Études sur les arbres binaires

Killian Reine

1 Introduction

1.1 Le projet

Le présent rapport s'inscrit dans le cadre de l'unité d'enseignement transversale « Algorithmique avancée », dispensée aux étudiants de troisième année de licence informatique enseignée par Stefan Balev et Véronique Jay à l'Université Le Havre Normandie.

Le projet réalisé avec le langage de programmation Java vise à comparer deux structures les **Arbres Binaires de Recherche – ABR** et les **Arbres Rouge et Noir – ARN**. Toutes deux appartiennent à la même famille de structures de données, celle des arbres binaires.

L'implémentation des structures doit s'appuyer sur le modèle l'interface `Collection<T>` en Java.

1.2 Présentation des Arbres binaires – AB

1.2.1 Définition et vocabulaire de base

Un **arbre binaire** est une structure de données arborescente dans laquelle chaque nœud peut avoir au maximum deux enfants, généralement appelés *fils gauche* et *fils droit*. Cette structure permet en fait d'organiser les données de manière hiérarchique, facilitant ainsi la recherche, l'insertion et la suppression d'éléments.
Les arbres binaires s'accompagnent aussi d'un certain vocabulaire que je m'écorcerai d'utiliser dans ce présent rapport.

— La **racine** représente le point d'entrée de l'arbre, « le sommet ».

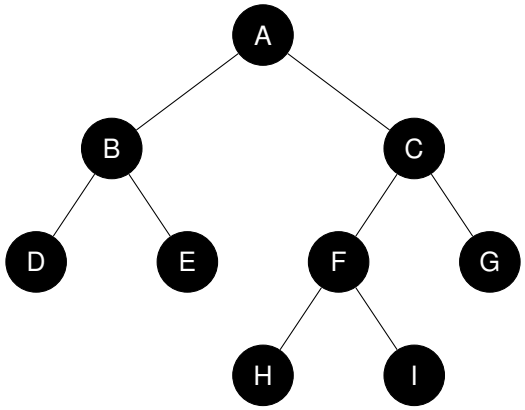
— Le **sous-arbre gauche** et **sous-arbre droit** tout deux représentent les deux fils que peut avoir un nœud, ces derniers pouvant être eux même un arbre binaire ou vide.

— Une **feuille** représente tous les nœuds de l'arbre qui ne possèdent aucun fils.

— La **hauteur** de l'arbre représente le nombre de niveau le plus grand entre la racine et une feuille.

— La **profondeur** est le nombre d'arêtes entre la racine et le nœud le plus éloigné.

1.2.2 Exemple d'un arbre binaire



1.3 Présentation des Arbres Binaires de Recherche – ABR

1.3.1 Définition et propriétés des Arbres Binaires de Recherche

Un **Arbre Binaire de Recherche (ABR)** est un arbre binaire particulier dans lequel chaque nœud contient une *clé* et satisfait la propriété suivante :

— La clé de tout nœud du **sous-arbre gauche** est **inférieure** à celle du nœud parent.

— La clé de tout nœud du **sous-arbre droit** est **supérieure** à celle du nœud parent.

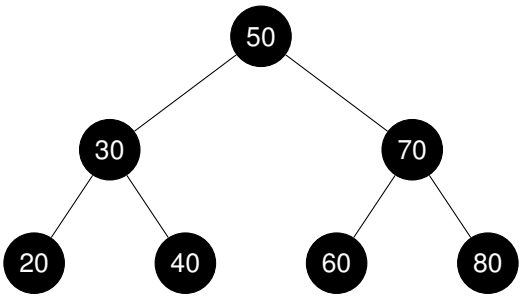
Cette structure permet d'effectuer des opérations de recherche, d'insertion et de suppression de manière efficace, généralement en temps proportionnel à la hauteur de l'arbre.

— La **racine** contient la clé de départ.

— Les **sous-arbres gauche et droit** sont eux-mêmes des ABR.

— Une **feuille** est un nœud qui n'a aucun enfant, donc la clé n'a pas de sous-arbre.

1.3.2 Exemple d'un ABR



1.4 Présentation des Arbres Rouges et Noirs – ARN

1.4.1 Généralités sur les Arbres rouges-noirs

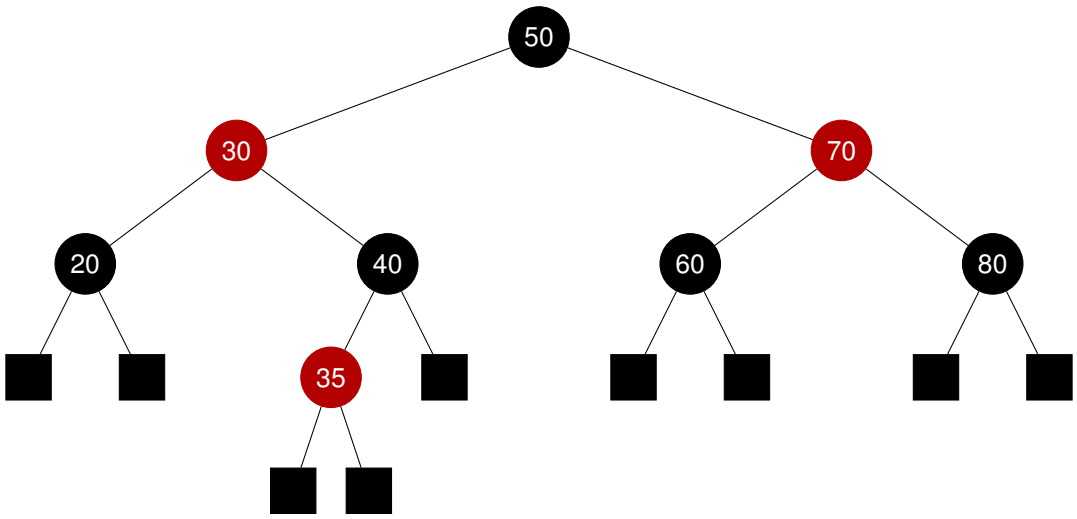
Un **Arbre Rouge et Noir (ARN)** est une structure de données appartenant à la famille des arbres équilibrés. On appelle **arbre équilibré** est un arbre qui conserve une profondeur relativement uniforme entre ses différentes branches, ce qui permet de réduire, en moyenne, le temps d'accès à une donnée, quelle que soit la clé recherchée.

Un arbre rouge-noir est un arbre binaire de recherche tel que :

1. Chaque nœud de l'arbre est soit rouge, soit noir.
2. La racine de l'arbre est noire.
3. Les feuilles sont noires.
4. Si un nœud est rouge alors ses deux fils sont noirs.
5. Pour chaque nœud le chemin reliant à des feuilles contiennent le même nombre de nœuds noirs.

On appelle **hauteur noire** notée $h_n(x)$ le nombre de nœuds noirs sans compter x sur le chemin de x vers la feuille.

1.4.2 Exemple d'un ARN



On remarque bien que chaque nœud est soit noir soit rouge. Chaque nœud rouge possède bien deux fils noir. La hauteur noire de cet arbre est donnée par $h_n = 1$ en partant de la racine, on ne rencontre qu'un seul nœud noir pour aller à une feuille.

Dans notre modèle, on considère que chaque nœud a deux fils initialement nuls. Ainsi, les feuilles représentent des nœuds qui n'encapsulent aucune valeur. Pour éviter de devoir stocker un même nœud `null` autant de fois qu'il y a de feuilles, on considère que ce nœud est unique et que toutes les feuilles pointent vers ce dernier, **la sentinelle**.

1.5 Présentation d'une structure linéaire : `ArrayList<T>`

`ArrayList<T>` est une classe générique du package `java.util`, elle implémente `List<T>`. Elle permet de **stocker les éléments dans un tableau dynamique**. Un tableau de taille ajustable automatiquement lors de l'ajout et la suppression d'éléments.

— Elles est basée sur la structure d'un tableau dynamique

— Elle conserve l'ordre d'insertion des éléments

— Elle autorise doublons et valeurs nulles

— L'accès aux éléments se fait via les indices

1.6 Comparaison ABR, `ArrayList<T>`

La principale différences entre les structures d'Arbre binaire de recherche et d'`ArrayList` réside dans la manière de stocker les éléments qui est rappelé ci-contre :

— **La structure `ArrayList`**
Les éléments sont ordonnés selon l'ordre d'insertion dans le temps.

— **L'Arbre Binaire de Recherche – ABR**
Les éléments sont stockés en respectant les règles de positions suivantes : plus petit que la valeur du nœud à gauche sinon à droite.

On remarque assez rapidement que en fin de compte, les éléments d'une `ArrayList` sont placés sans vraiment avoir de positions définies de manière logique alors que pour l'ABR, les éléments sont placés selon une règle logique.

Cette différence de taille, va en réalité révéler de grandes différences dans les performances des deux structures au niveau de l'ajout, et surtout lors de la recherche et de la suppression d'éléments.

2 Implémentation des structures

2.1 La classe `BinaryTree<T>`

2.1.1 Rôle général de la classe

La classe `BinaryTree<T>` est une classe abstraite qui permet de représenter de manière générale les caractéristiques communes à nos arbres binaires. L'ensemble des méthodes abstraites qu'elle comporte devront être implémentées dans les classes qui l'étendent comme `ABR<T>` que l'on détaillera plus tard.

2.1.2 La classe interne `Node`

Tout arbre binaire est naturellement constitué de nœuds liés entre eux. La classe `Node` est interne à la classe `BinaryTree<T>` de sorte à ce que toutes les classes qui étendront `BinaryTree<T>` aient accès en même temps à cette classe.

Les attributs de la classe `Node`

`Node`			
type	**attribut**	**visibilité**	**description**
`T`	`value_`	`private`	valeur encapsulée par le nœud
`Node`	`father_`	`private`	père du nœud
`Node`	`left_`	`private`	sous-arbre gauche du nœud
`Node`	`right_`	`private`	sous-arbre droit du nœud

Ayant décidé dans ma classe que toutes mes variables étaient privées, les accesseurs (getters) et mutateurs (setters) sont alors nécessaires. Chaque getters commencera par le préfixe `get` suivit du nom de la variable sans l'underscore et chaque setters suivra le même principe mais avec le préfixe `set`.

2.1.3 Retour sur la classe `BinaryTree<T>`

Attributs de la classe `BinaryTree`

BinaryTree			
type	**attribut**	**visibilité**	**description**
`Node`	`root_`	`protected`	la racine de l'arbre
`int`	`nbNode_`	`protected`	le nombre de nœuds dans l'arbre
`Comparato<? super T>`	`cmp_`	`protected`	comparateur permettant de déterminer un ordre des valeurs de type `T`

Les constructeurs

La classe `BinaryTree` possède au total dans l'implémentation trois constructeurs différents permettant chacun d'initialiser un arbre binaire.

```
1 // [1] Arbre vide et comparator null
2 public BinaryTree();
3
4 // [2] Arbre vide mais initialisation du comparator
5 public BinaryTree(Comparator<? super T> cmp);
6
7 // [3] Arbre initialisé avec une collection d'éléments
8 public BinaryTree(Collection<? extends T> collect);
```

Listing 1 – Constructeurs de la classe `BinaryTree`

— L'initialisation de l'arbre binaire vide implique une racine initialisée à `null` et un nombre de nœud à `0`.

— L'initialisation à l'aide d'une collection, créé un arbre binaire où les éléments sont ajoutés selon l'ordre donné par la collection.

Méthode abstraites de la classe `BinaryTree`

Comme évoqué un peu plus haut, la classe `BinaryTree<T>` est une classe générique abstraite, elle contient alors des méthodes dites abstraites [1] :

```
1  // Méthode d'ajout d'un/de plusieurs noeud.s
2  public abstract boolean add(T n_);
3  public abstract boolean addAll(Collection<? extends T> collect_);
4
5  // Méthode de suppression d'un/de plusieurs noeud.s
6  public abstract boolean removeValue(T n_);
7  public abstract boolean removeAllValue(Collection<? extends T> collect_);
8
9  // Rechercher un noeud
10 public abstract Node research(Object n_);
```

Listing 2 – Méthodes abstraites de la classe `BinaryTree`

Les méthodes ci-dessus devront être implémentées par toutes les classes qui héritent directement de `BinaryTree<T>`. Par exemple, la classe `ABR<T>` qui implémentera cette dernière devra définir une à une toutes ces méthodes. L'ensemble des méthodes abstraites seront une surcharge des méthodes de la classe `Collection<T>` adaptés aux arbres binaires.

Méthodes concrètes de la classe `BinaryTree`

Les méthodes concrètes représentent une sorte d'implémentation par défaut de méthodes qui peuvent au besoin être surchargés par les classes qui étendent `BinaryTree`. La réécriture des méthodes dépendra de l'arbre sur lequel on travail, par exemple sur les arbres rouge-noir, la méthode d'affichage devra être réécrire pour afficher en couleur les nœuds. Ceci n'est qu'un exemple simple.

1. **Méthode abstraite** – qui n'a pas de corps elles devront alors être définies par les classes qui étendent la classe abstraite.

Ci dessous une liste des signatures des méthodes définies directement dans la classe `BinaryTree`, ce sont les méthodes directement utilisables par les classes qui l'étendent. Chaque méthode est précédée de son utilité en commentaire, le code de ces dernières est disponible dans le dépôt rendu.

```java
// Renvoie le nombre de noeuds dans l'arbre
public int size();

// Vider l'arbre, le remettre à '0'
public void clear();

// Supprimer un ou plusieurs éléménents
public boolean remove(Object o);
public boolean removeAll(Collection<T> collect);

// Vérifier si une valeure est déjà dans l'arbre
public boolean contains(Object o);

// L'abre est vide ?
public boolean isEmpty();

// Méthode d'affichage
// Reprise de la méthode sur le TP ABR @author : Stefan BALEV
public String toString();
private void toString(Node x, StringBuffer buf, String path, int len);
```

Listing 3 – Méthodes concrètes de `BinaryTree`

Ainsi, l'utilité d'avoir des méthodes concrètes[2] est qu'on fournit une implémentation par défaut des méthodes, donc toutes les classes qui étendent `BinaryTree` auront aussi accès à ses méthodes pouvant alors les utiliser directement, ou alors les redéfinir à leur convenance selon les besoins.

La classe imbriquée : `TreeIterator`

La classe interne `TreeIterator` permet de fournir un **itérateur infixe** (in order) pour un arbre binaire en suivant le schéma suivant :

sous-arbre gauche → racine → sous-arbre droit

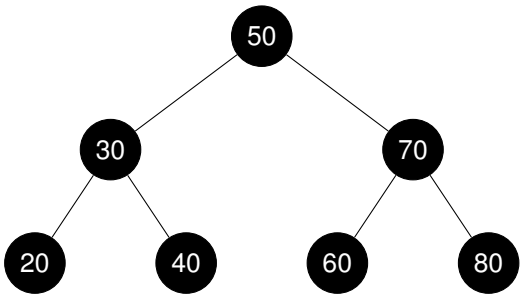
En fait, cela permet de parcourir des arbres comme les arbres binaires de recherche dans l'ordre des valeurs contenues dans les nœuds. La sous classe imbriquée implémente l'interface `java.util.Iterator<T>`, cela permet alors de parcourir l'arbre de manière naturelle en utilisant une **bouble for-each**.

Principe général de l'itérateur

- L'itérateur débute au plus petit élément de l'arbre obtenu grâce à la méthode `Node.minimum()`
- Il vérifie grâce à `Iterator.hasNext()` si il y a un nœud suivant
- Dans ce cas l'itérateur avance dans l'arbre en utilisant la méthode `Node.suivant()`.

La méthode d'affichage

Comme précisé un peu plus tôt, la méthode d'affichage utilisée pour les arbres binaires est celle du TP2 sur les ABR, la méthode d'affichage a été écrite par Stefan BALEV et elle permet d'afficher l'arbre de gauche dans le terminal à droite :




```java
// Affichage de l'arbre
// @author : Stefan BALEV

                    +-- 80
          +-- 70 --|
          |         +-- 60
-- 50 --|
          |         +-- 40
          +-- 30 --|
                    +-- 20
```

Listing 4 – Affichage de l'arbre binaire dans le terminal

2. **Méthode concrète –** qui a un corps, une définition

2.2 La classe `ABR<T>`

2.2.1 Présentation générale de la classe

Comme l'indique le nom de la classe, `ABR<T>` est une classe qui a pour principal objectif de traiter les Arbres Binaires de Recherche – ABR. Pour rappel, un arbre binaire de recherche est un arbre binaire où chaque nœud est placé selon les règles suivantes :

— La clé de tout nœud du **sous-arbre gauche** est **inférieure** à celle du nœud parent.

— La clé de tout nœud du **sous-arbre droit** est **supérieure** à celle du nœud parent.

La classe `ABR` étend la classe `BinaryTree` ce qui signifie qu'elle accède a toutes les variables et méthodes (sauf statiques et privées) de `BinaryTree`. Sans pour autant oublier que `BinaryTree` est une classe abstraite et qu'elle contient un certain nombre de méthodes abstraites qui doivent être définies dans la classe `ABR`.

2.2.2 Méthodes abstraites définies par `ABR<T>`

La méthode `add(T n_)`

La méthode `add(Object)` est une méthode qui est définie dans l'interface `Collection<T>` qui permet d'ajouter un élément dans une collection. Pour le cas des Arbres Binaire de Recherche – ABR, on doit faire en sorte d'ajouter un nœud tout en respectant, la règle des inférieurs à gauche et des supérieurs à droite. Il ne faut pas oublier de gérer les différents cas :

— L'arbre est vide

— L'arbre n'est pas vide

```
1  Ajouter(v)
2      // [ CAS 1 : Arbre vide ]
3      si arbre vide
4          racine = nouveau(v), retourner vrai
5
6      // [ CAS 2 : arbre non vide ]
7      courant = racine
8      tant que courant existe
9          si v = courant.valeur ALORS retourner faux
10         parent = courant
11         courant = (v < courant.valeur) ? courant.gauche : courant.droite
12
13     insérer v à gauche ou droite de parent
14     retourner vrai
```

Listing 5 – Algorithme de la méthode d ajout dans un ABR

CAS 1 – L'arbre est vide
La nœud ajouté devient la racine de l'arbre, on peut renvoyer vrai.

CAS 2 – L'arbre n'est pas vide
On va alors parcourir l'arbre jusqu'à tomber sur un nœud qui est `null`/vide, on descend dans l'arbre en fonction des différents résultats des comparaisons successives entre la valeur du nœud à ajouter et la valeur du nœud courant.

- SI le nœud est déjà dans l'arbre, on retourne faux car on l'a pas ajouté
- Si on arrive sur un nœud nul, alors c'est là qu'on va ajouter le nouveau nœud

L'ordre des éléments contenus dans les différents nœuds de l'arbre est créé grâce au `Comparator<T>` définit lors de la création de l'arbre ou lors de la comparaison de deux nœuds sur le type `T`.

La méthode `attachNode(Nod p_, Node n_)`

Notre classe `ABR` fournit une méthode appelée `attachNode(Node, Node)` prenant deux nœuds en paramètres, le père du nœud à ajouter et le nœud à ajouter lui même. Cette fonction va simplement attacher le nouveau nœud à l'arbre en l'attachant au parent à gauche si la valeur contenue est inférieure à celle du parent sinon à droite.

La méthode `removeValue(T v_)`

La méthode `removeValue(T v_)` est une méthode initialement déclarée dans la classe abstraite `BinaryTree`, elle permet de supprimer un élément de l'arbre, c'est elle qui sera utilisée lors de l'appel à la méthode `remove` de l'interface `Collection<T>`. La suppression d'un nœud dans un arbre binaire de recherche doit faire en sorte de toujours respecter les règles avant et après suppression.
Par ailleurs la méthode `removeAllValue` est la méthode qui permet de supprimer une liste d'éléments de l'arbre. Elle sera appelée par `removeAll` de l'interface `Colection`.

```
Supprimer(v)
    // [ CAS 1 : recherche du noeud ]
    z = rechercher(v)
    si z n existe pas
        retourner faux

    // [ CAS 2 : choix du noeud a supprimer y ]
    si z a 0 ou 1 enfant
        y = z
    sinon
        // z a 2 enfants : prendre son successeur
        y = successeur(z)

    // [ CAS 3 : x = enfant eventuel de y ]
    x = (y.gauche existe) ? y.gauche : y.droite

    // rattacher x au pere de y
    si x existe
        x.pere = y.pere

    // [ CAS 4 : mise a jour du pere de y ]
    si y.pere n existe pas
        racine = x
    sinon si y est le fils gauche
        y.pere.gauche = x
    sinon
        y.pere.droite = x

    // [ CAS 5 : copier la valeur si y != z ]
    si y != z
        z.valeur = y.valeur

    retourner vrai
```

Listing 6 – Algorithme de la méthode de supression

L'algorithme de suppression commence par rechercher le noeud contenant `v`. S'il n existe pas, la suppression échoue. Si le nœud a au plus un enfant, c est lui que l'on supprime. Sinon, on choisit son successeur `y`. On rattache ensuite l'enfant éventuel `x` de `y` au père de `y`. Si `y` était la racine, on remplace la racine par `x`. Enfin, si `y` n'est pas le nœud initial `z`, on copie la valeur de `y` dans `z`.

La méthode `research(Object o)`

C'est la méthode qui fait partie de l'interface `Collection`, dans notre implémentation ici dans la classe `ABR`, la méthode de recherche prend en argument une valeur et renvoie le nœud qui contient cette valeur dans l'arbre, s'il existe sinon elle renvoie `null`. L'ordre et les déplacements dans les différents sous-arbre est possible grâce au comparateur `cmp` définie à la construction de l'arbre ou à la volée.

2.3 La classe `ARN<T>`

2.3.1 Présentation générale de la classe

La classe `ARN<T>` permet de représenter et de traiter les Arbres rouge-noir – ARN. Un arbre rouge-noir est une arbre binaire de recherche qui possède quelques propriétés supplémentaires :

1. Chaque nœud de l'arbre est soit rouge, soit noir.
2. La racine de l'arbre est noire.
3. Les feuilles sont noires.
4. Si un nœud est rouge alors ses deux fils sont noirs.
5. Pour chaque nœud le chemin reliant à des feuilles contiennent le même nombre de nœuds noirs.

Puisque un arbre rouge-noir est un ABR alors dans l'implémentation, la classe `ARN` étend la classe `ABR`. Et, puisque la classe `ABR` étend elle même la classe `BinaryTree` alors, en fait la classe `ARN` aura accès aux mêmes méthodes de `BinaryTree` qu'à accès `ABR`. Ainsi, on obtient un schéma d'héritage suivant :

$$\texttt{ARN<T>} \longmapsto \texttt{ABR<T>} \longmapsto \texttt{BinaryTree<T>}$$

C'est ici que l'implémentation est rendue plus complexe :

— Les nœuds d'un ARN possèdent une couleur, alors dans un ABR non.

— Un ARN est un ABR avec des propriétés d'équilibrage.

Cela implique que les opérations d'insertion et de suppression doivent non seulement préserver la structure et les invariants d'un ABR, mais également respecter les règles d'équilibrage propres aux arbres rouge-noir. Les méthodes d'ajout et de suppression héritées de `ABR` risquent donc de violer ces propriétés ; il est nécessaire de les redéfinir ou de les étendre afin de garantir que les contraintes de l'ARN restent satisfaites.

2.3.2 Les nœuds d'un ARN

Pour rappel on définit le nœud d'un arbre binaire par la valeur qu'il encapsule, son fils droit, son fils gauche et son père si ils existent. Dans un ARN, il est nécessaire de rajouter un nouveau paramètre qui permet de représenter la couleur du nœud.

`NodeRN`			
type	**attribut**	**visibilité**	**description**
`T`	`value_`	`private`	valeur encapsulée par le nœud
`boolean`	`isRed`	`public`	le nœud est-il rouge ?
`NodeRN`	`father_`	`private`	père du nœud
`NodeTN`	`left_`	`private`	sous-arbre gauche du nœud
`NodeRN`	`right_`	`private`	sous-arbre droit du nœud

Dans la classe `NodeRN` interne à la classe `ARN` on a simplement étendu la classe `Node` de `BinaryTree` en lui ajoutant un nouvel attribut `isRed` qui permet de déterminer la couleur du nœud :

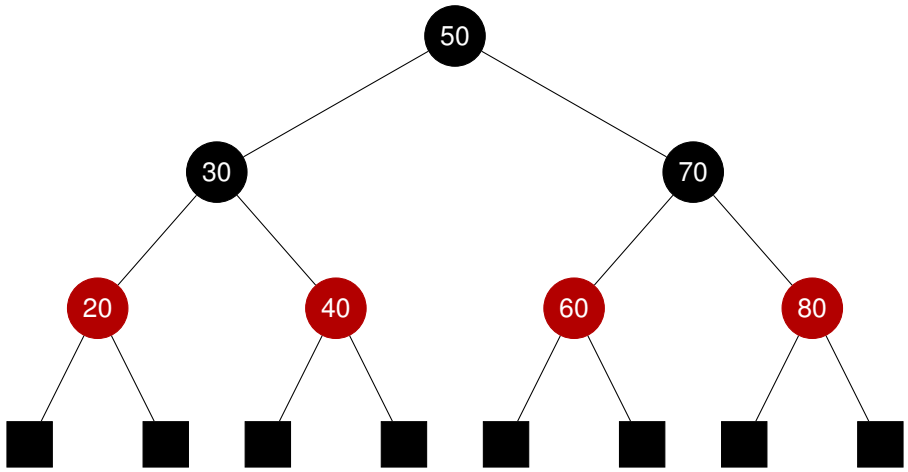
— Si `isRed == true` alors le nœud est rouge

— Si `isRed == false` alors le nœud est noir

Précisions :
La classe `NodeRN` réécrit aussi les getters et setters de la classe `Node` ainsi que toutes les méthodes `minimum`, `suivant` qui lui étaient associées afin qu'elles correspondent à la classe `NodeRN` et qu'elle soient utilisable par la classe `ARN`.

On considérera que tous les nœuds possèdent un fils gauche et un fils droit par défaut initialisés à `null`. Et puisque tous ses nœuds représenteront les feuilles de l'arbre et sont tous égaux alors, pour éviter de devoir stocker chaque nœud de manière unitaire, tous les nœuds feuilles pointeront vers un même nœud appelé **la sentinelle**.
Tous les nœuds feuilles qui pointeront vers la sentinelles seront représentés par le symbole suivant :

2.3.3 Affichage d'un arbre rouge-noir




```
// Affichage de l'arbre adapté à ARN

                              +-- ⊠
                  +-- 80 --|
                  |           +-- ⊠
         +-- 70 --|
         |        |           +-- ⊠
         |        +-- 60 --|
         |                    +-- ⊠
-- 50 --|
         |                    +-- ⊠
         |        +-- 40 --|
         |        |           +-- ⊠
         +-- 30 --|
                  |           +-- ⊠
                  +-- 20 --|
                              +-- ⊠
```

Listing 7 – Affichage de l'arbre rouge-noir avec sentinelles

2.3.4 Retour sur la classe `ARN`

Voici les listes des attributs associés à la classe `ARN`.

ARN			
type	**attribut**	**visibilité**	**description**
`NodeRN`	`root_`	`protected`	racine de l'arbre rouge-noir
`NodeRN`	`sentinel_`	`protected`	sentinelle de l'arbre
`String`	`RED`	`public`	code rouge terminal
`String`	`RESET`	`public`	code noir terminal

Le constructeur

Au départ l'arbre créé est initialement vide, alors la racine est représentée par la sentinelle. Et on définit la couleur de la sentinelle comme noire car les feuilles et la racine sont noires. Dans tous les cas la sentinelle est donc noire.

```
public ARN() {
    sentinel_.isRed = false; // La sentinelle est noire
    root_ = sentinel_;
}
```

Listing 8 – Constructeur vide de la classe ARN

Dans notre arbre, tous les nœuds `null` sont représentés par la sentinelle.

La méthode `add(T n_)`

On dispose de l'algorithme suivant pour ajouter un nœud.

```
Ajouter(v)
    // [ CAS 1 : arbre vide ]
    si racine = sentinel
        la racine devient rouge et encapsule v
        CorrigerAjout(racine)
        retourner vrai

    // [ CAS 2 : arbre non-vide]
    // recherche de la position du noeud
    courant = racine
    tant que courant != sentinel
        parent = courant
        si (v < courant.valeur) : courant = courant.gauche
        sinon si (v > courant.valeur) : courant = courant.droite
        sinon : retourner faux // Le noeud existe déjà
    Création et insertion du noeud n au parent

    // correction si propriétés violées
    CorrigerAjout(n)

    retourner vrai
```

Listing 9 – Méthode d'ajout (ARN)

La méthode `CorrigerAjout(Node)` permet de rééquilibrer, re-colorier l'arbre rouge-noir au cas où des propriétés aient été violée lors de l'insertion du nœud dans l'arbre. Dans notre implémentation la fonction est nommée `ajouterCorrection(Node z)`.

```
CorrigerAjout(z)
    tant que père(z) est rouge
        si père(z) est fils gauche de grand-père(z)
            y = grand-père(z).droit  // oncle de z
            si y.couleur == rouge
                père(z).couleur = noir
                y.couleur = noir
                grand-père(z).couleur = rouge
                z = grand-père(z)
            sinon
                si z == père(z).droit
                    z = père(z)
                    rotationGauche(z)
                père(z).couleur = noir
                grand-père(z).couleur = rouge
                rotationDroite(grand-père(z))
        sinon
            // Les cas miroirs
```

Listing 10 – Correction après ajout dans ARN

Cet algorithme, inspiré de celui présent dans le cours permet en fait de corriger les éventuelles déséquilibrages, erreurs de coloration lors de l'ajout d'un nœud dans l'arbre.

- Tant que le père de z est rouge, la propriété "un nœud rouge ne peut pas avoir de fils rouge" est violée.
- Si l'oncle de z est rouge (cas 1) : on recolore le père, l'oncle et le grand-père, puis on remonte z au grand-père.
- Si l'oncle est noir :
  - Si z est en ligne avec son père (cas 3), on recolore et on effectue une rotation pour rétablir les propriétés.
  - Sinon (cas 2, triangle), on effectue une rotation sur z pour transformer le triangle en ligne, puis rotation et recoloration.
- Enfin, la racine est recolorée en noir pour respecter la propriété que la racine doit toujours être noire.

La méthode `removeValue(T n_)`

```
Supprimer(n)
    // [ CAS 1 : arbre vide ]
    retourner faux car pas de noeud supprimable

    // [ CAS 2 : arbre non vide ]
    Si n a deux fils : on récupère le suivant dans y
    x est le fils unique ou la sentinelle de y
    on rattache le père de y à x

    si le père(y) = sentinelle : racine = x // y était racine
    sinon si y = oncle gauche : oncle-gauche(y)=x
    sinon : oncle-droit(y)=x

    supprimerCorrection(x)
    retourner vrai
```

Listing 11 – Méthode de suppression ARN

Cet algorithme `Supprimer(n)` gère la suppression d'un nœud dans un arbre rouge-noir (ARN). Dans le cas où l'arbre est vide, aucun nœud n'est supprimé et la fonction retourne `faux`. Si le nœud à supprimer a deux fils, on remplace le nœud par son successeur (le plus petit nœud du sous-arbre droit) et on rattache son fils unique ou la sentinelle au père du successeur. Ensuite, on ajuste la racine si nécessaire et on effectue la correction via `supprimerCorrection(x)` pour rétablir les propriétés de l'arbre rouge-noir. La fonction retourne `vrai` une fois la suppression effectuée.

```
supprimerCorrection(x)
    tant que x != racine et x est noir
        si x est fils gauche de son père
            w = frère de x (droit)
            si w est rouge
                recolorer w et père(x)
                rotationGauche(père(x))
                w = frère droit de x
            si w.gauche et w.droit noirs
                w = rouge
                x = père(x)
            sinon
                si w.droit noir
                    recolorer w et w.gauche
                    rotationDroite(w)
                    w = frère droit de x
                recolorer w et père(x)
                rotationGauche(père(x))
                x = racine
        sinon
            // Les cas miroir
    x = noir
```

Listing 12 – Correction de la suppression dans ARN

L'algorithme `supprimerCorrection(x)` restaure les propriétés d'un arbre rouge-noir après la suppression d'un nœud. Tant que le nœud `x` est noir et n'est pas la racine, on vérifie la couleur de son frère `w` et de ses enfants, et on applique les quatre cas classiques :

1. frère rouge alors recoloration et rotation pour transformer le problème.
2. frère noir avec deux enfants noirs alors recolorer le frère et remonter `x`.
3. frère noir avec un enfant rouge (triangle) alors rotation pour transformer en ligne.
4. frère noir avec enfant droit rouge (ligne) alors recoloration et rotation pour résoudre le déficit de noir.

Les cas miroirs sont traités symétriquement lorsque `x` est fils droit, et l'algorithme termine en recolorant `x` en noir.

La méthode `research(Object o)`

La méthode de recherche dans un ARN est exactement la même que celle implémentée dans la classe `ABR<T>` évoquée plus haut.

2.3.5 Les rotations

Les rotations gauche et droite sont des opérations fondamentales pour maintenir l'équilibre des arbres binaires de recherche auto-équilibrés, comme les arbres rouge-noir. Une rotation gauche (`leftRotate`) pivote un nœud vers la gauche : son fils droit devient le nouveau nœud racine du sous-arbre, le nœud pivoté devient fils gauche du nouveau nœud, et les sous-arbres concernés sont réattachés de manière à conserver l'ordre.

Symétriquement, une rotation droite (`rightRotate`) pivote un nœud vers la droite : son fils gauche devient la nouvelle racine du sous-arbre, et le nœud pivoté devient fils droit.

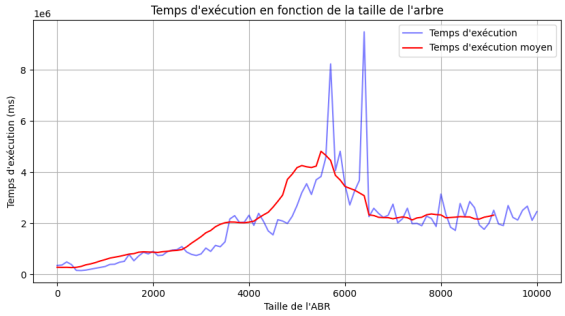
L'utilité de ces rotations est de rétablir les propriétés d'équilibre après une insertion ou une suppression, en permettant de corriger des déséquilibres ou des violations de couleur sans reconstruire l'arbre entier. Elles permettent également de préserver la structure de recherche binaire, car ne changent pas l'ordre des valeurs dans l'arbre.

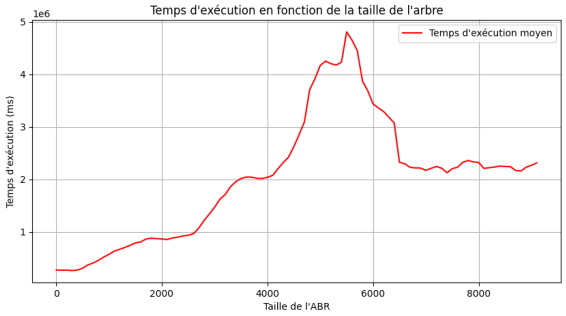
3 Comparaisons entre `ABR` et `ARN`

3.1 Comparaison à l'ajout

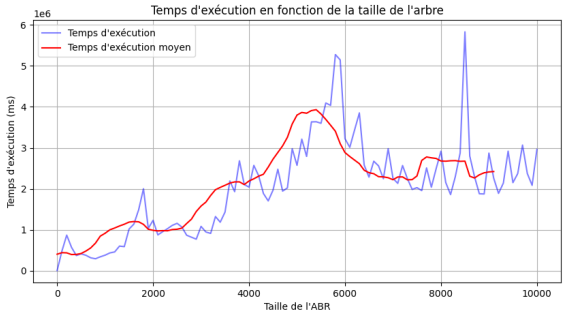
3.1.1 Cas basique : Ajout de valeurs aléatoires

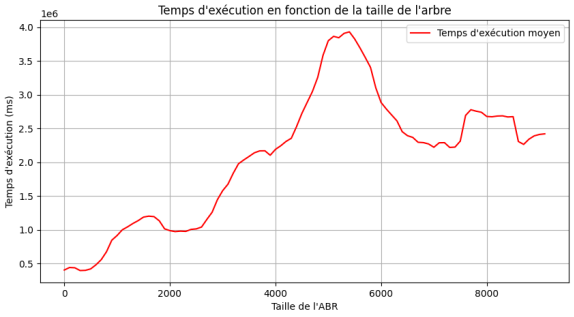
Courbes pour l'ajout – Arbre Binaire de Recherche ABR





Courbes pour l'ajout – Arbre Rouge-Noir ARN



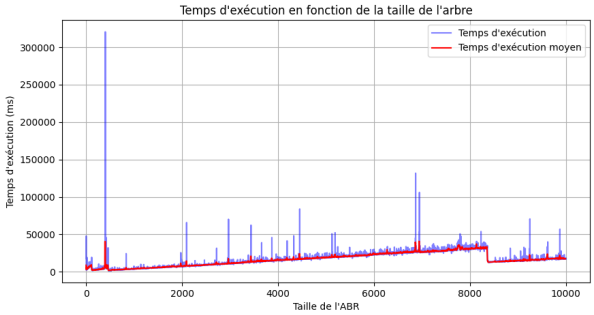


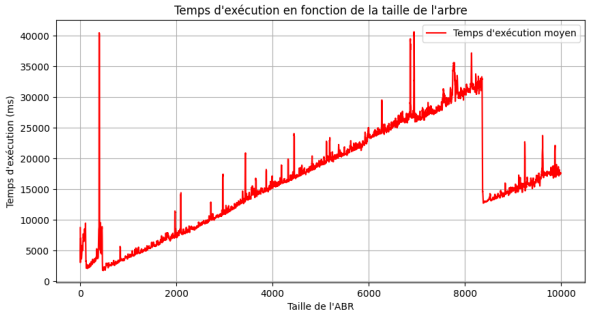
Les courbes[3] présentées ci-dessus illustrent les temps d'exécution pour l'ajout de valeurs aléatoires dans les deux types d'arbres étudiés. Les graphiques du haut concernent l'`Arbre Binaire de Recherche (ABR)`, tandis que ceux du bas concernent l'`Arbre Rouge-Noir (ARN)`. Dans chaque figure, la courbe bleue représente le temps d'exécution observé pour chaque insertion, et la courbe rouge correspond au temps d'exécution moyen ou lissé, ce qui permet de visualiser plus clairement la tendance générale sans être perturbé par les fluctuations ponctuelles. On observe que pour les ABR, la variabilité est plus importante, tandis que les ARN présentent des temps plus réguliers grâce à leur équilibrage automatique.

3. **Les courbes** – sont en meilleure qualité dans le dossier `python` du rendu.

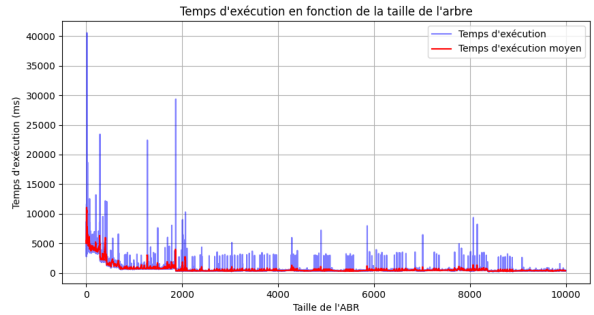
3.1.2 Cas défavorable : Ajout de valeurs dans l'ordre

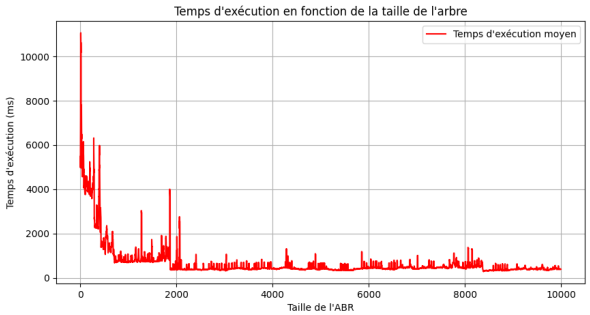
Courbes pour l'ajout empirique – Arbre Binaire de Recherche ABR





Courbes pour l'ajout empirique – Arbre Rouge-Noir ARN





Les graphiques ci-dessus illustrent les temps d'exécution pour l'ajout de valeurs dans l'ordre croissant, correspondant au cas défavorable pour les arbres. Les figures du haut concernent l'`Arbre Binaire de Recherche (ABR)`, tandis que celles du bas concernent l'`Arbre Rouge-Noir (ARN)`. On remarque que pour les ABR, le temps d'exécution augmente fortement avec la taille de l'arbre et présente de grandes variations, ce qui reflète le déséquilibre extrême de l'arbre dans ce scénario. En revanche, les ARN restent relativement stables grâce à leur mécanisme d'équilibrage automatique, comme le montre la courbe rouge lissée qui suit une tendance beaucoup plus régulière.

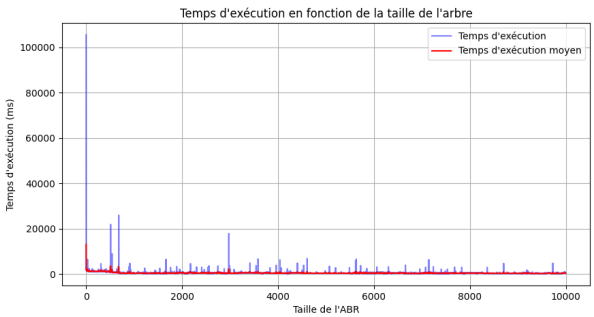
3.1.3 Conclusion pour l'ajout

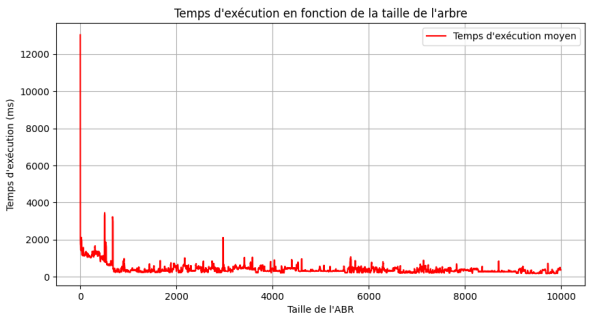
L'analyse des temps d'exécution montre l'impact de la structure interne des arbres sur les performances des opérations d'insertion. Pour des insertions aléatoires, les ABR présentent une variabilité plus importante que les ARN, la tendance générale reste croissante. En revanche, dans le cas défavorable d'insertion de valeurs triées, les ABR voient leur temps d'exécution croître de manière linéaire, illustrant le déséquilibre qui se crée. Les ARN, grâce à leur mécanisme d'équilibrage automatique, maintiennent des temps d'insertion réguliers et proches d'une complexité logarithmique.

Ces observations confirment que, bien que les ABR puissent être efficaces dans des cas généraux, les ARN sont eux efficaces face aux séquences d'insertion ordonnées.

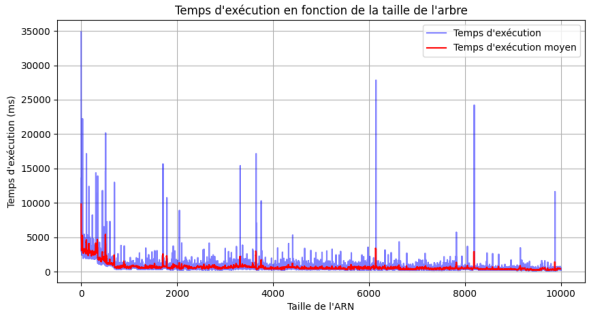
3.2 Comparaison à la suppression

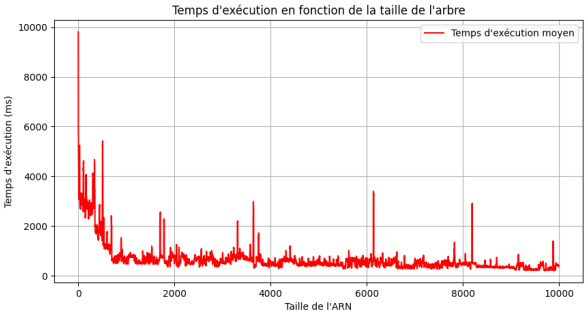
Courbes pour la suppression – Arbre Binaire de Recherche ABR





Courbes pour la suppression – Arbre Rouge-Noir ARN





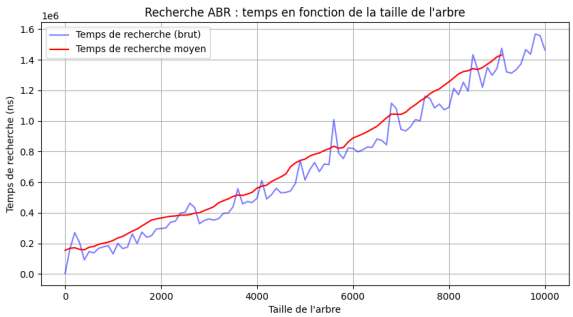
On part d'un arbre initial de 10000 éléments et on calcul le temps de suppression des n éléments selon la taille de l'arbre.

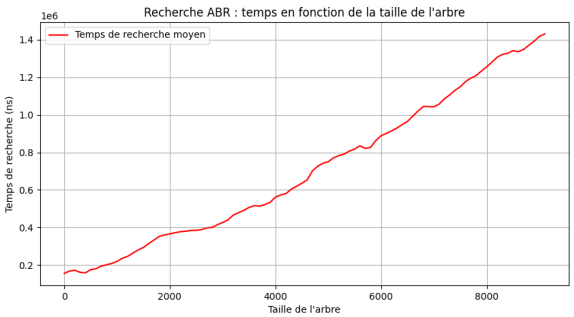
L'analyse des temps d'exécution de la suppression en fonction de la taille des arbres révèle des différences significatives entre les ABR et les ARN. Pour les arbres binaires de recherche, on observe un pic initial très prononcé atteignant environ 13 000 ms pour les petites tailles d'arbres (autour de 100 nœuds), suivi d'une décroissance rapide et d'une stabilisation autour de 500 ms pour les tailles supérieures à 2 000 nœuds. Cette volatilité initiale s'explique par le fait que les ABR ne soient pas équilibrés aux séquences d'insertions et de suppressions, pouvant conduire à des dégénérescences temporaires de la structure.

En revanche, les arbres rouge-noir présentent un comportement plus stable, avec un pic maximal d'environ 10 000 ms également en début de courbe, mais une décroissance plus progressive. On note toutefois des pics isolés aux positions 3 000, 6 000 et 8 000 nœuds, atteignant respectivement 2 000, 3 500 et 3 000 ms. Ces variations périodiques sont caractéristiques des opérations de rééquilibrage des ARN lors de suppressions complexes. Malgré ces pics, les ARN maintiennent une performance globalement plus prévisible que les ABR, avec une moyenne de temps d'exécution inférieure à 1 000 ms pour la majorité des tailles testées. Cette stabilité supérieure confirme l'intérêt des mécanismes d'auto-équilibrage pour garantir des performances logarithmiques même dans le pire cas.

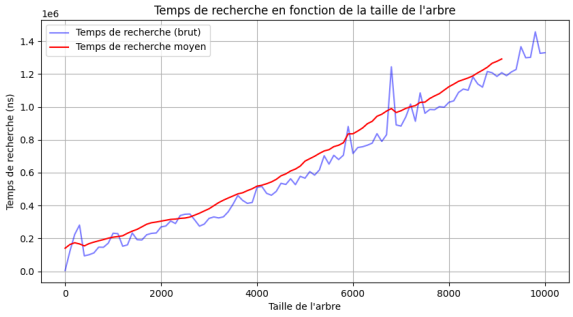
3.3 Comparaison à la recherche

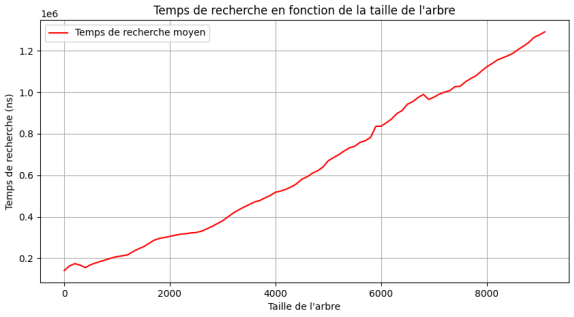
Courbes pour la recherche – Arbre Binaire de Recherche ABR





Courbes pour la recherche – Arbre Rouge-Noir ARN





Les courbes de recherche pour l'ABR et l'ARN présentent des comportements similaires. Dans les deux cas, le temps de recherche moyen (courbe lissée) suit une croissance quasi-linéaire en fonction de la taille de l'arbre, ce qui correspond au comportement attendu en $O(\log n)$ pour des arbres équilibrés. L'ARN présente toutefois des variations légèrement plus grandes, particulièrement visible sur le graphique de gauche avec des pics plus marqués autour de 6000-8000 nœuds. Cette différence s'explique par les mécanismes d'auto-équilibrage : l'ABR maintient un équilibre plus strict à chaque insertion, tandis que l'ARN tolère un léger déséquilibre temporaire. Les courbes moyennes restent néanmoins très proches, confirmant que les deux structures offrent des performances de recherche comparables pour des ensembles de données de taille moyenne.

4 Conclusion

Cette étude comparative met en évidence l'influence déterminante des mécanismes d'équilibrage sur les performances des structures arborescentes. Les `ABR` peuvent offrir des performances satisfaisantes dans des contextes favorables ou aléatoires, mais leur sensibilité à l'ordre des insertions et aux suppressions entraîne une forte variabilité des temps d'exécution. À l'inverse, les `ARN` garantissent des performances plus stables et prévisibles, aussi bien pour l'ajout, la suppression que la recherche, grâce à leur équilibrage automatique. Ainsi, bien que plus complexes à implémenter, les arbres rouge-noir apparaissent comme une solution plus robuste lorsque l'on souhaite assurer des performances logarithmiques dans le pire des cas.