Lycée Les 3 Sources **Bourg-Lès-Valence**
NSI T^le^ Année **2024-25**

Devoir surveillé n°5

Nom et prénom :

Exercice 1 : (11pt) *Cet exercice traite du thème architecture matérielle, et plus particulièrement des processus et leur ordonnancement.*

1) Avec la commande `ps -aef` on obtient l'affichage suivant :

```
PID  PPID  C STIME TTY       TIME CMD
8600    2  0 17:38 ?     00:00:00 [kworker/u2:0-fl]
8859    2  0 17:40 ?     00:00:00 [kworker/0:1-eve]
8866    2  0 17:40 ?     00:00:00 [kworker/0:10-ev]
8867    2  0 17:40 ?     00:00:00 [kworker/0:11-ev]
8887 6217  0 17:40 pts/0 00:00:00 bash
9562    2  0 17:45 ?     00:00:00 [kworker/u2:1-ev]
9594    2  0 17:45 ?     00:00:00 [kworker/0:0-eve]
9617 8887 21 17:46 pts/0 00:00:06 /usr/bin/firefox/firefox
9657 9617 17 17:46 pts/0 00:00:04 /usr/bin/firefox/firefox -contentproc -childID
9697 9617  4 17:46 pts/0 00:00:01 /usr/bin/firefox/firefox -contentproc -childID
9750 9617  3 17:46 pts/0 00:00:00 /usr/bin/firefox/firefox -contentproc -childID
9794 9617 11 17:46 pts/0 00:00:00 /usr/bin/firefox/firefox -contentproc -childID
9795 9794  0 17:46 pts/0 00:00:00 /usr/bin/firefox/firefox
9802 7441  0 17:46 pts/2 00:00:00 ps -aef
```

On rappelle que :

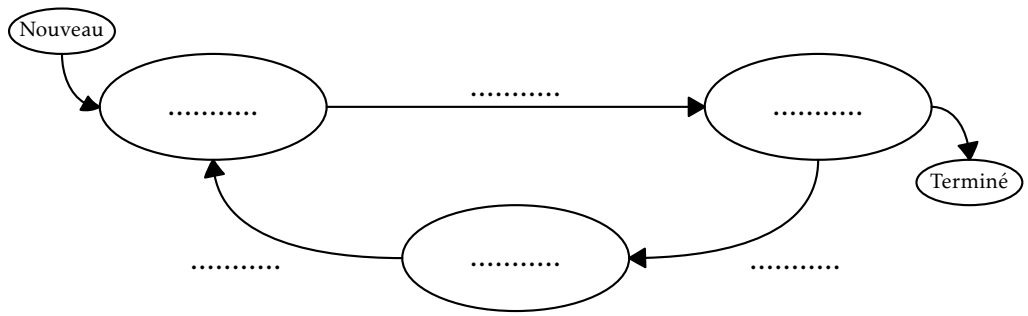
- `PID` : Identifiant d'un processus (Process IDentification)
- `PPID` : Identifiant du processus parent d'un processus (Parent Process IDentification)

a) Donner sous forme d'un arbre de PID la hiérarchie des processus liés à `firefox`.

b) Indiquer la commande qui correspond au processus parent du premier processus de `firefox`. .....

c) La commande `kill` permet de supprimer un processus à l'aide de son PID (par exemple `kill 8600`). Lorsqu'on supprime un processus, tous les sous-processus sont également supprimés.
Indiquer la commande qui permettra de supprimer tous les processus liés à `firefox` et uniquement ces processus. .....

2) a) Compléter le schéma ci-dessous avec les termes suivants concernant l'ordonnancement des processus : *Élu*, *En attente*, *Prêt*, *Blocage*, *Déblocage*, *Mise en exécution*.



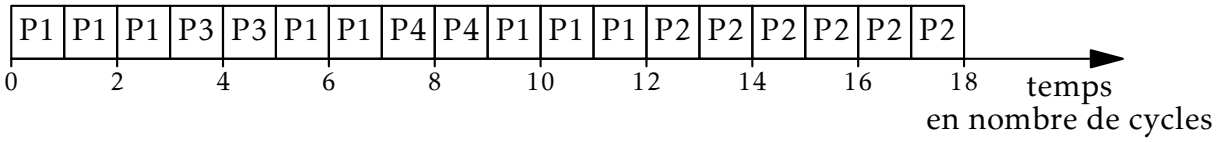
On donne dans le tableau ci-dessous quatre processus qui doivent être exécutés par un processeur. Chaque processus a un instant d'arrivée et une durée, donnés en nombre de cycles du processeur.

Processus	**P1**	**P2**	**P3**	**P4**
Instant d'arrivée	0	2	3	7
Durée	8	6	2	2

Les processus sont placés dans une file d'attente en fonction de leur instant d'arrivée.
On se propose d'ordonnancer ces quatre processus avec la méthode suivante :

- Parmi les processus présents en liste d'attente, l'ordonnanceur choisit celui dont la durée restante est la plus courte ;
- Le processeur exécute un cycle de ce processus puis l'ordonnanceur désigne de nouveau le processus dont la durée restante est la plus courte ;
- En cas d'égalité de temps restant entre plusieurs processus, celui choisi sera celui dont l'instant d'arrivée est le plus ancien ;
- Tout ceci jusqu'à épuisement des processus en liste d'attente.

On donne en exemple ci-dessous, l'ordonnancement des quatre processus de l'exemple précédent suivant l'algorithme ci-dessus.

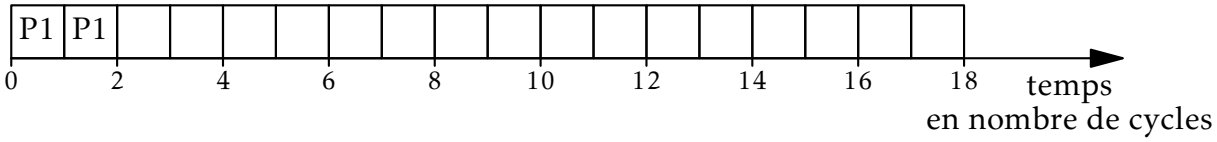


On définit le temps d'exécution d'un processus comme la différence entre son instant de terminaison et son instant d'arrivée.

b) Calculer la moyenne des temps d'exécution des quatre processus.

On se propose de modifier l'ordonnancement des processus. L'algorithme reste identique à celui présenté précédemment mais au lieu d'exécuter un seul cycle, le processeur exécutera à chaque fois deux cycles du processus choisi. En cas d'égalité de temps restant, l'ordonnanceur départagera toujours en fonction de l'instant d'arrivée.

c) Compléter le schéma ci-dessous donnant le nouvel ordonnancement des quatre processus.



d) Calculer la nouvelle moyenne des temps d'exécution des quatre processus et indiquer si cet ordonnancement est plus performant que le précédent.

On se propose de programmer l'algorithme du premier ordonnanceur. Chaque processus sera représenté par une liste comportant autant d'éléments que de durées (en nombre de cycles). Pour simuler la date de création de chaque processus, on ajoutera en fin de liste de chaque processus autant de chaînes de caractères vides que la valeur de leur date de création.

```
p1 = ['1.8', '1.7', '1.6', '1.5', '1.4', '1.3', '1.2', '1.1']
p2 = ['2.6', '2.5', '2.4', '2.3', '2.2', '2.1', '', '']
p3 = ['3.2', '3.1', '', '', '']
p4 = ['4.2', '4.1', '', '', '', '', '', '', '']
liste_proc = [p1, p2, p3, p4]
```

Une fonction `scrutation` (non étudiée) est chargée de parcourir la liste `liste_proc` de tous les processus et de renvoyer la liste d'attente des processus en fonction de leur arrivée. À chaque exécution de scrutation, les processus présents (sans chaînes de caractères vides en fin de liste) sont ajoutés à la liste d'attente. La fonction supprime pour les autres un élément de chaîne de caractères vides. Les processus qui sont terminés ne sont pas remis dans la liste d'attente.

3) a) La fonction `choix_processus` est chargée de sélectionner le processus dont le temps restant d'exécution est le plus court parmi les processus en liste d'attente.
Compléter la fonction `choix_processus` ci-dessous.

```
def choix_processus(liste_attente):
    """Renvoie l'indice du processus le plus court parmi
    ceux présents en liste d'attente liste_attente"""
    if liste_attente != []:
        mini = len(liste_attente[0])
        indice = 0









        return indice
```

Lors de l'éxecution d'un processus, on supprime son dernier élément, avec la méthode `pop`.

```
>>> proc = ['1.8', '1.7', '1.6', '1.5', '1.4']
>>> proc.pop()
'1.4'
>>> proc
['1.8', '1.7', '1.6', '1.5']
```

b) Compléter la fonction `ordonnancement` pour réaliser le fonctionnement désiré.

```
def ordonancement(liste_proc):
    """Exécute l'algorithme d'ordonnancement
    liste_proc -- liste des processus
    Renvoie la liste d'exécution des processus"""
    execution = []
    attente = scrutation(liste_proc, [])
    while attente != []:
        indice = choix_processus(attente)









        attente = scrutation(liste_proc, attente)
    return execution
```

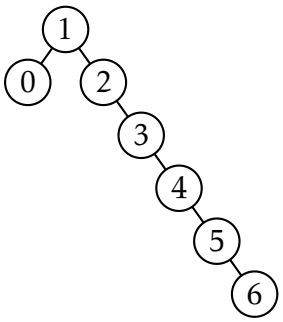
Exercice 2 : (12pt) *Cet exercice porte sur les arbres binaires de recherche, la POO et la récursivité.*

Nous disposons d'une classe `ABR` pour les arbres binaires de recherche dont les clés sont des entiers :

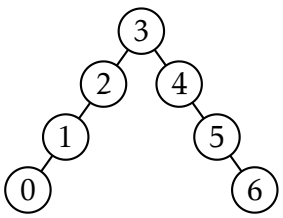
```
class ABR():
    def __init__(self) :
        # Initialise une instance d'ABR vide.

    def cle(self):
        # Renvoie la clé de la racine de l'instance d'ABR.

    def sad(self):
        # Renvoie le sous-arbre droit de l'instance d'ABR.

    def sag(self):
        # Renvoie le sous-arbre gauche de l'instance d'ABR.

    def est_vide(self):
        # Renvoie True si l'instance d'ABR est vide et False sinon.

    def inserer(self, cle_a_inserer):
        # Insère cle_a_inserer à sa place dans l'instance d'ABR.
```

Considérons ci-dessous trois arbres binaires de recherche :

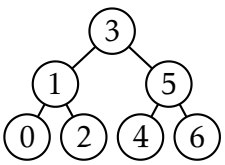
Arbre n°1



Arbre n°2



Arbre n°3



Dans tout l'exercice, nous ferons référence à ces trois arbres binaires de recherche et utiliserons la classe `ABR` et ses méthodes.

Partie A

1) Un arbre est une structure de données hiérarchique dont chaque élément est un nœud. Compléter le texte ci-dessous en choisissant des expressions parmi `au maximum`, `au minimum`, `exactement`, `feuille`, `racine`, `sous-arbre gauche` et `sous-arbre droit` :
   - Le nœud initial est appelé .....
   - Un nœud qui n'a pas de fils est appelé .....
   - Un arbre binaire est un arbre dans lequel chaque nœud a ..... deux fils.
   - Un arbre binaire de recherche est un arbre binaire dans lequel tout nœud est associé à une clé qui est :
     - supérieure à chaque clé de tous les nœuds de son .....
     - inférieure à chaque clé de tous les nœuds de son .....
2) Donner dans l'ordre les clés obtenues lors du parcours préfixe de l'arbre n°1.

Nom et prénom :

3) Donner dans l'ordre, les clés obtenues lors du parcours suffixe, également appelé post-fixe, de l'arbre n°2.

4) Donner dans l'ordre, les clés obtenues lors du parcours infixe de l'arbre n°3.

5) Compléter les instructions ci-dessous afin de définir puis de construire, en y insérant les clés dans un ordre correct (il y a plusieurs possibilités, on en demande une) , les trois instances de la classe `ABR` qui correspondent aux trois arbres binaires de recherche représentés plus haut.

```
arbre_no1 = ...
arbre_no2 = ...
arbre_no3 = ...
for cle_a_inserer in [..., ..., ..., ..., ..., ..., ...]:
    arbre_no1....
for cle_a_inserer in [..., ..., ..., ..., ..., ..., ...]:
    arbre_no2....
for cle_a_inserer in [..., ..., ..., ..., ..., ..., ...]:
    arbre_no3....
```

6) Voici le code de la méthode `hauteur` de la classe `ABR` qui renvoie la hauteur d'une instance d'`ABR` :

```
    def hauteur(self):
        if self.est_vide() :
            return -1
        else :
            return 1 + max(self.sag().hauteur(), self.sad().hauteur())
```

Donner, en vous basant sur cette fonction, la hauteur des trois instances `arbre_no1`, `arbre_no2` et `arbre_no3` de la classe `ABR` définies plus haut et qui correspondent aux trois arbres représentés plus haut.

7) Compléter le code de la méthode `est_presente` ci-dessous qui renvoie **`True`** si la clé `cle_a_rechercher` est présente dans l'instance d'`ABR` et **`False`** sinon :

```
    def est_present(self, cle_a_rechercher):
        if self.est_vide() :
            return ...
        elif cle_a_rechercher == self.cle() :
            return ...
        elif cle_a_rechercher < self.cle() :
            return ...
        else :
            return ...
```

8) Expliquer quelle instruction, parmi les trois ci-dessous, nécessitera le moins d'appels récursifs avant de renvoyer son résultat :

- `arbre_no1.est_presente(7)`
- `arbre_no2.est_presente(7)`
- `arbre_no3.est_presente(7)`

Partie B

9) On rappelle que la fonction `abs(x)` renvoie la valeur absolue de `x`. Par exemple :

```
>>> abs(3)
3
>>> abs(-2)
2
```

On donne la méthode `est_partiellement_equilibre(self)` de la classe `ABR`. Cette méthode renvoie `True` si l'instance de la classe `ABR` est l'implémentation d'un arbre partiellement équilibré et `False` sinon :

```
    def est_partiellement_equilibre(self) :
        if self.est_vide() :
            return True
        return abs(self.sag().hauteur() - self.sad().hauteur()) <= 1
```

D'après cette fonction, expliquer ce qu'on appelle ici un arbre *partiellement équilibré*.

Un arbre binaire est *équilibré* s'il est partiellement équilibré et si ses deux sous- arbres, droit et gauche, sont eux-mêmes équilibrés. Un arbre vide est considéré comme équilibré.

10) Justifier que, parmi les trois arbres définis plus haut, deux sont partiellement équilibrés.

11) Justifier que, parmi les trois arbres définis plus haut, un seul est équilibré.

12) Définir et coder la méthode récursive `est_equilibre` de la classe `ABR` qui renvoie `True` si l'instance de la classe `ABR` est l'implémentation d'un arbre équilibré et `False` sinon.