DIU Enseigner l'informatique au lycée

Notes de cours

Jean-Christophe Filliâtre

Jean-Christophe.Filliatre@lri.fr

Arbres binaires

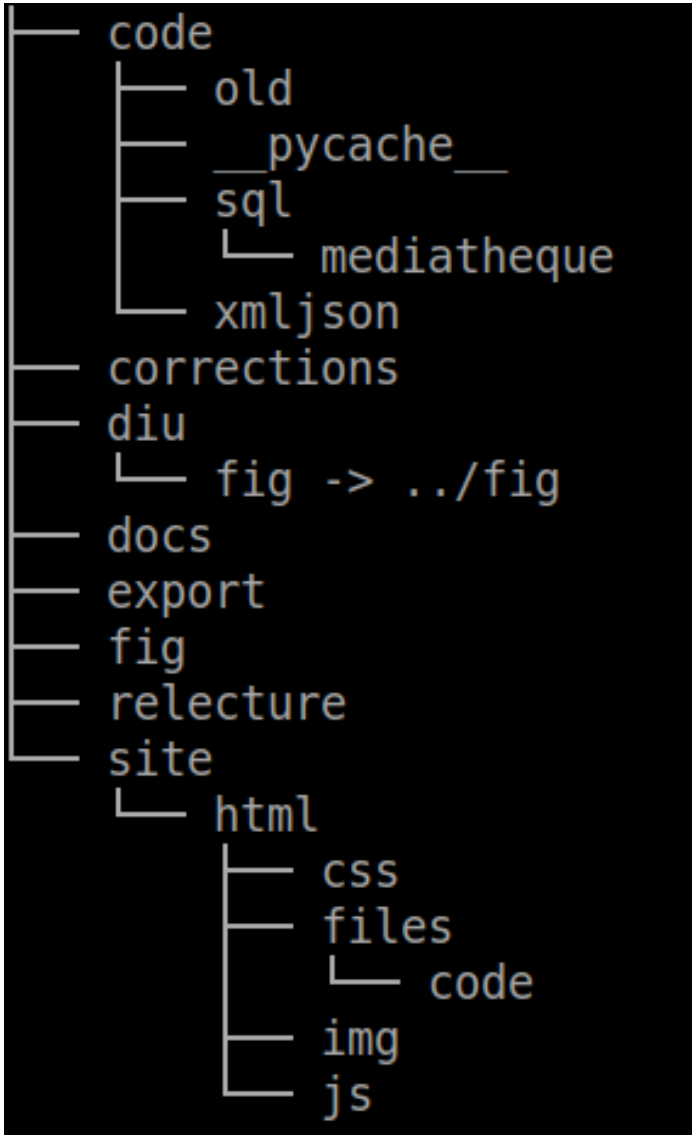
Arbres binaires de recherche

Arborescences

arborescence de fichiers

Le système de fichiers de votre ordinateur a la structure d'un arbre.

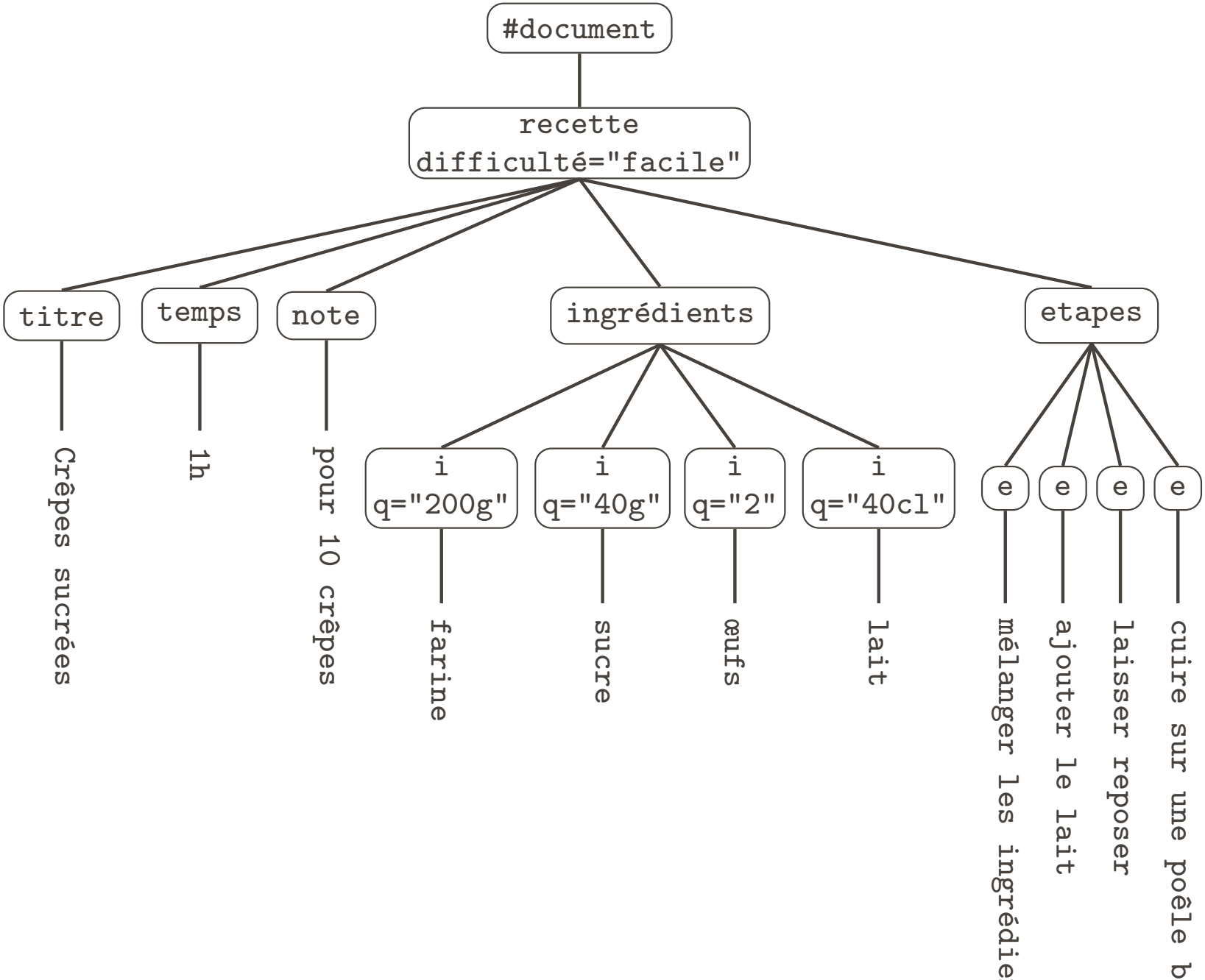
Un répertoire est un nœud, un fichier est une feuille.



document XML

```
<recette difficulté="facile">
 <titre>Crêpes sucrées</titre>
 <temps>1h</temps>
 <note>pour 10 crêpes</note>
 <ingredients>
    <i q="200g">farine</i>
    <i q="40g">sucre</i>
    <i q="2">œufs</i>
    <i q="40cl">lait</i>
 </ingredients>
 <etapes>
   <e>mélanger les ingrédients solides</e>
   <e>ajouter le lait</e>
   <e>laisser reposer</e>
   <e>cuire sur une poêle beurrée</e>
 </etapes>
</recette>
```

document XML

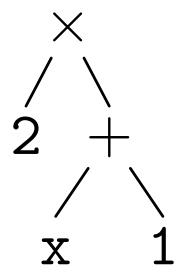


arbre de syntaxe

Quand vous écrivez

```
2 * (x + 1)
```

dans un programme Python, ceci est représenté en interne par un arbre de la forme



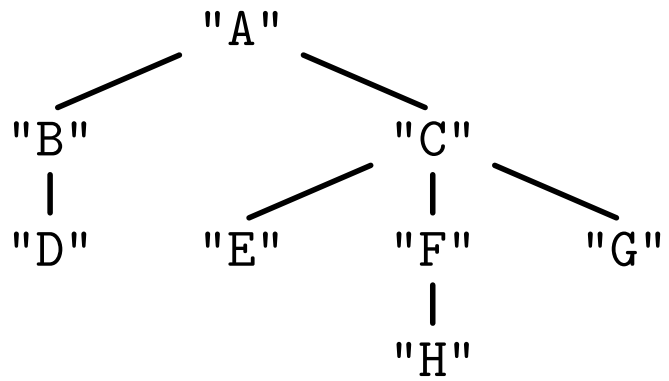
définition

Une arborescence est un ensemble non vide de nœuds qui sont organisés de la façon suivante :

- un nœud particulier constitue la racine de l'arborescence ;
- les autres nœuds sont partagés en n sous-ensembles distincts, qui forment autant d'arborescences ;
- le nœud racine est relié aux n racines de ces arborescences, qu'on appelle ses fils.

exemple

Une arborescence contenant 8 nœuds, étiquetés par des chaînes de caractères :



- la racine est le nœud étiqueté par `"A"` ;
- le nœud `"A"` possède deux fils (`"B"` et `"C"`) ;
- les nœuds `"B"` et `"F"` possèdent un seul fils ;
- le nœud `"C"` possède trois fils ;
- les nœuds `"D"`, `"E"`, `"H"` et `"G"` ne possède aucun fils (et sont appelés des feuilles).

Arbres binaires

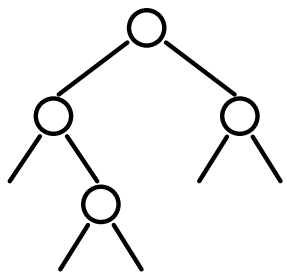
définition : arbre binaire

Un arbre binaire est un ensemble fini de nœuds correspondant à l'un des deux cas suivants.

- Soit l'arbre est vide, c'est-à-dire qu'il ne contient aucun nœud.
- Soit l'arbre n'est pas vide, et ses nœuds sont structurés de la façon suivante :
  - un nœud est appelé la racine de l'arbre ;
  - les nœuds restant sont séparés en deux sous-ensembles, qui forment récursivement deux sous-arbres appelés respectivement sous-arbre gauche et sous-arbre droit.

exemple

Voici un exemple d'arbre binaire contenant quatre nœuds.

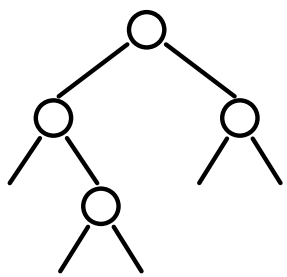


La racine de l'arbre est représentée en haut — en informatique, les arbres poussent vers le bas.

définition : taille d'un arbre

La taille d'un arbre binaire est définie comme son nombre de nœuds.

Exemple : l'arbre



a la taille 4.

feuille

Lorsqu'un nœud a deux sous-arbres vides, c'est-à-dire qu'il est de la forme

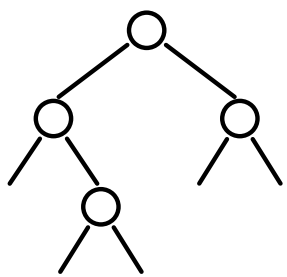
on parle de feuille.

Il est important de ne pas confondre une feuille, qui contient exactement un nœud, et un arbre vide, qui n'en contient pas.

définition : hauteur

On définit la hauteur d'un arbre comme le plus grand nombre de nœuds rencontrés en descendant de la racine jusqu'à une feuille (ou, de façon équivalente, jusqu'à un arbre vide). Tous les nœuds le long de cette descente sont comptés, y compris la racine et la feuille.

Exemple : l'arbre



a une hauteur 3.

définition équivalente de la hauteur

On peut définir la hauteur récursivement sur la structure de l'arbre de la manière suivante :

- l'arbre vide a pour hauteur 0 ;
- un arbre non vide a pour hauteur le maximum des hauteurs de ses deux sous-arbres, auquel on ajoute 1.

(On la programmera ainsi, plus loin.)

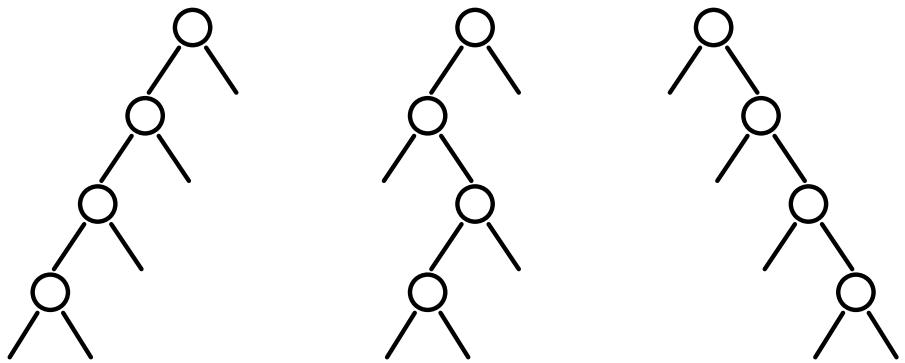
propriété

Si N désigne la taille d'un arbre binaire et si h désigne sa hauteur, alors on a :

$$h \leq N \leq 2^h - 1$$

borne de gauche atteinte

L'égalité $h = N$ est possible lorsque l'arbre est complètement linéaire :

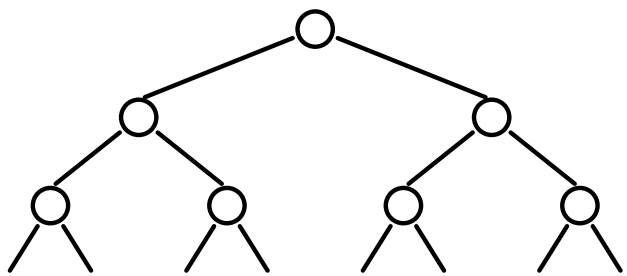


Les arbres comme celui de gauche ou celui de droite, où le sous-arbre non vide est systématiquement du même côté, sont appelés des peignes.

borne de droite atteinte

L'égalité $N = 2^h - 1$ est possible pour un arbre binaire parfait où toutes les feuilles sont exactement à la même profondeur.

Exemple : l'arbre



est un arbre binaire parfait de hauteur 3 et sa taille est $2^3 - 1 = 7$.

remarque

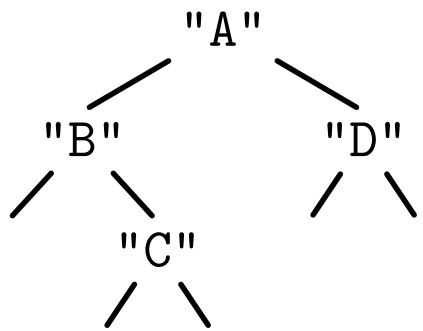
La hauteur est une notion importante. Elle joue notamment un grand rôle lorsque la complexité des algorithmes dépend directement de la hauteur des arbres.

Nous le verrons plus tard avec les arbres binaires de recherche.

de l'information dans les arbres

L'intérêt des arbres est d'y stocker de l'information. Pour cela, on attache une ou plusieurs valeurs à chaque nœud.

Exemple : un arbre avec une chaîne de caractères stockée dans chaque nœud



Les arbres vides, en revanche, ne contiennent pas d'information.

représentation en Python

De nombreuses solutions existent.

représentation en Python

De nombreuses solutions existent.

La plus simple : chaque nœud est un objet d'une classe

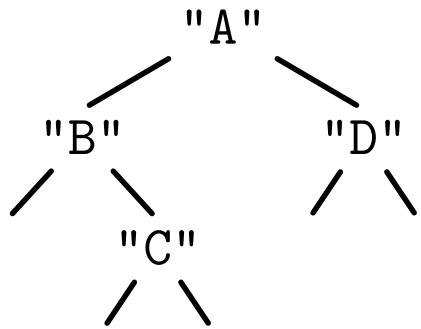
```
class Noeud:
    """un noeud d'un arbre binaire"""
    def __init__(self, g, v, d):
        self.gauche = g
        self.valeur = v
        self.droit  = d
```

Par ailleurs, l'arbre vide est représenté par la valeur `None`.

(C'est l'analogue de la classe `Cellule` pour les listes chaînées.)

exemple

L'arbre

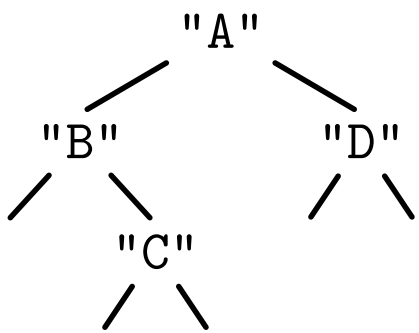


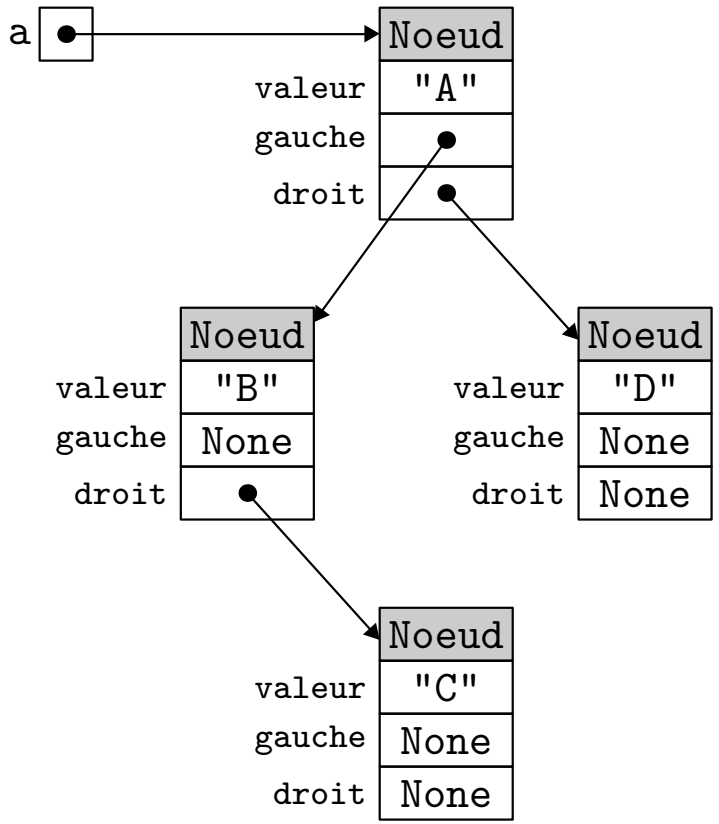
est construit et stocké dans la variable a avec cette instruction :

```
a = Noeud(Noeud(None, "B", Noeud(None, "C", None)),
          "A",
          Noeud(None, "D", None))
```

exemple

```
a = Noeud(Noeud(None, "B", Noeud(None, "C", None)),
          "A",
          Noeud(None, "D", None))
```





Algorithmique des arbres binaires

remarque

La définition d'arbre binaire étant récursive, il est naturel d'écrire des opérations sur les arbres binaires sous la forme de fonctions récursives.

exemple : calcul de la taille

Écrivons une fonction

```
def taille(a):
    """le nombre de noeuds de l'arbre a"""
```

exemple : calcul de la taille

```
def taille(a):
    """le nombre de noeuds de l'arbre a"""
    if a is None:
        return 0
    else:
        return 1 + taille(a.gauche) + taille(a.droit)
```

autre exemple : la hauteur

```
def hauteur(a):
    """la hauteur de l'arbre a"""
    if a is None:
        return 0
    else:
        return 1 + max(hauteur(a.gauche),
                       hauteur(a.droit))
```

complexité

Ces deux fonctions `taille` et `hauteur` ont une complexité directement proportionnelle au nombre de nœuds de l'arbre.

En effet, ces deux fonctions font un nombre d'opérations borné sur chaque nœud (addition, maximum) et parcourent une fois et une seule chaque nœud.

parcours infixe

Écrivons maintenant une fonction qui affiche les valeurs contenues dans tous les nœuds de l'arbre, par exemple une par ligne.

L'ordre dans lequel le parcours des nœuds est effectué devient important.

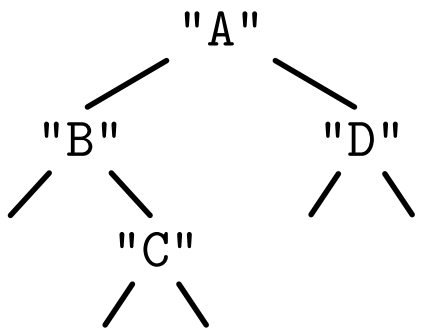
On peut par exemple parcourir le sous-arbre gauche, puis afficher la valeur de la racine, puis enfin parcourir le sous-arbre droit. On appelle cela un parcours infixe.

parcours infixe

```
def parcours_infixe(a):
    """affiche les éléments de l'arbre a
       dans un parcours infixe"""
    if a is None:
        return
    parcours_infixe(a.gauche)
    print(a.valeur)
    parcours_infixe(a.droit)
```

exemple

Sur l'arbre



on obtient

```
B
C
A
D
```

autres parcours

On pourrait également afficher la valeur avant de parcourir les sous-arbres (on parle alors de parcours préfixe) ou après (on parle alors de parcours postfixe).

Exercices

Arbres binaires de recherche

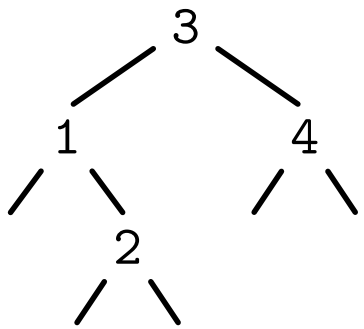
définition

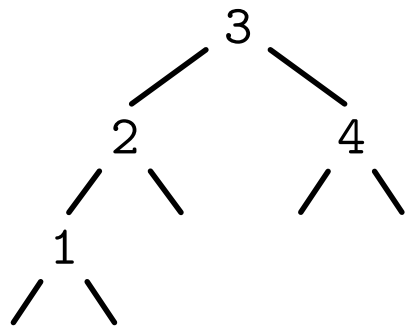
Un arbre binaire de recherche (ABR) est

- un arbre binaire ;
- dont les nœuds contiennent des valeurs qui peuvent être comparées entre elles ;
- et tel que, pour tout nœud de l'arbre,
  - toutes les valeurs situées dans le sous-arbre gauche sont plus petites que la valeur située dans le nœud,
  - toutes les valeurs situées dans le sous-arbre droit sont plus grandes que la valeur située dans le nœud.

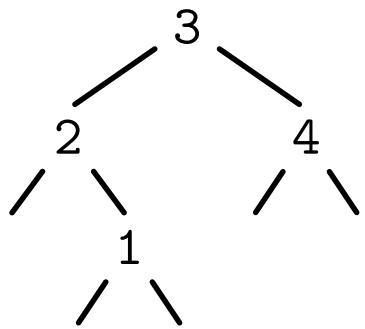
exemples

Les deux arbres





sont des ABR, mais l'arbre



n'en est pas un.

représentation en Python

On conserve la même représentation (c'est un arbre binaire) :

```
class Noeud:
    """un noeud d'un arbre binaire"""
    def __init__(self, g, v, d):
        self.gauche = g
        self.valeur = v
        self.droit  = d
```

La propriété d'ABR sera supposée ou garantie par les fonctions.

rechercher dans un ABR

Tout l'intérêt d'un ABR est de restreindre à chaque étape la recherche à un seul des deux sous-arbres.

C'est analogue à la recherche dichotomique (où on se restreint à un demi-tableau à chaque fois).

rechercher dans un ABR

```
def appartient(x, a):
    """détermine si x apparaît dans l'ABR a"""
```

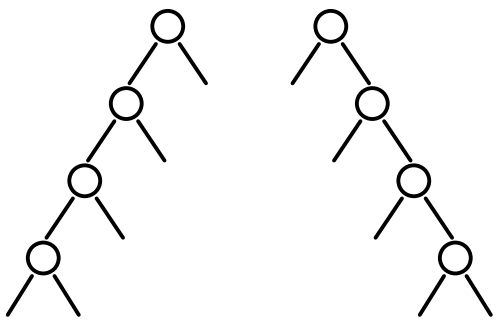
rechercher dans un ABR

```
def appartient(x, a):
    """détermine si x apparaît dans l'ABR a"""
    if a is None:
        return False
    if x < a.valeur:
        return appartient(x, a.gauche)
    elif x > a.valeur:
        return appartient(x, a.droit)
    else:
        return True
```

complexité

La complexité de la recherche dépend de la forme de l'arbre. Dans tous les cas, elle est bornée par la hauteur.

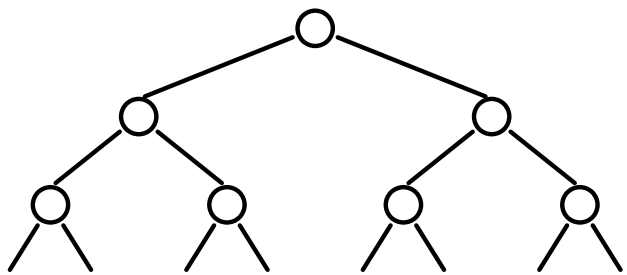
Dans le pire des cas, l'arbre est linéaire (par ex. un peigne) et la recherche peut être amenée à examiner tous les nœuds.



Un ABR linéaire n'est pas meilleur qu'une liste chaînée.

complexité

Si en revanche l'arbre est parfait, alors $h = \log_2(N+1)$ et la recherche est donc d'un coût logarithmique.



Exemple : $N = 10^6$ et $h = 20$.

Vingt comparaisons suffisent à trouver un élément parmi un million.

construire un ABR

Comment construire un ABR ?

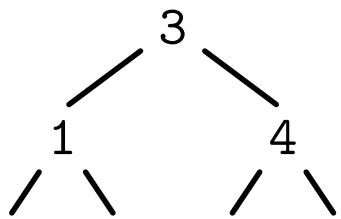
Par exemple, en ajoutant des éléments un par un avec une fonction

```
def ajoute(x, a):
    ...
```

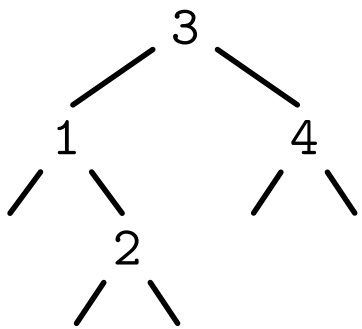
ajouter un élément

A priori, l'ajout procède comme la recherche : on descend à gauche ou à droite, jusqu'à trouver le point d'insertion.

Exemple : pour insérer 2 dans l'arbre



on descend à gauche, puis à droite, puis on greffe une feuille :



ajouter un élément dans un ABR

C'est plus facile à dire qu'à faire.

En particulier, la fonction

```
def ajoute(x, a):
    ...
```

doit-elle modifier l'arbre `a` sans rien renvoyer ?

Mais dans ce cas, que faire lorsque `a` est l'arbre vide, c'est-à-dire `None` ?

ajouter un élément dans un ABR

Il est plus simple d'adopter l'approche où

la fonction `ajoute` renvoie un nouvel arbre,
sans modifier l'arbre a reçu en argument

comme nous l'avons fait avec les listes chaînées (cf mardi).

ajouter un élément dans un ABR

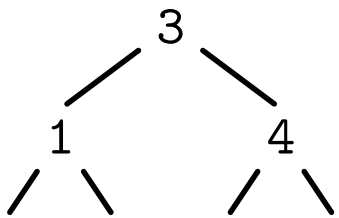
```
def ajoute(x, a):
    """ajoute x à l'arbre a,
       renvoie un nouvel arbre"""
```

ajouter un élément dans un ABR

```
def ajoute(x, a):
    """ajoute x à l'arbre a,
       renvoie un nouvel arbre"""
    if a is None:
        return Noeud(None, x, None)
    if x < a.valeur:
        return Noeud(ajoute(x, a.gauche),
                     a.valeur, a.droit)
    else:
        return Noeud(a.gauche, a.valeur,
                     ajoute(x, a.droit))
```

exemple

On construit l'arbre



en faisant par exemple

```
a = ajoute(3, None)
a = ajoute(1, a)
a = ajoute(4, a)
```

Ici, on remplace à chaque fois l'arbre contenu dans a par l'arbre renvoyé par `ajoute`.

efficacité

Ce n'est pas différent de `appartient` : la complexité dépend de la forme de l'arbre.

encapsulation dans un objet

Comme pour les listes, on peut retrouver une interface impérative en encapsulant nos ABR dans une classe.

encapsulation dans un objet

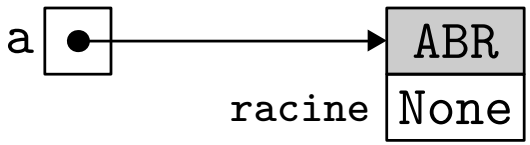
```
class ABR:
    """un arbre binaire de recherche"""
    def __init__(self):
        self.racine = None

    def ajouter(self, x):
        self.racine = ajoute(x, self.racine)

    def contient(self, x):
        return appartient(x, self.racine)
```

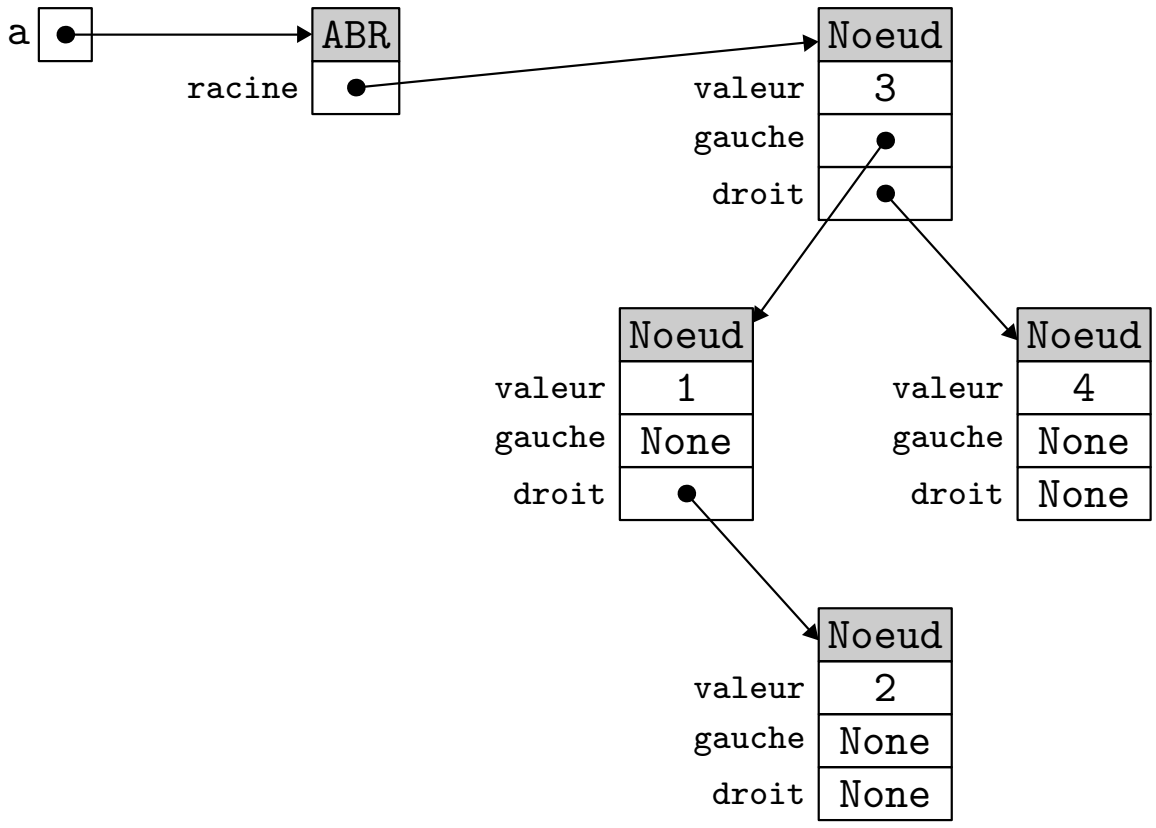
exemple

```
a = ABR()
```



exemple

```
a = ABR()
a.ajouter(3)
a.ajouter(1)
a.ajouter(4)
a.ajouter(2)
```



Exercices