Chapitre 7 - Structures hiérarchiques : arbres

I - Introduction

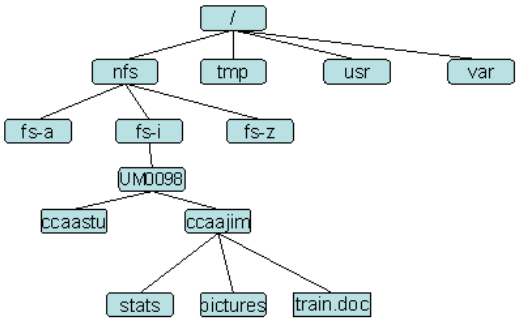
Il n'existe pas uniquement une façon linéaire de structurer les données comme les listes, les piles ou les files. Nous pouvons également structurer de façon hiérarchique.

Définition 1

Un arbre est une **structure hiérarchique** permettant de représenter de manière symbolique des informations structurées.

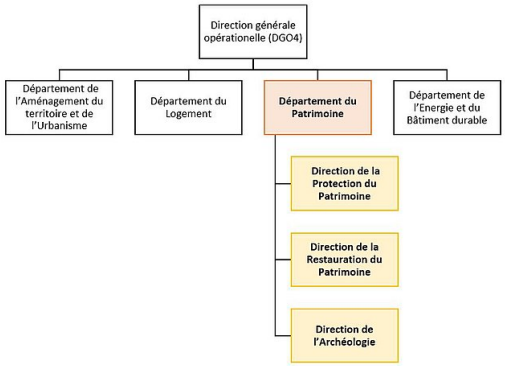
Exemples

- Un dossier, contenant des dossiers et des fichiers, chaque dossier pouvant contenir des dossiers et des fichiers :



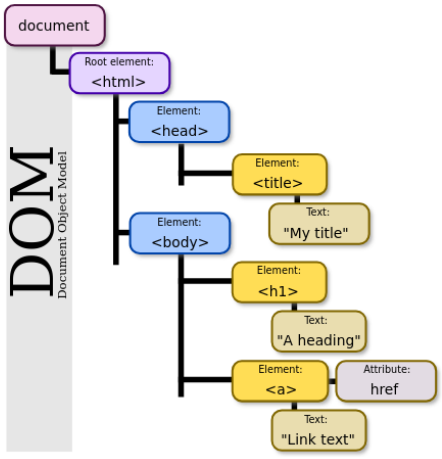
Crédits : Jimbotyson, CC BY-SA 3.0, via Wikimedia Commons.

- Un organigramme :



Crédits : Adriencld, CC BY-SA 4.0, via Wikimedia Commons

- La structure d'une page Web (le DOM (Document Object Model) permet alors de modifier une page Web en modifiant, en ajoutant ou en supprimant des noeuds de l'arbre) :



Crédits : Birger Eriksson, CC BY-SA 3.0, via Wikimedia Commons

Applications

Les applications des arbres sont nombreuses en informatique. Citons par exemple :

- les **arbres de jeu** qui permettent de représenter toutes les positions et tous les coups possibles d'un jeu (voir exercice 2)
- les **arbres syntaxiques** permettant de représenter la structure syntaxique d'une phrase ou d'un code source (lorsque l'interpréteur Python lit du code source, il construit d'abord l'arbre syntaxique du code)
- les **arbres binaires de recherche** qui permettent de rechercher ou d'insérer efficacement un élément dans un arbre (voir Thème 5, Chapitre 2)
- l'**arbre de Huffmann** sur lequel repose la méthode de compression de Huffmann, très utilisée pour compresser des données (textes, images, vidéos ; sons, etc.) et que l'on retrouve dans les compressions JPEG, MPEG, MP3, ZIP, etc.
- le **DOM (Document Object Model)** permet de représenter la structure d'une page Web affichée sous forme d'un arbre, mais aussi de modifier les éléments de la page en modifiant l'arbre (souvent grâce à JavaScript).

II - Arbres quelconques

1) Vocabulaire et définitions

Dans tous ces exemples, on a défini un cas où l'information est élémentaire (fichier, tâche élémentaire), et un cas général où l'information structurée contient deux ou plusieurs informations de même structure.

Dans la terminologie informatique, on utilise les termes de

- **feuille** pour les informations élémentaires,
- **noeud** pour chaque embranchement de l'arbre,
- **racine** pour le(s) noeud(s) principal(aux).

> **Attention** : l'analogie avec les arbres réels peut s'avérer trompeuse. Les arbres - en informatique - sont le plus souvent représentés avec la racine en haut, puis les noeuds, et les feuilles en bas.

Il s'agit d'une **structure de données absraite** permettant de représenter une collection de données par des **noeuds** organisés de manière hiérarchique : il y a un (parfois plusieurs) noeud racine et chaque noeud dépend d'un antécédent (sauf la racine) et a des descendants (sauf les feuilles).

Dans le vocabulaire des arbres on utilise les termes *père* et *fils* pour désigner respectivement un antécédent et les descendants.

- **Père** : chaque noeud possède exactement un seul noeud père, celui dont il est issu, à l'exception de la racine qui n'en a pas.
- **Fils** : chaque noeud peut avoir un nombre arbitraire de noeuds fils, dont il est le père.

Ainsi, avec ces définitions :

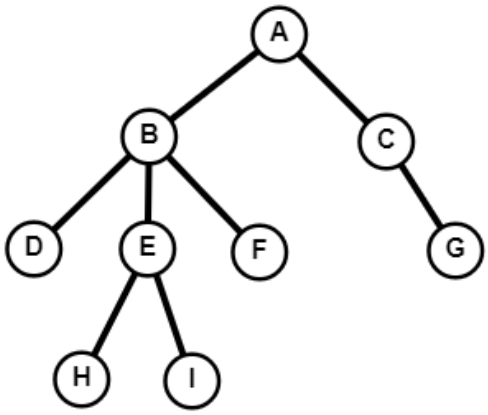
- les **feuilles** sont les noeuds qui n'ont pas de fils,
- la **racine** est un noeud qui n'a pas de père.
- les **noeuds internes** sont les noeuds qui ne sont pas des feuilles.

L'intérêt des arbres est d'y stocker de l'information. Pour cela, chaque noeud peut contenir une ou plusieurs valeurs. **L'information portée par un noeud s'appelle l'étiquette du noeud** (ou la valeur, ou la clé).

Exemple

Dans cet arbre :

- La racine est le noeud A.
- Le noeud B possède 3 fils (les noeuds D, E et F), le noeud C possède un fils (le noeud G), le noeud F ne possède aucun fils.
- Le noeud B a pour père le noeud A.
- Les feuilles sont les noeuds D, H, I, F et G (ceux qui n'ont pas de fils).



2) Caractéristiques d'un arbre

Définition 2

La **taille** d'un arbre est le nombre de noeuds qu'il possède.

Définition 3

La **profondeur** d'un noeud est la longueur du chemin le plus court entre ce noeud et la racine (la racine a donc une profondeur égale à 0).

Définition 4

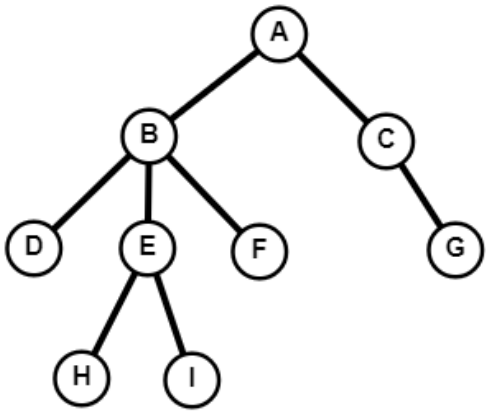
La **hauteur** d'un arbre est la profondeur maximale de ses noeuds (elle vaut 0 pour l'arbre réduit à sa racine et -1 par convention pour un arbre vide).

Attention : On trouve aussi dans la littérature, que la profondeur de la racine est égale à 1, ce qui modifie la hauteur de l'arbre également puisqu'alors l'arbre réduit à la racine a pour hauteur 1 et l'arbre vide a pour hauteur 0. Les deux définitions se valent, il faut donc bien lire celle qui est donnée.

Exemple

Dans cet arbre :

- La taille de l'arbre est égale à 9 (il possède 9 noeuds : 4 noeuds internes et 5 feuilles).
- Le noeud E a une profondeur égale à 2 (le chemin A-B-E a une longueur égale à 2).
- La hauteur de l'arbre est égale à 3 (la prodondeur maximale est égale à 3, c'est celle des noeuds les plus profonds : H et I).



III - Arbres binaires

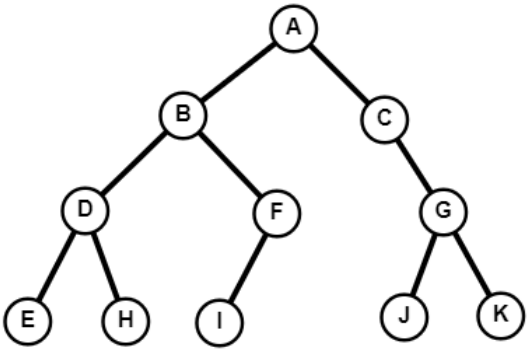
Seuls les arbres binaires sont au programme de Terminale NSI.

1) Définition et vocabulaire spécifique

Définition 5

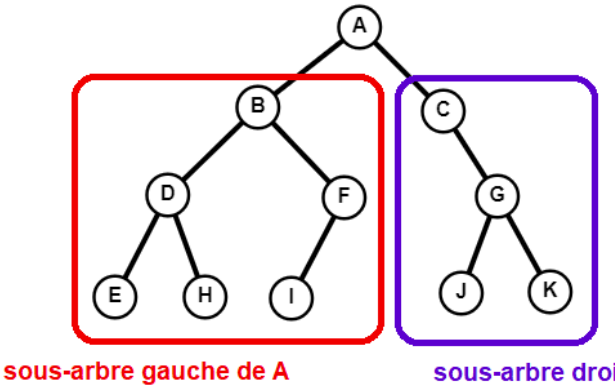
Un **arbre binaire** est un arbre dont tous les noeuds ont **au plus** deux fils.

L'arbre vu dans le paragraphe précédent n'est pas binaire car le noeud B possède 3 fils.
En revanche, l'arbre suivant est binaire.



Les définitions vues précédemment pour des arbres quelconques restent bien évidemment valables pour les arbres binaires. Dans le cas d'un arbre binaire, chaque noeud possède deux sous-arbres, éventuellement vides, que l'on appelle **sous-arbre gauche** et **sous-arbre droit**.

Par exemple, dans le cas de l'arbre binaire précédent, le noeud A possède un sous-arbre gauche et un sous-arbre droit comme la figure ci-contre le montre.



- Les sous-arbres gauche et droit de A sont eux-mêmes des arbres dont les racines sont respectivement B et C.
- Ces noeuds possèdent eux-même des sous-arbres gauche et droit. Par exemple, le noeud C possède un sous-arbre gauche, qui est vide, et un sous-arbre droit qui est l'arbre dont la racine est G.
- Ainsi de suite...

2) Type abstrait `Arbre binaire`

De manière générale, on peut construire un arbre binaire comme un noeud composé de deux sous-arbres.
L'arbre vide est représentée par la valeur `None`.
Ainsi, une feuille est un noeud avec les sous-arbres gauche et droit à `None`.
Pour annoter la structure de l'arbre avec des informations, on utilise des étiquettes pouvant être enregistrées à chaque noeud.
On peut ensuite parcourir un arbre par l'accès à son étiquette et à ses sous-arbres droit et gauche.
Un prédicat permet de distinguer les feuilles des noeuds.

On peut ainsi spécifier un arbre binaire par le type abstrait suivant :

- Constructeur : `noeud : Etiquette × Arbre binaire × Arbre binaire → Arbre binaire`
- Sélecteurs :
  - `droit : Arbre binaire → Arbre binaire`
  - `gauche : Arbre binaire → Arbre binaire`
  - `etiquette : Arbre binaire → Etiquette`
- Prédicat : `est_feuille : Arbre binaire → Booléen`

Implémentation

Il existe, comme toujours, plusieurs implémentations possibles d'un arbre binaire. Une implémentation classique consiste à représenter chaque noeud comme un objet d'une classe `Noeud`.

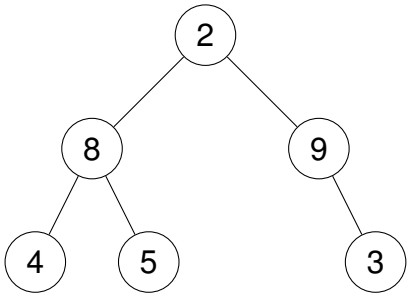
```
class Noeud:
    def __init__(self, e, g=None, d=None):
        self.etiquette = e
        self.gauche = g
        self.droit = d

    def est_feuille(self):
        return not self.gauche and not self.droit

    # Une representation possible de l'arbre
    def __repr__(self):
        ch = str(self.etiquette)
        if self.gauche or self.droit:
            ch = ch + '-(' + str(self.gauche) + ',' + str(self.droit) + ')'
        return ch
```

Il y a une petite subtilité à bien comprendre pour la méthode `__init__` : on a choisi de définir la valeur None par défaut aux arguments gauche et droit.
Cela permet de construire une feuille (d'étiquette 'a') en écrivant `Noeud('a')` au lieu de `Noeud('a', None, None)`.

La construction d'un arbre s'effectue alors avec des noeuds ayant soit un seul argument (cas des feuilles), soit trois (cas général).

```
A1 = Noeud(2, Noeud(8, Noeud(4), Noeud(5)), Noeud(9, None, Noeud(3)))
print(A1)
#OUTPUT
2-(8-(4,5),9-(None,3))
```

L'arbre `A1` ainsi construit représente l'arbre binaire ci-dessous.



Calcul de la taille et de la hauteur d'un arbre binaire

La définition d'un arbre binaire étant récursive, il est naturel d'écrire des **algorithmes récursifs** pour effectuer des opérations sur les arbres binaires.

En particulier, on peut écrire assez facilement deux fonctions récursives `taille` et `hauteur` qui calculent respectivement la taille et la hauteur d'un arbre binaire. Il suffit de parcourir récursivement l'arbre avec les méthodes `gauche` et `droit`.

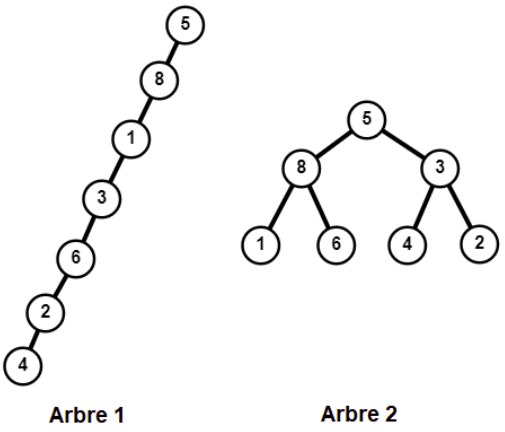
```
def taille(A):
    """Renvoie la taille d'un arbre binaire A."""
    if A is None:
        return 0
    else:
        return 1 + taille(A.gauche) + taille(A.droit)

def hauteur(A):
    """Renvoie la hauteur d'un arbre binaire A"""
    if A is None:
        return -1
    else:
        return 1 + max(hauteur(A.gauche), hauteur(A.droit))
```

On obtient alors :

```
>>> taille(A1)
6
>>> hauteur(A1)
2
```

Encadrement de la hauteur d'un arbre binaire

La hauteur d'un arbre binaire est la profondeur maximale de ses noeuds. Cependant un arbre binaire d'une taille donnée peut avoir un aspect totalement différent. En effet, les deux arbres binaires suivants sont de même taille (égale à 7) mais ont des "formes" très différentes.



Le premier est dit filiforme ou dégénéré tandis que le second est dit parfait (un arbre est dit parfait si tous les niveaux sont remplis c'est-à-dire si chaque noeud interne a exactement deux fils).

Si on cherche à encadrer la hauteur d'un arbre binaire, ces deux exemples fournissent les deux cas de figure extrêmes :

- Le premier arbre est l'arbre binaire de taille 7 dont la hauteur est maximale, elle vaut 6 (le noeud le plus profond a une profondeur égale à 6).
- Le second arbre un arbre binaire de taille 7 dont la hauteur est minimale, elle vaut 2 (les 4 feuilles sont toutes à une profondeur égale à 2).

De manière générale, on a l'encadrement suivant.

Si on note H la hauteur d'un arbre binaire à N noeuds, alors :

$$E(\log_2 N) \leq H \leq N-1$$

où $E(\log_2 N)$ est la partie entière du logarithme en base 2 de N, c'est-à-dire le nombre de bits nécessaire à son écriture en base 2 diminué d'une unité (c'est la définition des informaticiens).

On peut vérifier cela avec les arbres précédents de taille $N = 7$

- D'une part, $N - 1 = 6$ qui correspond bien à la taille de l'arbre filiforme (arbre 1).
- D'autre part, $7 = (111)_2$ donc il faut 3 bits pour écrire 7 en base deux. On en déduit que $E(\log_2 7) = 3 - 1 = 2$ qui est bien la hauteur de l'arbre parfait (arbre 2).

Ces deux arbres étant les cas extrêmes, un arbre binaire à 7 noeuds a une hauteur comprise entre 2 et 6.

IV - Algorithmes sur les arbres binaires

1) Parcours d'un arbre binaire

Parcourir un arbre, c'est visiter tous ses noeuds, afin de pouvoir opérer une action tour à tour sur eux. Un parcours d'arbre définit dans quel ordre les noeuds sont visités.

1).1 Parcours en largeur d'abord

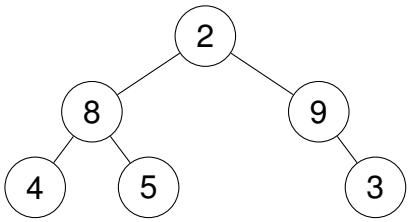
Définition 6

Dans le cas où un arbre est parcouru **niveau par niveau** (en commençant par la racine est en lisant de gauche à droite) on parle d'un **parcours en largeur** d'abord. On utilise le terme largeur car dans ce cas on explore les noeuds en balayant en largeur chaque niveau de l'arbre.

Exemple

Dans le cas d'un parcours de cet arbre binaire en largeur d'abord, les noeuds sont visités dans l'ordre : 2 - 8 - 9 - 4 - 5 - 3.

Un parcours en largeur d'abord n'est pas récursif.



Algorithme de parcours en largeur

L'utilisation d'une **file** permet d'écrire facilement l'algorithme de parcours en largeur d'abord. Le principe est le suivant :

- → On enfile l'arbre de départ
- → Tant que la file n'est pas vide :
  - → on défile un élément
  - → si celui-ci n'est pas un arbre vide :
    - → on affiche son étiquette
    - → on enfile ses fils gauche et droit (dont les racines sont les noeuds du niveau suivant)

On utilise ainsi la file pour y insérer et donc traiter (en défilant) tour à tour les noeuds, niveau par niveau. Pour l'implémentation qui suit, notre file est implémentée par un objet d'une classe `File`.

```
def parcours_en_largeur(A):
    """Affiche les etiquettes de l'arbre binaire A selon un parcours en largeur."""
    F = File()
    F.enfiler(A)
    while F.taille() != 0:
        a = F.defiler()  # renvoie le sommet
        if a is not None:
            print(a.etiquette, end=" ")
            F.enfiler(a.gauche)
            F.enfiler(a.droit)
```

On peut vérifier en appliquant cette fonction à l'arbre de l'exemple précédent :

```
A1 = Noeud(2, Noeud(8, Noeud(4), Noeud(5)), Noeud(9, None, Noeud(3)))
print(A1)
#OUTPUT : 2-(8-(4,5),9-(None,3))

parcours_en_largeur(A1)
#OUTPUT : 2 8 9 4 5 3
```

1).2 Parcours en profondeur

Définition 7

Dans le cas où on explore complètement l'un des deux sous-arbres avant le second on parle d'un **parcours en profondeur**. On utilise le terme profondeur car dans ce cas on tente toujours de visiter le noeud le plus éloigné de la racine à condition qu'il soit le fils d'un noeud déjà visité.

On distingue trois ordres particuliers pour explorer en profondeur les sous-arbres gauche, droit et la racine du noeud courant :

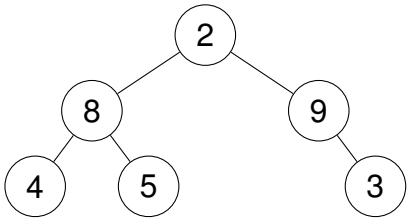
- **ordre préfixe** : On traite d'abord la racine de l'arbre, puis son sous-arbre gauche et son sous-arbre droit : **Racine - SAG - SAD**.
- **ordre infixe** : On traite d'abord le sous-arbre gauche, puis la racine, puis le sous-arbre droit : **SAG - Racine - SAD**
- **ordre suffixe (ou postfixe)** : On traite d'abord le sous-arbre gauche, puis le sous-arbre droit, puis la racine : **SAG - SAD - Racine**

Remarque : Les termes prefixe, infixe, et suffixe font référence à l'endroit où l'on place la racine dans l'ordre de parcours.

Exemple

Voici l'ordre des noeuds visités dans les 3 ordres de parcours en profondeur :

- Parcours préfixe : 2 - 8 - 4 - 5 - 9 - 3
- Parcours infixe : 4 - 8 - 5 - 2 - 9 - 3
- Parcours suffixe : 4 - 5 - 8 - 3 - 9 - 2



Algorithme de parcours en profondeur

Les algorithmes de parcours en profondeur s'écrivent facilement de manière **récursive**. Pour l'algorithme de parcours suivant l'ordre **préfixe** on procède ainsi :

→ si l'arbre n'est pas vide :
  → on affiche l'étiquette de sa racine
  → on parcourt récursivement son sous-arbre gauche
  → puis son sous-arbre droit
→ (sinon on ne fait rien)

Voici une fonction récursive qui implémente le parcours préfixe :

```
def parcours_prefixe(A):
    """Affiche les etiquettes de l'arbre binaire A selon un parcours prefixe."""
    if A is not None:
        print(A.etiquette, end=" ")  # etiquette affichee avant SAG et SAD
        parcours_prefixe(A.gauche)
        parcours_prefixe(A.droit)
```

```
parcours_prefixe(A1)
#OUTPUT : 2 8 4 5 9 3
```

Il suffit de changer l'ordre des lignes 4, 5 et dans le if pour retrouver les ordres infixe et suffixe de parcours des noeuds.

```
def parcours_infixe(A):
    """Affiche les etiquettes de l'arbre binaire A selon un parcours infixe."""
    if A is not None:
        parcours_infixe(A.gauche)
        print(A.etiquette, end=" ")  # etiquette affichee entre SAG et SAD
        parcours_infixe(A.droit)

def parcours_suffixe(A):
    """Affiche les etiquettes de l'arbre binaire A selon un parcours suffixe."""
    if A is not None:
        parcours_suffixe(A.gauche)
        parcours_suffixe(A.droit)
        print(A.etiquette, end=" ")  # etiquette affichee apres SAG et SAD
```

```
parcours_infixe(A1)
#OUTPUT : 8 4 5 2 9 3
parcours_suffixe(A1)
#OUTPUT : 8 4 5 9 3 2
```

1).3 Recherche dans un arbre binaire

Pour vérifier sur une étiquette `e` est présente dans un noeud d'un arbre binaire `A`, il faut le parcourir (en largeur ou en profondeur). Dans le pire cas, c'est-à-dire si l'étiquette est absente, il faut bien regarder tous les noeuds pour conclure. Ainsi, pour un arbre binaire de N noeuds, l'algorithme de recherche a un coût en temps de l'ordre de N (noté $O(N)$).

- La recherche dans un arbre binaire prend un temps similaire à la recherche dans un tableau ou dans une liste.
- Mais dans un tableau, on a vu que sous hypothèse de tri, on pouvait faire mieux : une recherche dichotomique ! (voir programme de Première NSI).
- Peut-on faire de même dans un arbre binaire ? Autrement dit, quelle hypothèse d'ordre faire pour permettre une recherche plus efficace ? Réponse : oui, en utilisant des **arbres binaires de recherche**.

2) Arbre binaire de recherche

Définition 8

Un **arbre binaire de recherche**, abrégé **ABR**, est un arbre binaire dans lequel tout noeud a une clé (= étiquette) :

- plus grande ou égale à celles de son sous-arbre gauche
- plus petite strictement que celles de son sous-arbre droit

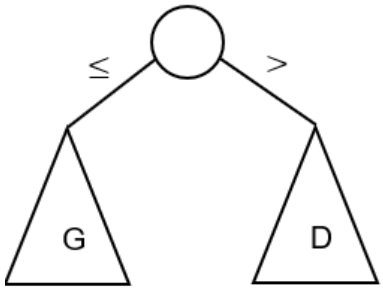
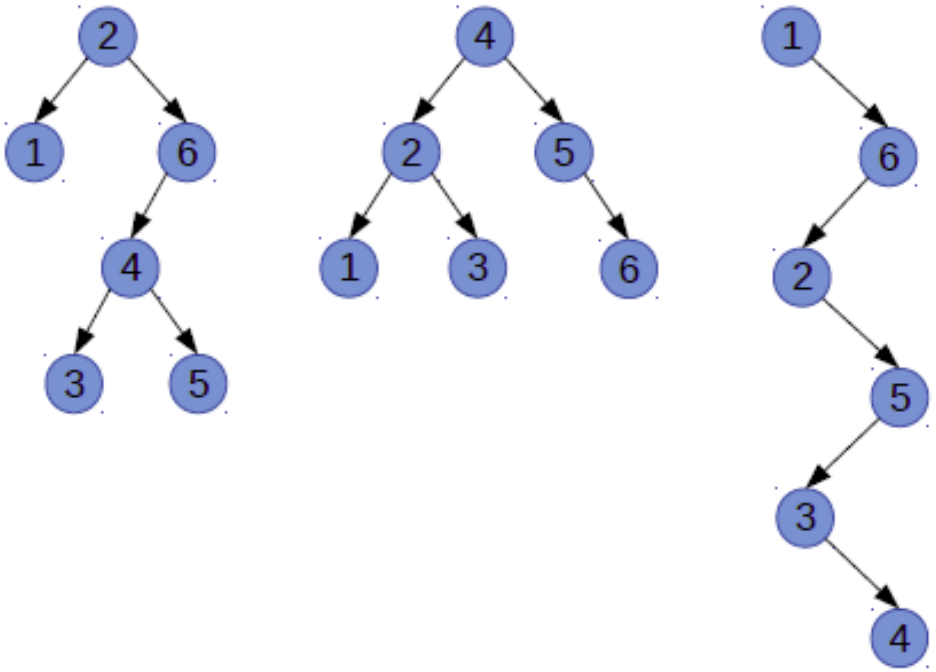


Schéma d'un ABR

Exemples : Voici quelques exemples d'ABR :



On construit l'arbre de gauche par l'objet `A2` de la classe `Noeud` ci-dessous :

```
A2 = Noeud(2, Noeud(1), Noeud(6, Noeud(4, Noeud(3), Noeud(5))))
print(A2)
#OUTPUT : 2-(1,6-(4-(3,5),None))
```

2).1 Rechercher une clé dans un ABR

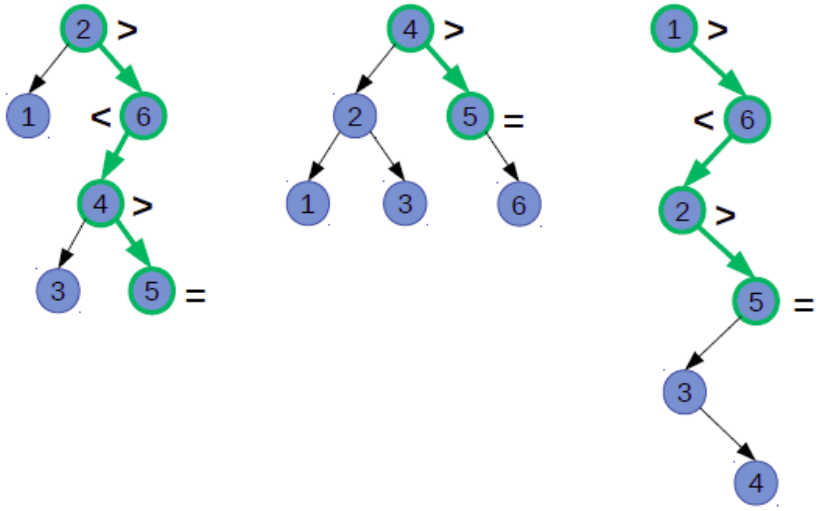
La propriété d'ordre en chaque noeud d'un ABR assure qu'il existe un unique chemin pour toute clé stockée : la comparaison en chaque noeud indique si la recherche doit être poursuivie à gauche ou à droite. La recherche est fructueuse si la clé est trouvée en un noeud ; infructueuse s'il est aboutit à un sous-arbre vide.

Cela permet d'écrire facilement l'algorithme récursif de recherche d'une clé dans un ABR :

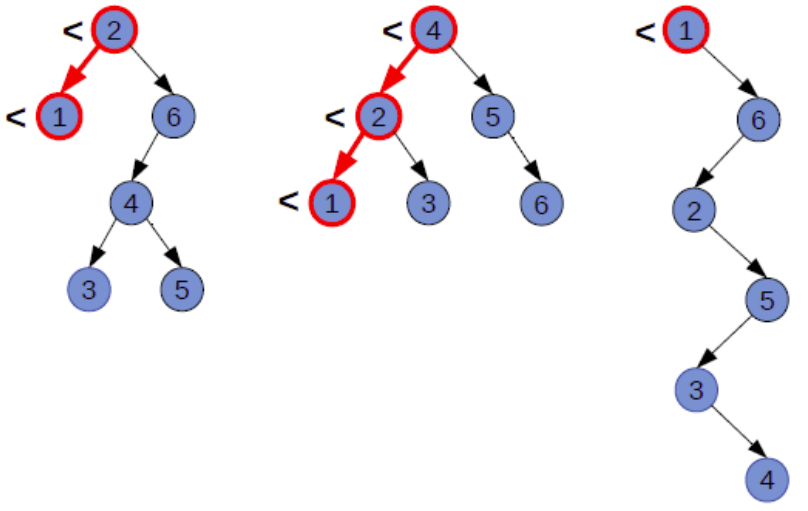
```
def etq_presente(A, e):
    """Renvoie True si l'etiquette e est presente dans l'ABR A, et False sinon."""
    if A is None:
        return False
    if e == A.etiquette:
        return True
    elif e < A.etiquette:
        return etq_presente(A.gauche, e)
    else:
        return etq_presente(A.droit, e)
```

Exemples

- `etq_presente(A, 5)` renvoie `True` dans les trois cas :



- `etq_presente(A, 0)` renvoie `False` dans les trois cas :



Remarques : La propriété d'ordre sur les clés d'un ABR implique :

- que l'on trouve le **minimum** en se déplaçant systématiquement à gauche : le dernier noeud atteint avant un sous-arbre gauche vide est le minimum ;
- que l'on trouve le **maximum** en se déplaçant systématiquement à droite : le dernier noeud atteint avant un sous-arbre droit vide est le maximum ;
- que le **parcours par ordre infixe** d'un ABR donne les clés dans l'**ordre croissant**.

2).2 Insérer une clé dans un ABR

Le principe de l'ajout d'une clé est simple : pour que l'élément que l'on va ajouter soit retrouvé lors d'une future recherche, il faut l'insérer à l'endroit où conduira cette recherche. Cela conduit à suivre un chemin unique dans l'ABR et on insère le nouveau noeud avec la clé dès qu'on aboutit à un sous-arbre vide.

On présente ici une version qui renvoie un nouvel arbre à chaque insertion car elle est simple à écrire et permet de gérer facilement le cas de l'insertion dans un arbre vide représenté par `None`.

> On peut écrire des versions avec modification en place de l'arbre passé en argument mais cela rend l'algorithme plus long à écrire et on doit réserver un cas particulier pour l'insertion dans un arbre vide.

Insérer une clé `e` dans un ABR `A` revient à construire un ABR qui contient `e` et toutes les clés de `A`. Le principe est relativement simple :

→ si l'ABR est vide, on construit un ABR possédant un unique noeud de clé `e`.

→ si `e` est inférieure ou égale à l'étiquette de `A` il faut insérer la clé dans le sous-arbre gauche de `A` ce qui revient à créer un nouvel ABR dont :

  → l'étiquette est celle de `A` (inchangée) ;

  → le sous-arbre gauche est le sous-arbre gauche de `A` dans lequel on insère la clé `e` ⇒ appel récursif

  → le sous-arbre droit est celui de `A` (inchangé).

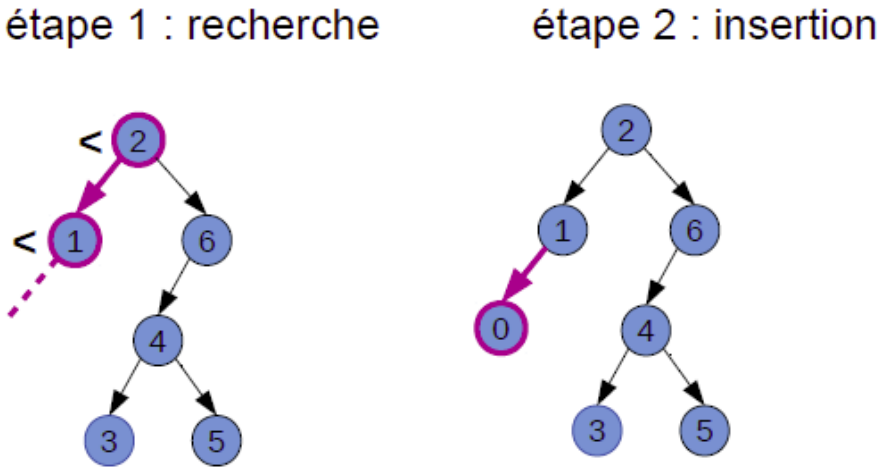
→ si `e` est strictement supérieure à l'étiquette de `A` il faut insérer la clé dans le sous-arbre droit de `A` en procédédant de manière similaire.

On aboutit à la fonction ajouter suivante :

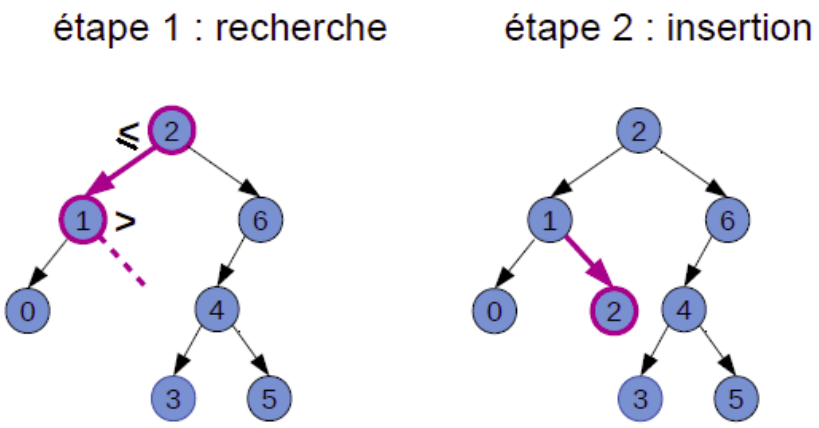
```
def ajouter(A, e):
    """Renvoie un nouvel ABR contenant les cles de l'ABR A ainsi que la cle e."""
    if A is None:
        return Noeud(e, None, None)
    elif e <= A.etiquette:
        return Noeud(A.etiquette, ajouter(A.gauche, e), A.droit)
    else:
        return Noeud(A.etiquette, A.gauche, ajouter(A.droit, e))
```

Remarques et exemples

- L'insertion revient à créer une feuille. Par exemple, `ajouter(A, 0)` :



- Même en cas d'égalité, on descend toujours jusqu'à un sous-arbre vide. Par exemple, `ajouter(A, 2)` :




```
A2 = Noeud(2, Noeud(1), Noeud(6, Noeud(4, Noeud(3), Noeud(5))))
A2 = ajouter(A2, 0)
print(A2)
#OUTPUT : 2-(1-(0,None),6-(4-(3,5),None))

A2 = ajouter(A2, 2)
print(A2)
#OUTPUT : 2-(1-(0,2),6-(4-(3,5),None))
```

2).3 Complexités (temporelles)

Coût de la recherche

La recherche d'une clé dans un ABR conduit, dans le pire cas, à parcourir un chemin de la racine jusqu'à une feuille. Le coût de cet algorithme est donc de l'ordre de la hauteur de l'arbre (= profondeur maximale des feuilles).
Ainsi, le coût de la recherche est de l'ordre de :

- N si l'arbre est filiforme, soit comme la recherche dans un tableau ou dans une liste ;
- $\log_2 N$ si l'arbre est parfait, soit le même coût que la recherche dichotomique dans un tableau trié.

Coût de l'insertion

L'insertion d'une clé se fait au niveau d'une feuille, ce qui conduit toujours à parcourir un chemin de la racine jusqu'à une feuille. On en déduit que le pire cas est égal au meilleur cas et que le coût de l'insertion est donc le même que celui de la recherche : de l'ordre de $\log_2 N$ dans le cas d'un ABR bien équilibré.

Source : Lycée Mounier - Angers