Liste doublement chaînée - merge-sort.

Algorithmique 3 - Travaux pratiques

Les travaux pratiques seront réalisés à partir de l'archive `base_code_lab3.tar.gz` fournie sur moodle et contenant la hiérarchie suivante :

`base_code_lab3`
–> `Code` : répertoire contenant le code source fourni devant être complété.

Il est demandé à l'étudiant de rédiger, dans un fichier au format **pdf**, un compte rendu de travaux pratiques contenant, pour chaque exercice :

- les réponses aux éventuelles questions posées pour l'exercice dans le sujet,
- une description synthétique des algorithmes mis en œuvre et une analyse de la complexité en temps et en espace de ces algorithmes,
- une synthèse de la démarche de programmation mise en œuvre, en précisant :
  - les références concernant les documentations et aides utilisées (livres, camarades, chat-GPT, site web ...),
  - les difficultés rencontrées et comment elles ont été résolues.

À l'issue de la séquence, l'étudiant devra déposer sur moodle une archive selon les spécifications suivantes :

- Format de l'archive identique au format initial (.tar.gz),
- Contenu de l'archive :
  - le compte rendu de TP au format pdf,
  - un dossier `Code` contenant uniquement le code source et le Makefile, à l'exclusion de tout autre fichier.

Le respect strict des consignes ci-dessus est un élément de l'évaluation des travaux déposés.

L'objectif de ce TP est d'écrire un module de gestion de collection selon la spécification étudiée en cours/TD `List`. Nous souhaitons mettre en œuvre une implantation de cette spécification en utilisant une représentation interne de liste doublement chaînée **avec sentinelle**. Nous souhaitons aussi disposer dans cette implantation des opérateurs `map` et `reduce` permettant d'appliquer des opérateurs de transformation de la liste elon le fonctionnement vu en cours-TD.
Enfin, nous souhaitons pouvoir trier, en utilisant l'algorithme de tri fusion, une liste selon une relation d'ordre fournie par l'utilisateur du module.

Le fichier source `main.c` fourni dans l'archive propose de tester l'implantation de ce module et des différents opérateurs ajoutés. **Ce fichier ne devra pas être modifié lors de ce TP**. Le programme exécutable prend un paramètre correspondant au numéro de l'exercice à tester.

Le fichier source `list.c` devra être complété lors de ce TP. Le code source initial des fonctions proposées dans le fichier list.c permet de compiler et d'exécuter le programme principal mais n'implantent pas les fonctionalités voulues. Ce code source devra être **remplacé** par le code de l'étudiant au fur et à mesure de l'avancement du TP.

La documentation, à générer par la commande `make doc`, fournit non seulement la documentation des opérateurs à programmer mais aussi les résultats espérés pour chaque exercice.

1 Définition du TAD `List`

Comme cela a été étudié en cours, le type abstrait de données `List` permet de gérer une collection linéaire d'informations et fournit des opérateurs d'insertion, de suppression et d'accès à n'importe quel élément de la collection.
En faisant l'hypothèse non restrictive que nous définissons une collection d'entiers, ce type abstrait peut être spécifié comme suit.

Signature

Sorte : List
Utilise : int, bool, env
ListFunctor : int × env → int
Opérateurs_Constructeurs :
list : → List
push_back : List × int → List
Opérateurs :
push_front : List × int → List
pop_back : List → List
pop_front : List → List
back : List → int
front : List → int
empty : List → bool
size : List → int
at : List × int → int
insert_at : List × int × int → List
remove_at : List × int → List
map : List × ListFunctor × env → List

Axiomatique

Préconditions :
pop_back(l), pop_front(l) ***défini ssi*** $\neg$*empty(l)*
back(l), front(l) ***défini ssi*** $\neg$*empty(l)*
Axiomes :
push_front(list, x) = push_back(list,x)
push_front(push_back(l,x), y) = push_back(push_front(l,y), x)
pop_back(push_back(l,x)) = l
pop_front(push_back(list,x)) = list
$l \neq list \rightarrow$ *pop_front(push_back(l,x)) = push_back(pop_front(l), x)*
back(push_back(l,x)) = x
front(push_back(list,x)) = x
$l \neq list \rightarrow$ *(front(push_back(l,x)) = front(l))*
empty(list) = true
empty(push_back(l,x)) = false
size(list) = 0 ;
size(push_back(l,x)) = 1+ size(l)
$i = size(l) \rightarrow$ *at(push_back(l,x), i) = x*
$i < size(l) \rightarrow$ *at(push_back(l,x), i) = at(l, i)*
insert_at(list, 0, x) = push_back(list,x)
$i = size(push\_back(l,x)) \rightarrow$ *insert_at(push_back(l,x),i,y) = push_back(push_back(l,x),y)*
$0 \leq i < size(push\_back(l,x)) \rightarrow$ *insert_at(push_back(l,x),i,y) = push_back(insert_at(l, i,y), x)*
$i = size(l) \rightarrow$ *remove_at(push_back(l,x),i) = l*
$0 \leq i < size(l) \rightarrow$ *remove_at(push_back(l,x),i) = push_back(remove_at(l, i),x)*
map(list, f, e) = list
map(push_back(l, x), f, e) = push_back(map(l, f, e), f(x, e))

Le fichier `list.h` fourni propose une interface pour l'implantation de ce TAD. **Ce fichier ne devra pas être modifié lors de ce TP**.

L'implantation de la liste à réaliser repose sur une représentation avec utilisation d'une sentinelle. Sur une liste doublement chaînée, la sentinelle est un élément de chaînage vérifiant l'invariant de structure suivant :

— sa durée de vie est la durée de vie de la liste,
— l'élément suivant la sentinelle est l'élément en tête de liste,
— l'élément précédent la sentinelle est l'élément en fin de liste,
— l'élément précédent la tête de liste est la sentinelle,
— l'élément suivant la fin de liste est la sentinelle,
— pour une liste vide, l'élément de tête et l'élément de fin correspondent la sentinelle.

2 Implantation des constructeurs et de l'opérateur `map`.

À partir de la représentation interne de la liste proposée dans le fichier list.c, programmer les opérateurs suivants :

1. `List* list_create(void)`. Remplacer le code par défaut de cet opérateur par le code d'allocation de la structure représentant la liste et l'initialisation correcte de cette structure.
   En prenant exemple sur l'implantation fournie à la séquence 2 du module `staticstack`, et en particulier du mécanisme d'allocation mémoire mis en œuvre dans la fonction `stack_create()`, vous vous assurerez de la compacité de la représentation mémoire de la liste et prendrez soin d'assurer, par construction, que la sentinelle à une durée de vie égale à celle de la liste.
2. `void list_delete(ptrList* l)`. Remplacer le code par défaut de cet opérateur par le code permettant de libérer l'ensemble des ressources mémoires allouées pour le stockage de la liste `l`. Après exécution, la liste, correspondant au pointeur `l`, doit être invalidée en mettant la constante `NULL` dans `l`.
3. `List* list_push_back(List* l, int v)`. Remplacer le code par défaut de cet opérateur pour ajouter l'élément `v` en fin de liste.
4. `List* list_map(List* l, Functor f, void* environment)`. Comme vu en cours-TD, cet opérateur permet d'exécuter la fonction `f` sur chaque élément de la liste en fournissant l'environnement utilisateur `environment` à cette fonction et de remplacer la valeur de l'élément par la valeur de retour de `f`.
5. `bool list_is_empty(const List* l)` Remplacer le code par défaut de cet opérateur par le code respectant la spécification.
6. `int list_size(const List* l)` Remplacer le code par défaut de cet opérateur par le code respectant la spécification. Attention, cet opérateur doit fournir le résultat en temps constant (complexité en O(1)).

Lorsque ces six opérateurs auront été programmés, vous devrez obtenir les résultats suivants lors de l'exécution du programme :

```
$ ./list_test 1
-------- TEST PUSH_BACK --------
List (10) : 0 1 2 3 4 5 6 7 8 9
$
```

3 Implantation de l'opérateurs `push_front`.

Implanter l'opérateur suivant :

1. `List* list_push_front(List* l, int v)`. Remplacer le code par défaut de cet opérateur pour ajouter l'élément `v` en tête de liste.

Lorsque ces deux opérateurs auront été programmés, vous devrez obtenir les résultats suivants lors de l'exécution du programme :

```
$ ./list_test 2
 -------- TEST PUSH_BACK --------
 List (10) : 0 1 2 3 4 5 6 7 8 9
 -------- TEST PUSH_FRONT --------
 List (10) : 9 8 7 6 5 4 3 2 1 0
 Sum is 45
$
```

4 implantation des opérateurs d'accès et de suppression en tête et en fin de liste.

Implanter les opérateurs suivants :

1. `int list_front(const List* l)`. Cet opérateur renvoie la valeur de l'élément se trouvant en tête de liste.
2. `int list_back(const List* l)`. Cet opérateur renvoie la valeur de l'élément se trouvant en fin de liste.
3. `List* list_pop_front(List* l)`. Cet opérateur supprime l'élément se trouvant en tête de liste.
4. `List* list_pop_back(List* l)`. Cet opérateur supprime l'élément se trouvant en fin de liste.

Lorsque ces quatre opérateurs auront été programmés et que le programme aura été compilé, vous devrez obtenir les résultats suivants :

```
$ ./list_test 3
-------- TEST PUSH_BACK --------
 List (10) : 0 1 2 3 4 5 6 7 8 9
 -------- TEST PUSH_FRONT --------
 List (10) : 9 8 7 6 5 4 3 2 1 0
 Sum is 45
 -------- TEST POP_FRONT --------
 Pop front : 9
 List (9) : 8 7 6 5 4 3 2 1 0
 -------- TEST POP_BACK --------
 Pop back : 0
 List (8) : 8 7 6 5 4 3 2 1
$
```

5 implantation des opérateurs d'accès, d'insertion et de suppression à une position donnée dans la liste.

Implanter les opérateurs suivants :

1. `List* list_insert_at(List* l, int p, int v)`. Cet opérateur insère l'élément `v` à la position `p` dans la liste `l`.

2. `List* list_remove_at(List* l, int p)`. Cet opérateur supprime l'élément à la position `p` dans la liste `l`.
3. `int list_at(const List* l, int p)`. Cet opérateur renvoie la valeur de l'élément se trouvant à la position `p` dans la liste `l`.

Lorsque ces quatre opérateurs auront été programmés, vous devrez obtenir les résultats suivants lors de l'exécution du programme :

```
$./list_test 4
-------- TEST PUSH_BACK --------
List (10) : 0 1 2 3 4 5 6 7 8 9
-------- TEST PUSH_FRONT --------
List (10) : 9 8 7 6 5 4 3 2 1 0
Sum is 45
-------- TEST POP_FRONT --------
Pop front : 9
List (9) : 8 7 6 5 4 3 2 1 0
-------- TEST POP_BACK --------
Pop back : 0
List (8) : 8 7 6 5 4 3 2 1
-------- TEST INSERT_AT        --------
List (10) : 0 2 4 6 8 9 7 5 3 1
-------- TEST REMOVE_AT        --------
List (4) : 2 6 9 5
List cleared (0)
-------- TEST AT       --------
List (10) : 0 1 2 3 4 5 6 7 8 9
$
```

6 Algorithme de tri fusion sur liste doublement chaînée.

Afin de pouvoir trier une liste d'entier, nous souhaitons étendre le module de gestion de liste implanté dans les exercices précédent en ajoutant l'opérateur `List* list_sort(List* l, OrderFunctor f)`. Cette fonction à pour objectif de trier la liste `l` selon la relation d'ordre définie par la fonction `f` passée en paramètre et dont la spécification est donné dans le fichier `list.h` et dans la documentation générée par `make doc`.

Afin de proposer un algorithme efficace pour le tri de la liste, cette fonction utilisera l'algorithme du tri fusion (cf https://fr.wikipedia.org/wiki/Tri_fusion).
Bien que l'on puisse appliquer l'algorithme de tri fusion sur les listes doublement chaînées circulaires, ce qui est le cas de notre liste, la sentinelle jouant le rôle de maillon de rebouclage, le fait que la sentinelle ne définisse pas un élément de la liste peut être gênant pour l'implantation directe de l'algorithme tel que décrit sur la page Wikipédia ci-dessus.
Pour implanter cet algorithme, nous allons donc suivre la démarche suivante :

1. Définir une structure de données `SubList` permettant de représenter de façon minimale (pointeur de tête et pointeur de queue) une liste doublement chaînée sans sentinelle. Cette structure de donnée doit-elle être déclarée dans le fichier `list.h` ou dans le fichier `list.c` ? Justifier votre choix.
2. Définir une fonction `SubList list_split(SubList l)` qui découpe une liste l en deux sous listes de tailles égales à 1 élément près. À partir de la liste doublement chaînée `l` cette fonction renvoie une structure `SubList` dont le pointeur de tête désigne le dernier élément

de la sous-liste gauche et le pointeur de queue le premier élément de la sous liste droite. Cette fonction doit-elle être déclarée dans le fichier `list.h` ou dans le fichier `list.c` ? Justifier votre choix.

3. Définir une fonction `SubList list_merge(SubList leftlist, SubList rightlist, OrderFunctor f)` permettant de fusionner les deux listes triées `leftlist` et `rightlist` en respectant l'ordre défini par la fonction passée en paramètre `f`. Cette fonction doit-elle être déclarée dans le fichier `list.h` ou dans le fichier `list.c` ? Justifier votre choix.
4. Définir une fonction récursive `SubList list_mergesort(SubList l, OrderFunctor f)` permettant de trier la liste `l` selon la relation d'ordre `f`. Cette fonction devra ,dans un premier temps, découper la liste l en deux sous liste (gauche et droite), puis effectuer deux appels récursif pour trier ces deux sous-listes et retourner le résultat de la fusion de ces deux sous listes triées. Cette fonction doit-elle être déclarée dans le fichier `list.h` ou dans le fichier `list.c` ? Justifier votre choix.
5. Enfin, dans la fonction `List* list_sort(List* l, OrderFunctor f)`, transformer la liste avec sentinelle en une liste minimale correspondant à votre structure `SubList`, trier cette liste et transformer le résultat pour remettre en place la sentinelle.

Lors de l'exécution de votre programme pour valider cet exercice, vous devrez obtenir les résultats suivants :

```
 $./list_test 5
 -------- TEST SORT      --------
Unsorted list    : List (8) : 5 3 4 1 6 2 3 7
Decreasing order : List (8) : 7 6 5 4 3 3 2 1
Increasing order : List (8) : 1 2 3 3 4 5 6 7
$
```