Faculté des Sciences et Ingénierie - Sorbonne université

Master Informatique parcours - STL / IMA

AlgAv - Algorithmique Avancée

Rapport de projet

Devoir de programmation

Réalisé par :
MAOUCHE Mounir - 21315128 - M1 IMA
BOUZOURINE Hichem - 21319982 - M1 STL

Supervisé par :
Antoine Genitrini

11 Décembre 2023

TABLE DES MATIÈRES

TABLE DES FIGURES

CHAPITRE 1

INTRODUCTION

Les données sont une composante fondamentale de l'informatique car elles représentent l'information que l'ordinateur est demandé de traiter. Il existe une multitude de traitement que l'on peut effectuer sur les données et il est apparu nécessaire d'agencer les données de telle manière à optimiser le traitement en question. C'est pourquoi les structures de données est un domaine qui a suscité un intérêt puissant de la part des chercheurs au fil des décennies.

Nous allons dans le cadre de ce projet étudier certains types de ces structures : les tas et les files de priorité, les arbres de recherche, ainsi que les tables de hachage. Nous allons les analyser selon différents aspects, principalement leurs caractéristiques, les opérations qui les définissent, ainsi que leur complexité.

CHAPITRE 2

ÉCHAUFFEMENT

2.1 Représentation de la clé 128 bits

Nous avons utilisé un tableau d'entiers de 4 éléments pour représenter. Une clé 128 bits est dont un quadruplet de 32 bits. Le constructeur de la classe reçoit une chaine en *Hexadécimal* utilise la primitive **BigInteger** pour la convertir en **binaire**.

Le code correspondant à cette section se trouve dans le fichier **"Cle128Bit.java"**.

2.2 Prédicat *inf*

La fonction **inf** dans la classe **Cle128Bit** compare deux clés de **128 bits**. Elle convertit ces clés en valeurs non signées sur 128 bits en utilisant des **masques** , puis détermine si la première clé est strictement inférieure à la seconde en parcourant et comparant les entiers 32 bits. Si une différence est détectée, elle renvoie **True** si la première clé est inférieure, sinon **False**.

2.3 Prédicat *eg*

Cette fonction marche d'une manière identique à la fonction *inf*, à la seule différence que si elle détecte une différence au niveau d'un des bits, elle renvoie **false**, indiquant que les clés ne sont pas égales. Sinon, elle retourne **true**.

CHAPITRE 3

TAS PRIORITÉ MIN

Le code correspondant à cette section se trouve dans le repertoire **"/P2TasPrioriteMin"**.

3.1 Définitions

La structure **ITasMin** est une interface définissant les opérations fondamentales sur un **tas min** telles que *SupprMin*, *Ajout* et *Construction*.

3.2 Représentation avec tableau

Le tas est représenté par un **tableau de ICle** dans lequel seront stockées les clés, le tableau a une capacité fixe et on l'augmente au besoin.

3.3 Représentation via arbre

Le tas est représenté par un **arbre binaire** où chaque noeud contient une clé et des références vers son parent et ses deux descendents. Chaque sous-arbre possède son minimum à la racine.

3.4 Fonctions utiles

Nous avons défini quelques fonctions utiles :

— **SupprMin** : Cette fonction supprime l'élément de clé minimale dans la structure du tas. Elle élimine la **racine** du tas puis réorganise les clés pour maintenir les propriétés du tas min.

— **Ajout** : La fonction **ajout()** permet d'insérer une nouvelle clé dans le tas.
Dans l'implémentation avec tableau, elle ajoute la clé à la fin du tableau, puis réorganise les éléments pour maintenir la propriété de tas min.

Dans l'implémentation avec arbre Elle exploite un système d'insertions récursives en respectant la propriété du tas min. Lorsque nécessaire, elle réajuste les nœuds et la position d'insertion pour garder la structure de tas minimum.

— **AjoutsIteratifs** : Cette fonction construit un tas de manière itérative à partir d'une liste d'éléments. Dans l'implémentation basée sur un tableau, elle ajoute successivement chaque élément de la liste au tas, appliquant à chaque fois les réorganisations nécessaires

3.5 Fonction *Construction*

3.5.1 Principe :

— **Commencer avec le tableau désorganisé** : On commence avec un tableau d'éléments qui ne forment pas nécessairement une structure de tas.

— **Identifier les nœuds non-feuilles** : On part de l'idée que la seconde moitié du tableau représente les feuilles du ***Tas***. L'indice du premier nœud non-feuille à partir de la fin dans un tableau est **(n/2) - 1**, où **n** est le nombre total d'éléments dans le tableau.

— **Appliquer le processus de tasification aux nœuds non-feuilles** : On effectue une opération de "tasification" (**heapify**) [2] sur chaque nœud non-feuille dans l'ordre inverse (du dernier au premier). Ce processus consiste à comparer le noeud avec ses enfants, et si la propriété de tas min (resp. max) n'est pas respectée, faire remonter le plus petit (resp. plus grand) enfant à la place du noeud. On répète cette opération de manière récursive sur touts les sous-abres affectés jusqu'à ce que la propriété de tas soit satisfaite pour le sous-arbre actuel.

— **Répéter jusqu'à ce que le tableau entier soit un tas** : On continue à appliquer la tasification à chaque nœud non-feuille jusqu'à ce qu'on atteigne la racine du tas, et à ce moment-là, l'ensemble du tableau sera transformé en un tas satisfaisant la propriété de tas.

3.5.2 Pseudo code

```
1  Function buildHeap(tableau):
2      n <- taille(tableau)
3
4      // On commence a partir du premier noeud non-feuille pour i de ((n / 2) - 1)
           a 0 faire:
5          heapify(tableau, n, i)
6
7  Function heapify(tableau, taille, indice):
8      plusGrand <- indice
9      gauche <- 2 * indice + 1
10     droit <- 2 * indice + 2
11
12     // Verifie si le noeud a un enfant gauche et si cet enfant est plus grand
           que le noeud courant
13     si gauche < taille et tableau[gauche] > tableau[plusGrand] alors:
14         plusGrand <- gauche
15
16     // Verifie si le noeud a un enfant droit et si cet enfant est plus grand que
           le noeud courant
17     si droit < taille et tableau[droit] > tableau[plusGrand] alors:
18         plusGrand <- droit
19
```

```
    // Si le plus grand element n'est pas le noeud courant, on effectue
    // un echange et on recursivement
    si plusGrand n'est pas egal a indice alors:
        echange(tableau[indice], tableau[plusGrand])
        heapify(tableau, taille, plusGrand)
```

3.6 Fonction *Union*

3.6.1 Principe :

Le principe ici est de réaliser l'union de deux tas sans clés communes en utilisant une approche de complexité linéaire.

1. Récupérer les tableaux représentant les tas.
2. Créer un ensemble (**unionSet**) pour stocker les clés des deux tas **sans doublons**.
3. Parcourir les deux tableaux, ajouter les éléments dans l'ensemble.
4. Convertir l'ensemble en une liste (**unionSansDoublon**) pour conserver l'ordre initial.
5. Appeler la fonction**construction** pour pour construire le tas correspondant à cette liste.

3.6.2 Pseudo code :

```
Function union(t2):
    tabTas1 <- getRepresentationTableau()
    tabTas2 <- t2.getRepresentationTableau()

    unionSet <- createEmptySet()

    // Ajouter les cles du premier tas dans l'ensemble
    for i from 0 to tabTas1.length - 1 do:
        unionSet.add(tabTas1[i])

    // Ajouter les cles du deuxieme tas dans l'ensemble
    for i from 0 to tabTas2.length - 1 do:
        unionSet.add(tabTas2[i])

    unionSansDoublon <- convertSetToList(unionSet)

    // Appeler la fonction de construction du tas avec la liste des cles uniques
    construction(unionSansDoublon)
```

3.7 Complexité

3.7.1 *SupprMin*

La complexité : $\Theta(logn)$

— Processus : Trouver la valeur la plus à droite dans le dernier niveau d'un arbre binaire de hauteur équilibrée, ce qui prend logarithmiquement (**log n**) du temps. puis on fait une réorganisation de la racine pour rétablir l'ordre correct, ce qui prend aussi (log n).

3.7.2 *Ajout*

La complexité : $\Theta(logn)$

— Processus : Identifier la position d'insertion pour une nouvelle valeur dans un arbre binaire équilibré, demandant (**log n**) opérations. Ensuite, ajuster l'arbre pour maintenir sa structure correcte (les remontées), également en (**log n**) opérations.

3.7.3 *AjoutsIteratifs*

La complexité : $\Theta(n.logn)$

— Processus : Chaque ajout individuel dans un arbre équilibré nécessite **log n** opérations. Avec **n** ajouts successifs, la complexité totale est de n fois le coût logarithmique, équivalant à (n * log n).

3.7.4 *Construction*

La complexité : $\Theta(n)$ Preuve :

— Le nombre de noeuds ayant une hauteur **h** est $\leq \frac{n}{2^h}$

— Hauteur de l'arbre : $\lfloor \log n \rfloor$

— Coût de permutation pour maintenir l'arbre à une hauteur **h** : h

— La complexité temporelle est donc la somme des nombres de noeuds de hauteur h fois le coût pour maintenir la propriété du tas min de chaque sous-arbre de hauteur h :

$$\sum_{h=0}^{\lfloor \log n \rfloor} \left(\frac{n}{2^h} \cdot h \right) = n \sum_{h=0}^{\lfloor \log n \rfloor} \frac{h}{2^h} \tag{3.1}$$

$$\leq n \sum_{h=0}^{\infty} \frac{h}{2^h} \tag{3.2}$$

$$\leq O(n) \tag{3.3}$$

$$= O(n) \tag{3.4}$$

3.7.5 *Union*

La complexité : $\Theta(n + m)$

— Processus : on a 2 tas de taille n et m, respectivement, pour les parcourir, chaque un a une complexité de $\Theta(n)$ et $\Theta(m)$ respectivement, puis pour construire la liste des élément (sans doublons) on a une complexité de $\Theta(n + m)$ puis la fonction construction, puisqu'elle est linéaire, sa complexité içi est en $\Theta(n + m)$

3.8 Vérification graphique de la complexité

3.8.1 *AjoutsIteratifs*

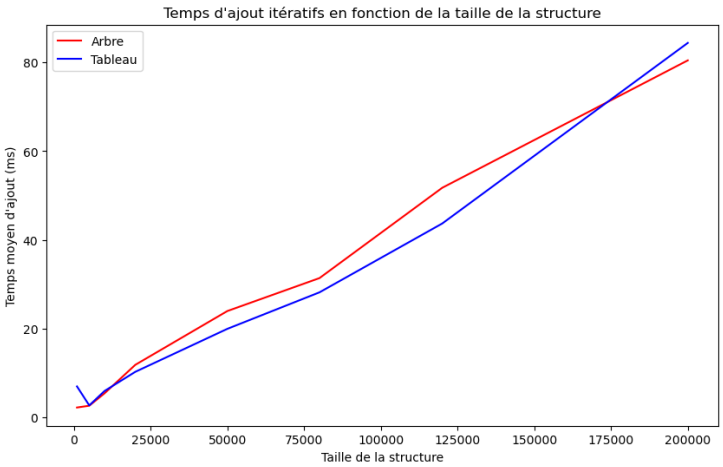

FIGURE 3.1 – Evaluation de AjoutsIteratifs pour les tas

Pour cette fonction, les deux représentations semblent relativement équivalentes.

3.8.2 *Construction*

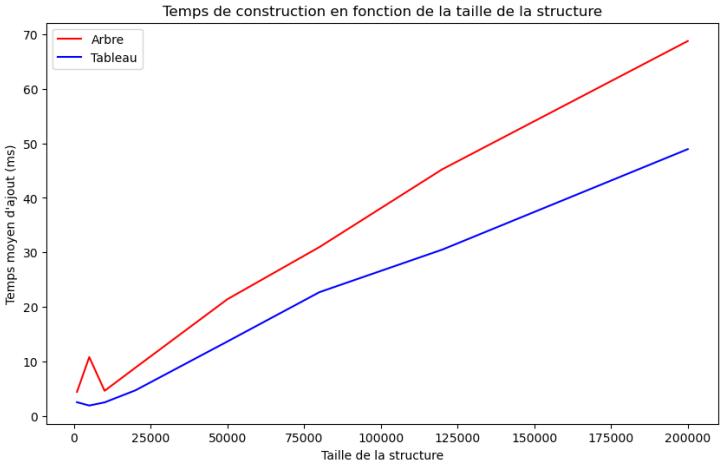

FIGURE 3.2 – Evalutation de Construction pour les tas

Nous pouvons remarquer que la représentation par tableaux est plus rapide que celle en arbre. De plus, remarquons que les temps maximums d'exécution ont maintenant diminué de 80 ms environs à moins de 70 ms, ce qui rejoint nos calculs théoriques sur la complexité de la fonction Construction et son avantage par rapport aux ajouts itératifs.

3.8.3 *SupprMin*

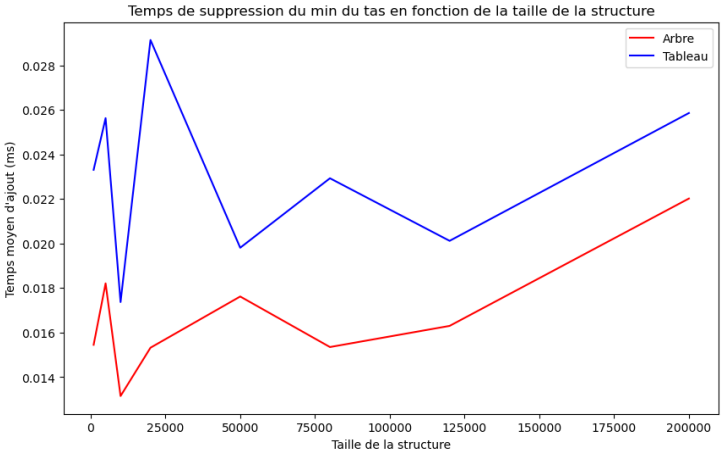

FIGURE 3.3 – Evaluation de SupprMin pour les tas

Les temps d'exécution fluctuent pour les instances de plus petites tailles. Cela semble étrange mais peut éventuellement être expliqué par le fait que dans certains cas on a eu de la chance et la valeur qui se trouve en dernière position lorsqu'elle a été remontée ne nécéssitait pas beaucoup d'échanges pour retrouver la structure de tas.

3.8.4 *Union* :

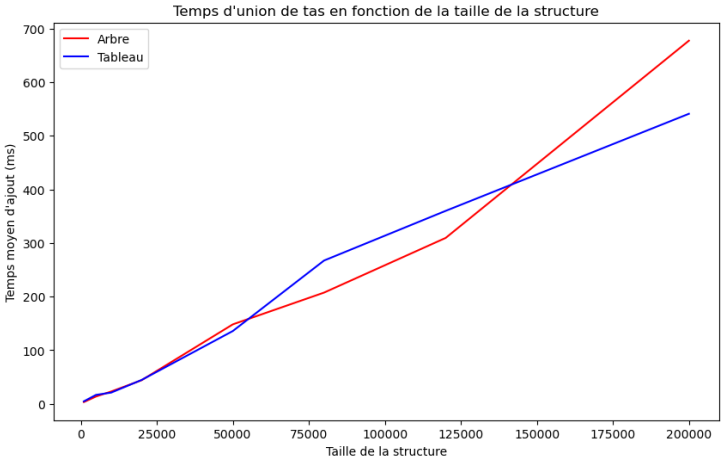

FIGURE 3.4 – Evaluation de Union pour les tas

Les deux structures semblent équivalentes au début mais dans les grandes tailles la représentation par tableau prend l'avantage.

3.8.5 Conclusion

Nous pouvons conclure à travers ces expérimentations que les complexités théoriques sont cohérentes par rapport aux temps d'exécution trouvés expérimentalement, et aussi que la représentation tabulaire est plus avantageuse en général, probablement parce que l'accès aux données, notemment aux fils et aux parents se fait d'une manière directe en fonction de l'index de la clé courante, et aussi car nous soupçonnons que le langage Java présente une certaine lenteur lors de l'allocation de la mémoire nécessaire pour les noeuds de la représentation arborescente.

CHAPITRE 4

FILES BINOMIALES

Nous rappelons qu'une **file binomiale** est une **suite de Tournois Binomiaux**, il nous faut donc définir une structure qui traite les tournois ainsi que leurs fonctions utiles.

Le code correspondant à cette section se trouve dans le repertoire **"/P3FilesBinomiales"**.

4.1 Tournois Binomiaux

Un tournoi binomial est une **structure arborescente** de données utilisée pour représenter un tas min. Il se compose de plusieurs **arbres binomiaux** où chaque arbre suit une structure spécifique, un nœud principal contenant une clé et des fils, et chaque fils étant lui-même un tournoi binomial de **degré inférieur**.

4.1.1 Primitives de bases

Nous avons défini les primitives de bases pour la gestion des Tournois.

— **union** : Fait l'union de 2 tournoi binomial de même taille.

— **decapite** : Renvoie la file binomiale obtenue en supprimant la racine du tournoi binomial

— **file** : Renvoie la file binomiale réduite au tournoi binomial courant

— **getFils** : Renvoie le fils présent à l'indice donnée en paramètre.

le code de cette partie se trouve dans les fichiers : '**ITournoiBinomial.java**' et '**TournoiBinomial.java**'

4.2 Files Binomiales

La classe "**FileBinomiale**" représente une file spéciale organisant les éléments en utilisant des "**tournois binomiaux**". Voici ses caractéristiques :

— **Degré** : Indique la taille de cette file.

— **Nombre de tournois** : Le nombre de tournois binomiaux actuellement présents dans cette file.

— **Ensemble de tournois** : Liste regroupant les tournois binomiaux formant cette file.

— **Tournoi de clé minimale** : Stocke le tournoi ayant la plus petite clé parmi ceux de la file binomiale.

4.2.1 Les fonctions

Nous avons défini quelques fonctions utiles :

— **Union** : Cette fonction réalise l'union de deux files binomiales. Elle sélectionne le tournoi de degré minimum dans chaque file et effectue des opérations basées sur la comparaison de degrés. Si **aucune retenue** n'est présente, elle fusionne les tournois de degré minimum. Si une **retenue existe**, elle la traite par rapport aux degrés des tournois minimaux des files, gérant les combinaisons possibles pour conserver la structure des files binomiales tout en fusionnant les tournois appropriés.

— **SupprMin** : Cette fonction supprime le tournoi minimum de la file binomiale. Tout d'abord, elle retire ce tournoi minimum de la file **tournoiMin**, puis elle réalise une opération d'**union** entre la file actuelle et le résultat de la **décapitation** du tournoi minimum. Cette opération d'union permet de maintenir la structure et les propriétés des files binomiales après la suppression du minimum.

— **Ajout** : Cette fonction permet d'insérer un **tournoi** dans la file. Elle réalise cette opération en appelant la fonction d'**union** *unionFile* entre la file courante et la file associée au tournoi à insérer. Cela garantit que la file résultante maintient la structure et les propriétés des files binomiales tout en ajoutant le tournoi.

— **Construction** : La fonction *construction* crée une file binomiale à partir d'une **liste d'éléments**. Elle initialise une file vide *file*, puis **itère** sur chaque élément de la liste. Pour chaque élément, elle **crée un tournoi binomial** contenant cette clé, puis utilise la fonction *ajout* pour insérer ce tournoi dans la file. En fin de compte, elle retourne la file binomiale construite à partir des éléments de la liste.

4.2.2 Vérification graphique de la complexité.

i *Construction*

Le temps d'execution de la fonction *Construction* pour une file binomiale est linéaire en la taille des instances données.

ii *Union*

Les résultats semblent difficiles à interpréter car la durée d'exécution devrait être croissante du début à la fin, néanmoins nous pouvons avancer l'hypothèse que cela peut être dû aux différentes retenues qui peuvent apparaître lors de l'union et qui peuvent augmenter le temps d'exécution si elles sont nombreuses.

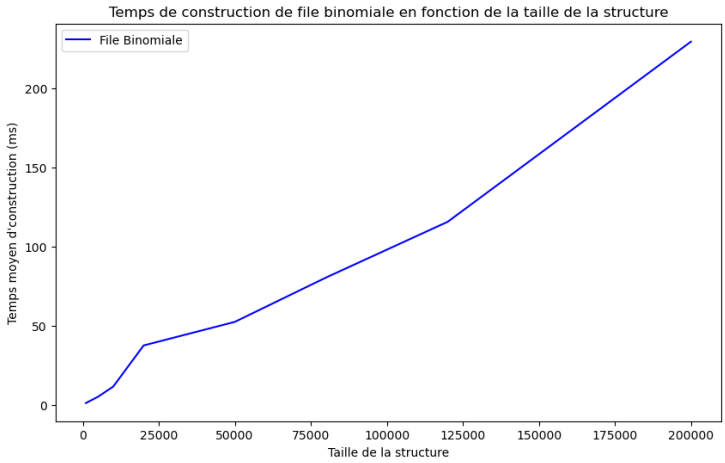

FIGURE 4.1 – Evaluation de Construction pour les files binomiales

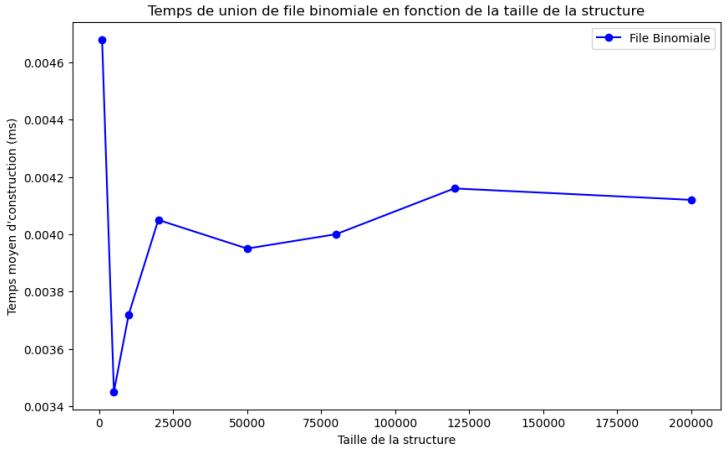

FIGURE 4.2 – Evaluation de Union pour les files binomiales

CHAPITRE 5

HACHAGE

Le code correspondant à cette section se trouve dans le repertoire **"/P4HachageMD5"**.

5.1 Présentation de MD5

MD5 est une méthode de hachage cryptographique qui permet de chiffrer un message sur code unique et de taille réduite (128 bits) tout en minimisant les collisions.

5.2 Description de l'implémentation

La classe **MD5** contient une implémentation de l'algorithme de hachage MD5 [3].

5.2.1 Les étapes de l'algorithme :

1. **Padding** : Ajouter un remplissage des bits jusqu'à atteindre une taille multiple de 512 bits, de telle sorte que les 64 derniers bits soient réservés à la taille du message original
   Exemple : Message de 500 bits. Ajouter un 1 puis des 0 jusqu'à arriver à 1024-64 = 960. Rajouter les 64 derniers bits contenant la taille du message.
2. Division sur n blocs de 512-bits
3. Initialisation des variables *a, b, d et c*
4. diviser les blocs de 512-bits en 16 mots de 32 bits
5. Pour 64 itérations et pour chaque sous-bloc, on effectue diverses operations $(AND, OR, XOR, leftshift...)$ et on met à jour les valeurs de $a, b, cetdaufuretàmesure$.
6. Retourner le code final en concaténant les 4 variables a, b, c, d.

CHAPITRE 6

ARBRE DE RECHERCHE

6.1 Arbre de Recherche AVL

L'implémentation de la structure d'arbre de recherche AVL repose sur une classe générique, `AVL`, permettant de la manipuler, et conforme aux spécifications définies dans l'interface `IAVL`.

La classe `AVL` expose des méthodes permettant l'insertion d'une clé, la recherche d'une clé, l'obtention de la hauteur de l'arbre, ainsi que la récupération des clés triées.

6.1.1 Explication du choix de la structure

Nous avons choisi un arbre AVL car il est adapté la tâche demandée, qui est de permettre la recherche en temps $O(logn)$. En effet, la caractéristique principale de cette structure est qu'elle garantit qu'à chaque noeud, la différence des hauteurs de chaque sous-arbre enraciné en ce noeud est au maximum 1, ce qui permet de maintenir une hauter égale à $logn$, n étant le nombre de données.

6.1.2 Structure des Données

Nos AVL sont constitués de noeuds, décris dans la classe `Noeud` et contenant une clé ainsi que des pointeurs vers un noeud parent ainsi que 2 noeuds descendants.

6.1.3 Insertion

La méthode `inserer` permet d'ajouter une clé dans l'arbre AVL. L'algorithme d'insertion maintient l'équilibre de l'arbre en effectuant d'éventuelles `rotations`.

6.1.4 Recherche

La méthode `rechercher` effectue la recherche d'une clé dans l'arbre AVL. Elle renvoie le nœud contenant la clé recherchée, ainsi que le nombre de comparaisons effectuées la pour trouver.

6.1.5 Résultats et Analyse

Pour évaluer les performances de recherche, des fichiers de clés aléatoires de différentes tailles sont utilisés. Les résultats de la recherche sont enregistrés dans un fichier CSV (`recherche_AVL.csv`), contenant la taille de l'arbre, la moyenne du nombre de comparaisons et le maximum de comparaisons effectuées pour chaque taille d'arbre.

La méthode `genererResultatRecherche` utilise ces fichiers pour évaluer la performance moyenne et maximale de recherche dans l'arbre pour chaque taille d'arbre testée.

Le code correspondant à cette section se trouve dans le repertoire **"/P5ArbreDeRecherche"**.

CHAPITRE 7

ÉTUDE EXPÉRIMENTALE

Dans cette section, nous mettons en place l'étude expérimentale demandée en tant que conclusion de notre projet. Nous avons effectué des tests sur les temps d'exécution des processus d'*ajout*, *construction*, *suppression* du minimum, ainsi que l'*union* pour les structure de tas dont les deux représentations, en arbre et en tableau, ont été comparées, ainsi que pour la structure de file binomiale.

7.1 Expérimentations sur MD5

Nous avons stocké le haché MD5 des mots de Shakespear dans une structure d'AVL et avons obtenu un total de **23086** mots différents pour un nombre de **0 collisions**. Ceci s'explique par le fait que le code final est sur 128 bits, ce qui donne $2^{128} > 3.4 * 10^{38}$ codes différents, alors que 23086 ne nécessite que 15 bits pour le représenter.

Nous rappelons que la probabilité d'avoir une collision avec le hachage **MD5** est $1.47 * 10^{-29}$ [1]

7.2 Comparaison des structures sur les données extraites

Dans cette section, nous montrons les résultats d'exécution des fonctions pour nos deux structures sur les données de l'oeuvre de Shakespeare

Lors de notre comparaison entre les files et les tas, nous avons choisi la représentation tabulaire pour les tas car elle a en général retourné de meilleurs résultats lors des tests précédents.

7.2.1 supprMin

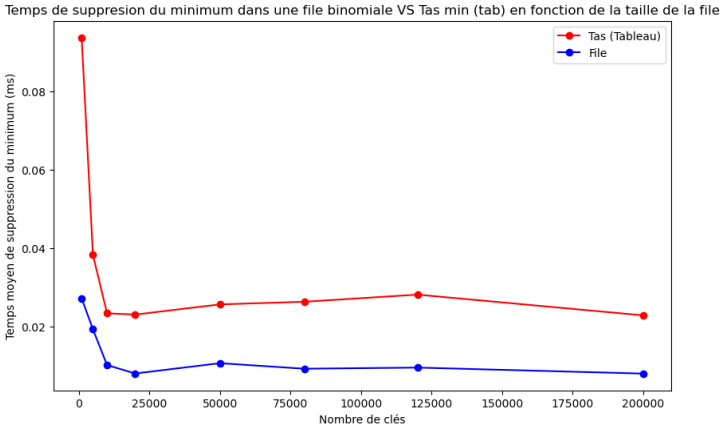

FIGURE 7.1 – Evaluation de supprMin pour les files binomiales vs les tas

7.2.2 Ajout

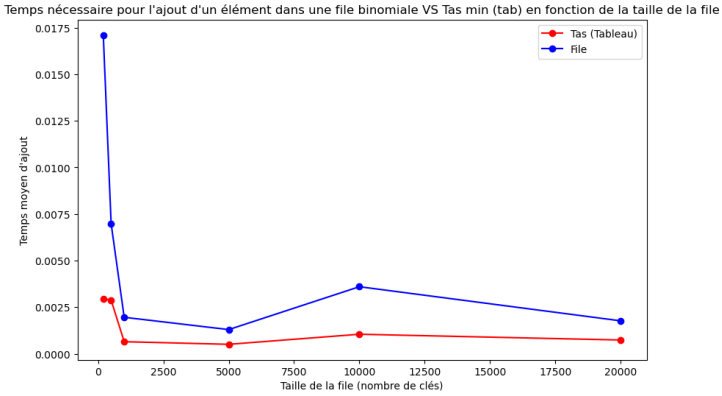

FIGURE 7.2 – Evaluation d'Ajout pour les files binomiales vs les tas

7.2.3 Construction

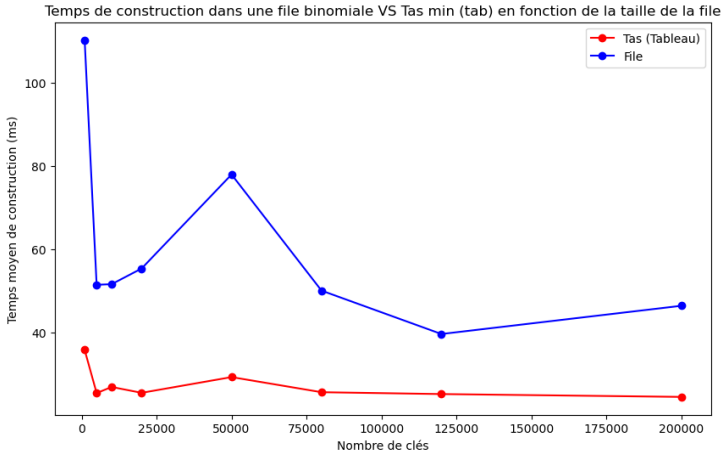

FIGURE 7.3 – Evaluation de Construction pour les files binomiales vs les tas

Pour les fonctions précédentes, nous remarquons quelques anomalies dans les résultats retournés car pour le temps est maximal pour les instances les plus petites et ensuite la complexité semble relativement constante.

7.2.4 Union

Dans ce cas de test, nous avons effectué l'union de structures crées à partir des données de Shakespeare avec des structures crées avec les données aléatoires fournies.

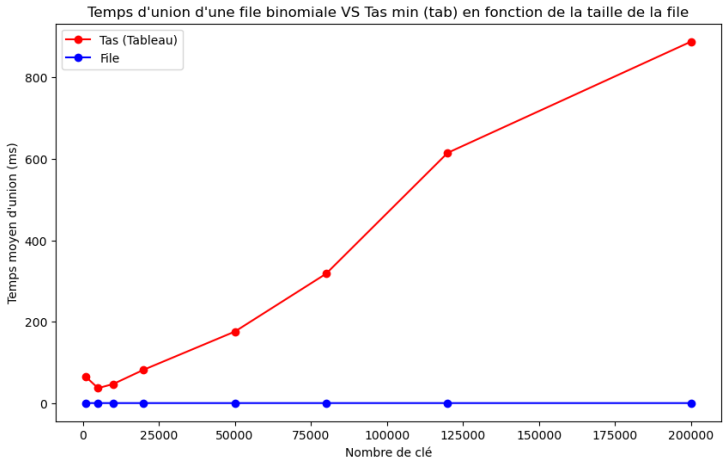

FIGURE 7.4 – Evaluation d'Union pour les files binomiales vs les tas

Ici, nous voyons bien l'avantage des files pour l'union par rapport aux tas qui se fait en temps linéaire. La courbe des files est difficilement visualisable à cause de l'échelle utilisée, en effet la complexité étant logarithmique, elle donne des valeurs très inférieures à une complexité linéaires notamment pour les grandes instances.

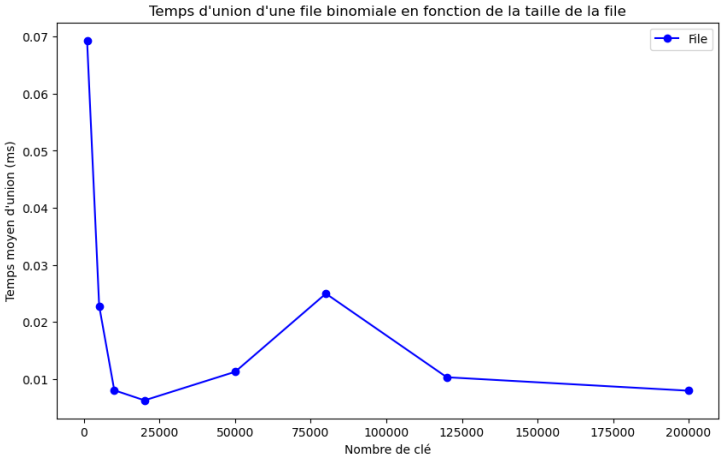

FIGURE 7.5 – Evaluation d'Union pour les files binomiales isolée

Ici, nous voyons la courbe des files isolée et pouvons voir qu'elle n'est pas constante, mais possède les mêmes anomalies que pour les autres fonctions, c'est à dire que son maximum est en l'instance la plus petite et n'est pas croissante tout le long.

CONCLUSION GÉNÉRALE

Dans le cadre de notre projet, nous avons analysé le comportement de différentes structures de données telles que les files et les tas de priorité, et avons cherché à optimiser le temps nécessaire à leur construction. Nous avons étudié une méthode de hachage cryptographique, MD5, et avons déterminé une structure de données pour accéder rapidement aux données chiffrées. Pour finir, nous avons effectué des expérimentations sur les différentes structures étudiées, et nous pouvons conclure que les complexités théoriques s'accordent avec les complexités obtenues empiriquement.

BIBLIOGRAPHIE

[1] Gorka Ramirez. Md5 : The broken algorithm, 2015. Accessed : 27 Nov 2023.

[2] Jeremy West. How can building a heap be o(n) time complexity ?, 2012. Accessed : 29 Nov 2023.

[3] Wikipedia. Md5, 2023. Accessed : 24 Nov 2023.