Projet ALGAV

—--------CHEN SHIYAO 28707756
—--------YAPI ATSE 3803860

I.Introduction

Le projet présenté vise à explorer et expérimenter diverses structures de données et algorithmes avancés dans le domaine de l'informatique. Au cours de ce projet, nous allons aborder des sujets tels que les tas min, les files binomiales, les fonctions de hachage MD5 et les arbres de recherche, tout en mettant l'accent sur l'analyse expérimentale de ces structures dans des contextes réels.

II.Une reformulation du sujet courte:

Dans les 6 exercices réalisés jusqu'à présent, nous avons:

1. exercice 1: Nous définirons une représentation des clés 128 bits, cruciale pour l'ensemble des problématiques qui suivent. Nous explorerons également des opérations de comparaison entre ces clés.
2. exercice 2**:** Les tas min, implémentés à la fois sous forme d'arbres binaires et de tableaux, seront examinés. Nous aborderons les fonctions fondamentales telles que la suppression du minimum, l'ajout d'éléments, la construction efficace à partir d'une liste, et l'union de deux tas.
3. exercice 3: Une autre structure de données, les files binomiales, sera présentée. Nous examinerons leur utilisation dans les opérations de suppression, d'ajout, de construction et d'union, tout en analysant leur complexité temporelle.
4. exercice 4: Nous aborderons la conception et l'implémentation de la fonction de hachage MD5, une technique largement utilisée pour produire une empreinte codée sur 128 bits unique pour des données de taille variable.
5. exercice 5: Une structure arborescente de recherche sera développée pour fournir une méthode efficace de recherche d'éléments dans un ensemble de clés.
6. exercice 6: Enfin, nous réaliserons une étude expérimentale en utilisant les mots de l'œuvre de Shakespeare. Nous stockerons les hachés MD5 de chaque mot dans une structure arborescente de recherche tout en créant une liste des mots uniques. Nous comparerons graphiquement les temps d'exécution des algorithmes sur des données réelles.

III.La descriptions des structures manipulées

De nombreuses structures ont été manipulé:

- une structure Cle_entier, représentant les entiers codés sur 128 bits, qui est composé par un quadruplet de 4 entiers non signés de 4 octets

```c
typedef struct cle_entier{
    uint32_t u1; // bit de poids faible
    uint32_t u2;
    uint32_t u3;
    uint32_t u4;  // bits de poids forts
}Cle_entier;
```

- une structure Cle_cell, regroupant les entiers qui ont 128 bits sous forme de liste chaînée

```c
typedef struct ce_cell{
    Cle_entier* cle;
    struct ce_cell* suiv;
}Ce_Cell;
```

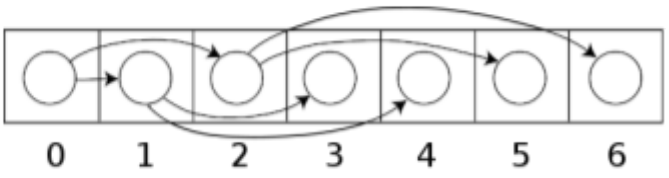
Un **tas min** est une structure de données utilisée notamment pour implémenter une file de priorité, on a 2 méthodes différentes pour le créer:

- une structure de tas min via arbre

```c
typedef struct tas_ab{
    Cle_entier* cle;
    struct tas_ab* fg;
    struct tas_ab* fd;
} Tas_Min_arbre;
```

- une structure de tas min vias tableau et et tab[i] ≤ tab[2i + 1] et tab[i] ≤ tab[2i + 2]

```c
typedef struct tas_min_tableau {
    int capacite; //case possible
    int taille;  //taille réelle
    //Ce_Cell* lcle;
    Cle_entier** tab;
} Tas_Min_tableau;
```



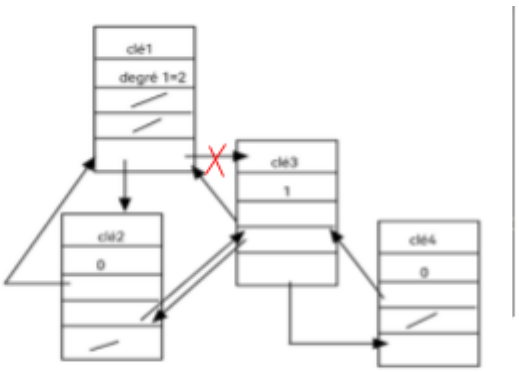
Un **file binomiale** est une structure de données en arbre utilisée pour représenter des tas binomiaux.

Voici quelques caractéristiques clés des files binomiales :

- une structure TournoisB, qui représente un arbre binomial
- une structure Tb_Cell, une liste chainéé de arbre binomiale, représentant une file binomiale

```c
//Tournois Binomiale Noued
typedef struct tournoisB{
    Cle_entier* cle; // Valeur st
    int degre;  // Degré du nœud
    struct tournoisB* parent; //
    struct tournoisB* enfant; //
    struct tournoisB* frere; // P
} TournoisB;
```

```c
//File Binomiale
typedef struct tb_cell{
    TournoisB* tb;
    struct tb_cell* suiv;
}TB_Cell;
```



exemple de tas binomiale

- une structure de AB_Rech, représentant un arbre de recherche

```c
typedef struct ab_rech{
    Cle_entier* cle;
    struct ab_rech* fils_g;
    struct ab_rech* fils_d;
    struct ab_rech* parent;
}AB_Rech;
```

Finalement , on a une structure pour regrouper les mot de l'œuvre de Shakespeare sous forme de liste chaînée

- une structure Words, représentant une liste chainée de mots

```c
typedef struct words{
    char* data;
    struct words* suiv;
}Words;
```

V. Les réponses aux questions:

Question 2.7

//via arbre

void ajout_tas_arbre(Tas_Min_arbre** tas, Cle_entier* cle) ⇒ O(n) (<>Cours = log(n))

Dans le pire cas, la fonction auxiliaire "ajout_cas_libre" insère la clé à la position libre (la dernière position) en visitant chaque noeud de l'arbre au plus deux fois.Le noeud du père est échangé avec le noeud le noeud qui vient d'être ajouté, et façon récursive avec son propre père. On fait au plus h (hauteur du tas) échanges à chaque ajout. Donc la complexité de la fonction "ajout_tas_arbre" est bien en O(n).

void ajout_iteratifs_tas_arbre(Tas_Min_arbre** tas, Ce_Cell* cles) ⇒ O(n²)
Dans le pire des cas, la fonction appelle n (taille de la liste de clés) fois la fonction ajout dont la complexité est en n. Donc la complexité de la fonction est bien en O(n²).

Cle_entier* suppr_min_tas_arbre(Tas_Min_arbre** tas) ⇒ O(log²(n)) (<> Cours : O(log(n))
Dans le pire des cas, la fonction appelle la fonction auxiliaire “dernier_elment” dont la complexité est en O(log²(n)) car dans celle-ci la fonction hauteur est appelée au plus h fois (h = hauteur de l’arbre). Après avoir récupéré la clef en dernière position, sa valeur est remplacée par celle de la racine. Ensuite l'élément en dernière position est détruit. Enfin le dernier élément mis à la racine est repositionné dans le tas en O(h). Donc la complexité de la fonction Suppr_min_tas_arbre est bien en O(log²(n)).

Tas_Min_arbre* construction_tas_arbre(Ce_Cell* cles) ⇒ O(n)
Dans le pire des cas, la fonction appelle la fonction auxiliaire “taille” dont la complexité est en O(n) (n=taille de la liste de clés). La complexité de la fonction “Construction_sous_arbre” est en O(2^(ht-1)) (ht = nombre de nœuds maximale sur un niveau de l’arbre). “Construction_sous_arbre” est appelée au plus h fois, en effet elle se charge de construire les sous arbres du tas à chaque niveau de l’arbre en partant du dernier niveau. Enfin, la fonction “equilibrer_tas_arbre” dont la complexité est en O(n) est appelée, réorganiser le tas afin de conserver la structure de tas priorité min. Donc la complexité de la fonction construction est bien en 0(n).

Tas_Min_arbre* union_tas_arbre(Tas_Min_arbre* tas1, Tas_Min_arbre* tas2) ⇒ O(n1+n2)
(n1 : taille du tas1, n2 : taille du tas2)
La complexité de la fonction auxiliaire “tas_arbre_en_liste” est en O(n) (n=taille du tas considéré) car elle parcourt le tas et récupère la clé de chaque nœud du tas. “tas_arbre_en_liste” est appelé 2 fois afin de récupérer les clés des tas 1 et tas 2. Enfin, la fonction “construction_tas_arbre” dont la complexité est en O(n) est appelée avec en paramètre la liste de clés de taille n1+n2. La complexité de la fonction union_tas_arbre est bien en O(n1+n2)

//via tableau

fonctions fondamentales:

void Ajout_Tab(Tas_Min_tableau* tas, Cle_entier* cle) ==>O(log n)

Dans le pire cas, la fonction doit remonter le tableau du tas jusqu'à la première case pour maintenir la propriété du tas min.

void Ajout_It_tab(Tas_Min_tableau* tas, Ce_Cell* lcle) ==> O(n log n)

La fonction Ajout_Tab est appelée pour chaque élément de la liste, et chaque appel a une complexité logarithmique O(log n)

void entasserMin(Tas_Min_tableau* tas, int indice) ==> Complexité : O(log n)

La fonction réorganise le tas à partir de l'indice donné pour maintenir la propriété du tas min.

unsigned int SupprMin(Tas_Min_tableau* tas) ==>Complexité : O(log n)

La fonction supprime l'élément minimum (racine) du tas et réorganise le tas pour maintenir la propriété du tas min,

il appele la fonction entasserMin dont sa complexité est O(log n).

construction:

Supposons que la hauteur du tas minimum soit h et que la profondeur actuelle soit i. Alors, le nombre total de nœuds parents est 2^h - 1, et chaque profondeur a 2^ i nœuds. Chaque opération sur le tas nécessite deux comparaisons (entre les nœuds enfants et entre le nœud enfant le plus grand et le nœud parent), et un total de i-1 opérations sont effectuées. Ainsi, la pire complexité est 2*i, et la meilleure est 2.

La complexité du pire cas est donc :

$2^0 \cdot h + 2^1 \cdot (h - 1) + 2^2 \cdot (h - 2) + ... + 2^{h-2} \cdot 2 + 2^{h-1} \cdot 1 + 2^h \cdot 0$

$=h + 2^1 \cdot (h - 1) + 2^2 \cdot (h - 2) + ... + 2^{h-2} \cdot 2 + 2^{h-1}$

Soit la formule ci-dessus s(h):

$s(h)=2s(h)-s(h) = -h + 2^1 + 2^2 + 2^3 + ... + 2^{h-1} + 2^h$

Selon la séquence géométrique, on obtient \($s(h) = s(h)=2^{h+1} - (h + 2)$.

Le nombre de nœuds n satisfait $n <= 2^{h+1}-1$, donc $n <= 2^{h+1}$. Ainsi,$s(h)=2^{h+1} - (h + 2)>=n$. Ainsi, la complexité temporelle dans le pire des cas est d'au moins O(n).

union:

Tas_Min_tableau* Union_tab(Tas_Min_tableau* tas1, Tas_Min_tableau* tas2);

==> O(m+n)

La complexité de cette fonction est dominée par l'appel de la fonction fusion et la reconstruction du tas à partir de la fusion.

1.Appel à fusion :

Complexité : O(m + n), où m est la taille de tas1->tab et n est la taille de tas2->tab

2.Reconstruction du tas avec remonter :

Complexité : O(m + n), où m est la taille de tas1->tab et n est la taille de tas2->tab.

La complexité totale de la fonction union_tab est donc O(m + n), où m et n sont les tailles des deux tas respectifs.

Cle_entier **fusion(Cle_entier **T1, int n1, Cle_entier **T2, int n2) ==> O(n1+n2)

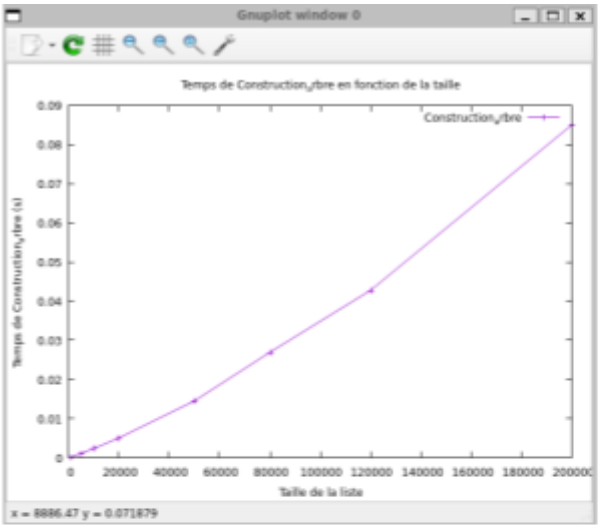
Cette fonction a une complexité linéaire par rapport à la somme des tailles des deux tableaux.

La complexité est O((n1+n2), où n1 est la taille de T1, et n2 est la taille de T2.

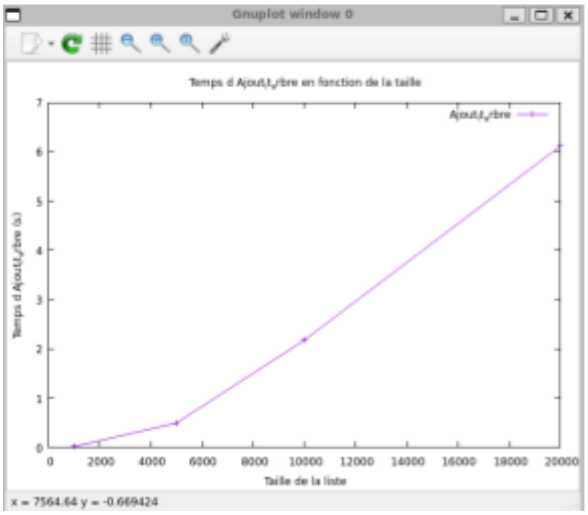
Et puis on verifier avec les complexités temporelles des fonctions

Question 2.8/Question 2.9

(1) Graphe du temps d'exécution de la construction et de l'ajout itératif du tas min par la structure arborescente

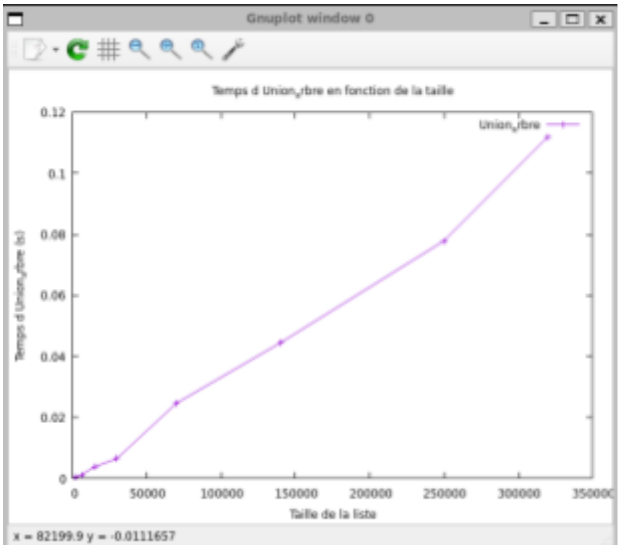


taille	Construction (arbre)
1000	0.000207
5000	0.001143
10000	0.002354
20000	0.005044
50000	0.014525
80000	0.026945
120000	0.042755
200000	0.085074



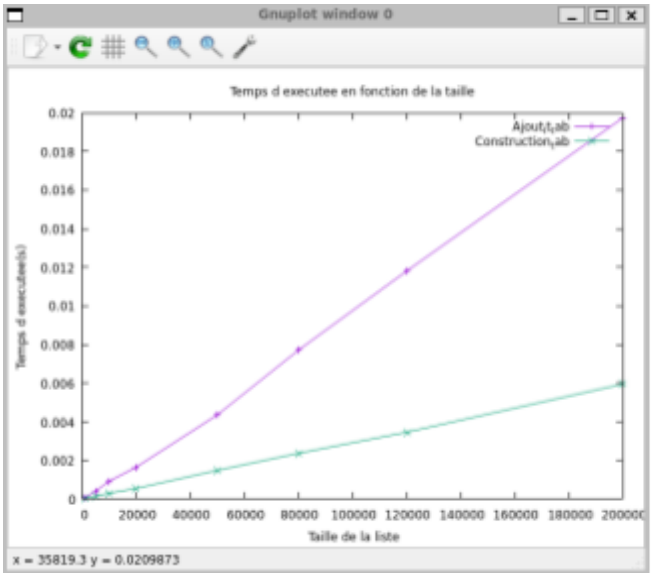
taille	ajout itératif (arbre)
1000	0.018265
5000	0.495164
10000	2.177184
20000	6.1155872

(2) Graphe du temps d'exécution de l'opération d'Union sur deux tas min représentés par la structure arborescente



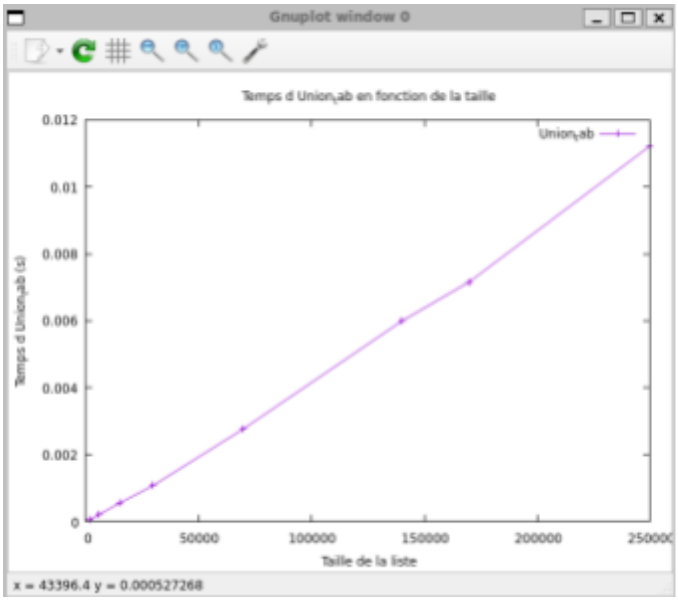
taille	union (arbre)
2000	0..000480
6000	0.001193
15000	0.003889
30000	0.006552
70000	0.024651
140000	0.044444
250000	0.077825
320000	0.111935

(3) Graphe du temps d'exécution de la construction et de l'ajout itératif du tas min par la structure tableau



taille	ajout itératif (tableau)	construction (tableau)
1000	0.000071	0.000028
5000	0.000403	0.000151
10000	0.000919	0.000299
20000	0.001646	0.000547
50000	0.004355	0.001484
80000	0.007707	0.002369
120000	0.011804	0.003460
200000	0.019742	0.005954

(4) Graphe du temps d'exécution de l'opération d'Union sur deux tas min représentés par la structure tableau

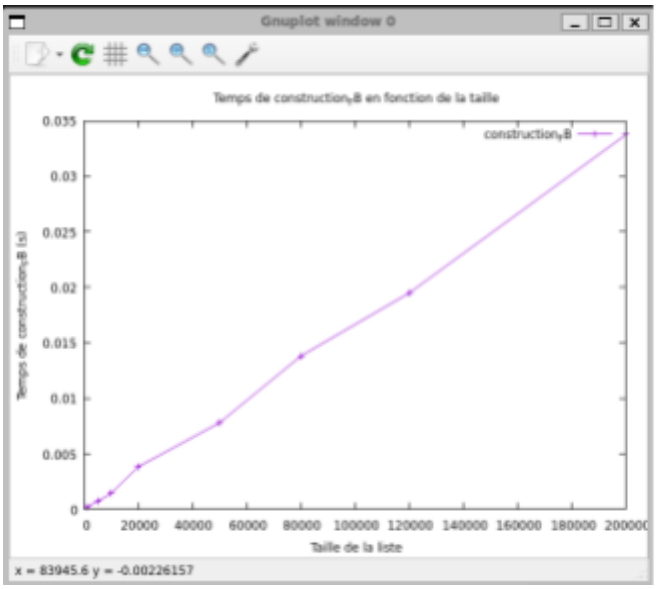


taille	union (tableau)
2000	0.000067
6000	0.000219
15000	0.000558
30000	0.001087
70000	0.002776
140000	0.005987
170000	0.007159
250000	0.011212

On sait que les complexités théoriques de la Construction et de l'Ajout Itératif sont respectivement O(n), représentant une droite, et O(n log n) (dans le pire cas). Sur le graphe 3, on observe que la courbe de la Construction forme une droite située en dessous de la courbe de l'Ajout Itératif, qui est également une droite. C'est normal, car l'ajout itératif commence avec un arbre vide et son coût amorti est O(1), donc sa complexité est O(n).De plus, la complexité de l'opération Union est en O(n+m), représentant également une droite, ce qui est conforme au graphe 4.

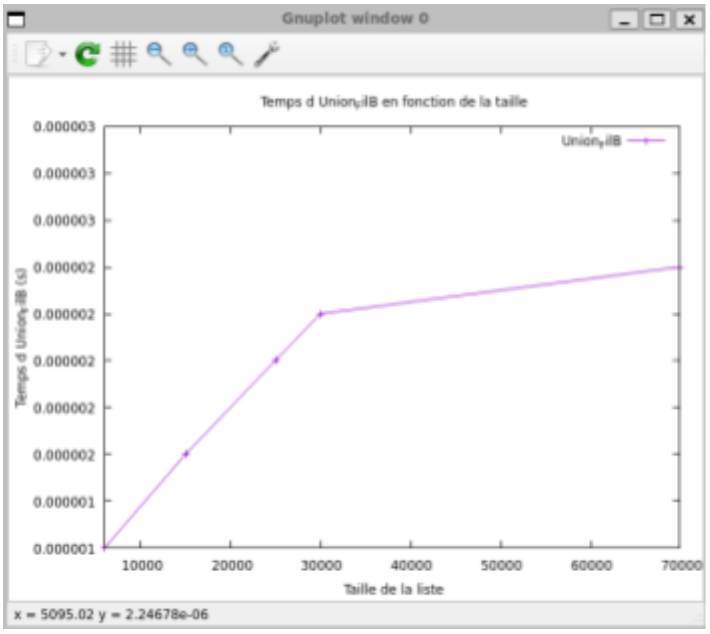
Question 3.12 / Question 3.13

(5) Graphe du temps d'exécution de la construction de la file binomiale



taille	construction(file binomiale)
1000	0.000256
5000	0.000723
10000	0.001458
20000	0.003845
50000	0.007806
80000	0.013812
120000	0.019484
200000	0.033744

(6) Graphe du temps d'exécution de l'opération d'Union sur deux files binomiales



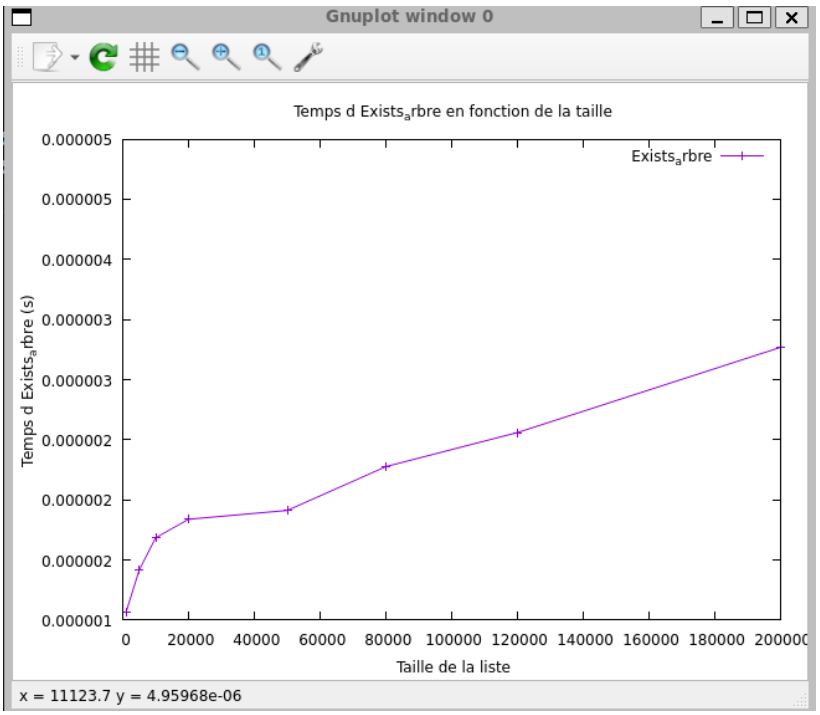
taille	union (tableau)
6000	0.0000012
15000	0.0000016
25000	0.0000020
30000	0.0000022
70000	0.00000242

On sait que la complexité théorique de la construction de la file binomiale est O(n), ce qui représente une droite. Du coup, elle confirme la courbe du graphe 5.

On constate que la courbe croît plus lentement que celle de n, ce qui correspond à la propriété de log n

Question 5

(7) Graphe du temps d'exécution de l'opération de recherche d'une clé dans un arbre binaire de recherche.

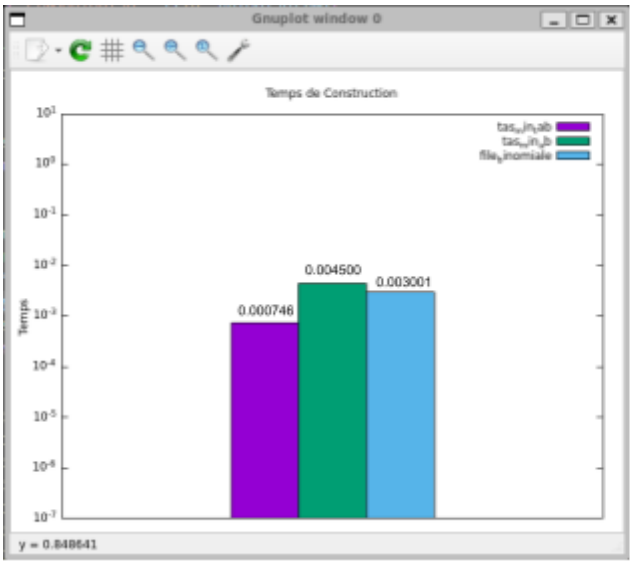


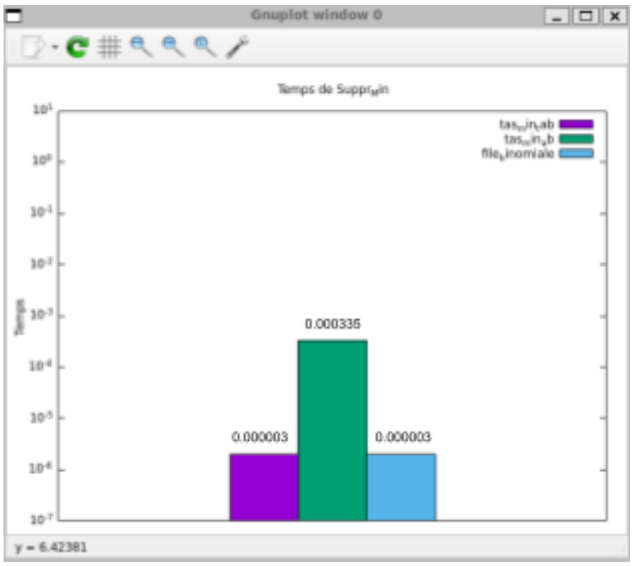
Question 6.15

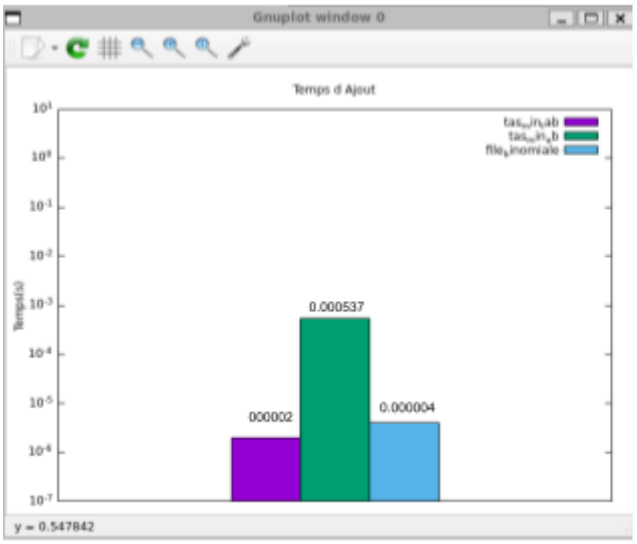
On constate qu'il y a 23 086 mots distincts dans l'œuvre de Shakespeare, et après l'exécution de la fonction de hachage MD5, on ne trouve aucun hachage identique, car les mêmes chaînes de caractères ne se trouvent pas dans la liste de mots. Ainsi, les collisions ne sont pas détectées pour MD5.

Question 6.16

Comparaison graphique des complexité temporelles : Tas Min & File Binomia







On trouve que les temps d'exécution du tas min et la file binomiale sont presque identiques, car ils ont la même complexité. Mais pour fusionner deux tas, la file binomiale est plus rapide car O(n+m) < O(n+m), où n et m sont les tailles des deux tas respectifs.

VI.Conclusion

Après ce projet, nous avons une meilleure compréhension des contenus des cours par rapport aux performances des structures de données et algorithmes examinés. Les tas min et les files binomiales, en tant que structures de données de base, ont démontré leur efficacité pour les opérations fondamentales telles que l'ajout, la suppression et la construction.

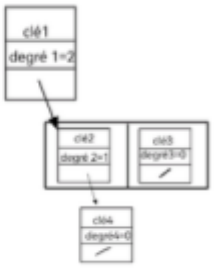
Les fonctions de hachage MD5 ont été utilisées avec succès pour créer des empreintes uniques des mots de l'œuvre de Shakespeare, démontrant ainsi leur utilité dans le contexte de la recherche et de l'intégrité des données. L'utilisation d'arbres de recherche a également été évaluée, mettant en évidence leur capacité à offrir une recherche efficace avec un temps logarithmique et à éliminer les mots répétés.

Il a été constaté que la file binomiale est la meilleure structure de données pour cette expérimentation.

Les chose corrigée après la soutenance:

Je refais l'exercice 3 avec la nouvelle structure dans EX_3.c

```
typedef struct tournoisB{
    Cle_entier* cle; // Valeur
    int degre;
    struct tournoisB* enfant; /
} TournoisB;
```



Mais après l'avoir testée, elle ne s'est pas révélée plus efficace que l'ancienne. Les complexités temporelles des fonctions sont presque les mêmes.

Et on change les graphes par rapport au tas min via arbre car on trouve que il y a problème pour les complexités alors on modifier les codes(ex3 et exo6).

On utilise codespace de github pour tester les données car c’est plus vite que mon ordinateur.