# Projet ALGAV - MU4IN500

Floria LIM 28706087
Myriam MABROUKI 28710344

Automne 2023

# Table des matières

# 1 Introduction

Pour ce projet, nous avons choisi d'utiliser le langage de programmation C++. En effet, celui-ci est orienté objet. Ainsi, nous avons la possibilité de représenter chaque structure de données comme un objet avec des méthodes leur étant propres.
Afin de représenter des clés de 128 bits, nous avons choisi de créer une classe dédiée `Key`. De ce fait, nous avons :

- un attribut de type `array<unsigned long, 4>` (un tableau de taille fixée à 4, où chaque élément `unsigned long` représente un entier non signé de 32 bits)
- des méthodes `inf` et `eg` (prédicats d'infériorité et d'égalité qui ne s'appliquent que sur des objets de type `Key`)

# 2 Tas priorité minimale

**Définition 1** *Un tas minimal est un arbre binaire étiqueté de façon croissante, dont toutes les feuilles sont situées au plus sur deux niveaux, les feuilles du niveau le plus bas étant positionnées le plus à gauche possible.*

Pour ce projet, nous devions représenter un tas minimal avec deux structures disctintes : via un arbre binaire et via un tableau. Dès lors, nous allons vous détailler notre implémentation de ces deux structures.

Pour les réaliser, nous avons choisi de créer deux classes distinctes : `Heap_array` pour la version avec un tableau et `Heap_tree` pour la version avec un arbre binaire. Voyons ensemble quels sont les attributs que nous avons choisis pour représenter ces deux classes.

- via un tableau (`Heap_array`) :
  - un `vector` de `Key` qui possède l'ensemble des clés du tas
  - un entier `size` qui représente le nombre de clés dans le tas
- via un arbre binaire (`Heap_tree`) :
  - un pointeur `value` sur une `Key` qui représente la valeur du noeud
  - deux pointeurs de type `Heap_tree` qui représentent le fils gauche `left` et le fils droit `right`
  - un entier `size` qui représente la taille de l'arbre

**Question 2.5**
La fonction `Construction` permet de construire un tas de priorité minimum de façon plus efficace que $n$ appels à `Ajout` comme dans `AjoutIteratifs`, $n$ étant le nombre d'éléments à insérer.
Détaillons le principe de cet algorithme dans les deux versions implémentées (via un tableau et via un arbre binaire).

Avec un tableau
La `construction` se fait d'abord en initialisant un tableau auquel nous ajoutons nos $n$ clés.
La suite de l'algorithme consiste en la descente des noeuds, excepté ceux du dernier niveau, jusqu'à atteindre la bonne position (ils doivent être plus grands que leur père et plus petits que leurs éventuels fils).
Les tas sont des arbres binaires presque complets, c'est-à-dire que tous les niveaux sont remplis sauf éventuellement le dernier. De plus, nous savons qu'à chaque niveau le nombre de noeuds double car il s'agit

d'un arbre binaire. De ce fait, nous parcourons uniquement la première moitié du tableau puisqu'elle correspond à l'ensemble des noeuds privé des feuilles. Lors de ce parcours, nous appelons une fonction auxiliaire, `Construction_aux`, qui permet d'échanger la position entre un noeud et ses fils. Cette fonction s'appelle récursivement tant qu'il reste des échanges à faire.

Voici le pseudo-code de notre algorithme `Construction` avec une structure sous forme de tableau.

**Algorithme 1 :** Construction

**Entrées :** $L$ une liste de $n$ clés toutes distinctes

**Sorties :** $T$ le tas sous forme de tableau créé à partir de $L$

$T \leftarrow$ tableau de taille $n$ initialisée à 0

**pour** *$i$ allant de 1 à $n$* **faire**
  $T[i] \leftarrow L[i]$

**fin**

**pour** *$i$ allant de n/2 à 1* **faire**
  $Construction\_aux(T, i)$

**fin**

**retourner** $T$

**Algorithme 2 :** Construction_aux

**Entrées :** $T$ un tas sous forme de tableau de taille $n$, $i$ un entier

$left \leftarrow 2 \times i$

$right \leftarrow 2 \times i + 1$

$max \leftarrow i$

**si** $left \leq n$ *et* $T[left] < T[max]$ **alors**
  $max \leftarrow left$

**si** $right \leq n$ *et* $T[right] < T[max]$ **alors**
  $max \leftarrow right$

**si** $i \neq max$ **alors**
  $echanger(T[i], T[max])$
  $Construction\_aux(T, max)$

Avec une structure arborescente

La `construction` se fait d'abord en insérant les clés dans une structure arborescente qui a la bonne forme. Pour cela, nous initialisons un tableau $T$ de taille $n$ ayant comme premier élément la racine de l'arbre auquel on y a assigné la première clé de la liste. L'algorithme parcourt le reste des clés de la liste et pour chaque clé, un nœud parent est extrait de $T$, et ses nœuds enfants sont créés avec les clés suivantes. Ces nœuds enfants sont ensuite ajoutés à $T$ pour être utilisés comme parents potentiels ultérieurement. A la fin du parcours de la liste de clés, $T$ correspond à un parcours en largeur de la structure arborescente créée. Une fois toutes les clés insérées, l'algorithme effectue des remontées dans l'arbre à partir des nœuds du bas vers la racine. Comme expliqué précédemment, nous parcourons uniquement la première moitié de $T$ puisqu'elle correspond à l'ensemble des noeuds privé des feuilles. Lors de ce parcours, nous appelons une fonction auxiliaire, `Construction_aux` qui compare les valeurs des nœuds avec celles de leurs enfants et échange leurs valeurs si nécessaire. Nous en profitons également pour mettre à jour la taille des noeuds lors de ce parcours.

Voici le pseudo-code de notre algorithme `Construction` avec une structure arborescente.

**Algorithme 3 :** Construction

**Entrées :** $L$ une liste de $n$ clés toutes distinctes

**Sorties :** $H$ le tas sous forme arborescente créé à partir de $L$

$T \leftarrow$ tableau de taille $n$ initialisée à 0

$T[1] \leftarrow (TasArbre(L[1], \epsilon, \epsilon))$

$indice \leftarrow 1$

$Tsize \leftarrow 1$

**pour** *i allant de 2 à n* **faire**

- $current \leftarrow T[indice]$
- $indice \leftarrow indice + 1$
- $current.left \leftarrow TasArbre(L[i], \epsilon, \epsilon)$
- $Tsize \leftarrow Tsize + 1$
- $T[Tsize] \leftarrow current.left$
- $i \leftarrow i + 1$
- **si** $i < n$ **alors**
  - $current.right \leftarrow TasArbre(L[i], \epsilon, \epsilon)$
  - $Tsize \leftarrow Tsize + 1$
  - $T[Tsize] \leftarrow current.right$

**fin**

**pour** *i allant de n/2 à 1* **faire**

- $Construction\_aux(T[i])$
- $T[i].size = T[i].left.size + T[i].right.size$

**fin**

$H \leftarrow T[1]$

**retourner** $H$

**Algorithme 4 :** Construction_aux

**Entrées :** $H$ un tas sous forme arborescente

**si** $H.right \neq \epsilon$ *et* $H.left.value < H.right.value$ **alors**

- **si** $H.value < H.left.value$ **alors**
  - $echanger(H.value,\ H.value.left)$
  - $Construction\_aux(H.left)$

**sinon si** $H.right \neq \epsilon$ *et* $H.value < H.right.value$ **alors**

- $echanger(H.value,\ H.value.right)$
- $Construction\_aux(H.right)$

**Question 2.6**

La fonction `Union` prend en arguments deux tas ne partageant aucune clé commune, et construit un tas qui contient l'ensemble de toutes les clés.

Détaillons le principe de cet algorithme dans les deux versions implémentées (via un tableau et via un arbre binaire).

Avec un tableau

Nous commençons tout d'abord par initialiser un tableau de taille n1 + n2 ainsi qu'un indice à 1. Nous itérons sur les deux tas tant que les deux ne sont pas vides. À chaque itération, l'algorithme compare les clés minimales des deux tas et extrait la plus petite des clés pour l'ajouter au tableau. Nous obtenons ainsi l'union des deux tas lorsque toutes les clés ont été extraites.
Voici le pseudo-code de notre algorithme `Union` avec une structure sous forme de tableau.

**Algorithme 5 :** Union

**Entrées :** $T1$ et $T2$ deux tas sous forme de tableau de taille $n1$ et $n2$ ne partageant aucune clé commune
**Sorties :** $T$ le tas qui contient l'ensemble de toutes les clés
$T \leftarrow$ tableau de taille $n1 + n2$ initialisée à 0
$indice \leftarrow 1$
**tant que** $Non(isVide(T1))$ *ou* $Non(isVide(T2))$ **faire**
  **si** $isVide(T2)$ *ou* $(Non(isVide(T1))$ *et* $T1[1] < T2[1])$ **alors**
    $T[indice] \leftarrow SupprMin(T1)$
    $indice \leftarrow indice + 1$
  **sinon**
    $T[indice] \leftarrow SupprMin(T2)$
    $indice \leftarrow indice + 1$
**fin**
**retourner** $T$

Avec une structure arborescente

L'algorithme d'`Union` sous une structure arborescente reprend la même procédure que celle avec un tableau. Nous extrayons à chaque itération la plus petite clé des deux tas puis l'insérons dans une structure arborescente avec un parcours en largeur. Ainsi, comme pour l'algorithme de construction, un tableau de noeud `Heap_tree` est initialisé pour pouvoir y stocker les noeuds du parcours. Aussi, il nous a été nécessaire de mettre à jour la taille `size` de chaque noeud parent une fois toutes les insertions terminées.
Voici le pseudo-code de notre algorithme `Union` avec une structure arborescente.

**Algorithme 6 :** Union

**Entrées :** $H1$ et $H2$ deux tas sous forme arborescente de taille $n1$ et $n2$ ne partageant aucune clé commune

**Sorties :** $H$ le tas qui contient l'ensemble de toutes les clés

$T \leftarrow$ tableau de taille $n1 + n2$ initialisée à 0

$indice \leftarrow 1$

$Tsize \leftarrow 1$

**tant que** $Non(isVide(H1))$ *ou* $Non(isVide(H2))$ **faire**

  $current \leftarrow T[indice]$

  $indice \leftarrow indice + 1$

  **si** $isVide(H2)$ *ou* $(Non(isVide(H1))$ *et* $H1.value < H2.value)$ **alors**

    $current.left \leftarrow SupprMin(H1)$

  **sinon**

    $current.left \leftarrow SupprMin(H2)$

  $Tsize \leftarrow Tsize + 1$

  $T[Tsize] \leftarrow current.left$

  **si** $Non(isVide(H1))$ *ou* $Non(isVide(H2))$ **alors**

    **si** $isVide(H2)$ *ou* $(Non(isVide(H1))$ *et* $H1.value < H2.value)$ **alors**

      $current.right \leftarrow SupprMin(H1)$

    **sinon**

      $current.right \leftarrow SupprMin(H2)$

    $Tsize \leftarrow Tsize + 1$

    $T[Tsize] \leftarrow current.right$

**fin**

**pour** *i allant de n/2 à 1* **faire**

  $T[i].size = T[i].left.size + T[i].right.size$

**fin**

$H \leftarrow T[1]$

**retourner** $H$

**Question 2.7**

Nous avons implémenter cinq fonctions, `Ajout, AjoutIteratifs, SupprMin, Construction` et `Union`, pour un tas de priorité minimale.

Prouvons les complexités de chacune d'entre elles pour la version avec un arbre binaire et celle avec un tableau.

Notons $n$ le nombre d'éléments dans notre tas.

1. via un tableau

   (a) `Ajout`

   Cet algorithme consiste d'abord à ajouter une case à la fin de notre tableau représentant notre tas. Cet ajout a une complexité amortie en $O(1)$ avec les primitives existantes sur la structure `vector` en C++ d'après [1]. En effet, on alloue une taille de 1 au départ et à chaque fois que l'on doit dépasser cette taille, on alloue le double de la taille du vecteur. Le reste du temps, l'ajout se fait en $O(1)$, d'où cette complexité amortie obtenue.

   Afin de conserver la propriété de tas, nous devons bien positionner l'élément ajouté (ses

ascendants doivent être inférieurs à lui tandis que ses descendants doivent être supérieurs).

Nous pouvons connaître la positions du père $pos\_pere$ d'un élément $i$ grâce à la formule suivante : $pos\_pere = \lfloor \frac{i-1}{2} \rfloor$

Ainsi, nous partons de l'élément ajouté et nous faisons des échanges en remontant, pour respecter la priorité du minimum, tant que cela est nécessaire. Au pire cas, nous devrons parcourir toute une branche, soit $log(n)$ comparaisons.

Par conséquent, nous avons une complexité pire cas pour `Ajout` en $O(log(n))$.

(b) `AjoutIteratifs`

Le principe d'`AjoutIteratifs` consiste en autant d'appels à `Ajout` que d'éléments à ajouter. Ainsi, nous obtenons une complexité pire cas pour `AjoutIteratifs` en $O(n \times log(n))$.

(c) `SupprMin`

L'idée derrière `SupprMin` est de, dans un premier temps, supprimer la racine, c'est-à-dire le premier élément du tableau, et de la remplacer par le dernier enfant ajouté, c'est-à-dire le dernier élément du tableau. Cette suppression se fait en $O(1)$ avec les primitives existantes sur la structure `vector` en C++.

De même que pour `Ajout`, nous devons conserver la propriété de tas concernant la position des éléments. De ce fait, nous devrons descendre l'élément ayant pris la place de racine jusqu'à que celui-ci se trouve à la bonne position.

Nous pouvons connaître les positions des fils $pos\_fils1$ et $pos\_fils2$ d'un élément $i$ grâce aux formules suivantes :

i. $pos\_fils1 = 2 \times i + 1$

ii. $pos\_fils2 = 2 \times i + 2$

Ainsi, nous devons, au plus, parcourir toute une branche et faire $log(n)$ comparaisons. Nous obtenons donc une complexité pire cas en $O(log(n))$ pour `SupprMin`.

(d) `Construction`

Concernant l'algorithme de `Construction`, nous initialisons d'abord un tableau auquel nous ajoutons nos $n$ clés. Tout cela se fait en $O(n)$. En effet, chaque ajout dans notre `vector` a un coût amorti en $O(1)$ comme expliqué pour l'algorithme d'`Ajout`.

La suite de l'algorithme consiste, grâce à la fonction auxiliaire `Construction_aux`, en la descente des noeuds de sorte à ce que la propriété de croissance soit respectée.

Cette fonction auxiliaire réalise un nombre constant de comparaisons, deux, et s'appelle récursivement le long d'une branche. De ce fait, il semblerait que la complexité de `Construction_aux` soit en $O(log(n))$. Comme $\frac{n}{2}$ appels sont faits à cette fonction nous pourrions croire que `Construction` réalise en $O(n \times log(n))$ comparaisons.

Procédons à une analyse plus fine.

Considérons dans notre arbre $h$ le niveau d'un noeud. La racine est de niveau $log(n)$ et les feuilles sont de niveau 0.

Prouvons d'abord par récurrence que :

$$\text{Pour tout tas de taille } n, \text{ il y a } \left\lfloor \frac{n}{2^h} \right\rfloor \text{ de noeuds de niveau } \geq h \tag{1}$$

**Preuve 1** *Cas de base :*

Le tas considéré est réduit au tas à un seul noeud, il n'y a donc qu'un seul niveau.
$\left\lfloor \frac{1}{2^0} \right\rfloor = \left\lfloor \frac{1}{1} \right\rfloor = 1$. Il n'y a bien qu'un seul noeud de niveau supérieur ou égal à 0.

*Cas d'induction :*

Supposons qu'il existe un $n \in N$ telle que la propriété que nous cherchons à démontrer soit vraie.
Montrons qu'elle est vraie au rang $n+1$.

Nous avons un tas de $n$ noeuds pour lequel la propriété est vérifiée
Rajoutons un noeud à notre arbre afin d'obtenir un tas de taille $n+1$. Nous distinguons alors deux cas :

— si $n+1$ n'est pas un multiple de $2^h$ :
  Comme nous récupérons l'entier inférieur, nous avons $\left\lfloor \frac{n}{2^h} \right\rfloor = \left\lfloor \frac{n+1}{2^h} \right\rfloor$
  Comme, par hypothèse de récurrence, nous avons $\left\lfloor \frac{n}{2^h} \right\rfloor \geq h$, nous obtenons bien $\left\lfloor \frac{n+1}{2^h} \right\rfloor \geq h$

— si $n+1$ est un multiple de $2^h$ :
  Nous avons alors $\left\lfloor \frac{n+1}{2^h} \right\rfloor = \left\lfloor \frac{n}{2^h} \right\rfloor + 1$.
  Par hypothèse de récurrence, comme $\left\lfloor \frac{n}{2^h} \right\rfloor \geq h$, nous avons bien $\left\lfloor \frac{n}{2^h} \right\rfloor + 1 \geq h$ et donc $\left\lfloor \frac{n+1}{2^h} \right\rfloor \geq h$

Avec $n$ noeuds dans notre tas, nous avons au total $log(n)$ niveaux.
Ainsi, nous avons
— $\lfloor \frac{n}{2} \rfloor$ noeuds de niveau $\geq 1$
— $\lfloor \frac{n}{4} \rfloor$ noeuds de niveau $\geq 2$
...
— $\left\lfloor \frac{n}{2^{log(n)}} \right\rfloor$ noeuds de niveau $\geq log(n)$

Notons $h$ le niveau d'un noeud. La fonction `Construction_aux` réalise des appels récursifs depuis un certain noeud jusqu'à atteindre des feuilles. Elle effectue donc autant de comparaisons que la différence entre la longueur d'une branche et la profondeur du noeud considéré. Cela correspond exactement au niveau d'un noeud. Ainsi `Construction_aux` réalise $h$ comparaisons par noeuds.

Par conséquent, nous obtenons bien :

$$\sum_{h=0}^{log(n)} \left\lfloor \frac{n}{2^h} \right\rfloor \times h \;=\; n \times \sum_{h=0}^{log(n)} \left\lfloor \frac{1}{2^h} \right\rfloor \times h \;=\; n \times \sum_{h=0}^{log(n)} \left\lfloor \frac{h}{2^h} \right\rfloor \tag{2}$$

Ainsi nous avons $O(n \times \sum_{h=0}^{log(n)} \lfloor \frac{h}{2^h} \rfloor)$ comparaisons pour `Construction_aux`.

Déterminons la limite suivante :

$$\lim_{h \to \infty} \frac{h}{2^h} \tag{3}$$

Il s'agit d'une forme indéterminée du type $\frac{\infty}{\infty}$.
Nous pouvons donc appliquer la règle de l'hôpital.

Notons $f$ et $g$ les fonctions $f : h \longmapsto h$ et $g : h \longmapsto 2^h$.
$f$ et $g$ sont dérivables sur R. Ainsi,

$$\lim_{h \to \infty} \frac{f(h)}{g(h)} = \lim_{h \to \infty} \frac{f'(h)}{g'(h)} = \lim_{h \to \infty} \frac{1}{2^h \times log(2)} \tag{4}$$

Comme nous avons $\lim_{h \to \infty} 1 = 1$ et $\lim_{h \to \infty} 2^h \times log(2) = \infty$ , nous obtenons :

$$\lim_{h \to \infty} \frac{h}{2^h} = 0 \tag{5}$$

Par conséquent, nous pouvons négliger la somme $\sum_{h=0}^{log(n)} \lfloor \frac{h}{2^h} \rfloor$ qui devient constante.

Finalement, nous nous retrouvons bien avec une complexité en $O(n)$.

(e) `Union`
Pour l'algorithme d'`Union`, nous itérons tant que les deux tas ne sont pas vides. À chaque itération, une comparaison ainsi qu'un appel à la fonction `SupprMin` sont effectués sur un élément. Ces deux opérations ayant une complexité en $O(1)$, chaque itération de la boucle se fait en temps constant et traite au plus un élément à la fois. La complexité de l'algorithme est donc linéaire en la somme des tailles des deux tas, soit $O(n1 + n2)$.

2. via un arbre binaire

(a) `Ajout`
Cet algorithme commence tout d'abord par ajouter le nœud à la fin du tas. La position du nœud à ajouter est décidée de manière récursive en fonction de la hauteur actuelle du tas.
À chaque appel récursif, nous pouvons connaître lequel des deux fils il faut parcourir de la manière suivante :

i. Si $n < \frac{2^{log_2(n)+1}}{2} + \frac{2^{log_2(n)+1}}{4}$ : fils gauche
ii. Sinon : fils droit

La recherche de cette position se fait en $log(n)$ car nous parcourons une branche d'un arbre qui est parfaitement équilibré dans le pire des cas. À chaque fin d'appel récursive, l'algorithme échange éventuellement la valeur du nœud avec celle de son fils pour préserver la propriété du tas. Cet échange s'effectuant en temps constant, l'algorithme de `Ajout` a donc une complexité pire cas en $O(log(n))$.

(b) `AjoutIteratifs`

De même que via un tableau, `AjoutIteratifs` réalise autant d'appels à `Ajout` que d'éléments à ajouter. Ainsi, nous obtenons également une complexité pire cas pour `AjoutIteratifs` en $O(n \times log(n))$.

(c) `SupprMin`

Concernant la fonction `SupprMin` via une forme arborescente, nous faisons dans un premier temps un appel à une première fonction auxiliaire `last_element` qui réalise le parcours d'une branche depuis la racine pour trouver le dernier noeud ayant été inséré. Ce parcours réalise $O(log(n))$ comparaisons.

Après avoir réalisé la suppression du noeud désiré, nous parcourons, avec une seconde fonction auxiliaire `SupprMin_aux`, de nouveau notre tas depuis la racine pour réaliser les échanges nécessaires à la propriété de croissance de l'arbre. De même que pour le précédent, ce parcours réalise $O(log(n))$ comparaisons.

Au total, nous réalisons $2 \times log(n)$ comparaisons, ce qui nous donne une complexité en $O(log(n))$.

(d) `Construction`

Pour l'algorithme de `Construction`, l'insertion des clés dans une structure arborescente ayant la bonne forme se fait en $O(n)$ via un tableau représentant un parcours en largeur. Tout comme pour la structure sous forme de tableau, une fonction auxiliaire récursive `Construction_aux` est appelée sur la moitié des noeuds de l'arbre qui représente l'ensemble des noeuds privés des feuilles, et cette dernière est négligeable comme démontré précédemment. L'algorithme de `Construction` s'effectue donc en $O(n)$.

(e) `Union`

Le principe de l'algorithme d'`Union` pour une structure arborescente est la même que celle pour la structure sous forme de tableau. Nous effectuons des opérations sur une clé en temps constant et chaque clé est traitée une seule fois. Cependant, on peut noter qu'un parcours sur la moitié des noeuds de l'arbre est effectuée en addition pour la mise à jour des tailles, nécessitant ainsi un traitement supplémentaire par rapport à la structure sous forme de tableau. Cette dernière s'effectuant en $O(\frac{n1+n2}{2})$, la complexité temporelle de l'algorithme reste en temps linéaire en la somme des tailles des deux tas $O(n1 + n2)$.

**Question 2.8**

Pour la mesure du temps d'exécution des algorithmes, nous avons généré des jeux de données aléatoires en plus allant jusqu'à 1000000 de clés afin d'obtenir une analyse des complexités plus précise. Pour cela, nous avons construit une fonction `generate_jeu` qui génère 5 fichiers .txt *nb_cles* clés sur 128 bits en hexadécimal. Afin de garantir que les clés générées soient toutes distinctes, nous avons fait appel à la structure d'ensemble `Set` fournie en C++.

Pour chaque nombre de clés donné, nous mesurons le temps d'exécution d'un algorithme pour 5 jeux de données différentes et calculons la moyenne des 5.

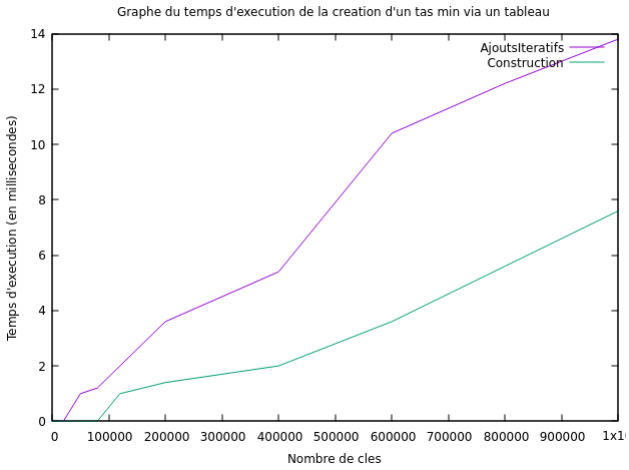


FIGURE 1 – Sous la forme d'un tableau

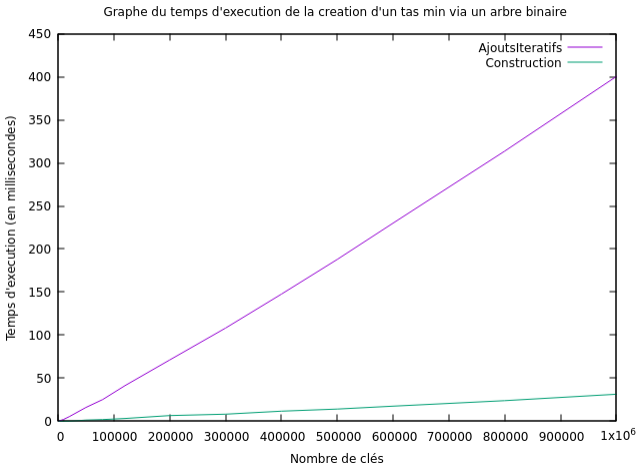


FIGURE 2 – Sous une forme arborescente

Nous pouvons observer une courbe linéaire pour les fonctions `AjoutIteratifs` et `Construction` dans les deux structures. L'algorithme de `Construction` possède un temps d'exécution plus rapide que `AjoutIteratifs` dans nos deux cas de structure, vérifiant ainsi les complexités démontrées précédemment c'est-à-dire $O(n \times log(n))$ pour `AjoutIteratifs` et en $O(n)$ pour `Construction`.
De plus, nous pouvons noter que le temps d'exécution pour la structure arborescente est globalement plus longue que pour la structure sous forme de tableau. Cela peut s'expliquer par le fait que les données contenues dans la structure sous forme de tableau sont contiguës en mémoire. En effet, l'accès d'un élément se fait donc directement contrairement à la forme arborescente où il est nécessaire d'effectuer un parcours. Aussi, une allocation dynamique d'un noeud est nécessaire à chaque ajout d'un élément dans la structure arborescente, ce qui se traduit par un temps d'exécution plus long, alors que nous réservons dès le début la bonne taille en mémoire pour le tableau.

**Question 2.9**

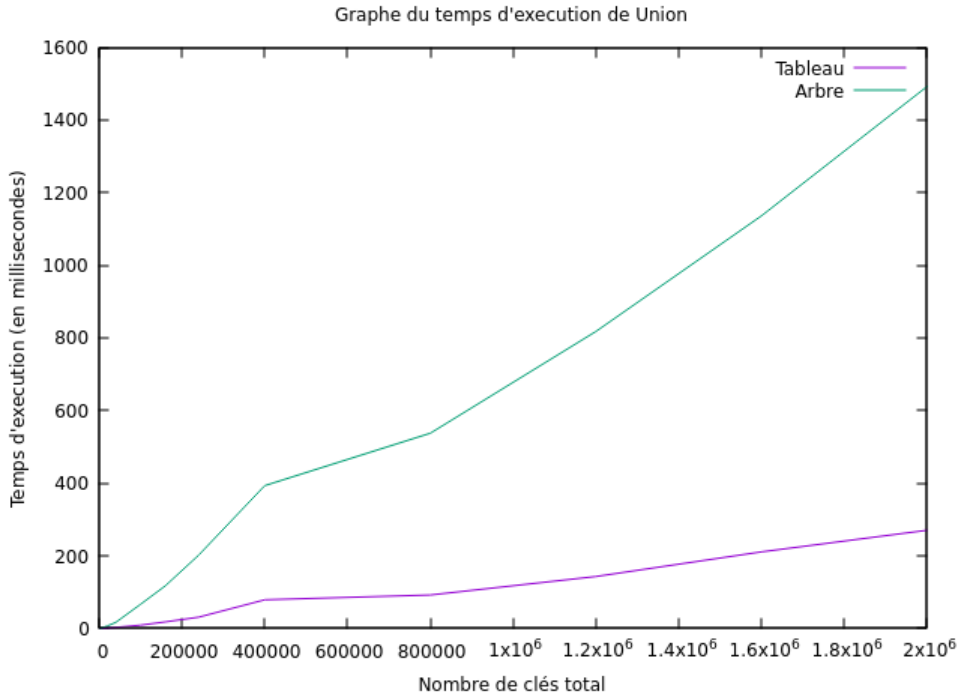


FIGURE 3 – Union

Pour la mesure du temps d'exécution de `Union`, nous avons effectué une union entre deux tas différents de même taille.
Le temps d'exécution de l'algorithme `Union` a bien une forme linéaire, vérifiant ainsi la complexité en $O(n1 + n2)$. Comme pour l'algorithme de `Construction`, la structure sous forme de tableau est plus efficace que celle sous forme arborescente. Cela peut s'expliquer par le parcours de la moitié des noeuds que nous effectuons en plus dans la structure arborescente afin de mettre à jour les tailles des noeuds.

# 3 Files Binomiales

**Définition 2** *Arbre binomial*

- $B_0$ *est l'arbre réduit à un seul nœud*
- *Étant donnés 2 arbres binomiaux* $B_k$ *, on obtient* $B_{k+1}$ *en faisant de l'un des* $B_k$ *le premier fils à la racine de l'autre* $B_k$*.*

**Définition 3** *Un tournoi binomial est un arbre binomial étiqueté croissant (croissance sur tout chemin de la racine aux feuilles)*

**Définition 4** *Une file binomiale est une suite de tournois binomiaux de tailles strictement décroissantes*

Pour ce projet, nous devions également représenter une file binomiale. Pour ce faire, nous avons créer les classes `TournoiBinomial` et `FileBinomiale`. Regardons les attributs que nous avons choisi pour ces deux classes :

- `TournoiBinomiale`
  - un `vector` de `TounoiBinomial` qui représente la forêt liée à la racine du tournois
  - un entier, `size`, qui représente le nombre de noeuds du tournoi
- `FileBinomiale`
  - un `vector` de `TounoiBinomial` qui représente la suite de tournois binomiaux dont est composée la file
  - un entier, `size`, qui représente le nombre de noeuds de la file

**Question 3.12**

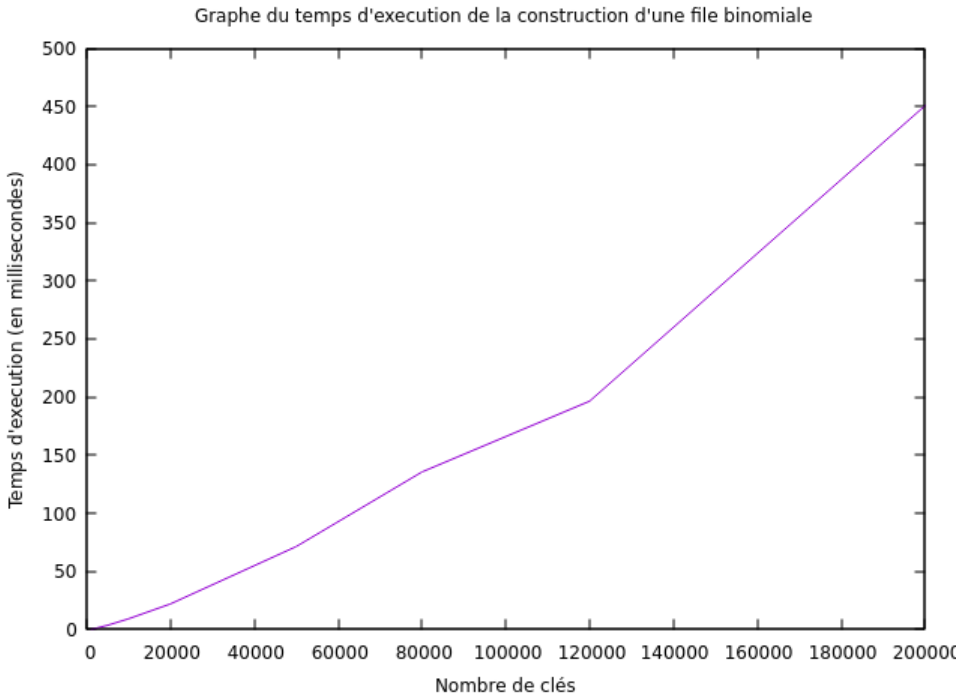


FIGURE 4 – Construction

Pour l'algorithme de `Construction`, nous appelons la fonction `Ajout` $n$ fois. Nous observons que le temps d'exécution est linéaire, ce qui correspond bien à une complexité en $O(n)$.

**Question 3.13**

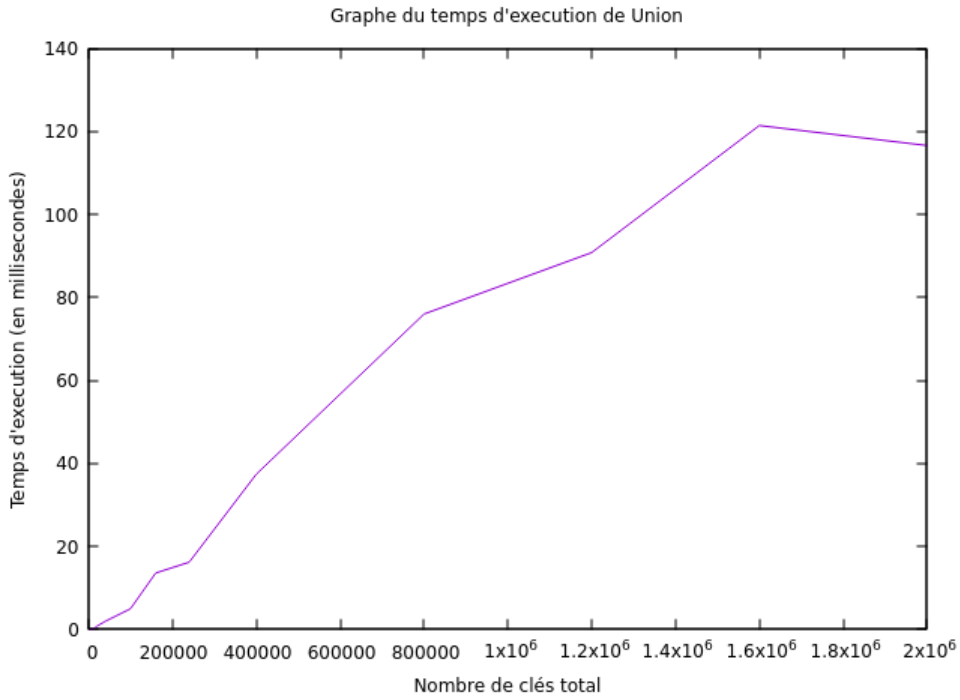


FIGURE 5 – Union

Pour la fonction `Union`, nous avons appliqué à la lettre le pseudo-code donné en cours. Il nous a été nécessaire de tester cet algorithme sur des jeux allant jusqu'à 1000000 de clés pour nous permettre de conclure sur la forme de la courbe. Nous observons donc une allure logarithme qui correspond bien à la complexité en $O(log(n+m))$.

# 4 Arbre de recherche

Pour notre structure arborescente de recherche, nous avons tout d'abord décidé d'implémenter un simple ABR. Notre structure possède :

- un pointeur `key` de type `Key` qui représente le haché MD5 d'un mot
- une chaîne de charactères `value` où on y stocke le mot
- deux pointeurs de type `BinarySearchTree` qui représentent le fils gauche `left` et le fils droit `right`
- un entier, `size`, qui représente la taille de l'arbre

La recherche dans un arbre binaire de recherche possède en moyenne une complexité temporelle en $O(log(n))$, cependant elle peut être au pire des cas en $O(n)$ si cette dernière est mal équilibrée. C'est pourquoi nous avons vérifié la hauteur de la branche la plus longue de notre ABR pour l'ensemble des mots de Shakespeare (23086 mots différents). Nous observons que cette dernière possède une hauteur de 32, ce

qui est relativement proche de $log(23086) \approx 15$ qui est la hauteur de l'arbre dans le cas où il serait équilibré. Nous avons ainsi considéré que notre structure était déjà assez pertinente pour ce contexte particulier et c'est pourquoi nous avons garder cette structure pour le reste du projet.

# 5 Étude expérimentale

**Question 6.15**

Lorsque l'on souhaite déterminer les ensembles de mots différents dans l'oeuvre de Shakespeare qui entrent en collisions pour `MD5`, on n'en trouve aucun.
En effet, cela s'explique par la quantité de valeurs de hachage qu'il est possible d'obtenir grâce à la fonction de hachage cryptographique considérée très supérieure au nombre de mots différents dans l'oeuvre de Shakespeare.
Montrons-le plus formellement.

Peu importe la longueur de la chaîne considérée, la fonction de hachage cryptographique `MD5` produit une empreinte sur 128 bits. Ainsi, nous avons $2^{128}$ valeurs de hachage possibles.
Dans l'oeuvre de Shakespeare, nous obtenons 23086 mots différents.

Analysons le nombre de collisions primaires qu'il pourrait y avoir.
Nous prendrons comme hypothèse l'uniformité de la fonction de hachage (nous pouvons vérifier expérimentalement l'effet d'avalanche que produit `MD5`).
La probabilité $P$ qu'il y ait au moins une collision est de :

$$P = 1 - \prod_{i=0}^{n-1}(1 - \frac{i}{m}) \approx 1 - e^{-\frac{n^2}{m}} \tag{6}$$

avec $m$ le nombre d'entrées dans la table de hachage et $n$ le nombre de clés.
De ce fait, nous avons $m = 2^{128}$ et $n = 23086$.
En approximant, grâce à [2], en valeur numérique ce calcul, on obtient le résultat suivant :

$$P = 1 - \prod_{i=0}^{23085}(1 - \frac{i}{2^{128}}) \approx 1 - e^{-\frac{23086^2}{2^{128}}} \approx 10^{-30} \approx 0 \tag{7}$$

Ainsi, on obtient une probabilité quasi nulle d'obtenir une collisions avec le hachage `MD5` sur l'ensemble des mots différents dans l'oeuvre de Shakespeare.

Si on calcule cette probabilité sur l'ensemble des mots de la langue anglaise, soit environ 200000, on obtient un résultat autour de $10^{-28}$. Il est donc toujours quasiment impossible d'obtenir une collision s'il l'on prend l'ensemble des mots de la langue anglaise.

Cela n'a finalement rien de surprenant. En effet, $2^{128} \approx 10^{38}$ ce qui est très largement supérieur à 23086 ou même à 200000, d'autant plus lorsque l'on sait que l'univers compte aux alentours $10^{80}$ particules.

**Question 6.16**

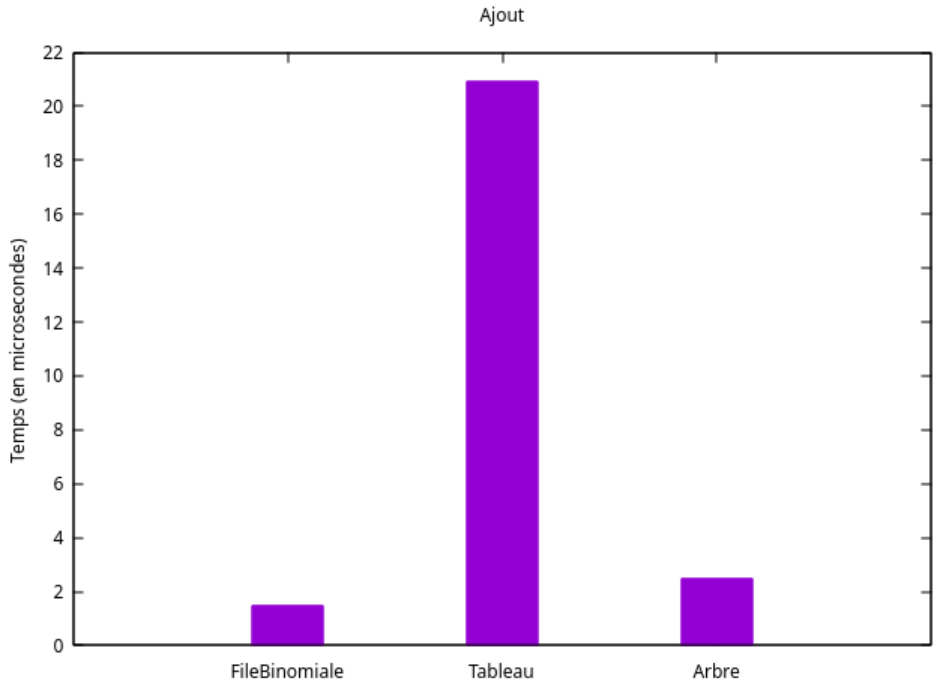


FIGURE 6 – Ajout sur une file, un tas via un tableau et un tas via un arbre

Non pouvons observer que le temps d'exécution de `Ajout` pour une structure sous forme de tableau est nettement plus longue que pour les autres structures. En effet, un ajout d'un élément dans un tableau dont la taille a été fixée au préalable nécessite la ré-allocation du tableau dans un espace mémoire plus grand, se traduisant ainsi par ce temps d'exécution plus long. En re-dimensionnant la taille du tableau avec la primitive `resize` fournie en C++ avant la mesure du temps d'exécution de `Ajout`, nous obtenons cette fois-ci un temps d'exécution de 0 microsecondes pour la structure sous forme de tableau. Cette différence d'exécution entre celle-ci et la structure arborescente s'explique par l'accès aux éléments qui se fait en temps constant dans le tableau (les données étant contiguës en mémoire), contre la structure arborescente qui nécessite un parcours. Aussi, le temps d'exécution plus faible pour une file binomiale peut s'expliquer par le fait qu'un ajout correspond à une `Union` dont la complexité pire cas est en $O(log(n))$. Dans le meilleur cas on obtient une complexité constante si la file binomiale ne contient pas de tournoi $TB_0$. Le tas sous forme arborescente, en revanche, a un temps d'exécution plus long car nous sommes toujours obligés de parcourir une branche du tas dans sa totalité pour ajouter un élément.

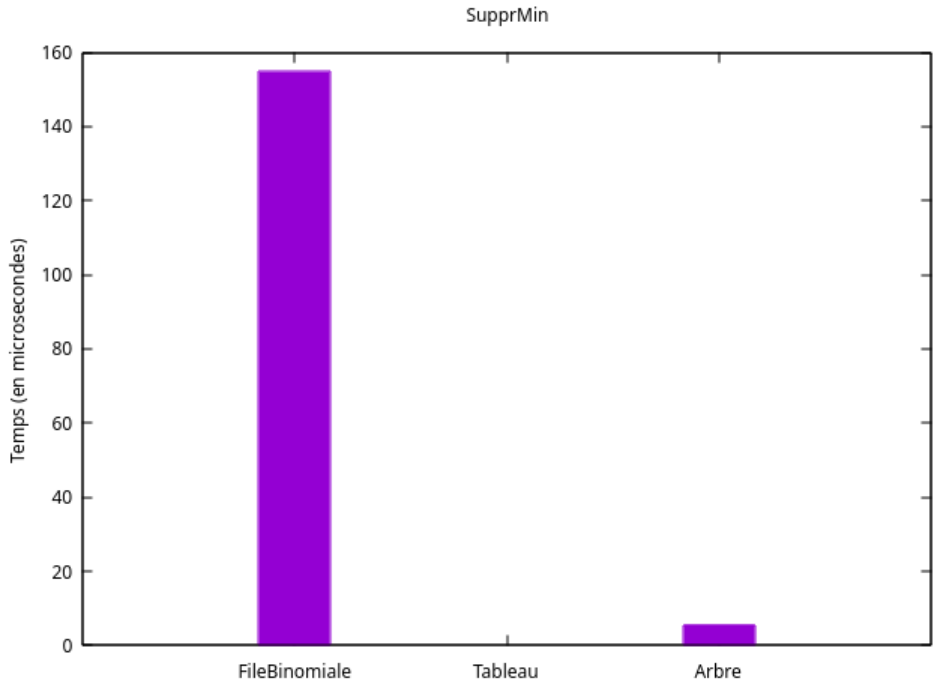


FIGURE 7 – SupprMin sur une file, un tas via un tableau et un tas via un arbre

Nous pouvons constater un temps d'exécution de `SupprMin` plus long pour les files binomiales que pour les tas. En effet, bien que ces structures présentent la même complexité, la file binomiale réalise d'abord $log(n)$ comparaisons pour trouver le tournoi de racine minimale parmi les tournois dont elle est composée, puis $log(n)$ comparaisons pour décapiter le tournoi trouvé dans la file et, enfin, $log(n)$ comparaisons pour faire l'union entre notre file initiale et la file issue de la décapitation du tournoi de racine minimale. Ainsi, `SupprMin` se réalise en $3 \times log(n)$ comparaisons pour une file binomiale.
Les tas, quant à eux, réalisent moins de comparaisons pour la suppression. En effet, pour la version avec un tableau on supprime le dernier élément du tableau (cela se fait en temps constant car il n'y a aucune désallocation) et on ne parcourt qu'une seule branche pour faire des échanges si besoin. Nous obtenons alors $log(n)$ comparaisons au pire des cas (cela peut être moins s'il n'est pas nécessaire de parcourir tout la branche). La version arborescente, elle, réalise deux parcours : un premier depuis la racine jusqu'à une feuille pour trouver l'élément à supprimer, ainsi qu'un second depuis la racine pour échanger les noeuds si besoin. Cela amène à $2 \times log(n)$ comparaisons au pire cas (cela peut être moins également pour les mêmes raisons qu'énoncées précédemment).
De ce fait, nous pouvons mieux comprendre pourquoi le temps d'exécution de `SupprMin` est plus long pour les files binomiales que pour les tas, bien qu'on ait une complexité similaire pour ces deux structures.

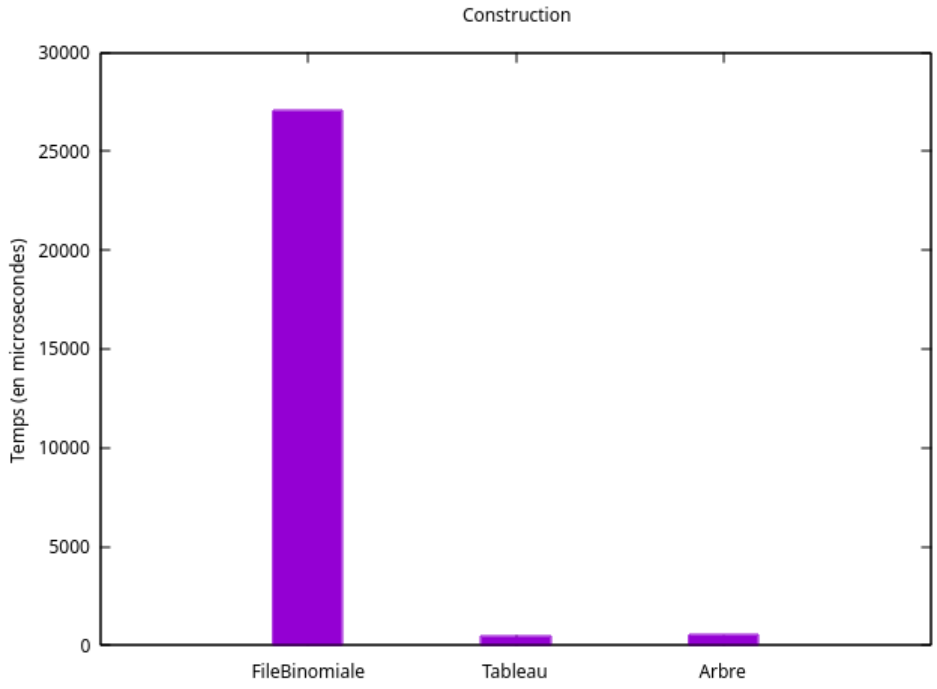


FIGURE 8 – Construction sur une file, un tas via un tableau et un tas via un arbre

Nous pouvons constater que le temps d'exécution de `Construction` est beaucoup plus long pour une file binomiale que pour un tableau et un arbre. En effet, l'algorithme de `Construction` est en fait équivalent à celle d'un `AjoutIteratifs` dans le sens où il fait un appel à `Ajout` sur chaque élément à ajouter. À chaque itération, une file contenant l'élément est allouée et une `Union` est ensuite effectuée entre cette file et la file actuelle. La complexité de `Ajout` étant en $O(log(n))$ comme vu précédemment, `Construction` se fait donc, *a priori*, en $O(n \times log(n))$ comparaisons pour une file binomiale. Cependant, un ajout dans une file binomiale ne coûte $log(n)$ comparaisons que lorsque la taille de la file vaut $2^x - 1$. De plus, l'ajout ne coûte aucune comparaison lorsque la taille de file est paire (c'est-à-dire la moitié des appels à `Ajout` dans notre construction). Bien que la complexité de `Construction` a été prouvée en $O(n)$ pour une file, il est possible que celle-ci soit réalise tout de même plus de comparaisons que dans un tas expliquant alors les différences de temps d'exécution.

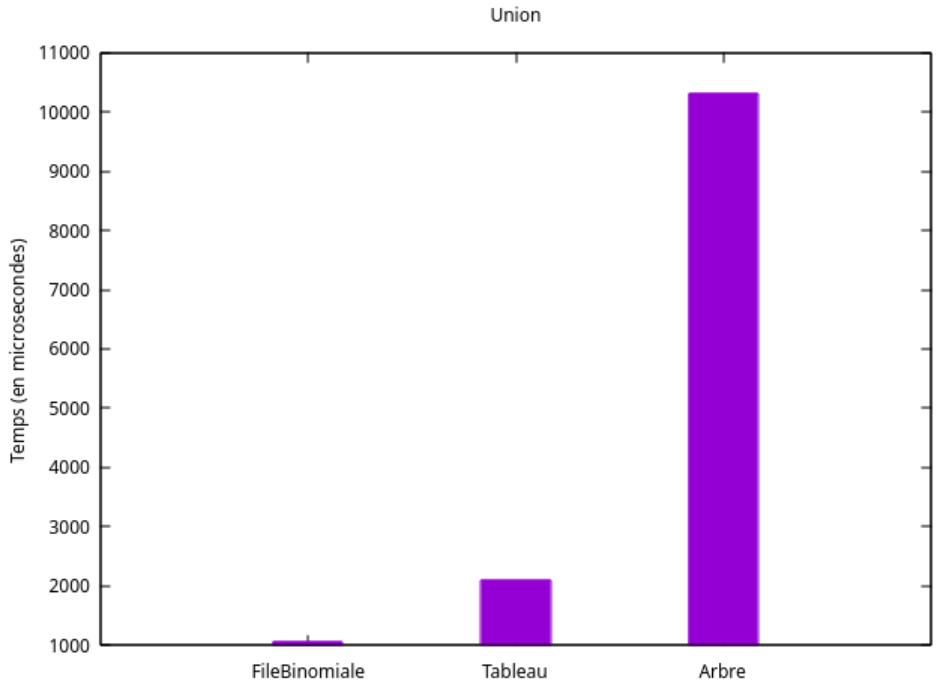


FIGURE 9 – Union sur une file, un tas via un tableau et un tas via un arbre

Afin de comparer les temps d'exécution de l'`Union`, nous avons choisi de diviser l'ensemble des mots de l'oeuvre de Shakespeare en deux parties distinctes de même taille.

Cette fois-ci, nous pouvons constater que cet algorithme s'exécute beaucoup plus rapidement pour une file binomiale que pour un tas. Par ailleurs, nous pouvons constater que la version arborescente s'exécute beaucoup plus rapidement pour une version via un tableau que via un arbre. En effet, pour une file binomiale nous sommes autour de 1 $ms$ de temps d'exécution. Cependant, avec un tas nous doublons notre temps d'exécution via un tableau et nous le multiplions par 10 via un arbre.
Cette différence de temps d'exécution s'explique par le fait que dans un tas, nous devons créer une nouvelle structure où nous ajoutons successivement les éléments minimums de nos deux tas à fusionner. Afin d'obtenir ces éléments, nous faisons des appels à la fonction `SupprMin` qui est en $O(log(n))$. Ainsi, nous devons comparer chaque élément de nos deux tas de départ ce qui nous donne bien une complexité en $O(n+m)$. Par ailleurs, nous pouvons noter que dans la version arborescente, nous réalisons un second parcours en $O(n+m)$ afin de mettre à jour les tailles des éléments qui ne sont pas des feuilles. Cela explique le temps d'exécution 5 fois plus long avec une complexité similaire pour un tas sous forme arborescente.
En revanche, lorsque nous souhaitons faire l'union de deux files binomiales nous ne comparons non pas chaque élément des deux files mais seulement les racines des tournois dont sont composées les files. Cela correspond bien à une complexité en $O(log(n+m))$.
Ces deux complexités correspondent bien aux complexités théoriques vues en cours.

# 6 Conclusion

En conclusion, le choix d'une structure contenant des données se fait en fonction des besoins spécifiques que l'on possède, mais dépend également des données sur lesquelles on travaille.
Le tas priorité min est une structure de données utile et efficace lorsque l'on souhaite ajouter successivement des éléments bruts. Quant à la file binomiale, celle-ci montre son efficacité lorsqu'on part de structures déjà existantes que l'on souhaite unir.

Par ailleurs, ce projet nous a permis de mieux nous familiariser avec les structures étudiées en cours. Nous avons pu vérifier et valider les complexités qui nous ont été fournies en classe, tout en évaluant en temps réel leur efficacité et ainsi prendre conscience de leur différence.

À travers ce projet, nous avons pu nous rendre compte que le choix du langage jouait une part importante quant à la réalisation de notre implémentation. En effet, le langage C++ réalise, plus souvent qu'on ne le pense, des copies ainsi que des réallocations implicites de la mémoire pour nos structures de données contrairement à des langages comme le C où nous devons gérer manuellement toutes les allocations. Cela implique des mesures de temps qui ne sont pas toujours concordantes avec les résultats attendus. Même avec l'algorithme le plus performant qui soit, une trop grande quantité de ré-allocations mémoires peut tout simplement empêcher la réalisation de tests sur cet algorithme. Bien qu'il soit nécessaire, lors de mesures, de faire la différence entre les temps système et utilisateur, il s'agit tout de même d'une part non négligeable lors des analyses de complexités.

Ainsi, ce projet nous aura permis de comparer l'utilisation des structures de données selon différents cas en fournissant une implémentation correcte qui respecte les complexités attendues.

# Références

[1] *C++ Reference*. 2023. URL : https://en.cppreference.com.

[2] *WolframAlpha*. 2023. URL : https://www.wolframalpha.com/.