BABU Bastien
L2 MIDL
22302768

Compte Rendu TP Algo 3
Séquence 4 : Liste à raccourci – Skiplist.

Ce TP porte sur l'implémentation des Skiplists, une structure de données linéaire probabiliste permettant l'optimisation des opérations de type dictionnaire (insertion, recherche, suppression). Cette structure de données est particulièrement efficace du fait de son excellente complexité moyenne en **O(log(n)).**

L'objectif de ce TP est de comprendre et de mettre en place cette structure, tout en étudiant ses performances et en comparant son efficacité par rapport à d'autres structures de données. Nous y retrouverons une gestion de la probabilité pour l'attribution des niveaux, la robustesse et la performance des algorithmes d'insertion et de suppression, ainsi que la validation des résultats via des tests automatisés fournis.

1) Description des choix d'implémentation

- **Construction et opérations d'une liste à raccourcis :**

La Skiplist est implémentée comme une structure de données basée sur une liste doublement chaînée avec des raccourcis grâce aux niveaux probabilistes. La conception suit les principes expliqués dans le sujet et reprend certaines techniques vu lors de la dernière séquence.

L'implémentation de la Skiplist repose sur trois structures principales, conçues pour garantir de bonnes performances et assurer le bon fonctionnement de la Skiplist. Dans un premier temps, nous avons la strucutre *s_Link* qui représente un lien entre deux nœuds. Elle contient un pointeur *prec* de type *struct s_Noeud* qui pointe vers le nœud précédent et un pointeur *suiv* aussi de type *struct s_Noeud* qui pointe vers le nœud suivant. Cette structure est essentielle pour former la base de la structure doublement chaînée et du tableau de liens par la suite. Ensuite, nous avons la structure *s_Noeud* qui représente un nœud de la Skiplist. Elle contient l'entier *nbLevel* qui est le nombre de niveaux associés au nœud et défini de manière probabiliste, un tableau dynamique de type *struct s_Link* contenant des liens bidirectionnels pour chaque niveau et un entier *value* qui correspond à la valeur stockée dans le nœud. Cette structure est aussi essentielle pour représenter les éléments (nœud) de la Skiplist et de les relier entre eux. Enfin, nous avons la structure qui *s_Skiplist* qui représente la Skiplist elle-même. Elle contient une sentinelle qui est pointeur de type *struct s_Noeud* permettant de simplifier la gestion des bornes (début et fin de liste), un entier *nbLevel* qui est cette fois-ci le nombre maximal de niveaux de la Skiplist, un entier *taille* qui représente le nombre d'éléments présents dans la liste et enfin, *prob* de type *RNG* (définie dans les fichiers fournis) qui permet de stocker la probabilité générée et de l'utilisé pour attribuer les niveaux probabilistes des nœuds. Cette structure globale est essentielle pour centraliser la gestion de la liste.

La fonction de création skiplist_create initialise une liste vide, en configurant la sentinelle, en allouant dynamiquement les liens nécessaires et en initialisant la probabilité grâce à la fonction *rng_initialize*. La fonction *skiplist_delete* libère les nœuds, et leurs liens en parcourant la liste ce qui assure une libération complète et sécurisée de la mémoire.

La performance de la Skiplist est assurée par la gestion des opérations d'insertion et de suppression de manière efficace, notamment avec l'utilisation des raccourcis entre les niveaux. Les fonctions d'insertion (*skiplist_insert*) et de suppression (*skiplist_remove*) utilise le système de niveaux de la liste afin de réduire le nombre de nœuds (éléments) à visiter, ce qui garantit une complexité en un temps logarithmique **O(log(n))** dans la plupart des cas. De part, la détermination aléatoire du niveau des nœuds, on assure une performance optimale en évitant une complexité linéaire dans la manière de rechercher les nœuds. Bien que, les fonctions d'accès à un élément précis (*skiplist_at*), ou de recherche (*skiplist_search*) ont une complexité linéaire **O(n)** dans le pire des cas, elles bénéficient tout de même des raccourcis ce qui leurs permet de réduire le nombre de nœuds à visiter. Elles sont donc globalement plus efficaces qu'une structure de liste simplement chaînée.

La robustesse est aussi assurée de par la vérification des pré-conditions pour les fonctions : *skiplist_insert*, *skiplist_remove, skiplist_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* particulièrement lors des ajouts ou suppressions aux extrémités de la liste.

En ce qui concerne la fonction *skiplist_map*, elle applique une transformation à chaque élément via un opérateur *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.

En conclusion, l'implémentation des fonctions de construction et d'opérations de la Skiplist repose sur une structure performante et robuste, capable de manipuler efficacement des données triées. La combinaison de performances O(1) à O(log(n)) et de flexibilité en fait un outil essentiel pour gérer des données de manière fiable et rapide.

- **<u>Itérateur et recherche linéaire d'une valeur dans une Skiplist :</u>**

L'implémentation de l'itérateur pour la Skiplist permet de parcourir les éléments de la liste dans deux directions avant (*FORWARD_ITERATOR*), ou arrière (*BACKWARD_ITERATOR*), tout en restant dans un temps linéaire **O(n).**
Celui-ci est implémenter à l'aide d'une structure *s_SkipListIterator* contenant le nœud actuellement visité par l'itérateur (*noeudCourant*) qui est un pointeur de type *Nœud* et qui est mise à jour au fur et à mesure que l'itérateur progresse dans la liste. De plus, cette structure contient la direction dans laquelle parcourir la liste, soit *FORWARD_ITERATOR* pour un parcours avant (de la tête vers la queue) *BACKWARD_ITERATOR* pour un parcours arrière (de la queue vers la tête). Enfin, la structure de l'itérateur contient bien entendu une skiplist qui est un pointeur de type *SkipList*, pour parcourir la structure de donnée. Cette structure est efficace, car elle permet des déplacements directs et rapides grâce aux liens bidirectionnels présents dans chaque nœud.

Les fonctions de créations (*skiplist_iterator_create*), et de suppression (*skiplist_iterator_delete*) de l'itérateur sont performante notamment par leur complexité en un temps constant **O(1)**. En effet, *skiplist_iterator_create* initialise un nouvel itérateur en y spécifiant la direction et en positionnant directement le pointeur *noeudCourant* sur la sentinelle. La fonction *skiplist_iterator_delete* libère entièrement la mémoire allouée par la création de l'itérateur évitant toutes fuites de

mémoires possibles. De plus, les fonctions *skiplist_iterator_next*, *skiplist_iterator_begin*, *skiplist_iterator_end* et *skiplist_iterator_value* permettant respectivement de mettre à jour le pointeur du nœud courant (*noeudCourant*) pour passer au suivant ou au précédent selon la direction, de placer l'itérateur au départ de la liste, de vérifier si le pointeur *noeudCourant* a atteint la fin de liste et enfin, de retourner la valeur du nœud actuellement pointé par *noeudCourant*, sont toutes en un temps constant **O(1)**.

L'implémentation de l'itérateur est robuste notamment par les vérifications aux bornes ce qui évite les dépassements en cas de parcours de la liste vide ou après avoir atteint la fin, mais aussi pour son importante flexibilité grâce aux deux directions de parcours possibles (avant ou arrière), ce qui rend l'itérateur adaptable à plusieurs scénarios auxquelles il pourrait être confronté.

L'implémentation de l'itérateur de la skiplist repose sur une structure simple et performante, contenant des fonctions garantissant des opérations constantes **O(1)**. La combinaison de performances, robustesse et de flexibilité fait de l'itérateur un outil essentiel dans la manipulation des données d'une skiplist de manière sécurisée et efficace.

2) Description du comportement du programme sur les jeux de tests

Le programme a été testé avec succès sur les différentes fonctionnalités de la Skiplist à l'aide des fonctions de test fournies. Chaque test a permis de valider le bon fonctionnement des algorithmes en s'assurant que les résultats obtenus correspondent aux attentes.

- **Test de la construction d'une Skiplist :**

La fonction *test_construction* utilise les données du fichier de test pour construire une Skiplist. Les niveaux les valeurs à insérer sont lus dans le fichier grâce à la fonction *buildlist*, puis celle-ci initialise une Skiplist vide, et insère successivement les valeurs. Une fois la liste construite, *test_construction* affiche la taille de la liste suivie des valeurs présentes dedans dans l'ordre croissant.

Tous les tests de cette fonction fonctionnent correctement, ce qui confirme que les éléments sont insérés dans le bon ordre et donc que la fonction *skiplist_insert* fonctionne.

- **Test de la recherche dans une Skiplist :**

La fonction *test_search* effectue une série de recherche dans une Skiplist. Les valeurs à rechercher sont lues depuis un fichier de test. Pour chaque valeur, la fonction skiplist_search est appelée pour déterminer si la valeur est présente dans la liste et pour compter le nombre d'opérations qui ont été nécessaires. Enfin les statistiques du nombre total de valeurs recherchées, celles trouvées et non trouvées ainsi que le nombre minimal, maximal, moyen d'opérations effectuées pour les recherches sont affichées à la fin du test. Lors de la mise en œuvre de cette fonction j'avais une marge de +/- 1 sur l'affichage donc une modification a été apportée sur la fonction *skiplist_search* en ajoutant la ligne : *(*nb_operations)++;* qui a permis d'avoir l'affichage correct.

- **Test de la recherche avec itérateur dans une Skiplist :**

La fonction *test_search_iterator* réalise le même type de recherche que *test_search* mais en utilisant un itérateur. En effet, un itérateur est initialisé avec un parcours dans la direction avant (*FORWARD_ITERATOR*). La liste est ensuite parcourue noeud par noeud à l'aide des fonctions *skiplist_iterator_begin*, *skiplist_iterator_end* et *skiplist_iterator_next* afin de trouver les valeurs recherchées. Enfin, les mêmes statistiques que dans le test précédent sont calculées et affichées. Le test a validé l'utilisation correcte de l'itérateur pour parcourir la liste et effectuer les recherches. Cependant, le nombre d'opérations était systématiquement supérieur à celui obtenu avec *skiplist_search*, car l'itérateur effectue un parcours linéaire sans exploiter le système de niveaux mis en place par la Skiplist.

- **Question : Expliquez dans votre rapport de devoir les raisons de la différence en temps de calcul constatée, particulièrement pour le test numéro 4 (Pour *test_search* et *test_search_iterator*) ?**

La différence entre *test_search* et *test_search_iterator* vient de leur méthode de recherche. En effet *test_search* utilise le système de hiérarchies des niveaux de la Skiplist ce qui lui permet d'avoir une complexité moyenne en O(log(n)), tandis que *test_search_iterator* effectue un parcours linéaire avec une complexité de O(n). Cette différence est particulièrement visible sur des grandes listes, par exemple dans le test n°4, où *test_search* est bien plus rapide.

- **Test de la suppression dans une Skiplist :**

La fonction *test_remove*, teste la suppression des valeurs dans la Skiplist. Les valeurs à supprimer sont lues depuis un fichier de test. Chaque valeur est supprimée à l'aide de *skiplist_remove*, qui supprimer et ajuste les pointeurs des nœuds. Une fois la suppression terminée, la liste est affichée dans l'ordre décroissant en utilisant un itérateur avec la direction arrière (*BACKWARD_ITERATOR*). Les tests effectués ont montré que la suppression fonctionne correctement, même pour les cas limites.

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

La réalisation de ce TP a été particulièrement exigeante et m'a demandé beaucoup de temps, notamment pour l'implémentation des fonctions fondamentales comme *skiplist_insert* et la gestion des structures notamment les structures de la Skiplist (*struct s_Link*, *struct s_Noeud* et *struct s_SkipList*).

L'une des parties les plus difficiles a été l'implémentation de la fonction skiplist_insert. En effet, la gestion des niveaux et la mise à jour correcte des liens (précédent comme suivant) après chaque insertion ont posé beaucoup de problèmes. De plus, la définition de la structure de données a également été complexe, car dans les autres TP celle-ci était fournie tandis que dans ce TP il a fallu la concevoir de manière autonome, ce qui a nécessité une réflexion et de nombreux changement au fur et à mesure que j'avançais dans le TP.

En ce qui concerne la fonction *skiplist_delete*, la difficulté principale était d'éviter les fuites mémoire sur une structure aussi complexe que les skiplists. La

suppression des nœuds et de leurs liens dans une structure aussi imbriquée a causé plusieurs erreurs de type *segmentation fault* en raison de pointeurs mal gérés. Ces erreurs ont aussi été retrouvé dans l'implémentation de la fonction *skiplist_insert*. Une attention particulière a été portée à libérer chaque élément correctement pour éviter toute fuite de mémoire. Nonobstant, une fois ces problèmes résolus, les autres implémentations ont découlés plus facilement.

L'implémentation de l'itérateur s'est révélée relativement simple, car elle reprenait des concepts vus en cours. Cependant, il a quand même fallu s'adapter aux spécificités de la Skiplist, ce qui m'a demandé quelques ajustements pour gérer correctement les directions et les bornes.

Parmi les autres problèmes rencontrés, la mise à jour des liens après insertion et suppression a causé plusieurs erreurs, notamment dans la fonction *skiplist_remove*, où de mauvaises mises à jour des pointeurs *prec* ont conduit à des résultats incorrects lors de l'exécution des fonctions de tests. Par exemple, pour la fonction *test_search_iterator*, les tests 1 et 3 fonctionnaient mais les tests 2 et 4 renvoyaient des erreurs de segmentation.

Afin de résoudre tous ces problèmes j'me suis beaucoup rapproché de mes camarades, notamment en travaillant en groupe et en comprenant nos erreurs et le fonctionnement précis d'une liste à raccourcis. Ces échanges et aides de mes camarades ont été précieux pour comprendre et corriger mes problèmes. En outre, j'ai beaucoup débogué à la « main » en mettant des *print* afin de voir d'où provenaient mes erreurs. Enfin, bien que, l'utilisation de ChatGPT ait fournis quelques pistes, la complexité de la Skiplist a limité son efficacité. Celui-ci m'a notamment aidé pour comprendre certaines erreurs comme l'erreur des « Double free », mais aussi à mettre les *print* pour déboguer et enfin à faire ce qu'on appelle de l'allocation dynamique contigüe pour diminuer la dispersion mémoire.

Pour finir, ce TP a été particulièrement long et exigeant mais très formateur notamment car il a permis d'approfondir la compréhension des skiplists et de son implémentation. Les défis rencontrés ont également renforcé l'importance d'une gestion rigoureuse des pointeurs et des allocations mémoire pour garantir le bon fonctionnement des tests.

4) 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 :

- Camarades : Ils ont été la principale source d'aide, m'apportant des explications et des conseils précieux pour résoudre les problèmes complexes que je rencontrais.
- ChatGPT : Discussions et échanges d'idées pour clarifier certains concepts et erreurs rencontrés.
- Fichiers de présentation des Skiplists en anglais : Fournis par les professeurs, ils ont permis de mieux comprendre les concepts théoriques de la liste à raccourcis.
- PDF du cours : Matériel fourni en cours, essentiel pour la compréhension des concepts abordés dans le TP et la réalisation de celui-ci.