TP Allocation mémoire

Le but de ce TP et d'écrire un allocateur dynamique de mémoire utilisant un tableau alloué en début de programme. Afin d'éviter de manipuler différents types de pointeurs, notre allocateur ne renverra que des pointeurs de type `uint64_t *`. Nous utiliserons dans ce TP l'alias `size_t` pour désigner le type `uint64_t`. La signature de notre allocateur sera donc :

```
size_t *my_malloc(size_t n);
void my_free(size_t *p);
```

La fonction `my_malloc` prend en paramètre une taille `n` et renvoie un pointeur `p` de type `size_t *` vers une portion mémoire pouvant acceuillir `n` entiers `size_t`. Si le tas ne dispose plus de place, elle renvoie le pointeur !NULL !. La fonction `my_free` est utilisée avec un pointeur non nul renvoyé par `my_malloc`. Son rôle est de libérer la mémoire.

Afin de gérer notre tas, nous utiliserons les variables globales `heap` et `heap_size` ainsi que les fonctions `alloc_heap` et `free_heap` suivantes.

```
size_t *heap;
size_t heap_size = 32;

void alloc_heap() {
    heap = malloc(heap_size * sizeof(size_t));
}

void free_heap() {
    free(heap);
}
```

Enfin, afin d'initialiser nos zones mémoires, nous utiliserons la fonction suivante.

```
void init_memory(size_t *p, size_t n, size_t value) {
    for (size_t i = 0; i < n; i++) {
        p[i] = value;
    }
}
```

1 Implémentation naïve

Dans cette partie, nous organisons notre mémoire `heap` comme une suite contigüe de portions mémoire entre les indices `1` et `heap_size - 1`. La case !heap[0] ! joue un rôle spécifique : elle contient un entier `end` tel que les portions réservées se trouvent parmi les cases d'indices !1 !, . . . , `end - 1`. Lors de la prochaine allocation, la nouvelle portion sera placée à partir de l'indice `end`. Dans cette stratégie naïve, la libération de mémoire n'a pas d'effet sur le tas. La fonction `my_free` s'écrit donc comme ceci :

```
void my_free(size_t *p) {}
```

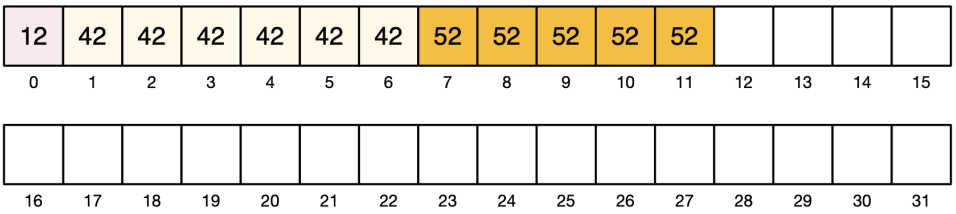
Afin d'initialiser notre structure, c'est-à-dire la première case du tableau `heap`, on écrit une fonction de signature

```
void init_heap();
```

Notre code pourra donc être utilisé de la manière suivante. On crée ici 2 tableaux d'entiers de taille respective 6 et 5 que l'on initialise respectivement avec les valeurs 42 et 52. On libère enfin ces tableaux.

```
alloc_heap();
init_heap();
size_t *p1 = my_malloc(6);
size_t *p2 = my_malloc(5);
init_memory(p1, 6, 42);
init_memory(p2, 5, 52);
my_free(p2);
my_free(p1);
free_heap();
```

Après initialisation de le mémoire, le tableau `heap` est dans l'état suivant. Dans toute la suite de ce TP, une case vide signalera soit une case mémoire qui n'est pas initialisée, soit une case mémoire dont la valeur ne nous intéresse plus.



1. Écrire la fonction `init_heap` qui initialise le tas pour que `heap[0]` contienne une valeur appropriée pour cette stratégie.
2. Écrire la fonction `my_malloc` pour cette stratégie.

2 Réservations de blocs de taille fixe

Nous cherchons désormais à permettre la réutilisation de la mémoire libérée. Nous proposons pour cela une nouvelle stratégie d'implémentation. Nous fixons une variable globale `block_size` et nous placerons les portions mémoires à l'intérieur de blocs ayant une taille fixe de `block_size` cases contigües dans le tableau `heap`.

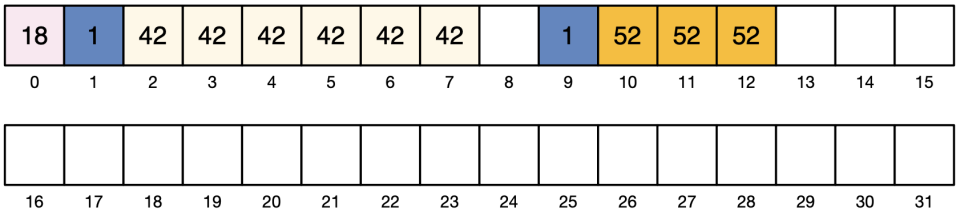
```
size_t block_size = 8;
```

Une portion mémoire `i` réservée avec la taille n (où $n+1 \leqslant$ `block_size`) occupe $n+1$ cases d'un tel bloc. Lorsque nous parlerons de la portion mémoire `i`, cet entier fera référence à l'index de la première case accessible à l'utilisateur. L'*en-tête* sera donc accessible à l'aide de `heap[i - 1]` qui vaut 1 si la portion est encore réservée, ou 0 si elle a été libérée. Les cases suivantes sont utilisées pour stocker les données. Comme précédemment, la case `heap[0]` est réservée pour donner l'index !i! de la prochaine portion libre pour créer un bloc lorsqu'aucun recyclage de bloc libéré n'est possible.

Une fois le tas initialisé et les opérations suivantes effectuées

```
size_t *p1 = my_malloc(6);
size_t *p2 = my_malloc(3);
init_memory(p1, 6, 42);
init_memory(p1, 3, 52);
```

l'état du tas est donné par le tableau ci-dessous. Le pointeur `p1` pointe vers la portion mémoire d'index `i = 2` tandis que le pointeur `p2` pointe vers la portion mémoire d'index `i = 10`.



1. Écrire la fonction `init_heap` pour cette stratégie.
2. Écrire une fonction `bool is_free(size_t i)` renvoyant `true` si la portion mémoire `i` est libre, et `false` sinon. Créez de même les fonctions `void set_free(size_t i)` et `void set_used(uin64_t i)` permettant de changer l'en-tête de la portion mémoire d'index `i`.

Pour réserver une nouvelle portion, on cherche en priorité à réutiliser un bloc laissé libre par une précédente libération. La portion libre pointée par `heap[0]` n'est utilisée que si un tel bloc n'existe pas.

3. Écrire la fonction `my_malloc` pour cette stratégie d'implémentation. On renverra `NULL` lorsque la taille `n` est trop grande vis-à-vis de `block_size`.
4. Écrire la fonction `my_free` pour cette stratégie d'implémentation.

3 Portions mémoire 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 mémoire. Nous munissons pour cela chaque portion d'une 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 zone de taille n impaire, nous attribuerons une taille $n+1$ à cette zone, mais le programmeur n'est pas autorisé à consulter cet espace supplémentaire. Si la taille de la portion est n et la portion est utilisée, l'en-tête et le pied de page contiendront la valeur $n+1$. Si la portion est libre, ils contiendront la valeur n.

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 fixée donnée par la variable globale suivante.

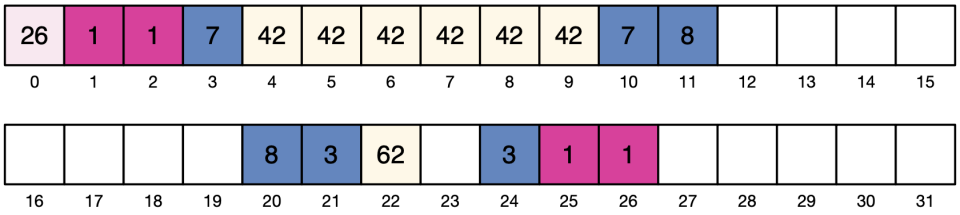
```
size_t prologue = 2;
```

La case `heap[0]` indique l'index 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.

Après initialisation du tas et appels suivants par le programmeur

```
size_t *p1 = my_malloc(6);
size_t *p2 = my_malloc(7);
size_t *p3 = my_malloc(1);
init_memory(p1, 6, 42);
init_memory(p2, 7, 52);
init_memory(p3, 1, 62);
my_free(p2);
```

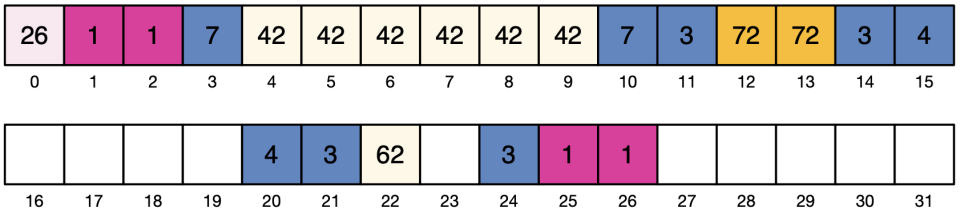
la figure suivante présente le contenu de `heap`.



1. Écrire les fonctions `bool is_free(size_t i)` et `size_t read_size(size_t i)` permettant de savoir si une portion mémoire `i` est libre où non et de connaître sa taille.
2. Écrire les fonctions `void set_free(size_t i, size_t n)` et `void set_used(size_t i, size_t n)` permettant de définir l'en-tête d'une portion de mémoire `i` et de fixer sa taille à `n`.
3. Écrire les fonctions `size_t next(size_t i)` et `size_t previous(size_t i)` permettant respectivement d'obtenir l'index de la portion mémoire suivante et de la portion mémoire précédente. Ces fonctions renverront respectivement les indexs du prologue et de l'épilogue lorsque la portion mémoire `i` est la première ou la dernière zone de notre tas.
4. Écrire la fonction `init_heap` pour cette stratégie.

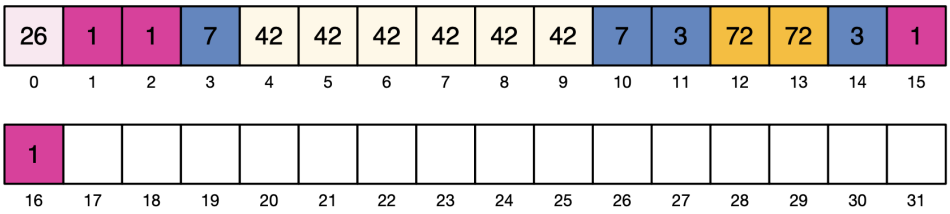
À part les deux portions spéciales épilogue et prologue, toutes les portions ont une taille strictement positive. Lors de la réservation d'une nouvelle portion, on réserve en priorité dans la zone mémoire comprise entre le prologue et l'épilogue, et en dernier recours, on déplace l'épilogue. Si une portion libre est suffisamment grande, une réservation dans cette zone la sépare en une portion réservée et une portion libre. La figure suivante illustre ce mécanisme en présentant le contenu de la mémoire de la figure précédente après l'appel

```
size_t *p4 = my_malloc(2);
init_memory(p4, 2, 72);
```



5. Écrire la fonction `my_malloc` pour cette stratégie.

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 `my_free`. La figure ci-dessous illustre ce mécanisme en présentant le contenu de `heap` après l'appel de `my_free(p3)`.



6. Écrire la fonction `my_free` pour cette stratégie. Expliquer l'intérêt des portions prologues et épilogues.

4 Chaînage explicite des portions libres

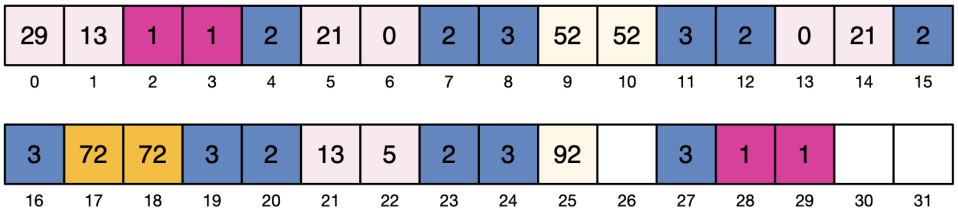
Nous souhaitons maintenant améliorer l'implémentation de la partie précédente pour accélerer la recherche de portions libérées. Nous allons pour cela maintenir une *chaîne des portions libres*. Il s'agit d'une séquence de portions libres organisée de la manière suivante.

— Dans chaque portion libre `i` dans la chaîne, on stocke une information dans les cases `heap[i]` et `heap[i + 1]`.
  — La case `heap[i]` contient l'index de la portion libre prédécesseur dans la chaîne.
  — La case `heap[i + 1]` contient l'index de la portion libre successeur dans la chaîne.
— L'index de l'entrée de la chaîne est stockée dans la case `heap[1]`. Par convention, elle vaut 0 si et seulement si la chaîne est vide. Si la chaîne n'est pas vide, son premier élément est une portion dont le prédécesseur vaut 0. De même, elle contient un dernier élément, éventuellement égal au premier, dont le successeur vaut 0. Toutes les autres portions de la chaîne ont des prédécesseurs et sucesseurs non nuls.

Après initialisation du tas et appels suivants par le programmeur

```
size_t *p1 = my_malloc(2);
size_t *p2 = my_malloc(2);
size_t *p3 = my_malloc(2);
size_t *p4 = my_malloc(2);
size_t *p5 = my_malloc(1);
size_t *p6 = my_malloc(1);
init_memory(p1, 2, 42);
init_memory(p2, 2, 52);
init_memory(p3, 2, 62);
init_memory(p4, 2, 72);
init_memory(p5, 1, 82);
init_memory(p6, 1, 92);
my_free(p1);
my_free(p5);
my_free(p3);
```

la figure suivante présente le contenu de `heap`.



1. Écrire la fonction `void add_begin_chain(size_t i)` qui ajoute la portion libre `i` en tête de la chaîne et en fait son premier élément, la portion `i` n'appartenant pas à la chaîne au moment de l'appel.
2. Écrire la fonction `void remove_from_chain(size_t i)` qui supprime la portion libre `i` de la chaîne.
3. Écrire la fonction `init_heap` pour cette stratégie.
4. Écrire la fonction `my_malloc` pour cette stratégie.

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 la chaîne la portion libre créée par la fusion.

5. Écrire la fonction `my_free` pour cette stratégie.
6. Commenter les avantages de cette stratégie par rapport à la stratégie précédente.