

Data Structures (Not UML)

Pseudo-code Syntax, Trees

Amandine Decker & Marie Cousin

M1 TAL/SC 2023–2024

Université de Lorraine, LORIA

Class Organisation

Second Lab :

- Reminder : optional exercise TD2 (due October 27th) ;
- hand back on a sheet of paper (October 27th), send it by e-mail, or hand back on Arche.
- Correction of the second lab (except bonus exercise) will be uploaded today (October 20th)
- Points and feedback on the optional exercise of TD1 before Wednesday.

How to find the Arche repository :

- on your Arche top bar (blue-ish), click on “Home” (“Accueil”) ;
- then “LORRAINE MANAGEMENT” (purple) ;
- and search for “Data Structure” ;
- Full name of the course is “Data Structures - Beginners”, password is “Nox”.

Syntax

Variable

Variable

A **variable** is an object, or a buffer. You can create it, initialise it, modify it. It is used to store data, like a result for example.

- **Create** and **initialise** a variable : $new_var \leftarrow init_value$
- **Modify** a variable : $var \leftarrow new_value$
 - Create and initialise empty structures like last week :
 - 1: $S \leftarrow S_0$ -- creates an empty stack
 - But it can work with any type of data :
 - 1: $c \leftarrow 5$ -- creates a variable c of value 5
 - 2: $c \leftarrow 0$ -- modifies the variable c , it is now of value 0
 - 3: $c \leftarrow c + 3$ -- modifies the variable c , it is now of value 3 ($0 + 3$)

Variable

- There exists variables of any type ;
- When you create a variable, you can assign it to a value of any type :
 - Integer :
1: $var \leftarrow 5$
 - Boolean :
1: $var \leftarrow true$
2: $var \leftarrow false$
3: $iable \leftarrow var$
4: $iable \leftarrow true$
 - String :
1: $var \leftarrow$
 "*tadaaaaa*"
 - List :
1: $var \leftarrow [5, 8, 6, 9, 2]$
 - Dictionary :
1: $var \leftarrow$
 $\{(6, [12, 24]), (7, [7, 14]), (2, [6])\}$
 - Queue :
1: $var \leftarrow Q_0$
 - Stack :
1: $var \leftarrow S_0$

If Block

If/Then/Else block

If the **condition** is satisfied, instructions 1, 2, etc. are executed. If the condition is not satisfied, instructions A, B, etc. are executed.

Note : It is not mandatory to have the "else" part; in that case, when the condition is not satisfied, no instructions are executed.

```
1: if condition then  
2:   instruction 1  
3:   instruction 2  
4:   etc.  
5: else   -- optional part  
6:   instruction A  
7:   instruction B  
8:   etc.  
9: end if   -- but this one is mandatory
```

Loops (1)

For loop

In a *for loop*, we create a **variable** that will take several values one after the other, *i.e.*, we will iterate over some **elements** and the variable will take one value at a time. The loop is thus executed a **known definite number of times**. The instructions 1, 2, etc. will be executed as many times as there are elements.

Note : The loop variable (here X) can be used in the loop.

```
1: for X in elements do  
2:   instruction 1  
3:   instruction 2  
4:   etc.  
5: end for
```

Loops (2)

While loop

The *loop condition* is a condition that is true or false, it may use some variable used elsewhere in the algorithm. The instructions 1, 2, etc. will be executed *as long as* the *loop condition* remains satisfied (i.e., true).

- 1: **while** loop condition **do**
- 2: instruction 1
- 3: instruction 2
- 4: etc.
- 5: **end while**

Return instruction

This instruction returns *result*, and **ends** the algorithm.

(Note : If you encounter a **return** in the middle of an algorithm, **THE FOLLOWING INSTRUCTIONS WILL NOT BE EXECUTED!**)

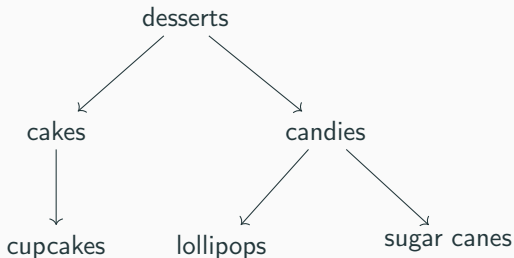
1: **return** *result*

Note : If you want to return **more than one** thing, use **comas**!

1: **return** result1, result2, result3

Trees

Trees : Idea



"I want to organise my desserts!"

- Nox wants to organise its favourite desserts ;
- In a kind of diagram, he draws each subtype of dessert below surtype.
- We have a **tree** of desserts !

Trees : Definition

Tree

A **Tree** is a data structure composed of **nodes** and **branches**.

Trees : Definition

Tree

A **Tree** is a data structure composed of **nodes** and **branches**.

- The **unique** top node is the **root**;

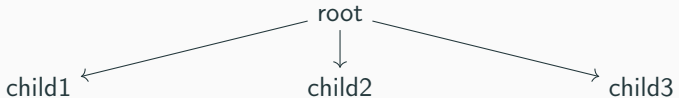
root

Trees : Definition

Tree

A **Tree** is a data structure composed of **nodes** and **branches**.

- The **unique** top node is the **root** ;
- Each node may have **children** (or descendants) nodes ;
- The parent node and its children are **linked by branches** (arrows from the parent to the child) ;

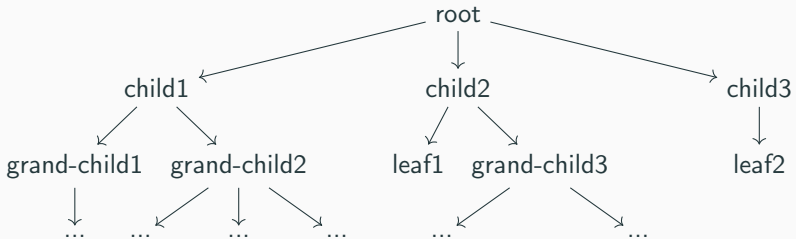


Trees : Definition

Tree

A **Tree** is a data structure composed of **nodes** and **branches**.

- The **unique** top node is the **root** ;
- Each node may have **children** (or descendants) nodes ;
- The parent node and its children are **linked by branches** (arrows from the parent to the child) ;
- A node that has no child (or descendant) is called a **leaf**.



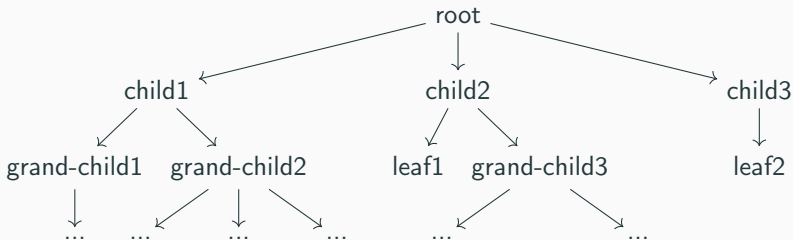
Trees : Definition

Tree

A **Tree** is a data structure composed of **nodes** and **branches**.

- The **unique** top node is the **root** ;
- Each node may have **children** (or descendants) nodes ;
- The parent node and its children are **linked by branches** (arrows from the parent to the child) ;
- A node that has no child (or descendant) is called a **leaf**.

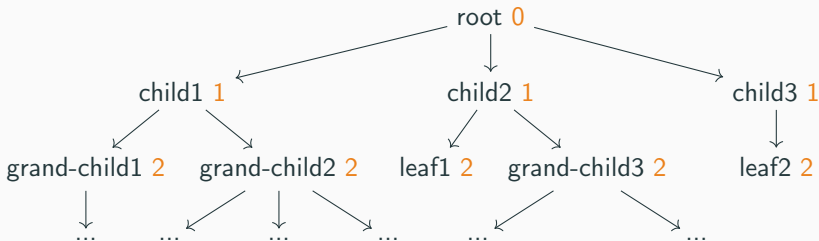
→ *Like a family tree in human life for example*



Trees : Definition

Properties

- The **root** of a tree is its unique top node ;
- The **depth** of a node is the number of ancestor it has (the depth of the root is 0) (*cf. the tree below, in orange*) ;
- The **height** of a tree is the depth of the deepest leaf ;
- If each node of a tree has 0 to 2 children, the tree is said **binary**.



Trees : Some Formalism

If we consider a set of elements E , we can write $Trees(E)$ the set of all the trees containing elements of E . The empty tree $\{ \}$ is in $Trees(E)$.

We will consider **only binary trees** in this class. A tree (or a sub-tree) is characterised by its root, its left sub-tree and its right sub-tree. We represent each tree or sub-tree by a dictionary that has 3 pairs :

$\{('root', value), ('left\ child', left\ sub-tree), ('right\ child', right\ sub-tree)\}$

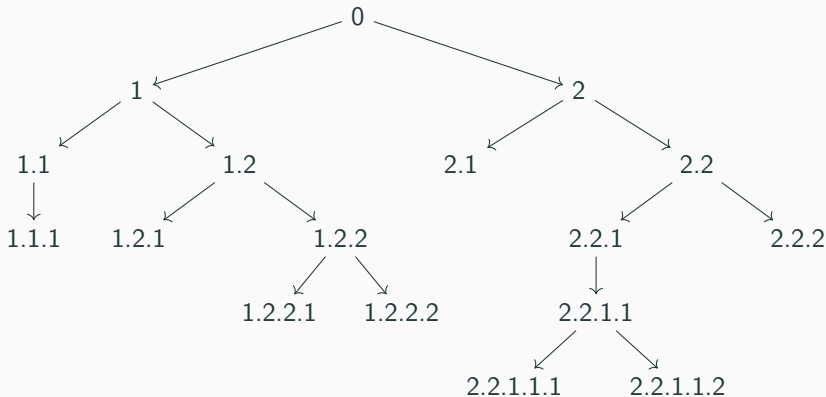
If $e \in E$, and $T \in Trees(E)$, the following **operations** exist :

Operations

- $isEmpty(T)$ returns *true* if the tree is empty, false otherwise ;
- $T['root']$ returns the value of root of the tree ;
- $T['left\ child']$ return the left sub-tree of T (it is a tree) ;
- $T['right\ child']$ return the right sub-tree of T (it is a tree).

Trees : Some Formalism

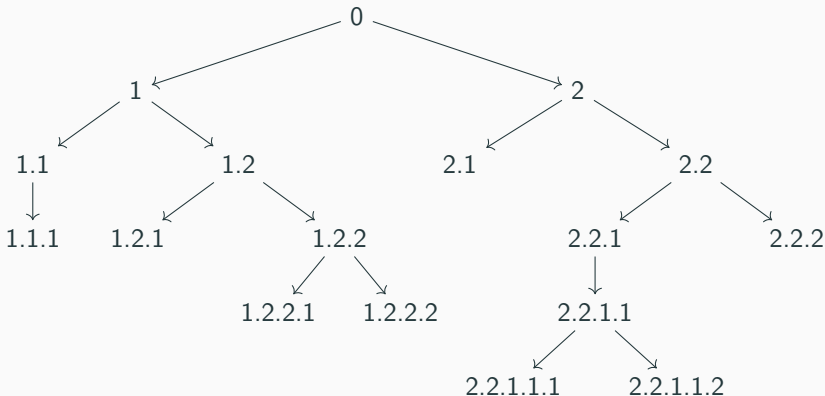
Let T be the following tree.



Trees : Some Formalism

Let T be the following tree. Then :

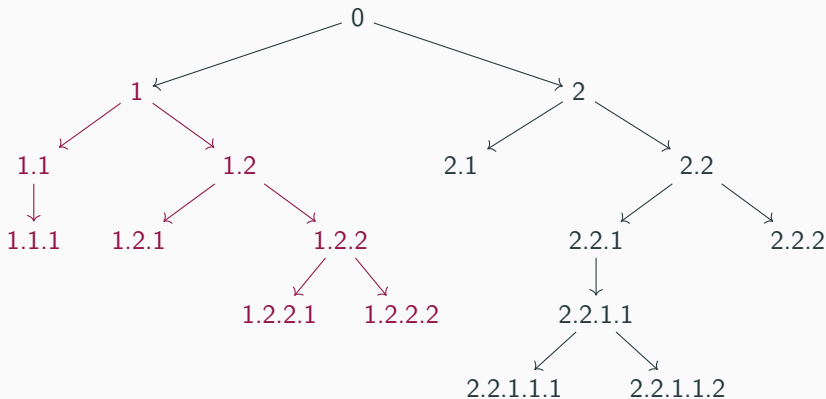
- $T[\text{'root'}] = 0$;
- The height of T is 5.



Trees : Some Formalism

Let T be the following tree. Then :

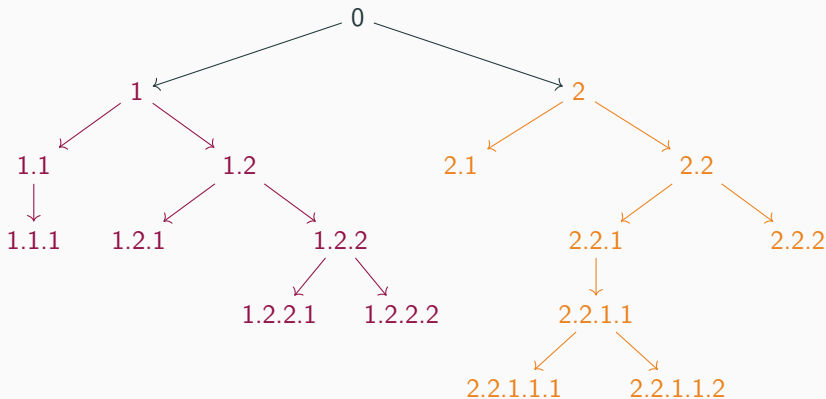
- $T[\text{'left child'}]$ will return the purple tree



Trees : Some Formalism

Let T be the following tree. Then :

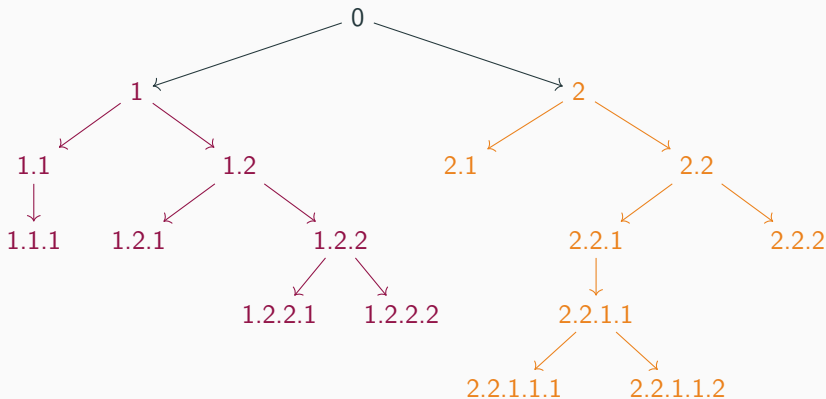
- $T[\text{'right child'}]$ will return the orange tree



Trees : Some Formalism

Let T be the following tree. Then :

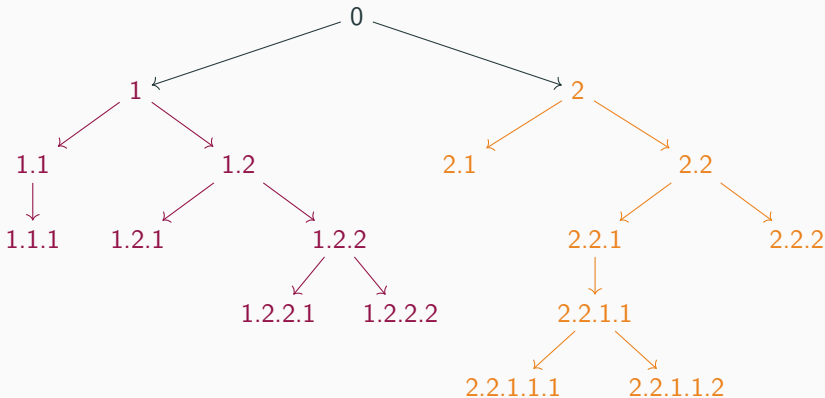
- $T[\text{'left child'}][\text{'right child'}][\text{'right child'}][\text{'root'}] = 1.2.2$



Trees : Some Formalism

Let T be the following tree. Then :

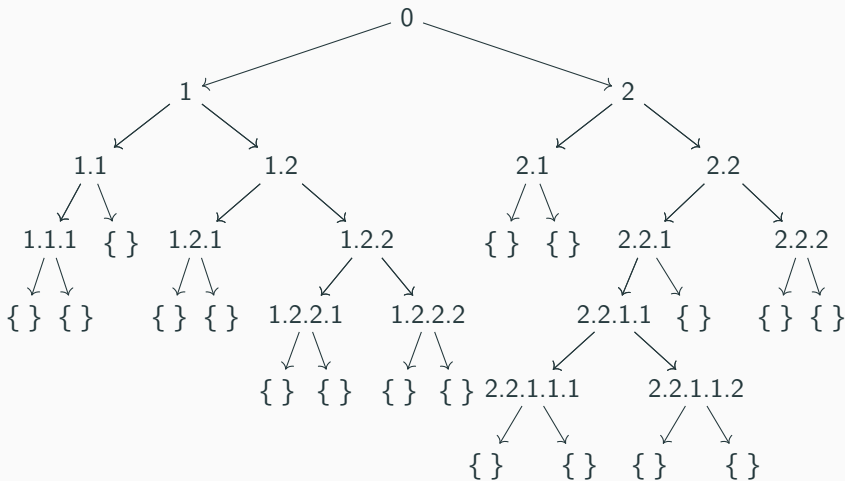
- $isEmpty(T['right\ child']['right\ child']['left\ child']['right\ child'])$
 $= true$



Trees : Some Formalism

Let T be the following tree. Then :

- $isEmpty(T['right\ child']['right\ child']['left\ child']['right\ child'])$
= *true*. Indeed, you actually encode this :



Trees : Don't panic

Regarding trees, we expect from you :

- to be able to represent a tree, either in its "drawing" form, or in its dictionary form ;
- to be able to *apply* a given algorithm on a given tree ;
- to understand a given algorithm on trees (and be able to say what it does) ;
- to be able to write a single basic instruction or a basic algorithm ("how do I get the value of *this* node in *this* tree" for instance, for a given tree and node) ;
- to give the *principle* (not the algorithm itself) of an algorithm to answer a question ;

We do not expect from you :

- to write a whole algorithm on trees.

Trees : Example

Require: T , $value$

```
1:  $S \leftarrow S_0$ 
2:  $add(T, S)$ 
3: while  $!isEmpty(S)$  do
4:    $tree \leftarrow get(S)$ 
5:    $remove(S)$ 
6:   if  $!isEmpty(tree)$  then
7:     if  $tree['root'] == value$  then
8:       return true
9:     end if
10:    if  $!isEmpty(tree['right child'])$  then
11:       $add(tree['right child'], S)$ 
12:    end if
13:    if  $!isEmpty(tree['left child'])$  then
14:       $add(tree['left child'], S)$ 
15:    end if
16:  end if
17: end while
18: return false
```

Question

What does this algorithm do?

Trees : Example

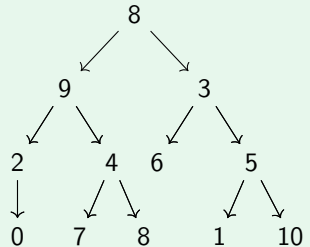
Require: T , $value$

```
1:  $S \leftarrow S_0$ 
2:  $add(T, S)$ 
3: while  $!isEmpty(S)$  do
4:    $tree \leftarrow get(S)$ 
5:    $remove(S)$ 
6:   if  $!isEmpty(tree)$  then
7:     if  $tree['root'] == value$  then
8:       return true
9:     end if
10:    if  $!isEmpty(tree['right child'])$  then
11:       $add(tree['right child'], S)$ 
12:    end if
13:    if  $!isEmpty(tree['left child'])$  then
14:       $add(tree['left child'], S)$ 
15:    end if
16:  end if
17: end while
18: return false
```

Question

What does this algorithm do?

Try with the following tree and value 5 :



Trees : Example

Require: T , $value$

```
1:  $S \leftarrow S_0$ 
2:  $add(T, S)$ 
3: while  $!isEmpty(S)$  do
4:    $tree \leftarrow get(S)$ 
5:    $remove(S)$ 
6:   if  $!isEmpty(tree)$  then
7:     if  $tree['root'] == value$  then
8:       return true
9:     end if
10:    if  $!isEmpty(tree['right child'])$  then
11:       $add(tree['right child'], S)$ 
12:    end if
13:    if  $!isEmpty(tree['left child'])$  then
14:       $add(tree['left child'], S)$ 
15:    end if
16:  end if
17: end while
18: return false
```

Question

What does this algorithm do?

It takes a tree and a value as input, and returns *true* if the input value is the value of one of the node of the input tree, and *false* otherwise.

TD1 Optional Exercise

TD1 Optional Exercise : General Feedback

- Several solutions were possible, we put one of them on Arche ;
- Almost all of you had the right idea/principle concerning your algorithm ;
- Read the question, and follow the given constraint ! If we say "this algo should use at most 2 structures" do not use more than 2 structures ;
- Use the operations and instructions given during the class (no `queue.pop()`, no `unqueue`, etc.) ;
- In general, **ALWAYS** comment your algorithm, and **ALWAYS** add a few lines explaining the principle of your algorithm ;

TD1 Optional Exercise : General Feedback

- Several algorithms had syntax problems :
 - A **break** instruction will make you go out of the current loop without taking into the consideration the loop instruction (for loop) or the loop condition (while loop). *If you put a **break** just before a **return**, you will not return for instance. Tip : do not use **break**, but define your loop instruction/condition more precisely ;*
 - If you encounter the instruction **return** while executing (or applying) your algorithm, the execution will stop. All the following instructions will not be executed. *So if you use two **return** one after the other, the second one will never be read ;*
 - If you forgot the **end if**, **end for** or **end while**, make sure you respected the indentation. If you forgot both (the **end** – and the indentation) your algorithm becomes unreadable or false ;

TD1 Optional Exercise : General Feedback

- Several algorithms had queues and stacks related problems :
 - Do not forget that a queue is FIFO, and a stack is LIFO ;
 - If Q is a queue, get(Q) will **read** the **first** element of the queue. It will NOT remove it. If you want to **remove** the element you have to use the instruction remove(Q) (Caution, remove(Q) does NOT read the element.);
 - If S is a stack, get(S) will **read** the **last** element of the stack. It will NOT remove it. If you want to **remove** the element you have to use the instruction remove(S) (Caution, remove(S) does NOT read the element.);
 - You **can't** iterate over a queue or a stack. You have to (get the element, and then) remove the element.

Summary

Summary

- **Trees** are a complex data structure ;
- They are made of **nodes** and **branches** ;
- We will consider only **binary** trees (trees such that each of their nodes has 0 to 2 children) ;
- The unique top node of a tree is the **root** and a node without any child is a **leaf** ;
- Each node has a **depth** and a tree has a **height** ;
- We have chosen to represent trees with **dictionaries** ;
- Their root, left child, and right child are accessible via the keys of the dictionary representing them ;
- Do not forget that a sub-tree is also a tree, and thus represented by a dictionary too.