TP Arbre Rouge-Noir

Ce TP a pour finalité de comparer la structure de l'arbre rouge-noir (ou arbre équilibré) avec celle de l'arbre de binaire de recherche pour déterminer si l'arbre équilibré a un intéret.

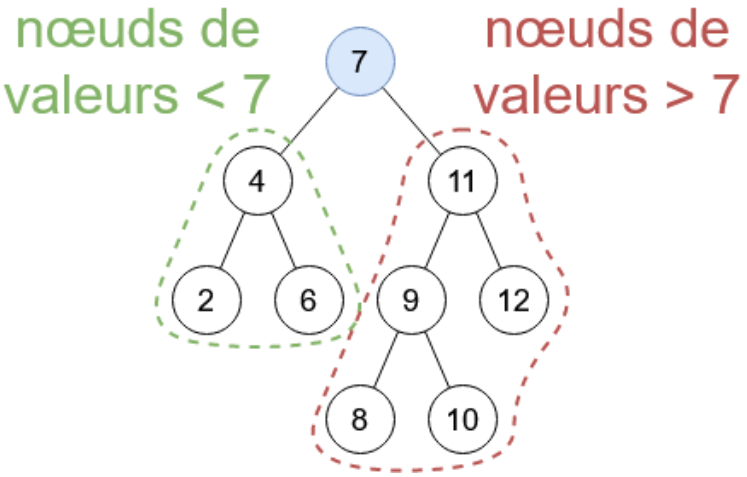
Dans ce document j'explique d'abord le concept de l'arbre rouge-noir puis mon code, ensuite je compare son utilisation et sa complexité à d'autres structures de données, puis enfin pour être sûr que mon code est juste je compare sa vitesse d'exécution avec celle de l'arbre binaire de recherche que l'on a codé au TP précédent.

▼ Qu'est-ce qu'un Arbre Rouge-Noir

Les arbres rouges et noirs sont une variante modifiée des arbres binaires de recherche.

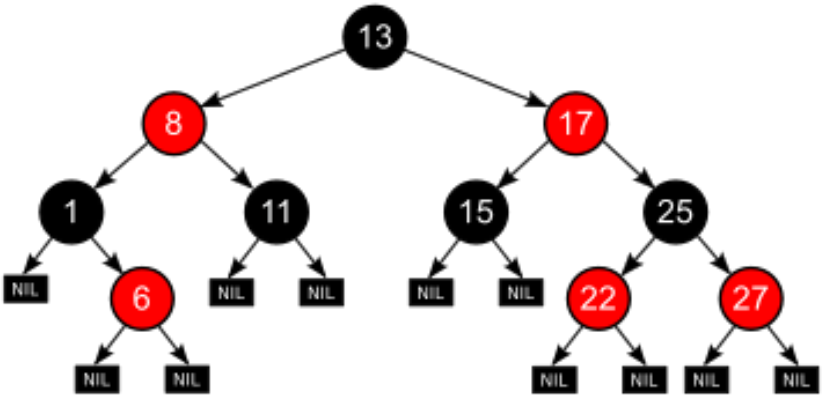
Pour rappel les propriétés de l'arbre binaire de recherche sont les suivantes :

- Chaque Noeud possède une clé.
- Tous les nœuds du sous-arbre gauche ont une clé inférieure à celle du nœud en question, et tous les nœuds du sous-arbre droit ont une clé supérieure.
- Chaque nœud a au plus deux enfants : un nœud gauche et un nœud droit.



L'idée principale de l'arbre rouge-noir est de maintenir un équilibre entre les branches de l'arbre, pour ce faire l'arbre possède des propriétés supplémentaires à celle de l'arbre binaire de recherche.

- Chaque nœud dans l'arbre est coloré, soit rouge soit noir.
- La racine de l'arbre est forcément noire.
- Chaque feuille est représentée par un noeud null et est noire.
- Si un nœud est rouge, alors ses deux enfants sont noirs.
- Tout chemin de la racine à une feuille doit avoir le même nombre de nœuds noirs. Cela garantit que l'arbre est équilibré et évite les chemins beaucoup plus longs que d'autres.



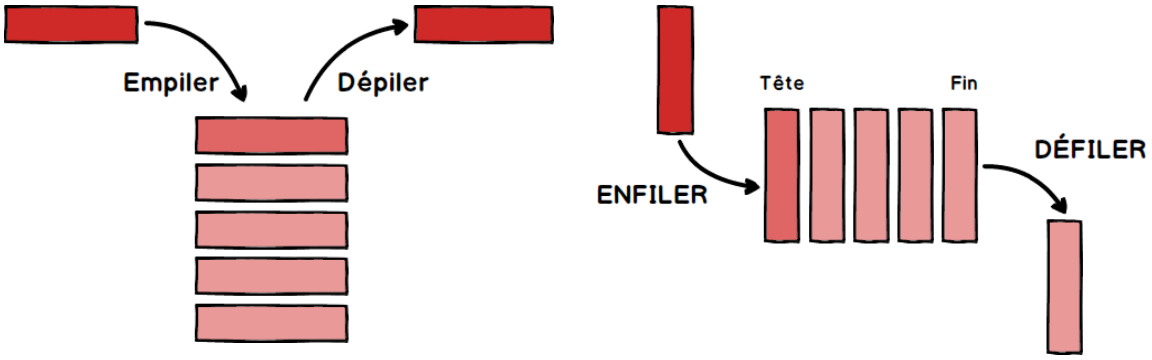
L'ajout ou la suppression d'un nœud peut violer les propriétés de couleur, mais l'arbre est rééquilibré à chaque opération pour maintenir ces propriétés. Les rotations et les changements de couleurs sont utilisés pour rétablir l'équilibre.

Comparaison avec d’autres structures de données

On peut comparer très simplement la structure de l’arbre rouge-noir avec d’autres structures de données de base, ici j’ai choisi de le faire avec la pile, la file et la liste chainée.

La pile et la file

La pile utilise le principe de “dernier arrivé, premier sorti” (LIFO) et la file utilise le principe de “premier arrivé,premier sorti” (FIFO). C’est-à-dire que dans les faits, on n’a accès qu’à une seule valeur de la structure à la fois, le haut de la pile ou le début de la file.



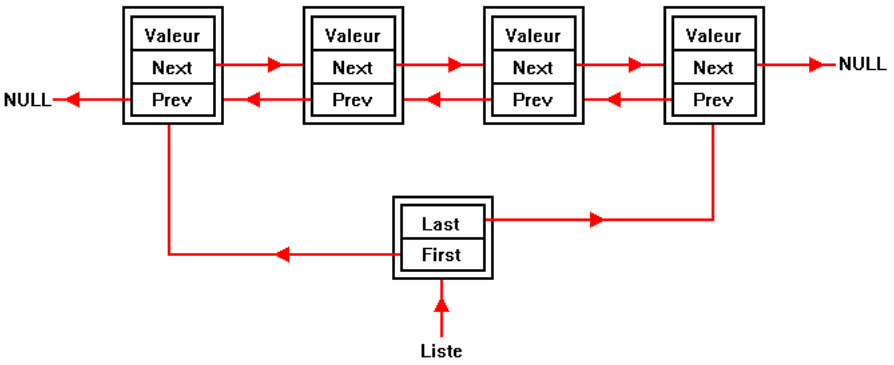
On sera donc obligé de dépiler/défiler à chaque fois que l’on veut supprimer, déplacer ou ajouter au milieu une donnée, et pour rechercher un élément on sera forcément obligé de traverser une partie voire la totalité de la structure, ce qui ralentit considérablement la manipulation de données.

La complexité pour une manipulation de donnée que ce soit dans la pile ou la file sera dans tous les cas O(n), où n est le nombre de données à traiter, dans le pire des cas ça sera le tableau entier.

La Liste Chainée

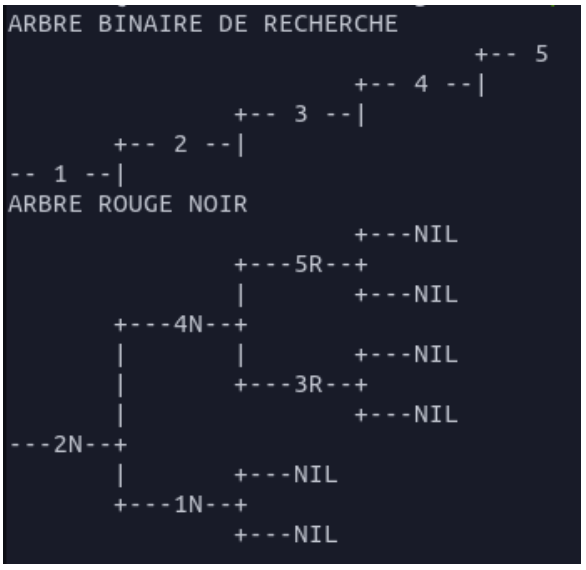
La liste chainée donne accès à la première et/ou à la dernière valeur contenue dans les noeuds de tête et de queue. Les noeuds connaissent le noeud précédent et/ou le noeud suivant ce qui permet un ajout en milieu de liste bien plus aisé. Supprimer et rechercher une valeur reste complexe dans le sens où on doit toujours traverser une partie voire la totalité de la liste pour retrouver la donnée que l’ont veut.

La complexité sera plutôt de O(1) dans les meilleurs cas où on doit ajouter, supprimer ou rechercher la tête ou la queue. Elle sera de O(n) dans les autres cas.



▼ Arbre Binaire de recherche et Arbre équilibré

L'arbre binaire de recherche et l'arbre équilibré ont la même structure de base qui a été décrite dans la première partie de ce document. En remplissant les 2 avec les même clés on voit tout de suite la différence dans le traitement :



> Après l'exécution de MainTest.java sur le tableau {1,2,3,4,5} on voit bien que l'arbre binaire n'a que des fils droits comme tous les chiffres ajoutés après la racine lui sont supérieurs alors que l'arbre rouge-noir est équilibré

En calculant la complexité on peut alors déterminer le tableau suivant :

ABR		Au pire des cas	Au cas moyen	Au meilleur des cas
	Ajouter un Noeud	O(n)	O(log n)	O(1)
	Rechercher un Noeud	O(n)	O(log n)	O(1)
	Supprimer un Noeud	O(n)	O(log n)	O(1)
ANR		Au pire des cas	Au cas moyen	Au meilleur des cas
	Ajouter un Noeud	O(log n)	O(log n)	O(1)
	Rechercher un Noeud	O(log n)	O(log n)	O(1)
	Supprimer un Noeud	O(log n)	O(log n)	O(1)

On doit maintenant tester la vitesse d'exécution de notre code pour voir si celui-ci est bien plus rapide que celui de l'arbre binaire du TP précédent.

▼ Implantation de ANR

J'ai décidé de ne pas partir d'un héritage de ABR mais directement de AbstractCollection puisqu'il y a de toute façon beaucoup de méthodes à réécrire.

A tous les endroits où on vérifiait si un noeud était feuille dans ABR(donc qu'il était null) on vérifie maintenant que c'est la sentinelle.

▼ Variables de la Classe ANR

```
private Noeud racine;
private final Noeud sentinelle = new Noeud();
private int taille;
private Comparator<? super E> cmp;
```

J'ai rajouté une constante représentant la sentinelle, le noeud feuille noir avec une clé nulle.

On utilise la même sentinelle pour toutes les feuilles pour gagner de la mémoire.

▼ Code de la classe Noeud

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

    Noeud (){
      cle = null;
      gauche = droit = pere = null;
      couleur = 'N';
    }

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

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

    public Noeud suivant() {
      if (droit != sentinelle) return droit.minimum();
      else {
        Noeud p = pere;
        Noeud courant = this;
        while (p != sentinelle && courant == p.droit) {
          courant = p;
          p = p.pere;
        }
        return p;
      }
    }

    public char getCouleur() {
      return couleur;
    }

    public E getCle() {return cle;}

    public Noeud getGauche() {
      return gauche;
    }

    public Noeud getDroite() {
      return droit;
    }

    public Noeud getPere() {
      return pere;
    }

    public void setPere(Noeud pere2) {
      this.pere = pere2;
```

```
    }

    public void setGauche(Noeud x) {
      this.gauche = x;

    }

    public void setDroite(Noeud x) {
      this.droit = x;

    }

    private void setCouleur(char c)  {
      if(c=='N' || c=='R') couleur = c;
      else System.out.println("Cette couleur n'est pas autorisée");
    }

  }
```

J’ai ajouté la couleur sous forme d’un caractère, soit ‘R’ pour rouge, soit ‘N’ pour noir.

On a aussi ajouté un constructeur sans argument pour la sentinelle qui fait un noeud noir, avec une clé nulle, sans enfants et sans parent.

On part du principe que quand on crée un noeud avec une clé alors il ne sera pas null. Ce noeud est créé rouge et une fois dans l’arbre on appliquera des corrections pour garder les propriétés de l’arbre.
Les setters et getters seront utilisés tout au long du code, c’est un choix personnel car je trouve ça plus lisible.

▼ Constructeurs de ANR

```
public ANR() {
    racine = sentinelle;
    taille = 0;
    cmp = ((e1,e2) -> ((Comparable<E>) e1).compareTo(e2));
  }
  public ANR(Comparator<E> cmp) {
    racine = sentinelle;
    taille = 0;
    this.cmp = cmp;
  }

  public ANR(Collection<? extends E> c) {
    this();
    this.addAll(c);
  }
```

Les arbres vides ont maintenant la sentinelle comme racine.

▼ ToString de ANR

```
public String toString() {
      StringBuilder buf = new StringBuilder();
      toString(racine, buf, "", maxStrLen(racine), true);
      return buf.toString();
  }

  private void toString(Noeud x, StringBuilder buf, String path, int len, boolean isRight) {
      if (x == sentinelle) {
          buf.append(getIndent(path, len, isRight) + "---NIL\n");
          return;
      }

      toString(x.droit, buf, path + "D", len, true);
      buf.append(getIndent(path, len, isRight) + "---" + x.cle.toString() + x.getCouleur() + "--+\n");
      toString(x.gauche, buf, path + "G", len, false);
  }

  private String getIndent(String path, int len, boolean isRight) {
      StringBuilder indent = new StringBuilder();
      for (int i = 0; i < path.length(); i++) {
          for (int j = 0; j < len + 6; j++)
              indent.append(' ');

          char c = ' ';
          if (i == path.length() - 1)
              c = isRight ? '+' : '+';  // Correction ici, les deux côtés devraient être "|"
          else if (path.charAt(i) != path.charAt(i + 1))
              c = '|';
```

```
        indent.append(c);
      }
      return indent.toString();
  }
  private int maxStrLen(Noeud x) {
    return x.cle == null ? 0 : Math.max(x.cle.toString().length(),
        Math.max(maxStrLen(x.gauche), maxStrLen(x.droit)));
  }
```

Le toString() est modifié pour pouvoir afficher les noeuds feuilles, on écrit donc NIL pour les représenter. On écrit également le caractère qui représente la couleur à coté de la clé.

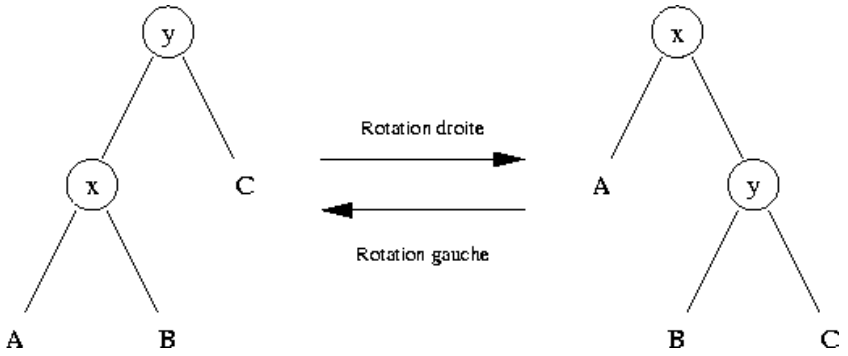
▼ Rotations

```
private void rotationGauche(Noeud x) {
    Noeud y = x.getDroite();
    x.setDroite(y.getGauche());
    if(y.getGauche()!=sentinelle) {
      y.getGauche().setPere(x);
    }
    y.setPere(x.getPere());
    if(x.getPere()==sentinelle) {
      racine = y;
    }else {
      if(x==x.getPere().getGauche()) {
        x.getPere().setGauche(y);
      }else {
        x.getPere().setDroite(y);
      }
    }
    y.setGauche(x);
    x.setPere(y);

  }

  public void rotationDroite(Noeud y) {
    Noeud x = y.getGauche();
    y.setGauche(x.getDroite());
    if(x.getDroite()!=sentinelle) {
      x.getDroite().setPere(y);
    }
    x.setPere(y.getPere());
    if(y.getPere()==sentinelle) {
      racine = x;
    }else {
      if(y==y.getPere().getDroite()) {
        y.getPere().setDroite(x);
      }else {
        y.getPere().setGauche(x);
      }
    }
    x.setDroite(y);
    y.setPere(x);
  }
```

2 nouvelles méthodes, les rotations ! Elle sont utilisées par corrigerAdd() et corrigerSuppr(). Le but étant de passer d'un état à l'autre pour garder les propriétés de l'arbre correctes.



▼ Ajout d'un noeud

```
public boolean add(E e) {
    if (e == null) {
      System.out.println("Ajout impossible, clé nulle");
      return false;
    }
    Noeud z = new Noeud(e);
    Noeud y = sentinelle;
    Noeud x = racine;

    while(x!=sentinelle) {
      y=x;
      x=(cmp.compare(z.getCle(), x.getCle())<0) ? x.getGauche() : x.getDroite();
    }
    z.setPere(y);
    if(y==sentinelle) {
      racine = z;
    }else {
      if(cmp.compare(z.getCle(),y.getCle())<0) {
        y.setGauche(z);
      }else {
        y.setDroite(z);
      }
    }
    z.setGauche(sentinelle);
    z.setDroite(sentinelle);
    corrigerAdd(z);
    return true;
  }

  private void corrigerAdd(Noeud z)  {
    //comme le père de la racine est fixée a null, ça ne sert a rien de regarder si son père est rouge, donc on ne rentre pas d
    //On en sortira également si jamais en remontant dans l'arbre on se retrouve sur la racine
    //Et la racine est toujours remise a noir après la boucle.

    while(z!=racine && z.getPere().getCouleur()=='R') {

      //Si le père de z est le fils gauche de son père.
      if(z.getPere()==z.getPere().getPere().getGauche()) {
        //y est l'oncle de z, on prend le fils droit du grand père de z
        Noeud y = z.getPere().getPere().getDroite();

        if(y.getCouleur()=='R') {
          //cas 1 l'oncle de z est ROUGE,
          //on doit mettre z, son père et son oncle en noir
          //et mettre le grand père de z en rouge pour continuer a respecter le nombre de noeuds noirs.

          z.getPere().setCouleur('N'); //le père de Z est remis en noir
          y.setCouleur('N'); // l'oncle de z est remis en noir
          z.getPere().getPere().setCouleur('R'); // le grand père de z est mis en rouge
          //ici on a bien corrigé pour avoir 2 noeuds noirs sous un noeud rouge !

          z=z.getPere().getPere();
          //on remonte le noeud courant au grand père de z
          //qui est rouge et on revérifiera au prochain passage de boucle si son père existe et n'est pas rouge
        } else {
          if(z==z.getPere().getDroite()) {
            //cas 2 si l'oncle de z est noir et z est le fils droit de son père
            //on remonte sur le père et on fait une rotation gauche
            z=z.getPere();
            rotationGauche(z);
            //z devient le fils gauche et on arrive au cas 3
          }
          //cas 3 : z est le fils gauche et son oncle est noir
          z.getPere().setCouleur('N');
          z.getPere().getPere().setCouleur('R');
          rotationDroite(z.getPere().getPere());
        }
      }else {
        //le père de Z est le fils droit de son père
        //on fait les même 3 cas mais en miroir.
        Noeud y = z.getPere().getPere().getGauche();
        //l'oncle de z est a gauche
        if(y.couleur=='R') {
          //cas 1 en mirroir
          z.getPere().setCouleur('N');
          y.setCouleur('N');
          z.getPere().getPere().setCouleur('R');
          z = z.getPere().getPere();
        }else {
          if(z==z.getPere().getGauche()) {
            //cas 2
            z=z.getPere();
            rotationDroite(z);
```

```
          }
          //cas 3
          z.getPere().setCouleur('N');
          z.getPere().getPere().setCouleur('R');
          rotationGauche(z.getPere().getPere());
        }
      }
    }
    racine.setCouleur('N');

  }
```

J'ai modifié add pour que tous les null soient mis à sentinelle mais aussi que les noeuds enfants du noeud ajouté en bout d'arbre soient aussi la sentinelle.
J'ai également ajouté l'appel à corrigerAdd() qui permet de corriger la coloration et la répartition de l'arbre pour garder toutes les propriétés après chaque ajout.

- A la fin on remet bien la racine en noir pour être sur de respecter toutes les contraintes.

▼ Supression d'un noeud

```
private Noeud supprimer(Noeud z)  {
    Noeud y; // nœud à détacher

      if (z.getGauche() == sentinelle || z.getDroite() == sentinelle) {
          y = z;
      } else {
          y = z.suivant();
      }

      // x est le fils unique de y ou null si y n'a pas de fils
      Noeud x = (y.getGauche() != sentinelle) ? y.getGauche() : y.getDroite();


      x.setPere(y.getPere());


      if (y.getPere() == null) { // suppression de la racine
          racine = x;
      } else {
          if (y == y.getPere().getGauche()) {
              y.getPere().setGauche(x);
          } else {
              y.getPere().setDroite(x);
          }
      }

      if (y != z) {
          z.cle = y.cle;
      }
      if(y.getCouleur()=='N') corrigerSuppr(x);

      return z.suivant();
  }

  private void corrigerSuppr(Noeud x)  {
    while(x!= racine && x.getCouleur()=='N') {
      if (x == x.getPere().getGauche()) {
            Noeud w = x.getPere().getDroite(); // le frère de x
            if (w.getCouleur() == 'R') {
              // cas 1
              w.setCouleur('N');
              x.getPere().setCouleur('R');
              rotationGauche(x.getPere());
              w = x.getPere().getDroite();
            }
            if (w.getGauche().getCouleur() == 'N' && w.getDroite().getCouleur() == 'N') {
              // cas 2
              w.setCouleur('R');
              x = x.getPere();
            } else {
              if (w.getDroite().getCouleur() == 'N') {
                // cas 3
                w.getGauche().setCouleur('N');
                w.setCouleur('R');
                rotationDroite(w);
                w = x.getPere().getDroite();
              }
              // cas 4
              w.setCouleur(x.getPere().getCouleur());
```

```
            x.getPere().setCouleur('N');
            w.getDroite().setCouleur('N');
            rotationGauche(x.getPere());
            x = racine;
          }
        } else {
          Noeud w = x.getPere().getGauche(); // le frère de x
          if (w.getCouleur() == 'R') {
            // cas 1
            w.setCouleur('N');
            x.getPere().setCouleur('R');
            rotationDroite(x.getPere());
            w = x.getPere().getGauche();
          }
          if (w.getGauche().getCouleur() == 'N' && w.getDroite().getCouleur() == 'N') {
            // cas 2
            w.setCouleur('R');
            x = x.getPere();
          } else {
            if (w.getGauche().getCouleur() == 'N') {
              // cas 3
              w.getDroite().setCouleur('N');
              w.setCouleur('R');
              rotationGauche(w);
              w = x.getPere().getGauche();
            }
            // cas 4
            w.setCouleur(x.getPere().getCouleur());
            x.getPere().setCouleur('N');
            w.getGauche().setCouleur('N');
            rotationDroite(x.getPere());
            x = racine;
        }
      }
  }
  x.setCouleur('N');
}
```

Dans supprimer j'ai aussi effectué quelques modifications, si le noeud détaché est noir on brise la règle du nombre de noeuds noirs par chemins, on doit donc corriger la coloration de l'arbre.

- A la fin on remet bien la racine en noir pour être sur de respecter toutes les contraintes.

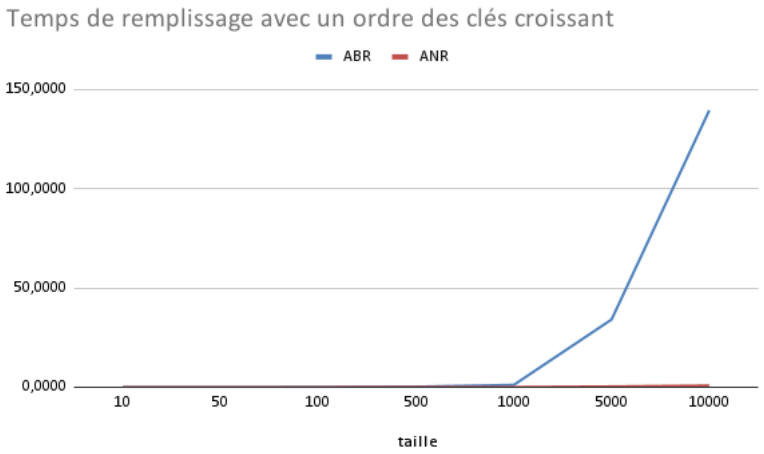
▼ Comparaison de temps d'exécution de ABR et ANR

J'ai ajouté une Classe FichierTemps afin de tester le temps d'exécution du remplissage de chaque type d'arbre, cette classe prend en paramètre un nom de fichier et un tableau et exécute 10000 fois le remplissage de chaque arbre de 2 façons différentes (faisant donc 40000 remplissages en tout), il remplit d'abord l'arbre binaire avec le tableau croissant, puis mélange le tableau pour le remplir de manière aléatoire, il fait ensuite de même pour l'arbre rouge-noir.
A chaque remplissage l'arbre est évidemment réinitialisé et il relève le temps écoulé pendant ce remplissage avant de l'écrire dans le fichier.

Une fois les fichiers créés j'ai pu tirer les graphiques suivants des données récupérées :

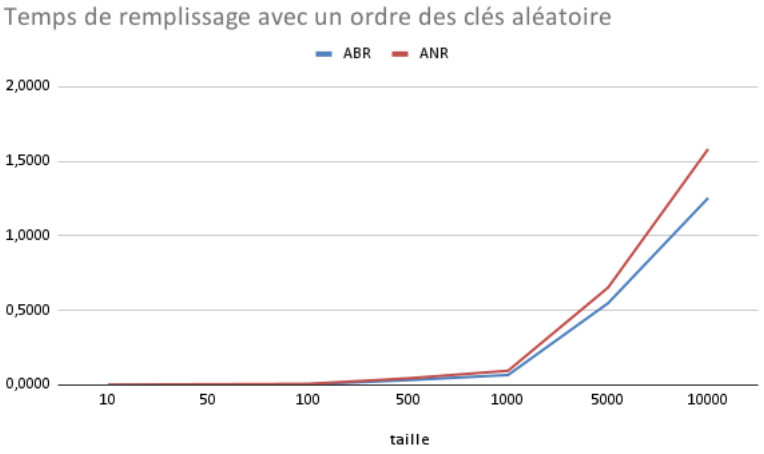
▼ Remplissage avec des clés croissantes :

> Le temps en ms est en ordonée et la taille du tableau utilisé pour le remplissage est en abscisse.
> Ce remplissage correspond au pire des cas où toutes les clés sont dans l'ordre et pour ABR on a toutes les valeurs qui seront mises à gauche, créant un arbre de plus en plus haut.
> On peut voir que le temps d'exécution de ABR est bien plus haut que ANR quand on a un grand nombre de valeurs, ce qui parait logique au vue de la différence des complexités pour ce cas.

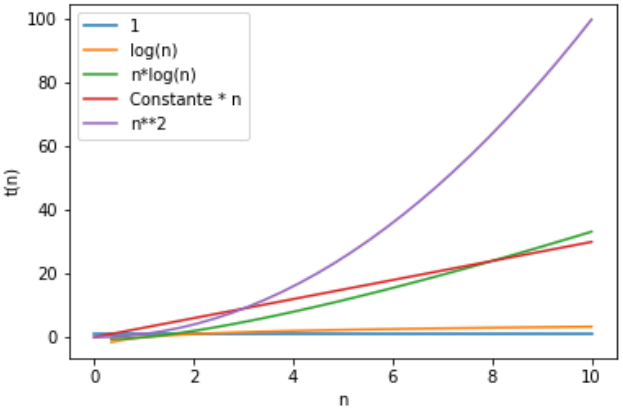


▼ Remplissage avec des clés croissantes :

> Le temps en ms est en ordonée et la taille du tableau utilisé pour le remplissage est en abscisse.
> Ce remplissage correspond au cas moyen où les clés ne sont pas triées. Dans ce cas ci les 2 remplissages sont très similaires ce qui correspond aux complexités de la partie précédente.



De plus si on compare ces courbes avec les courbes d'ordre de complexité on voit bien qu'elles correspondent à peu près aux complexités vues plus tôt : soit log n pour l'ANR dans tous les cas, log n pour l'ABR dans le cas moyen et n dans le pire des cas de l'ABR.



Avec tout cela on peut déterminer que dans tous les cas on aura plus intéret à utiliser l'arbre rouge-noir pour gérer une structure de donnée qui nécessite de manipuler un grand nombre de données et de pouvoir ajouter, rechercher et supprimer n'importe quelle données qui se trouvent dans la structure.