Sorbonne Université - Master STL

MU4IN500 - ALGAV

Année 2023-2024

Devoir de Programmation

Langage C

Alex XU & Stéphanie HUANG

SOMMAIRE

1. Echauffement.....2
2. Structure 1 : Tas priorité min.....2
3. Structure 2 : Files binomiales.....13
4. Fonction de hachage.....16
5. Arbre de Recherche.....16
6. Etude expérimentale..... 17
7. Conclusion.....21

1. Echauffement

Question 1.1

Pour représenter les entiers codés sur 128 bits, on suivra l'exemple donné dans l'énoncé. On utilisera donc un quadruplet de *4 entiers non signés de 32 bits* pour représenter notre clé de 128 bits.

```
struct clef128{
    unsigned int b32_1; //poids faibles
    unsigned int b32_2;
    unsigned int b32_3;
    unsigned int b32_4; //poids forts
    char* clef_hexa; // cles en hexa
};
```

On utilisera notre fonction *hexaToUnsigned* qui prend en entrée une chaîne de caractère (qui est un entier 128 bits hexadécimal), elle transformera l'hexadécimal en Clef128, c'est à dire en un quadruplet de 4 entiers non signés de 32 bits et le renverra.

Question 1.2

Le prédicat *inf*, prend en entrée 2 clés 128 bits, il commence par comparer les entiers de 32 bits de poids fort de chacun des 2 clés. Si la clef1 est inférieure à la clef2 on renvoie 1, sinon on regarde s'il est supérieur à la clef2 et on renvoie 0 si c'est le cas. Et ainsi, si les 2 entiers de 32 bits sont égaux, on exécute les mêmes opérations de comparaisons pour chacun des entiers de 32 bits suivants, jusqu'au dernier de poids faible.

Question 1.3

Le prédicat *eg*, prends en entrée 2 clés 128 bits, on comparera chaque paires d'entiers de 32 bits de la clef1 et de la clef2, si la paire n'est pas égale on renvoie 0, sinon on compare les paires suivantes, si elles sont toutes égales on renvoie 1.

2. Structure 1 : Tas priorité *min*

Question 2.4

```
struct tasTableau {
    Clef128 * tab;
    int taille;
    int capacite;
};
```

- *Clef128 supprMinTasTableau (TasTableau * tas)* :

On commence par vérifier que le tas n'est pas vide, si c'est le cas on renvoie une clé nulle. Sinon on commence par retirer la racine et on la remplace par le dernier élément inséré. Ensuite on rééquilibre le tas, pour respecter les propriétés de la structure. Notre fonction *reequilibrerTas*, prend le tas et l'indice du nœud à descendre à la bonne profondeur pour respecter la propriété du tas minimum. On regardera donc pour l'indice 0, qui est l'élément à faire descendre. On le compare à son fils gauche et droit, on échangera avec le fils minimum, sinon l'arbre est équilibré. Si on a effectué un échange, on fait une récursion jusqu'à ce que les fils du nœud ne lui soient plus inférieurs. On retourne la clé supprimée.

- *void ajout (TasTableau * tas, Clef128 clef)* :

Pour ajouter un élément au tasTableau, il suffit de faire un ajout en queue du tableau et pour respecter la propriété du tas minimum, on remonte la clef jusqu'à ce que son nœud parent lui soit inférieur.

- *TasTableau * ajoutsIteratifs (Clef128 * clefs[], int deb, int fin)* :

Comme son nom l'indique il s'agit de faire une suite d'ajout itérativement dans le tas. Dans une boucle, on appellera successivement la fonction *ajout* sur le tas créer que l'on renverra.

Question 2.4 Bis

```
struct tasArbre {
    Clef128* clef;
    struct tasArbre * fg;
    struct tasArbre * fd;
    struct tasArbre * parent; // parent du noeud
    int hauteur; // hauteur de l'arbre
    int noeud; // nombre de noeud present dans l'arbre
};
```

- *Clef128 supprMinTasArbre (TasArbre * tas, TasArbre * racine)* :

La fonction supprMinTasArbre est résursive. Elle prend en paramètre 2 Tas Arbre*, l'un pour parcourir l'arbre et récupérer le dernier nœud ajouté et l'autre pour récupérer la racine, le nœud minimum pour pouvoir faire l'échange lors de sa suppression.
On va donc parcourir l'arbre pour récupérer le dernier en fonction du nombre de nœuds dans les sous arbres.

<u>Cas1:</u>

- Si le sous arbre gauche est rempli, c'est à dire n = (2^h)-1 avec n : le nombre de nœuds et h : la hauteur de l'arbre, on vérifie le sous arbre droit. Si le sous arbre droit est rempli, on vérifie si les feuilles des 2 sous arbres sont au même niveau. Si c'est le cas, on appelle la fonction sur le sous arbre droit.

<u>Cas2:</u>

- Si les feuilles des 2 sous arbres ne sont pas au même niveau, on appelle la fonction sur le sous arbre gauche.

<u>Cas3:</u>

- Si le sous arbre droit n'est pas rempli, on appelle la fonction sur le sous arbre droit.

<u>Cas4:</u>

- Si le sous arbre gauche n'est pas rempli, on appelle la fonction sur le sous arbre gauche.

Lorsque le dernier nœud est trouvé, on met à jour la hauteur et le nombre de nœuds dans les nœuds parents avant de le retirer. Puis on fait l'échange avec la racine et on rééquilibre l'arbre en plaçant au bon endroit le dernier nœud tel que le parent doit être inférieur au nœud.

- *void ajout (TasArbre * tas, Clef128 clef)* :

De la même manière que la fonction de suppression, on va ajouter les nœuds soit dans le sous arbre gauche soit dans le sous arbre droit selon si le sous arbre est rempli ou non. Lors de l'ajout, on va mettre à jour la hauteur et le nombre de nœuds dans les nœuds parents et on effectue un rééquilibrage à la remontée.

Le rééquilibrage à la remontée échange les clefs avec celle du parent si inférieur et rappelle la fonction sur les nœuds parents jusqu'à la racine.

- *TasArbre * ajoutsIteratifs (Clef128 * clefs[], int len)* :

La fonction prend en entrée un tableau de clés et renvoie un TasArbre contenant les clés. Elle va appeler successivement la fonction ajout, len fois, et ajouter chaque clés du tableau dans le tas initialisé au début de la fonction.

Question 2.5

*TasTableau *construction (Clef128 * clefs[], int deb, int fin)* :

La fonction construction comme pour *ajoutsIteratif,* ajoute successivement les clés dans le tas, sans utiliser la fonction d'*ajout* qui en plus d'insérer les clés, rééquilibre le tas pour respecter la propriété du tas minimum. On effectue le rééquilibrage seulement après l'insertion de toutes les clés. Le rééquilibrage se fait sur tous les nœuds parents. On renvoie ensuite le tas rééquilibré.

Pseudo code :

```
TasTableau * constructionTasTableau (Clef128 * clefs[], int taille)
{
    TasTableau * tas = initTas(taille);
    //remplir le tas par insertion successifs sans respecter la propriété du tas
    min O(n)
    pour i allant de 0 à taille :
        tas->tab[i] = *clefs[i];
        tas->taille++;
    //on regarde si tous les noeuds parents sont plus petit que leurs enfants sinon
    on doit échanger les noeuds O(logn)
    pour i allant de (taille-1)/2 à 0 :
        reequilibrerTas(tas, i);
    return tas;
}

void reequilibrerTas (TasTableau * tas, int i)
{
    // il faut comparer i avec ses enfants, si les enfants sont inférieur à i alors
    on échange et on répète l'opération
    int min = i;
    int gauche = 2*i+1;
    int droite = 2*i+2;
    si le noeud gauche existe et si la clef gauche est inférieur à la clef min :
        min = gauche;
    si le noeud droite existe et si la clef droite est inférieur à la clef min :
        min = droite;
    // Si le minimum a changé, on échange et on rééquilibre le sous-arbre
    si min a changé :
        i echange avec son fils le plus petit
        reequilibrerTas(tas, min);
    }
```

```
}
```

Question 2.5 Bis

*TasArbre *construction (Clef128 * clefs[], int len)* :

La fonction prend en entrée un tableau de clés et renvoie un TasArbre construit en O(n).

Pour avoir une complexité en O(n) avec n le nombre de clés, l'ajout doit être en O(1) et le rééquilibrage inférieur à O(n).

Pour cela, on procède différemment en effectuant un parcours en largeur et en ajoutant les clés successivement. De cette manière, on aura une complexité en O(1) pour l'ajout de n clés. Après avoir ajouté toutes les clés, on rééquilibre seulement les nœuds parents de l'arbre, donc n/2 nœuds. Ce qui nous donne une complexité en O(logn).

On créer 2 structures pour le parcours en largeur:

- une liste doublement chaînée contenant un noeud de l'arbre
- une liste qui contient 2 pointeurs, un sur le premier élément et l'autre sur le dernier élément de la liste doublement chaînée

```
typedef struct Element Element;
struct Element {
    TasArbre* noeud;
    struct Element* suiv;
    struct Element* pre;
};

typedef struct Liste Liste;
struct Liste
{
    Element* tete;
    Element* queue;
};
```

Pseudo code:

```
TasArbre * construction(Clef128 * clefs[], int len) :
      //Cas de base
      TasArbre * premier = creerNoeud(clefs[0])
      // Le 1er élément de la liste est la racine du tas
      Liste * liste->tete = ajoutListe(premier)
      liste->queue = liste->tete
      //boucle principale
      pour i allant de 1 à len :
            si le sous arbre gauche de la tête de liste est vide:
                  sous arbre gauche  = creerNoeud(clefs[i])
                  ajout du nouveau noeud dans la liste du parcours
                  mis à jour du pointeur queue sur le nouveau élément
                  mis à jour du pointeur précédent du nouveau noeud
            sinon
                  si le sous arbre droit de la tête de liste est vide:
                        sous arbre droit = creerNoeud(clefs[i])
                        ajout du nouveau noeud dans la liste du parcours
                        mis à jour du pointeur queue sur le nouveau élément
                        mis à jour du pointeur précédent du nouveau noeud
                        liste->tête = liste->tête->suiv
```

```
        reequilibrageDescente(liste des éléments du parcours, taille de la liste)
        retourner la tête de la liste de départ
```

Question 2.6

La fonction *UnionTasTableau* prend en argument deux TasTableau et renvoie l'union des deux.
On commence par insérer n éléments du tas1 en O(1), puis m éléments du tas2 aussi en O(1) dans le nouveau TasTableau créé. On finit ensuite par rééquilibrer le tas à l'instar de la fonction *construction,* avec la fonction *rééquilibrer* qui se fait en O(logn).
Ainsi la complexité de la fonction *UnionTasTableau* s'effectue bien en temps linéaire O(n+m).

```
TasTableau * UnionTasTableau(TasTableau *tas1, TasTableau *tas2) {
    TasTableau * tasUnion = initTas(tailleTotale);
    Pour i allant de 0 à taille(tas1) :
        insérer tas1[i] à la suite du tableau de tasUnion
    Pour i allant de 0 à taille(tas2) :
        insérer tas2[i] à la suite du tableau de tasUnion
    Rééquilibrer le TasUnion
    return tasUnion;
}
```

Question 2.6 Bis

Afin de garantir la complexité linéaire en la somme des tailles des deux tas, on parcourt chaque tas et on insère chaque nœud dans un nouveau tas. Le parcours se fait en largeur pour pouvoir insérer les nœuds aux bons endroits et obtenir l'ajout d'un nœud en 0(1). On a donc l'ajout des clefs des deux tas en O(n+m) avec n, la taille du tas1 et m, la taille du tas2.
Le rééquilibrage se fait à la toute fin, en O(logn) car la fonction est appelée seulement sur les nœuds parents donc n/2 nœuds.
Nous pouvons également implémenter la fonction différemment en appelant la fonction construction() de la question précédente. On récupère les clefs des 2 tas à fusionner, puis on appelle la fonction construction sur la liste de clefs. La complexité de la fonction est bornée par celle de construction qui est en O(n) avec n, le nombre clef. Nous avons une complexité totale de la fonction Union en O(n+m).
Cette implémentation nous semble plus optimisée que la première version. Cependant lorsque l'on a testé le temps d'exécution, l'algorithme prenait davantage de temps à compiler que la première version.
Nous avons donc préféré prendre la première version pour l'étude expérimentale.

Pseudo code_v1:

```
TasArbre * Union(TasArbre * tas1, TasArbre * tas2) :
        //Cas de base
        TasArbre * premier = creerNoeud(tas1->clef)
        TasArbre * premier2 = creerNoeud(tas2->clef)
        // Le 1er élément de la liste est la racine du tas
        // newliste contient à la fin de la fonction tous les noeuds de tas1 et tas2
        // copy contient les noeuds de tas1
        // copy2 contient les noeuds de tas2
        Liste * newliste->tete = ajoutListe(premier)
        newliste->queue = newliste->tete
```

```
	Liste * copy->tete = ajoutListe(premier)
	copy->queue = copy->tete
	Liste * copy2->tete = ajoutListe(premier2)
	copy2->queue = copy2->tete
	// 1ère boucle parcourant tas1
	tant que copy->tete not NULL:
		// parcours en largeur pour ajouter les noeuds de tas1
		si sous arbre gauche non vide:
			ajouter noeud gauche dans copy
			mettre à jour la fin de la liste // pointeur queue
		si sous arbre droit non vide:
			ajouter noeud droit dans copy
			mettre à jour la fin de la liste

		copy->tete = copy->tete->suiv // on passe au suivant ici car le premier élément est déjà ajouté dans la nouvelle liste (cas de base)
		si copy->tete is NULL:
			break //car il y a un décalage d'une itération à cause du 1er élément ajouté dans le cas de base

		// ajout des noeuds dans la nouvelle liste
		si sous arbre gauche de la tete de newliste vide:
			creerNoeud(copy->tete->clef)
			ajouter le nouveau noeud dans la liste newliste
			mettre à jour le pointeur de fin de liste
			mettre à jour le pointeur précèdent du nouveau noeud
		sinon
			si sous arbre droit de la tete de newliste vide:
				creerNoeud(copy->tete->clef)
				ajouter le nouveau noeud dans la liste newliste
				mettre à jour le pointeur de fin de liste
				mettre à jour le pointeur précèdent du nouveau noeud
				passer au noeud suivant dans newliste

	//fin tant que de tas1

	tant que copy2->tete not NULL:
		// parcours en largeur pour ajouter les noeuds de tas2
		si sous arbre gauche non vide:
			ajouter noeud gauche dans copy2
			mettre à jour la fin de la liste // pointeur queue
		si sous arbre droit non vide:
			ajouter noeud droit dans copy2
			mettre à jour la fin de la liste

		// ajout des noeuds dans la nouvelle liste
		si sous arbre gauche de la tete de newliste vide:
```

```
                    creerNoeud(copy2->tete->clef)
                    ajouter le nouveau noeud dans la liste newliste
                    mettre à jour le pointeur de fin de liste
                    mettre à jour le pointeur précèdent du nouveau noeud
             sinon
                    si sous arbre droit de la tete de newliste vide:
                           creerNoeud(copy2->tete->clef)
                           ajouter le nouveau noeud dans la liste newliste
                           mettre à jour le pointeur de fin de liste
                           mettre à jour le pointeur précèdent du nouveau noeud
                           passer au noeud suivant dans newliste
             copy2->tete = copy2->tete->suiv

      //fin tant que tas2

      reequilibrageDescente(newliste, taille de la liste)
      retourner la tête de la liste de départ
```

Pseudo code_v2:

```
void * UnionParcours(ListeChainee * liste, TasArbre * tas) :
      si tas is NULL:
            sortir
      liste = insertMap(liste, tas->clef);
      UnionParcours(liste->suiv, tas->fg);
      UnionParcours(liste->suiv, tas->fd);
//fin UnionParcours

TasArbre * Union(TasArbre * tas1, TasArbre * tas2) :
      ListeChainee* liste = NULL;
      UnionParcours(liste, tas1);
      UnionParcours(liste, tas2);
      int taille = sizeMap(liste) // recupère la taille de la liste
      Clef128* clefs[taille];
      int i = 0;
      tant que liste is not NULL:
            clefs[i] = liste->clef;
            i++;
            liste = liste->suiv;
      // fin tant que
      retourner construction(clefs, taille);
```

Question 2.7

Complexité théorique	Tas	File Binomiale
Supp Min (1 élément parmi n)	O(log n)	O(log n)
Ajout (1 élément parmi n)	O(log n)	O(log n)
Construction (n éléments)	O(n)	O(n)
Union (n éléments et m éléments	O(n+m)	O(log(n+m))

TasTableau :
SupprMin:
Analyse pire cas :
→ L'accès à l'élément minimum du tableau se fait en O(1) a l'indice 0.
→ Pour récupérer le dernier nœud ajouté, l'accès se fait également en O(1) a l'indice taille-1 ou taille correspond au nombre de nœuds ajoutés dans le tableau.
→ L'échange de clef est en O(1).
→ Le rééquilibrage du tas est effectué en O(logn).
La complexité la plus grande est celle du rééquilibrage, nous avons donc une complexité totale en O(logn).

Ajout:
Analyse du pire cas :
→ L'ajout se fait à la fin du tableau, nous avons donc une complexité en O(1)
→ Puis nous effectuons des échanges successivement avec les noeuds parents.
Nous obtenons une complexité en temps logarithmique, O(logn).

AjoutsItératifs:
Analyse du pire cas :
→ Nous faisons n ajouts successifs en appelant la fonction ajout() en O(logn).
Ainsi nous avons donc une complexité totale en O(nlogn).

Construction:
Analyse du pire cas :
→ Nous appelons la fonction initTas() en O(1) qui permet d'allouer n (nombre de clefs) cases du tableau
→ Nous effectuons ensuite n insertions en O(1) dans le tas, ce qui revient à faire O(n) opérations.
→ Le rééquilibrage du tasTableau est effectué à la fin de la construction sur la moitié des nœuds seulement.
Le rééquilibrage étant en O(logn), et la boucle principale en O(n), nous obtenons une complexité totale en O(n).

Union:
Le pire cas est représenté par la fusion de deux tas de tailles différentes.
Analyse du pire cas :
→ Nous appelons la fonction initTas() en O(1).
→ Nous parcourons les deux tas en entrée afin d'ajouter les nœuds dans le nouveau tableau en O(1). La construction du tableau final est donc en O(n+m) avec n, la taille du premier tas et m, la taille du deuxième tas.
→ Le rééquilibrage en O(logn) s'effectue à la fin, de la même manière que la fonction construction().
Ainsi, nous obtenons une complexité totale de la fonction en O(n+m).

TasArbre :
SupprMin :
Analyse du pire cas :
→ Nous parcourons en O(logn) pour trouver le dernier nœud ajouté.
→ Puis nous mettons à jour la hauteur et le nombre de nœuds dans les nœuds parents en O(logn).
→ A la fin de la fonction, nous appelons la fonction echangeClef() en O(1) entre le nœud racine et le dernier récupéré, et la fonction echangeRacine() pour positionner le nœud racine dans l'arbre en O(logn).
Nous obtenons une complexité totale bornée par O(logn).

Ajout:
Analyse du pire cas :
→ Nous parcourons en O(logn) pour trouver le dernier nœud ajouté.
→ Puis nous mettons à jour la hauteur et le nombre de nœuds dans les nœuds parents en O(logn).
→ Lors de l'ajout, nous effectuons un rééquilibrage sur tous les nœuds parents en O(logn).
Nous obtenons à la fin une complexité totale en O(logn).

AjoutsItératifs:
Analyse du pire cas :
→ Nous appelons la fonction Initialisation() qui est en O(1).
→ Puis nous effectuons n (nombre de clefs à ajouter) ajouts successifs en O(logn).
Nous obtenons une complexité totale en O(nlogn).

Construction:
Analyse du pire cas :
→ Nous effectuons un parcours en largeur pour trouver l'emplacement du nouveau nœud à ajouté, ce qui nous donne une complexité en O(n) avec n, le nombre de clefs à ajouter car l'ajout se fait en O(1).
→ A la fin des ajouts, nous effectuons un rééquilibrage en O(logn) sur les nœuds parents seulement, donc n/2 nœuds du tas.
Nous obtenons ainsi une complexité de: O(n) + n/2O(logn) avec n/2 négligeable, ce qui nous donne une complexité totale bornée par O(n).

Union:
Le pire cas est représenté par la fusion de deux tas de tailles différentes.
Analyse du pire cas :

→ Nous effectuons un parcours en largeur sur les deux tas en entrée. Au fur et à mesure des parcours, nous ajoutons les nœuds successivement dans un nouveau tas en O(1). Ainsi, nous obtenons un nouveau tas en O(n+m) avec n, la taille du premier tas et m, la taille du deuxième tas en entrée.
→ Après la création du nouveau tas, nous effectuons un rééquilibrage similaire à la fonction construction(), sur les nœuds parents seulement.
Nous obtenons une complexité totale en O(n+m).

Question 2.8
Pour les tests, on utilisera les ensembles de données disponibles sur Moodle. Chaque quintuplet de données pour chaque taille sera testé pour obtenir une moyenne des temps d'exécution des opérations suivantes : ajouts itératifs, construction, suppression du minimum et union.
Méthode pour mesurer le temps d'exécution :
Au début de l'exécution de la fonction dont le temps doit être mesuré, le nombre de ticks est enregistré dans une variable *deb*. Une fois l'opération terminée, une autre variable, *fin*, est définie pour enregistrer le nombre de ticks après l'exécution de la fonction. Le temps d'exécution en secondes est alors calculé en utilisant la formule : (double)(fin - deb) / CLOCKS_PER_SEC.

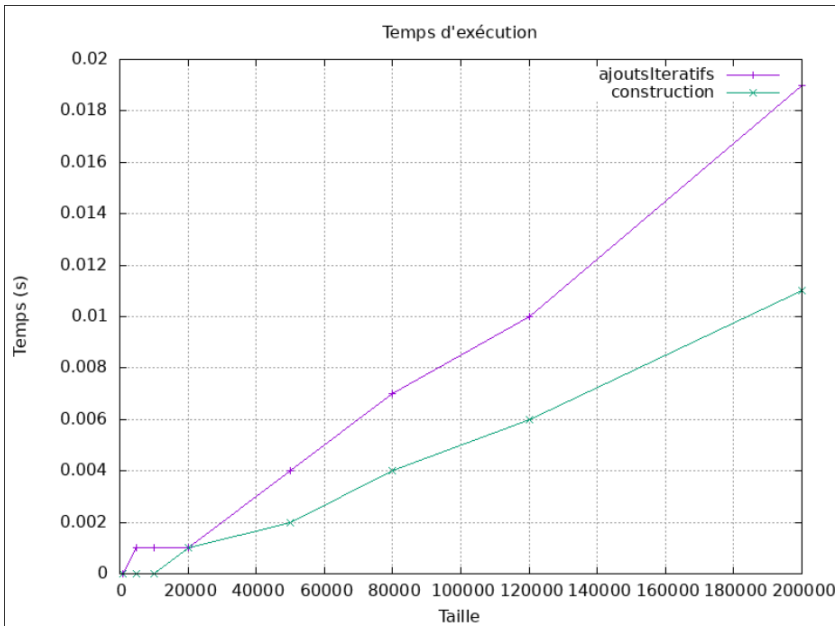

Figure 1 - Comparaison Ajouts Itératifs et Construction de Tas Min Tableau

Dans la figure 1, on observe que la construction est plus efficace que les ajouts itératifs. On explique la différence par la complexité donnée précédemment. Les ajouts itératifs se font en O(nlogn) tandis que la construction se fait en O(n).

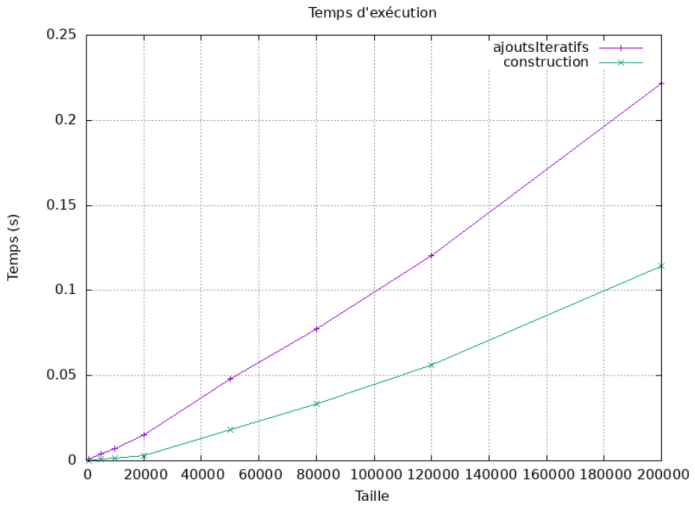

Figure 2 - Comparaison Ajouts Itératifs et Construction de Tas Min Arbre

Dans la figure 2, nous remarquons que les courbes générées montrent que la fonction AjoutItératifs() prend plus de temps à construire un tasArbre que la fonction Construction(). De plus, nous savons que la fonction AjoutsItératifs() est en O(nlogn) et Construction() en O(n). Cela peut expliquer l'écart de temps entre les 2 courbes car la croissance est plus lente pour les fonctions nlogn.

Enfin, nous constatons que les performances des fonctions AjoutsItératifs() et Construction() du tasTableau est largement plus rapide que celles de tasArbre (facteur 10).
Selon nous, cet écart est sûrement dû à la différence de parcours et d'accès au nœud. Dans tasTableau, les accès direct en O(1) tandis que pour tasArbre, nous devons parcourir les nœuds en logn.

Question 2.9

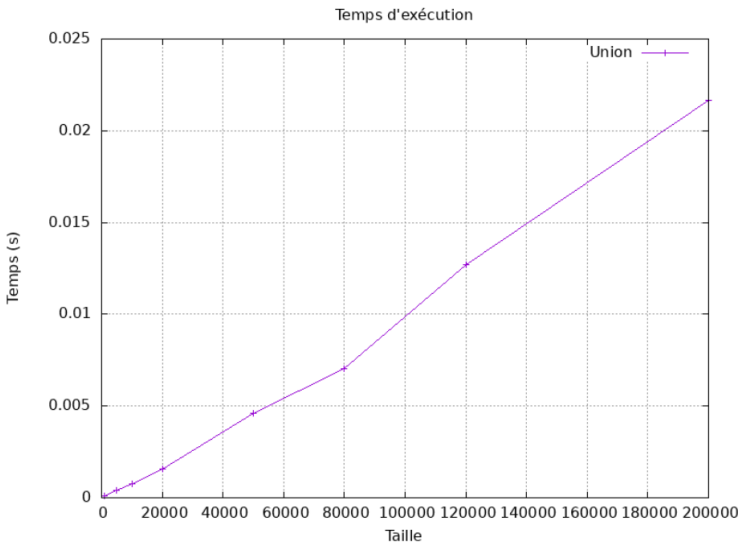

Figure 3 - Union de Tas Min Tableau de même taille

Dans la figure 3, l'union du Tas Min Tableau se fait bien en temps linéaire cohérent avec la complexité théorique O(n+m).
On a testé pour la fusion des tas de même taille, si elles sont de taille différente l'allure de la courbe ne changerait pas mais le temps d'exécution pourrait être légèrement augmenté.

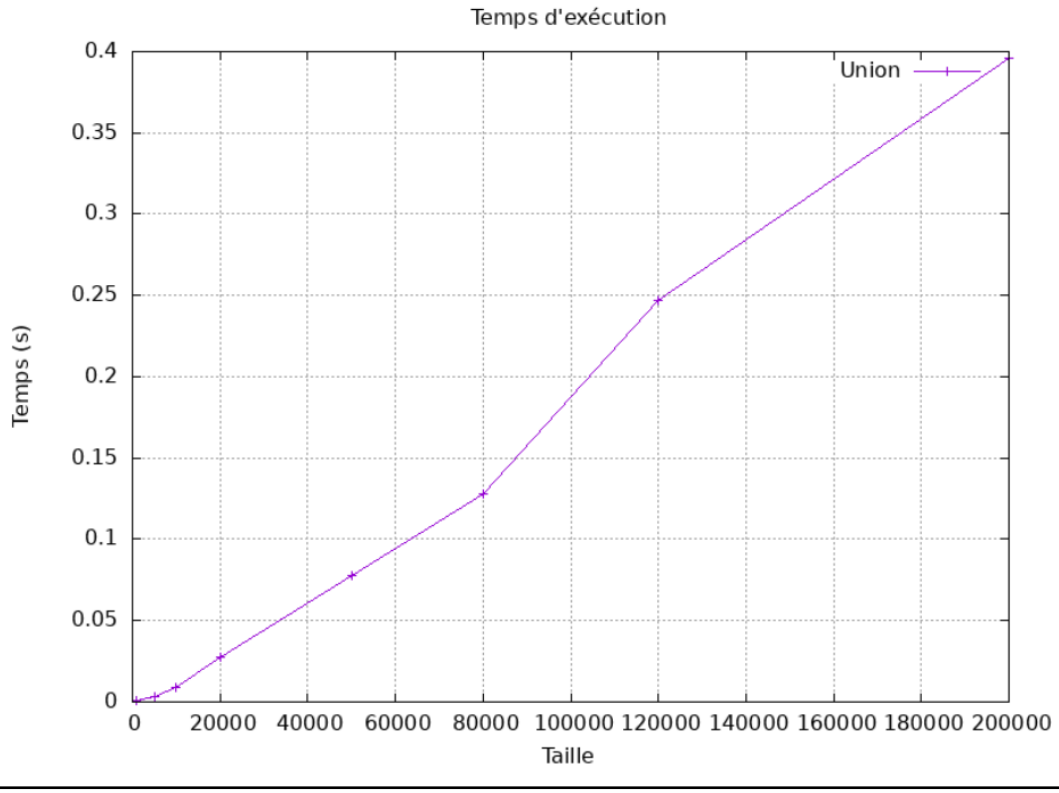

Figure 4 - Union de Tas Min Arbre de même taille

Dans la figure 4, nous pouvons observer que la courbe générée par la fonction Union() sur des jeux de données de même taille est linéaire. Ce qui vérifie la complexité de notre fonction en O(n).

De la même manière que les courbes de AjoutsItératifs() et Construction(), nous remarquons une grande différence de temps d'exécution entre les deux structures.

3. Structure 2 : Files binomiales

Question 3.10
On définit les Tournois Binomiaux, qui ont une clef, un degré, une référence vers le premier fils ainsi qu'un frère. Le parcours des fils d'un nœud se fait en accédant au premier fils du nœud, on parcourt ensuite frères jusqu'à un nœud NULL, comme une liste chaînée.

```c
typedef struct tournoiBinomial
{
    Clef128 clef;
    int degre;
    struct tournoiBinomial * fils;
    struct tournoiBinomial * frere; // frere droit
} TournoiBinomial;

typedef struct elementFile
{
    TournoiBinomial * tournoi;
    struct elementFile * suivant;
} ElementFile;

typedef struct fileBinomiale
{
    ElementFile * tete;
    ElementFile * queue;
} FileBinomiale;
```

On a défini toutes les primitives citées dans le cours pour les Tournois Binomiaux ainsi que pour les Files Binomiales.

Question 3.11

- *FileBinomiale * AjoutMin(FileBinomiale * F, TournoiBinomial * T) :*

La fonction ajoute le nouveau tournoi binomial en tête, puis équilibre la file binomiale avec la fonction *Equilibrage,* qui prend la file en argument et renvoie la file qui respecte la propriété de tournoi de degré unique dans la file.

- *FileBinomiale * SupprMin(FileBinomiale * F) :*

La fonction supprime l'élément minimum de la file. Pour cela, on cherche le tournoi ayant la plus petite racine, avec *TournoiMin* qui prend la file en argument et retourne le tournoi avec le minimum. On utilise ensuite la fonction *Decapite,* qui prend le tournoi minimum en argument, et renvoie la file binomiale composé des fils du tournoi minimum. On finit par fusionner la file nouvellement créée avec la file principale avec la fonction *Union.*

- *FileBinomiale * Construction (Clef128 ** clefs, int debut, int fin) :*

Cette fonction consiste simplement à faire des ajouts successifs de tournois binomiaux, en utilisant la fonction *AjoutMin* précédemment créée. Cette fonction construction est différente des fonctions construction de TasTableau et TasArbre où l'on rééquilibre la structure après les insertions successives des clefs sans se soucier de la propriété de la structure. Ici, construction correspond aux ajoutsIteratifs des structures précédentes, où le rééquilibrage s'effectue à chaque ajout.

- *FileBinomiale * Union(FileBinomiale * F1, FileBinomiale * F2) :*

La fonction appelle *UFret,* qui prend les deux files à fusionner, ainsi qu'un tournoi binomial, qui correspond aux tournois fusionnés que l'on doit ajouter dans la file union.
Notre code pour *Union* est tiré du pseudo code du cours.

Question 3.12

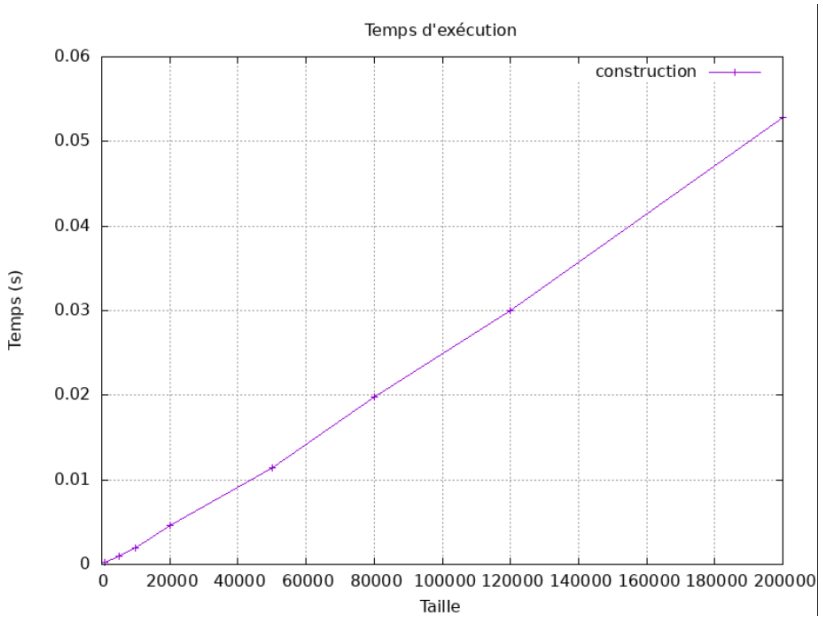

Figure 5 - Temps d'exécution de la fonction Construction pour la File Binomiale

On constate que la complexité théorique O(n) est vérifiée par ce graphe, la courbe est linéaire.

Question 3.13

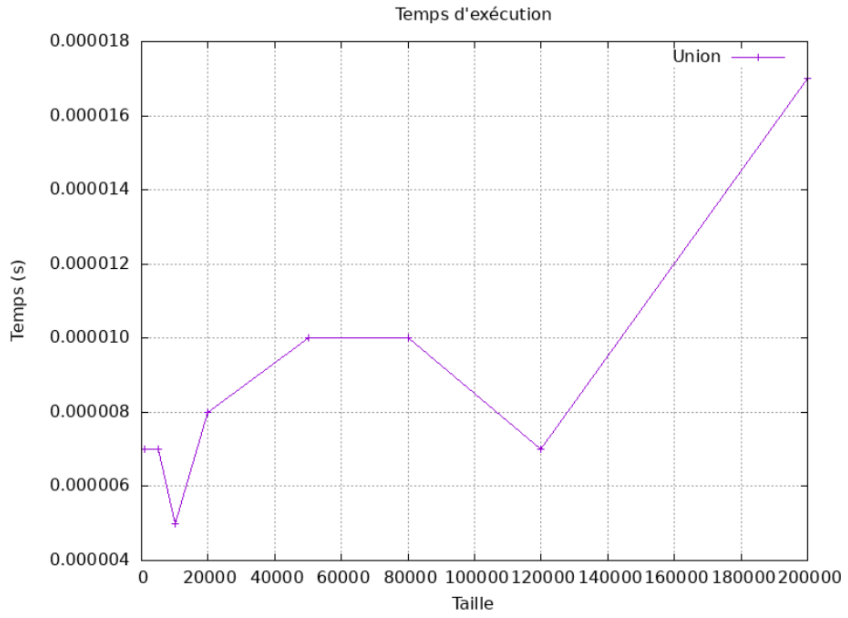

Figure 6 - Temps d'exécution de la fonction Union pour la File Binomiale

Dans la figure 5, la courbe ne semble pas être de la forme O(log(n+m)). Cependant le temps d'exécution d'Union est très court de l'ordre de 4 à 17 microsecondes (10^-6) pour l'union d'une paire de file binomiale de taille 1000 à 200 000.
A chaque exécution de la fonction Union avec les mêmes paramètres, l'allure de la courbe varie. Ce comportement peut être causé par notre méthode de mesure du temps. On définit le nombre de ticks du début et de la fin de l'exécution d'Union. Il se peut que le comportement de la courbe soit dû à l'ordonnancement des processus, auquel cas ce n'est pas le réel temps exécuté par l'opération.

4. Fonction de hachage

La difficulté principale rencontrée lors de l'implémentation de la fonction md5() a été le padding de la chaîne de caractère. Nous avons commencé à travailler sur des bits pour remplir le message de 0, puis en octet dans la suite du code.

```
//Preparation du message(padding) :
int lenInit = strlen(chaine);
int len = lenInit*8; // taille du mot en bits
int nbZero = (448-(len+1))%512;
int offset = (nbZero > 0) ? nbZero : 512-nbZero; //nombre bit à 0 à ajouter
int lenFinal_bit = len+1+offset+64; //taille du message après preparation en bits
int bloc = lenFinal_bit/512; // nombre de bloc de 512 bits = 64 octets
int lenFinal_octet = 64*bloc;
unsigned char* msg = (unsigned char*)malloc(sizeof(unsigned char)*lenFinal_octet);
memset(msg, 0, lenFinal_octet);
memcpy(msg, chaine, lenInit);
msg[lenInit] = (unsigned char)0x80;
int indice = lenFinal_octet-8;
memcpy(msg+indice, &len, 4);
```

5. Arbre de Recherche

Nous avons implémenté un arbre de Recherche équilibré pour respecter la recherche en temps logarithmique et éviter d'avoir des arbres peignes, qui augmenteraient le temps d'exécution.

```
struct noeud {
    Clef128 * clef;
    struct noeud * gauche;
    struct noeud * droit;
    int hauteur;
}Noeud;
```

- La fonction d'insertion va parcourir l'arbre en O(logn) pour connaître l'emplacement du nouveau nœud puis équilibrer l'arbre en vérifiant la hauteur de chaque sous arbre pour ensuite faire des rotations en O(logn).

A chaque ajout d'un nœud, l'arbre de recherche sera équilibré car nous faisons un rééquilibrage a chaque ajout et non à la fin de la construction de l'arbre. Nous n'obtiendrons donc jamais un peigne, la complexité du rééquilibrage est bien en O(logn).

- La fonction d'ajoutsIteratifs appelle la fonction d'insertion n fois. Nous avons donc une complexité totale en O(nlogn).
- La fonction de recherche évalue le nœud racine puis appel récursivement sur le nœud gauche si la clef est plus petite sinon sur le nœud droit. Nous obtenons une complexité en O(logn).

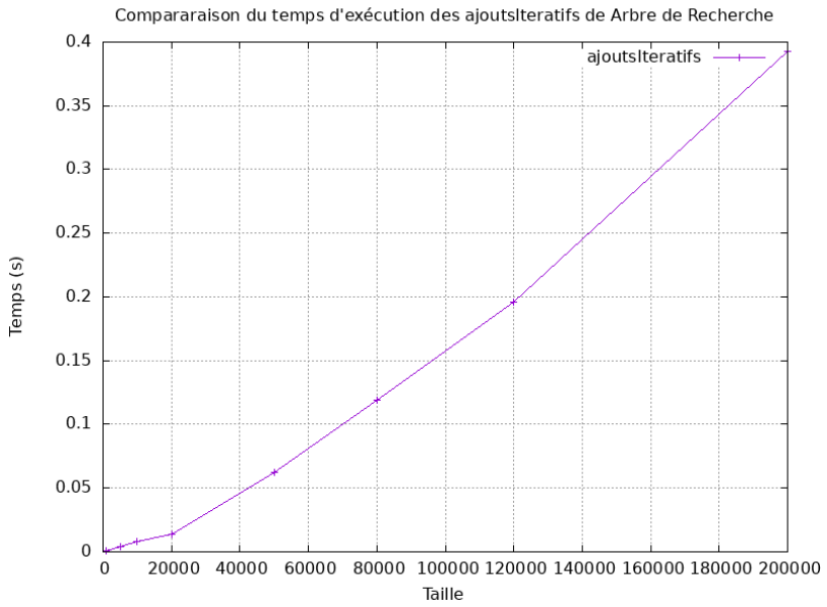

Figure 7 - Temps d'exécution de la fonction AjoutsItératifs de l'arbre de recherche

Dans la figure 7, on observe que l'allure de la courbe est logarithmique car on remarque qu'elle remonte vers la fin. Si l'on teste sur un plus grand nombre de données, on est censé pouvoir mieux observer une complexité en O(nlogn).

6. Etude expérimentale

Question 6.14

La question a été traité dans le fichier ../tests/shakespeare.c
Nous avons construit une liste chaînée pour récupérer tous les mots de l'archive Shakespeare. Chaque mot est haché avec la fonction md5() puis ajouté à un arbre de recherche équilibré, ce qui nous permettra par la suite de faire des recherches en temps logarithmique.
Nous trouvons 2086 mots uniques dans les jeux de données de Shakespeare.

Question 6.15

La question a été traité dans le fichier ../tests/shakespeare.c

Pour trouver des collisions dans les mots de Shakespeare, nous avons regardé pour toutes les clefs si elle se trouvait dans l'arbre de recherche équilibré.
Nous trouvons 0 collisions au sein des clefs. On sait que le hachage md5 produit une clef représentée sur 128 bits, donc assez grand pour qu'il y ait peu voire pas de collisions dans nos 2086 clefs.

Question 6.16

Nous avons créé un script genere_graph_6_16.sh qui permet de générer les graphes de cette question.

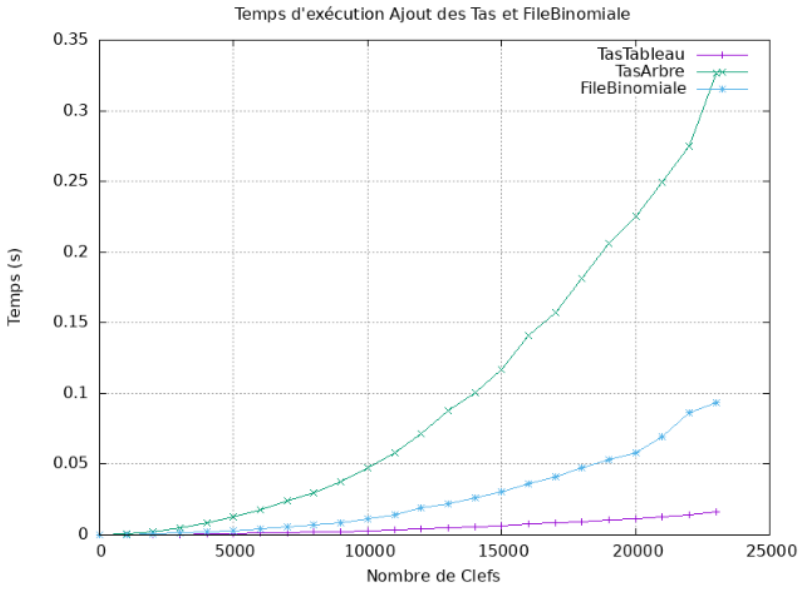

Figure 8 - Temps d'exécution de la fonction Ajout pour les structures de tas et file binomiale

Dans la figure 8, nous observons que le temps d'exécution de la fonction Ajout de la structure tableau du tas est plus rapide que la structure de l'arbre et de la file binomiale.
Les courbes ont une allure de O(nlogn). Car on fait n ajout en O(logn).
On peut expliquer l'écart entre tasTableau et tasArbre par la recherche de l'emplacement pour insérer le dernier nœud. Dans tasTableau, l'insertion se fait en queue du tableau donc en O(1) car nous avons stocké le nombre de clef dans l'attribut "taille" de la structure. Dans tasArbre, nous devons parcourir en logn pour pouvoir insérer le nouveau nœud. Ce qui rallonge le temps d'exécution de la fonction Ajout de tasArbre.
Pour file binomiale, l'insertion se fait en tête de liste donc en O(1) et le rééquilibrage en O(logn) en parcourant la file binomiale. Puis on fusionne les tournois de même degré en O(1). Nous obtenons également la même complexité pour tasTableau. Ainsi, nous pouvons en déduire que le facteur dans la file binomiale est supérieur à tasTableau.
La différence entre tasArbre et file binomiale s'explique de la même manière que la différence entre tasArbre et tasTableau. La structure de la file binomiale ajoute un nouveau nœud en O(1), tandis que celle de tasArbre, parcours en logn avant d'ajouter.

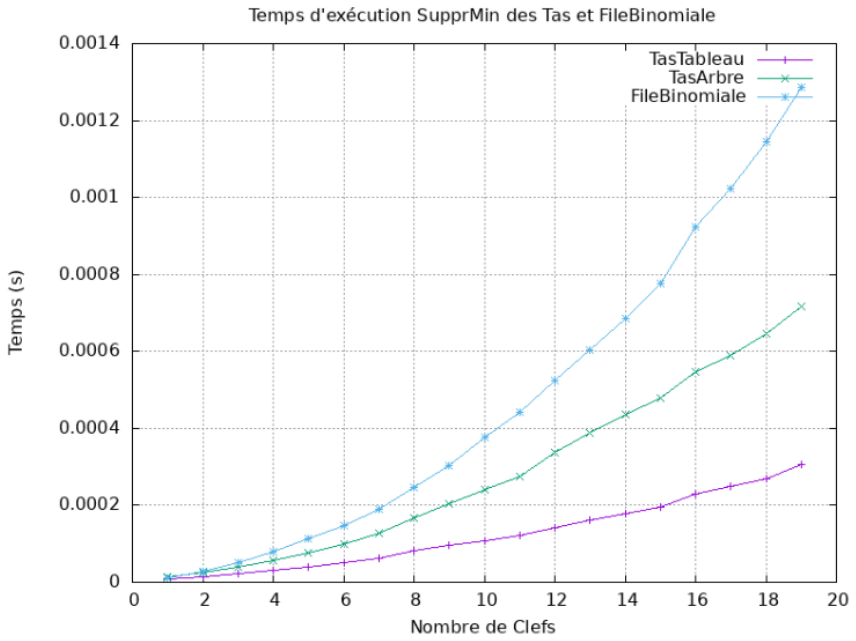

Figure 9 - Temps d'exécution de la fonction SupprMin pour les structures de tas et file binomiale

Dans la figure 9, nous observons que la fonction de SupprMin de tasTableau est le plus rapide, suivi de tasArbre puis de la file binomiale.
Les courbes ont une allure de O(nlogn). Car on fait n supprMin en O(logn).
Pour tasTableau, la suppression se fait en O(1) et le rééquilibrage en O(logn).
Pour tasArbre, la recherche du dernier nœud se fait O(logn), l'échange en O(1) et le rééquilibrage en O(logn).
Pour la file binomiale, il faut parcourir la file en O(logn) à la recherche du tournoi contenant l'élément minimum (se trouvant à la racine). On fusionne la file obtenue après décapitation du tournoi minimum, avec la file principale, sachant que Union est en O(log(n+m)), on obtient une complexité logarithmique qui correspond bien à la complexité théorique.
Nos trois courbes sont de mêmes complexités mais un facteur correspondant aux différentes opérations de chaque structure augmente le temps d'exécution d'une structure à une autre (comme les parcours de structure en temps logarithmique).

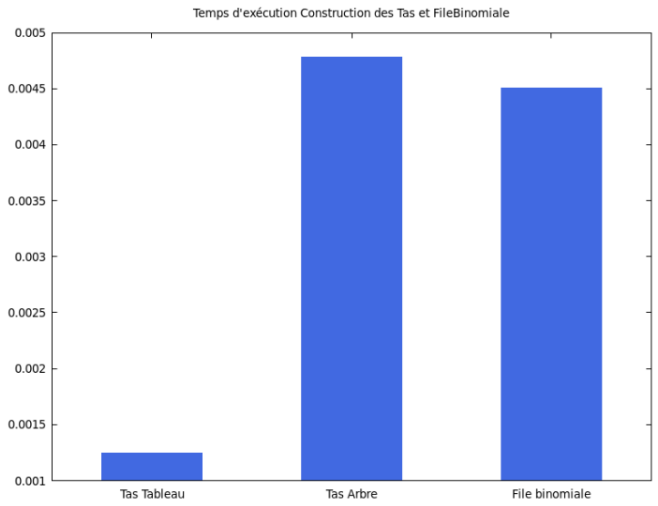

Figure 10 - Temps d'exécution de la fonction Construction pour les structures de tas et file binomiale

Dans la figure 10, nous constatons une nouvelle fois, que l'exécution de la structure tasTableau est beaucoup plus rapide que la structure tasArbre (~15fois plus lent) et la file binomiale (~10fois plus lent).
On sait que nos fonctions construction() sont toutes en O(n) avec n, le nombre de clefs. La différence se trouve donc dans l'implémentation. On explique cette différence du fait que tasTableau ne nécessite pas un parcours de la structure, lors de l'insertion, contrairement aux 2 autres structures, qui pour TasArbre doit trouver l'emplacement où insérer. Pour la File Binomiale, la fonction construction effectue le rééquilibrage à chaque ajout, tandis que TasTableau et TasArbre font le rééquilibrage une seule fois.

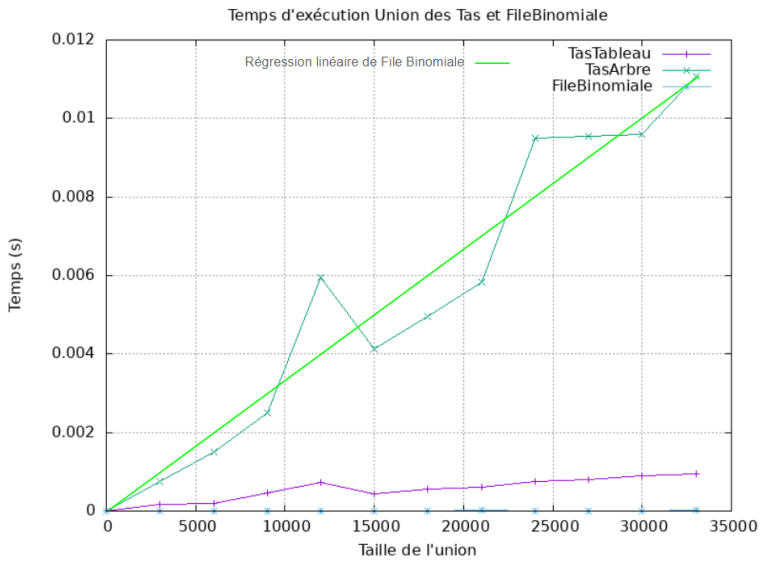

Figure 11 - Temps d'exécution de la fonction Union pour les structures de tas et file binomiale

Pour l'algorithme Union, nous avons pris des données de tailles différentes afin de représenter le pire cas. Taille totale = Taille1 + Taille2
Dans la figure 11, nous remarquons que le temps d'exécution de la fonction Union de la file binomiale est négligeable, par rapport aux deux autres structures.
TasTableau a une allure logarithmique, cependant si l'on augmente la taille des données, la courbe aura une allure linéaire comme dans la figure 3.
Comme vu précédemment entre la figure 3 et 4, il y a un facteur 10 entre le temps d'exécution de l'Union de tasTableau et TasArbre, le graphe est cohérent avec les résultats précédents.
L'écart entre TasTableau, TasArbre et File Binomiale se creusera d'autant plus si l'on augmente la taille de fusion, on l'explique car Union de FileBinomiale est en temps logarithmique tandis que les TasArbre et TasTableau sont en temps linéaire.

7. Conclusion

En conclusion, d'après nos expérimentations les fonctions d'ajout, de suppression du minimum, de construction et d'union pour chacune des structures correspondent bien aux complexités théoriques donnés en cours. La structure arbre du tas min est pour toutes les opérations moins performantes que la structure tableau du tas min. Il se peut pour l'union du tas arbre que notre implémentation ne soit pas optimale, et donc que le temps d'exécution ne soit pas représentatif. Il se pourrait que pour une meilleure implémentation du tas arbre il y ait des opérations plus performantes que celles du tas tableau.