

Ensuite, pour remettre les éléments dans le bon ordre, on dépile tous les éléments de la pile temporaire dans la pile de départ.

```
[ ] : def supprime_toutes_occurrences(pile, element):
    # BOUCLE 1: parcourir et dépiler entièrement 'pile' dans pile_temp
    # SAUF les occurrences de element
    # -> invariant: pile est vide des éléments parcourus
    # -> invariant: pile_temp contient, à l'envers, les éléments parcourus
    # SAUF les occurrences de element
    pile_temp = pile()
    # -> condition d'arrêt: pile est vide
    while not (pile.est_vide()):
        element_courant = pile.depiler()
        # maintenir de l'invariant
        # l'élément courant n'est pas élément
        if element_courant != element:
            # on ajoute l'élément courant dans la pile temporaire
            pile_temp.empiler(element_courant)
    # fin BOUCLE: tout pile est parcouru + invariant
    # => pile_temp contient toutes les valeurs de pile sauf élément
    # => pile est vide
    # a l'envers
    # fin BOUCLE 2: parcourir pile_temp et tout dépiler dans pile
    while not (pile_temp.est_vide()):
        pile.empiler(pile_temp.depiler())
    # de element n'y sont plus !
```

## 2021 - J1 - Métropole 2

### Exercice 2

**Q1.a** Écrire une suite d'instructions permettant de créer une instance de la classe Pile affectée à une variable pile1 contenant les éléments 7, 5 et 2 insérés dans cet ordre.

Ainsi, à l'issue de ces instructions, l'instruction pile1.afficher() produit l'affichage : 7, 5, 2.

#### R1.a

```
[ ] : pile1 = Pile
      pile1.empiler(7)
      pile1.empiler(5)
      pile1.empiler(2)
```

**Q1.b** Donner l'affichage produit après l'exécution des instructions suivantes.

```
[ ] : element1 = pile1.depiler()
      pile1.empiler(5)
      pile1.empiler(element1)
      pile1.afficher()
```

**R1.b** Le dernier élément de la pile1 est dépilé et stocké dans la variable element1. La pile contient donc (en respectant l'affichage de l'énoncé) : 7, 5.

Ensuite on empile (dans cet ordre) : 5 puis 2.

L'affichage de pile1 produit donc 7, 5, 5, 2.

**Q2** On donne la fonction mystere suivante :

```
[ ] : def mystere(pile, element):
      pile2 = Pile()
      nb_elements = pile.nb_elements()
      for i in range(nb_elements):
          elem = pile.depiler()
          pile2.empiler(elem)
          if elem == element:
              return pile2
      return pile2
```

**Q2.a cas n°1.** Quel est l'affichage de



```
[ ]: >>> pile.afficher()
7, 5, 2, 3
>>> mystere(pile, 2).afficher()
```

**R2.a cas n°1.** L'appel `mystere(pile, 2)` renvoie une pile correspondant à pile retournée jusqu'à rencontrer la valeur 2.

La méthode `afficher()` appliquée à cette pile affiche donc : 3, 2

**Q2.a cas n°2.** Quel est l'affichage de

```
[ ]: >>> pile.afficher()
7, 5, 2, 3
>>> mystere(pile, 9).afficher()
```

**R2.a cas n°2.** L'appel `mystere(pile, 9)` renvoie une pile correspondant à pile retournée car 9 n'appartient pas à pile.

La méthode `afficher()` appliquée à cette pile affiche donc : 3, 2, 5, 7.

**Q2.a cas n°3.** Quel est l'affichage de

```
[ ]: >>> pile.afficher()
7, 5, 2, 3
>>> mystere(pile, 9).afficher()
```

**R2.a cas n°3.** L'appel `mystere(pile, 3)` renvoie une pile correspondant à pile retournée jusqu'à rencontrer 3. Ce qui arrive dès le premier tour de boucle.

La méthode `afficher()` appliquée à cette pile affiche donc : 3.

**Q2.a cas n°4.** Quel est l'affichage de

```
[ ]: >>> pile.est_vide()
True
>>> mystere(pile, 3).afficher()
```

**R2.a cas n°4.** L'appel `mystere(pile, 3)` renvoie une pile vide car aucun tour de boucle n'est effectué.

La méthode `afficher()` appliquée à cette pile affiche donc : 3.

**Q2.b** Expliquer ce que permet d'obtenir la fonction `mystere`.

**R2.b** Si l'argument `element` n'appartient pas à `pile`, la fonction `mystere` renvoie une copie de pile retournée. Si `element` est présent, elle renvoie un extrait jusqu'à



element de la copie de la pile retournée.

**Q3** Écrire une fonction `etendre(pile1, pile2)` qui prend en arguments deux objets `Pile` appelés `pile1` et `pile2` et qui modifie `pile1` en lui ajoutant les éléments de `pile2` rangés dans l'ordre inverse. Cette fonction ne renvoie rien.

On donne ci-dessous les résultats attendus pour certaines instructions.

```
[ ]: >>> pile1.afficher()
7, 5, 2, 3
>>> pile2.afficher()
1, 3, 4
>>> etendre(pile1, pile2)
>>> pile1.afficher()
7, 5, 2, 3, 4, 3, 1
>>> pile2.est_vide()
True
```

**R3** Il suffit de dépiler `pile2` et d'empiler au fur et à mesure les éléments dans `pile1`. À la fin `pile2` sera complètement vide.

```
[ ]: def etendre(pile1, pile2):
    # BOUCLE: parcours de pile2
    # -> invariant: pile1 contient les éléments de piles2 parcourus
    # -> invariant: pile2 ne contient plus les éléments parcourus
    # -> condition d'arrêt: pile2 est vide
    while not (pile2.est_vide()):
        element = pile2.depiler()
        pile1.empiler(element)

    # fin BOUCLE: tout pile2 a été parcouru + invariants
    # => pile2 est vide ET pile1 possède les éléments de pile2
```

**Q4** Écrire une fonction `supprime_toutes_occurences(pile, element)` qui prend en arguments un objet `Pile` appelé `pile` et un élément `element` et supprime tous les éléments `element` de `pile`.

On donne ci-dessous les résultats attendus pour certaines instructions.

```
[ ]: >>> pile.afficher()
7, 5, 2, 3, 5
>>> supprime_toutes_occurences (pile, 5)
>>> pile.afficher()
7, 2, 3
```

**R4** L'idée est de dépiler tous les elements de la pile, sauf occurrence, dans une pile temporaire.

La pile passée en argument est donc vide.