Master 1 : STL

MU4IN500 : Algorithmique Avancée

Devoir de Programmation

Alallah Yassine 28707696
Pham Thanh Tung 28631029

Année 2023-2024

Sommaire

Introduction

Le but du problème consiste à visualiser graphiquement les temps d'exécution sur des données réelles des algorithmes et structures de données que nous avons introduits dans les chapitres 1, 2 et 3 du module.

Certaines fonctions utilisées par nos fonctions principales sont documentées en annexes.

Structure du code

- cles_alea : répertoire contenant des fichiers contenant des clés 128 bits aléatoires
- cles_sorted_2M : répertoire contenant un fichier comprenant deux millions de clés 128 bits en ordre croissant
- lib
  - hachage
    - MD5.py
  - sdd_bigint : répertoire contenant les structures compatibles 128 bits
    - abr_bigint.py
    - bigint.py
    - file_bigint.py
    - tasmin_binaire_bigint.py
    - tasmin_tableau_bigint.py
  - sdd_integer : répertoire contenant les structures en Integer afin de mieux les visualiser sur des petits entiers
    - abr.py
    - file.py
    - tasmin_binaire.py
    - tasmin_tableau.py

  - utilitaire.py : contient des fonctions annexes permettant notamment de manipuler des fichiers

- test
  - test_sdd_bigint : répertoire contenant un fichier test par structure
  - test_sdd_integer
  - test_MD5
  - test_Shakespeare : partie expérimentale

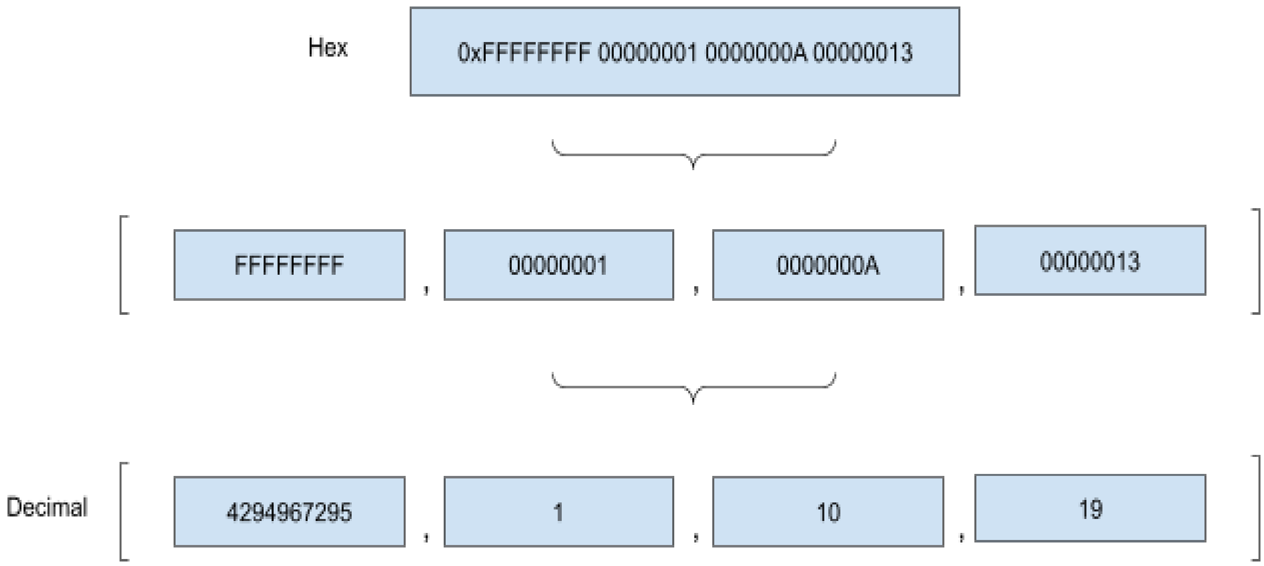
- main.py

Échauffement

Représentation d’une clé 128 bits

Pour représenter un clé 128 bits, nous avons procédé comme ceci :

- diviser les 32 caractères d’une clé 128 bits en 4 chaînes de 8 caractères chacune, soit 4 clés de 32 bits
- puis convertir chacune des 4 chaînes en décimal, on obtient alors un liste de 4 nombres compris entre 0 et 4 294 967 295 (car non signé).



How To:

```
#Importer les fichiers nécessaire
import lib.sdd_bigint.bigint as bi

#Création des entiers 128Bits
#Le constructeur accepte des strings de longueur de 32 ou 34 caractères à
cause des suffixes “0x” et les convertit
grand_entier1 = bi.Cle128Bits(“0xdf6943ba6d51464f6b02157933bdd9ad”)
grand_entier2 = bi.Cle128Bits(“d192acf4c06fe7c7df042f07d290bdd4”)
print(grand_entier1.toString())
print(grand_entier2.toString())
#attendu:
```

```
[3748217786, 1834042959, 1795298681, 868080045]
[3516050676, 3228559303, 3741593351, 3532701140]
#L'affichage en décimal des quatres parties en 32 bits des clés
```

Prédicats

```
Pseudo-Code de inf: La fonction compare les quadruplets des deux clés
Cle128Bits et renvoie True si la clé1 < clé2 sinon False.
```

```
Inf(clé1, clé2):
    Pour chaque i de 0 à 3 inclus:
        Si self.quadruplet[i] > quad.quadruplet[i]:
            Retourner False
        Sinon si self.quadruplet[i] < quad.quadruplet[i]:
            Retourner True
        Sinon:
            Si i égal à 3:
                Retourner False
            Sinon:
                Continuer
```

Complexité: **O(1)**

```
Pseudo-Code de eq: Vérifie si deux clés Cle128Bits sont égales.
```

```
Eq(clé1, clé2):
    Retourner clé1.quadruplet == clé2.quadruplet
```

Complexité: **O(1)**

How To:

```
import lib.sdd_bigint.bigint as bi #Importer les fichiers nécessaire

#Création des entiers 128Bits
grand_entier1 = bi.Cle128Bits("0xdf6943ba6d51464f6b02157933bdd9ad")
grand_entier2 = bi.Cle128Bits("0xd192acf4c06fe7c7df042f07d290bdd4")

#Affichage
print(grand_entier2.inf(grand_entier1))
print(grand_entier1.eq(grand_entier2))
```

```
#attendu: True
          False
```

Structure 1 : Tas priorité min

En cours, nous avons introduit la structure de données de tas min. Nous allons la représenter en mémoire avec deux structures distinctes : via un arbre binaire et via un tableau.

Structure 1.1 : Tas priorité min en arbre

Représentation

```
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
```

Fonctions fondamentales d’un tas min en arbre

Ajout et AjoutsItératifs

```
Pseudo-Code de Ajout: Ajoute un élément avec la clé spécifiée au tas binaire.
```

```
Ajout(clé):
    nouveau_noeud = NoeudTasMin(clé)
    # Si la racine est vide, le nouveau nœud est la racine
    si racine != nil:
        racine = nouveau_noeud
    sinon:
    # Appel de la fonction récursive rec_Ajout pour naviguer et insérer
nouveau_noeud dans la racine
        rec_Ajout(racine, nouveau_noeud)
```

Complexité: La complexité dépend de la hauteur de l'arbre binaire. Dans le pire cas, l'arbre est complètement équilibré, et la hauteur est logarithmique par rapport au nombre de nœuds avec taille = log2(hauteur) pour un tas. Donc, la complexité de l'ajout est **O(log n)**.

How To:

```
#Importation des fichiers nécessaires
import lib.sdd_bigint.tasmin_binaire_bigint as tasabr
import lib.sdd_bigint.bigint as bi

#Création de l'arbre représentant le tas
tas = tasabr.TasMin()
#Ajout d'une clé dans l'arbre
tas.Ajout(bi.Cle128Bits("0xdf6943ba6d51464f6b02157933bdd9ad"))
#Affichage de l'arbre
tas.afficher_arbre(tas.racine)
# attendu:

└── [3748217786, 1834042959, 1795298681, 868080045]
```

Pseudo-Code de rec_Ajout: Ajoute un nœud au tas.

```
rec_Ajout(parent, nouveau_noeud):

    si parent.gauche == nil:
        parent.gauche = nouveau_noeud
        # Met à jour la référence du parent pour le nouveau_noeud
        nouveau_noeud.parent = parent

        # Ajoute le noeud à la liste des noeuds
        last_node += [nouveau_noeud]
    sinon si parent.droite == nil:
        parent.droite = nouveau_noeud
        # Met à jour la référence du parent pour le nouveau_noeud
```

```
        nouveau_noeud.parent = parent
        # Ajoute le noeud à la liste des noeuds
        self.last_node.append(nouveau_noeud)
    sinon:
        # Ajoute le nœud à la branche appropriée
        si parent.gauche.hauteur == parent.droite.hauteur:
            # Si les hauteurs des branches gauche et droite sont égales
            si parent.gauche.taille == parent.droite.taille:
                # Si les branches gauche et droite ont la même taille, on
continue à explorer la gauche
                rec_Ajout(parent.gauche, nouveau_noeud)
            sinon si parent.gauche.taille % 2 != 0:
                # Si la taille de la branche gauche est impaire, on continue
à explorer la droite
                rec_Ajout(parent.droite, nouveau_noeud)
            sinon:
                # Sinon, on continue à explorer la gauche
                rec_Ajout(parent.gauche, nouveau_noeud)
        sinon:
            # Si les hauteurs des branches gauche et droite sont différentes
            si parent.gauche.taille % 2 != 0 et parent.gauche.gauche.taille
== parent.gauche.droite.taille:
                # Si la taille de la branche gauche est impaire et ses deux
sous-branches ont la même taille, on explore la droite
                rec_Ajout(parent.droite, nouveau_noeud)
            sinon:
                # Sinon, on explore la gauche
                rec_Ajout(parent.gauche, nouveau_noeud)

    # Réorganise le tas après l'ajout
    parent.majNoeud()
```

Complexité: Ici, toutes les opérations sont en O(1), **get_hauteur()** et **majNoeud()**. Donc la complexité dépend de la hauteur de l'arbre binaire. Soit log(n) avec n, la taille du tas. On a donc O(log(n)).

Pseudo-Code de AjoutsIteratifs: Ajoute les éléments de la liste en paramètres au tas binaire.

```
AjoutsIteratifs(liste):
    # Itération sur la liste et ajout au tas
    Pour chaque élément i dans liste:
        Ajout(i)
```

Complexité : N (taille de la liste) * complexité de **rec_Ajout**, soit N*log(n). Donc on a dans le pire cas **O(n*log(n))**

How To:

```
#Importation des fichiers nécessaires
import lib.utilitaire as ut
import lib.sdd_bigint.tasmin_binaire_bigint as tasabr

#Création de l'échantillon d'éléments (10 éléments)
#La fonction prog_list_constr renvoie une liste de listes (dans ce cas spécifique une liste contenant une liste de 10 éléments). Comme cette fonction a été codé pour la phase de test final sur l'ensemble des clés aléa, une description plus détaillé de la fonction est fourni dans le code source en commentaire
samples = ut.prog_list_constr(10, 10, 0)
#Création de l'arbre représentant le tas
tas = tasabr.TasMin()
#Ajout des éléments au tas
tas.AjoutsIteratifs(samples[0])
#Affichage de l'arbre
tas.afficher_arbre(tas.racine)

#Comment analyser l'arbre affiché:
   -  Plus on descends dans l'arbre, plus les lignes sont indentés
   -  La branche gauche est affiché en premier
#attendu:
└── [602580185, 426232481, 3964731641, 138455936] #racine
      ├── [1356453159, 3293574679, 2727937134, 3290195753] #branche gauche
      |    ├── [1389535929, 832807153, 3526368565, 363335293]
      |    |    ├── [3983630590, 954020349, 4102041063, 576193144]
      |    |    └── [3411245654, 3059252211, 3928924417, 1609824262]
      |    └── [2398824147, 3480035218, 487421168, 3961739784]
      |         └── [3395898027, 1353677783, 1152707669, 2983853358]
      └── [1816088954, 2257925604, 3579840626, 1581852887] #branche droite
           ├── [1834156317, 1832645408, 132768699, 622939054]
           └── [2269505948, 1979310886, 1859389282, 643936639]
```

SupprMin

Pseudo-Code de SupprMin: Supprime l'élément à la racine du tas binaire et garantie les propriétés du tas.

```
SupprMin():
    # Vérifie si la racine est nulle
    si racine == nil:
        retourner nil

    # Cas où la racine n'a pas d'enfants
    si self.racine.gauche == nil et non self.racine.droite == nil:
        r = racine.cle
        racine = nil
        retourner r

    # Vérifie si last_node est vide, si oui, appelle last_one() pour le
remplir
    si longueur(last_node) == 0:
    # Va chercher le dernier élément dans le tas
        last_one(racine)

    # Initialise tmp avec le dernier élément de last_node
    tmp = last_node[-1]

    # Parcours de la branche
    tant que tmp.parent n'est pas racine:
        # Diminue la taille des ancêtres du nœud à supprimer
        tmp.parent.taille -= 1
        tmp = tmp.parent

    # Échange avec le dernier élément
    dernier_noeud = last_node[-1]

    # Switch des clés entre la racine et le dernier_noeud
    r = self.racine.cle
    switch(racine.cle, dernier_noeud.cle)
    # On supprime le dernier élément
    last_node.pop()

    # Retirer le dernier nœud (anciennement le nœud de clé minimale)
    si dernier_noeud.parent != nil:
        si dernier_noeud.parent.gauche == dernier_noeud:
            dernier_noeud.parent.gauche = nil
        sinon:
            dernier_noeud.parent.droite = nil
    sinon:
        self.racine = None

    # Fait descendre le premier élément pour maintenir la propriété du tas
    descendre(racine)
    retourner r
```

```
Complexité: Ici SupprMin est en O(log(n)), plus précisément O(3*log(n)), car on présente dans le corps de la fonction une boucle while qui permet de remonter d'un noeud à la racine, soit en log(n) et on dépend de deux fonction : descendre() qui parcourt le chemin et inverse les clés si besoin et last_one() qui parcourt le chemin de la racine au dernier noeud ajouté, en log(n) aussi.
```

```
How To:

#Importation des fichiers nécessaires
import lib.utilitaire as ut
import lib.sdd_bigint.tasmin_binaire_bigint as tasabr

#Création de l'échantillon d'éléments
samples = ut.prog_list_constr(10, 10, 0)
#Création de l'arbre représentant le tas
tas = tasabr.TasMin()
#Ajouts des éléments dans le tas
tas.AjoutsIteratifs(samples[0])
#Suppression de l'élément minimal du tas
tas.SupprMin()
#Affichage de l'arbre
tas.afficher_arbre(tas.racine)
#attendu:
└── [1356453159, 3293574679, 2727937134, 3290195753]
     ├── [1389535929, 832807153, 3526368565, 363335293]
     │   ├── [3395898027, 1353677783, 1152707669, 2983853358]
     │   │   ├── [3983630590, 954020349, 4102041063, 576193144]
     │   │   └── [3411245654, 3059252211, 3928924417, 1609824262]
     │   └── [2398824147, 3480035218, 487421168, 3961739784]
     └── [1816088954, 2257925604, 3579840626, 1581852887]
         ├── [1834156317, 1832645408, 132768699, 622939054]
         └── [2269505948, 1979310886, 1859389282, 643936639]
```

Construction

```
Pseudo-Code de Construction: Construction réalise la construction d'un tas binaire à partir d'une liste d'éléments.
```

```
Construction(liste):

    # Vérification si le tas a déjà une racine (déjà construit)
    si racine != nil:
        retourner
```

```
    # Initialisation de la liste à construire et appel de rec_Construction()
pour démarrer la construction
    to_build = liste
    rec_Construction(Vrai, None, longueur(liste))
```

Complexité: La complexité dépend de rec_Construction.

How To:

```
#Importation des fichiers nécessaires
import lib.utilitaire as ut
import lib.sdd_bigint.tasmin_binaire_bigint as tasabr

#Création de l'échantillon d'éléments
samples = ut.prog_list_constr(10, 10, 0)
#Création de l'arbre représentant le tas
tas = tasabr.TasMin()
#Via la méthode Construction
tas.Construction(samples[0])
#Affichage de l'arbre
tas.afficher_arbre(tas.racine)
#attendu:
└── [602580185, 426232481, 3964731641, 138455936]
      ├── [1389535929, 832807153, 3526368565, 363335293]
      │   ├── [1816088954, 2257925604, 3579840626, 1581852887]
      │   │   ├── [3983630590, 954020349, 4102041063, 576193144]
      │   │   └── [2269505948, 1979310886, 1859389282, 643936639]
      │   └── [3395898027, 1353677783, 1152707669, 2983853358]
      │       └── [3411245654, 3059252211, 3928924417, 1609824262]
      └── [1356453159, 3293574679, 2727937134, 3290195753]
          ├── [2398824147, 3480035218, 487421168, 3961739784]
          └── [1834156317, 1832645408, 132768699, 622939054]
```

Pseudo-Code de rec_Construction: Construction réalise la construction d'un tas binaire. D'abord en créant un tas puis le rééquilibre.

```
rec_Construction(gd, parent, taille):
   """
    Arguments:
       - gd: Un booléen indiquant si le nœud actuel est le fils gauche (True)
ou le fils droit (False) du nœud parent.
       - parent: Le nœud parent actuel pour lequel le fils (gauche ou droit)
est en cours de construction.
       - taille: La taille totale du sous-arbre en cours de construction.
   """
```

```
    # Cas de base: arrêt de la construction si la taille est nulle ou la liste
à construire est vide
    si taille == 0 ou longueur(to_build) == 0:
        retourner

    # Récupération des informations sur la répartition des nœuds dans le
sous-arbre
    # Calcul noeud retourne le nombre de noeud à gauche et à droite
    res = calculNoeud(taille)

    # Cas initial: construction de la racine du tas
    si non parent:
        racine = NoeudTasMin(to_build.pop())
    # Gauche
        rec_Construction(Vrai, racine, res[1])
    # Droite
        rec_Construction(Faux, racine, res[2])
    sinon:
        # Construction du fils gauche ou droit en fonction de 'gd'
        si gd:
            parent.gauche = NoeudTasMin(self.to_build.pop())
            si res[1] != 0:
                # si le nombre de noeuds à ajouter au fils gauche est
différent de 0
                rec_Construction(Vrai, parent.gauche, res[1])
            si res[2] != 0:
                # si le nombre de noeuds à ajouter au fils droit est différent
de 0
                rec_Construction(Faux, parent.gauche, res[2])
            # Maj noeud
            parent.taille = 1 + parent.gauche.taille
            parent.gauche.parent = parent
        sinon:
            parent.droite = NoeudTasMin(to_build.pop())
            si res[1] != 0:
                # si le nombre de noeuds à ajouter au fils gauche est
différent de 0
                rec_Construction(Vrai, parent.droite, res[1])
            si res[2] != 0:
                # si le nombre de noeuds à ajouter au fils droit est différent
de 0
                rec_Construction(Faux, parent.droite, res[2])
            # maj des attributs
            parent.taille += parent.droite.taille
            parent.droite.parent = parent

    # Descendre pour maintenir la propriété du tas
```

```
    si parent == nil:
        retourner
    descendre(parent)
```

Complexité: Ici on fait appel à plusieurs fonctions dont pop() en O(1), **calculNoeud()** en O(1), **descendre()** en O(log(n)). On crée le tas, en initialisant chaque nœud de l'arbre via **CalculNoeud**() qui nous permet de connaitre le nombre de nœuds à gauche et à droite à chaque tour. Étant donné qu'on parcourt d'abord chaque nœud, on est en O(n) puis avant le dépilement on appelle **descendre**() à partir des parents de la base.

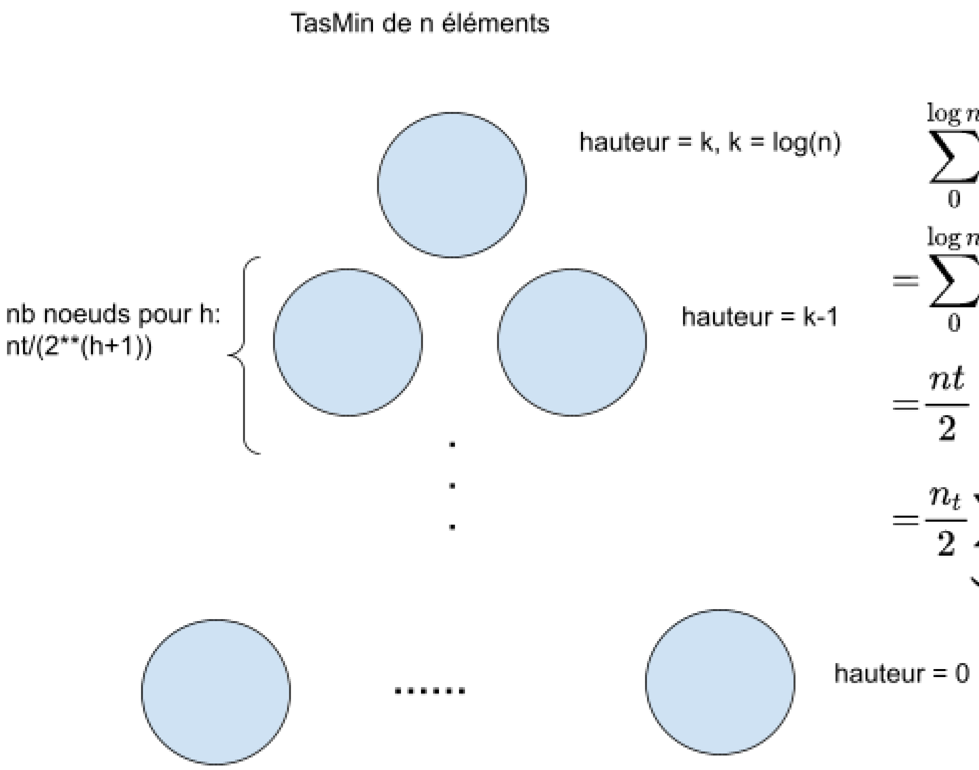
Chaque appel réalisera h appel à **descendre**() avec h la hauteur du nœud dans le pire cas.

On se retrouve alors en **O(n)** pour la formation du tas +
W = [somme(nb_noeuds_internes_h * O(h)) avec nb_noeuds_internes_h, le nombre de noeuds à chaque hauteur, soit (nombre de noeuds total dans le tas divisé par (2**(h+1))]

En développant **W**, on se retrouve avec une suite de la forme :
(nb_total_noeud_tas/2) * somme(h/2**h), lorsque h tend vers l'infini le deuxième terme tend vers 2. On a donc pour W = nb_total_noeud_tas -> **O(n)**

Enfin, on se retrouve en O(2*n) -> **O(n)**

Démonstration:



$$\sum_0^{\log n}\left[\frac{n_t}{2^{h+1}}\right]\times O(h)$$

$$=\sum_0^{\log n}\left[\frac{nt}{2^h\times 2}\right]\times O(h)$$

$$=\frac{nt}{2}\sum_0^{\log n}\left[\frac{h}{2^n}\right]$$

$$=\frac{n_t}{2}\underbrace{\sum_0^{\infty}\left[\frac{h}{2^n}\right]}_{\text{tend vers 2}}$$

Union

Pseudo-Code de Union: La fonction Union combine deux tas binaires minimaux en fusionnant leurs feuilles respectives dans un nouveau tas binaire minimal.

```
Union(Tas1 = self, Tas2):
    # Remise à 0 de liste feuilles
     Tas1.liste_feuille = []
     tas2.liste_feuille = []

    # Obtention des listes de feuilles de chaque tas
    l1 = liste_feuille()
    l2 = liste_feuille()

    if l2 == []:
     return Tas1
     if l1 == []:
     return Tas2

    # Fusion des listes de feuilles
    pour chaque élément i dans l2:
        l1.append(i)

    # Création d'un nouvel objet TasMin et construction du tas binaire minimal
résultant
    tasres = TasMin()
    retourner tasres.Construction(l1)
```

Complexité: Ici on appelle **liste feuille()** en O(n) avec n la taille de la liste. On a donc O(n), pour le tas1 et O(m) pour le tas2, puis l'ajout du tas2 dans tas 1 en O(max(n, m)) et enfin construction sur la combinaison des 2 tas en O(n+m).
On peut majorer chaque complexité par O(n+m) et obtenir, O(4*(n+m)), Soit O(n+m)

How To:

```
#Importation des fichiers nécessaires
import lib.utilitaire as ut
import lib.sdd_bigint.tasmin_binaire_bigint as tasabr

#Création des échantillons d'éléments (5 éléments chaque)
samples1 = ut.prog_list_constr(5, 5, 0)
samples2 = ut.prog_list_constr(5, 10, 5)
#Création des arbres représentant les deux tas à unir
tas1 = tasabr.TasMin()
tas2 = tasabr.TasMin()
```

```
tas1.Construction(samples1[0])
tas2.Construction(samples2[0])
#Affichage de l'union
tas1.afficher_arbre(tas1.Union(tas2).racine)
#attendu:
└── [602580185, 426232481, 3964731641, 138455936]
      ├── [1816088954, 2257925604, 3579840626, 1581852887]
      │   ├── [2269505948, 1979310886, 1859389282, 643936639]
      │   │   ├── [3983630590, 954020349, 4102041063, 576193144]
      │   │   └── [3395898027, 1353677783, 1152707669, 2983853358]
      │   └── [1834156317, 1832645408, 132768699, 622939054]
      │        └── [2398824147, 3480035218, 487421168, 3961739784]
      └── [1356453159, 3293574679, 2727937134, 3290195753]
          ├── [3411245654, 3059252211, 3928924417, 1609824262]
          └── [1389535929, 832807153, 3526368565, 363335293]
```

Structure 1.2 : Tas priorité min en tableau

Représentation

```
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)
```

Fonctions fondamentales d'un tas min en arbre

Ajout et Ajouts Itératifs

```
Pseudo-Code de Ajout: Ajoute un élément avec la clé spécifiée au tas binaire.
```

```
Ajout(Tas, elt):
  Tas += [Cle128Bits(elt)]
  # On vérifie que la structure est toujours un tas depuis le dernier élément
  monter(Tas, longueur(Tas) - 1)
  retourner Tas
```

Complexité: La fonction d'ajout ajoute l'élément en queue de tableau en O(1) et appelle la fonction monter, qui a une complexité temporelle en O(log n) dans le pire cas. Ainsi, la complexité temporelle totale de la fonction Ajout est également en O(log n).

Pseudo-Code de AjoutsIteratifs: Ajoute les éléments de la liste en paramètres au tas binaire.

AjoutsIteratifs(Tas, data):
    si longueur(data) == 0:
      retourner Tas
 	
    Pour i allant de 0 à longueur(data)-1:
      Ajout(Tas, data[i])
 	
    retourner Tas

Complexité: La fonction itère sur la liste de données et appelle Ajout pour chaque élément. Si la taille de la liste de données est n et la taille actuelle du tas est n, la complexité temporelle pire cas serait en O(n * log n), car pour chaque élément on remonte vers la racine en O(log(n)) pire cas pour s'assurer les propriété du tas. Cas d'un chemin qui doit être parcouru entièrement : un tableau de valeurs décroissantes.

How To:

```
#Importation des fichiers nécessaires
import lib.sdd_bigint.tasmin_tableau_bigint as tastab
import lib.utilitaire as ut

#Création de l'échantillon d'éléments
samples = ut.prog_list_constr(10, 10, 0)
#Création du tas sous forme de tableau
tas = []
#En utilisant AjoutsIteratifs
tas = tastab.AjoutsIteratifs(tas, samples[0])
#Affichage du tas
tastab.Print(tas)
#attendu:
PARENT : [602580185, 426232481, 3964731641, 138455936] LEFT CHILD : [1356453159, 3293574679, 2727937134, 3290195753] RIGHT CHILD : [1816088954, 2257925604, 3579840626, 1581852887]
 PARENT : [1356453159, 3293574679, 2727937134, 3290195753] LEFT CHILD : [1389535929, 832807153, 3526368565, 363335293] RIGHT CHILD : [2398824147, 3480035218, 487421168, 3961739784]
 PARENT : [1816088954, 2257925604, 3579840626, 1581852887] LEFT CHILD : [1834156317, 1832645408, 132768699, 622939054] RIGHT CHILD : [2269505948, 1979310886, 1859389282, 643936639]
 PARENT : [1389535929, 832807153, 3526368565, 363335293] LEFT CHILD : [3983630590, 954020349, 4102041063, 576193144] RIGHT CHILD : [3411245654, 3059252211, 3928924417, 1609824262]
```

```
PARENT : [2398824147, 3480035218, 487421168, 3961739784]
```

SupprMin

Pseudo-Code de SupprMin: Supprime l’élément à la racine du tas binaire et garantie les propriétés du tas.

```
SupprMin():
   si longueur(Tas) == 0:
      retourner

   min_val = Tas[0]        # On garde l'élément minimal
   switch(Tas[0], Tas[-1]) # On place le dernier élément en tête
   Tas.pop()               # On supprime le dernier élément
   descendre(Tas, 0)       # On vérifie que la structure est toujours un tas
depuis la racine

   retourner min_val
```

Complexité: La fonction supprime le minimum en échangeant la racine avec le dernier élément et en descendant la racine jusqu'à une position appropriée. La complexité temporelle est en O(log n), où n est la taille du tas. En effet, dans le pire cas, on parcourt tout le tableau.

How To:

```
#Importation des fichiers nécessaires
import lib.sdd_bigint.tasmin_tableau_bigint as tastab
import lib.utilitaire as ut

#Création de l’échantillon d’éléments
samples = ut.prog_list_constr(10, 10, 0)
#Création du tas sous forme de tableau
tas = []
#En utilisant AjoutsIteratifs
tas = tastab.AjoutsIteratifs(tas, samples[0])
#Suppression de l’élément minimal du tas
tastab.SupprMin(tas)
#Affichage du tas
tastab.Print(tas)
#attendu:
PARENT : [1356453159, 3293574679, 2727937134, 3290195753] LEFT CHILD :
[1389535929, 832807153, 3526368565, 363335293] RIGHT CHILD : [1816088954,
2257925604, 3579840626, 1581852887]
 PARENT : [1389535929, 832807153, 3526368565, 363335293] LEFT CHILD :
```

```
[3395898027, 1353677783, 1152707669, 2983853358] RIGHT CHILD : [2398824147,
3480035218, 487421168, 3961739784]
 PARENT : [1816088954, 2257925604, 3579840626, 1581852887] LEFT CHILD :
[1834156317, 1832645408, 132768699, 622939054] RIGHT CHILD : [2269505948,
1979310886, 1859389282, 643936639]
 PARENT : [3395898027, 1353677783, 1152707669, 2983853358] LEFT CHILD :
[3983630590, 954020349, 4102041063, 576193144] RIGHT CHILD : [3411245654,
3059252211, 3928924417, 1609824262]
```

Construction

Pseudo-Code de Construction: Construction réalise la construction d'un tas binaire à partir d'une liste d'éléments.

```
Construction(liste):
  Tas = []
  pour chaque élément x dans liste:
    Tas += [bi.Cle128Bits(x)]  # Conversion de la clé en Clé128Bits
  Pour i allant de longueur(Tas)//2 à 0 en décroissant:
      # On part de la base (n total/2^h, avec h=0 à la base)
      # pour remonter sur toute la hauteur du tas
      # Les derniers noeuds n'ayant pas de fils, on peut passer directement à leur racine
      # On équilibre puis on monte, ce qui revient à aller à l'élément précédent dans le tableau
      # Puis on réapplique la fonction de descente jusqu'à ce que le tas soit bien un tas min
     descendre(Tas, i - 1)
  retourner Tas
```

Complexité: La fonction Construction parcourt la moitié du tas (en partant de la base jusqu'à la racine) car la base n'a pas besoin d'appeler **descendre**() et le nombre de d'éléments à la base est nb_total_noeuds/2**(h (0 à la base) + 1) puis appelle descendre à chaque itération. Dans le pire cas, la complexité temporelle est en O(n), où n est la taille du tas.

Démonstration : **W** = [somme(nb_noeuds_internes_h * O(h)) avec nb_noeuds_internes_h, le nombre de noeuds à chaque hauteur, soit (nombre de noeuds total dans le tas divisé par (2**(h+1))]

En développant **W**, on se retrouve avec une suite de la forme : (nb_total_noeud_tas/2) * somme(h/2**h), lorsque h tend vers l'infini le deuxième terme tend vers 2. On a donc pour W = nb_total_noeud_tas -> **O(n)**

```
How To:

#Importation des fichiers nécessaires
import lib.sdd_bigint.tasmin_tableau_bigint as tastab
import lib.utilitaire as ut

# Création de l’échantillon d'éléments
samples = ut.prog_list_constr(10, 10, 0)
#Création du tas sous forme de tableau
tas = []
#Via la méthode Construction
tas = tastab.Construction(samples[0])
#Affichage du tas
tastab.Print(tas)
#attendu:
PARENT : [602580185, 426232481, 3964731641, 138455936] LEFT CHILD :
[1356453159, 3293574679, 2727937134, 3290195753] RIGHT CHILD : [1816088954,
2257925604, 3579840626, 1581852887]

PARENT : [1356453159, 3293574679, 2727937134, 3290195753] LEFT CHILD :
[2398824147, 3480035218, 487421168, 3961739784] RIGHT CHILD : [1389535929,
832807153, 3526368565, 363335293]

PARENT : [1816088954, 2257925604, 3579840626, 1581852887] LEFT CHILD :
[1834156317, 1832645408, 132768699, 622939054] RIGHT CHILD : [2269505948,
1979310886, 1859389282, 643936639]

PARENT : [2398824147, 3480035218, 487421168, 3961739784] LEFT CHILD :
[3983630590, 954020349, 4102041063, 576193144] RIGHT CHILD : [3411245654,
3059252211, 3928924417, 1609824262]

PARENT : [1389535929, 832807153, 3526368565, 363335293]
```

Union

```
Pseudo-Code de Union: La fonction Union combine deux tas binaires minimaux en
fusionnant leurs feuilles respectives dans un nouveau tas binaire minimal.
```

```
Union(Tas1, Tas2):
  # Cas où les deux tas sont vides
  si longueur(Tas1) == 0 et longueur(Tas2) == 0:
      retourner Aucun

  # Cas où le premier tas est vide
  si Tas1 == []:
      retourner Construction(Tas2)

  # Cas où le deuxième tas est vide
  si Tas2 == []:
      retourner Construction(Tas1)

  # Ajout des éléments du deuxième tas au premier tas
  Pour chaque élément i dans Tas2:
      Tas1 += [i]

  # Rééquilibrage du tas résultant de l'union
  Construction(Tas1)
  retourner Tas1
```

Complexité: La fonction Union ajoute les éléments du deuxième tas au premier tas, puis applique la fonction Construction pour rééquilibrer le tas résultant. La complexité temporelle est dominée par la construction et est en O(n), où n est la taille du tas résultant.

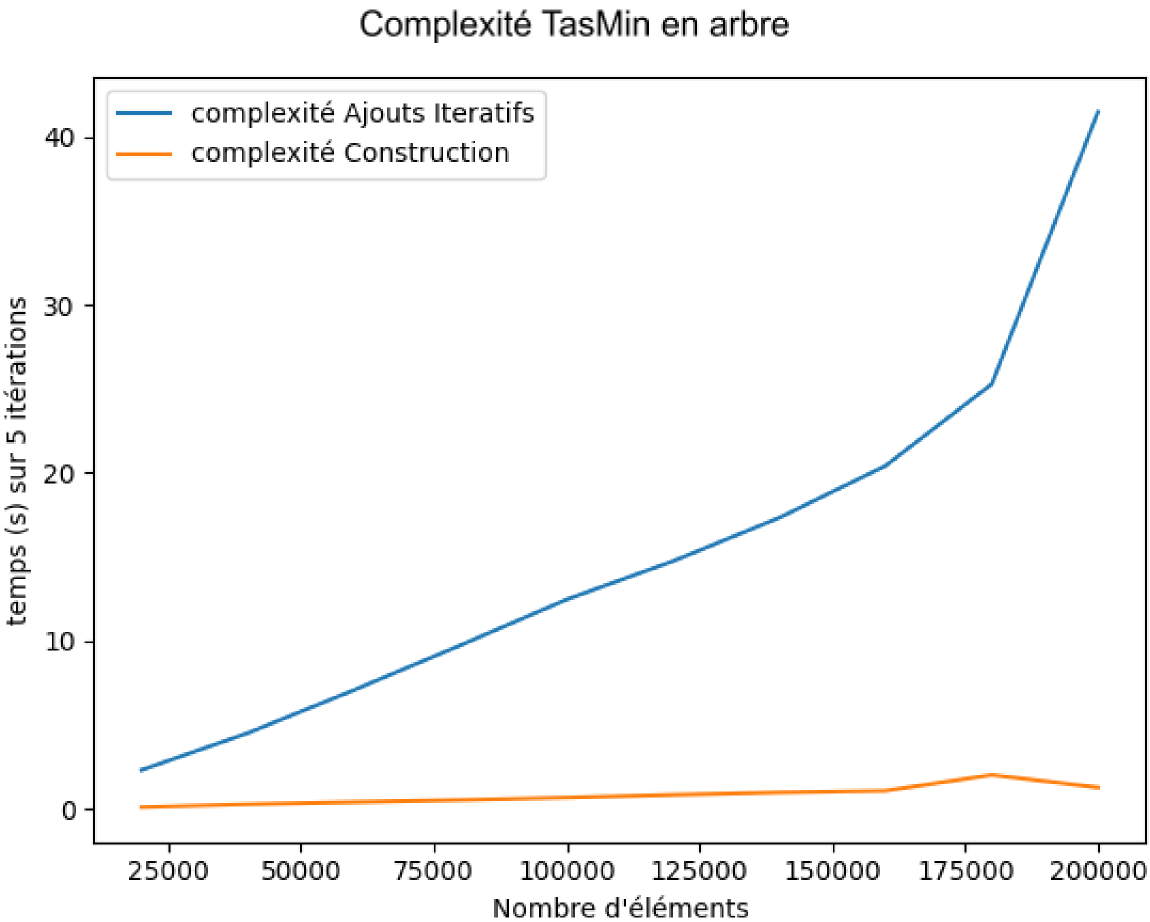
How To:

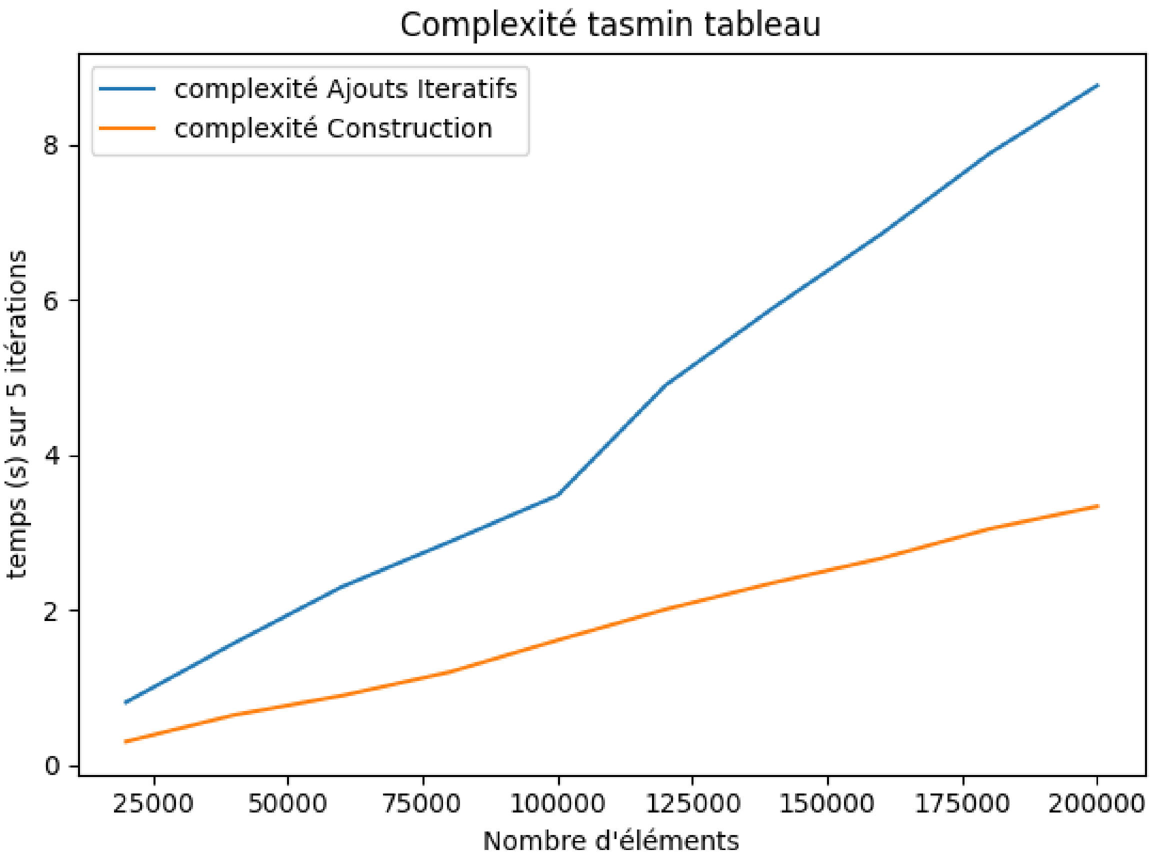
```
#Importation des fichiers nécessaires
import lib.sdd_bigint.tasmin_tableau_bigint as tastab
import lib.utilitaire as ut

#Création des échantillons d'éléments
samples1 = ut.prog_list_constr(5, 5, 0)
samples2 = ut.prog_list_constr(5, 10, 5)
#Création des tas
tas1 = []
tas2 = []
tas1 = tastab.Construction(samples1[0])
tas2 = tastab.Construction(samples2[0])
#Affichage de l'union des deux tas
tastab.Print(tastab.Union(tas1,tas2))
#attendu:
```

PARENT : [602580185, 426232481, 3964731641, 138455936] **LEFT CHILD** : [1389535929, 832807153, 3526368565, 363335293] **RIGHT CHILD** : [1834156317, 1832645408, 132768699, 622939054]
PARENT : [1389535929, 832807153, 3526368565, 363335293] **LEFT CHILD** :

```
[3411245654, 3059252211, 3928924417, 1609824262] RIGHT CHILD : [2398824147,
3480035218, 487421168, 3961739784]
 PARENT : [1834156317, 1832645408, 132768699, 622939054] LEFT CHILD :
[1356453159, 3293574679, 2727937134, 3290195753] RIGHT CHILD : [1816088954,
2257925604, 3579840626, 1581852887]
 PARENT : [3411245654, 3059252211, 3928924417, 1609824262] LEFT CHILD :
[3983630590, 954020349, 4102041063, 576193144] RIGHT CHILD : [2269505948,
1979310886, 1859389282, 643936639]
 PARENT : [2398824147, 3480035218, 487421168, 3961739784]
```

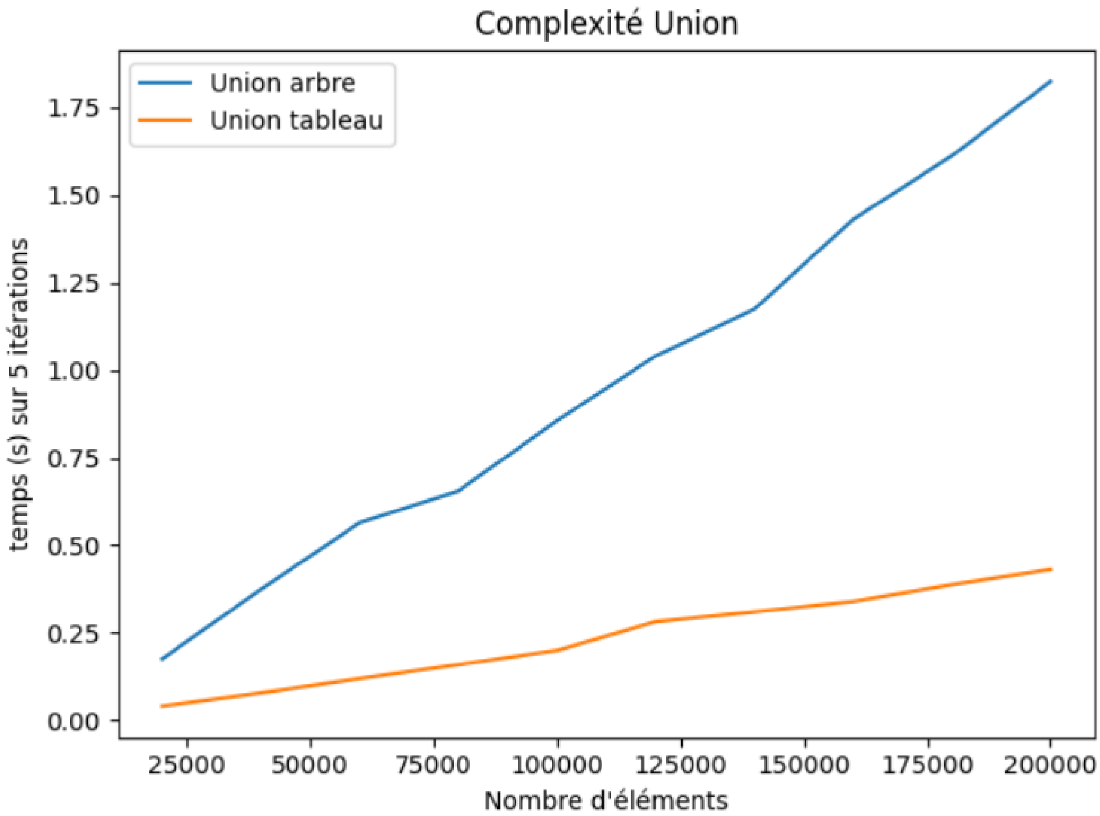
Observations graphiques





On peut observer que les temps d'exécutions sont bien inférieurs pour construction en O(n) comparé à l’ajout itératif en O(n*log(n)).

Pour le cas de la structure en tableau, on a dû forcer le pire cas en créant une liste de clé 128 bits ordonnées en ordre décroissant. Pour le tas en arbre, nous sommes restés sur la liste de clés fournis pour ce devoir.



Le temps d'exécution est moindre sur la structure en tableau, cela peut s'expliquer par le temps d'exécution liés à la récupération des éléments des deux tas en arbre.

Structure 2 : Files binomiales

Représentation

```
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
```

Primitives d'un Tournoi

Pseudo-Code de EstVide: Vérifie si le tournoi est vide.
EstVide():   retourner len(T) == 0
Complexité: **O(1)**

Pseudo-Code de Degre: Renvoie le degré de la racine du tournoi.
Degre():   retourner Degre
Complexité: **O(1)**

Pseudo-Code de Union2Tid: Renvoie l'union de deux tournois de même taille.

```
Union2Tid(T1, T2):
  si T1.Degre != T2.Degre:
        retourner "Les deux tournois ne sont pas de même taille"

        # On compare la racine des 2 tournois
        sinon si T1.Racine().inf(T2.Racine()):
            T += [T2] # On ajoute le tournoi T2 à la fin du tournoi T : T1 =
[racine, fils = T2]
            T1.Degre += 1
            retourner T1
        sinon:
            T2.T += [T1] # On ajoute le tournoi T à la fin du tournoi T2 : T2
= [racine, fils = T1]
            T2.Degre += 1
            retourner T2
```

Complexité: **O(1)**

Pseudo-Code de Racine: Renvoie la racine du tournoi.
Racine():   retourne T[0]
Complexité: **O(1)**

Pseudo-Code de Decapite: Renvoie la file binomiale obtenue en supprimant la

racine du tournoi.

```
Decapite(T1):
       si T1.EstVide() ou len(T1.T) == 1:
            retourner

        F = File()
        Pour i allant de len(T) - 1 à 0 avec un pas de -1: # File décroissante
mais liste des tournois croissante donc on inverse
            F.AjoutMin(T1.T[i])
        retourner F
```

Complexité: **O(log(n))** car l'opération appelle j = log(n) fois **AjoutMin** en O(1). En effet, un tournoi k de n éléments est composé de k tournois avec k = log(n). Exemple: T3 est composé de T0, T1 et T2 pour un total de 2**3 = 8 éléments.

Pseudo-Code de File: Renvoie une file binomiale réduite au tournoi T_k -> <T_k>.

```
File(T1):
       file = File()
       file.AjoutMin(T1)
       retourner file
```

Complexité: **O(1)**

Primitives d'une File binomiale

Pseudo-Code de EstVide: Renvoie vrai si la file est vide, sinon faux.

```
EstVide(F):
  retourner len(F.LT) == 0
```

Complexité: O(1)

Pseudo-Code de MinDegre: Renvoie le tournoi de degré minimal dans la file.

```
MinDegre(F):
     si F.MinDegreIndex != nil
```

```
        retourner F.LT[F.MinDegreIndex]
    sinon
        retourner nil
```

Complexité: O(1)

Pseudo-Code de Reste: Renvoie la file privée de son tournoi de degré minimal.

```
Reste(F):
    si non F.EstVide():
        tk = F.LT.pop(-1)
        F.taille -= 2**tk.Degre() # Réduit la taille de la file
    retourner self
```

Complexité: elle dépend de l’opération pop(), étant donné que l’on retire le dernier élément, l'opération est en **O(1)**.

Pseudo-Code de AjoutMin: Renvoie la file obtenue en ajoutant le tournoi comme tournoi de degré minimal de la file initiale.

```
AjoutMin(F, Tk):
    F.LT += [Tk]             # Ajout du Tournoi en queue de la File: O(1)
    F.MinDegreIndex = -1     # Maj index
    F.taille += 2**Tk.Degre() # Màj taille
    retourner self
```

Complexité : O(1)

Pseudo-Code de UFret: La file binomiale résultant de l'union de la file courante avec F2 et d'un tournoi en retenue.

```
UFret(F = self, F2, T):  # Fonction du cours
    si non T ou T.EstVide():  # pas de tournoi en retenue
           si F.EstVide():
               retourner F2
           si F2.EstVide():
               retourner F
           T1 = F.MinDeg()
           T2 = F2.MinDeg()

           si T1 et T2 et T1.Degre() < T2.Degre():
```

```
                # Cas où le degré du tournoi de la file courante est inférieur
à celui de F2
                retourner F.Reste().UnionFile(F2).AjoutMin(T1)
            si T1 et T2 et T2.Degre() < T1.Degre():
                # Cas où le degré du tournoi de F2 est inférieur à celui de la
file courante
                retourner F.UnionFile(F2.Reste()).AjoutMin(T2)
            si T1 et T2 et T1.Degre() == T2.Degre():
                # Cas où les deux tournois ont le même degré
                retourner self.Reste().UFret(F2.Reste(), T1.Union2Tid(T2))
     sinon:  # T tournoi en retenue
            si F.EstVide():
                # Cas où la file courante est vide
                retourner T.File().UnionFile(F2)
            si F2.EstVide():
                # Cas où F2 est vide
                retourner self.UnionFile(T.File())

            T1 = self.MinDeg()
            T2 = F2.MinDeg()
            si T1 et T2 et T.Degre() < T1.Degre() et T.Degre() < T2.Degre():
                # Cas où le degré du tournoi de retenue est inférieur à celui
de T1 et T2
                retourner F.UnionFile(F2).AjoutMin(T)
            si T1 et T2 et T.Degre() == T1.Degre() et T.Degre() == T2.Degre():
                # Cas où le degré de retenue est égal à celui de T1 et T2
                retourner F.Reste().UFret(F2.Reste(),
T1.Union2Tid(T2)).AjoutMin(T)
            si T1 et T2 et T.Degre() == T1.Degre() et T.Degre() < T2.Degre():
                # Cas où le degré de retenue est égal à celui de T1 et
inférieur à celui de T2
                retourner F.Reste().UFret(F2, T1.Union2Tid(T))
            si T1 et T2 et T.Degre() == T2.Degre() et T.Degre() < T1.Degre():
                # Cas où le degré de retenue est égal à celui de T2 et
inférieur à celui de T1
                retourner F.UFret(F2.Reste(), T2.Union2Tid(T))
```

Complexité : **O(log(n+m))** car on parcourt les 2 files complètement dans le pire des cas et chaque file a log(n) tournois avec n le nombre d’éléments dans ses tournois. De plus, toutes les primitives appelées sont en O(1) : **Reste**(), **Union2Tid**(), **AjoutMin**(), **MinDeg**() et **EstVide**(). **UnionFile**() revient à appeler **UFret**.

Pseudo-Code de UnionFile: Renvoie la file binomiale union des deux files F1 et F2.

```
UnionFile(F = self, F2):
     retourner F1.UFret(F2, None)
```

Complexité : dépend de la complexité de **UFret**

How To:

```
#Importation des fichiers nécessaires
import lib.sdd_bigint.file_bigint as filebi
import lib.utilitaire as ut

#Création des échantillons d'éléments
samples1 = ut.prog_list_constr(5,5,0)
samples2 = ut.prog_list_constr(5,10,5)

#Création des files
file1 = filebi.File()
file2 = filebi.File()
#Avec la méthode Construction
file1.Construction(samples1[0])
file2.Construction(samples2[0])
#Affichage de l'union des files
file1.UnionFile(file2).afficheFile()
#attendu:
File10 =  (3, racine : [602580185, 426232481, 3964731641, 138455936]) (1,
racine : [1389535929, 832807153, 3526368565, 363335293])
```

Pseudo-Code de Ajout: Ajoute une clé à la file binomiale.

```
Ajout(F = self, clé):
     t = Tournoi(clé)  # Conversion en bigint et création du tournoi
     res = F.UnionFile(t.File())  # Union de la file actuelle avec la file
formée d'un seul tournoi
     retourner res
```

Complexité : dépend de la complexité de **UnionFile** dans le meilleur cas, car on ajoute un **seul** tournoi de degré minimum et donc **seul** un **AjoutMin** dans **UFret** est appelé et retourné, soit **O(1)**.

How To:

```
#Importation des fichiers nécessaires
import lib.sdd_bigint.file_bigint as filebi

#Création de la file
file = filebi.File()
```

```
#Ajout d'une clé dans la file
file.Ajout("0xdf6943ba6d51464f6b02157933bdd9ad")
#Affichage de la file
file.afficheFile()
#attendu:
File1 =  (0, racine : [3748217786, 1834042959, 1795298681, 868080045])
```

Pseudo-Code de Construction: Renvoie la file binomiale construite à partir de la liste.

`Construction(F = self, liste):`
`     Pour x dans liste:`
`            F.Ajout(x)`

Complexité : dépend de la complexité de **Ajout,** on a N fois la complexité de **Ajout**. soit O(N*1) -> **O(N)**
How To:

```
#Importation des fichiers nécessaires
import lib.sdd_bigint.file_bigint as filebi
import lib.utilitaire as ut

#Création de l'échantillon d'éléments
samples = ut.prog_list_constr(10,10,0)

#Création de la file
file = filebi.File()
#Via la méthode Construction
file.Construction(samples[0])
#Affichage de la file
file.afficheFile()
#attendu
File10 =  (3, racine : [602580185, 426232481, 3964731641, 138455936]) (1,
racine : [1356453159, 3293574679, 2727937134, 3290195753])
```

Pseudo-Code de SupprMin: Renvoie la file binomiale en supprimant la racine minimale de la file.

`SupprMin(F = self):`
`        si F.EstVide():`
`            retourner File()`
`        sinon:`

```
            # Recherche du tournoi avec la racine minimale
            index = -1
            min = Cle128Bits(0xFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF) # Plus grande
clé
            Pour i allant de (len(self.LT)) à 0:  # O(log(n)) car il y'a
log(n) éléments dans la liste
                tk = F.LT[i]
                si tk.Racine() < min:
                    min = tk.Racine()
                    index = i

            # Suppression du tournoi de degré minimal de la liste
            tksup = F.LT.pop(index)  # O(log(n)) car il y'a log(n) éléments
dans la liste

            # Réduction de la taille de la file
            F.taille -= (2**tksup.Degre())

            # Obtention de la file résultant de la décapitation du tournoi de
degré minimal
            res = tksup.Decapite()  # O(log(n))

            # Union de la file actuelle avec la file résultante
            res = F.UnionFile(res)
            retourner F
```

Complexité : On réalise plusieurs opérations à la suite avec une complexité supérieure à 1. Un **boucle** en **O(log(n))** sur la File, la suppression d'un Tournoi de la File en **O(log(n))** puis on appelle **Decapite** en **O(log(n))** et enfin **UnionFile** en **O(log(n))** car les 2 Files on à 2, N-1 éléments.

On a donc une fonction en O(4*log(n)) -> **O(log(n))**

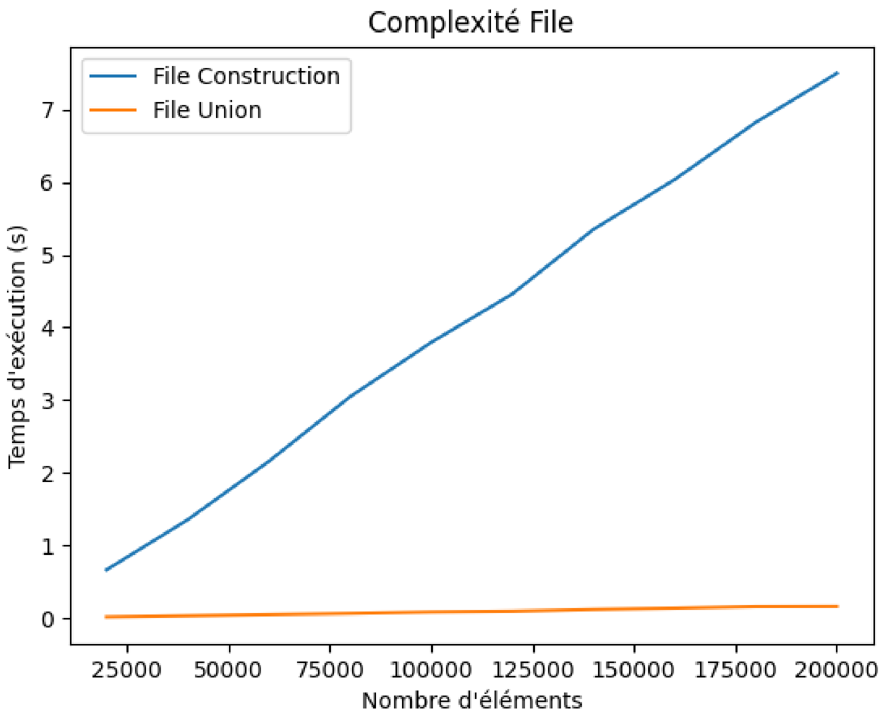
How To:

```
#Importation des fichiers nécessaires
import lib.sdd_bigint.file_bigint as filebi
import lib.utilitaire as ut

#Création de l'échantillon d'éléments
samples = ut.prog_list_constr(10,10,0)

#Création de la file
file = filebi.File()
#Avec la méthode Construction
file.Construction(samples[0])
#Suppression du tournoi minimal
```

```
file.SupprMin()
#Affichage de la file
file.afficheFile()
#attendu:
File9 =  (3, racine : [1356453159, 3293574679, 2727937134, 3290195753]) (0,
racine : [3411245654, 3059252211, 3928924417, 1609824262])
```

Observations graphiques

Les graphiques ont été produits à partir des clés fournies pour ce devoir.



La construction est bien linéaire.

La fonction Union est bine O(log(n)) et est bien plus rapide que l'union sur tas en tableau qui est en O(n+m) et plus rapide que la construction de la file en O(n).

Fonction de hachage

MD5: Permet d'obtenir l'empreinte numérique d'un fichier
Voir fichier lib/hachage/MD5.py.

```
How To:

mot = "which"
# MD5 attendu : 8b7af514f25f1f9456dcd10d2337f753
print(md5.md5_to_hex(md5.md5(mot.encode('utf-8'))))
```

Arbre de Recherche

Représentation

Pour assurer, la recherche en **O(log(n))**, on a donc dû implémenter une forme d'ABR, l'AVL. L'AVL nécessite un auto-équilibrage à chaque opération sur l'arbre.

```
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
```

Fonctions principales

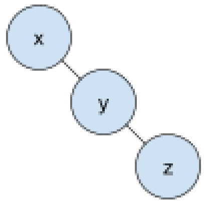
Pour assurer une recherche dans l'ABR en **O(log(n))**, on s'assure que celui-ci soit équilibré.

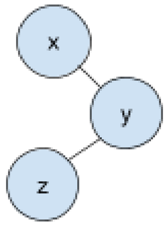
```
Pseudo-Code de Balance: Calcul de la balance du nœud, définie comme la
différence entre les hauteurs du sous-arbre droit et du sous-arbre gauche.

Balance(A = self):
     retourner (A.droite.hauteur + 1 si A.droite != nil sinon 0) -
(A.gauche.hauteur + 1 si A.gauche != nil sinon 0)

Complexité : O(1)
```

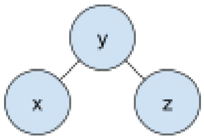
```
Pseudo-Code de Rotation: Effectue les rotations nécessaires pour équilibrer
l'arbre après une opération d'insertion et assure également que le nœud avec
```

la clé maximale est à droite et le nœud avec la clé minimale est à gauche.

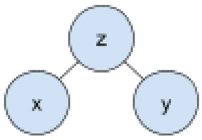
```
Rotation(A = self):
    facteur_equilibre = A.Balance()

    # Cas de l'arbre déséquilibré à droite
    si facteur_equilibre > 1:
        # Cas de la rotation simple à gauche
        si A.droite.balance() >= 0:
            A.RG()
```



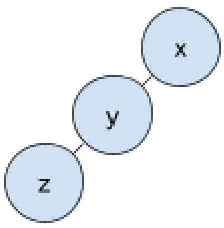


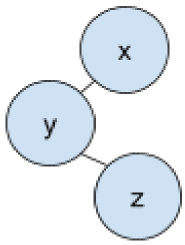



```
       # Cas de la rotation droite-gauche
       sinon:
            A.droite.RD()
            A.RG()
```

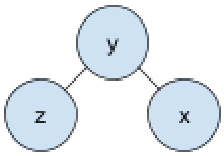


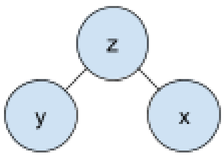

```
    # Cas de l'arbre déséquilibré à gauche
    si facteur_equilibre < -1:
```






```
    # Cas de la rotation simple à droite
        si A.gauche.balance() <= 0:
            A.RD()
        # Cas de la rotation gauche-droite
        sinon:
            A.gauche.RG()
            A.RD()
```






```
    # Assure que le nœud avec la clé maximale est à droite
    si A.droite et A.cle.sup(A.droite.cle):
        Switch(A.cle, A.droite.cle)

    # Assure que le nœud avec la clé minimale est à gauche
    si A.gauche et (A.gauche.cle.sup(A.cle) ou A.gauche.cle.eq(A.cle)):
        Switch(A.cle, A.gauche.cle)
```

Complexité : **O(1)**

How To:

```
#Importation des fichiers nécessaires
import lib.sdd_integer.abr as abr
import lib.utilitaire as ut

#Création de l’échantillon d’éléments
random_int_list1 = random.sample(range(1, 25), 5)
random_int_list2 = random.sample(range(1, 25), 10)
random_int_list3 = random.sample(range(1, 25), 20)
random_int_list4 = random.sample(range(1, 25), 10)
#Échantillon d'éléments croissant puis décroissant
random_int_list4.sort()
random_int_list5 = random.sample(range(1, 25), 10)
random_int_list5.sort(reverse=True)
#Création du premier noeud de l’arbre
arbre1 = abr.ABR(random_int_list1[0])
arbre2 = abr.ABR(random_int_list2[0])
arbre3 = abr.ABR(random_int_list3[0])
arbre4 = abr.ABR(random_int_list4[0])
arbre5 = abr.ABR(random_int_list5[0])
#Avec la méthode Construction
arbre1.construction(random_int_list1[1:])
arbre2.construction(random_int_list2[1:])
arbre3.construction(random_int_list3[1:])
arbre4.construction(random_int_list4[1:])
arbre5.construction(random_int_list5[1:])
#Affichage des arbres
abr.PrintTree(arbre1)
abr.PrintTree(arbre2)
abr.PrintTree(arbre3)
abr.PrintTree(arbre4)
abr.PrintTree(arbre5)
#attendu
      18
   /¯¯¯ ¯¯¯\
  11      20
 /¯ ¯\
 6  14


         9
    /¯¯¯¯¯¯   ¯¯¯¯¯¯\
    4              23
   /¯¯¯ ¯¯¯\       /¯¯¯ ¯¯¯\
   1       8      19      24
```

```
         /‾      /‾ ‾\
         5      14  20

                               9
               /------------       ------------\
               4                              17
       /------   ------\               /------   ------\
       2               6              13               21
   /--- ---\       /--- ---\       /--- ---\       /--- ---\
   1       3       5       8      12      16      19      22
                                 /‾      /‾      /‾ ‾\       ‾\
                                11      15      18  20      24

              12
       /------   ------\
       7              18
   /--- ---\       /--- ---\
   2      10      14      22
                 /‾ ‾\       ‾\
                13  17      24

              18
       /------   ------\
       9              22
   /--- ---\       /--- ---\
   7      16      19      23
 /‾      /‾ ‾\
 5      14  17
```

Résultats des arbres si il n’y avait pas eu de rotation effectué lors des insertions

Pseudo-Code de RD: Effectue une rotation à droite sur le nœud actuel dans un arbre AVL.

```
RD(A = self):

    # Effectue les échanges
    switch(A.cle, A.gauche.cle)  # switch(p, q)
    switch(A.gauche.droite, A.droite)  # switch(V, W)
    switch(A.gauche.gauche, A.droite)  # switch(U, V)
    switch(A.gauche, A.droite)  # switch(ABR(q), U)
```

```
    # Met à jour la hauteur des sous-arbres
```

Complexité : **O(1)** dans le cas normal
Note : Nous avons eu du mal à mettre à jour notre attribut hauteur dans le cas d'une rotation donc nous sommes passer par une fonction en O(log(n)) afin de la recalculer. Cependant, aucune contrainte n'a été indiquée pour l'insertion dans l'arbre, donc j'imagine que ce n'est pas pénalisant.

Pseudo-Code de RG: Effectue une rotation à gauche sur le nœud actuel dans un arbre AVL.

```
RG(A = self):

    # Effectue les échanges
    switch(A.cle, A.droite.cle)  # switch(p, q)
    switch(A.gauche, A.droite.droite)  # switch(V, W)
    switch(A.droite.gauche, A.droite.droite)  # switch(U, V)
    switch(A.gauche, A.droite)  # switch(ABR(q), U)

    # Met à jour la hauteur des sous-arbres
```

Complexité : **O(1)** dans le cas normal
Note : Nous avons eu du mal à mettre à jour notre attribut hauteur dans le cas d'une rotation donc nous sommes passer par une fonction en O(log(n)) afin de la recalculer. Cependant, aucune contrainte n'a été indiquée pour l'insertion dans l'arbre, donc j'imagine que ce n'est pas pénalisant.

Pseudo-Code de **Recherche**: Recherche une clé dans l'arbre AVL.

```
Recherche(A = self, clé): #clé : instance de classe Cle128Bits
    # Cas où la clé du nœud actuel est égale à la clé recherchée
    si A.cle.eq(clé):
        retourner True
    # Cas où la clé recherchée est inférieure à la clé du nœud actuel
    sinon si cle.inf(A.cle):
        # Si le sous-arbre gauche est vide, la clé n'est pas présente
        si A.gauche est None:
            retourner False
        sinon:
            # Sinon, récursion pour rechercher la clé dans le sous-arbre
gauche
            retourner A.gauche.recherche(cle)
    sinon:
```

```
        # Cas où la clé recherchée est supérieure à la clé du nœud actuel
        # Si le sous-arbre droit est vide, la clé n'est pas présente
        si A.droite est None:
            retourner False
        sinon:
            # Sinon, récursion pour rechercher la clé dans le sous-arbre droit
            retourner A.droite.recherche(cle)
```

Complexité : **O(log(n))**, l’arbre étant équilibré.

How To:

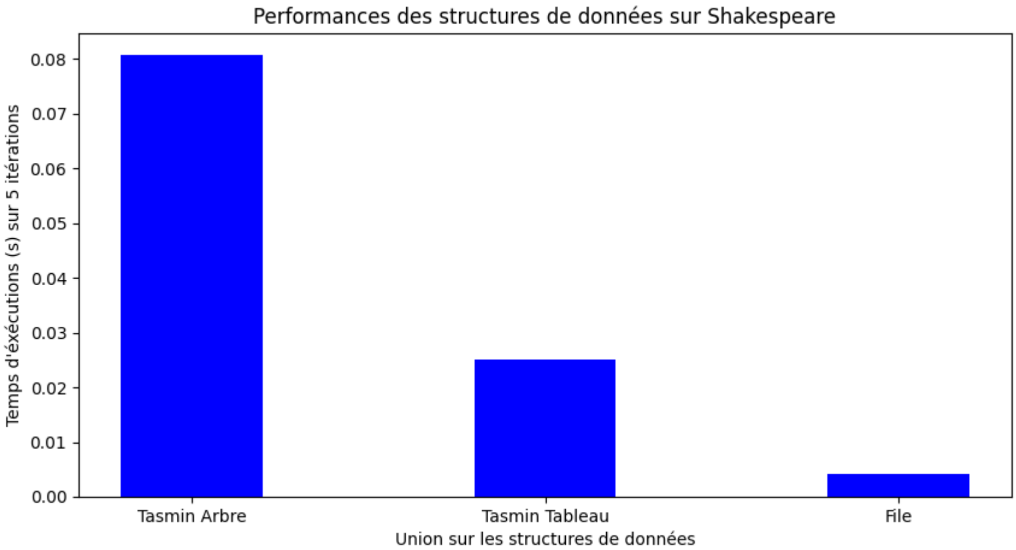
```
#Importation des fichiers nécessaires
import lib.sdd_bigint.abr_bigint as abr
import lib.utilitaire as ut

#Création de l’échantillon d’éléments
samples = ut.prog_list_constr(10,10,0)
#Création du premier noeud de l’arbre
arbre = abr.ABR(samples[0][0])
#Avec la méthode Construction avec le reste de la liste
arbre.construction(samples[0][1:])
#Affichage du résultat de la recherche
print(arbre.recherche(samples[0][2]))
#attendu:
True
```

Étude expérimentale

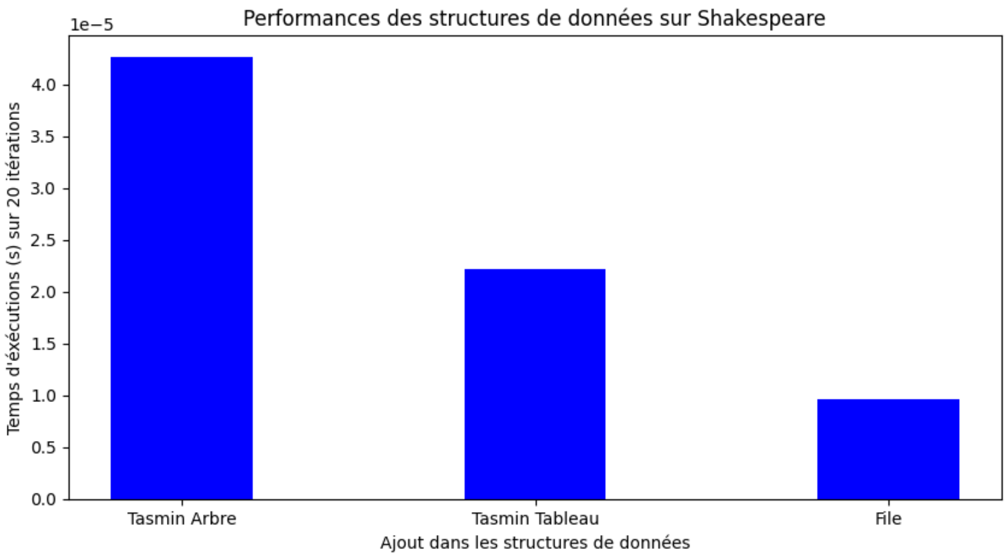
Pseudo-Code de Collision: Renvoie l'ensemble des mots qui entre en collision pour MD5

How To:

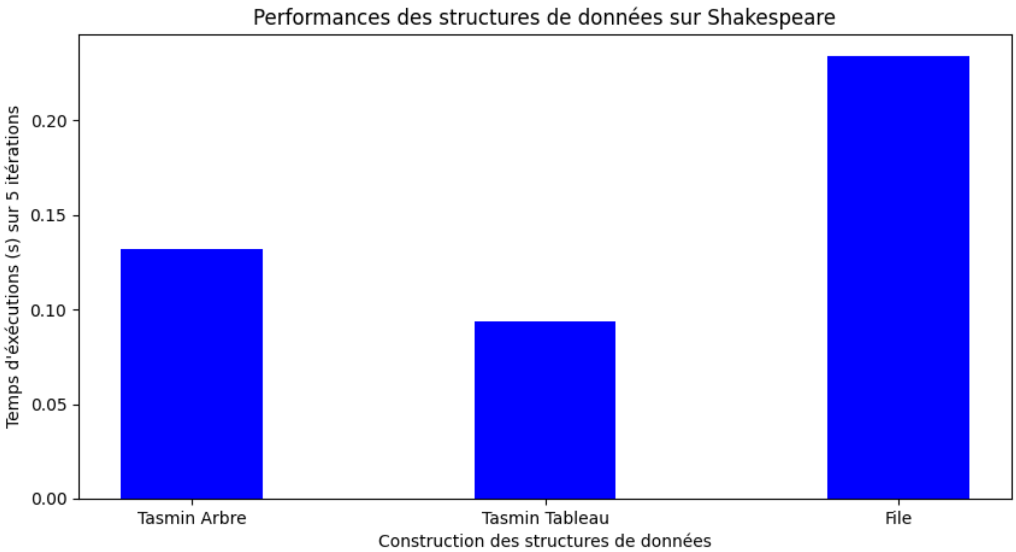
```
import test.test_experimentation.test_Shakespeare as shake

liste = [‘out’, ‘hello’, ‘home’, ‘out’]

# attendu : {'c68271a63ddbc431c307beb7d2918275': ['out', 'out']}
print(shake.collisions(liste))
```



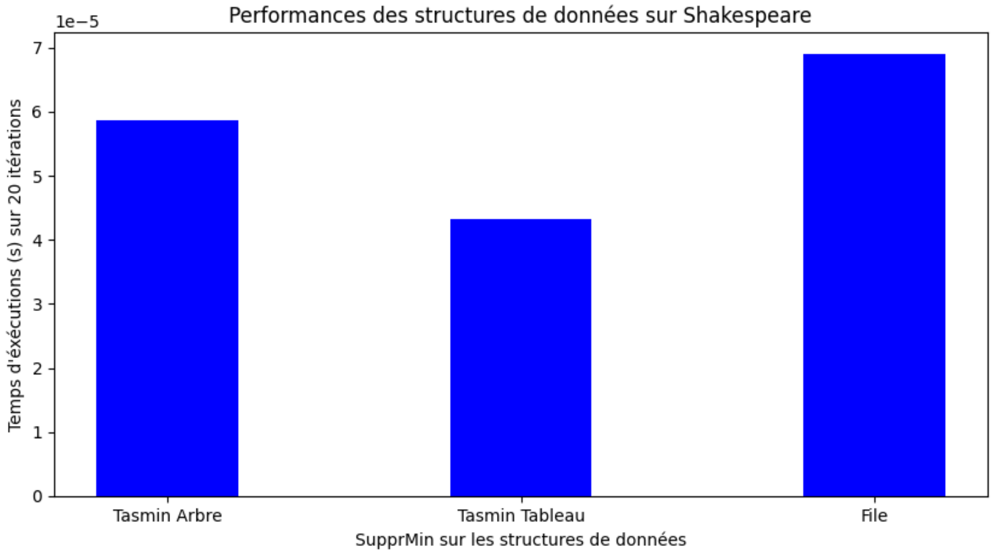
On observe que sur les données Shakespeare, le temps d'exécution est bien plus faible sur la file que sur le tas. En effet, ce nous montre bien la différence entre une fonction en O(n+m) et une fonction en O(log(n+m)).



L'ajout de la file est comme prévu plus rapide que le tas. En effet, cette dernière est en O(log(n+m)). L'ajout dans le tas en tableau est comme prévu plus rapide que l'ajout dans l'arbre car celui-ci profite de l'ajout en queue du tableau en O(1) puis les deux tas procèdent à des remontées.



La construction des trois structures est en O(n). Le tas a de meilleures performance que la file dans notre cas.



Les trois structures ont des temps d'exécutions quasi semblables pour Supprmin, cependant, la file est quand plus lente du fait des opérations de décapitation de tournoi ainsi que la recherche de la racine minimale qui ne peut être déduite que par l'itération sur toute la file.

Annexe

Fonctions annexes du Tas en arbre

Pseudo-Code de **get_hauteur**: Renvoie la hauteur du nœud basée sur la taille.
get_hauteur(Tas = self):   retourner int(math.log2(Tas.taille))
Complexité : **O(1)**

Pseudo-Code de **majNoeud**: Met à jour le nœud en comparant ses clés avec celles de ses enfants. La fonction effectue des échanges si nécessaire, met à jour la taille du nœud en fonction de ses enfants, et ajuste les parents des enfants.

```
majNoeud(Noeud = self):
     # Vérifie si le nœud a un enfant gauche
     si Noeud.gauche:
        # Compare les clés du nœud et de l'enfant gauche, et échange si
nécessaire
        si Noeud.gauche.cle.inf(Noeud.cle):
            Switch(Noeud.cle, Noeud.gauche.cle)

        # Met à jour la taille du nœud en fonction de l'enfant gauche
        Noeud.taille = 1 + Noeud.gauche.taille
        # Ajuste le parent de l'enfant gauche
        Noeud.gauche.parent = Noeud

    # Vérifie si le nœud a un enfant droit
    si Noeud.droite:
        # Compare les clés du nœud et de l'enfant droit, et échange si
```

```
nécessaire
        si Noeud.droite.cle.inf(Noeud.cle):
            Switch(Noeud.cle, Noeud.droite.cle)

        # Met à jour la taille du nœud en fonction de l'enfant droit
        Noeud.taille += Noeud.droite.taille
        # Ajuste le parent de l'enfant droit
        Noeud.droite.parent = Noeud
```

Complexité : **O(1)**

Pseudo-Code de **last_one**: Identifie et ajoute le dernier nœud d'une branche du tas binaire à la liste last_node. La fonction explore récursivement l'arbre en suivant des règles spécifiques pour déterminer le dernier nœud.

```
last_one(Tas = self, node):
     si non node:
        retourner

    # Cas où le nœud a un enfant gauche mais pas de fils droit
    si node.gauche et non node.droite:
        Tas.last_node += [node.gauche]
        retourner

    # Cas où le nœud n'a ni fils gauche ni fils droit
    sinon si non node.droite et non node.gauche:
        Tas.last_node += [node]
        retourner

    # Cas particulier d'un tas avec 3 éléments
    sinon si node.droite et node.gauche et node.droite.taille == 1 et
node.gauche.taille == 1:
        Tas.last_node += [node.droite]
        retourner

    sinon:
        # Si les fils ont la même taille, on continue à explorer à droite
        si node.gauche et node.droite et node.gauche.taille ==
node.droite.taille:
            Tas.last_one += [node.droite]

        # Si la branche gauche est pleine et la droite est vide, on explore la
gauche forcément
        sinon si node.gauche.taille == (2**(node.get_hauteur()) - 1) et
node.droite.taille == (2**(node.get_hauteur() - 1) - 1):
```

```
            Tas.last_one(node.gauche)

        # Si la branche gauche est pleine et la droite n'est pas vide, on explore la droite
        sinon si node.gauche.taille == (2**(node.get_hauteur()) - 1) et node.droite.taille > (2**(node.get_hauteur() - 1) - 1):
            Tas.last_one(node.droite)

        # Si la branche droite est vide et la gauche n'est pas encore pleine, on explore la gauche
        sinon si node.gauche.taille < (2**(node.get_hauteur()) - 1) et node.droite.taille == (2**(node.get_hauteur() - 1) - 1):
            Tas.last_one(node.gauche)
```

Complexité : **O(log(n))**, simple parcours en profondeur dans un tas.

Pseudo-Code de **descendre**: La fonction descendre fait descendre un nœud dans le tas en échangeant avec son plus petit enfant jusqu'à ce que la propriété du tas soit rétablie.

```
descendre(Tas = self, node):
     # Boucle pour faire descendre le nœud dans le tas
     tant que Vrai:
        si non noeud:
            return

        gauche = noeud.gauche
        droite = noeud.droite
        plus_petit = noeud

        # Comparaison avec le plus petit enfant
        si gauche et gauche.cle.inf(plus_petit.cle):
            plus_petit = gauche

        si droite et droite.cle.inf(plus_petit.cle):
            plus_petit = droite

        # Échange si nécessaire avec le plus petit enfant
        si plus_petit n'est pas noeud:
            Switch(noeud.cle, plus_petit.cle)
            noeud = plus_petit
        sinon:
            return
```

Complexité : **O(log(n)),** simple parcours vers la racine. Le chemin ayant au maximum log(n) éléments, avec n la taille du tas.

Pseudo-Code de **calculNoeud**: Prend en entrée la taille d'une liste d'éléments et retourne une liste d'informations sur la construction d'un nœud du tas.

```
calculNoeud(Tas = self, taille):
    # Vérification si la taille est nulle
    si taille == 0:
        retourner
    # Calcul de la hauteur de l'arbre
    h = entier(log2(taille))

    # Cas où la taille est une puissance de 2 moins 1
    si taille == (2**(h + 1) - 1):
        taille -= 1
        retourner [1, taille // 2, taille - taille // 2]
    sinon:
        # Calcul du reste après avoir construit un tas binaire complet
        reste = ((taille + 1) - ((2**h) - 1)) - 1
        liste_g = (taille - reste) // 2
        liste_d = (taille - reste) // 2

        # Répartition du reste entre les sous-arbres gauche et droit
        si reste <= ((2**h) // 2):
            liste_g += reste
        sinon:
            liste_g += ((2**h) // 2)
            liste_d += reste - ((2**h) // 2)

        retourner [1, liste_g, liste_d]
```

Complexité : **O(1)**

Fonctions annexes du Tas en tableau

Pseudo-Code de **monter**: La fonction compare le nœud à son parent et les échange si nécessaire jusqu’à la racine pour maintenir la propriété du tas.

```
monter(Tas, pos):
    # position du pere
    pere = (pos - 1) // 2
    si Tas[pos].inf(Tas[pere]):
        # switch
        Switch(Tas[pere], Tas[pos])

        si pere > 0:  # On continue de monter si le pere n'est pas à la racine
du tas (pos = 0)
            monter(Tas, pere)
```

Complexité : **O(log(n)),** simple remontée vers la racine dans le pire cas.

Pseudo-Code de **descendre**: La fonction compare le nœud avec ses fils gauche et droit (s'il en a) et échange avec le plus petit des fils si nécessaire.

```
descendre(Tas, pos):
    pere = pos
    fils_gauche = 2 * pos + 1
    fils_droit = 2 * (pos + 1)

    # Recherche des fils gauche et droit du nœud
    noeud = []
    si fils_droit < longueur(Tas):
        noeud = [Tas[pere], Tas[fils_gauche], Tas[fils_droit]]
    sinon si fils_gauche >= longueur(Tas):
        retour
    sinon:
        noeud = [Tas[pere], Tas[fils_gauche], Tas[fils_droit]]

    min = 0
    # Trouver l'index du plus petit élément parmi le nœud et ses fils
    si longueur(noeud) == 2:
        min = 0 si noeud[0].inf(noeud[1]) sinon 1
    sinon:
        si noeud[0].inf(noeud[1]) et noeud[0].inf(noeud[2]):
            min = 0
        sinon si noeud[1].inf(noeud[0]) et noeud[1].inf(noeud[2]):
            min = 1
        sinon si noeud[2].inf(noeud[0]) et noeud[2].inf(noeud[1]):
            min = 2

    si min != 0:  # Teste si le parent est bien l'élément le plus petit, si
différent de 0 alors
        # on doit échanger le père avec son plus petit fils
```

```
        index_min = 2 * pos + min
        Tas[pere], Tas[index_min] = Tas[index_min], Tas[pere]

        si (2 * index_min + 1) < longueur(Tas):
        # si le fils gauche est inférieur à la taille du tableau, on peut
continuer
            descendre(Tas, index_min)
```

Complexité : **O(log(n)),** simple descente vers le dernier fils dans le pire cas, donc au pire un chemin de taille = log(n).

Pseudo-Code de **Inserer**: Insére une clé dans l'arbre AVL et le rééquilibre.

```
Inserer(A = self):

    # Cas où le nœud actuel est vide
    si A.cle est None:
        A.cle = bi.Cle128Bits(valeur)
    # Cas où la valeur à insérer est inférieure à la valeur du nœud actuel ou
égale à la valeur du nœud actuel
    sinon si bi.Cle128Bits(valeur).inf(A.cle) ou
bi.Cle128Bits(valeur).eq(A.cle):
       # Gestion des collisions si les valeurs sont égales
        si bi.Cle128Bits(valeur).eq(A.cle) et valeur dans ABR.liste_collision:
            ABR.liste_collision[valeur] += 1
        sinon:
            ABR.liste_collision[valeur] = 1

        # Si le sous-arbre gauche est vide, créé un nouveau nœud gauche avec
la valeur
        si A.gauche est None:
            A.gauche = ABR(valeur)
        sinon:
            # Sinon, récursion pour insérer la valeur dans le sous-arbre
gauche
            A.gauche.insertion(valeur)
    sinon:
        # Cas où la valeur à insérer est supérieure
        # Si le sous-arbre droit est vide, créé un nouveau nœud droit avec la
valeur
            si A.droite est None:
                A.droite = ABR(valeur)
            sinon:
                # Sinon, récursion pour insérer la valeur dans le sous-arbre
droit
```

```
                A.droite.insertion(valeur)

    # Met à jour la hauteur du nœud
    A.hauteur = A.hauteurabr()

    # Applique les rotations pour maintenir l'équilibre de l'arbre AVL
    A.rotation()

    retourner
```

```
How To:

#Importation des fichiers nécessaires
import lib.sdd_bigint.abr_bigint as abr
import lib.utilitaire as ut

#Création de l’échantillon d’éléments
samples = ut.prog_list_constr(10,10,0)
#Création du premier noeud de l’arbre
arbre = abr.ABR(samples[0][0])
#Avec la méthode Insertion
arbre.insertion(samples[0][1])
arbre.insertion(samples[0][2])
arbre.insertion(samples[0][3])
arbre.insertion(samples[0][4])
arbre.insertion(samples[0][5])
abr.PrintTree(arbre)

#attendu:
      18
   /¯¯¯ ¯¯¯\
  13      23
 /¯ ¯\      ¯\
60  18      34

#Les clés sont sous affichage simplifié et n’ont que les deux premiers
chiffres de la première partie comparés à leur affichage complet pour plus de
lisibilité
```