University of Liege

faculty of applied sciences

INFO0027-2 Programming techniques - ALGORITHMICS
PROJECT

Bytestream mapper

Academic Year: 2024-2025

Professor: Laurent Mathy

ANDRIAMAMPIANINA Bruce : s2302253
NOEL Thomas : s194057

Assistant professor: Jakub Duchateau

Contents

1 Algorithme

Le projet repose sur l'utilisation d'un arbre rouge-noir (Red-Black Tree, RBTree) pour gérer efficacement les opérations de décalage et de suppression de bytes dans un flux de données. L'algorithme implémente plusieurs fonctions pour gérer ces opérations à l'aide de structures de données équilibrées, permettant de maintenir une complexité logarithmique pour les insertions, suppressions et recherches.

1.1 Initialisation des arbres

L'initialisation de l'arbre rouge-noir se fait dans la fonction `RBTreeInit`. Chaque arbre contient un NIL spécial, représentant une feuille vide, qui sert également de sentinelle pour éviter les erreurs d'accès. Cette structure est utilisée dans toute l'implémentation pour représenter des positions vides ou des sous-arbres inexistants. Un arbre rouge-noir est ainsi configuré avec un nœud racine pointant vers NIL.

1.2 Insertion et rotations

L'insertion dans un arbre rouge-noir se fait par la fonction `RBTreeInsert`, qui insère un nouveau nœud dans l'arbre tout en maintenant les propriétés de l'arbre rouge-noir (équilibre des couleurs et des hauteurs). Lors de l'insertion, si un nœud parent est rouge, l'arbre effectue des rotations à gauche et à droite (`leftRotate` et `rightRotate`) pour restaurer l'équilibre. Les rotations ajustent également le champ `lazyShift` des nœuds pour gérer efficacement les décalages.

1.3 Suppression et gestion des décalages

La fonction `MAGICremove` est responsable de l'ajout d'une opération de suppression dans l'arbre `deleteTree` et d'un décalage dans l'arbre `shiftTree`. Le processus est marqué par un `timestamp` qui permet de suivre les suppressions et les modifications dans l'ordre des événements. Lorsqu'une suppression est effectuée, l'algorithme enregistre le décalage dans l'arbre des décalages tout en marquant la suppression dans l'arbre des suppressions.

1.4 Recherche et mappage

La fonction `MAGICmap` permet de rechercher la position d'un élément après application des opérations de suppression et de décalage. L'algorithme recherche dans `shiftTree` puis dans `deleteTree` pour déterminer si un décalage ou une suppression a eu lieu sur la position donnée. Si un élément a été supprimé, la fonction retourne -1.

2 Complexité

L'algorithme repose sur l'utilisation d'un arbre rouge-noir, une structure de données équilibrée permettant d'assurer des opérations efficaces. Analysons la complexité des principales opérations.

2.1 Complexité de l'insertion

L'insertion dans un arbre rouge-noir se fait en `O(log n)`, où n est le nombre de nœuds dans l'arbre. Cela est dû à la nécessité de maintenir l'équilibre de l'arbre après chaque insertion, ce qui peut nécessiter jusqu'à `O(log n)` rotations pour restaurer les propriétés de l'arbre.

2.2 Complexité de la suppression

La suppression d'un nœud dans un arbre rouge-noir se fait également en `O(log n)`, car elle nécessite de parcourir l'arbre pour localiser le nœud à supprimer, puis de rééquilibrer l'arbre, ce qui peut également nécessiter jusqu'à `O(log n)` rotations.

2.3 Complexité de la recherche

La recherche dans l'arbre rouge-noir pour un élément donné (que ce soit pour un décalage ou une suppression) a une complexité de `O(log n)`, car l'arbre est équilibré et chaque recherche consiste à parcourir un chemin de hauteur logarithmique.

2.4 Complexité des opérations de mappage

L'opération `MAGICmap` effectue une recherche dans deux arbres rouges-noirs distincts, `shiftTree` et `deleteTree`. Chaque recherche dans ces arbres a une complexité de `O(log n)`, ce qui donne une complexité totale de `O(log n)` pour l'opération de mappage.

2.5 Complexité de la gestion des timestamps

L'utilisation de `timestamp` permet de suivre les modifications de manière efficace. L'incrémentation du `timestamp` est une opération constante, `O(1)`, ce qui ne contribue pas à la complexité globale des autres opérations.

3 Test plan and performance study

3.1 Objectifs du Test

Le plan de test vise à :

- Vérifier l'intégrité et la cohérence des opérations d'ajout, suppression et modification de bytes.
- Tester la gestion de l'arbre rouge-noir utilisé pour structurer le flux de données.
- Évaluer la robustesse du système face à des entrées invalides ou extrêmes.
- Assurer des performances acceptables pour différentes tailles de flux de données.

3.2 Stratégie de Test

3.2.1 Types de Test

Les tests seront divisés en plusieurs catégories :

- **Tests unitaires** : Vérification des fonctions individuelles.
- **Tests d'intégration** : Vérification des interactions entre les modules.
- **Tests de performance** : Évaluation du temps d'exécution et de la consommation mémoire.
- **Tests de robustesse** : Vérification du comportement du programme en cas d'erreurs.

3.3 Cas de Test

ID	Description	Statut
T1	Ajouter un byte à un emplacement vide	En attente
T2	Supprimer un byte existant	En attente
T3	Ajouter une séquence de bytes et vérifier leur insertion	En attente
T4	Tester la suppression d'une plage de bytes	En attente

T5	Vérifier la cohérence des positions après plusieurs ajouts et suppressions successifs	En attente
T6	Tester `MAGICmap` après insertion pour s'assurer que les positions sont correctes	En attente
T7	Tester `MAGICmap` après suppression pour s'assurer que les décalages sont correctement gérés	En attente
T8	Tester `MAGICmap` sur des zones non modifiées	En attente
T9	Effectuer un test de stress avec un grand volume de données	En attente
T10	Supprimer une plage dépassant la taille du flux et vérifier le comportement	En attente

3.4 Critères d'Acceptation

Un test est considéré comme réussi si :

- Le résultat obtenu correspond exactement au résultat attendu.
- Aucune erreur imprévue n'est générée.
- Le programme maintient une performance stable même pour de grandes quantités de données.

4 Conclusion