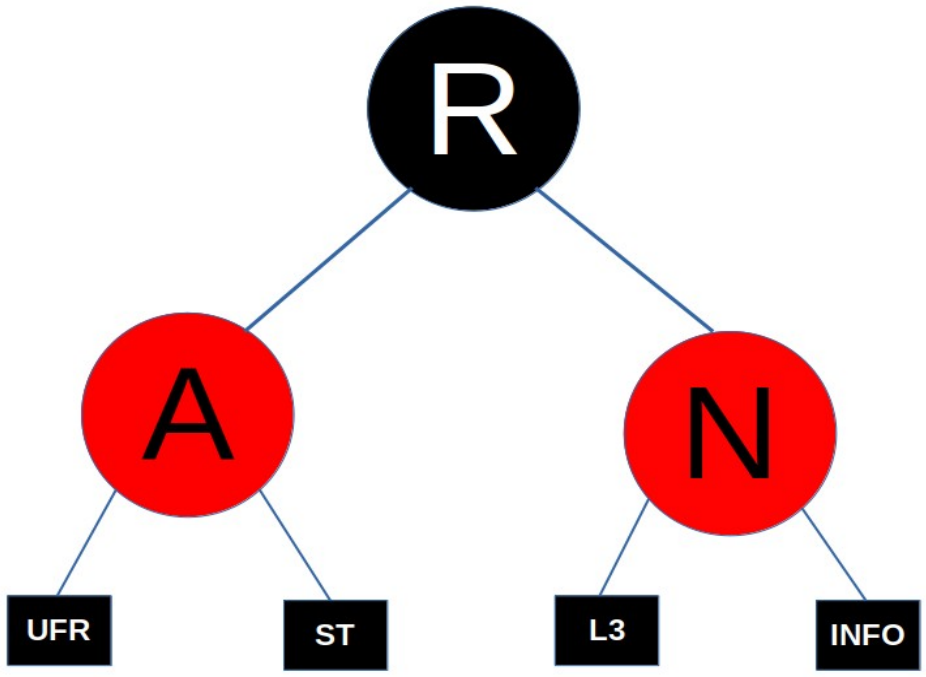
UNIVERSITÉ LE HAVRE NORMANDIE

Algorithmique Avancée

Rapport d'étude sur les Arbres Rouges-Noirs (ARN)



I. Introduction

Les arbres binaires de recherche équilibrés sont des structures de données très utiles dans le domaine de l'informatique, qui permettent la gestion rapide et efficace des données tout gardant une structure bien ordonnée. Ils sont utilisés pour implémenter une variété d'algorithmes, tels que la recherche, l'insertion et la suppression.

Les arbres rouge-noir, dont nous allons discuter dans ce rapport, sont une variante des arbres équilibrés traditionnels qui garantissent une hauteur logarithmique de l'arbre, tout en simplifiant les opérations d'insertion et de suppression.

Définition et propriétés :

Un arbre Rouge-Noir est un arbre binaire de recherche(ABR) qui chaque nœuds est soit rouge, soit noir et qui respecte les propriétés suivantes :

- La racine est **noire**.
- Les feuilles (ici sentinelle) sont noires.
- Si un nœud est rouge, alors ses deux fils sont noirs.
- Pour chaque nœud, tous les chemins le reliant à des feuilles contiennent le même nombre de nœuds noirs.

A partir de ces propriétés, on voit que l'arbre est « équilibré ». En effet, la hauteur de tout sous-arbre est au plus deux fois plus grande que la hauteur du sous-arbre le plus court.

Exemple :

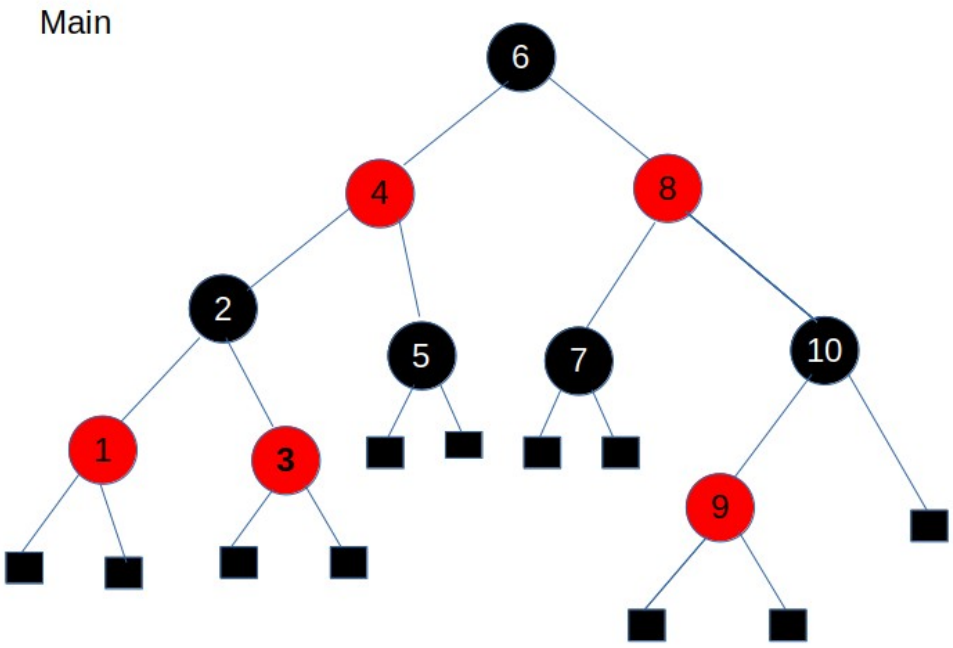


Figure 1 - Exemple d'arbre rouge noir

Les ARN vs les structures linéaires et les ABR

1. Structures linéaires :

Les structures linéaires(comme les tableaux, les listes ..) permettent un accès séquentiel aux éléments, tandis que les arbres Rouge-Noir(ici ARN) offrent un accès rapide aux éléments par recherche binaire. Les ARN sont plus efficaces que ces structures pour les opérations de recherche, d'insertion et de suppression, car elles ont une complexité en temps logarithmique, alors que les structures linéaires ont une complexité en temps linéaire.

2. ABR classiques :

Les ARN sont similaires aux ABR classiques, mais avec des règles supplémentaires pour maintenir un équilibre. La principale différence réside dans l'ajout de l'information de couleur(rouge ou noire) dans chaque nœud des ARN, ce qui permet de maintenir un équilibre lors des modifications de l'arbre.

Les arbres Rouge-Noir présentent également des avantages par rapport aux ABR classiques. Les arbres rouge-noir sont plus efficaces pour les opérations de recherche et d'insertion sur des arbres déséquilibrés.

En effet, les ABR classiques peuvent devenir déséquilibrés(cas défavorable) lorsque les clés sont ajoutées de la plus petite à la plus grande valeur, ce qui peut entraîner une dégradation des performances des opérations de recherche car pour chercher la la plus grande clé de l’arbre, on est obligé de parcourir toute la taille de l’arbre.

Exemple : Cas défavorable des ABR.

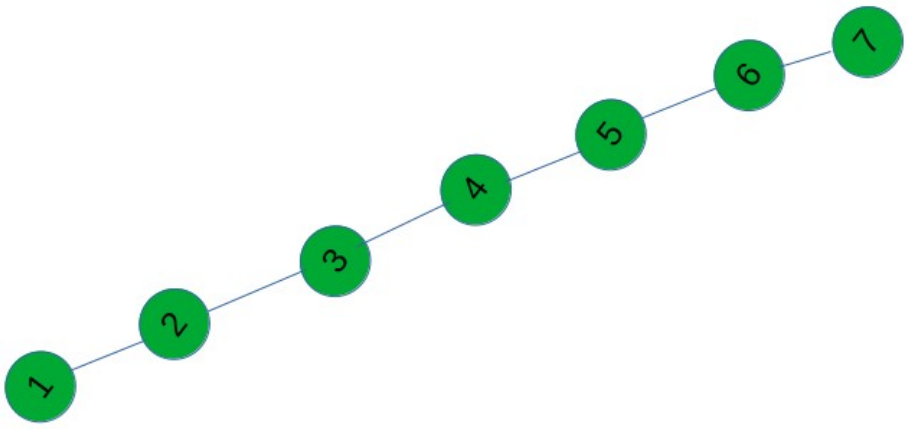


Figure 2 - Arbre binaire de recherche (Ajout de clés croissantes)

Les arbres rouge-noir, quant à eux, sont conçus pour rester équilibrés, ce qui garantit des performances constantes pour ces opérations(voir figure 1).

II. Implantation spécifique des ARN :

En plus des attributs **`cle`**, **`pere`**, **`gauche`** et **`droit`** des nœuds d'un ABR, on ajoute un attribut supplémentaire `couleur` qui peut prendre les valeurs `N` pour Noir ou `R` pour Rouge et un autre attribut `sentinelle`(nœud de clé nulle et de couleur N) dans la classe ABR.

L'implantation de la classe ARN se fait comme suit :

Utilisation de la sentinelle :

Une *sentinelle* est employée pour représenter les nœuds externes, simplifiant le code en évitant les comparaisons constantes de ***`null`***.
Par exemple la méthode « *suivant()* » de ABR :

```
    Noeud suivant() {
        if (droit != null) {
            return droit.minimum();
        }
        Noeud y = pere;
        Noeud x = this;
        while (y != null && x == y.droit) {
            x = y;
            y = y.pere;
        }
        return y;
    }
}
```

Figure 3: ABR avec la présence de la constante "null"

Une fois dans l'ARN, on obtient ceci :

```
    public Noeud suivant() {
        if (this.droit != sentinelle) {
            return this.droit.minimum();
        }
        Noeud y = pere;
        Noeud x = this;
        while (y != sentinelle && x == y.droit) {
            x = y;
            y = y.pere;
        }
        return y;
    }
}
```

Figure 4: ARN avec l'utilisation de la sentinelle

Méthode de Recherche :

La recherche d'une clé dans un ARN se déroule de la même façon que pour un ABR : on parcourt l'arbre en partant de la racine, on compare la clé recherchée à celle du nœud courant de l'arbre. Si ces clés sont égales, la recherche est terminée et on renvoie le nœud courant. Sinon, on choisit de descendre vers le nœud enfant gauche ou droit selon que la clé recherchée est inférieure ou supérieure. Si une feuille est atteinte, la valeur recherchée ne se trouve pas dans l'arbre, la sentinelle est renvoyée.

La couleur des nœuds de l'arbre n'a pas d'impact direct sur la recherche. Cependant, les arbres rouge-noir, à la différence des arbres binaires de recherche classiques, garantissent par construction un temps d'exécution de la recherche en O(log n), y compris dans le pire des cas.

Rotations :

En utilisant les méthodes d'ajout et de suppression des ABR, celles-ci ne préservent pas les toutes les propriétés des arbres équilibres(ici ARN). Si certaines propriétés sont violées, il faudra les réparer en effectuant des rotations et des ré-coloriages.

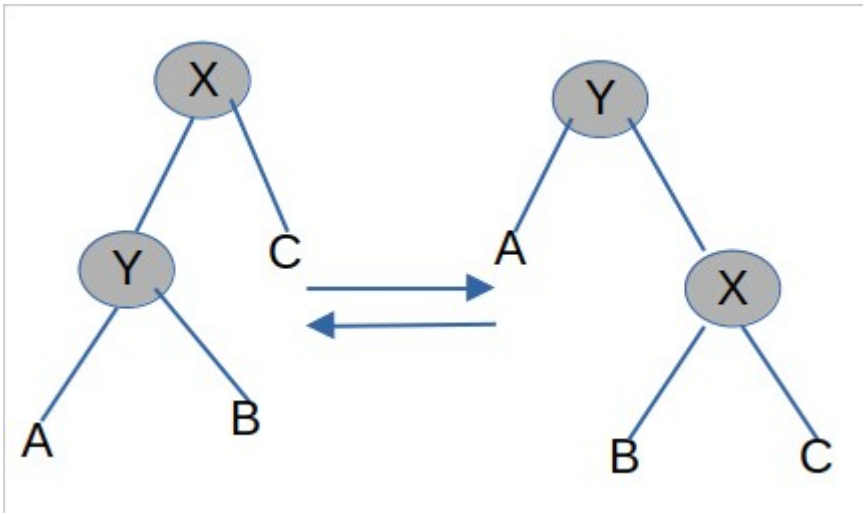


Figure 5: Rotation Gauche - Droite

Les rotations dans un arbre binaire sont des opérations locales permettant d'échanger un nœud avec l'un de ses fils pour réorganiser la structure. La rotation droite positionne un nœud comme fils droit de son ancien fils gauche, tandis que la rotation gauche le place comme fils gauche de son ancien fils droit. Pour réaliser cela, on a implémenter deux méthodes qui effectuent les rotations :
`rotationDroite(Noeud n)` et `rotationGauche(Noeud n).`
L'opération de rotation est réalisable en temps O(1).

Insertion :

Le principe d'insertion d'un élément dans un ABR se fait comme une insertion dans un ABR classique en créant un nœud de couleur rouge.

Cependant, cette opération peut violer la propriété des ARN si le père du nouveau nœud est également rouge. Pour corriger cela, des rotations gauche et droite sont utilisées pour réorganiser les nœuds et ajuster leurs couleurs, éliminant ainsi les conflits et maintenant l'équilibre de l'arbre. Ces corrections sont effectuées par la méthode `ajouterCorrection(Noeud z).`

Il y a plusieurs cas possibles pour rétablir les propriétés de l'arbre, à partir du nœud inséré.

Cas 1 (et 1') : Le père et l'oncle du nœud inséré sont rouges.

On change la couleur du père et de l'oncle en noir, et on colore le grand-père en rouge. Ensuite, on répète le processus de correction à partir du grand-père.

Cas 2 (et 2') : Le père est rouge, mais l'oncle est noir, et le nœud inséré est le fils droit du père (ou le fils gauche).

On effectue une rotation gauche (ou droite) autour du père, transformant le cas en Cas 3

Cas 3 (et 3') : Le père est rouge, l'oncle est noir, et le nœud inséré est le fils gauche du père (ou le fils droit).

On change la couleur du père en noir, du grand-père en rouge, puis on effectue une rotation droite (ou gauche) autour du grand-père.

En terme de complexité, la méthode d'insertion est en O(h) = O(log n).

Suppression :

Tout Comme pour l'insertion, la suppression d'une valeur dans un ARN commence par supprimer un nœud comme dans un arbre binaire de recherche. Cependant, on doit apporter quelques modifications à l'algorithme de suppression d'un ABR classique.

Si le nœud contenant la clé à supprimer était de couleur rouge, aucune des propriété des ARN n'est violée. Par contre, si celles-ci sont violées si ce nœud est de couleur noire. Il faut donc apporter une correction avec la méthode `supprimerCorrection(Noeud z)` qui restaure les propriété des ARN. Pour cette correction on va procéder cas par cas.

Cas 1 (et 1'): Le frère de nœud à détacher est rouge. Pour rétablir l'équilibre, on change la couleur du frère en noir, la couleur du parent en rouge, et on effectue une rotation gauche(ou droite pour le cas miroir).

Cas 2 (et 2'): Le frère est noir, et ses deux fils sont noirs. Pour rétablir l'équilibre, on colore le frère en rouge. Si le père du nœud à supprimer est rouge, on le colore en noir. Sinon, on déplace le problème vers le père (en recolorant et en vérifiant les cas de nouveau).

Cas 3 (et 3’): Le frère est noir, son fils gauche est rouge, et son fils droit est noir. On change la couleur du fils gauche en noir, la couleur du frère en rouge, et on effectue une rotation (droite ou gauche).

Cas 4 (et 4’): Le frère est noir et son fils droit est rouge. Pour rétablir l'équilibre, on change la couleur du frère en la couleur du parent, la couleur du parent en noir, la couleur du fils droit en noir, et on effectue une rotation gauche(ou droite pour 4’).

En terme de complexité, la suppression est en O(h) = O(log n).

Conclusion

Toutes les opérations sur les ARN sont en O(h), c’est à dire O(log n). Ce qui justifie l’utilisation des arbres rouge-noir par rapport aux arbres binaires classiques.

III. Étude expérimentale des performances

Pour réaliser cette étude expérimentale, on va effectuer dans un premier temps une série de tests unitaires qui permettent de vérifier le bon fonctionnement des méthodes implantées, puis une dans un second temps une étude qui vise à comparer les performances des ARN avec les ABR dans deux scénarios distincts.

1. Cas moyen :

Insertion de clés aléatoires

Dans cas, on effectue une série d'insertion de clés de façon aléatoire de 0 à 100 000 éléments dans les ARN et ABR puis on calcule le temps d'ajout de chaque arbre.

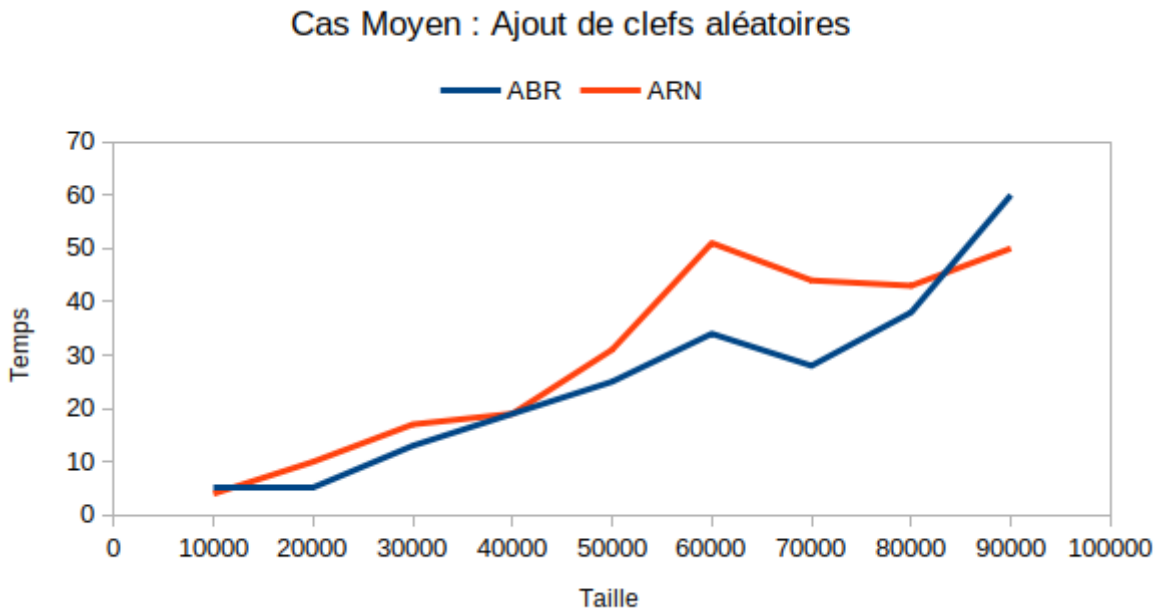


Figure 6: Evolution du temps en fonction de la taille des ABR et ARN lors d'ajout de clés aléatoires

On peut voir sur ce graphe que lors de l'ajout de clés aléatoires, les ARN démontrent une croissance plus rapide que les ABR jusqu'à une taille particulière, après quoi les deux structures commencent à se chevaucher en termes de temps d'exécution.

Recherche de clés aléatoires

Après un ajout aléatoire de clés, on se met à effectuer une recherche dessus puis on mesure le temps de recherche des arbres rouge-noir et arbres binaires de recherche classiques.

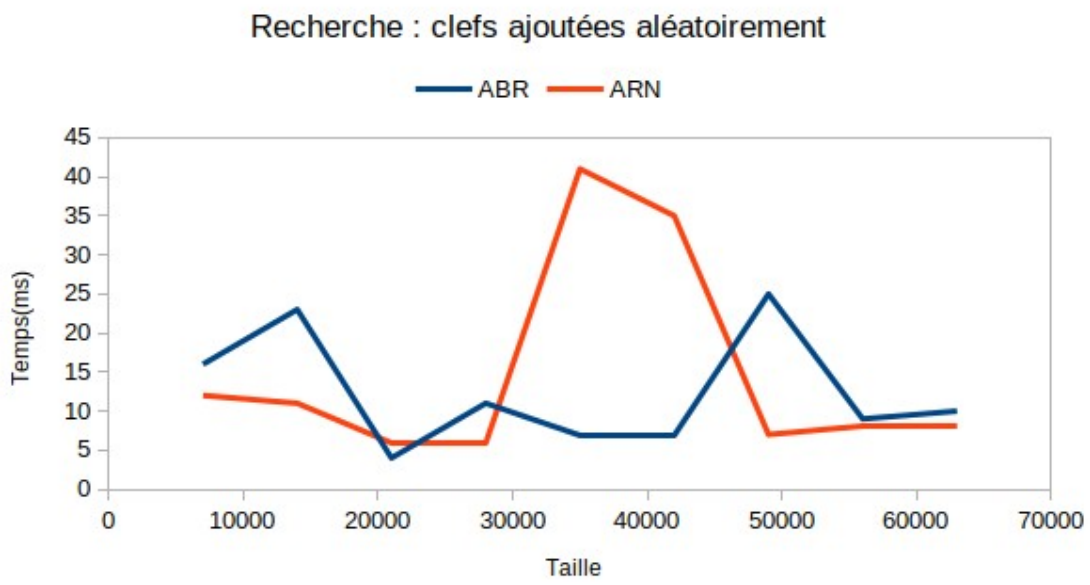


Figure 7: Evolution du temps en fonction de la taille des ABR et ARN lors de la Recherche de clés aléatoires

L'évolution des temps de recherche pour les clés dans les ARN et ABR montre des tendances intéressantes. On observe sur ce graphe qu'à une certaine taille, les temps de recherche des ABR sont compétitifs, mais à mesure que la taille de l'arbre augmente, les ARN surpassent les ABR en maintenant des temps de recherche plus constants et généralement inférieurs.

2. Cas défavorable :

Insertion de clés croissantes :

Dans cette étude, on s'amuse à ajouter des clés croissantes dans les deux structures d'arbres pour évaluer leurs comportements.

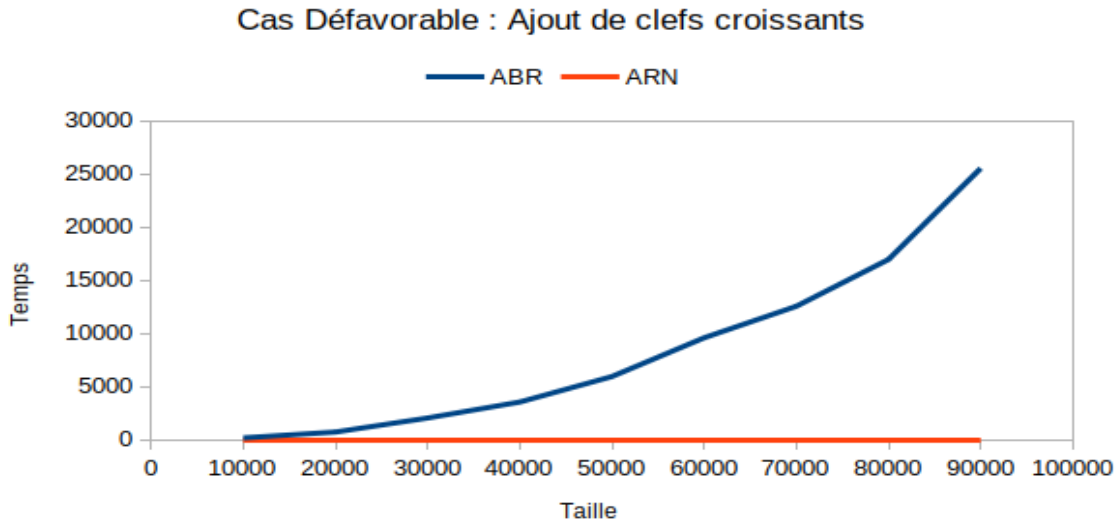


Figure 8: Evolution du temps en fonction de la taille des ABR et ARN lors d'ajout de clés croissantes

L’ajout de clés croissantes dans un ABR entraîne un déséquilibrage complète de l’arbre, quand aux ARN qui sont plus robustes et sont capables de maintenir une hauteur logarithmique de l'arbre, même en cas d'insertion de clés croissantes. C’est pourquoi dans ce graphe ci dessus on observe que la courbe des ABR s’éloigne de plus en plus quand la taille augmente. Ainsi les ABR prennent plus de temps.

Recherche de clés croissantes :

Dans ce scénario, le temps de recherche est évalué pour les clés allant de 10 000 à 2 * N avec N=70000, en considérant que les N premières clés sont présentes dans l'arbre, tandis que les N clés suivantes ne le sont pas.

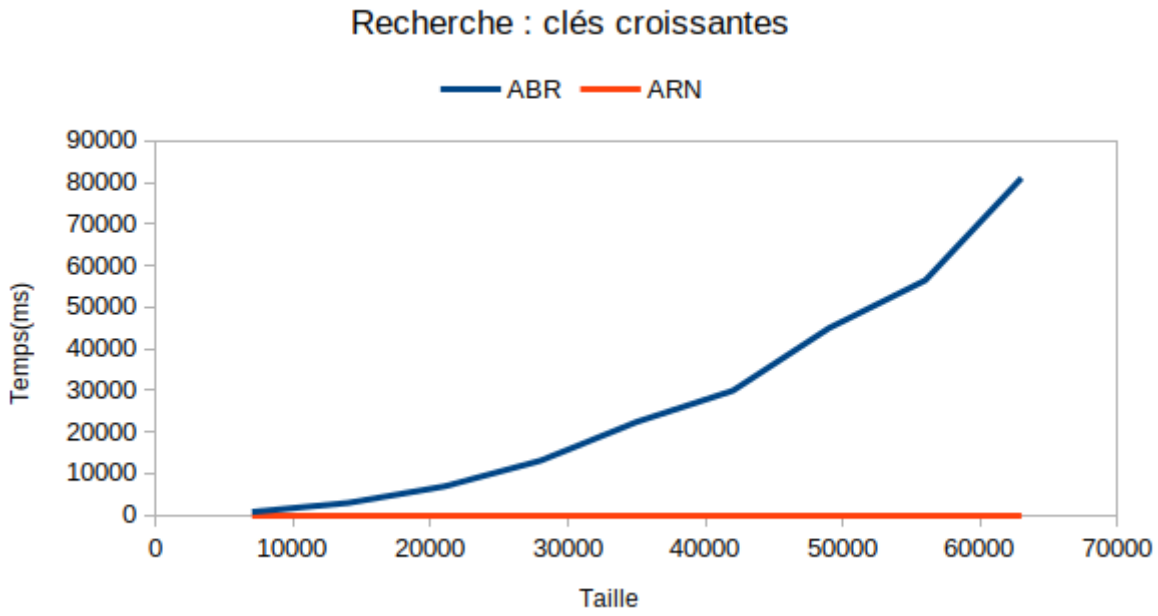


Figure 9: Evolution du temps en fonction de la taille des ABR et ARN lors de la Recherche de clés croissantes

La courbe de recherche montre que les ARN sont également plus rapides que les ABR pour rechercher une valeur dans un arbre de taille croissante. En effet, les ARN sont en moyenne une fois plus rapides que les ABR pour la recherche de valeurs dans un arbre de taille 70 000.

En conclusion, l'étude expérimentale révèle que les arbres Rouges-Noirs (ARN) surpassent les arbres binaires de recherche classiques (ABR) en termes de performances, offrant des temps d'insertion et de recherche plus constants et généralement inférieurs, même dans des cas défavorables comme l'ajout de clés croissantes. Ces résultats justifient l'utilisation préférentielle des ARN dans diverses applications nécessitant des opérations sur des structures de données équilibrées.

Bibliographie :

- ***Cours sur les Arbres Rouges-Noirs*** - S. Balev, Université Le Havre https://www-apps.univ-lehavre.fr/forge/balevs/algorithmique-avanc-e/-/blob/master/cours/black-red-trees1.md
- ***Arbre bicolore***- Wikipédia https://fr.wikipedia.org/wiki/Arbre_bicolore
- ***Arbres Rouges-Noirs en Java*** - MIASHS https://miashs-www.u-ga.fr/prevert/Prog/Java/CoursJava/arbresRougeNoir.html
- ***INAL 5 - Structures de données avancées*** - LIP6 https://www-npa.lip6.fr/~blin/Enseignement_files/INAL/INAL_5.pdf
- ***Algorithmique - Les Arbres*** - MCours.net

  http://www.mcours.net/cours/pdf/yass1/yass1clic1156.pdf