SEARLE Oliver 22210680

Rapport sur la gestion de la mémoire et l'analyse des performances

Gestion de la mémoire

Après chaque étape complète, je vérifiais l'usage mémoire avec Valgrind. En cas de fuites de mémoire, j'ajoutais les appels à free ou à skiplist_delete là où ils manquaient. Cela m'a permis d'identifier et de corriger les problèmes de mémoire dans mes implémentations.

Problèmes rencontrés et solutions

1. **Fonction de recherche skiplist_search :**
   Au départ, j'obtenais des résultats incorrects. J'ai remarqué que j'utilisais mal la fonction rng, ce qui provoquait la création excessive de nœuds à des niveaux élevés. Après avoir corrigé cette utilisation et révisé la logique de la recherche, j'ai constaté que le dernier décalage (à la fin de skiplist_search) n'était pas pris en compte. Une fois ces ajustements effectués, tous les tests sont passés avec succès.
2. **Fonction de suppression skiplist_remove :**
   Dans les spécifications du sujet, il est indiqué que cette fonction retourne un booléen. Cependant, dans le fichier skiplist.h, la déclaration spécifiait un retour de type SkipList*. J'ai donc modifié le fichier skiplist.h pour le rendre cohérent avec les spécifications.

Analyse des performances et complexité temporelle

Fonctions principales :

1. skiplist_create(int nb_levels)
   - **Complexité temporelle : O(nb_levels)**
   - Raison : Initialisation des pointeurs pour chaque niveau.
2. skiplist_delete(SkipList** d)
   - **Complexité temporelle : O(N)**
   - Raison : Parcours tous les nœuds par le niveau 0.
3. skiplist_size(const SkipList* d)
   - **Complexité temporelle : O(1)**
   - Raison : La taille est simplement retournée.
4. skiplist_at(const SkipList* d, unsigned int i)
   - **Complexité temporelle : O(i)**
   - Raison : Parcours linéaire jusqu'au nœud d'index i.
5. skiplist_map(const SkipList* d, ScanOperator f, void* user_data)
   - **Complexité temporelle : O(N)**
   - Raison : Itération sur chaque nœud de la liste.
6. skiplist_insert(SkipList* d, int value)
   - **Complexité temporelle : O(log N)**
   - Raison : Recherche pour la position d'insertion (**O(log N)**) et mise à jour des pointeurs pour chaque niveau.
7. skiplist_search(const SkipList* d, int value, unsigned int* nb_operations)

    - **Complexité temporelle : O(log N)**
    - Raison : Recherche optimisée en sautant des niveaux.
8. skiplist_remove(SkipList* d, int value)
    - **Complexité temporelle : O(log N)**
    - Raison : Recherche du nœud à supprimer (**O(log N)**) et mise à jour des pointeurs pour chaque niveau.

Fonctions liées à l'itérateur :

1. skiplist_iterator_create(SkipList* d, IteratorDirection w)
    - **Complexité temporelle : O(N)** (pire cas si BACKWARD_ITERATOR).
    - Raison : Recherche du dernier nœud si l'itération est inversée.
2. skiplist_iterator_delete(SkipListIterator** it)
    - **Complexité temporelle : O(1)**
    - Raison : Libération simple de l'itérateur.
3. skiplist_iterator_begin(SkipListIterator* it)
    - **Complexité temporelle : O(N)** (pire cas si BACKWARD_ITERATOR).
    - Raison : Positionnement de l'itérateur au dernier élément.
4. skiplist_iterator_end(SkipListIterator* it)
    - **Complexité temporelle : O(1)**
    - Raison : Vérification simple de l'état de l'itérateur.
5. skiplist_iterator_next(SkipListIterator* it)
    - **Complexité temporelle : O(1)**
    - Raison : Passage au nœud suivant/précédent.
6. skiplist_iterator_value(SkipListIterator* it)
    - **Complexité temporelle : O(1)**
    - Raison : Lecture simple de la valeur.

Appréciation du TP

J'ai apprécié ce TP, particulièrement la création et la manipulation de la structure de données skiplist, bien plus que la précédente. Ce TP m'a posé moins de problèmes, et j'ai trouvé agréable l'inclusion d'un script de test avec des résultats attendus et des retours clairs sous forme de OK pour chaque étape.

Analyse des performances

En s'appuyant sur le test 4, on observe un écart significatif de performances entre l'utilisation de la skiplist dans test_search et l'itération normale des données dans test_iteration. Cela illustre bien la différence de complexité temporelle entre les deux approches :

1. **Recherche avec la skiplist (test_search) :**
        - Nombre minimal d'opérations : 1
        - Nombre maximal d'opérations : 30
        - Nombre moyen d'opérations : 12
    - **Complexité temporelle :**

        - Meilleur cas : **O(1)**
        - Pire cas : **O(log N)**
2. **Recherche par itération (test_iteration) :**
        - Nombre minimal d'opérations : 2
        - Nombre maximal d'opérations : 10,922
        - Nombre moyen d'opérations : 9,707
    - **Complexité temporelle :**
        - Meilleur cas : **O(1)**
        - Pire cas : **O(N)**

Conclusion

Ces résultats confirment l'intérêt des skiplists pour améliorer l'efficacité des recherches. La complexité logarithmique observée avec skiplist_search en fait une structure adaptée pour manipuler des ensembles de données de grande taille, contrairement à une recherche par itération dont le pire cas est linéaire. Pour faire ce TP j'ai utilisé un mix de Chat GPT, stack Overflow pour comprendre comment implémenter les Skiplist basiques, les informations dans le .h surtout de rng.h pour connaître l'implémentation désirée et finalement le TP3 pour les listes doublement chaînées basiques.