Introduction to Programming

Bertrand Meyer

Introduction to Programming

Piotr Nienaltowski

Lecture 18: Container data structures

Last revised 11 January 2006

Topics for this lecture

- Containers and generality
- Container operations
- Linked lists
- Arrays
- Assessing algorithm performance: Big-O notation
- Hash table
- Stack
Container data structures

- Contain other objects ("items")
- Possible operations on a container:
  - Insert an item
  - Find out if an element is present
  - Remove an element
  - "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 container operation speed.

A familiar container: list

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.
  
  \texttt{LINKED\_LIST [\textit{G}]} \texttt{LINKED\_LIST \textit{[PERSON]}} \texttt{LINKED\_LIST \textit{[STRING]}} \texttt{LINKED\_LIST \textit{[ANY]}}
A standardized naming schema

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)

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:
  \[ 0 \leq \text{index} \leq \text{count} + 1 \]
- 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: linked lists

Adding a cell

The corresponding command

```
put_right(v: A)  -- Add v to right of cursor position; do not move cursor.
    require
        not after; not after
    local
        p: LINKABLE(p)
    do
        create p.make(v)
        if before then
            p.put_right(first_element)
            first_element = p
            active = p
        else
            p.put_right(active
            active = p
        end
    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 an 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.

```
lower    item (4)     upper
1        2            3        4        5        6        7

Valid index values
```

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

```
item (i) : INTEGER; 6

-- 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 number of items.
- Unlike most other programming languages, Eiffel allows resizing an array (\texttt{resize}).
- Feature \texttt{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

$f$ is $O(g(n))$ means there exists a constant $K$ such that for all $n$:

$$\frac{|f(n)|}{g(n)} \leq K$$

- Provides the measure as a function of the size ($count$) of the data structure.
- Defines the function not by an exact formula but by an order of magnitude ("$O$ of $count$").

Some 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.

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.
**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 interval value for every element of \( K \).
- Whenever two different keys give the same hash value, a collision occurs.

---

**Collision handling**

Open hashing:

\[ \text{ARRAY[LINKED_LIST[6]]} \]

---

**A better technique: closed hashing**

EiffelBase HASH_TABLE implements closed hashing:

\( \text{HASH_TABLE} \) uses a single \( \text{ARRAY[6]} \) to store the items. At any time some of the positions are occupied and some free:

<table>
<thead>
<tr>
<th>Capacity</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>Occupied</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Free</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Closed hashing continued

If the hash function yields an already occupied position, the mechanism will try a succession of other positions \((j, j2, j3)\) 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 \(O(1)\).

An example

```helen
helen: PERSON
personnel_directory: HASH_TABLE(PERSON, STRING)

create helen
create personnel_directory, make(100)

personnel_directory, put(helen, "Helen")

personnel_directory, item("Helen")
```

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): chose the element inserted most recently → stack.
  - First-In First-Out (FIFO): chose the oldest element not yet removed → queue.
  - Priority queue: chose the element with 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 (push)
- Pop the top element (remove)
- Access the top element (item)

Using stacks

```
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 the 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 $2 \text{ a b + c d - +}$

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

4

5

6
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.

Cost of linked list operations

<table>
<thead>
<tr>
<th>Operation</th>
<th>Feature</th>
<th>Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Insert at cursor position</td>
<td><code>put_right</code></td>
<td>O (1)</td>
</tr>
<tr>
<td>Remove at cursor position</td>
<td><code>remove</code></td>
<td>O (1)</td>
</tr>
<tr>
<td>Insertion at end</td>
<td><code>extend</code></td>
<td>O (1) or O (count)</td>
</tr>
<tr>
<td>Search</td>
<td><code>has</code></td>
<td>O (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(count)</td>
</tr>
<tr>
<td>Search</td>
<td>has</td>
<td>O(count)</td>
</tr>
</tbody>
</table>

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>Key-based insertion</td>
<td>put, extend</td>
<td>O(count)</td>
</tr>
<tr>
<td>Key-based replacement</td>
<td>replace</td>
<td>O(1)</td>
</tr>
<tr>
<td>Removal</td>
<td>remove</td>
<td>O(1)</td>
</tr>
</tbody>
</table>

An overview

- Use linked list:
  - 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
An overview

- Use hash table:
  - for data structures that are bounded
  - when elements have an associated key
  - when elements are mainly accessed based on their keys
- Use stacks:
  - when there is the need of a LIFO dispenser

End of lecture 18