



I - MAUVAISE Implémentation MUTABLE : modèle tableau dynamique

On ne peut pas faire une bonne implémentation dont les primitives sont toutes à coût CONSTANT.

Choix 1

L'Arrière est la première case du tableau t.

→ enfiler est à coût LIN avec _____

L'Avant est la dernière case du tableau t.

→ défiler est à coût CST avec _____

Choix 2

L'Avant est la première case du tableau t.

→ défiler est à coût LIN avec _____

L'Arrière est la dernière case du tableau t.

→ enfiler est à coût CST avec _____

!!! MAUVAISE IMPLEMENTATION !!!!! (exemple avec le CHOIX 2)

01	def nouvelle_file() -> 'File':	coût _____
02	return []	
03		
04	def est_file_vide(f:'File') -> bool:	coût _____
05	return f == []	
06		
07	def enfiler(f:'File', elt:'Element') -> None:	coût _____
08	f.append(elt) # on enfile en tant que dernière case	
09		
10	def defiler(f:'File NON VIDE') -> 'Element':	coût _____
11	return f.pop(0) # on défile la première case	
12		
13	def lire_avant(f:'file NON VIDE') -> 'Element':	coût _____
14	return f[0] # on lit la première case, l'Avant	

II - Bonne implémentation MUTABLE : modèle liste chaînée Tête Fin

Inutile d'espérer une bonne implémentation en utilisant une **liste chaînée de tuples** ou une **liste chaînée d'objets-cellules référençant uniquement la cellule de tête** :

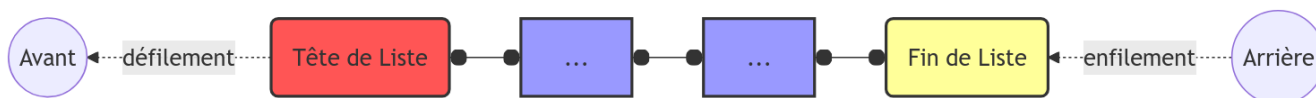
→ les actions sur la Tête sont à coût _____ → les actions sur la Fin sont à coût _____

Par contre, on peut utiliser une **liste chaînée d'objets référençant la Tête ET la Fin**. En effet ;

→ les actions sur la Tête sont à coût _____ → l'insertion en Fin est à coût _____

→ la suppression de la Fin est à coût _____

Il n'y a donc qu'une façon de faire :

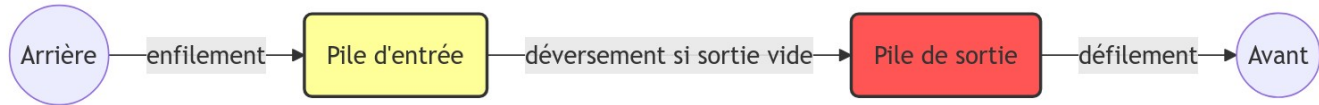


01	from liste_chaine_objet_tete_fin import nouvelle_liste	
02	from liste_chaine_objet_tete_fin import est_liste_vide	
03	from liste_chaine_objet_tete_fin import inserer_fin	
04	from liste_chaine_objet_tete_fin import supprimer_tete	
05	from liste_chaine_objet_tete_fin import premier	
06		
07	# Fonctions primitives de la File	
08		
09	def nouvelle_file() -> 'File':	coût _____
10	return nouvelle_liste()	
11		
12	def est_file_vide(f:'File') -> bool:	coût _____
13	'''Prédicat qui renvoie True si la File est vide'''	
14	return est_liste_vide(f)	
15		
16	def enfiler(f:'File', elt:'Element') -> None:	coût _____
17	'''Enfile elt dans la File'''	
18	inserer_fin(f, elt)	# On insère en Fin de liste
19		
20	def defiler(f:'File NON VIDE') -> 'Element':	coût _____
21	'''Défile la file f et renvoie l'élément supprimé'''	
22	return supprimer_tete(f)	# On supprime la Tête de liste
23		
24	def lire_avant(f:'File NON VIDE') -> 'Element':	coût _____
25	'''Renvoie la valeur située à l'Avant de la File f'''	
26	return premier(f)	

III - Bonne implémentation MUTABLE : modèle à deux Piles

Principe général

La file est un simple tableau contenant deux Piles : $f = [pEntree, pSortie]$



Enfiler dans la File revient à _____ dans la **Pile d'entrée** $f[]$

Défiler la File _____ revient à _____ la **Pile de sortie NON VIDE** $f[]$

Astuce (à connaître) : Si la Pile de sortie est VIDE lorsqu'on veut défiler, il faut d'abord _____ la Pile d'entrée dans la Pile de sortie.

Implémentation d'une File MUTABLE (module avec dépendance vers une Pile immuable)

```
01 from pile_immuable import nouvelle_pile
02 from pile_immuable import est_pile_vide
03 from pile_immuable import empiler
04 from pile_immuable import depiler
05 from pile_immuable import lire_sommet
06
07 def nouvelle_file() -> 'File':
08     pile_entree = nouvelle_pile() # inputStack en anglais
09     pile_sortie = nouvelle_pile() # outputStack en anglais
10     return [pile_entree, pile_sortie]
11
12 def est_file_vide(f:'File') -> bool:
13     return est_pile_vide(f[0]) and est_pile_vide(f[1]) # Les deux doivent être vides
14
15 def enfiler(f:'File', elt:'Element') -> None:
16     f[0] = empiler(f[0], elt)
17
18 def defiler(f:'File NON VIDE') -> 'Element':
19     if est_pile_vide(f[1]):
20         deverser_entree_vers_sortie(f)
21     valeur = lire_sommet(f[1])
22     f[1] = depiler(f[1])
23     return valeur
24
25 def lire_avant(f:'File NON VIDE') -> 'Element':
26     if est_pile_vide(f[1]):
27         deverser_entree_vers_sortie(f)
28     lire_sommet(f[1])
29
30 # Fonctions hors interface
31 def deverser_entree_vers_sortie(f:'File') -> None:
32     '''Déverse entièrement la pile d'entrée dans la pile de sortie'''
33     while not est_pile_vide(f[0]):
34         elt = lire_sommet(f[0])
35         f[0] = depiler(f[0])
36         f[1] = empiler(f[1], elt)
```

IV - Autre implémentation : modèle d'un type dict contenant un dict list

Le modèle est un dictionnaire contenant un tableau qui contient les éléments stockés dans la File, ainsi que d'informations permettant de savoir où insérer et où supprimer. Attention, ici la taille de la File est limitée par la taille du tableau associé à la clé 'data'.

```
>>> file = nouvelleFile()
>>> file
{'avant': None, 'arriere': None, 'taille': 0, 'max': 5, 'data': [None, None, None, None, None]}
>>> enfiler(10, file)
>>> file
{'avant': 0, 'arriere': 0, 'taille': 1, 'max': 5, 'data': [10, None, None, None, None]}
>>> enfiler(20, file)
>>> file
{'avant': 0, 'arriere': 1, 'taille': 2, 'max': 5, 'data': [10, 20, None, None, None]}
>>> enfiler(30, file)
>>> file
{'avant': 0, 'arriere': 2, 'taille': 3, 'max': 5, 'data': [10, 20, 30, None, None]}
>>> enfiler(100, file)
>>> file
{'avant': 0, 'arriere': 3, 'taille': 4, 'max': 5, 'data': [10, 20, 30, 100, None]}
>>> defiler(file)
10
>>> file
{'avant': 1, 'arriere': 3, 'taille': 3, 'max': 5, 'data': [10, 20, 30, 100, None]}
>>> defiler(file)
20
>>> file
{'avant': 2, 'arriere': 3, 'taille': 2, 'max': 5, 'data': [10, 20, 30, 100, None]}
```