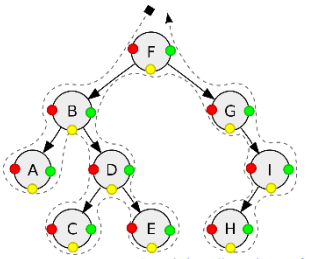
Cours – Algorithmes sur les arbres

Objectifs :
- ⇨ Découvrir les algorithmes de parcours d'arbres
- ⇨ Voir les fonctions de recherche et d'insertion dans un ABR



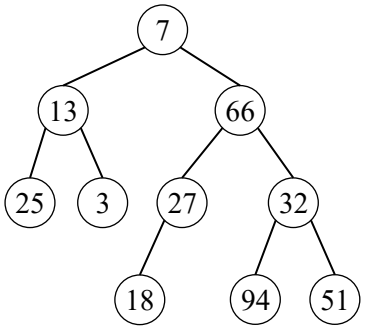
D'après https://www.lyceum.fr

I – Parcours d'un arbre

Nous prendrons ici le cas des arbres binaires (pas nécessairement de recherche), que l'on pourra ensuite généraliser aux arbres ayant un nombre quelconque de fils par nœud.
Dans le chapitre de présentation des arbres binaires, on a déjà parcouru des arbres avec les fonctions `taille` et `hauteur`, mais dans ce cas l'ordre de parcours n'avait pas d'importance.
Si on décide par exemple d'afficher une à une les valeurs de tous les nœuds, alors l'ordre dans lequel on examine les nœuds a une importance.

Application 1 :
Dans l'arbre ci-contre, on imagine 4 parcours différents qui donneront les affichages suivants :
Algorithme 1 : 7, 13, 25, 3, 66, 27, 18, 32, 94, 51
Algorithme 2 : 25, 13, 3, 7, 18, 27, 66, 94, 32, 51
Algorithme 3 : 25, 3, 13, 18, 27, 94, 51, 32, 66, 7
Algorithme 4 : 7, 13, 66, 25, 3, 27, 32, 18, 94, 51

Pour chacun de ces parcours écrivez en pseudo-code l'algorithme de parcours qui permet d'obtenir cet affichage (et qui fonctionnerait pour un arbre quelconque).



1) Parcours en largeur et en profondeur

On parle de **parcours en largeur** lorsqu'on examine tous les nœuds d'un même niveau successivement avant de passer au niveau suivant.
Les **parcours en profondeur** correspondent aux parcours où on va d'abord chercher à descendre le plus bas possible dans l'arbre avant de remonter pour explorer tous les chemins.

Application 2 :
Parmi les parcours envisagés à l'application 1, le(s)quel(s) est(sont) un(des) parcours en profondeur et le(s)quel(s) est(sont) un(des) parcours en largeur ?

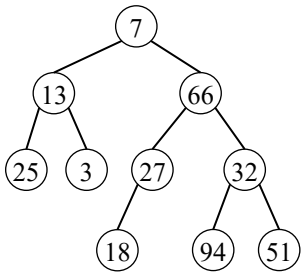
Il existe 3 types de parcours en profondeur : infixe, prefixe et postfixe

- Parcours **infixe** :
  - ⇒ On parcourt le sous-arbre gauche
  - ⇒ On traite le nœud (on affiche sa valeur par exemple)
  - ⇒ On parcourt le sous-arbre droit

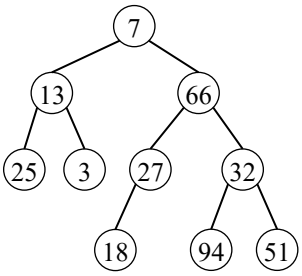
Mais le moment où on traite le nœud peut être avant ou après le parcours des sous-arbres :

- Parcours **préfixe** :
  - ⇒ On traite le nœud
  - ⇒ On parcourt le sous-arbre gauche
  - ⇒ On parcourt le sous-arbre droit
- Parcours **postfixe** :
  - ⇒ On parcourt le sous-arbre gauche
  - ⇒ On parcourt le sous-arbre droit
  - ⇒ On traite le nœud

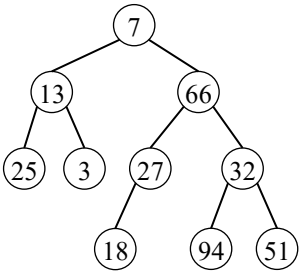
Application 3 :
1) Affecter à chaque parcours de l'application 1 le nom correct (infixe, préfixe, postfixe).
2) Sur les figures ci-dessous, tracer le parcours suivi dans chaque cas. On fera un trait qui passe sur le nœud à chaque fois que l'algorithme le traite ou à côté si le nœud est juste traversé.



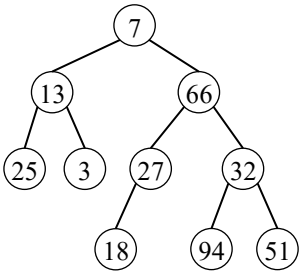
Parcours infixe



Parcours préfixe

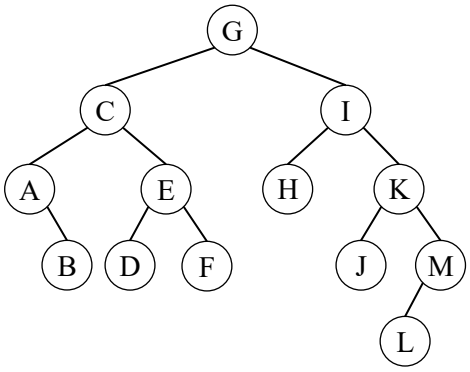


Parcours postfixe



Parcours en largeur

Application 4 :
Pour l'arbre représenté ci-contre, donner :
a) le parcours infixe
b) le parcours préfixe
c) le parcours postfixe
d) le parcours en largeur



Application 5 :
En s'appuyant sur la classe `Noeud` (dans le fichier « Noeud.py ») et la classe File (dans le fichier « Files.py »), programmer les 4 parcours dans 4 fonctions (parcours_infixe, parcours_prefixe, parcours_postfixe et parcours_en_largeur) qui affichent les nœuds visités successifs.
En utilisant l'arbre de l'application 1 (`arbre1`) ou celui de l'application 4 (`arbre4`) donnés plus bas, vous pourrez vérifier que vos fonctions donnent bien le résultat attendu.

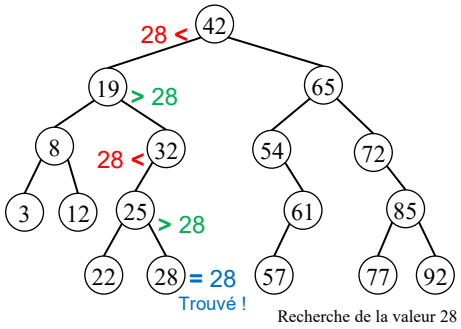
```
arbre1 = Noeud(Noeud(Noeud(None,25,None),13,Noeud(None,3,None)),7,
               Noeud(Noeud(Noeud(None,18,None),27,None),66,
                     Noeud(Noeud(None,94,None),32,Noeud(None,51,None))))
```

```
arbre4 = Noeud(Noeud(Noeud(None,'A',Noeud(None,'B',None)),'C',
                 Noeud(Noeud(None,'D',None),'E',Noeud(None,'F',None))),'G',
             Noeud(Noeud(None,'H',None),'I',Noeud(Noeud(None,'J',None),'K',
                                  Noeud(Noeud(None,'L',None),'M',None))))
```

II - Recherche et insertion dans un ABR

Dans cette partie on va voir les algorithmes qui font tout l'intérêt des arbres binaires de recherche (ABR).

1) Recherche dans un ABR



Recherche de la valeur 28

La recherche dans un ABR est assez simple et ressemble fortement à la recherche dichotomique vue en classe de première :

1. La valeur recherchée est égale à celle du nœud courant ? Si oui, on a trouvé
2. Si elle est plus petite et qu'il existe un fils gauche, on s'y déplace et on reprend au 1. S'il n'y a pas de fils gauche et que la valeur est plus petite c'est qu'elle n'est pas dans l'arbre.
3. Si elle est plus grande et qu'il existe un fils droit, on s'y déplace et on reprend au 1. S'il n'y a pas de fils droit et que la valeur est plus grande c'est qu'elle n'est pas dans l'arbre.

Application 6 :

1) Ecrire une fonction `recherche_ABR(n, val)` qui recherche la valeur `val` dans l'ABR dont on donne le nœud racine `n`. La fonction renvoie le nœud contenant la valeur ou bien None si elle n'est pas dans l'arbre de recherche.

Vous pouvez écrire une fonction récursive ou itérative (les deux sont de difficulté équivalente). Dans un premier temps, vous pouvez afficher la valeur du nœud examiné à chaque étape pour vérifier que votre algorithme fonctionne bien. Vous pouvez aussi utiliser l'arbre vu plus haut dont la définition en python est donnée juste en dessous :

```
ABR1 = Noeud(Noeud(Noeud(Noeud(None,3,None),8,Noeud(None,12,None)),19,
       Noeud(Noeud(Noeud(None,22,None),25,Noeud(None,28,None)),32,None)),42,
       Noeud(Noeud(None,54,Noeud(Noeud(None,57,None),61,None)),
       65,Noeud(None,72,Noeud(Noeud(None,77,None),85,Noeud(None,92,None)))))
```

Pour les plus rapides, vous pouvez écrire l'autre version (itérative ou récursive) également.

2) Quelle est la complexité de cette fonction ? Justifier.

2) Insertion dans un ABR

L'insertion d'une valeur dans un ABR se fait exactement à l'endroit où la valeur aurait dû se trouver quand on la cherche. Il suffit alors de remplacer le fils gauche ou droit manquant par un nœud contenant la valeur à insérer (et aucun fils gauche ou droit : c'est une feuille).

Dans le cas où la valeur est déjà présente dans l'arbre, on rajoute la valeur dans le sous-arbre gauche du nœud où se trouve la valeur.

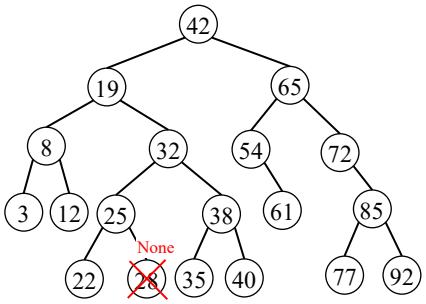
Application 7 :

1) En vous inspirant fortement de la fonction précédente, écrire une fonction `insertion_ABR(n, val)` qui insère la valeur `val` dans l'arbre binaire de recherche dont on donne le nœud racine `n`.
2) Quelle est la complexité de cette fonction ? Justifier.
3) Ecrire alors une fonction `insertions_ABR(n, t)` qui insère toutes les valeurs contenues dans le tableau `t` dans l'ABR dont on donne le nœud racine `n`.

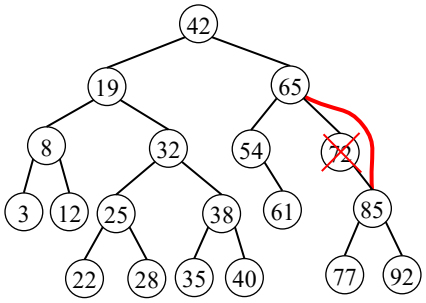
3) Suppression dans un ABR (hors programme)

La suppression dans un ABR n'est pas au programme de terminale NSI, mais on peut tout de même en présenter le principe :

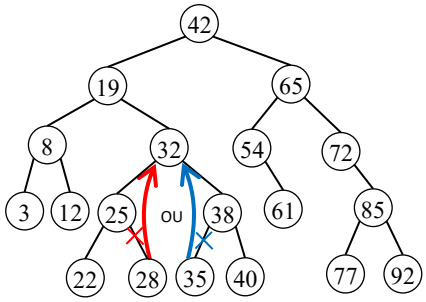
- On recherche la valeur à supprimer dans l'arbre (voir algorithme de recherche)
- Si le nœud contenant la valeur est une feuille, on le supprime en faisant pointer son parent sur None.
- Si le nœud contenant la valeur a un seul fils, on le supprime en faisant pointer son parent vers son unique fils.
- Si le nœud contenant la valeur a deux fils, on doit remplacer sa valeur soit par le maximum de son sous-arbre gauche, soit par le minimum de son sous-arbre droit (et supprimer ce maximum ou minimum).



Suppression de la valeur 28



Suppression de la valeur 72



Suppression de la valeur 32

4) Complexité et équilibrage des arbres

On a vu que la complexité de la recherche est proportionnelle à la hauteur de l'arbre (profondeur à laquelle il faut descendre pour avoir testé toutes les possibilités). Si l'arbre est dégénéré (chaque nœud a au plus 1 fils), sa hauteur est égale à sa taille n et on a donc une recherche en temps linéaire ce qui n'est pas mieux que dans un tableau.
En revanche si l'arbre est parfait on a $h = \log_2(n)$ et la recherche se fait donc en temps logarithmique.
D'une manière générale, si l'arbre est équilibré sa hauteur est alors quasiment égale à $\log_2(n)$ (à une unité près) et la complexité de la recherche est alors logarithmique donc efficace.
Pour que les algorithmes sur les ABR soient efficaces, il faut donc s'assurer que l'arbre est bien équilibré. Cela peut se faire en optimisant les algorithmes d'insertion et de suppression de manière à ce que ces opérations maintiennent l'arbre équilibré. Il existe de nombreuses façons de maintenir les arbres équilibrés (mais aucune n'est au programme de NSI). Les plus connues sont les arbres AVL et les arbres Rouges-Noirs.

Références :

Recherche sur les Arbres binaires : https://www.lri.fr/~fiorenzi/Teaching/AL/C4.pdf
Arbres binaires de recherche : http://pageperso.lif.univ-mrs.fr/~francois.denis/algoMPCI/chap1.pdf
Parcours d'un arbre binaire : http://math.univ-lyon1.fr/irem/IMG/pdf/parcours_arbre_avec_solutions-2.pdf
Arbres AVL (pages 1 à 19) : https://www.lri.fr/~fiorenzi/Teaching/AL/C5.pdf
Arbres AVL (implémentation en Java) : https://miashs-www.u-ga.fr/prevert/Prog/Java/CoursJava/arbresAVL.html
Arbres Rouges-Noirs : https://slides.com/rolandin0/algorithmique-avancee-les-arbres-rouges-noirs

Exercices – **Algorithmes sur les arbres**

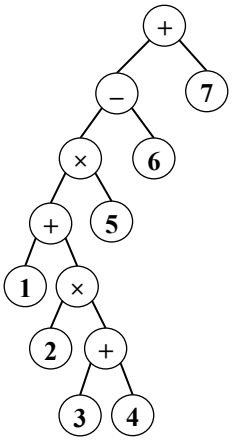
Exercice 1 : Différents parcours, différents arbres

Donner un arbre binaire de hauteur aussi petite que possible dont le parcours infixe serait : 4, 23, 9, 13, 45, 24, 37. Même question pour un parcours préfixe, postfixe et en largeur.

Exercice 2 : Parcours croissant

Quel est le parcours qui permet de lister les éléments d'un ABR dans l'ordre croissant ?

Exercice 3 : Retour sur la notation polonaise inverse (RPN)



L'arbre binaire ci-contre représente un calcul.

1) En rajoutant des parenthèses autour de chaque groupe « opérateur et opérandes », donner l'écriture du calcul correspondant.
2) Un des parcours (préfixe, infixe, suffixe) correspond à notre notation usuelle, un autre à la notation polonaise inversée. Identifier ces parcours.
3) Construire l'arbre correspondant au calcul en RPN : 2 4 3 × + 7 × 8 + 2 3 5 + × -

Exercice 4 : Classe ABR

Le fichier « ABR.py » contient la classe `ABR` qui reprend les fonctions écrites pendant le cours en les implémentant sous forme de méthodes de la classe `ABR`. Analyser le code de ce fichier et noter les différences avec la programmation sous forme de fonctions.
Comment les fonctions récursives `parcours_infixe`, `parcours_prefixe` et `parcours_postfixe` ont-elles été adaptées sous forme de méthodes ?

Exercice 5 : Ordre d'insertion

1) Lorsqu'on crée un ABR par insertion de valeurs successives à partir d'un arbre vide, l'ordre dans lequel on insère les valeurs a-t-il une importance ? Justifier.
2) Peut-on obtenir le même arbre en insérant les mêmes valeurs mais dans un ordre différent ? Donner un (contre) exemple.
3) Dans quel ordre doit-on insérer les valeurs pour obtenir l'arbre étudié dans la partie II du cours ?

Exercice 6 : Tri arborescent

1) Ecrire une fonction `tableau(n)` qui prend en argument le nœud racine `n` d'un arbre binaire et renvoie un tableau contenant toutes les étiquettes des nœuds de l'arbre dans l'ordre infixe.
2) En utilisant la fonction précédente, écrire une fonction `tri(t)` qui prend en argument un tableau t et renvoie un tableau trié contenant les mêmes éléments. On pourra utiliser la classe `ABR` du fichier « ABR.py » pour ne pas avoir à réécrire trop de code. Quel est la complexité de cet algorithme ?