

Nom: MOSAAD

Prénom: Chehab

Numéro étudiant : 22106126

TP3 : Liste doublement chaînée - merge-sort

1) Définition du TAD List :

2) Implantation des constructeurs et de l'opérateur map :

- 1) Dans la fonction `List *list_create()`, je commence par allouer de la mémoire pour un pointeur de type `List`. Ensuite, un élément sentinelle est créé en allouant de la mémoire pour un pointeur de type `LinkedElement`. La sentinelle est initialisée de telle sorte que son pointeur `next` et son pointeur `previous` pointent tous deux vers elle-même ; car la liste est vide. Enfin, j'initialise la variable `size` à 0 et je retourne un pointeur vers la nouvelle liste.
- 2) Dans la fonction `List *list_push_back(List *l, int v)`, je commence par allouer de la mémoire pour un nouvel élément de la liste. L'entier `v` est ensuite assignée à l'attribut `value` du nouvel élément. Puis j'insère le nouvel élément à la fin de la liste. Ensuite, je modifie les pointeurs `next` et `previous` de l'élément sentinelle et de l'élément précédent. La variable `size` de la liste est également mise à jour. Enfin, je retourne un pointeur vers la liste mise à jour.
- 3) Dans la fonction `List *list_map(List *l, SimpleFunctor f)`, je commence par itérer à travers la liste à l'aide d'une boucle `for`. La boucle commence à l'élément suivant de l'élément sentinelle et s'arrête lorsque l'élément courant est l'élément sentinelle. À chaque itération de la boucle, la fonction `f` est appliquée à la valeur de l'élément courant. Puis je remplace la valeur d'origine de l'élément par la valeur retournée par `f`. Enfin, je retourne un pointeur vers la liste modifiée.
- 4) Dans la fonction `void list_delete(ptrList *l)`, je commence par libérer la mémoire allouée pour la liste. Cela libère toute la mémoire allouée pour la structure de la liste, y compris l'élément sentinelle et tous les éléments de la liste. Ensuite, je mets à jour le pointeur de pointeur de liste en le mettant à `NULL`. Cela permet de s'assurer que le pointeur pointe maintenant vers une adresse nulle, empêchant ainsi toute tentative d'accès à la mémoire qui a été libérée.
- 5) Dans la fonction `bool list_is_empty(List *l)`, je compare simplement la taille de la liste, `size`, à zéro et retourne `"true"` si elle est égale à zéro, sinon retourne `"false"`.
- 6) Dans la fonction `int list_size(List *l)`, je retourne simplement la taille de la liste.

Les résultats obtenus lors de l'exécution du programme :

```
PS C:\Users\future> cd 'D:\INFO PAUL SABATIER\12\s4\algo3\tp3\Code\'
PS D:\INFO PAUL SABATIER\12\s4\algo3\tp3\Code> make
"Generating in release mode"
make[1]: Entering directory 'D:/INFO PAUL SABATIER/12/s4/algo3/tp3/Code'
make[1]: Leaving directory 'D:/INFO PAUL SABATIER/12/s4/algo3/tp3/Code'
PS D:\INFO PAUL SABATIER\12\s4\algo3\tp3\Code> .\list_test.exe 1
----- TEST PUSH_BACK -----
List (10) : 0 1 2 3 4 5 6 7 8 9
```

3) Implantation des opérateurs push front et reduce :

- 1) Dans la fonction `List *list_push_front(List *l, int v)`, je commence par allouer de la mémoire pour le nouvel élément à ajouter à la liste. Ensuite, j'initialise la valeur de l'élément avec la valeur donnée `v`.

Puis, je mets à jour les pointeurs de l'élément nouvellement créé de telle sorte qu'il pointe vers le premier élément actuel de la liste en tant que suivant et vers la sentinelle de la liste en tant que précédent. Ainsi, le premier élément de la liste, qui est le suivant de la sentinelle, doit avoir comme précédent la sentinelle. Ensuite, je mets à jour les pointeurs de la sentinelle de la liste pour que le nouvel élément soit le premier élément de la liste et j'incrmente la taille de la liste. Enfin, je retourne un pointeur vers la liste mise à jour.

- 2) Dans la fonction `List *list_reduce(List *l, ReduceFunctor f, void *userData)`, je commence par parcourir tous les éléments de la liste en partant du premier élément, en utilisant une boucle `for`. La boucle est initialisée avec un pointeur vers le premier élément de la liste et continue jusqu'à ce que l'élément atteint soit égal à la sentinelle, qui est le dernier élément de la liste. Ainsi, la boucle itère sur tous les éléments de la liste, sauf la sentinelle elle-même. Pour chaque élément de la liste, je remplace sa valeur par le résultat de la fonction `f`. La fonction de réduction est appelée avec deux arguments : la valeur de l'élément courant de la liste et un pointeur générique `userData`. Enfin, après avoir appliqué la fonction de réduction à tous les éléments de la liste, je retourne un pointeur vers la liste mise à jour.

Les résultats obtenus lors de l'exécution du programme :

```
PS D:\INFO PAUL SABATIER\12\s4\algo3\tp3\Code> .\list_test.exe 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 :

- 1) Dans la fonction `int list_front(List *l)`, j'accède à la sentinelle de la liste et elle accède au premier élément de la liste en utilisant le pointeur `next` de la sentinelle, qui pointe vers le premier élément de la liste. Enfin, je retourne la valeur de cet élément.
- 2) Dans la fonction `int list_back(List *l)`, j'accède à la sentinelle de la liste et elle accède à l'avant-dernier élément de la liste en utilisant le pointeur `previous` de la sentinelle, qui pointe vers l'avant-dernier élément de la liste. Enfin, elle retourne la valeur de cet élément.
- 3) Dans la fonction `List *list_pop_front(List *l)`, je commence par accéder à la sentinelle de la liste puis j'accède au deuxième élément de la liste en utilisant le pointeur `next` de la sentinelle, qui pointe vers le premier élément de la liste. Ensuite je mets à jour le champ `next` de la sentinelle pour pointer vers le deuxième élément de la liste, en ignorant ainsi le premier élément. Puis le champ `previous` du nouvellement choisi premier élément pour qu'il pointe vers la sentinelle, et donc le premier élément de la liste est supprimé de la liste. Également, je mets à jour la taille de la liste pour refléter la suppression de l'élément. Enfin, je retourne un pointeur vers la liste mise à jour.
- 4) Dans la fonction `List *list_pop_back(List *l)`, je commence par accéder à la sentinelle de la liste puis j'accède à l'avant-dernier élément de la liste en utilisant le pointeur `previous` de la sentinelle, qui pointe vers l'avant-dernier élément de la liste. Ensuite je mets à jour le champ `previous` de la sentinelle pour pointer vers l'avant-dernier élément de la liste, en ignorant ainsi le dernier élément. Puis le champ `next` du nouvellement choisi avant-dernier élément pour qu'il pointe vers la sentinelle, et donc le dernier élément de la liste est supprimé de la liste. Également, la taille de la liste est mise à jour pour refléter la suppression de l'élément. Enfin, je retourne un pointeur vers la liste mise à jour.

Les résultats obtenus lors de l'exécution du programme :

```
PS D:\INFO PAUL SABATIER\12\s4\algo3\tp3\Code> .\list_test.exe 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 :

- 1) Dans la fonction `List *list_insert_at(List *l, int p, int v)`, je commence par accéder à la sentinelle de la liste. Ensuite, je boucle sur la liste pour atteindre l'élément précédant la position `p`, en mettant à jour le pointeur `insert_at` à chaque itération. Une fois que j'atteins l'élément précédant la position `p`, je crée un nouvel élément à insérer dans la liste en allouant de la mémoire pour un nouvel élément et en initialisant ses champs `value`, `next`, et `previous` avec les valeurs fournies. Ensuite, je mets à jour les pointeurs `next` et `previous` des éléments environnants pour inclure le nouvel élément. Également, la taille de la liste est mise à jour pour refléter l'ajout de l'élément. Enfin, je retourne un pointeur vers la liste mise à jour.
- 2) Dans la fonction `List *list_remove_at(List *l, int p)`, je commence par accéder au premier élément de la liste (c'est-à-dire celui après la sentinelle). Ensuite, je boucle sur la liste pour atteindre l'élément à la position `p`, en mettant à jour le pointeur `remove_at` à chaque itération. Une fois que j'atteins l'élément à la position `p`, je mets à jour les pointeurs `next` et `previous` des éléments environnants pour retirer l'élément de la liste. Également, la taille de la liste est mise à jour pour refléter la suppression de l'élément. Enfin, je retourne un pointeur vers la liste mise à jour.
- 3) Dans la fonction `int list_at(List *l, int p)`, je commence par accéder au premier élément de la liste (c'est-à-dire celui après la sentinelle). Ensuite, je boucle sur la liste pour atteindre l'élément à la position `p`, en mettant à jour le pointeur `value_at` à chaque itération. Une fois que j'atteins l'élément à la position `p`, je renvoie sa valeur.

Les résultats obtenus lors de l'exécution du programme :

```
PS D:\INFO PAUL SABATIER\12\s4\algo3\tp3\Code> .\list_test.exe 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 :

- 1) J'ai constitué la structure de données SubList de trois membres ; 1) Le head qui est un pointeur vers le premier élément de la sous-liste. 2) Le tail qui est un pointeur vers le dernier élément de la sous-liste. 3) La size qui est la taille de la sous-liste.

Le code d'une nouvelle structure de données comme SubList doit être écrit dans le fichier d'en-tête list.h afin que les autres fichiers source qui incluent l'en-tête peuvent utiliser le type de données SubList. Parce que si je définisse la structure SubList dans list.c au lieu de list.h, les autres fichiers source qui incluent list.h ne pourront pas utiliser la structure SubList.

- 2) Dans la fonction SubList list_split(SubList l), je commence par créer une nouvelle SubList appelée split_Sublist. Je détermine ensuite la position de l'élément milieu de la liste d'entrée en divisant la taille de la liste par deux. Cela me permettra de déterminer où couper la liste en deux. Ensuite, j'itère sur les éléments de la liste d'entrée en utilisant un pointeur SubList_at pour suivre l'emplacement de l'élément actuel. L'itération s'arrête après avoir atteint l'élément du milieu de la liste. À ce stade, la sous-liste de la deuxième moitié commence juste après l'élément du milieu, donc le premier élément de split_Sublist est défini sur cet élément et le dernier élément de split_Sublist est défini sur le prochain élément de la liste d'entrée, car cela sera la nouvelle fin de la deuxième sous-liste. Finalement, je retourne un pointeur vers la deuxième sous-liste nouvellement créée.

La fonction list_split doit être écrite dans le fichier list.c parce qu'elle opère sur la structure SubList et implique la manipulation de la structure de données de la liste chaînée. Par conséquent, il appartient au fichier list.c, où j'ai déjà implémenté les fonctions qui manipulent la sous-liste, telles que SubList_create et SubList_delete. Également, je dois ajouter un prototype de fonction pour list_split dans le fichier list.h afin que d'autres parties du programme puissent l'utiliser.

- 3) Dans la fonction SubList list_merge(SubList leftlist, SubList rightlist, OrderFunctor f), je commence par créer une nouvelle sous-liste chaînée appelée "mergedList" qui sera utilisée pour stocker la liste fusionnée. Ensuite, j'initialise deux pointeurs "right_current" et "left_current" pour pointer vers le premier élément de chaque sous-liste chaînée. J'utilise ensuite une boucle "while" pour parcourir les deux sous-listes chaînées simultanément, en comparant les éléments actuels de chaque liste en utilisant la fonction de comparaison f. Si cette fonction de comparaison retourne vrai pour un élément de la sous-liste gauche, alors cet élément est ajouté à la fin de la sous-liste fusionnée. Si la fonction de comparaison retourne faux, alors l'élément de la sous-liste droite est ajouté à la fin de la sous-liste fusionnée. Une fois que la boucle "while" a terminé de parcourir les deux sous-listes chaînées, je vérifie s'il reste des éléments dans l'une des sous-listes qui n'ont pas été ajoutés à la sous-liste fusionnée. Si la sous-liste droite a des éléments restants, je parcours ces éléments et les ajoute à la fin de la sous-liste fusionnée. Si la sous-liste gauche a des éléments restants, je parcours ces éléments et les ajoute également à la fin de la sous-liste fusionnée. Enfin, je renvoie un pointeur vers la sous-liste fusionnée "mergedList".

La fonction list_merge doit être écrite dans le fichier list.c. car elle implémente le comportement de la structure de données SubList et qu'elle n'est pas destinée à être exposée à du code externe. En l'ajoutant à list.c, je garde les détails d'implémentation cachés et maintenez l'encapsulation. Je dois déclarer le prototype de fonction dans list.h pour l'utiliser dans d'autres fonctions définies dans le même fichier d'en-tête.

- 4) Dans la fonction SubList list_mergesort(SubList l, OrderFunctor f), je vérifie tout d'abord si la sous-liste contient plus d'un élément. Si c'est le cas, je divise la sous-liste en deux parties égales en appelant la fonction list_split et crée deux nouvelles sous-listes pour stocker ces parties. Ensuite, j'appelle récursivement la fonction list_mergesort pour trier chacune des deux sous-listes créées précédemment. Une fois que les deux sous-listes ont été triées, j'appelle la fonction list_merge pour fusionner les deux sous-listes triées dans une seule sous-liste triée. Finalement, je renvoie la sous-liste triée résultante. Si la

sous-liste initiale ne contient qu'un seul élément ou moins, je la renvoie directement sans effectuer de tri.

Cette fonction doit être implémentée dans le fichier list.c avec les autres fonctions liées à la structure SubList.

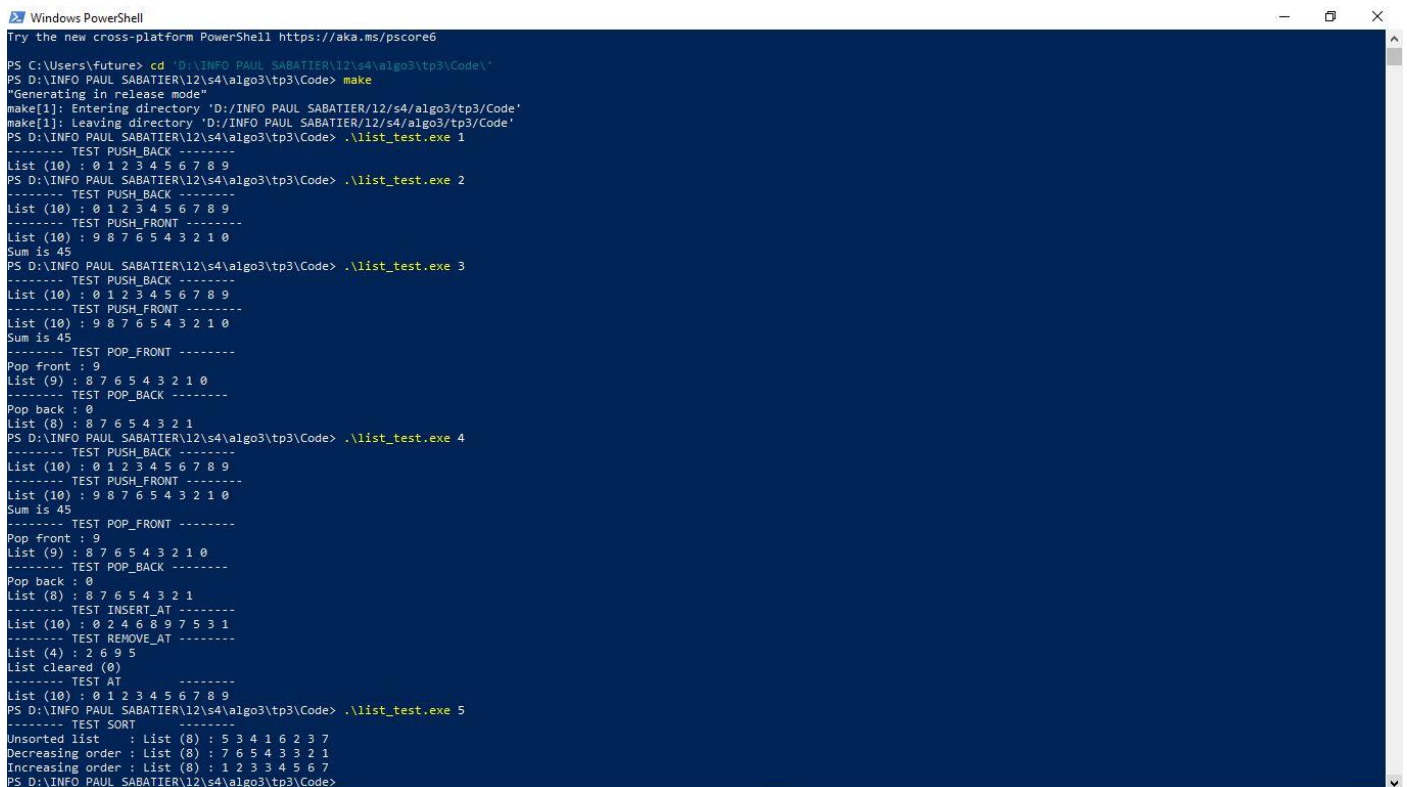
- 5) Dans la fonction `List *list_sort(List *l, OrderFunctor f)`, je commence par créer une nouvelle sous-liste à partir des éléments de la liste initiale l. Je parcours la liste initiale en ajoutant chaque élément à la fin de la nouvelle sous-liste. Ensuite, j'appelle la fonction `list_mergesort` pour trier les éléments de la sous-liste en utilisant la fonction de comparaison f. Une fois que la sous-liste triée est renvoyée, je parcours à nouveau la liste initiale en remplaçant chaque élément par son équivalent trié. Cela est fait en parcourant simultanément la sous-liste triée et la liste initiale et en écrasant la valeur de chaque élément dans la liste initiale avec la valeur correspondante de la sous-liste triée. Enfin, je renvoie la liste initiale triée.

Cette fonction doit être implémentée dans le fichier list.c car elle opère sur la structure de données List définie dans ce fichier.

Les résultats obtenus lors de l'exécution du programme :

```
PS D:\INFO PAUL SABATIER\12\s4\algo3\tp3\Code> .\list_test.exe 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
```

Tous les tests ensembles :



```
Windows PowerShell
Try the new cross-platform PowerShell https://aka.ms/powershell

PS C:\Users\Futurs> cd 'D:\INFO PAUL SABATIER\12\s4\algo3\tp3\Code\'
PS D:\INFO PAUL SABATIER\12\s4\algo3\tp3\Code> make
"Generating in release mode"
make[1]: Entering directory 'D:\INFO PAUL SABATIER\12\s4\algo3\tp3\Code\'
make[1]: Leaving directory 'D:\INFO PAUL SABATIER\12\s4\algo3\tp3\Code\'
PS D:\INFO PAUL SABATIER\12\s4\algo3\tp3\Code> .\list_test.exe 1
----- TEST PUSH_BACK -----
List (10) : 0 1 2 3 4 5 6 7 8 9
PS D:\INFO PAUL SABATIER\12\s4\algo3\tp3\Code> .\list_test.exe 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
PS D:\INFO PAUL SABATIER\12\s4\algo3\tp3\Code> .\list_test.exe 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
PS D:\INFO PAUL SABATIER\12\s4\algo3\tp3\Code> .\list_test.exe 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
PS D:\INFO PAUL SABATIER\12\s4\algo3\tp3\Code> .\list_test.exe 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
PS D:\INFO PAUL SABATIER\12\s4\algo3\tp3\Code>
```