in die Programmierung

Prof. Dr. Bertrand Meyer

Lecture 13: (Container-)Datenstrukturen

Themen dieser Lektion

Container und Generizität

Container-Operationen

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

Wichtige Datenstrukturen:

- Listen
- Arrays
- Hashtabellen
- Stapelspeicher (*Stacks*, fortan Stapel)
- Warteschlangen (*Queues*)

Container-Datenstrukturen

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

Fundamentale Operationen (siehe nächste Folie):

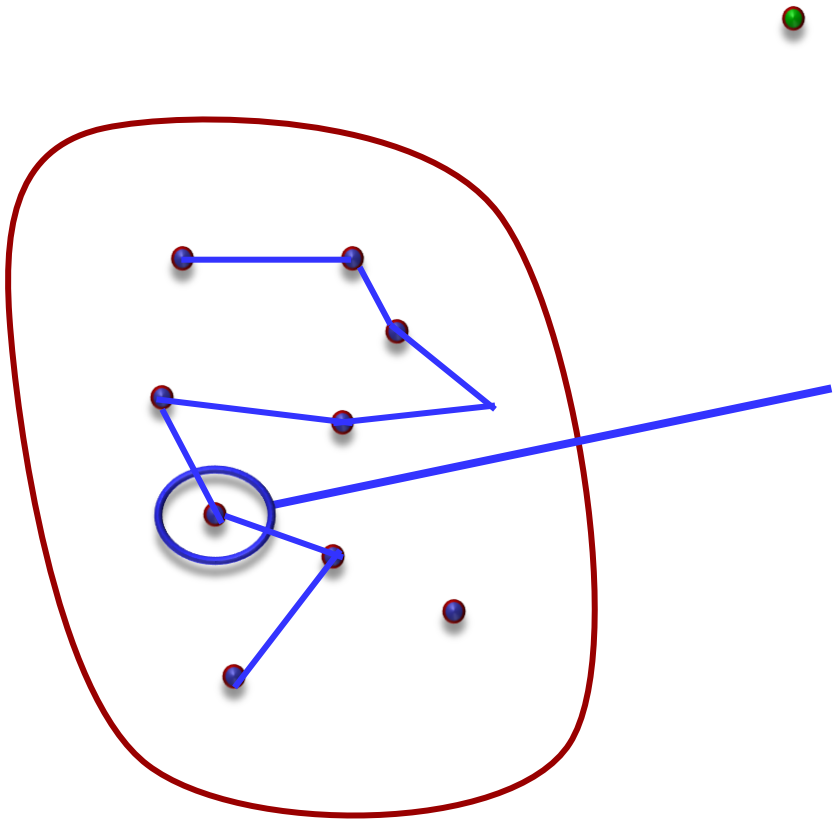
- Einfügen: Ein Element hinzufügen
- Löschen: Ein Vorkommen eines Elements (falls vorhanden) löschen
- Ausradieren (Wipeout): Alle Vorkommen eines Elements löschen
- Suchen: Ist ein bestimmtes Element vorhanden?
- Traversierung/Iteration: Eine gegebene Operation auf jedes Element anwenden

Die Implementation eines Containers bestimmt:

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

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

Container-Operationen



- 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

Container-Datenstrukturen

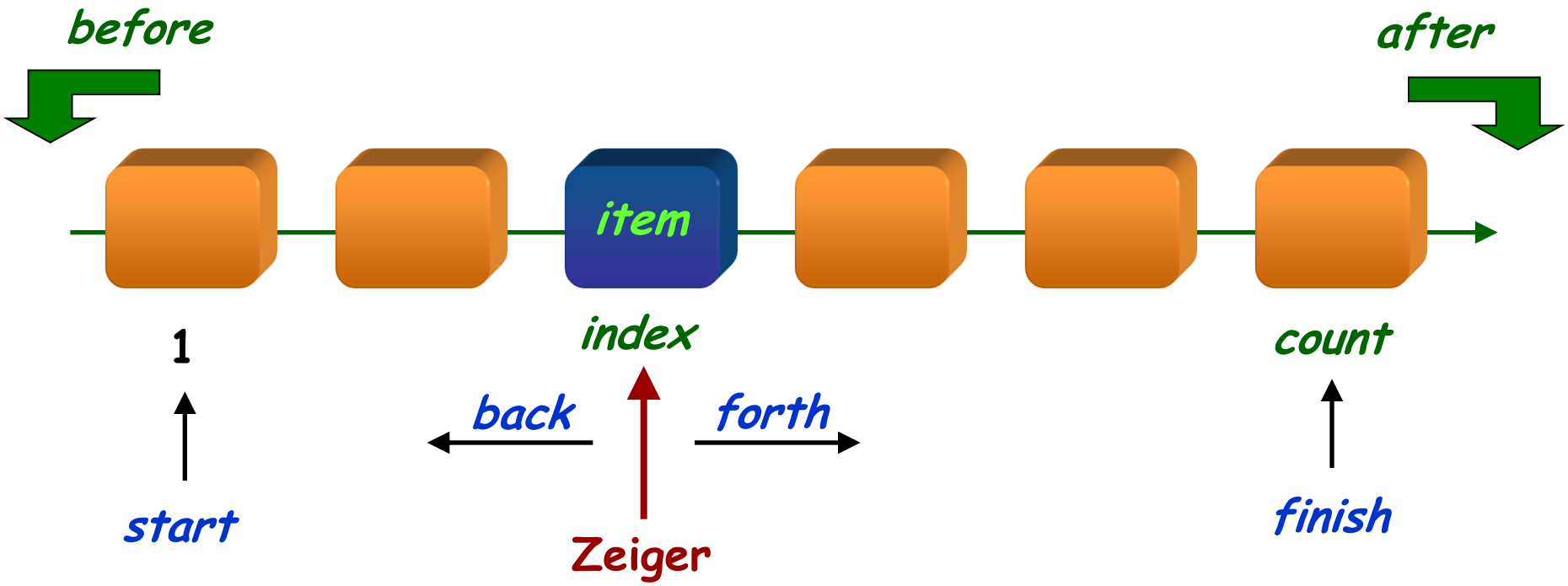
Fundamentale Operationen:

- 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:

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

Ein bekannter Container: Die Liste



Abfragen
Befehle

Um die Iteration und andere Operationen zu vereinfachen haben unsere Listen Zeigern (intern oder extern)

Ein standardisiertes Namensschema

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
```

Stilregel: wenn angebracht, benutzen auch Sie diese Namen in Ihren eigenen Klassen

Begrenzte Repräsentationen

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

Container und Generizität

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

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

Containerstrukturen sind als generische Klassen implementiert:

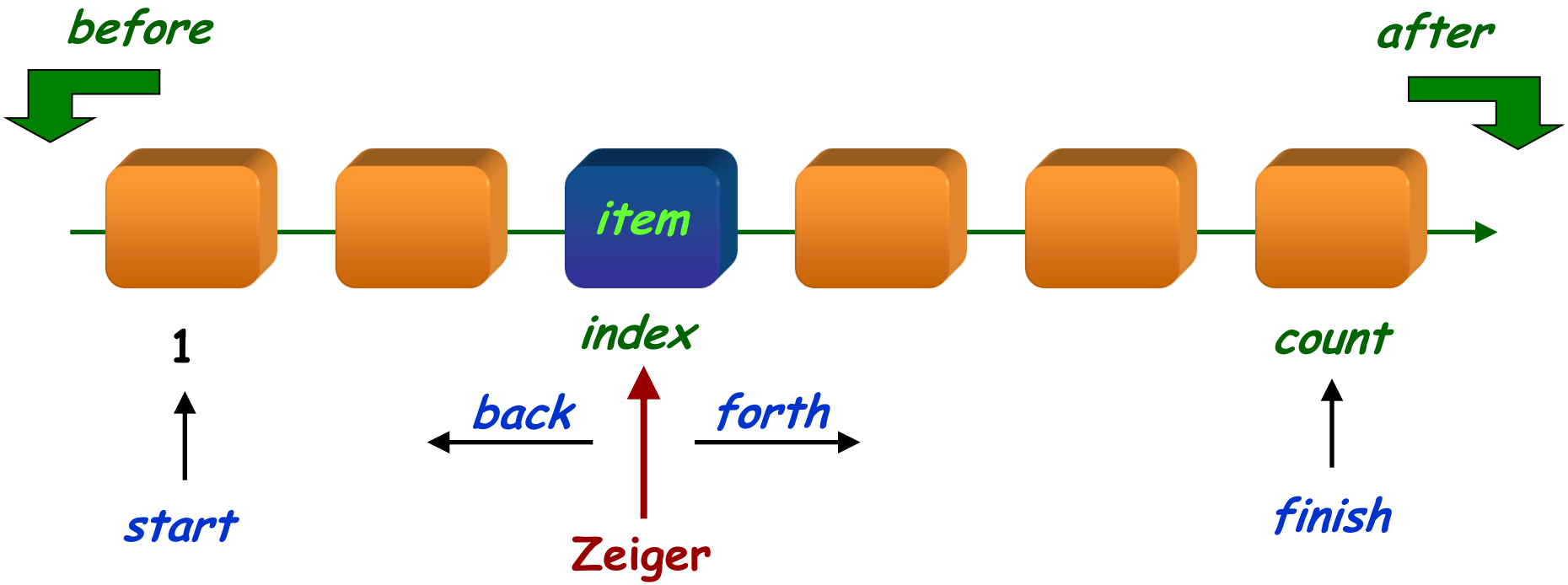
LINKED_LIST [G]

pl : LINKED_LIST [PERSON]
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 Zeigern (*cursors*)



Zeiger-Eigenschaften (Alle in der Klasseninvariante!)

Der Zeiger reicht von 0 bis *count* + 1:

$$0 <= index <= count + 1$$

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

$$before = (index = 0)$$

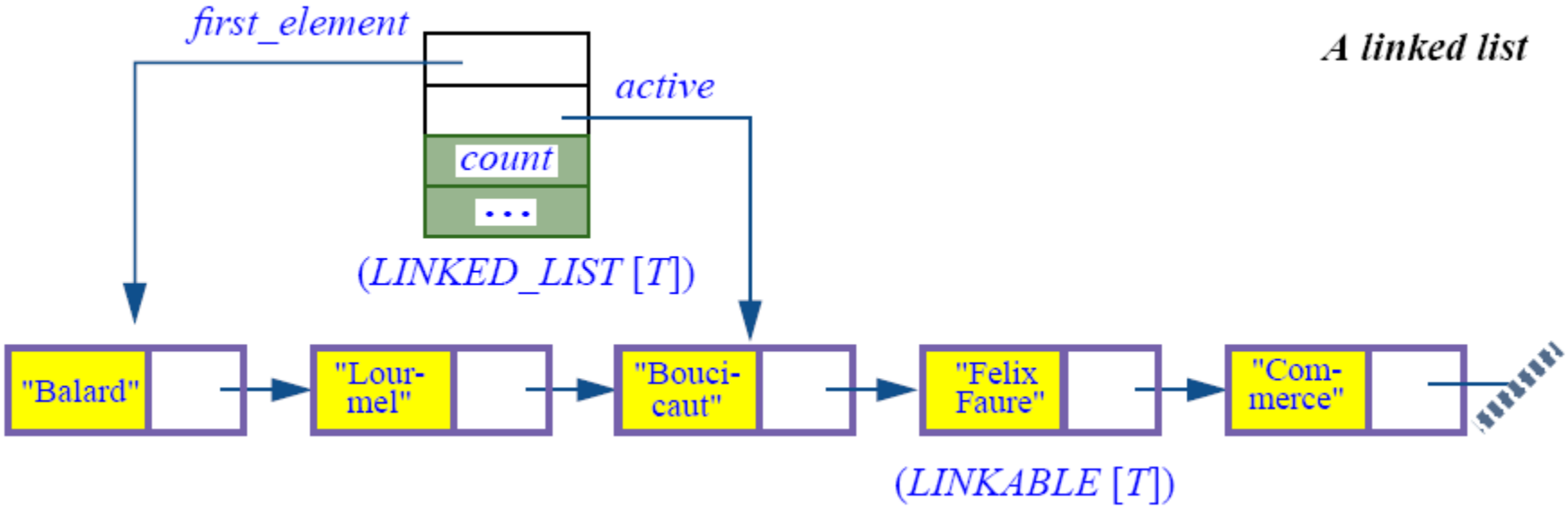
Der Zeiger ist auf Position count + 1 gdw *after* wahr ist:

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

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

is_empty **implies** ((*index* = 0) **or** (*index* = 1))

Eine spezifische Implementation: (einfach) verkettete Liste

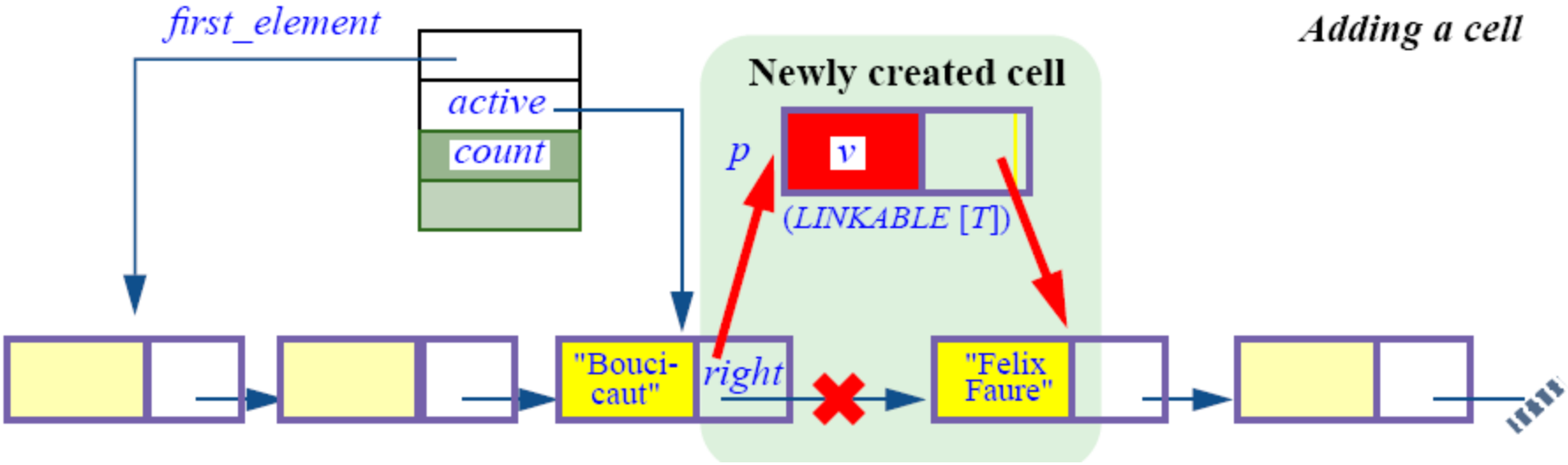


Warnung

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

- Leere Struktur
- Volle Struktur (bei endlicher Kapazität)

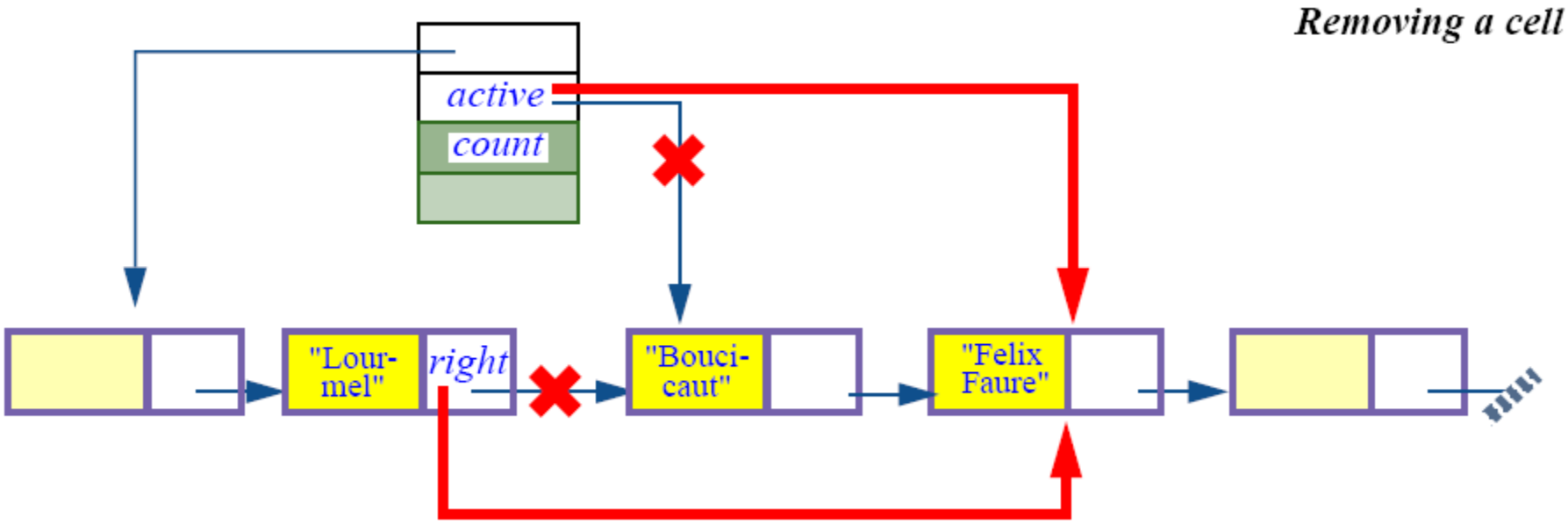
Eine Zelle hinzufügen



Der dazugehörige Befehl (in *LINKED_LIST*)

```
put_right (v : G)
            -- v rechts der Zeigerposition einfügen, Zeiger 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
```

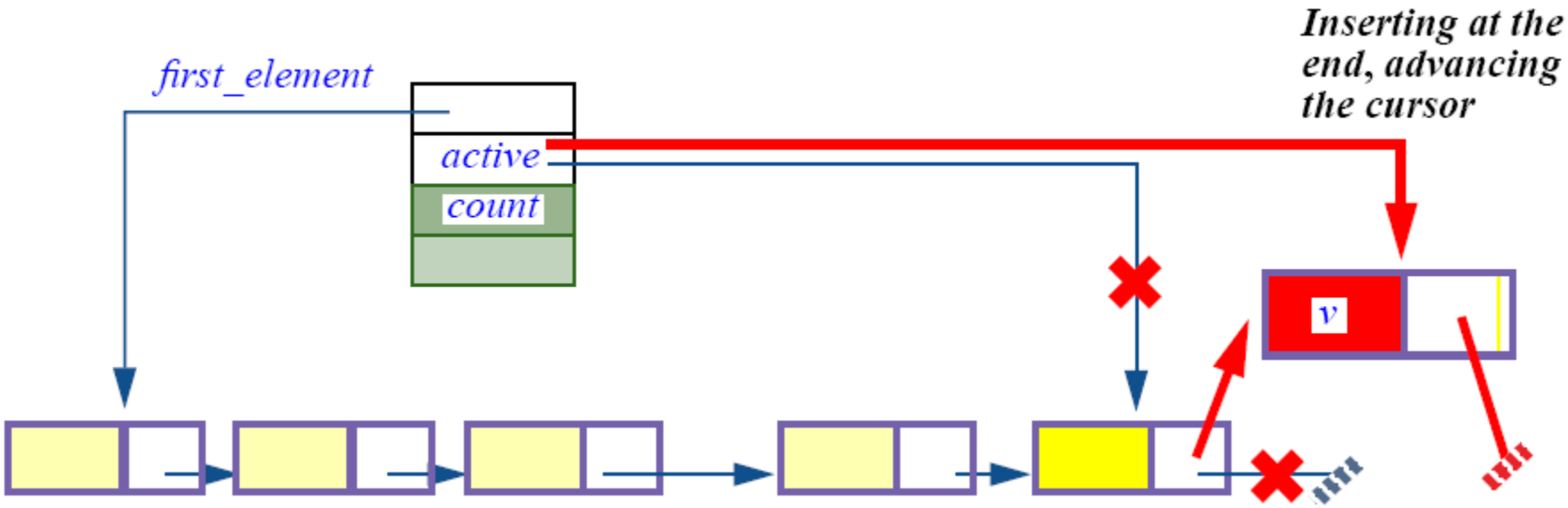
Eine Zelle löschen



Der zugehörige Befehl

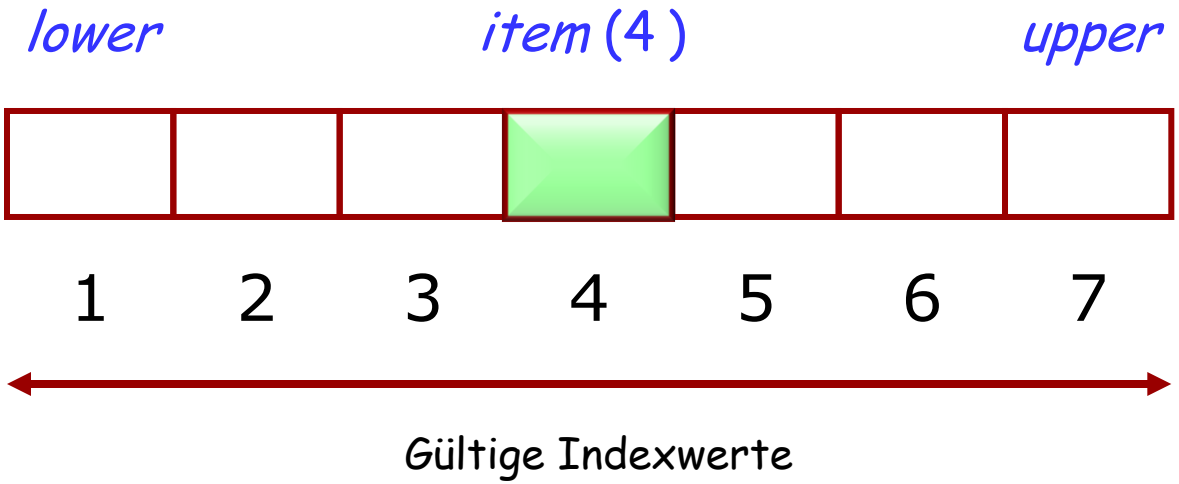
Entwerfen Sie *remove* als Übung!

Am Ende einfügen: *extend*



Arrays

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



Grenzen und Indizes

Arrays haben Grenzen:

```
lower: INTEGER
        -- Minimaler Index.
upper: INTEGER
        -- Maximaler Index.
```

Die Kapazität eines Arrays ist durch die Grenzen bestimmt:

```
capacity = upper - lower + 1
```

Auf Elemente eines Arrays zugreifen und diese verändern

```
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 >= lower and i <= upper

Bemerkung zu Eiffel: Vereinfachung der Notation

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 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 <= *count* <= *capacity*

Eine leere Liste hat keine Elemente:

is_empty = (*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.

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

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

Beispiele

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 Zeigerposition einzufügen

force von *ARRAY*: **O** (count)

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

(Erinnerung) *put_right* in *LINKED_LIST*

```
put_right (v : G)
            -- v rechts der Zeigerposition einfügen, Zeiger 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
```

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?

Varianten der Algorithmenkomplexitäten

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 anders erwähnt, beziehen wir uns in dieser Diskussion auf die Leistung im schlechtesten Fall

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

Für beide Grenzen: $\Theta(f(n))$

(wir benutzen $O(f(n))$ für Einfachkeit)

Kosten der Operationen einer einfach verketteten Liste

Operation	Feature	Komplexität
Einfügen rechts vom Zeiger	*put_right*	O (1)
Einfügen am Ende	*extend*	O (*count*) O (1)
Rechten Nachbarn löschen	*remove_right*	O (1)
Element beim Zeiger löschen	*remove*	O (*count*)
Index-basierter Zugriff	*i_th*	O (*count*)
Suchen	*has*	O (*count*)

Kosten der Operationen einer doppelt verketteten Liste

Operation	Feature	Komplexität
Einfügen rechts vom Zeiger	*put_right*	O (1)
Einfügen am Ende	*extend*	O (1)
Rechten Nachbarn löschen	*remove_right*	O (1)
Element beim Zeiger löschen	*remove*	O (1)
Index-basierter Zugriff	*i_th*	O (*count*)
Suchen	*has*	O (*count*)

Kosten der Operationen eines Arrays

Operation	Feature	Complexity
Index-basierter Zugriff	*item*	O (1)
Index-basierte Ersetzung	*put*	O (1)
Index-basierte Ersetzung ausserhalb der Grenzen	*force*	O (*count*)
Suchen	*has*	O (*count*)
Suchen in sortiertem Array	-	O (log *count*)

Hashtabellen

Können wir die Effizienz von Arrays erreichen:

- Zeitlich konstanter Zugriff
- Zeitlich konstante Änderungen

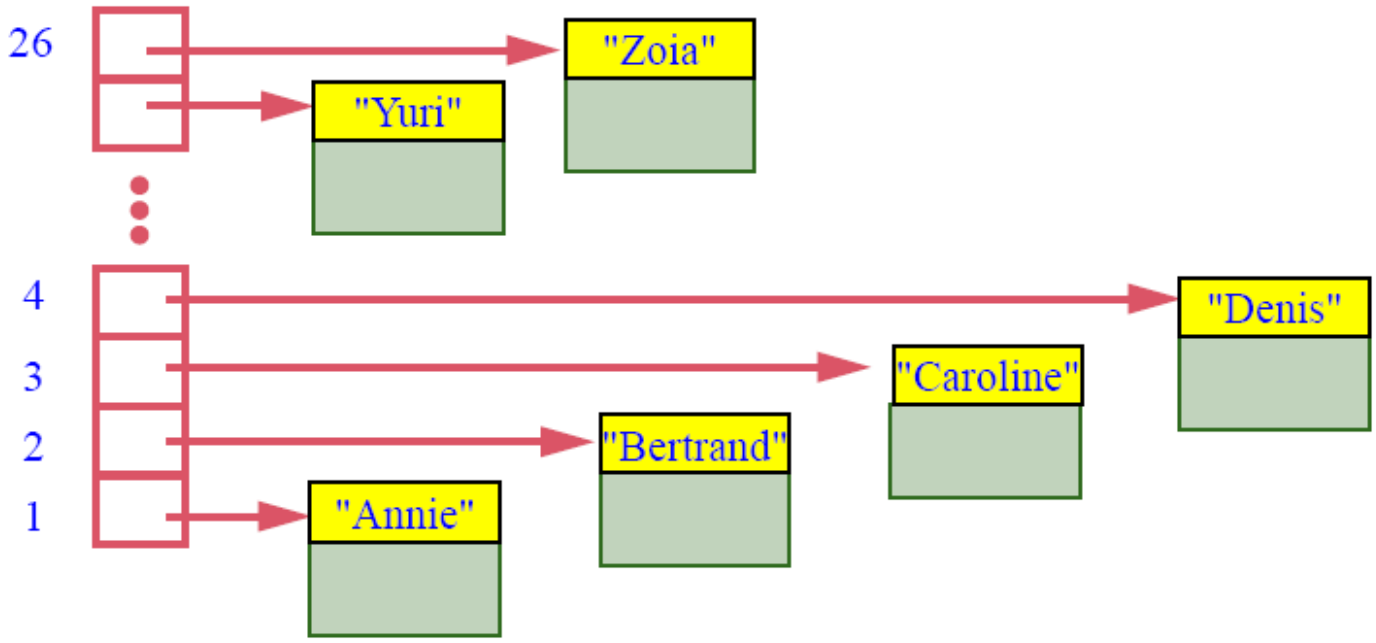
… ohne uns dabei auf ganze Zahlen als Schlüssel zu beschränken?

Die Antwort: Hashtabellen (… jedenfalls beinahe)

Hashtabellen

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.



Eine Abbildung (mapping)

Der Gebrauch von Hashtabellen

```
person, person1 : PERSON
personal_verzeichnis : HASH_TABLE [PERSON, STRING]

create personal_verzeichnis.make (100000)
```

Ein Element speichern:

```
create person1

personal_verzeichnis.put (person1, "Annie")
```

Ein Element abfragen:

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

Eingeschränkte Generizität und die Klassenschnittstelle

```
class
    HASH_TABLE [G, K -> HASHABLE]
feature
    item alias "[]" (key: K): G assign force

    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 aktualisieren, so dass new das Element
                -- ist, das mit key assoziiert wird.
        ...
end
```

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

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

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

Das Beispiel, neu geschrieben

```
person, person1 : PERSON
personal_verzeichnis : HASH_TABLE [PERSON, STRING]

create personal_verzeichnis.make (100000)
```

Kein guter Stil - Wieso?

Ein Element speichern:

```
        create person1

        personal_verzeichnis ["Annie"] := person1
```

Ein Element abfragen:

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

Die Hashfunktion

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

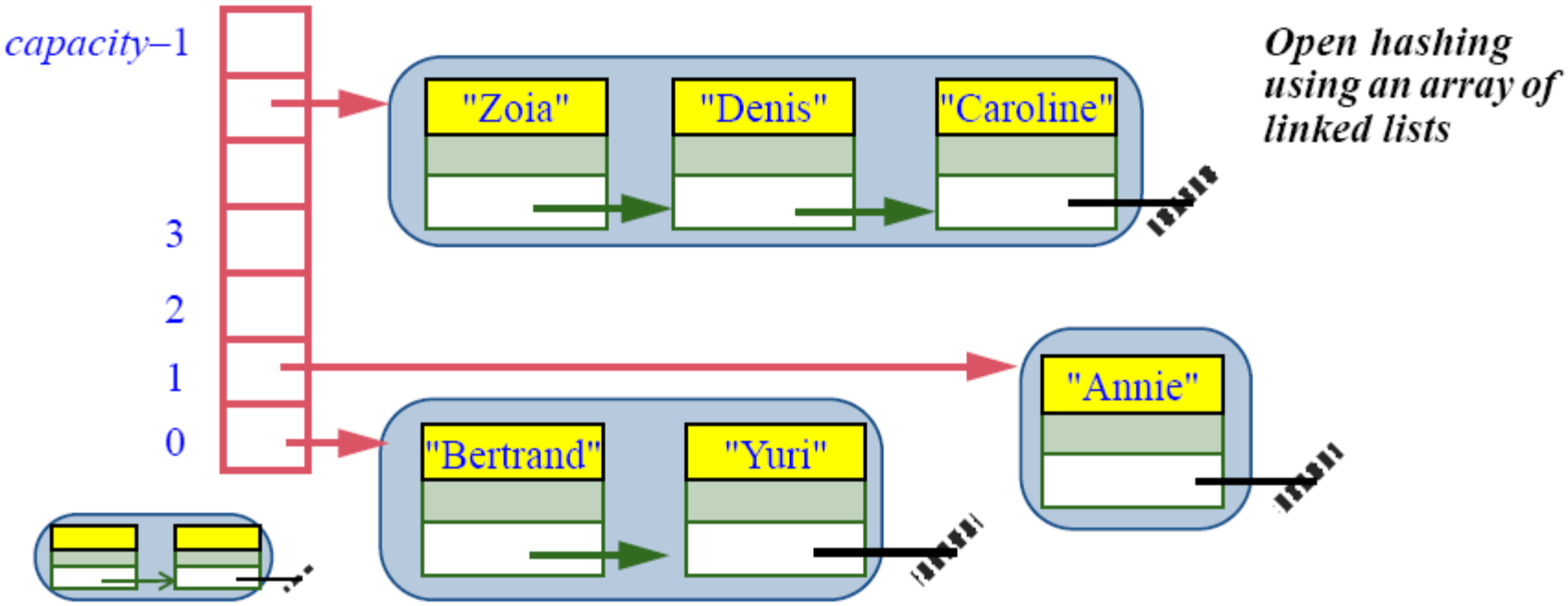
Eine perfekte Hashfunktion ergibt für jedes Element von *K* einen anderen Integer

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

Behandlung von Kollisionen

Offenes Hashing:

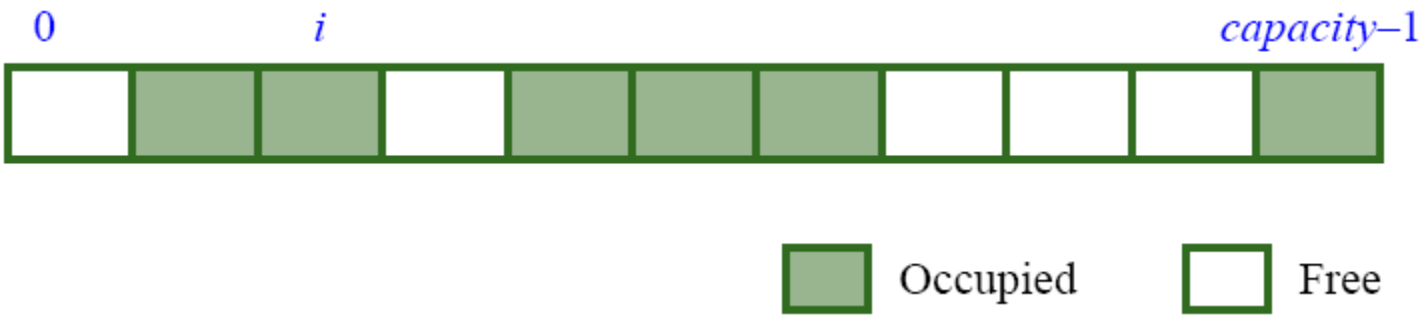
ARRAY[LINKED_LIST[G]]



Eine andere Technik: Geschlossenes Hashing

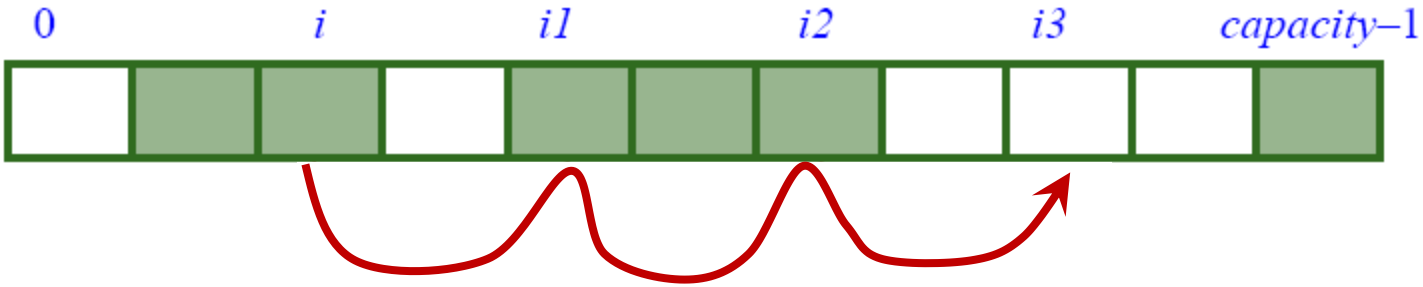
Die Klasse *HASH_TABLE [G, H]* implementiert geschlossenes Hashing:

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 (*i1, i2, i3*) zu erreichen, bis er eine freie Stelle findet.

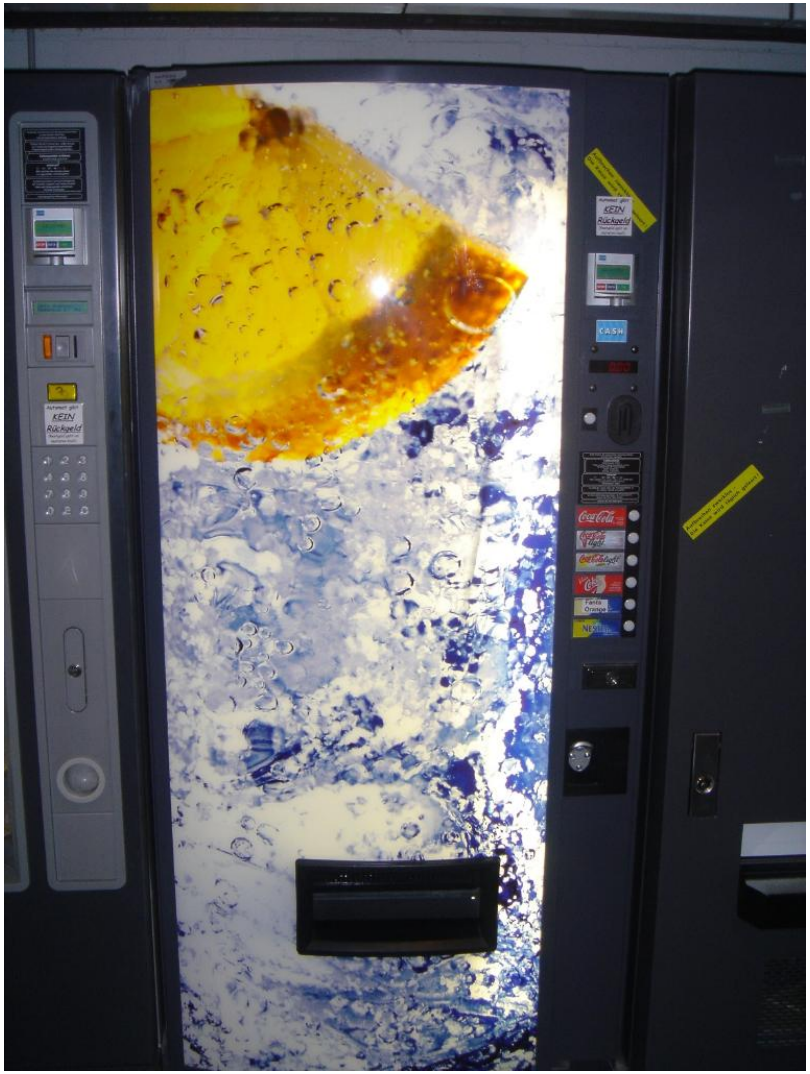


Mit dieser Richtlinie und einer guten Wahl der Hashfunktion ist das Suchen und das Einfügen in Hashtabellen in **O** (1) erreichbar.

Kosten der Operationen einer Hashtabelle

Operation	Feature	Komplexität
Schlüsselbasierter Zugriff	*item*	$O(1)$ $O(count)$
Schlüsselbasierte Einfügung	*put, extend*	$O(1)$ $O(count)$
Löschung	*remove*	$O(1)$ $O(count)$
Schlüsselbasiertes Ersetzen	*replace*	$O(1)$ $O(count)$
Suchen	*has*	$O(1)$ $O(count)$

Dispenser



Dispenser

Für Elemente von **Dispensern** gibt es keinen Schlüssel oder andere identifizierende Informationen

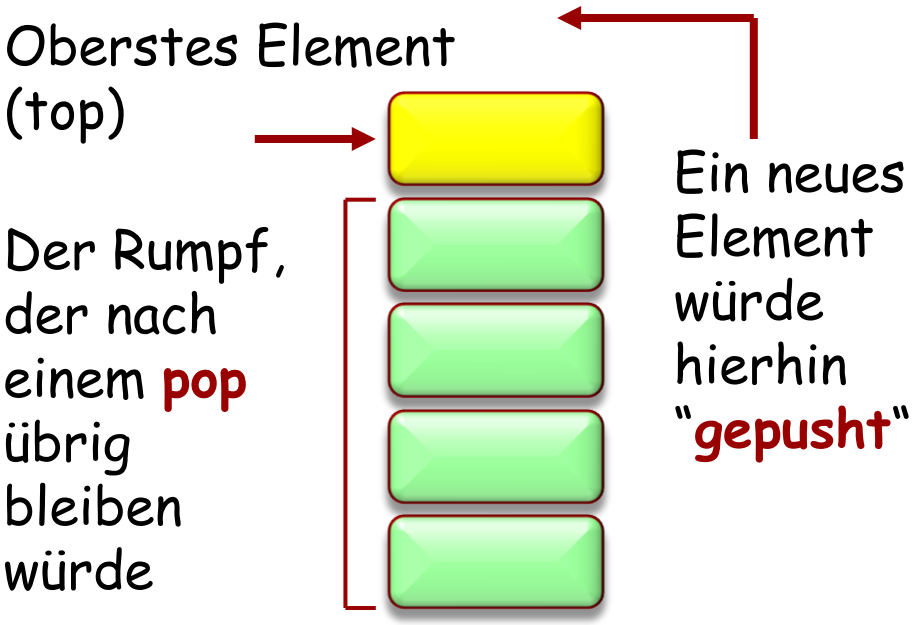
Dispenser besitzen eine spezifische Ausgaberegel, z.B.:

- Last In First Out (LIFO): Wähle das Element, das zuletzt eingefügt wurde
  → Stapel (*stack*)
- First In First Out (FIFO): Wähle das älteste, noch nicht entfernte Element
  → Warteschlange (*queue*)
- Wähle das Element mit der höchsten Priorität:
  → Vorrangswarteschlange (*priority queue*)

Stapel

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

- Ein Element auf den Stapel "drücken" (*put*)
- Das oberste Element wegnehmen (*remove*)
- Zugriff auf das oberste Element (*item*)



Anwendungen von Stapel

Viele!

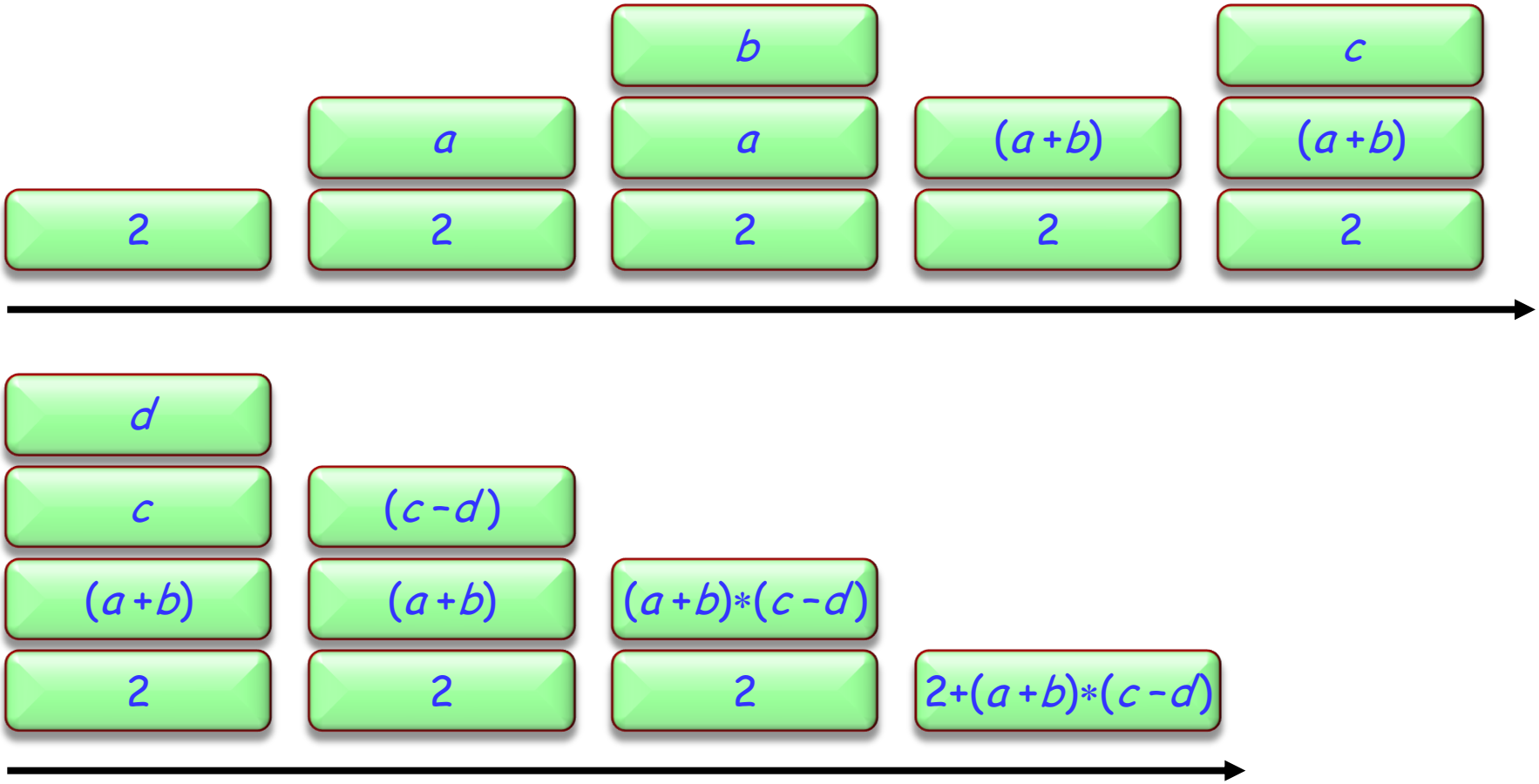
Allgegenwärtig in der Implementation von Programmiersprachen:

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

Ein Beispiel: Die Präfixnotation (auch: Polnische Notation)

```
from
until
        "Alle Token wurden gelesen"
loop
        "Lese nächsten Token"
        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 pushe
                        -- das Resultat:
                s.put (application (x, op2, op1))
        end
end
```

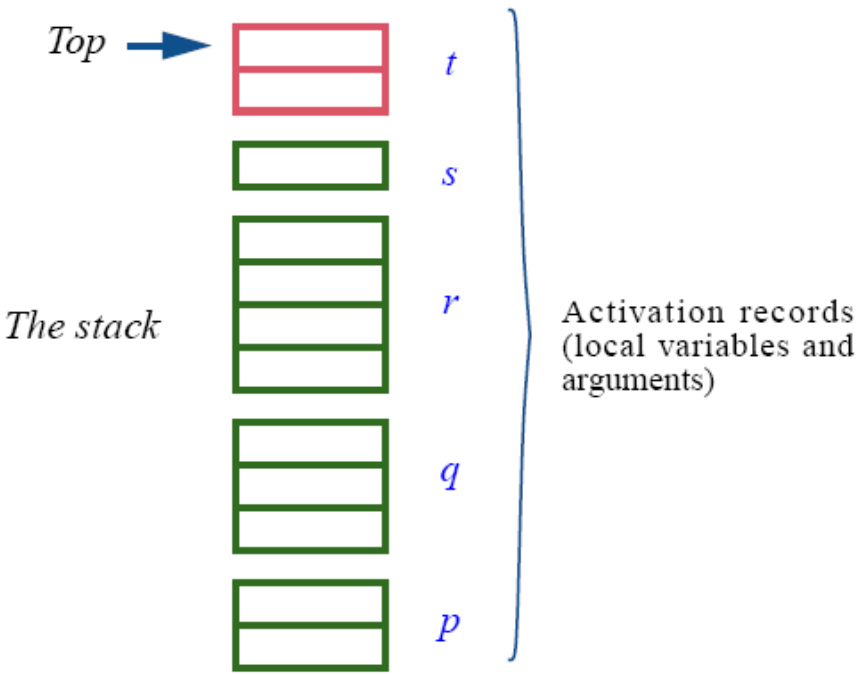
Auswertung von 2 a b + c d - * +

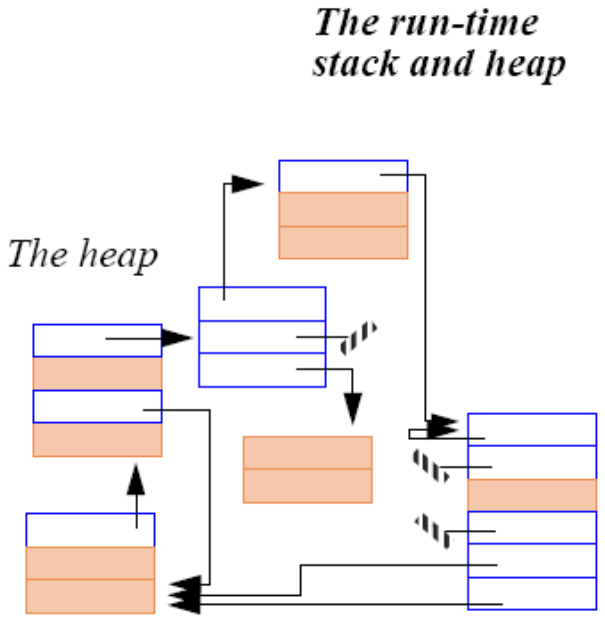


Der Laufzeit-Stapel

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

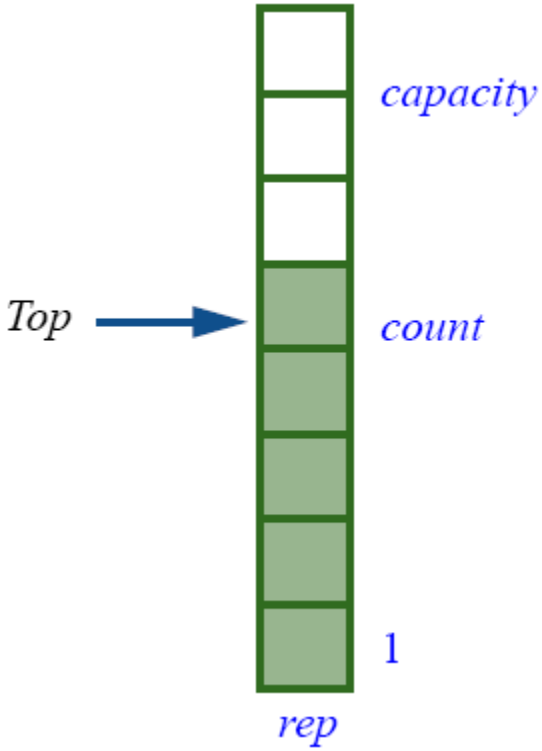
Ein Aktivierungseintrag beinhaltet die lokalen Variablen einer Routine (Argumente und lokale Entitäten).

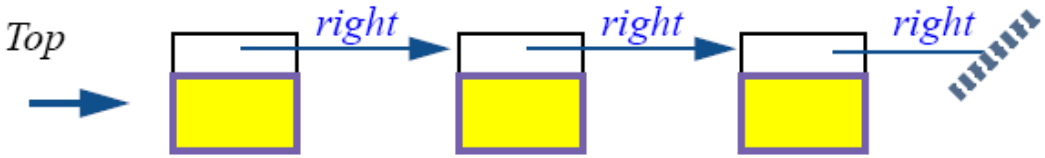




Stapel implementieren

Die häufigsten Implementationen eines Stapel sind entweder verkettet oder indiziert (arrayed).





Zwischen den Datenstrukturen auswählen

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 Elemente 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 Stapel:

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

Benutzen Sie eine Warteschlange:

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

Was wir in dieser Vorlesung gesehen haben

Container-Datenstrukturen: Grundlegende Begriffe, Schlüsselbeispiele

Ein wenig Komplexitätstheorie ("Big-O")

Wann welcher Container zu wählen ist