Topics for this lecture

- Containers and genericity
- Container operations
- Lists
- Arrays
- Assessing algorithm performance: Big-O notation
- Hash tables
- Stacks and queues

The ultimate question

LINKED_LIST or ARRAY?
**Container data structures**

Contain other objects ("items")

Some fundamental operations on a container:

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

Various container implementations, as studied next, determine:

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

This lecture is just an intro; see "Data Structures and Algorithms" (second semester course) for an in-depth study

---

**A familiar container: the list**

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

---

**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
  \[ \text{count} \leq \text{capacity} \]

Containers and genericity

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

Solution: using genericity allows explicit type parameterization consistent with static typing principles

Container data structures are typically implemented as generic classes

\[
\text{LINKED\_LIST}[G]
\]

\[
\begin{align*}
\text{pl} & : \text{LINKED\_LIST}[\text{PERSON}] \\
\text{sl} & : \text{LINKED\_LIST}[\text{STRING}] \\
\text{al} & : \text{LINKED\_LIST}[\text{ANY}]
\end{align*}
\]

Lists

A list is a container keeping items in a certain order. Lists in EiffelBase have cursors.

\[
\begin{align*}
\text{before} & \\
\text{start} & \\
\text{index} & \\
\text{back} & \\
\text{forth} & \\
\text{count} & \\
\text{after} & \\
\text{finish} & \\
\end{align*}
\]
Cursor properties (all in class invariant!)

The cursor ranges from 0 to count + 1:
\[ 0 \leq index \leq count + 1 \]

The cursor is at position 0 if and only if before is True:
\[ before = (\text{index} = 0) \]

It is at position count + 1 if and only if after is True:
\[ after = (\text{index} = count + 1) \]

In an empty list the cursor is at position 0:
\[ \text{is_empty} = (\text{count} = 0) \]

A specific implementation: (singly) linked lists

Caveat

Whenever you define a container structure and the corresponding class, pay attention to borderline cases:
> Empty structure
> Full structure (if finite capacity)
Adding a cell

The corresponding command

```
put_right(v : G)
  -- Add v to right of cursor position; do not move cursor.
  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 := 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
```

Removing a cell
The corresponding command

Do remove as an exercise

Inserting at the end: extend

Arrays

An array is a container storing items in a fixed (at any specific time) set of contiguous memory locations. Each memory location is identified by an integer index.

<table>
<thead>
<tr>
<th>lower</th>
<th>item (4)</th>
<th>upper</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Valid index values
Bounds and indexes

Arrays are bounded:

lower: INTEGER  -- Minimum index
upper: INTEGER  -- Maximum index

The capacity of an array is determined by the bounds:
capacity = upper - lower + 1

Accessing and modifying array items

item (i: INTEGER) : G
  -- Entry at index i, if in index interval.
require
valid_key: valid_index (i)

put (v: G, i: INTEGER)
  -- Replace i-th entry, if in index interval, by v.
require
valid_key: valid_index (i)
ensure
inserted: item (i) = v

Eiffel note: simplifying the notation

Feature item is declared as

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

This allows the following synonym notations:

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

These facilities are available to any class
A class may have at most one feature aliased to "[]"
Resizing an array

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).

Feature force resizes an array if required: unlike put, it has no precondition. Resizing usually requires reallocating the array and copying the old values. Such operations are costly!

Using an array to represent a list

See class ARRAYED_LIST in EiffelBase.

Introduce count (number of elements in the list)

The number of list items ranges from 0 to capacity:

\[ 0 \leq \text{count} \leq \text{capacity} \]

An empty list has no elements:

\[ \text{is_empty} = (\text{count} = 0) \]

Linked or arrayed list?

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 (\( \text{count} \)).

\( f \) is \( O(g(n)) \)

means that there exists a constant \( k \) such that \( \forall n: \)

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

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

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

\[ \text{count}^2 \times 2^{\text{count}} \text{ is } O(\text{count}^2) \]

Examples

\textit{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.

\textit{force} of ARRAY: \( O(\text{count}) \)

At worst the time for this operation grows proportionally to the number of elements in the array.
Why neglect constant factors?

Consider algorithms with complexity

- $O(n)$
- $O(n^2)$
- $O(2^n)$

Assume your new machine (Christmas is coming!) is 1000 times faster?

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

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 singly-linked list operations

<table>
<thead>
<tr>
<th>Operation</th>
<th>Feature</th>
<th>Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Insert right to cursor</td>
<td><code>put_right</code></td>
<td>$O(1)$</td>
</tr>
<tr>
<td>Insert at end</td>
<td><code>extend</code></td>
<td>$O(\text{count})$ $O(1)$</td>
</tr>
<tr>
<td>Remove right neighbor</td>
<td><code>remove_right</code></td>
<td>$O(1)$</td>
</tr>
<tr>
<td>Remove at cursor position</td>
<td><code>remove</code></td>
<td>$O(\text{count})$</td>
</tr>
<tr>
<td>Index-based access</td>
<td><code>i_th</code></td>
<td>$O(\text{count})$</td>
</tr>
<tr>
<td>Search</td>
<td><code>has</code></td>
<td>$O(\text{count})$</td>
</tr>
</tbody>
</table>
Introduction to Programming, lecture 13: Inheritance & Genericity

Cost of doubly-linked list operations

<table>
<thead>
<tr>
<th>Operation</th>
<th>Feature</th>
<th>Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Insert right to cursor</td>
<td>put_right</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>Insert at end</td>
<td>extend</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>Remove right neighbor</td>
<td>remove_right</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>Remove at cursor position</td>
<td>remove</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>Index-based access</td>
<td>$i_{th}$</td>
<td>$O(\text{count})$</td>
</tr>
<tr>
<td>Search</td>
<td>has</td>
<td>$O(\text{count})$</td>
</tr>
</tbody>
</table>

Cost of array operations

<table>
<thead>
<tr>
<th>Operation</th>
<th>Feature</th>
<th>Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Index-based access</td>
<td>item</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>Index-based replacement</td>
<td>put</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>Index-based replacement outside of current bounds</td>
<td>force</td>
<td>$O(\text{count})$</td>
</tr>
<tr>
<td>Search</td>
<td>has</td>
<td>$O(\text{count})$</td>
</tr>
<tr>
<td>Search in sorted array</td>
<td>-</td>
<td>$O(\log \text{count})$</td>
</tr>
</tbody>
</table>

Hash tables

Both arrays and hash tables are indexed structures; item manipulation requires an index or, in case of hash tables, a key. Unlike arrays hash tables allow keys other than integers.
An example

```lisp
person, person1 : PERSON
personnel_directory : HASH_TABLE[PERSON, STRING]
create personnel_directory.make(100)

Storing an element:
create person1
personnel_directory.put(person1, "Annie")

Retrieving an element
person := personnel_directory.item("Annie")
```

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 \( K \).
Whenever two different keys give the same hash value a collision occurs.

Collision handling

Open hashing:
```
ARRAY[LINKED_LIST[6]]
```
A better technique: closed hashing

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:

![Diagram of closed hashing]

Closed hashing

If the hash function yields an already occupied position, the mechanism will try a succession of other positions 
\((i_1, i_2, i_3)\) until it finds a free one:

![Diagram of closed hashing sequence]

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

Cost of hash table operations

<table>
<thead>
<tr>
<th>Operation</th>
<th>Feature</th>
<th>Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Key-based access</td>
<td>item</td>
<td>(O(1)) (O(\text{count}))</td>
</tr>
<tr>
<td>Key-based insertion</td>
<td>put, extend</td>
<td>(O(1)) (O(\text{count}))</td>
</tr>
<tr>
<td>Removal</td>
<td>remove</td>
<td>(O(1)) (O(\text{count}))</td>
</tr>
<tr>
<td>Key-based replacement</td>
<td>replace</td>
<td>(O(1)) (O(\text{count}))</td>
</tr>
<tr>
<td>Search</td>
<td>has</td>
<td>(O(1)) (O(\text{count}))</td>
</tr>
</tbody>
</table>
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 \( \rightarrow \) stack
- First In First Out (FIFO): choose the oldest element not yet removed \( \rightarrow \) queue.
- Priority queue: choose the element with the highest priority.

Stacks

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

- Push an item to the top of the stack \( (\text{push}) \)
- Pop the top element \( (\text{remove}) \)
- Access the top element \( (\text{item}) \)
An example: Polish expression evaluation

from
until
"All terms of Polish expression have been read"
loop
"Read next term x in Polish expression"
if "x is an operand" then
    s.put(x)
else -- x is a binary operator
    -- Obtain and pop two top operands:
    op1 := s.item; s.remove
    op2 := s.item; s.remove
    -- Apply operator to operands and push result:
    s.put(application(x, op2, op1))
end
end

Applications of stacks

Many!

Ubiquitous in programming language implementation:
- Parsing expressions (see next)
- Managing execution of routines ("THE stack")
  Special case: implementing recursion
- Traversing trees
- …

Evaluating 2 a b + c d - * +

2
a
2
b
2
2
(a+b)
2
(c-d)
2
(a+b)^2(c-d)
2
2
2
2

2+(a+b)^2(c-d)
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).

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
What we have seen

Container data structures: basic notion, key examples

Algorithm complexity ("Big-O")

How to choose a particular kind of container

End of lecture 16