1. **Quelle structure de données choisir pour chacune de ces tâches ?**

|  |
| --- |
| Pile |
| File |
| Pile |
| File |
| Pile |
| Dictionnaire |

* *Stocker l’historique des actions effectuées dans un logiciel (Undo)*
* *Serveur d’impression (buffer de requêtes)*
* *Revenir en arrière dans un navigateur (page précédente)*
* *Mise en liste d’attente téléphonique lors d’un appel au FAI par exemple*
* *Gestion des évènements (ex. frappe clavier)*
* *Représenter un répertoire téléphonique*

1. **Pile et palindromes (difficulté \*)**

Un palindrome est un mot qui se lit de la même façon de gauche à droite et de droite à gauche. Par exemple, *kayak* et *coloc* sont des palindromes. Par opposition, *informatique* n’en est pas un.

Les piles peuvent être utiles pour détecter un palindrome: on peut lire le mot jusqu’à sa moitié, et empiler les lettres qu’on lit, puis arrivé à la moitié on lit les lettres tout en dépilant et en regardant si le résultat du dépilage correspond à la lettre lue. Si ce n’est pas le cas, le mot en entrée n’est pas un palindrome.

S’il y a toujours égalité, c’est un palindrome.

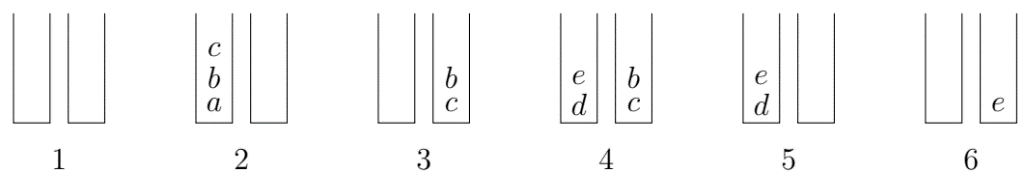
Il faut faire attention à distinguer les mots de longueur paire et impaire.

1. Créer un fichier python palindrome.py
2. Implémenter à l’aide d’une pile une fonction palindrome() qui prend en entrée un mot et renvoie True si c’est un palindrome et False sinon.
3. Tester votre programme avec différents mot/phrases : palindromes ou non
4. **Inversion d’une file à l’aide d’une pile (difficulté \*)**
5. Ecrire en python une procédure qui inverse une file d'éléments qui lui est passée en paramètre, à l’aide d’un Pile. Il existe en effet une méthode très simple pour inverser une file en utilisant une pile.

Obligatoire : Ne pas utiliser de tableau ou de liste de travail pour effectuer l'inversion, mais utiliser une pile.

1. **Réalisation d’une File à l’aide de 2 Piles (difficulté \*\*)**
2. Il est possible d’implémenter la structure de données File à l’aide de deux piles (pileDroite, pileGauche). Regarder l’exemple illustré ci-dessous et faites le lien avec l’algorithme qui suit l’illustration.

*1. On crée une file vide*

*2. On enfile les éléments a, b et c*

*3. On défile un élément*

*4. On enfile les éléments d et e*

*5. On défile deux éléments*

*6. On défile un élément.*

Algorithme de la méthode *défiler* la file :

***Fonction*** *defiler*

***Si*** *pileDroite.estVide()* ***Alors***

***Tant Que not*** *pileGauche.estVide()* ***Faire***

*pileDroite.empiler(pileGauche.depiler())*

***Fin Tant Que***

***Fin Si***

***Retourner*** *pileDroite.depiler()*

***Fin fonction***

1. Implémenter en programmation orientée objet une File à l’aide de 2 Piles (pileDroite et pileGauche). Ne faire que les méthodes **estVide**, **enfiler**, **defiler**
2. **Validité du parenthésage d'une expression à l’aide d’une Pile (difficulté \*\*)**

Un problème fréquent pour les compilateurs et les traitements de textes est de déterminer si les parenthèses d’une chaîne de caractères sont équilibrées et proprement incluses les unes dans les autres. On désire donc écrire une fonction qui teste la validité du parenthésage d’une expression :

– on considère que les expressions suivantes sont valides : "()", "[([bonjour+]essai)7plus- ];"

– alors que les suivantes ne le sont pas : "(", ")(", "4(essai]".

Notre but est donc d’évaluer la validité d’une expression en ne considérant que ses parenthèses et ses crochets. On suppose que l’expression à tester est dans une chaîne de caractères, dont on peut ignorer tous les caractères autres que ‘(’, ‘[’, ’]’ et ‘)’.

1. Écrire en Python la fonction valide qui renvoie vrai si l’expression passée en paramètre est valide, faux sinon.
2. **File et ordonnancement (difficulté \*\*\*)**

L’ordonnancement consiste, pour le système d’exploitation, à optimiser l’utilisation du processeur en lui affectant tour à tour différentes tâches à exécuter. On appelle processus un programme en cours d’exécution. Il peut y en avoir des centaines à la fois sur une machine, et que quelques processeurs (souvent 4).

L’ordonnanceur va répartir le temps de calcul entre les programmes, afin que tous puissent avancer dans leur exécution de manière satisfaisante, et que les programmes qui n’ont pas besoin de temps processeur à un certain moment (par exemple parce qu’ils attendent une réponse de l’utilisateur avant de continuer) ne gaspillent pas de temps de calcul.

La plupart des ordonnanceurs modernes utilisent des files pour garder en mémoire de façon optimale les programmes à exécuter. En effet, tout comme la pile était une structure naturelle pour gérer les palindromes à l’exercice précédent, la file est parfaitement adaptée à l’ordonnancement : les programmes qui demandent du temps de calcul sont insérés en bout de file, et ceux qui seront défilés pour obtenir effectivement du temps processeur sont ceux qui attendent depuis le plus longtemps.

1. Créer un fichier python scheduler.py.
2. Créer une classe Activite pour modéliser des activités avec :

* un constructeur \_\_init\_\_() initialisant ayant trois attributs privés : name, time et priority.
* un accesseur get\_time() qui renvoie la valeur de l’attribut time.
* un accesseur get\_priority() qui renvoie la valeur de l’attribut priority.
* une méthode publique execute() qui décrémente l’attribut time d’une valeur passée en paramètre à la méthode et qui renvoie un booléen indiquant si time est nul (True) ou non (False).

NB : time ne peut en aucun cas être < 0.

* une méthode spéciale \_\_repr\_\_(self) renvoyant une chaîne représentant l’activité selon le format : "<nom activité>: <temps>s [<priorité>]"

1. Créer une classe Ordonnanceur sur le patron suivant :

class Ordonnanceur:

def \_\_init\_\_(self, quota=0):

self.\_\_file = pf.File()

self.\_\_quota = int(quota)

def set\_quota(self, quota : int) -> int:

# à compléter

def add\_activity(self, activity : object):

# à compléter

def step(self):

# à compléter

def run(self):

# à compléter

1. Compléter la méthode add\_activity() qui ajoute une activité passée en paramètre à la file de processus de l’ordonnanceur.
2. Modifier la méthode step() qui effectue un “tour” d’ordonnancement comme suit :

* si la file est vide, on l’affiche et on ne fait rien.
* s’il y a au moins une activité dans la file, on exécute l’activité en affichant son nom et sa durée. On décrémente son temps d’une unité et si son quota arrive à 0, on défile l’activité.

1. Modifier la méthode run() qui itère step jusqu’à obtenir une file de processus vides.
2. Créer une liste de 10 activités de durée et de priorité aléatoires (durée entre 1 et 10 et priorité entre 0 et 2).
3. A l’aide d’une boucle, mettre toutes les activités dans la file de l’ordonnanceur puis exécuter l’ordonnanceur.

Ordonnancement préemptif, la stratégie “round-robin”

L’ordonnancement à l’aide d’une file vu précédemment est dit **coopératif** : l’ordonnanceur laisse à chaque processus tout le temps dont il a besoin. Dans une telle stratégie, c’est le processus qui doit rendre la main de lui-même, parce qu’il a terminé ou parce qu’il se met en attente (on n’a pas modélisé cette éventualité).

Dans les systèmes d’exploitation récents, l’ordonnancement est **préemptif** : au bout d’un certain temps, si le processus auquel est actuellement alloué le temps processeur n’a pas terminé, on le met en attente et on alloue le processeur à un autre processus.

1. Reprendre votre ordonnanceur pour qu’il prenne à l’initialisation un paramètre tmax dénotant le temps maximal laissé à un processus. Modifier ensuite la méthode step pour qu’à chaque étape elle considère une activité et :

* soit le temps d’exécution est inférieur à tmax et on exécute l’activité que l’on a défilée.
* soit ce temps d’exécution est strictement supérieur à tmax, alors on exécute l’activité défilée pendant tmax, puis on la renfile avec un temps d’exécution diminué de tmax.

Ordonnancement multi-files : les priorités

Dans les systèmes d’exploitation modernes, il y a une notion de priorité : certains processus plus critiques doivent être exécutés avant les autres. On suppose ici que les priorités des activités sont des entiers entre 0 et 2 (0 est la plus haute priorité, 2 la plus faible). Une stratégie approchant ce qui se fait dans les systèmes modernes est la suivante :

* on crée une file pour chaque priorité 0, 1 et 2.
* à l’ajout dans l’ordonnanceur, on ajoute l’activité à la bonne file.
* la fonction “step” de l’ordonnanceur se comporte comme un round-robin, mais qui commence sur la file de priorité 0. Si elle est vide, il cherche une tâche dans la file de priorité 1. Si elle est vide, on regarde la file de priorité 2.

1. Implémenter un tel ordonnanceur et le faire tourner sur un exemple.