ÉTUDES DES STRUCTURES DE DONNÉES
ARBRE BINAIRE DE RECHERCHE & ARBRE ROUGE ET NOIR

Table des matières

Introduction:

Le monde évolue en permanence, le flux de donnée aussi. Ainsi pour répondre à cette demande incessante, nous devons nous développeur rendre ces flux plus optimisés. Afin que les ressources demandées soit moins nombreuses et gourmandes en énergie. La meilleure façon de parvenir à cette tâche est de construire des structures de données plus optimisés afin que le stockage, la recherche dans la structure, ou même la suppression soit plus rapide. En effet, les flux de donnée grossissent leur taille dépassant souvent et largement l'unité du Gigaoctet. Ainsi, il est devenu important de parvenir à stocker, parcourir ces données efficacement. Les mathématiciens & informaticiens se sont penchés sur la question tels que:

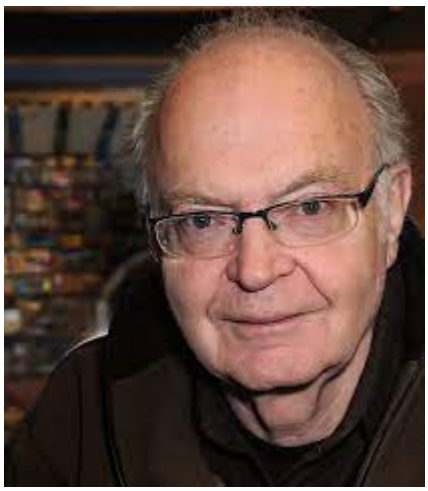


Figure I

Donald Knuth dans son livre 'The Art of Computer Programming,' Knuth a expliqué le fonctionnement des arbres binaires de recherche.

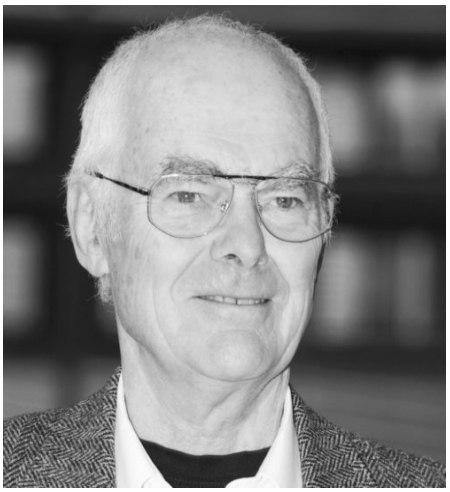


Figure II

Rudolf Bayer (& Edward M. McCreight), ils ont introduit les arbres B, qui sont à la base des arbres rouges et noirs. Les arbres B sont des arbres équilibrés utilisés pour la recherche et le stockage de données.

Les Arbres Binaires de Recherche (ABR)

I. Implémentation en JAVA

Pour implémenter nos ABR, nous aimerions qu'ils acceptent des données de n'importe quel type comparable. Cette classe implémentera la classe AbstractCollection<E> car on peut dire que cette structure de donnée est une collection d'objet de type E. Chaque donnée sera encapsulée dans un nœud.

A. Implémentation de la classe Nœud :

La classe noeud est une sous-classe des ABR correspondante à l'essentiel du fonctionnement d'un ABR. Chaque noeud aura une clé correspondant à la valeur que l'on souhaite insérer dans l'arbre. Il aura également un noeud dit 'à sa gauche' et un noeud 'à sa droite'. Nous verrons plus tard dans ce rapport à quoi ces noeuds gauche et droit servent. Un noeud possèdera deux constructeur:

- Un constructeur vide

- Un constructeur avec un Element de type E. Ce type permet de stocker la majorité des objets en java.

```
class Noeud {
  E cle;
  Noeud gauche;
  Noeud droit;
  Noeud pere;

  private Noeud() {
      cle = null;
      gauche = null;
      droit = null;
      pere = null;
  }

  private Noeud(E cle) {
    this.cle = cle;
    gauche = null;
    droit = null;
    pere = null;

  }
}
```

Lorsqu'on utilise le constructeur vide, on instancie un objet de classe noeud. Il aura une clé null, un noeud gauche null, un noeud droit null et un noeud pere à null. D'un autre côté, quant on utilise le constructeur avec un Element, on affecte cet élément à l'attribut clé de l'objet. Ses noeuds sont quant à eux vide. Voici un brève aperçu de comment cela se traduit en java:

Maintenant que l'on sait comment créer un noeud, nous aimerions connaître le noeud minimum à l'instance du noeud sur laquel nous sommes.

Nous aimerions également savoir quel est le noeud suivant sur l'instance du noeud. Afin de pouvoir itérer sur l'arbre de manière plus efficace.

Ainsi voici un petit aperçu des méthodes minimum() et suivant() appartenantes à la classe noeud en java.

```
Noeud minimum() {
  Noeud courrant = this;
  while (courrant.gauche != null) {
    courrant = courrant.gauche;
  }
  return courrant;
}
```

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

Pour récupérer le noeud minimum de l'instance, nous allons explorer la branche gauche de ce noeud jusqu'à tomber sur le dernier noeud de cette branche, traduit par un pointeur à gauche équivalent à null.

Pour récupérer quant à lui le noeud suivant, nous allons premièrement regarder l'existence d'un noeud droit sur celui que nous sommes afin de récupérer le minimum de ce noeud qui est donc le noeud suivant. Si le noeud n'a pas de noeud à sa droite, c'est qu'il est déjà le noeud minimum de sa branche droite, donc il est inutile d'appeler la fonction minimum(). Une fois que nous sommes au minimum de la branche droite de notre instance, nous allons 'remonter l'ABR' à l'aide des pointeurs sur les pères des noeud que nous allons rencontrer jusqu'à trouver pour quel noeud courrant est un fils gauche qui lui sera donc notre suivant. Cas spécial, courrant est en réalité le noeud ayant la clé maximale, il n'existe donc pas de suivant, ainsi on retourne 'null' comme pour dire 'absence de noeud suivant'.

Il existe trois types de noeud:

- L'unique noeud racine de l'Arbre Binaire. Il se trouve tout en haut de l'arbre. Il n'a pas de père.
- Les noeuds centraux. Ils sont chacun un père et au moins un enfant.
- Les feuilles. Ils n'ont aucun enfant.

Pour déterminer si l'instance du noeud sur lequel nous sommes est une feuille, nous utiliserons la méthode isFeuille() qui retourne un boolean si il ne possède pas d'enfant.

Pour résumer, un noeud possède comme attribut:

1. Une clé de type E. Permettant de stocker une majorité d'objet de java.
2. Un noeud gauche. Ce noeud sera inférieur au noeud d'instance. Nous utiliserons un comparator pour vérifier l'infériorité de la valeur. Ce noeud aura comme valeur null si le noeud d'instance est une feuille.
3. Un noeud droit. Ce noeud sera supérieur au noeud d'instance. Nous utiliserons un comparator également pour vérifier la supériorité de la valeur. Ce noeud aura comme valeur null si le noeud d'instance est une feuille.
4. Un noeud père. Ce noeud sera le père du noeud actuel et sera null si le noeud d'instance est la racine.
5. Nous disposons de trois méthodes, l'une permettant de récupérer le noeud minimum du noeud d'instance, une autre pour récupérer dans tous les cas (sauf quand le noeud est le noeud ayant la clé maximale) son noeud suivant et une dernière pour savoir si le noeud d'instance est une feuille ou non.

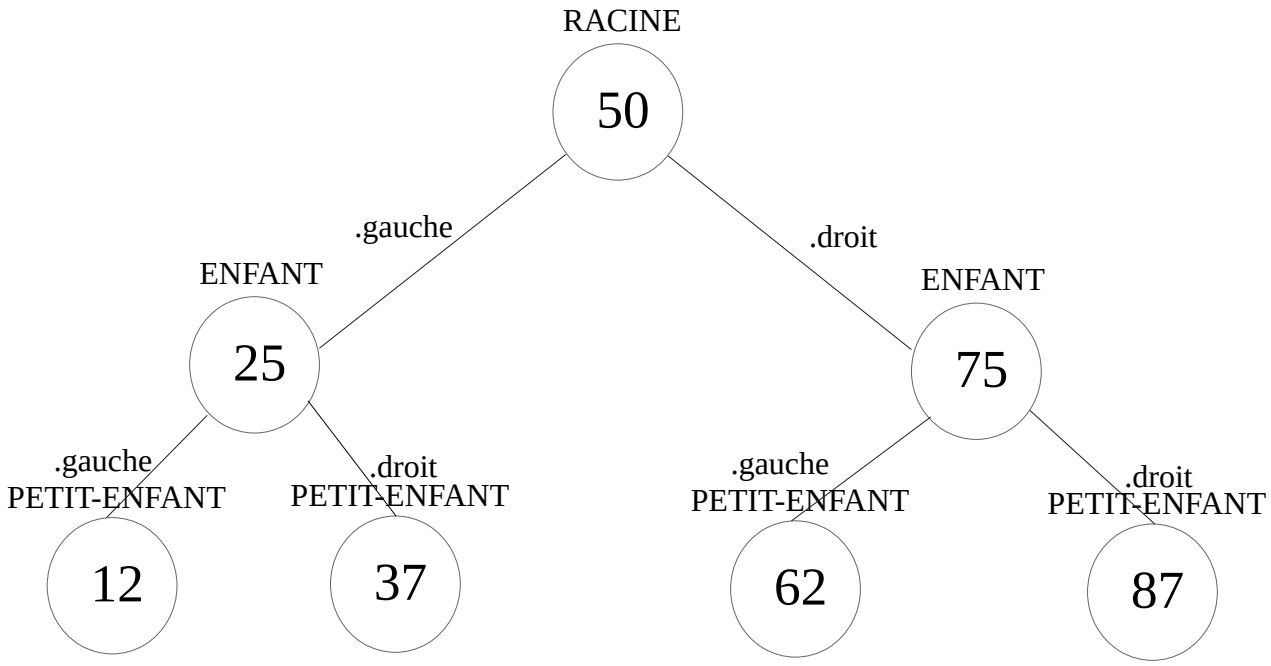
B. Les constructeurs de la classe ABR :

Dorénavant qu'on dispose de la classe noeud, nous disposons d'un système permettant de relier des noeuds entre-eux. Le plus dur est en réalité déjà réalisé. Nous allons dorénavant voir comment créer une structure de noeuds reliés les uns des autres. Et pour se faire, nous aurons donc 3 constructeurs différents.

1. Un constructeur vide. Permettant de disposer d'un comparateur par défault.
2. Un constructeur dans lequel on définit le comparateur.
3. Un constructeur par recopie qui réalise une copie d'un ABR.

Chaque Arbre Binaire de Recherche disposera comme paramètre:

Un noeud racine. Ce noeud est l'ABR en entier en réalité car les enfants de ce noeud auront ou non des enfants droit et gauche jusqu'à tomber sur les feuilles. Ce phénomène s'illustre ci-dessous:



Chaque arbre dispose également d'une taille équivalent au nombre de noeud que possède l'arbre en tout et d'un comparateur permettant de comparer deux types de données afin de déterminer l'ordre de l'arbre. Voici un bref aperçu des constructeurs de la classe ABR.

Le constructeur vide:

```
public ArbreBinaireDeRecherche(Comparator<? super E> cmp) {
  racine = null;
  taille = 0;
  this.cmp = cmp;
}
```

Le constructeur dans lequel on définit un comparateur:

```
public ArbreBinaireDeRecherche(Comparator<? super E> cmp) {
  racine = null;
  taille = 0;
  this.cmp = cmp;
}
```

Le constructeur par recopie:

```java
public ArbreBinaireDeRecherche(Collection<? extends E> c) {
  Iterator<? extends E> it = c.iterator();
  while (it.hasNext()) {
    E noeudToAppend = (E) it.next();
    this.add(noeudToAppend);
  }
}
```

On utilise l'itérateur des ABR pour parcourir la structure entrée en paramètre et ajouter à chaque itération le prochain noeud à l'aide de la méthode add(noeud Exemple) que l'on verra par la suite.

C. L'Itérator des ABR :

Afin de naviguer plus facilement dans la structure de donnée, on peut utiliser un Iterator. Pour cela on implément la sous-classe ABRIterator. Cette sous-classe implémente l'interface Iterator. Elle possède comme unique paramètre courrant (oui, j'écris courrant comme ça) une copie d'une noeud racine afin d'itérer sur l'arbre sans rien trop bouger au sein de la structure. C'est d'ailleurs dans le constructeur la classe que l'on va affecter au paramètre le noeud racine de l'ABR. Pour créer cette classe implémentant l'interface Iterator<E>, nous devons surcharger les méthodes hasNext() et next().

1. Pour la méthode hasNext() le principe est de savoir si le noeud courrant admet un noeud suivant. Comme nous disposons de la méthode suivant() de la sous-classe noeud, il est aisé de déterminer ainsi l'existence d'un tel noeud.
2. Pour la méthode next() le principe est d'affecter courrant à son noeud suivant. Attention si il n'existe pas de suivant, on doit renvoyer null car la méthode renvoie un objet de type E.

Voici un bref aperçu des méthodes hasNext() et next() :

```java
@Override
public boolean hasNext() {
    Noeud tampon = courrant;
    boolean test = tampon.suivant() != null;
    return test;
}
```

```java
@Override
public E next() {
    if (courrant == null){
        return null;
    }
    this.courrant.suivant();
    return courrant.cle;
}
```

Comme nous n'utilisons pas les autres méthodes de l'interface Iterator dans le reste du code, il est inutile de surcharger les autres méthodes de l'Interface. N'étant pas fan des Iterator, j'ai réalisé la classe sans vraiment l'utiliser.

D. Ajouter un noeud dans un ABR :

Pour ajouter un noeud dans cette structure de donnée implémentant l'interface AbstractCollection, nous allons surcharger la méthode add de cette interface. La méthode prend en paramètre une clé du type E que l'on souhaite insérer et retourne un boolean pour savoir si l'ajout s'est réalisé avec succès. On refuse la valeur null comme valeur à ajouter. Ainsi on retourne false si jamais l'utilisateur souhaite ajouter null comme valeur. Maintenant ce petit cas spécial mis de côté, attaquons-nous aux autres.

- Premier cas, on ajoute une valeur dans un arbre vide. Un arbre vide est traduit par le fait que sa racine équivaut à null. Dans ce cas, c'est simple, on créer un noeud avec le constructeur dans lequel on définit un élément et on affecte ce nouveau noeud à la racine. On instancie ses fils à null. On incrémente taille de 1 et on retourne true.
- Second cas, l'arbre possède taille noeud(s). Dans ce cas, on créer un noeud (noeudToAppend) avec le constructeur dans lequel on définit un élément. On parcourt l'arbre jusqu'à tomber sur une feuille. Pour cela on regarde la valeur de courrant avec cmp.compare(noeudToAppend.cle, tampon.cle) tampon étant une copie du noeud courrant. Si la clé de noeudToAppend est inférieure, on affecte à courrant son fils gauche. Sinon son fils droit. Ainsi on arrive sur la feuille où il faut ajouter noeudToAppend. Si la clé est inférieure à ce noeud, on affecte à courrant.gauche noeudToAppend sinon on affecte à courrant.droit noeudToAppend. Enfin, on affecte le père de noeudToAppend à courrant. On finit par affecter les fils de noeudToAppend à null car c'est une feuille et on incrémente de 1 la taille pour finir par retourner true pour dire que l'opération s'est bien effectué.

Voici un petit aperçu de la fonction add de la classe ABR:

```
public boolean add(E element) {
      if( element == null ) return false;
      Noeud noeudtoAppend = new Noeud(element);
      Noeud tampon = null;
      Noeud courrant = racine;
      while (courrant != null){
          tampon = courrant;
          courrant = this.cmp.compare(noeudtoAppend.cle, courrant.cle) < 0 ? courrant.gauche : courrant.droit;
      }
      noeudtoAppend.pere = tampon;
      if(tampon == null)
          racine = noeudtoAppend;
      else{
          if(this.cmp.compare(noeudtoAppend.cle, tampon.cle) < 0)
              tampon.gauche = noeudtoAppend;
          else tampon.droit  = noeudtoAppend;
          noeudtoAppend.pere = tampon;
      }
      noeudtoAppend.gauche = noeudtoAppend.droit = null;
      taille++;
      return true;
  }
```

E. Rechercher un noeud dans un ABR :

La méthode rechercher prend en paramètre une clé de classe E que l'on souhaite insérer et retourne le noeud recherché pour effectuer diverse opération dessus. Comme toujours, si on recherche une valeur équivalente à null, on retourne null. Pour cette méthode il existe deux cas:

- La clé recherchée est dans l'ABR. On parcourt l'ABR jusqu'à ce que la clé de courrant équivaut la clé recherchée. On retourne courrant.
- La clé recherchée n'est pas dans l'ABR. On parcourt l'ABR on arrive jusqu'à une feuille, on compare la clé recherchée et la feuille. Comme la clé recherchée n'est pas dans l'arbre, la boucle s'arrête car on a affecté courrant à null car feuille.gauche ou feuille.droit est null et on retourne courrant donc null.

Voici un bref aperçu de la méthode rechercher de la classe ABR:

```
public Noeud rechercher(Object o) {
    if(o == null) return null;
    Noeud courrant = racine;
    while (courrant != null && courrant.cle != (E) o) {
        courrant = cmp.compare(courrant.cle, (E) o) > 0 ? courrant.gauche : courrant.droit;
    }
    return courrant;
}
```

F. Supprimer un noeud dans un ABR :

On surcharge la méthode remove() de la classe Collection. La méthode remove prend en paramètre une clé de classe Object que l'on souhaite supprimer dans l'ABR et retourne un boolean. Comme à son habitude, on accepte pas le fait de supprimer une valeur entrée en paramètre équivalente à null et donc nous retournons false. Evidemment, supprimer un élément dans un arbre vide retourne également false. Cette méthode possède trois cas spéciaux:

1. Le noeud a supprimé est une feuille. On va affecter au père null au noeud à supprimé. Si le noeud à supprimé est un fils droit, alors on affecte nulle à son père.droit sinon on affecte à son père père.gauche à null.
2. Le noeud a supprimé possède un fils. On va affecter au père, le fils au noeud à supprimé. Si le noeud à supprimé est un fils droit, alors on affecte son fils à père.droit sinon on affecte à père.gauche son fils.
3. Le noeud a supprimé possède deux fils. Là c'est déjà plus compliqué. Premièrement, on va récupérer le fils gauche du noeud courrant. On affecte le père du fils, son grand-père (père de courrant). Cas spécial le père de courrant est null donc on a atteint la racine alors on affecte la racine avec le fils. Ensuite nous allons gérer la réaffectation du noeud fils quand le noeud courrant est en cours de suppression. Si le noeud courrant est l'enfant gauche de son père, alors on définit au fils du père de courrant le fils sinon on définit au fils du père de courrant le fils. En d'autres termes, il maintient la propriété de l'arbre binaire de recherche après la suppression en s'assurant que le nœud père du nœud en cours de suppression pointe vers le bon nœud enfant après la suppression.

On désincrémente la taille à chaque fois que la suppression à correctement fonctionné.

Voici un bref aperçu de la fonction remove():

```
private boolean supprimer(Noeud toDelete) {
      if (racine == null || toDelete == null) {
          return false;
      }
      Noeud courrant;
      Noeud fils;
      // Si le noeud a supprimer est enfant unique on le supprime direct.
      courrant = toDelete.gauche == null || toDelete.droit == null ? toDelete : toDelete.suivant();
      if (courrant == null) {
          return false;
      }
      // On recupere le fils du noeud courant.
      fils = courrant.gauche != null ? courrant.gauche : courrant.droit;
      if (fils != null) {
          // Le pere du fils devient le pere de courant.
          fils.pere = courrant.pere;
      }
      if (courrant.pere == null) {
          // Si noeud courrant -> racine.
          this.racine = fils;
      } else {
          // Sinon -> le fils gauche ou droit du pere devient le fils.
          if (courrant.equals(courrant.pere.gauche)) {
              courrant.pere.gauche = fils;
          } else {
              courrant.pere.droit = fils;
          }
      }
      if (!courrant.equals(toDelete)) {
          // la cle a supprimer vaut la cle de courrant
          toDelete.cle = courrant.cle;
      }
      taille--; // desincrement la taille de 1 car on vient de supprimer le noeud toDelete
      return true;
  }

  @SuppressWarnings("unchecked")
  @Override
  public boolean remove(Object o){ // Appel de la fonction supprimer par le biais de collection
      return this.supprimer(this.rechercher((E) o));
  }
```

G. Afficher un ABR :

Après toutes ces explications assomantes, nous aimerions bien voir ce que ça donne dans la console au final. Pour cela on surcharge la méthode toString() de la classe Collection. On commence par créer un StringBuffer. Cette classe est très utile quant à sa maniabilité à créer des chaînes assez complexes telle que la notre. Un fois cela réaliser, nous allons appeler une autre méthode semblable à toString() mais différente par ses paramètres qui fait d'elle une fonction bien à part de celle surchargée. Elle se nomme également toString mais accepte en paramètre un noeud, un StringBuffer (celui que l'on vient de créer) une chaine correspondant au chemin du noeud et un entier correspondant à la taille correspondant au nombre de caractère du plus grand objet sur un niveau de l'arbre. Cette méthode sera récursive.

On fait son premier appel à la racine, avec un StringBuffer (buf) vide, un chemin de noeud vide (") puisqu'on est sur la racine, et on récupère la taille maximum avec maxStrLen(noeud enparamètre). Cette méthode, renvoie 0 si le noeud en paramètre est null. Sinon grâce à Math.max, la méthode va renvoyer la taille maximale entre la clé du noeud en paramètre, la clé maximale de son sous-arbre droit et gauche. On fait cela pour avoir des liens de taille correspondantes. Une fois dans la fonction récursive, notre condition d'arrêt sera si le noeud parcouru est null, alors on retourne void. Ensuite, on parcourt l'ABR tout à sa droite à l'aide d'appel récursif. Donc le reste du programme sera fait sur tout les niveaux de l'arbre et on commencera le programme sur le fils le plus à droite de l'ABR pour chaque niveau. Une fois les appels récursifs terminés, nous nous trouvons sur le noeud n'ayant pas de suivant. (Le noeud ayant la clé maximale de l'ABR).

Une fois que nous avons parcouru tout le côté droit de l'arbre, nous commençons à construire notre chaîne. Pour chaque niveau de l'arbre, nous ajoutons un certain nombre d'espaces à notre StringBuffer pour bien aligner notre affichage. Le nombre d'espaces ajoutés dépend de la longueur de la clé la plus longue sur ce niveau de l'arbre (c'est pour cela que nous avons calculé len précédemment). Ensuite, nous ajoutons un caractère spécial à notre chaîne. Ce caractère est '+' si nous sommes sur le dernier niveau de l'arbre, et '|' sinon. Cela permet de bien visualiser les différents niveaux de l'arbre.

Après cela, nous ajoutons la clé du nœud actuel à notre chaîne. Si le nœud a des enfants, nous ajoutons également '–' à notre chaîne pour indiquer qu'il y a des nœuds en dessous. Enfin, nous ajoutons un certain nombre de '-' à notre chaîne pour aligner correctement l'affichage, puis nous ajoutons '|'.

Une fois que nous avons terminé avec le nœud actuel, nous faisons un appel récursif à la méthode toString() sur le fils gauche du nœud. Nous continuons ce processus jusqu'à ce que nous ayons parcouru tout l'arbre. À la fin, nous retournons notre StringBuffer converti en chaîne de caractères.

Ceci conclut notre Implémention des Arbres Binaire de Recherche en Java.

II. Etude de la complexité & Temps pour des applications sur grands ABR :

A. Construction des données :

Dorénavant, nous aimerions bien savoir la complexité de cette structure de donnée en moyenne et dans le pire des cas. L'étude de la complexité fait très souvent intervenir le 'pire des cas'. Ce cas dans un ABR est simple, c'est un arbre dit 'complètement désordonné' où chaque valeur de clé est supérieur (ou l'inverse fonctionne aussi) à la dernière. Ainsi, nous avons un arbre avec une seule branche. Aussi on prendra le cas moyen, soit un arbre généré aléatoirement. Par convention, on instancira la clé de la racine à la valeur maximale générée par l'aléatoire divisée par 2. Ceci permet d'avoir un arbre nettement plus équilibré. Pour se faire, nous avons créer une classe Main.java dans laquelle nous construirons reload fois un ABR de manière aléatoire, et un ABR de manière croissant afin d'obtenir des valeurs moyennes pour chacun de ses arbres. Car il est possible que nos temps soient biaisés par un processus externe à cet algorithme. Pour stocker les résultats, nous allons utiliser les ArrayList.

1. aleaTime - Stocke les temps pour la construction d'un ABR aléatoire.
2. worstCase - Stocke les temps pour la construction d'un ABR croissant.
3. searchInTimeAlea - Stocke les temps pour la recherche de clés présentes dans un ABR aléatoire.
4. searchOutTimeAlea - Stocke les temps pour la recherche de clés absentes dans un ABR aléatoire.
5. searchInWorstCase - Stocke les temps pour la recherche de clés présentes dans un ABR croissant.
6. searchOutWorstCase - Stocke les temps pour la recherche de clés absentes dans un ABR croissant.

Ensuite on boucle reload fois. L'entier reload correspond au nombre de fois que l'on refait les opérations pour en calculer les moyennes en terme de temps. À chaque reload, on créer un arbre d'abord de treeSize éléments jusqu'à treeSize*10 éléments. Le pas est de treeSize. Ainsi, nous réalisons toujours des arbres de taille équivalente.

Pour réaliser un arbre aléatoire, on place la racine à treeSize/2. C'est la valeur médiane de notre génération de nombre aléatoire. Ensuite on génère un nombre aléatoire entre 0 et treeSize pour chaque éléments ajoutés. On ajoute cet élément dans une ArrayList qui va nous servir plus tard pour rechercher N clés présentes dans l'ABR. Une fois les Arbres de N éléments réalisés, on récupère le temps mis par l'algortihme à l'aide de System.currentTimeMillis et on le stocke dans aleaTime.

Pour réaliser un arbre croissant, rien de plus simple. On boucle sur 0 à N éléments, on ajoute l'itérateur k à l'ABR (une autre instance que l'ABR aléatoire) puis on ajoute k à une liste qui pareil va nous servir plus tard pour rechercher N clés présentes dans l'ABR croissant. Une fois les Arbres

de N éléments réalisés, on récupère le temps mis par l'algortihme à l'aide de System.currentTimeMillis et on le stock dans worstCase.

Pour rechercher N clés présentes dans l'ABR aléatoire, on parcourt la liste dans laquelle les clés on étaient ajoutées au préalable. On utilise la fonction rechercher() pour chacune d'elle. Puis on récupère le temps, et on le stocke dans searchInAleaTime. On fait de même avec l'ABR croissant mais on stocke cette fois-ci le temps dans searchInWorstCase.

Pour rechercher N clés absentes, il faut sortir de la portée de l'ABR aléatoire. Pour cela on va rechercher des clés supérieur à sa taille. On va rechercher une clé équivalente à treeSize + 1 + k où k correspond à l'itérateur de la boucle. On récupère le temps et on le stocke dans serachOutAleaTime. On fait pareil pour l'ABR croissant mais on va stocker les temps dans searchOutWorstCase.

Maintenant que nous avons nos données, nous récupérons pour chaque liste le nombre moyen pour chaque Nbéléments (treeSize * N). Ensuite à l'aide de la librairie XChart (petit remercie à Lucas LEMARCHAND pour m'avoir parlé de cette librairie incroyable), on obtient par pas de 1000 éléments les courbes suivantes.

B. Complexités des courbes :

Maintenant que nous avons visuellement nos données, calculons la complexité de:

1. Construction d'un ABR aléatoire (en moyenne).
2. Construction d'un ABR croissant.
3. Recherche de clé présente dans un ABR aléatoire (en moyenne).
4. Recherche de clé absente dans un ABR aléatoire (en moyenne).
5. Recherche de clé présente dans un ABR croissant.
6. Recherche de clé absente dans un ABR croissant.

1. Complexité de la construction d'un ABR aléatoire et croissant :

La complexité de la méthode add() revient au parcour que fait l'algorithme:

```
while (courrant != null){
    tampon = courrant;
    courrant = this.cmp.compare(noeudtoAppend.cle, courrant.cle) < 0 ? courrant.gauche : courrant.droit;
}
```

En moyenne, la complexité de ce parcour équivaut à O(log(n)). où n équivaut au nombre d'élément à insérer. Le reste des instructions équivalent toutes O(1).

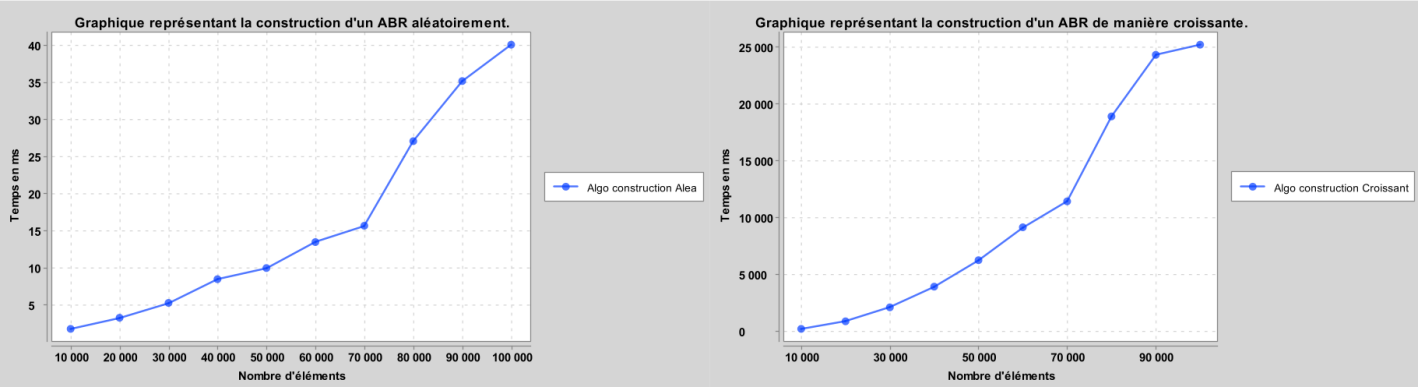
Donc la complexité dans le pire des cas et dans chaque cas, et parce que l'aléatoire en moyenne génère des arbres équilibrés, est de O(log(n)). La complexité dans le pire des cas, est O(n) car on est obligé de parcourir tout l'arbre pour ajouter un noeud.

2. Complexité de la recherche de clé présente dans un ABR aléatoire et croissant :

Pour la méthode rechercher(), la complexité dépend de la hauteur de l'ABR. Le fait de savoir si la clé est présente ou non n'influe pas sur la complexité de la méthode. De même la complexité dans un ABR aléatoire équivaut à O(log(n)) car cela dépend de la hauteur de l'ABR. Sinon la complexité vaut O(n) pour un arbre dégénéré car on doit parcourir n noeud dans le pire des cas.

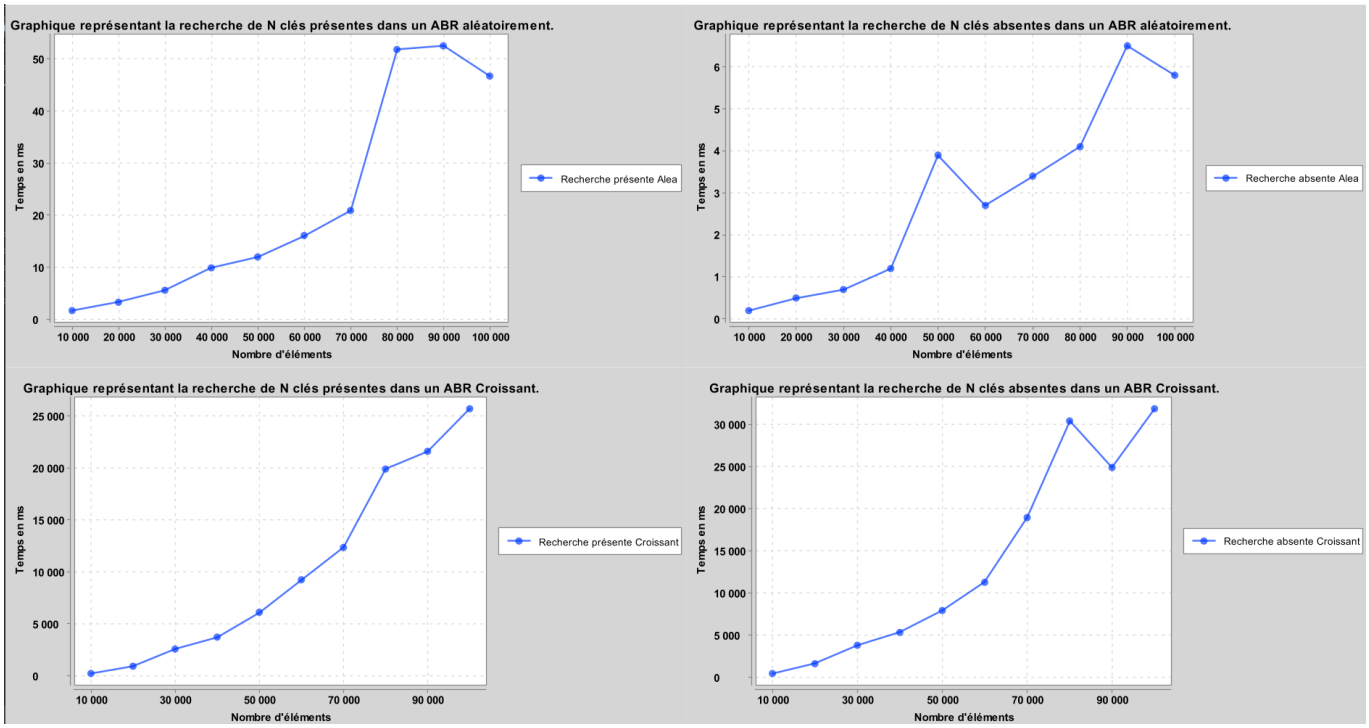
C. Interprétations des résultats :

Voici une visualisation graphique des temps de constructions des Arbres Binaire de Recherche en fonction du nombre d'éléments qui compose l'Arbre:



Sur le graphique de gauche, on remarque que l'échelle du temps est majorée à droite par]0, 40]ms. Sur le graphique de droite, on remarque que l'échelle du temps est majorée à droite par]0, 25 000]ms. La construction d'un ABR aléatoire est nettement plus rapide que la construction d'un ABR croissant.

Voici une visualisation graphique des temps des recherches dans des Arbres Binaire de Recherche en fonction du nombre d'éléments qui compose l'Arbre:



I

D. Conclusion :

Nous avons vus comment implémenter les Arbres Binaire de Recherche en Java. Nous avons étudiés le temps de construction d'Arbres Binaire de Recherche plus ou moins équilibrés, étudiés le temps de construction d'Arbres Binaire de Recherche désordonnés, étudiés la recherche d'éléments présents comme absents pour chaque cas. Nous avons calculés pour l'ajout d'un élément (construction d'un ABR) la complexité. On a vu qu'elle était équivalente à O(log(n)). Tout comme la recherche. Nous avons comparer les résultats moyens pour les deux différents Arbres. Ainsi nous avons conclus que l'ABR aléatoire était nettement plus rapide tant dans sa complexité que dans ses résultats. Dorénavant, nous allons étudiés une structure de données qui corrige à l'écriture (add() et supprimer()) l'emplacement de ses noeuds grâce au coloriage. On les nomme les Arbres Rouge et Noir.

Les Arbres Rouges et Noirs (ARN)

L'ARN ou d'arbre bicolore ou encore Arbre Rouge et Noir est un type d'Arbre Binaire de Recherche équilibré. Chaque nœud de l'Arbre, possède un attribut 'couleur' (rouge ou noir). Cette propriété permet de garantir que l'arbre est équilibré : lorsque des éléments sont insérés ou supprimés, certaines propriétés concernant la relation entre les nœuds et leurs couleurs doivent être conservées, ce qui évite que l'arbre ne devienne trop déséquilibré, même dans le pire des cas.

Voici les propriétés de l'ARN qui préviennent le déséquilibre :

1. Chaque noeud est rouge ou noir.

2. La racine est toujours un noeud noir.

3. Les feuilles sont toujours noires.

4. Si un noeud est rouge, alors ses noeuds enfants sont noirs.

5. Pour chaque noeud, si on emprunte n'importe quel chemin jusqu'à une feuille, on peut compter pour chaque chemin le même nombre de noeud(s) noir(s).

I. Implémentation en Java :

Comme nous avons expliqués en détail le fonctionnement, l'implémentation en JAVA de nos Arbres Binaire de Recherche, et comme le fonctionnement des ARN et similaire excepté la gestion de correction des noeuds, cette partie sera plus courte que la précédente. Pour implémenter nos ARN, nous aimerions qu'ils acceptent également des données de n'importe quel type comparable. Cette classe implémentera donc aussi la classe AbstractCollection<E>.

A. La classe noeud :

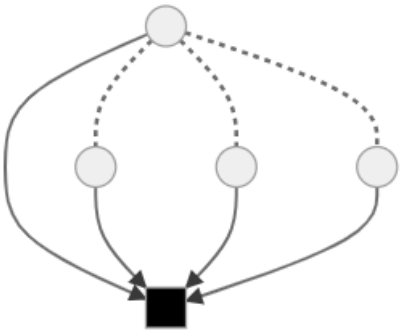
Cette sous-classe est strictement la même que celle des ABR à un détail près, nous rajoutons un paramètre à la sous-classe noeud nommé 'couleur'. Afin de gérer la couleur du noeud pour la correction de l'Arbre.

Voici un petit aperçu de notre sous-class noeud :

```
class Noeud {
  E cle;
  Noeud gauche;
  Noeud droit;
  Noeud pere;
  char couleur;

  private Noeud(E cle) {
    this.cle = cle;
    gauche = null;
    droit = null;
    pere = null;
    couleur = 'R';
  }

  Noeud minimum() {
    Noeud courrant = this;
    while (courrant.gauche != sentinelle) {
      courrant = courrant.gauche;
    }
    return courrant;
  }

  Noeud minimum(Noeud courrant) {
    while (courrant.gauche != sentinelle) {
      courrant = courrant.gauche;
    }
    return courrant;
  }

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

B. Les constructeurs de la classe ArbreBicolore :

Pour la classe Bicolore, on ajoute comme paramètre d'instance de la classe un noeud sentinelle, le reste des paramètres d'instance de classe et les constructeurs ne change pas. C'est un noeud vide qui se pointer par toutes les feuilles. de cette façon:



Tous les fils qui pointerait sur null dans un ABR pointeront sur un noeud sentinelle dans un ARN.

Aperçu de la création d'un noeud sentinelle avec la méthode sentinelle() :

```
public Noeud sentinelle() {
  Noeud sentinelle = new Noeud(null);
  sentinelle.gauche = null;
  sentinelle.droit = null;
  sentinelle.couleur = 'N';
  return sentinelle;
}
```

On n'affichera pas les noeuds sentinelles dans le toString().

Lorsqu'on créer un ARN vide la racine est donc une sentinelle. Un ARN ne comportant que la racine a comme enfant gauche sentinelle et enfant droit sentinelle etc...

C. L'Itérator des ARN :

Pour la sous-classe ARNIterator, seules les méthodes hasNext() et Next() sont modifiés par rapport à l'Itérator des ABR. En effet dans hasNext() des ABR on vérifiait si le noeud suivant de courrant était différent de null. Maintenant, dans ARNIterator on vérifie avec sentinelle. On fait de même pour next().

D. Ajouter un noeud dans un ARN :

Nous pouvons remarquer que la méthode add() de la classe ArbreBicolore semble en tout point être similaire à celle de la classe ArbreBinaireDeRecherche... Sauf l'avant dernière ligne ! En effet, dans un ARN on ajoute un noeud comme dans un ABR sauf qu'une fois placé, on va corrigeait sa position afin de rendre son placement plus optimisé pour les opérations que nous allons faire par la suite. Forcément nous devrons garder les propriétés fonctionnelles dans l'ARN. Pour cela nous appelons ajoutercorrection sur le noeud que nous voulons ajouter.

Explication de la méthode ajouterCorrection() :

La méthode ajouterCorrection() est appelée pour garder les propriétés d'un ARN après l'ajout d'un noeud. En effet, l'ajout d'un noeud dans un ARN peut violer les propriétés citées plus haut.

On commence par vérifier si le père du nouveau noeud est rouge. Si c'est le cas, cela veut dire que nous avons deux noeuds rouges consécutifs, ce qui est interdit dans un ARN. On va utiliser une combinaison de 'recoloration de noeuds' et de rotations d'arbre afin de corriger les propriétés violées.

Il y a trois cas possibles :

1. Si l'oncle du noeud qu'on veut corriger est rouge, alors on change la couleur du père de notre noeud et de l'oncle en noir. La couleur du grand père devient rouge. Ensuite, on continue la correction en remontant dans l'arbre, en commençant par le grand père. Enfin lorsqu'on retombe sur la racine la méthode se finit.

2. Si l'oncle du noeud qu'on veut corriger est noir et que le noeud est un enfant droit, alors on effectue une rotation à gauche sur le père. Cela transforme la configuration de l'arbre en une configuration qui correspond au cas 3.

3. Si l'oncle du noeud qu'on veut corriger est noir et que le noeud est un enfant gauche, alors on change la couleur du père en noir, la couleur du grand père en rouge, et on effectue une rotation à droite sur le grand père.

Explication des rotations droites & gauches :

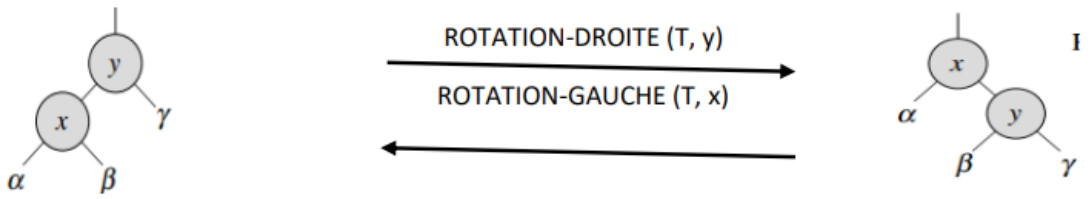


Figure 7 : Rotations sur un arbre binaire de recherche

RotationGauche(noeud courrant) :

- On crée une variable tampon pour stocker le noeud droit du noeud courant.
- On fait pointer le noeud droit du noeud courant vers le noeud gauche du noeud tampon.
- Si le noeud gauche du noeud tampon n'est pas une sentinelle, on fait pointer son père vers le noeud courant.
- On fait pointer le père du noeud tampon vers le père du noeud courant.
- Si le père du noeud courant est une sentinelle, on fait de tampon la nouvelle racine de l'arbre.

- Si le noeud courant est le fils gauche de son père, on fait pointer le fils gauche du père du noeud courant vers tampon. Sinon, on fait pointer le fils droit du père du noeud courant vers tampon.
- On fait pointer le noeud gauche du noeud tampon vers le noeud courant.
- On fait pointer le père du noeud courant vers le noeud tampon.

RotationGauche(noeud courrant) :

Cette méthode est très similaire à la méthode de rotation à gauche, mais elle effectue une rotation dans la direction opposée. Elle utilise le noeud gauche du noeud courant au lieu du noeud droit, et elle met à jour les pointeurs en conséquence. La logique est la même que celle de la rotation à gauche, mais inversée.

Ces corrections sont appliquées jusqu'à ce que les propriétés de l'ARN soient respectées. À la fin de la méthode, on s'assure que la racine de l'arbre est noire, pour respecter le propriété de l'ARN qui stipule que la racine d'un ARN doit être noire.

ajouterCorrection() garantit que l'arbre reste équilibré, ce qui permet d'optimiser les opérations de recherche, d'insertion et de suppression. C'est cette optimisation qui différencie les ARN des ABR. On va voir cela en illustration plus tard.

E. Recherche une valeur dans un ARN :

L'algo de recherche est une copie de l'algo de recherche de la classe ABR. La seule différence est la comparaison entre le noeud courrant avec le noeud sentinelle dans la boucle while.

Comme une image vaut mille mots :

```
public Noeud rechercher(Object o) {
  Noeud courrant = racine;
  while (courrant != sentinelle && courrant.cle != (E) o) {
    courrant = cmp.compare(courrant.cle, (E) o) > 0 ? courrant.gauche : courrant.droit;
  }
  return courrant;
}
```

F. Supprimer une valeur dans un ARN :

Comme avec la fonction add(), la fonction supprimer de la classe ARN n'a qu'une ligne de différence avec la méthode supprimer de la classe ABR. Une fois la valeur supprimée il se peut que des propriétés de l'ARN soient violées. Donc nous allons corriger l'ARN pour garder l'équilibre de l'ARN et le respect des propriétés.

Explication de la méthode supprimer() :

• On entre dans une boucle tant que le noeud courant n'est pas la racine et que sa couleur est noire.

• Si le noeud courant est le fils gauche de son père :

- On crée une variable brother pour stocker le frère du noeud courant.
- Si la couleur du frère est rouge (cas 1) :
  - On change la couleur du frère en noir et celle du père en rouge.
  - On effectue une rotation à gauche sur le père.
  - On met à jour la variable brother.
- Si les deux fils du frère sont noirs (cas 2) :
  - On change la couleur du frère en rouge.
  - On met à jour le noeud courant pour qu'il pointe vers son père.
- Sinon, si le fils droit du frère est noir (cas 3) :
  - On change la couleur du fils gauche du frère en noir et celle du frère en rouge.
  - On effectue une rotation à droite sur le frère.
  - On met à jour la variable brother.
- Enfin, dans le cas 4 :
  - On change la couleur du frère pour qu'elle soit celle du père du noeud courant.
  - On change la couleur du père du noeud courant et celle du fils droit du frère en noir.
  - On effectue une rotation à gauche sur le père du noeud courant.
  - On fait pointer le noeud courant vers la racine.

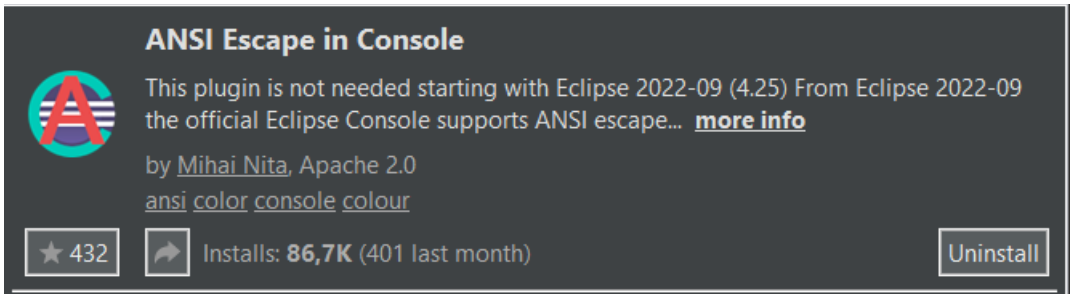
• Si le noeud courant est le fils droit de son père, on effectue les mêmes opérations mais en inversant gauche et droite.

• À la fin, on change la couleur du noeud courant en noir.

E. Afficher un ARN :

Le toString() de la classe ARN est exactement comme le toString() de la classe ABR. Seules différences on ne compare plus le noeud courrant avec null mais avec sentinelle. On commence à en avoir l'habitude. De plus quand un noeud doit être rouge, on affiche le code ANSI "\u001B[31m" + la clé du noeud courrant + "\u001B[0m".

ATTENTION : Sur eclipse, il faudra installer le plugin suivant :

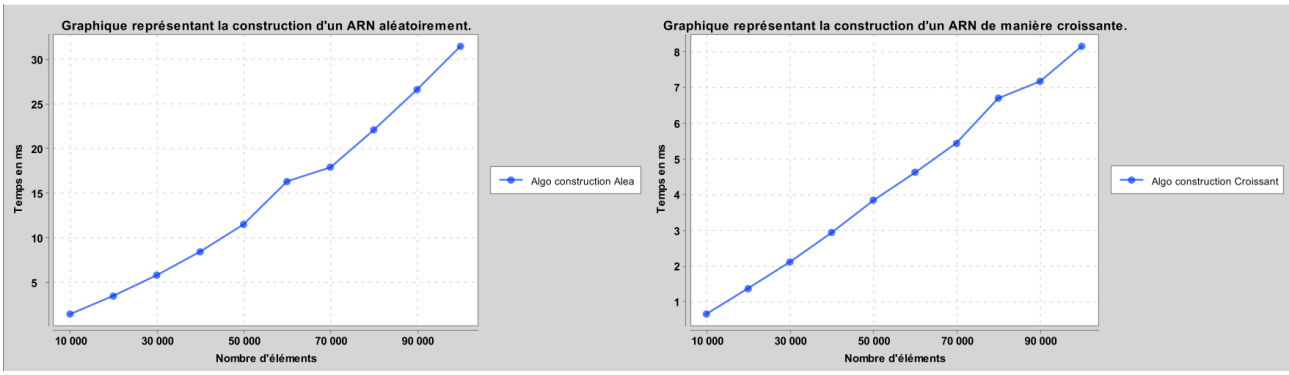


Pour se faire, rendez-vous dans Help -> Eclipse marketplace -> Rechercher "ANSI Escape in Console" et télécharger-le. N'oubliez pas de relancer eclipse.

II. Etude de la complexité et collecte des données pour la classe ArbreBicolore :

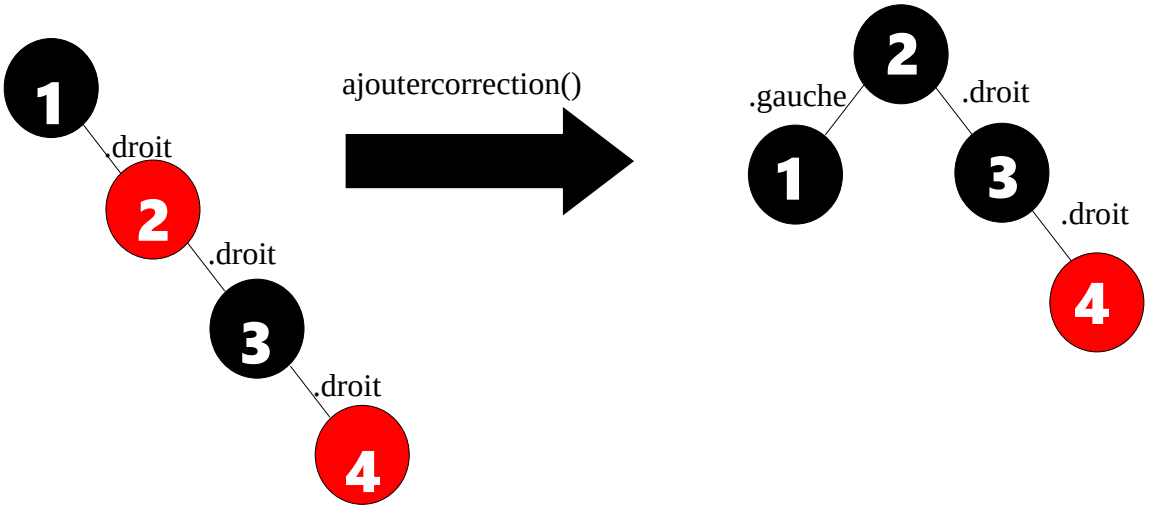
A. Temps & Graphiques & Complexité des Constructions des ARN croissants et aléatoires

Pour la collecte des données, on utilise la même méthode mais on construit des Arbres Bicolores à la place des Arbres Binaire de Recherche. Donc nous avons totalement copié collé la méthode buildDataABR() pour construire buildDataARN() en changeant juste les classes des Arbres. (Pour pour le refactoring on verra ça plus tard). Ainsi on obtient pour la construction ces courbes suivantes:



En regardant ces courbes, quelque chose devrait vous choquer. En effet, un arbre dégénéré est plus rapide qu'un arbre Aléatoire. C'est devenu complètement contradictoire avec ce qu'on disait dans la complexité des Arbres Binaire de Recherche. Mais c'est normal. En effet à chaque fois qu'on va placer un noeud dans un ARN croissant on va le disposer à la meilleur place possible.

Voici une petite illustration de ce phénomène :



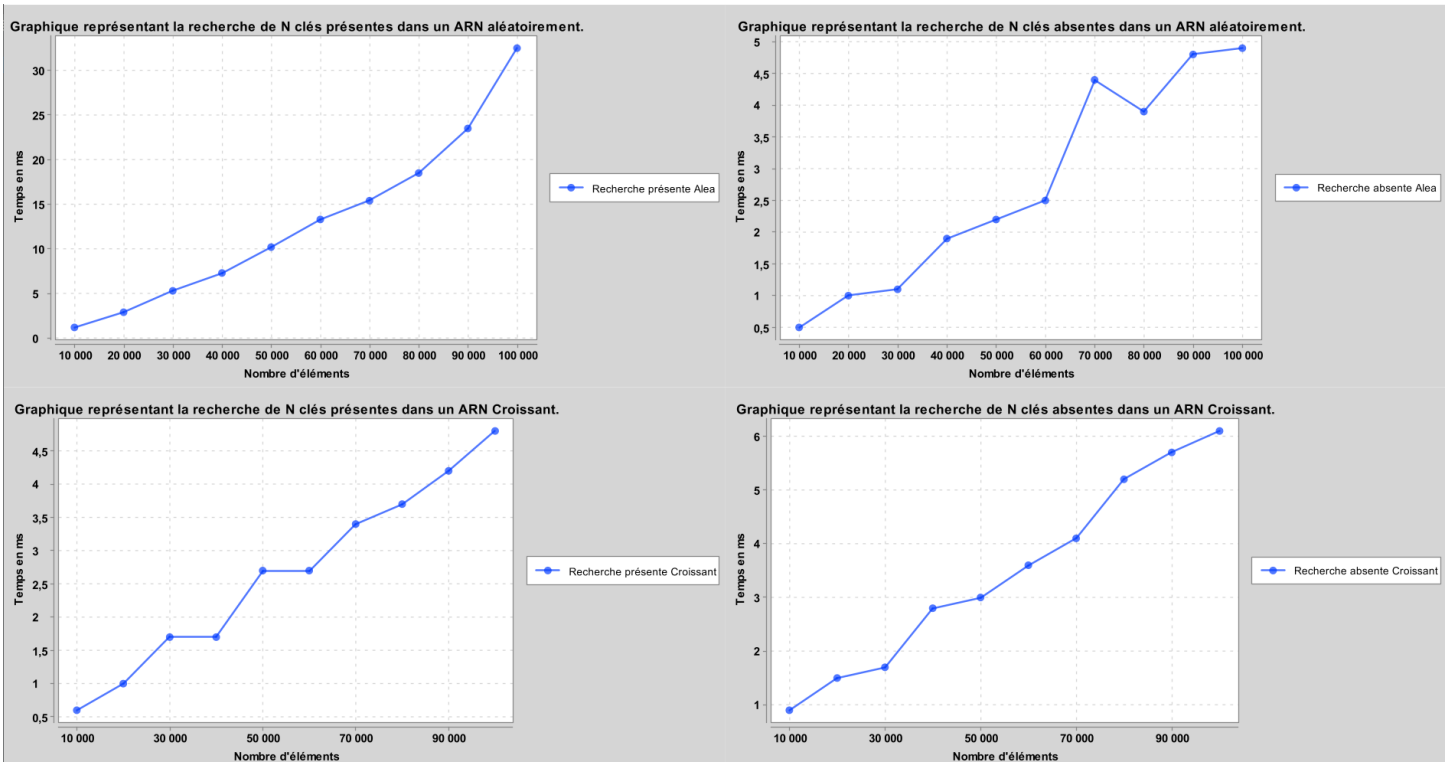
Lorsque nous insérons des données aléatoires dans un ARN, chaque insertion peut nécessiter une restructuration de l’arbre pour maintenir ses propriétés d’équilibrage. Ces restructurations, qui peuvent inclure des rotations et des changements de couleur, prennent du temps. De plus, la recherche du point d’insertion approprié dans un arbre non trié peut également prendre plus de temps. Alors que, lorsqu'on insére des données dans un ordre croissant, chaque nouvelle donnée est ajoutée à la fin de l’arbre. Cela signifie qu’il y a moins de chances que l’arbre ait besoin d’être restructuré, ce qui rend le processus plus rapide.

Construire un noeud possède une complexité logarithmique. En effet, il s'agit de la même complexité que celle des ABR. Mais nous ne sommes jamais dans des cas aussi mal "construits" que ceux des ABR. La complexité joue un rôle majeur mais l'optimisation des structures de données également.

Passons maintenant à la recherche de N clés présentes / Absentes.

B.Temps & Graphiques des Temps pour la recherche de N clés présentes et absentes dans un ARN aléatoire et croissant.

Voici les graphiques obtenus pour la recherche de N clés présentes / absentes dans un ARN :



Sur ces graphiques on se rend compte que la recherche de N clés présentes dans un ARN aléatoire et nettement plus longue que celle des clés absentes. Cela devrait être l'inverse. Lorsque la clé est trouvée, le programme de recherche s'arrête. Alors que la recherche absente devra obligatoirement parcourir toute la branche la plus à droite afin d'arrêter la recherche.

Conclusion Finale :

Pour manipuler efficacement les données, il faut des outils essentiels tels que les arbres de recherche binaires et les arbres rouges et noirs. Dans le monde de l'informatique, les Arbre Binaire de Recherche sont très populaires en raison de leur capacité à effectuer des opérations de recherche, d'insertion et de suppression en O(logn). Il faut cependant noter que cette efficacité peut parfois se détériorer et devenir O(n).

Lorsque les données sont insérées dans un ordre spécifique, tel que croissant ou décroissant, il existe une possibilité que ce soit le pire des cas. Après chaque insertion ou suppression, les ARN maintiennent l'équilibre de l'arborescence grâce à l'ajout de colorations de nœuds et d'application de propriétés, conduisant finalement à une complexité de O(logn) et à la résolution de ce problème. Les pires scénarios n’ont aucune chance contre les opérations de recherche, d’insertion et de suppression.

Dans les applications qui nécessitent des performances élevées et des données immenses, les ARN sont préférables aux ABR, mais leur mise en œuvre est plus complexe. Décider d'utiliser un ABR ou un ARN est une question d'identification des besoins particuliers de la tâche. Il est donc essentiel de comprendre ces structures de données et leurs compromis afin de prendre une décision éclairée.