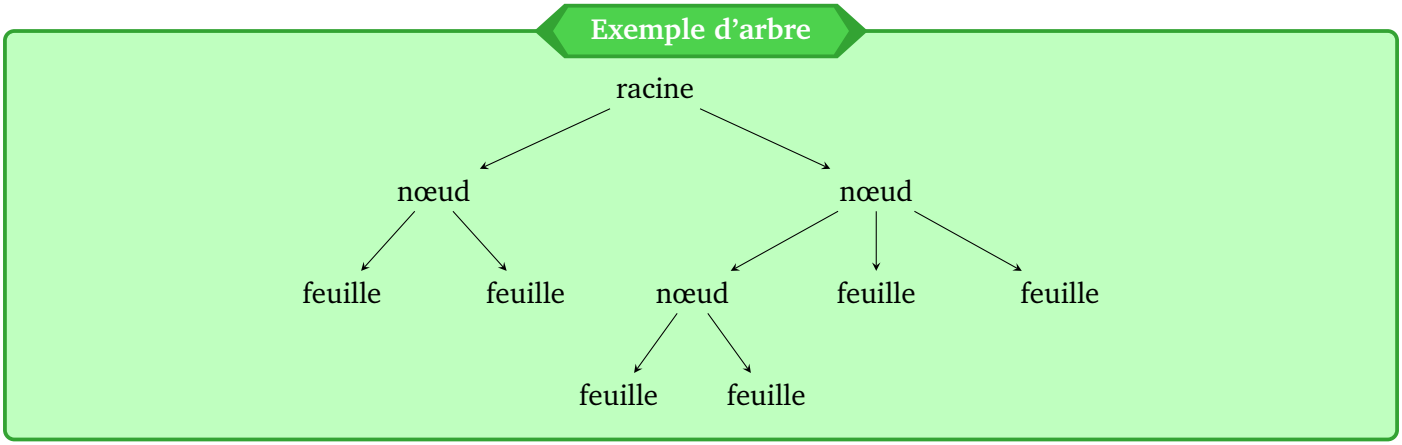
Arbres et récursivité

Florent Bouchez Tichadou

1er juillet 2020

Les arbres sont le deuxième type le plus commun de structures de données récursives après les listes chaînées. Les arbres sont composés de *nœuds*, chaque nœud pouvant comporter zéro ou plusieurs *enfants*, également des nœuds. On peut voir les arbres comme une généralisation des listes chaînées : dans une liste chaînée, chaque cellule a exactement un successeur—sauf la queue qui n'en a aucun—, dans un arbre chaque *nœud* a un ou plusieurs successeurs—sauf les *feuilles* qui n'en ont aucun.

Notez la particularité des arbres en informatique qui ont leur racine en haut, poussent vers le bas, et donc ont leurs feuilles en bas :-)



Pour un arbre non vide, on peut sélectionner un nœud particulier qu'on appelle la *racine* de l'arbre, qui est le point d'entrée de la structure. C'est l'équivalent de la *tête* pour une liste chaînée. Les successeurs d'un nœud sont appelés les *enfants* ou *fils* ; un nœud sans enfant est une *feuille*, les autres nœuds sont appelés les *nœuds internes*.

On entrevoit bien l'aspect récursif de la structure, en remarquant que les enfants d'un nœud interne sont eux-même les racines de *sous-arbres* de l'arbre général. L'utilisation de fonctions récursives est ainsi particulièrement adaptée aux arbres. Nous utiliserons donc le plus souvent possible des fonctions récursives dans ce document.

Comme pour les listes chaînées, les nœuds contiennent en général une information supplémentaire, leur *valeur*, qui peut être de n'importe quel type. Les arbres servent ainsi de structure de données, c'est-à-dire de *contenant* pour stocker un certain nombre d'éléments. Comme les tableaux et les listes chaînées, on peut ainsi stocker un *ensemble*, une *séquence* d'éléments ou plus généralement un *ensemble ordonné*.

1 Introduction aux Arbres binaires

Ce document va essentiellement parler d'*arbres binaires*, qui sont un sous-ensemble des arbres. Dans un arbre binaire, chaque nœud peut avoir au plus deux enfants. Ces enfants sont appelées les *fils gauche* et *droit* et sont les racines des *sous-arbres gauche* et *droit*. Un nœud peut n'avoir qu'un seul enfant—auquel cas l'autre fils est `Nil`—ou aucun enfant : c'est alors une feuille—et ses deux « fils » sont `Nil`.

1.1 Définition

On peut définir un arbre comme étant un nœud (sa racine), mais il est également possible d'encapsuler l'arbre dans une structure contenant un seul champ, `racine` (de la même manière qu'une séquence sous forme de

liste chaînée peut être représentée avec une structure contenant un seul champ, la `tête`). Dans ce document, nous allons prendre la première définition qui a l'avantage d'être plus claire et permet d'écrire des algorithmes récursifs plus facilement.

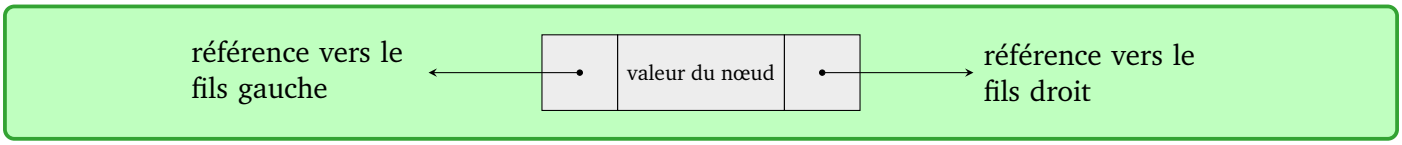
type nœud :
- valeur : élément
- gauche : référence vers nœud
- droit : référence vers nœud

type arbre : référence vers nœud
(définition utilisée par la suite)

Note : il est également possible comme pour les listes chaînées, d'encapsuler la racine dans une structure comme suit :

type arbre :
- racine : référence vers nœud

Une représentation possible des nœuds d'un arbre est la suivante, calquée sur la représentation des cellules dans les listes chaînées.

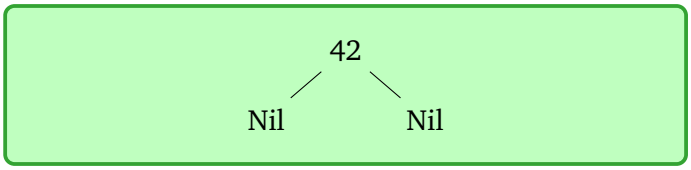


Cependant, en général, on ne représente pas les nœuds comme une structure avec les champs « valeur », « gauche » et « droit », on se contente d'une représentation plus épurée comme dans l'exemple ci-dessous, où l'on fait partir les liens de chaînage directement de la valeur du nœud, et même le sens des flèches n'est plus indiqué (elles sont toujours orientées du haut vers le bas).

1.2 Création

On suppose pour simplifier que, à la création d'un nouveau nœud, les champs gauche et droit sont initialisés à Nil.

```
r ← nouveau nœud
r.valeur ← 42
retourner r
```

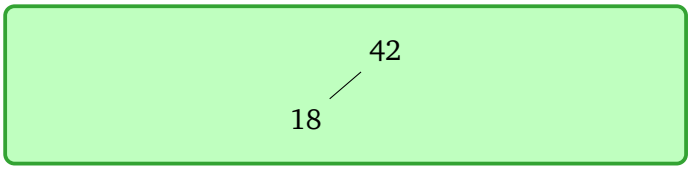


Note : par la suite, on ne représente plus les Nil, les feuilles sont simplement des nœuds sans descendant.

1.3 Ajouter un nœud

On suppose ici qu'on ajoute une feuille—i.e., on n'insère pas un nœud interne au milieu ou à la racine de l'arbre—en tant que fils gauche d'un nœud n.
⚠ si n avait déjà un fils gauche, on perd la référence vers l'ancien fils.

```
x ← nouveau nœud
x.valeur ← 18
n.gauche ← x
```



1.4 Fonctions récursives et parcours d'arbre

Dans le cas d'une liste chaînée, il est assez simple de faire un parcours à l'aide d'une boucle « tant que ». Pour un arbre binaire, c'est beaucoup plus compliqué, car il n'y a pas qu'un seul « suivant », mais deux ! On voudrait ainsi aller à gauche et à droite *en même temps*...

Si on regarde la tentative ci-dessous (fausse !), on se rend compte qu'on ne va parcourir que les fils droit des nœuds que l'on rencontre, et on « loupe » ainsi la majeure partie de l'arbre.

```
afficher_faux (r : racine d'un arbre)
    n ← r
    tant que n ≠ Nil faire                                   ⚠
        afficher (n.valeur)
        n ← n.gauche
        n ← n.droit      /* Mais que fait-on ? ? ? Ça ne marche PAS */
```

La solution la plus simple consiste à utiliser des fonctions récursives : en effet, une fois la branche gauche de l'arbre parcourue (par l'appel récursif), le programme « revient » au nœud où l'on se trouve et l'on peut continuer sur la branche droite.

```
afficher_rec (r : racine d'un arbre)
    si r = Nil alors
        retourner      /* rien à faire */
    sinon
        afficher (n.valeur)
        afficher_rec(n.gauche)
        afficher_rec(n.droit)
```

1.5 Fonctions récursives et modifications d'arbres

Il est assez fastidieux de créer des arbres « à la main », aussi, nous souhaitons définir des fonctions capables de modifier des arbres, par exemple pour y insérer de nouveaux nœuds. Considérons un problème simple, qui nous permettra de poser une question fondamentale…

Problème : on souhaite créer un arbre représentant un ensemble d'entiers E en partant d'un arbre vide (racine à Nil) et en y ajoutant tous les éléments de E. Pour ajouter un élément, trois possibilités s'offrent à nous : soit la racine est Nil et on y insère l'élément, soit on l'insère dans le sous-arbre gauche, soit dans le sous-arbre droit.

Proposition : créer une fonction récursive qui insère au hasard à gauche ou à droite.

```
cree_arbre_aleatoire(E : ensemble d'entiers) : arbre
    racine ← Nil
    pour chaque e ∈ E faire
        insere_alea(racine, e)
    retourner racine
```

Ces algorithmes sont **faux** ! Essayez de trouver pourquoi.

```
insere_alea(n, e)
    si n = Nil alors
        n ← nouveau nœud
        n.valeur ← e
    sinon
        si random_bool() alors
            insere_alea(n.gauche, e)
        sinon
            insere_alea(n.droit, e)
```

Comme indiqué dans l'encadré, les algorithmes proposés sont faux…

Ils incluent une erreur **fondamentale** faite par les étudiants (mais pas que) à tous les niveaux de la scolarité—y compris chez de nombreux étudiants de master…

Vous l'avez trouvée ? Si oui bravo ! Si non accrochez-vous et soyez particulièrement attentifs à la suite des explications.

Dans notre exemple, la fonction « insere_aleatoire » n'effectue que des **modifications locales** : à la création d'un nouveau nœud, l'argument n est modifié, mais **cette modification n'est pas visible par la fonction appelante**. Après chaque appel à « insere_aleatoire », racine reste donc à Nil . . .

Nous avons alors à résoudre un problème crucial : comment s'assurer d'une modification effectuée dans une fonction est bien visible depuis la fonction appelante ? En particulier : comment modifier la racine d'un arbre dans une fonction (récursive) ?

Il existe plusieurs manières de contourner ce problème, nous en présentons ici quatre :

1. renvoyer systématiquement la (nouvelle) racine ou nœud comme valeur de retour des fonctions ;
2. donner un argument supplémentaire aux fonctions : le père du nœud en cours ;
3. rajouter à la structure nœud une référence vers le père ;
4. utiliser des références de références : une racine étant déjà une référence sur une structure, pour la modifier il faut une référence sur cette référence. . .

La méthode n°4 est la plus efficace et la plus « propre » car elle n'alourdit pas les fonctions avec des arguments supplémentaires et évite les recopies inutiles. Cependant, elle nécessite d'être parfaitement au clair avec le concept des pointeurs à plusieurs niveaux en C ce qui la rend délicate à manipuler. En Python par contre elle nécessite de contourner le système pour obtenir des références de références et est complètement inadaptée.

La méthode n°2 peut être utilisée si l'on veut éviter des recopies inutiles, mais elle alourdit le code par l'ajout de nombreuses conditions qui permettent par exemple retrouver si le nœud en cours est le fils gauche ou droit du parent avant de faire une modification.

La n°3 est très semblable à la 2. On perd un peu de place mémoire car on doit stocker une information supplémentaire dans les nœuds.

Enfin, la méthode n°1 est conceptuellement la plus simple et permet d'obtenir un code propre et très compréhensible. C'est celle que nous utiliserons dans le reste du document par soucis d'uniformisation et pour conserver une ligne directrice.

Solution

```
cree_arbre_aleatoire(E : ensemble d'entiers) :
  arbre
    racine ← Nil
    pour chaque e ∈ E faire
      racine ← insere_alea(racine, e)
    retourner racine
```

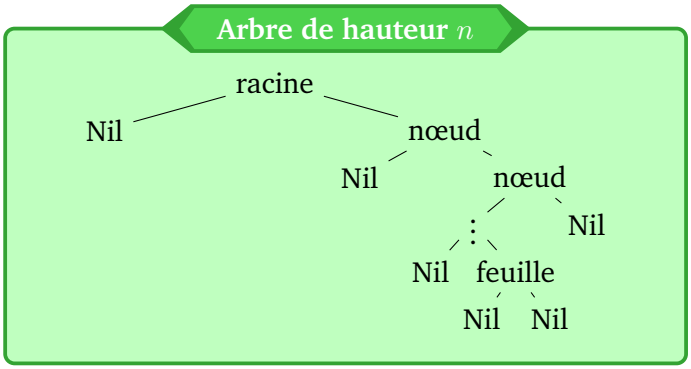
Version correcte, avec utilisation de la valeur de retour des fonctions pour mettre à jour les liens de chaînage dans l'arbre.

```
insere_alea(n : arbre, e) : arbre
    si n = Nil alors
      n ← nouveau nœud
      n.valeur ← e
    sinon
      si random_bool() alors
        n.gauche ← insere_alea(n.gauche, e)
      sinon
        n.droit ← insere_alea(n.droit, e)
    retourner n
```

2 Hauteur des arbres binaires

La *hauteur* (ou *profondeur*) d'un arbre est définie comme le nombre de nœuds dans le plus long chemin de la racine à une feuille. Par définition, un arbre vide a comme hauteur zéro, et un arbre avec un seul nœud (une feuille) a pour hauteur un. On peut calculer la hauteur facilement grâce à la fonction récursive suivante ci-après à gauche.

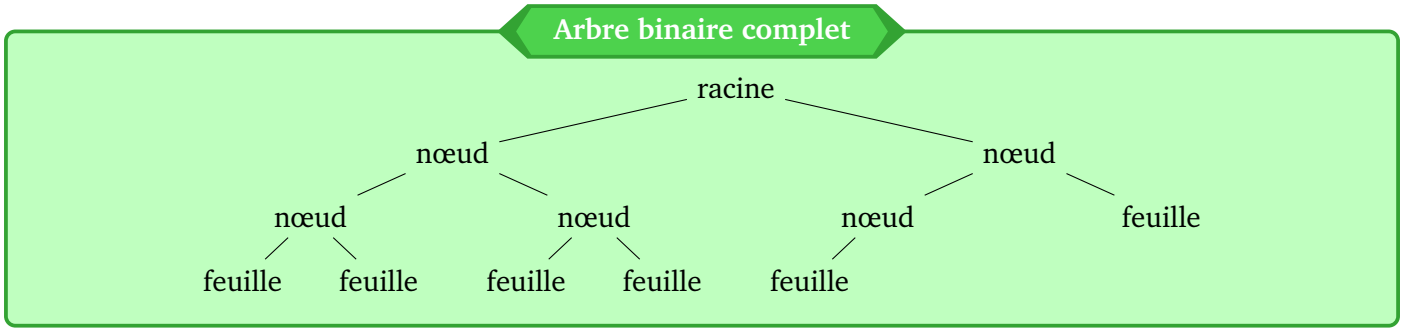
```
hauteur(n) : entier
    si n = Nil alors
        retourner 0
    sinon
        hg ← hauteur (n.gauche)
        hd ← hauteur (n.droit)
        retourner 1+ max (hg, hd)
```



Posons nous la question suivante : soit un arbre binaire à n nœuds, quelle est sa hauteur (en fonction de n) ? Même si on ne peut répondre de manière certaine pour tous les arbres, on peut au moins borner cette hauteur.

Considérons un arbre binaire contenant n nœuds. Dans le pire des cas, on peut créer un arbre de hauteur exactement n : chaque nœud a exactement un fils, l'autre étant `Nil` comme le montre l'exemple ci dessus à droite. En l'absence de propriété particulière sur un arbre sa hauteur est donc en $O(n)$.

À l'inverse, sur un arbre binaire complet, chaque niveau est complètement « rempli »—sauf éventuellement le dernier. Ceci signifie que si la hauteur de l'arbre est h, chaque chemin de la racine à une feuille contient h ou $h-1$ nœuds.



Calculons la hauteur h en fonction de n. Si l'on appelle la racine le niveau 0, le dernier niveau est le $h-1$, et tous remplis sauf éventuellement le niveau $h-1$. Appelons $niveau(l)$ le nombre de nœud au niveau l. Puisque chaque nœud dans les niveaux 0 à $h-2$ a exactement deux fils, $niveau(l) = 2 \times niveau(l-1)$ pour $l \in \{1, h-2\}$.

Nous avons $niveau(0) = 1$ (la racine), donc nous pouvons conclure que $niveau(l) = 2^l$ pour $l \in \{0, h-2\}$.

De plus $1 \leqslant niveau(h-1) \leqslant 2^{h-1}$, nous avons alors :

$$\begin{aligned} n &= \sum_{l=0}^{h-1} niveau(l) \\ &= \sum_{l=0}^{h-2} 2^l + niveau(h-1) \\ &= 2^{h-1} - 1 + niveau(h-1) \end{aligned}$$

et donc,

$$\begin{aligned} 2^{h-1} - 1 + 1 \leqslant \quad n \quad &\leqslant 2^{h-1} - 1 + 2^{h-1} \\ 2^{h-1} \leqslant \quad n \quad &\leqslant 2^h - 1 \end{aligned}$$

En passant au logarithme tous les membres de nos inégalités, nous obtenons $h - 1 \leqslant \log n$ et $h \geqslant \log(n+1)$. Donc $\log(n+1) \leqslant h \leqslant \log n + 1$.

Pour un arbre binaire complet, la hauteur est ainsi $\log n$ (logarithme en base 2). Le plus important à retenir est que, pour un arbre *équilibré*, la hauteur est en $O(\log n)$. Il est également possible de prouver qu'en moyenne, un arbre binaire est équilibré, c'est-à-dire qu'un arbre binaire aléatoire de n nœuds a une hauteur en $O(\log n)$.

3 Suppressions dans un arbre binaire

Les algorithmes présentés dans cette section modifient les arbres. Il est donc très important de s'assurer que les modifications dans les fonctions récursives sont bien visibles depuis les fonctions appelantes (voir ou re-voir la section 1.5). Nous allons encore une fois utiliser les valeurs de retour pour « renvoyer » les racines des sous-arbres modifiées.

3.1 Suppression d'une feuille

Problème : écrire un algorithme qui cherche une feuille (n'importe laquelle) et la supprime.

Proposition : on descend dans l'arbre à gauche ou à droite jusqu'à trouver une feuille que l'on supprime. On utilise la fonction auxiliaire « est_feuille » qui renvoie Vrai si son argument est une feuille et Faux sinon.

```
est_feuille (n) : booléen
    retourner n.gauche = Nil et n.droit = Nil

supprime_feuille (n) : arbre
    si est_feuille (n) alors
        libérer n
        retourner Nil
    sinon
        si n.gauche ≠ Nil alors
            n.gauche ← supprime_feuille (n.gauche)
        sinon
            n.droit ← supprime_feuille (n.droit)
        retourner n
```

3.2 Supprimer un nœud interne

Avant de supprimer un nœud interne n, il faut s'assurer qu'on ne va pas perdre une partie de l'arbre. Pour garder les fils gauche et droit de n, on va chercher à remplacer n par un autre nœud de l'arbre, par exemple une feuille f qui n'a donc pas d'enfant. La méthode que nous allons voir ici convient si la position dans l'arbre n'a pas d'importance (par exemple si l'arbre sert à stocker un ensemble non ordonné).

En revanche, si l'arbre possède des propriétés particulières, il est important de bien choisir quel nœud prendra la place de n (voir à ce sujet l'exercice sur les arbres binaires de recherche).

Pour le remplacement de n par un autre nœud f, nous avons la possibilité soit de copier l'élément (la valeur) de f dans n (puis supprimer f au lieu de n), soit de modifier les liens de chaînage pour mettre f à la place de n. La première possibilité est efficace si les éléments contenus dans les nœuds sont petits (par exemple des entiers), en revanche il faut choisir la deuxième si les éléments prennent beaucoup de place (par exemple, si un nœud contient un tableau).

Pour cet algorithme, nous allons faire appel à une nouvelle fonction auxiliaire pour plus de clarté : `trouve_feuille` qui cherche une feuille dans un sous-arbre, la déconnecte, et renvoie un couple contenant le nouveau sous-arbre et la feuille.

⚠ Lors de l'utilisation, il est important de « raccrocher » le sous-arbre après la suppression ; par exemple, pour supprimer le fils gauche d'un nœud x, il faut utiliser la fonction ainsi :
« x.gauche ← supprime(x.gauche) ».

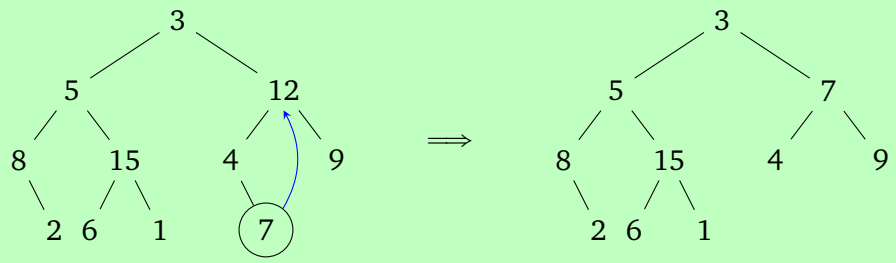
```
trouve_feuille (n) : (arbre, nœud)
/* Renvoie la feuille la plus à gauche de l'arbre ainsi que
   l'arbre modifié (sans la feuille)                          */
si n =Nil alors Erreur ("Arbre vide !")
si est_feuille (n) alors
    retourner (Nil , n)
sinon
    si n.gauche ≠ Nil alors
        (g', f) ← trouve_feuille (n.gauche)
        n.gauche ← g'
        retourner (n, f)
    sinon
        (d', f) ← trouve_feuille (n.droit)
        n.droit ← d'
        retourner (n, f)
```

```
supprime (n) : arbre
/* Version avec copie de l'élément.                           */
/* n supposé nœud interne.                                    */
(n', f) ← trouve_feuille (n)
n'.valeur ← f.valeur
libérer f
retourner n'
```

```
supprime (n) : arbre
/* Version avec modification des liens de chaînage.           */
(n', f) ← trouve_feuille (n)
f.gauche ← n'.gauche
f.droit ← n'.droit
libérer n'
retourner f
```

Exemple : suppression du nœud contenant 12

Le nœud est supprimé et remplacé par une feuille du sous-arbre enraciné en 12.



4 Complexité des opérations sur arbres binaires

La plupart des opération sur les arbres binaires dépendent soit du nombre de nœuds n, soit de la hauteur de l'arbre, notée h. Retenir que h peut-être $O(n)$ ou $O(\log n)$ selon l'équilibrage de l'arbre.

ajout : l'ajout d'un fils à un nœud se fait en $O(1)$. S'il faut chercher dans l'arbre une place libre où insérer le nœud (i.e., un nœud qui a au moins un fils `Nil`), alors l'ajout se fait en $O(h)$: on ne descend que à gauche ou à droite, et dans le pire des cas on parcours toute la hauteur et on tombe sur une feuille.

est_feuille : trivialement en $O(1)$.

supprime_feuille : dans le pire des cas, on part de la racine et la feuille la plus à gauche est la plus profonde : $O(h)$.

trouve_feuille : idem, $O(h)$.

supprime : $O(h)$ pour trouver une feuille, puis $O(1)$ pour faire le remplacement, donc $O(h)$ au total.

afficher_rec : beaucoup de fonctions sont en $O(h)$ car elles sont appelées récursivement uniquement sur un des deux fils, en revanche, pour afficher l'arbre, il nous faut parcourir tout l'arbre. Dans cette fonction récursive, on va parcourir chaque nœud exactement une fois, la complexité est donc $O(n)$, que l'arbre soit équilibré ou non.

hauteur : idem, $O(n)$. Notez que même si la hauteur *propriété de l'arbre*, peut être en $O(\log n)$, la *fonction* de calcul de cette hauteur nécessite bien de parcourir tout l'arbre !

5 Parcourir les arbres

Dans cette section, nous nous intéressons aux parcours d'arbres de manière générale. Nous avons déjà vu précédemment comment afficher un arbre et calculer la hauteur, fonctions qui nécessitent de parcourir tous les nœuds de l'arbre. Nous recensons ici les schémas génériques de parcours.

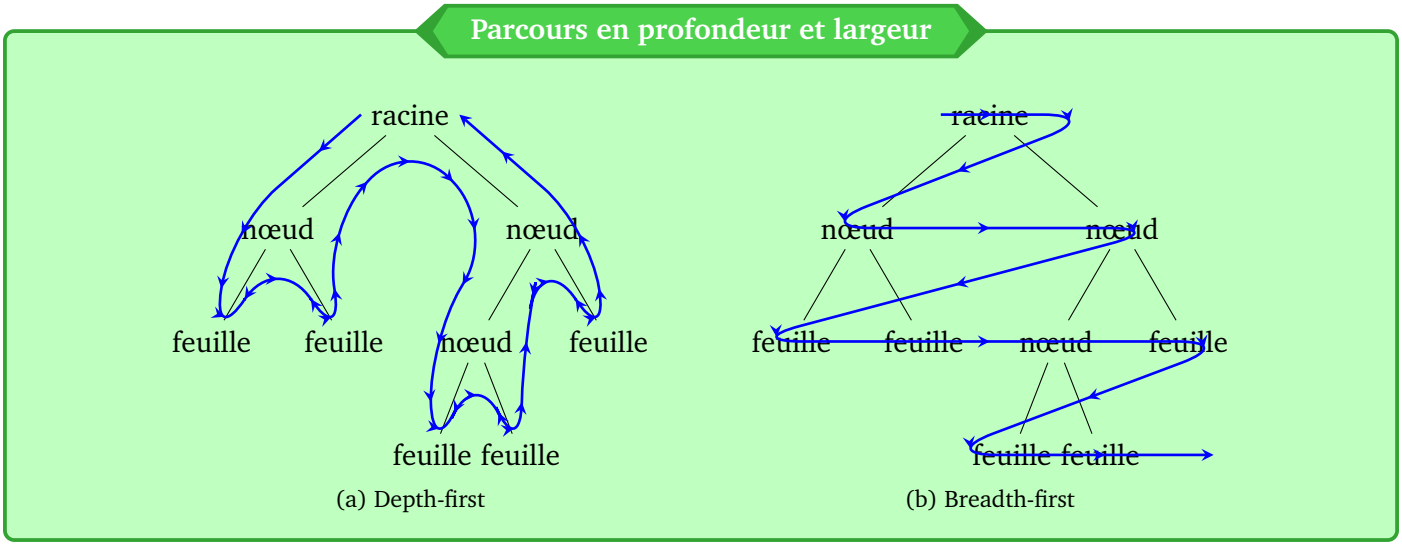
5.1 Parcours complets

Le fait d'exécuter une action sur chaque nœud d'un arbre (dans un ordre particulier ou non) est appelé un *parcours*. C'est une opération extrêmement courante, par exemple simplement pour afficher le contenu de chacun des nœuds de l'arbre à des fins de déboggage. Nous décrivons ici uniquement les parcours sur les arbres binaires, les algorithmes se généralisent facilement à des arbres quelconques.

Les deux parcours les plus courants diffèrent par l'ordre dans lequel ils rendent visite aux nœuds (voir Figure 1).

Parcours en profondeur : (*Depth-first*) on va d'abord en profondeur, en parcourant tous les nœuds d'un sous-arbre avant de visiter l'autre sous-arbre d'un nœud ; c'est le type de parcours le plus simple.

Parcours en largeur : (*Breadth-first*) on parcourt les nœuds par niveau : d'abord la racine, puis ses fils, puis les fils des ses fils, etc. Il est techniquement plus difficile.



(a) Depth-first (b) Breadth-first

FIGURE 1 – L'ordre de visite des nœuds dépend du type de parcours.

5.1.1 Parcours en profondeur

Il y a en fait trois variantes de parcours en profondeur, car on passe par un nœud trois fois en utilisant un algorithme récursif : quand on arrive à ce nœud depuis son père, quand on revient du sous-arbre gauche, puis quand on revient du sous-arbre droit. Nous avons donc trois possibilités pour faire notre action sur le nœud en cours :

Préfixe : on exécute l'action dès qu'on arrive pour la première fois dans le nœud, donc *avant* de parcourir les enfants ;

Infixe : on exécute l'action après le parcours du fils gauche, mais avant celui du fils droit, donc *entre* les appels récursifs ;

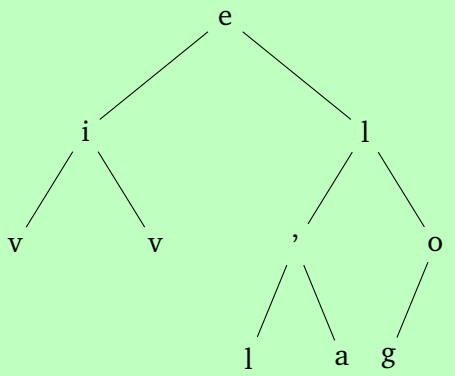
Postfixe : on exécute l'action une fois qu'on a parcouru les deux sous-arbres, donc *après* les appels récursifs.

Il est bien évidemment possible de mélanger ces trois ordres, et faire ainsi des actions, avant, entre, et après les visites aux enfants. Dans l'algorithme ci-dessous, c'est ce qu'on utilise pour faire un affichage de l'arbre avec délimitation des sous-arbres à base de parenthèses.

Algorithme récursif La manière la plus simple de faire un parcours en profondeur est d'utiliser la récursion, car la structure de donnée est elle-même récursive.

```
visite (n)
    si n = Nil alors retourner
    afficher "("                          /* Action préfixe */
    si n.gauche ≠ Nil alors
        visite (n.gauche)
    afficher n.valeur                     /* Action infixe */
    si n.droit ≠ Nil alors
        visite (n.droit)
    afficher ")"                          /* Action postfixe */
```

Parcours préfixe infixe et postfixe



L'action de `visite` sur l'arbre ci à gauche produirait la chaîne de caractères suivante :

```
((( v ) i ( v )) e ((( l ) ' ( a ) l (( g ) o ))))
```

Complexité : comme pour le calcul de la hauteur, on va parcourir exactement chaque nœud une fois (ou trois fois, donc un nombre constant). La complexité va donc dépendre de l'action effectuée sur chaque nœud. Si a est cette complexité, alors le parcours est en $O(n \times a)$.

Algorithme itératif Dans des cas extrêmement particuliers, il peut-être préférable d'utiliser un algorithme itératif (par exemple pour des arbres de profondeurs gigantesques, où l'on risque de dépasser la pile d'appels (*stack overflow*). Soyons clairs : ce genre de cas est rarissime, et si nous présentons l'algorithme ici, ce n'est pas pour l'utiliser en pratique mais parce qu'il permet de se familiariser avec le principe avant de l'utiliser pour le parcours en largeur.

Lorsqu'on utilise des fonctions récursives, c'est la pile des appels qui se « souvient » de où on vient dans l'arbre (chaque fonction appelée est empilée dans la *stack* et dépilé quand elle se termine). Sans récursivité, nous avons besoin de gérer nous même cette « mémoire » du chemin parcouru depuis la racine.

Dans notre cas, nous devons donc gérer une pile, par exemple avec un tableau ou une liste chaînée dans laquelle on met les nœuds qu'il nous reste à parcourir. On considère les deux actions classiques de gestion de pile, `push` qui rajoute un élément sur le haut de la pile, et `pop` qui enlève l'élément en haut de la pile et le renvoie.

Notez qu'il est plus difficile dans cette version d'avoir des actions infixe ou postfixe car on ne « passe » par un

nœud qu'une seule fois. L'algorithme ci-dessous effectue ainsi un parcours prefixe.

```
visite_itératif (racine : arbre)
    pile ← ∅
    n ← racine
    tant que n ≠ Nil ou pile ≠ ∅ faire
        tant que n = Nil et pile ≠ ∅ faire
            n ← pop (pile)                    /* On cherche le prochain nœud à parcourir dans la pile */
        si n = Nil alors
            retourner                         /* Plus de nœud à parcourir : on quitte la fonction */
        sinon
            afficher n.valeur                 /* Action préfixe */
            push (pile, n.droit)
            n ← n.gauche
```

Complexité : on a remplacé la pile d'appels récursifs par une pile que l'on gère « à la main », ce qui ne change pas la complexité : $O(n \times a)$, à condition que les opérations sur la pile soit bien en $O(1)$.

5.1.2 Parcours en largeur

Pour un parcours en largeur, il n'est pas possible d'utiliser la récursion car on passe notre temps à sauter d'un sous-arbre à un autre. De même que pour le parcours en profondeur itératif, on a besoin d'une structure de données auxiliaire qui va garder les informations sur les nœuds qu'il nous reste à parcourir. Cependant, dans ce cas, les nœuds que nous insérons doivent être parcourus en dernier (sinon on descendrait tout de suite en profondeur). On utilise alors une *file* d'attente au lieu d'une pile : au lieu de l'action `push`, on utilise l'action `append` qui ajoute en fin de file.

```
parcours_largeur (A : arbre)
    file ← { racine }
    tant que file ≠ ∅ faire
        n ← pop (file)
        afficher n.valeur
        si n.gauche ≠ Nil alors
            append (file, n.gauche)
        si n.droit ≠ Nil alors
            append (file, n.droit)
```

Complexité : on garde la même complexité que pour le parcours en profondeur : $O(n \times a)$ car on parcourt encore une fois tous les nœuds de l'arbre, mais dans un ordre différent (il faut bien entendu pouvoir faire les opérations sur la file en $O(1)$, ce qui est possible avec une liste chaînée par exemple).

6 Exercises

Exercice 1 (Échange de fils)

Écrire un algorithme qui échange les fils gauche et droit de tous les nœuds d'un arbre binaire.

Exercice 2 (Fusion d'arbres)

Soit deux arbres binaires A et B enracinés en r_A et r_B. On cherche à fusionner ces arbres pour n'en avoir plus qu'un.

Écrire un algorithme qui insère r_B à la première place trouvée dans A (à la place d'un fils Nil).

Exercice 3 (Arbres binaires de recherche (ABR))

Dans un arbre binaire de recherche, chaque nœud n contient une valeur entière v. Pour chaque valeur x contenue dans un nœud du sous-arbre gauche de n, on a $x \leqslant v$. Symétriquement, pour chaque valeur y contenue dans un nœud du sous-arbre droit de n, on a $v < y$.

Question 3.1 Écrire un algorithme qui recherche une valeur v dans un ABR. Quelle est sa complexité ?

Question 3.2 Comparez la complexité avec la recherche d'une valeur dans une séquence triée sous forme de tableau avec longueur. Lister les avantages et inconvénients à utiliser un ABR ?

Question 3.3 On souhaite afficher tous les éléments d'un ABR dans l'ordre croissant. Quel est le type de parcours à utiliser ? Et si on veut les afficher en ordre décroissant ?

Question 3.4 Expliquez comment l'insertion d'un nouveau nœud dans un ABR diffère de l'algorithme générique présenté section 1.3. Détaillez cet algorithme.

Question 3.5 De même, expliquez comment la suppression d'un nœud dans un ABR diffère de l'algorithme générique de la section 3.1. Écrivez l'algorithme de suppression d'une valeur v dans un ABR.

Exercice 4 (Tours de Hanoï)

(Texte introductif extrait de Wikipedia) *Les tours de Hanoï (originellement, la tour d'Hanoïa) sont un jeu de réflexion imaginé par le mathématicien français Édouard Lucas [. . .] Dans le grand temple de Bénarès, au-dessous du dôme qui marque le centre du monde, trois aiguilles de diamant, plantées dans une dalle d'airain, hautes d'une coudée et grosses comme le corps d'une abeille. Sur une de ces aiguilles, Dieu enfila au commencement des siècles, 64 disques d'or pur, le plus large reposant sur l'airain, et les autres, de plus en plus étroits, superposés jusqu'au sommet. C'est la tour sacrée du Brahmâ. Nuit et jour, les prêtres se succèdent sur les marches de l'autel, occupés à transporter la tour de la première aiguille sur la troisième, sans s'écarter des règles fixes que nous venons d'indiquer, et qui ont été imposées par Brahma. Quand tout sera fini, la tour et les brahmes tomberont, et ce sera la fin des mondes !*

Les règles sont les suivantes :

— on ne peut déplacer plus d'un disque à la fois ;
— on ne peut placer un disque que sur un autre disque plus grand que lui ou sur un emplacement vide.

On suppose que cette dernière règle est également respectée dans la configuration de départ.

Écrire un algorithme récursif qui résout le problème des tours de Hanoï pour n disques. Quelle est sa complexité ? Quand la fin du monde aura-t-elle lieu ?