Introduction to Programming

Bertrand Meyer

Last revised 1 December 2003
Lecture 13: Container data structures
Container data structures

- Used to contain other objects ("items")
- Example: a metro line is — among other things — a container of stops
- What you can do with 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
- Many kinds of containers: lists (incl. "linked list", "doubly-linked lists"), circular lists, arrays, stacks, queues, priority queues, hash tables...
A basic issue with containers

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

- Metro line: list of stops
- Route: list of segments
- Phone directory: list of directory entries
- Agenda: list of appointments
- ...
Without genericity

class METRO_LINE feature
   start is do ... end
   forth is do ... end
   item: METRO_STOP is do ... end
   put_right (x: METRO_STOP) is
      -- Add x right of cursor.
      do
         ... Something involving x ...
      end
   extend (x: METRO_STOP) is do ... end
      ...
end
Without genericity

class ROUTE feature
  start is do ... end
  forth is do ... end
  item: SEGMENT is do ... end
  put_right (x: SEGMENT) is
    -- Add x right of cursor.
    do
      ... Something involving x ...
    end
  extend (x: SEGMENT) is do ... end
    ...
end
Without genericity

class AGENDA feature
  start is do ... end
  forth is do ... end
  item: APPOINTMENT is do ... end
  put_right (x: APPOINTMENT) is
    -- Add x right of cursor.
      do
        ... Something involving x ...
      end
  extend (x: APPOINTMENT) is do ... end
    ...
end
A non-generic linked list class

class \textit{LIST1} feature
\begin{align*}
\textit{start} & \text{ is do } \ldots \text{ end} \\
\textit{forth} & \text{ is do } \ldots \text{ end} \\
\textit{item: ANY} & \text{ is do } \ldots \text{ end} \\
\textit{put\_right} (x: \textit{ANY}) & \text{ is } \\
\text{ do } & \text{ -- Add } x \text{ right of cursor.} \\
\text{ do } & \text{ -- Something involving } x \ldots \\
\text{ end } & \text{ end} \\
\textit{extend} (x: \textit{ANY}) & \text{ is do } \ldots \text{ end} \\
\end{align*}
Using the general list

\[
\begin{align*}
\text{my\_route}: & \text{ LIST1} & \text{seg}: & \text{SEGMENT} \\
\text{my\_agenda}: & \text{ LIST1} & \text{app}: & \text{APPOINTMENT}
\end{align*}
\]

\[
\begin{align*}
\text{my\_route}.\text{extend} & (\text{seg}) \\
\text{my\_agenda}.\text{extend} & (\text{app})
\end{align*}
\]

\[
\begin{align*}
\text{seg} & := \text{my\_route}.\text{item} \\
\text{app} & := \text{my\_agenda}.\text{item}
\end{align*}
\]

\[
\begin{align*}
\text{app} & := \text{my\_route}.\text{item} \quad -- \ ???? ??????
\end{align*}
\]
- Repeat code (not really acceptable)

- Allow conversions, or “casts”
  - Unchecked: C, C++
  - Checked: Java, C#

```plaintext
app ?= my_agenda.item
if app /= Void then
  ...
end
```

- Make type parameterization explicit (Eiffel): genericity
Solution: genericity

class LINKED_LIST [G] feature
    start is do ... end
    forth is do ... end
    item: G is do ... end
    put_right (x: G) is
        -- Add x right of cursor.
        do
            ... Something involving x ...
        end
    extend (x: G) is do ... end
        ...
end
Using the **generic list class**

```plaintext
\[
\begin{align*}
\text{my\_route: LIST [SEGMENT]} & & \text{seg: SEGMENT} \\
\text{my\_agenda: LIST [APPOINTMENT]} & & \text{app: APPOINTMENT}
\end{align*}
\]
```

```plaintext
\[
\begin{align*}
\text{my\_route.extend (seg)} \\
\text{my\_agenda.extend (app)} \\
\text{seg := my\_route.item} \\
\text{app := my\_agenda.item} \\
\text{app := my\_route.item} & -- \text{Type-wrong, rejected}
\end{align*}
\]
```
Static typing

- Every entity of the program is declared with a type
- Every assignment and feature call must satisfy type compatibility rules
- Goal: never apply to an object a feature that is not defined for that object
Errors in the software lifecycle

It’s better to catch an error early than late

Better in analysis than in design
Better in design than implementation
Better in compilation than testing
Better in testing than actual operation
A generic class: \textit{LINKED\_LIST}

- Demo (see EiffelStudio)
highest_name (line: METRO_LINE): STRING is
   -- Alphabetically greatest station name of line

require
   line_exists: line /= Void

do
   from
      fancy_line.start ; Result := ""
   invariant ... variant ...
   until
      fancy_line.after
   loop
      Result := greater (Result, line.item.name)
   end

end

ensure
   Result /= Void and then not Result.empty

end
do
  from
    fancy_line.start ; Result := ""
  until
    fancy_line.after
  loop
    Result := greater (Result, line.item.name)
    fancy_line.forth
  end
end
The routine body, version 2

local

\[ i: INTEGER \]

do

from

\[ i := 0 ; Result := "" \]

until

\[ i > n \]

loop

\[ i := i + 1 \]

\[ Result := greater (Result, line.i_{th} (i).name) \]

end

end
local

\[ i: INTEGER \]

do

from

\[ i := count + 1 ; Result := "" \]

until

\[ i = 0 \]

loop

\[ i := i - 1 \]

\[ Result := greater (Result, line.i_{th} (i).name) \]

end

end
How fast is the algorithm?

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

- But most fundamentally depends on the algorithm!
Estimating essential efficiency

How does the algorithm’s execution time (and memory occupation) behave as a function of the size \textit{count} of the data, when that size becomes large?

- Version 1: time roughly proportional to \textit{count}.
- Version 2 and 3: could be proportional to \textit{count}, or to \textit{count}^2!

\[
1 + 2 + \ldots + \text{count} = \text{count} \times (\text{count} + 1) / 2
\]
“Big O” notation

- \( O(f(n)) \), where \( n \) represents the size of the input, means “on the order of \( f(n) \)”. 
End of lecture 13