Chapitre 2

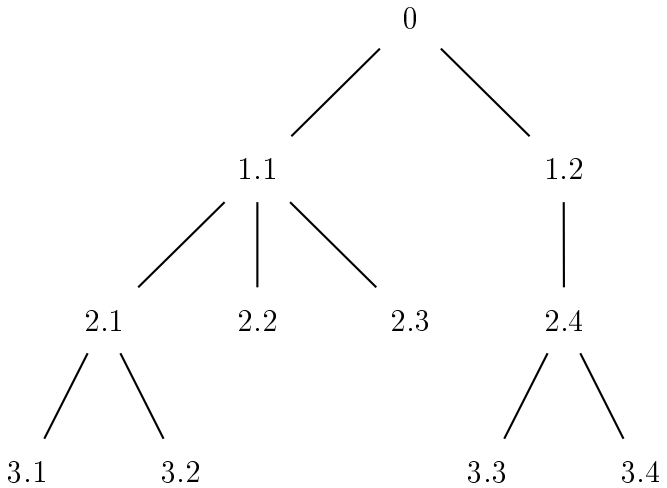
Arbres binaires

2.1 Arbre

2.1.1 Généralités

Un **arbre** est une structure abstraite hiérarchisée d'éléments. Ces éléments constituant l'arbre sont appelés **nœuds**, chaque nœud étant relié par des **arêtes** à un ou plusieurs **sous-arbres** enfants ou à rien. Un nœud n'ayant aucun sous-arbre enfant est appelé **feuille**. Le premier nœud de l'arbre est quant à lui appelé **racine**.

Exemple : Dans l'arbre ci-dessous, 0 est la racine et 1.1 et 1.2 sont ses nœuds enfants. 2.2, 2.3 et 3.1 à 3.4 sont des feuilles car ils ne possèdent aucun sous-arbre enfant.



Les arbres se rencontrent notamment en informatique afin de représenter les situations suivantes :

— L'arborescence des fichiers dans un ordinateur (à laquelle il est possible d'accéder à l'aide de la commande `tree` dans le terminal sous Linux).

— L'arborescence des processus d'un ordinateur.

— L'organisation d'une page HTML : le DOM. Lequel est à la base de la dynamisation des pages Web à l'aide de JavaScript.

Remarque : contrairement aux listes, piles et files, les arbres ne sont pas des structures linéaires ; même s'il est possible d'écrire un arbre sous forme d'une liste comme on le verra plus bas. On remarquera toutefois que cette liste n'est pas homogène : elle contient différents types de valeurs, ce qui ne correspond pas à la définition de la structure abstraite de liste.

2.1.2 Propriétés

Les caractéristiques suivantes sont valables pour tout type d'arbre.

Arité de l'arbre : nombre maximal d'enfants par nœud.

Profondeur d'un nœud : longueur du chemin entre la racine et du nœud (comptée en nombre de nœuds, ceux de départ et d'arrivé compris).

Hauteur de l'arbre : valeur de la profondeur maximale de l'arbre.

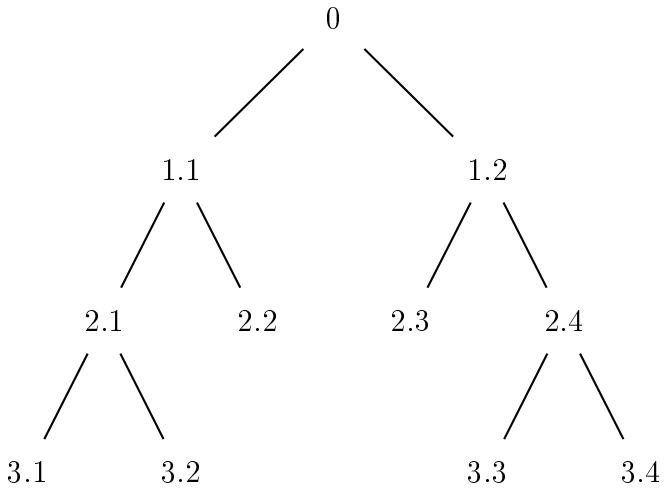
Taille de l'arbre : nombre d'éléments de l'arbre.

Exemple : on reprend l'arbre donné en exemple ci-dessus. Celui-ci est d'arité 3, de taille 11 et de hauteur 4. L'élément 2.4 est situé à une profondeur de 2.

2.1.3 Arbres binaires

Un **arbre binaire** est un arbre particulier où le nombre d'enfants de chaque nœud est au plus égal à deux. Autrement, chaque nœud peut avoir zéro, un ou deux enfants.

Exemple : L'arbre du premier n'est pas binaire car il possède un nœud ayant trois enfants. L'arbre ci-dessous est binaire, tous ces nœuds ont deux enfants au plus.



Il est possible noter un arbre sous la forme de liste avec la convention ci-dessous.

`[noeud, [sous-arbre gauche], [sous-arbre droit]]`,

où les sous-arbres obéissent à la même structure. Une feuille revient à avoir des sous-arbres vides, on peut pourra donc les noter sous la forme

`[feuille, [], []]`.

Exemple : l'arbre binaire de l'exemple ci-dessus peut s'écrire sous la forme :

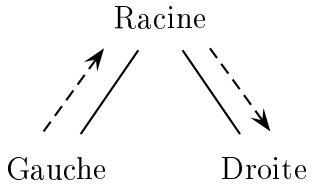
```
[0, [1.1, [2.1, [3.1, [], []], [3.2, [], []]], [2.2, [], []]], [1.2, [2.3, [],
[]], [2.4, [3.3, [], []], [3.4, [], []]]]].
```

2.2 Parcours en profondeur d'arbres

Un parcours d'arbre consiste à donner la liste des nœuds l'arbre dans un certain ordre. Nous allons voir trois parcours dits en profondeur : infixe, suffixe et préfixe.

2.2.1 Parcours infixe

Ce parcours peut se résumer par la règle Enfant Gauche, Racine, Enfant Droit (ou GRD). On parcourt récursivement le sous-arbre de gauche, puis la racine et enfin le sous-arbre de droite.



Exemple : le parcours infixe de l'arbre binaire donné en exemple ci-dessus est

3.1 - 2.1 - 3.2 - 1.1 - 2.2 - 0 - 2.3 - 1.2 - 3.3 - 2.4 - 3.4

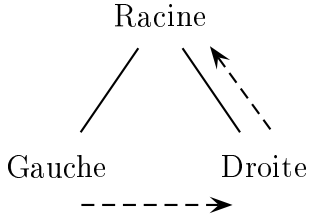
Plus précisément, la fonction de parcours infixe est donnée par l'algorithme récursif ci-dessous. Le cas de base étant un arbre vide auquel nous sommes certains de finir par arriver.

Algorithme 1 : Parcours infixe

```
1 Fonction infixe(arbre) :
2 |
3 |   Si arbre non vide  :
4 |   |
5 |   |   infixe(enfant gauche)
6 |   |   Afficher : racine
7 |   |   infixe(enfant droit)
```

2.2.2 Parcours suffixe

Ce parcours peut se résumer par la règle Enfant Gauche, Enfant Droit, Racine (ou GDR). On parcourt récursivement le sous-arbre de gauche, puis le sous-arbre de droite et enfin la racine.



Exemple : le parcours suffixe de l'arbre binaire donné en exemple ci-dessus est

3.1 - 3.2 - 2.1 - 2.2 - 1.1 - 2.3 - 3.3 - 3.4 - 2.4 - 1.2 - 0

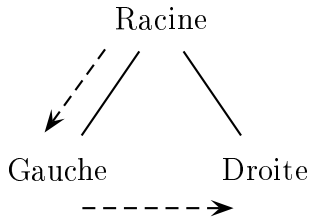
Plus précisément, la fonction de parcours suffixe est donnée par l'algorithme récursif ci-dessous. Le cas de base étant un arbre vide auquel nous sommes certains de finir par arriver.

Algorithme 2 : Parcours suffixe

```
1 Fonction suffixe(arbre) :
2 |
3 |  Si arbre non vide :
4 |  |
5 |  |  suffixe(enfant gauche)
6 |  |  suffixe(enfant droit)
7 |  |  Afficher : racine
```

2.2.3 Parcours préfixe

Ce parcours peut se résumer par la règle Racine, Enfant Gauche, Enfant Droit (ou RGD). On parcourt récursivement la racine puis le sous-arbre de gauche et enfin sous-arbre de droite.



Exemple : le parcours suffixe de l'arbre binaire donné en exemple ci-dessus est

0 - 1.1 - 2.1 - 3.1 - 3.2 - 2.2 - 1.2 - 2.3 - 2.4 - 3.3 - 3.4

Plus précisément, la fonction de parcours préfixe est donnée par l'algorithme récursif ci-dessous. Le cas de base étant un arbre vide auquel nous sommes certains de finir par arriver.

Algorithme 3 : Parcours préfixe

```
1 Fonction préfixe(arbre) :
2 |
3 |  Si arbre non vide :
4 |  |
5 |  |  Afficher : racine
6 |  |  préfixe(enfant gauche)
7 |  |  préfixe(enfant droit)
```

2.3 Parcours en largeur d'arbres

Le parcours en largeur (BFS pour *Breadth First Search* en anglais) est un parcours d'arbres où tous les nœuds d'un niveau sont donnés de la gauche vers la droite avant de passer au niveau suivant.

Exemple : toujours avec le même arbre binaire des exemples ci-dessus, on a pour parcours en largeur :

0 - 1.1 - 1.2 - 2.1 - 2.2 - 2.3 - 2.4 - 3.1 - 3.2 - 3.3 - 3.4

Bien qu'étant un parcours intuitif, l'algorithme donnant ce résultat l'est moins. En effet, il est fait intervenir la notion de file : à chaque fois qu'un nœud est considéré, on stocke dans une file d'attente (laquelle est initialisée avec la racine) ses enfants avant de les faire défiler.

Exemple : reprenons notre arbre exemple et détaillons à chaque étape la file d'attente et les nœuds affichés / considérés. La file est initialisée avec la racine.

Considéré : None
File : 0
Affichés :

Considéré : 0
File : 1.1, 1.2
Affichés : 0

Considéré : 1.1
File : 1.2, 2.1, 2.2
Affichés : 0, 1.1

Considéré : 1.2
File : 2.1, 2.2, 2.3, 2.4
Affichés : 0, 1.1, 1.2

Considéré : 2.1
File : 2.2, 2.3, 2.4, 3.1, 3.2
Affichés : 0, 1.1, 1.2, 2.1

Considéré : 2.2
File : 2.3, 2.4, 3.1, 3.2
Affichés : 0, 1.1, 1.2, 2.1, 2.2

Considéré : 2.3
File : 2.4, 3.1, 3.2
Affichés : 0, 1.1, 1.2, 2.1, 2.2, 2.3

Considéré : 2.4
File : 3.1, 3.2, 3.3, 3.4
Affichés : 0, 1.1, 1.2, 2.1, 2.2, 2.3, 2.4

À partir de cette dernière étape, il ne reste plus que des feuilles dans la file. En la faisant défiler, on obtient le parcours final :

0 - 1.1 - 1.2 - 2.1 - 2.2 - 2.3 - 2.4 - 3.1 - 3.2 - 3.3 - 3.4

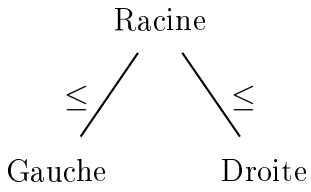
L'algorithme du parcours en larguer est donné en pseudo-code ci-dessous. Notons par ailleurs qu'il existe un version récursive de ce parcours ; l'algorithme correspondant n'est pas donné cependant.

Algorithme 4 : Parcours en largeur

```
1  Fonction parcours_largeur(arbre) :
2
3      Si arbre non vide :
4
5          file ← enfiler(racine)
6
7          Tant que file non vide :
8
9              nœud ← défiler(file)
10             Afficher : nœud
11
12             Si sous-arbre_gauche(nœud) non vide :
13
14                 file ← enfiler(sous-arbre_gauche(nœud))
15
16             Si sous-arbre_droit(nœud) non vide :
17
18                 file ← enfiler(sous-arbre_droit(nœud))
```

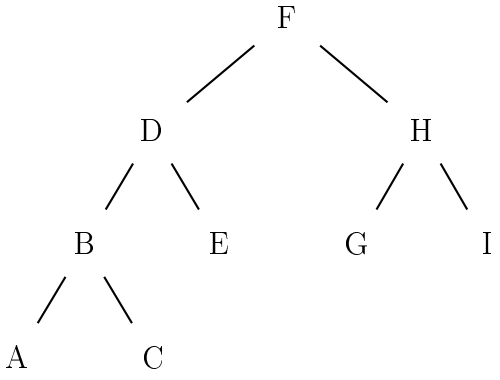
2.4 Arbres binaires de recherche

Un **Arbre Binaire de Recherche** (**ABR**) est un arbre binaire dans lequel, pour tout nœud, toutes les valeurs des nœuds du sous-arbre gauche sont inférieures à celle du nœud considéré, laquelle est inférieure à toutes celles du sous-arbre droit. Autrement dit, les valeurs sont rangées dans l'ordre croissant (numériquement ou alphabétiquement) selon le motif



$$\text{valeurs gauches} \leq \text{valeur nœud} \leq \text{valeurs droites}.$$

Remarque : La plupart des opérations, notamment de recherche, dans un arbre binaire de recherche sont rapides puisque tous les éléments y sont triés. On a une complexité temporelle de recherche en $O(\ln(n))$ identique à celle de la recherche dichotomique, ce qui est naturel dans la mesure où le processus est similaire.



Exemple : l'arbre ci-contre est un arbre binaire de recherche, toutes ses valeurs sont triées dans l'ordre alphabétique.

L'insertion d'une valeur dans un arbre binaire de recherche se fait assez simplement. On parcourt l'arbre tant que l'on n'est pas arrivé à une feuille selon le principe : si ma valeur à insérer est plus petite que le nœud en cours, je vais dans le sous-arbre gauche, sinon dans le droit. Une fois arrivé à la feuille correspondante, soit j'insère à gauche, soit à droite afin de respecter l'ordre. Cela est donné par l'algorithme ci-dessous.

Algorithme 5 : Parcours en largeur

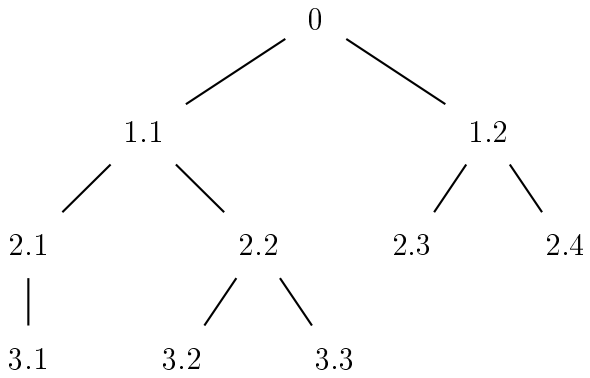
```
1  Fonction inserer(ABR, valeur) :
2  |
3  |  Tant que ABR non vide :
4  |  |
5  |  |  arbre ← ABR // mémorisé pour le cas où on a une feuille
6  |  |
7  |  |  Si valeur < racine(ABR) :
8  |  |  |  ABR ← sous-arbre_gauche
9  |  |  Sinon
10 |  |  |  ABR ← sous-arbre_droit
11 |  |
12 |  Si valeur < racine(arbre) :
13 |  |  insérer à gauche
14 |  Sinon
15 |  |  insérer à droite
```

2.5 Exercices

2.5.1 Démarrage

Exercice 2.1. On considère l'arbre ci-contre.

1. Quel type d'arbre est-ce ?
2. Quelle est la profondeur du nœud 2.3 ?
3. Quelle est la hauteur de cet arbre ?
4. Quelle est la taille de cet arbre ?
5. Écrire cet arbre sous la forme d'une liste.



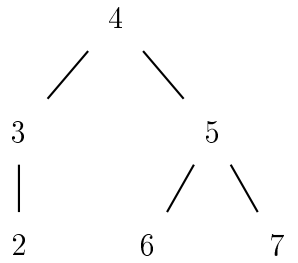
Exercice 2.2. Dessiner l'arbre correspondant à la liste suivante.

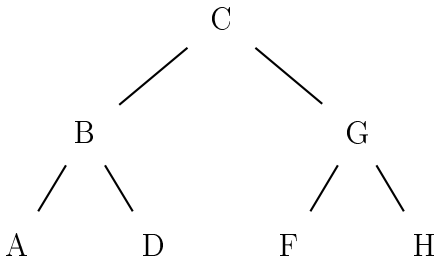
`[0, [1.1, [2.1, [], []], []], [1.2, [2.2, [], []], [2.3, [3.3, [], []], [3.4, [], []]]]]`.

Exercice 2.3. Donner les parcours infixe, suffixe et préfixe de l'arbre de l'exercice 2.1.

Exercice 2.4. Donner le parcours en largeur de l'arbre de l'exercice 2.1 en détaillant à chaque étape l'état de la file et des éléments affichés.

Exercice 2.5. Les arbres ci-dessous sont-ils des arbres binaires de recherche ? Si non, proposer une modification afin qu'ils en deviennent.





2.5.2 Approfondissement

Exercice 2.6. [Taille maximale d'un arbre] Quelle est la taille maximale d'un arbre binaire de hauteur n ? Même question avec un arbre d'arité quelconque.

Exercice 2.7. [Arbre binaire et POO] Le but de cet exercice est de programmer une classe **Arbre Binaire**. Pour cela, nous allons d'abord commencer par programmer une classe **Nœud** contenant les attributs d'un nœud (donnés dans le schéma UML ci-dessous) puis nous programmerons ensuite la classe **Arbre Binaire** à partir de celle-ci.

Nœud
valeur
enfant gauche
enfant droit

1. Programmer une classe **Nœud** répondant au schéma ci-contre. Quels seront les types et valeurs par défaut de « enfant gauche » et « enfant droit » ?
2. Quels seront les attributs et méthodes de la classe arbre ? Donner son schéma UML.
3. Programmer la classe **Arbre Binaire**.
4. Programmer une méthode récursive donnant l'arbre sous forme de liste.
5. Programmer une méthode récursive donnant la hauteur l'arbre.
6. Programmer une méthode récursive donnant la taille l'arbre.

Exercice 2.8. [Parcours en profondeur] Ajouter des méthodes permettant d'effectuer les trois parcours en profondeur : infixe, suffixe et préfixe à la classe **Arbre**.

Exercice 2.9. [Parcours en largeur] Ajouter une méthode permettant d'effectuer le parcours en largeur à la classe **Arbre**.

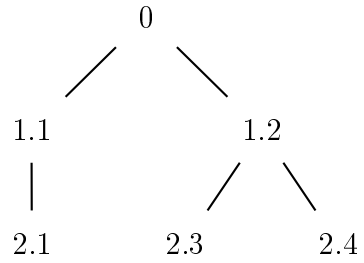
Exercice 2.10. Quel parcours permet d'obtenir les valeurs des nœuds d'un arbre binaire de recherche triées dans l'ordre croissant ?

Exercice 2.11. [Arbre binaire de recherche] Le but de cet exercice est de programmer une classe **Arbre Binaire de Recherche**.

1. Comment sera la classe **Arbre Binaire de Recherche** par rapport à la classe **Arbre Binaire** ?
2. Programmer une nouvelle méthode permettant d'insérer une valeur dans un arbre binaire de recherche selon l'algorithme du cours.
3. En s'inspirant de la recherche dichotomique, programmer une méthode déterminant si une valeur est présente ou non dans un arbre binaire de recherche.

2.5.3 Entraînement

Exercice 2.12. On considère l'arbre ci-contre.



1. Quel type d'arbre est-ce ?
2. Quelle est la profondeur du nœud 1.1 ?
3. Quelle est la hauteur de cet arbre ?
4. Quelle est la taille de cet arbre ?
5. Écrire cet arbre sous la forme d'une liste.

Exercice 2.13. Dessiner l'arbre correspondant à la liste suivante.

```
[0, [1.1, [], []], [1.2, [2.2, [3.1, [], []], [3.2, [], []]], [2.3, [3.3, [],
                                []], [3.4,[],[]]]]].
```

Exercice 2.14. Donner les parcours infixe, suffixe et préfixe de l'arbre de l'exercice 2.12.

Exercice 2.15. Donner le parcours en largeur de l'arbre de l'exercice 2.12 en détaillant à chaque étape l'état de la file et des éléments affichés.

2.6 Ressources supplémentaires

— Vidéo sur les arbres, généralités
— Vidéo sur les parcours en profondeur d'arbres
— Vidéo sur le parcours en largeur d'arbres
— Vidéo sur les arbres binaires de recherche