Einführung in die Programmierung
Introduction to Programming

Prof. Dr. Bertrand Meyer

Exercise Session 9
Today

- Feedback on the mock exam

- Recursion
  - Recursion
    - Recursion
      - Recursion

- Basic data structures
  - Arrays
  - Linked Lists
  - Hashtables
Recursion: an example

- Fibonacci numbers:
  0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, ...

- How can we calculate the n-th Fibonacci number?

- Recursive formula:
  
  \[ F(n) = F(n-1) + F(n-2) \text{ for } n > 1 \]
  
  with \( F(0) = 0, F(1) = 1 \)
Recursion: a second example

Another example of recursion

Source: en.wikipedia.org/wiki/Recursion
A recursive feature

fibonacci(n: INTEGER): INTEGER
    do
        if n = 0 then
            Result := 0
        elseif n = 1 then
            Result := 1
        else
            Result := fibonacci(n-1) + fibonacci(n-2)
        end
    end
The general notion of recursion

A definition for a concept is **recursive** if it involves an instance of the concept itself.

- The definition may use more than one “*instance of the concept itself*”
- *Recursion* is the use of a recursive definition
Thoughts

“To iterate is human, to recurse - divine!”

but … computers are built by humans

Better use iterative approach if reasonable?
Iteration vs. recursion

- Every recursion could be rewritten as an iteration and vice versa.
- BUT, depending on how the problem is formulated, this can be difficult or might not give you a performance improvement.
Be careful when using recursion!

Possible stack overflow detected. The application has been paused to let you examine its current status.
Exercise: Printing numbers

- If we pass $n = 4$, what will be printed?

```
print_int (n: INTEGER)
  do
    print (n)
    if n > 1 then
      print_int (n - 1)
  end
end
```

```
print_int (n: INTEGER)
  do
    if n > 1 then
      print_int (n - 1)
    end
  end
  print (n)
end

4321

1234
```
Exercise: Reverse string

- Print a given string in reverse order using a recursive function.
class APPLICATION

create
   make

feature make
   make
      local
         s: STRING
      do
         create s.make_from_string ("poldomangia")
         invert(s)
      end

   invert (s: STRING)
      require
         s \= Void
      do
         if not s.is_empty then
            invert (s.substring (2, s.count))
            print (s[1])
         end
      end
end
end
Exercise: Sequences

- Write a recursive and an iterative program to print the following:

111,112,113,121,122,123,131,132,133,
211,212,213,221,222,223,231,232,233,
311,312,313,321,322,323,331,332,333,

- Note that the recursive solution can use loops too.
Exercise: Recursive solution

cells: ARRAY [INTEGER]

handle_cell (n: INTEGER)
    local
        i: INTEGER
    do
        from
            i := 1
        until
            i > 3
        loop
            cells [n] := i
            if (n < 3) then
                handle_cell (n+1)
            else
                print (cells [1].out+cells [2].out+cells [3].out+"","")
            end
            i := i + 1
        end
    end
Exercise: Iterative solution

\begin{verbatim}
from
  i := 1
until
  i > 3
loop
  from
    j := 1
until
    j > 3
loop
  from
    k := 1
until
    k > 3
loop
  print (i.out+j.out+k.out+"","")
  k := k + 1
end
  j := j + 1
end
i := i + 1
end
\end{verbatim}
Arrays

An array is a very fundamental data-structure, which is very close to how your computer organizes its memory. An array is characterized by:

- Constant time for random reads
- Constant time for random writes
- Costly to resize (including inserting elements in the middle of the array)
- Must be indexed by an integer
- Generally very space efficient

In Eiffel the basic array class is generic, \texttt{V\_ARRAY\ [G]}.
Using Arrays

Which of the following lines are valid? Which can fail, and why?

- my_array : V_ARRAY [STRING]  
  Valid, can’t fail
- my_array ["Fred"] := "Sam"  
  Invalid
- my_array [10] + "’s Hat"  
  Valid, can fail
- my_array [5] := "Ed"  
  Valid, can fail
- my_array.force ("Constantine", 9)  
  Valid, can’t fail

Which is not a constant-time array operation?
Linked Lists

- Linked lists are one of the simplest data-structures
- They consist of linkable cells

```class LINKABLE [G]
class LINKABLE [G]

create
set_value

feature
set_value (v : G)
do
   value := v
end

value : G

set_next (n : LINKABLE[G])
do
   next := n
end

next : LINKABLE[G]
end```

Using Linked Lists

Suppose you keep a reference to only the head of the linked list, what is the running time (using big O notation) to:

- Insert at the beginning: $O(1)$
- Insert in the middle: $O(n)$
- Insert at the end: $O(n)$
- Find the length of the list: $O(n)$

What simple optimization could be made to make end-access faster?
A binary search tree is a binary tree where each node has a **COMPARABLE** value.

- Left sub-tree of a node contains only values less than the node’s value.
- Right sub-tree of a node contains only values greater than or equal to the node’s value.
Exercise: Adding nodes

- Implement command `put (n: INTEGER)` in class `NODE` which creates a new `NODE` object at the correct place in the binary search tree rooted by `Current`.

- Test your code with a class `APPLICATION` which builds a binary search tree using put and prints out the values using the traversal feature.

- Hint: You might need to adapt the traversal feature such that the values are printed out in order.
Exercise: Solution

- See code in IDE.
Exercise: Searching

- Implement feature $has \ (n: \text{INTEGER}) : \text{BOOLEAN}$ in class $\text{NODE}$ which returns true if and only if $n$ is in the tree rooted by $\text{Current}$.

- Test your code with a class $\text{APPLICATION}$ which builds a binary search tree and calls $has$. 
Exercise: Solution

- See code in IDE.