Chapitre n° 5 : structure de données hiérarchiques, les arbres

1. Les arbres, une structure hiérarchique

1.1. Présentation

Les listes, les piles et les files sont des structures de données linéaires. Des structures non linéaires peuvent être intéressantes dans la représentation des données lorsque des relations existent entre ces données. En particulier, lorsque les données sont hiérarchisées, il est possible d'utiliser des arbres : on parle alors de structure arborescente.

1.2. Exemples d'utilisation

On trouve des arbres dans des domaines divers.

- Arbres généalogiques.
- Arbres lexicographiques, qui présentent plusieurs mots, où les préfixes communs à plusieurs mots n'apparaissent qu'une seule fois.
- Organigramme d'une société avec directeur, sous-directeurs, secrétaires, etc.
- Contenu d'un livre avec parties, chapitres, sections, sous-sections, etc.
- Données présentées sous forme arborescente par l'explorateur d'un système d'exploitation.
- Etc.

2. Vocabulaire associé aux arbres

Définition : *un arbre est soit une structure vide, soit est composé d'un ou plusieurs nœuds.*

Définition : *une relation entre deux nœuds s'appelle arête (ou lien).*

Définition : *les fils sont l'ensemble des nœuds reliés à un même nœud par des arêtes entrantes.*

Définition : *le père (ou parent) est le nœud relié à chacun de ses nœuds fils par une arête sortante.*

Définition : *un sous-arbre est l'ensemble des nœuds et arêtes d'un nœud parent, de ses fils et de leurs éventuels descendants.*

Définition : *la racine est le seul nœud sans père.*

Définition : *une feuille est un nœud sans fils.*

Définition : *un nœud interne est un nœud qui n'est pas une feuille.*

Définition : *un chemin est une liste de nœuds reliés par des arêtes.*

Définition : *une branche est le chemin le plus court reliant un nœud à la racine.*

Définition : *la taille d'un arbre est le nombre de nœuds de l'arbre.*

Définition : *la profondeur d'un nœud est le nombre d'arêtes sur la branche qui le relie à la racine.*

Hormis la racine, qui a une profondeur de 0, chaque nœud de profondeur k est le fils d'un nœud de profondeur $k-1$.

Définition : *la hauteur d'un arbre est la profondeur maximale de l'ensemble des nœuds de l'arbre.*

Remarque : en informatique, les arbres sont traditionnellement présentés avec la racine en haut et les feuilles en bas.

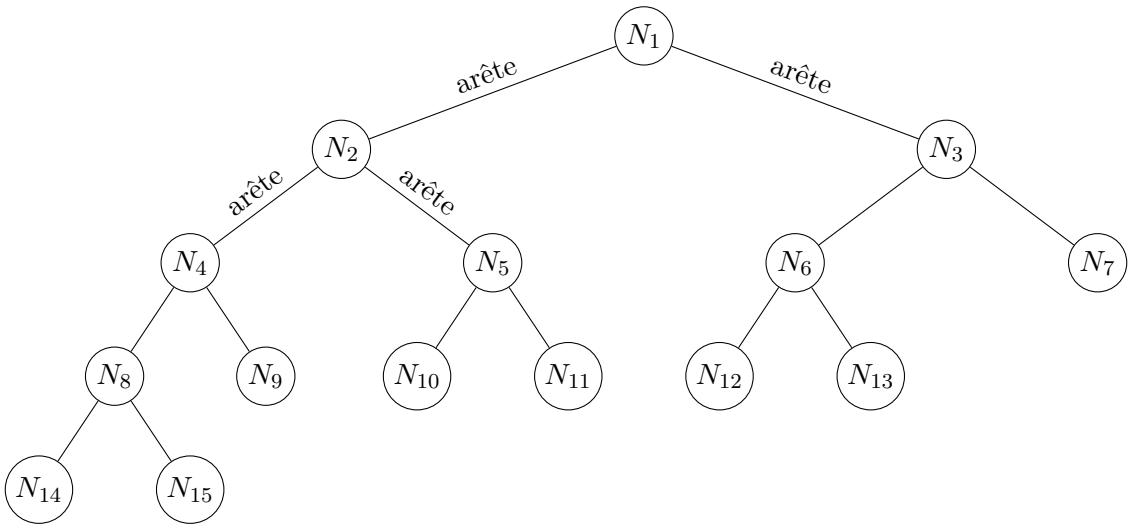


FIGURE 1 – Exemple d'arbre binaire

Exercice : les questions portent sur l'arbre ci-dessus.

1. Quel nœud est la racine de cette arbre ?
2. Quels nœuds sont des feuilles ?
3. Quels nœuds sont des nœuds intérieurs ?
4. Quels sont les fils du nœud N_5 ?
5. Quel est le père du nœud N_7 ?
6. Quel(s) nœud(s) constitue(nt) le sous-arbre gauche de N_3 ? Le sous-arbre droit ?
7. Par quels nœuds passe le chemin reliant N_{14} à N_{11} ?
8. Par quels nœuds passe la branche de N_{12} ?
9. Quelle est la taille de cet arbre ?
10. Quelle est la profondeur du nœud N_9 ?
11. Quelle est la hauteur de cet arbre ?

Solution :

1. La racine de cet arbre est le nœud N_1.
2. Les feuilles sont les nœuds N_7, N_9, N_{10}, N_{11}, N_{12}, N_{13}, N_{14} et N_{15}.
3. Les nœuds intérieurs sont les nœuds N_1, N_2, N_3, N_4, N_5, N_6 et N_8.
4. Les fils de N_5 sont N_{10} et N_{11}.
5. Le père de N_7 est N_3.
6. Le sous-arbre gauche de N_3 est constitué par N_6, N_{12} et N_{13}.
   Le sous-arbre droit est constitué par N_7.
7. Par quels nœuds passe le chemin reliant N_{14} à N_{11} ?
8. Par quels nœuds passe la branche de N_{12} ?
9. La taille de cet arbre est 15.
10. La profondeur de N_9 est 3.
11. La hauteur de cet arbre est 4.

3. Arbres binaires

3.1. Définitions

Dans le cas général, un nœud peut avoir un nombre quelconque de fils.

Définition : *un arbre binaire est un arbre pour lequel chaque père a au plus deux fils.*

Remarque : tout arbre peut être ramené à un arbre binaire.

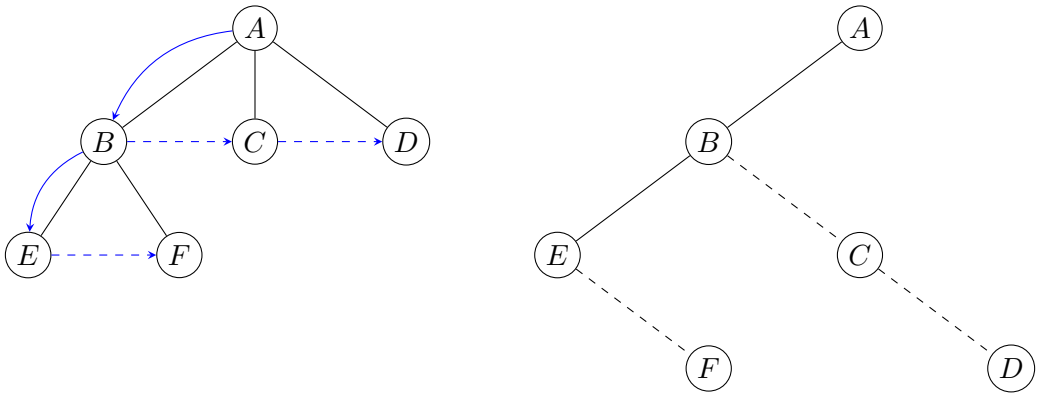


FIGURE 2 – Exemple d'arbre quelconque transformé en arbre binaire

Propriété : *un arbre binaire peut être défini de façon récursive.*

- *Soit c'est un arbre vide, que nous noterons* Δ.
- *Soit c'est un triplet* (e, g, d)*, appelé nœud, avec :*
  - e *son étiquette,*
  - g *son sous-arbre binaire gauche,*
  - d *son sous-arbre binaire droit.*

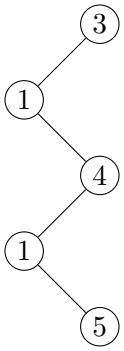
La première partie de la définition assure l'arrêt et donc la cohérence de la définition récursive.

Exercice : dessiner chacun des arbres ci-dessous. Donner également la taille, la hauteur et le nombre de feuilles de chaque arbre dessiné.

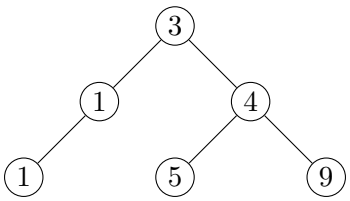
1. $(1, \Delta, \Delta)$
2. $(3, (1, \Delta, (4, (1, \Delta, (5, \Delta, \Delta)), \Delta)), \Delta)$
3. $(3, (1, (1, \Delta, \Delta), \Delta), (4, (5, \Delta, \Delta), (9, \Delta, \Delta)))$
4. $(3, (1, (1, \Delta, \Delta), (5, \Delta, \Delta)), (4, (9, \Delta, \Delta), (2, \Delta, \Delta)))$

1er arbre : taille de 1, hauteur de 0 et 1 feuille.

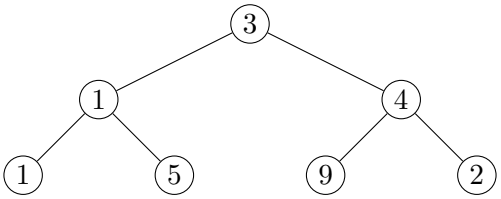
2e arbre : taille de 5, hauteur de 4 et 1 feuille.



3e arbre : taille de 6, hauteur de 2 et 3 feuilles.



4e arbre : taille de 7, hauteur de 2 et 4 feuilles.



3.2. Notions de hauteur et de profondeur

Exercice : soit un arbre binaire de hauteur h.

1. Combien de feuilles et de nœuds comporte-t-il au minimum ?
2. Combien de feuilles et de nœuds comporte-t-il au maximum ?

Solution :

1. À chaque profondeur, il n'y a qu'un seul nœud, chaque père n'a qu'un seul fils. *Il y a donc au minimum* $h+1$ *nœuds et une seule feuille.*
2. Chaque père possède deux fils. À chaque profondeur p, il y a 2^p nœuds. *Le nombre maximal de nœuds est donc* $\sum_{p=0}^{h} 2^p = 2^{h+1} - 1$ *et le nombre maximal de feuilles est* 2^h.

Un arbre de hauteur h *comporte au minimum* $h+1$ *nœuds dont une seule feuille (dans ce cas, on parle d'arbre filiforme) et au maximum* $2^{h+1}-1$ *nœuds dont* 2^h *feuilles (dans ce cas, on parle d'arbre complet).*

3.3. Implémentation des arbres binaires

Comme pour les autres TAD, pour implémenter un arbre binaire nous aurons besoin de constructeurs, de sélecteurs et d'un prédicat.

Constructeurs :

- une fonction qui crée un arbre binaire vide ;
- une fonction *nœud* qui crée un arbre binaire (e, g, d) à partir d'une étiquette e et de deux arbres binaires g et d.

Sélecteurs :

- une fonction qui renvoie l'étiquette e à partir de l'arbre binaire (e, g, d) ;
- une fonction qui renvoie l'arbre binaire gauche g à partir de l'arbre binaire (e, g, d) ;
- une fonction qui renvoie l'arbre binaire droit d à partir de l'arbre binaire (e, g, d).

Ces sélecteurs ne pouvant pas s'appliquer sur un arbre binaire vide, il faut également le prédicat suivant.

Prédicat :

- une fonction qui renvoie le booléen indiquant si l'arbre binaire est vide.

3.4. Arbres binaires complets

Définition : *un arbre binaire complet est un arbre pour lequel chaque père, hormis les feuilles, a exactement deux fils et pour lequel les branches issues de la racine ont exactement même longueur.*

Remarque : il est possible d'implémenter un arbre binaire complet avec une simple liste. Dans ce cas, les deux fils du nœud d'indice i se trouvent aux indices $2i + 1$ et $2i + 2$.

Exemple : implémentation d'un arbre complet avec une liste simple.

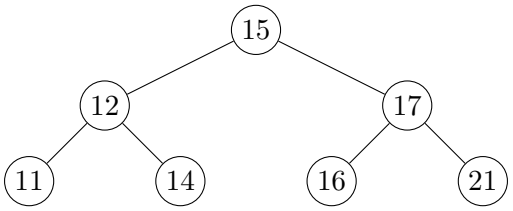


FIGURE 3 – Exemple d'arbre complet

Implémentation de l'arbre.

```
1 arbre = [15, 12, 17, 11, 14, 16, 21]
```

4. Algorithmes sur les arbres binaires

4.1. Présentation

Ces algorithmes classiques sont à connaître et sont exigibles à l'examen.

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

4.2.1. Taille d'un arbre binaire

Il s'agit de déterminer le nombre total de nœuds de l'arbre.

Algorithme donnant la taille d'un arbre :

- *si l'arbre est vide, il y a* 0 *nœud ;*
- *sinon, nous comptons le nœud racine, plus le nombre de nœuds du sous-arbre gauche, plus le nombre de nœuds du sous-arbre droit.*

Le TP n° 8 présente des exemples d'implémentations de cet algorithme.

4.2.2. Hauteur d'un arbre binaire

Il s'agit de déterminer le nombre maximal de nœuds se trouvant entre la racine et une feuille.

Algorithme donnant la hauteur d'un arbre :

- *si l'arbre est vide, il contient* 0 *nœud ;*
- *sinon, nous comptons le nœud racine, plus le maximum entre la hauteur du sous-arbre gauche et la hauteur du sous-arbre droit.*

Le TP n° 8 présente des exemples d'implémentations de cet algorithme.

4.3. Parcours d'un arbre, ordre en profondeur d'abord

4.3.1. Présentation

On parcourt un arbre pour calculer sa taille ou sa hauteur, pour chercher une valeur particulière ou pour afficher les différentes valeurs, etc. Il existe différentes façons d'effectuer ce parcours.

Exemple : pour illustrer ces différents parcours, nous utiliserons l'exemple de l'arbre binaire complet suivant.

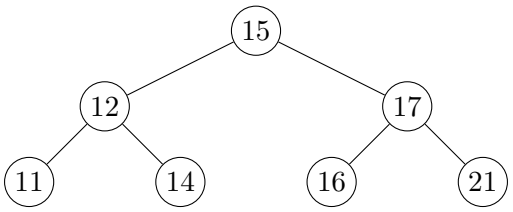


FIGURE 4 – Exemple d'arbre complet

4.3.2. Parcours d'un arbre, ordre en profondeur d'abord

Définition : *les parcours en profondeur d'abord (*DFS *pour* Depth-First Search*) consistent à explorer chaque branche complètement avant d'explorer la branche voisine.*

Remarque : la programmation récursive est indiquée.

Algorithme de parcours d'un arbre, ordre en profondeur d'abord :

- *si l'arbre est non vide, on parcourt de manière récursive son sous-arbre gauche puis son sous-arbre droit ;*
- *sinon, c'est terminé.*

Définition : *on distingue trois cas suivant le moment où est traité une racine d'un sous-arbre. Traiter une racine signifie par exemple afficher la valeur.*

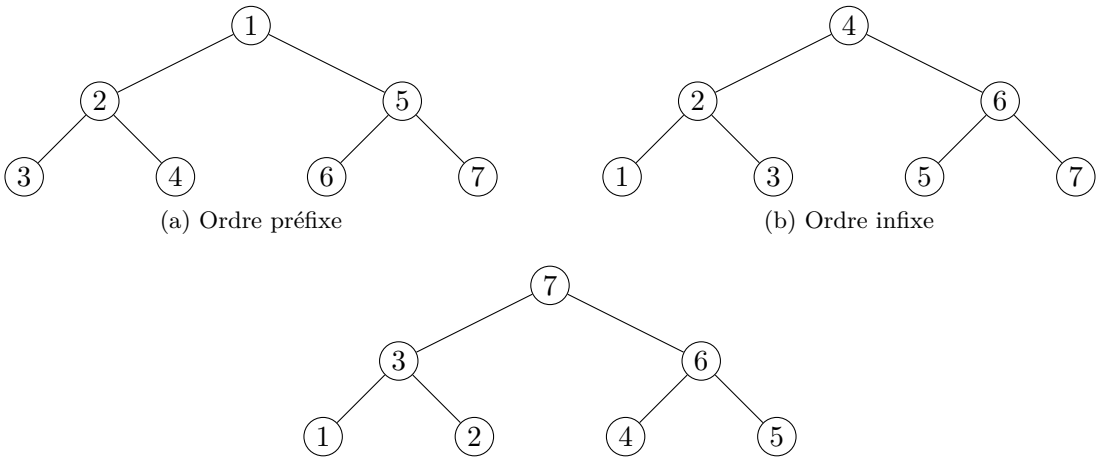
- *Si la racine est traitée avant ses deux sous-arbres, il s'agit d'un ordre préfixe.*
- *Si la racine est traitée entre ses deux sous-arbres, il s'agit d'un ordre infixe.*
- *Si la racine est traitée après ses deux sous-arbres, il s'agit d'un ordre postfixe.*

Exercice : dans le cas de l'arbre présenté en exemple à la figure 4, donner la liste des valeurs des nœuds dans le cas d'un parcours en profondeur d'abord.

1. Dans le cas d'un ordre préfixe.
2. Dans le cas d'un ordre infixe.
3. Dans le cas d'un ordre postfixe.

Solution :

1. $15, 12, 11, 14, 17, 16, 21$.
2. $11, 12, 14, 15, 16, 17, 21$.
3. $11, 14, 12, 16, 21, 17, 15$.



(a) Ordre préfixe

(b) Ordre infixe

(c) Ordre postfixe

FIGURE 5 – Parcours d'un arbre, ordre en profondeur d'abord

Le TP n° 9 présente des exemples d'implémentations de cet algorithme.

4.3.3. Parcours d'un arbre, ordre en largeur d'abord

Définition : *le parcours en largeur d'abord (*BFS *pour* Breath-First Search*) consiste à explorer l'arbre niveau par niveau, en considérant tous les sommets de chaque niveau.*

Remarque : on commence donc par la racine, puis les deux racines de chacun des deux sous-arbres, puis les quatre racines de chacun des quatre sous-arbres et ainsi de suite.

Exercice : dans le cas de l'arbre présenté en exemple à la figure 4, donner la liste des valeurs des nœuds dans le cas d'un parcours en largeur d'abord.

Solution : $15, 12, 17, 11, 14, 16, 21$.

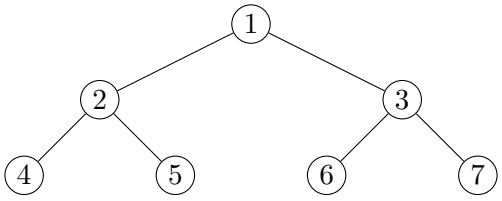


FIGURE 6 – Parcours d'un arbre, ordre en largeur d'abord

Algorithme de parcours d'un arbre, ordre en largeur d'abord.
La méthode la plus pratique consiste à utiliser une structure de file.

- *On suppose l'arbre non vide.*
- *On place l'arbre dans la file.*
- *Tant que la file n'est pas vide :*
  - *on défile un élément, qui est un arbre ;*
  - *on affiche la valeur de sa racine ;*
  - *on place dans la file chacun de ses sous-arbres, s'ils ne sont pas vides.*

Remarque : il s'agit d'un algorithme itératif.

Le TP n° 9 présente des exemples d'implémentations de cet algorithme.

5. Algorithmes sur les arbres binaires de recherche

5.1. Définition

Définition : *un arbre binaire de recherche (ANR) est un arbre binaire pour lequel les enfants à gauche d'un nœud on des valeurs inférieures à celle du nœud et les enfants à droite de nœud on des valeurs supérieures à celle du nœud.*

Exercice : est ce que l'arbre présenté en exemple à la figure 4 est un arbre binaire de recherche.

Solution : oui, il respecte la définition.

5.2. Insertion d'une clef dans un ABR

On peut reprendre l'implémentation en POO d'un arbre binaire, avec l'utilisation d'une classe *Arbre*.

```
class Arbre:
    def __init__(self, val):
        self.valeur = val
        self.gauche = None
        self.droit = None

    def ajout_gauche(self, val):
        self.gauche = Arbre(val)

    def ajout_droit(self, val):
        self.droit = Arbre(val)
```

Les méthodes *ajout_gauche* et *ajout_droit* ajoutent un enfant au nœud, sans se soucier de la valeur. Pour construire un ABR, ces deux méthodes vont être remplacées par une méthode *ajoute* qui permettra d'ajouter une valeur en respectant la propriété d'un arbre binaire de recherche.

```
class ABR:
    def __init__(self, val):
        self.valeur = val
        self.gauche = None
        self.droit = None

    def ajoute(self, val):
        if val < self.valeur:
            if self.gauche is None:
                self.gauche = ABR(val)
            else:
                self.gauche.ajoute(val)
        elif val > self.valeur:
            if self.droit is None:
                self.droit = ABR(val)
            else:
                self.droit.ajoute(val)
```

Remarque : on suppose que les valeurs à ajouter à l'arbre de recherche sont toutes différentes.

Définition : *on parle de valeur ou de clef et donc d'insertion d'une valeur ou d'une clef dans un ABR.*

Remarque : *cet algorithme, classique, est à connaître et est exigible à l'examen.*

5.3. Recherche d'une clef dans un ABR

Pour la recherche d'une clef, on écrit une méthode *recherche* dans la classe *ABR* en suivant le même principe que pour l'insertion.

```
    def recherche(self, valeur):
        if val < self.valeur:
            if self.gauche is None:
                return False
            else:
                return self.gauche.recherche(val)
        elif val > self.valeur:
            if self.droit is None:
                return False
            else:
                return self.droit.recherche(val)
        return True
```

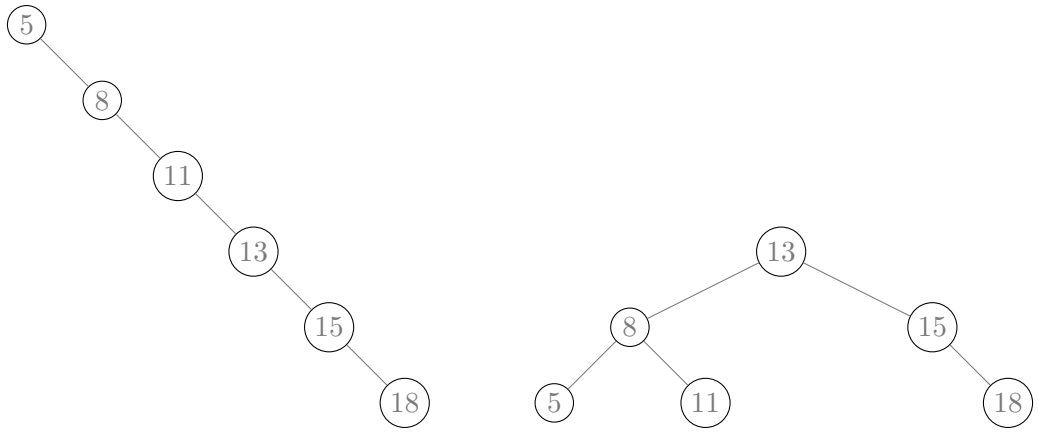
Remarque : le principe utilisé est celui de la dichotomie. Suivant la valeur du nœud où on se trouve, on descend vers la gauche si cette valeur est plus grande que la valeur cherchée et vers la droite sinon. La recherche est terminée si la valeur est trouvée ou si on atteint une branche qui n'a pas de fils gauche ou de fils droit, suivant celui que l'on souhaiterait.

5.4. Coût d'une recherche ou d'une insertion dans un ABR

Exercice : dessiner les arbres *abr1* et *abr2* obtenus en ajoutant les valeurs dans l'ordre suivant.

1. *abr1 :* 5, 8, 11, 13, 15, 18.
2. *abr1 :* 13, 8, 5, 11, 15, 18.
3. Pour quel arbre la recherche d'une clef sera la plus rapide ?

Solution :



(a) abr1 (b) abr2

Dans *abr1*, au pire cas, la recherche va devoir parcourir les 6 niveaux de l'arbre. Dans *abr2*, au pire cas, la recherche va devoir parcourir les 3 niveaux de l'arbre : elle y sera donc plus rapide.

Remarque : il existe des méthodes pour équilibrer l'arbre, si nécessaire, ce qui permet de diminuer significativement le coût d'une recherche.

Propriété : *si l'arbre de recherche est équilibré et possède* n *noeuds, alors la recherche d'une clef a un coût de l'ordre de la hauteur* h *de l'arbre, c'est à dire du logarithme binaire de* n.

C'est le nombre de chiffres utilisés dans l'écriture binaire de n.

Une insertion a la même coût.

Capacités exigibles

- Identifier des situations nécessitant une structure de données arborescente.
- Évaluer quelques mesures des arbres binaires (taille, encadrement de la hauteur, etc.).
- Calculer la taille et la hauteur d'un arbre.
- Parcourir un arbre de différentes façons (ordres infixe, préfixe ou suffixe ; ordre en largeur d'abord).
- Rechercher une clef dans un arbre de recherche, insérer une clef.