Structures hiérarchiques: arbres.

Résumé :

Ce chapitre est consacré aux arbres, l'un des concepts algorithmiques les plus importants de l'informatique. Les arbres servent à représenter un ensemble de données structurées hiérarchiquement.

Table de matière :

I. ARBRES

1. Définitions

- Un arbre est une structure dynamique d'éléments appelés aussi parfois « sommet » ou « Nœud ». Ses nœuds sont organisés d'une manières hiérarchique (Père, Fils , Petit-fils,…). Les nœuds sont reliés par des « Arcs » tel que chaque Nœud (à part la racine) a exactement un arc pointant vers lui. La «racine est un Nœud particulier car il n'a pas de prédécesseur. Les feuilles sont les nœuds sans successeurs. En fin, chaque nœud est composé de données et de pointeurs.

- Un arbre est composé d'un nœud particulier appelé racine ; de plusieurs nœuds intermédiaires possédant chacun un et un seul nœud appelé père, et des nœuds possédant éventuellement un ou plusieurs fils. Les nœuds qui ne possèdent pas de fils sont appelés feuilles.

  La figure 1 résume le différent composant d'un arbre :

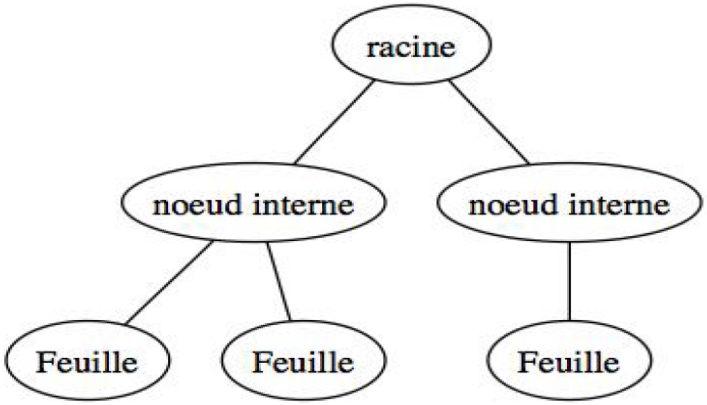


Figure 1 Description des différents composants d'un arbre.

2. Terminologie de base

2.1. Arité d'un arbre

L'arité de l'arbre est le nombre de fils qu'il possède. Un arbre dont les nœuds ne comporteront qu'au maximum *n* fils sera d'arrité *n*. On parlera alors d'arbre n-aire. Il existe un cas particulièrement utilisé : c'est l'arbre binaire. Dans un tel arbre, les nœuds ont au maximum 2 fils. On parlera alors de fils gauche et de fils droit pour les nœuds constituant ce type d'arbre.
L'arité n'impose pas le nombre minimum de fils, il s'agit d'un maximum, ainsi un arbre d'arité 3 pourra avoir des nœuds qui ont 0,1,2 ou 3 fils, mais en tout cas pas plus.
On appelle degré d'un nœud, le nombre de fils que possède ce nœud.

2.2. Taille et hauteur d'un arbre

On appelle la taille d'un arbre, le nombre de nœud interne qui le compose. C'est à dire le nombre nœud total moins le nombre de feuille de l'arbre.
On appelle également la profondeur d'un nœud la distance en terme de nœud par rapport à l'origine. Par convention, la racine est de profondeur 0. Dans l'exemple suivant (Figure 2) le nœud F est de profondeur 2 et le nœud H est de profondeur 3.

La hauteur de l'arbre est alors la profondeur maximale de ses nœuds. C'est à dire la profondeur à laquelle il faut descendre dans l'arbre pour trouver le nœud le plus loin de la racine.

On peut aussi définir la hauteur de manière récursive : la hauteur d'un arbre est le maximum des hauteurs de ses fils. C'est à partir de cette définition que nous pourrons exprimer un algorithme de calcul de la hauteur de l'arbre.

La hauteur d'un arbre est très importante. En effet, c'est un repère de performance. La plupart des algorithmes que nous verrons dans la suite ont une complexité qui dépend de la hauteur de l'arbre. Ainsi plus l'arbre aura une hauteur élevée, plus l'algorithme mettra de temps à s'exécuter

Exemple :

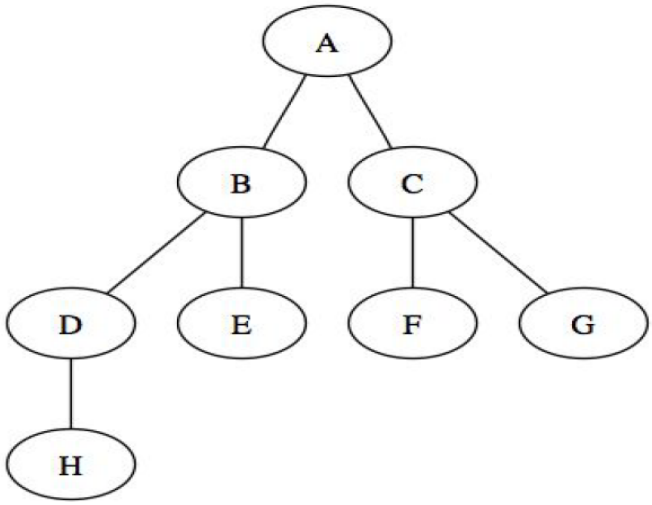


Figure 2 Représentation d'un arbre.

- {***A,B,C,,D,E,F,G,H***} est l'ensemble de nœuds de l'arbre
- Le nœud ***A*** est appelé la racine de l'arbre ;
- Le nœud ***E , F, G*** et ***H*** sont appelés les feuilles de l'arbre.
- La hauteur d'un arbre est le plus long chemin qui mène de la racine à une feuille =3.

II. ARBRES BINAIRES

1. Définitions :

- Un arbre binaire est un cas particulier des arbres.
- Un arbre binaire est un arbre où chaque nœud possède au plus deux fils : le fils droit et le fils gauche. Même si un nœud possède un seul fils, il peut être un fils gauche ou un fils droit.
- Un arbre binaire *A* est :
    - Soit l'arbre vide, noté ϕ.
    - Soit un triplet (*Ag,r,Ad*) où :
        - r est un nœud, appelé la racine de *A*.
        - Ag est un arbre binaire, appelé le sous-arbre gauche de *A*.
        - Ad est un arbre binaire, appelé le sous-arbre droit de *A*.

Exemple : l'arbre binaire de l'expression arithmétique (a+b)*(c-d)

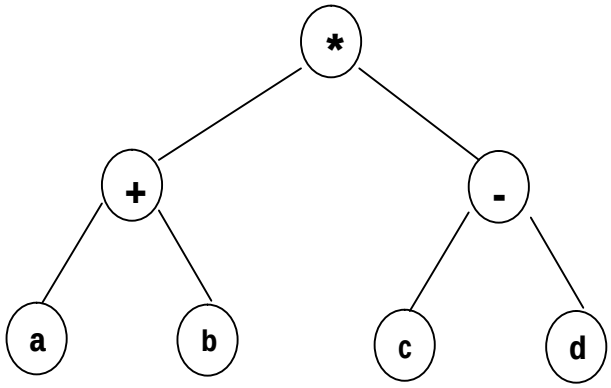


Figure 3 Arbre binaire de l'expression arithmétique (a+b)*(c-d)

2. Implémentation

Un nœud est une structure (ou un enregistrement) qui contient au minimum trois champs : un champ contenant l'élément du nœud, c'est l'information qui est importante. Cette information peut être un entier, une chaîne de caractère ou tout autre chose que l'on désire stocker. Les deux autres champs sont le fils gauche et le fils droit du nœud. Ces des fils sont en fait des arbres, on les appelle généralement les sous arbres gauches et les sous arbres droit du nœud. Chaque nœud possède la représentation de la figure :

Valeur	Ag	Ad

Figure 4 Représentation d'un nœud.

De part cette définition, un arbre ne pourra donc être qu'un pointeur sur un nœud. Voici donc une manière d'implémenter un arbre binaire :

```
Type Arbre= pointeur( Noeud)
Noeud= structure
Valeur= TElement
Ag = Arbre ; {Fils gauche}
Ad = Arbre ; {Fils droit}
Fin
```

On remplacera le type TElement pas type ou la structure de données que l'on veut utiliser comme entité significative des nœuds de l'arbre. De par cette définition, on peut donc aisément constater que l'arbre vide sera représenté par la constante NULL.

Exemple : représentation de l'expression (a+b)*(c-d)

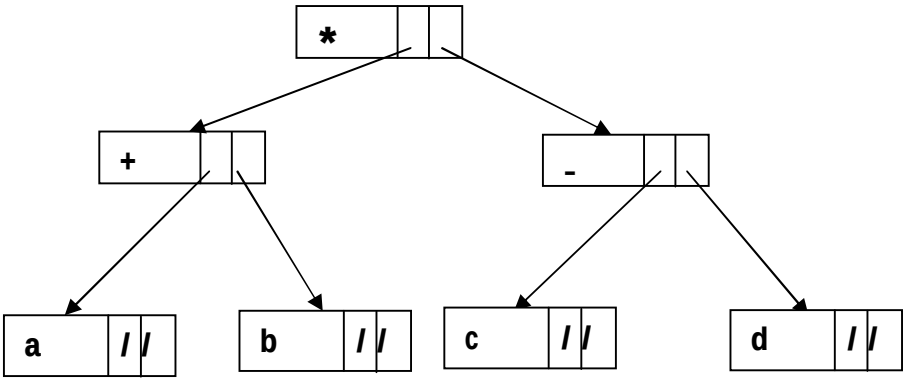


Figure 5 Représentation de l'expression (a+b)*(c-d).

3. Les fonctions de base sur la manipulation des arbres

Afin de faciliter notre manipulation des arbres, nous allons créer quelques fonctions :

- **La fonction qui détermine si un arbre est vide :**

```
fonction EstVide( T : arbre ) renvoie un booléen
si T == Null alors
renvoyer vrai;
sinon
renvoyer faux;
fin si
```

- **Les fonctions qui vont nous permettre de récupérer le fils gauche ainsi que le fils droit d'un arbre** :Il faut faire attention à un problème : le cas où l'arbre est vide. En effet, dans ce cas, il n'existe pas de sous arbre gauche ni de sous arbre droit. Pour régler ce problème nous décidons arbitrairement de renvoyer l'arbre vide comme fils d'un arbre vide.

```
fonction FilsGauche( T : arbre ) renvoie un arbre
si EstVide(T) alors
renvoyer arbre_vide;
sinon
renvoyer sous arbre gauche;
fin si
```

La fonction qui retourne le fils droit sera codée de la même manière mais tout simplement au lieu de renvoyer le fils gauche, nous renvoyons le fils droit.

- **La fonction qui détermine si le nœud est une feuille :**

```
fonction EstUneFeuille(T : arbre) renvoie un booléen.
{
si EstVide(T) alors
renvoyer faux;
sinon
        si EstVide(FilsGauche(T)) et EstVide(FilsDroit(T)) alors
        renvoyer vrai;
        sinon
        renvoyer faux;
        fin si
fin si
}
```

- **La fonction qui détermine si un nœud est un nœud interne :**

Pour ce faire, deux méthodes : soit on effectue le test classique en regardant si un des fils n'est pas vide, soit on utilise la fonction précédente.
Voici le pseudo code utilisant la fonction précédente :

```
fonction EstNoeudInterne( T : arbre ) renvoie un booléen
si EstUneFeuille(T) alors
renvoyer faux;
sinon
renvoyer vrai;
fin si
```

4. Parcours d'arbres

Un parcours d'arbres est un algorithme qui permet de visiter chacun des nœuds de cet arbre. Nous distinguerons deux types de parcours : le parcours en profondeur et le parcours en largeur. Le parcours en profondeur permet d'explorer l'arbre en explorant jusqu'au bout une branche pour passer à la suivante. Le parcours en largeur permet d'explorer l'arbre niveau par niveau. C'est à dire que l'on va parcourir tous les nœuds du niveau 1 puis ceux du niveau deux et ainsi de suite jusqu'à l'exploration de tous les nœuds.

4.1. Parcours en profondeur

On définit trois parcours en profondeur privilégiés qui sont :

- Le parcours préfixé (pré-ordre),
- Le parcours infixe,
- Le parcours postfixé (post-ordre).

De manière formelle :

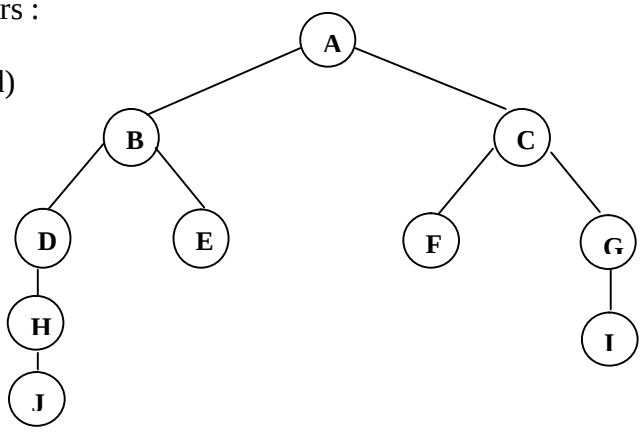
Les parcours préfixé (infixé, suffixé) sont définis comme suit.

- Si A est l'arbre vide, alors pref(A) = inf(A) = suf(A) = ϕ;
- Si A = (Ag; r;Ad), et e(r) est le nom de r, alors :

pref(A) = e(r)pref(Ag)pref(Ad)
inf(A) = inf(Ag)e(r)inf(Ad)
suf(A) = suf(Ag)suf(Ad)e(r)

Exemple:

- ✓ Préfixe: ABDHJECFGH
- ✓ Infixé: JHDBEAFCIG
- ✓ Postfixé: JHDEBFIGCA



4.1.1. Le parcours préfixé (pré-ordre)

Le parcours préfixé consiste à effectuer les opérations suivantes :

- Traitement de la racine ;
- Parcours du sous-arbre gauche ;
- Parcours du sous-arbre droit.

➔ NGD (Nœud, Gauche, Droite).

```
Procedure parcours_prof_prefixe( A : arbre )
si non EstVide(A) alors
traiter_racine(A);
parcours_prof_prefixe(FilsGauche(A));
parcours_prof_prefixe(FilsDroit(A));
fin si
```

4.1.2. Le parcours infixé

Le parcours infixé consiste à effectuer les opérations suivantes :

- Le parcours du sous-arbre gauche ;
- Traitement de la racine ;
- Parcours du sous arbre-droit.

➔GND (Gauche, Nœud, Droite).

```
Procédure parcours_prof_infixe( A : arbre )
si non EstVide(A) alors
parcours_prof_infixe(FilsGauche(A));
traiter_racine(A);
parcours_prof_infixe(FilsDroit(A));
fin si
```

4.1.3. Parcours postfixé ou suffixé

Le parcours postfixé consiste à effectuer les opérations suivantes :

- Parcours du sous-arbre gauche ;
- Parcours du sous-arbre droit ;
- Traitement de la racine .

➔GDN (Gauche, Droite, Nœud).

```
Procédure parcours_prof_suffixe(T : arbre )
si non EstVide(T) alors
parcours_prof_suffixe(FilsGauche(T));
parcours_prof_suffixe(FilsDroit(T));
traiter_racine(T);
fin si
```

4.2. Parcours en largeur (ou par niveau)

Nous allons aborder un type de parcours un peu plus compliqué, c'est le parcours en largeur. Il s'agit d'un parcours dans lequel, on traite les nœuds un par un sur un même niveau. On passe ensuite sur le niveau suivant, et ainsi de suite.
Le parcours en largeur de l'arbre précédant est : ABCDEFGHIJ.

Une méthode pour réaliser un parcours en largeur consiste à utiliser une structure de données de type file d'attente.

Le principe est le suivant :

- Lorsque nous somme sur un nœud nous traitons ce nœud (par exemple nous l'affichons) puis nous mettons les fils gauche et droit non vides de ce nœud dans la file d'attente, puis nous traitons le prochain nœud de la file d'attente.

- Au début, la file d'attente ne contient rien, nous y plaçons donc la racine de l'arbre que nous voulons traiter.
- L'algorithme s'arrête lorsque la file d'attente est vide. En effet, lorsque la file d'attente est vide, cela veut dire qu'aucun des nœuds parcourus précédemment n'avait de sous arbre gauche ni de sous arbre droit. Par conséquent, on a donc bien parcouru tous les nœuds de l'arbre.

```
procedure parcours_largeur(A : arbre)
Creer_File_D'attente F
ajouter(F,racine (A));
Tant que F n'est pas vide faire
        X <- extraire(F);
        Traiter_racine(X);
        Si  non EstVide(FilsGauche(X)) alors
                ajouter(F,FilsGauche(X));
        fin si
        si  non EstVide(FilsDroit(X)) alors
                ajouter(F,FilsDroit(X));
        fin si
fin TQ
```

5. Application : arbres d'expressions

Une expression arithmétique peut-être représentée de trois façons différentes : infixée, postfixée et préfixée.
Exemple : l'expression arithmétique **a+b**

- Exp. Préfixée : + a b
- Exp. Infixée : a + b
- Exp. Postfixée : a b +

Les expressions post-fixées et préfixées ne nécessitent pas de parenthèses pour être évaluées. Les expressions infixées doivent être transformées en expressions postfixées ou préfixées pour pouvoir être évaluées par les langages de programmation.

Transformation d'une expression infixée en une expression postfixée :

A * B ⟶ A B *

A * B + C ⟶ A B * C +

A + B * C ⟶ A B C * +

A + B * C - D / E ⟶ A B C * + D E / -

Evaluation d'une expression postfixée

- On utilise une pile pour empiler les opérandes.
- Lorsque le caractère lu est un opérateur, on dépile les deux derniers éléments de la pile, on effectue l'opération correspondante, et on remet le résultat dans la pile.

Représentation d'une expression par un arbre :

On utilise un arbre binaire ayant les propriétés suivantes :

1. Chaque feuille est un opérande.
2. La racine et les nœuds internes sont des opérateurs.
3. Les sous-arbres représentent des sous-expressions.

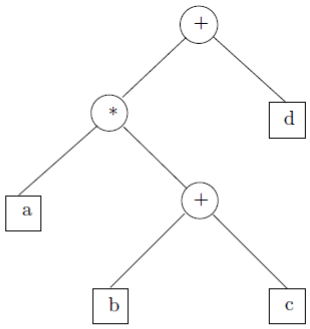


Figure 6 Arbre de l'expression a*(b+c)+d.

III. ARBRES BINAIRES DE RECHERCHE

1. Introduction :

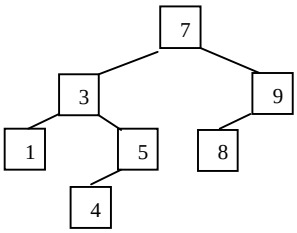
La recherche d'un élément dans un tableau contenant des données ordonnées était très efficace: recherche binaire, en temps logarithmique. Cependant, l'insertion et la suppression d'un élément dans un tableau est très inefficace. Inversement, dans le cas d'une liste chaînée, l'insertion et la suppression sont très efficaces, mais la recherche d'un élément se fait par une procédure séquentielle. Comment combiner les deux avantages (recherche efficace, et insertion/suppression efficace): Arbre binaire de recherche.

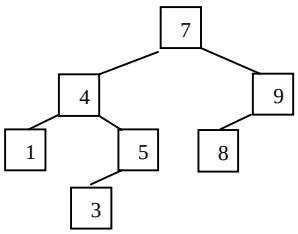
2. Définition :

Un arbre binaire de recherche est un arbre binaire ayant les propriétés suivantes (définition récursive):
1. Tous les nœuds du sous arbre gauche de la racine ont des valeurs inférieures ou égales à la valeur de la racine.
2. Tous les nœuds du sous arbre droit de la racine ont des valeurs strictement supérieures à la valeur de la racine.
3. Les sous-arbre gauche et droit sont eux mêmes des arbres binaires de recherche.

Dans la suite, on travaillera avec un arbre binaire représenté de façon chaînée par la représentation fils gauche, fils droit. Pour un nœud *N* :
– *N*.G (resp. *N*.D) représente la racine du sous-arbre gauche (resp. droit) de *N* (ou l'arbre vide si ce sous-arbre gauche (resp. droit) est vide).
– *N*.Valeur représente la valeur contenue dans le nœud *N*.

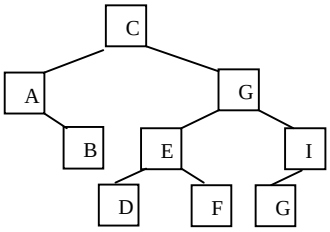
Exemples :
Exemple 1: Les clés sont des chiffres.





Contre exemple

Exemple 2: Les clés sont ici des lettres. L'ordre des clés est basé sur l'ordre alphabétique



3. Recherche d'un élément

```
Fonction Chercher(X, A) : Booléen
{ Rôle : la fonction recherche si un élément X appartient à un ABR A : Le résultat est
Vrai si X appartient, Faux sinon }
Paramétres Donnée
A : ABR {on travaille sur l'arbre A}
X : <type de base> {la valeur à rechercher}
Début
Si A = Vide
Alors Retourner (Faux ) {Recherche négative}
Sinon
        Si X = A.Valeur
        Alors Retourner (Vrai ) {Recherche positive}
        Sinon
                        Si X < A.Valeur
                        Alors Retourner (Chercher(X,A.G))
                        Sinon Retourner (Chercher(X,A.D))
                        Fin Si
        Fin Si
Fin Si
Fin
```

4. Adjonction d'un élément

Dans un ABR, l'opération de l'adjonction se décompose en deux étapes :

- Une étape de recherche, qui renvoie des informations sur la place où doit être inséré le nouvel élément ;
- Une deuxième étape de l'adjonction proprement dite.

1.1. Insertion aux feuilles

Dans cette méthode, on rattache un nouveau nœud à une feuille de l'arbre de recherche. L'ajout est précédé d'une phase de recherche de la place d'insertion pour qu'après insertion la propriété d'arbre binaire de recherche soit respectée.

La procédure récursive est donnée par :

```
Procédure Ajoutefeuille(X, A)
{ Rôle : Adjonction d'un élément X aux feuilles d'un arbre binaire de recherche A }
Paramètre Donnée
X : <type de base> { la valeur à ajouter}
Paramètre Donnée/Résultat
A : ABR { on travaille sur l'arbre A à modifier}
Début
Si A = Vide
Alors
A←Créer ABR()
A.Valeur ← X
A.G ← Vide
A.D ← Vide
Sinon
        Si X <A.Valeur
        Alors Ajoutefeuille(X,A.G)
        Sinon Ajoutefeuille(X,A.D)
        Fin Si
Fin Si
Fin
```

1.2. **Adjonction à la racine**

Pour pouvoir mettre une valeur *X* à la racine de l'arbre de recherche, il faut que toutes les valeurs précédemment contenues dans l'arbre de recherche soient séparées en deux arbres de recherche :

- un arbre qui sera le sous-arbre gauche de la racine et qui contiendra des valeurs inférieures ou égales à *X*,
- et l'autre arbre qui sera le sous-arbre droit de la racine et qui contiendra des valeurs strictement supérieures à *X*.

On considère deux étapes :

1. Une première étape qui consiste à couper l'arbre binaire de recherche en deux sous-arbres binaires de recherche.

2. Une deuxième étape qui consiste à rassembler les nouveaux sous-arbres crées autour de la nouvelle racine *X*.

- **Première étape**

La procédure de coupure de l'arbre binaire de recherche initial en deux sous-arbres binaires de recherche est donnée par :

```
Procédure Couper(X, A, G, D)
{ Rôle : coupure de l'ABR A, selon l'élément X, en deux ABR G et D;
G contient tous les éléments de A inférieurs ou égaux à X, et D contient tous les éléments de A
strictement supérieurs à X. }
Paramètre Donnée
X : <type de base> {la valeur autour coupe}
Paramètre Donnée/Résultat
A : ABR {ABR à couper}
Paramètres Résultat
G, D : ABR {deux nouveaux ABRs}
Début
Si A = Vide
Alors
G ← Vide
D ←Vide
Sinon
        Si X ≤A.Valeur Alors
        D ← A
        Couper(X, A.G, G, D.G)
        Sinon
        G ← A
        Couper(X, A.D, G.D, D)
        Fin Si
Fin Si
Fin
```

- **Deuxième étape**

La procédure de rassemblement des deux ABRs autour de la racine *X* est donnée par :

```
Procédure Ajouter Racine(X, A, G, D)
{ Rôle : la procédure réalise l'adjonction de l'élément X à la racine de l'arbre binaire de
recherche A; elle utilise la procédure de coupure d'un arbre binaire de recherche }
Paramètre Donnée
X : <type de base> {la valeur à insérer}
Paramètre Donnée/Résultat
A : ABR {ABR dans lequel on insère X}
Variable
R : ABR {de type ABR}
Début
R ← Créer ABR()
R.Valeur ← X
Couper(X, A, R.G, R.D)
A ← R
Fin
```

5. Suppression d'un élément

Dans un arbre binaire de recherche, l'opération de la suppression se décompose en deux étapes :

- En une étape de recherche ;
- Une deuxième étape de suppression qui dépend de la place de l'élément *X* dans l'arbre binaire de recherche.

Pour cette suppression, plusieurs cas sont possibles :

1. Si le nœud contenant la valeur est sans fils, on peut supprimer le nœud directement.

2. Si le nœud contenant la valeur possède un fils, on peut remplacer le nœud par son fils, et on obtient directement un arbre binaire de recherche.

3. Si le nœud contenant la valeur possède deux fils, plusieurs solutions sont possibles.

Dans ce cas, remplacer le nœud à supprimer par la feuille du sous-arbre gauche contenant la valeur maximale, ou par la feuille du sous-arbre droit contenant la valeur minimale.

• Fonction de suppression de l'élément *Max* dans arbre binaire de recherche :

```
Fonction Sup Max(A) : <type de base>
{ Rôle : cette fonction supprime le nœud contenant l'élément maximum dans un ABR A non
vide ; En résultat, elle renvoie l'élément Max de A, qui est privé de ce maximum. }
Paramètre Donnée/Résultat
A : ABR {ABR : on cherche (privé) de Max}
Variable
Max : <type de base> {Max élément à chercher}
Début
Si A.D = Vide Alors
        Max ← A.V aleur
        A ← A.G
        Retourner (Max)
Sinon
        Retourner (Sup Max(A.D))
Fin
```