BABU Bastien
L2 MIDL
22302768

UNIVERSITÉ TOULOUSE III PAUL SABATIER

Compte Rendu TP Algo 3
Séquence 3 : Liste doublement chaînée – merge sort.

1) Description des choix d'implémentation

- **Implémentation des constructeurs et des opérateurs du TAD *List* :**

Les fonctions qui implémentent les constructeurs et les opérateurs du TAD *List* sont performante et robuste notamment de par l'utilisation d'une liste doublement chaînée circulaire avec sentinelle. En effet, l'utilisation d'une sentinelle simplifie le traitement des bornes, puisque la tête et la queue sont toujours reliées facilitant ainsi les insertions et suppressions d'éléments dans la liste.

La performance est assurée par une complexité en temps constant **O(1)**, en ce qui concerne les fonctions d'ajout et de suppression aux bornes (tête et queue de liste) : *list_push_front*, *list_push_back*, *list_pop_front*, *list_pop_back*. De plus, les fonction vérifiant l'état de la liste comme la taille et si elle est vide comme *list_is_empty* et *list_size* ont aussi une complexité en temps constant **O(1)** ce qui garantit à nouveau des performances remarquables. En ce qui concerne les fonctions d'ajout, suppression et d'accès à un élément précis (hors bornes) : *list_insert_at*, *list_remove_at*, *list_at,* malgré une baisse des performances dû à une complexité en un temps linéaire **O(n)**, ces algorithmes restent quand même le plus optimisé grâce à la structure chainée qui permet un parcours sans décalage d'éléments.

La robustesse est aussi assurée de par la vérification des pré-conditions pour les fonctions : *list_pop_front*, *list_pop_back,* *list_insert_at*, *list_remove_at*, *list_at* qui vérifient si la position en paramètre est valide et si la liste est non vide, ce qui empêche les accès hors bornes. En outre, l'utilisation d'une sentinelle assure une résistance aux erreurs lors de la manipulation aux bornes de la liste, en réduisant le risque d'erreur de segmentation ou de référencement *NULL*.

En ce qui concerne la fonction *list_map* celle-ci est très particulière. En effet, elle applique une transformation à chaque élément via un foncteur *f* tout en ayant une complexité en un temps linéaire **O(n)**. Ce parcours évite les appels redondants ce qui améliore l'efficacité pour des opérations appliquées sur tous les éléments. Enfin, cet algorithme est robuste, car il est indépendant des transformations effectuées par le foncteur f, ce qui implique que cet algorithme a une flexibilité accrue en fonction des usages.

- **Algorithme de tri fusion sur liste doublement chaînée :**

L'algorithme de tri fusion est utilisé ici pour trier une liste doublement chaînée circulaire en divisant la liste en sous-listes et en les fusionnant de manière ordonnée. Ce type de trie est particulièrement efficace notamment pour les listes chaînées grâce à sa complexité en un temps quasi-linéaire O(n.log(n)) et à son absence de besoin de déplacements d'éléments en mémoire.

1) **Structure** *SubList* : La structure *SubList* est conçue pour stocker des segments de la liste, notamment avec ces deux pointeurs (*head* et *tail*). L'utilisation de

cette structure facilite le découpage et la fusion en segments indépendants ce qui réduit la complexité.

2) **Fonction** *list_split* **:** Cette fonction découpe une liste en deux sous-listes de taille égale à 1 élément près. Elle utilise deux pointeurs (*ptr1* et *ptr2*) pour parcourir la liste afin de trouver le milieu de celle-ci. Comme ptr2 avance deux fois plus vite que *ptr1*, alors, quand *ptr2* arrive à la fin *ptr1* est au milieu qui est l'endroit où il faut séparer la liste en deux. Cette approche est le résultat d'une complexité en **O(n)** de par le parcours unique pour trouver le milieu. De plus, cette méthode est très efficace notamment en réduisant le nombre de manipulations nécessaire en parcourant la liste une fois. Enfin cette fonction est robuste car elle vérifie le cas *NULL* et si la tête est égale à la queue en renvoyant tout simplement la liste telle quelle.
3) **Fonction** *list_merge* **:** Cette fonction prend deux sous-listes triées (*leftlist* et *rightlist*) et les fusionne en une liste triée. Cette fonction réalise la fusion en un temps linéaire **O(n)** car elle parcourt les deux-sous listes en une seule fois. La fonction est optimisée de par l'utilisation de pointeurs qui permet de relier les éléments de chaque sous-liste, ce qui évite les copies de données. Enfin, en utilisant le foncteur *f*, cette fonction s'adapte à différents ordres de tri (croissant, décroissant, etc.)
4) **Fonction** *list_mergesort* **:** Cette fonction applique la méthode du tri fusion de manière récursive. Elle divise, en effet, la liste en sous-listes en faisant appel à *list_split* jusqu'à obtenir des sous-listes vides ou d'un seul élément. En sus, cette fonction admet des performances remarquables de par sa complexité en **O(n.log(n))** ce qui en fait un des algorithmes de tri le plus efficace pour des structures de données comme les listes chaînées. On a, en effet, besoin de **log(n)** étape pour la division et chaque étape nécessitent **O(n)** opérations pour la fusion. Enfin, cette méthode de tri est performant aussi car il ne requiert pas d'accès aléatoire. En effet, la liste est divisée puis fusionnée sans copies supplémentaires tout en limitant l'utilisation mémoire.
5) **Fonction** *list_sort* **:** La fonction *list_sort* permet d'initialiser le processus de tri fusion, qui se fait sur la structure *SubList*, sur la liste principale de structure *s_List*. En utilisant *list_mergesort* elle trie la liste *subList* de type *SubList*. Une fois le tri effectué, elle reconnecte la liste triée à la sentinelle pour maintenir la structure circulaire doublement chaînée. Grâce à l'appel de *list_mergesort* cette fonction conserve la complexité en **O(n.log(n))**. Enfin, cette fonction est performante aussi car elle maintient la structure circulaire de la liste afin de simplifier l'utilisation globale de la liste, même après le tri.

- **Question : Pour la structure** *SubList* **et les fonctions** *list_split*, *list_merge* **et** *list_mergesort*, **doivent-elles être déclarées dans le fichier** *list.h* **ou dans le fichier** *list.c* **?**

Les fonctions *list_split*, *list_merge*, *list_mergesort* et la structure *SubList* doivent être déclarées dans le fichier *list.c*, afin d'éviter d'exposer des détails d'implémentation dans l'interface publique (*list.h*). Ces éléments sont utilisés uniquement en interne (*list.c*) pour l'implémentation du tri fusion. De ce fait, en les plaçant dans le fichier *list.c*, on simplifie l'interface du fichier *list.h* en ne rendant accessible que la fonction *list_sort*, qui est la seule nécessaire pour l'utilisateur.

2) Analyse personnelle du travail effectué, problèmes rencontrés et solutions mises en place

Dans le cadre de ce TP, j'ai implémenté toutes les fonctions nécessaires du TAD *List* doublement chaînée avec sentinelle. J'ai aussi implémenté un tri fusion pour les listes chaînées notamment avec les fonctions *list_split*, *list_merge*,

list_mergesort, et *list_sort*. Globalement, l'implémentation des constructeurs et des opérateurs pour le TAD List se sont révélées sans difficulté particulière car nous les avions vu en cours et restaient pour la plupart simple à implémenter.

La principale difficulté a été dans l'implémentation du tri fusion, surtout dans la fonction *list_merge*. J'ai passé beaucoup de temps à analyser et à dérouler minutieusement cet algorithme pour m'assurer de la bonne fusion des éléments en deux sous-listes triées. Ce travail a nécessité attention particulière aux détails et aux étapes de comparaison, de lien des éléments, ainsi qu'à la gestion des bornes.

Pour la fonction *list_split*, j'avais initialement pensé à calculer la taille de la liste afin d'en déterminer le milieu. Seulement, cette approche est moins efficace que la méthode via les deux pointeurs qui parcourt en même temps la liste, qui, elle optimise la complexité de l'algorithme. Je tiens à signaler que cette méthode de parcourt m'a été donné par ChatGPT en lui demandant, comment je pouvais optimiser ce parcourt.

En ce qui concerne la fonction *list_mergesort*, je n'ai rencontré aucune grande difficulté. En effet, la version récursive simplifie énormément la tâche comparée à une version itérative qui est bien plus complexe.

Enfin, un des aspects les plus minutieux de la mise en place des algorithmes du tri fusion a été la gestion des cas particuliers de la structure *SubList* du fait que celle-ci ne contient pas de sentinelle. En effet, il a fallu vérifier systématiquement les bornes de chaque sous-liste avant d'effectuer des opérations. Néanmoins, le point positif de la chose est que grâce à ces précautions la robustesse du code est garanti.

3) Références concernant les documentations et aides utilisées

Pour la réalisation de ce projet, j'ai consulté plusieurs sources afin de m'assurer que mon implémentation était correcte et efficace :

- ChatGPT : Pour obtenir des conseils techniques (optimisation, robustesse) et résoudre des problèmes spécifiques de codage comme des cas de bornes non pris en compte.
- Camarades : Discussions et échanges d'idées pour clarifier certains concepts.
- Wikipedia : Pour une compréhension théorique de l'algorithme du tri fusion.
- PDF du cours : Matériel fourni en cours, essentiel pour la compréhension des concepts abordés dans le TP et la réalisation de celui-ci.