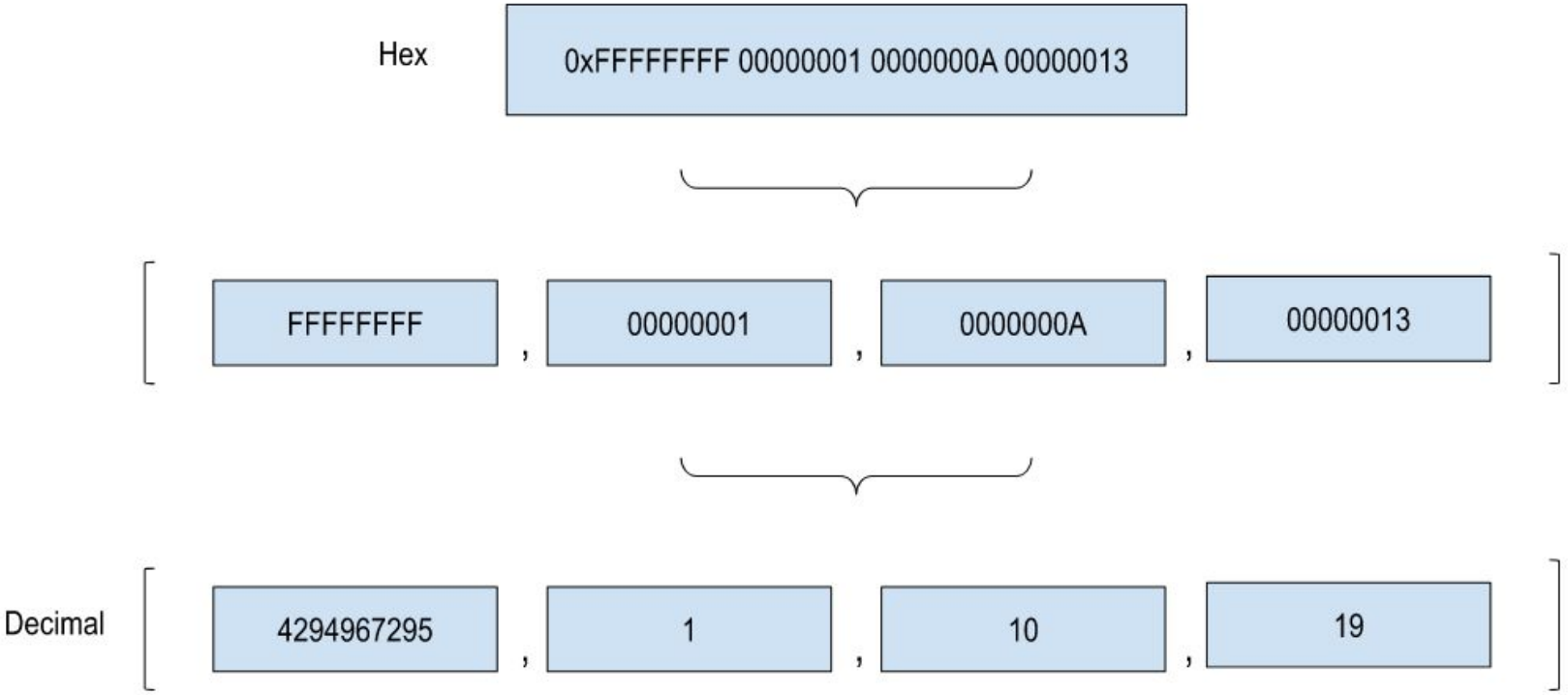
Master 1 : STL
MU4IN500 : Algorithmique Avancée

Devoir de Programmation

Alallah Yassine 28707696
Pham Thanh Tung 28631029

Année 2023-2024

Echauffement



Tas priorité Min

```
NoeudTasMin: # Structure représentant un noeud du tas

    clé = bi.Cle128Bits(clé)
    gauche = None
    droite = None
    parent = None
    taille = 1  # Initialise la taille à 1 car il s'agit d'un nouveau nœud
    hauteur = 0  # Initialise la hauteur à 0 car il s'agit d'une feuille

TasMin: # Structure représentant tas comme un ensemble de noeuds

    racine = None
    derniers_noeuds = [] # Liste des derniers noeuds ajoutés au tas
    liste_feuilles = [] # Liste des feuilles de l'arbre
```

Arbre

```
TasMin = [] # Notre tas est un simple tableau dont le père à
l'index i, a un fils gauche à la position 2 * index + 1 et un fils
droit à la position 2 * (index + 1)
```

Tableau

Fonction principales

AjoutsItératifs
(méthode Williams)

N insertions

Parcours d'une seule branche de l'arbre à chaque itération en **O(log(n))**

Construction
(méthode Floyd)

Construction d'un arbre respectant les propriétés d'un tas pendant le parcours ce dernier en **O(n)**

Puis appel à **descendre** sur chaque noeud interne afin de garantir les propriétés du tas priorité Minimum en **O(h)**

Union

Récupérer les éléments des deux TasMin en **O(n)**

Construction d'un tas à partir des clés des deux TasMin

SupprMin

Parcours de la structure pour récupérer le dernier élément si pas directement accessible en **O(1)** ou **O(log(n))**

Puis appel à **descendre** en **O(log(n))**

Complexité de deux fonctions auxiliaires importantes

CalculNoeud:

Prends en argument une liste de clés et renvoie une liste contenant la racine et deux listes contenant les clés des branches gauche et droite

Complexité en O(1)

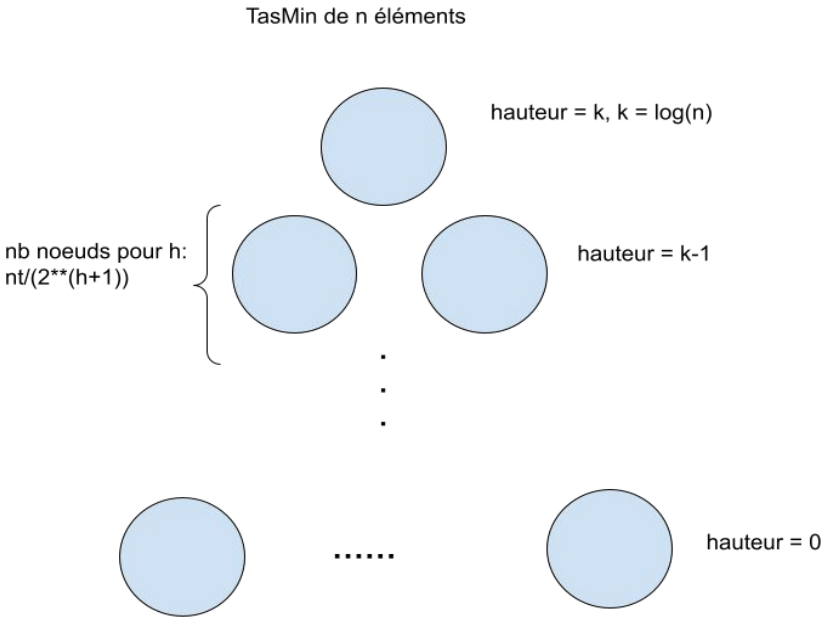
Descendre:

Fais la descente d'une clé dans l'arbre

- Recherche parmis les fils
- Vérifie que le parent est bien plus petit que l'un des fils
- Descends le parent appelle la fonction récursivement

Complexité en O(log(n))

Complexité de Construction



$$\sum_{0}^{\log n}\left[\frac{n_t}{2^{h+1}}\right] \times O(h)$$

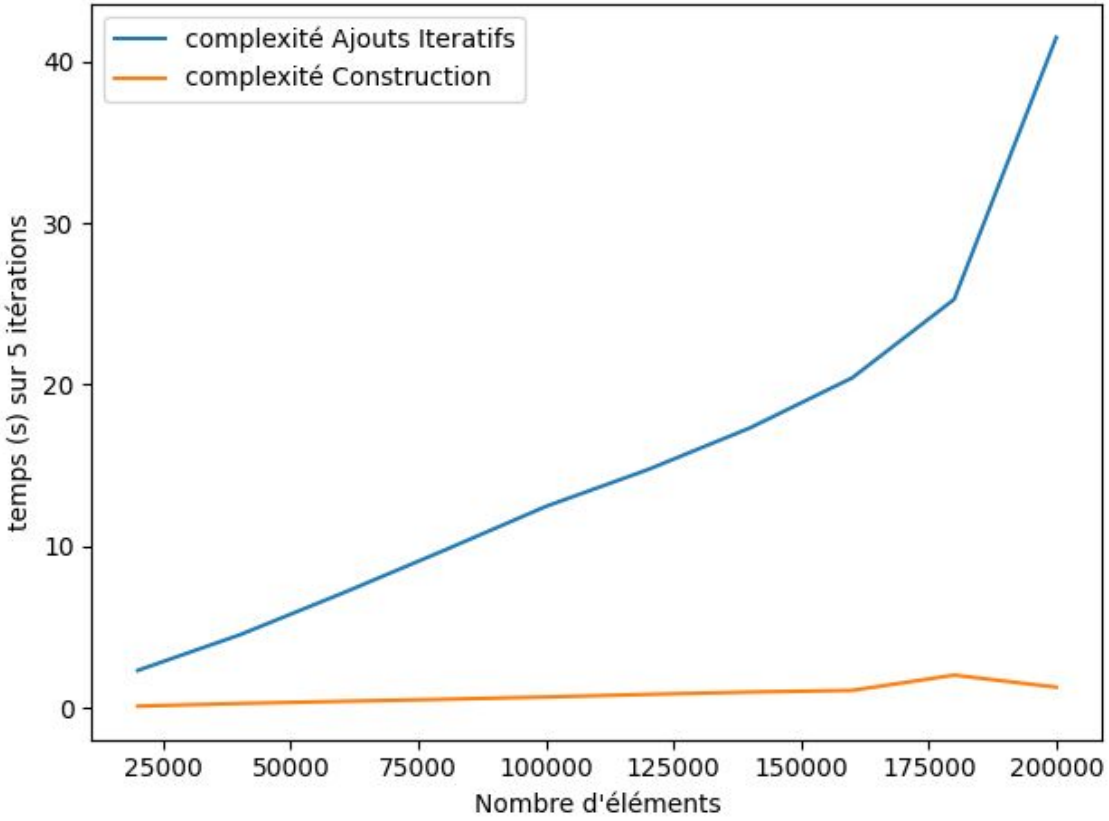
$$= \sum_{0}^{\log n}\left[\frac{nt}{2^h \times 2}\right] \times O(h)$$

$$= \frac{nt}{2}\sum_{0}^{\log n}\left[\frac{h}{2^n}\right]$$

$$= \frac{n_t}{2}\underbrace{\sum_{0}^{\infty}\left[\frac{h}{2^n}\right]}_{\text{tend vers } 2}$$

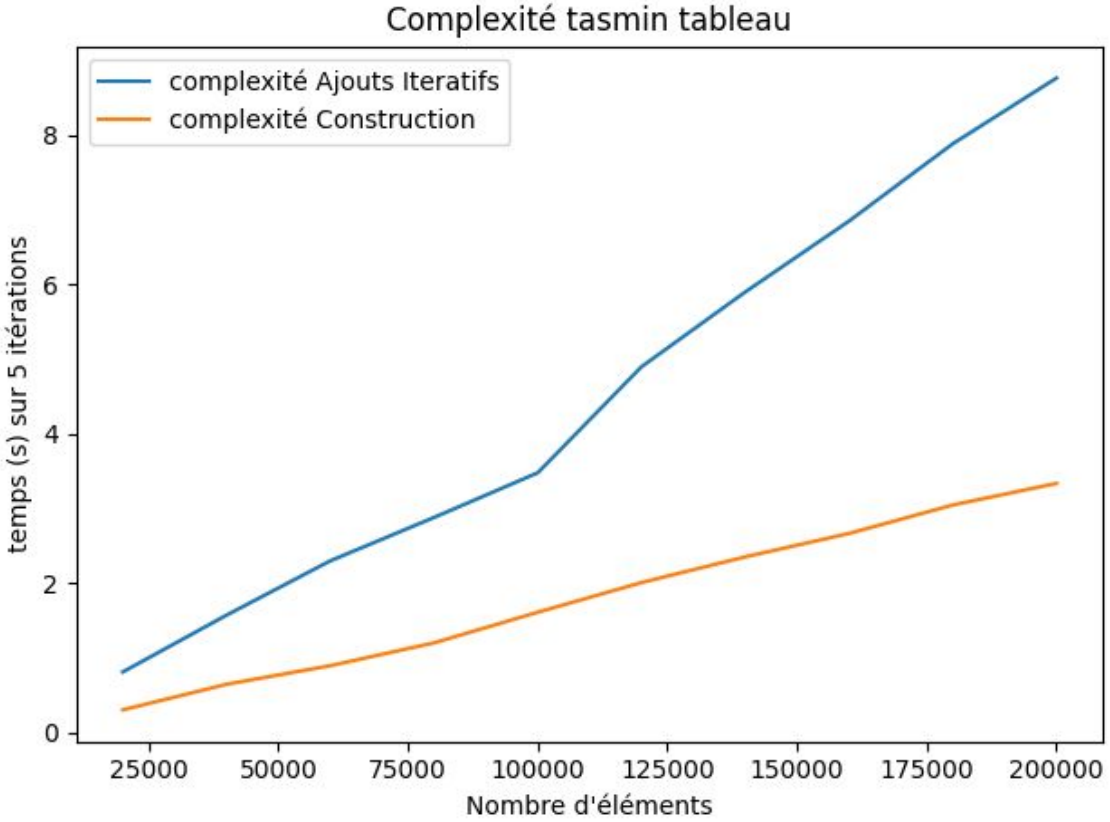
O(n)

Observations graphique



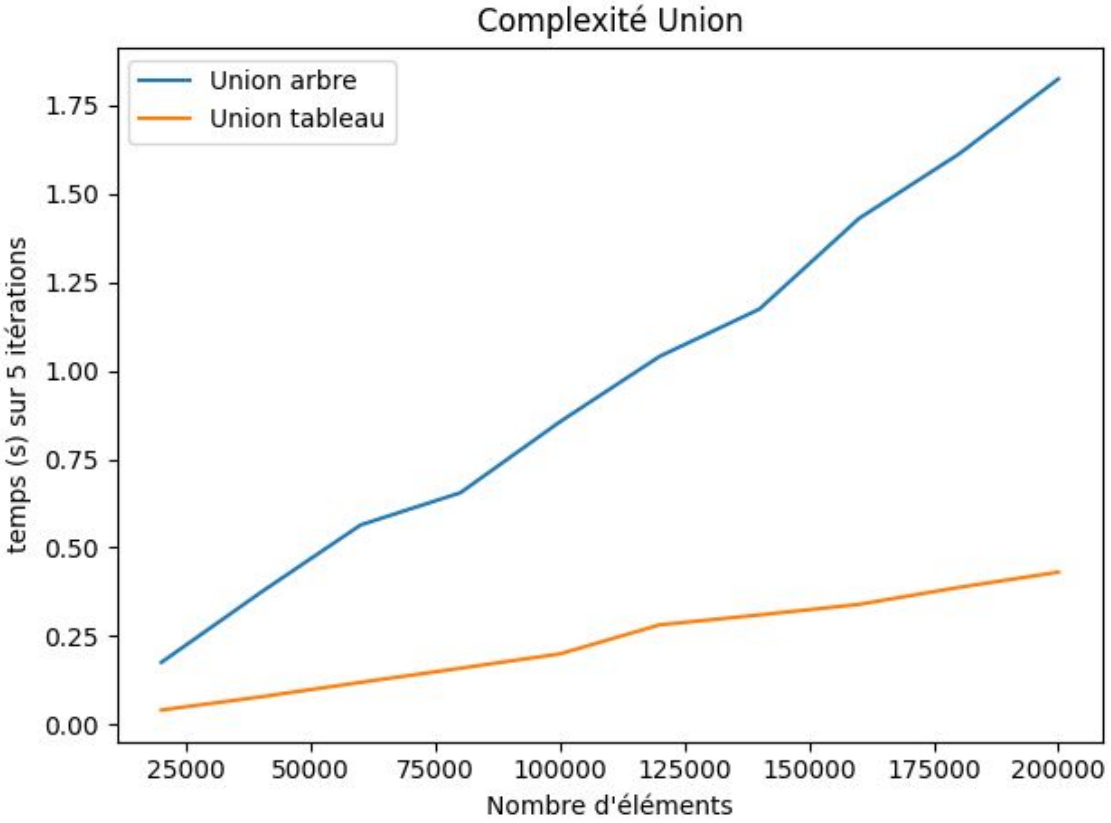
On a un meilleur temps d'exécution pour Construction O(n)

Observations graphique



On a un meilleur temps d'exécution pour Construction O(n)

Observations graphique



Meilleur temps d'exécution pour le tableau malgré la même complexité du fait des opérations plus coûteuses pour récupérer tous les noeuds de l'arbre.

File binomiale

```
Tournoi:
     T = [] # Contient la racine ainsi que les fils eux-même Tournoi
            # Tk = [racine, T0, T1, …, Tk-1]
     Degré # Degré du tournoi
File:
     LT = [] # Ensemble des Tournois de la file
     MinDegreIndex # Index du plus petit tournoi
     taille # Taille de la file
```

File10 = (3, racine : [602580185, 426232481, 3964731641, 138455936]) (1, racine : [1389535929, 832807153, 3526368565, 363335293])

Fonction principales

Construction

Opération:

1. **Création** d'une **file** avec un **seul tournoi** représentant notre élément en **O(1)**
2. Appel à **Union** sur notre file et la file à un élément en **O(1)**

N opérations en O(n)

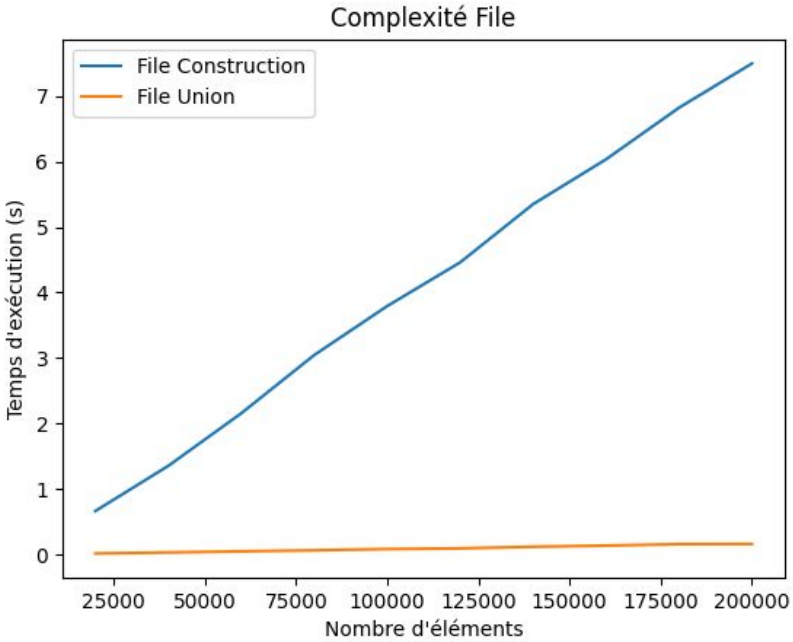
Union

Parcours des deux files, **comparaison** des éléments un à un et **insertion** du tournoi minimal dans la nouvelle file résultante, si deux tournois de même degré, **ajouter** le tournoi résultant de leur **fusion**

SupprMin

1. **Parcours** de la file en **log(n)** car log(n) éléments dans la file
2. **Retirer** le tournoi dont la racine est minimale et **former** une file à partir des **tournois fils** de l'élément
3. **Union** de la file courante à celle formée du tournoi sans sa racine

Observations graphique



Construction : O(n)

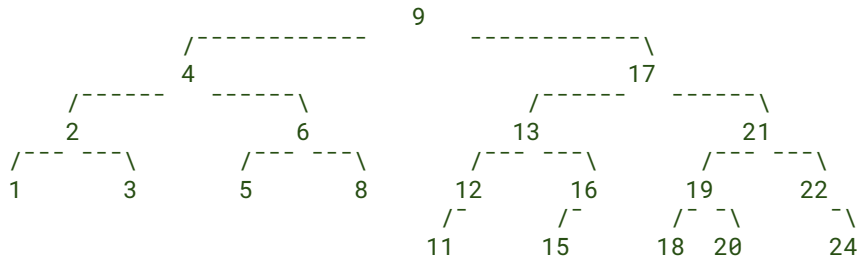
Union : O(log(n+m))

Arbre de recherche

Structure d’un arbre de recherche

```
ABR:
    # Attribut de classe partagé par toutes les
instances de la classe permettant de stocker les
clés qui entrent en collision
    liste_collision = {}
    # Conversion en Cle128Bits
    clé = Cle128Bits(clé)
    gauche = None
    droite = None
    parent = None
    hauteur = 0
```

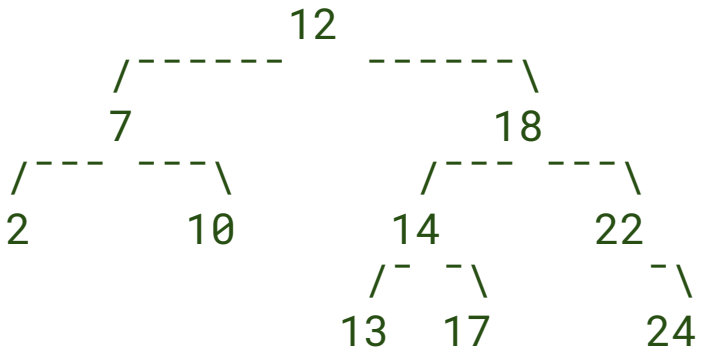
Représentation graphique de l’arbre



Arbre de recherche

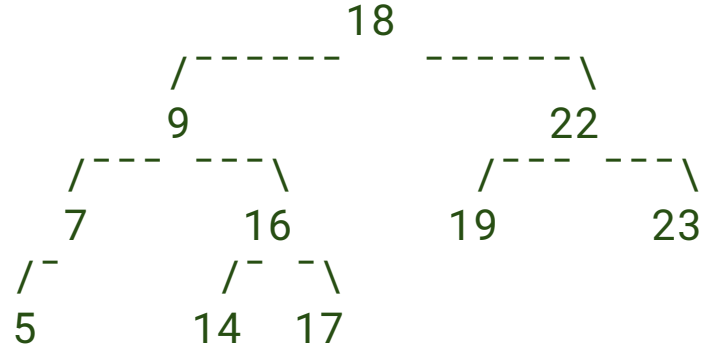
Arbre résultant d’une liste croissante

[2,7,10,12,13,14,17,18,22,24]



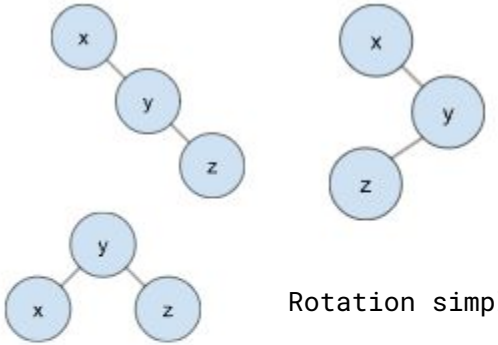
Arbre résultant d’une liste décroissante

[23,22,19,18,17,16,14,9,7,5]

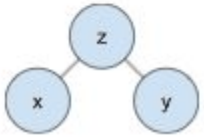


Arbre de recherche

Cas d'un arbre déséquilibré à droite

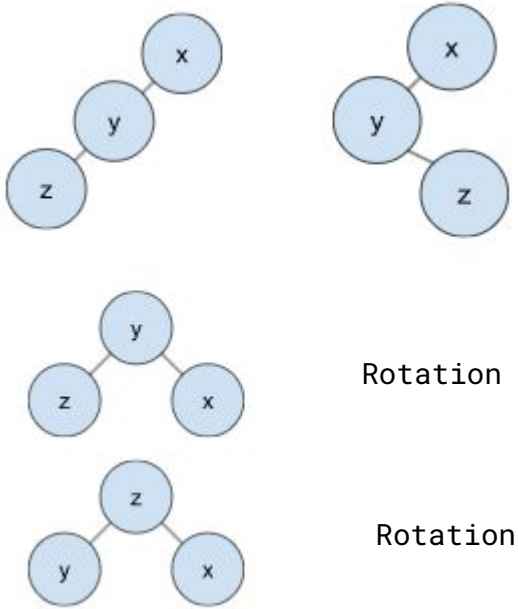


Rotation simple gauche



Rotation droite-gauche

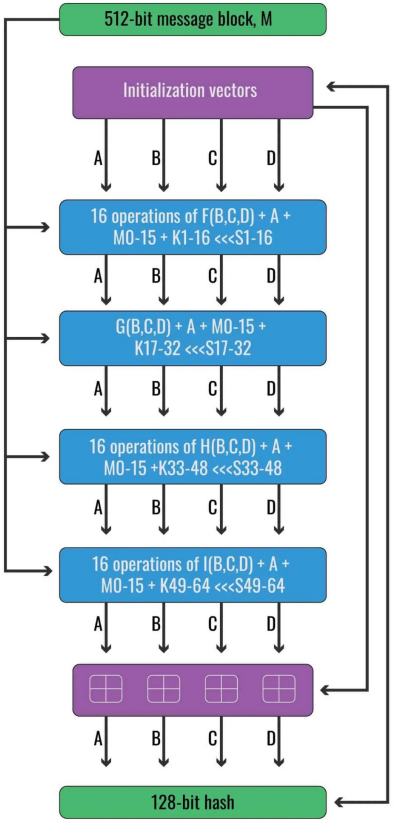
Cas d'un arbre déséquilibré à gauche



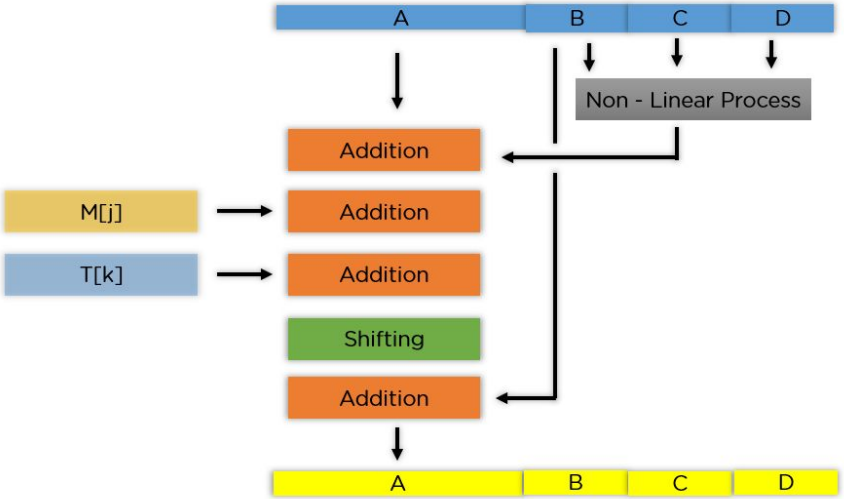
Rotation simple droite

Rotation gauche-droite

MD5

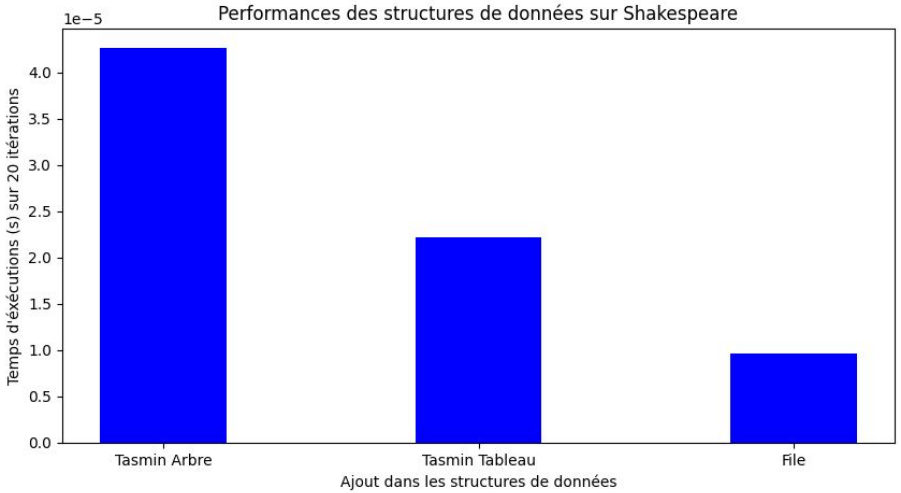


F G K R W H



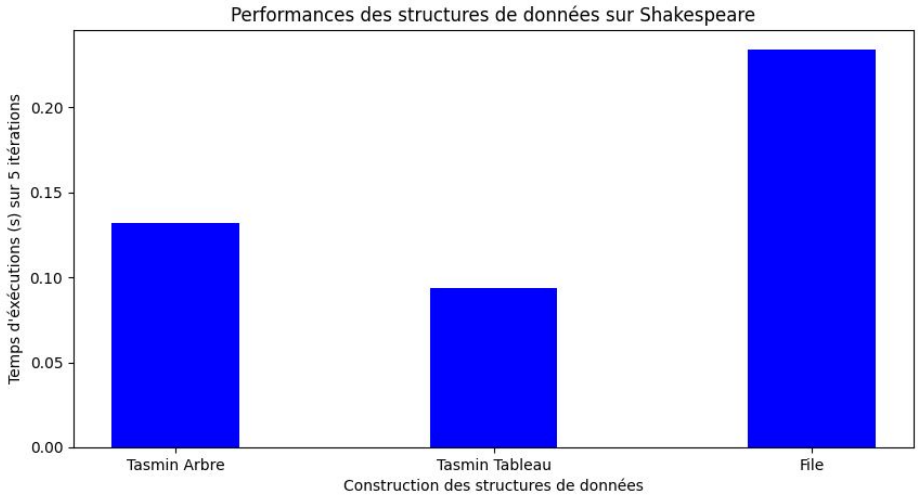
On observe aucune collision sur les données Shakespeare

Etude expérimentale sur Shakespeare



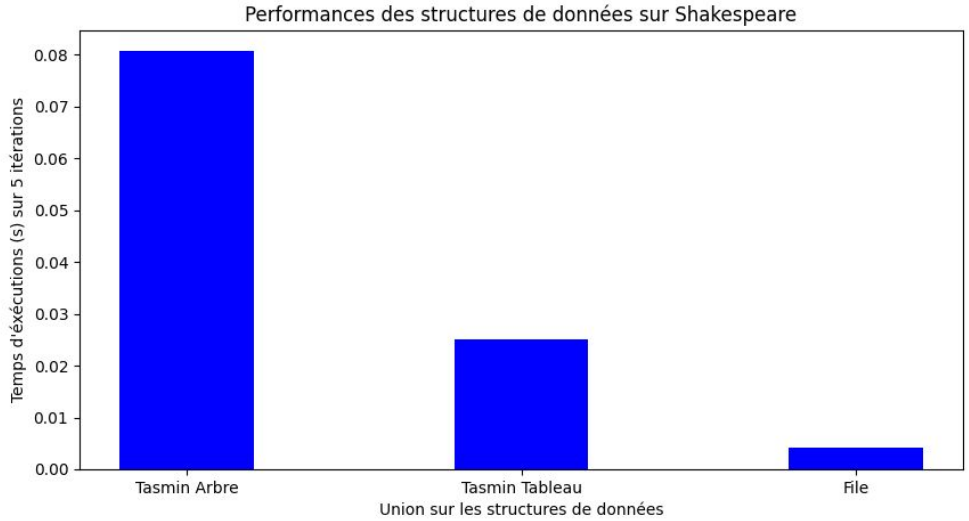
On observe que l’arbre est le moins performant car il doit chercher récursivement dans quelle branche de l’arbre insérer le nouvel élément

Etude expérimentale sur Shakespeare



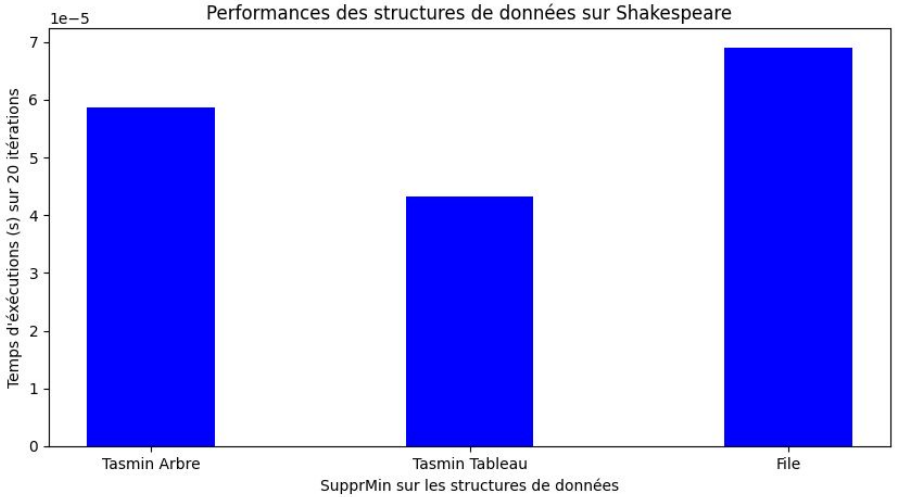
On observe que la file est la moins performante car à chaque ajout de noeud, elle fait l'union entre la file formée de la clé à ajouter et la file actuelle

Etude expérimentale sur Shakespeare



On observe que l’arbre est le moins performant à cause de l’opération pour récupérer tous les éléments dans les deux arbres qui est très coûteuse

Etude expérimentale sur Shakespeare



On observe que la file est la moins performante car elle fait l'union entre la file privé du tournoi de degré minimal et la file actuelle