Cours de programmation séquentielle

File de priorité

1 Buts

— Introduction à la file de priorité.
— Utilisation de liste chaînée.
— Utilisation des pointeurs.
— Utilisation pointeurs de fonctions.

2 Introduction théorique

Une file de priorité (ou priority queue en anglais) est une structure abstraite de file un peu particulière. Chaque élément possède une priorité associée. Les éléments sont retirés en fonction de leur priorité:

— pour une file *max*, on retire l'élément avec la priorité la plus élevée en premier;
— pour une file *min*, on retire l'élément avec la priorité la plus faible en premier.

Ici, nous nous concentrerons sur les files de priorités *max*.

En général, il existe trois fonctions importantes sur les files (outre leur création et leur destruction):

1. La fonction `is_empty()` permettant de vérifier si la file est vide
2. La fonction `push()` permettant d'insérer un élément.
3. La fonction `pop()` permettant dde retourner l'élément avec la plus haute priorité.

On supplémente souvent ces trois opérations avec la quatrième fonction suivante

4. La fonction `peek()` permettant de retourner une copie de l'élément avec la plus haute priorité.

Les files de priorités sont utilisées pour différents algorithmes comme le calcul du plus court chemin dans un graphe (algorithme de Dijkstra), les simulations à événements discrets, ou encore l'algorithme A*.

Il existe différentes implémentations des files de priorité, avec des structures de données sous-jacentes différentes. En fonction de l'implémentation l'efficacité (ou la complexité) des opérations ci-dessus peut varier. Dans ce travail, vous allez implémenter une file de priorité à l'aide d'une liste chaînée.

3 Énoncé

3.1 File de priorité d'entiers

La file de priorité d'entiers n'est autre qu'une liste chaînée

```
typedef struct _element {
   int data;
   struct _element* next;
} element;

typedef element* priority_queue;
```

Chaque élément de notre liste, contient des données (`data`) et un pointeur vers l'élément suivant.

Puis il faut implémenter les fonctions suivantes:

1. La fonction `priority_queue_init(priority_queue *pq, int val)` qui initialise une file de priorité contenant un élément contenant la valeur `val`.
2. Une fonction `priority_queue_is_empty(priority_queue *pq)` qui détermine si la file de priorité est vide.
3. Une fonction `priority_queue_push(priority_queue *pq, int element, bool (* cmp)(int, int` qui ajoute `element` en fonction de sa priorité qui est déterminée par la fonction `cmp`.
4. La fonction `priority_queue_pop(priority_queue *pq, int *element)` qui retire le premier élément de la file de priorité et le stocke dans `element`.
5. La fonction `priority_queue_peek(priority_queue *pq, int *element)` qui copie le premier élément de la file dans `element`.
6. La fonction `priority_queue_empty(priority_queue *pq)` qui vide un vecteur et libère la mémoire.

La fonction `priority_queue_push()` a une particularité: elle prend la fonction `cmp()` en argument. Elle permet de comparer l'élément que vous souhaitez insérer avec l'élément courant de votre file d'attente. Pour une file *max* d'entiers, la fonction `cmp()` n'est autre que la fonction `>`: elle retourne vrai, si l'élément à ajouter est plus grand que l'élément courant de la file, faux sinon. La fonction `priority_queue_push()` fonctionne donc de la façon suivante

1. Si `element` est plus grand que le premier élément de la file, on insère élément avant le premier élément.
2. Sinon on parcourt les éléments de la file jusqu'à ce que `element` soit plus grand que l'élément courant de la file (ou jusqu'à ce qu'on atteigne le dernier élément de la file) et on insère `element`.

3.1.1 Complexité algorithmique

La complexité d'un algorithme est le temps que prend l'algorithme pour s'exécuter. Dans le cas de la file de priorité chacune fonction ci-dessus a une complexité algorithmique différente, mais on peut l'exprimer en fonction de la longueur N de

la file. Ainsi la fonction `priority_queue_is_empty(priority_queue *pq)` est dite de complexité constante, car peu importe la longueur de la file cela prendra le temps que prend une comparaison pour savoir si elle est vide ou non. En revanche, pour pour insérer un élément cela se complique un peu. Pourriez-vous donner une asymptote vers laquelle tend la complexité de la fonction `push()` pour une liste de longueur N uniquement en fonction de N? Quand vous aurez trouvé demandez à M. Künzli ou à M. Chassot si vous avez fait juste ou non. Si vous ne comprenez pas la question essayez d'abord de vous documenter un peu (p.ex. ici). Si cela ne va toujours pas demandez au personnel enseignant.

3.2 File de priorité générique

Une fois cette première partie terminée, vous devrez implémenter une file de priorité générique. Cela signifie que le type de `data` ne sera plus `int` mais `void *`. Comme ce que vous ajoutez est maintenant un pointeur, les choses se compliquent un peu pour la gestion de la mémoire, mais pas tant que ça. Pour la gestion des erreurs cela simplifie même pas mal de choses.

Les fonctions à implémenter cette fois sont:

```
typedef struct _g_element {
   void *data;
   struct _g_element* next;
} g_element;

typedef g_element* priority_queue;
```

Chaque élément de notre liste, contient des données (`data`) et un pointeur vers l'élément suivant.

Puis il faut implémenter les fonctions suivantes:

1. La fonction `g_priority_queue_init(priority_queue *pq, void *val)` qui initialise une file de priorité contenant un élément contenant le pointeur `val`.
2. Une fonction `g_priority_queue_is_empty(priority_queue *pq)` qui détermine si la file de priorité est vide.
3. Une fonction `g_priority_queue_push(priority_queue *pq, void *element, bool (* cmp)(voi` qui ajoute `element` en fonction de sa priorité qui est déterminée par la fonction `cmp`.
4. La fonction `g_priority_queue_pop(priority_queue *pq)` qui retire le premier élément de la file de priorité et le retourne. Si la file est vide il faut retourner `NULL`. En effet, `NULL` est un pointeur ne pouvant jamais être lu (déréférencé) et permet de vérifier si nous n'avons pas fait une opération interdite (comme faire un `pop` sur une file vide).
5. La fonction `g_priority_queue_peek(priority_queue *pq)` qui copie le premier élément de la file et le retourne. Gérez correctement le cas où la file est vide.
6. La fonction `g_priority_queue_empty(priority_queue *pq, void (*custom_free)(void *))` qui vide un vecteur et libère la mémoire.

Les fonctions 1-5 se comportent de façon très similaire à la version "entiè-

re" de la file de priorité. La différence principale se situe dans la fonction `g_priority_queue_empty()`. On constate ici, qu'on passe en argument une fonction `custom_free()` qui s'occupera de libérer la mémoire de chaque élément de la file.