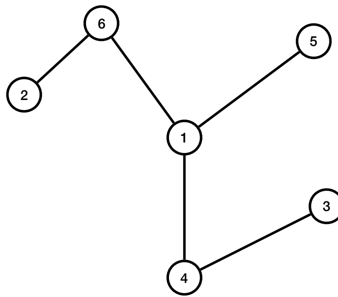
Arbres

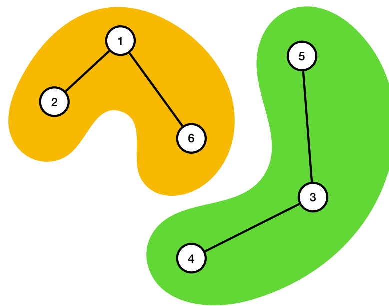
Benjamin Monmege

2021/2022

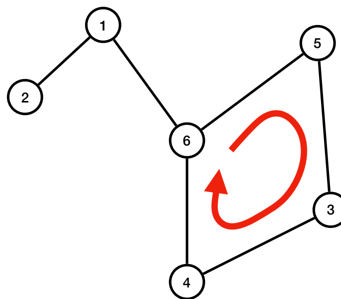
Un arbre est un graphe non orienté connexe et sans cycle.
Voilà donc un exemple d'arbre :



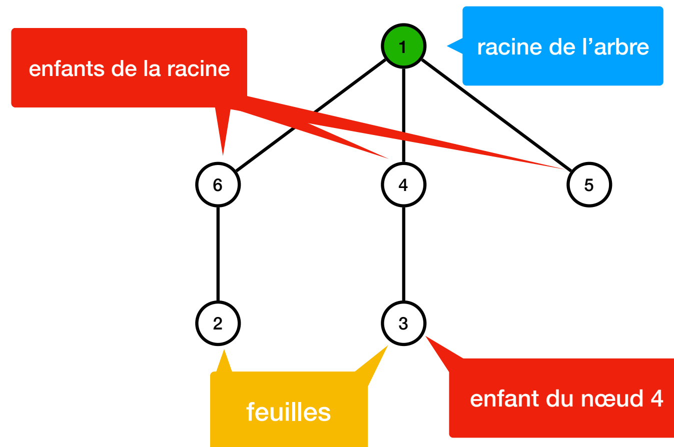
En revanche, l'exemple ci-dessous n'est pas un arbre puisque ce graphe n'est pas connexe : il est en deux morceaux détachés



Finalement, ce dernier exemple ci-dessous est bien connexe, mais possède un cycle : ce n'est donc pas non plus un arbre



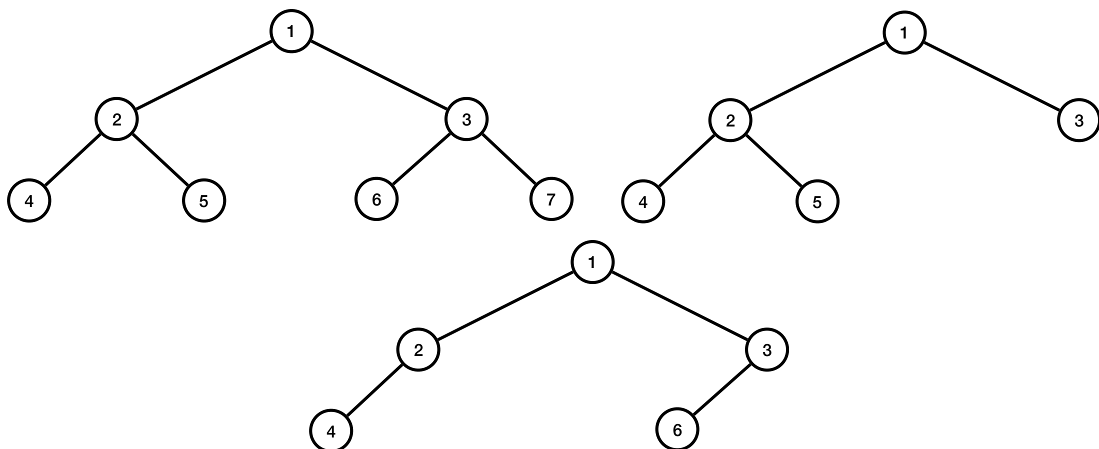
Une particularité de certains arbres est qu'ils ont un sommet ayant un statut particulier : c'est la *racine* de l'arbre. Il s'agit simplement d'un sommet désigné que l'on dessine généralement tout en haut de l'arbre (tout en bas dans le cas de l'arbre généalogique, tout à gauche dans le cas de l'arbre de probabilités). Un arbre possédant une racine s'appelle souvent un *arbre enraciné*. On peut donc reprendre l'arbre précédent, choisir 1 comme sommet racine, puis comme si on remontait le sommet 1 tout en haut en le tenant entre nos doigts, on obtient la représentation du même arbre :



Dans ce cas, les sommets 4, 5 et 6 sont les *enfants* du sommet 1. De même, le sommet 3 est l'unique enfant du sommet 4. Les sommets 2, 3 et 5 n'ont pas d'enfants : on dit qu'ils sont des *feuilles*, pour conserver l'analogie botanique. On dit d'ailleurs qu'un chemin allant de la racine à une feuille est une *branche* de l'arbre.

1 Arbres binaires

Parmi tous les arbres possibles, on distingue une classe importante en pratique : les *arbres binaires*. Ce sont des arbres enracinés dont chaque sommet a au plus 2 enfants : on distingue alors d'ailleurs son *enfant gauche* de son *enfant droit*. Les arbres ci-dessous sont donc des arbres binaires :

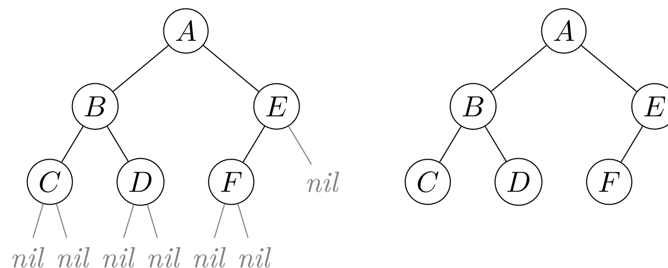


On peut alors donner une définition différente de ces arbres, plus proche de leur représentation dans un ordinateur. Il s'agit d'une définition *récursive* : cela veut dire qu'on s'autorise à utiliser la notion d'arbre binaire pour définir ce qu'est un arbre binaire. On fait de même lorsqu'on décide qu'un entier (naturel) est soit 0 soit le successeur $n+1$ d'un entier naturel n. Pour les arbres, on fait ainsi :

Définition 1. Un arbre binaire est :
— soit l'arbre vide, qu'on note souvent *nil* (et qu'on ne représente généralement pas dans les dessins) ;
— soit une racine ayant un enfant gauche et un enfant droit, qui sont tous deux des arbres binaires.

Dans le cas d'arbres enracinés, on appelle parfois *nœuds* les sommets de l'arbres.

À gauche ci-dessous le graphe est un arbre binaire, qu'on représente dans la suite comme dessiné à droite, sans les arbres *nil* :

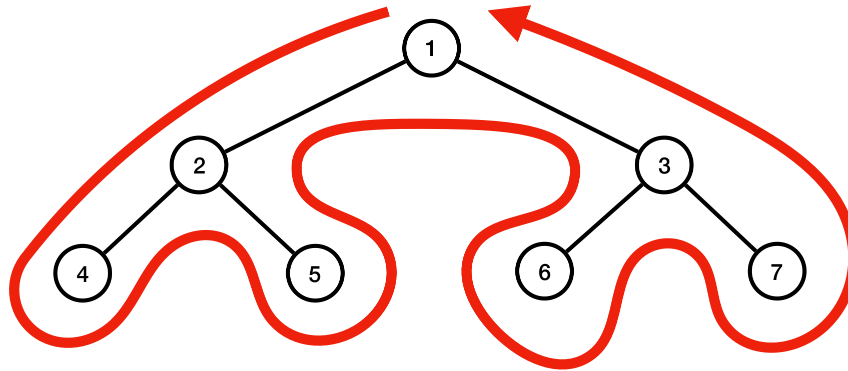


La racine de cet arbre est le nœud A qui a deux enfants : l'enfant gauche a B pour racine, et l'enfant droit a E pour racine. Les enfants du nœud C sont deux arbres vides *nil*. Une feuille est un nœud qui possède l'arbre vide comme enfants gauche et droit : les feuilles de l'arbre du dessus sont C, D et F.

2 Affichage d'une arborescence de fichiers : parcours préfixe

Parfois, on a besoin d'afficher une arborescence de fichier sous forme d'une liste de fichiers. Un outil permet de faire cela sous un système d'opérations Unix, tel que Linux ou Mac : l'utilitaire `ls` qu'on peut exécuter dans un terminal avec l'option `-R` pour signifier qu'on souhaite rentrer récursivement dans les sous-répertoires. Le programme `ls` commence par imprimer les noms des fichiers et répertoires contenus directement dans le répertoire racine. Ensuite, il imprime le contenu du premier répertoire entièrement, avant de passer au second répertoire, puis le troisième répertoire, etc.

Remis dans le contexte d'un arbre binaire (plutôt qu'un arbre quelconque comme pour l'arborescence de fichiers), cela revient à considérer le parcours suivant

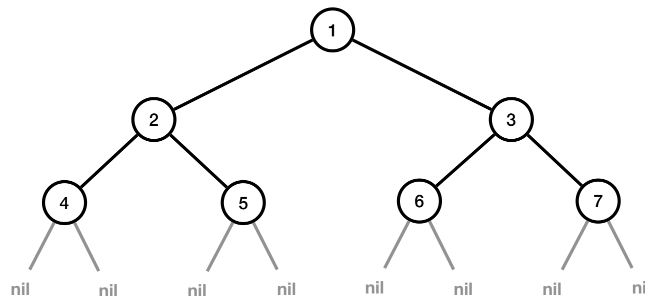


De plus, on imprime le contenu du sommet la première fois qu'on le visite, ce qui fait que, dans ce cas, on imprimerait donc les sommets dans l'ordre

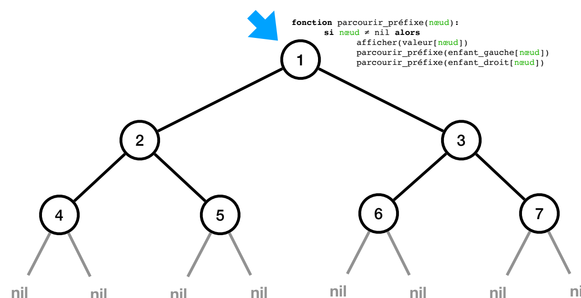
Un tel parcours, qu'on appelle *parcours préfixe* puisqu'il traite chaque nœud *avant* de traiter ses enfants, peut être implémenté par un algorithme récursif (nous avons déjà vu un algorithme récursif lorsque nous avons étudié le tri par fusion) :

```
fonction parcourir_préfixe(nœud) :
    Si nœud ≠ nil alors
        afficher(valeur[nœud])
        parcourir_préfixe(enfant_gauche[nœud])
        parcourir_préfixe(enfant_droit[nœud])
```

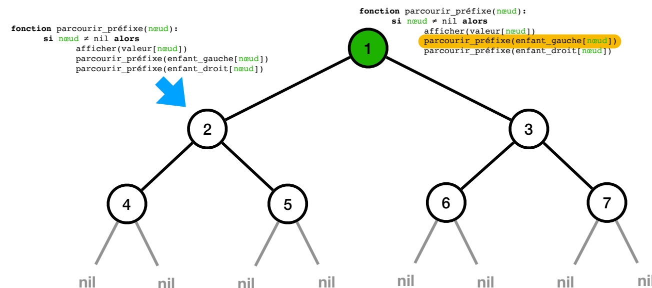
Testons-le sur l'arbre précédent pour mieux comprendre comment s'exécute un algorithme récursif. On commence par faire réapparaître les arbres vides :



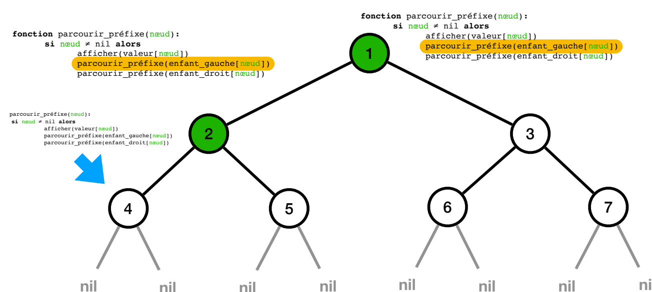
et on lance l'algorithme sur le nœud racine :



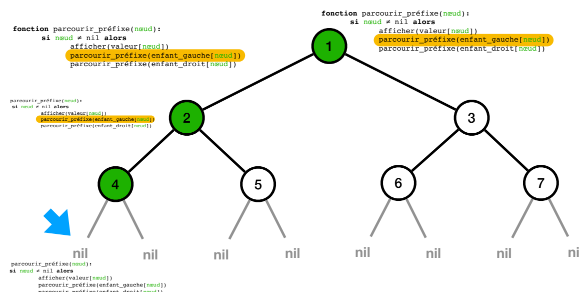
On commence par le test `nœud` $\neq$ `nil` qui est vrai puisque le nœud 1 n'est pas l'arbre vide. On exécute donc les trois lignes internes, en commençant par afficher la valeur du nœud. On appelle ensuite récursivement le même algorithme sur l'enfant gauche. Cela veut dire qu'on met en pause l'exécution courante et qu'on démarre une exécution indépendante depuis le début sur le nœud 2 :



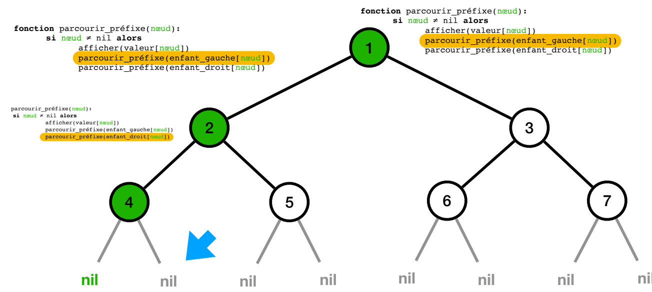
De même, on commence par imprimer le contenu du nœud 2, puis on fait l'appel récursif dans l'enfant gauche :



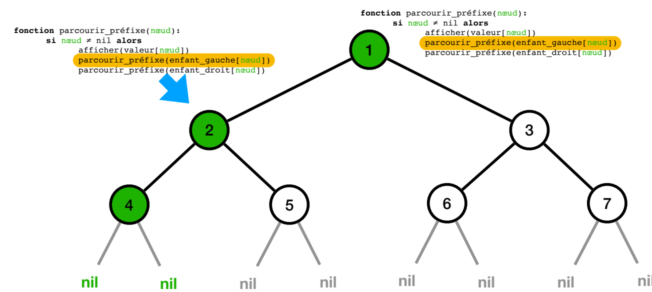
De même on affiche le contenu du nœud 4 et on s'appelle récursivement sur l'enfant gauche :



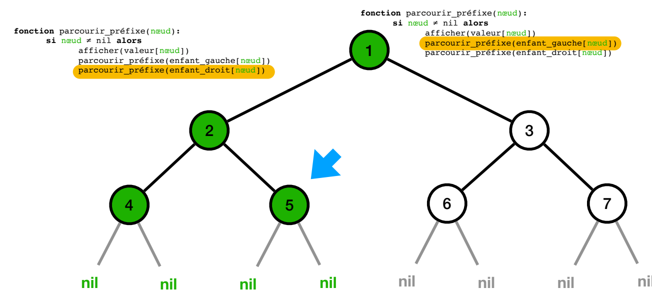
Cette fois, ce nœud est l'arbre vide. Le test `nœud` $\neq$ `nil` est donc faux ce qui implique que la fonction s'arrête tout de suite, sans rien faire. On revient donc dans l'algorithme mis en pause sur le nœud 4 : c'est comme si vous aviez le contrôle du direct sur plusieurs télévisions en parallèle, dès qu'une émission se termine sur l'une, le programme se remet en route sur une autre... La prochaine ligne de l'algorithme demande à exécuter l'algorithme sur le sous-arbre droit, vide également :



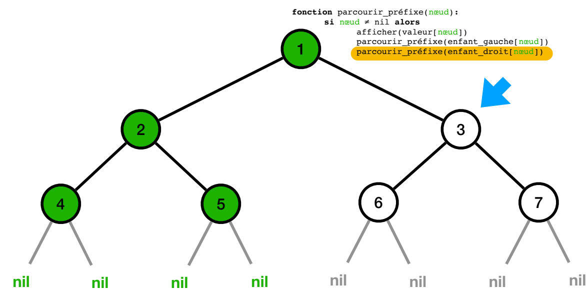
L'exécution s'arrête donc également immédiatement et on revient dans l'exécution sur le nœud 4 : mais celle-ci est arrivée à son terme et s'arrête donc naturellement. On retourne donc finalement dans l'exécution sur le nœud 2, là où on s'en était arrêté :



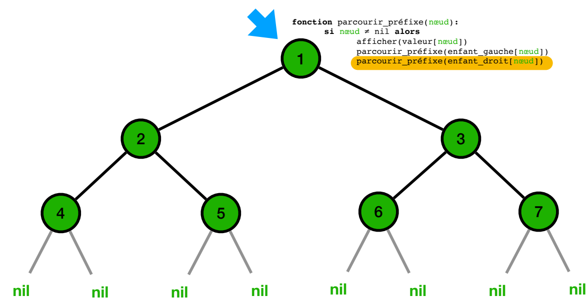
La prochaine ligne consiste à s'appeler récursivement sur l'enfant droit :



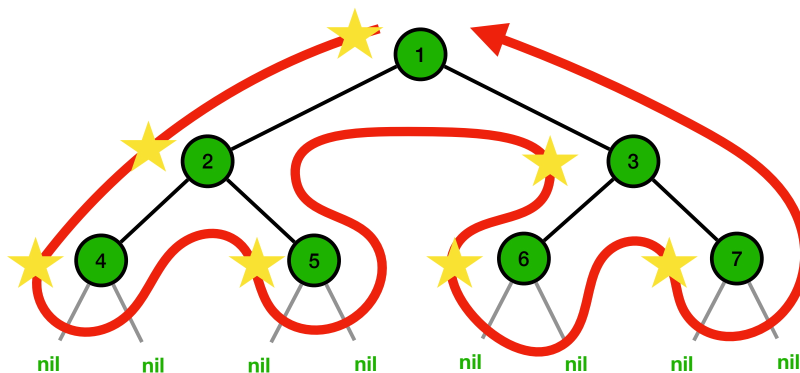
Comme pour le nœud 4, on va donc imprimer le contenu du nœud 5, puis s'appeler récursivement sur les deux arbres vides avant de terminer. L'exécution du nœud 2 termine ainsi, et on retourne donc dans l'exécution du nœud 1 :



Le parcours de son enfant droit permet d'imprimer les contenus des nœuds 3, puis 6, puis 7, avant de revenir à la fin de l'exécution du nœud 1 et de terminer :

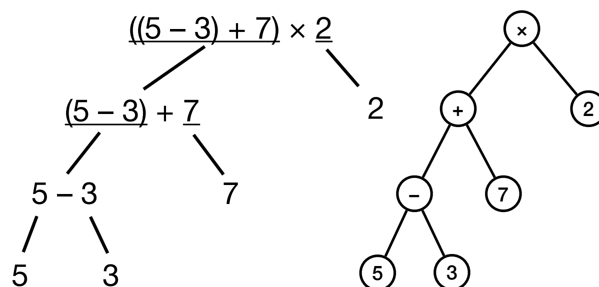


Au final, on a donc imprimé les contenus des nœuds dans l'ordre 1, 2, 4, 5, 3, 6, 7. Le long du parcours d'arbre, on traite donc les nœuds la première fois qu'on les a visités, c'est-à-dire sur les étoiles dans le diagramme suivant :



3 Expressions arithmétiques et parcours postfixe

Les arbres sont partout... même dans vos calculatrices. Lorsque vous tapez un calcul, par exemple $\big((5-3)+7\big) \times 2$, la calculatrice, elle, stocke un arbre binaire en mémoire. Pour cela, elle essaie de décomposer le calcul en deux tant que c'est possible. Au début, elle trouve donc l'opération la plus prioritaire, le produit, pour décomposer le calcul en le produit de $(5-3)+7$ et de 2. À nouveau, elle décompose le sous-calcul $(5-3)+7$ comme la somme de $5-3$ et de 7, et ainsi de suite jusqu'à ce que les sous-calculs soient tous des constantes. On obtient donc la représentation par l'arbre ci-dessous, qu'on résume à droite en conservant dans chaque nœud uniquement l'opération qui permet la décomposition :



Qu'est-ce qu'un calcul pour une calculatrice ? Considérons, pour simplifier la présentation, une calculatrice très simple avec uniquement les opérations de somme, de soustraction, de

produit et de division. Ainsi, ce que reçoit la calculatrice de l'utilisateur est une expression arithmétique qui est :

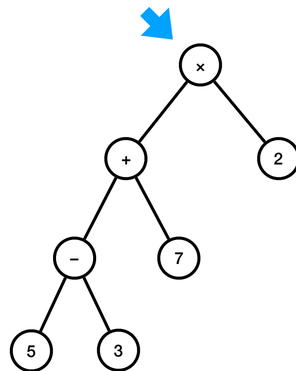
— soit un nombre n ;
— soit une somme $(e_1 + e_2)$;
— soit une soustraction $(e_1 - e_2)$;
— soit un produit $(e_1 \times e_2)$;
— soit une division $(e_1 \div e_2)$;

avec, à chaque fois, e_1 et e_2 deux expressions arithmétiques. On retrouve ici une définition par récurrence, où on utilise la notion d'expression arithmétique pour définir une expression arithmétique. C'est parce que cette définition est récursive qu'elle se prête bien à la représentation par un arbre :

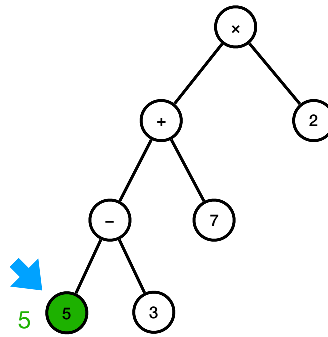
— un nombre n est représenté par une feuille contenant n ;
— une somme $(e_1 + e_2)$ est représentée par un nœud contenant $+$ avec l'arbre binaire de e_1 en enfant gauche et l'arbre binaire de e_2 en enfant droit ;
— et de même pour les trois autres opérations.

Si la calculatrice stocke l'arbre de l'expression dans sa mémoire, comment fait-elle pour l'évaluer, c'est-à-dire pour répondre à l'utilisateur que le résultat du calcul $\big((5-3)+7\big) \times 2$ est 18 ? Elle fait comme nous si nous voulions le faire de tête : elle commence par les calculs « les plus à l'intérieur ». En effet, dans le calcul précédent, la seule chose qu'on peut calculer, sans utiliser de propriétés sur les opérateurs arithmétiques, est le sous-calcul $5-3$, qui vaut 2. On peut donc remplacer dans le calcul originel et obtenir $(2+7) \times 2$. À nouveau, on exécute le sous-calcul le plus à l'intérieur, soit $2+7$, dont on sait que cela vaut 9. On obtient alors 9×2 qui s'évalue en 18.

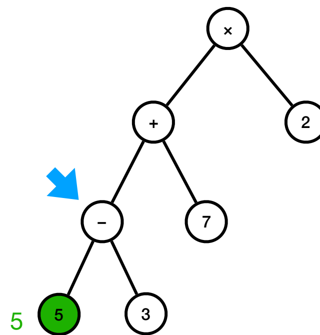
Il se trouve que ce calcul consistant à d'abord évaluer les sous-calculs les plus à l'intérieur est à nouveau un parcours de l'arbre. On commence donc à la racine de l'arbre.



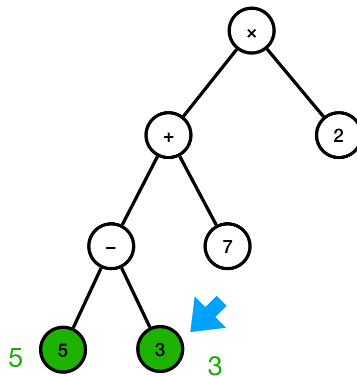
Mais contrairement au parcours préfixe, on affiche rien et on part directement dans l'enfant gauche. Petit à petit, appel récursif après appel récursif, on atterrit donc la feuille 5 qui s'évalue directement en 5 sans avoir besoin d'appels récursifs :



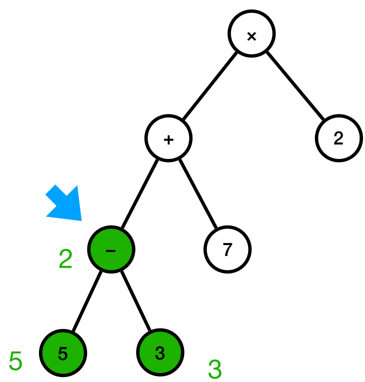
On peut donc remonter dans le nœud parent :



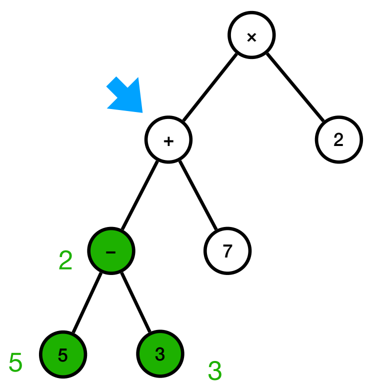
Comme pour le parcours préfixe, on passe ensuite à l'appel récursif sur l'enfant droit :



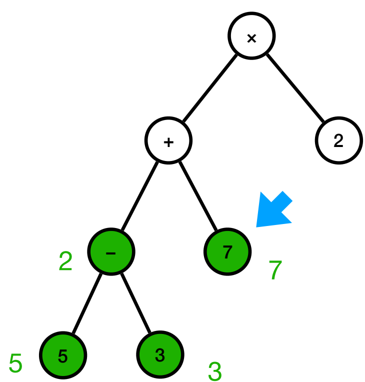
puis on remonte une dernière fois vers le nœud $-$ où on a désormais toutes les informations pour pouvoir évaluer l'opération $5 - 3$:



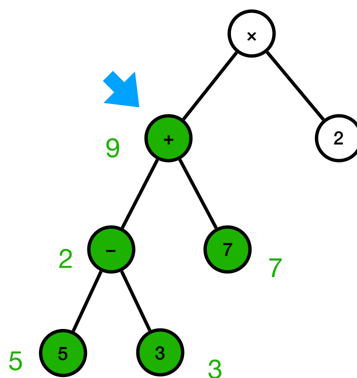
On continue donc à remonter dans l'arbre, sur le nœud + où, à nouveau, on n'a pas encore tous les éléments pour effectuer le calcul :



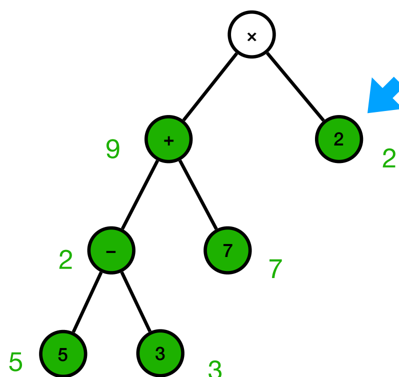
Il faut d'abord parcourir l'enfant droit :



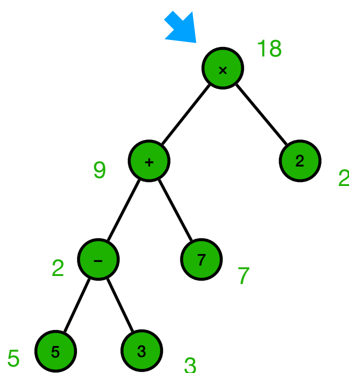
puis revenir pour évaluer le calcul $2 + 7$:



On repart à la racine de l'arbre, pour ensuite aller évaluer son enfant droit :



puis finalement évaluer le calcul 9×2 à la racine :



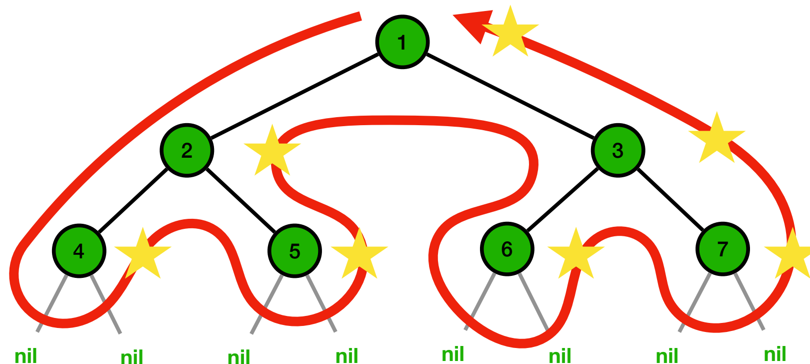
Voici l'opération qu'on a effectué :

```
fonction évaluer_expression(nœud) :
    si nœud est une feuille alors
        retourner valeur[nœud]
    sinon
        m := évaluer_expression(enfant_gauche[nœud])
        n := évaluer_expression(enfant_droit[nœud])
        si valeur[nœud] = '+' alors
            retourner (m + n)
        sinon si valeur[nœud] = '−' alors
            retourner (m − n)
```

```
        sinon si valeur[nœud] = '×' alors
            retourner (m × n)
        sinon    # dans ce cas valeur[nœud] = '÷'
            retourner (m ÷ n)
```

Notons que les appels récursifs sont désormais exécutés avant de traiter plus longuement le nœud (traitement qui consiste en l'occurrence à prendre le résultat de l'enfant gauche et de l'enfant droit et d'exécuter le calcul élémentaire demandé).

En matière d'affichage d'un arbre binaire, l'évaluation d'une expression arithmétique consiste en le parcours suivant :



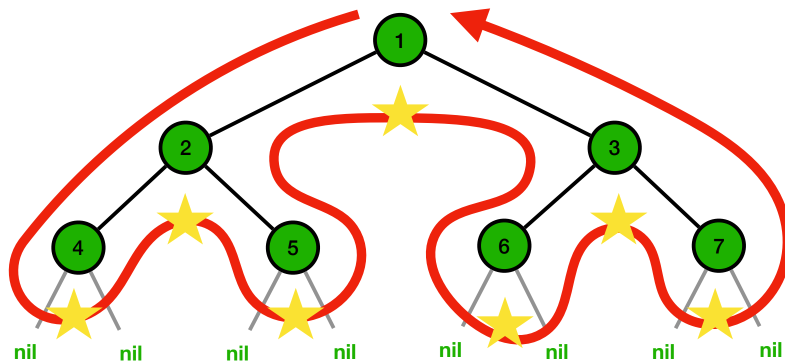
dans lequel on affiche chaque nœud lors du dernier passage par celui-ci, après le traitement de ses enfants gauche et droit. En opposition au parcours préfixe précédent, on appelle ce parcours le *parcours postfixe*. On peut l'écrire de la manière suivante à l'aide d'un algorithme récursif :

```
fonction parcourir_postfixe(nœud) :
    Si nœud ≠ nil alors
        parcourir_postfixe(enfant_gauche[nœud])
        parcourir_postfixe(enfant_droit[nœud])
        afficher(valeur[nœud])
```

Pour l'arbre précédent, le parcours postfixe affiche donc les nœuds dans l'ordre 4, 5, 2, 6, 7, 3 et 1.

4 Parcours infixe : affichage d'une expression

Nous avons vu deux parcours d'arbres différents : le parcours préfixe dans lequel on traite d'abord le nœud avant de traiter son enfant gauche puis son enfant droit, et le parcours postfixe où on commence par traiter les deux enfants avant de traiter le nœud lui-même. On peut aisément imaginer un troisième parcours dans lequel on traite le nœud *entre* le traitement de l'enfant gauche et celui de l'enfant droit : on appelle cela le *parcours infixe*. Il peut se visualiser ainsi



et affiche donc les nœuds dans l'ordre 4, 2, 5, 1, 6, 3 et 7. C'est comme si on avait aplati l'arbre en le passant sous une enclume. L'algorithme récursif de ce parcours est naturellement le suivant :

```
fonction parcourir_infixe(nœud) :
    Si nœud ≠ nil alors
        parcourir_infixe(enfant_gauche[nœud])
        afficher(valeur[nœud])
        parcourir_infixe(enfant_droit[nœud])
```

Les trois parcours ne diffèrent donc que par l'ordre dans lequel on visite la racine, l'enfant gauche et l'enfant droit.

Le parcours infixe est très naturel, en particulier à nouveau dans les calculatrices. Il permet d'afficher à l'écran le calcul (stocké sous forme d'arbre par la calculatrice) sous un format compréhensible par nous. En fait, afficher une expression,

— c'est afficher le nombre n si on est à une feuille ;
— sinon, c'est afficher une parenthèse ouvrante, afficher l'enfant gauche, afficher l'opération contenue dans le nœud courant, puis afficher l'enfant droit et enfin afficher une parenthèse fermante.

5 Arbres binaires de recherche

Pour finir, considérons une dernière application des arbres en informatique. Pour cela, revenons à la question du stockage d'un annuaire téléphonique dans lequel on veut pouvoir chercher facilement un nom (pour connaître le numéro de téléphone qui lui est associé dans l'annuaire). Nous avons proposé une solution sous forme de tableau en début de cours. En comparant deux algorithmes de recherche dans un tableau (et donc de recherche d'un nom dans un annuaire), l'algorithme de recherche séquentielle et l'algorithme de recherche dichotomique dans un tableau trié, nous pouvons déduire que la meilleure implémentation d'un annuaire jusque-là est un tableau trié (qui représente bien l'annuaire physique où les noms sont triés dans l'ordre alphabétique) : l'algorithme de recherche dichotomique permet alors de rechercher un nom dans l'annuaire avec une complexité $O(\log n)$ si l'annuaire contient n noms. Même si on place dans l'annuaire 60 millions d'habitants, le temps de recherche reste minuscule, puisque $\log_2(60\,000\,000) \approx 26$.

Que se passe-t-il si un nouvel abonné arrive en cours d'année, ou si un abonné ne souhaite plus avoir son nom dans l'annuaire ? Avec les bottins imprimés, pas d'autre solution que d'attendre la prochaine édition où la totalité de l'annuaire sera réimprimée. Que peut-on faire

si on utilise un annuaire numérique ? Imaginons un annuaire stocké dans un tableau trié par ordre alphabétique des noms :

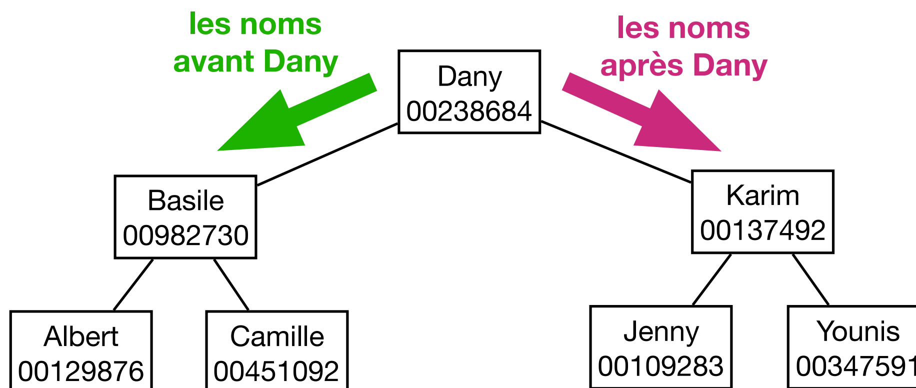
Albert 00129876	Camille 00451092	Dany 00238684	Jenny 00109283	Karim 00137492	Younis 00347591

Basile est un nouvel abonné qu'il nous faut ajouter dans l'annuaire. On peut utiliser la recherche dichotomique pour trouver très rapidement la position dans le tableau où il faut l'insérer (entre Albert et Camille).

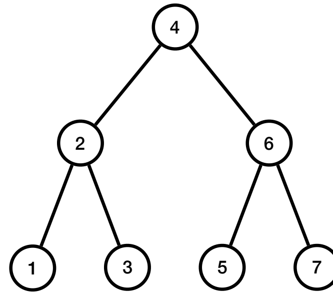
Albert 00129876	Basile 00982730	Camille 00451092	Dany 00238684	Jenny 00109283	Karim 00137492	Younis 00347591

Mais l'insérer dans le tableau nécessite de « déplacer » tous les noms qui le suivent dans l'annuaire : c'est donc très coûteux ! Pour un bottin, cela veut dire qu'il faut refaire toute la mise en page des pages qui suivent celle où on l'a ajouté. Même pour un tableau stocké dans la mémoire d'un ordinateur, cela demande de déplacer d'une case vers la droite le contenu de toutes les cases de la dernière jusqu'à la case qu'on doit libérer pour insérer Basile : cet algorithme a donc complexité $O(n)$ dans le pire des cas (c'est-à-dire quand on doit insérer un nouvel abonné en tête du tableau).

À la place, nous allons proposer une nouvelle structure de données pour stocker un annuaire (ou un dictionnaire) dont l'insertion d'un nouvel élément sera, comme la recherche, de complexité $O(\log n)$. Il s'agit d'un arbre binaire :



Plus précisément, c'est un *arbre binaire de recherche* (ABR). Il s'agit d'un arbre tel que chaque nœud x de l'arbre vérifie la propriété suivante : toutes les valeurs des nœuds dans l'enfant gauche de x sont inférieures à la valeur de x, et toutes les valeurs des nœuds dans l'enfant droit de x sont supérieures à la valeur de x. Sur l'exemple précédent, tous les noms avant Dany dans l'annuaire sont à gauche de la racine, et tous les noms après Dany dans l'annuaire sont à sa droite. Mais aussi, parmi tous les noms à gauche de Dany, ceux avant Basile sont à sa gauche et ceux après Basile sont à sa droite. Voici un autre ABR où les nœuds sont des entiers :



À gauche de la racine se trouve des entiers inférieurs à 4. À droite du nœud 6, ne se trouve que des entiers supérieurs à 6.

Comment rechercher une valeur dans un ABR, comme par exemple un nom dans l'annuaire ? On peut utiliser un algorithme récursif qui *descend* dans l'arbre en partant de la racine et en continuant à gauche ou à droite à l'aide d'une comparaison entre la valeur à chercher et le nœud courant. Notez la proximité entre cet algorithme et la recherche dichotomique qui décidait aussi de continuer dans la moitié gauche ou droite du tableau restant, à l'aide d'une comparaison entre la valeur cherchée et le milieu du tableau : c'est la racine de l'ABR qui joue le rôle du milieu du tableau.

```
fonction est_présent_abr(nœud, x)
    si nœud = nil alors
        retourner Faux
    sinon si x = valeur[nœud] alors
        retourner Vrai
    sinon si x < valeur[nœud] alors
        retourner est_présent_abr(enfant_gauche[nœud], x)
    sinon
        retourner est_présent_abr(enfant_droit[nœud], x)
```

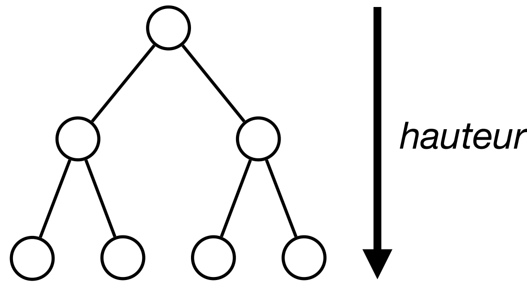
Dans l'ABR précédemment donné en exemple, la recherche de la valeur 3 se déroule comme suit :

- à la racine, on compare 3 et 4 : puisque $3 < 4$, on continue la recherche dans l'enfant gauche, le nœud 2 ;
- puisque $2 < 3$, on continue la recherche dans l'enfant droit ;
- c'est le nœud 3 et la recherche s'arrête donc avec succès.

Si on recherche la valeur 8, voici l'exécution :

- à la racine, on a $4 < 8$, donc on continue la recherche dans l'enfant droit, le nœud 6 ;
- puisque $6 < 8$, on continue à nouveau la recherche à droite, dans le nœud 7 ;
- à nouveau $7 < 8$, on continue donc la recherche à droite ;
- c'est désormais l'arbre vide `nil` : nous n'avons pas trouvé la valeur 8 dans l'arbre et on peut s'arrêter sur cet échec.

Quelle est la complexité de cet algorithme ? À chaque appel récursif, on plonge dans l'enfant gauche ou l'enfant droit, donc on descend d'un niveau à chaque appel. Ainsi, les appels récursifs suivent une branche (un chemin de la racine à l'une des feuilles de l'arbre) de l'arbre et la complexité est donc proportionnelle à la longueur maximale d'une de ses branches. Par analogie avec nos dessins, on appelle *hauteur* cette longueur maximale d'une branche de l'arbre. Par exemple, l'ABR suivant a hauteur 2, de sorte qu'un arbre réduit à sa racine aura hauteur 0 par convention :



Comment la hauteur d'un ABR se compare-t-elle au nombre n de nœuds que l'arbre contient ? Malheureusement pas toujours très bien : la hauteur peut être de l'ordre de n dans le pire des cas. Mais si l'ABR est *équilibré* (comme c'est le cas au-dessus), c'est-à-dire qu'on essaie de conserver autant de nœuds dans l'enfant gauche et l'enfant droit de n'importe quel nœud de l'arbre, alors la hauteur devient proportionnelle à $\log_2 n$. Dans ce cas, la complexité de l'algorithme de recherche dans un ABR a complexité $O(\log n)$, comme pour la recherche dichotomique.

Attelons-nous finalement au problème initial : l'ajout de nouveaux abonnés dans l'annuaire ! Cela correspond à l'insertion d'une nouvelle valeur dans un ABR. En fait, nous n'avons pas beaucoup à faire, le principal étant de savoir chercher la position où devrait se trouver l'élément à insérer. De deux choses l'une :

— soit on trouve l'élément à insérer : dans ce cas, on renvoie une erreur puisqu'on ne souhaite pas avoir deux occurrences de la même valeur dans l'ABR ;

— soit on ne le trouve pas, ce qui veut dire que l'algorithme de recherche s'est retrouvé en-dessous d'une feuille, dans un arbre vide *nil* (comme lorsqu'on a recherché 8 dans l'ABR précédemment) : il ne reste plus alors qu'à remplacer cet arbre vide par une nouvelle feuille avec la valeur à insérer. On obtient bien un nouvel ABR avec tous les nœuds de l'ABR initial auquel s'ajoute la valeur à insérer.

L'algorithme se résumant finalement à une recherche dans l'ABR, sa complexité est du même ordre que la complexité de la recherche : $O(n)$ dans le pire des cas, et $O(\log n)$ dans le cas d'un ABR équilibré.