ECOLE POLYTECHNIQUE - ESPCI
ECOLES NORMALES SUPERIEURES

CONCOURS D'ADMISSION 2021

JEUDI 15 AVRIL 2021
16h30 - 18h30

FILIERES MP-PC-PSI
Epreuve n° 8
INFORMATIQUE B (XELCR)

Durée : 2 heures

L'utilisation des calculatrices n'est pas autorisée pour cette épreuve

Gestion d'un allocateur dynamique de mémoire

L'utilisation des calculatrices **n'est pas autorisée** pour cette épreuve. Le langage de programmation sera **obligatoirement** `Python`.

Complexité. La complexité, ou le temps d'exécution, d'une fonction P est le nombre d'opérations élémentaires (addition, multiplication, affectation, test, etc...) nécessaires à l'exécution de P. Lorsqu'il est demandé de donner la complexité d'un programme, le candidat devra justifier cette dernière si elle ne se déduit pas directement de la lecture du code.

Rappels concernant le langage `Python`. *L'utilisation de toute fonction* `Python` *sur les listes autre que celles mentionnées dans ce paragraphe est interdite.*

Si `a` désigne une liste en `Python` :

- `len(a)` renvoie la longueur de cette liste, c'est-à-dire le nombre d'éléments qu'elle contient ; la complexité de `len` est en $\mathcal{O}(1)$.
- `a[i]` désigne le `i`-ème élément de la liste, où l'indice `i` est compris entre 0 et `len(a) - 1` ; la complexité de cette opération est en $\mathcal{O}(1)$.

  On pourra aussi utiliser la fonction `range` pour constuire une liste d'entiers et réaliser des itérations.

Introduction. Dans ce sujet, on s'intéresse à un environnement de programmation qui a une mémoire limitée, et où l'on veut limiter au maximum la création de nouvelles structures (comme les listes `Python`) en mémoire. Pour cela, nous souhaitons fournir au programmeur un environnement de gestion dynamique de la mémoire dans lequel il pourra *réserver* des *portions* de mémoire pour y lire et écrire des données, puis *libérer* certaines portions quand il n'en a plus besoin. Pour simplifier, nous ne permettons au programmeur de lire et écrire dans ces portions de mémoire que des valeurs de type caractère, c'est-à-dire des chaînes de caractères de longueur 1. Une portion qui n'est pas (ou n'est plus) réservée est dite *libre*. Chaque portion agit comme un tableau de caractères : elle a une taille, et on peut lire et écrire les caractères aux positions comprises entre 0 et $t-1$, si t est la taille demandée pour la portion considérée lors de sa réservation.

Ce service est réduit aux cinq fonctions suivantes :

- `p = reserver(n, c)` qui renvoie une nouvelle portion `p` réservée qui était libre précédemment. La portion est de taille `n` et toutes ses `n` cases contiennent le même caractère `c`. La fonction renvoie `None` si la mémoire est trop pleine pour permettre cette réservation.

- `liberer(p)` qui libère une portion `p` qui était réservée précédemment.
- `c = lire(p, i)` renvoie le caractère `c` stocké à la case en position `i` dans la portion `p`.
- `ecrire(p, i, c)` qui met à jour la case en position `i` dans la portion `p` avec le caractère `c`.
- `demarrage()` qui est appelée une seule fois pour initialiser correctement la mémoire, avant tout appel aux quatre fonctions précédentes.

Dans ce sujet, on considère que le programmeur fera un usage licite de ces fonctions en respectant les préconditions suivantes :

- il ne lira et n'écrira qu'en utilisant les fonctions `lire` et `ecrire`, dans les portions actuellement réservées et à des positions comprises entre 0 et $t-1$, si t est la taille demandée pour la portion considérée lors de sa réservation ;
- il n'utilisera les fonctions `libere`, `lire` et `ecrire` qu'avec des portions `p` obtenues par un appel à `reserver` et qui n'auront pas été explicitement libérées depuis cet appel ;
- il n'utilisera `p = reserver(n, c)` que lorsque l'entier `n` est strictement positif ;
- il n'utilisera les fonctions `libere`, `lire`, `ecrire` et `reserver` que sur une mémoire correctement initialisée avec un appel à `demarrage()`.

Ces propriétés sont appellées *hypothèses de bon usage*.

Pour implémenter ces services, nous créons initialement une liste `Python` `mem` de taille `TAILLE_MEM` (intialisée avec des `0`). Les variables `mem` et `TAILLE_MEM` sont des variables globales. **Aucune autre liste `Python` ne doit être manipulée dans ce sujet**, exception faite des listes créées avec la fonction `range` pour réaliser des itérations.

```
mem = [0] * TAILLE_MEM
```

On suppose que `mem` vient juste d'être initialisée de cette façon au moment de l'appel de `demarrage()`. Cette liste va contenir les différents contenus des portions qui seront réservées et libérées. Chaque case contiendra soit un caractère placé par le programmeur, soit un entier que nous placerons pour organiser notre structure de données. Grâce aux hypothèses de bon usage, le programmeur n'accédera jamais à des cases contenant des entiers.

Dans tout le sujet, les services de lecture et écriture sont donnés par les fonctions suivantes :

```
def lire(p, i):
    return mem[p+i]

def ecrire(p, i, c):
    mem[p+i] = c
```

Par conséquent, nous désignerons par une portion un indice valide `p` de `mem`, tel que `mem[p]` ... `mem[p+n-1]` sont les cases réservées par le programmeur lors de l'appel à

`p = reserver(n,c)`. Nous confondons donc la portion `p` avec la première position, accessible au programmeur, des cases ainsi réservées.

Ce sujet est conçu pour être traité linéairement. Les différentes hypothèses et spécifications listées dans cette introduction sont valables et importantes pour les quatre parties du sujet. Chaque partie propose une stratégie différente des fonctions `demarrage`, `reserver` et `liberer` de gestion dynamique de la mémoire. Seule la fonction `initialiser` décrite plus bas est la même dans chaque partie.

Partie I : Implémentation naïve

Dans cette partie, nous organisons notre mémoire `mem` comme une suite contiguë de portions entre les indices `1` et `TAILLE_MEM-1`. La case `mem[0]` joue un rôle spécifique : elle contient un entier `prochain` tel que les portions réservées jusqu'à présent se trouvent parmi les cases `mem[1]` ... `mem[prochain-1]`. Lors de la prochaine réservation, la nouvelle portion sera placée à partir de l'indice `prochain`. Ici, une portion `p` de taille `n` est constituée de `n` cases `mem[p]`, ..., `mem[p+n-1]`. Mais dans cette stratégie naïve d'implémentation, la libération de portion n'a pas d'effet sur `mem`, ce qui est correct mais peu efficace en termes de consommation mémoire : aucun *recyclage* de mémoire n'est possible. La fonction de libération est donc simplement [1] :

```
def liberer(p):
    pass
```

La Figure 1 présente le contenu du début de `mem` après l'appel des services suivants par le programmeur :

```
p1 = reserver(6, 'a')
p2 = reserver(9, 'b')
p3 = reserver(3, 'c')
ecrire(p1, 0, 'A')
```

19	A	a	a	a	a	a	b	b	b	b	b	b	b	b	b	c	c	c	0	0	0	0	0	0	0	0	0	0	0
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29

Figure 1 – Exemple de contenu de la mémoire - Stratégie naïve.

Question 1. Écrire une fonction `initialiser(p, n, c)` qui initialise une portion de taille `n` à l'indice `p` en remplissant chaque case avec le caractère `c`.

Question 2. Écrire la fonction `demarrage()` pour cette stratégie.

Question 3. Écrire la fonction `reserver(n, c)` pour cette stratégie. Préciser la complexité de `reserver(n, c)` en fonction de `n` et `TAILLE_MEM`.

1. En `Python`, `pass` est l'instruction qui ne fait rien.

À partir de maintenant, les seules fonctions manipulant *directement* la liste `mem` seront fournies par l'énoncé. Vos solutions doivent donc s'appuyer sur les fonctions auxiliaires qui vous seront fournies et ne pas contenir de lecture (de la forme `a=mem[p]`) ou écriture (de la forme `mem[p]=a`) directes à `mem`.

Partie II : Réservations de blocs de tailles fixes

Nous cherchons maintenant à permettre la réutilisation de la mémoire libérée. Nous proposons pour cela une nouvelle stratégie d'implémentation. Nous fixons une constante globale `TAILLE_BLOC` et nous placerons les portions à l'intérieur de blocs de taille fixe de `TAILLE_BLOC` cases contiguës dans la liste `mem`. Une portion `p` reservée avec la taille n (où $n+1 \leq$ `TAILLE_BLOC`) occupe $n+1$ cases au début d'un tel bloc. La première case `mem[p-1]` contient un *en-tête* qui vaut 1 si la portion est encore réservée, ou 0 si elle a été libérée. Les cases suivantes sont utilisées pour stocker les caractères écrits par le programmeur. Comme précédemment, la case `mem[0]` est réservée pour pointer sur la prochaine portion libre pour créer un bloc (si aucun recyclage de bloc libéré n'était possible).

La Figure 2 présente le contenu du début de `mem` après l'appel des services suivants par le programmeur :

```
p1 = reserver(6, 'a')
p2 = reserver(4, 'b')
p3 = reserver(3, 'c')
liberer(p2)
ecrire(p1, 0, 'A')
```

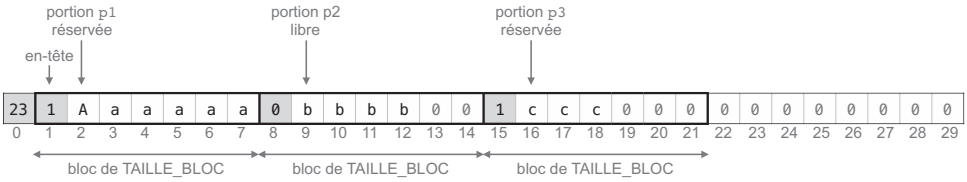

Figure 2 – Exemple de contenu de la mémoire avec `TAILLE_BLOC=7` - Partie II.

Pour manipuler les cases d'en-tête des portions et la prochaine portion libre, nous vous fournissons les fonctions suivantes :

```
def lire_prochain():
    return mem[0]

def ecrire_prochain(p):
    mem[0] = p

def est_reservee(p):
    return mem[p-1] == 1

def est_libre(p):
    return mem[p-1] == 0
```

```
def marque_reservee(p):
    mem[p-1] = 1
```

```
def marque_libre(p):
    mem[p-1] = 0
```

Question 4. Écrire la fonction `demarrage()` pour cette stratégie d'implémentation.

Pour réserver une nouvelle portion, on cherche en priorité à réutiliser un bloc laissé libre par une précédente libération. La portion libre pointée par `lire_prochain()` n'est utilisée que si un tel bloc n'existe pas.

Question 5. Écrire la fonction `reserver(n, c)` pour cette stratégie d'implémentation. Renvoyer `None` si la taille `n` est trop grande vis-à-vis de `TAILLE_BLOC`. Préciser la complexité de `reserver(n, c)` en fonction de `n` et `TAILLE_MEM`.

Question 6. Écrire la fonction `liberer(p)` pour cette stratégie d'implémentation.

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

Nous abordons maintenant une autre stratégie d'implémentation qui permettra de ne pas limiter autant la taille de chaque portion. Nous munissons pour cela chaque portion d'un en-tête mais aussi d'un *pied de page*. Pour chaque portion, ces deux cases additionnelles contiennent la même valeur : un entier encodant deux informations sur la portion courante. La première information indique si la portion est réservée ou pas. La deuxième indique la taille réservée à cette portion. Nous imposons que cette taille soit toujours un entier pair. Si un programmeur demande à réserver une portion de taille n impaire, nous attribuerons une taille $n+1$ à cette portion, mais le programmeur n'est pas autorisé à consulter cet espace supplémentaire, en vertu des hypothèses de bon usage dictées en début de sujet.

Nous vous fournissons toutes les fonctions nécessitant un accès direct à `mem` :

```
def est_reservee(p):
    return mem[p-1] % 2 == 1

def marque_reservee(p, taille):
    mot = taille + 1
    mem[p-1] = mot
    mem[p+taille] = mot

def lire_taille(p):
    return 2 * (mem[p-1] // 2)

def precedent_est_libre(p):
    return mem[p-2] % 2 == 0
```

```
def est_libre(p):
    return mem[p-1] % 2 == 0

def marque_libre(p, taille):
    mot = taille
    mem[p-1] = mot
    mem[p+taille] = mot

def lire_taille_precedent(p):
    return 2 * (mem[p-2] // 2)

def precedent_est_reserve(p):
    return mem[p-2] % 2 == 1
```

Question 7. Expliquer en quelques lignes le fonctionnement des fonctions `est_reservee`, `marque_reservee`, `lire_taille` et `precedent_est_libre`.

Nous utiliserons deux portions spéciales, une *portion prologue* et une *portion épilogue*. Ces deux portions spéciales sont de taille nulle et toujours marquées réservées. Nous les plaçons en début et en fin de la zone de réservation. La portion prologue se situe à une position fixe donnée par la variable globale suivante :

```
PROLOGUE = 2
```

La case `mem[0]` indique la position courante de l'épilogue. L'épilogue est contigu au prologue au démarrage, puis est déplacé vers des indices plus élevés quand la zone des portions réservées doit être agrandie. Nous vous fournissons les fonctions permettant de lire et modifier les informations concernant cette portion spéciale :

```
def lire_position_epilogue():
    return mem[0]
```

```
def ecrire_position_epilogue(p):
    mem[0] = p
```

La Figure 3 présente [2] le contenu du début de `mem` après l'appel des services suivants par le programmeur :

```
p1 = reserver(6, 'a')
p2 = reserver(7, 'b')
p3 = reserver(1, 'c')
liberer(p2)
ecrire(p1, 0, 'A')
```

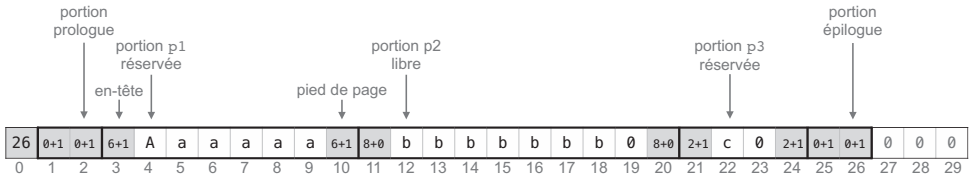

Figure 3 – Exemple de contenu de la mémoire - Partie III.

Question 8. Écrire la fonction `demarrage()` pour cette stratégie.

A part les deux portions spéciales épilogue et prologue, toutes les portions ont une taille strictement positive. Lors de la réservation d'une nouvelle portion, on réserve en priorité dans la zone de mémoire comprise entre le prologue et l'épilogue, et en dernier recours on déplace l'épilogue. Si une portion libre est suffisamment grande, une réservation dans cette zone la sépare en une portion réservée et une portion libre.

2. Dans les en-têtes et pieds de page de ces figures, la notation `t + b` désigne l'encodage d'une paire formée d'une taille `t` et d'un bit d'état de réservation `b` $\in \{0, 1\}$.

La Figure 4 illustre ce mécanisme en présentant le contenu de la mémoire de la Figure 3, après l'appel à `p4 = reserver(2,'d')`.

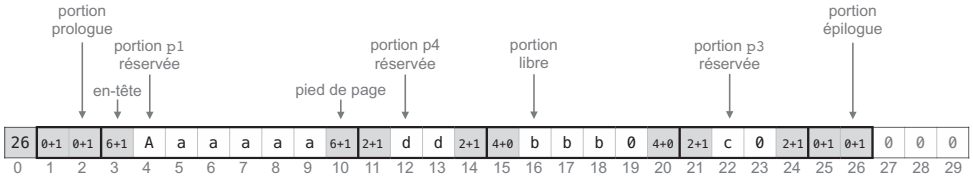

Figure 4 – Exemple de contenu de la mémoire après nouvelle réservation - Partie III.

Question 9. Écrire la fonction `reserver(n, c)` pour cette stratégie. Préciser la complexité de `reserver(n, c)` en fonction de `n` et `TAILLE_MEM`.

Lors de la libération d'une portion, on étudie les portions adjacentes libres et on réalise si possible une *fusion* afin qu'il n'y ait jamais deux portions adjacentes libres après un appel à la fonction `liberer`.

La Figure 5 illustre ce mécanisme en présentant le contenu de la mémoire de la Figure 4, après l'appel à `liberer(p3)`.

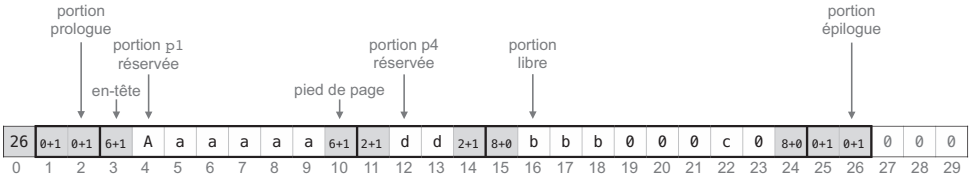

Figure 5 – Exemple de contenu de la mémoire après nouvelle libération - Partie III.

Question 10. Écrire la fonction `liberer(p)` pour cette stratégie. Justifier la correction et la complexité de `liberer(p)` en fonction de `TAILLE_MEM`, en précisant le nombre maximal de fusions effectuées à chaque appel, et en expliquant l'interêt des portions prologues et épilogues.

Partie IV : Chaînage explicite des portions libres

Nous souhaitons maintenant améliorer l'implémentation de la partie précédente pour accélérer la recherche de portions libérées. Nous allons pour cela maintenir une *chaîne des portions libres*. Il s'agit d'une séquence de portions libres organisée de la façon suivante :

- dans chaque portion libre `p` dans la chaîne, on stocke une information dans les cases `mem[p]` et `mem[p+1]` :

  - la case `mem[p]` contient la position de la portion libre prédécesseur dans la chaîne ;
  - la `mem[p+1]` contient la position de la portion libre successeur dans la chaîne ;

- la position de l'*entrée* de la chaîne est stockée dans la case `mem[1]`. Elle vaut `0` si et seulement si la chaîne est vide.
- si la chaîne n'est pas vide, son entrée est une portion dont le prédécesseur vaut `0` ;
- si la chaîne n'est pas vide, elle contient une portion *de fin* (éventuellement égale à celle d'entrée) dont le successeur vaut `0` ;
- toutes les autres portions de la chaîne ont des prédécesseurs et successeurs non nuls.

Pour manipuler cette structure, nous vous fournissons les fonctions suivantes :

```
def lire_entree_chaine():
    return mem[1]

def ecrire_entree_chaine(p):
    mem[1] = p

def lire_predecesseur(p):
    return lire(p, 0)

def lire_successeur(p):
    return lire(p, 1)

def ecrire_predecesseur(p, predecesseur):
    ecrire(p, 0, predecesseur)

def ecrire_successeur(p, successeur):
    ecrire(p, 1, successeur)

def chaine_est_vide():
    return lire_entree_chaine()==0
```

Par ailleurs, on redéfinit la portion prologue comme suit :

```
PROLOGUE = 3
```

La Figure 6 présente le contenu du début de `mem` après l'appel des services suivants par le programmeur :

```
p1 = reserver(2, 'a')
p2 = reserver(2, 'b')
p3 = reserver(2, 'c')
p4 = reserver(2, 'd')
p5 = reserver(1, 'e')
p6 = reserver(1, 'f')
liberer(p1)
liberer(p5)
liberer(p3)
```

La Figure 7 présente la chaîne associée, qui commence par la portion `p3=13` , suivi de `p5=21` et enfin de `p1=5`.

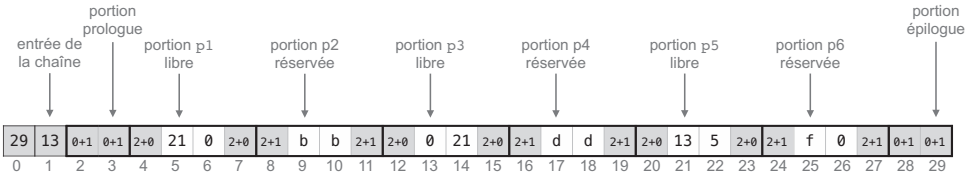

Figure 6 – Exemple de contenu de la mémoire - Partie IV.

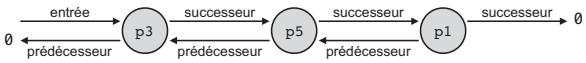

Figure 7 – Chaîne associée à la mémoire de la Figure 6.

Question 11. Écrire la fonction `ajoute_en_entree_de_chaine(p)` qui ajoute la portion libre `p` en tête de la chaîne et en fait son entrée. La portion `p` n'appartient pas à la chaîne au moment de l'appel.

Question 12. Écrire la fonction `supprime_dans_chaine(p)` qui supprime la portion `p` de la chaîne. La portion `p` est libre et appartient à la chaîne.

Question 13. Écrire la fonction `demarrage()` pour cette stratégie.

Question 14. Écrire la fonction `reserver(n, c)` pour cette stratégie. Préciser la complexité de `reserver(n, c)` en fonction de `n`, `TAILLE_MEM`, et tout autre quantité que vous jugerez utile pour mettre en valeur le gain en efficacité de cette stratégie d'implémentation. Comme cette fonction est très similaire à la fonction `reserver(n, c)` écrite pour la partie précédente, vous pouvez indiquer seulement les différences entre cette version et la précédente (en numérotant les lignes de la fonction précédente).

La Figure 8 présente le résultat de l'opération `liberer(p6)` sur la mémoire de la Figure 6.

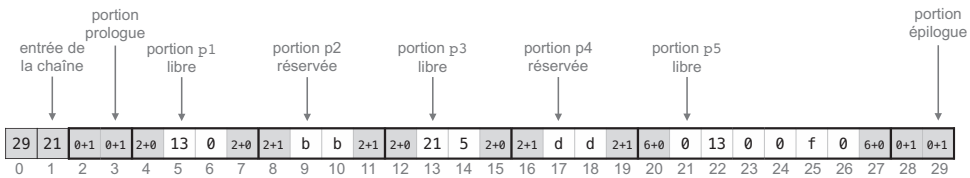

Figure 8 – Exemple de contenu de la mémoire après libération de `p6` - Partie IV.

Pour libérer une portion, on réalise comme dans la partie précédente une fusion avec les éventuelles portions libres adjacentes en mémoire, mais en les supprimant au préalable de la chaîne, puis en ajoutant en entrée de chaîne la portion libre créée par la fusion.

Question 15. Écrire la fonction `liberer(p)` pour cette stratégie. Préciser la complexité de `liberer(p)` en fonction de `TAILLE_MEM`. Là encore, il vous suffit de préciser les différences avec la fonction `liberer(p)` de la partie précédente.

* *
*