**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 <stdio.h>
#include <stdlib.h>

int main(void) {
  unsigned int n = 1;
  int* i = malloc(n * sizeof(int));
  if (i != NULL)
    printf("%d\n", *i);
  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**. On souhaite implémenter une structure de données d'ensemble dont les éléments sont des chaînes de caractères et pour laquelle l'opération de vérification d'appartenance d'un élément à l'ensemble doit être efficace. Parmi les types concrets suivants, lequel est le moins adapté?

- ☐ Un arbre binaire de recherche
- ■ Une liste triée doublement chaînée
- ☐ Un tableau trié dynamique
- ☐ Un tableau trié 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

```
void queue_push(Queue* q, unsigned int value)
```

qui enfile la valeur `value` dans la file `q` en maintenant les invariants mentionnés ci-haut. Vous n'avez pas à gérer le cas où l'allocation dynamique échoue en raison d'un manque de mémoire. 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, si `malloc` échoue, alors `i` sera `NULL`, on n'effectuera aucun affichage avec `printf`, puis on appellera `free` avec `NULL`, ce qui ne pose aucun problème. Si `malloc` réussit, alors on affichera une valeur quelconque sur la sortie standard (car `*i` n'a pas été initialisée), puis on libèrera l'espace mémoire réservé. Bref, le programme aura une exécution normale, peu importe le cas.

**Q2**. Comme on souhaite représenter un ensemble d'entier et qu'on veut pouvoir trouver rapidement une valeur, on doit minimalement maintenir les valeurs triées pour faciliter la recherche en utilisant une fouille binaire. L'arbre binaire de recherche et les tableaux (statique ou dynamique) sont bien adaptés à cette situation. Par contre, les listes (simplement ou doublement) chaînées sont moins adaptées, car on n'a pas facilement un accès direct à un élément quelconque: il faut parcourir plusieurs noeuds de façon linéaire avant de repérer l'élément qui nous intéresse. C'est donc la liste doublement chaînée qui est la moins adaptée à la situation.

**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_push` doit gérer différemment le champ `first`, alors que lorsqu'il y a au moins un noeud dans la file, il faut bien rattacher l'ancien dernier noeud au nouveau dernier noeud. On devrait aussi idéalement traiter le cas où la fonction `malloc` échoue. Dans tous les cas, on n'oublie pas de mettre à jour la taille de la file.

En suivant cette logique-là, on arrive à une solution similaire à celle-ci:

```
void queue_push(Queue* q, unsigned int value) {
  struct QueueNode* node = malloc(sizeof(struct QueueNode));
  if (node == NULL) {
    fprintf(stderr, "error: out of memory\n");
    return;
  }
  node->value = value;
  node->next = NULL;
  if (q->size == 0) {
    node->prev = NULL;
    q->first = node;
    q->last = node;
  } else {
    node->prev = q->last;
    q->last->next = node;
    q->last = node;
  }
  ++q->size;
}
```

On observe qu'il y a une certaine redondance dans les blocs conditionnels. On peut donc simplifier un peu comme suit:

```
void queue_push(Queue* q, unsigned int value) {
  struct QueueNode* node = malloc(sizeof(struct QueueNode));
  if (node == NULL) {
    printf("error: out of memory\n");
    return;
  }
  node->value = value;
  node->prev = q->last;
  node->next = NULL;
  if (q->size == 0)
    q->first = node;
  else
    q->last->next = node;
  q->last = node;
  ++q->size;
}
```