Matthieu GRALL

Cours d’algorithmique

Partie 2

Attention : ce document est un complément aux séances de cours et de travaux pratiques.

Table des matières

1 Recherches

1.1 Problématique

Dans les systèmes informatiques, les données sont représentées de différentes manières : tableau à une dimension, tableau à N dimensions, arborescences, listes chaînées, …

Une fois représentées, on cherche à manipuler ces données : ajout et suppression de données, consultation et recherche des données.

On s'interroge sur :
- la présence ou l'absence d'une information → on communique alors la réussite de l'opération ou la position relative de la valeur dans l'ensemble
- la satisfaction de critères → on communique alors la donnée optimisant un critère, ou toutes les données satisfaisant un critère

On a deux cas de figure :
- l'ensemble des valeurs est dans le désordre
  ex. : 46 -1 9 52 -14 -8 4 45
- l'ensemble des valeurs respecte un ordre
  ex. : -14 -8 -1 4 9 45 46 52

Les recherches que nous allons étudier utiliserons des données représentées sous la forme de tableaux de nombres entiers (une **classe T1D** ayant pour attributs la **Table** des valeurs et le **NombreEléments** de la Table).

1.2 La recherche séquentielle

1.2.1 Recherche de la présence d'une valeur

- Algorithme de la méthode de la classe T1D :

MFonction RechercheSéquentielle (val) retourne (booléen)
{retourne vrai si val appartient au tableau Cible, faux sinon}
Paramètres :
(D) Cible : T1D
(D) val : entier
Variables :
Trouvé : booléen
Ind : entier
Début
Ind ← 0
Trouvé ← faux
Tant Que ((Ind < NombreEléments) et (non Trouvé))
Ind ← Ind + 1
Trouvé ← (Table[Ind] = val)
Fin Tant Que
Retourne (Trouvé)
Fin

- Algorithme de cette recherche si elle n'est pas conçue comme une méthode de la classe T1D :

Fonction RechercheSéquentielle (Ens, val) retourne (booléen)
{retourne vrai si val appartient au tableau Ens, faux sinon}
Paramètres :
(D) Ens : T1D
(D) val : entier
Variables :
Trouvé : booléen
Ind : entier
Début
Ind ← 0
Trouvé ← faux
Tant Que ((Ind < Ens.RetourneNombreEléments) et (non Trouvé)) *{RetourneNombreEléments est Fonction permettant d'accéder à l'attribut NombreEléments}*
Ind ← Ind + 1
Trouvé ← (Ens[Ind] = val)
Fin Tant Que
Retourne (Trouvé)
Fin

1.2.2 Recherche de la position d'une valeur

- Algorithme de la méthode de la classe T1D :

MFonction RechercheSéquentielle (val) retourne (entier)
{retourne la position de val dans Table si val appartient au tableau Cible, -1 sinon}
Paramètres :
(D) Cible : T1D
(D) val : entier
Variables :
Trouvé : booléen
Ind : entier
Début
Ind ← 0
Trouvé ← faux
Tant Que ((Ind < NombreEléments) et (non Trouvé))
Ind ← Ind + 1
Trouvé ← (Table[Ind] = val)
Fin Tant Que
Si (Trouvé) alors
Retourne (Ind)
Sinon
Retourne (-1)
Fin Si
Fin

- Algorithme de cette recherche si elle n'est pas conçue comme une méthode de la classe T1D :

Fonction RechercheSéquentielle (Ens, val) retourne (entier)
{retourne la position de val dans Table si val appartient au tableau Cible, -1 sinon}
Paramètres :
(D) Ens : T1D
(D) val : entier
Variables :
Trouvé : booléen
Ind : entier
Début
Ind ← 0
Trouvé ← faux
Tant Que ((Ind < Ens.RetourneNombreEléments) et (non Trouvé))
Ind ← Ind + 1
Trouvé ← (Ens[Ind] = val)
Fin Tant Que
Si (Trouvé) alors
Retourne (Ind)
Sinon
Retourne (-1)
Fin Si
Fin

1.2.3 Recherche de la position d'une valeur dans un tableau ordonné

MFonction RechercheSéquentielle (val) retourne (entier)
{retourne la position de val dans Table si val appartient au tableau Cible, -1 sinon}
Paramètres :

(D)	*Cible*	*:*	*T1D*
(D)	*val*	*:*	*entier*

Variables :

Trouvé	*:*	*booléen*
valDépassée	*:*	*booléen*
Ind	*:*	*entier*

```
Début
    Ind ← 0
    Trouvé ← faux
    valDépassée ← faux
    Tant Que ((Ind < NombreEléments) et (non (Trouvé ou valDépassée)))
        Ind ← Ind + 1
        Trouvé ← (Table[Ind] = val)
        valDépassée ← (Table[Ind] > val)
    Fin Tant Que
    Si (Trouvé) alors
        Retourne (Ind)
    Sinon
        Retourne (-1)
    Fin Si
Fin
```

1.3 Les recherches séquentielles avec critères

1.3.1 Recherche de valeurs plus petites qu'une valeur passée en paramètre

MProcédure RechPlusPetits (val, TabInfVal)
{TabInfVal contiendra la suite des éléments de Table inférieurs à val}
Paramètres :

(D)	*Cible*	*:*	*T1D*
(R)	*TabInfVal*	*:*	*T1D*
(D)	*val*	*:*	*entier*

Variables :

IndD	*:*	*entier*
IndR	*:*	*entier*

```
Début
    IndD ← 0
    IndR ← 0
    Tant Que (IndD < NombreEléments)
        IndD ← IndD + 1
        Si (Table[IndD] ≤ val) alors
            IndR ← IndR + 1
            TabInfVal[IndR] ← Table[IndD]
        Fin Si
    Fin Tant Que
    TabInfVal.MemNombreEléments(IndR)   {MemNombreEléments est une méthode mettant à jour
                                         La valeur de l'attribut NombreEléments}
Fin
```

1.3.2 Recherche de la plus petite valeur

```
MFonction RechercheMin retourne (entier)
{retourne le plus petit élément de Table}
Paramètres :
        (D)     Cible           :       T1D
Variables :
                Ind             :       entier
                Min             :       entier
Début
        Ind ← 0
        Min ← Table[1]
        Tant Que (Ind < NombreEléments)
                Ind ← Ind + 1
                Si (Table[Ind] < Min) alors
                        Min ← Table[Ind]
                Fin Si
        Retourne (Min)
Fin
```

1.3.3 Recherche de l'indice de la plus petite valeur

```
MFonction RechercheMin retourne (entier)
{retourne la position du plus petit élément de Table}
Paramètres :
        (D)     Cible           :       T1D
Variables :
                Ind             :       entier
                Min             :       entier
                IndMin          :       entier
Début
        Ind ← 1
        IndMin ← 1
        Min ← Table[1]
        Tant Que (Ind < NombreEléments)
                Ind ← Ind + 1
                Si (Table[Ind] < Min) alors
                        IndMin ← Ind
                        Min ← Table[Ind]
                Fin Si
        Retourne (IndMin)
Fin
```

1.4 La recherche dichotomique dans un tableau ordonné

Le principe est de viser l'élément du milieu dans un ensemble de valeurs triées. Soit la valeur cherchée est plus petite, on reprend alors la recherche à gauche de la valeur du milieu. Soit la valeur cherchée est plus grande, on reprend alors la recherche à droite de la valeur du milieu. Et ainsi de suite…

```
MFonction RechercheDichotomique (val) retourne (entier)
{retourne la position de val dans Table si val appartient au tableau Cible, -1 sinon}
Paramètres :
        (D)     Cible           :       T1D
        (D)     val             :       entier
Variables :
                IndDébu         :       entier
                IndFin          :       entier
                IndMilieu       :       entier
Début
        IndDébut ← 0
        IndFin ← NombreEléments + 1
        Min ← Table[1]

        {parcours dichotomique}
        Tant Que ((IndFin – IndDébut) > 1)
                IndMilieu ← ((IndFin + IndDébut) / 2)
                Si (Table[IndMilieu] ≤ val) alors
                        {val peut être dansla moitié droite}
                        IndDébut ← IndMilieu
                Sinon
                        {val peut être dansla moitié gauche}
                        IndFin ← IndMilieu
                Fin Si

        {interprétation de la sortie de boucle}
        Si (IndDébut == 0) alors
                {val n'est pas dans le tableau car plus petite que toute les valeurs du tableau}
                Retourne (-1)
        Sinon
                Si (Table[IndDébut] = val) alors
                        {val est dans le tableau}
                        Retourne (IndDébut)
                Sinon
                        {val n'est pas dans le tableau}
                        Retourne (-1)
                Fin Si
        Fin Si
Fin
```

2 Complexité

2.1 Problématique

Combien de temps et de place-mémoire sont nécessaires pour résoudre tel ou tel problème avec tel ordinateur au moyen de tel algorithme ?
Est-il possible d'optimiser le coût en temps et en place-mémoire simultanément ?

Etude à mener :
- Evaluer le plus précisément possible le nombre d'opérations élémentaires (complexité temporelle)
- Evaluer la place-mémoire (complexité spatiale)

2.2 Complexité asymptotique

- On doit étudier la complexité pour de grosses quantités de données.

Exemple :
2 algorithmes pour une même tâche :
- l'algorithme 1 effectue n^2 opérations
- l'algorithme 2 effectue $n.\log_2 n$ opérations

2 machines :
- la machine 1 effectue 2^{10} ($\sim 10^3$) opérations par seconde
- la machine 2 effectue 2^{20} ($\sim 10^6$) opérations par seconde

Le temps de calcul est donc le suivant :

Nombre d'opérations	Machine 1		Machine 2	
	Algorithme 1	Algorithme 2	Algorithme 1	Algorithme 2
$n = 2^{10}$	2^{10} s	10 s	1 s	$<10^{-2}$ s
$n = 2^{20}$	2^{30} s	20.2^{10} s	2^{20} s	20 s

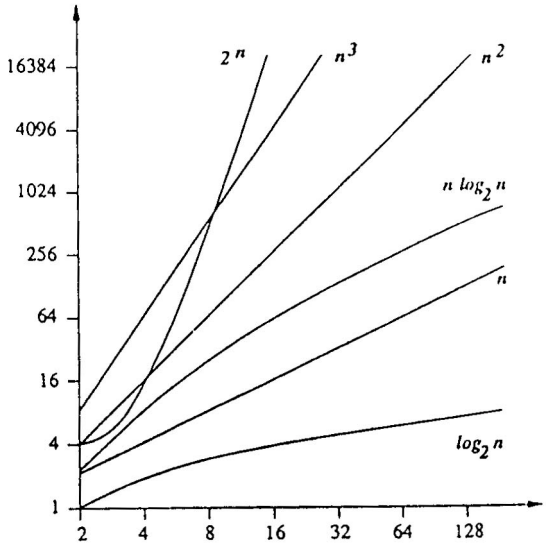
- Notation O

f = O(g) signifie que f est dominée asymptotiquement par g
ex. : 2n = O(n), et 2n = $O(n^2)$

- Notation θ

f = θ(g) signifie que f et g sont de même ordre de grandeur asymptotique
ex. : 2n = θ (n), mais 2n n'est pas un $\theta(n^2)$

- Rapidité de croissance comparées de certaines fonctions usuelles :



La croissance peut être
- polynomiale : f = O(P), f se comporte comme P → à rechercher
- exponentielle : f = O(c^n) → à éviter

2.3 Calcul de la complexité de quelques algorithmes

2.3.1 Calcul de la valeur d'un polynôme en un point

- Première solution

```
…
p ← a[0]
pour i ← 1 à n faire
        xpi ← puissance(x,i)
        p ← p + a[i] * xpi
```

On a dans ce cas n(n+1)/2 multiplications et n additions, l'algorithme est donc en O(n^2).

- Deuxième solution (mieux)

```
…
p ← a[0]
xpi ← 1
pour i ← 1 à n faire
        xpi ← xpi * x
        p ← p + a[i] * xpi
```

On a dans ce cas 2n multiplications et n additions, l'algorithme est donc en O(n).

- Troisième solution (encore mieux)

```
…
p ← a[n]
pour i ← n à 1 par pas de –1 faire
        p ← p * x + a[i-1]
```

On a dans ce cas n multiplications et n additions, l'algorithme est donc en O(n).

2.3.2 Calcul de la complexité des algorithmes de recherche simples

Les opérations élémentaires retenues sont les comparaisons.

- Recherche séquentielle dans un tableau non trié

Complexité au pire : n comparaisons
Complexité moyenne : (n+n/2)/2 = 3n/4 comparaisons
→ Algorithme en O(n)

- Recherche séquentielle dans un tableau trié

Complexité au pire : n comparaisons
Complexité moyenne : (n/2+n/2)/2 = n/2 comparaisons
→ Algorithme en O(n)

- Recherche dichotomique

Complexité au pire = Complexité moyenne = nombre d'intervalles considérés
→ Algorithme en $O(\log_2 n)$

3 Les représentations arborescentes

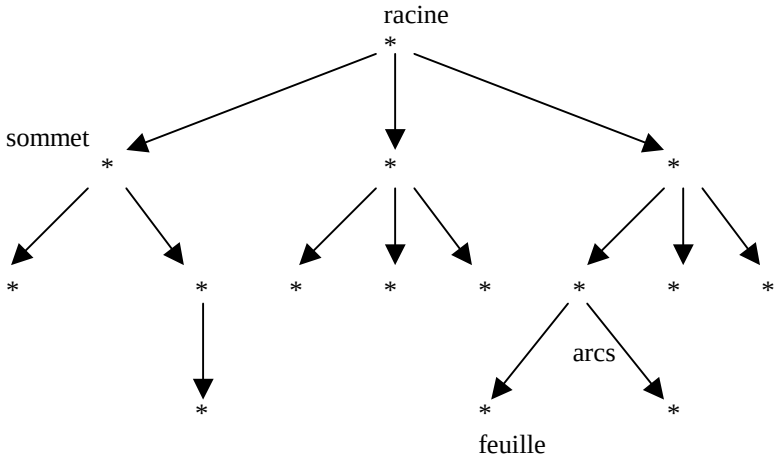
3.1 Définitions

Un **graphe** G=(X, U) est un ensemble X de sommets reliés entre eux par un ensemble U d'arcs non orientées entre ces sommets.

Un **graphe orienté** est un graphe dont les arcs sont orientés.

Un **arbre** est un graphe non orienté connexe sans cycle (connexe signifiant qu'il existe toujours une chaîne entre deux sommets).

Une **arborescence** est un graphe orienté connexe sans cycle. Elle se caractérise par :
- sa **racine**
- ses **sommets**
- ses **arcs** reliant les sommets entre eux
- ses **feuilles** (sommets qui ne sont origines d'aucun arc)



Un sommet qui est origine d'arcs est dit **père** des sommets extrémités de ces arcs (sommets **fils**).

Une **sous-arborescence** de racine est l'arborescence formée par le sommet a et toute sa descendance.

Le **niveau** ou la **profondeur** d'un sommet a est le nombre d'arcs qu'il faut parcourir pour atteindre a en venant de la racine.

La **hauteur** d'une arborescence est la profondeur maximale de ses feuilles.

Une **arborescence binaire** (appelée aussi abusivement **arbre binaire**) est une arborescence dans laquelle chaque sommet ne peut avoir plus de 2 sommets fils.

Un **arbre binaire parfait** est un arbre binaire dont tous les niveaux comportent le nombre maximal de sommets (soit 2^h sommets au niveau h) sauf peut-être le niveau le plus profond.

Un **tas** est un arbre binaire parfait étiqueté tel que la valeur portée en chaque sommet interne est au moins égale à celle portée par chacun de ses fils.

Un **tas partiel** est un arbre binaire parfait tel que la valeur portée par chaque sommet interne, différent de la racine, est au moins égale à celle de chacun de ses fils.

Un **arbre binaire de recherche** (ABR) est un arbre binaire dans lequel toutes les valeurs étiquetant les sommets du sous-arbre gauche de la racine sont inférieurs ou égales à la valeur étiquetant la racine, tandis que toutes les valeurs étiquetant les sommets du sous-arbre droit de la racine sont supérieures à la valeur étiquetant la racine !

3.2 La classe ArbreBinaire

3.2.1 Attributs

Etiquette : Info
Père : ArbreBinaire
FilsGauche : ArbreBinaire
FilsDroit : ArbreBinaire

3.2.2 Méthodes de consultation et de mise à jour de l'information portée

MFonction GetEtiquette retourne (Info)

```
Paramètres (D) Cible : ArbreBinaire
{retourne la valeur qui étiquette la Cible}
Début
    Retourne (Etiquette)                {on pourrait écrire « Retourne (Cible.Etiquette) »}
Fin
```

MProcédure SetEtiquette (ValRacine)

```
Paramètres (D/R) Cible     : ArbreBinaire
           (D)   ValRacine : Info
{Etiquette la Cible avec la valeur ValRacine}
Début
    Etiquette ← ValRacine
Fin
```

3.2.3 Méthodes d'accès à la structure de la représentation

MFonction GetPére retourne (ArbreBinaire)

```
Paramètres (D) Cible : ArbreBinaire
{retourne l'arbre dont la Cible est un sous-arbre}
Début
    Retourne (Père)
Fin
```

MFonction SetPére (ValPère)

```
Paramètres (D/R) Cible   : ArbreBinaire
           (D)   ValPère : ArbreBinaire
{change la valeur de Père}
Début
    Père ← ValPère
Fin
```

MFonction GetFilsGauche retourne (ArbreBinaire)

```
Paramètres (D) Cible : ArbreBinaire
{retourne le sous-arbre gauche de la Cible}
Début
    Retourne (FilsGauche)
Fin
```

MFonction SetFilsGauche (ValFilsGauche)

```
Paramètres (D/R) Cible        : ArbreBinaire
           (D)   ValFilsGauche : ArbreBinaire
{change la valeur de FilsGauche}
Début
    FilsGauche ← ValFilsGauche
Fin
```

MFonction GetFilsDroit retourne (ArbreBinaire)

```
Paramètres      (D)     Cible   :       ArbreBinaire
{ retourne le sous-arbre droit de la Cible }
Début
        Retourne (FilsDroit)
Fin
```

MFonction SetFilsDroit (ValFilsDroit)

```
Paramètres      (D/R)   Cible           :       ArbreBinaire
                (D)     ValFilsDroit    :       ArbreBinaire
{change la valeur de FilsDroit}
Début
        FilsDroit ← ValFilsDroit
Fin
```

MFonction estFeuille retourne (booléen)

```
Paramètres      (D)     Cible   :       ArbreBinaire
{retourne vrai si la Cible n'a pas de fils, faux sinon}
Début
        Retourne ((FilsGauche = Nil) ET (FilsDroit = Nil))
Fin
```

MFonction estRacine retourne (booléen)

```
Paramètres      (D)     Cible   :       ArbreBinaire
{retourne vrai si la Cible n'a pas de père, faux sinon}
Début
        Retourne (Père = Nil)
Fin
```

MFonction estVide retourne (booléen)

```
Paramètres      (D)     Cible   :       ArbreBinaire
{retourne vrai si la Cible ne contient pas de sommet, faux sinon}
Début
        Retourne ((FilsGauche = Nil) ET (FilsDroit = Nil) ET (Père = Nil) ET (Etiquette = Nil))
Fin
```

3.2.4 Méthodes de construction de la représentation arborescente

MFonction Création (ValRacine) retourne (entier)

```
Paramètres      (D/R)   Cible           :       ArbreBinaire
                (D)     ValRacine       :       Info
{retourne       0 si la Cible est vide (ValRacine est alors affectée à la racine),
                -1 si intervient un problème d'allocation,
                1 si la Cible est un arbre non vide}
Début
        Si (estVide) alors
                Retourne (1)                    {« Retourne » provoque la sortie de la fonction}
        Fin Si

        Cible ← Nouveau (ArbreBinaire)
        Si (Cible = Nil) alors
                Retourne (-1)
        Fin Si

        SetPère (Nil)
        SetFilsGauche (Nil)
        SetFilsDroit (Nil)
        SetEtiquette (ValRacine)
        Retourne (0)
Fin
```

MProcédure AjoutFilsGauche (CodePb, SAG)

```
Paramètres      (D/R)   Cible       :   ArbreBinaire
                (R)     CodePb      :   entier
                (R)     SAG         :   ArbreBinaire
{SAG sera le sous-arbre gauche ajouté, CodePb vaudra    0 si tout se passe bien,
                                                        1 si l'allocation est impossible,
                                                        -1 si la Cible a déjà un fils gauche}
Début
    Si (GetFilsGauche != Nil) alors
        CodePb ← -1
    Sinon
        Si (SAG.Création (Nil) = -1) alors
            CodePb ← 1
        Sinon
            CodePb ← 0
            SAG.SetPère (Cible)
            SetFilsGauche (SAG)
        Fin Si
    Fin Si
Fin
```

MProcédure AjoutFilsDroit (CodePb, SAD)

```
Paramètres      (D/R)   Cible       :   ArbreBinaire
                (R)     CodePb      :   entier
                (R)     SAD         :   ArbreBinaire
{SAD sera le sous-arbre droit ajouté, CodePb vaudra     0 si tout se passe bien,
                                                        1 si l'allocation est impossible,
                                                        -1 si la Cible a déjà un fils droit}
Début
    Si (GetFilsDroit != Nil) alors
        CodePb ← -1
    Sinon
        Si (SAD.Création (Nil) = -1) alors
            CodePb ← 1
        Sinon
            CodePb ← 0
            SAD.SetPère (Cible)
            SetFilsDroit (SAD)
        Fin Si
    Fin Si
Fin
```

3.2.5 Algorithmes de parcours d'un arbre binaire

Les méthodes suivantes sont de bons exemples de traitements récursifs, essayez de les :

MFonction CompteSommets retourne (entier)

```
Paramètres      (D)     Cible       :   ArbreBinaire
{retourne le nombre de sommets de la cible}
Début
    Si (estVide) alors
        Retourne (0)
    Fin Si
    Retourne (1 + FilsGauche.CompteSommets + FilsDroit.CompteSommets)
Fin
```

MProcédure AffichePréfixe

```
{affiche l'étiquette de la cible, puis les étiquettes du sous-arbre gauche, puis celles du sous-arbre droit}
Paramètres    (D)    Cible    :    ArbreBinaire
Variables            FG       :    ArbreBinaire
                     FD       :    ArbreBinaire
Début
    Si (non estVide) alors
        {affichage de la valeur portée par la racine}
        Afficher (Etiquette)

        {affichage de toutes les valeurs du sous-arbre gauche}
        FG ← FilsGauche
        FG.AffichePréfixe

        {affichage de toutes les valeurs du sous-arbre droit}
        FD ← FilsDroit
        FD.AffichePréfixe
    Fin Si
Fin
```

MProcédure AfficheSuffixe

```
{affiche les étiquettes du sous-arbre gauche, puis celles du sous-arbre droit, puis l'étiquette de la cible}
Paramètres    (D)    Cible    :    ArbreBinaire
Variables            FG       :    ArbreBinaire
                     FD       :    ArbreBinaire
Début
    Si (non estVide) alors
        {affichage de toutes les valeurs du sous-arbre gauche}
        FG ← FilsGauche
        FG.AfficheSuffixe

        {affichage de toutes les valeurs du sous-arbre droit}
        FD ← FilsDroit
        FD.AfficheSuffixe

        {affichage de la valeur portée par la racine}
        Afficher (Etiquette)
    Fin Si
Fin
```

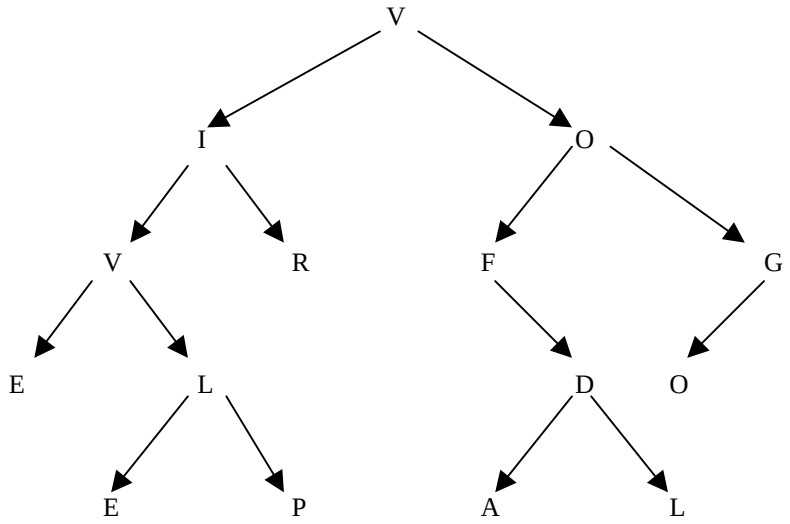
MProcédure AfficheInfixe

```
{affiche les étiquettes du sous-arbre gauche, puis l'étiquette de la cible, puis celles du sous-arbre droit}
Paramètres    (D)    Cible    :    ArbreBinaire
Variables            FG       :    ArbreBinaire
                     FD       :    ArbreBinaire
Début
    Si (non estVide) alors
        {affichage de toutes les valeurs du sous-arbre gauche}
        FG ← FilsGauche
        FG.AfficheInfixe

        {affichage de la valeur portée par la racine}
        Afficher (Etiquette)

        {affichage de toutes les valeurs du sous-arbre droit}
        FD ← FilsDroit
        FD.AfficheInfixe
    Fin Si
Fin
```

Essayez les trois algorithmes d’affichage sur cette arborescence :



En fait, on peut avoir besoin de l’une ou l’autre selon ce qu’on veut afficher.

3.2.6 Algorithme du tri par tas

Soit une suite de nombres.
- Etiqueter les sommets de l'arbre binaire avec les valeurs à trier, lors d'un parcours en largeur de cet arbre
- Réorganiser l'arbre binaire étiqueté en un tas
- Boucler :
  - Retirer la valeur portée par la racine et la remplacer par la valeur portée par la feuille la plus à droite
  - Supprimer cette feuille
  - Réorganiser l'arbre binaire en un tas

Les valeurs retirées tour à tour forment la suite triée.

3.2.7 Les arbres binaires de recherche

Afin de créer un arbre binaire de recherche (pour chaque sommet : l'étiquette du fils gauche est inférieure ou égale à celle du père, l'étiquette du fils droit est supérieure à celle du père), on peut ajouter la méthode suivante à la classe ArbreBinaire :

```
MFonction AjoutDansABR (Val) retourne (booléen)
    Paramètres   (D/R)  Cible  :  ArbreBinaire
                 (D)    Val    :  Info
    Début
        Si Cible.estVide alors          {on pourrait se contenter d'écrire « Si estVide alors »}
            Retourne (Cible.Création (Val)
        Sinon
            Si (Val ≤ Cible.GetEtiquette) alors
                Si (Cible.FilsGauche.estVide) alors
                    Cible.AjoutFilsGauche (CodePb, FG)
                    Si (CodePb = 0) alors
                        FG.SetEtiquette (Val)
                    Fin Si
                Sinon
                    CodePb ← Cible.FilsGauche.AjoutDans ABR (Val)
                Fin Si
            Sinon
                Si (Cible.FilsDroit.estVide) alors
                    Cible.AjoutFilDroit (CodePb, FD)
                    Si (CodePb = 0) alors
                        FD.SetEtiquette (Val)
                    Fin Si
                Sinon
                    CodePb ← Cible.FilsDroit.AjoutDansABR (Val)
                Fin Si
            Retourne (CodePb)
        Fin Si
    Fin
```

L'affichage des valeurs triées nécessite alors un parcours infixe, à l'aide de la méthode vue plus haut.