CM 6 : Récursivité (2)

Info1.Algo1

2022-2023 Semestre Impair

Plan

Modification (?) d'une liste chaînée

Exemple

Écrire la fonction récursive `substituer_premiere_occurence` qui accepte en paramètres :

- une liste chaînée d'entiers `liste`
- un entier `ancienne_valeur`
- un entier `nouvelle_valeur`.

La fonction retourne une nouvelle liste dans laquelle toutes les occurrences de `ancienne_valeur` sont remplacées par `nouvelle_valeur`.

Plan

Cas général et motivation

Définition informelle

Un **arbre** est une **structure de données récursive**. Chaque élément de cette structure est appelé **nœud**. Chaque nœud est constitué :

- d'une **valeur** associée au nœud.
- d'un nombre quelconque de références vers ses **descendants** ou sous-arbres.

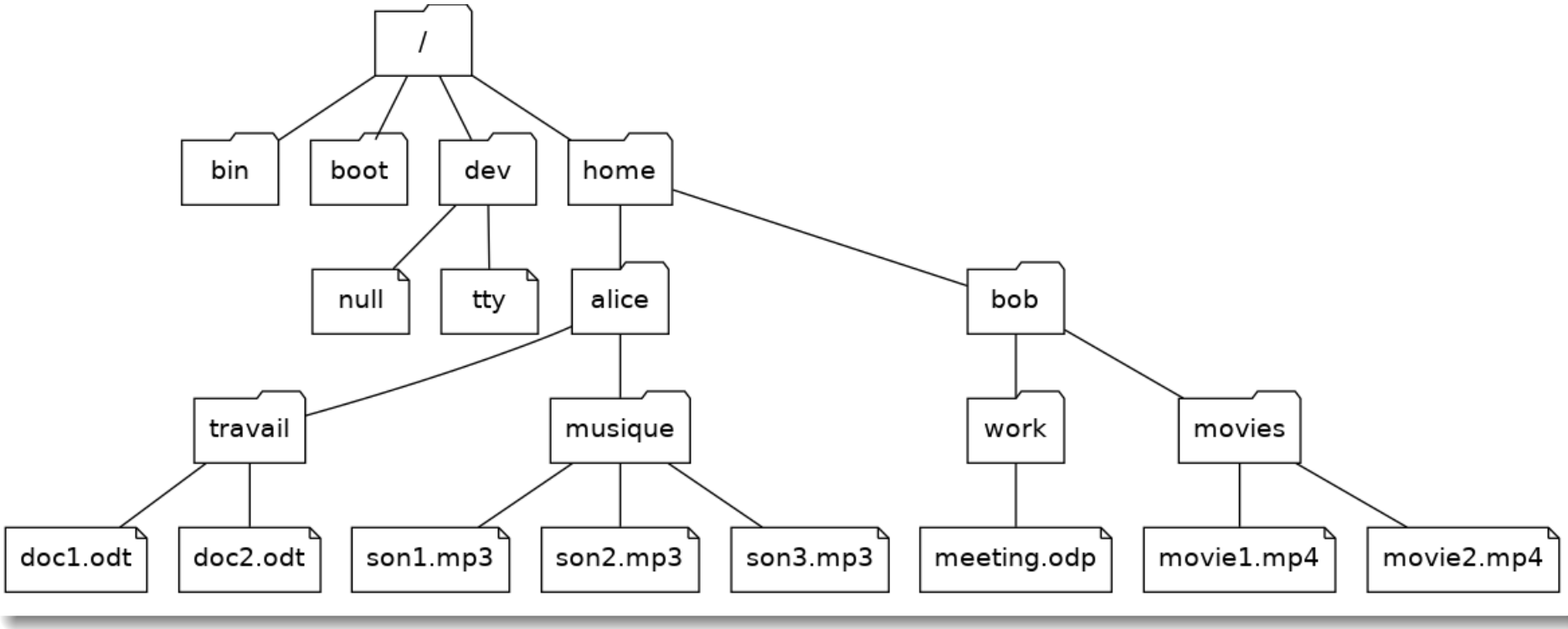
Le premier nœud (qui n'est le descendant d'aucun autre nœud) est appelé **racine** de l'arbre.

Cas général et motivations

Motivation 1 : dans un système d'exploitation

Le **système de fichiers** ainsi que l'**organisation des processus** en cours d'exécution sont tous deux représentés sous forme d'**arbre**.

Exemple 1.1 : système de fichiers



Cas général et motivations

Exemple 1.2 : organisation des processus

Sur un système Linux, les processus en cours d'exécution sont accessibles via la commande `pstree`.

```
alexander@alexander-vu:~$ pstree -np
systemd(1)─┬─systemd-journal(206)
           ├─systemd-udevd(226)
           ├─systemd-timesyn(373)───{sd-resolve}(391)
           ├─cupsd(663)─┬─dbus(739)
           │            └─dbus(740)
           ├─acpid(679)
           ├─anacron(682)
           ├─rsyslogd(688)─┬─{in:imuxsock}(726)
           │               ├─{in:imklog}(727)
           │               └─{rs:main Q:Reg}(728)
           ├─accounts-daemon(694)─┬─{gmain}(708)
           │                      └─{gdbus}(741)
           ├─snapd(696)─┬─{snapd}(722)
           │            ├─{snapd}(725)
           │            ├─{snapd}(730)
           │            └─{snapd}(734)
```

Cas général et motivations

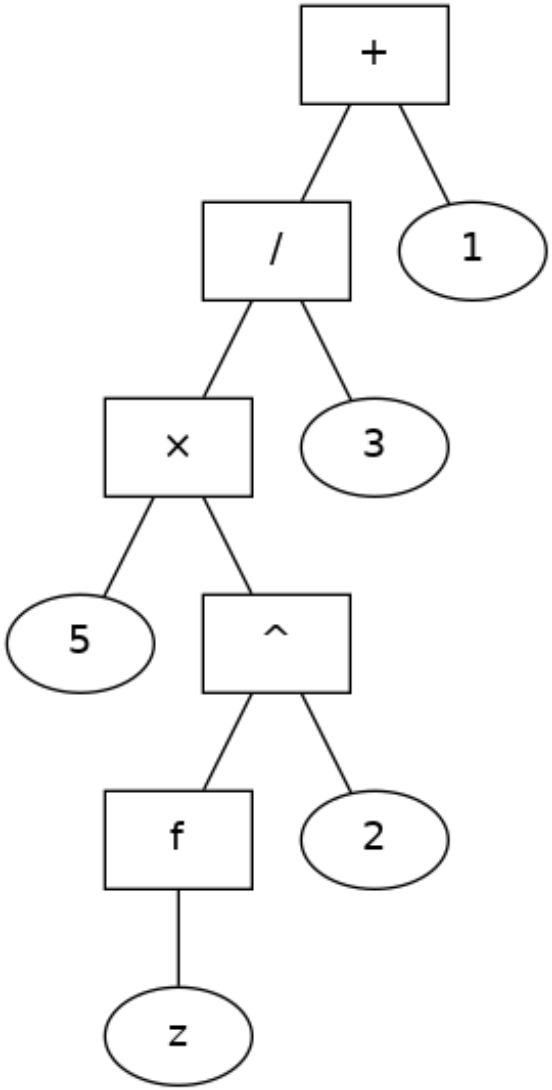
Motivation 2 : arbres syntaxiques

Lors de l'analyse syntaxique d'un langage (expressions algébriques ou logiques, langage de programmation,...), l'analyseur produit un **arbre syntaxique** qui représente la structure syntaxique de ce qui a été lu.

Exemple 2.1

L'arbre syntaxique ci-contre représente la formule mathématique :

$$\frac{5\ f(z)^2}{3} + 1$$

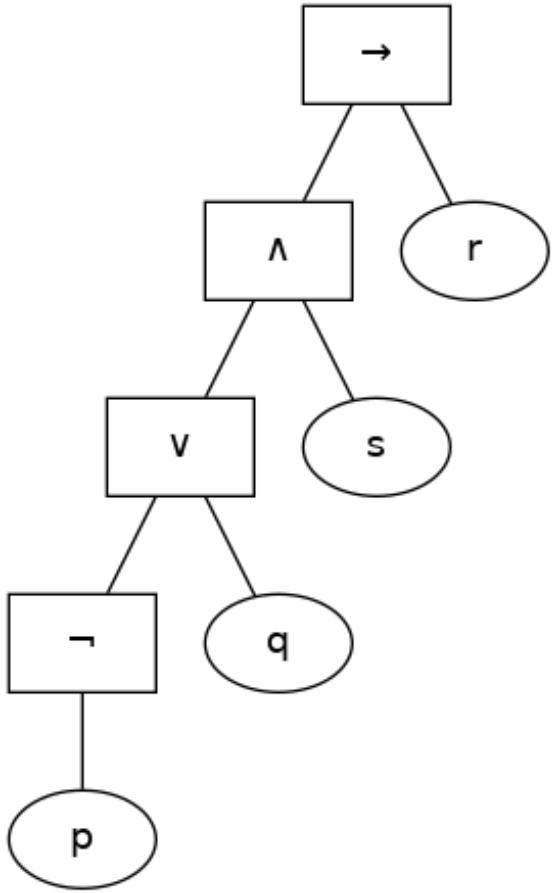


Cas général et motivations

Exemple 2.2

L'arbre syntaxique ci-contre représente la formule de la logique des propositions :

$$(\neg p \vee q) \wedge s \rightarrow r$$



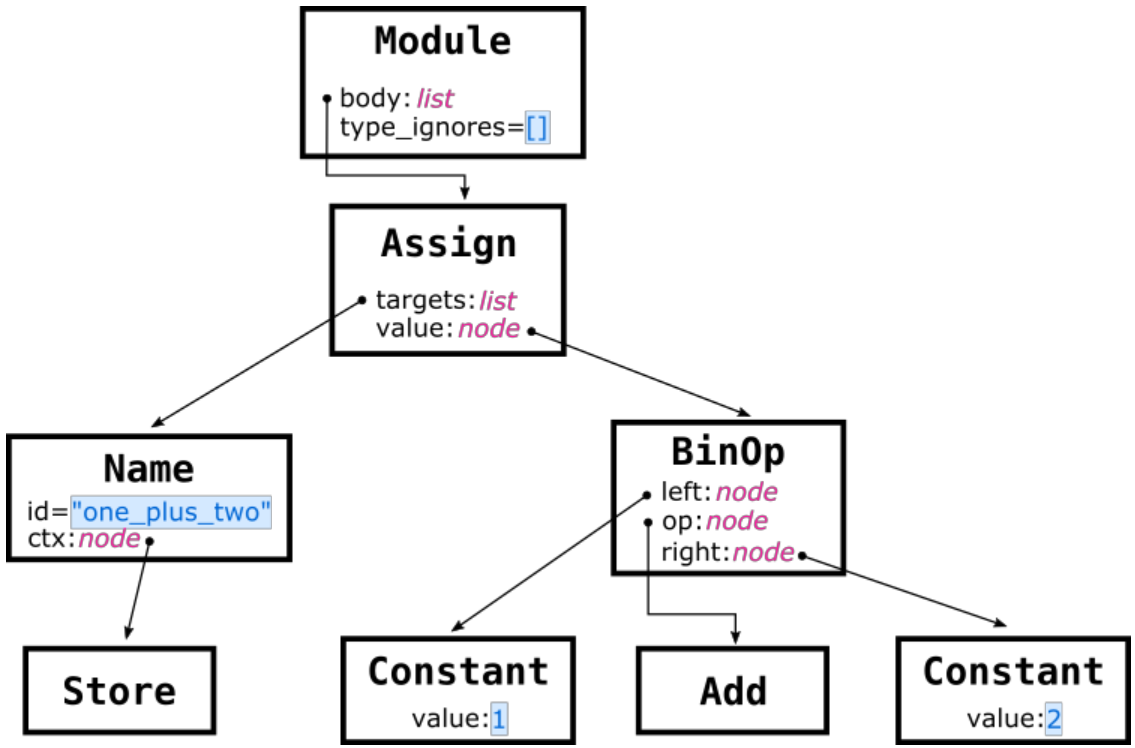
(Cf. UE Info1.DS1 - Structures discretes 1)

Cas général et motivations

Exemple 2.3

L'arbre syntaxique ci-dessous représente le code suivant :

```
1 one_plus_two = 1+2
```



Cas général et motivations

Exemple 2.3 (suite)

L'arbre syntaxique peut être obtenu avec le module ast :

```
1 import ast
2 print(ast.dump(ast.parse("one_plus_two = 1+2"),indent=4))
```

```
Module(
    body=[
        Assign(
            targets=[
                Name(id='one_plus_two', ctx=Store())],
            value=BinOp(
                left=Constant(value=1),
                op=Add(),
                right=Constant(value=2)))],
    type_ignores=[])
```

Cas général et motivations

Motivation 3 : en algorithmique

La structure de données arbre est aussi un **puissant outil** pour la résolution de certains problèmes *(non nécessairement exprimés sous forme d'arbre)*.

Problème posé	Arbre utilisé
Tri d'un tableau	Tas *(en anglais : heap)*
Représentation d'un tableau associatif	Arbre binaire de recherche Treap, AVL, Arbre bicolore B-arbre, Arbre splay...

Arbres binaires

Définition

Un **arbre binaire** est :

- soit un **arbre vide**.
- soit un **noeud** ayant **exactement 2 descendants** (**gauche** et **droite**) qui sont eux-mêmes des **arbres binaires**.

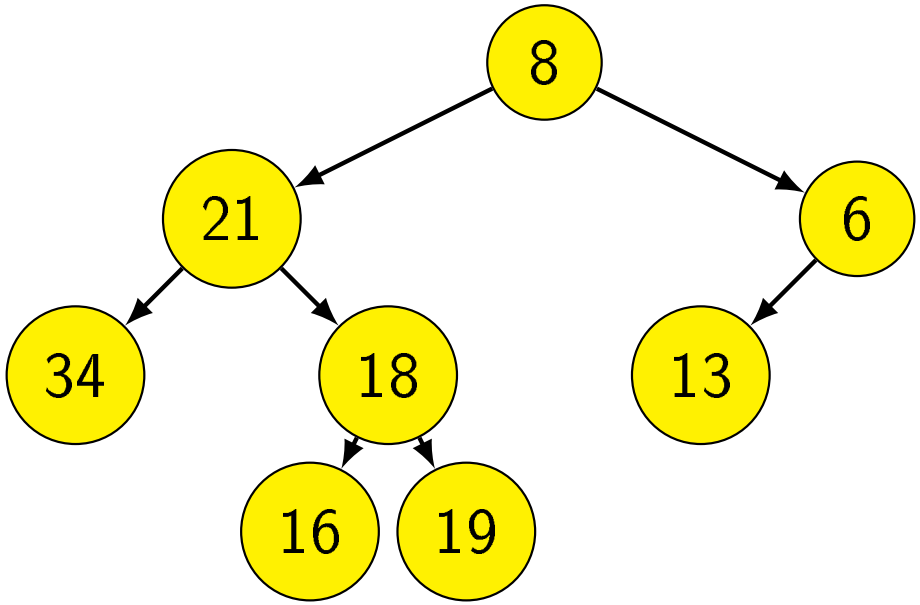
L'un ou l'autre des descendants peut être **vide**, ce qui permet de représenter :

- Les nœuds ayant **un seul vrai descendant**.
- Les **feuilles** (n'ayant aucun vrai descendant).

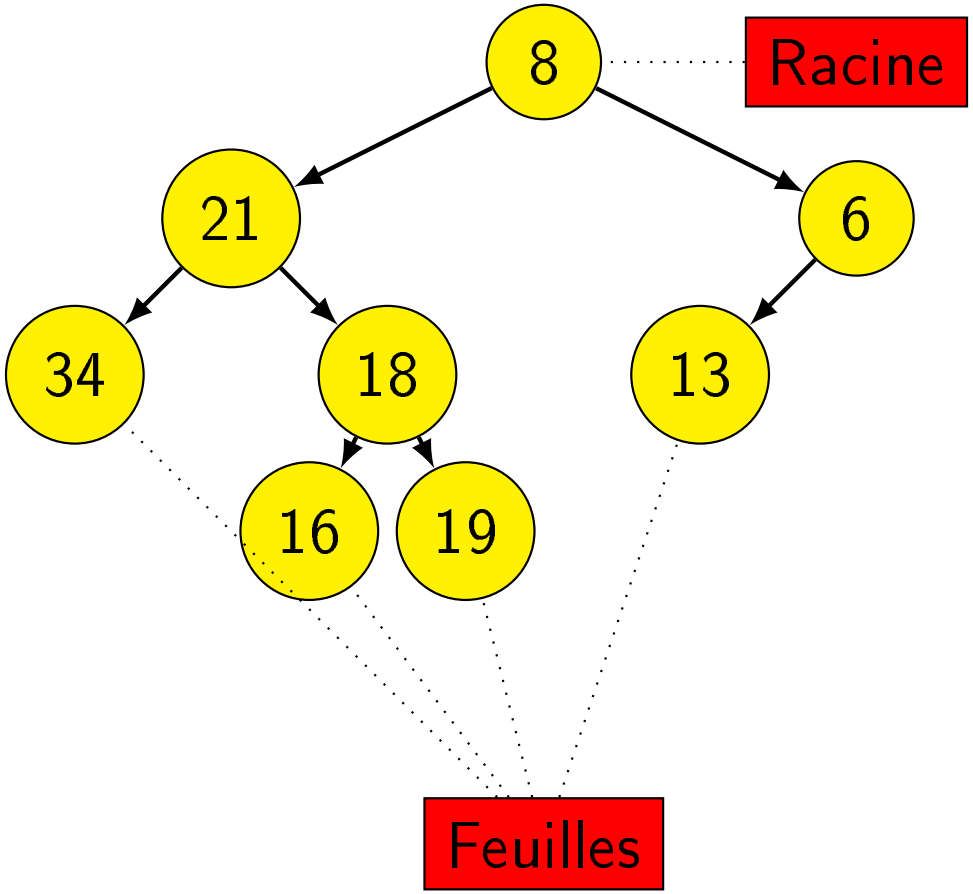
Chaque noeud est **porteur d'une information** (un entier, une chaîne, ...).
L'unique nœud qui n'a pas de parent est la **racine**.

Arbres binaires

Exemple



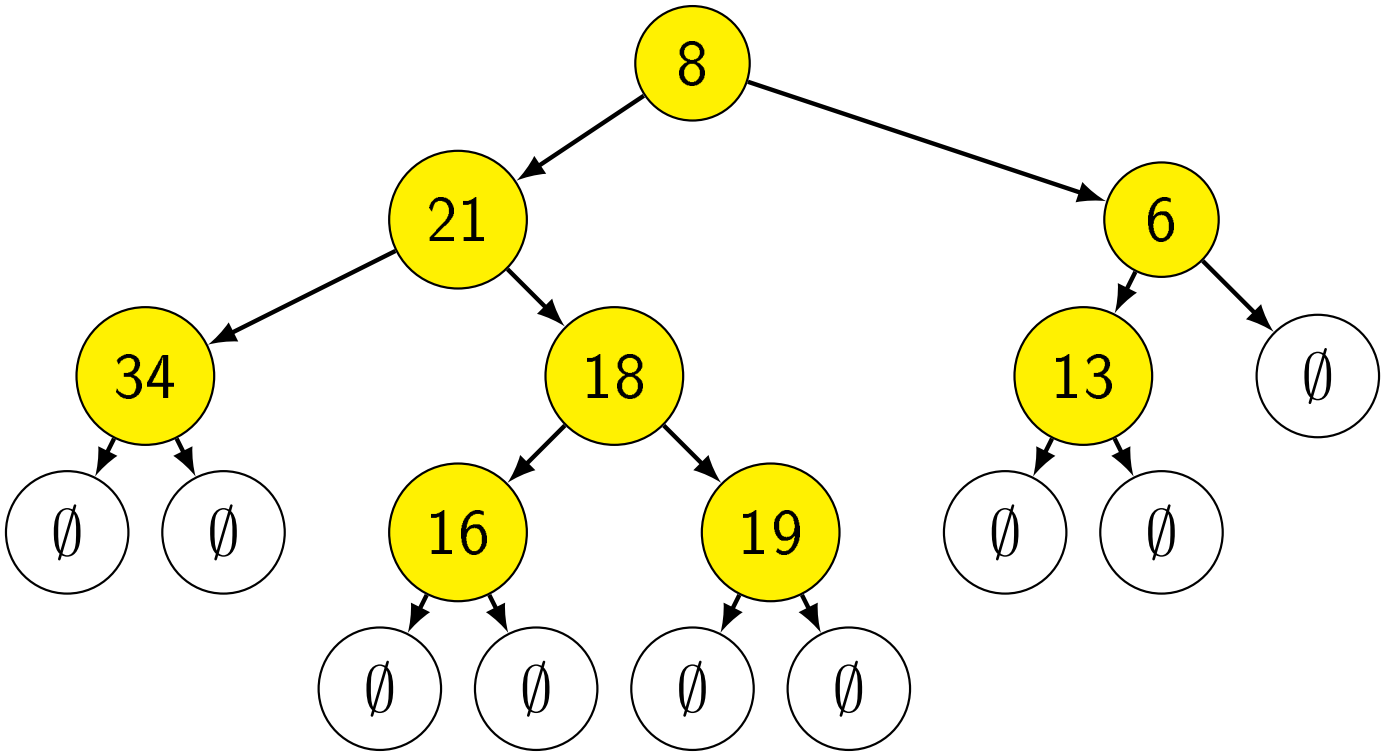
Exemple (suite)



Arbres binaires

Exemple (suite)

Avec tous les noeuds vides :



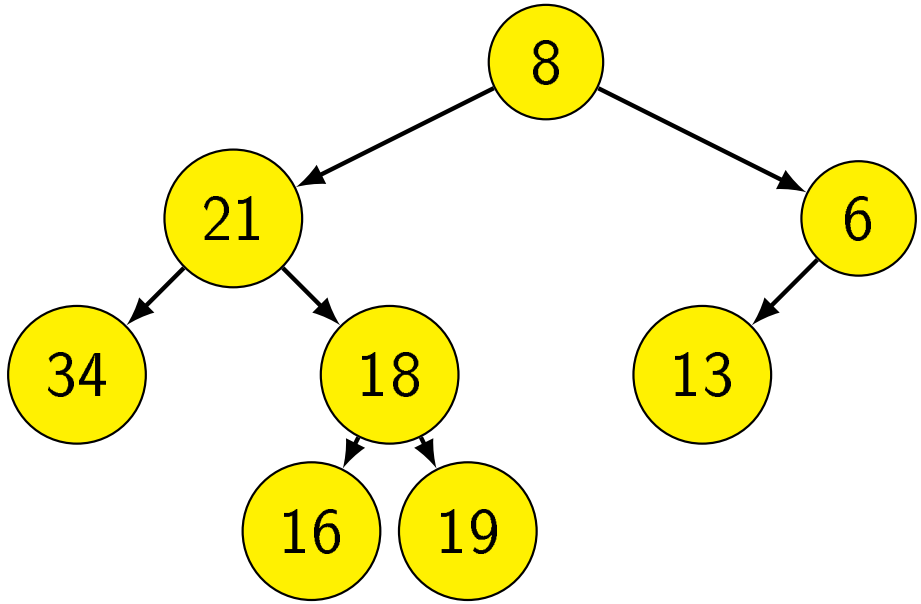
Arbres binaires

Caractéristiques

- On appelle **taille** d'un arbre le nombre de nœuds de cet arbre.
- On appelle **hauteur** d'un arbre le nombre de nœuds du plus long chemin entre la racine et une feuille quelconque de l'arbre.
  La hauteur d'un **arbre vide** est 0.

Exemple (suite)

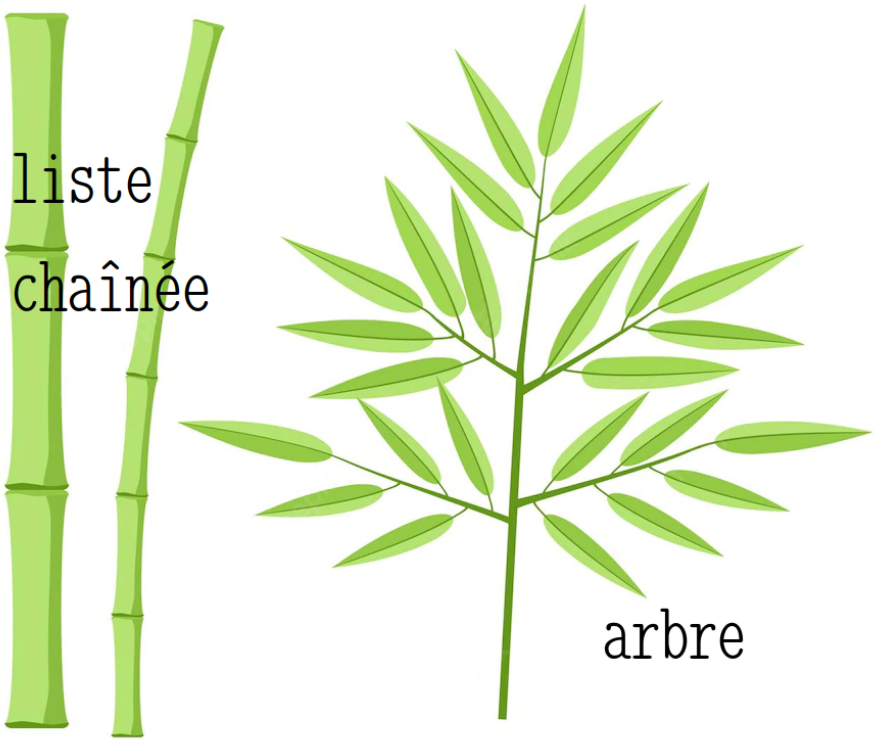
- **taille** : 8
- **hauteur** : 4



Description du type

Principe général

- Toutes les valeurs associées aux noeuds sont de **même type**.
- C'est une structure **récursive** : on distingue l'élément **racine** et le reste (son **sous-arbre gauche** et son **sous-arbre droit**).



Haiku :

Parmi les arbres
Un bambou
Liste chaînée

Description du type

Opérations autorisées

- Opérations de construction
  - Création d'un arbre vide.
  - Création d'un arbre à partir de la valeur de sa racine et de ses deux sous-arbres (gauche et droit)
- Opérations d'accès
  - Accès à la valeur associée à la racine
  - Accès aux sous-arbres gauche et droit
  - Test d'arbre vide

Implémentation

Spécification des fonctions d'interface

Les opérations autorisées seront accessibles via les seules fonctions suivantes :

- Opérations de création
  - `creer_arbre_vide()` : retourne un arbre vide.
  - `creer_arbre(r,g,d)` : retourne l'arbre constitué de la racine `r` et des sous-arbres gauche `g` et droit `d`.
- Opérations d'accès
  - `racine(arbre)` : retourne la valeur racine de l'arbre passé en paramètre.
  - `gauche(arbre)` et `droite(arbre)` : retournent respectivement les sous-arbres gauche et droit de l'arbre passé en paramètre.
  - `est_vide(arbre)` : retourne `True` si l'arbre passé en paramètre est vide, `False` sinon.

Implémentation

Principe

Représentation à l'aide du type `tuple` :

- un arbre vide est représenté par la valeur `None`
- un arbre non vide par un tuple de trois éléments :
  - le premier est la valeur associée à la **racine** (`arbre[0]`)
  - le deuxième est le **sous-arbre gauche** (`arbre[1]`)
  - le troisième est le **sous-arbre droit** (`arbre[2]`)

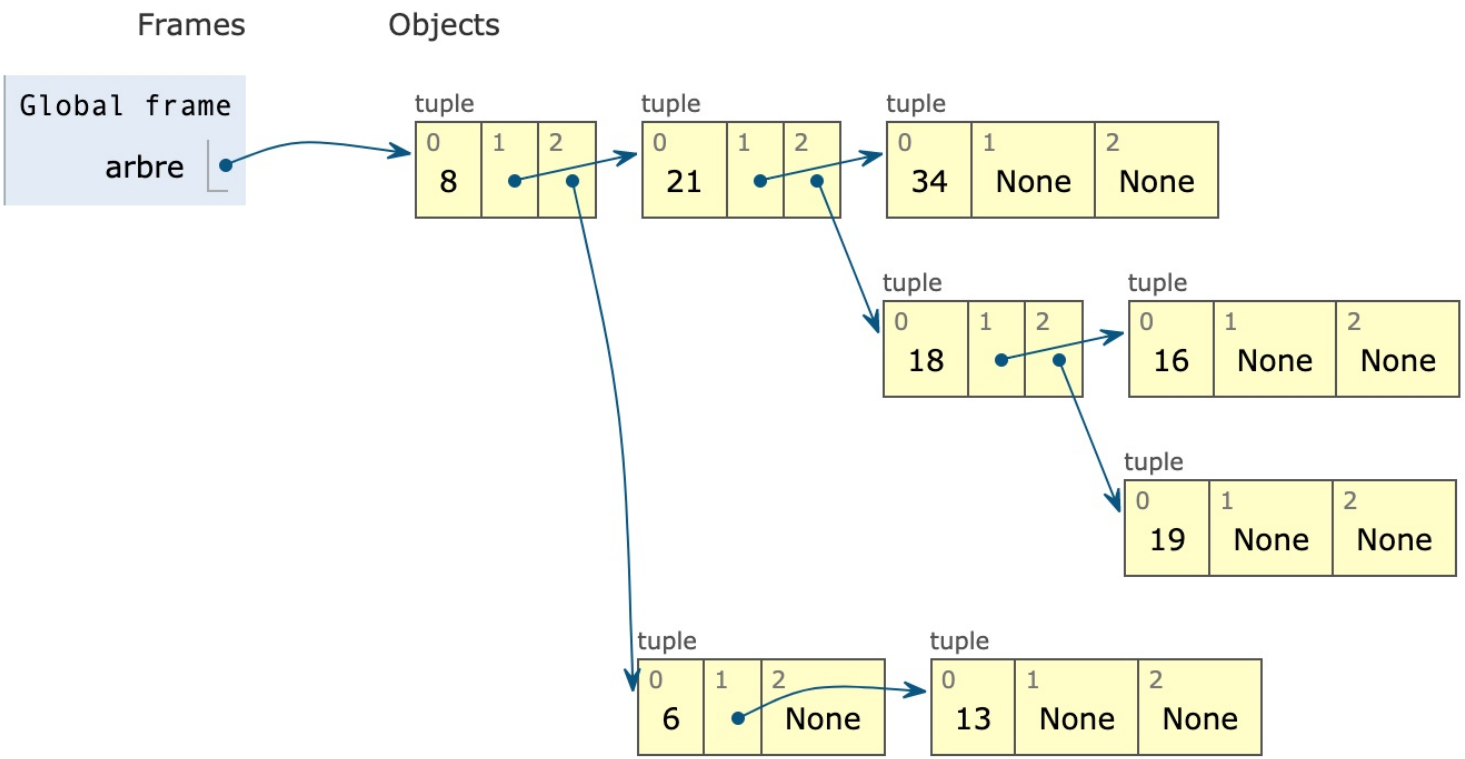
Remarques :

- On ne peut pas modifier un tuple $\Rightarrow$ sécurisation du type.
- On pourra changer de convention plus tard (exemple : représenter l'arbre vide par un tuple vide), cela **ne doit en aucun cas** changer quoique ce soit des autres fonctions qui utiliseront nos arbres !

Exemple (suite)

Représentation de l'arbre donné en exemple :

```
(8, (21, (34, None, None), (18, (16, None, None), (19,
None, None))), (6, (13, None, None), None))
```



Implémentation

Détails de l'implémentation

L'implémentation particulière choisie ici est la suivante :

- Un arbre vide est représentée par `None`.
- Un arbre non vide est un tuple `(r,g,d)`.

```
1  def creer_arbre_vide():
2      return None
3
4  def creer_arbre(r,g,d):
5      return r,g,d
6
7  def racine(arbre):
8      return arbre[0]
9
10 def gauche(arbre):
11     return arbre[1]
12
13 def droite(arbre):
14     return arbre[2]
15
16 def est_vide(arbre):
17     return arbre==None
```

Implémentation

Autres implémentations possibles

- pas de module standard en Python (mais on trouve divers modules à installer comme `tree`)
- tuple ou objet en Python, objet en JAVA, ...
- struct plus pointeurs en C
- tableaux dynamiques

Écriture de fonctions

La récursivité est particulièrement adaptée pour traiter des structures de données récursives comme les arbres

Méthodologie

- **Cas d'arrêt :**
  - En général, arbre vide.
  - Éventuellement feuille : ses deux sous-arbres sont vides.
- **Cas récursif :**
  - En général, appeler la fonction **récursivement sur les deux sous-arbres**.
  - Éventuellement sur un seul sous-arbre : problème dissymétrique ou élément déjà trouvé sur le premier sous-arbre exploré...
  - **On se convainc que la fonction termine** : on a diminué la taille de l'arbre en passant aux sous-arbres.

Écriture de fonctions

Exemple 1 (retournant un entier)

Écrire la fonction récursive `taille` qui accepte en paramètre un `arbre` et renvoie sa **taille**.
Écrire la fonction récursive `hauteur` qui accepte en paramètre un `arbre` et renvoie sa **hauteur**.

Exemple 2 (retournant un arbre)

Écrire la fonction récursive `ajout_a_droite` qui accepte en paramètres un `arbre` ainsi qu'un entier `valeur` et retourne un nouvel arbre résultat de l'ajout d'un noeud à l'arbre donné en paramètre.
Le noeud sera ajouté le plus à droite de l'arbre, et aura pour valeur l'entier passé en paramètre.
Dans l'arbre vu au début, on ajouterait un fils droit au nœud contenant la valeur 6.

Parcours

2 types de parcours

Lors d'un traitement, on doit déterminer l'**ordre** selon lequel les différents nœuds sont examinés, ce qui conditionne l'ordre de traitement des informations.
Deux politiques de parcours sont possibles :

- Parcours en **profondeur** (avec différentes variantes).
- Parcours en **largeur**

Parcours

Parcours en profondeur

Lors d'un **parcours en profondeur préfixe** d'un arbre :

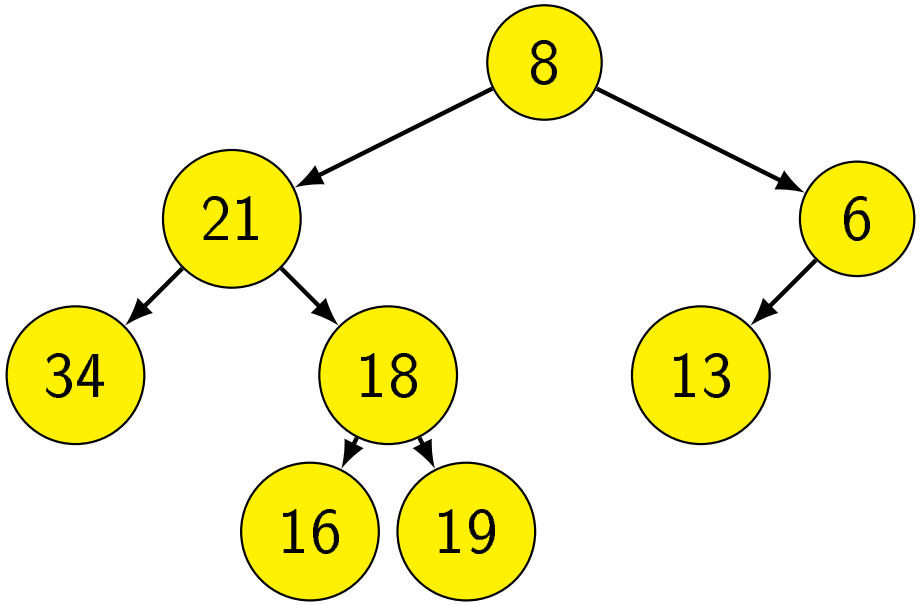
- On traite d'abord la **racine** de l'arbre.
- On effectue un parcours en profondeur préfixe du sous-arbre **gauche**, puis **droit**.

De même on peut définir les **parcours en profondeur infixe et suffixe**.

type de parcours en profondeur	ordre de traitement
préfixe	**racine** gauche droite
infixe	gauche **racine** droite
suffixe	gauche droite **racine**

Parcours

Exemple (suite)



type de parcours en profondeur	ordre de traitement
préfixe	8 21 34 18 16 19 6 13
infixe	34 21 16 18 13 8 13 6
suffixe	34 16 19 18 21 13 6 8

Parcours

Exemple

Écrire les fonctions récursives `parcours_prefixe`, `parcours_infixe` et `parcours_suffixe` qui acceptent en paramètre un `arbre` et **affichent** les valeurs des noeuds de cet arbre selon un parcours en profondeur préfixe, infixe et suffixe.

Parcours

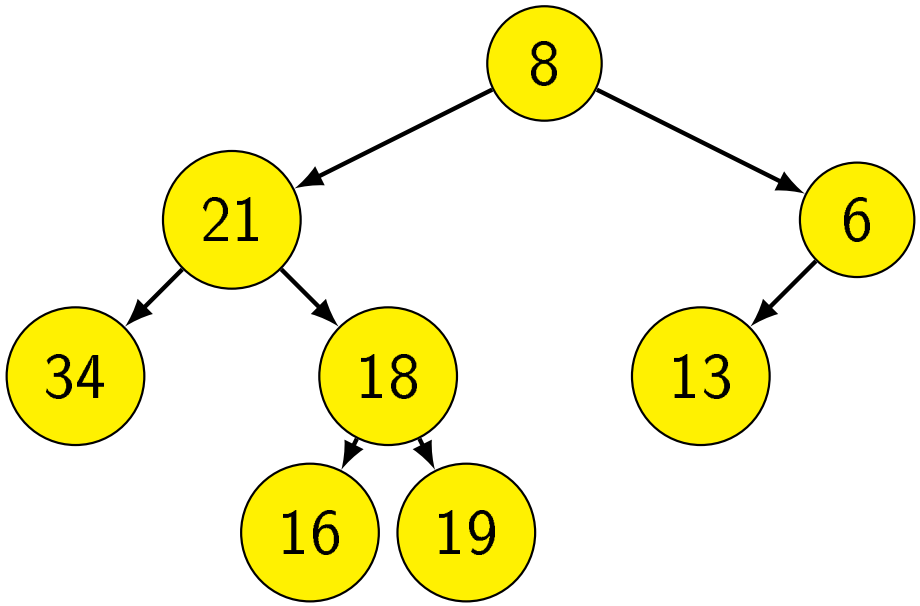
Parcours en largeur

Lors d'un **parcours en largeur** d'un arbre, on traverse l'arbre **niveau par niveau**, en partant du niveau 0 jusqu'au niveau le plus profond.
Pour chaque niveau, le parcours des nœuds s'effectue de la gauche vers la droite.

Remarque : L'écriture d'une fonction de parcours en largeur nécessite une structure de données **file** *(vue utérieurement)*.

Parcours

Exemple (suite)



Ordre de traitement pour le parcours en largeur :
8 21 6 34 18 13 16 19