Informatique MPSI

Arthur Jacquin

23 décembre 2021

Table des matières

1 Présentation

Ce document tente de résumer les cours d'informatique de MPSI (Informatique Pour Tous et option Informatique).

Le langage utilisé en Informatique Pour Tous est Python 3. Créé en 1991 par Guido Van Rossum, Python est un language interprété de haut niveau et multi-paradigmes : object, impératif et fonctionnel notamment. La version 3 utilisée est parue en 2009. Le typage est dynamique.

Le langage utilisé en option Informatique est OCaml, créé en 1996 par l'Institut National de Recherche en Informatique et en Automatique. La version utilisée est OCaml 4. Le typage est fort et statique.

De nombreuses notions sont communes aux deux enseignements, ainsi une bonne partie de ce document traite des deux langages simultanément. Les extraits de code sont généralement présentés en parallèle, Python à gauche, Ocaml à droite, afin de faciliter l'identification des points communs et des différences.

Enfin, une section est destinée au SQL, au programme d'Informatique Pour Tous.

On définit un algorithme comme une **suite d'instruction** acceptant des **données d'entrée** et renvoyant un résultat.

2 Types et variables

2.1 Types principaux

Catégorie	Types Python	Types OCaml
Vide	`None`	`unit`
Booléen		`bool`
Numérique	`bool`, `int`, `float`, `complex`, ...	`int`, `float`, `complex`, ...
Séquence	`list`, `str`, `tuple`	`list`, `string`, `array`
Dictionnaire	`dict`	*c.f.* `Hashtbl`
Fonction	`function`	`fun`
Autres		`char`

2.2 Opérations élémentaires

2.2.1 Python

```
not, (or), (and) # Operations sur booleens
(==), (!=), (<), (<=), (>), (>=) # Comparaisons
(+), (-), (*), (**), (/), (//), (%) # Operations numeriques
abs, round # Approximation
(|), (^), (&), (>>), (<<) # Operations sur les bits
([not] in), (+), .append, .copy, .pop # Operations sequentielles
(+), .split, .strip, .replace # Operations sur chaines de caracteres
bool, int, float, complex, str, tuple, list # Conversions de types

print # Affichage sur sortie standard
max, min # Recherche d'extremum
range # cf section Structures usuelles
chr, ord # Traduction caractere - code ASCII
len # Operations sur conteneurs
iter, next # Operations sur iterateurs
eval, exec # Controle de flot sur chaine de caracteres
open # cf section Gestion de fichiers
```

En Python, il y a *promotion automatique de type* : les objets sont convertis dans le type le plus complexe avant les calculs.

Pour accéder aux éléments d'un tuple ou liste, on utilise l'instruction `tab[i]`. Les indices négatifs fonctionnent également. On peut extraire des listes par *slicing* : `tab[deb:]`, `tab[:fin]`, `tab[deb:fin]` ou encore `tab[deb:fin:step]`. On note que l'indice `fin` n'est jamais atteint.

Un dernier méchanisme intéressant est la création de listes *en compréhension* : `[expression for var in iter [if condition]]`.

2.2.2 OCaml

```ocaml
not, (&&), (||) (* Operations sur booleens *)
(==), (!=), (<), (<=), (>), (>=), max, min (* Comparaisons *)
(+), (-), ( * ), (/), (mod), abs (* Operations sur entiers *)
(+.), (-.), ( *. ), (/.) (* Operations sur flotants *)
exp, log, ( ** ), sqrt (* Exponentielle et logarithme (flotants) *)
[a]cos[h], [a]sin[h], [a]tan[h] (* Trigonometrie (flottants) *)
ceil, floor, truncate (* Approximation (flotants) *)
(^) (* Operations sur chaines de caracteres *)
fst, snd (* Operations sur couples *)
(@) (* Operations sur listes *)
ref, (!), (:=), incr, decr (* References *)
exception, raise, failwith (* Gestion d'erreurs *)

(* Conversion de types *)
float_of_int, int_of_float (* int <> float *)
string_of_bool, bool_of_string (* bool <> string *)
string_of_int, int_of_string (* int <> string *)
string_of_float, float_of_string (* float <> string *)
int_of_char, char_of_int (* char <> int *)
Char.escaped, String.get (* char <> string *)

(* Affichage sur sortie standard *)
print_char, print_string, print_int, print_float, print_newline
```

2.3 Variables

Les variables permettent de stocker des valeurs. Déclarer une variable revient à associer un nom à une adresse mémoire où sera stockée la valeur associée à la variable.

```python
nom_variable = valeur
```

```ocaml
let nom_variable = valeur;;
```

Certains objets, dits *mutables*, sont modifiés *en place*. Il faut donc se méfier des *shallow copy* (par opposition aux *deep copy*) et de l'*aliasing* : une copie trop simple (attribution de la même adresse mémoire) entraîne la modification systématique des deux objets. Les objets *immutables* ne connaissent pas ce problème car toute attribution donne lieu à un changement d'adresse mémoire.

On appelle *scope* les *environnements* dans lesquels on peut accéder à une variable. Une variable *locale* n'est accessible uniquement dans la définition de la fonction (Python) ou dans l'instruction suivante (OCaml : `let var = value in instruction`). Une variable *globale* est accessible partout. La modification d'une variable globale dans la définition d'une fonction Python doit être précédée de l'instruction `global var`.

Les appels aux fonctions forment une *pile d'appels* et une pile d'environnements associés. L'appel d'une variable cherchera la définition la plus récente (dans la pile) de la variable.

2.4 Représentation des réels

2.4.1 Complément à 2

Soit n bits. Soit $x \in [\![-2^{n-1} ; 2^{n-1} [\![$. Si x est positif, on le représente par son écriture en base 2, sinon on le représente par l'écriture binaire de $x + 2^n$.

2.4.2 Ecriture en virgule flottante normalisé

$$\forall x \in \mathbb{R}, \exists!(\epsilon, m, e) \in \{-1; 1\} \times [1; 2[\times \mathbb{Z}, \quad x = \epsilon \cdot 2^e \cdot m$$

Un réel est codé sur 64 bits par :

- 1 bit de signe (ϵ) : `0` pour les réels positifs.
- 11 bits pour l'exposant (e) codé en complément à 2. Des cas particuliers permettent de représenter des valeurs infinies, des zéros signés et des `NaN` (*not a number*).
- 52 bits pour la mantisse correspondant à la représentation binaire de $m - 1$.

Tout nombre n'est donc pas représentable : les tailles allouées à l'exposant et à la mantisse sont nécessairement limitées.

3 Structures usuelles

En Python, les blocs d'instructions sont définis par les tabulations. En OCaml, on peut transformer une suite d'instructions élémentaires en instruction élémentaire par l'utilisation de `begin` et `end` :

```
instruction_bloc_0
    instruction_bloc_1
        instruction_bloc_2
        instruction_bloc_2
    instruction_bloc_1
```

```
begin
    instruction_1;
    ...
    instruction_n;
end
```

3.1 Instructions conditionnelles

```
if condition:
    bloc_instructions
[elif condition:
    bloc_instructions]*
[else:
    bloc_instructions]
```

```
if condition then
    instruction
[else
    instruction]
```

3.2 Boucles

3.2.1 Boucles inconditionnelles

```
for variable in iterable:
    bloc_instructions
```

```
for variable = deb to fin do
    instruction
done
```

En Python, les itérables usuels sont les objects séquentiels ou l'instruction `range`, dont le fonctionnement est à rapprocher du *slicing* :

— `range(fin)` créera les valeurs `0`, `1`, ..., `fin - 1`.
— `range(deb, fin)` créera les valeurs `deb`, `deb` + `1`, ..., `fin - 1`.
— `range(deb, fin, pas)` créera les valeurs (strictement inférieures à `fin`) `deb`, `deb` + `pas`, ...

En OCaml, la `variable` prendra successivement les valeurs `deb`, `deb` + `1`, ..., `fin`.

3.2.2 Boucles conditionnelles

```
while condition:
    bloc_instructions
```

```
while condition do
    instruction
done
```

3.2.3 Contrôle de flot

Deux commandes permettent de contrôler plus finement les boucles :

— La commande `break` force une sortie de boucle.
— La commande `continue` force le passage à l'itération suivante.

3.3 Filtrage (OCaml)

Le filtrage (ou *pattern matching*) est une structure très utilisée en OCaml. Elle permet de détecter le constructeur de l'objet en entrée. Si le *pattern matching* est également disponible avec Python depuis la version 3.10., on s'intéresse ici à son utilisation en OCaml. La structure générale de la commande est la suivante.

```
match variable with
[| constructeur [when condition] -> instruction]*
```

On parle de **filtrage avec gardes** lorsque des conditions sont précisées. Le filtrage est dit **exhaustif** si tous les cas de figure sont traités, sinon il y a **non exhaustivité du filtrage**. Attention, le compilateur ne connaît pas les hypothèses faites sur les données en entrée. Ainsi, le filtrage suivant (à gauche) d'un entier à valeurs dans $\{0; 1\}$ ne sera pas considéré exhaustif. Pour rendre un filtrage exhaustif, ou plus généralement filtrer positivement n'importe quel constructeur, il est possible d'utiliser l'*underscore* (à droite) :

```
match val_bool_entiere with
| 0 -> false
| 1 -> true
```

```
match val_bool_entiere with
| 0 -> false
| _ -> true
```

Comme ce sont les constructeurs qui sont comparés, il n'est pas possible de comparer une variable à une autre. L'algorithme de gauche renverra toujours `true` car il est possible d'attribuer la valeur de `n` (un entier) à `a`, dont la valeur sera écrasée. Pour contourner le problème, on peut utiliser un filtrage avec gardes (à droite) :

```
let a = 42;;
match n with
| a -> true
| _ -> false;;
```

```
let a = 42;;
match n with
| _ when a = n -> true
| _ -> false;;
```

Enfin, il est possible de filtrer plusieurs variables en même temps :

```
let rec fusion l1 l2 =
    match l1, l2 with
    | [], _ -> l2
    | _, [] -> l1
    | t1::q1, t2::q2 when t1 < t2 -> t1::(fusion q1 l2)
    | t1::q1, t2::q2 -> t2::(fusion l1 q2);;
```

4 Fonctions

Les fonctions permettent de sauvegarder une suite d'instructions pour y faire appel par la suite.

4.1 Structure

Les p représentent ici des paramètres.

```python
def nom_fonc([p][, p]*):
    bloc_instructions

nom_fonc = lambda [p][, p]*: instruction
```

```ocaml
let [rec] nom_fonc [p][ p]* =
    instruction

let nom_fonc = fun p[ p]* -> instruction
```

4.2 Paramètres

En Python, les paramètres sont de la forme `nom_param[: type][ = val_defaut]`. Il est donc possible d'expliciter le type du paramètre, ainsi qu'une valeur par défaut, valeur que prendra `nom_param` si aucune valeur n'est spécifiée. Un paramètre présentant une valeur par défaut est dit *optionnel*, sinon le paramètre est dit *obligatoire*. Les différents paramètres sont séparés par des virgules. Les paramètres obligatoires doivent figurer avant les paramètres optionnels.

En OCaml, les paramètres sont de la forme `nom_param` ou `(nom_param : type)`. Tous les paramètres sont donc obligatoires. Les différents paramètres sont séparés par des espaces.

4.3 Valeur renvoyée

En Python, l'instruction `return value` termine la fonction, qui renvoie alors `value`. Il peut y avoir plusieurs points de retour dans la fonction. En OCaml, la valeur renvoyée est celle de la dernière instruction.

4.4 Fonctions récursives

Une fonction est dite récursive lorsqu'elle s'appelle elle-même. En OCaml, il faut déclarer qu'une fonction est récursive avec le mot clé `rec` (*c.f.* Structures). En Python, aucune précision n'est demandée.

4.5 Paradigmes procédural et fonctionnel

On dit qu'une fonction est une **procédure** et agit par *effets de bord* lorsqu'elle modifie l'environnement (les variables, la sortie standard, ...) et ne renvoie rien. De telles fonctions sont caractéristiques du paradigme procédural.

Par opposition aux procédures, les fonctions ne modifiant pas l'environnement et renvoyant un résultat sont dites **fonctions pures** et sont caractéristiques du paradigme fonctionnel.

4.6 Exemples

```python
def somme_pwr(a: list, b = 2):
    res = 0
    for elem in a:
        res += elem**b
    return res
```

```ocaml
let rec pgcd (a: int) b =
    if b = 0 then a
    else pgcd b (a mod b);;
```

5 Analyse

5.1 Terminaison

Il faut s'assurer que les programmes écrits se terminent. Deux schémas peuvent provoquer la non-terminaison : les boucles conditionnelles (si la condition reste toujours vraie) et les fonctions récursives. Dans les deux cas, prouver la terminaison revient à exhiber un **variant**, quantité entière positive strictement décroissante. En effet, il n'existe pas de suites strictement décroissante définie sur $\mathbb{N}$ à valeurs entières positives, d'où la terminaison.

5.2 Validité

Il faut également s'assurer que le résultat renvoyé par l'algorithme est bien celui souhaité. Prouver la validité revient à exhiber un **invariant**, prédicat pour lequel il faut prouver :

1. L'**initialisation** : les préconditions rendent le prédicat vrai.
2. La **conservation** : la valeur logique de ce prédicat n'est pas modifée par l'éxécution de l'algorithme. Pour les boucles inconditionnelles, il peut être intéressant d'indexer le prédicat.
3. La **terminaison** : une fois l'algorithme exécuté, l'invariant fournit une propriété utile à la preuve de la validité.

Par exemple, pour les boucles conditionnelles et inconditionnelles :

```
# H = invariant
preconditions
# H vrai
while cond:
    # Si H vrai
    bloc_instructions
    # alors H vrai
# H vrai et cond fausse
```

```
(* H = invariant *)
preconditions
(* H vrai *)
while cond do
    (* Si H vrai *)
    instruction
    (* alors H vrai *)
done;
(* H vrai et cond fausse *)
```

```
# H(k) = invariant
preconditions
# H(deb) vrai
for k in range(deb, fin):
    # Si H(k) vrai
    bloc_instructions
    # alors H(k+1) vrai
# H(fin) vrai
```

```
(* H(k) = invariant *)
preconditions
(* H(deb) vrai *)
for k=deb to fin do
    (* Si H(k) vrai *)
    instruction
    (* alors H(k+1) vrai *)
done;
(* H(fin+1) vrai *)
```

5.3 Complexité

Prouver la terminaison et la validité d'un algorithme permet de s'assurer qu'il renvoie le résultat souhaité. En revanche, cela ne dit rien de son efficacité. L'étude de la complexité tente de pallier ce problème. La complexité reflète le temps pris (complexité temporelle) et la place occupée en mémoire (complexité spatiale) par l'éxécution d'un alogrithme. L'étude de la complexité spatiale n'est pas au programme : on se contentera d'éviter la multiplication des listes, en essayant au maximum de faire des opérations en place.

L'objectif serait de prédire le temps d'éxécution d'un algorithme pour une taille n de données à traiter définie. Malheuresement, les multiples facteurs en jeu (architecture de l'ordinateur, gestion de la mémoire, puissance du processeur, ...) rendent le calcul très difficile.

En dépit d'un calcul précis, on s'intéresse donc principalement à l'évolution de ce temps d'éxécution (tous autres facteurs égaux) en fonction de n. On procède de la façon suivante :

1. Choix d'un paramètre n (*e.g.* taille d'une liste, ...).
2. Choix de l'unité (correspondant à l'étape la plus significative : nombre d'opérations élémentaires, nombre d'appels récursifs, ...).
3. Choix de la nature de la complexité calculée : pire des cas, cas moyen, meilleur des cas, ... Bien souvent, on s'intéresse au pire des cas.
4. Calcul. Il est fréquent de trouver une formule de récurrence. On peut alors utiliser différentes stratégies :
   — S'intéresser à des cas particulers (*e.g.* les 2^k) puis conclure sur les autres cas en considérant la complexité croissante en fonction de n.
   — S'il existe $(a,b) \in \mathbb{R}^2$ tel que $a \neq 1$ et pour tout $n \in \mathbb{N}$, on ait $c_{n+1} = a \cdot c_n + b$. On note alors $l = \frac{b}{1-a}$, qui donne $(c_{n+1} - l) = a \cdot (c_n - l)$, d'où $c_n = a^n \cdot (c_0 - l) + l$.
   — Utiliser le théorème maître.
5. Conclusion selon le tableau suivant.

Complexité	Appelation
$O(1)$	Constante
$O(\log(n))$	Logarithmique
$O(n)$	Linéaire
$O(n\log(n))$	Pseudo-linéaire
$O(n^2)$	Quadratique
$O(n^p),\ p > 2$	Polynomiale
$O(2^n)$	Exponentielle
$O(n!)$	Factorielle

En pratique, la complexité pseudo-linéaire est la complexité maximale pour laquelle on peut espérer des résultats satisfaisants.

5.3.1 Théorème maître

Supposons qu'il existe $(a,b) \in \mathbb{N}^2 \backslash \{(0,0)\}$, $k \geq 2$ et $f \in \mathbb{R}^{\mathbb{N}}$ une fonction croissante tels que :

$$\forall n \in [\![2, +\infty[\![, \quad c_n = a\, c_{\lceil \frac{n}{k} \rceil} + b\, c_{\lfloor \frac{n}{k} \rfloor} + f(n)$$

On note $\alpha = \ln_k(a+b)$ et β tel que $f(n) = O(n^\beta)$. Alors :
— $\beta > \alpha \quad \Rightarrow \quad c_n = 0(n^\beta)$
— $\beta = \alpha \quad \Rightarrow \quad c_n = 0(n^\alpha \ln(n))$
— $\beta < \alpha \quad \Rightarrow \quad c_n = 0(n^\alpha)$

6 Tris

6.1 Importance des tris

Il est possible d'améliorer de façon très significative la complexité d'un algorithme si les données d'entrée, de type séquentiel, sont triées. Les tris sont donc très couramment utilisés. Comme il en existe de nombreux types, cette section vise uniquement à présenter les tris usuels.

Les algorithmes naïfs sont quadratiques (en prenant pour paramètre le nombre d'éléments à trier) tandis que les meilleurs tris généraux sont pseudo-linéaires. Il ne faut cependant pas émettre de trop grandes généralités : quelques hypothèses sur les données d'entrée permettent ainsi de réaliser un tri linéaire.

6.2 Tris naïfs

6.2.1 Tri par insertion

```
let inserer l x =
    let rec aux l x mem insere =
        if insere = true then
            match mem with
            | [] -> l;
            | t::q -> aux (t::l) x q true
        else
            match l with
            | [] -> aux (x::l) x mem true;
            | t::q when x < t -> aux (x::l) x mem true;
            | t::q -> aux q x (t::mem) false
    in aux l x [] false;;

let rec tri_insertion l =
    match l with
    | [] -> l;
    | t::q -> inserer (tri_insertion q) t;;
```

6.3 Tris efficaces

6.3.1 Tri fusion

```
let rec partage l =
    match l with
    | [] -> [], [];
    | t::[] -> l, [];
    | a::b::q -> let l1, l2 = partage q in a::l1, b::l2;;

let rec fusion l1 l2 =
    match l1, l2 with
    | [], _ -> l2;
    | _, [] -> l1;
    | t1::q1, t2::q2 when t1 < t2 -> t1::(fusion q1 l2);
    | t1::q1, t2::q2 -> t2::(fusion l1 q2);;

let rec tri_fusion l =
    let l1, l2 = partage l in
        fusion (tri_fusion l1) (tri_fusion l2);;
```

6.3.2 Tri rapide

```
let remplacer tab a b =
    let tmp = tab.(a) in
        begin
            tab.(a) <- tab.(b);
            tab.(b) <- tmp;
        end;;

let tri_rapide tab =
    let rec aux tab i j =
        if j > i then
            let pivot = tab.(j) and f = ref i in
            (* H(t) := les cases i..f-1 sont inferieures a pivot
             *                    f..t-1 sont superieures a pivot*)
                begin
                    (* reorganisation des elements *)
                    for t = i to j - 1 do
                        if tab.(t) < pivot then
                            begin
                                remplacer tab !f t;
                                incr f;
                            end
                    done;
                    (* positionnement du pivot *)
                    remplacer tab !f j;
                    (* appels recursifs *)
                    aux tab i (!f-1);
                    aux tab (!f+1) j;
                end
    in aux tab 0 (Array.length tab - 1);;
```

6.4 Un tri linéaire

Supposons qu'il existe vmax tel que les données d'entrée soient un tableau à valeurs dans $[\![0; v_{max}[\![$. On peut aisément montrer que l'algorithme suivant termine, est valide et linéaire.

```
def tri_lineaire(tab):
    # Calcul du vmax
    vmax = 0
    for x in tab:
        if x >= vmax:
            vmax = x + 1
    # Denombrement
    comp = [0]*vmax
    for x in tab:
        comp[x] += 1
    # Generation du resultat
    res = []
    for i in range(vmax):
        for _ in range(comp[i]):
            res.append(i)
    return res
```

7 Arbres (OCaml)

Les arbres peuvent être implémentés de multiples manières, avec diverses notations. On s'intéresse donc plus aux notions classiques qu'à la retranscription des fonctions usuelles, qui varie selon l'implémentation. On retiendra principalement que les fonctions récursives utilisant le *pattern-matching* sont omniprésentes.

7.1 Implémentation classique

```
type tree =
    | F of feuille;
    | N of tree * racine * tree;;
```

Toute **racine** est donc **père** d'un **fils droit** et d'un **fils gauche**. On définit naturellement les notions d'ancêtres et de descendants.

Les **noeuds** sont l'ensemble des **feuilles** (noeuds externes) et des **racines** des sous-arbres de l'arbre total (noeuds internes). La **profondeur** d'un noeud est le nombre de générations le séparant de la racine de l'arbre total (profondeur nulle pour la feuille des arbres feuilles). Les noeuds de même profondeur forment le **niveau** de profondeur associée. La **hauteur** d'un arbre correspond à la profondeur maximale de ses noeuds.

```
let rec ni t =
    (* nb de noeuds internes *)
    match t with
    | F f -> 0;
    | N (g, r, d) ->
        1 + (ni g) + (ni d);;
```

```
let rec h t =
    (* hauteur de l'arbre *)
    match t with
    | F f -> 0;
    | N (g, r, d) ->
        1 + max (h g) (h d);;
```

On peut prouver par induction sur les arbres que pour tout arbre T :

$$nb\_feuilles(T) = nb\_racines(T) + 1$$

$$h(T) + 1 \leq nb\_feuilles(T) \leq 2^{h(T)}$$

Des terminaisons correpondent aux cas d'égalité. Ainsi, un arbre est dit *peigne* si et seulement si $h(T) + 1 = nb\_feuilles(T)$, et *complet* si et seulement si $nb\_feuilles(T) = 2^{h(T)}$.

7.2 Parcours

Parcourir un arbre est chose commune. Il y a plusieurs façons de le faire :

— les parcours dits **en largeur**, qui traite les noeuds par niveaux croissants.

— les parcours dits **en profondeur**, qui traite les noeuds par branches. Dans cette optique, on traite l'arbre `N (g, r, d)` en traitant récursivement les fils gauche `g` et droite `d`. On distingue les parcours préfixe, infixe, et postfixe selon que l'on traite la racine `r` avant, entre ou après les fils.

```ocaml
let parcours_largeur t =
    let rec aux todo =
        match todo with
        | [] -> ();
        | (F f)::q ->
          begin
            traiter_feuille f;
            aux q;
          end;
        | (N (g, r, d))::q ->
          begin
            traiter_noeud r;
            aux todo@[g; d];
          end
    in aux [t];;
```

```ocaml
let rec parcours_prefixe t =
    match t with
    | F f -> traiter_feuille f;
    | N (g, r, d) ->
        begin
            traiter_noeud r;
            parcours_prefixe g;
            parcours_prefixe d;
        end;;
```

Il y a unicité des parcours préfixes et postfixes, ce qui les rends très intéressants. Le parcours postfixe est ainsi utilisé par la notation polonaise inversée (RPN), qui manipule une pile au lieu de représenter les expressions sous forme d'arbre. Montrons l'unicité par l'exhibition de la bijection réciproque :

```ocaml
let rec liste_prefixe t =
    match t with
    | F f -> [("F", f)];
    | N (g, r, d) ->
        ("N", r)::((liste_prefixe g)@(liste_prefixe d));;

let inv l =
    let rec aux l res =
        match l with
        | [] -> res;
        | t::q -> aux q (t::res)
    in aux l [];;

let arbre_prefixe l =
    let rec aux l res =
        match l, res with
        | [], t::[] -> t;
        | ("F", f)::q, _ -> aux q ((F f)::res);
        | ("N", r)::q1, g::d::q2 -> aux q1 ((N (g, r, d))::q2);
        | _, _ -> failwith "Arbre invalide."
    in aux (inv l) [];;
```

Un arbre est également parcourue lors de la recherche d'un élément ou d'un extremum :

```ocaml
let rec rech_elem x t =
    match t with
    | F f -> f = x;
    | N (g, r, d) ->
          (r = x)
        || (rech_elem x g)
        || (rech_elem x d);;
```

```ocaml
let rec rech_max t =
    match t with
    | F f -> f;
    | N (g, r, d) ->
        let mg = rech_max g
        and md = rech_max d in
            max r (max mg md);;
```

7.3 Arbres binaires homogènes, ABR

```
type 'a arbre =
    | Vide;
    | N of ('a arbre) * 'a * ('a arbre);;
```

Les arbres binaires de recherche (ABR) sont des arbres binaires homogènes à valeurs dans un ensemble entièrement ordonné et dont le parcours infixe renvoie une liste croissante. Tout l'intêret des ABR apparaît lorsque la hauteur h est minimale, car h est alors de l'ordre de $\log_2(n)$, d'où des complexités logarithmiques pour les opérations élémentaires :

— Recherche d'un extremum.
— Recherche d'un élément.
— Insertion d'un élément.
— Suppression d'un élément.

```
let rec rech_max_ABR t =
    match t with
    | N (_, r, Vide) -> r;
    | N (_, _, g) -> rech_max_ABR g;;
    | _ -> failwith "Arbre vide.";;

let rec rech_elem_ABR x t =
    match t with
    | Vide -> false;
    | N (g, r, d) when x < r -> rech_elem_ABR x g;
    | N (g, r, d) when x > r -> rech_elem_ABR x d;
    | _ -> true;;

let rec inserer_ABR x t =
    match t with
    | Vide -> N (Vide, x, Vide);
    | N (g, r, d) when x < r -> N (inserer_ABR x g, r, d);
    | N (g, r, d) when x > r -> N (g, r, inserer_ABR x d);
    | _ -> t;;

let rec fusion_ABR t tp =
    match tp with
    | Vide -> t;
    | N (g, r, d) ->
        inserer_ABR r (fusion_ABR (fusion_ABR t g) d);;

let rec suppr_ABR x t =
    match t with
    | Vide -> failwith "Valeur non presente.";
    | N (g, r, d) when x < r -> N (suppr_ABR x g, r, d);
    | N (g, r, d) when x > r -> N (g, r, suppr_ABR x d);
    | _ -> fusion_ABR g d;;
```

Le maintien d'une hauteur minimale peut se faire par **rotations** droites et gauches, qui ne brisent pas le caractère ABR.

L'insertion successive des éléments d'une liste (pseudo-linéaire) dans un ABR initialement vide, puis son parcours infixe (linéaire) est donc une façon optimale (pseudo-linéaire) de trier une liste !

```
let rec liste_infixe t =
    match t with
    | Vide -> [];
    | N (g, r, d) -> (liste_infixe g)@[r]@(liste_infixe d);;

let tri_ABR l =
    let rec abr_of_list todo =
        match todo with
        | [] -> Vide;
        | t::q -> inserer_ABR t (abr_of_list q)
    in liste_infixe (abr_of_list l);;
```

7.4 Preuves par induction

On peur prouver la validité d'un prédicat P par induction sur les arbres. Pour cela, il faut montrer dans le cas de l'implémentation classique (respectivement celles des arbres binaires homogènes) :

— P est vrai pour tout arbre feuille (respectivement vide).

— P est vrai pour tout arbre de forme `N (ta, r, tb)` où `r` est une racine quelquonque et `ta` et `tb` des arbres pour lesquels P est vrai.

8 Gestion de fichiers (Python)

8.1 Création d'une connexion

8.1.0.1 `open(file: str, mode: str)` création d'une connexion avec le fichier `file`. Il existe différents types de connexion, permettant d'accéder à différentes méthodes.

8.2 Types de connexion

`mode`	Signification	Description
`'r'`	*read*	Lecture seule
`'w'`	*write*	Ecriture (écrasement)
`'a'`	*append*	Ecriture (ajout en fin de fichier)

8.3 Méthodes

Méthode	Modes	Typage	Description
`.close()`	Tous	`unit > unit`	Suppression de la connexion
`.readlines()`	Lecture	`unit > str list`	Récupération des lignes
`.readline()`	Lecture	`unit > str`	Récupération d'une ligne (itérateur)
`.write(text: str)`	Ecriture	`str > unit`	Ecriture de `text`

8.4 Remarques

— Une connexion ouverte doit être fermée !
— Les sauts de ligne sont à traiter manuellement. La méthode `.strip()` peut-être utile.
— Pour le traitement de fichiers `.csv`, la méthode `.split(separator: str)` peut-être utile.

9 Modules Python

9.1 Utilisation

Les modules suivants doivent être importés avant utilisation. Les différentes méthodes d'imports font varier la façon d'accéder aux éléments du module. La première méthode peut provoquer des *collisions*. On choisira donc préférentiellement les méthodes deux et trois.

```
from nom_module import obj[, obj]* # Acces aux objets : obj
import nom_module # Acces aux objets: nom_module.obj
import nom_module as surnom # Acces aux objets: surnom.obj
```

9.2 `maths`

```
ceil, floor, trunc # Approximation
factorial, comb # Combinatoire
gcd, lcm # Arithmetique
exp, log # Exponentielle et logarithme naturel
sqrt # Identique a **.5
[a]cos[h], [a]sin[h], [a]tan[h] # Trigonometrie
pi, e, tau # Constantes
```

9.3 `random`

```
randrange([start, ]stop[, step]) # Fonctionne comme range
randint(start, stop) # Entier entre start et stop inclus
choice(sequence) # Element de sequence
random() # Flotant entre 0 et 1 exclus
```

9.4 `time`

```
monotonic() # Temps depuis une origine fixe quelconque
time() # Temps depuis epoch
sleep(n) # Suspend l'execution n secondes
strftime(fmt: str) # Convertit localtime() en chaine de caractere
                   # Utiliser les commandes ci-dessous
                   # ex: 'Il est %H heure %M'
```

Commande	Résultat
`%c`	Représentation usuelle (totale)
`%x`	Représentation usuelle (date)
`%X`	Représentation usuelle (heure)
`%Y`	Année
`%m`	Mois ($[\![01; 12]\!]$)
`%d`	Jour du mois ($[\![01; 31]\!]$)
`%H`	Heure ($[\![00; 23]\!]$)
`%M`	Minute ($[\![00; 59]\!]$)
`%s`	Seconde ($[\![00; 59]\!]$)
`%A`	Jour de la semaine (nom)

9.5 `numpy`

Introduit le type `ndarray`, qui diffère du type `list` :
— la taille des `ndarray` et le type de ses éléments sont fixés à la création des objets
— les opérations sont considérablement plus rapides et moins gourmandes en mémoire

— la multidimensionalité est mieux gérée
— les opérations vectorielles (éléments par éléments) sont possibles : `np.zeros(3)` + 1 correspond à `np.array([1, 1, 1])`

```
array([[1, 2], [4, 5]]) # Creation d'un ndarray
                        # Creation en comprehension possible
zeros(taille) # Creation avec valeurs initialisees a zero
identity(n) # Creation d'une matrice identite
linspace(a, b, n) # Creation de n points uniformement repartis
any(cond), all(cond) # Verification d'un predicat sur une matrice
sum(arr), prod(arr), transpose(arr), dot(arr1, arr2) # Operations
arr[1, 1::] # Acces aux elements (slicing possible)
arr.ndim # Dimension
arr.shape # Taille selon les dimensions
arr.copy() # Deepcopy
```

9.6 matplotlib.pyplot

```
plot(x, y[, fmt: str]) # Trace une figure
                       # Utiliser les commandes ci-dessous
                       # ex: '.r'
xlabel(name), ylabel(name) # Nommer les axes
show() # Affiche la figure
savefig(path) # Enregistre la figure
imread(path) # Lecture de l'image sous forme de ndarray
imshow(arr) # Affiche l'image deduite d'un ndarray
imsave(path, arr) # Enregistre l'image
```

Marqueur	Style de ligne	Lettre	Couleur
.	Points	k	Noir
,	Pixels	b	Bleu
x	Croix	g	Vert
-	Ligne continue	r	Rouge
:	Ligne pointillée	y	Jaune

9.7 turtle

```
forward(n) # Avance de n pixels
right(d), left(d) # Tourne de d degres
goto(x, y), setx(x), sety(y) # Defini la position
penup(), pendown(), isdown() # Controle du stylo
showturtle(), hideturtle() # Controle de la tortue
```

9.8 sqlite3

Fonctionne de façon similaire à la gestion de fichiers.

```
import sqlite3 as sql
con = sql.connect(db_path) # Ouverture d'une connexion
cur = con.cursor() # Creation d'un curseur
cur.execute(query) # Requetage
for row in cur: # Lecture du resultat de la requete
    bloc_instructions
con.commit() # Commit changes
con.close() # Close the connexion
```

10 Modules OCaml

10.1 Utilisation

La majorité des modules suivants font partie de la librairie standard. Il suffit d'utiliser l'instruction `nom_module.objet` pour accéder aux objets de ces modules. Le module `Graphics` n'en fait partie : il faut donc importer le module (commandes à la section `Graphics`).

10.2 String

```
get s i (* Acces au caractere d'indice i, identique a s.[i] *)
length s (* Longueur de s *)
concat sep s_list (* Concatenation des chaines de s_list *)
sub s debut longueur (* Acces a une sous-chaine *)
```

10.3 Array

```
length t (* Taille du tableau *)
get t i (* Acces a l'element d'indice i, identique a t.(i) *)
set t i x (* Ecriture de l'element i, identique a t.(i) <- x *)
make n def (* Creation d'un tableau de taille n ou tous
              les elements sont initialises a def *)
make_matrix x y def (* Creation d'une matrice de format (x, y) *)
sub t debut longueur (* Creation d'un sous-tableau *)
copy t (* Deepcopy *)
map f t (* Mapping *)
fold_left f accu t (* Pliage a gauche *)
sort f t (* Tri croissant selon f
            ex: Array.sort (fun a b -> b - a) tab *)
```

10.4 List

```
type 'a list =
    | []
    | (::) of 'a * 'a list;;
length l (* Taille de la liste *)
hd l (* Acces au premier element *)
tl l (* Acces au reste de la liste *)
nth l i (* Acces a l'element d'indice i *)
rev l (* Inversion de liste (non tail-recursive) *)
append l1 l2 (* Concatenation des listes, identique a l1@l2 *)
map f l (* Mapping *)
fold_left f accu l (* Pliage a gauche *)
sort f l (* Tri croissant selon f *)
```

10.5 Queue (FIFO) et Stack (LIFO)

```
create (* Creation d'une file/pile *)
push x q (* Ajoute un element *)
is_empty q (* Verifie si la file/pile est vide *)
pop q (* Supprime et renvoie l'element le plus
             ancien (module Queue)
          ou recent (module Stack) *)
length q (* Taille de la file/pile *)
clear q (* Vide la file/pile *)
copy q (* Copie de la file/pile *)
fold f accu q (* Pliage a gauche *)
```

10.6 Hashtbl

```
create n (* Creation d'une table de taille n *)
mem table cle (* Test d'existence d'une valeur *)
add table cle valeur (* Ajout d'une valeur *)
find table cle (* Lecture d'une valeur *)
remove table cle (* Suppression d'une valeur *)
```

10.7 Random

```
int n (* Entier aleatoire entre 0 et n - 1 *)
float f (* Flotant aleatoire entre 0 et f *)
```

10.8 Graphics

```
load "graphics.cma";; open Graphics;; (* Import du module *)
open_graph "" (* Affiche l'ecran de trace *)
moveto x y (* Defini la position du pointeur *)
lineto x y (* Trace un trait de la position actuelle
              a la position precisee (deplace le pointeur) *)
close_graph () (* Supprime l'ecran de trace *)
```

11 Méthodes classiques

11.1 Recherche d'extremum ou d'une valeur dans un tableau, une liste

```python
def rech_max(l):
    if len(l) == 0:
        return None
    else:
        M = l[0]
        for i in range(len(l)):
            if l[i] > M:
                M = l[i]
        return M
```

```ocaml
let rec rech_elem l x =
    match l with
    | [] -> false;
    | t::q when t = x -> true;
    | t::q -> rech_elem q x;;
```

Dans le cas d'un tableau trié, on peut chercher de façon dichotomique :

```ocaml
let rech_trie t x =
    let rec aux i j =
        if i > j then false
        else let m = (i + j)/2 in
            if t.(m) = x then true
            else if t.(m) > x then aux (m + 1) j
            else aux i (m - 1)
    in aux 0 (Array.length t - 1);;
```

11.2 Exponentiation rapide

```python
def expo(x, n):
    if n == 0:
        return 1
    elif n%2 == 1:
        return x * expo(x*x, n//2)
    else:
        return expo(x*x, n//2)
```

```ocaml
let rec expo x n =
    if n = 0 then 1
    else if n mod 2 = 1 then
        x * (expo (x*x) (n/2))
    else
        expo (x*x) (n/2);;
```

11.3 Décomposition en base b

```python
def decomp(n, b):
    if n == 0:
        return []
    else:
        return decomp(n//b, b) + [n%b]
```

```ocaml
let decomp n b =
    let rec aux n =
        if n = 0 then []
        else [n mod b]::(aux (n/b))
    in List.rev (aux n);;
```

11.4 Pivot de Gauss

```
def cherche_pivot(A, j):
    maxi, indice = abs(A[j][j]), j
    for i in range(j, len(A)):
        if abs(A[i][j]) > maxi:
            maxi, indice = abs(A[i][j]), i
    return indice

def echange(M, i1, i2):
    M[i1], M[i2] = M[i2], M[i1]

def transvection(M, i1, i2, mu):
    for j in range(len(M[0])):
        M[i1][j] = M[i1][j] + mu * M[i2][j]

def dilatation(M, i, mu):
    for j in range(len(M[0])):
        M[i][j] = mu * M[i][j]

def gauss(A, Y):
    # Echelonnage
    for j in range(len(A)):
        pivot = cherche_pivot(A, j)
        echange(A, j, pivot)
        echange(Y, j, pivot)
        for k in range(j + 1, len(A)):
            coef = - A[k][j] / A[j][j]
            transvection(A, k, j, coef)
            transvection(Y, k, j, coef)

    # Diagonale de 1
    for i in range(len(A)):
        coef = 1 / A[i][i]
        dilatation(A, i, coef)
        dilatation(Y, i, coef)

    # Remontee
    for j in range(len(A) - 1, 0, -1):
        for k in range(j):
            coef = - A[k][j]
            transvection(A, k, j, coef)
            transvection(Y, k, j, coef)
```

11.5 Recherche de zéros

Soit une fonction f définie et continue sur $I = [a; b]$ et telle que $f(a)\,f(b) \leq 0$. Par théorème des valeurs intermédiaires, il existe $\alpha \in I$ pour lequel f s'annule. Approchons ce α :

— Si f est strictement monotone : méthode dichotomique.

— Si f est dérivable et sa dérivée est connue : méthode de Newton.

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$$

— Si f est dérivable : méthode de la sécante.

$$x_{n+1} = x_n - \frac{x_n - x_{n-1}}{f(x_n) - f(x_{n-1})} f(x_n)$$

— Si on a $(a_0, b_0) \in I^2$ tels que $f(a_0)\, f(b_0) \leq 0$ et $|f(b_0)| \leq |f(a_0)|$: méthode de Dekker :

1. On pose $b_{-1} = a_0$.
2. Tant que $|b_n - a_n| > eps$:
   (a) On pose lorsque c'est possible $\alpha_n = b_n - \frac{b_n - b_{n-1}}{f(b_n) - f(b_{n-1})} f(b_n)$ et $\beta_n = \frac{a_n + b_n}{2}$. Si α_n n'est pas défini ou n'est pas compris entre b_n et β_n, on pose $b_{n+1} = \beta_n$, sinon on pose $b_{n+1} = \alpha_n$.
   (b) Si $f(a_n)\, f(b_{n+1}) \leq 0$, on pose $a_{n+1} = a_n$. Sinon, on pose $a_{n+1} = b_n$. On vérifie qu'on a ainsi $f(a_{n+1})\, f(b_{n+1}) \leq 0$.
   (c) Si $|f(a_{n+1})| < |f(b_{n+1})|$, on échange les valeurs de a_{n+1} et b_{n+1}.

Seule la dichotomie donne une réponse systématique, mais sa vitesse de convergence n'est pas très bonne :

```
def dicho(f, a, b, eps):
    c = (a + b)/2
    if b - a < 2*eps:
        return c
    elif f(c) * f(a) <= 0:
        return dicho(f, a, c, eps)
    else:
        return dicho(f, c, b, eps)

def newton(f, f_, x0, eps): # Arret : x_{n+1} - x_n < eps
    dx = f(x0)/f_(x0)
    if dx < eps:
        return x0
    else:
        return newton(f, f_, x0 - dx, eps)
```

11.6 Equations différentielles

```
def euler(F, a, b, n, y0):
    h = (b - a)/n
    t, y = [a], [y0]
    tc, yc = a, y0
    for k in range(n):
        yc, tc = yc + h * F(yc, tc), tc + h
        y.append(yc)
        t.append(tc)
    plt.plot(t, y)
    plt.show()
    return t, y
```

11.7 Intégration

```
def integrale(f, a, b, n): # cf sommes de Riemann
    s, h = 0, (b - a)/n
    for i in range(n):
        s += f(a + i*h)
    return s*h
```

11.8 *Backtracking*

Objectif : trouver une configuration valide pour un ensemble d'éléments à valeurs dans E un ensemble fini et ordonné :

1. Lister dans `todo` les indices des éléments à trouver.
2. Initialiser un indice `k` à `0`.
3. Tant que k est inférieur strict à `len(todo)` :
   (a) S'il n'y a pas de valeurs à l'indice `todo[k]`, y mettre la première valeur de E. Sinon, mettre la suivante. S'il n'y pas de valeurs suivante, c'est qu'aucune des valeurs de E convient, *i.e.* le début de configuration n'est pas valide, donc décrémenter k et recommencer cette étape.
   (b) Si la configuration a une chance d'être valide, incrémenter k.
4. La solution est valide.

11.9 Méthode du rejet

Objectif : simuler un loi uniforme sur un sous-ensemble A de E :

1. Fonction fE simulant une loi uniforme sur E
2. Faire une fonction qui utilise fE tant que l'object généré n'est pas dans A

11.10 Paradigmes

— Diviser pour régner
— Algorithmes gloutons
— Programmation dynamique (bas en haut)
— Mémoïsation (haut en bas)

12 Requêtes SQL

Les languages de requêtes structurées (*Structured Query Language*) permettent de gérer efficacement des bases de données. Seule la requête `SELECT` est exigible en classes préparatoires.

12.1 SELECT

```
SELECT [DISTINCT] [MIN|AVG|MAX|COUNT|SUM(]oid|*|column_name[)] [AS temp_name] [, ...]
FROM table_name [AS temp_name]
[JOIN table_name [AS temp_name] ON condition]*
[WHERE condition]
[GROUP BY column_name [HAVING condition]]
[ORDER BY column_name [ASC|DESC] [, ...]]
[LIMIT number_of_row];
```

12.2 Conditions

Les conditions peuvent avoir plusieurs formats :

```
column_name ==|!=|<[=]|>[=] value
column_name [NOT] IN (sql_statement)|(value [, ...])
column_name [NOT] BETWEEN min_value AND max_value
column_name IS [NOT] NULL
column_name LIKE pattern
```

Les commandes `AND`, `OR` et `NOT` permettent de former des conditions plus complexes. Enfin, il est possible d'utiliser les *fonctions d'aggrégation* (telles que `MIN` ou `AVG`) dans les conditions liées à `HAVING`. Elles agissent alors sur les regroupements créés par la commande `GROUP BY`.

Les *pattern* peuvent tirer profit des *wildcards* suivantes :
- `_` représente un unique caractère
- `%` représente une chaine de caractères quelquonque (éventuellement vide)

13 *Object Oriented Programming* (Python)

13.1 Généralités

Une classe permet de définir de nouveaux types d'objets, ainsi que leurs attributs et méthodes associées. L'utilisation de *docstrings* est vivement recommandée.

```
class NomClasse: # Creation d'une classe

    def nom_methode(self[, args]): # Definition d'une methode
        pass

a = NomClasse([args]) # Creation d'un objet
a.attr # Acces a un attribut
a.attr = value # Definition de l'attribut
del a.attr # Suppression de l'attribut
```

13.2 Méthodes spéciales

Python réserve certains noms de méthodes à des **méthodes spéciales**, appelées dans des contextes définis. La liste suivante présente les méthodes spéciales les plus fréquentes. Notons qu'il n'est pas nécessaire de définir toutes ces méthodes : une erreur `NotImplemented` sera levée lors d'un appel abusif. Le premier argument de ces méthodes est toujours `self` ; on note `nbarg` le nombre d'arguments supplémentaires recquis par la méthode.

13.2.1 Général

Méthode	`nbarg`	Appel
`__init__`	n	Appelé lors de la création d'un object.
`__repr__`	0	`x` (dans le toplevel)

13.2.2 Comparaison

Méthode	`nbarg`	Appel
`__eq__`	1	`x == y`
`__ne__`	1	`x != y`
`__lt__`	1	`x < y`
`__le__`	1	`x <= y`
`__gt__`	1	`x > y`
`__ge__`	1	`x >= y`

13.2.3 Conversion de type

Méthode	`nbarg`	Appel
`__bool__`	0	`bool(s)`
`__int__`	0	`int(s)`
`__str__`	0	`str(s)`
`__float__`	0	`float(s)`
`__complex__`	0	`complex(s)`

13.2.4 Container

Méthode	`nbarg`	Appel
`__len__`	0	`len(s)`
`__getitem__`	1	`s[key]`
`__delitem__`	1	`del s[key]`
`__setitem__`	2	`s[key] = item`
`__contains__`	1	`x in s`

13.2.5 Itérateurs

Méthode	`nbarg`	Appel
`__iter__`	0	`for i in s:`
`__next__`	0	`next(s)`

Pour les itérateurs à support finis, on utilise l'instruction `raise StopIteration` dans la définition de `__next__`.

13.2.6 Maths

Méthode	`nbarg`	Appel
`__add__`	1	`x + y`
`__sub__`	1	`x - y`
`__mul__`	1	`x * y`
`__matmul__`	1	`x @ y`
`__truediv__`	1	`x / y`
`__floordiv__`	1	`x // y`
`__mod__`	1	`x % y`
`__pow__`	1	`x ** y`
`__lshift__`	1	`x << y`
`__rshift__`	1	`x >> y`
`__and__`	1	`x & y`
`__xor__`	1	`x ^ y`
`__or__`	1	`x | y`

En préfixant la méthode d'un `i`, on définit les opérateurs d'assignment associés. `__isub__` est ainsi appelé par `a -= 3`.

13.2.7 Opérateurs unaires

Méthode	`nbarg`	Appel
`__neg__`	0	`-x`
`__abs__`	0	`abs(x)`
`__round__`	1	`round(x, n)`
`__trunc__`	0	`trunc(x)`
`__floor__`	0	`floor(x)`
`__ceil__`	0	`ceil(x)`

13.3 Un exemple : les arbres

```python
from itertools import chain

class Tree:
    ''' Implementation de la structure d'arbre '''

    def __init__(self, *arg):
        ''' Initialisation
            Appel: Tree(arg)
        '''
        if len(arg) == 0:
            self.vide = True
            self.fg = None
            self.racine = None
            self.fd = None
        elif len(arg) == 1:
            self.vide = False
            self.fg = Tree()
            self.racine = arg[0]
            self.fd = Tree()
        elif len(arg) == 3:
            self.vide = False
            self.fg = arg[0]
            self.racine = arg[1]
            self.fd = arg[2]
        else:
            raise NotImplemented

    def is_empty(self):
        ''' Predicat de nullite
            Appel: is_empty(t)
        '''
        return self.vide

    def __iter__(self):
        ''' Parcours prefixe
            Appel: for i in t:
        '''
        if self.vide:
            return iter([])
        else:
            return chain(iter([self.racine]), iter(self.fg), iter(self.fd))

    def __len__(self):
        ''' Hauteur de l'arbre
            Appel: len(s)
        '''
        if self.vide == True:
            return 0
        else:
            return 1 + max(len(self.fd), len(self.fg))
```

```python

    def __contains__(self, item):
        ''' Recherche d'un element
            Appel: x in t
        '''
        for i in self:
            if i == item:
                return True
        return False

    def __bool__(self):
        ''' Conversion vers booleen
            Appel: bool(t), if t
        '''
        return self.vide == False

    def __eq__(self, other):
        ''' Comparaison d'arbres
            Appel: t1 == t2
        '''
        if self.vide == other.vide:
            if self.vide == True:
                return True
            else:
                return (self.racine == other.racine) and (self.fg == other.fg) and (self.fd == other.fd)
        else:
            return False

    def __repr__(self):
        ''' Representation de l'objet
            Appel: t (in toplevel)
        '''
        if self:
            if self.fg:
                if self.fd:
                    return f"Tree({self.fg}, {self.racine}, {self.fd})"
                else:
                    return f"Tree({self.fg}, {self.racine}, Vide)"
            else:
                if self.fd:
                    return f"Tree(Vide, {self.racine}, {self.fd})"
                else:
                    return f"Tree({self.racine})"
        else:
            return ""

    def __str__(self):
        ''' Conversion vers chaine
            Appel: str(t)
        '''
        return repr(self)
```

14 Astuces, élégance du code, vrac

14.1 `PEP 8`

Python est un langage assez souple. Pour simplifier le partage du code et la démocratisation du langage, la `PEP 8` (*Python Enhancement Proposal*) fixe des conventions lors de l'écriture de code Python. Un parcours rapide de son contenu (accessible en ligne ici) est vivement recommandé pour améliorer la clarté et la compréhensibilité du code.

14.2 Gestion des erreurs

```python
class CustomError(Exception):
    pass
raise error
```

```ocaml
exception CustomError;;

raise error;;
```

En OCaml, il est également possible de lancer des messages d'erreurs avec `failwith`, de typage `string > 'a` (s'adapte au reste de l'algorithme).

14.3 *Unpacking*, *packing*, utilisation de l'*underscore*

La formation de tuples (*packing*) et l'extraction des valeurs d'un tuple (*unpacking*) sont des mécanismes fréquemment utilisés, parfois de façon non explicite, par exemple lors de l'échange de valeurs de deux variables :

```python
a, b = b, a
```

```ocaml
let a, b = b, a;;
```

Parfois, toutes les valeurs du tuple ne sont pas intéressantes : on peut alors utiliser l'*underscore*, dont le fonctionnement est à rapprocher celui d'une variable jetable :

```python
a, _, b = 1, 2, 3

for _ in range(42):
    print("Hello World!")
```

```ocaml
let a, _, b = 1, 2, 3;;

for _ = 1 to 42 do
    print_string "Hello World!"
done;;
```

14.4 Commandes Python utiles

- Les *f-string*, à utiliser à l'occasion des `exec`.

14.5 Commandes OCaml utiles

- `incr` et `decr` , de typage `int ref > unit`.
- `min` et `max` de typage `'a > 'a > 'a`.

14.6 Conseils et notes

- Commenter au préalable le code, puis le code lui-même.
- Commencer par réfléchir au typage de la fonction demandée.
- Penser à réutiliser les fonctions précédentes.
- Vérifier régulièrement la syntaxe.
- Eviter les conditions complexes.