Cet exercice porte sur les arbres binaires, les files et la programmation orientée objet. Cet exercice comporte deux parties indépendantes.

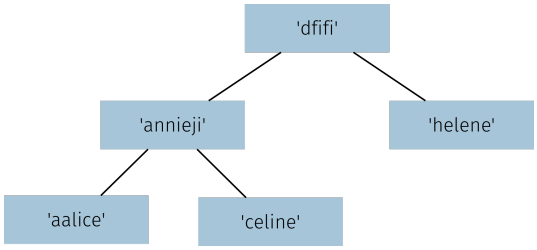
Partie 1

Une entreprise stocke les identifiants de ses clients dans un arbre binaire de recherche.
On rappelle qu'un arbre binaire est composé de nœuds, chacun des nœuds possédant éventuellement un sous-arbre gauche et éventuellement un sous-arbre droit.
La taille d'un arbre est le nombre de nœuds qu'il contient. Sa hauteur est le nombre de nœuds du plus long chemin qui joint le nœud racine à l'une des feuilles. On convient que la hauteur d'un arbre ne contenant qu'un nœud vaut 1 et celle de l'arbre vide vaut 0.
Dans cet arbre binaire de recherche, chaque nœud contient une valeur, ici une chaîne de caractères, qui est, avec l'ordre lexicographique (celui du dictionnaire) :

– strictement supérieure à toutes les valeurs des nœuds du sous-arbre gauche ;

– strictement inférieure à toutes les valeurs des nœuds du sous-arbre droit.

Ainsi les valeurs de cet arbre sont toutes distinctes. On considère l'arbre binaire de recherche suivant :



1. Donner sa taille et sa hauteur.

2. Recopier ci-dessous cet arbre après l'ajout des identifiants suivants : `'davidbg'` et `'papicoeur'` dans cet ordre.

3. On décide de parcourir cet arbre pour obtenir la liste des identifiants dans l'ordre lexicographique. Quel parcours doit-on utiliser ?

Pour traiter informatiquement les arbres binaires, nous allons utiliser une classe `ABR` .
Un arbre binaire de recherche, nommé `abr` dispose des méthodes suivantes :

- `abr.est_vide()` : renvoie `True` si `abr` est vide et `False` sinon.
- `abr.racine()` : renvoie l'élément situé à la racine de `abr` si `abr` n'est pas vide et `None` sinon.
- `abr.sg()` : renvoie le sous-arbre gauche de `abr` s'il existe et `None` sinon.
- `abr.sd()` : renvoie le sous-arbre droit de `abr` s'il existe et None sinon.

On a commencé à écrire une méthode récursive `present` de la classe `ABR`, où le paramètre `identifiant` est une chaîne de caractères et qui renvoie `True` si `identifiant` est dans l'arbre et `False` sinon.

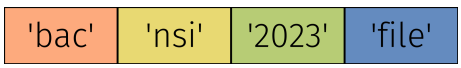
4. Compléter ce code

Python

```
def present(self, identifiant):
    if self.est_vide():
        return False
    elif self.racine() == identifiant:
        return ...
    elif self.racine() < identifiant:
        return self.sd(). ...
    else:
        return ...
```

Partie 2

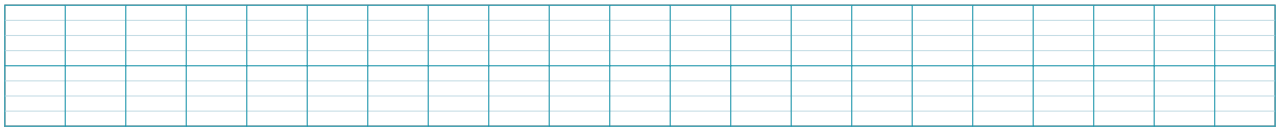
On considère une structure de données file que l'on représentera par des éléments en ligne, l'élément à droite étant la tête de la file et l'élément à gauche étant la queue de la file. On appellera `f1` la file suivante :



On suppose que les quatre fonctions suivantes ont été programmées préalablement en langage Python :

- `creer_file()` : renvoie une file vide ;
- `est_vide(f)` : renvoie `True` si la file `f` est vide et `False` sinon ;
- `enfiler(f, e)` : ajoute l'élément `e` à la queue de la file `f` ;
- `defiler(f)` : renvoie l'élément situé à la tête de la file `f` et le retire de la file.

5. Donner le résultat renvoyé après l'appel de la fonction `est_vide(f1)`.



6. Représenter la file `f1` après l'exécution du code `defiler(f1)`.

7. Représenter la file `f2` après l'exécution du code suivant :

Python

```
f2 = creer_file()
liste = ['castor', 'python', 'poule']
for elt in liste:
    enfiler(f2, elt)
```

8. Compléter la fonction `longueur` qui prend en paramètre une file `f` et qui renvoie le nombre d'éléments qu'elle contient.
Après un appel à la fonction, la file `f` doit retrouver son état d'origine.

Python

```
def longueur(f):
    resultat = 0
    g = creer_file()
    while ... :
        elt = defiler(f)
```

```
        resultat = ...
        enfiler(... , ...)
    while not(est_vide(g)):
        enfiler(f, defiler(g))
    return resultat
```

Un site impose à ses clients des critères sur leur mot de passe. Pour cela il utilise la fonction `est_valide` qui prend en paramètre une chaîne de caractères `mot` et qui retourne `True` si mot correspond aux critères et `False` sinon.

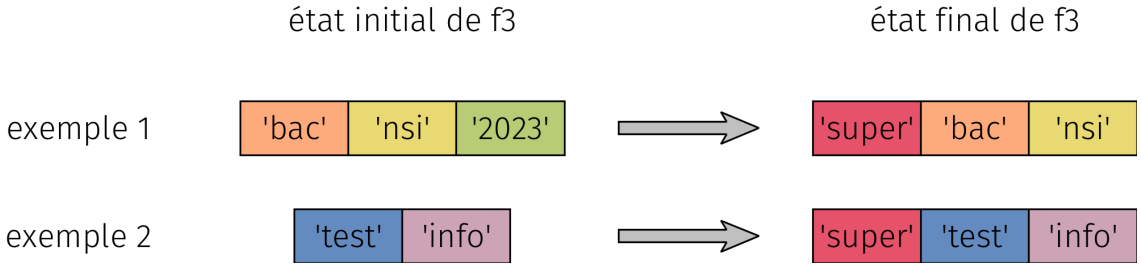
Python

```
def est_valide(mot):
    if len(mot) < 8:
        return False
    for c in mot:
        if c in ['!', '#', '@', ';', ':']:
        return True
    return False
```

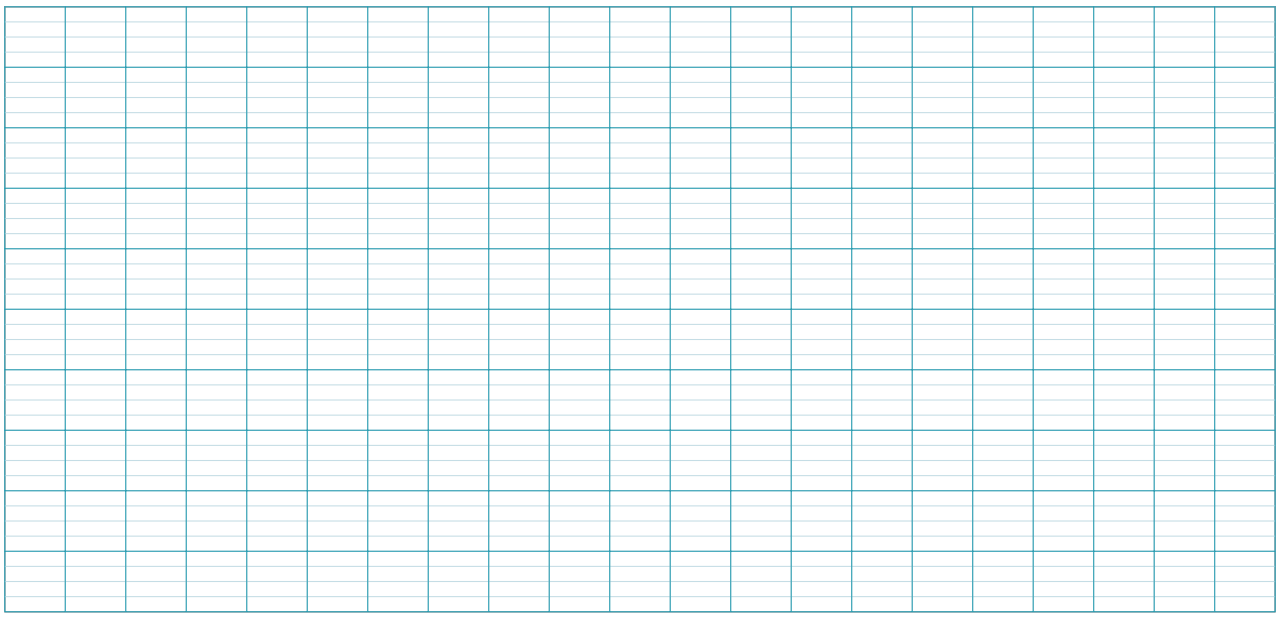
9. Entourer le ou les mots validés par cette fonction.

- `'best@'`
- 'paptap23'
- '2!@59fgds'

La figure suivante montre, sur deux exemples, l'évolution d'une file `f3` après l'exécution de l'instruction `ajouter_mot(f3, 'super')` :



10. Écrire le code de cette fonction `ajouter_mot` qui prend en paramètres une file `f` (qui a au plus 3 éléments) et une chaîne de caractères valide `mdp` . Cette fonction met à jour la file de stockage `f` des mots de passe en y ajoutant `mdp` et en défilant, si nécessaire, pour avoir au maximum trois éléments dans cette file.
On pourra utiliser la fonction `longueur` définie précédemment.



Pour intensifier sa sécurité, le site stocke les trois derniers mots de passe dans une file et interdit au client lorsqu'il change son mot de passe d'utiliser l'un des mots de passe stockés dans cette file.

11. Compléter la fonction `mot_file` :

- qui prend en paramètres une file `f` et `mdp` de type chaîne de caractères ;
- qui renvoie `True` si le mot de passe est un élément de la file `f` et `False` sinon.

Après un appel à cette fonction, la file `f` doit retrouver son état d'origine.

Python

```
def mot_file(f, mdp):
    g = creer_file()
    present = False
    while not(est_vide(f)):
        elt = defiler(f)
        enfiler(g, elt)
        if ...:
            present = ...
    while not(est_vide(g)):
        enfiler(f, defiler(g))
    return present
```

12. Écrire une fonction `modification` qui prend en paramètres une file `f` et une chaîne de caractères `nv_mdp`. Si le mot de passe `nv_mdp` répond bien aux deux exigences des questions **9.** et **11.** , alors elle modifie la file des mots de passe stockés et renvoie `True` . Dans le cas contraire, elle renvoie `False`.
On pourra utiliser les fonctions `mot_file` , `est_valide` et `ajouter_mot`.