Compte rendu ALGO

L3 informatique

Nom : DOUADI

Prenom : Anis

Les Arbres Binaires de Recherche (ABR)

- ## Définition :

  Les Arbres Binaires de Recherche (ABR) sont des structures de données arborescentes utilisées en informatique pour organiser et stocker des données de manière efficace, facilitant la recherche, l'insertion et la suppression d'éléments. La particularité d'un ABR réside dans le fait que chaque nœud de l'arbre a au plus deux enfants, et pour chaque nœud, tous les éléments situés dans le sous-arbre gauche sont inférieurs ou égaux à cet élément, tandis que tous les éléments situés dans le sous-arbre droit sont supérieurs.

- ## Propriétés :

  ➔ Pour chaque nœud de l'arbre, tous les éléments situés dans son sous-arbre gauche sont inférieurs ou égaux à cet élément, et tous les éléments situés dans son sous-arbre droit sont supérieurs.

  ➔ Les ABR sont des structures arborescentes où chaque nœud a au plus deux enfants : un à gauche et un à droite. Cette structure hiérarchique facilite la recherche et la manipulation des données.

  ➔ Chaque sous-arbre d'un ABR est également un ABR. Cette propriété récursive facilite l'application d'opérations telles que la recherche, l'insertion et la suppression de manière récursive. Ces opérations de recherche, d'insertion et de suppression ont une complexité moyenne de O(log n), où n est le nombre d'éléments dans l'arbre. Cependant, dans le pire des cas, la complexité peut devenir linéaire (O(n)).

  ➔ Les ABR peuvent être parcourus de différentes manières, telles que le parcours préfixe (infixe), le parcours infixé (inorder), et le parcours postfixe (postorder), fournissant différentes perspectives sur les données stockées dans l'arbre.

- Implantation :

Notre classe ABR est une représentation d'un arbre binaire dans lequel chaque nœud a au plus deux enfants, un à gauche et un à droite. La structure de données est paramétrée par le type des clés stockées dans l'arbre, et elle implémente l'interface Collection.

On utilise une classe interne Noeud pour représenter les éléments individuels de l'arbre. Chaque nœud contient une clé, des références vers ses enfants gauche et droit, et une référence vers son parent.

La classe Noeud comprend des méthodes telles que minimum() pour trouver le nœud contenant la clé minimale dans un sous-arbre, et suivant() pour obtenir le successeur d'un nœud dans l'ordre des clés.

Notre classe contient des variables de classe pour suivre la racine de l'arbre, la taille totale de l'arbre et un comparateur pour définir l'ordre des éléments. On utilise trois constructeurs pour initialiser l'arbre de différentes manières : vide, avec un comparateur spécifié, ou en copiant une collection existante.

La méthode add(E e) est responsable de l'ajout d'éléments à l'arbre. On utilise une façon itérative en parcourant l'arbre à partir de la racine jusqu'à trouver la position appropriée pour l'insertion en fonction du comparateur. Les éléments sont ajoutés tout en maintenant la propriété de recherche binaire.

On utilise la méthode rechercher(Object o) pour rechercher une clé dans l'arbre. Elle parcourt l'arbre de manière itérative en comparant les clés jusqu'à trouver la clé recherchée ou atteindre une feuille.

La méthode supprimer(Noeud z) est responsable de la suppression d'un nœud spécifique de l'arbre tout en préservant la propriété de recherche binaire. Elle gère les cas où le nœud à supprimer à zéro, un, ou deux enfants.

La classe implémente également l'interface Collection, fournissant des méthodes telles que iterator() pour itérer à travers les éléments de l'arbre, size() pour obtenir la taille de l'arbre, et toString() pour générer une représentation visuelle de l'arbre sous forme de chaîne de caractères.

Les arbres rouge-noir (ARN)

- Définition :

  Les Arbres Rouges-Noirs (ARN) sont une structure de données arborescente qui étend les propriétés des Arbres Binaires de Recherche (ABR) classiques pour garantir un équilibre relatif de l'arbre. L'idée clé derrière les Arbres Rouges-Noirs est d'assigner des couleurs, soit rouge soit noir, aux liens entre les nœuds de l'arbre. Cette coloration est utilisée pour imposer des règles spécifiques qui maintiennent un équilibre optimal, assurant ainsi des performances de recherche, d'insertion et de suppression efficaces, même dans des situations défavorables.

- Propriétés :

  ➔ Couleur des Nœuds :Chaque nœud dans un Arbre Rouge-Noir est attribué l'une des deux couleurs possibles : rouge ou noir. Cette coloration est une caractéristique fondamentale qui permet d'imposer des contraintes spécifiques tout en maintenant la structure ordonnée de l'arbre.

  ➔ Les ARN conservent la propriété d'ordre des ABR classiques. Pour chaque nœud, tous les éléments dans le sous-arbre gauche sont inférieurs ou égaux à ce nœud, et tous les éléments dans le sous-arbre droit sont supérieurs.

  ➔ Les liens rouges et noirs suivent des règles spécifiques. Aucun chemin simple de la racine à une feuille ne peut avoir deux liens rouges consécutifs, la racine de l'arbre est toujours noire. Les feuilles (nœuds nuls ou sentinelles) sont également considérées comme noires. Ces règles simplifient l'application des propriétés de couleur dans l'arbre.

  ➔ Pour chaque nœud non feuille, tous les chemins simples de ce nœud aux feuilles descendants ont le même nombre de nœuds noirs

- Implantation :

Notre classe ARN est une représentation d'une structure d'arbre binaire de recherche (ABR) avec l'utilisation spécifique des arbres rouges et noirs Le choix d'utiliser des arbres rouges et noirs découle de leur capacité à maintenir un arbre relativement équilibré tout en évitant les pires cas de performance.

Dans la classe ARN on utilise une approche générique avec un type de données paramétré <E>. Chaque élément est encapsulé dans un objet Noeud qui possède des attributs tels que la clé (cle), des références vers les nœuds fils gauche (gauche) et droit (droit), une référence vers le nœud parent (pere), et une indication de la couleur du nœud (couleur). La couleur rouge ou noire des nœuds est utilisée pour maintenir l'équilibre de l'arbre.

On crée plusieurs constructeurs pour créer des instances d'arbres rouges et noirs. Ces derniers permettent la création d'arbres vides, d'arbres avec un comparateur personnalisé pour définir l'ordre des éléments, ou encore la création d'un arbre à partir d'une collection existante.

On utilise la méthode add pour l'insertion d'un nouvel élément dans l'arbre tout en maintenant l'équilibre structurel requis par les propriétés des arbres rouges et noirs. On utilise la couleur rouge pour indiquer le nœud nouvellement ajouté, et on appelle la méthode insCorrection pour garantir le respect des règles des arbres rouges et noirs après chaque insertion. Cette approche garantit que l'arbre demeure équilibré même après des opérations d'insertion fréquentes.

La suppression d'un élément de l'arbre est gérée par la méthode remove, qui, en plus de supprimer le nœud, s'assure que l'arbre reste équilibré en appelant la méthode suppCorrection. Cette méthode corrige les violations potentielles des propriétés des arbres rouges et noirs tout en maintenant l'équilibre global de la structure.

La méthode toString est une représentation de l'arbre sous forme de chaîne de caractères. On représente la structure et la couleur des nœuds, offrant ainsi une visualisation claire de la disposition des éléments dans l'arbre.

On propose une classe interne privée ARNIterator qui implémente l'interface Iterator. Cela permet d'itérer à travers les éléments de l'arbre dans l'ordre, offrant ainsi une manière pratique de parcourir les éléments de l'arbre.

Résultats de l'étude expérimentale.

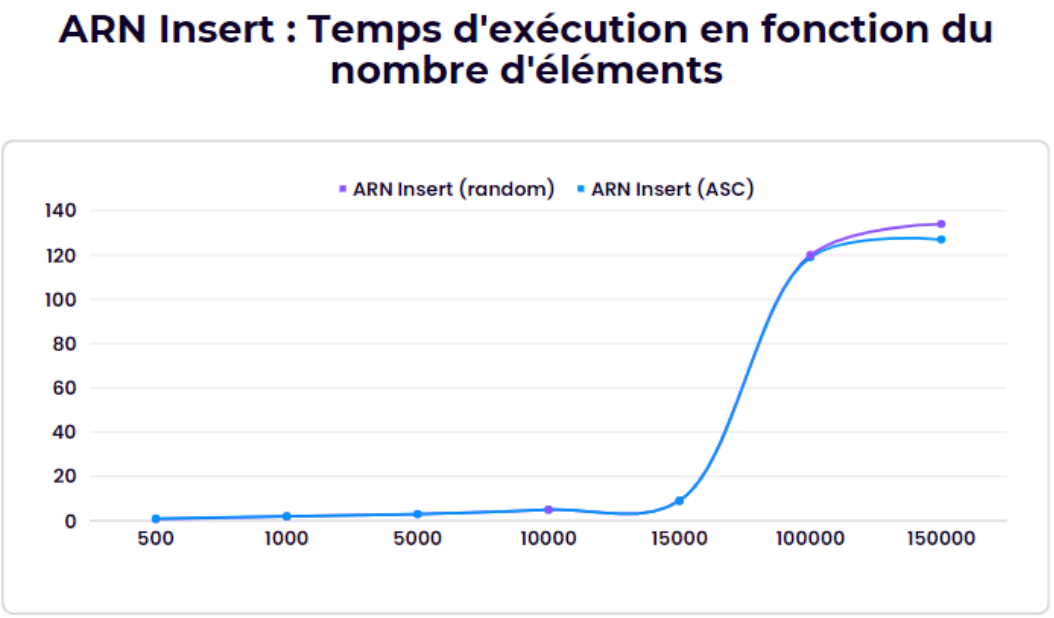
- **L’insertion:**

  - **Résultats de l’insertion sur l’ARN:**

N/t(ms)	ARN Insert (random)	ARN Insert (ASC)
500	0.661	1
1000	2	2
5000	3	4
10000	6	6
15000	9	9
100000	118	123
150000	133	125

Le tableau représente le temps d’insertion sur un ARN en fonction du nombre d'éléments.

Représentation des résultats obtenus sur la courbe suivante :



Analyse de la courbe : Le graph résultant de l'exécution du programme ARN révèle des conclusions intéressantes. Les essais ont porté sur l'insertion aléatoire et l'ajout d'éléments déjà triés par ordre croissant, avec une variation de la taille de l'arbre. Les courbes obtenues pour ces deux méthodes d'ajout présentent des similitudes, indiquant que le temps d'exécution reste stable, quel que soit le nombre d'éléments. En somme, le programme ARN maintient une performance constante, démontrant ainsi son efficacité peu importe la distribution des données dans l'arbre.

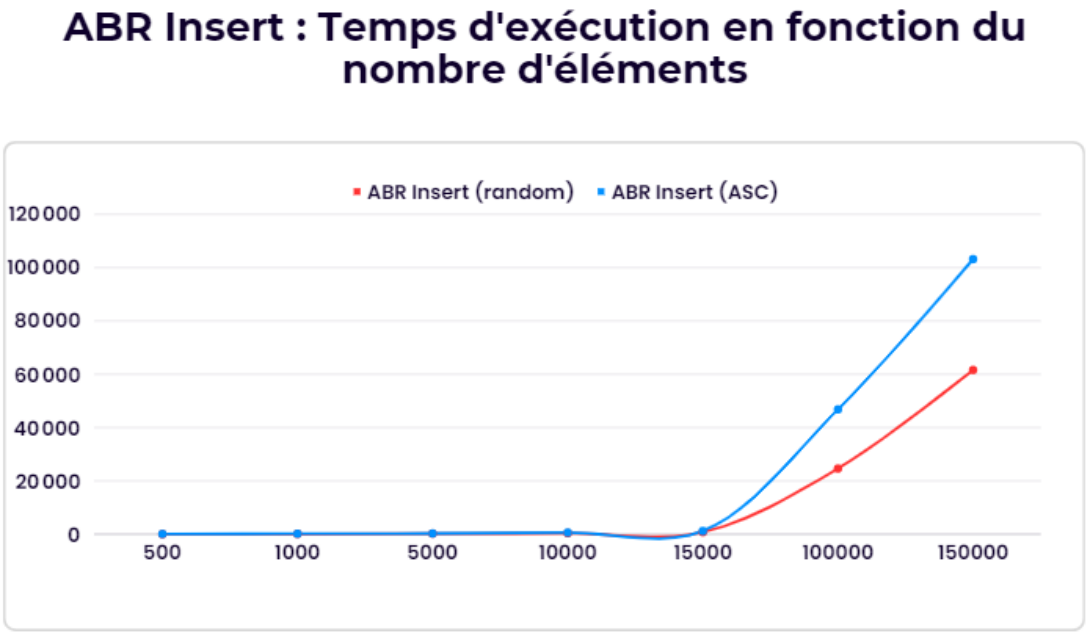
❖ Résultats de l'insertion sur l'ARB:

N/t(ms)	ABR Insert (random)	ABR Insert (ASC)
500	3	87
1000	98	176
5000	226	287
10000	372	617

15000	729	1191
100000	24697	46813
150000	61523	102715

Le tableau représente le temps d'insertion sur un ABR en fonction du nombre d'éléments.

Représentation des résultats obtenus sur la courbe suivante :



Analyse de la courbe :les résultats de l'exécution de l'insertion aléatoire et de l'insertion d'éléments déjà triés de manière croissante révèlent une observation significative. L'insertion d'éléments de manière aléatoire se révèle plus rapide que celle des éléments triés de façon croissante.

- ***Pire cas:***

Dans le pire des cas, l'insertion d'éléments croissants, par exemple, {10, 15, 20, 25, 30}, crée un arbre où tous les éléments sont situés du côté droit du nœud. Cette configuration génère une complexité de O(N) lors de la recherche du plus grand élément, nécessitant un parcours intégral de l'arbre.

- ***meilleur cas :***

Dans le meilleur cas, l'insertion d'éléments aléatoires (courbes bleues) s'avère plus rapide que l'insertion d'éléments croissants. Cela s'explique par le fait qu'aucun parcours intégral de l'arbre n'est requis pour insérer un élément, conduisant à une complexité de O(log(n)).

- ***Comparaison entre ARN et ABR :***

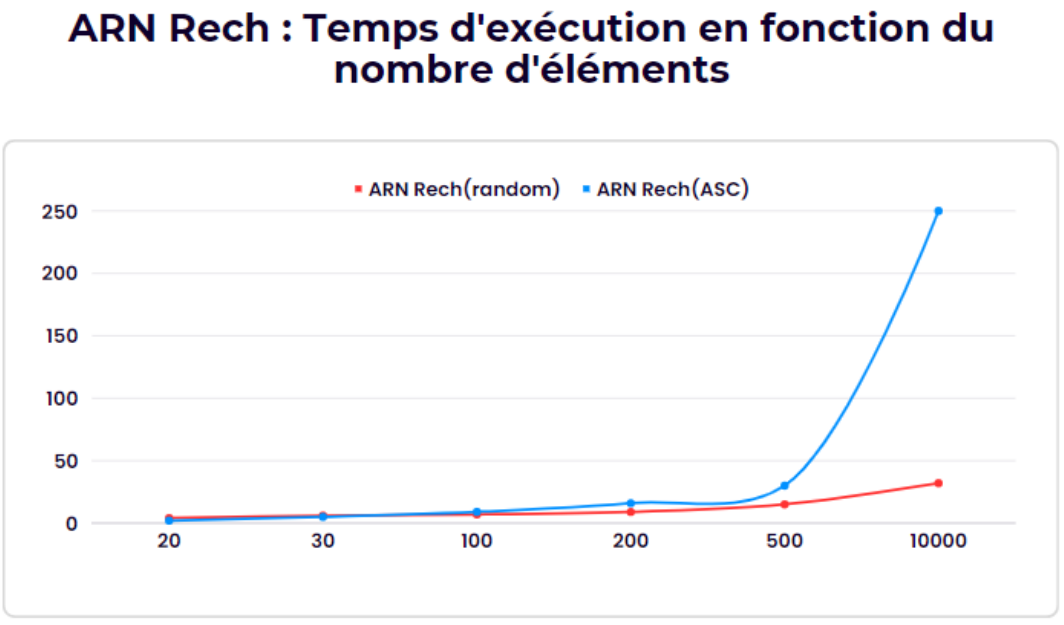
Les graphiques indiquent clairement que l'ABR affiche une lenteur significative lors de l'insertion d'éléments, que ce soit de manière aléatoire ou croissante. Par conséquent, la complexité de l'ABR est évaluée à O(n), tandis que l'ARN maintient une complexité plus avantageuse de O(log(n)). Ces observations soulignent l'efficacité supérieure de l'ARN dans le contexte des opérations d'insertion par rapport à l'ABR, ce qui peut être crucial dans des scénarios de gestion de données massives.

- **La Recherche:**
  - **Résultats de la recherche sur l'ARN:**

N/t(ms)	ARN Rech(random)	ARN Rech (ASC)
20	3	3
30	6	7
100	8	10
200	11	17
500	17	33
10000	30	260

Le tableau représente le temps de recherche sur un ARN en fonction du nombre d'éléments.

Représentation des résultats obtenus sur la courbe suivante :



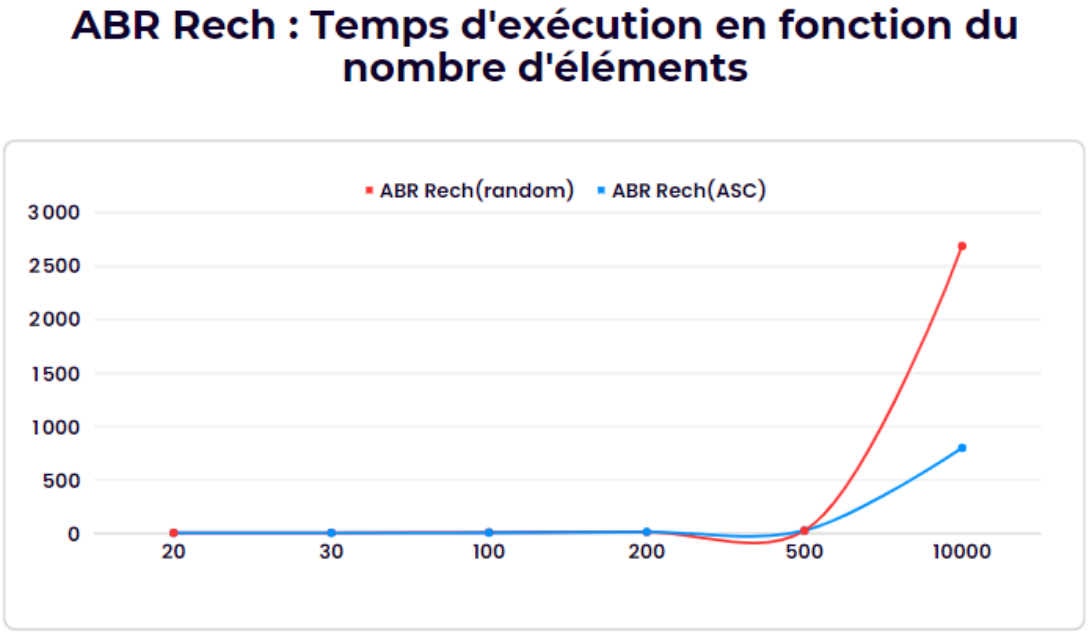
Analyse de la courbe : En analysant le graphe issu de l'exécution de la recherche de nombres en fonction du nombre d'éléments de l'arbre, suite à des insertions réalisées de manière aléatoire ou ordonnée de façon croissante, une tendance significative émerge. Il est notable que la recherche dans un arbre où les éléments sont insérés de manière croissante est plus lente par rapport à une insertion aléatoire. Cette observation suggère une complexité de O(log(n)), où n représente le nombre d'éléments à insérer. Ainsi, la séquence d'insertion influe directement sur l'efficacité des opérations de recherche dans l'arbre.

- **Résultats de la recherche sur l'ABR:**

N/t(ms)	ABR Rech(random)	ABR Rech (ASC)
20	5	5
30	6	5
100	10	7
200	13	15
500	26	27
10000	2690	800

Le tableau représente le temps de recherche sur un ABR en fonction du nombre d'éléments.

Représentation des résultats obtenus sur la courbe suivante :



❖ Pire cas :

le pire des cas lors de la recherche du dernier nœud, nécessitant une traversée complète de l'arbre. Dans de telles situations, la complexité de l'opération de recherche atteint O(N), indiquant une inefficacité accrue.

❖ meilleur cas :

Dans le meilleur des cas, où le nœud est le premier dans l'arbre, la complexité atteint son minimum, notée O(1).En revanche, le scénario optimal le plus courant se produit lors d'insertions aléatoires dans l'arbre. Dans ces cas, le nombre de comparaisons nécessaire pour localiser un nœud est généralement limité à la hauteur de l'arbre, soit log(N). Cette approche offre une efficacité notable dans les opérations de recherche, avec une complexité de O(log(N)).

❖ comparaison entre ARN et ABR :

Les courbes du graphique démontrent que la recherche dans l'Arbre Rouge-Noir (ARN) est plus efficace que dans l'Arbre Binaire de Recherche (ABR), surtout lorsque le nombre d'éléments dans l'arbre est considérable.

★ **Conclusion :**

La conclusion tirée de notre étude est que dans un premier abord, les différences de complexité entre l'Arbre Binaire de Recherche (ABR) et l'Arbre Rouge-Noir (ARN) ne soient pas particulièrement significatives pour des ensembles d'éléments plus restreints, une distinction nette émerge lorsque la taille de l'ensemble d'éléments augmente. Dans ce contexte, la complexité logarithmique de l'ARN, notée O(log(N)), démontre sa supériorité en termes d'efficacité par rapport à la complexité linéaire de l'ABR, notée O(N). Ainsi, à mesure que le nombre d'éléments à traiter augmente, l'utilisation de l'Arbre Rouge-Noir se révèle plus avantageuse pour les opérations d'insertion, de recherche et de suppression, grâce à sa capacité à maintenir un équilibre structurel.