

Rapport Algo Avancée

Table des matières

Présentation des différents arbres de recherche.....	2
Arbres Binaires de Recherche (ABR)	2
Principe de base :	2
Performance :	2
Arbre Rouge et Noir (ARN)	2
Principe de base :	2
Fonctionnement :	2
Performance :	3
Présentation de l'Implémentation	3
Arbres Binaires de Recherche (ABR)	3
1. Structure de base :	3
2. Contraintes et Interface :	3
3. Opération add (E e)	3
4. Opération contains(Object o) et findNode(E e).....	4
5. Opération remove(Object o).....	4
6. Opération iterator()	4
Arbre Rouge et Noir (ARN)	4
1. Le Choix de la Sentinelle (NIL) :	4
2. Le Nœud :	5
3. Opérations de Rééquilibrage :	5
4. Opérations de Collection :	6
Résultats de l'étude expérimental.....	6
Comparaison insertion.....	6
Comparaison insertion :	7
Conclusion.....	8

Présentation des différents arbres de recherche

Arbres Binaires de Recherche (ABR)

Les **Arbres Binaires de Recherche (ABR)** (en anglais, *Binary Search Trees* ou **BST**) sont une structure de données fondamentale utilisée pour stocker des éléments de manière organisée.

Principe de base :

Un ABR est un arbre binaire où chaque nœud possède au maximum deux enfants. La propriété clé qui définit un ABR est la suivante :

- Pour tout nœud, toutes les clés dans son sous-arbre gauche sont **inférieures** à la clé du nœud.
- Pour tout nœud, toutes les clés dans son sous-arbre droit sont **supérieures** à la clé du nœud.

Performance :

La hauteur de l'arbre est le facteur déterminant de la performance pour les opérations d'insertion, de suppression et de recherche.

- **Cas Idéal (Arbre Équilibré) :** Si l'arbre est parfaitement équilibré, sa hauteur est logarithmique, $O(\log(n))$, où n est le nombre de nœuds. Les opérations s'exécutent donc en $O(\log(n))$.
- **Cas le Plus Défavorable (Arbre Dégénéré) :** Si les éléments sont insérés dans un ordre strictement croissant ou décroissant, l'arbre se dégrade en une **liste chaînée**. Sa hauteur devient $O(n)$, et les opérations se dégradent à $O(n)$.

Ce cas défavorable est la **limitation majeure** des ABR classiques.

Arbre Rouge et Noir (ARN)

Les **Arbres Rouge et Noir (ARN)** (en anglais, *Red-Black Trees* ou **RBT**) sont une variante sophistiquée des ABR conçue spécifiquement pour garantir de bonnes performances en tout temps. Ils font partie de la famille des *arbres auto-équilibrés* (comme les arbres AVL).

Principe de base :

Un ARN est un ABR qui maintient son équilibre en associant une couleur (rouge ou noire) à chaque nœud et en respectant **cinq propriétés strictes** :

1. **Propriété ABR :** C'est un ABR valide.
2. **Propriété de Couleur :** Chaque nœud est soit rouge, soit noir.
3. **Propriété de la Racine :** La racine est noire.
4. **Propriété des Nœuds Rouges :** Si un nœud est **rouge**, ses enfants doivent être **noirs**. (Pas deux nœuds rouges consécutifs sur un chemin de la racine à une feuille.)
5. **Propriété de Profondeur Noire :** Pour tout nœud, tout chemin simple de ce nœud à une feuille (nœud NIL) contient le **même nombre** de nœuds noirs. Ce nombre est appelé la *profondeur noire*.

Fonctionnement :

Après chaque opération d'insertion ou de suppression, l'arbre peut violer les propriétés ARN. Pour rétablir l'équilibre, deux mécanismes sont utilisés :

1. **Recoloration :** Changer la couleur d'un ou plusieurs nœuds.
2. **Rotations :** Modifier la structure de l'arbre localement (rotations gauches et droites) pour modifier les relations parent-enfant tout en préservant la propriété ABR.

Ces mécanismes garantissent que l'ARN reste **partiellement équilibré**. La propriété 5 assure que le plus long chemin de la racine à une feuille est au maximum deux fois plus long que le chemin le plus court. Cela signifie que la hauteur maximale h de l'arbre est toujours bornée par $O(\log(n))$.

Performance :

En garantissant une hauteur logarithmique, les ARN offrent une **complexité temporelle garantie** pour les opérations de base, même dans le pire des cas :

- Recherche : $O(\log(n))$
- Insertion : $O(\log(n))$
- Suppression : $O(\log(n))$

Présentation de l'Implémentation

Arbres Binaires de Recherche (ABR)

1. Structure de base :

Classe Interne `Nœud` :

- Elle encapsule la structure de base de l'arbre : une `cle` (clé/valeur de type `E`), un pointeur vers le sous-arbre gauche et un pointeur vers le sous-arbre droit.
- **Justification** : La clé `E` doit être de type générique `Comparable<E>` pour permettre la comparaison et le respect des propriétés de l'ABR.

Champs de la Classe `ArbreBinaireRecherche` :

- `Racine` : Le point de départ de l'arbre.
- `Taille` : Un compteur qui suit le nombre d'éléments, permettant une complexité $O(1)$ pour l'opération `size()`.
- **Justification** : Ces champs sont essentiels pour gérer l'état de l'arbre et fournir des métriques de base efficaces.

2. Contraintes et Interface :

Généricité et Contrainte « `Comparable` » :

- La classe est déclarée comme `ArbreBinaireRecherche<E> extends Comparable<E>>`.
- **Justification** : Un ABR repose sur la capacité de comparer les clés pour décider où aller (gauche si `key < noeud`, droit si `key > noeud`). L'utilisation de `Comparable` assure que la méthode `compareTo` est disponible pour tous les éléments insérés, ce qui est fondamental pour les opérations d'insertion, de recherche et de suppression.

Implémentation de `Collection<E>` :

- **Justification** : Implémenter cette interface permet à votre arbre d'être traité comme n'importe quelle autre collection standard de Java (comme `ArrayList` ou `HashSet`), garantissant l'accès aux méthodes courantes (`add`, `contains`, `remove`, `size`, `iterator`, etc.).

3. Opération `add(E e)`

Choix : Implémentation itérative (boucle `while`) :

- Elle recherche la position correcte en naviguant dans l'arbre (gauche ou droite) jusqu'à atteindre `null`, en gardant une trace du parent du nœud actuel (courant).

Justification :

- **Propriété ABR** : L'insertion est correcte car elle maintient la propriété du BST : si $e < \text{parent.cle}$, l'insertion se fait à gauche ; sinon, à droite.
- **Gestion des Duplicatas** : Elle retourne `false` et ne fait rien si l'élément est déjà présent (`cmp == 0`), garantissant que l'arbre ne contient pas de doublons.
- **Efficacité** : L'insertion a une complexité temporelle de $O(h)$ où h est la hauteur de l'arbre. Dans le cas moyen d'un arbre équilibré, cela est $O(\log(n))$.

4. Opération `contains(Object o)` et `findNode(E e)`

Choix : Utilisation d'une méthode utilitaire itérative `findNode` qui parcourt l'arbre.

Justification :

- **Efficacité** : La recherche tire parti de la propriété du BST, en éliminant la moitié de l'arbre à chaque comparaison, ce qui conduit à une complexité de $O(h)$.
- **Sécurité** : L'implémentation de `contains(Object o)` gère correctement la conversion de type et les exceptions `ClassCastException`, conformément aux bonnes pratiques de l'interface `Collection`.

5. Opération `remove(Object o)`

Choix : Implémentation qui utilise le **successeur immédiat** (le plus petit élément du sous-arbre droit) pour remplacer le nœud à supprimer ayant deux enfants.

Justification :

- **Gestion des Cas** : L'implémentation gère les trois cas de suppression :
 - **0 enfant (feuille)** : Le nœud est simplement déconnecté de son parent.
 - **1 enfant** : Le nœud est remplacé par son unique enfant.
 - **2 enfants** : Le nœud à supprimer est remplacé par son successeur (le minimum du sous-arbre droit).

6. Opération `iterator()`

Choix : Implémentation d'un itérateur basé sur une **pile** (`Stack`) pour effectuer un **parcours en ordre**.

Justification :

- **Ordre Croissant** : Un parcours en ordre d'un ABR produit une séquence des clés triées par ordre croissant. C'est le comportement attendu et le plus utile pour une collection ordonnée comme un ABR.
- **Itération sans Récursion** : L'utilisation de la pile permet de simuler la récursivité nécessaire au parcours en profondeur tout en offrant une approche **itérative** et en évitant les problèmes de débordement de pile (`StackOverflowError`) pour les arbres très profonds. L'initialisation (`init`) pousse tous les nœuds gauches jusqu'à la racine sur la pile, et `next()` désempile, visite, puis pousse le sous-arbre droit.
- **Efficacité** : Chaque nœud est visité une seule fois. La complexité de l'itération totale est $O(n)$, et l'appel à `next()` est amorti en $O(1)$.

Arbre Rouge et Noir (ARN)

1. Le Choix de la Sentinelle (`NIL`) :

Implémentation : On utilise un nœud constant et unique, `NIL`, de couleur `NOIR`, pour représenter toutes les feuilles nulles (externes) et la référence parent de la racine.

Justification :

- **Simplification de l'Algorithme :** Les algorithmes d'Arbre Rouge-Noir (surtout `fixAfterInsert` et `fixAfterDelete`) nécessitent de vérifier la couleur des enfants ou du parent. Si `null` était utilisé, il faudrait gérer `NullPointerException` ou des conditions `if (x != null)` constantes. La sentinelle `NIL` (toujours `NOIR`) permet de traiter uniformément tous les enfants/parents manquants comme des nœuds noirs, rendant le code des rotations et des corrections beaucoup plus propre et sûr.
- **Initialisation :** La racine est initialisée à `NIL` lorsque l'arbre est vide, simplifiant le test d'arbre vide (`racine == NIL`).

2. Le Nœud :

Implémentation : Le nœud stocke la `cle`, les pointeurs `gauche`, `droit`, et `parent`, ainsi qu'un booléen `color` (`ROUGE` ou `NOIR`).

Justification :

- **Parent Pointer :** Le pointeur `parent` est indispensable pour les opérations de rééquilibrage (`rotate` et `fix`). Contrairement à un ABR simple, un ARN a besoin de remonter l'arbre pour effectuer des rotations et des recolorations.
- **Couleur (Par Défaut ROUGE) :** Un nouveau nœud est toujours inséré comme **ROUGE**.

3. Opérations de Rééquilibrage :

Rotations (`rotateLeft`, `rotateRight`) :

- **Implémentation :** Mise en œuvre des rotations standard de l'arbre binaire (gauche et droite).
- **Justification :**
 - **Maintien de la Propriété ABR :** Les rotations sont le mécanisme central pour changer la structure de l'arbre tout en préservant la propriété de l'Arbre Binaire de Recherche (tous les éléments à gauche de `X` sont $\leq X$, et tous ceux à droite sont $\geq X$).
 - **Utilisation de NIL :** L'utilisation de `NIL` pour tester si `x` est la racine (`x.parent == NIL`) simplifie la gestion des bords de l'arbre.

Correction après Insertion (`fixAfterInsert`) :

- **Implémentation :** Gère les trois cas classiques de violation des propriétés (deux nœuds rouges consécutifs), avec des symétries pour le parent gauche/droit.
- **Justification :**
 - **Cas 1 (Oncle ROUGE) :** Nécessite une simple **recoloration** (`parent`, `oncle` \rightarrow `NOIR` ; `grand-père` \rightarrow `ROUGE`). La violation remonte au grand-père.
 - **Cas 2 (Triangle, Oncle NOIR) :** Nécessite une **rotation simple** pour transformer le "triangle" en un "cas 3 en ligne", puis on procède au cas 3.
 - **Cas 3 (Ligne, Oncle NOIR) :** Nécessite une **rotation double** (une rotation du `parent`, suivie de la rotation du `grand-père`) et une recoloration finale. Le `grand-père` devient `ROUGE` et le nouveau `parent` (ancien `parent/enfant` de l'arbre) devient `NOIR`.
 - **Complexité :** La boucle `while` monte vers la racine. Elle est conçue pour garantir que la hauteur noire de tous les chemins reste constante et que l'arbre demeure équilibré, assurant une complexité $O(\log(n))$ pour l'insertion.

Correction après Suppression (`fixAfterDelete`) :

- **Implémentation :** C'est la partie la plus complexe, impliquant quatre cas pour restaurer les propriétés (hauteur noire constante) après la suppression d'un nœud `NOIR`.
- **Justification :**

- La suppression d'un nœud NOIR (ou d'un ROUGE qui remplace un NOIR) crée un "noir en trop" ou un "noir manquant" dans le chemin. Le `fixAfterDelete` vise à propager ce déséquilibre vers le haut jusqu'à ce que la hauteur noire soit restaurée ou que la racine soit atteinte.
- **Cas 1 à 4 :** Ces cas gèrent les différentes configurations de couleur entre le nœud affecté (x), son frère (w), et les enfants du frère, en utilisant des rotations et des recolorations complexes pour absorber le "double noir" ou le propager.
- `transplant` : La méthode utilitaire `transplant` (remplacer u par v) est la façon standard de manipuler les liens parents/enfants de manière atomique lors de la suppression.

4. Opérations de Collection :

`add(E e)` : Utilise la recherche de position standard de BST pour trouver l'emplacement, gère `parent` avec `NIL` correctement, et appelle `fixAfterInsert`.

`remove(Object o)` : Met en œuvre la logique standard de suppression :

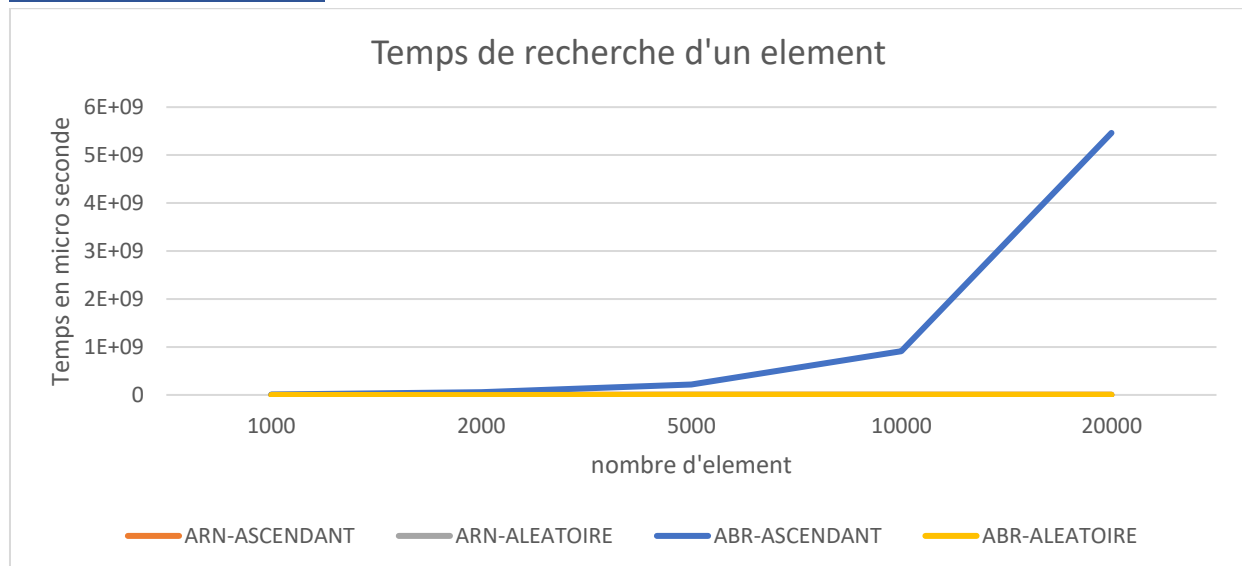
- Recherche du successeur (si deux enfants).
- Utilisation de `transplant` pour re-lie les nœuds.
- Appel crucial à `fixAfterDelete(x, xparent)` si le nœud retiré (ou remplacé), y , était NOIR.

`iterator()` : L'itération est implémentée avec une `Stack` pour un parcours **en ordre**.

- **Justification** : C'est la même logique que pour un ABR simple. L'itération produit un ensemble de clés **triées**. Elle utilise également `NIL` comme condition d'arrêt, assurant la cohérence avec la structure de l'arbre.

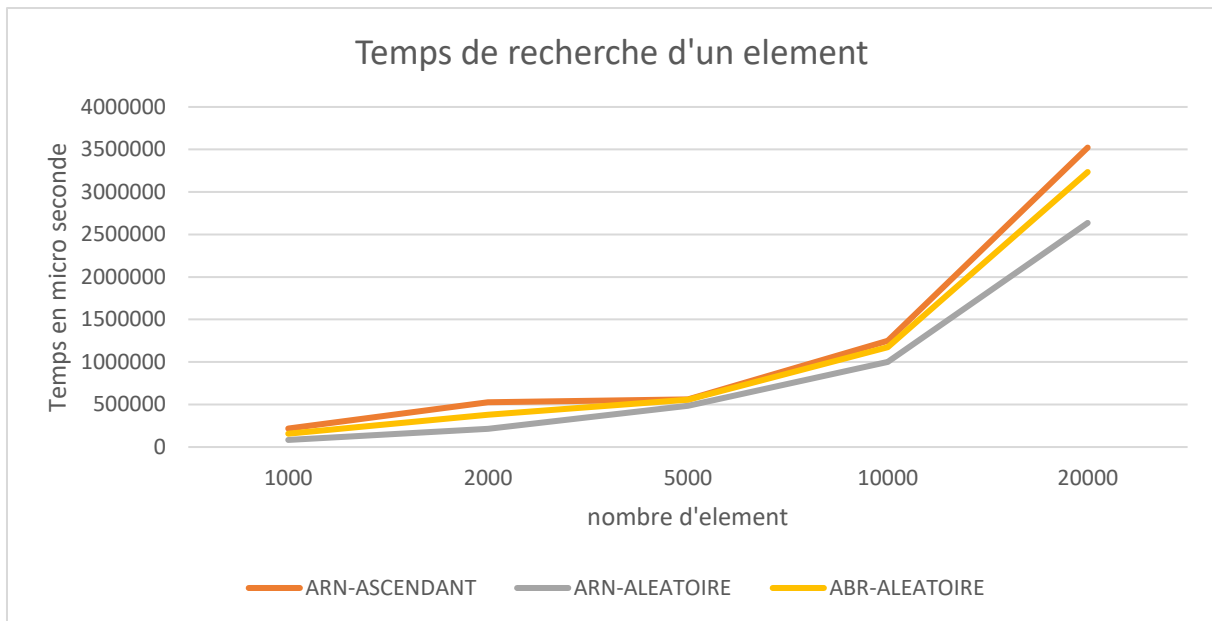
Résultats de l'étude expérimental

Comparaison insertion



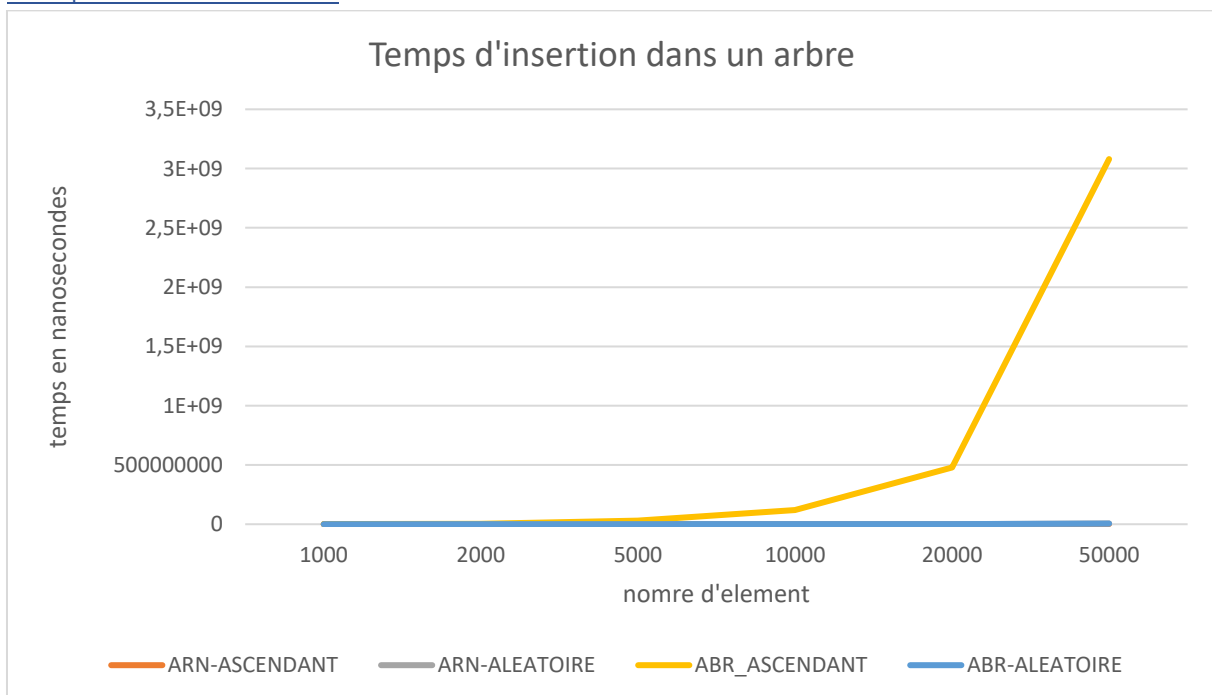
On remarque sur le graphe que le temps de recherche de l'ABR en ascendant est très élevé. Elle a une croissance linéaire. Ce qui correspond à la complexité attendu $O(n)$. En effet, les valeurs sont insérées de manière linéaire (1 puis 2 puis 3) donc cela ressemble à une liste chaînée.

L'analyse précise des temps des autres scénarios est difficile en raison de la courbe ABR-ASCENDANT, qui domine largement le graphique. Son temps d'exécution étant beaucoup plus élevé, il écrase visuellement les différences entre les autres, rendant leur comparaison moins évidente.



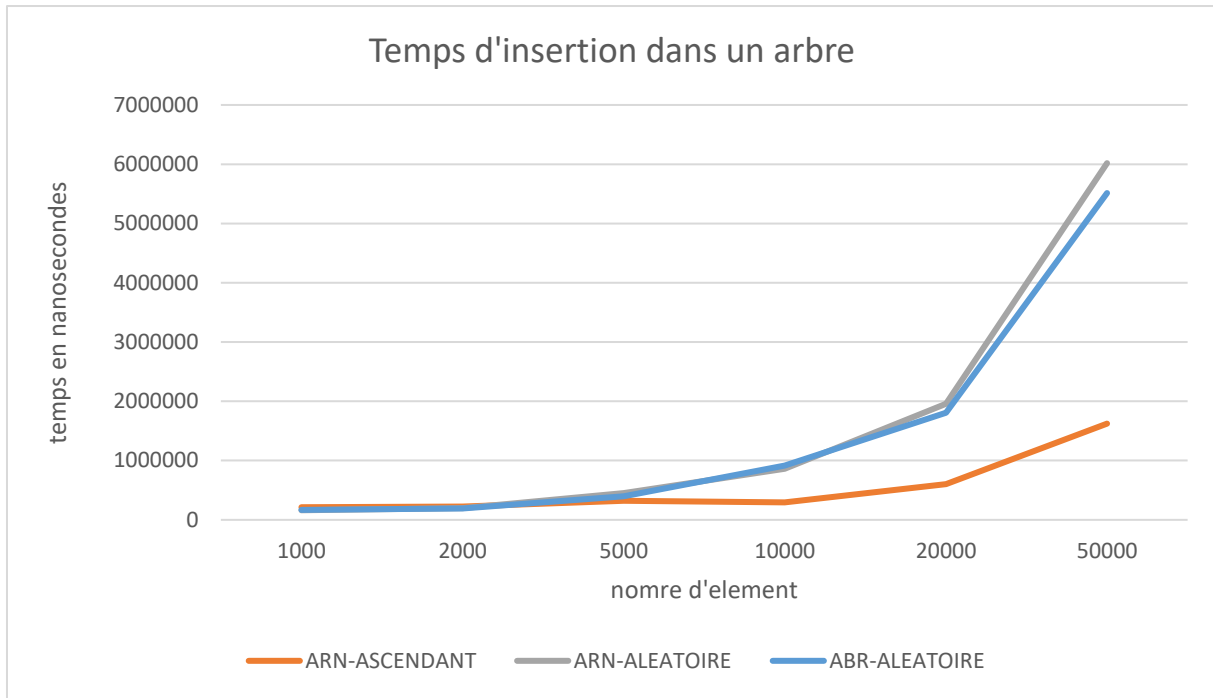
On remarque que les 3 autres courbes se suivent de près. Les 3 courbes suivent une tendance logarithmique. Cela est conforme à la complexité théorique attendue $O(\log(n))$. En effet un arbre rouge et noir est équilibré, donc, quelle que soit la taille de l'arbre. Cela implique que la complexité est de $O(\log(n))$. Pour l'ABR-ALEATOIRE, l'ordre d'insertion est aléatoire. Grâce à cette insertion aléatoire, l'arbre a de fortes chances de rester relativement équilibré (en moyenne donc : $O(\log(n))$).

Comparaison insertion :



On remarque sur le graphe que le temps d'Insertion de l'ABR en ascendant est très élevé. Elle a une croissance linéaire. Ce qui correspond à la complexité attendu $O(n)$. En effet, les valeurs sont insérées de manière linéaire (1 puis 2 puis 3) donc cela ressemble à une liste chaîner.

L'analyse précise des temps des autres scenarios est difficile en raison de la courbe ABR-ASCENDANT, qui domine largement le graphique. Son temps d'exécution étant beaucoup plus élevé, il écrase visuellement les différences entre les autres, rendant leur comparaison moins évidente.



On remarque que les que la courbe ARN-ALEATOIRE et ABR-ALEATOIRE sont pratiquement identique. Ils suivent une complexité de $O(\log(n))$. Cette complexité était attendue.

Cependant, on remarque que ARN-ASCENDANT est largement en dessous des 2 autres courbes. Et donc, il ne suit pas la complexité attendu $O(\log(n))$ attendu. En effet, théoriquement, il devrait pratiquement se superposer au 2 autres courbes. Cela est dû, peut-être, a une erreur dans le codage. Ou, cela est peut-être dû au fait que l'insertion se fait de manière linéaire.

Conclusion

L'ABR offre une insertion, une recherche et une suppression théoriques en $O(\log(n))$ lorsqu'il est équilibré. Son implémentation est relativement simple. Cependant, le cœur de sa limitation a été mis en évidence par l'analyse expérimentale : lorsque les données sont insérées dans un ordre séquentiel (ascendant ou descendant), l'ABR se dégénère en une liste chaînée, transformant sa complexité en $O(n)$. Le graphique de comparaison des insertions, largement dominé par le scénario ABR-ASCENDANT, illustre de manière spectaculaire cette vulnérabilité majeure.

L'ARN a été présenté comme la solution à cette limitation. En ajoutant une propriété de couleur et en intégrant des mécanismes d'auto-équilibrage (Recolorations et Rotations), l'ARN garantit que la hauteur maximale h de l'arbre est toujours bornée par $O(\log(n))$, même dans le pire des cas.

En conclusion, si l'ABR offre une solution efficace pour un ensemble de données dont on suppose qu'il est inséré de manière aléatoire, l'Arbre Rouge et Noir s'impose comme la solution de choix lorsque la complexité temporelle garantie dans le pire des cas est un critère essentiel. Il est la référence dans les bibliothèques logicielles (comme `TreeMap` ou `TreeSet` en Java) pour fournir un ensemble ordonné fiable.