ECOLE POLYTECHNIQUE - ESPCI - ECOLES NORMALES SUPERIEURES
Épreuve d'informatique pour tous 2021
Un corrigé

Gestion d'un allocateur dynamique de mémoire

Partie I : Implémentation naïve

Question 1.

```
def initialiser(p, n, c):
    for i in range(n):
        ecrire(p,i,c)
```

Question 2.

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

Question 3.

```
def reserver(n, c):
    p = mem[0]
    if p+n >= TAILLE_MEM: # plus assez de place en mémoire
        return None
    else :
        initialiser(p, n, c)
        mem[0] = p+n      # on met à jour la première position libre
        return p
```

S'il n'y a pas assez de place, il y a juste une affectation et une comparaison, le coût est en $O(1)$. Sinon, l'appel de la fonction `initialiser` a un coût en $\mathcal{O}(n)$ (on effectue n affectations dans une liste, chaque affectation étant de coût unitaire). Les autres opérations (deux lectures dans une liste, deux additions) ayant un coût unitaire, on peut les négliger.

conclusion : la complexité de `reserver(n, c)` est en $\begin{cases} \mathcal{O}(1) & \text{s'il n'y a pas assez de place} \\ \mathcal{O}(n) & \text{sinon} \end{cases}$

Partie II : Réservations de blocs de tailles fixes

Question 4.

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

Question 5.

```
def reserver(n, c):
    if n >= TAILLE_BLOC:
        return None
    else :
        p = 2
        # on recherche le premier bloc libre
        while p < TAILLE_MEM and est_reservee(p):
            p = p + TAILLE_BLOC
        if p >= TAILLE_MEM: # la mémoire est pleine
            return None
        else :
            initialiser(p,n,c)
            marque_reservee(p)
            if p >= lire_prochain():      # mise à jour de mem[0] si besoin
                prochain = p+TAILLE_BLOC
                ecrire_prochain(prochain)
            return p
```

Le coût de la recherche du premier bloc libre est au pire proportionnel au nombre de blocs et le nombre de blocs est au plus égal à $\left\lfloor \frac{\texttt{TAILLE\_MEM}}{\texttt{TAILLE\_BLOC}} \right\rfloor$, le coût de cette partie de recherche est donc en $\mathcal{O}\left(\left\lfloor \frac{\texttt{TAILLE\_MEM}}{\texttt{TAILLE\_BLOC}} \right\rfloor \right)$. Le coût de l'initialisation est comme dans la partie I en $\mathcal{O}(n)$ et la mise à jour éventuelle de `mem[0]` est un nombre fixe d'opérations en $\mathcal{O}(1)$.

conclusion : la complexité de `reserver(n, c)` est en $\begin{cases} \mathcal{O}(1) & \text{si } n \geqslant \text{TAILLE\_BLOC} \\ \mathcal{O}(n) + \mathcal{O}\left(\left\lfloor \frac{\texttt{TAILLE\_MEM}}{\texttt{TAILLE\_BLOC}} \right\rfloor \right) & \text{sinon} \end{cases}$

Question 6.

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

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

Question 7. `est_reserve(p)` étudie la parité de l'en-tête de la portion `p`. Si la parité est impaire, la portion est réservée (la fonction retourne `True`), sinon elle est libre (la fonction retourne `False`).
`marque_reserve(p, taille)` permet de réserver une portion de la mémoire de taille `taille` (entier pair) en position `p` en codant correctement l'en-tête et le pied de page.
`lire_taille(p)` permet d'obtenir la taille de la portion `p`. Si la taille de la portion est $2k$, la valeur de l'en-tête est égale à $e = 2k$ si la portion a été libérée ou $e = 2k + 1$ si la portion est réservée. Dans tous les cas, la quantité retournée est égale à $2 \left\lfloor \frac{e}{2} \right\rfloor = 2k =$ taille de la portion.
`precedent_est_libre(p)` retourne `True` si la portion précédente est libre et `False` sinon. Pour cela, il suffit de regarder la parité du pied de page de la portion précédente qui est situé à l'adresse $p - 2$.

Question 8.

```
1  def demarrage():
2      ecrire_position_epilogue(PROLOGUE+2)
3      marque_reservee(PROLOGUE,0)        # création de la portion prologue
4      marque_reservee(PROLOGUE+2, 0)     # création de la portion épilogue
```

Question 9.

```
1  def reserver(n, c):
2      m = 2 *((n+1)//2)     # calcul de la taille de la portion
3      p = lire_position_epilogue()
4      q = PROLOGUE
5      # recherche de la première portion libre de bonne taille
6      while q < p and (est_reservee(q) or lire_taille(q) < n):
7          q = q + lire_taille(q) + 2
8      if q  + n >= TAILLE_MEM:
9          print('reserver : Erreur, taille limite atteinte')
10         return None
11     else :
12         taille = lire_taille(q)
13         marque_reservee(q, m)
14         initialiser(q,n,c)
15         if m < taille:
16             marque_libre(q+m+2, taille-m-2)
17         if p == q: # on décale l'épilogue si nécessaire
18             nouv_p = q+m+2
19             marque_reservee(nouv_p,0)
20             ecrire_position_epilogue(nouv_p)
21         return q
```

Le coût de la recherche de la première portion libre de taille suffisante est, dans le pire des cas, proportionnel aux nombres de portions présentes. Chaque portion non vide ayant une taille minimale de 4, on a au plus `TAILLE_MEM`/4 portions, la complexité de la boucle des lignes 6 et 7 est donc en $\mathcal{O}(\texttt{TAILLE\_MEM})$. Le coût de l'initialisation est toujours en $\mathcal{O}(n)$ et toutes les autres instructions étant de complexité $\mathcal{O}(1)$.

conclusion : la complexité de `reserver(n, c)` est en $\begin{cases} \mathcal{O}(\texttt{TAILLE\_MEM}) & \text{s'il n'y a pas assez de place} \\ \mathcal{O}(n) + \mathcal{O}(\texttt{TAILLE\_MEM}) & \text{sinon} \end{cases}$

remarque : Le nombre de portions présentes en mémoire est plus adapté pour établir la complexité de la fonction `reserver(n, c)`. On peut remplaer $\mathcal{O}(\texttt{TAILLE\_MEM})$ par $\mathcal{O}(p)$ dans l'expression précédente, où p est le nombre total de portions.

Question 10. La fonction `bloc_suivant` donne l'adresse de la portion qui suit la portion `p`. La fonction `fusionne_precedent` fusionne la portion (supposée libre) `p` avec la portion précédente (elle-même supposée libre) pour former une seule portion libre.

```
def bloc_suivant(p):
    return p+lire_taille(p)+2

def fusionne_precedent(p,n):
    n_prec = lire_taille_precedent(p)
    p = p-n_prec-2
    marque_libre(p,n+n_prec+2)

def liberer(p):
    n = lire_taille(p)
    marque_libre(p, n)
    # cas où le bloc précédent est libre
    if precedent_est_libre(p):
        fusionne_precedent(p,n)
    # cas où le bloc suivant est libre
    q = bloc_suivant(p)
    if est_libre(q):
        fusionne_precedent(q, lire_taille(q))
```

On effectue au plus deux fusions, l'une avec le bloc précédent si celui est libre et l'autre avec le bloc suivant si il est libre. Avec les hypothèses de bon usage, on ne doit jamais avoir deux portions libres contiguës. Toutes les opérations se font à coût constant, elles ne dépendent ni de n, ni de la taille de la mémoire.

conclusion : la complexité de `liberer(n, c)` est en $\mathcal{O}(1)$

Les portions prologues et épilogues sont deux portions qui ne sont jamais libérées et qui permettent un bon fonctionnement de la fonction `liberer`. En effet, ces deux portions ne peuvent pas être fusionnée avec une portion libre puisqu'elles ne sont jamais libérées

Partie IV : Chaînage explicite des portions libres

Question 11.

```
def ajoute_en_entree_de_chaine(p):
    entree = lire_entree_chaine()
    ecrire_entree_chaine(p)
    ecrire_predecesseur(p,0)
    ecrire_successeur(p,entree)
    if entree != 0 :
        ecrire_predecesseur(entree, p)
```

Pour la fonction `ajoute_en_entree_de_chaine`, il faut faire attention au cas de la chaîne vide, 0 n'ayant jamais de prédécesseur.

Question 12.

```
def supprime_dans_chaine(p):
    pred = lire_predecesseur(p)
    succ = lire_successeur(p)
    if pred != 0:
        ecrire_successeur(pred, succ)
    else :
        ecrire_entree_chaine(succ)
    if succ != 0:
        ecrire_predecesseur(succ,pred)
```

Pour la fonction `supprime_dans_chaine`, il faut faire attention aux cas l'élément à supprimer est le premier élément de la chaîne ou le dernier.

Question 13.

```
def demarrage():
    ecrire_position_epilogue(PROLOGUE+2)
    marque_reservee(PROLOGUE,0)       # création de la portion prologue
    marque_reservee(PROLOGUE+2,0)     # création de la portion épilogue
    ecrire_entree_chaine(0)
```

Question 14.

```
def reserver(n, c):
    m = 2 *((n+1)//2)     # calcul de la taille de la portion
    epilogue = lire_position_epilogue()
    q = lire_entree_chaine()
    # recherche de la première portion libre de bonne taille
    while q != 0 and  lire_taille(q) < n:
        q = lire_successeur(q)
    if q == 0 and q  + epilogue + 2 >= TAILLE_MEM:
        print('reserver : Erreur, taille limite atteinte')
        return None
    else :
        if q == 0:
            q = epilogue
            nouv_p = q+m+2
            marque_reservee(nouv_p,0)
            ecrire_position_epilogue(nouv_p)
        else :
            supprime_dans_chaine(q)
        taille = lire_taille(q)
        marque_reservee(q, m)
        initialiser(q,n,c)
        if m < taille:
            marque_libre(q+m+2, taille-m-2)
            ajoute_en_entree_de_chaine(q+m+2)
        return q
```

Le principe est le même que celui de la fonction de la question 9 sauf qu'ici, au lieu de parcourir tous les blocs à la recherche du premier bloc libre de taille suffisante, on ne parcourt que les blocs libres.

conclusion :

la complexité de `reserver(n, c)` est en $\begin{cases} \mathcal{O}(p) & \text{si place insuffisante} \\ \mathcal{O}(n)+\mathcal{O}(p) & \text{sinon} \end{cases}$ où p est le nombre de blocs libres

Question 15.

```
def bloc_suivant(p):
    return p+lire_taille(p)+2

def fusionne_precedent(p,n):
    supprime_dans_chaine(p)
    n_prec = lire_taille_precedent(p)
    p = p-n_prec-2
    marque_libre(p,n+n_prec+2)

def liberer(p):
    n = lire_taille(p)
    marque_libre(p, n)
    ajoute_en_entree_de_chaine(p)
    if precedent_est_libre(p): # cas où le bloc précédent est libre
        fusionne_precedent(p,n)
    q = bloc_suivant(p)
    if est_libre(q):           # cas où le bloc suivant est libre
        fusionne_precedent(q, lire_taille(q))
```

Par rapport à la solution de la question 10, on ne fait qu'ajouter et supprimer un ou deux éléments dans la chaîne des portions libres. L'ajout et la suppression dans la chaîne des portions libres se faisant à coût constant, on ne change pas l'ordre de grandeur de la complexité.

conclusion : la complexité de `liberer(p)` est en $\mathcal{O}(1)$