ETH zürich
Chair of Software Engineering

Объектно-Ориентированное Программирование

Проф. Мануель Маццара
Проф. Джюйонг Ли
Проф. Бертран Мейер

С материалом из курса ЭТХ Цюрих
«Введние в Программирование»

12ая Лекция: Рекурсия

(Бертран Мейер)

2

What is recursion?

Normally, when we introduce a new concept C, we define it only in terms of previously known concepts

Example: "A word is a sequence of letters"

- New concept C: word
- Definition D: "sequence of letters"
- Set of existing concepts E: {sequence, letter}

A definition is recursive if C is a member of D

Example:

- "A ***word*** is either empty, or a letter followed by a ***word***"

In fact, a recursive definition is not a definition!

3

The story of the universe*

*According to Édouard Lucas, Récréations mathématiques, Paris, 1883.
This is my translation; the original is on the next page.

In the great temple of Benares, under the dome that marks the center of the world, three diamond needles, a foot and a half high, stand on a copper base.

God on creation strung 64 plates of pure gold on one of the needles, the largest plate at the bottom and the others ever smaller on top of each other. That is the tower of Brahmâ.

The monks must continuously move the plates until they will be set in the same configuration on another needle.

The rule of Brahmâ is simple: only one plate at a time, and never a larger plate on a smaller one.

When they reach that goal, the world will crumble into dust and disappear.

4

The story of the universe*

*According to Édouard Lucas, Récréations mathématiques, Paris, 1883.

Dans le grand temple de Bénarès, sous le dôme qui marque le centre du monde, repose un socle de cuivre équipé de trois aiguilles verticales en diamant de 50 cm de haut.

A la création, Dieu enfila 64 plateaux en or pur sur une des aiguilles, le plus grand en bas et les autres de plus en plus petits. C'est la tour de Brahmâ.

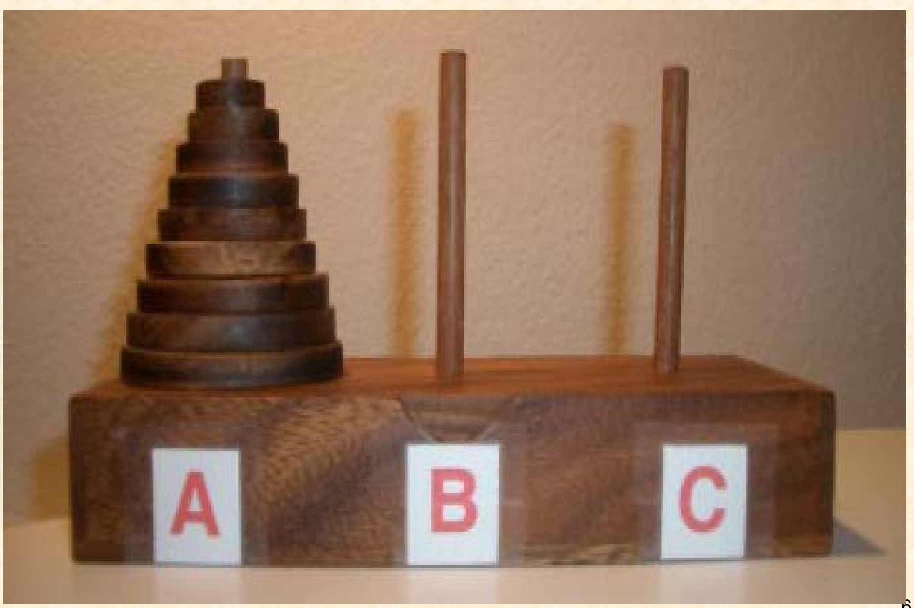
Les moines doivent continûment déplacer les disques de manière que ceux-ci se retrouvent dans la même configuration sur une autre aiguille.

La règle de Brahmâ est simple: un seul disque à la fois et jamais un grand plateau sur un plus petit.

Arrivé à ce résultat, le monde tombera en poussière et disparaîtra.

5

The towers of Hanoi



6

How many moves?

Assume *n* disks ($n \geq 0$); three needles *source*, *target*, *other*

The largest disk can only move from *source* to *target* if it's empty; all the other disks must be on *other*

So the minimal number of moves for any solution is:

Move $n - 1$ from *source* to *other*

Move *largest* from *source* to *target*

$$H_n = \boxed{H_{n-1}} + 1 + \boxed{H_{n-1}}$$

$$= 2 * \boxed{H_{n-1}} \; 1$$

Move $n-1$ from *other* to *target*

Since $H_1 = 1$, this implies:

$$H_n = 2^n - 1$$

7

This reasoning gives us an algorithm!

```
hanoi (n : INTEGER ; source, target, other : CHARACTER)
            -- Transfer n disks from source to target,
            -- using other as intermediate storage.

      require
            non_negative: n >= 0
            different1: source /= target
            different2: target /= other
            different3: source /= other
      do
            if n > 0 then
                  hanoi (n – 1, source, other, target)

                  move (source, target )

                  hanoi (n – 1, other, target, source)
            end
      end
```

Recursive calls

8

The tower of Hanoi

9

A possible implementation for *move*

```
move (source, target : CHARACTER)
            -- Prescribe move from source to target.
      require
            different: source /= target
      do
            io.put_character (source)
            io.put_string (" to ")
            io.put_character (target )
            io.put_new_line
      end
```

10

An example

Executing the call

hanoi (4, 'A', 'B', 'C')

will print out the sequence of fifteen (2^4 -1) instructions

A to *C*	*B* to *C*	*B* to *A*
A to *B*	*A* to *C*	*C* to *B*
C to *B*	*A* to *B*	*A* to *C*
A to *C*	*C* to *B*	*A* to *B*
B to *A*	*C* to *A*	*C* to *B*

11

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

12

Examples

- Recursive definition
- Recursive routine
- Recursive grammar
- Recursively defined programming concept
- Recursive data structure
- Recursive proof

13

(From inheritance lecture) what is a type?

(To keep things simple let's assume that a class has zero or one generic parameter)

A type is of one of the following two forms:

- C, where C is the name of a non-generic class
- $D\ [T\ ]$, where D is the name of a generic class and T is a type

14

Recursive routine

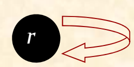
Direct recursion: body includes a call to the routine itself

Example: routine *hanoi* for the preceding solution of the Towers of Hanoi problem

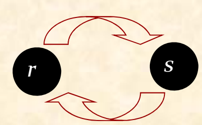
15

Recursion, direct and indirect

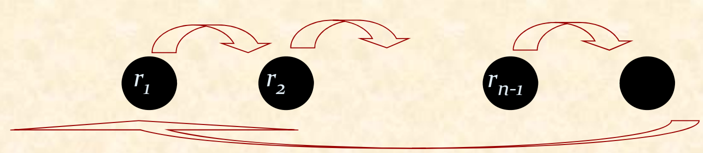
Routine *r* calls itself



r calls *s*, and *s* calls *r*



r_1 calls r_2 calls ... calls r_n calls r_1



16

Recursive grammar

Instruction ::= *Assignment* | *Conditional* | *Compound* | ...

Conditional ::= **if** *Expression* **then** *Instruction*

else *Instruction* **end**

17

Defining lexicographical order

Problem: define the notion that word *w1* is "**before**" word *w2*, according to alphabetical order.

Conventions:

- A **word** is a sequence of zero or more letters.
- A **letter** is one of:
  A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
- For any two letters it is known which one is "**smaller than**" the other; the order is that of this list.

18

Examples

ABC	before *DEF*
AB	before *DEF*
empty word	before *ABC*
A	before *AB*
A	before *ABC*

19

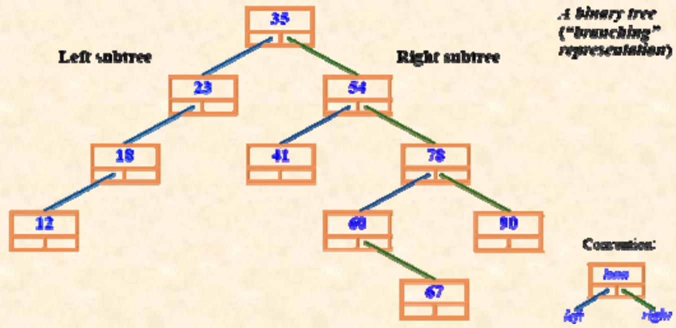
A recursive definition

The word x is "**before**" the word y if and only if one of the following conditions holds:

- x is empty and y is not empty
- Neither x nor y is empty, and the first letter of x is **smaller** than the first letter of y
- Neither x nor y is empty and:
  - Their first letters are the same
  - The word obtained by removing the first letter of x is before the word obtained by removing the first letter of y

20

Recursive data structure



A binary tree over a type G is either:

- Empty
- A node, consisting of three disjoint parts:
  - A value of type G the root
  - A binary tree over G, the left subtree
  - A binary tree over G, the right subtree

21

Nodes and trees: a recursive proof

Theorem: to any node of any binary tree, we may associate a binary tree, so that the correspondence is one-to-one

Proof:

- If tree is empty, trivially holds
- If non-empty:
  - To root node, associate full tree.
  - Any other node n is in either the left or right subtree; if B is that subtree, associate with n the node associated with n in B

Consequence: we may talk of the left and right subtrees of a node

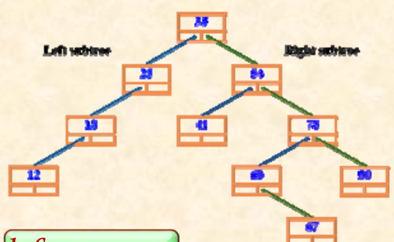
22

Binary tree class skeleton

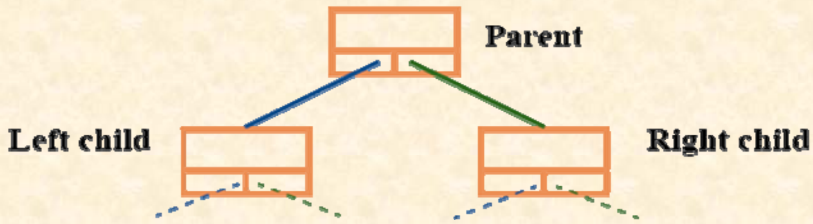
```
class BINARY_TREE [G ] feature

    item : G

    left :  BINARY_TREE  [G ]

    right : BINARY_TREE  [G ]

    ... Insertion and deletion commands ...

end
```


A recursive routine on a recursive data structure

```
count : INTEGER
        -- Number of nodes.
    do
        Result := 1
        if left /= Void then
            Result := Result + left.count
        end
        if right /= Void then
            Result := Result + right.count
        end
    end
```



Children and parents



Theorem: Single Parent

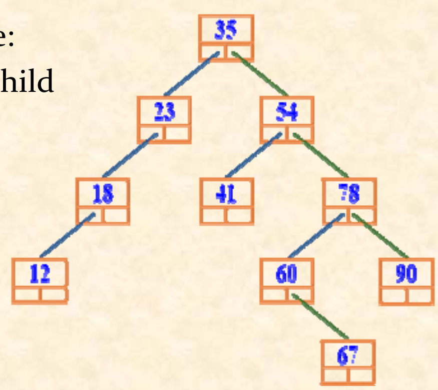
Every node in a binary tree has exactly one parent, except for the root which has no parent.

25

More binary tree properties and terminology

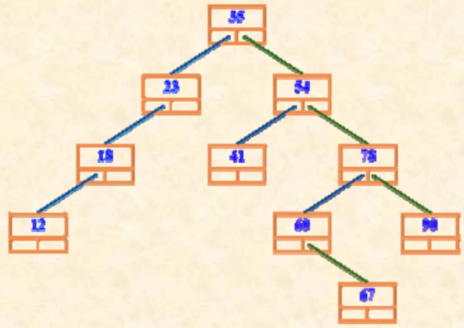
A node of a binary tree may have:

- Both a left child and a right child
- Only a left child
- Only a right child
- No child



26

More properties and terminology



Upward path:

- Sequence of zero or more nodes, where any node in the sequence is the parent of the previous one if any.

Theorem: Root Path

- From any node of a binary tree, there is a single upward path to the root.

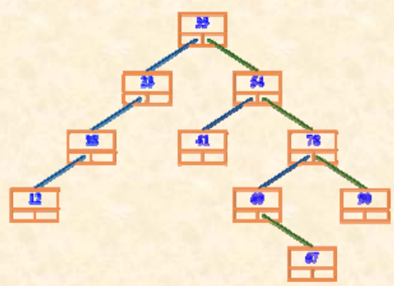
Theorem: Downward Path

- For any node of a binary tree, there is a single downward path connecting the root to the node through successive applications of *left* and *right* links.

27

Height of a binary tree

Maximum numbers of nodes on a downward path from the root to a leaf



```
height : INTEGER
        -- Maximum number of nodes
        -- on a downward path.
    local
        lh, rh : INTEGER
    do
        if left /= Void  then lh :=  left.height   end
        if right /= Void then rh := right.height end
        Result := 1 + lh.max (rh)
    end
```

28

Binary tree operations

```
add_left (x : G )
        -- Create left child of value x.
    require
        no_left_child_behind: left = Void
    do
        create left.make (x)
    end

add_right (x : G ) ...Same model...

make (x : G )
        -- Initialize with item value x.
    do
        item := x
    ensure
        set: item = x
    end
```

29

Binary tree traversals

```
print_all
        -- Print all node values.
    do
        if left /= Void then left.print_all end
        print (item)
        if right /= Void then right.print_all end
    end
```

30

Binary tree traversals

Inorder:

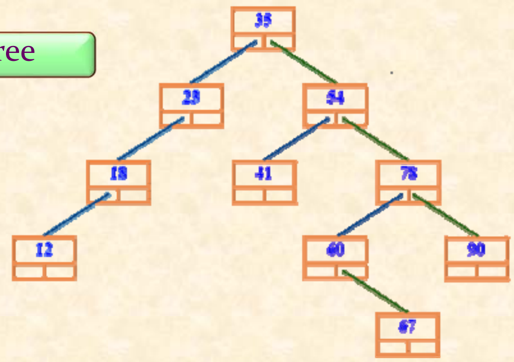
- traverse left subtree
- visit root
- traverse right subtree

Preorder:

- visit root
- traverse left
- traverse right

Postorder:

- traverse left
- traverse right
- visit root



31

Binary search tree

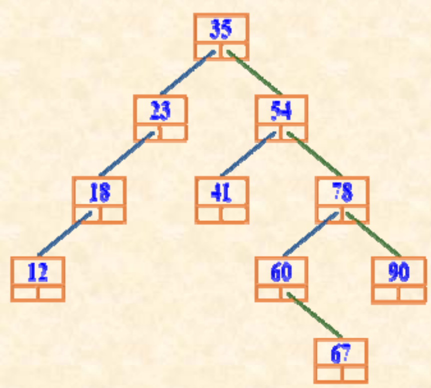
A binary tree over a sorted set *G* is a binary search tree if for every node *n* :

- For every node *x* of the left subtree of *n* :

  $x.item \leq n.item$

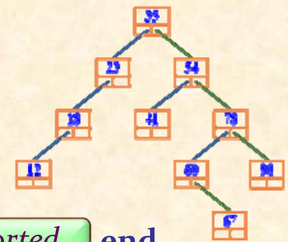
- For every node *x* of the right subtree of *n* :

  $x.item \geq n.item$



32

Printing elements in order



```
class BINARY_SEARCH_TREE [G ...] feature
    item : G
    left, right : BINARY_SEARCH_TREE [G ]

    print_sorted
            -- Print element values in order.
        do
            if left /= Void then left.print_sorted end

            print (item)

            if right /= Void then right.print_sorted end
        end
end
```

33

Searching in a binary search tree

```
class BINARY_SEARCH_TREE [G ...] feature
    item : G

    left, right : BINARY_SEARCH_TREE [G ]

    has (x : G ): BOOLEAN
                -- Does x appear in any node?
            require
                argument_exists: x /= Void
            do
                if x = item then
                    Result := True
                elseif x < item and left /= Void then
                    Result := left.has (x )
                elseif x > item and right /= Void then
                    Result := right.has (x )
                end
            end
end
```

34

Insertion into a binary search tree

Do it as an exercise!

35

Why binary search trees?

Linear structures: insertion, search and deletion are $O(n)$

Binary search tree: average behavior for insertion, deletion and search is $O(\log(n))$

But: worst-time behavior is $O(n)$!

- Improvement: Red-Black Trees

Note measures of complexity: best case, average, worst case.

36

Well-formed recursive definition

A useful recursive definition should ensure that:

- R1 There is at least one non-recursive branch
- R2 Every recursive branch occurs in a context that differs from the original
- R3 For every recursive branch, the change of context (R2) brings it closer to at least one of the non-recursive cases (R1)

37

"Hanoi" is well-formed

```
hanoi (n : INTEGER ; source, target, other : CHARACTER)
        -- Transfer n disks from source to target,
        -- using other as intermediate storage.

    require
        non_negative: n >= 0
        different1: source /= target
        different2: target /= other
        different3: source /= other
    do
        if n > 0 then
            hanoi (n – 1, source, other, target)
            move (source, target )
            hanoi (n – 1, other, target, source)
        end
    end
```

38

What we have seen so far

A definition is recursive if it takes advantage of the notion itself, on a smaller target

What can be recursive: a routine, the definition of a concept…

Still some mystery left: isn't there a danger of a cyclic definition?

39

Recursion variant

Every recursive routine should use a recursion variant, an integer quantity associated with any call, such that:

- The variant is always >= 0 (from precondition)
- If a routine execution starts with variant value v, the value v' for any recursive call satisfies

$$0 \leq v' < v$$

40

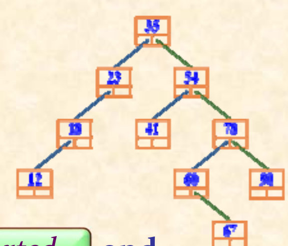
Hanoi: what is the variant?

```
hanoi (n : INTEGER ; source, target, other : CHARACTER)
        -- Transfer n disks from source to target,
        -- using other as intermediate storage.

    require
        ...
    do
        if n > 0 then

            hanoi (n – 1, source, other, target)
            move (source, target)
            hanoi (n – 1, other, target, source)

        end
    end
```

41

Printing: what is the variant?

```
class BINARY_SEARCH_TREE [G ...] feature
    item : G
    left, right : BINARY_SEARCH_TREE [G ]

    print_sorted
        -- Print element values in order.
    do
        if left /= Void then left.print_sorted end

        print (item)

        if right /= Void then right.print_sorted end
    end
end
```



42

Contracts for recursive routines

```
hanoi (n : INTEGER; source, target, other : CHARACTER)
		-- Transfer n disks from source to target,
		-- using other as intermediate storage.
		-- variant: n
		-- invariant: disks on each needle are piled in
		-- decreasing size
	require
		...
	do
		if n > 0 then
			hanoi (n – 1, source, other, target)
			move (source, target)
			hanoi (n – 1, other, target, source)
		end
	end
```


McCarthy's 91 function

$M(n) =$

- $n - 10$ if $n > 100$
- $\boxed{M}(\boxed{M}(n + 11))$ if $n \leq 100$

Another function

$bizarre\ (n) =$

- 1 if $n = 1$
- $bizarre\ (n\ /\ 2)$ if n is even
- $bizarre\ ((3 * n + 1)\ /\ 2)$ if $n > 1$ and n is odd

45

Fibonacci numbers

$fib\ (1) = 0$

$fib\ (2) = 1$

$fib\ (n) = fib\ (n - 2) + fib\ (n - 1)$ for $n > 2$

46

Factorial function

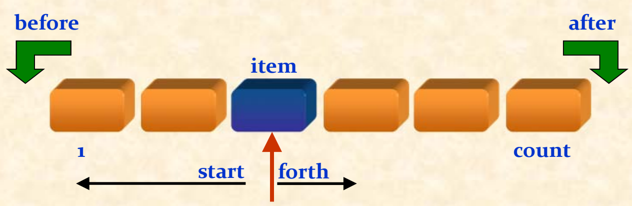
$0\,! = 1$

$n\,! = n * (n-1)\,!$ for $n > 0$

Recursive definition is interesting for demonstration purposes only; practical implementation will use loop (or table)

Our original example of a loop

```
highest_name : STRING
            -- Alphabetically greatest station name of line f.
      do
            from
                  f.start ; Result := ""
            until
                  f.after
            loop
                  Result := greater (Result, f.item.name)
                  f.forth
            end
      end
```



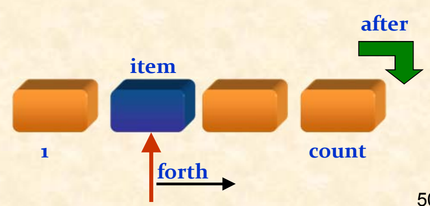
A recursive equivalent

```
highest_name : STRING
        -- Alphabetically greatest station name
        -- of line f.
    require
        not f.is_empty
    do
        f.start
        Result := f.highest_from_cursor
    end
```

49

Auxiliary function for recursion

```
highest_from_cursor : STRING
        -- Alphabetically greatest name of stations of
        -- line f starting at current cursor position.
    require
        f /= Void; not f.off
    do
        Result := f.item.name
        f.forth
        if not f.after then
            Result := greater (Result, highest_from_cursor )
        end
        f.back
    end
```



50

Loop version using arguments

```
maximum (a : ARRAY [STRING]): STRING
            -- Alphabetically greatest item in a.
      require
            a.count >= 1
      local
            i : INTEGER
      do
            from
                  i := a.lower + 1; Result := a.item (a.lower)
            invariant
                  i > a.lower ; i <= a.upper + 1
                  -- Result is the maximum element of a [a.lower .. i − 1]
            until
                  i > a.upper
            loop
                  if a.item (i ) > Result then Result := a.item (i ) end
                  i := i + 1
            end
      end
```

51

Recursive version

```
maxrec (a : ARRAY [STRING]): STRING
            -- Alphabetically greatest item in a.
      require
            a.count >= 1
      do
            Result := max_sub_array (a, a.lower)
      end

max_sub_array (a : ARRAY [STRING]; i : INTEGER): STRING
            -- Alphabetically greatest item in a starting from index i.
      require
            i >= a.lower; i <= a.upper
      do
            Result := a.item (i )
            if i < a.upper then
                  Result := greater (Result, max_sub_array (a, i + 1))
            end
      end
```

52

Recursion elimination

Recursive calls cause (in a default implementation without optimization) a run-time penalty: need to maintain stack of preserved values

Various optimizations are possible

Sometimes a recursive scheme can be replaced by a loop; this is known as recursion elimination

"Tail recursion" (last instruction of routine is recursive call) can usually be eliminated

53

Recursion elimination

r (*n*)
    do
        ... Some instructions ...

        r (*n'*)

        ... More instructions ...
    end

n := *n'*
goto *start_of_r*

-- e.g. *r* (*n* – 1)

May need *n* !

After call, need to revert to previous values of arguments and other context information

54

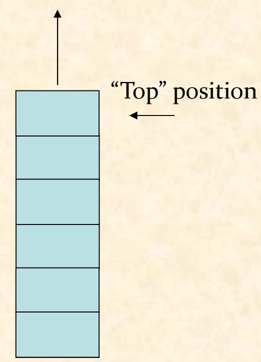
Using a stack

Queries:

- Is the stack empty? *is_empty*
- Top element, if any: *item*

Commands:

- Push an element on top: *put*
- Pop top element, if any: *remove*



Before a call: push on stack a "frame" containing values of local variables, arguments, and return information

After a call: pop frame from stack, restore values (or terminate if stack is empty)

Recursion elimination

```
r (n )
    do
start:        ... Some instructions ...

              -- r (n' )

after:        ... More instructions ...

    end
```

Push frame
n := *n'*
goto start

if *stack not empty* **then**
    Pop frame
    goto after
end

Minimizing stack use

r (*n*)
do
start: ... Some instructions ...

-- *r* (*n*')

after: ... More instructions ...

end

Push frame
n := *n*'
goto start

-- e.g. *r* (*n* – 1)

if *stack not empty* **then**
Pop frame
goto after
end

No need to store or retrieve from the stack for simple transformations, e.g. $n := n - 1$, inverse is $n := n + 1$

Recursion as a problem-solving technique

Applicable if you have a way to construct a solution to the problem, for a certain input set, from solutions for one or more smaller input sets

What we have seen

- The notion of recursive definition
- Lots of recursive routines
- Recursive data structures
- Recursive proofs
- The anatomy of a recursive algorithm: the Tower of Hanoi
- What makes a recursive definition "well-behaved"
- Binary trees
- Binary search trees
- Applications of recursion
- Basics of recursion implementation

59