Année 2014-2015 Session 1

Logo_ENSEIRB-MATMECA.png

Atelier algorithmique et programmation
PG116

Filière : Informatique, Année : 1

Date de l'examen : 06/03/2015 Durée de l'examen : 30mn+1h

Sujet de : G. Eyrolles, F. Herbreteau, L. Simon, B. Bramas

Documents ☐ autorisés ☒ non autorisés Calculatrice ☐ autorisée ☒ non autorisée

Exercice 1 (5 points)

Nous considérons le TAD `fifo` muni des opérations :

```
empty    : fifo
push     : fifo * int -> fifo
top      : fifo -> int
pop      : fifo -> fifo
is_empty : fifo -> bool
```

avec les axiomes suivants :

```
top(empty)           undefined
top(push(empty, x))  = x
top(push(f, x))      = top(f)            [is_empty(f) = false]

pop(empty)           undefined
pop(push(empty, x))  = empty
pop(push(f, x))      = push(pop(f), x)   [is_empty(f) = false]

is_empty(empty)      = true
is_empty(push(f, x)) = false
```

1. La traduction litérale des opérations `pop` et `push` conduit aux prototypes de fonctions C suivants :

   ```
   struct fifo * fifo__pop(struct fifo const *);
   struct fifo * fifo__push(struct fifo const *, int);
   ```

   Des cas similaires ont été vus en TD. Quels choix de prototypes avons-nous faits et pourquoi ?
2. Quel prototype de la fonction `set__empty` permet d'éviter de définir la structure `struct fifo` dans le fichier en-tête ?
3. Quelle fonction doit être ajoutée pour gérer la mémoire ?
4. Écrire le fichier `fifo.h` en suivant les réponses aux 3 questions précédentes.
5. Dans l'exercice 3, la mise en œuvre du TAD `fifo` est faite à l'aide de la structure `struct link` définie dans la feuille de TD 2 (rappelée en annexe A).

   Expliquer précisément pourquoi nous allons utiliser le couple de fonctions `lnk__add_tail`/`lnk__remove_head` plutôt que le couple `lnk__add_head`/`lnk__remove_tail` dans la réalisation des fonctions `fifo__push` et `fifo__pop` ?

♦

Exercice 2 (5 points)

Nous souhaitons stocker des données de type quelconque dans les listes chaînées, par exemple `struct complex` définie ci-dessous. Pour cela, nous remplaçons `int data` par `void * data` dans la définition dans la définition de `struct lelement`.

```
struct complex {
  long real;
  long imaginary;
};
```

1. Expliquer pourquoi il est nécessaire, dans ce cas, d'utiliser un pointeur pour le champ `data`.

Pour la nouvelle version de la liste chaînée, nous cherchons à écrire la fonction `int lnk__count(struct link const *, void const * sdata)` qui calcule le nombre d'occurences de la donnée `sdata` dans la structure chainée.

2. Expliquer pourquoi il n'est pas possible de calculer directement l'égalité sur la valeur des données (par exemple l'égalité sur des `int` ou d'égalité sur les champs de `struct complex`).
3. On peut tester l'égalité des données en utilisant un pointeur de fonction. Donnez le prototype correspondant de la fonction `lnk__count`.
4. Définir une fonction d'égalité pour le type `struct complex` compatible avec la fonction `lnk__count`.

♦

Exercice 3 (10 points)

Exercice sur machine

Écrire la réalisation du TAD `fifo` dans le fichier `fifo.c` fourni.
Dans ce fichier `fifo.c`, la représentation du type fifo est définie par la structure suivante :

```
struct fifo {
  struct link * list;
};
```

Cette mise en œuvre doit
— respecter le prototype des fonstions du fichier en-tête `fifo.h` fourni (et rappelé dans le fichier `fifo.c`),
— faire appel aux fonctions du module `link` pour la gestion du chainage (voir annexe A).
— gèrer la mémoire par allocation dynamique,
— utiliser des assertions pour les axiomes indéfinis (« undefined ») et les cas d'erreur.

Pour vérifier ce travail, nous vous fournissons des tests unitaires. Ces tests comparent votre réalisation avec une mise en œuvre de référence qui respecte les consignes données. Le test d'une de vos fonctions fait appel aux autres fonctions de la mise en œuvre de référence.

Si vous rencontrer des difficultés à faire passer les tests pour une fonction, il suffit de la mettre en commentaire avant de passer à la suivante.

♦

Accès à la machine examen.

Suivre les étapes suivantes :

1. **Se connecter sur la machine examen :**
   (a) **Entrer le login et le password sans se connecter**
   (b) **Choisir « Machine virtuelle examens » dans le menu en haut à droite**
   (c) **Se connecter**
2. **Créer un répertoire de travail à la racine du compte et aller dans ce répertoire :**

   ```
   mkdir ~/travail
   cd ~/travail
   ```

3. **Une fois l'examen démarré, récupérer les fichiers :**

   ```
   make -f ../sujets/Makefile
   ```

   Cette commande effectue les opérations suivante dans le répertoire courant : copie le fichier à compléter (s'il n'est pas présent), crée l'exécutable et l'exécute.

   Pour cette examen, vous devez avoir deux fichiers dans votre répertoire un fichier source `fifo.c` et un exécutable `fifo`.

   Les fichiers `fifo.h` et `link.h` se trouvent dans le répertoire `../sujets/fifo`.
4. **Durant l'examen, vous pouvez compiler et tester votre code avec la commande ci-dessus. Pensez à lancer régulièrement la commande `rendu`.**

A Fichier en-tête « link »

```
#ifndef __LINK_H__
#define __LINK_H__

struct lelement {
  int data;
  struct lelement * next;
};

struct link {
  struct lelement * head;
  struct lelement * tail;
};

struct link lnk__empty(void);

void lnk__add_head(struct link *, struct lelement *);
struct lelement * lnk__remove_head(struct link *);

void lnk__add_tail(struct link *, struct lelement *);
struct lelement * lnk__remove_tail(struct link *);

int lnk__is_end_mark(struct link const *, struct lelement const *);

#endif
```