
Fondamenti della Programmazione: Metodi Evoluti

Prof. Enrico Nardelli

Lezione 12: «*Container*» data structures

Topics for this lecture

Containers and genericity

Container operations

Assessing algorithm performance: Big-O notation

Important Data Structures:

- Tuples
- Lists
- Arrays
- Hash tables
- Stacks and queues

Container data structures

Contain other objects (“**items**”)

Various container implementations, as studied next, determine:

- Which of these operations are available
- Their speed
- The storage requirements

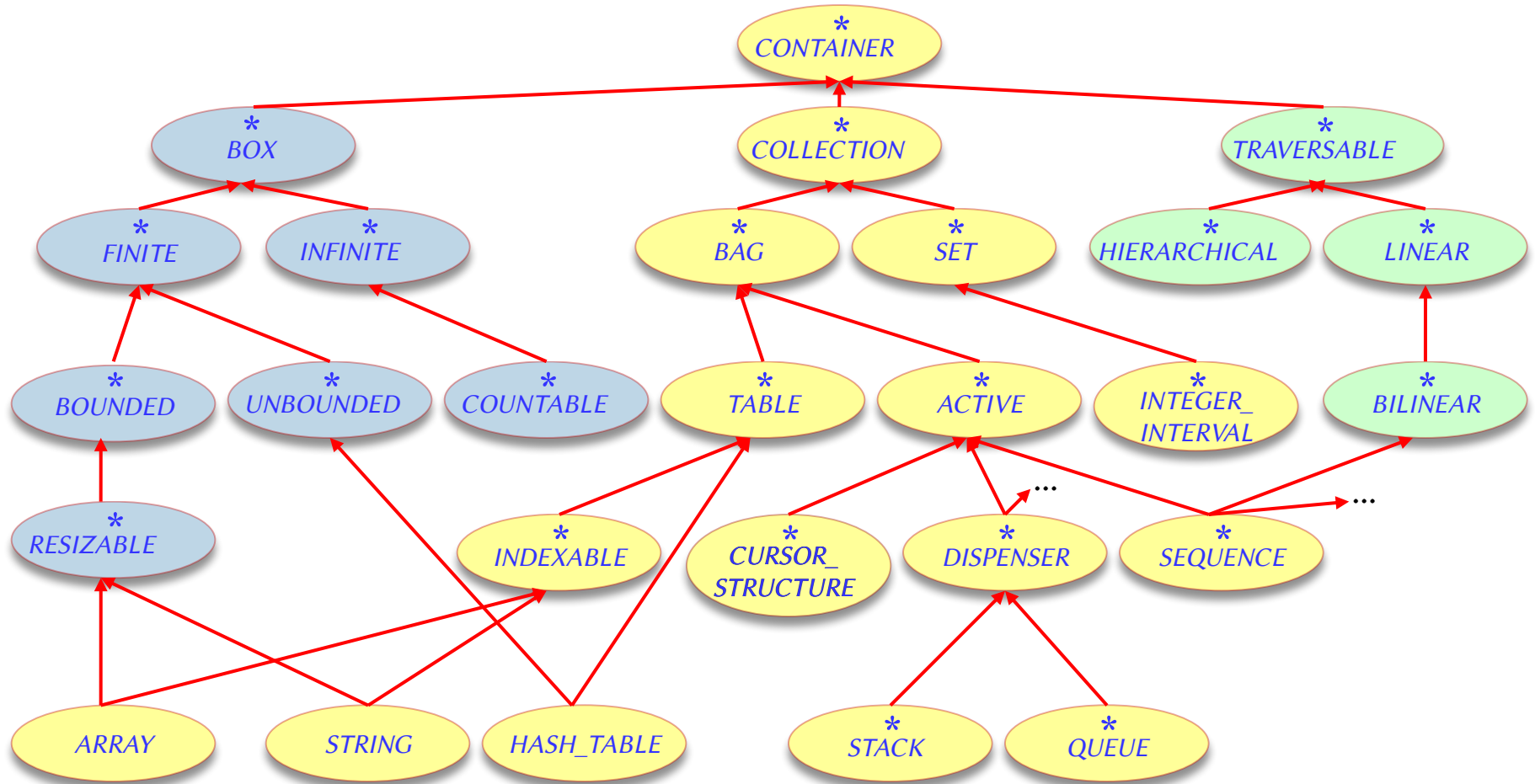
This lecture is just an intro; “Data Structures and Algorithms” is a large and important section of Informatics

Container data structures: classification

- A container can be studied from three viewpoints: **storage**, **access**, and **traversal**
- For each of these viewpoint the Base library offers a hierarchy of **deferred** classes:
 - **BOX** describes **storage** properties, such as being bounded or unbounded.
 - **COLLECTION** describes **access** properties (defining how to access a container's items, for example through an index or according to a last-in, first-out policy).
 - **TRAVERSABLE** describes **traversal** properties, such as sequential or hierarchical.

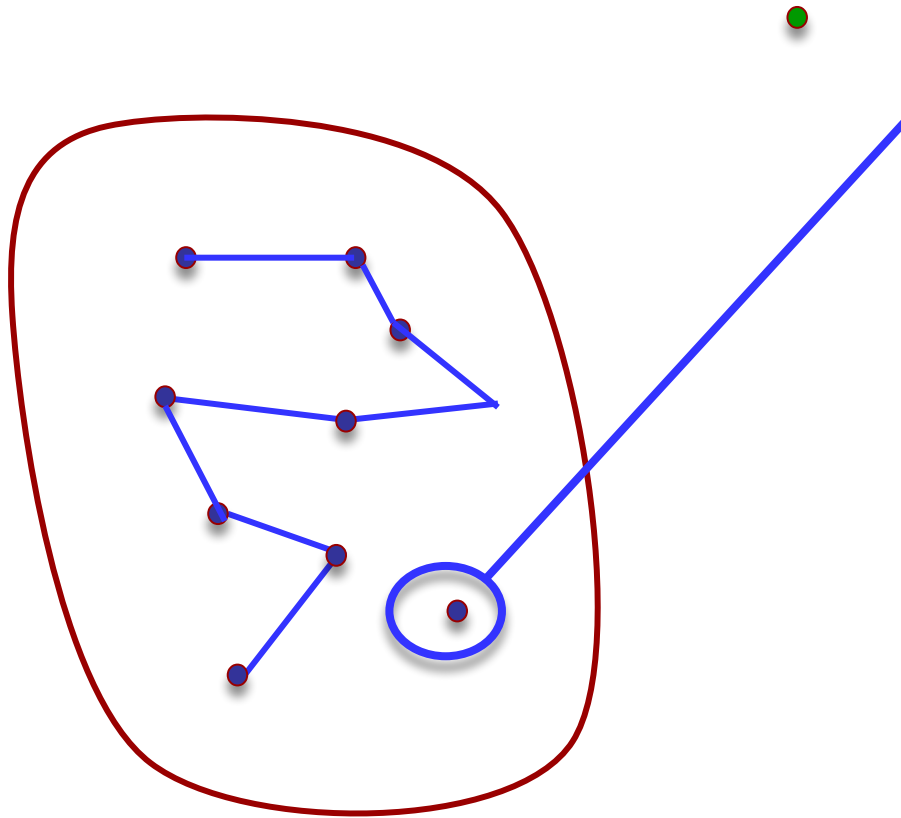
Deferred classes in EiffelBase

From lesson 11-EREDITARIETÀ



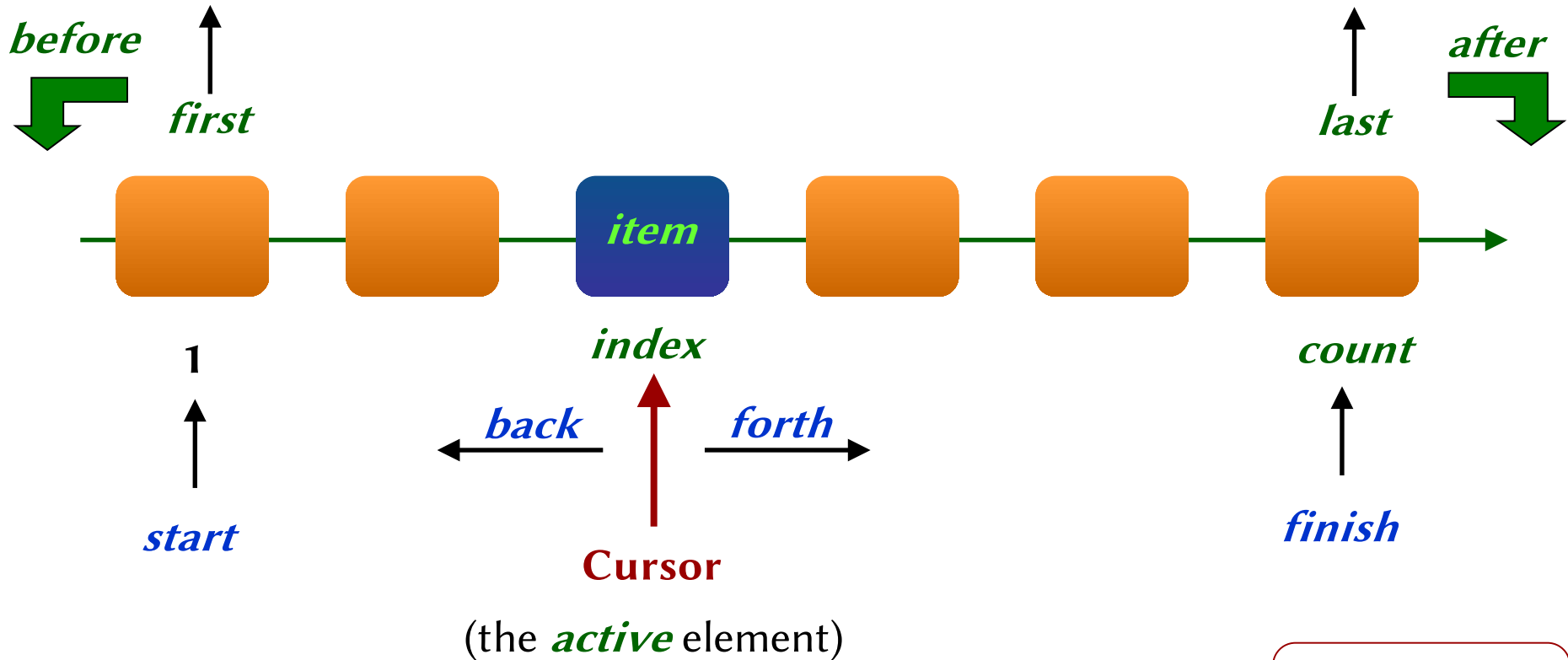
* deferred

Container data structure operations



- **Insertion**: add an item
- **Search**: find out if a given item is present
- **Removal**: remove an occurrence (if any) of an item
- **Wipeout**: remove all occurrences of items
- **Traversal/Iteration**: apply a given operation to every item

A familiar container: the list



To facilitate iteration and other operations, our lists have **cursors** (here internal, can be external)

Queries
Commands

A standardized naming scheme

Container classes in EiffelBase use standard names for basic container operations:

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

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

Whenever applicable, use them in your own classes as well

Bounded representations

In designing container structures, avoid hardwired limits!

“**Don’t box me in**”: EiffelBase is paranoid about hard limits

- Most structures conceptually unbounded
- Even arrays (bounded at any particular time) are resizable

When a structure is bounded, the maximum number of items is called *capacity*, with an invariant

count \leq *capacity*

Containers and genericity

How do we handle variants of a container class distinguished only by the item type?

Solution: **genericity** allows explicit type parameterization consistent with static typing

Container structures are implemented as generic classes:

LINKED_LIST [G]

pl : LINKED_LIST [PERSON]

sl : LINKED_LIST [STRING]

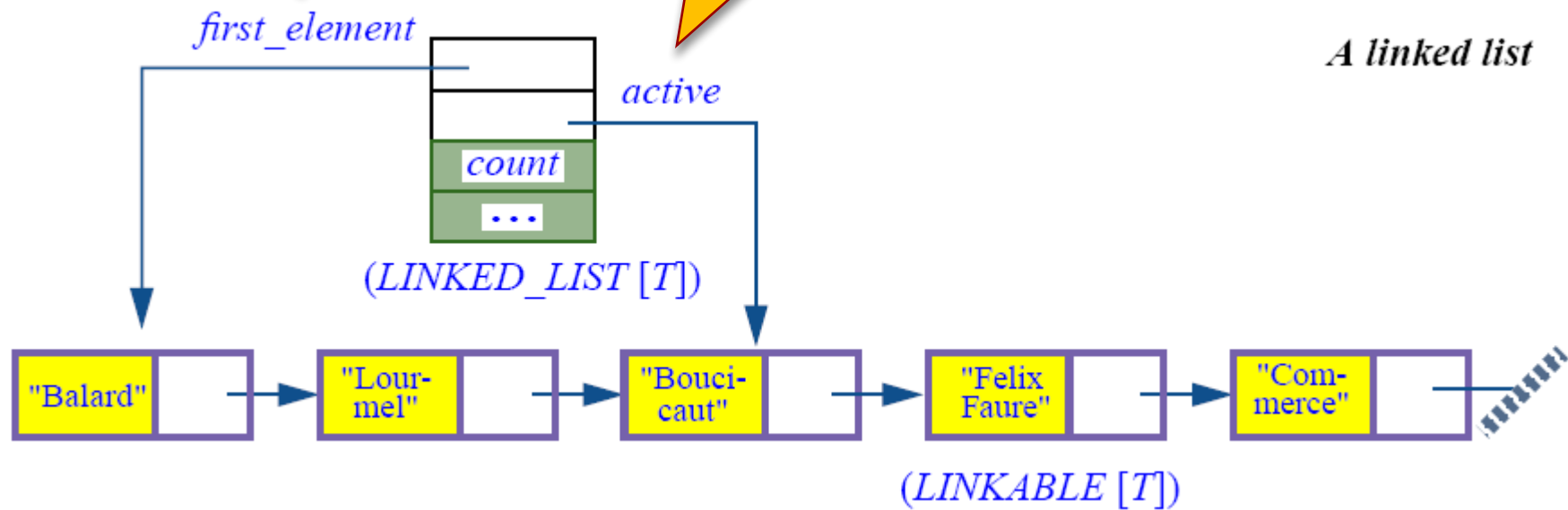
al : LINKED_LIST [ANY]

A specific implementation: (singly) linked lists

EiffelBase class: *LINKED_LIST* [T]

not exported: use *first*

not exported: use *item*




Caveat

Whenever you define a container structure and the corresponding class, pay attention to borderline cases:

- Empty structure
- Full structure (if finite capacity)

Key features

Attributes  are the key features in manipulating a container since are the fundamental places keeping information on its status

Queries provide derived information on container status

Commands change the container status (hence attributes value)

The **attributes** of *LINKED_LIST*[*G*] are ONLY these ones (some were defined as **deferred** in ancestor classes):

- *active* is the current element (possibly **Void**), export {*LINKED_LIST*}
- *first_element* is the first element (possibly **Void**), export {*LINKED_LIST*}
- *count* is the number of elements
- *before* is true when the cursor is *before* the first element
- *after* is true when the cursor is *after* the last element

Class invariants defining properties (inherited)

from *FINITE*: *is_empty* = (*count* = 0)

NB: it's how the **function** *is_empty* is defined

from *TRAVERSABLE*: *is_empty* **implies** *off*

NB: *off* is **deferred** in *TRAVERSABLE*; defined here as
off = *before* **or** *after*

from *LINEAR*: *after* **implies** *off*

from *BILINEAR*: *before* **implies** *off*

from *BILINEAR*: **not** (*after* **and** *before*)

Class invariants defining properties (inherited)

from *CHAIN*: $0 \leq \text{index} \leq \text{count} + 1$

NB: *index* is a **function** (it traverses the list)
providing the current cursor position

from *CHAIN*: $\text{off} = ((\text{index} = 0) \text{ or } (\text{index} = \text{count} + 1))$

from *CHAIN*: $\text{isfirst} = ((\text{not } \text{is_empty}) \text{ and } (\text{index} = 1))$

from *CHAIN*: $\text{islast} = ((\text{not } \text{is_empty}) \text{ and } (\text{index} = \text{count}))$

from *CHAIN*: **not** *off* **implies** $(\text{item} = i_th(\text{index}))$

NB: *item* and *i_th* are **functions** providing the
element at a given integer

from *LIST*: $\text{before} = (\text{index} = 0)$

from *LIST*: $\text{after} = (\text{index} = \text{count} + 1)$

Class invariants defining further constraints

In an **empty list** there is no *active* nor *first_element* :

is_empty **implies** ((*first_element* = **Void**) and (*active* = **Void**))

Also:

(*active* = **Void**) **implies** *is_empty*

therefore in a **not empty list** *active* cannot be **Void**

Note the following constraints:

before **implies** (*active* = *first_element*)

after **implies** (*active* = *last_element*)

In these cases *active* is on the first or the last element, but they cannot be accessed since *item* requires **not off**

Queries: accessing elements

item : returns *value* (of type *G*) stored in the element at the current cursor position (i.e., *active.item*)

require not off

first : returns *value* stored in the first *element* (i.e., *first_element.item*)

last : returns *value* stored in the last *element* (i.e., *last_element.item*)

N.B.: *last_element* is a **function** traversing the list!

require not is_empty (for both)

i_th (*i*: *INTEGER*) returns *value* (of type *G*) stored in the element in the *i_th* position, i.e. at index *i*

require $1 \leq i \leq count$

Commands with contracts: moving the cursor

start : move the cursor to first position (no effect if empty)

ensure (not *is_empty*) **implies** *isfirst*

ensure *is_empty* **implies** *after*

finish : move the cursor to last position (no effect if empty)

ensure (not *is_empty*) **implies** *islast*

ensure *is_empty* **implies** *before*

forth : move the cursor to next position

require not *after*

ensure *index* = **old** *index* + 1

back : move the cursor to previous position

require not *before*

ensure *index* = **old** *index* - 1

go_i_th : move the cursor to *i*-th position

require $0 \leq i \leq \text{count} + 1$

ensure *index* = *i*

Tuples

A **tuple** is a container storing items of **arbitrary** type in a set of contiguous memory locations, each

optionally

identified by name

author

year

title

Collodi

1883

Pinocchio

Example:

`a_book: TUPLE [author: STRING, year: INTEGER, title: STRING]`

where **author** and **year** and **title** denote the tuple components and are called **tags**

Tags facilitates access to individual elements...

... but can be omitted if you do not need to use them...

Tuples: tags for accessing and modifying

Given:

`a_person: TUPLE [name: STRING, age: INTEGER]`

`a_person := ["Mario", 56]`

Assigns to `a_person` the expression `["Mario", 56]`, called a **manifest tuple**

`print(a_person.name)`

Prints the string `Mario`

`a_person.age := 48`

Uses tag `age` as an assigner command for the tuple component

NB: `a_person: TUPLE [STRING, INTEGER]` is a correct definition for a tuple requiring no access to specific component

Tuple type as anonymous class

A tuple is like a class providing **only**

- Attributes
- All public
- With setter procedures
 - without precondition
 - doing nothing else beyond setting attribute value

In cases like this a tuple is a more economic modeling solution than a class

Known also as *anonymous classes*

Tuples: conformance rules

The type *TUPLE* describes tuples of arbitrary length and composition

The type *TUPLE* [T] describes tuples whose first component is of type T and the rest is of arbitrary length and composition

The type *TUPLE* [T, V] describes tuples whose first component is of type T, second component is of type V, and the rest is of arbitrary length and composition

... and so on...

Then, given

p0: *TUPLE*

p1: *TUPLE* [STRING]

p2: *TUPLE* [STRING, INTEGER]

Assignments like:

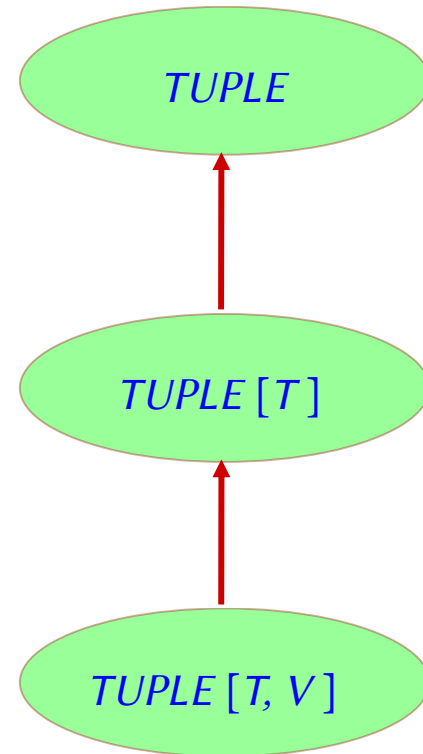
p0 := *p1*; *p1* := *p2*

are conforming,

while assignments like

p2 := *p1*; *p1* := *p0*

are **not conforming**

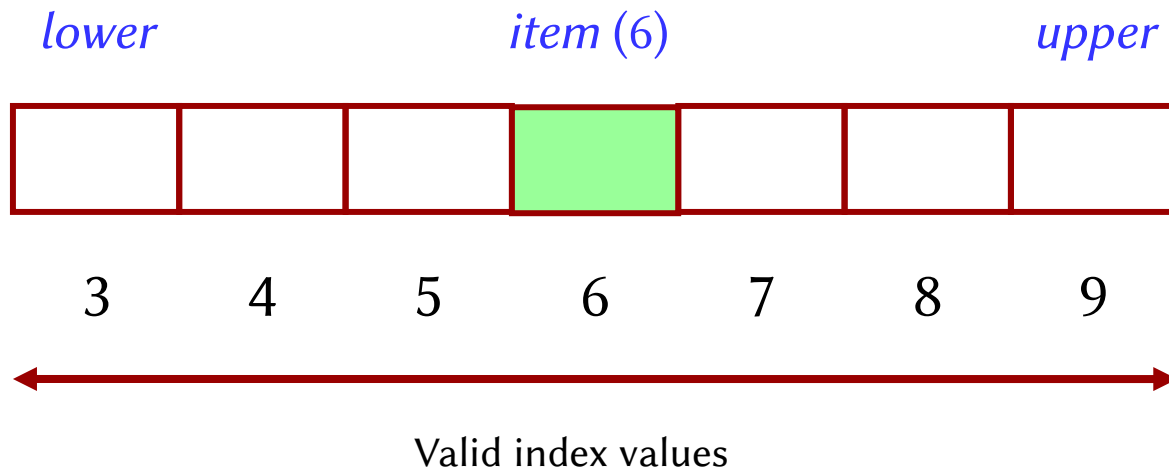


Arrays

An **array**

ARRAY [*G*]

is a container storing items of type **conforming** to *G* in a set of contiguous memory locations, each identified by an integer index



Bounds and indexes (1)

Arrays are bounded:

lower: INTEGER

-- Minimum index.

upper: INTEGER

-- Maximum index.

Usage:

my_array:ARRAY[INTEGER]

...

create my_array.make_filled (0, 4, 17)

default value

bounds

The capacity (or count) of an array is:

capacity = count = upper – lower + 1

Bounds and indexes (2)

Since class invariant requires

$$\textit{count} \geq 0$$

We have

$$\textit{lower} \leq \textit{upper} + 1$$

An array has one element ($\textit{count} = 1$) if and only if

$$\textit{lower} = \textit{upper}$$

An array is empty ($\textit{count} = 0$) if and only if

$$\textit{lower} = \textit{upper} + 1$$

Accessing and modifying array items

item (i : *INTEGER*) : *G*

-- Entry at index i , if in the *index* interval.

require

valid_key: *valid_index* (i)

put (v : *G* ; i : *INTEGER*)

-- Replace i -th entry, if in the *index* interval, by v .

require

valid_key: *valid_index* (i)

ensure

inserted: *item* (i) = v

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

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

Safe insertion in arrays

put ($v: G ; i: \text{INTEGER}$)

-- Replace i -th entry, if in the index interval, by v .

require

valid_key: *valid_index* (i)

ensure

inserted: *item* (i) = v

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

A *put* (v, i) might fail if i is not a valid index.

Use *force* (v, i) if you don't want to check the precondition. It "forces" the resizing of the array is the index is not in the validity range...

Resizing an array (1)

At any point in time arrays have a fixed lower and upper bound, and thus a fixed capacity

Unlike most other programming languages, Eiffel allows resizing an array (*resize*)

resize (min_index, max_index: INTEGER)

-- Enlarge array, preserving existing items,
-- down to *min_index* and up to *max_index*.

require

valid_bounds: min_index <= max_index

ensure

no_low_lost: lower = min_index.min (old lower)

no_high_lost: upper = max_index.max (old upper)

attempts to increase *lower* or to decrease *upper* will be discarded

It is a costly operation, since involves allocating a new zone and copying all elements

Resizing an array (2)

Inserting element with `put` can be done only within the bound. What if one needs to insert an element outside the current bounds?

Use feature *force* which, unlike *put*, has no precondition and can insert outside bounds. If required, it resizes the array:

force ($v: G ; i: \text{INTEGER}$)

-- Assign i -th entry to v .

ensure

inserted: $\text{item}(i) = v$

higher_count: $\text{count} \geq \text{old count}$

By default, *force* enlarge the size when needed by 50% so that in a linear sequence of *force*d insertions only a logarithmic number of costly resizing operation is performed.

REMEMBER: simplifying the notation

Feature *item* is declared as

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

From
03 – Features p.42

This allows the following synonym notations:

<i>a</i> [<i>i</i>]	for	<i>a.item</i> (<i>i</i>)
<i>a.item</i> (<i>i</i>) := <i>x</i>	for	<i>a.put</i> (<i>x</i> , <i>i</i>)
<i>a</i> [<i>i</i>] := <i>x</i>	for	<i>a.put</i> (<i>x</i> , <i>i</i>)

From
06 – Visibilità
p.8

A class may have at most one feature, with any number of arguments, **alias**-ed to “[]”

Linked list or array?

The choice of a container data structure depends on the speed of its container operations

The speed of a container operation depends on how it is implemented, on its underlying algorithm

How fast is an algorithm?

Depends on the hardware, operating system, load on the machine...

But most fundamentally depends on the algorithm!

Algorithm complexity: “big-O” notation

Let n be the size of the data structure ($count$).

“ f is $O(g(n))$ ”

means that there exists a constant k such that:

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

Defines function not by exact formula but by order of magnitude, e.g.

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

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

Why is the order of magnitude important?

Consider algorithms with complexity

$$O(n)$$

$$O(n^2)$$

$$O(2^n)$$

Assume for your next birthday they have promised you a new PC 1'000 times faster...

How much bigger a problem than today can you solve in one day of computation time?

Assume for your master degree they have promised you a new PC 1'000'000 times faster...

How much bigger a problem than today can you solve in one day of computation time?

Examples

put_right of *LINKED_LIST*: **O** (1)

Regardless of the number of elements in the linked list it takes a constant time to insert an item at cursor position.

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

At worst the time for this operation grows proportionally to the number of elements in the array.

Variants of algorithm complexity

We may be interested in

- Worst-case performance
- Best-case performance (seldom)
- Average performance (needs statistical distribution)

Unless otherwise specified this discussion considers worst-case

Lower bound notation: $\Omega(n)$

Cost of **linked list** operations

Operation	Feature	Complexity
Insert right to cursor	<i>put_right</i>	O (1)
Insert at end	<i>extend</i>	O (1)
Move cursor ahead	<i>forth</i>	O (1)
Move cursor back	<i>back</i>	O (<i>count</i>)
Remove right neighbor	<i>remove_right</i>	O (1)
Remove at cursor position	<i>remove</i>	O (<i>count</i>)
Index-based access	<i>i_th</i>	O (<i>count</i>)
Search	<i>has</i>	O (<i>count</i>)

Cost of **array** operations

Operation	Feature	Complexity
Index-based access	<i>item</i>	O (1)
Index-based replacement	<i>put</i>	O (1)
Index-based replacement outside of current bounds	<i>force</i>	O (<i>count</i>)
Search	<i>has</i>	O (<i>count</i>)
Search in sorted array	-	O (<i>log count</i>)

Hash tables

Can we get the efficiency of arrays

- Constant-time access
- Constant-time update

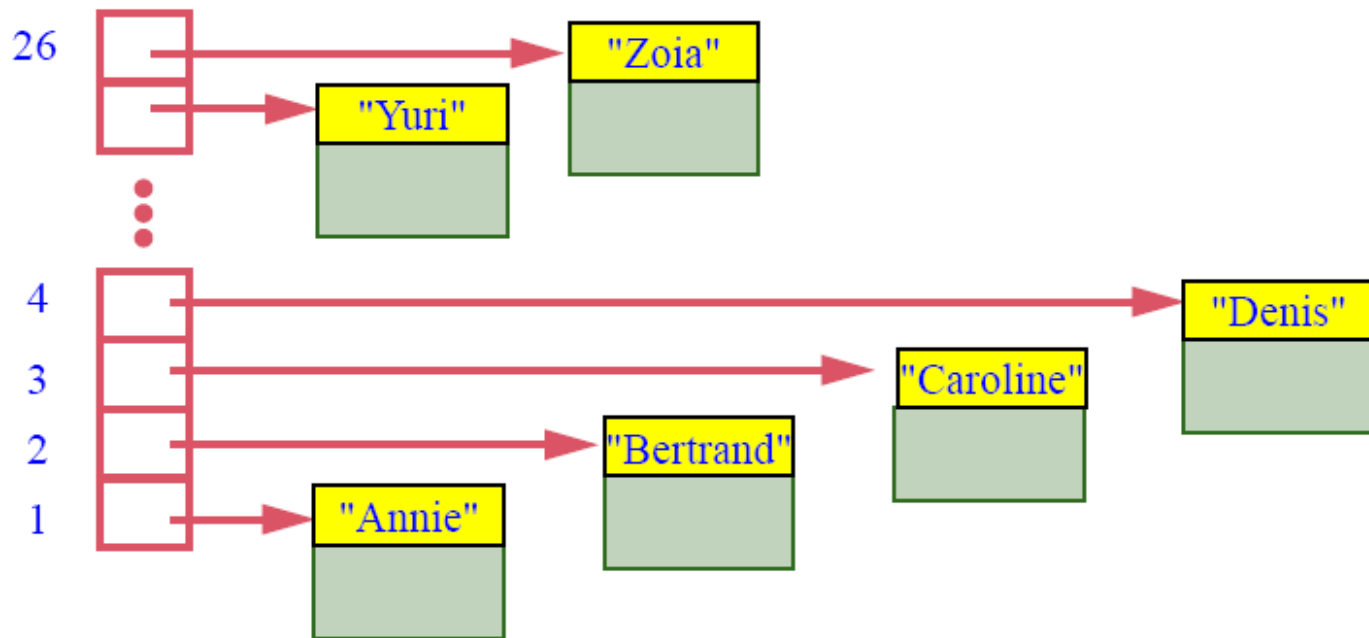
without limiting ourselves to keys that are integers in a fixed, contiguous interval?

Hash table answer: **almost!**

Hash tables

Arrays allows to access elements through an integer **index**. What if we want another kind of index, e.g. strings?

Hash tables allow **keys** other than integers, e.g. strings.
A trivial example:



Hash function

The hash function maps K , the set of possible keys, into an integer interval $a..b$.

A **perfect** hash function gives a different integer value for every element of K .

Otherwise, whenever two different keys give the same hash value a **collision** occurs.

A mapping structure

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!](#)

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.

Cool Sites

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

Using hash tables

person, person1 : PERSON

personnel_directory : HASH_TABLE [PERSON, STRING]

create *personnel_directory.make* (100000)

Storing an element:

create *person1*

personnel_directory.put (*person1*, "Annie")

Retrieving an element

person := personnel_directory.item ("Annie")

Constrained genericity & the class interface

class

HASH_TABLE [G, K -> HASHABLE]

Allows *h* ["ABC"] for *h* • *item* ("ABC")

Allows *h* • *item* ("ABC") := *x*
for *h* • *put* (*x*, "ABC")

feature

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

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

- Insert *new* with *key* if no other item
- is associated with same key, otherwise
- do nothing

Together, allow
h ["ABC"] := *x*
for *h* • *put* (*x*, "ABC")

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

- Update table so that *new* will be
- the item associated with *key*.

...

end

The example rewritten

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

```
create personnel_directory.make (100000)
```

Storing an element:

```
create person1
```

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

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

Not good style, why?

Retrieving an element

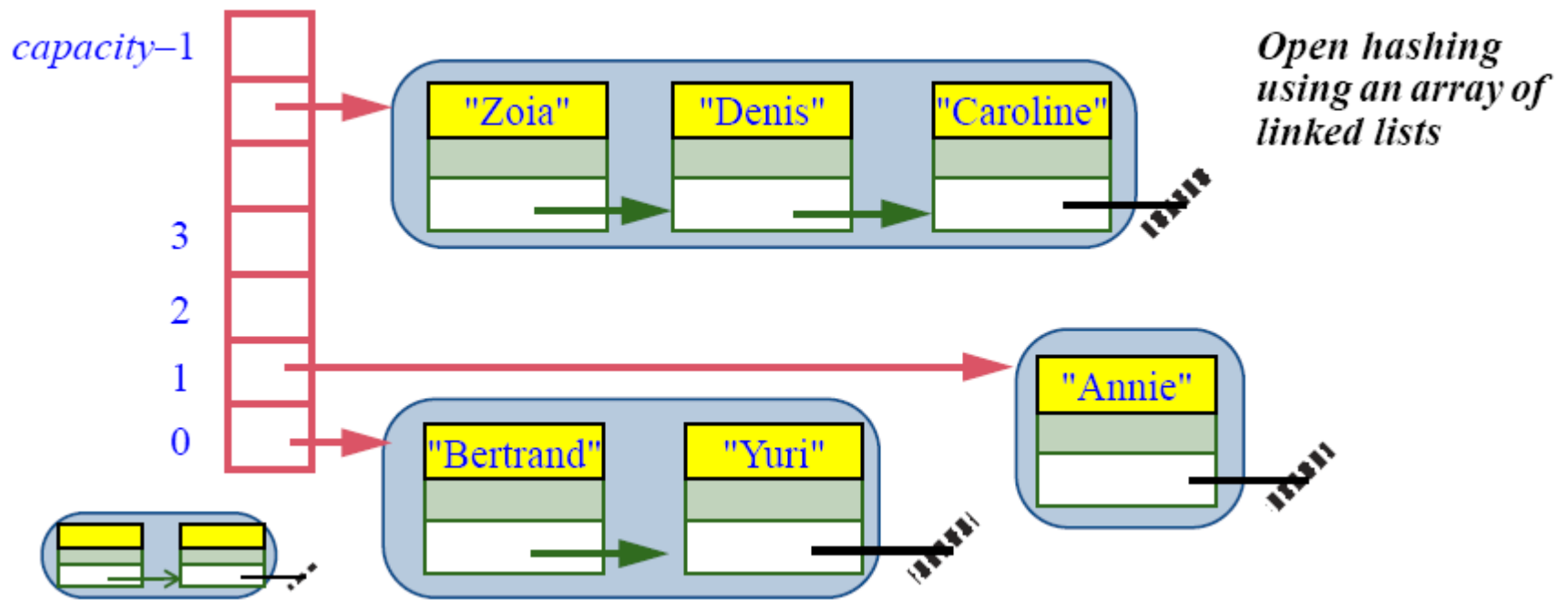
```
person := personnel_directory.item ("Annie")
```

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

Collision handling

Open hashing:

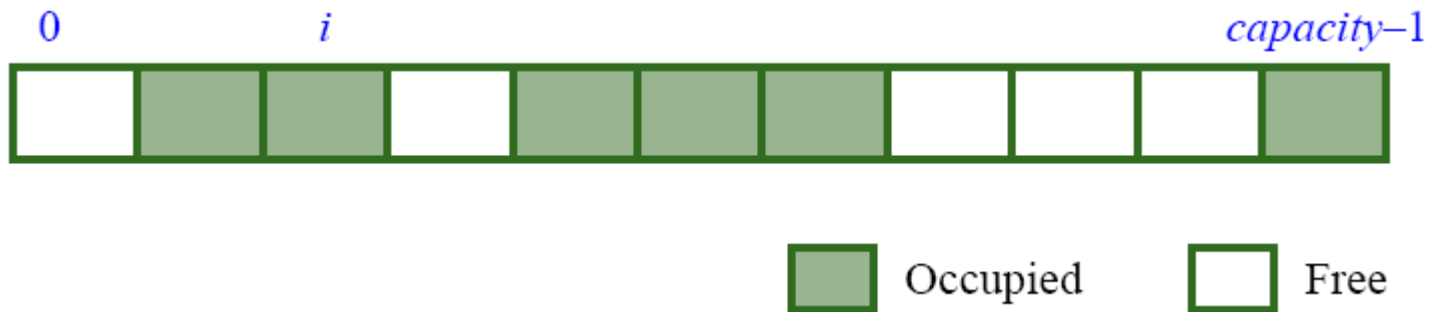
$ARRAY[LINKED_LIST[G]]$



A better technique: closed hashing

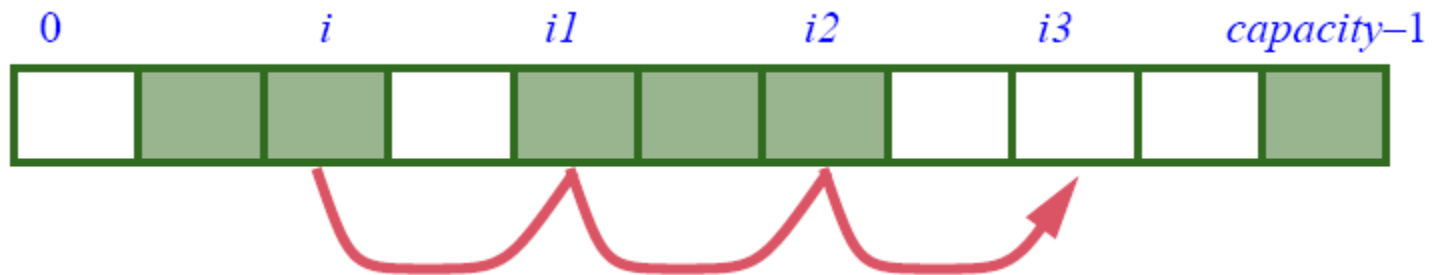
In EiffelBase the class *`HASH_TABLE [G, H]`* implements closed hashing:

`HASH_TABLE [G, H]` uses a single *`ARRAY [G]`* to store the items. At any time some of positions are occupied and some free:



Closed hashing

If the hash function yields an already occupied position, the mechanism will try a succession of other positions (*i1*, *i2*, *i3*), provided by adding a suitably defined increment, until it finds a free one:



With this policy and a good choice of hash function search and insertion in a hash table are $O(1)$...

...save for the need of enlarging the table when it becomes almost full: an $O(count)$ operation (this affects *put* and *force*). Eiffel use 80% as a threshold to decide when enlarging the table

Cost of hash table operations

Operation	Feature	Complexity
Key-based access	<i>item</i>	O (1)
Key-based insertion	<i>put, force</i>	O (count)
Removal	<i>remove</i>	O (1)
Key-based replacement	<i>replace</i>	O (1)
Search	<i>has</i>	O (1)

Dispensers



Dispensers

Unlike indexed structures, as arrays and hash tables, there is no key or other identifying information for dispenser items.

Dispensers are container data structures that prescribe a specific retrieval policy:

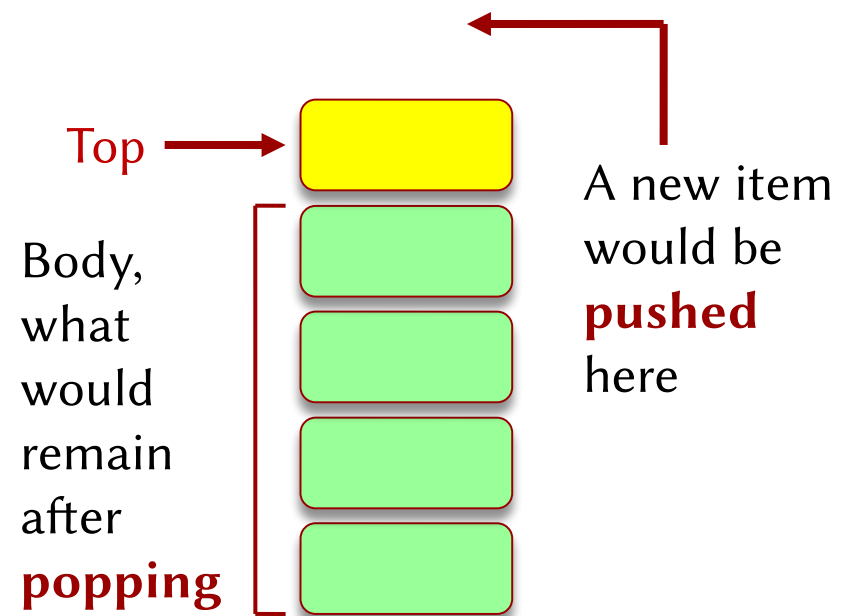
- **Last In First Out (LIFO)**: choose the element inserted most recently → **stack**.
- **First In First Out (FIFO)**: choose the oldest element not yet removed → **queue**.
- **Priority queue**: choose the element with the highest priority.

A stack is a dispenser applying a LIFO policy. The basic operations are:

Access the **top** element (*item*)

Pop the top element (*remove*)

Push an item to the top of the stack (*put*)



Applications of stacks

Many!

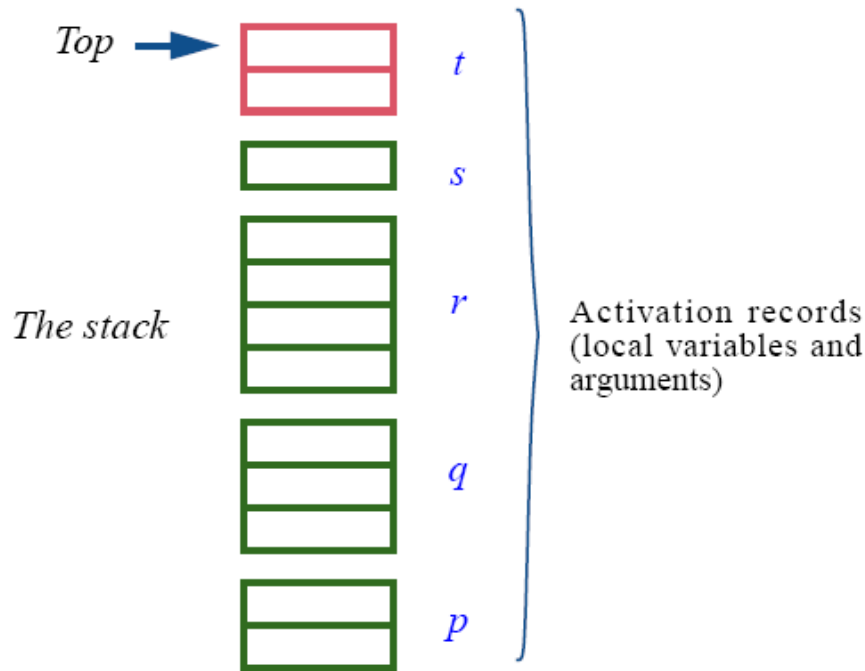
Ubiquitous in programming language implementation:

- Parsing expressions
- Managing execution of routines (“**THE** stack”)
Special case: implementing **recursion**
- Traversing trees
- ...

The run-time stack

The run-time stack contains the *activation records* for all currently active routines.

An *activation record* contains a routine's locals (arguments and local entities).

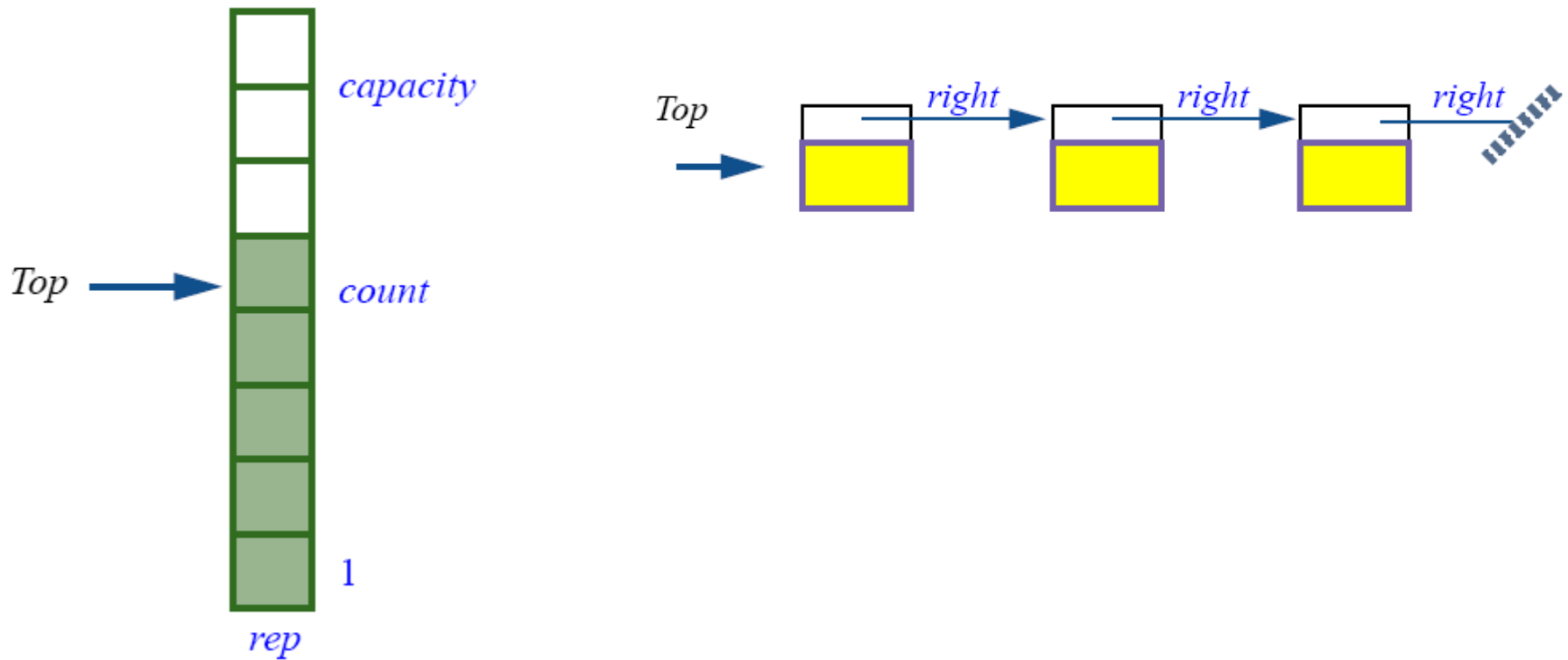


Before a call: push on stack a “frame” containing values of local variables, arguments, and return information

After a call: pop frame from stack, restore values (or terminate if stack is empty)

Implementing stacks

Common stack implementations are either arrayed or linked.



Choosing between data structures

Use a linked list if:

- Order between items matters
- The main way to access them is in that order
- (Bonus condition) No hardwired size limit

Use an array if:

- Each item can be identified by an integer index
- The main way to access items is through that index
- Hardwired size limit (at least for long spans of execution)

Use a hash table if:

- Every item has an associated key
- The main way to access them is through these keys
- The structure is bounded

Use a stack:

- For a LIFO policy
- Example: traversal of nested structures such as trees

Use a queue:

- For a FIFO policy
- Example: simulation of FIFO phenomenon

Learning how to use data structures

Study the relevant sections of Eiffel on-line documentation

EiffelBase, The Kernel

<https://www.eiffel.org/doc/solutions/EiffelBase%2C%20The%20Kernel>

EiffelBase Data Structures Overview

<https://docs.eiffel.com/book/solutions/eiffelbase-data-structures-overview>

Accessible also through the course web site

What we have seen

Container data structures: basic notion, key examples

Algorithm complexity (“Big-O”)

How to choose a particular kind of container