Master 1 : STL

Algorithmique avancée
Devoir de programmation

Année 2023/2024

Mengxiao LI 21304592

Zhengdao YU(21304260)

12 décembre 2023

Table des matières

I. Échauffement

1. La Structure

Ci-contre la structure que nous proposons pour représenter les clés de 128 bits sous le nom **Key128**, une clé de 128 bits est divisée en 4 parties dont chacune contient 32 bits.

```
typedef struct {
    unsigned int part1;
    unsigned int part2;
    unsigned int part3;
    unsigned int part4;
} Key128;
```

2. Les Fonctions

Ci-dessous les fonctions implémentées :

- **createKey** permet de créer une clé 128 bits
- **inf** pour savoir si key1 est inférieur à key2
- **eg** pour vérifier l'égalité entre deux clés passées en arguments

```
Key128 createKey(unsigned int p1, unsigned int p2, unsigned int p3, unsigned int p4);
//Q1.2
bool inf(Key128 key1, Key128 key2);
//Q1.3
bool eg(Key128 key1, Key128 key2);
```

Ensuite, nous avons implémenté d'autres fonctions/procédures pour effectuer des tests (création, conversion, lecture de fichier, etc.) avec les jeux de données fournis.

```
Key128 hexTokey128(const char* hex);
void convertHexToKey128(const char *inputFile, const char *outputFile);
void convertAllFilesInFolder(const char *inputFolder, const char *outputFolder);
//从文件创建数组
Key128* buildArrayFromFile(const char* filename, int* size);
Key128* resizeArray(Key128* array, int newSize);
Key128* processFile(const char* filename, int* size);
Key128* processFile1(const char* filename, int* size);
```

II. Structure 1 : Tas priorité min

Tableau

1.1 Structure

Voici la structure que nous avons définie pour représenter le tas min sous forme d'un tableau, cette structure est composée de trois valeurs qui sont :

```
typedef Key128 HPDataType;
//tas tableau
typedef struct HP
{
    HPDataType* a;
    int size;
    int capacity;
}HP;
```

- **a** : la clé 128 bits
- **size** : la taille du tableau
- **capacity** : la capacité du tableau

1.2 les 3 fonctions fondamentales d'un tas min

Premièrement, un tas min doit répondre à la propriété suivante : soit un nœud N dont l'identifiant est x, alors l'identifiant de son enfant gauche doit être 2x+1, et l'identifiant de son enfant droit est 2x+2. L'identifiant représente la position du nœud dans notre structure, que ce soit tableau ou arbre binaire.

De là, cet identifiant nous servira par la suite à effectuer des échanges de nœuds ou faire des comparaisons de valeurs.

Les trois fonctions fondamentales sont Ajout, SupprMin et AjoutsIteratifs, nous allons expliquer leur fonctionnement et montrer leur complexité :

- **Ajout(tas, data)**

Fonctionnement : tout d'abord, data est ajouté à la fin de notre tas, puis, on compare la valeur du data et son parent grâce à leur identifiant, si data est plus petit que son parent, alors on effectue l'échange, et ainsi de suite jusqu'à ce que data soit plus grand que son parent.

Complexité : un tas min est un arbre binaire complet, soit un tas min de taille n, alors sa hauteur est log(n)+1, donc le nombre d'échanges maximum est 2*(log(n)+1), alors sa complexité au pire est O(log n).

- **SupprMin(tas)**

  Fonctionnement : elle supprime la valeur minimale du tas qui se situe à la racine. Dans un premier temps, on remplace la valeur de la racine par celle du dernier nœud, et on supprime le dernier nœud du tas. Ensuite, on compare la valeur de la racine avec ses enfants, et de le faire descendre récursivement dans le tas en l'échangeant avec l'enfant qui a la valeur minimale, jusqu'à la fin du tas ou lorsqu'il est plus petit que ses enfants.

  Complexité : la suppression du dernier nœud et l'échange entre racine et le dernier nœud peuvent être réaliser en O(1), seule la descente de la racine impacte la complexité. Comme la hauteur d'un arbre binaire complet est log(n)+1 alors, la complexité au pire cas est O(log n).

- **AjoutsIteratifs(tas, data[])**

  Fonctionnement : elle ajoute progressivement une liste des clés dans le tas en utilisant la fonction Ajout.

  Complexité : Soit n la taille de la liste, m la taille du tas et log(m)+1 la hauteur du tas, comme on effectue n ajout de clés dans le tas de taille m, alors la complexité est O(n log m).

1.3 Construction

Pour construire un tas à partir d'une liste de clés, on ajoute tout d'abord toutes les clés dans notre tas, ensuite, on effectue les opérations de descente des nœuds pour les ranger dans l'ordre à partir du parent du dernier nœud jusqu'à la racine, en utilisant une fonction auxiliaire aux(minHP, IDParent).

Dans la fonction aux, on suppose que le parent contient la clé minimale dans un premier temps. Grâce à la propriété du tas, on connaître l'identifiant de tous les nœuds fils, alors on peut comparer la clé du parent avec ses enfants, puis chercher si le parent a effectivement la clé minimale. Si ce n'est pas le cas, alors on échange la valeur du fils avec ce parent, et on continue la récursion sur ce fils qui vient de recevoir la clé du parent, jusqu'au dernier nœud.

2.3.1 Le pseudo code de la fonction

```
Aux (INOUT minHP , index )
      smallest = index;
      idFg = 2*index +1
      idFd = 2*index +2
      si ( idFg < minHp->size && minHp->a[idFg]<minHp->a[smallest])
            smallest=idFg
      si( idFd < minHp->size && minHp->a[idFd]<minHp->a[smallest])
            smallest=idFg
      si ( smallest != index )
            change( minHp->a[smallest] , minHp->a[index])
            Aux(minHP , samllest)

Construction( INOUT minHP, keys[])
      n = keys.length()
      Pour i de 0 à n faire      //Pour insere
            minHp->a[i] = key [i];
     Pour i de (n-2)/2 à 0 faire
            Aux(minHP , i );
```

2.4 Union

Comme **Construction**, on ajoute les clés des deux tas dans un seul nouveau tas, ensuite, on effectue les opérations de descente des nœuds comme décrit précédemment dans la partie Construction afin de ranger les nœuds dans l'ordre pour obtenir un tas min.

2.4.1 le pseido de la fonction

```
Union ( IN minHP1, IN minHp2 )
     minHpUnion=minHp1+minHp2
     n=minHpUnion->size ;
     Pour i de (n-2)/2 à 0 faire
           Aux(minHP , i );
      Fin pour
```

2.5 Prouver les complexités de Construction et Union

La complexité de l'insertion des clés d'une liste de taille n dans un tas est O(n).

Soit notre tas (arbre binaire complet) de hauteur h, et le nœud ciblé se situe à la hauteur x. Le nombre d'échanges maximum est h-x, et chaque échange nécessite 2 comparaisons, ainsi, le nombre de comparaisons maximales est 2(h-x).
De là, le nombre de comparaison de chaque nœud du tas est :

$$2^0 \times\ 2(h-1) + 2^1 \times 2 \times 2(h-2) + \cdots + 2^{h-2} \times 2\big(h-(h-1)\big)$$

$On\ obtient\ alors$:

$$A = \sum_{i=1}^{h-1} 2^{i-1} \times (h-i) = \sum_{i=1}^{h-1} 2^{i} \times (h-1)$$

$Avec\ h = log\ n + 1$

$Soit\ j = h - i, alors\ i = h - j$

$$\sum_{j=h-1}^{1} 2^{h-j} \times j = \sum_{j=logn}^{1} 2^{\log n} \times 2 \times 2^{-j} \times j$$
$$= \sum_{j=logn}^{1} 2n \times 2^{-j} \times j$$
$$= 2n \sum_{j=h}^{1} 2^{-j} \times j$$

$Soit \quad S = \sum_{j=h}^{1} 2^{-j} \times j$

$donc \quad S\ = 1 \times \frac{1}{2} + 2 \times \frac{1}{4} + 3 \times \frac{1}{8} + \cdots + h \times \frac{1}{2^h}$

$on\ a \quad \frac{1}{2}S = 1 \times \frac{1}{4} + 2 \times \frac{1}{8} + \cdots + (h-1) \times \frac{1}{2^h} + h \times \frac{1}{2^{h+1}}$

$Donc\ S - \frac{1}{2}S\ =\ \frac{1}{2}S = \frac{1}{2} + \frac{1}{4} + \frac{1}{8} + \cdots + \frac{1}{2^h} - \frac{1}{2^{h+1}}$

$$= \frac{1-\left(\frac{1}{2}\right)^h}{1-\frac{1}{2}} - \frac{h}{2^{h-1}}$$

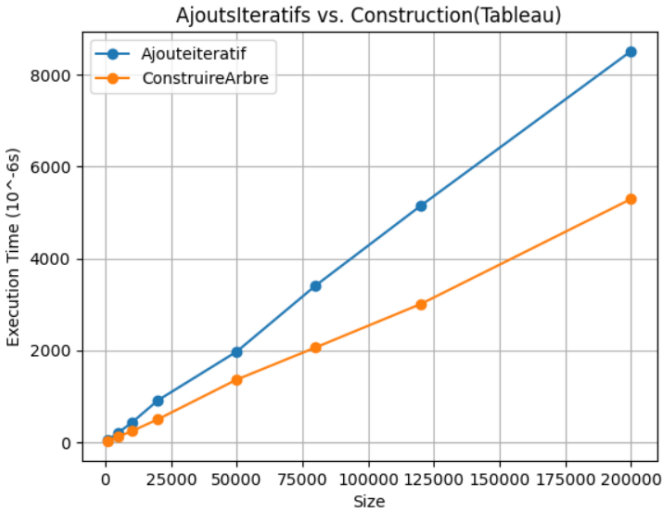
$$=\ 1 - \frac{1}{2^h} - \frac{h}{2^{h-1}}$$

$$Donc\ S = 2 - \frac{1}{2^{h-1}} - \frac{h}{2^h} \leq 2$$
$$A = 2nS \leq 2n \times 2$$
$$A \leq 4n$$

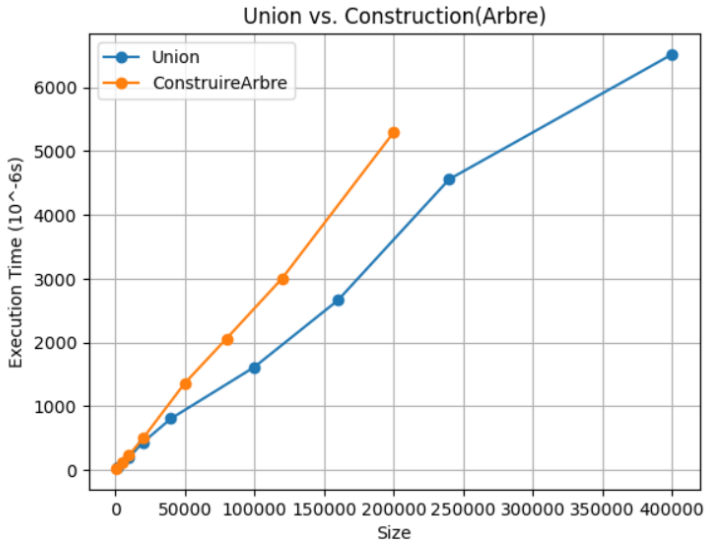
Donc la complexité de **Construction** est O(n).
Donc le complexité de **Construction** est même O(n).

2.6 Visualisation des complexités par les graphiques

Dans les tests que nous avons effectué, le temps d'exécution des fonctions AjoutsIteratifs et ConstruireArbre est linéare en fonction de la taille des données. Et on peut aussi faire la remarque que la construction est plus rapide que ajout sur la figure ci-dessous.



La figure représente le temps d'exécution de la fonction Union sur deux fichier de même taille, dont la taille est précisée sur l'axe des abscisse. On note également que le temps reste linéaire.
L'axe des x de ce graphique représente le nombre total de clés. Nous constatons que bien que l'union et la construction aient le même principe, l'union prend moins de temps car les deux tas sont déjà des tas minimaux.



Arbre Binaire

2.1 Structure

Voici la représentation de notre tas min en tant qu'un arbre binaire, nous avons choisi d'implémenter une structure composée aussi de trois valeurs qui sont :

- **data** : la clé 128 bits
- **fg** : enfant gauche
- **fd** : enfant droit

```
typedef Key128 HPType;
typedef struct HPArb {
    HPType data;
    struct HPArb* fG;  // fils gauche
    struct HPArb* fD;  // fils droit
} HPArb;
```

Dans cette structure, on a pas accès directement au parent du nœud, donc nous avons implémenter une autre structure **Queue** pour pouvoir y accéder pendant les différentes opérations.

```
typedef struct QueueNode {
    HPArb* data;
    struct QueueNode* next;
} QueueNode;


typedef struct Queue {
    QueueNode* front;
    QueueNode* rear;
} Queue;

Queue* createQueue();
void enqueue(Queue* q, HPArb* data);
HPArb * dequeue(Queue* q);
```

Cette structure **Queue** possède la propriété First In, First Out (FIFO), c'est-à-dire premier entré, premier sorti, qui pourra nous aider à gérer le tas min sous forme d'un arbre binaire.

2.2 Les 3 fonctions fondamentales d'un tas min

Comme pour les tas min avec la structure du tableau, on présente dans cette partie les trois fonctions fondamentales.

- **Ajout(tas, data)**

  Fonctionnement : l'idée reste identique, mais l'intégration de **Queue** rend l'algorithme différent. Pendant l'insertion, on utilise le parcours en largeur pour trouver la position idéal du nouveau nœud à insérer, puis, on retourne l'identifiant du nœud inséré. Et durant la remontée de ce nouveau nœud, nous appelons la fonction `*HPArb* findNode(HPArb* tas, int index)*`, qui lui utilise le parcours en largeur pour localiser le nœud dont l'index est passé en argument. Et grâce à cette fonction, on peut échanger et/ou comparer le nœud courant et son parent.

  Complexité : L'insertion du nœud coûte O(n), et l'étape de la modification du tas coûte O(log n) car chaque échange nécessite l'appel de la fonction **findNode** pour trouver le parent du nœud courant. Donc la complexité finale est O(n log n).

- **SupprMin(tas)**

  Fonctionnement : Même idée, mais dans cette fonction, nous utilisons trois fonctions auxiliaires :
  `int countNodes(HPArb* tas)` qui retourne le nombre de nœud du tas, qui correspond aussi à l'identifiant du dernier nœud
  `void deleteLastNode(HPArb **tas, int idlast)` à l'aide de la fonction **countNodes** et **findNode** pour trouver son parent, on peut alors utiliser cette fonction pour supprimer le dernier nœud du tas.
  `void desentre(HPArb *tas)` a la même fonctionnalité que la fonction auxiliaire **Aux** qui consiste à faire les opérations en descente présenter précédemment, pour ranger les nœuds dans l'ordre afin de constituer un tas min.

  Complexité : la complexité de **countNodes** et de **deleteLastNode** sont O(n), et la complexité de **descendre** est O(long n), donc celle de **SupprMin** est O(n).

- **AjoutsIteratifs(tas, data[])**

  Fonctionnement : pareil que la structure tableau.

  Complexité : soit la liste de taille n et le tas de taille m. On effectue n fois l'opération Ajout dont la complexité est O(m log m), donc la complexité finale est O(n * m * log m)).

2.3 Construction

Pour réaliser la construction de tas min avec la structure arbre binaire, nous utilisons un tableau de pointeur nommé **nodeRefs**, chaque élément de ce tableau point vers un nœud du tas. Donc pendant la construction, à chaque création de nœud dans l'arbre, le pointeur correspondant du nœud est ajouté dans **nodeRefs**, et nous permettra d'y accéder sans besoin de parcourir tout l'arbre.

2.3.1 Le pseudo code de la fonction

```
Construction( INOUT minHP, keys[])
    length = keys.length()
    CreateABC(minHp,keys[],length,lastId,nodeRefs[])
    Aux(lastId , nodeRefs[])

Aux(lastId ,nodeRefs[])
    Size= lastId
    Pour i de (n-2)/2 à 0 faire
        index =i
        tant que TRUE
            smallest=i
            idFg = 2*index +1
            idFd = 2*index +2
            Si idFg>size alors break
            parentNode = nodeRefs[index]
            filsG = nodeRefs[idFg];
            Si idFd<=size alors filsD= nodeRefs[idFd]
            Sinon alors filsD est NULL

            si ( filsG != NULL && filsG->data < parentNode ->data)
                smallest=idFg
            si ( filsD != NULL && filsD->data < parentNode ->data)
                smallest=idFg
            si ( smallest != i )
                change( nodeRefs[smallest] , parentNode)
                index=smsamllest
            sinon
                Sortir de la boucle
        fin tant que
    fin pour
```

```
CreateABC(INOUT minHp,keys[],length,INOUT lastId,INOUT nodeRefs)
    Index=0
    minHp=nouveauNoeud(keys[0]) ;
    nodeRefs[index++]=tas
    enqueue(queue,tas)
    current = 1
    tant que current<length et index<length
        parent = dequeu(queue)
        Si Parent.filsGouche() est NULL
            Parent.filsGouche()=nouveauNoeud(keys[current])
            Enqueue(queue, Parent.filsGouche())
            nodeRefs[index++]=parent-> filsGouche()
            current++
        Si Parent.filsDroit() est NULL
            Parent. filsDroit ()=nouveauNoeud(keys[current])
            Enqueue(queue, Parent. filsDroit ())
            nodeRefs[index++]=parent-> filsDroit()
            current++
    fin tant que
    lastId++
```

2.4 Union

Dans la procédure Union de la structure arbre binaire, nous avons implémenté une fonction auxiliaire **treeToArray** qui fait un parcours en largeur, permettant de construire un tableau de clé 128 bits à partir d'un arbre binaire, et de nous le retourner. La variable **length** correspond au nombre d'élément dans le tableau, et elle est actualisée à la fin du parcours.

```
treeToArry (minHp , INOUT length): Key128[]
    Si(minHp==NULL)
        length=0
        return NULL
    count=0
    enqueue(queue,minHp)
    tant que queue est non vide
        node=deqieie(queue)
        array[count++]=node.data()
        si(node.filsGauche() est non vide)
            enqueue(queue, node.filsGauche())
        si(node.filsDauche() est non vide)
            enqueue(queue, node.filsDauche())
    fin tantque
    length=count
    return array

Union( IN minHP1, IN minHp2 ):  HPArb
    array1= treeToArry(minHp1 ,array1[] ,arrayLength1)
    array2= treeToArry(minHp2 ,array2[] ,arrayLength2)

    unionArr[]=array1+array2;
    construction(minHp,unionArr[])
```

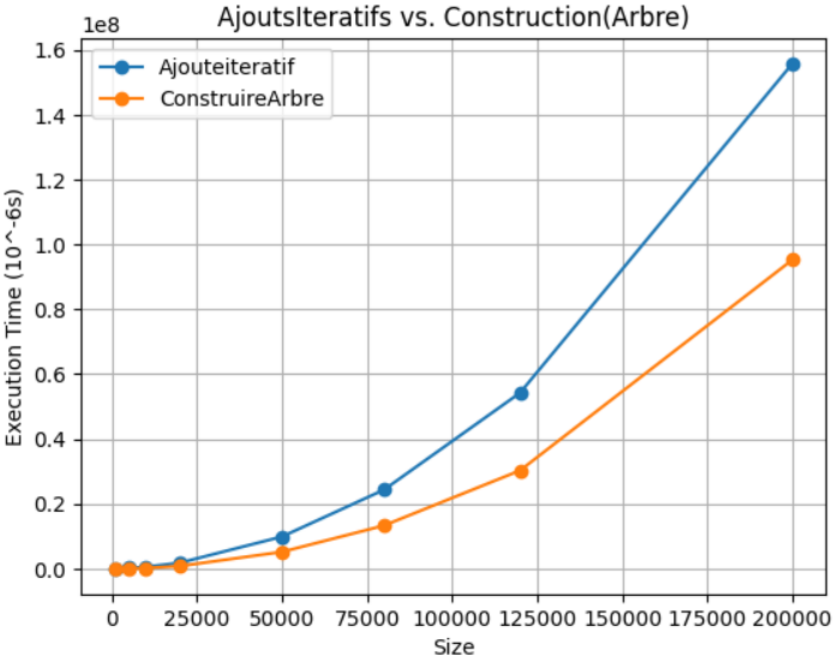
2.5 Prouver les complexités de Construction et Union

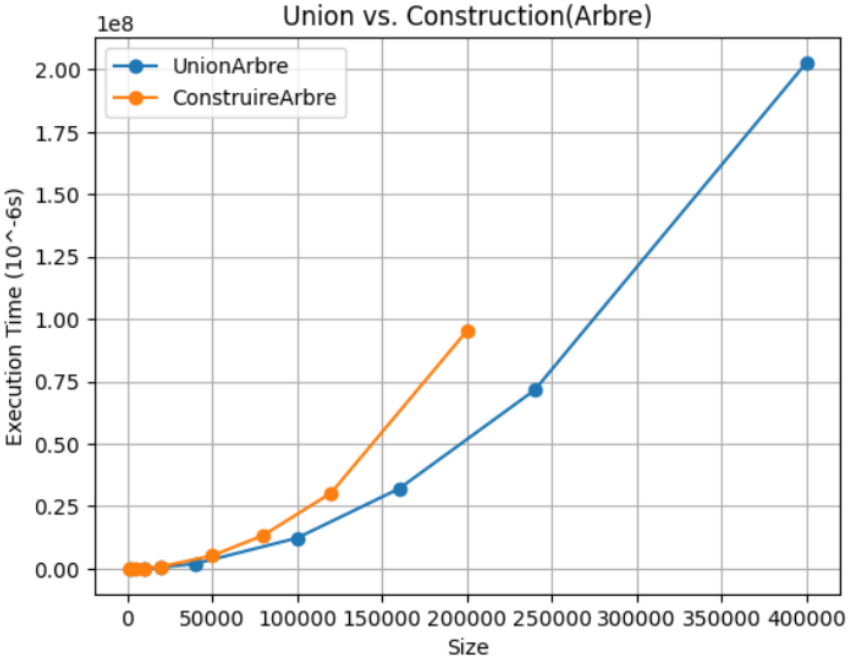
Pour la construction, la fonction **CreateABC** coûte O(n), car elle parcourt tout le tableau pour construire l'arbre binaire. Et la fonction **Aux** utilise cette fois le tableau des pointeurs **nodeRefs** mentionné dans la précédente partie, donc la complexité de la modification du tas devrait être identique que la structure avec tableau, qui est O(n). Ainsi, la complexité finale est O(n).

Pour Union, nous concatenons simplement les deux listes, puis appelle la procédure **Construction** pour réaliser l'union des clés des deux listes en un arbre binaire, donc la complexité finale doit être O(n), comme pour la construction.

2.6 Visualisation des complexités par les graphiques

La complexité de la construction devrait être O(n) d'après notre analyse précédemment, cependant, dans nos tests, la courbe ne semble pas être linéaire. Nous supposons que le problème provient de l'allocation de mémoire qui n'a pas pu être faite de manière optimale, ce qui entraîne une augmentation dans les temps d'exécution.





.

III. Structure 2 : FileBinomiale

1 Structure

```c
//struct arbre binomiale node
typedef struct ArbreBinomialeNode {
    Key128 data; // data of node
    struct ArbreBinomialeNode *child; // Pointe vers le nœud enfant le plus à gauche
    struct ArbreBinomialeNode *frere; // Pointe vers le nœud frère
    struct ArbreBinomialeNode *parent; // Pointe vers le nœud Parent
} ArbreBinomialeNode;
```

La structure de la file binomiale est basée sur notre définition dans TD.
Pour une file binomiale Bkn, elle a Bk0, Bk1, ... Bk(n-1) fils.

- Chaque nœud de l'arbre binomial (ArbreBinomialeNode) stocke les données de type key128, a un pointeur vers le fils le plus à gauche (child Bk0), un pointeur vers le prochain fils (frere Bk1), et un pointeur vers le parent (parent).
- Un tournoi (Tournoi) contient simplement un pointeur vers sa racine.
- La file binomiale (FileBinomiale) est composée d'un pointeur qui pointe d'une liste de tournois et d'un indicateur de taille (size) pour indiquer combien de tournois elle contient.

```c
//struct tournoi binomial
typedef struct Tournoi {
    ArbreBinomialeNode *racine; // pointe vers la racine
} Tournoi;

//struct file binomiale
typedef struct FileBinomiale{
    Tournoi ** file; //pointe vers un pointeur
    int size;
//    int index;??
}FileBinomiale;
```

2 Les focntions primitives mise en oeuvre

```
//primitive tournois
bool EstVide_T(Tournoi* t);
int Degre(Tournoi* t);
Tournoi* Union2Tid(Tournoi* t1,Tournoi* t2);
//Renvoie le tournoi de degre minimal dans la file
Tournoi* MinDeg(FileBinomiale* fb);
//primitive file binomiale
FileBinomiale* File(Tournoi* t);
bool EstVide_FB(FileBinomiale* fb);
```

```
FileBinomiale* Reste(FileBinomiale* fb);
FileBinomiale* AjoutMin(Tournoi* t,FileBinomiale* fb);
FileBinomiale* Ajout_FB(Tournoi* t,FileBinomiale* fb);
FileBinomiale* UnionFile(FileBinomiale* fb1,FileBinomiale* fb2);
FileBinomiale* UFret(FileBinomiale* fb1,FileBinomiale* fb2,Tournoi* t);
FileBinomiale* SupprMin_FB(FileBinomiale* fb);
FileBinomiale* Construction_FB(Tournoi** list_tournois,int size);
void freeArbreBinomialeNode(ArbreBinomialeNode *node);
void freeTournoi(Tournoi *t);
void freeFileBinomiale(FileBinomiale* fb);
//primitive supplementaire-copy
ArbreBinomialeNode* copyArbreBinomialeNode(ArbreBinomialeNode* node);
Tournoi* copyTournoi(Tournoi* tournoi);
//primitive supplementaire-print
void printFileBinomiale(FileBinomiale* fb);
void printArbreBinomialeNode(ArbreBinomialeNode *node);
```

L'implémentation de ces méthodes est basée sur notre cours.

3 Deux méthodes spéciales (copyArbreBinomialeNode, copyTournoi) - pourquoi et leur utilité

```
ArbreBinomialeNode* copyArbreBinomialeNode(ArbreBinomialeNode* node) {
    if (node == NULL) {
        return NULL;
    }
    ArbreBinomialeNode* newNode = malloc( size: sizeof(ArbreBinomialeNode));
    if (newNode == NULL) {
        // erreur memoire
        fprintf(stderr, "Error: Unable to allocate memory for ArbreBinomialeNode.\n");
        return NULL;
    }
    newNode->data = node->data;
    newNode->child = copyArbreBinomialeNode( node: node->child);
    newNode->frere = copyArbreBinomialeNode( node: node->frere);
    newNode->parent = NULL; // on set le pointeur dans copyTournoi,Sinon,
                            // cela pourrait etre une boucle récursive infinie.
    return newNode;
}
```

```c
Tournoi* copyTournoi(Tournoi* tournoi) {
    if (tournoi == NULL) {
        return NULL;
    }
    Tournoi* newTournoi = malloc( size: sizeof(Tournoi));
    newTournoi->racine = copyArbreBinomialeNode( node: tournoi->racine);
    // mise a jour le pointeur "parent"
    if (newTournoi->racine && newTournoi->racine->child) {
        newTournoi->racine->child->parent = newTournoi->racine;
    }
    return newTournoi;
}
```

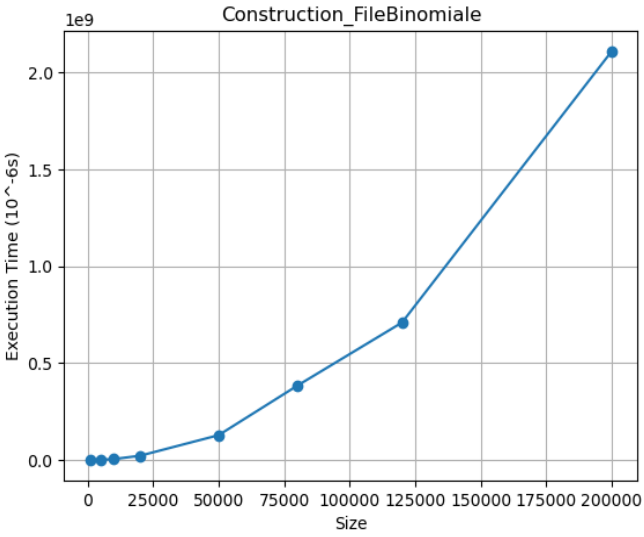
Dans notre code, tous les paramètres sont passés par référence (pointeurs) pour éviter les problèmes de mémoire et pour bien arranger les modification des files,tournois.

Cependant, cela peut entraîner un problème. Par exemple, lorsque nous construisons une file binomiale en utilisant un pointeur vers un tournoi(Primitive:File), si ce tournoi est modifié ultérieurement, alors les tournois dans cette file binomiale seront également modifiés en conséquence. Nous ne voulons pas que cela se produise, c'est pourquoi dans des fonctions similaires à celles de la File(Tournois t), nous devons créer une copie des paramètres sur laquelle nous baserons pour créer la file binomiale.

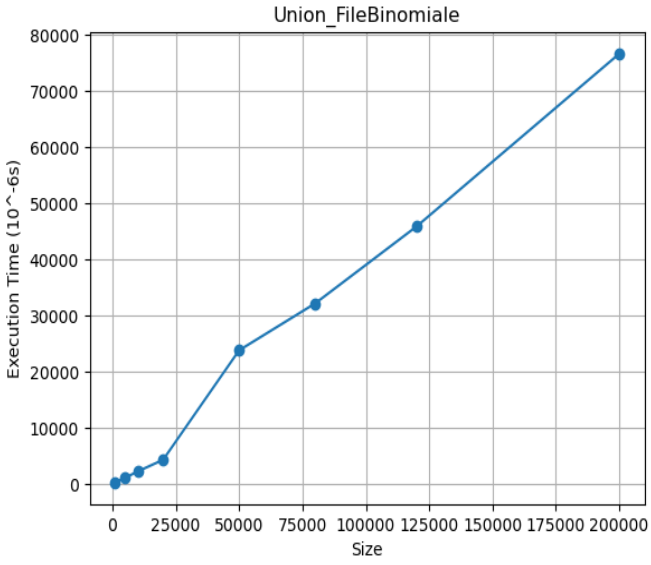
```c
FileBinomiale* File(Tournoi* T) {
    // creer un nouveau file binomiale vide
    FileBinomiale* newFb = malloc( size: sizeof(FileBinomiale));
    newFb->size = 1; // cette file n'a que un seul tournois
    newFb->file = malloc( size: sizeof(Tournoi*) * newFb->size);
    // copie T et attribue : newFb->file[0]
    Tournoi* copiedT = malloc( size: sizeof(Tournoi));
    if (copiedT == NULL) {
        fprintf(stderr, "Error: Unable to allocate memory for Tournoi.\n");
        free(newFb->file);
        free(newFb);
        return NULL;
    }
    copiedT->racine = copyArbreBinomialeNode( node: T->racine);
    newFb->file[0] = copiedT;
    return newFb;
}
```

4 Visualisation des complexités par les graphiques

- **Construction:**
  O(nlogn)



- **Union:**
  O(logn)



Ces deux courbes générées correspondent essentiellement à nos attentes en ce qui concerne la complexité temporelle de la construction (Construction : O(n log n)) et de l'union (Union : O(log n)), bien que pour l'union, cela puisse ne pas être très évident.
De plus, ils ont tous deux fonctionné avec succès avec 200 000 données sans rencontrer de problèmes de mémoire.

IV. Fonction de hachage :MD5

1 Structure

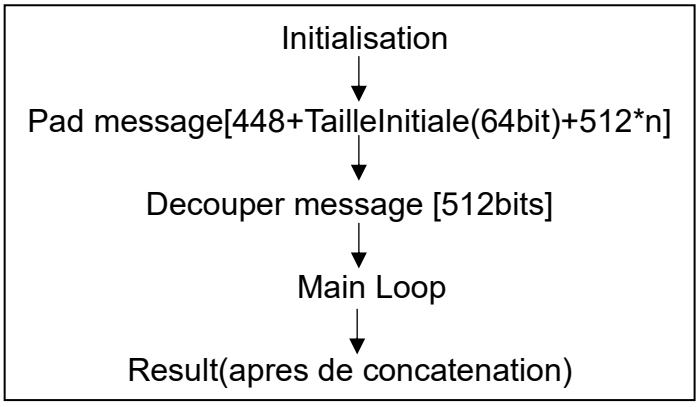
```c
typedef struct {
    uint64_t data_deja;//Taille du message traité
    uint32_t buffer[4];//Valeur de hachage actuelle
    uint8_t data_current[64];//Tampon d'entrée utilisé p
    uint8_t res[16];//Résultat de hachage
} MD5Context;
```

La structure pour stocker “L’ETAT” de Context de MD5
"data_deja" stocke les messages que nous avons déjà traités, "buffer" stocke la valeur de hachage actuelle, "data_current" stocke la valeur de hachage actuelle, et "res" stocke la valeur de hachage finale après traitement.

2 Les primitives fonctions

```c
//void md5_k_Init(uint32_t k[64]);
void md5_context_Init(MD5Context *ctx);
void md5_DecouperMessage(uint8_t *paddedMessage, size_t paddedMessageLen, MD5Context *ctx);
void md5_PadMessage(uint8_t *message, size_t messageLen, uint64_t messageLen_Debut);
void md5_MainLoop(MD5Context *ctx);
void md5_Final(MD5Context *ctx);
void md5(const uint8_t *message, size_t messageLen,uint8_t *res);
```

3 Les processus principal (tout est conforme au pseudo-code de la Wiki) :



- *`md5_context_Init(MD5Context *ctx)`* : Initialisation du struct MD5Context
- *`md5_DecouperMessage(uint8_t *paddedMessage, size_t paddedMessageLen, MD5Context *ctx)`* : Diviser le message en blocs de 512 bits
- *`md5_PadMessage(uint8_t *message, size_t messageLen, uint64_t messageLen_Debut)`* : Étendre l'entrée du message à 448 + taille initiale du message (64 bits) + 512 * n bits => 512 * n bits
- *`md5_MainLoop(MD5Context *ctx)`*: La boucle principale
- *`md5_Final(MD5Context *ctx)`* : Traiter la valeur de hachage finale
- *`md5(const uint8_t *message, size_t messageLen,uint8_t *res)`* : Intégrer l'ensemble des méthodes ci-dessus pour implémenter l'algorithme MD5. Lorsque on entre des messages, il stockera la valeur de hachage finale dans "res"

4 Comment on realise la code en little endian

nous implémentons la petite-endian (petit-boutiste) en effectuant une rotation vers la gauche du nombre de bits requis. Par exemple, dans la boucle principale, nous devons diviser un bloc de message de 512 bits en 16 petits blocs de 32 bits.

```
for (i = 0; i < 16; i++) {
    w[i] = (uint32_t) ctx->data_current[i * 4] |
           (uint32_t) ctx->data_current[i * 4 + 1] << 8 |
           (uint32_t) ctx->data_current[i * 4 + 2] << 16 |
           (uint32_t) ctx->data_current[i * 4 + 3] << 24;
}
```

Ici, dans cette étape, nous mettons en œuvre une boucle qui déplace le bit le plus significatif (MSB) vers la gauche par le biais de l'opérateur de décalage << du nombre requis de positions, puis le bit suivant en importance, puis le suivant, et ainsi de suite. Nous répétons continuellement cette opération, ce qui nous permet de coder en petite-endian (petit-boutiste).

V.Arbre de Recherche

1.Structure

Voici la structure de l'arbre de recherche que nous avons choisi d'implémenter en utilisant un arbre binaire. Cette structure est composée de trois valeurs :

```
typedef Key128 Type;
typedef struct ABR {
    Type data;
    struct ABR* fG;  // fils Gauche
    struct ABR* fD;  // fils Droit
} ABR;
```

- **data** : la clé (128 bits)
- **fg** : enfant gauche
- **fd** : enfant droit"

Voici les fonctions que nous avons définies pour l'arbre binaire de recherche.

` *CreateNode(Type data)*`: Pour créer un Arbre de Recherche Binaire
` *InsertNode(ABR* root, Type data)*` *:* Pour insérer un nœud
` *estDans(ABR * root,Type data)*` Pour v*érifier si une valeur (data) est présente dans l'Arbre*

VI. Étude expérimentale

1 Collision MD5

1.1 Idée principale

La méthode que nous avons mise en œuvre consiste à parcourir initialement tous les fichiers txt, puis à construire une liste contenant uniquement des mots uniques [23086 mots uniques au total] (méthode : nous utilisons un booléen pour indiquer si ce nouveau mot rencontré existe déjà dans la liste ; s'il existe, nous l'ignorons, sinon nous l'insérons dans la liste).

Pour étudier s'il y a des collisions parmi ces mots, nous appliquons d'abord l'algorithme MD5 à cette liste contenant uniquement des mots uniques. Nous obtenons ainsi une nouvelle liste contenant les valeurs de hachage de ces mots. Notre idée est d'utiliser la fonction predefini

"estdans" de notre structure ABR pour explorer si cette liste contient deux valeurs de hachage identiques. Si c'est le cas, comme nous savons que la liste avant la conversion ne contient que des mots uniques, cela signifie que ces deux valeurs de hachage identiques proviennent de deux mots différents, ce qui prouve l'existence d'une collision. Au final, nous avons trouvé 0 collision.

```
bool estDans(ABR *root, Type data) {
    if (root == NULL) {
        return false;
    }
    if (eg( key1: data,  key2: root->data)) {
        return true;
    } else if (inf( key1: data,  key2: root->data)) {
        return estDans( root: root->fG, data);
    } else {
        return estDans( root: root->fD, data);
    }
}
```

1.2 La collision

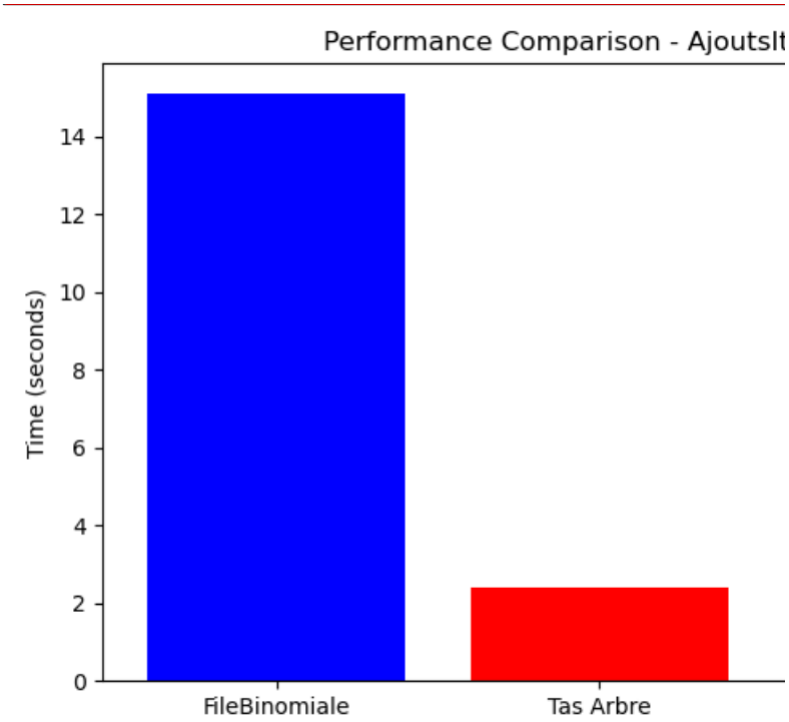
Le fait que nous ayons observé 0 collision avec l'algorithme de hachage MD5 est tout à fait normal compte tenu de nos données d'échantillonnage. MD5 présente une probabilité très faible de collisions, mais cette probabilité n'est pas nulle.

```
-----------Q6.2-----------
-----Q6.2no Collision, il y a 23086
```

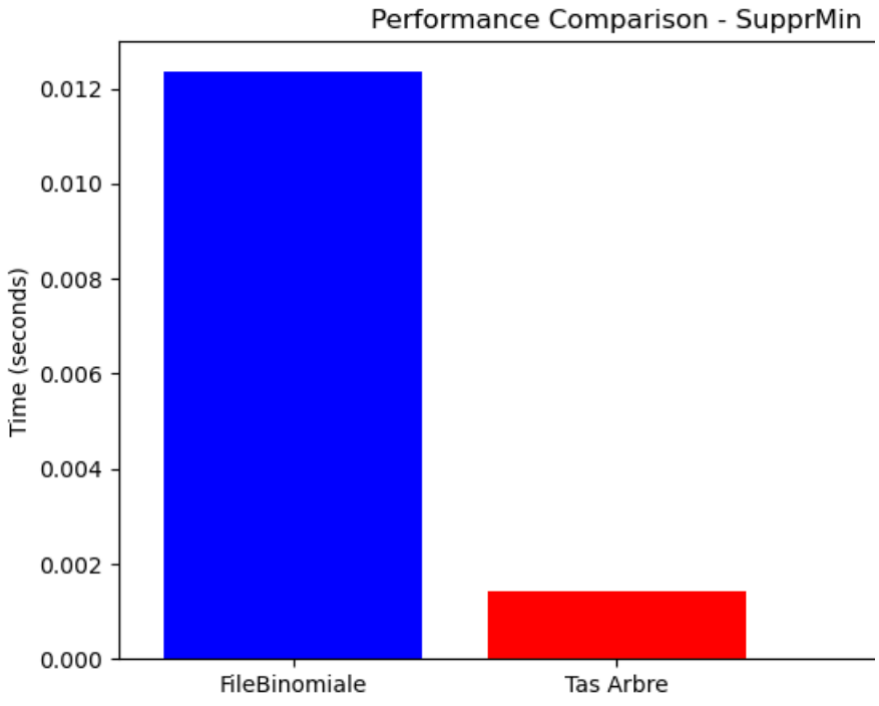
Cependant, si nous augmentions la quantité de données de notre échantillon, il serait possible d'observer des collisions avec l'algorithme MD5. C'est d'ailleurs l'une des raisons pour lesquelles MD5 est progressivement abandonné, ce qui le rend moins sécurisé pour certaines applications.

2 La comparaison des temps d'exécution entre la file binomiale et le tas

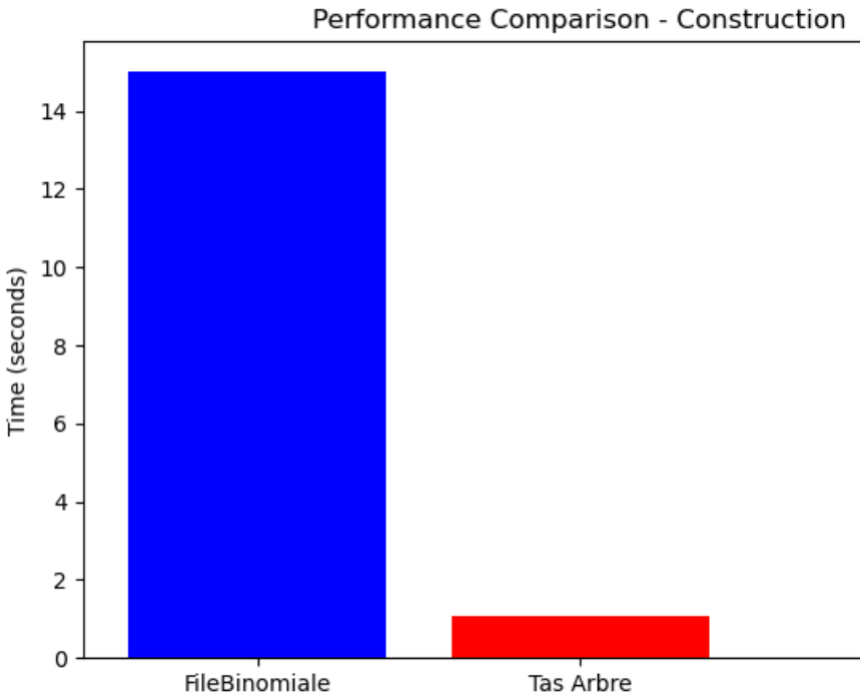
2.1 Ajout:



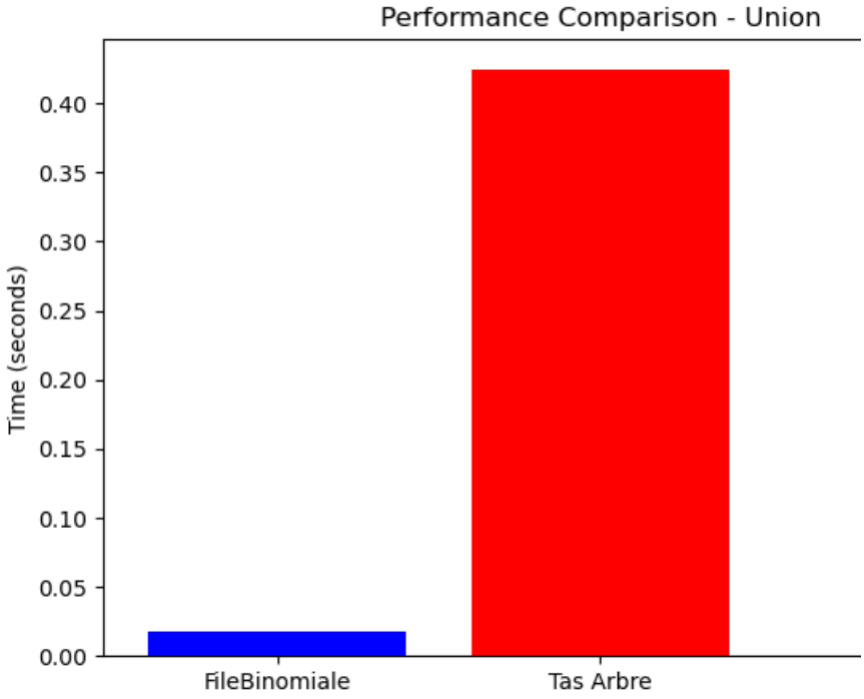
2.2 suppriMIN:



2.3 Construction:



2.4 Union :



3 conclusion

En fait, les résultats que nous avons observés correspondent dans une certaine mesure à nos attentes.

- **Union**

Par exemple, en ce qui concerne l'opération d'union, la file binomiale est plus efficace que le tas, car sa conception lui permet de gérer efficacement les opérations de fusion. La file binomiale est composée d'une série d'arbres binomiaux, chacun suivant des règles spécifiques. Fusionner deux files binomiales ne nécessite que la comparaison et la liaison des listes de nœuds racines, ce qui peut être réalisé avec une complexité temporelle relativement basse.

En général, pour les opérations d'ajout, de suppression du minimum (SupprMin) et de construction (Construction), la structure de tas (Tas) est généralement plus efficace que celle de la file binomiale (FileBinomiale).

- **SupprMin**

Dans un Tas, la suppression du minimum implique généralement de déplacer le dernier élément du tas à la position racine, puis d'effectuer une opération de descente pour maintenir la propriété du tas. La complexité temporelle de cette opération est en O(log n). En revanche, dans une FB, l'opération SupprMin peut être plus complexe car elle nécessite de trouver l'arbre binaire avec la racine la plus petite parmi plusieurs arbres binomiaux, puis de réorganiser le reste du tas. Ce processus peut impliquer plus d'étapes.

- **Ajoute(Ajout d'éléments)**

L'ajout d'éléments dans un Tas est généralement simple : ajouter le nouvel élément à la fin du tas, puis effectuer une opération de montée. Cette opération a également une complexité temporelle de O(log n). Dans une FB, l'ajout d'un nouvel élément peut nécessiter la création d'un nouvel arbre binaire, suivi de sa fusion avec le tas existant, ce qui peut impliquer davantage d'opérations de fusion et de réorganisation.

- **Construction**

La construction d'un Tas peut être effectuée de manière efficace, par exemple en utilisant l'algorithme de construction de tas de Floyd, avec une complexité temporelle de O(n). En revanche, la construction d'une FB peut nécessiter davantage d'étapes, en particulier lorsque de nombreux éléments doivent être intégrés dans différents arbres binomiaux. Par conséquent, pour ces opérations spécifiques, un tas binaire (Tas) a généralement de meilleures performances que la file binomiale (FB), principalement en raison de la simplicité de sa structure de données et de la directivité de ses opérations.