Sorbonne Université - MU4IN500

Rapport du projet d'Algorithmique Avancée
Devoir de programmation

Laura Rattanavan, 28633935
Julien Fang, 28605540

15 décembre 2023

Table des matières

Introduction

L'objectif de ce projet est de visualiser concrètement les courbes des temps d'exécution des structures de données introduites en cours : les tas min et files binomiales. Nous nous intéresserons entre autres aux algorithmes Ajout, Construction, SupprMin et Union.

Pour ce faire, nous utiliserons notamment des jeux de données aléatoires, ainsi que les mots de l'oeuvre de Shakespeare auxquels nous appliquerons la fonction de hachage *Message Digest 5* (MD5) afin d'obtenir des clés codées sur 128 bits. Enfin, nous implémenterons une structure arborescente de recherche : les arbres binaires de recherche.

Le langage de programmation utilisé au cours de ce projet est Python en programmation orientée objet. Les graphes sont tracés à l'aide de la fonction plot de la bibliothèque matplotlib. Les fichiers fournis sont listés en annexe à la fin de ce rapport.

1 Présentation

1.1 Échauffement

Question 1.1

Nous représentons une clé 128 bits sous forme de classe `Cle128`. Une clé est composée de 4 entiers non signés 32 bits et se lit de `v4` à `v1`, `v1` étant la valeur la plus grande.

```
class Cle128:
    def __init__(self, v1, v2, v3, v4):
        self.v1 = np.uint32(v1)
        self.v2 = np.uint32(v2)
        self.v3 = np.uint32(v3)
        self.v4 = np.uint32(v4)
```

Question 1.2

Le prédicat **inf** permet de vérifier si une clé cle1 est strictement inférieure à une clé cle2.
Afin de vérifier le prédicat, il faut comparer chaque attribut de la classe. Si un des attributs de cle1 est supérieur à cle2, cela retourne vrai et dans le cas échéant, faux.

Question 1.3

Le prédicat **eg**, permet de vérifier l'égalité entre cle1 et cle2. Il faut alors vérifier l'égalité de tous les attributs des deux clés à un à un.

2 Structure 1 : Tas de priorité min

Question 2.4

Pour la structure de tas min représentée en arbre binaire, nous avons créé une classe **Noeud**. Un **Noeud** est composé d'une clé (étiquette), d'un enfant gauche et un enfant droite.
Nous avons notamment jugé utile de définir un attribut parent qui sera utilisé lors de la remontée dans l'arbre que nous expliquerons par la suite. De même, nous avons accès au nombre de noeuds (taille) d'un **Noeud**, qui sera utilisé dans.

```
class Noeud:
def __init__(self,cle):
    self.cle = cle
    self.nbNoeuds = 1
    self.parent = None
    self.gauche = None
    self.droite = None

class TasMinArbre :
def __init__(self):
    self.racine = None
```

Ainsi, un **TasMinArbre** est composé d'une racine qui est un **Noeud**. Les méthodes de cette classe sont les suivantes :

- **supprMin(tas)** : Cette fonction supprime le minimum d'un tas min, arrange le tas en une structure de tas min correcte et retourne l'élément min supprimé. Étant donné notre structure, il s'agit donc de supprimer la racine du tas qui est le min, puis de retourner une structure de tas min correcte.

  Pour cela, nous pouvons distinguer deux cas simples : si le tas est vide ou si le tas est composé d'un unique noeud (aucun enfant). Le premier cas ne nécessite aucun traitement. Quant au 2e cas, il s'agit de stocker la clé de la racine et de mettre la racine à `None`.
  Sinon, le tas est composé de plus d'un noeud. Nous récupérons alors le dernier noeud du tas et supprimons ce dernier noeud (fonction **supprDernierNoeud**) afin de le remplacer avec la racine du tas. Enfin, nous devons réorganiser le tas afin qu'il corresponde à une structure de tas min correcte (fonction **descente**).

  — La fonction récursive **supprDernierNoeud(tas, noeud)** cherche le dernier noeud d'un tas (à partir du sous-arbre **noeud**), le supprime du tas, met à jour le nombre de noeuds et retourne le dernier noeud.

  Pour cela, nous regardons si le noeud courant est le dernier du tas (aucun enfant). Si c'est le cas, le supprimer revient à supprimer l'enfant correspondant du parent de ce noeud dans le tas. Il se peut que ce noeud soit la racine (pas de parent, cas trivial à traiter).

  Sinon, si le noeud possède au moins un enfant, il faut chercher si le dernier noeud appartient à l'enfant gauche ou droite du sous-arbre courant (afin de lancer l'appel récursif). Soit le noeud courant ne possède qu'un enfant gauche (appel récursif trivial), soit il possède un enfant gauche et droite. Pour le 2e cas, nous utilisons le fait qu'un arbre complet possède $2^{h+1} - 1$ noeuds, avec $h = \lfloor \log_2(n) \rfloor$ sa hauteur. Ainsi, son sous-arbre (enfant) complet possède $\frac{2^{h+1} - 1 - 1}{2} = 2^h - 1$ noeuds.

Nous pouvons alors effectuer des comparaisons avec la taille de l'enfant pour savoir si les enfants gauche et droite sont complets (dans ce cas, le dernier noeud se trouve à droite). Sinon, si l'enfant droite n'est pas complet, il faut regarder si le dernier étage du tas est vide ou occupé. Pour cela, nous utilisons le nombre de noeuds d'un tas complet sans son dernier étage (profondeur h) pour le soustraire à la taille de l'enfant droite et ainsi tester si le dernier étage est vide ou occupé (si occupé, le dernier est à droite). Si le dernier étage est vide ou l'enfant gauche n'est pas complet, il faut donc descendre à gauche.

— La fonction **descente(tas, noeud)** réorganise le tas min en structure correcte à partir d'un noeud (la racine) : elle fait descendre le noeud tant qu'il est plus grand que ses enfants.
Il faut tout d'abord trouver le nouveau min du tas. Ensuite, tant que le noeud n'a pas atteint le bas du tas (au moins l'enfant gauche existe) et qu'il y a des noeuds à échanger (gauche ou droite plus petit que noeud), nous échangeons le noeud avec son enfant gauche ou droite plus petit que lui.

- **Ajout(tas, cle)** : Fonction qui ajoute une clé dans le tas. Dans un premier temps, il faut créer le nouveau noeud contenant la clé à ajouter. Ensuite, si la racine est vide, ce nouveau noeud devient la racine. Sinon, nous ajoutons le nouveau noeud dans le tas (place disponible en bas du tas) puis nous réordonnons le tas min en structure correcte (fonction **AjoutRec**).

  — La fonction récursive **AjoutRec(tas, noeud, nouveau_noeud)** ajoute un nouveau noeud au tas à partir du sous-arbre courant noeud, met à jour le nombre de noeuds, puis réorganise le tas min avec la fonction **remonter**.
  Pour cela, nous regardons si un des enfants de noeud est nul. Sinon, la même méthode de calcul que **supprDernierNoeud** nous permet d'effectuer des comparaisons avec la taille des enfants. Toutefois, il s'agit ici de trouver une place disponible dans le tas et non son dernier noeud. Enfin, sans oublier d'incrémenter le nombre de noeuds, nous faisons appel à **remonter**.

  — La fonction **remonter(tas, noeud)** réorganise le tas min en remontant le noeud tant qu'il est plus petit que son parent.

- **AjoutsIteratifs(tas, liste_cle)** :
  Nous parcourons la liste de clés, et nous faisons appel à **Ajout** afin d'ajouter les clés dans le tas min de structure arbre binaire.

La structure de tableau sera plus simple, une simple liste pour les éléments du tas suffit.

```
class TasMinTableau:
def __init__(self):
    self.tas=[]
```

Cette structure permet d'accéder à n'importe quel fils et père grâce aux indices de la liste. En particulier, le premier élément de la liste est la racine du tas et pour un noeud à la position i dans la liste, ses enfants gauche et droite sont respectivement à la position $2i+1$ et $2i+2$.

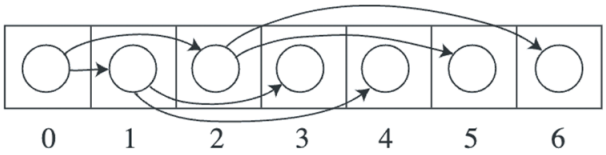

FIGURE 1 – Représentation de Tas Min sous forme de tableau

Dans la classe **TasMinTableau**, nous avons les fonctions :

- **supprMin(tas)** : Cette fonction est la version de SupprMin pour un tas min tableau. Nous traitons les cas suivants :
  — Si le tableau est vide, il n'y a pas de min à supprimer.
  — Si le tableau ne contient qu'un seul élément, nous retirons cet élément (min à retourner).
  — Sinon, nous relevons d'abord la racine du tas (élément minimum à la première case du tableau, **min_element**) puis nous remplaçons la racine du tas par le dernier élément du tableau. Ensuite, nous devons réordonner le tas afin d'avoir une structure correcte. Pour cela, nous devons descendre la nouvelle racine dans le tas tant qu'elle est plus grande que ses enfants. Or, pour un noeud à la position i dans le tableau, nous pouvons calculer la position de ses enfants gauche et droite (respectivement $2i+1$ et $2i+2$). Ainsi, nous cherchons l'indice du plus petit entre le noeud i (racine dans le tas avant de réordonner) et ses deux enfants. Si cet indice est différent de celle du noeud i, il faut échanger le noeud avec le plus petit enfant trouvé et continuer à traiter le noeud. Nous arrêtons ce traitement si le noeud ne peut plus être échangé avec un de ses enfants.

- **Ajout(tas, élément)** :
  Nous ajoutons d'abord l'élément à la fin du tableau. Ensuite nous récupérons l'indice (**ifils**) de l'élément ajouté.
  Nous récupérons l'indice du père de **ifils** dans le tas.
  Nous réalisons une boucle, tant que nous n'avons pas atteint la racine et que le fils est plus petit que son pere, alors :
  — Nous échangeons les valeurs dans le tableau à l'indice **ifils** et celui du père
  — **ifils** récupère l'indice du père
  — L'indice du père change afin de remonter dans l'arbre (ipere = (ifils − 1)//2

- **AjoutsIteratifs(tas, liste_cle)** :
  Nous parcourons la liste de cle, et nous faisons appel à **Ajout** pour ajouter les clés dans le tas min de structure tableau.

Question 2.5

Ci-dessous figure le pseudo-code de la fonction **Construction** pour la construction d'un Tas Min arbre. Par hypothèse, nous supposons que le tas passé en paramètre est vide.
Remarque : contrairement à notre implémentation où certaines fonctions n'ont pas de valeur de retour, ici les retours ont été ajoutés pour faciliter la lecture du pseudo-code.

Algorithme 1 : Construction Tas Min Arbre

Entrées : tas (vide), listeCles
Sorties : tas construit avec les clés de listeCles
Si listeCles est non vide **:**
  tas ← **AjoutConstruction(tas, listeCles)**
  MiseAJourNbNoeuds(tas.racine)
  tas ← **RemonterConstruction(tas.racine)**
retourner tas

Tout d'abord, nous commençons par vérifier un cas de base, lorsque la liste des clés est vide. Ensuite, nous faisons appel à la fonction **AjoutContruction** qui permet de créer un tas sans se soucier des valeurs de chacune des clés.

La fonction **AjoutConstruction** prend le premier élément du tableau et le met à sa racine. Puis elle fait appel à **AjoutConstructionRec**.

La fonction **AjoutConstructionRec** utilise les indices (**position**) afin de savoir si l'élément de la liste pour un indice donné sera ajouté comme père, fils gauche ou droit dans le tas. La position sert à savoir quel élément de la liste nous devons récupérer pour ajouter au tas. Il y a plusieurs cas :

- Si le noeud est vide ou la position est supérieur ou égal a la taille de la liste, nous avons fini d'ajouter les clés dans le tas.
- Sinon, nous récupérons les indices de l'élément à ajouter comme enfant gauche et droit :
  - gauche_position $= 2 \times$ position $+ 1$
  - droite_position $= 2 \times$ position $+ 2$
  - Si **gauche_position** est inférieur à la taille de la liste des clés, il y a un fils gauche à ajouter au noeud : la clé à la position **gauche_position** dans la liste. Puis nous faisons un appel récursif de AjoutConstructionRec pour ajouter les enfants de ce nouvel enfant gauche.
  - Même traitement pour **droite_position**.

Ensuite, nous utilisons **MiseAJourNbNoeuds** pour mettre à jour le nombre de noeuds de chaque noeud du tas (taille de chaque sous-arbre).
Enfin, nous réalisons la remontée en partant du bas du tas pour réordonner le tas en structure de tas min.

Ci-dessous figure le pseudo-code de la fonction **Construction** pour la struction d'un tableau.

Algorithme 2 : Construction Tas Min Tableau

Entrées : tas (vide) , listeCles
Sorties : tas construit avec les clés de listeCles
tas ← listeCles
taille ← longueur du tas min tableau
pour i **de** taille//2 **à** 0 **faire**
  tas ← **RemonterConstruction(tas, i)**
retourner tas

Tout d'abord, listeCles devient le tas non ordonné. Ensuite, la fonction récupère la taille de la liste afin de parcourir du bas vers le haut du tas en appelant la fonction **RemonterConstruction**. En commençant par la moitié de la liste, nous ignorons les feuilles qui seront traitées par la suite dans **RemonterConstruction**.

La fonction **RemonterConstruction** effectue une boucle **true** et récupère d'abord les indices des fils gauche et droite du noeud courant.

- gauche = $2 \times$ indice $+ 1$
- droite = $2 \times$ indice $+ 2$
- indice_min = indice **indice_min** représente le plus petit élément entre le père et ses fils

Dans la boucle, nous cherchons qui est le plus petit entre le noeud et ses enfants gauche et droite. Plusieurs conditions sont ainsi vérifiées :

— Si gauche est inférieur à la taille de la liste (l'enfant existe) et cet enfant gauche est plus petit que le min trouvé, alors l'enfant gauche devient le min (nous récupérons son indice dans le tas).
— De même pour droite.
— Si **indice_min** est différent de son indice initial passé en argument, un des enfants du noeud est plus petit que le noeud. Nous échangeons donc les deux clés dans le tas et nous descendons dans le tas pour continuer à ordonner et trouver la bonne place du noeud dans le tas.
— Sinon, nous sortons de la boucle avec **break** : il n'y a plus d'échange à effectuer.

Question 2.6

Voici le pseudo-code de **Union** de tas min pour une structure d'arbre binaire :

Algorithme 3 : Union Tas Min Arbre

Entrées : tas1 , tas2
Sorties : Union des deux tas ne partageant aucune clé commune
tas ← **TasMinArbre()**
listeCles ← obtenirListeCles(tas1)
listeCles = listeCles + obtenirListeCles(tas2)
tas ← **Construction(tas, listeCles)**
retourner tas

Dans cette fonction, nous commençons par initialiser un nouveau Tas Min vide. Puis nous récupérons la liste des clés **listeCles** du **tas1** ainsi que celle du **tas2** à l'aide de la fonction **ob-**

tenirListeCles. Ensuite, nous réalisons la **Construction** indiquée dans la question précédente, avec le tas vide précédemment créé et les clés de **listeCles**.

Voici le pseudo-code de **Union** pour une structure de tableau :

Algorithme 4 : Union Tas Min Tableau

Entrées : tas1 , tas2
Sorties : union des deux tas ne partageant aucune clé commune
tas ← **TasMinTableau()**
tas.tas ← tas1.tas + tas2.tas
tas.tas ← **Construction(tas, tas.tas)**
retourner tas

Le principe est similaire à l'**Union** pour une structure d'arbre. Nous concaténons ici les tas1 et tas2 représentés par leur liste de clés dans le tas tableau du nouveau tas créé.

Question 2.7

Toutes nos complexités sont exprimées par rapport à n le nombre d'éléments dans la structure.

Les complexités de nos diverses fonctions pour la structure d'un tas min arbre binaire :

- **SupprMin** : La suppression du minimum consiste à supprimer la racine par le dernier élément du tas, puis effectuer une descente afin de rétablie la propriété du tas.
  La racine est accessible en O(1). Puis récupérer le dernier élément du tas dépend de la hauteur du tas qui est en O(log n), et de même pour la descente.
  Donc la complexité de **SupprMin** est en O(log n).

- **Ajout** : Lors de l'ajout, nous insérons l'élément que à ajouter à l'étage le plus bas du tas, ce qui nécessite une descente. Donc l'opération d'insertion dans le pire cas est O(log n).
  Nous effectuons ensuite sa remontée pour rétablir la propriété du tas min. Cette opération a également une complexité en O(log n) si le noeud ajouté doit être remonté à la racine.
  Donc la complexité de l'ajout est en O(log n).
- **AjoutsIteratifs** : La complexité de cette fonction est O(m log n), où m est la longueur de la liste de clés à ajouter et n est le nombre d'élément dans le tas avant l'opération. Cette fonction effectue alors m appels à **Ajout**.

- **Construction** : La complexité de **AjoutConstruction** est en $O(n)$. Elle crée un noeud pour chaque clé de la liste de clés et la récursion permet de descendre dans l'arbre en ajoutant des noeuds à chaque niveau.
  Ensuite, la complexité de la fonction **MiseAJourNbNoeuds** dépend du nombre d'additions effectuées. Dans le pire cas, nous effectuons 2 additions.
  La complexité s'exprime alors $T(n) = 2T\left(\frac{n}{2}\right) + O(2) = O(n)$ par le théorème maître.
  Enfin, **RemonterConstruction** réalise des appels récursifs afin d'être dans les noeuds les plus bas du tas. Puis elle réalise pour chacun une remontée si nécessaire afin de respecter la règle du tas min. Elle a une complexité en $O(n)$.
  Donc, la **Construction** est en $O(n)$.

- **Union** : La fonction récupère la liste de clé des deux tas. Si le tas 1 et le 2 possèdent respectivement n et m éléments, cette opération est en O(n+m). Puis elle fait appel à **Construction** qui a une complexité O(n+m).
  Donc la complexité de l'**Union** est en O(n+m).

Les complexités pour la structure d'un tas min tableau :

- **SupprMin** : La suppression du minimum consiste à remplacer la racine par le dernier élément du tableau qui se fait en temps constant. Puis réordonner le tas dépend de la hauteur du tas que nous devons parcourir.
  La complexité de suppression du minimum est donc en O(log n).

- **Ajout** : L'ajout d'un élément à la fin du tas (liste) s'effectue en O(1). Puis dans le pire cas, il faut faire la remontée, ce qui dépend de la hauteur du tas qui est en O(log n).
  Donc la complexité est en O(log n).

- **AjoutsIteratifs** : Cette fonction a une complexité en O(m log n), puisqu'elle réalise m fois la fonction **Ajout** sachant que **Ajout** est en O(log n), d'où la complexité.

- **Construction** : La fonction parcourt chaque élément du tableau ce qui donne une complexité O(n) et effectue une remontée en O(log n) pour maintenir la propriété du tas.
  La complexité est en O(n).

- **Union** : La fonction récupère les éléments des deux tas tas 1 et tas 2, respectivement de taille n et m. Puis la fonction **Construction** est appelé avec une complexité O(n+m).
  Donc la complexité de Union est en O(n+m).

Question 2.8

Nous avons tracé nos courbes de complexité avec les clés aléatoires fournies. Comme chaque taille de jeux de données est fournie avec 5 fichiers de clés, nous faisons donc des moyennes sur 5 pour nos tests de performance. Nous utilisons par ailleurs la fonction `process_time()` de la bibliothèque `time` afin de relever le temps d'exécution pour le morceau de code exécuté. Enfin, nos jeux de données contiennent des clés en hexadécimal. Pour créer nos clés 128, nous coupons donc en quatre les clés fournies.

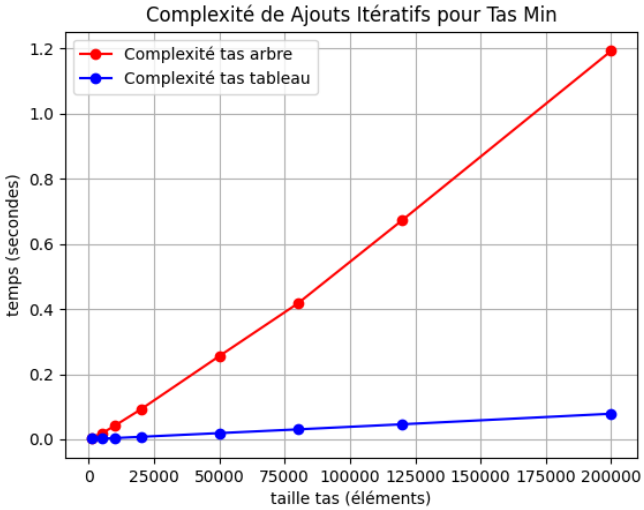

FIGURE 2 – Graphe de complexité de Ajouts Itératifs pour Tas Min

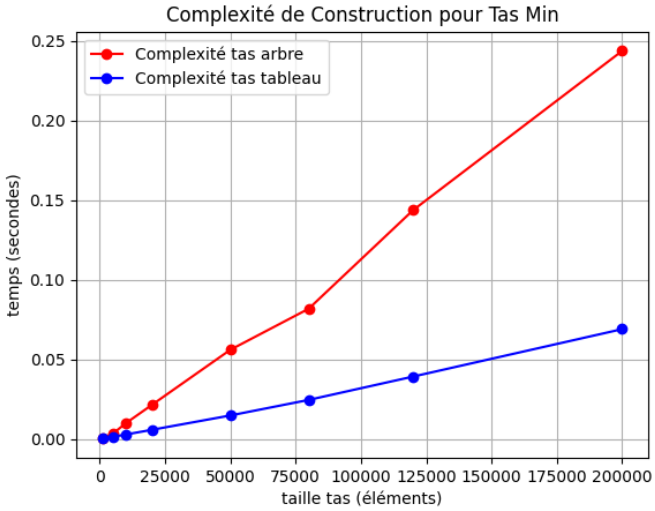

FIGURE 3 – Graphe de complexité de Construction pour Tas Min

Nous pouvons observer que comme attendu, **Construction** est plus rapide que **Ajouts Itératifs**.

Question 2.9

Afin de réaliser nos courbes de complexité pour la fonction **Union**, la somme d'éléments du tas est proche de la taille des fichiers de clés de cles_alea et utilise ce jeux de données en faisant

des moyennes sur 5 comme les courbes précédentes. L'utilisation de la fonction `process_time()` de la bibliothèque `time` était nécessaire pour obtenir le temps d'exécution du code exécuté.

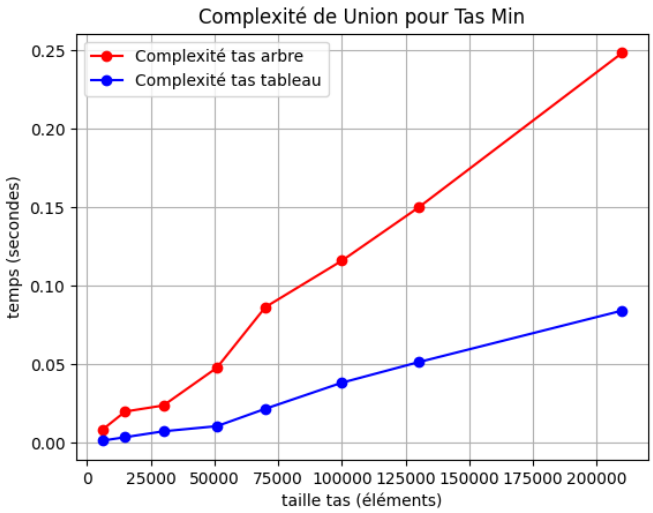

FIGURE 4 – Graphe de complexité de Union pour Tas Min

La courbe tracée devrait avoir une complexité en O(n+m), avec n et m le nombre d'éléments des deux tas. Nous pouvons observer que la structure du tableau est meilleure que celle de l'arbre.

3 Structure 2 : Files binomiales

Question 3.10

Pour implémenter une file binomiale, il est nécessaire d'implémenter un tas binomial. Afin de construire une structure de tas binomial, nous avons créé une classe **Node** pour la racine du tas binomial. La classe **Node** est uniquement constituée d'une clé.
Ensuite, la classe **TasBinomial** est définie par sa racine, de classe **Node**, son degré et une liste de tas enfants (**children**). Son degré est égal à la taille de la liste de ses enfants.

```
class Node:
    def __init__(self, cle):
        self.cle = cle

class TasBinomial:
def __init__(self,cle=None):
    if cle is None:
       self.racine = None
       self.degre = 0
       self.children = []
    else:
       self.racine = Node(cle)
       self.degre = 0
       self.children = []
```

Une File Binomiale est alors représentée par une liste de Tas Binomiaux.

```
class FileBinomiale:
def __init__(self, liste_tas=[]):
    self.liste = liste_tas
```

Les primitives de Files Binomiale sont les suivantes :

Algorithme 5 : estVide File Binomiale

Entrées : file
Sorties : Vérifie si la file est vide
retourner file.liste == []

Algorithme 6 : MinDegre File Binomiale

Entrées : file
Sorties : Le tournoi de degré minimal de file
Si estVide(file) :
  retourner TasBinomial()
else :
  min ← dernier tas de la file
  retourner min

Algorithme 7 : Reste File Binomiale

Entrées : file
Sorties : file privée de son tournoi de degré minimal si elle n'est pas vide, une file vide sinon
Si estVide(file) :
  retourner FileBinomial()
file ← file privée de son dernier tas binomial
retourner file

Algorithme 8 : AjoutMin File Binomiale

Entrées : file, tas
Sorties : Renvoie la file obtenue en ajoutant le tournoi tas comme tournoi de degré minimal
Si non estVide(tas) :
  file.liste.append(tas)
retourner file

Question 3.11

Voici les fonctions fondamentales d'une file binomiale :

- **SupprMin(file)** :
  — Si la file est vide, nous renvoyons None.
  — Nous utilisons **min_tas**, le tas ayant la plus petite clé, que nous trouvons à l'aide d'un parcours sur la liste de la file. Il suffit de comparer la racine des tas de la file pour trouver la clé min.
  — Nous retirons le **min_tas** de la liste grâce a la fonction `remove`.
  Nous récupérons ensuite la file binomiale obtenue en supprimant du **min_tas** sa racine, à l'aide de la fonction **Decapite** (**fileDecapite**). Enfin, nous utilisons **UnionFile** pour réaliser l'union entre **fileDecapite** et la file passée en paramètre.
  Nous renvoyons alors cette nouvelle file.

- **Ajout(file, tas)** :
  — Si la file est vide, il suffit d'ajouter le tas dans la liste de la file puis nous retournons la file.
  — Sinon, nous créons une file contenant le tas avec la fonction **File** (**file2**). Puis nous renvoyons l'union entre la file passée en argument et **file2**.

- **Construction(file, listeCles)** :
  — Nous parcourons listeCles, pour créer un **TasBinomial** à chacun des clés, et nous faisons **Ajout(tas)** du tas crée dans la file.
  Enfin, nous renvoyons la file.

- **UnionFile(file1, file2)** : Cette fonction construit la file union des files binomiales file1 et file2. Elle fait appel à la fonction **UFret** qui implémente l'algorithme donné dans le cours.

Nous pouvons remarquer que dans notre projet, la complexité de **Construction** est en $O(n\log(n))$

comme indiqué dans le sujet **"Construction se fait par ajouts itératifs"**. Or, l'opération d'ajout est en O(log n). Malheureusement, nous n'avons pas trouvé un algorithme de **Construction** suivant la complexité en O(n) du cours.

Question 3.12

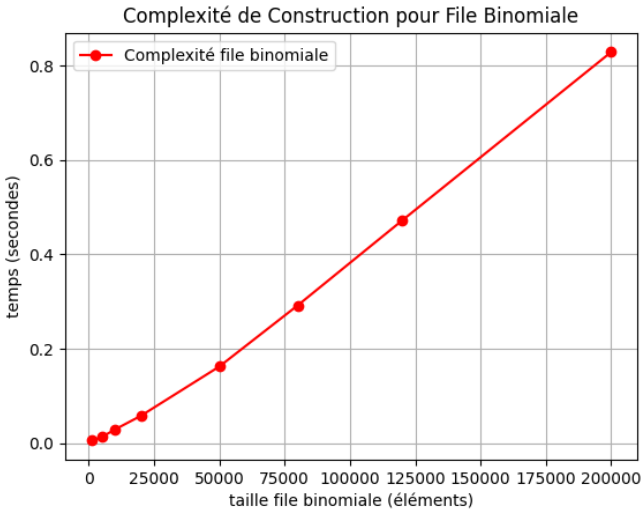

FIGURE 5 – Graphe de complexité de Construction pour File Binomiale

Cette courbe a été tracée comme les précédentes. Graphiquement, la courbe de complexité est une droite et semble être meilleure que celle de Tas Min arbre et tableau.

Question 3.13

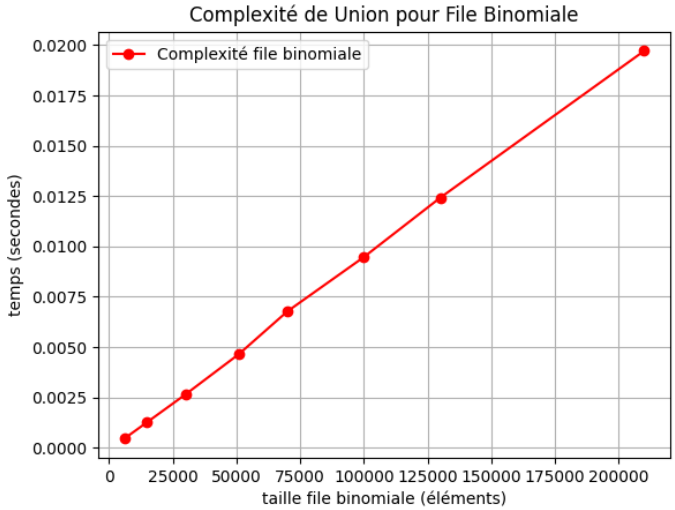

FIGURE 6 – Graphe de complexité de Union pour File Binomiale

La courbe ci-dessus a été tracée comme les précédentes. Nous pouvons remarquer qu'il aurait peut-être fallu générer des valeurs supplémentaires ou faire des unions sur des tas de taille plus importante afin d'observer l'allure d'un log sur le graphe.

4 Fonction de hachage

Le pseudo-code donné sur la page Wikipédia implémente la fonction de hachage MD5. Voici quelques explications :

- Nous commençons par initialiser **r** étant un tableau de 64 entiers.
- Nous initialisons les constantes **k** dans un tableau, grâce à la formule indiquée : $\lfloor |\sin(i+1)| \rfloor \times 2^{32}$.
- Nous initialisons quatre autres variables étant des entiers **h0**, **h1**, **h2**, **h3**.
- Pour réaliser le padding, il faut commencer par convertir le message en octet puis nous obtenons sa longueur en multipliant par 8. Nous ajoutons un bit **1** au message, suivi d'autant de bits **0** que nécessaire afin que la taille du message soit égale à 448 $\mod 512$ bits.
- Puis la taille du message du message est ajoutée en 64 bits little-endian.
- Par la suite, nous traitons des blocs de 512 bits du message, et dans chaque bloc, nous réalisons les étapes suivantes :
  - Chaque bloc est divisé en 16 mots de 32 bits.
  - Nous initialisons des valeurs de hachage qui sont les variables **a**, **b**, **c**, **d**
  - Par la suite, la boucle principale consiste à faire 64 tours de boucle à l'aide d'une variable **i**. Pour **i<=15**, **i<=31**, **i<=47**, **i<=63**, des opérations logiques et arithmétiques sont effectuées.
- Nous ajoutons alors les résultats trouvés dans la boucle précédente, aux blocs précédents.
- Enfin, nous retournons **empreinte** qui correspond à la concaténation de **h0**, **h1**, **h2**, **h3** convertie dans l'ordre little-endian, de longueur 4 octets (32bits).

5 Arbre de Recherche

Nous avons choisi d'implémenter un Arbre Binaire de Recherche (ABR) (non équilibré). Pour cela, nous avons une structure correspondant à un noeud. Cette classe prend en compte une clé, ainsi qu'un fils gauche et droit.

```
class NoeudABR:
    def __init__(self,cle):
        self.cle = cle
        self.gauche = None
        self.droite = None
```

Ensuite, l'implémentation de la structure arborescente de recherche aura besoin de la classe **NoeudABR**.

```
class ABR :
    def __init__(self, cle=None):
        if cle is None:
            self.racine = None
        else:
            self.racine = NoeudABR(cle)
```

Cette structure arborescente de recherche nécessite plusieurs fonctions comme primitives :

- **EstABRVide(abr)** : Vérifie si l'ABR abr est vide. Pour cela, nous vérifions si la racine de **abr** est égale à `None`.

- **AjoutABR(abr,cle)** : Cette fonction ajoute une clé à l'ABR abr. Pour cela :
  — Nous créons un **NoeudABR** contenant la clé.
  — Si la racine de **abr** est vide alors nous ajoutons le nouveau noeud à la racine.
  — Sinon :
    — Si la clé de la racine est inférieure à la clé à ajouter, il faut ajouter la clé dans l'enfant droit de abr. Si le fils droit de la racine vaut **None** alors nous insérons la clé. Sinon, nous continuons le parcours dans le fils droit avec **AjoutABRrec** pour insérer la clé.
    — Même traitement si la clé est plus petite que la racine, il faut alors insérer à gauche.
    — Sinon, cela signifie que la clé est déjà dans l'arbre.

  La fonction **AjoutABR** est utilisée comme point d'entrée du parcours puis appelle **AjoutABRrec**. **AjoutABRrec** est effectuée de manière récursive afin de parcourir et réaliser le travail d'ajout de la clé.

- **SupprABR(abr, cle)** : Cette fonction supprime la clé cle de l'arbre.
  — Si la racine de **abr** et la clé sont égales alors **supprRacine** est appelée pour retirer.
  — Sinon, **supprRec** est directement appelée.

  La fonction **supprRacine** permet de supprimer la racine s'il s'agit bien de l'élément à retirer. La fonction commence par traiter les cas de base :
  — Si le noeud est une feuille, nous renvoyons None.
  — Si le noeud courant n'a pas de fils gauche, nous renvoyons son fils droit et nous le retirons son fils droit.

— Si le noeud courant n'a pas de fils droit, nous renvoyons son fils gauche et nous le retirons son fils gauche.
— Nous remplaçons la racine par le minimum du sous-arbre droit en utilisant la fonction **min**. Puis nous supprimons le minimum du sous-arbre droit.

La fonction **supprRec** permet de supprimer le noeud de l'arbre. Elle s'effectue simplement avec des conditions :
— Si le noeud courant vaut **None**, il n'y a rien à supprimer.
— Si la clé du noeud et la clé sont identiques, nous utilisons **supprRacine** pour retirer le noeud.
— Si la clé est inférieure à la clé du noeud, nous faisons un appel récursif **supprRec** sur le fils gauche du noeud.
— Sinon, nous faisons **supprRec** sur le fils droit du noeud.
Cette fonction retourne le noeud.

- **RechercheABR(abr,noeud,cle)** : Cette fonction recherche si la clé dans l'ABR en vérifiant si elle est dans le sous-arbre courant (noeud).
  — Si le noeud vaut **None**, nous retournons **False**.
  — Si la clé du noeud courant (la racine du sous-arbre courant) et la clé sont égales, nous retournons **True**.
  — Si la clé est inférieure à la clé du noeud, nous faisons un appel récursif de **RechercherABR** en partant sur le fils gauche du noeud.
  — Sinon, nous réalisons un appel récursif **RechercherABR** en allant sur le fils droit du noeud.

  La complexité en moyenne de **RechercheABR** est en O(log n). Sauf dans le cas où nous avons simplement des arbres déséquilibrés de manière à ce que chaque noeud ait un seul descendant (un arbre peigne).

  Afin de vérifier si implémenter un ABR non équilibré suffisait pour obtenir des arbres plus ou moins équilibrés (pas trop déséquilibrés), ainsi que vérifier la complexité de recherche en moyenne en O(log n), nous avons mené deux expériences :

  - Réaliser l'ABR des mots uniques récupérés dans l'oeuvre de Shakespeare :
    — Nombre total de noeuds : 23 086
    — Hauteur minimale fils gauche : 6
    — Hauteur minimale fils droit : 5
    — Hauteur maximale fils gauche : 35
    — Hauteur maximale fils droit : 33
    — Nombre de noeuds fils gauche : 10 009
    — Nombre de noeuds fils droit : 10 076
  - Parcourir l'ensemble des fichiers "cles_alea" fournis et construire un ABR pour chaque fichier. Et nous obtenons les valeurs suivantes en faisant des sommes :
    — Nombre total de noeuds : 2 430 000
    — Hauteur minimale fils gauche : 204
    — Hauteur minimale fils droit : 204
    — Hauteur maximale fils gauche : 1194
    — Hauteur maximale fils droit : 1239
    — Nombre de noeuds fils gauche : 1 085 197
    — Nombre de noeuds fils droit : 1 344 763

Selon nos expériences, l'utilisation de l'arbre binaire de recherche est suffisante pour obtenir une recherche en O(log n) sur des données concrètes. Il n'est pas nécessaire de réaliser un arbre AVL ou un Arbre 2-3-4.

6 Étude expérimentale

Question 6.14

Afin de traiter les mots de l'oeuvre de Shakespeare, nous utilisons la fonction de hachage MD5 qui donne des valeurs en hexadécimal. Puis nous coupons ces valeurs en 4 afin de créer des clés 128.

Question 6.15

Sur l'ensemble des mots différents de l'oeuvre de Shakespeare, nous n'obtenons aucune collision pour MD5.
La raison s'explique par le fait que nous avons une probabilité assez basse de collision avec 1 chance sur 2^{128}. En particulier, par le paradoxe des anniversaires, il aurait fallu hacher 2^{64} valeurs pour qu'il y ait 1 chance sur 2 d'obtenir une collision. Or ici, nous n'avons que 23086 mots différents dans l'oeuvre.

Question 6.16

Pour la réalisation des histogrammes, nous avons d'abord construit les deux structures de Tas Min et la File Binomiale à l'aide des mots uniques de l'oeuvre de Shakespeare.

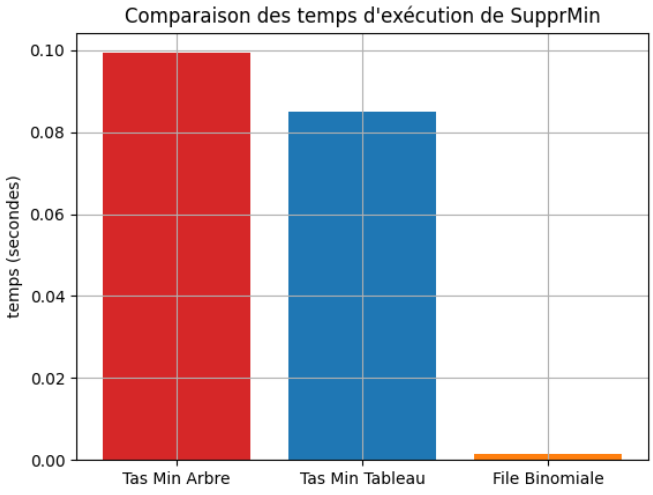

FIGURE 7 – Histogramme des temps d'exécution de SupprMin pour un Tas min et File Binomiale

L'histogramme représentant la suppression de l'ensemble des éléments des structures de données. Sans surprise, la structure du tableau est légèrement plus rapide que l'arbre. Malheureusement notre SupprMin pour la File Binomiale semble incorrecte, car lors de nos tests, elle effectue simplement 5 fois l'opération SupprMin dans notre cas. Nous n'avons pas trouvé l'erreur malgré le fait que nous pensons que nos étapes pour réaliser SupprMin soit correct.

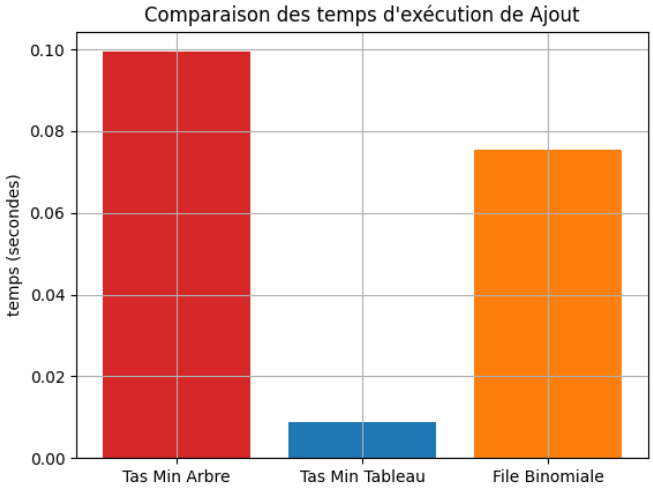

FIGURE 8 – Histogramme des temps d'exécution de Ajout pour un Tas min et File Binomiale

Nous pouvons observer que la structure du tableau est bien plus rapide que celle de l'arbre et de la file binomiale. De plus, l'ajout par file binomiale est plus rapide que la structure d'arbre.

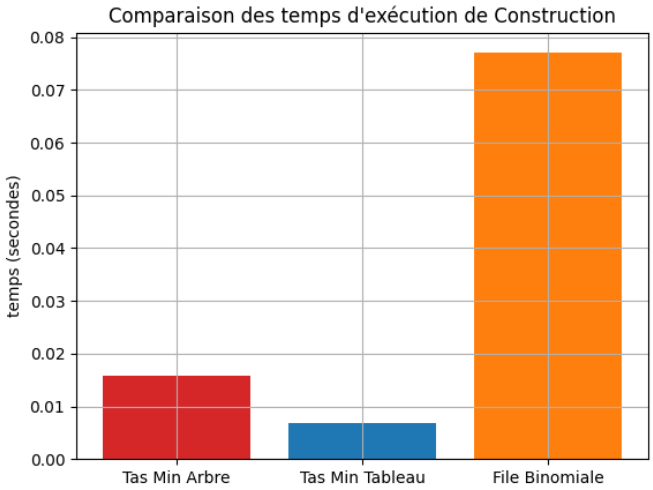

FIGURE 9 – Histogramme des temps d'exécution de Construction pour un Tas min et File Binomiale

Par cet histogramme, nous constatons que la Construction pour une file binomiale est bien plus lente que les deux structures de tas min. Le tas min tableau reste toujours plus rapide que les deux autres structures pour cette opération. Nous avons après tout implémenté un algorithme de Construction en O(n log(n)) contrairement aux deux implémentations pour Tas Min qui sont en O(n).

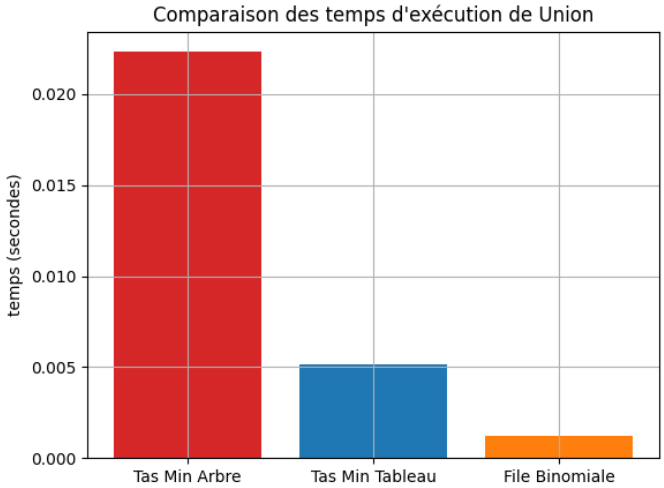

FIGURE 10 – Histogramme des temps d'exécution de Union pour un Tas min et File Binomiale

Cet histogramme montre que la structure de file binomiale est plus rapide que les deux autres. La raison est que la file binomiale effectue une union en complexité logarithmique, contrairement aux deux autres. Nous pouvons ainsi remarquer qu'en pratique, il serait plus intéressant de construire des files binomiale à l'aide de Union au lieu de Construction.

7 Annexe : Liste des fichiers

Fichiers de code :

- projet.py : code source
- tests.py : fichier des tests
- testsPerf_FileBino.py : fichier pour la partie de File Binomiale
- testsPerf_TasMin.py : fichier pour la partie de Tas Min
- Shakespeare.py :
- etudeExp : fichier pour la partie Etude experimentale

Fichiers de données et images :

- AjoutsIteratifs_TasMin.png
- Construction_FileBino.png
- Construction_TasMin.png
- Exp_Ajout.png
- Exp_Construction.png

— Exp_SupprMin.png
— Exp_Union.png
— tas_tab.png
— Union_FileBino.png
— Union_TasMin.png