Corrigé : gestion d'un allocateur dynamique de mémoire (X PC 2021)

Partie I. Implémentation naïve

Question 1.

```
def initialiser(p, n, c):
    for i in range(n):
        mem[p + i] = c
```

Question 2. Il s'agit simplement d'indiquer que la prochaine allocation de mémoire aura lieu à partir de l'indice 1 :

```
def demarrage():
    mem[0] = 1
```

Question 3. Ici, il faut initialiser la zone de mémoire *ad hoc* puis mettre à jour le première case.

```
def reserver(n, c):
    p = mem[0]
    if p + n <= TAILLE_MEM:
        initialiser(p, n, c)
        mem[0] += n
        return p
```

La fonction `initialiser` possède une complexité en $O(n)$ donc celle de `reserver` est en $O(1)$ si $p+n >$ `TAILLE_MEM`, et en $O(n)$ sinon.

Partie II. Réservation de blocs de tailles fixes

Question 4.

```
def demarrage():
    ecrire_prochain(2)
```

Question 5.

```
def reserver(n, c):
    if n < TAILLE_BLOC:
        q = lire_prochain()
        p = 2
        while p < q and est_reservee(p):
            p += TAILLE_BLOC
        if p + n <= TAILLE_MEM:
            marque_reservee(p)
            initialiser(p, n, c)
            if p == q:
                ecrire_prochain(p + TAILLE_BLOC)
    return p
```

La complexité de cette fonction est un $O(1)$ si $n \geqslant$ `TAILLE_BLOC`.
Dans les autres cas, la recherche d'un emplacement libre est un $O\left(\frac{\texttt{TAILLE\_MEM}}{\texttt{TAILLE\_BLOC}}\right)$ et enfin, si l'écriture est possible, celle-ci a une complexité en $O(n)$.

Question 6.

```
def liberer(p):
    marque_libre(p)
```

Partie III. Portions avec en-tête et pied de page

Question 7. Dans la case `mem[p-1]` se trouve l'en-tête de p : un entier de la forme $2q+r$ où $2q$ est la taille de la portion (nécessairement paire) et r un entier qui vaut 1 si la portion est réservée, 0 sinon. Ainsi, `mem[p-1] % 2` renvoie la valeur de r, ce qui explique la fonction `est_reservee`, et `mem[p-1] // 2` renvoie la valeur de q, ce qui explique la fonction `lire_taille`.
Dans la fonction `marque_reservee`, la variable `mot` contient la quantité $2q+1$ où $2q$ est la taille de la portion, les deux lignes suivantes servant à initialiser l'en-tête et le pied de page.
Enfin, dans la fonction `precedent_est_libre`, la case `mem[p-2]` contient le pied de page du bloc précédent, et sa valeur modulo 2 indique si le bloc précédent est libéré ou pas.

Question 8. On positionne l'épilogue puis on réserve prologue et épilogue.

```
def demarrage():
    ecrire_position_epilogue(4)
    marque_reservee(PROLOGUE, 0)
    marque_reservee(4, 0)
```

Question 9.

```
1  def reserver(n, c):
2      taille = n + n % 2  # pour le cas où n est impair
3      p = lire_position_epilogue()
4      q = p
5      while q > PROLOGUE:
6          t = lire_taille_precedent(q)
7          q -= t + 2
8          if est_libre(q) and taille <= t:
9              break
10     if q > PROLOGUE:
11         marque_reservee(q, taille)
12         initialiser(q, n, c)
13         if taille < t - 2:
14             marque_libre(q + taille + 2, t - taille - 2)
15         return q
16     elif p + taille + 2 < TAILLE_MEM:
17         marque_reservee(p, taille)
18         initialiser(p, n, c)
19         ecrire_position_epilogue(p + taille + 2)
20         marque_reservee(lire_position_epilogue(), 0)
21         return p
```

Les lignes 5-9 cherchent un bloc libre de taille adéquate à gauche de l'épilogue.
Si on en trouve un, les lignes 10-15 le remplissent, et s'il reste un reliquat de mémoire, créent un nouveau bloc libre (lignes 13-14).
Enfin, si on en trouve pas et si la taille restante est suffisante, les lignes 17-21 remplissent un nouveau bloc à la place de l'épilogue, et décalent ce dernier.
La recherche d'un emplacement libre a une complexité en O(`TAILLE_MEM`), donc la complexité totale de cette fonction est en O(`TAILLE_MEM` $+n$).

Question 10.

```
1  def liberer(p):
2      marque_libre(p, lire_taille(p))
3      q = p + lire_taille(p) + 2
4      if est_libre(q):
```

```
5           marque_libre(p, lire_taille(p) + lire_taille(q) + 2)
6       if precedent_est_libre(p):
7           q = p - lire_taille_precedent(p) - 2
8           marque_libre(q, lire_taille(p) + lire_taille(q) + 2)
```

Puisqu'il n'y a jamais deux blocs libres consécutifs, une libération de mémoire nécessitera au plus deux fusions, avec le bloc suivant (lignes 3-5) et le bloc précédent (lignes 6-8). La complexité de cette fonction est donc constante.

Partie IV. Chaînage explicite des portions libres

Question 11. Il faut distinguer le cas de la chaîne vide.

```
def ajoute_en_entree_de_chaine(p):
    if not chaine_est_vide():
        q = lire_entree_chaine()
        ecrire_predecesseur(q, p)
        ecrire_successeur(p, q)
    ecrire_predecesseur(p, 0)
    ecrire_entree_chaine(p)
```

Question 12.

```
def supprime_dans_chaine(p):
    q = lire_predecesseur(p)
    r = lire_successeur(p)
    if q != 0:
        ecrire_successeur(q, r)
    if r != 0:
        ecrire_predecesseur(r, q)
```

Question 13.

```
def demarrage():
    ecrire_entree_chaine(0)
    ecrire_position_epilogue(5)
    marque_reservee(PROLOGUE, 0)
    marque_reservee(5, 0)
```

Question 14. Seule la partie consistant à chercher un emplacement libre avant l'épilogue (lignes 4-9) a changé :

```
1  def reserver(n, c):
2      taille = n + n % 2  # pour le cas où n est impair
3      p = lire_position_epilogue()
4      q = lire_entree_chaine()
5      while q != 0:
6          if taille <= lire_taille(q):
7              break
8          q = lire_successeur(q)
9      if q != 0:
10         marque_reservee(q, taille)
11         initialiser(q, n, c)
12         if taille < t - 2:
13             marque_libre(q + taille + 2, t - taille - 2)
14         return q
15     elif p + taille + 2 < TAILLE_MEM:
16         marque_reservee(p, taille)
17         initialiser(p, n, c)
18         ecrire_position_epilogue(p + taille + 2)
19         marque_reservee(lire_position_epilogue(), 0)
20         return p
```

Question 15. Il suffit d'ajouter à la fonction précédente la suppression éventuelle des blocs libres à gauche et à droite, et insérer dans la chaîne le nouveau bloc libre.

```
def liberer(p):
    marque_libre(p, lire_taille(p))
    q = p + lire_taille(p) + 2
    if est_libre(q):
        supprime_dans_chaine(q)
        marque_libre(p, lire_taille(p) + lire_taille(q) + 2)
    if precedent_est_libre(p):
        q = p - lire_taille_precedent(p) - 2
        supprime_dans_chaine(q)
        marque_libre(q, lire_taille(p) + lire_taille(q) + 2)
        p = q
    ajoute_en_entree_de_chaine(p)
```