Identification /5

Prénom: .....

Nom: .....

Code permanent: .....

Veuillez choisir **une seule réponse** par question, avec une croix (×) ou un crochet (✓).

Q1. Que se passera-t-il si on exécute le programme suivant?

```
#include <stdbool.h>
#include <stdlib.h>

int main(void) {
  unsigned int n = 1;
  int* i = malloc(n * sizeof(int));
  while (i != NULL) {
    n *= 2;
    i = realloc(i, n * sizeof(int));
  }
  free(i);
  return 0;
}
```

- ☐ Il accèdera à de la mémoire non réservée
- ☐ Il déclenchera une erreur de segmentation
- ■ Il entraînera une fuite de mémoire
- ☐ Son exécution sera normale

Q2. Parmi les types concrets suivants, lequel semble le moins adapté pour implémenter une pile?

- ■ Un arbre binaire de recherche
- ☐ Une liste simplement chaînée
- ☐ Un tableau dynamique
- ☐ Un tableau statique

Q3. Considérez les déclarations suivantes permettant d'implémenter une file d'attente à l'aide d'une liste doublement chaînée:

```
struct QueueNode {
  // Valeur du noeud
  unsigned int value;
  // Noeud precedent
  struct QueueNode* prev;
  // Noeud suivant
  struct QueueNode* next;
};

typedef struct {
  // Premier noeud de la file
  struct QueueNode* first;
  // Dernier noeud de la file
  struct QueueNode* last;
  // Nombre de noeuds dans la file
  unsigned int size;
} Queue;
```

On souhaite maintenir les invariants suivants, pour toute file `f` et pour tout noeud `n` de la file `f`:

- `f.first` pointe vers le prochain noeud à défiler de la file
- `f.first` est `NULL` si et seulement si la file est vide
- `f.last` pointe vers le dernier noeud enfilé dans la file
- `f.last` est `NULL` si et seulement si la file est vide
- `f.size` est égal au nombre de noeuds dans la liste doublement chaînée
- `n.prev` est `NULL` si et seulement si `n` est le premier noeud de la file
- `n.next` est `NULL` si et seulement si `n` est le dernier noeud de la file

Proposez une implémentation de la fonction

```
unsigned int void queue_pop(Queue* q)
```

qui retire le premier noeud de la file `q` et qui retourne la valeur correspondante, tout en maintenant les invariants mentionnés ci-haut. Utilisez le verso de la feuille pour répondre.

Voir page suivante

Q1. D'une part, rappelons que les fonctions `malloc` et `realloc` retournent `NULL` lorsqu'il n'y a plus de mémoire disponible. D'autre part, lorsqu'on appelle la fonction `free` avec `NULL`, il n'y a pas de problème ni de comportement indéterminé: il ne se passe tout simplement rien. Dans le programme ci-haut, une première observation importante est que la fonction `free` est appelée avec `NULL`, peu importe qu'on entre ou non dans la boucle `while`, étant donné que la condition d'arrêt de la boucle est justement que `i` soit égal à `NULL`. Par conséquent, ce programme ne libère jamais d'espace mémoire. On en conclut donc que si on fait au moins un tour de boucle, on aura une fuite de mémoire. Aussi, on observe que le programme n'utilise jamais de mémoire non réservée (aucun problème de ce côté) et ne déclenche pas d'erreur de segmentation, car il n'utilise pas de déréférencement.

Q2. Nous avons vu en classe qu'on pouvait implémenter une pile à l'aide d'une liste simplement chaînée. Il est aussi possible d'utiliser un tableau statique ou un tableau dynamique, en utilisant le dernier élément comme sommet de pile. Dans le cas d'une pile, l'arbre binaire de recherche n'est pas du tout adapté, car il n'y a pas de concept de clé qui ait du sens. On pourrait artificiellement tout de même stocker les valeurs de la pile toujours dans le sous-arbre gauche (ou le sous-arbre droit), mais ça reviendrait à implémenter une liste simplement chaînée: étant donnée la plus grande complexité des opérations agissant sur un arbre binaire de recherche, ce n'est pas une approche efficace.

Q3. Avant de répondre à cette question, il est important de bien réfléchir aux différents cas qui peuvent survenir. Par exemple, lorsque la file est vide, l'opération `queue_pop` ne peut pas être appliquée. On devrait s'attendre aussi à traiter séparément le cas où la file contient un seul élément, car alors il faut s'assurer que les champs `first` et `last` deviennent `NULL` après avoir retiré l'élément. Dans le cas où il y a plus d'un élément, il faut s'assurer de mettre correctement à jour le champ `first` pour qu'il référence le deuxième noeud, puis mettre à jour le champ `prev` du nouveau premier noeud pour maintenir l'invariant correspondant. Finalement, on ne doit pas oublier de libérer l'espace mémoire dont on n'a plus besoin avec la fonction `free`. Dans tous les cas, on n'oublie pas de mettre à jour la taille de la file et on retourne l'élément qui a été défilé.

Lorsqu'il y a plusieurs cas à traiter, il est souvent préférable de traiter les cas les plus faciles en premier. En suivant cette logique-là, on arrive à une solution similaire à celle-ci:

```
unsigned int queue_pop(Queue* q) {
  if (q->size == 0) {
    fprintf(stderr, "error: cannot pop from empty queue");
    exit(1);
  }
  struct QueueNode* node = q->first;
  unsigned int value = node->value;
  if (q->size == 1) {
    q->first = NULL;
    q->last = NULL;
  } else {
    q->first = q->first->next;
    q->first->prev = NULL;
  }
  free(node);
  --q->size;
  return value;
}
```

Le code peut être un peu simplifié en observant que l'affectation à `q->first` est la même dans le cas où la file contient un élément que dans le cas où elle en contient 2 ou plus. On obtient alors la fonction suivante, qui est un peu plus courte:

```
unsigned int queue_pop(Queue* q) {
  if (q->size == 0) {
    fprintf(stderr, "error: cannot pop from empty queue");
    exit(1);
  }
  struct QueueNode* node = q->first;
  unsigned int value = node->value;
  q->first = q->first->next;
  if (q->size == 1)
    q->last = NULL;
  else
    q->first->prev = NULL;
  free(node);
  --q->size;
  return value;
}
```