SORBONNE UNIVERSITÉ

4I500

Algorithmique Avancée

Rapport de Projet

Étudiants: Numéros:
Yannick LI 3305496
Olivier Latiri 3520328

Décembre 2018

Algorithmique Avancée
Devoir de programmation

Notre devoir de programmation est réalisé en C++.

1. Échauffement

Dans ce devoir on a dû utiliser des clés de 128 bits. Pour les stocker, nous avons utilisé une classe Clef (fichier Clef.h et Clef.cpp) dans laquelle on représente nos clés à l'aide de 4 entiers non signés de 32 bits (4 unsigned int : i1, i2, i3, i4 avec i1 étant l'entier de poids fort).

Question 1.1

Pour pouvoir déterminer si une cle1 est strictement inférieure à une cle2, on va comparer en premier l'entier de poids le plus fort (i.e. i1) s'il y a égalité on compare i2 ainsi de suite.
Dans notre code, cette méthode a pour signature : bool operator<(Clef a)
On compare l'appelant avec le paramètre a.

Question 1.2

Pour pouvoir déterminer si une cle1 est égales à une cle2, on va renvoyer true si et seulement si les 4 entiers représentant chacunes des deux clés sont égales.
Dans notre code, cette méthode a pour signature : bool operator==(Clef a)
On compare l'appelant avec le paramètre a.

2. Structure 1 : Tas priorité min

Question 2.3/2.4

1. Implémentation du tas avec la structure de tableau (fichier TasMinTab.h)

1.1. Implémentation de SupprMin (l.37-42)

Cette fonction sert à supprimer le plus petit élément du tas. Elle est en O(log(n)).
Comme on est dans un tas le plus petit élément du tas est le premier élément du tableau, donc on va le remplacer par le dernier élément du tableau (copie) et on va enlever le dernier élément. Ces opérations sont en O(1). Ensuite on va appeler la fonction redescendre sur le nouveau premier élément du tableau pour le placer au bon endroit du tableau. Redescendre compare les valeurs des clefs du père et de ses fils, et échange la position du père avec celui du fils le plus petit. Redescendre est ensuite appelé de manière récursive sur le nouveau fils jusqu'à ce que le noeud

sur lequel on utilise Redescendre n'ait pas de fils plus petit que lui même. Cette descente de fils en fils, on dans le pire des cas en O(log(n)) si on compare tous les fils du branche.

1.2. Implémentation de Ajout (l.68-72)

La fonction ajout sert à ajouter une clef dans notre tas. Elle est en O(log(n)).
Pour réaliser cet ajout, on ajoute l'élément à la fin du tableau O(1) et on appelle ensuite la fonction remonter sur cet élément.
La fonction remonter va comparer le noeud avec son père et l'échanger s'il est plus petit que lui. Cette fonction est appelé de manière récursive jusqu'à ce que le père du noeud appelé soit plus petit que lui. On avance de père en père, on est dans le pire des cas en O(log(n)).

1.3. Implémentation de ConsIter (l.81-90)

Cette fonction va construire notre tas de manière itérative à partir du vector qu'on lui donne en entrée. Cette fonction est en O(n).
Pour réaliser cette fonction, on ajoute chacun des éléments en paramètre dans le tableau (O(n)).
Puis, à on appeler la fonction redescendre sur chaque noeud en partant du noeud le plus à droit de l'avant dernière hauteur et on va aller de droite à gauche et remonter dans les hauteurs de notre tas. La fonction "redescendre" permet de vérifier que : pour chaque noeud, si sa clef est plus grande que celle de l'un de ses fils (on vérifie d'abord le fils gauche), on échange les positions père/fils et on appelle "redescendre" récursivement sur le nouveau fils.
Preuve de la complexité :
Soit h la hauteur du tas. Au niveau i, le nombre d'échange est ≤ h-i pour chaque noeud, et on sait qu'il y a au maximum 2i noeuds. On a donc au total un nombre d'échange ≤ $\Sigma(h - i)\cdot 2^i$ (i allant de 1 à h).
Soit j = h-i, alors i = h-j et $\Sigma(h - i)\cdot 2^i = \Sigma j\ 2^{(h-j)} = (2^h)\Sigma\ j*1/(2^j)$
Or, $\Sigma\ j*1/(2^j)$= ½ + 2 ¼ + 3 ⅛ + 4 1/16 + … <= 2
Alors on a $2^h\ \Sigma\ j\ 1/(2^j)$ <= 2* 2^h = 2 n O(n) (car h est en O(log n))
Et $2^h\ \Sigma j/(2^j)$ ≤ $2^{(h+1)}$
D'où le nombre d'échanges est en O(n)

1.4. Implémentation de Union (l.91-98)

Cette fonction fait l'union des deux tas passés en paramètre. Cette fonction est en O(n+m).
On appelle constIter sur l'ensemble d'éléments, c'est à dire les éléments des deux tableaux que l'on souhaite rassembler. Si n et m correspondent au nombre d'éléments des deux tableaux, on est donc en O(n+m).

2. Implémentation du tas avec la structure d'arbre (fichier TasMinArbre.h)

2.1. Implémentation de SupprMin (l.228-249)

On va retrouver la position du dernier noeud avec la fonction DonneMoiLeDernier qui est en O(log(n)). On va echanger le dernier noeud avec la racine, ensuite on va appeler la fonction tamiser_racine sur la nouvelle racine sans oublier de supprimer l'élément minimum qui est devenu le dernier élément de l'arbre. tamiser_racine compare les valeurs des clefs du père et de ses fils, et échange la position du père avec celui du fils le plus petit. tamiser_racine est ensuite appelé de manière récursive sur le nouveau fils jusqu'à ce que le noeud sur lequel on utilise tamiser_racine n'ait pas de fils plus petit que lui même. Cette descente de fils en fils, on dans le pire des cas en O(log(n)) si on compare tous les fils du branche.

2.2. Implémentation de Ajout (l.68-72)

La fonction ajout sert à ajouter une clef dans notre tas. Elle est en O(log(n)).
Pour réaliser cet ajout, on verifie si on a une racine:
Si la racine n'existe pas, l'élément qu'on ajoute devient la nouvelle racine (O(1)).
Sinon on ajoute l'élément à la fin de l'arbre grâce à la fonction DonnePereNoeudVide qui nous permet d'obtenir le père de la où l'on doit insérer le noeud et on appelle ensuite la fonction tamiser_haut sur cet élément.
La fonction tamiser_haut va comparer le noeud avec son père et l'échanger s'il est plus petit que lui. Cette fonction est appelé de manière récursive jusqu'à ce que le père du noeud appelé soit plus petit que lui. On avance de père en père, on est dans le pire des cas en O(log(n)).

2.3. Implémentation de ConsIter (l.81-90)

Cette fonction va construire notre tas de manière itérative à partir du vector qu'on lui donne en entrée. Cette fonction est en O(n).
Pour réaliser cette fonction, on ajoute chacun des éléments en paramètre dans un arbre construit de manière arbitraire ce qui se fait en O(n).
Puis, à on appeler la fonction tamiser_bas sur chaque noeud en partant du noeud le plus à droit de l'avant dernière hauteur et on va aller de droite à gauche et remonter dans les hauteurs de notre tas. La fonction tamiser_bas permet de vérifier que : pour chaque noeud, si sa clef est plus grande que celle de l'un de ses fils (on vérifie d'abord le fils gauche), on échange les positions père/fils et on appelle tamiser_bas récursivement sur le nouveau fils.
Preuve de la complexité :
Soit h la hauteur du tas. Au niveau i, le nombre d'échange est ≤ h-i pour chaque noeud, et on sait qu'il y a au maximum 2i noeuds. On a donc au total un nombre d'échange ≤ $\Sigma(h - i)\cdot 2^i$ (i allant de 1 à h).
Soit j = h-i, alors i = h-j et $\Sigma(h - i)\cdot 2^i = \Sigma j\ 2^{(h-j)} = (2^h)\Sigma\ j*1/(2^j)$
Or, $\Sigma\ j*1/(2^j)$= ½ + 2 ¼ + 3 ⅛ + 4 1/16 + … <= 2

Alors on a 2^h Σ j 1/(2^j) <= 2* 2^h = 2 n O(n) (car h est en O(log n))
Et 2^h Σj/(2^j) ≤ 2^(h+1)
D'où le nombre d'échanges est en O(n).

2.4. Implémentation de Union (l.91-98)

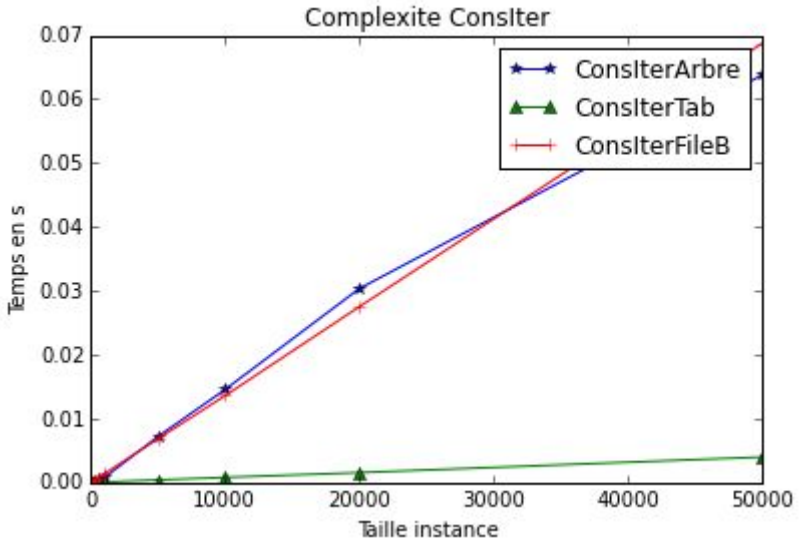
Cette fonction fait l'union des deux tas passés en paramètre. Cette fonction est en O(n+m).
On appelle constIter sur l'ensemble d'éléments, c'est à dire les éléments des deux tableaux que l'on souhaite rassembler. Si n et m correspondent au nombre d'éléments des deux tableaux, on est donc en O(n+m).

Question 2.5

Pour vérifier graphiquement la complexité temporelle de la fonction ConsIter, on utilise la fonction complexiteConsIterArbre() et complexiteConsIterTab() qui sont dans le fichier main.cpp. Pour cela on va mesurer pour chaque taille, le temps d'exécution des 5 jeux et on va faire une moyenne.

On obtient les valeurs suivantes:
ConsIterArbre (moyenne des 5 jeux):
Taille : 100 ; Temps : 9.58e-05
Taille : 200 ; Temps : 0.0002108
Taille : 500 ; Temps : 0.0004208
Taille : 1000 ; Temps : 0.0008434
Taille : 5000 ; Temps : 0.0070818
Taille : 10000 ; Temps : 0.0134284
Taille : 20000 ; Temps : 0.0273536
Taille : 50000 ; Temps : 0.0574186
ConstIterTab (moyenne des 5 jeux):
Taille : 100 ; Temps : 1.02e-05
Taille : 200 ; Temps : 1.7e-05
Taille : 500 ; Temps : 3.46e-05
Taille : 1000 ; Temps : 6.56e-05
Taille : 5000 ; Temps : 0.0003808
Taille : 10000 ; Temps : 0.0007798
Taille : 20000 ; Temps : 0.0016288
Taille : 50000 ; Temps : 0.0035216

Ce qui nous donne le graphe suivant :



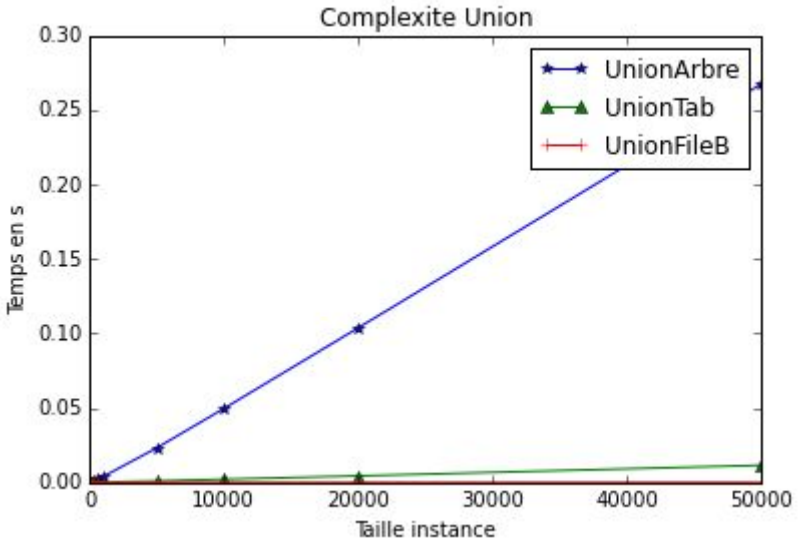
On remarque que la complexité de consIter pour un tas en tableau est bien inférieur à celle pour la réprésentation en arbre. C'est en moyenne 15 fois plus rapide de faire consIter avec un tableau qu'avec un arbre.

Question 2.6

Pour vérifier graphiquement la complexité temporelle de la fonction Union, on utilise la fonction complexiteUnionArbre() et complexiteUnionTab() qui sont dans le fichier main.cpp. Pour cela, on va mesurer le temps d'exécution pour chaque taille, en faisant l'union des 5 jeux, c'est-à-dire que l'on va faire 4 unions donc on divise le temps finale par 4.

On obtient les valeurs suivantes:
UnionArbre (moyenne des 4 union des 5 jeux):
Taille : 100 ; Temps : 0.00030025
Taille : 200 ; Temps : 0.0006445
Taille : 500 ; Temps : 0.0018375
Taille : 1000 ; Temps : 0.00388675
Taille : 5000 ; Temps : 0.0231805
Taille : 10000 ; Temps : 0.0495845
Taille : 20000 ; Temps : 0.103665
Taille : 50000 ; Temps : 0.267189
UnionTab (moyenne des 4 union des 5 jeux):
Taille : 100 ; Temps : 2.2e-05
Taille : 200 ; Temps : 3.975e-05
Taille : 500 ; Temps : 9.1e-05
Taille : 1000 ; Temps : 0.000176
Taille : 5000 ; Temps : 0.0010035
Taille : 10000 ; Temps : 0.0021495
Taille : 20000 ; Temps : 0.00439
Taille : 50000 ; Temps : 0.011522

Ce qui nous donne le graphe suivant :



On remarque que la complexité de Union en représentation de tableau est bien inférieur à celle de la représentation en arbre. C'est environ 20 fois plus rapide de faire une union avec les tableaux qu'avec un arbre.

Question 2.7

Nos tests nous permettent de conclure que la représentation des tas en tableau est beaucoup plus efficace que celle en tableau lorsqu'on l'on souhaite faire des opérations de construction et d'union. Cette représentation en tableau est en moyenne 15 à 20 fois plus rapide que la représentation en arbre, ce qui est très significative.

3. Structure 2 : Files binomiales

Question 3.8/3.9 :

Pour implémenter nos tournois binomiaux on a implémenté les fonctions suivantes (dans le fichier TournoiB.h/TournoiB.cpp) :

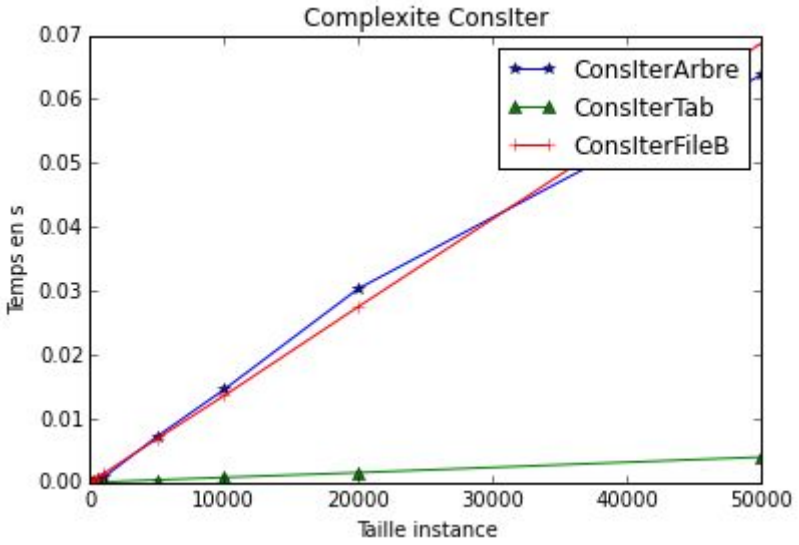
- getter/setter
- addFils(TournoiB* t) -> ajoute t comme fils du tournoi courant
- estVide(TournoiB* t) -> renvoie si le tournoi donné en paramètre est vide
- degre(TournoiB* t) -> renvoie le degré de t
- union2Tid(TournoiB *t1, TournoiB * t2) -> retourne l'union de deux tournois TBx en un seul tournoi TBx+1 : on compare la valeur des clefs des deux racines, et la plus petite devient le fils de l'autre.
- decapite(TournoiB *t) -> retourne la file binomiale composée des fils du tournoi courant
- afficher(TournoiB *tb) -> affiche la clef de chaque noeud formant le tournoi

Pour implémenter nos files binomiaux on a implémenté les fonctions suivantes (dans les fichiers FileB.h/FileB.cpp) :

- getter
- estVide(FileB* f) -> renvoie si la file donnée en paramètre est vide
- minDeg(FileB *f) -> retourne le tournoi de plus faible degré de la file (il se trouve au bout de la file).
- reste(FileB *f) -> retourne la file privée de son tournoi minimal (que l'on retire du bout de la liste).
- ajoutMin(TournoiB *t, FileB *f) -> retourne une file possédant un nouveau tournoi de degré minimal, que l'on ajoute à la fin de la file (pas de vérification : on part du principe que le tournoi fourni est bel et bien celui de degré min).
- uFret(FileB *f1, FileB *f2, TournoiB *t) -> pseudo code du cours
- unionFile(FileB* f1, FileB *f2) -> pseudo code du cours
- afficher(FilB* f) -> appelle les fonctions "affiche" de chaque tournoi de la file
- afficherTaille(FileB* f) -> affiche le degré des fils de la file
- supprMin(FileB *f) -> retourne la file privée de son tournoi minimal, que l'on remplace par une liste de tournoi obtenu avec "decapite"
- ajout(FilB *f, Clef *c) -> on crée un tournoi avec la clef donnée en paramètre, puis on l'ajoute à la file
- consIterFileB(std::vector<Clef *> *v) -> on crée une nouvelle file vide. Puis, avec chaque clef, on crée un tournoi que l'on utilise en paramètre d'une nouvelle file, que l'on peut alors fusionner avec la file vide de départ.

Question 3.10

En mesurant le temps de calcul de consIter sur une file binomiale (avec la fonction complexiteConsIterFileB()), on a obtenu les temps suivants :
ConsIterFileB (moyenne des 5 jeux):
Taille : 100 ; Temps : 0.0001744
Taille : 200 ; Temps : 0.0003436
Taille : 500 ; Temps : 0.0008666
Taille : 1000 ; Temps : 0.0017294
Taille : 5000 ; Temps : 0.0086422
Taille : 10000 ; Temps : 0.017425
Taille : 20000 ; Temps : 0.035557
Taille : 50000 ; Temps : 0.0867894
Ce qui donne le graphe suivant :



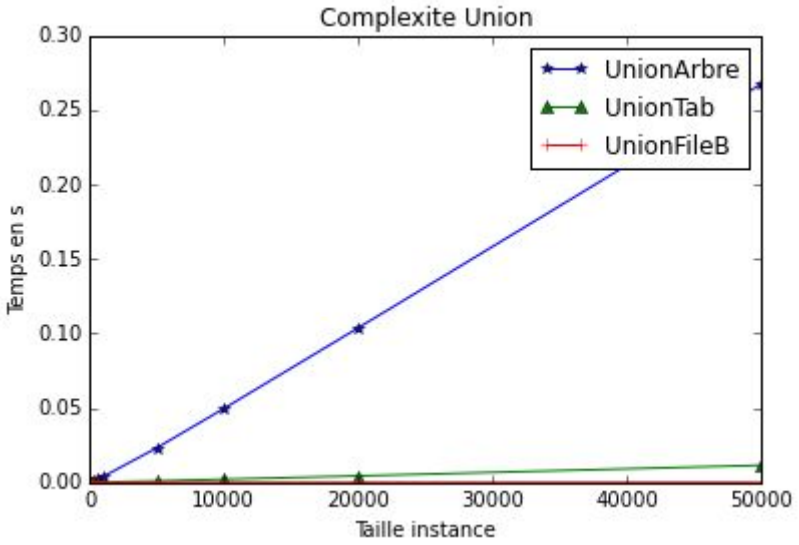
On remarque que consIter sur une file binomiale est aussi rapide que avec un tas min en représentation d'arbre. Ce résultat est plutôt normale car les deux complexités sont en O(n).

Question 3.11

Avec la même méthode que à la question 2.6, on a mesuré le temps de calcul d'union sur une file binomiale (avec la fonction complexiteUnionFileB()), on a obtenu les temps suivants :

UnionFileB (moyenne des 4 union des 5 jeux):
Taille : 100 ; Temps : 3e-06
Taille : 200 ; Temps : 2.75e-06
Taille : 500 ; Temps : 1.35e-05
Taille : 1000 ; Temps : 7e-06
Taille : 5000 ; Temps : 7e-06
Taille : 10000 ; Temps : 7.75e-06
Taille : 20000 ; Temps : 1.3e-05
Taille : 50000 ; Temps : 9e-06

Ce qui donne le graphe suivant :



On remarque que l'union sur une file binomiale est encore plus rapide que l'union pour un tas min sous forme de tas. Environ 100 fois plus rapide. Ce qui est normale car la complexité de l'union pour une file binomiale est en log(n+m) contrairement au tableau qui est en O(n+m).

4. Fonction de hachage

Nous avons implémenté l'algorithme de md5 fournis par wikipedia dans le fichier md5.h. La fonction de test es la fonction testMD5() défini dans le fichier ABR.cpp.
Par exemple pour la phrase "The quick brown fox jumps over the lazy dog", on obtient bien le hachage suivant : 9e107d9d372bb6826bd81d3542a419d6

5. Arbre de Recherche

Pour notre structure arborescente de recherche, on a décidé d'implémenter un arbre binaire de recherche non équilibré. Cette structure est implémenté dans les fichiers ABR.h/ABR.cpp. Cette structure permet bien de savoir en moyenne si un élément est contenu dans la structure en O(log(n)). Pour implémenter cette structure, on a défini les fonctions suivantes:

- getter/setter
- estFeuille(ABR*A) -> vérifie si le noeud passé en paramètre à des fils, s'il n'en a pas, c'est qu'il est une feuille
- ajout(ABR* A, std::string hash) -> on parcourt l'arbre jusqu'à atteindre la position correspondant à l'expression que l'on cherche, et on l'intègre à l'arbre
- ajout(ABR* A, std::string hash, string s, list<string>*l) -> même chose que l'ajout précédent, mais on ajoute en plus l'expression à la liste de toutes les expressions déjà ajoutés

- consIterABR(vector<Clef *>* vec) -> construit un ABR à l'aide d'un vecteur de clef. On commence par mettre à la racine du nouvel arbre la dernière clef du vecteur, puis on appelle la fonction "ajout (ABR*, string)" sur chaque autre clef pour les intégrer à l'arbre
- consIterABR(vector<string>* vec, list<string> *l) -> cette fois, on construit l'arbre avec pour racine le premier élément d'un vecteur de string, puis on ajoute cette string à la liste de string. On complète l'arbre avec les autres éléments du vecteur, à l'aide de la fonction "ajout (ABR*, , string, string, list<string>)", qui va se charger d'ajouter les string à la liste.
- afficher(ABR *A) -> affiche l'arbre (dans l'ordre filsG/noeud/filsD)

6. Étude expérimentale

Question 6.12

Pour stocker le haché MD5 de chaque mot de l'oeuvre de Shakespeare, on a utilisé la fonction testStructureABR(). Le fichier toto contient la concaténation de tous nos fichiers sur les oeuvres de Shakespeare. On a lancé la lecture de ce fichier et on a stocké tous les md5 dans une liste de string. En prenant la taille de la liste on obtient qu'il y a 23086 mots différents dans toutes les oeuvres de Shakespeare.

Question 6.13

Dans l'oeuvre de Shakespeare on n'a aucun mots différents qui sont en collision pour MD5. Ce résultat est tout à fait normal car si on se base sur le paradoxe des anniversaires, on peut démontrer que pour avoir une chance sur deux d'avoir une collision il faudrait 2^64 hashé, alors que nous n'avons même pas 2^15 hashé.

Question 6.14

Nous avons testé nos deux structures de tas min et celle de files binomiales. Pour faire l'union nous avons effectué l'union des 5 jeux différents et donc divisé le temps par les 4 unions nécessaires.

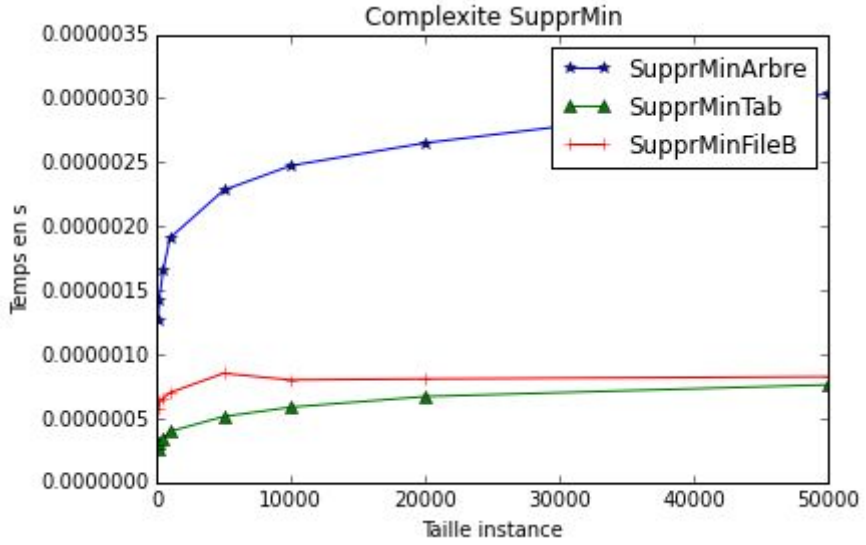
1. Pour SupprMin,

on a obtenu les temps suivants :

SupprMinArbre (1 operation):
Taille : 100 ; Temps : 1.262e-06
Taille : 200 ; Temps : 1.429e-06
Taille : 500 ; Temps : 1.6588e-06
Taille : 1000 ; Temps : 1.9092e-06
Taille : 5000 ; Temps : 2.28224e-06
Taille : 10000 ; Temps : 2.47398e-06
Taille : 20000 ; Temps : 2.65117e-06
Taille : 50000 ; Temps : 3.03377e-06

SupprMinTab (1 operation):
Taille : 100 ; Temps : 2.56e-07
Taille : 200 ; Temps : 3.14e-07
Taille : 500 ; Temps : 3.464e-07
Taille : 1000 ; Temps : 3.974e-07
Taille : 5000 ; Temps : 5.14e-07
Taille : 10000 ; Temps : 5.8694e-07
Taille : 20000 ; Temps : 6.7029e-07
Taille : 50000 ; Temps : 7.62296e-07
SupprMinFileB (1 operation):
Taille : 100 ; Temps : 5.84e-07
Taille : 200 ; Temps : 6.32e-07
Taille : 500 ; Temps : 6.568e-07
Taille : 1000 ; Temps : 7.01e-07
Taille : 5000 ; Temps : 8.5188e-07
Taille : 10000 ; Temps : 7.9984e-07
Taille : 20000 ; Temps : 8.0846e-07
Taille : 50000 ; Temps : 8.25936e-07

À partir de ces données, on a obtenu le graphe suivant :



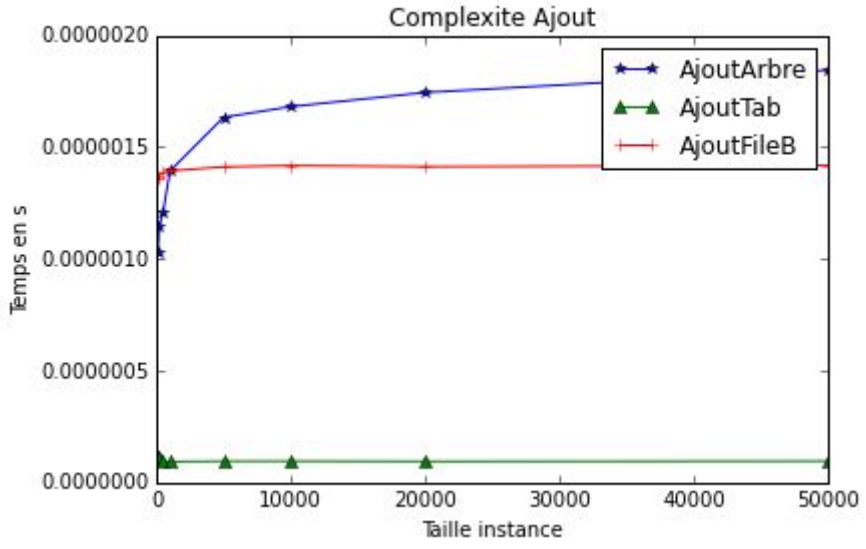
On remarque que si l'on souhaite effectuer beaucoup de supression on devrait plutôt utiliser la structure de tas min en tableau ou la file binomiale. Ces deux implémentations ont un temps environ 3 à 4 fois meilleurs que celui de l'implémentation en tas min sous forme d'arbre.

2. Pour Ajout,

on a obtenu les temps suivants :
AjoutArbre (1 operation):
Taille : 100 ; Temps : 1.03e-06
Taille : 200 ; Temps : 1.144e-06
Taille : 500 ; Temps : 1.2116e-06

Taille : 1000 ; Temps : 1.3928e-06
Taille : 5000 ; Temps : 1.633e-06
Taille : 10000 ; Temps : 1.68066e-06
Taille : 20000 ; Temps : 1.74383e-06
Taille : 50000 ; Temps : 1.84391e-06
AjoutTab (1 operation):
Taille : 100 ; Temps : 1.14e-07
Taille : 200 ; Temps : 1.15e-07
Taille : 500 ; Temps : 9.4e-08
Taille : 1000 ; Temps : 9.3e-08
Taille : 5000 ; Temps : 9.52e-08
Taille : 10000 ; Temps : 9.544e-08
Taille : 20000 ; Temps : 9.413e-08
Taille : 50000 ; Temps : 9.556e-08
AjoutFileB (1 operation):
Taille : 100 ; Temps : 1.36e-06
Taille : 200 ; Temps : 1.372e-06
Taille : 500 ; Temps : 1.386e-06
Taille : 1000 ; Temps : 1.3936e-06
Taille : 5000 ; Temps : 1.41132e-06
Taille : 10000 ; Temps : 1.41726e-06
Taille : 20000 ; Temps : 1.41238e-06
Taille : 50000 ; Temps : 1.41791e-06

À partir de ces données, on a obtenu le graphe suivant :



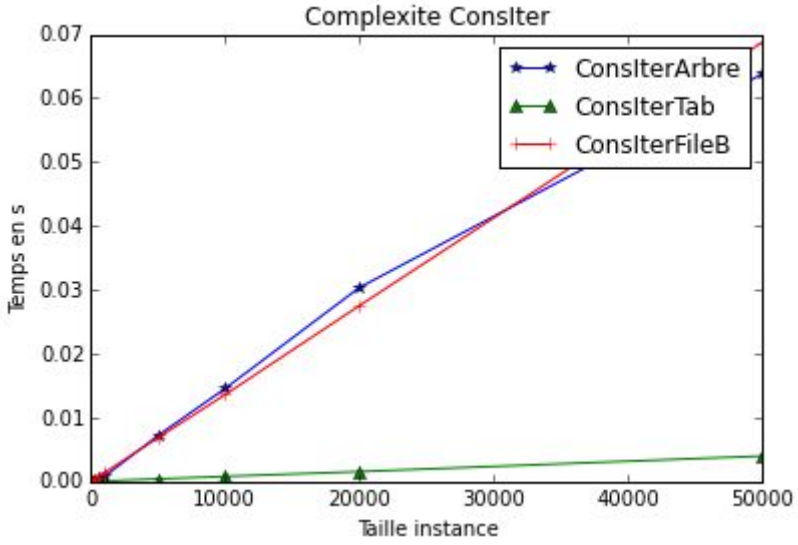
On remarque que la structure de tas min en tableau a des temps bien meilleur que pour la file binomiale (environ 10 fois plus rapide).

3. Pour ConsIter,

on a obtenu les temps suivants :
ConsIterArbre (moyenne des 5 jeux):

Taille : 100 ; Temps : 8.68e-05
Taille : 200 ; Temps : 0.0002324
Taille : 500 ; Temps : 0.0004234
Taille : 1000 ; Temps : 0.0007464
Taille : 5000 ; Temps : 0.0071914
Taille : 10000 ; Temps : 0.0145456
Taille : 20000 ; Temps : 0.0303476
Taille : 50000 ; Temps : 0.0637792
ConsIterTab (moyenne des 5 jeux):
Taille : 100 ; Temps : 9.6e-06
Taille : 200 ; Temps : 1.7e-05
Taille : 500 ; Temps : 3.78e-05
Taille : 1000 ; Temps : 7.26e-05
Taille : 5000 ; Temps : 0.0004
Taille : 10000 ; Temps : 0.0007904
Taille : 20000 ; Temps : 0.0015426
Taille : 50000 ; Temps : 0.0039764
ConsIterFileB (moyenne des 5 jeux):
Taille : 100 ; Temps : 0.000124
Taille : 200 ; Temps : 0.0002588
Taille : 500 ; Temps : 0.000685
Taille : 1000 ; Temps : 0.0013546
Taille : 5000 ; Temps : 0.0068424
Taille : 10000 ; Temps : 0.0135868
Taille : 20000 ; Temps : 0.0275114
Taille : 50000 ; Temps : 0.0687178

À partir de ces données, on a obtenu le graphe suivant :



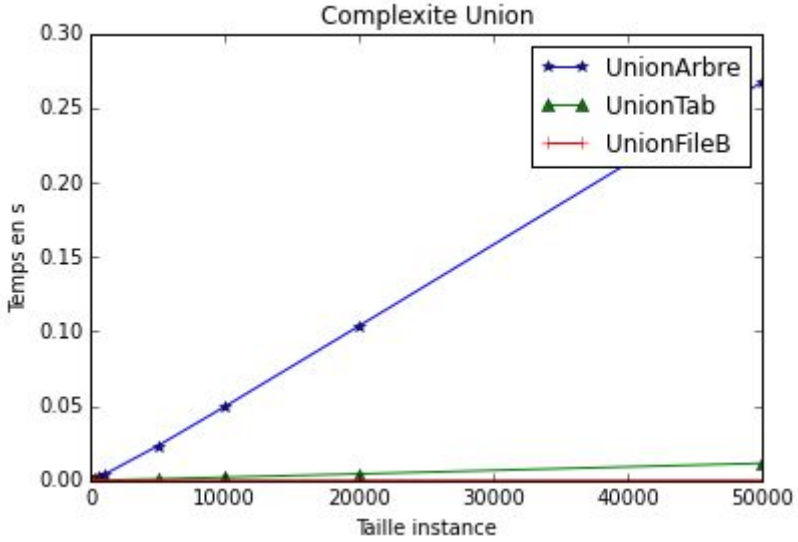
On remarque que la structure de tas min en tableau a des temps bien meilleur que pour la file binomiale (environ 10 fois plus rapide).

4. Pour Union,

on a obtenu les temps suivants :

UnionArbre (moyenne des 4 union des 5 jeux):
Taille : 100 ; Temps : 0.00030025
Taille : 200 ; Temps : 0.0006445
Taille : 500 ; Temps : 0.0018375
Taille : 1000 ; Temps : 0.00388675
Taille : 5000 ; Temps : 0.0231805
Taille : 10000 ; Temps : 0.0495845
Taille : 20000 ; Temps : 0.103665
Taille : 50000 ; Temps : 0.267189
UnionTab (moyenne des 4 union des 5 jeux):
Taille : 100 ; Temps : 2.2e-05
Taille : 200 ; Temps : 3.975e-05
Taille : 500 ; Temps : 9.1e-05
Taille : 1000 ; Temps : 0.000176
Taille : 5000 ; Temps : 0.0010035
Taille : 10000 ; Temps : 0.0021495
Taille : 20000 ; Temps : 0.00439
Taille : 50000 ; Temps : 0.011522
UnionFileB (moyenne des 4 union des 5 jeux):
Taille : 100 ; Temps : 3.5e-06
Taille : 200 ; Temps : 3.75e-06
Taille : 500 ; Temps : 7.25e-06
Taille : 1000 ; Temps : 7.75e-06
Taille : 5000 ; Temps : 1.05e-05
Taille : 10000 ; Temps : 1.05e-05
Taille : 20000 ; Temps : 1.25e-05
Taille : 50000 ; Temps : 1.475e-05

À partir de ces données, on a obtenu le graphe suivant :



On remarque que les temps pour faire l’union entre deux file binomiales est environ 100 fois plus rapide qu’avec le tas min en tableau qui était déjà 20 fois plus rapide qu’avec la représentation en arbre.

Conclusion :

Pour conclure, on peut dire que si l’on souhaite faire beaucoup d’union sur nos données alors la structure très largement conseillé sera la file binomiale, ce qui est normale car on aura une complexité en O(log(n+m)) contrairement au tas qui est en O(n+m).
Par contre si on souhaite plutôt faire majoritairement des constIter alors on aura tendance à préférer le tas min implémenté avec une structure de tableau qui a un temps d'exécution significativement inférieur à celui des files binomiales.

FIN
