Topics for this lecture

Containers and genericity
Container operations
Lists
Arrays
Assessing algorithm performance: Big-O notation
Hash tables
Stacks and queues
Introduction to Programming, lecture 18: Container data structures

The ultimate question

LINKED_LIST or ARRAY?

Container data structures

Contain other objects ("items")

Possible operations on a container:
- Insert an item
- Find out if an element is present
- Remove an item
- "Traverse" the structure to apply an operation to every item
- ...

The way in which a container stores its items determines the required storage space and the operation speed.

A familiar container: list

A familiar container: list
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)`

Containers and generality

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

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

Container data structures are typically implemented as generic classes.

```
LINKED_LIST [G] l1 : LINKED_LIST [PERSON]
l2 : LINKED_LIST [STRING]
l3 : LINKED_LIST [ANY]
```

Lists

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

The cursor ranges from 0 to count + 1:

\[ 0 \leq index \leq count + 1 \]

If the cursor is at position 0 before is True:

\[ \text{before} = (\text{index} = 0) \]

If the cursor is at position count + 1 after is True:

\[ \text{after} = (\text{index} = \text{count} + 1) \]

In an empty list the cursor is at position 0:

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

A specific implementation: (singly) linked lists

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

Do remove as exercise.
Inserting at the end: extend

Arrays

An array is a container storing items in contiguous memory locations. Each memory location is identified by an integer index.

Bounds and indexes

Arrays are bounded:

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

The capacity of an array is:

capacity = upper - lower + 1

The number of array items ranges from 0 to capacity:

0 <= count <= capacity

An empty array has no elements:

is_empty = (count = 0)
Accessing and modifying array items

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

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.
Resizing usually requires reallocating the array and copying the old values. Such operations are costly!

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!

Big-O notation

\[ f(n) \in O(g(n)) \text{ means there exists a constant } k \text{ such that for all } n: \]
\[ \frac{|f(n)|}{g(n)} \leq k \]

Provides measure as function of size (count) of data structure.
Defines function not by exact formula but by order of magnitude, e.g. \( O(1), O(\log \text{count}), O(\text{count}), O(\text{count}^2), O(2^{\text{count}}) \).

\( \text{count}^2 + 2\text{count} + 7 \in O(\text{count}^2) \)

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: \( O(\text{count}) \)

At worst the time for this operation grows proportionally to the number of elements in the array.
### 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>put_right</td>
<td>(O(1))</td>
</tr>
<tr>
<td>Insert at end</td>
<td>extend</td>
<td>(O(\text{count}))</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(\text{count}))</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 (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</td>
<td>force</td>
<td>(O(\text{count}))</td>
</tr>
<tr>
<td>Index-based replacement</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

helen: PERSON
personnel_directory: HASH_TABLE [PERSON, STRING]
create personnel_directory.make (100)

Storing an element
create helen
personnel_directory.put (helen, "Marie-Helene")

Retrieving an element
helen := personnel_directory.item ("Marie-Helene")

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[G]]**

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:

Closed hashing continued

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

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)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>O(count)</td>
</tr>
<tr>
<td>Key-based insertion</td>
<td>put, extend</td>
<td>O(1)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>O(count)</td>
</tr>
<tr>
<td>Removal</td>
<td>remove</td>
<td>O(1)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>O(count)</td>
</tr>
<tr>
<td>Key-based replacement</td>
<td>replace</td>
<td>O(1)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>O(count)</td>
</tr>
<tr>
<td>Search</td>
<td>has</td>
<td>O(1)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>O(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 → stack.
- First In First Out (FIFO): choose the oldest element not yet removed → 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 (put)
- Pop the top element (remove)
- Access the top element (item)
Using stacks

```
from until
loop
  "All terms of Polish expression have been read"
  "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
```

Evaluating 2 a b + c d - * +

```
2 \ a \ a \ (a+b) \ c
\b \ a \ (a+b)
\ 2 \ 2 \ 2 \\
\ d
\c \ (c-d)
\(a+b) \ (a+b)
\(a+b)*(c-d)
\ 2 \ 2 \ 2 \\
2+(a+b)*(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.

Use linked lists:
- for data structures that are unbounded
- when sequencing of elements is vital
- when elements are mainly accessed in the given sequence

Use arrays:
- for data structures that are bounded
- when elements have an integer index
- when elements are mainly accessed based on their indexes

Use hash table:
- when elements have an associated key
- when elements are mainly accessed based on their keys
- (rather) for data structures that are bounded

Use stacks:
- when there is need for a LIFO dispenser

Use queues:
- when there is need for a FIFO dispenser
End of lecture 18