Programation Orienté Objet

Arbre Rouge Noir

Douh ZAYNAB

Table des matières

Introduction:

L'objectif de ce rapport est de documenter le code de l'arbre rouge-noir **(ARN)** demandé en TP de Java. Avant de passer à la documentation il est important d' expliquer c'est quoi un arbre binaire de recherche **(ABR)** puisque l'arbre rouge-noir **(ARN)** est l'un de ses types particulier .

Un arbre de recherche binaire est une structure de données arborescente qui permet de représenter un ensemble de données

Un ABR (BST en anglais), c'est-à-dire qu'il se compose d'un nœud racine et de deux sous-arbres, gauche et droit. Chaque nœud de l'arbre contient une clé, qui représente une valeur de l'ensemble de données.

Les nœuds de l'arbre sont organisés de manière tel que :

- Les clés des nœuds du sous-arbre gauche soient toujours inférieures à la clé du nœud courant.
- les clés des nœuds du sous-arbre droit soient toujours supérieures à la clé du nœud courant.

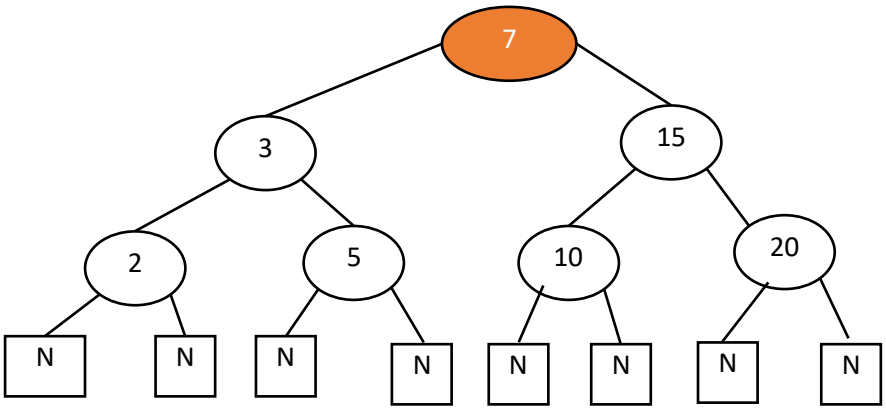
Les opérations caractéristiques sur les ABR sont l'insertion, la suppression et la recherche d'une clé.

Les opérations possibles sur un ABR :

Insertion d'une clé :

L'insertion d'une clé dans un arbre binaire de recherche consiste à trouver l'endroit où la clé doit être insérée, puis à créer un nouveau nœud avec la clé et à le placer à cet endroit.Pour trouver la feuille où la clé doit être insérée, on commence par la racine de l'arbre. On compare la clé à insérer à la clé du nœud Courant Si la clé à insérer est inférieure à la clé du nœud courant, on explore le sous-arbre gauche si elle est supérieure, on explore le sous-arbre droit . Ce processus se répète de manière récursive jusqu'à ce que l'emplacement approprié pour l'insertion soit trouvé.

Exemple:



Voici un exemple de graphe d'insertion pour insérer les clés 7, 15, 20 ,10,3 et 5 dans un arbre binaire de recherche vide:

- L'arbre est vide, donc la clé 7 devient la racine de l'arbre.
- La clé 15 est supérieure à la clé 7, donc elle est placée dans le sous-arbre droit de la racine.
- La clé 20 est supérieure à la clé 7 et 15, donc elle est placée dans le sous-arbre droit de la clé 15.
- La clé 10 est inférieure à la clé 7 mais supérieure à la clé 15, donc elle est placée dans le sous-arbre gauche de la clé 15.
- La clé 3 est inférieure à 7 donc elle est placée dans le sous-arbre gauche de la clé 7.
- La clé 5 est inférieure à 7 mais supérieure à 3 , donc elle est placée dans le sous-arbre droit de la clé 3.

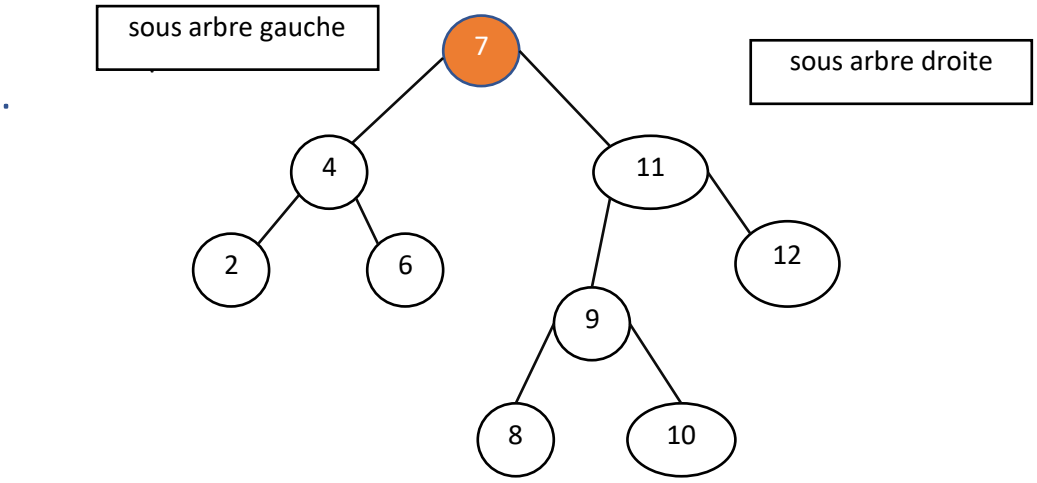
Recherche d'une clé :

On commence par se placer à la racine de l'arbre.

- Si la clé recherchée est égale à la clé du nœud courant, l'algorithme a trouvé la clé et renvoie `vrai`.
- Sinon, l'algorithme compare la clé recherchée à la clé du nœud courant.
    - Si la clé recherchée est inférieure à la clé du nœud courant, l'algorithme va à la branche gauche.
    - Sinon, l'algorithme va à la branche droite.

La recherche s'arrête lorsque la clé est rencontrée ou que l'on a atteint l'extrémité d'une branche .

Exemple :



L'algorithme de recherche commence par la racine de l'arbre. La clé recherchée, 6, est inférieure à la clé du nœud racine, 7. Par conséquent, l'algorithme descend dans la branche gauche.

Le nœud suivant est 3, qui est inférieur à la clé recherchée 6. Par conséquent, l'algorithme descend dans la branche à droite.

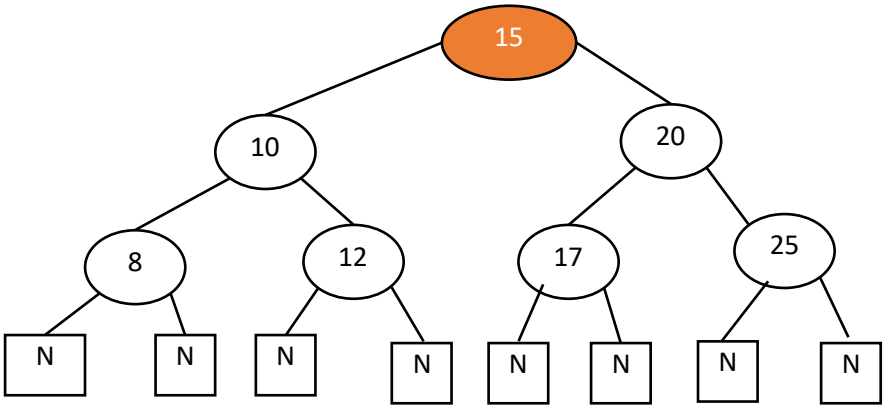
Le nœud suivant est 6, qui est égal à la clé recherchée. Par conséquent, l'algorithme a trouvé la clé et renvoie `vrai`.

Suppression d'une clé :

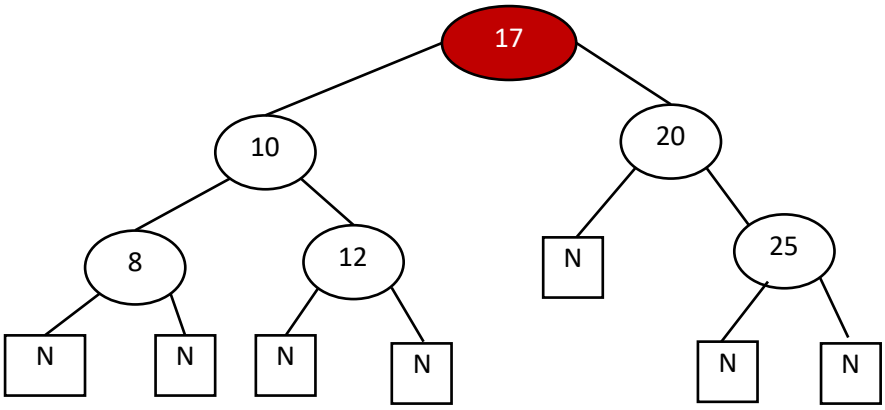
L'opération dépend du nombre de fils du nœud a supprimer.

- Cas 1 : le nœud a supprimer est une feuille. Il suffit de décrocher le nœud de l'arbre. Si le père n'existe pas l'arbre devient vide.
- Cas 2 : le nœud a supprimer a un fils et un seul. Le nœud est remplacé par son fils unique.
- Cas 3 : le nœud a supprimer p a deux fils. Il doit prendre comme clé celle de son suivant. Soit q le nœud de son sous-arbre gauche qui a la clé la plus grande (on peut prendre indifféremment le nœud de son sous-arbre droit de clé la plus petite). Il suffit de recopier la clé de q dans le nœud p et de supprimer le nœud q

Exemple:



On supprime 15 on obtient :

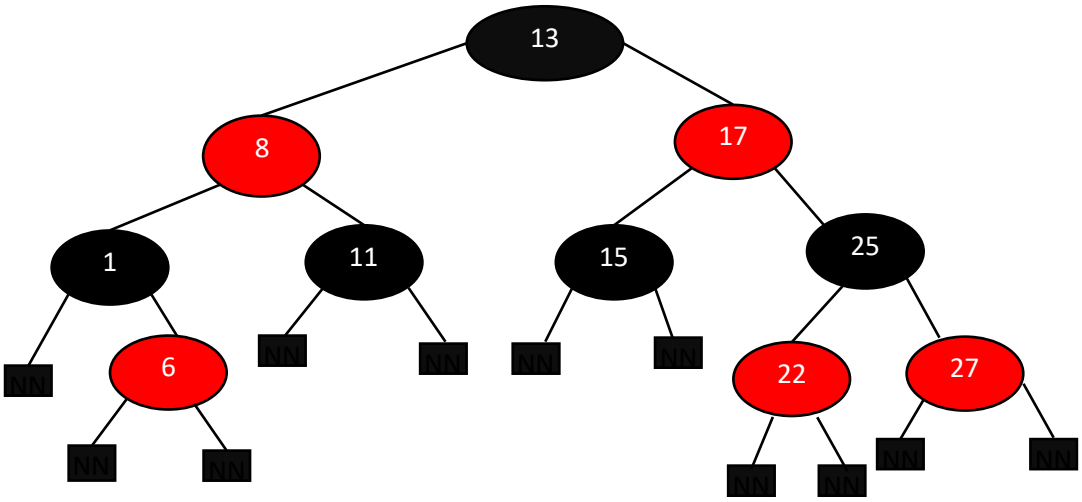


Les opérations possibles sur un arbre rouge-noir :

Définition :

L'arbre rouge noir est un ABR équilibré. Chaque nœud de l'arbre possède en plus de ses données propres un attribut binaire qui est souvent interprété comme sa "couleur" (rouge ou noir), ce dernier permet de garantir l'équilibre de l'arbre. Les propriétés qui permettent à l'ABR d'être équilibré sont :

1. Chaque nœud est soit rouge, soit noir.
2. La racine est noire.
3. Les feuilles (sentinelle) sont noire.
4. Si un nœud est rouge,ses deux fils sont noirs.
5. Pour tout nœud X, tous les chemins reliant X a une feuille contient le même nombre de nœud noirs.

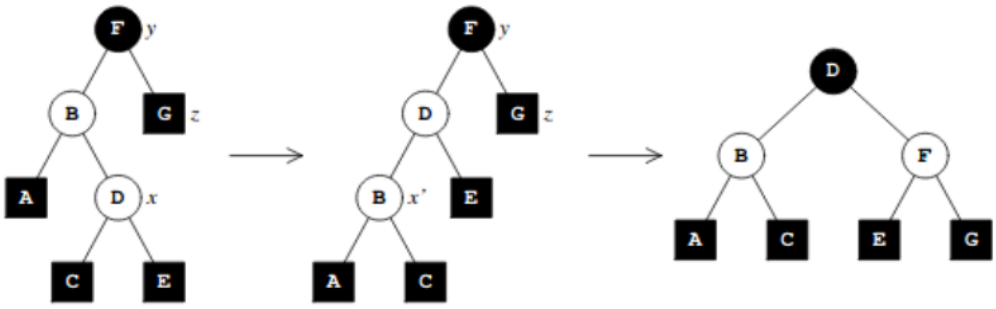


La recherche dans un arbre rouge-noir se déroule de la même manière que dans les arbres binaires de recherche classiques. Toutefois, lors de l'insertion ou de la suppression de nœuds, des ajustements spécifiques doivent être effectués sur les relations entre les nœuds et leurs couleurs pour garantir le respect des propriétés fondamentales de l'arbre rouge-noir. Ainsi, après l'ajout ou la suppression d'une clé, des réarrangements de nœuds ou des modifications de couleur sont nécessaires pour maintenir l'équilibre de l'arbre et conserver ses propriétés spécifiques.

Insertion d'une clé :

L'ajout d'une clé à un arbre Rouge-Noir débute par la procédure standard d'insertion dans un arbre binaire de recherche conventionnel. Un nouveau nœud est inséré et coloré en rouge, puis l'arbre résultant est ajusté pour rétablir les propriétés spécifiques aux arbres Rouge-Noir qui pourraient avoir été enfreintes. Il est pertinent de noter initialement que seules les propriétés 2 et 4 nécessitent une correction.

- **cas 1:** où le nœud z est rouge, deux sous-cas sont traités de manière similaire, en fonction de la position de x en tant que fils gauche ou droit. L'application d'une coloration corrige les propriétés de l'arbre Rouge-Noir, à l'exception éventuelle de la deuxième propriété, et aucune modification n'est apportée à la hauteur noire.Cependant, il est important de noter que le père de y pourrait être rouge, ou y pourrait être la racine. Dans de tels cas, la correction doit être effectuée au niveau supérieur pour assurer la conformité aux propriétés de l'arbre Rouge-Noir.
- **Cas 2 :** Si z est noir, cette situation est démontrée dans la figure . Pour résoudre ce cas, nous simplifions d'abord le problème en considérant le scénario où x est un fils gauche, réalisant une rotation gauche initiale (indiquée par la première flèche). Ensuite, nous procédons à une rotation droite suivie d'une étape de coloration (illustrée par la seconde flèche). À la suite de cette séquence d'opérations, le sous-arbre résultant conserve les propriétés d'un arbre Rouge-Noir, et aucune modification n'est apportée à la hauteur noire. Ainsi, la correction est considérée comme complète.



Suppression d'une clé :

Étape 1 : Recherche du nœud à supprimer

La suppression d'un élément d'un arbre rouge-noir commence par la recherche du nœud à supprimer. L'algorithme de recherche d'un nœud dans un arbre binaire de recherche est applicable aux arbres rouge-noirs.

Étape 2 : Suppression du nœud

Une fois le nœud à supprimer trouvé, il faut le supprimer. Il existe deux cas possibles :

- Si le nœud a un seul enfant, ce dernier remplace le nœud supprimé.
- Si le nœud n'a pas d'enfants, le nœud supprimé est simplement supprimé.

Étape 3 : Rééquilibrage de l'arbre

La suppression d'un nœud noir peut entraîner une violation de la propriété 4 des arbres rouge-noirs, qui stipule que tous les chemins descendants d'un nœud à une feuille contiennent le même nombre de nœuds noirs. Pour rétablir cette propriété, il faut effectuer une opération de rééquilibrage.

Il existe quatre cas de rééquilibrage possibles :

- **Cas 1 : le nœud supprimé est rouge**

Dans ce cas, aucune opération de rééquilibrage n'est nécessaire.

- **Cas 2 : le nœud supprimé est noir et son seul enfant est noir**

Dans ce cas, le nœud supprimé est remplacé par son enfant, qui devient noir. Cette opération ne viole pas la propriété 4 des arbres rouge-noirs.

- **Cas 3 : le nœud supprimé est noir et son seul enfant est rouge**

Dans ce cas, le nœud noir est remplacé par son enfant rouge, qui devient noir. Cette opération ne viole pas la propriété 4 des arbres rouge-noirs.

- **Cas 4 : le nœud supprimé est noir et ses deux enfants sont noirs**

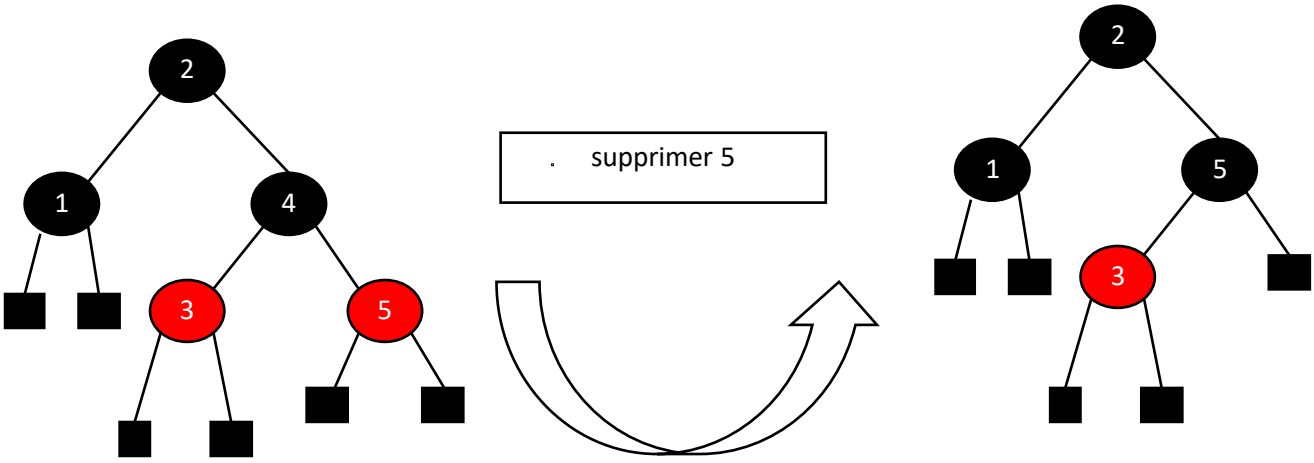
Dans ce cas, le nœud supprimé est remplacé par son enfant le plus à gauche ou le plus à droite. Cette opération viole la propriété 4 des arbres rouge-noirs, il faut donc effectuer une rotation pour rétablir cette propriété.

Explication du cas difficile

Le cas difficile est le cas 4, lorsque le nœud supprimé X était noir sans être la racine et il n'avait pas de fils rouge. La situation est que la hauteur noire des feuilles en dessous de F est maintenant en déficit de 1 par rapport aux autre nœuds.

Pour rétablir la propriété 4, il faut effectuer une rotation sur le nœud F. La rotation consiste à intervertir la position de F et de son parent. La couleur de F et de son parent est également inversée.

La rotation a pour effet de déplacer F vers le haut de l'arbre. Cela permet de rétablir la propriété 4, car le nœud F a désormais un fils rouge.



Comparaison ARN vs structure linéaire :

- Les arbres rouges noir sont équilibrés, ce qui garantit que les opérations de recherche, d'insertion et de suppression sont efficaces, même si l'arbre est très grand. Les structures linéaires ne sont pas équilibrées, ce qui peut entraîner une dégradation des performances des opérations de recherche, d'insertion et de suppression si l'arbre est déséquilibré.
- Les opérations de recherche, d'insertion et de suppression dans un arbre rouge noir sont en moyenne O(log(n)). Les opérations de recherche, d'insertion et de

suppression dans une structure linéaire sont en moyenne O(n) donc ARN est pluis efficace

- Les arbres rouges noir utilisent une couleur supplémentaire pour chaque nœud, ce qui peut entraîner une utilisation supplémentaire de l'espace mémoire. Les structures linéaires n'utilisent pas d'espace supplémentaire

Comparaison ARN vs ABR :

Les opérations de recherche, d'insertion et de suppression dans un arbre rouge noir sont en moyenne O(log(n)). Les opérations de recherche, d'insertion et de suppression dans un arbre de recherche binaire classique sont en moyenne O(log(n)) pour la recherche et O(n) pour l'insertion et la suppression.

Les arbres rouges noir sont une structure de données plus efficace que les arbres de recherche binaires classiques. Cependant, les arbres de recherche binaires classiques sont plus simples à implémenter et à comprendre

Les classes utilisées pour implémenter l'arbre:

Rappels :

- ✓ L'arbre binaire de recherche et l'arbre rouge-noir sont des structures de donnée. Alors une implémentation de l'interface `Collection` basée sur les arbres rouge-noir.
- ✓ **AbstractCollection** est une classe Java qui fournit des méthodes de base pour manipuler des collections. Elle peut être utilisée pour implémenter un arbre rouge noir. La méthode **iterator()** de la classe **AbstractCollection** retourne un objet **Iterator.** Un objet **Iterator** est un objet qui a pour rôle de parcourir une collection.
- ✓ **L'arbre binaire** est une structure de données qui suppose que les éléments peuvent être comparés par une relation d'ordre. En Java, cette hypothèse est exprimée en demandant que la classe des éléments implémente l'interface **Comparable**. L'interface **Comparable** déclare une seule méthode : Cette méthode retourne un entier qui représente la relation d'ordre entre les deux éléments comparés

```
interface Comparable {
    int compareTo ( Object o ) ;
}
```

Cette méthode retourne un entier <0, 0,ou >0 selon que l'objet auquel elle est appliquée précède,est égal ou suit o.

La classe Noeud :

On va commencer par implementation de la classe externe Noeud

Les attributs :

La classe Noeud contient cinq attributs qui sont :

Attribut	Description
E cle	Représente l'information d'un nœud.
Noeud gauche	Représente le fils gauche du nœud courant
Noeud droit	Représente le fils droit du nœud courant
Noeud pere	Représente le nœud père du nœud courant.
Char couleur	Représente la couleur du nœud($N_{(noir)}$ ou $R_{(rouge)}$)

Les Consturteurs :

La classe Noeud <E> a deux constructeurs :

a. le premier est un constructeur sans paramètre qui construit un objet de type Noeud ces attributs sont initialiser à sentnelle et sa couleur est N (noir).
b. Le deuxième est un constructeur paramétrique . Ce dernier contient comme clé et la couleur les deux paramètres passé au paramétre , son gauche et son droit sont la sentinelle.

Les methodes:

Noeud suivant()

```
Noeud suivant() {
    Noeud courant = this;
    // Cas où le Noeud possède un fils droit
    if (courant.droit != sentinelle) {
        // Si le fils droit existe, renvoyer le minimum du sous-arbre droit
        return courant.droit.minimum();
    }
    // Si le noeud n'a pas de fils droit, remonter dans l'arbre
    Noeud suivant = courant.pere;
    // On remonte tant que le noeud suivant existe et que le courant est le fils droit du suivant
    while (suivant != sentinelle && courant == suivant.droit) {
        courant = suivant;
        suivant = suivant.pere;
    }
    return suivant;
}
```

Le but de la méthode **`suivant()`** est de retourner le noeud suivant dans l'arbre rouge-noir, par rapport au noeud actuel. Cette méthode est utile pour itérer sur l'arbre rouge-noir dans l'ordre croissant.

Noeud minumum():

```
Noeud minimum() {
    Noeud courant = this;
    // Parcours à gauche
    while (courant.gauche != sentinelle) {
        courant = courant.gauche;
    }
    return courant;
}
```

Le but de Cette méthode retourne le nœud dont la clé est la plus petite clé du sous-arbre du nœud courant elle procède de la manière suivante : Pour accéder à la plus petite clé dans un ABR il suffit de descendre sur le fils gauche autant que possible. Le dernier nœud visité, qui n'a pas de fils gauche , porte la clé la plus petite de l'arbre.Le but de Cette méthode retourne le nœud dont la clé est la plus petite clé du sous-arbre du nœud courant elle procède de la manière suivante : Pour accéder à la plus petite clé dans un ABR il suffit de descendre sur le fils gauche autant que possible. Le dernier nœud visité, qui n'a pas de fils gauche , porte la clé la plus petite de l'arbre.

La classe ABRIterator :

Cette classe implémente l'interface Iterator. Elle permet de créer un iterateur qui parcourt un arbre rouge-noir.

Les attribus :

Cette classe contirnt juste deux attributs :

Attribut	Description
Noeud<E> suivant	Représente le nœud qui suit le noeud courant .
Noeud <E> precedent	Représente le nœud qui précède le noeud courant

Constructeur:

La classe ARNIterator a un constructeur sans paramètre qui initialise précédent a la sentinelle et initialise suivant au minimum de l'arbre et précédent a la sentinelle.

Les méthodes :

Boolean hasNext():

```
public boolean hasNext() {
    return suivant != sentinelle;
}
```

Retourne true si l'arbre possède encore des éléments à itérer.

E next() :

```java
// Méthode pour obtenir l'élément suivant dans le parcours
public E next() throws IllegalStateException {
    // S'il n'y a pas d'élément suivant, lance une exception
    if (!hasNext()) {
        throw new IllegalStateException();
    }
    precedent = suivant;
    suivant = suivant.suivant();
    return precedent.cle;

}
```

Cette méthode retourne la clé du nœud suivant si le courant a un suivant et le courant devient son suivant, sinon :NoSuchElementException.Elle utilise la méthode hasNext().

void remove () :

la méthode `remove()` supprime le dernier élément renvoyé par `next()`. Si `remove()` est appelée sans qu'aucun appel préalable de `next()` n'ait eu lieu, elle lance une exception **`IllegalStateException`.**

De plus, elle spécifie explicitement l'utilisation de la méthode `supprimer(Noeud<V>)` de la classe ARN pour effectuer la suppression du nœud.

```java
public void remove() throws IllegalStateException {
    // Vérifie si la méthode remove() est appelée sans appel préalable de next()
    if (precedent == sentinelle) {
        throw new IllegalStateException();
    }
    supprimer(precedent);
    // Réinitialise le nœud précédent à la sentinelle après la suppression
    precedent = sentinelle;

}
```

La class ARN<E>:

Les attributs:

Cette classe contient quatre attributs les voici :

Attribut	Description
Noeud racine	Représente le nœud racine de l'arbre. C'est le point de départ de l'arbre rouge-noir.
int taille	Représente la taille de l'arbre, c'est-à-dire le nombre total de nœuds dans l'arbre. Cette variable est souvent mise à jour lors de l'insertion et de la suppression de nœuds.
Noeud sentinelle	Représente un nœud spécial appelé sentinelle. Dans les arbres rouge-noir, la sentinelle represente un nœud null
Comparator<? super E> cmp	Représente un comparateur utilisé pour comparer les clés des éléments stockés dans l'arbre. Cet attribut est généralement utilisé

	pour déterminer l'ordre des éléments dans l'arbre et peut être fourni lors de la création de l'arbre pour spécifier comment comparer les éléments.

Les constructeurs :

La classe **ARN<E>** a trois constructeurs :

`public ARN():` un constructeur sans parametre qui crée un arbre vide.

`public ARN(Comparator<? super E> comp:` ce constructeur prend un argument un Comparator comp et crée un arbre vide. Les éléments sont comparés selon l'ordre imposé par le Comparateur comp.

`public ARN(Collection<? extends E> colle):` Un constructeur par recopie qui prend en argument une Collection collec et crée un arbre qui contient les mêmes éléments que collec. L'ordre des éléments est l'ordre naturel.

Les méthodes :

void rotationGauche(Noeud n): Le but de la méthode rotation gauche est de corriger une violation de la propriété de propriété de déséquilibre de l'arbre rouge-noir. La rotation gauche est une opération qui consiste à échanger la position de deux noeuds dans l'arbre. La rotation gauche est effectuée sur un noeud noir dont le fils gauche est rouge.

Noeud rotationDroite(Noeud n): Cette méthode permet de faire la rotation droite du nœud n. n nœud devient le fils droit du nœud qui était son fils gauche.

Noeud rechercher(Object o) :

```
1 usage
private Noeud rechercher(Object o) {
    Noeud courant = racine;
    while (courant != sentinelle && cmp.compare((E) o, courant.cle) != 0) {
        courant = cmp.compare((E) o, courant.cle) < 0 ? courant.gauche : courant.droit;
    }
    return courant;
}
```

le but de la méthode est de rechercher un élément dans l'arbre rouge-noir en utilisant la clé spécifiée, en parcourant l'arbre de manière itérative et en mettant à jour le nœud courant en fonction de la comparaison des clés jusqu'à ce que l'élément soit trouvé ou que la sentinelle soit atteinte.

void ajouter(Noeud ajout):

Insère un nouveau noeud à l'arbre rouge-noir et corrige l'équilibre de l'arbre.

On passe par 3 etapes :

- Trouve la position du nouveau noeud.
- Insère le nouveau noeud.

- Corrige l'équilibre de l'arbre.

Après l'insertion du nœud elle appelle **`ajouterCorrection(correction)`** pour organiser l'arbre si ces propriétés sont violées.

`void ajouterCorrection(Noeud correction):`

Cette méthode réarrange l'arbre après l'insertion plus clair elle corrige les violations de la propriété de déséquilibre de l'arbre rouge-noir après l'ajout d'un nouveau noeud.

Noeud supprimer(Noeud n) :

Cette méthode supprime un nœud. Après la suppression, elle appelle **`supprimerCorrection(Noeud x)`** pour organiser l'arbre si ces propriétés sont violées.

supprimerCorrection(Noeud x):

Cette méthode reorganisation de l'arbre, en remontant vers la racine. Si le nœud supprimé était rouge, elle ne change rien puisque tous les propriétés sont valide. Si le nœud supprimé était noir, alors la porpriété 5 est violée.

boolean addAll(Collection<? extends E> c): ajoute tous les éléments de la collection spécifiée à l'arbre. Elle itère à travers chaque élément de la collection et utilise la méthode `add` pour ajouter l'élément à l'arbre. Si au moins un élément est ajouté avec succès, la méthode renvoie `true`, indiquant une modification de l'arbre. Sinon, elle renvoie `false`.

int maxStrLen(Noeud x) :`maxStrLen` retourne la longueur maximale des représentations en chaîne (string) des clés des nœuds dans un arbre binaire, en utilisant une approche récursive. Si le nœud x est la sentinelle, la longueur est 0 ; sinon, elle compare la longueur de la clé de x avec les longueurs maximales des clés dans les sous-arbres gauche et droit de x, renvoyant le maximum de ces valeur.

Les méthodes redéfini:

Parmi les methode redefini qu'on a:

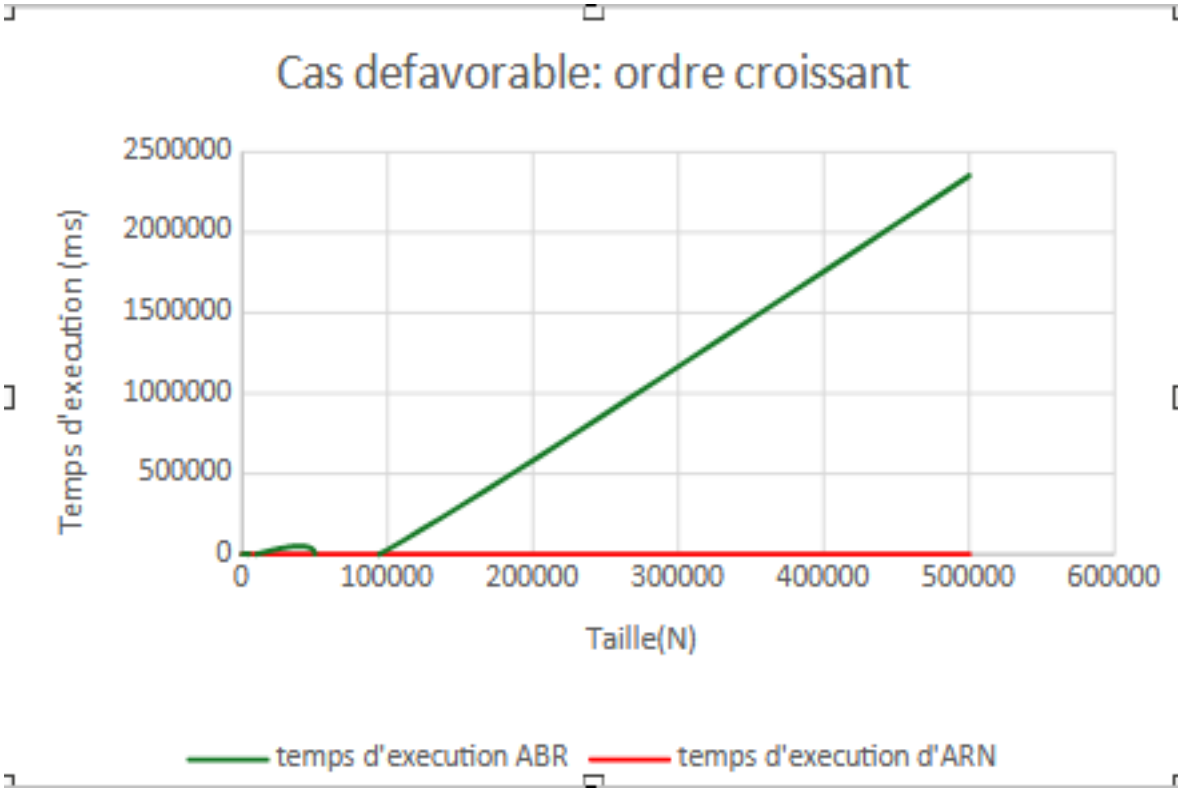
- ✓ **int size():** Retourne le nombre de nœud dans l'arbre.
- ✓ **boolean add(E element):** Cette méthode crée un nœud **Z** avec la clé **element** puis appelle la méthode ajouter(z) qui insère le nœud ensuite ajouterCorrection(z) qui organise l'arbre après l'insertion.
- ✓ **Boolean contains(Object o) :** Cette méthode vérifie si un objet existe dans l'arbre en appelant la méthode rechecher(o).

Le temps d'exécution :

Remarque:

On trouve le temps d'exécution grâce la méthode currentTimeMillis qui renvoie la valeur en (ms).

Cas défavorable d'insertion :

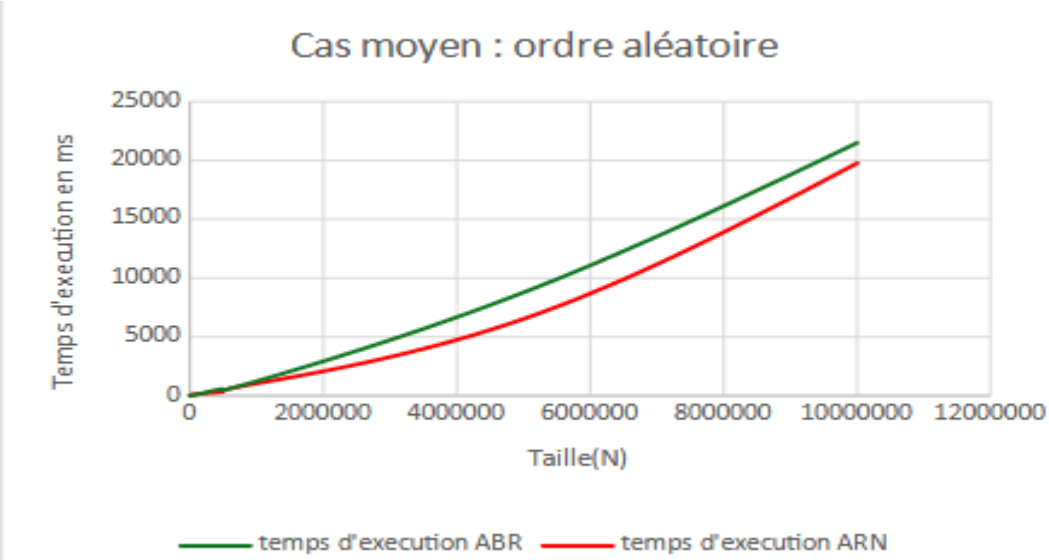


Ce graphe represente le temps d'exécution d'un arbre binaire de recherche (ABR) et d'un arbre rouge-noir (ARN) en fonction de la taille de l'entrée.

Le graphe montre que le temps d'exécution de l'ABR augmente rapidement que celui de l'ARN en fonction de la taille de l'entrée. Pour une taille d'entrée de 500000 d'éléments, l'ABR prend environ 2351573 ms à s'exécuter, tandis que l'ARN prend environ 161 millisecondes.

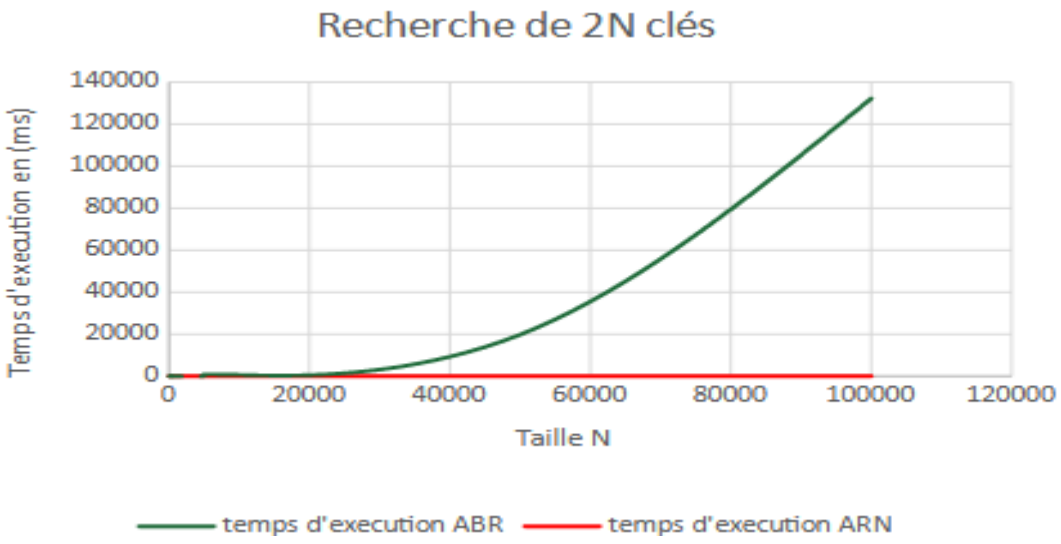
Ainsi, il existe une nette distinction dans les temps de traitement entre un Arbre Binaire de Recherche (ABR) et un Arbre Rouge-Noir (ARN). L'observation du graphe révèle que le temps d'exécution de l'ARN demeure constant.

Cas moyen d'insertion :



Ce graphique illustre le temps d'exécution d'un Arbre Binaire de Recherche (ABR) par rapport à celui d'un Arbre Rouge-Noir (ARN), en fonction de la taille de l'entrée. Pour de petites tailles, les temps d'exécution de l'ABR et de l'ARN sont quasiment identiques, avec peu de différence perceptible. Cependant, à mesure que la taille de l'entrée augmente, on observe une croissance plus rapide du temps d'exécution de l'ABR par rapport à celui de l'ARN. Par exemple, pour une taille d'entrée de 10 000 éléments, l'ABR nécessite environ 21 ms, tandis que l'ARN demande environ 19 ms. À une taille d'entrée de 500 000 éléments, l'ABR atteint environ 63 ms, alors que l'ARN prend 49 ms.

Le temps d'execution pour la recherche :



Ce diagramme représente l'évolution du temps d'exécution en fonction de la taille de l'entrée pour un arbre binaire de recherche (ABR) et un arbre rouge-noir (ARN).

On observe que, jusqu'à une taille d'environ 500 éléments, le temps d'exécution des deux structures est presque identique, avec une légère différence. Cependant, à mesure que la taille de l'entrée augmente, le temps d'exécution de l'arbre rouge-noir (ARN) augmente de manière relativement constante, semblant atteindre une stabilité. En revanche, le temps d'exécution de l'arbre binaire de recherche (ABR) augmente de manière significative.

Pour une taille d'entrée de 100 000 éléments, l'ABR nécessite environ 132 172 ms pour s'exécuter, tandis que l'ARN ne prend que 71 ms. Ainsi, le temps d'exécution de l'ABR est plus de 1 861 fois supérieur à celui de l'ARN.

Conclusion:

D’aprés les études des trois cas :

Recheche d’un nœud : L’algorithme de recherche dans un arbre rouge-noir est exactement le même algorithme que pour la recherche dans un arbre binaire de recherche. On a donc que la complexité en temps de la recherche dans un arbre rouge-noir est en O(log n).

Ajouter un noeud : L'ajout d'un nœud à l'arbre se déroule en trois phases distinctes. Dans la première phase, la recherche de la position du nouveau nœud requiert un temps O(log n). La deuxième phase, consacrée à l'insertion du nœud nouvellement créé, s'effectue en tempsO(1). Enfin, la troisième phase, qui concerne la réorganisation de l'arbre, prend un temps O(log n). Cette dernière étape implique au maximum log(n) opérations de recoloration, chacune exigeant un temps O(1). Par conséquent, l'ensemble du processus d'insertion d'une clé dans un arbre rouge-noir nécessite un temps total de O(log n).

D’apres une étude expérimentale le temps d'exécution d’ABR augmente de manière quadratique en fonction de la taille de l'entrée. L'arbre rouge-noir est une structure de données plus complexe qui est plus efficace que l'arbre binaire pour les grandes tailles d'entrée. L'ARN garantit que la hauteur de l'arbre est en **O(log n)**, ce qui signifie que le temps d'exécution de l'ARN est en **O(log n)**, même dans le pire des cas.

En conclusion, les arbres rouge-noir sont une meilleure option que les arbres binaires de recherche pour les grandes tailles d'entrée. Ils offrent de meilleures performances et sont plus efficaces.