Liste doublement chaînée - merge-sort.

Structures de données - Travaux dirigés sur machines

Séance 3

Il est demandé aux étudiants de réaliser chaque exercice dans un répertoire séparé.
Les travaux seront réalisés à partir de l'archive `SD-TP3.tgz` fournie sur moodle et contenant la hiérarchie suivante :

`SD-TP3`
–> `Test` : répertoire contenant les fichiers de test.
–> `Code` : répertoire contenant le code source fourni devant être complété.

Il est demandé aux étudiants de rédiger, dans un fichier texte contenu dans le dossier correspondant, les réponses aux questions posées pour chaque exercice.
À la demande de l'enseignant, l'étudiant devra pouvoir fournir une archive similaire à l'archive de départ contenant le résultat de son travail.

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**.
De plus, nous souhaitons étendre les fonctionnalités offertes par l'implantation de ce TAD en proposant une fonction permettant d'appliquer un opérateur quelconque sur un élément de la liste. Un tel opérateur, par exemple, peut servir pour afficher le contenu de la liste de façon efficace.

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 des fonctions proposées dans le fichier list.c permet de compiler et d'exécuter le programme. 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`, permettant de gérer une collection linéaire d'informations et fournissant des opérateurs d'insertion, de suppression et d'accès à n'importe quel élément de la collection peut être spécifié comme rappelé ci-dessous (en faisant l'hypothèse non restrictive que nous définissons une collection d'entiers).

Sorte : LIST
Utilise : INT, BOOL

Opérateurs_Constructeurs :
list : $\rightarrow$ LIST
push_back : LIST $\times$ INT $\rightarrow$ LIST
Opérateurs :
push_front : LIST $\times$ INT $\rightarrow$ LIST
pop_back : LIST $\rightarrow$ LIST
pop_front : LIST $\rightarrow$ LIST
back : LIST $\rightarrow$ INT
front : LIST $\rightarrow$ INT
empty : LIST $\rightarrow$ BOOL
size : LIST $\rightarrow$ INT
ith : LIST $\times$ INT $\rightarrow$ INT
insert_at : LIST $\times$ INT $\times$ INT $\rightarrow$ LIST
remove_at : LIST $\times$ INT $\rightarrow$ LIST

Préconditions :
pop_back(q), pop_front(q), back(q), front(q) ***défini ssi*** $\neg$*empty(q)*

Axiomes :
push_front(list, x) = push_back(list,x)
push_front(push_back(q,x), y) = push_back(push_front(q,y), x)
pop_back(push_back(q,x)) = q
pop_front(push_back(list,x)) = list
$q \neq list \rightarrow$ *(pop_front(push_back(q,x)) = push_back(pop_front(q), x))*
back(push_back(q,x)) = x
front(push_back(list,x)) = x
$q \neq list \rightarrow$ *(front(push_back(q,x)) = front(q))*
empty(list) = true
empty(push_back(q,x)) = false
size(list) = 0 ;
size(push_back(q,x)) = 1+ size(q)
$i = size(q) \rightarrow$ *ith(push_back(q,x), i) = x*
$i < size(q) \rightarrow$ *ith(push_back(q,x), i) = ith(q, i)*
insert_at(list, 0, x) = push_back(list,x)
$i = size(push\_back(q,x)) \rightarrow$ *insert_at(push_back(q,x),i,y) = push_back(push_back(q,x),y)*
$0 \leq i < size(push\_back(q,x)) \rightarrow$ *insert_at(push_back(q,x),i,y) = push_back(insert_at(q, i,y), x)*
$i = size(q) \rightarrow$ *remove_at(push_back(q,x),i) = q*
$0 \leq i < size(q) \rightarrow$ *remove_at(push_back(q,x),i) = push_back(remove_at(q, i),x)*

Le fichier `list.h` fourni propose une interface à 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 :

— 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()`. 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.
2. `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.
3. `List * list_map(List *l, SimpleFunctor f)`. Comme vu en cours, cet opérateur permet d'exécuter la fonction `f` sur chaque élément de la liste et de remplacer la valeur de l'élément par la valeur de retour de `f`.
4. `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`.
5. `bool list_is_empty(List *l)` Remplacer le code par défaut de cet opérateur par le code respectant la spécification.
6. `int list_size(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 des opérateurs `push_front` et `reduce`.

Implanter les opérateurs suivants :

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.
2. `List *list_reduce(List *l, ReduceFunctor f, void *userData)`. Comme vu en cours, cet opérateur permet d'exécuter la fonction `f` sur chaque élément de la liste en fournissant l'environnement utilisateur `userData` à cette fonction et de remplacer la valeur de l'élément par la valeur de retour de `f`.

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(List *l)`. Cet opérateur renvoie la valeur de l'élément se trouvant en tête de liste.
2. `int list_back(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(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 exercice 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. A 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
$
```