Les piles et les files sont des structures permettant de stocker des objets. Les deux structures disposent de règles d'ajout et de retraits des éléments les distinguant :

— Dans une pile (*stack*) lorsqu'on veut retirer un élément, c'est le dernier élément à être entré que l'on retire. Ce comportement est nommé LIFO (*Last In First Out*). Pour visualiser cela, il suffit d'imaginer une pile d'assiette : on prend généralement l'assiette en haut de la pile, assiette qui y a été posée en dernier.

— Dans une file (*queue*) lorsqu'on veut retirer un élément, c'est le premier élément à être entré que l'on retire. Ce comportement est nommé FIFO (*First In First Out*). Pour visualiser cela, il suffit d'imaginer une file d'attente : premier arrivé, premier servi.

Dans ce chapitre nous allons détailler des interfaces minimales décrivant ces deux structures et voir des exemples d'utilisation avant d'en proposer des implémentations.

1 Interface et utilisation des piles

1.1 Interface des piles

Pour une pile d'éléments de type `T`, on dénote le type de la pile par `Pile[T]`. Pour décrire les interfaces, on va utiliser l'écriture suivante décrivant les types des arguments et du résultat :

```
fct(<type arg>) -> <type res>
```

Écrivons notre interface :

1 - On veut pouvoir créer une pile vide :

```
creer_pile() -> Pile[T]
```

2 - On veut également pouvoir tester si une pile est vide :

```
est_vide(Pile[T]) -> bool
```

3 - On veut pouvoir ajouter un élément au début de la pile :

```
empiler(Pile[T], T) -> None
```

4 - On veut pouvoir retirer et renvoyer le premier élément de la pile :

```
depiler(Pile[T]) -> T
```

1.2 Exemple d'utilisation des piles

Avec cette structure, on peut par exemple implémenter le bouton de retour en arrière d'un navigateur web. Pour cela, on garde en mémoire l'adresse courante que l'on empile si on veut accéder à une nouvelle page. Ensuite, si on veut revenir en arrière, il suffit de dépiler pour retrouver la bonne adresse.

```
adresse_courante = ""
adresses_precedentes = creer_pile()

def aller_a(adresse_cible):
    empiler(adresses_precedentes, adresse_courante)
    adresse_courante = adresse_cible

def retour():
    adresse_courante = depiler(adresses_precedentes)
```

2 Interface et utilisation des files

2.1 Interface des files

Pour une file d'éléments de type `T`, on dénote le type de la file par `File[T]`.

Écrivons notre interface :

1 - On veut pouvoir créer une file vide :

```
creer_file() -> File[T]
```

2 - On veut pouvoir tester si une file est vide :

```
est_vide(File[T]) -> bool
```

3 - On veut pouvoir ajouter un élément à la fin de la file :

```
ajouter(File[T], T) -> None
```

4 - On veut pouvoir retirer et renvoyer le premier élément de la file :

```
retirer(File[T]) -> T
```

2.2 Exemple d'utilisation des files

Avec cette structure, on peut par exemple programmer un jeu de bataille :

```
paquet_alice = creer_file()
paquet_bob = creer_file()

distribuer(paquet_alice, paquet_bob)

while not(est_vide(paquet_alice) or est_vide(paquet_bob)):
    a = retirer(paquet_alice)
    b = retirer(paquet_bob)
    if valeur(a) == valeur(b):
        ajouter(paquet_alice, a)
        ajouter(paquet_bob, b)
    elif valeur(a) > valeur(b):
        ajouter(paquet_alice, a)
        ajouter(paquet_alice, b)
    else:
        ajouter(paquet_bob, a)
        ajouter(paquet_bob, b)

if est_vide(paquet_alice):
    print("Bob a gagne !")
else:
    print("Alice a gagne !")
```

Plusieurs choses à préciser sur ce programme. Déjà on suppose disposer :

- d'un codage désignant les cartes (par exemple des chaînes de caractère de la forme `"2P"`, `"10C"`) ;
- d'une fonction `valeur(<carte>) -> int` donnant un score à chaque carte ;
- d'une fonction `distribuer(File[<carte>], File(<carte>))` qui distribue les cartes dans les deux paquets à l'aide la fonction `ajoute`.

De plus, on renvoie les cartes des deux joueurs dans leurs paquets respectifs en cas d'égalité, ce qui ne correspond pas à la bataille.

Travail personnel possible : implémenter ce qu'il manque à ce programme pour qu'il fonctionne correctement et régler le problème de l'égalité.

3 Implémentations

3.1 Implémentation des piles avec des listes chaînées

On a vu au chapitre précédent que les listes chaînées se comportaient naturellement comme une pile (FIFO), on peut donc écrire l'implémentation suivante :

```
class Pile :
    def __init__(self, v, s):
        self.valeur = v
        self.suivante = s

def est_vide(p):
    return p is None

def creer_pile():
    return None
```

```
11
12 def empiler(p, e):
13     p = Cellule(e, p)
14
15 def depiler(p):
16     res = p.valeur
17     p = p.suivante
18     return res
```

3.2 Implémentation des piles avec des tableaux Python

Les tableaux Python disposent de méthodes de piles, on peut donc les utiliser. Attention cela dit, en Python les tableaux sont gérés de sorte a avoir des méthodes efficaces mais ce n'est pas le cas dans tous les langages :

```
1  def creer_pile():
2      return []
3
4  def est_vide(p):
5      return p == []
6
7  def empiler(p, e):
8      p.append(e)
9
10 def depiler(p):
11     return p.pop()
```

Remarque : Ici la tête de la pile est le dernier élément du tableau.

3.3 Implémentation des files avec des listes chaînées mutables

Pour implémenter une file, on peut utiliser des listes chaînées mutables (qu'on modifie). L'idée est ici de garder en attributs la tête de la liste (premier élément, prêt à sortir) et sa queue (dernier élément, dernier rentré). Les fonctions d'ajout et de retrait doivent alors être adaptées :

```
1  class File:
2      def __init__(self):
3          self.tete = None
4          self.queue = None
5
6  def creer_pile():
7      return File()
8
9  def est_vide(p):
10     return p.tete is None
11
12 def ajouter(p, e):
13     #on cree la cellule devant etre mise en fin de liste
14     c = Cellule(e, None)
15     if est_vide(p):
16         #dans ce cas, la cellule ajoutee est en tete
17         p.tete = c
18     else :
19         #un lien de l'ancienne vers la nouvelle queue
20         p.queue.suivante = c
21     #quoiqu'il arrive la cellule ajoutee est en queue
22     p.queue = c
23
24 def retirer(p):
25     if p.tete is None :
26         raise IndexError("File Vide !")
27     #on va renvoyer la valeur de la tete
28     res = p.tete.valeur
29     #on met a jour la tete
30     p.tete = p.tete.suivante
31     if p.tete is None:
32         #si cela vide la file, la queue est egalement vide
33         p.queue = None
34     return res
```