Rapport d'Algorithmique Avancée

Rapport Réalisé par :
Adam Abbas ABBAS
Et le code en Duo avec :
SALIMATOU BAH

Encadrant:
Stefan BALEV
et
Véronique JAY

Le 15/12/2024

TABLE DES MATIERES

INTRODUCTION

Un arbre rouge-noir (ARN) est un **arbre binaire de recherche équilibré**. Chaque nœud de l'arbre est associé à une couleur, soit **rouge** ou **noir**. Ces couleurs ne sont pas simplement esthétiques ; elles sont essentielles pour maintenir l'équilibre de l'arbre et garantir des opérations efficaces. Et pour cela, des règles sont nécessaires à être respecté.

A. Mécanisme de fonctionnement des ARN

Les arbres rouge-noir utilisent un ensemble de règles et de mécanismes pour maintenir leur équilibre. Voici comment ils fonctionnent en pratique tout en respectant les règles fondamentales si dessous :

A.1 Les règles fondamentales des ARN :

1.Chaque nœud est soit rouge, soit noir.
2.La racine de l'arbre est toujours noire.
3.Toutes les feuilles nulles (nœuds sans valeur, sentinelles ou nœuds externes) sont considérées comme noires.
4.Si un nœud est rouge, alors ses deux enfants sont noirs.
5.Tout chemin depuis un nœud donné jusqu'à ses feuilles descendantes contient le même nombre de nœuds noirs.

Apres chaque action (Insertion ou suppression), ces règles cites peuvent se briser, et donc il faut passer par ces mécanismes pour corriger l’équilibre et le bon fonctionnement de l’arbre tout en respectant les règles fondamentales des Arbres Rouge-Noir (on peut dire c'est la correction des règles brisées)

A.2. Les mécanismes pour garder le bon fonctionnement

1. Insertion : Lorsqu'un nouveau nœud est ajouté, il est initialement coloré en rouge. Si l'insertion viole l'une des règles fondamentales (par exemple, si le parent du nouveau nœud est rouge), des ajustements sont nécessaires. Ces ajustements se font par des rotations et des recolorations pour restaurer les propriétés de l'arbre rouge-noir.

- Rotations : Il existe deux types de rotations (gauche et droite) qui réorganisent les nœuds pour corriger les violations. Une rotation gauche fait pivoter un sous-arbre vers la gauche, tandis qu'une rotation droite le fait vers la droite.

- Recolorations : Changer la couleur des nœuds est une méthode utilisée pour équilibrer les sous-arbres après une insertion.

2. Suppression : Lorsqu'un nœud est supprimé, les règles doivent également être respectées. Si le nœud supprimé est rouge, il n'y a généralement pas de violation des règles. Cependant, si le nœud supprimé est noir, cela peut créer un déséquilibre que les rotations et recolorations doivent corriger.

C. Comparaison des ARN aux Structures Linéaires et aux ABR classiques.

C.1. Comparaison des ARN aux Structures Linéaires.

Les **structures linéaires** (listes chaînées, tableaux) organisent les données de manière séquentielle. Elles sont simples à comprendre et à implémenter, mais difficiles pour faire la recherche rapide : Dans le pire des cas. L'insertion et la suppression sont efficaces pour les listes chaînées si la position est connue, mais coûteuses (O**(n)**) dans les tableaux en raison des décalages nécessaires. Leur simplicité en fait un choix adapté pour des besoins basiques, bien que les listes chaînées ne soient pas accessibles par index.

Les **arbres rouge-noir (ARN)** , quant à eux, sont des arbres binaires de recherche équilibrés qui garantissent une recherche rapide même pour de grands ensembles de données, tout en maintenant automatiquement leur ordre. Cependant, leur implémentation est plus difficile que celle des structures linéaires, et ils nécessitent une légère surcharge mémoire pour stocker la couleur des nœuds.

C.2. Comparaison entre ABR classiques et Arbre Rouge-Noir

Les ABR classiques sont des arbres binaires sans mécanisme d'équilibrage, ce qui les rend simples à implémenter et performants (O(logn)) lorsque les données sont insérées de manière aléatoire. Cependant, en cas de déséquilibre (par exemple, insertion en ordre croissant), leur performance peut se dégrader jusqu'à O(n), rendant les opérations inefficaces. Leur simplicité est donc contrebalancée par l'absence de garantie de complexité logarithmique.

Les **ARN**, en revanche, ajoutent des règles de coloration pour maintenir un équilibre strict. Cela garantit une hauteur minimale et des performances stables (**O(logn)**) pour toutes les opérations, même dans les pires scénarios (nous allons démontrer par les graphes). Toutefois, cette robustesse s'accompagne d'une complexité accrue dans l'implémentation, ainsi que d'un léger surcoût en temps et en mémoire dû aux rotations et recolorations nécessaires pour préserver l'équilibrage

C.3. Récapitulatif de comparaison dans un tableau

Caractéristiques	Structures Linéaires	ABR Classiques	Arbres Rouge-Noir
Complexité Recherche	O(n)	O (log n) à O(n)	O (log n)
Équilibrage	O (1) à O(n)	O (log n) à O(n)	O (log n)
Complexité Insertion	N/A	Non garanti	Garanti
Simplicité d'Implémentation	Facile	Moyenne	Complexe
Usage Mémoire	Faible à Modérée	Modérée	Modérée
Usage Optimisé Pour	Petits ensembles, accès séquentiel	Données aléatoires, simplicité	Grandes données, performances stables

Table 1 : *Tableau de Comparaison des Structures Linéaires, ABR Classiques et Arbres Rouge-Noir.*

D. Les choix spécifiques d'implémentation :

Pour notre projet, nous avons choisi d'implémenter une classe d'Arbre Rouge-Noir (ARN) qui implémente l'interface Collection de Java. Ce choix s'explique par le besoin de gérer efficacement des ensembles de données de grande taille tout en maintenant un équilibre optimisé. Les ARN sont particulièrement adaptés à cette tâche en raison de leurs propriétés de rééquilibrage automatique, qui garantissent des opérations de recherche, d'insertion et de suppression en temps logarithmique.

D.1 COMMENT FONCTIONNE NOTRE CLASSE ARN IMPLEMENTEE

Pour bien comprendre le fonctionnement de notre classe ARN implémentée, il est essentiel de se pencher sur les méthodes clés qui permettent de maintenir l'équilibre de l'arbre. Ces méthodes sont conçues pour gérer les insertions et les suppressions tout en respectant les règles fondamentales de l'ARN, évitant ainsi les déséquilibres. Voici ces méthodes plus détaillées :

D.1.1. Méthode *add(E key) :*

La méthode add insère une nouvelle clé dans l'arbre rouge-noir. Après chaque insertion, des règles peuvent se briser, et surtout la règle 4 qui dit : "si un nœud est rouge, alors ses deux fils

sont noirs". En rappelle, chaque nœud est inséré en tant que rouge dans l'arbre. Cependant, voici un Scenario d'insertion d'un nœud :

On recherche la position d'insertion en : Parcourant l'arbre depuis la racine pour trouver où insérer la clé. Si la clé est plus petite, on avance vers le sous-arbre gauche, sinon vers le droit.

Insertion du nœud après avoir trouvé sa position d'insertion en tant que nœud rouge, inséré comme enfant gauche ou droit du nœud trouvé.

Correction des propriétés ARN : Après insertion, les règles de l'ARN peuvent être violées et La méthode **corrigerProprietesApresInsertion** est appelée pour rééquilibrer l'arbre et restaurer ses propriétés

D.1.2. Méthode corrigerProprietesApresInsertion(Noeud<E> Noeud)

Cette méthode corrige les violations des règles ARN après une insertion dans diffèrent cas pour que l'**ARN** respecte les 5 règles d'un arbre rouge noir

Si le parent est noir : Aucune correction n'est nécessaire (les propriétés ARN sont respectées).
Si le parent est rouge :
Cas 1 : L'oncle du nœud est rouge. Alors :
Changer la couleur du parent et de l'oncle à noir.
Changer la couleur du grand-parent à rouge.
Continuer la correction au niveau du grand-parent.
Cas 2 : L'oncle est noir ou absent.
Rotation gauche ou droite, selon la position relative du nœud.
Ajuster les couleurs pour rétablir les règles ARN.

D.1.3. Méthode remove(Object key)

La méthode remove supprime une clé de l'arbre.

Localisation du nœud à supprimer : Parcours de l'arbre pour trouver le nœud contenant la clé.

Suppression du nœud : Si le nœud a deux enfants, il est remplacé par son successeur en ordre croissant. Si le nœud a un ou aucun enfant, il est directement supprimé.

> **Correction des propriétés ARN** : Si le nœud supprimé est noir, cela peut violer la règle du chemin noir. La méthode **corrigerProprietesApresSuppression** est appelée pour restaurer les propriétés.

D.1.4. Méthode corrigerProprietesApresSuppression(Noeud<E> Noeud)

Cette méthode corrige les violations après une suppression.

Cas 1 : Le frère du nœud est rouge.

Rotation autour du parent pour rendre le frère noir.

Ajuster les couleurs.

Cas 2 : Le frère et ses enfants sont noirs.

Le frère devient rouge.

La correction remonte au niveau du parent.

Cas 3 : Le frère a au moins un enfant rouge.

Rotation autour du parent ou du frère, selon les positions.

Ajustement des couleurs pour équilibrer l'arbre.

D.1.5. Méthodes de Rotation

Méthode **rotationGauche**(Noeud<E> Noeud):La rotation gauche est utilisée pour rééquilibrer l'arbre en cas de déséquilibre à droite. Alors ces étapes s'imposent :

1. Le sous-arbre droit de Noeud devient son parent.
2. Le sous-arbre gauche de l'ancien sous-arbre droit devient le sous-arbre droit de Noeud.
3. Les pointeurs sont mis à jour pour relier correctement les nœuds.

Méthode **rotationDroite**(Noeud<E> Noeud):La rotation droite est utilisée pour rééquilibrer l'arbre en cas de déséquilibre à gauche. Alors ces étapes s'imposent :

1. Le sous-arbre gauche de Noeud devient son parent.
2. Le sous-arbre droit de l'ancien sous-arbre gauche devient le sous-arbre gauche de Noeud.
3. Les pointeurs sont ajustés pour refléter ces changements.

D.1.6 Autres Méthodes

Méthode getOncle(Noeud<E> Noeud): identifie l'oncle d'un nœud (le frère de son parent). Utile pour la correction après insertion.

Méthode findMinimum(Noeud<E> Noeud): Trouve le nœud avec la plus petite clé dans un sous-arbre donné. Utilisé lors de la suppression pour localiser le successeur en ordre croissant.

Méthode replaceParentsChild(Noeud<E> parent, Noeud<E> oldChild, Noeud<E> newChild): Remplace un enfant d'un parent par un nouveau nœud. Utilisé lors de la suppression pour reconnecter correctement les nœuds.

E. Les résultats de l'étude expérimentale.

Pour l'étude expérimentale, On effectue des tests sur ARN et ABR de taille N clé avec N nombres des clés pour évaluer les performances de deux arbres. On effectue essentiellement les tests sur l'ajout des clés du plus petit au plus grand dans les deux arbres, un deuxième ajout des clés aléatoires dans les deux arbres, la recherche de toutes les clés qui sont dans les deux arbres et les clés qui ne sont pas dans les arbres en commençant par exemple la recherche de N à 2N ce qui donne les clés qui n'existent pas dans l'arbre.

E.1 Etude Expérimentale en temps d'Insertion

- **Courbe de l'Ajout des clés aléatoires sur l'ARN vs l'ABR :**

Tests effectue avec n=100000.les mêmes valeurs sont ajoutées dans ABR et ARN

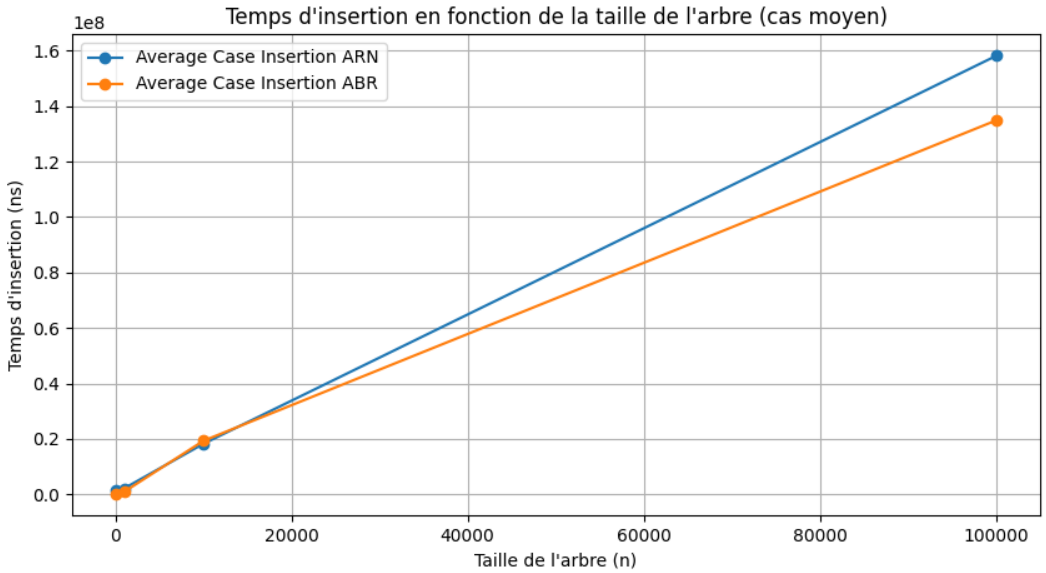

Figure 1 : *Courbe de l'Ajout des clés aléatoires sur l'ARN vs l'ABR : cas moyens*

REMARQUE : en voyant les graphes, on constate que la courbe en bleu qui est celle de l'ARN prend plus de temps a commencé l'ajout des clés, cela est dû par rapport aux corrections qu'effectuent l'ARN pour équilibrer les clés. Sur ce test, l'ABR prend moins de temps pour s'exécuter car les valeurs sont tirées de façon aléatoire parce que l'ABR n'effectue aucune correction sur les clés. L'équilibrage de l'ARN sur les clés de grande valeur aléatoire fait que l'ARN s'exécute moins rapidement que l'ABR. Cela ne justifie pas que l'ARN est moins efficace que l'ABR mais plutôt le contraire car un arbre rouge-noire offre une garantie

d'équilibre, tandis que l'ABR classique dépend de la distribution des clés pour maintenir des performances optimales.

- **Courbe de l'Ajout des clés croissantes sur l'ARN vs l'ABR :**

Courbe de l'Ajout des clés trie (du plus petit au plus grand) sur l'ARN vs l'ABR :
Tests effectue avec n=100000.les mêmes valeurs sont ajoutées dans ABR et ARN.

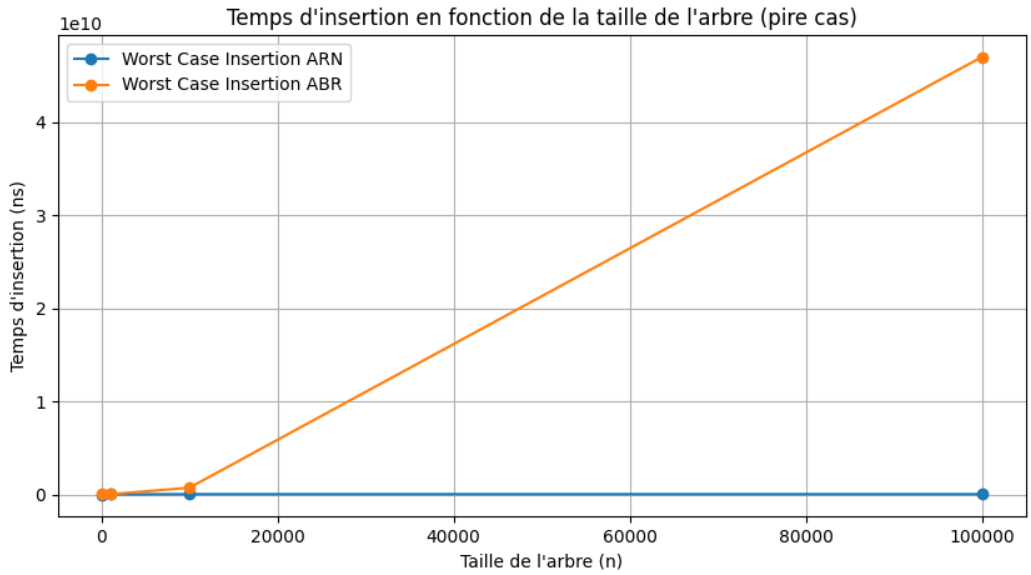

Figure 2 : *Courbe de l'Ajout des clés trie sur l'ARN vs l'ABR : Pire des cas.*

Remarque : une grande différence en exécution du temps entre l'ARN et l'ABR est remarqué .Les clés sont insérées dans l'ordre croissant, donc l'ajout dans un ARN est relativement rapide. Les Nœuds sont insérés comme des nœuds rouges à gauche et les rotations sont minimisées. La complexité est O(n) où n est le nombre de clés. Tandis que l'ABR dépend de l'équilibre de l'arbre. Si l'arbre n'est pas équilibré l'ABR devient une liste chaînée linéaire, et l'ajout peut être en O(n) dans les pires des cas. Si l'arbre est équilibré, l'ajout peut être proche de O(nlog n) en moyenne.

E.2 Etude Expérimentale en temps de Recherche

❖ **Courbe de la recherche des clés (cas moyens) dans l'ARN vs l'ABR :**

Courbe de recherche des clés dans l'ARN vs l'ABR :

Tests effectue avec n=100000.les mêmes valeurs sont recherchées dans ABR et ARN.

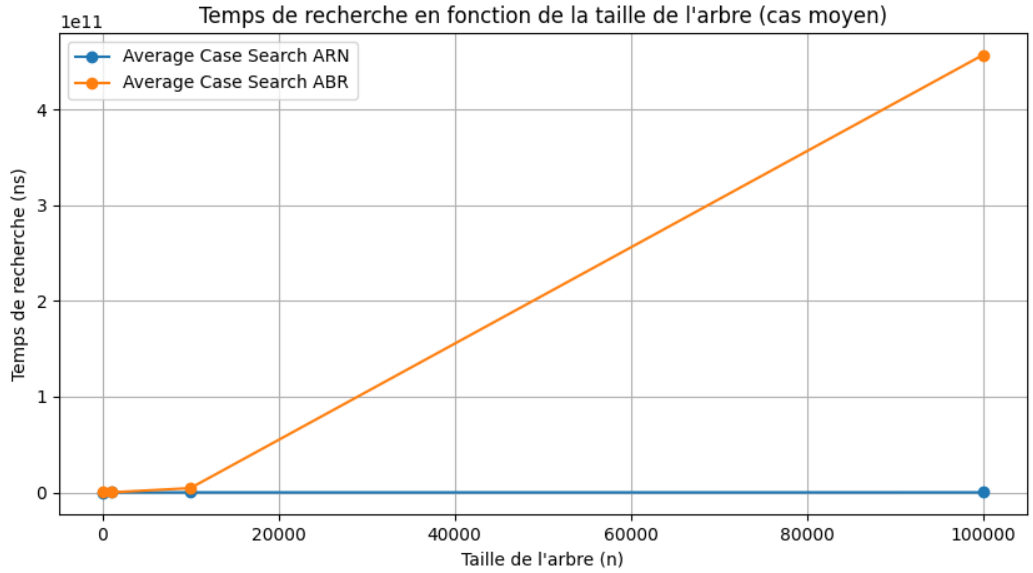

Figure 3 : *Courbe de la recherche des clés dans l'ARN vs l'ABR : cas moyens*

Remarque : On constate une différence énorme en exécution en temps entre l'ARN et l'ABR.Même si les clés sont insérées de façon ordonnée, l'ajout de rotations et des ajustements de couleur dans l'ARN maintient l'équilibre. Tandis que l'ABR dépend toujours de l'équilibre. Si l'ABR est équilibré la recherche des clés est comme dans le cas de l'ARN, la complexité est O(log n) . Le cas où l'ABR n'est pas équilibré, ce qui est le cas sur la courbe. Les clés sont insérées de manière ordonnée, la structure de l'arbre devient une liste chaînée, et la complexité devient linéaire, O(n) où n est le nombre de nœuds. Comme chaque nœud est comme fils droit du précédent, formant une structure linéaire.

❖ **Courbe de la recherche des clés dans l'ARN vs l'ABR** :

Courbe de recherche des clés (clés triées) dans l'ARN vs l'ABR :
Tests effectue avec n=100000.les mêmes valeurs sont recherchées dans ABR et ARN.

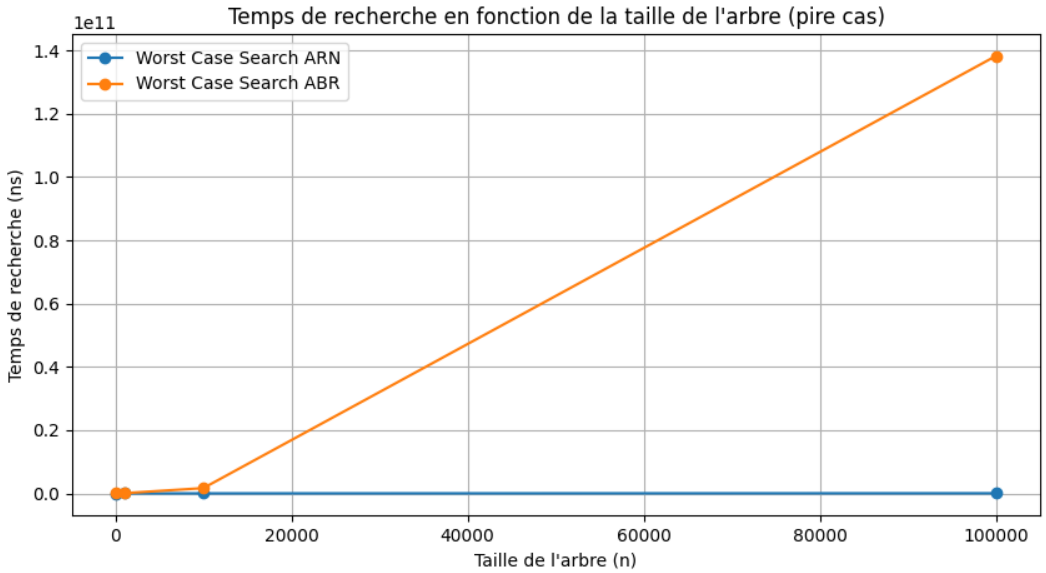

Figure 4 : *Courbe de la recherche des clés dans l'ARN vs l'ABR : pire des cas.*

Remarque : On constate une grande différence en exécution en temps entre l'ARN et l'ABR.L'algorithme de recherche suit un parcours logarithmique à travers l'arbre lorsque l'on recherche des clés croissantes qui ne sont pas dans l'ARN ou l'ABR.La complexité moyenne de recherche en ARN est O(log n), où n est le nombre de nœuds dans l'arbre. La propriété d'équilibrage de l'ARN, qui maintient la hauteur de l'arbre à un niveau logarithmique par rapport au nombre de nœuds, en est la cause. La complexité de la recherche pour un ABR dépend de l'équilibre de l'arbre. Lorsque l'ABR est équilibré, la complexité de la recherche est généralement également O(log n). Cependant, dans le pire des cas, la complexité peut atteindre O(n), où n est le nombre de nœuds, lorsque l'ABR est déséquilibré et ressemble à une liste chaînée.

CONCLUSION

En résumé, l'Arbre Rouge-Noir (ARN) et l'Arbre Binaire de Recherche (ABR) partagent la propriété fondamentale de permettre des opérations de recherche, d'insertion et de suppression efficaces. Cependant, la distinction clé réside dans la garantie d'équilibre offerte par l'ARN, assurant une complexité logarithmique prévisible même dans des scénarios d'insertion non

idéaux. En revanche, la performance de l'ABR dépend fortement de l'équilibre spécifique de l'arbre, ce qui peut conduire à des variations de complexité allant de logarithmique à linéaire en fonction de la distribution des clés. À noter qu'il existe également des arbres AVL (AVL tree est nommé après ses chercheurs : Georgy **A**delson-**V**elsky and Evgenii **L**andis), qui imposent un équilibre encore plus strict que les ARN, garantissant une hauteur minimale pour optimiser les temps de recherche au détriment d'une complexité légèrement accrue pour les mises à jour.