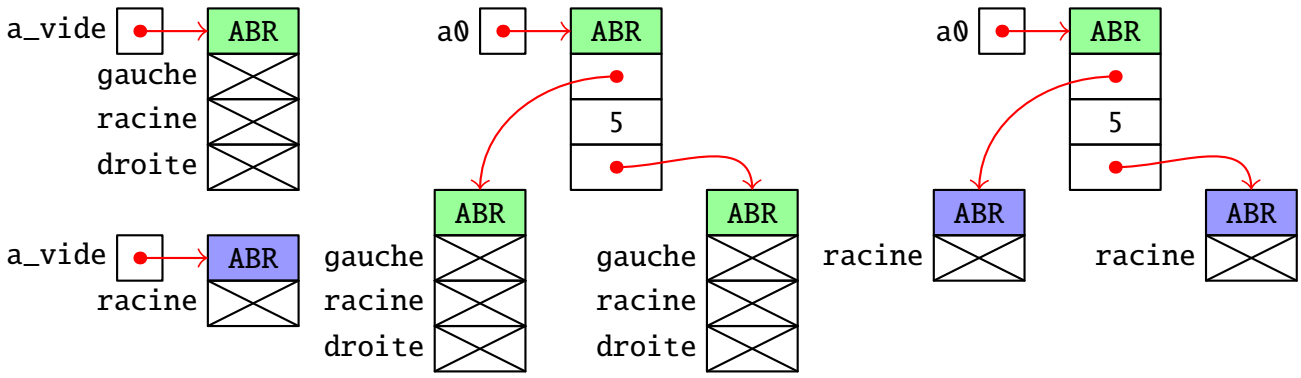
Python – Arbres binaires de recherche

Avoir la classe

Afin de représenter les ABR, nous allons utiliser une classe. Contrairement aux listes chaînées où nous avions utilisées 2 classes, nous ferons tout avec une seule. Cela pose le problème de l'arbre vide. En effet, si l'objet est égal à **None**, alors on ne peut pas utiliser les méthodes de la classe. Pour contourner ce problème, on peut soit faire une fonction `est_vide(arbre)` en dehors de la classe, soit utiliser quelque chose dans les attributs de la classe pour indiquer qu'il est vide.

```
class ABR:
    def __init__(self, gauche=None, racine=None, droite=None):
        self.racine = racine
        self.gauche = gauche
        self.droite = droite

    def est_vide(self):
        return self.racine is None
```

L'arbre vide est donc représenté par une racine égale à **None**. Ainsi, il suffit de faire `ABR()` pour créer un arbre vide.
Voici la représentation d'un arbre vide et d'un arbre ne contenant qu'un seul nœud. Pour simplifier la représentation des arbres vides, on pourra ne montrer que l'attribut `racine`.



Pour afficher les arbres, nous allons reprendre la méthode d'affichage des arbres binaires. Il faut rajouter la première fonction hors de la classe et la méthode `affichage` dans la classe `ABR`.

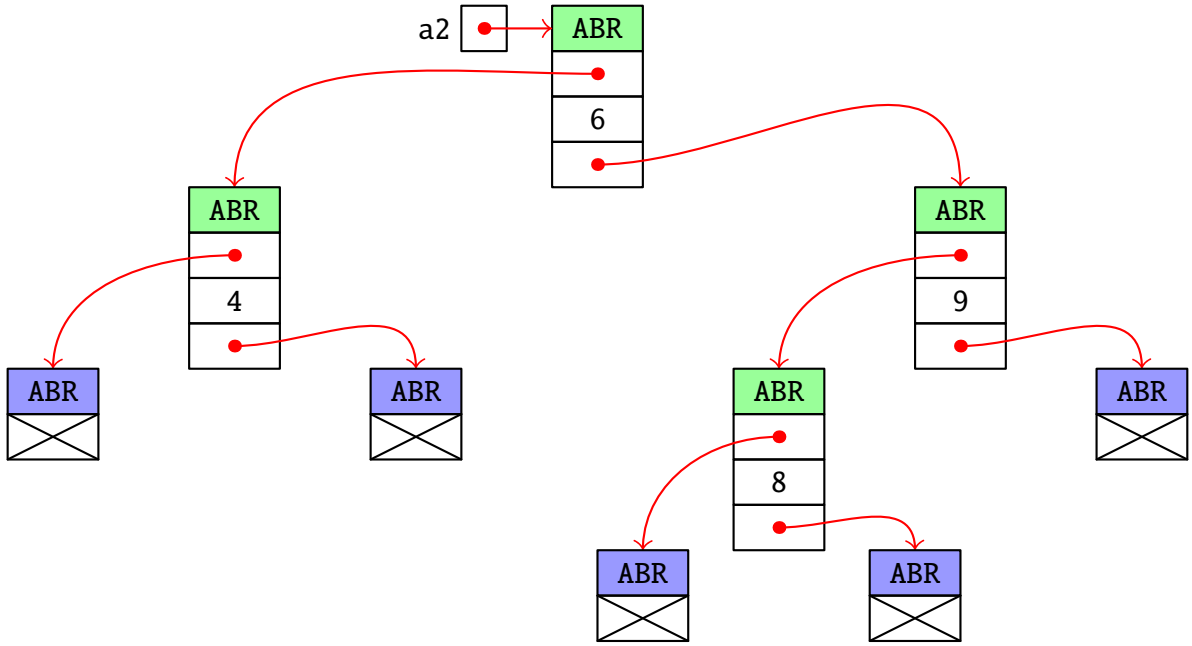
```
def faire_prefixe(instr):
    res = ""
    for i in range(1, len(instr)):
        if instr[i-1] != instr[i]:
            res += "\u2502 "
        else:
            res += "  "      # Bien mettre 2 espaces
    if len(instr) > 0:
        if instr[-1] == "g":
            res += "\u2514\u2500"
        else:
```

```
            res += "\u250C\u2500"
    return res

class ABR:
    ...

    def affichage(self, instr=""):
        if not self.est_vide():
            self.droite.affichage(instr + "d")
            print(faire_prefixe(instr) + str(self.racine))
            self.gauche.affichage(instr + "g")
```

```
>>> a1 = ABR(ABR(ABR(), 1, ABR()), 5, ABR(ABR(), 7, ABR()))
>>> a1.affichage()
┌─7
5
└─1
```

Exercice 1 (sur papier) : Représenter les objets correspondant à l'arbre a1.

Exercice 2 : Écrire l'expression correspondant à l'arbre a2.



Si les ajouts se font manuellement, comme c'est le cas au dessus, il se peut que l'arbre produit ne soit pas un ABR. Il faut donc rajouter une méthode permettant de vérifier que l'arbre est bien un ABR.

Exploitation

Exercice 3 : Écrire la méthode `hauteur(self)` qui renvoie la hauteur de l'arbre. La hauteur de l'arbre vide est 0.

```
>>> a1.hauteur()
2
```

Exercice 4 : Écrire la méthode `__len__(self)` qui renvoie le nombre de nœuds contenus dans l'arbre.

```
>>> len(a1)
3
```

Exercice 5 : Écrire la méthode `contient(self, val)` qui renvoie un booléen indiquant si `val` est dans `self`. Il ne faut pas explorer l'intégralité de l'arbre, mais juste prendre le chemin menant directement à la valeur cherchée.

```
>>> a1.contient(5)
True
>>> a1.contient(1)
True
>>> a1.contient(3)
False
```

Exercice 6 : Écrire la méthode `minimum(self)` qui renvoie le minimum contenu dans l'arbre. Si l'arbre est vide, il faut lever une exception.

```
>>> a1.minimum()
1
>>> a2.minimum()
4
```

Ajout d'un élément

Comme nous l'avons vu, créer un ABR à la main n'est pas si aisé, ni intéressant. Nous allons donc écrire une méthode permettant un tel ajout et garantissant que l'arbre modifié reste un ABR. Pour cela, nous pourrons utiliser la méthode suivante qui permet de transformer l'arbre vide en feuille :

```
class ABR:
    ...

    def mettre_valeur(self, val):
        self.racine = val
        self.gauche = ABR()
        self.droite = ABR()
```

```
>>> a3 = ABR()
>>> a3.mettre_valeur(7)
>>> a3.gauche.mettre_valeur(2)
>>> a3.droite.mettre_valeur(8)
>>> a3.affichage()
```

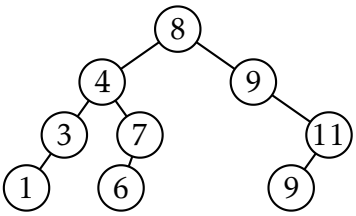
Exercice 7 : Écrire la méthode `insere(self, val)` qui place `val` au bon endroit dans l'ABR. Si la valeur se trouve déjà dans l'arbre, la nouvelle est insérée dans le descendant droit.

```
>>> a4 = ABR()
>>> a4.insere(6)
>>> a4.insere(3)
>>> a4.insere(8)
>>> a4.insere(7)
>>> a4.affichage()
┌─8
│ └─7
6
└─3
```

Exercice 8 : Écrire la fonction `creer_abr(liste)` qui renvoie un nouvel ABR où les valeurs de la liste sont insérées dans l'ordre.

```
>>> a5 = creer_abr([7,1,4,6,3,9])
>>> a5.affichage()
┌─9
7
│   ┌─6
│ ┌─4
│ │ └─3
└─1
```

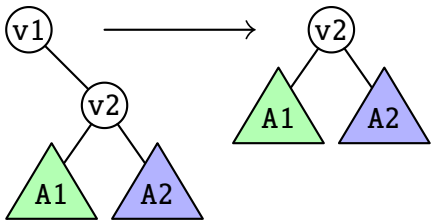
Exercice 9 (sur papier) : Trouver dans quel ordre mettre les valeurs dans la liste pour obtenir l'arbre suivant avec la commande `creer_abr(liste)`.



Suppression

La suppression d'un élément dans un ABR n'est pas au programme de terminale, mais cela ne veut pas dire que c'est un problème si complexe que cela.
Pour cela, il faut pouvoir supprimer le minimum d'un arbre non vide. Il y a deux cas :

- Soit le minimum de l'arbre est une feuille, et dans ce cas, il suffit de mettre ses 3 attributs à `None`.
- Soit le minimum est sur un nœud qui a un descendant droit non vide, et dans ce cas, il suffit de le remplacer par ce descendant, comme indiqué ci-contre.



Exercice 10 (sur papier) :

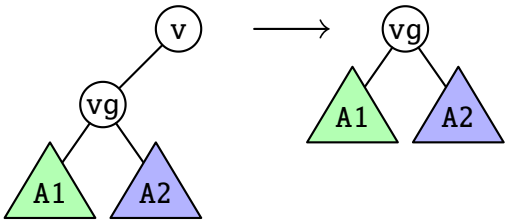
1) Comment savoir que la racine de l'arbre considéré est le minimum ?
2) On sait que la racine de l'arbre considéré est le minimum. Comment savoir si cet arbre est une feuille ou pas ?

Exercice 11 : Écrire la méthode `supprimer_minimum(self)` qui renvoie le minimum de l'arbre, tout en le supprimant. On supposera que l'arbre n'est pas vide.

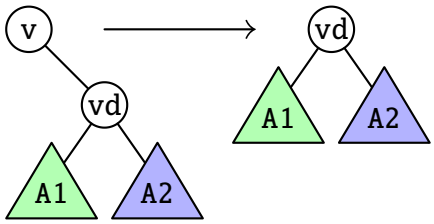
```
>>> a = creer_abr([6, 2, 8, 3, 7])
>>> a.affichage()
┌─8
│ └─7
6
│ ┌─3
└─2
>>> a.supprime_minimum()
2
>>> a.affichage()
┌─8
│ └─7
6
└─3
```

Pour supprimer une valeur dans l'arbre, il y a 4 cas :

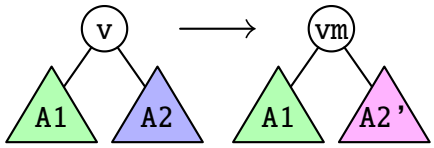
- La valeur est sur une feuille et dans ce cas, il suffit de mettre les 3 attributs à None.
- La valeur est sur un nœud qui n'a qu'un descendant à gauche, et il suffit de le remplacer par son descendant gauche.



- La valeur est sur un nœud qui n'a qu'un descendant à droite, et il suffit de le remplacer par son descendant droite.



- Sinon, on enlève le minimum du descendant droit et on le met à la place de la valeur à supprimer dans la racine. Dans l'exemple `vm` correspond au minimum de l'arbre `A2` et `A2'` est le même arbre auquel on a enlevé `vm`.



Exercice 12 : Écrire la méthode `supprime(self, val)` qui supprime une occurence de `val` dans l'arbre. On suppose que `val` est bien contenu dans l'arbre.

```
>>> a = creer_abr([6, 2, 8, 3, 7])
>>> a.supprime(6)
>>> a.affichage()
┌─8
7
│ ┌─3
```

```
└─2
>>> a.supprime(2)
>>> a.affichage()
┌─8
7
└─3
```

Pour aller plus loin

L'arbre obtenu après l'ajout de plusieurs valeurs peut être totalement déséquilibré, rendant les recherches moins efficaces. Il existe des versions spéciales des ABR qui essaient de maintenir un certain équilibre. Pour cela, la structure de l'arbre peut être modifiée à chaque ajout ou suppression.
Les plus connus sont les arbres AVL, du nom de leurs créateurs Georgii Adelson-Velsky et Evguenii Landis qui ont inventé ces arbres en 1962. Lors d'un ajout, on vérifie que la différence de hauteur entre deux sous-arbres n'est pas supérieure à 2. Si c'est le cas, l'arbre est rééquilibré.
Il existe aussi, entre autres, les arbres bi-colores, ou arbres rouge-noir, inventé par Rudolf Bayer en 1972. Là, le principe est plutôt d'identifier des nœuds à côté desquels il est possible de rajouter de nouvelles valeurs sans avoir besoin de rééquilibrer l'arbre.
Vous pouvez voir ces arbres en action à l'aide de ces simulateurs en ligne :
https://www.cs.usfca.edu/ galles/visualization/AVLtree.html
https://www.cs.usfca.edu/ galles/visualization/RedBlack.html
Les deux arbres reposent sur le principe de la rotation simple ou double qui permettent de rééquilibrer un arbre.