TP #15 — Listes impératives par listes chaînées en OCaml & C

Le but de ce petit sujet est d'implémenter une structure de données de liste impérative de trois façons différentes (pourquoi s'arrêter à une ?) : avec une liste *simplement chaînée* en OCaml, en C, et avec une liste *doublement* chaînée en OCaml.
On commence par rappeler les spécifications de la structure de liste impérative que l'on souhaite implémenter ; on souhaite disposer :

- d'un constructeur de liste vide ;
- d'une fonction *is_empty* qui teste si son argument est une liste vide ;
- d'un transformateur *insert_head* : paramètres : une valeur v, une liste x ; modifie x pour y ajouter un élément contenant v en tête ;
- d'un transformateur *remove_head* : paramètres : une liste x ; modifie x pour en supprimer le premier élément (et éventuellement le renvoyer) s'il existe ;
- d'un accesseur *get_head* : paramètres : une liste x ; renvoie le premier élément de x si non vide ;
- d'un accesseur *get_tail* : paramètres : une liste x ; renvoie la queue xs de x (la liste formée de tous ses éléments sauf sa tête) ;
- d'un transformateur *insert_after* : paramètres : une valeur v, une élément e d'une liste x ; modifie x pour y ajouter un élément contenant v entre e et son (éventuel) successeur ;
- d'un transformateur *remove* : paramètres : une liste x, un élément e de x ; modifie x pour en supprimer l'élément e ;
- d'un accesseur *next* : paramètres : un élément e d'une liste x ; renvoie l'élément suivant e s'il existe.

Exercice 1. *Liste simplement chaînée en OCaml*
On définit les deux types et l'exception suivante :

```
type 'a slist_e = { mutable dat : 'a;
                    mutable nxt : 'a slist_e option }
type 'a  slist = 'a slist_e option ref
exception Empty
```

Une `'a slist` vide est simplement une référence sur `None` ; une liste non vide est une référence vers (`Some` de) sa tête (son premier élément (ou « maillon ») de type `'a slist_e` ; tout élément `'a slist_e` contient une référence vers une option sur son éventuel successeur dans la liste.

1. Écrivez une fonction OCaml :
   `make_empty : unit -> 'a slist`

de construction de liste vide.

2. Écrivez une fonction OCaml :

   `is_empty : 'a slist -> bool`

   qui teste si son argument est une liste vide.
3. Écrivez une fonction OCaml :

   `insert_head : 'a -> 'a slist -> unit`

   qui modifie son second argument en y ajoutant (un élément de valeur `dat` égale à) son premier en tête.
4. Testez.
5. Écrivez une fonction OCaml :

   `insert_after : 'a -> 'a slist_e -> unit`

   qui intercale (un élément de valeur `dat` égale à) son premier argument entre son second et le successeur (`nxt`) de ce dernier.
6. Testez.
7. Écrivez une fonction OCaml :

   `remove_head : 'a slist -> unit`

   qui modifie son argument pour en supprimer la tête (c'est à dire qu'après un appel `remove_head x`, `x` doit avoir été modifiée pour contenir la queue qu'elle avait avant appel).
   De plus, cette fonction doit lever une exception `Empty` si son argument est une liste vide.
8. Testez.
9. Écrivez une variante silencieuse (ou idempotente) :

   `remove_head_s : 'a slist -> unit`

   identique à `remove_head` si ce n'est qu'elle ne fait rien si son argument est une liste vide. Cette fonction devra faire appel à `remove_head` et rattraper l'éventuelle exception que celle-ci peut lever.
10. Testez.
11. Écrivez une fonction OCaml :

    `get_head : 'a slist -> 'a slist_e`

    qui s'évalue en la tête de son argument si celui-ci est non vide, et lève une exception `Empty` sinon.
12. Écrivez une fonction OCaml :

    `get_head_dat : 'a slist -> 'a`

    qui s'évalue en la valeur du champ `dat` la de tête de son argument si celui-ci est non vide, et lève une exception `Empty` sinon.
13. Testez.
14. Écrivez une fonction OCaml :

    `remove_get_head : 'a slist -> 'a slist_e`

    qui lève une exception `Empty` si son argument est une liste vide, et sinon :

— s'évalue en sa tête ;
— le modifie pour en supprimer la tête.

15. Écrivez une fonction OCaml :

`remove_get_head_dat : 'a slist -> 'a`

qui lève une exception `Empty` si son argument est une liste vide, et sinon :

— s'évalue en la valeur du champ `dat` de sa tête ;
— le modifie pour en supprimer la tête.

16. Testez.

17. Écrivez une fonction OCaml :

`get_tail : 'a slist -> 'a slist`

qui lève une exception `Empty` si son argument est vide, et sinon s'évalue en la queue de celui-ci.

18. Écrivez également une variante silencieuse :

`get_tail_s : 'a slist -> 'a slist`

19. Testez.

20. Écrivez une fonction OCaml :

`iter : ('a -> 'a) -> 'a slist -> unit`

qui lève une exception `Empty` si son second argument est vide, et sinon modifie tous les éléments de celui-ci en changeant les valeurs de leurs champs `dat` en leurs images par son premier argument. Ces modifications devront être faites dans l'ordre de la tête vers la queue (c'est à dire que le premier argument devra être évalué pour la première fois sur la (valeur du champ `dat` de la) tête du second argument, etc.).

21. Testez avec un exemple rigolo.

22. Écrivez une fonction OCaml

`remove : 'a slist_e -> 'a slist -> unit`

qui modifie son second argument pour en supprimer un élément égal à son premier argument si un tel élément y apparaît, et lève une exception `Not_found` sinon.

Il pourra être utile de définir une ou plusieurs fonctions auxiliaires.

23. Testez.

24. Écrivez une variante silencieuse :

`remove_s : 'a slist_e -> 'a slist -> unit`

25. Testez.

Exercice 2. *Liste simplement chaînée en C*

Remarque préliminaire générale : vous êtes libres de traiter à votre guise tout cas non couvert par les spécifications des fonctions demandées (par exemple, si un argument est une valeur de pointeur nul).

On commence par définir le type :

```
struct slist_e
{
    int dat;
    struct slist_e *nxt;
};
```

Une liste simplement chaînée est alors représentée par un objet de type `struct slist_e *` : une liste vide est simplement une valeur de pointeur nul, et une liste non vide a pour valeur l'adresse de sa tête (son premier élément) de type `struct slist_e` ; tout élément `struct slist_e` contient dans son champ `nxt` l'adresse de son éventuel successeur dans la liste, et une valeur de pointeur nul s'il n'y en a pas.

On définit également la fonction utilitaire :

```
void die(char *msg)
{
    fputs(msg, stderr);
    exit(EXIT_FAILURE);
}
```

qui affiche (sur la sortie d'erreur standard) son argument (de type « chaîne de caractères ») et termine le programme sur une erreur.

1. Écrivez une fonction C de signature :
   `void *ckd_malloc(size_t n)`
   qui tente d'allouer un objet de `n` octets de durée de stockage allouée et renvoie un pointeur contenant son adresse. Dans le cas où cette allocation ne s'est pas déroulée avec succès, cette fonction devra afficher un message sur la sortie d'erreur standard et terminer sur une erreur.

Dans toute la suite de l'exercice, on s'astreindra à utiliser exclusivement `ckd_malloc` pour toute création d'objet de durée de stockage allouée (pour toute « allocation sur le tas »).

2. Écrivez une fonction C de signature :
   `struct slist_e *make_empty(void)`
   de construction de liste vide.
3. Écrivez une fonction C de signature :
   `void insert_head(int v, struct slist_e **x)`

qui modifie la liste dont son second argument contient l'adresse afin d'y ajouter en tête un élément dont la valeur du champ `dat` est égale à son premier argument.

4. Testez.
5. Écrivez une fonction C de signature :

   ```
   void insert_after(int v, struct slist_e *x)
   ```

   qui intercale un (élément de valeur `dat` égale à) son premier argument entre l'élément dont son second argument contient l'adresse et l'éventuel successeur de ce dernier.
6. Testez.
7. Écrivez une fonction C de signature :

   ```
   bool remove_head(struct slist_e **x)
   ```

   qui renvoie `false` si son argument pointer vers une liste vide, et sinon modifie la liste pointée pour en supprimer la tête et renvoie `true`.
8. Testez.
9. Écrivez une fonction C de signature :

   ```
   struct slist_e *get_head(struct slist_e *x)
   ```

   qui renvoie la tête de la liste représentée par son argument si celle-ci est non vide, et une valeur de pointeur nul sinon. (Vous pouvez également écrire une variante `get_head_dat` si vous le souhaitez, mais attention à la gestion d'erreur !)
10. Écrivez une fonction C de signature :

    ```
    void destroy(struct slist_e **x)
    ```

    qui met fin à la durée de vie (« libère la mémoire ») de la liste dont l'adresse est donnée par son premier argument, ainsi qu'à celle de tous les éléments qu'elle contient (éventuellement).

    Prenez bien garde aux fuites mémoires et aux *use after free*. Aucun n'est acceptable (pas même les « petits *use after free* »).
11. Testez, y compris pour l'absence de fuites mémoires et de *use after free* (du mieux que vous le pouvez).
12. Écrivez une fonction C de signature :

    ```
    void print_content(struct slist_e *x)
    ```

    qui affiche tous les éléments contenus dans la liste représentée par son argument, dans l'ordre de la tête vers la queue.
13. Testez.
14. Écrivez une fonction C de signature :

    ```
    bool remove_elem(struct slist_e *e, struct slist_e **x)
    ```

    qui supprime l'élément dont son premier argument contient l'adresse de la liste dont son second argument contient l'adresse s'il s'y trouve et renvoie `true` dans ce cas, et renvoie `false` sinon. Dans tous les cas, cette fonction

doit également mettre fin à la durée de vie (libérer la mémoire) de l'objet pointé par son premier argument.

15. Testez, notamment pour l'absence de fuite mémoire et de *use after free* (du mieux que vous le pouvez).

Exercice 3. *Liste doublement chaînée en OCaml*

On souhaite maintenant écrire une implémentation utilisant une liste *doublement chaînée* : chaque élément (maillon) de la liste maintient une référence à la fois sur son (éventuel) successeur (comme dans le cas d'une liste simplement chaînée) **et** sur son (éventuel) *prédécesseur*.

On définit pour cela les deux types et l'exception suivante :

```
type 'a dlist_e = { mutable dat : 'a;
                    mutable prv : 'a dlist_e option;
                    mutable nxt : 'a dlist_e option }
type 'a  dlist = 'a dlist_e option ref
exception Empty
```

1. Écrivez des fonctions OCaml :
   - `make_empty : unit -> 'a dlist`
   - `is_empty : 'a dlist -> bool`
   - `insert_head : 'a -> 'a dlist -> unit`
   - `insert_after : 'a -> 'a dlist_e -> unit`
   - `remove_head : 'a dlist -> unit`
   - `remove_head_s : 'a dlist -> unit`
   - `get_head : 'a dlist -> 'a dlist_e`
   - `get_head_dat : 'a dlist -> 'a`
   - `remove_get_head : 'a dlist -> 'a dlist_e`
   - `remove_get_head_dat : 'a dlist -> 'a`
   - `get_tail : 'a dlist -> 'a dlist`
   - `get_tail_s : 'a dlist -> 'a dlist`
   - `iter : ('a -> 'a) -> 'a dlist -> unit`

   qui adaptent celles écrites pour les `'a slist` au type `'a dlist`.
2. Testez avec précaution.
3. Écrivez une fonction OCaml :

   `insert_before : 'a -> 'a dlist_e -> 'a dlist -> unit`

   qui intercale (un élément de valeur `dat` égale à) son premier argument entre l'éventuel prédecesseur de son second argument (élément de la liste donnée par son troisième argument) et ce dernier.
4. Testez.
5. Écrivez une fonction OCaml :

   `remove : 'a dlist_e -> 'a dlist -> unit`

qui modifie son second argument pour en supprimer un élément égal à son premier argument si un tel élément y apparaît, et lève une exception `Not_found` sinon.
Cette fonction devrait être significativement plus efficace que celle opérant sur des `'a slist`.

6. Testez.
7. Écrivez une variante silencieuse :
   `remove_s : 'a dlist_e -> 'a dlist -> unit`
8. Testez.