ETH Zürich
Chair of Software Engineering

Einführung in die Programmierung
Introduction to Programming

Prof. Dr. Bertrand Meyer

Exercise Session 10

Today

- Basic data structures
  - Arrays
  - Linked Lists
  - Hashtables
- Another data structure: Tree

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, *ARRAY [G]*.

Using Arrays

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

- my_array : 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]

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?

Hashtables

Hashtables provide a way to use regular objects as keys (sort of like how we use INTEGER "keys" in arrays).

This is essentially a trade-off:

- We have to provide a *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.
- Our hash function should be good (minimize collisions) ☹
- Our hashtable will always take up more space than it needs to ☹

Good points about Hashtables

Hashtables aren't all that bad though, they provide us with a great solution: they can store and retrieve objects quickly by key! This is a *very* common operation.

For each of the following, define what the key and the values could be:

- A telephone book — Name → Telephone Number
- The index of a book — Concept → Page
- Google search — Search String → Websites

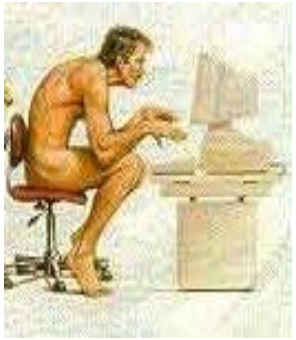
Would you use a hashtable or an array for storing the pages of a book?

Data structures

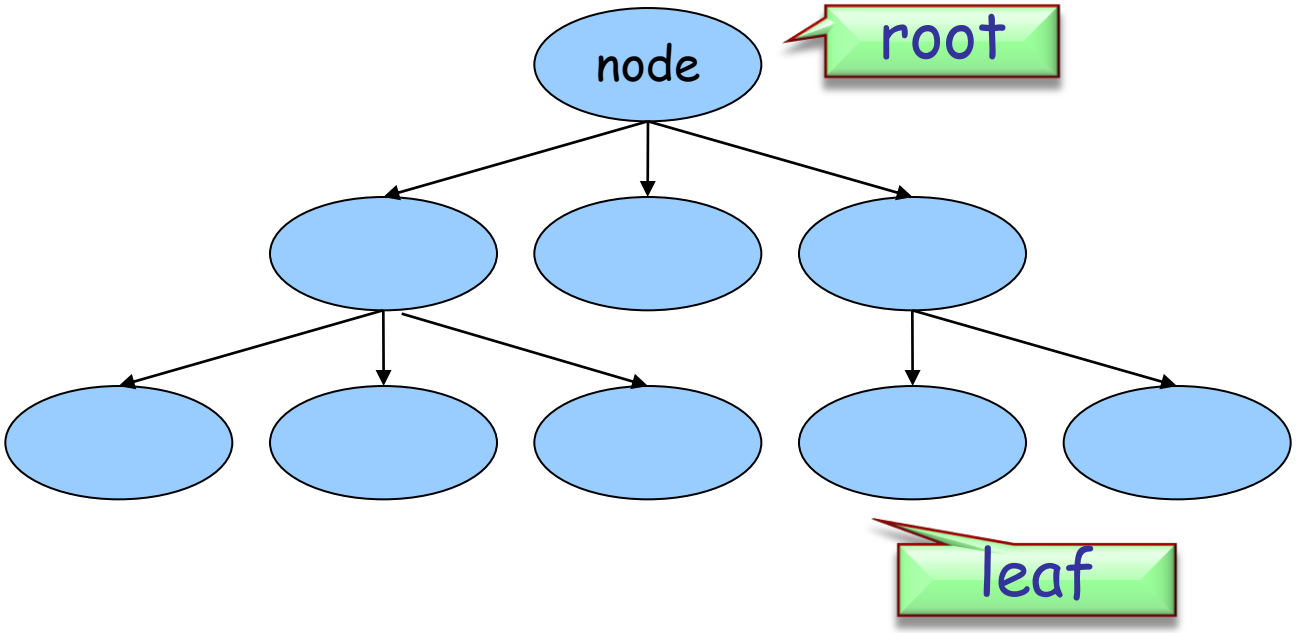
- You have seen several data structures
  - *ARRAY*, *LINKED_LIST*, *HASH_TABLE*, …
- We will now look at another data structure and see how recursion can be used for traversal.

Tree

Tree

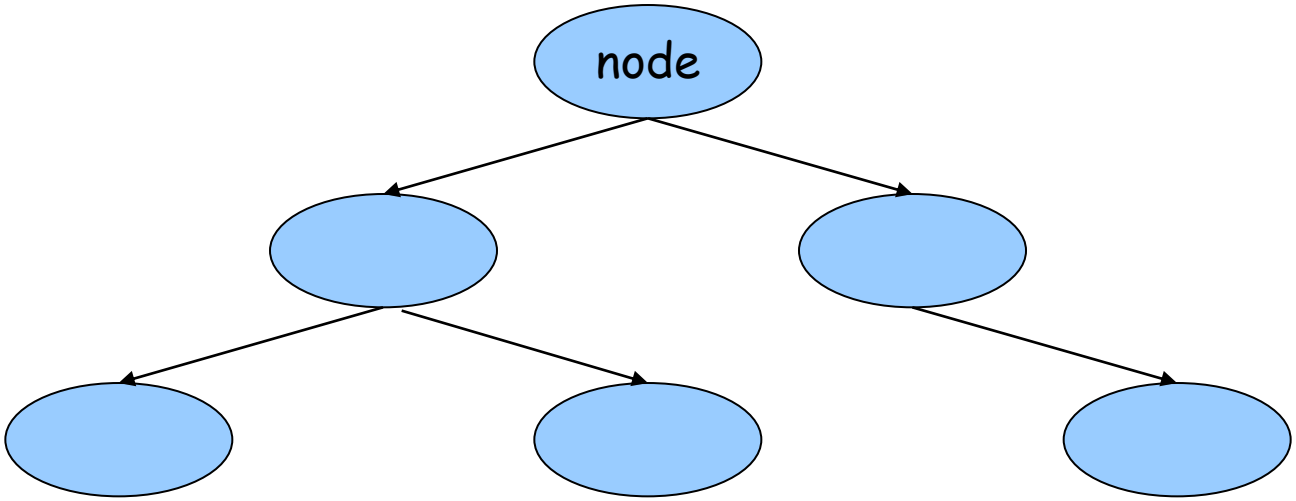


Tree: A more abstract way



- A non-empty tree has one root. An empty tree does not have a root.
- Every non-leaf node has links to its children. A leaf does not have children.
- There are no cycles.

Binary tree



- A binary tree is a tree.
- Each node can have at most 2 children (possibly 0 or 1).

Exercise: Recursive traversal

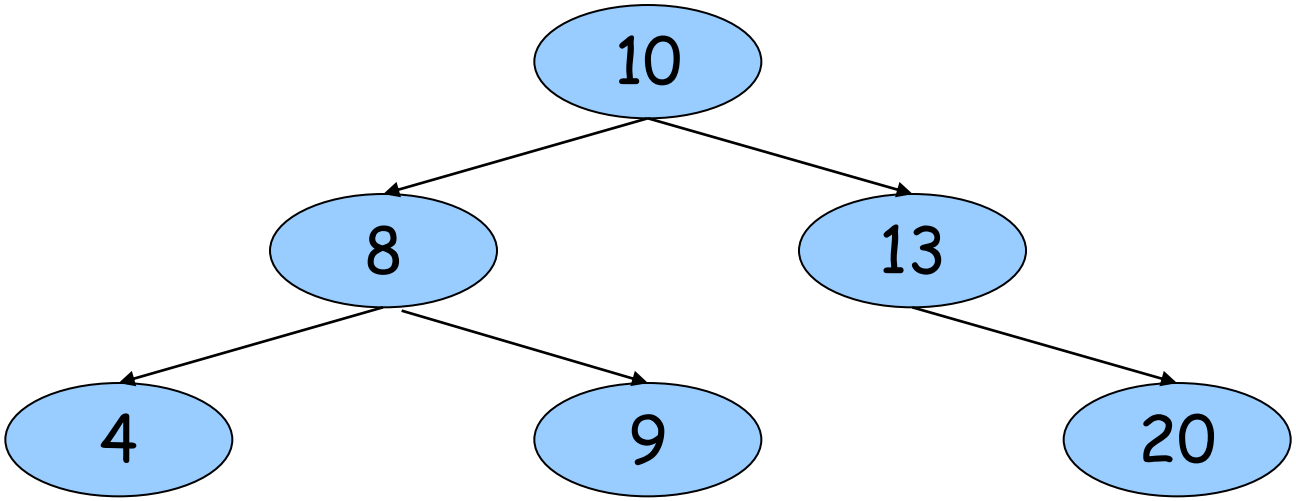
Hands-On

- Implement class *NODE* with an *INTEGER* attribute.
- In *NODE* implement a recursive feature that traverses the tree and prints out the *INTEGER* value of each *NODE* object.
- Test your code with a class *APPLICATION* which builds a binary tree and calls the traversal feature.

Exercise: Solution

- See code in IDE.

Binary search tree



- 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

Hands-On

- Implement feature *has (n: INTEGER): BOOLEAN* in class *NODE* which returns true if and only if *n* is in the tree rooted by **Current**.
- Test your code with a class *APPLICATION* which builds a binary search tree and calls *has*.

Exercise: Solution

- See code in IDE.

The End

End of slides

Time left?

Here's another recursion example ...

Exercise: Magic Squares

- A magic square of size NxN is a NxN square such that:
  - Every cell contains a number between 1 and N^2.
  - The sum in every row and column is constant.
  - The numbers are all different.

4	3	8
9	5	1
2	7	6

Exercise: Magic Squares

- Finding a 3x3 magic square is related to finding the permutations of 1 to 9.
- There exist 72 magic 3x3 squares.

123456789
123456798
123456879
123456897
123456978
123456987
...
987654321

Exercise: Magic Squares

Hands-On

- Write a program that finds all the 3x3 magic squares.

- Hints
  - Reuse the previous recursive algorithm by applying it to permutations (enforce no repetitions).
  - Use two arrays of 9 elements, one for the current permutation and one to know if a number has already been used or not.

Exercise: Solution

- See code in IDE.