SORBONNE UNIVERSITÉ

UE D'ALGAV

# Rapport de Projet ALGAV


Enseignants:

Antoine GENITRINI

Ariane ZHANG

Xue YANG


M1 INFO-STL 2023 - 2024

# 1 Présentation

Le langage de programmation choisi est c, car nous avons l'expérience de manipuler les arbres dans ce langage.

# 2 Clé à 128 bits

## Structure de la clé

La clé de 128 bits est composée de 4 entiers de 32 bits formant un tableau. Voici la définition du type.

```
typedef struct{
    // Tableau de 4 entiers de 32 bits
    uint32_t parts[4];
} uint128_t;
```

Nous utilisons les fonctions *inf* et *eg* pour comparer les valeurs des clés et vérifier leur égalité

## Structure de la liste de clé

Une liste de clés est implémentée car elle est utile pour la suite du projet.

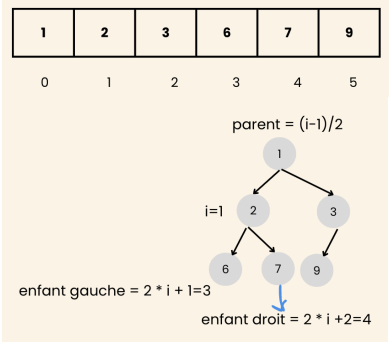
```
typedef struct Element Element;
struct Element{
    uint128_t cle;
    Element *suivant;
};

typedef struct ListeCle ListeCle;
struct ListeCle{
    Element *premier;
};
```

# 3 Tas min avec la structure tableau

## Structure

```c
//  Structure de tas minimum avec un arbre binaire
typedef struct Noeud{
    uint128_t cle;
    struct Noeud *gauche, *droite;
} Noeud;
// Structure de tas minimum avec un tableau
typedef struct{
    uint128_t *cles; // Tableau des clés
    int taille;      // Nombre d'éléments dans le tas
    int capacite;    // Capacité du tableau
} Tas;
```




## Pseudo code de construction

La fonction construction prend un tableau de clés et les organise dans une structure de tas. Elle utilise la procédure **`SiftDown`** pour assurer que le tas respecte la condition de l'ordre croissant du tas à partir de racine jusqu'à les nœuds.

```
construction(tas, listeCle, nbCles):
   tas.taille <- nbCles
    pour i de (nbCles - 2) / 2 jusqu'à 0 faire
        SiftDown(tas, i)
    retourner tas
```

## Pseudo code de union

La fonction UnionTas combine deux tas en augmentant la taille du premier si nécessaire, copie les éléments du second dans le premier, et réorganise le nouveau tas pour respecter les propriétés d'un tas.

```
UnionTas(sortie tas1, tas2):
    tailleUnion <- tas1->taille + tas2->taille
    nouvellesCles <- redimensionnerMemoire(tas1->cles, tas1->capacite *
tailleDe(uint128_t))
    tas1->cles <- nouvellesCles
    tas1->taille <- tailleUnion
    pour i de (tailleUnion - 2) / 2 décrément jusqu'à 0 faire
        Descendre(tas1, i)
    retourner tas1
```

## Complexité des fonctions

La fonction “ajout” ajoute une clé à la fin du tableau et effectue une remonté si nécessaire en échangeant la clé courant à son parent. La remonté peut faire au plus h échanger où h est la hauteur. Donc elle est de complexité O(log(n)).

La suppression de la clé minimum supprime la clé de la racine et la remplace par la clé positionné à la fin du tableau, puis effectue une descente si nécessaire en échangeant la clé courant à l'un de ces fils. L'algorithme effectue au plus h échanges. Ainsi on trouve O(log(n)).

L'ajout itératif effectue n ajouts individuels, chacun avec une complexité de O(log n), ce qui donne une complexité totale de O(n log n).

La construction ajoute n clés dans un tas et effectue des descentes non naïve. Ce qui donne une complexité O(n).

La fonction Union fusionne deux tas en ajoutant les éléments du second à la suite du premier, puis effectue un tri par descente à partir du dernier parent pour rétablir la propriété de tas min. La complexité de cette opération est O(m), où m est la taille totale du tas résultant.

## Analyse

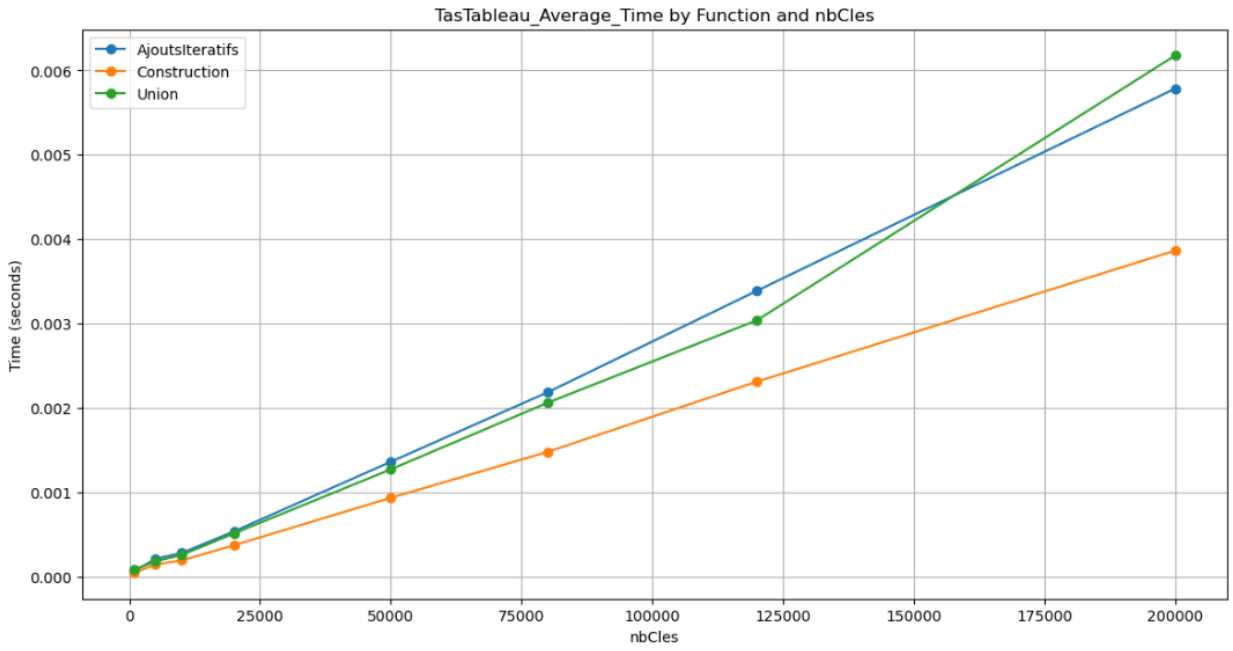
Le graphe ci-dessus illustre les courbes du temps d'exécution moyen des méthodes ajoutItératif, construction et union en fonction du nombre de clés. Et puis, les données utilisées proviennent du jeu de données fourni dans l'énoncé.

La courbe "Union" représente l'opération d'union de deux tas ayant des clés différentes mais de taille identique.

D’après le graphe, la fonction construction et union semblent linéaires, ce qui concorde avec l’explication précédente. Cependant, la courbe d'ajout itératif présente

également une linéarité. Cela peut s'expliquer par le nombre d'éléments ajoutés. Si chaque ajout nécessite un temps constant ou une opération en O(1), c'est à dire sans remonter, le temps total d'ajout pour n éléments serait proportionnel à n, conduisant à une complexité linéaire.

Il est également notable que la construction est plus rapide que l'ajout itératif, ce qui est cohérent.




# 4 Tas min avec la structure arbre

## Structure

```c
//  Structure de tas minimum avec un arbre binaire
typedef struct TasArbre{
    uint128_t cle; // Clé de l'élément
    struct TasArbre *gauche;
    struct TasArbre *droite;
} TasArbre;
```

## Pseudo code de construction

```
construction(out tas,listeCle):
      ajout simple de tout les clés le tas
      appliquer la fonction descendre pour chaque noeud de hauteur
      h-1, puis h-2 jusqu'à la racine
```

## Pseudo code de union

```
union(tas1,tas2):
      tant que tas1 ou tas2 non vide
            compare la racine tas1 et tas2
            ajout simple du minimum entre les racines des deux tas dans
un tas res
            supprime la racine du tas initial (tas1 ou tas2)
      tantque le tas restant (tas1 ou tas2) non vide
            ajout simple la racines au tas res
            supprime la racine du tas restant (tas1 ou tas2)
      retourner tas res
```

## Complexité des fonction

Les complexité du tas min via la structure tableau ne sont pas similaires à ceux du tableau, car la recherche de d'un parent d'un nœud, la recherche du dernier nœud ou bien la dernière place libre pour insérer une clé coûte plus cher. Les complexité du tas min via la structure arbre sont très mauvaise. Nous n'avons pas pu optimiser ces méthodes.

**ajout** : Dans la structure d'arbre utilisée, la recherche de la dernière place libre peut nécessiter un parcours de l'arbre, ce qui contribue à une complexité de O(n). Après avoir identifié cet emplacement, l'ajout de l'élément suivi de l'opération de remontée implique une nouvelle recherche du parent, également avec une complexité de O(n). En somme, cela aboutit à une complexité totale de O(2n), qui est équivalente à O(n) dans le contexte de la notation big-O, où les termes constants sont négligés.

**Sup min**: La fonction vise à supprimer la clé à la racine du tas et la remplacer par le nœud situé le plus bas et le plus à gauche possible. La recherche de ce nœud peut avoir une complexité au pire cas de O(n). Après avoir trouvé et remplacé la valeur de la racine, il est nécessaire de la faire descendre si elle est supérieure à ses fils. Cette opération de descente a une complexité de O(log(n)) dans le pire cas, étant donné la nature des tas. Ainsi, en combinant ces deux opérations, la complexité totale de la fonction suppMin est de O(n + log(n)), mais dans l'analyse asymptotique, on retient la complexité dominante, ce qui équivaut à O(n).

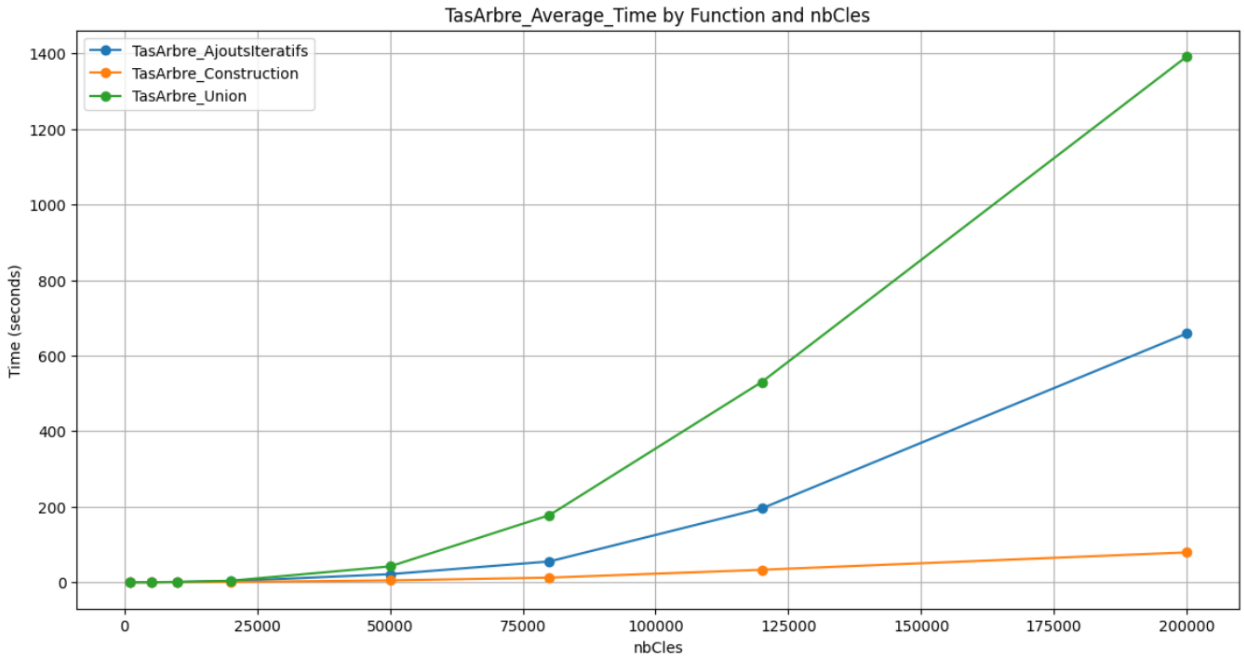
**ajout itératif** : Cette méthode appelle la fonction ajout, dont la complexité est O(n). Par conséquent, la complexité totale est $O(n^2)$

**construction**: La fonction vise à créer un tas à partir d'une liste de clés. Tout d'abord, elle parcourt la liste, ajoutant au tas sans prendre en compte de l'ordre du tas. Cette première étape a une complexité linéaire O(n). Puis elle fait des descentes si nécessaire de chaque nœud de hauteur h-1, h-2 jusqu'à la racine. La recherche des ensembles de nœuds du même niveau est de l'ordre de O(n).  Cette opération est répétée h-1 fois, ce qui est généralement proche de O(log(n)). Ainsi, la complexité totale est O(nlog(n)) + O(n),, ce qui se simplifie à O(nlog(n)).

**union**: L'ajout simple de chaque clé des deux tas coûte O(n)+O(m) ou n est la taille du tas1 et m la taille du tas. La suppression du minimum du tas1 coûte O(log(k)) et répète n fois avec k qui décrémente de 1 à chaque tour,  la suppression duminimum du tas2 coûte O(log(l)) et répète n fois avec l qui décrémente de 1 à chaque tour. La complexité totale est donc O(n+m)+nO(log(k))+mO(log(l)).

## Analyse de graphe

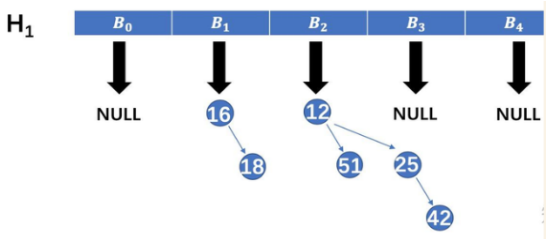
Le graphe ci-dessous représente trois courbes. Les courbes de ajout itératif et union représente le temps d'exécution d'un seul jeu de donné par nombre de clé car l'implémentation de la structure est très lente. La courbe de la construction représente la moyenne d'un ensemble de jeux de données par nombre de clés. On voit bien que la construction est beaucoup plus rapide que l'ajout itératif.




# 5 File Binomial

## Structure

```
// Binomial Tree Node
struct Node {
    uint128_t cle;
     int degree;
    struct Node *parent;
    struct Node *enfant;
    struct Node *frere;
};
typedef struct Node * BinNode;
// Binomial Heap
struct Heap {
    int size;
    int capacity;
    BinNode *list;
};
typedef struct Heap * BinHeap;
typedef struct Node * BinTree;
```




## Les fonctions principales

*fb_union* réalise l'union de deux files binomiales en combinant leurs arbres de même degré et en ajustant la taille du tas résultant.La complexité de cette fonction est O(log n), car elle doit potentiellement traverser et fusionner les arbres de tous les degrés jusqu'au degré maximal, qui est lié au logarithme de la taille du tas.

*ajout* insère une nouvelle clé dans un files binomiales en créant un nouvel arbre binomial et en l'unissant au tas existant. La complexité de l'ajout est O(log n), car l'insertion peut entraîner une série de fusions d'arbres binomiaux, chacune prenant un temps proportionnel au nombre de degrés dans le tas.

*fb_SupprMin* retire l'élément avec la clé minimum du tas binomial, en restructurant le tas pour maintenir ses propriétés.La complexité de la suppression du minimum est O(log n). Bien que trouver l'élément minimum soit en O(1), la restructuration après suppression nécessite un parcours et une possible fusion des arbres, ce qui se fait en temps logarithmique par rapport à la taille du tas.

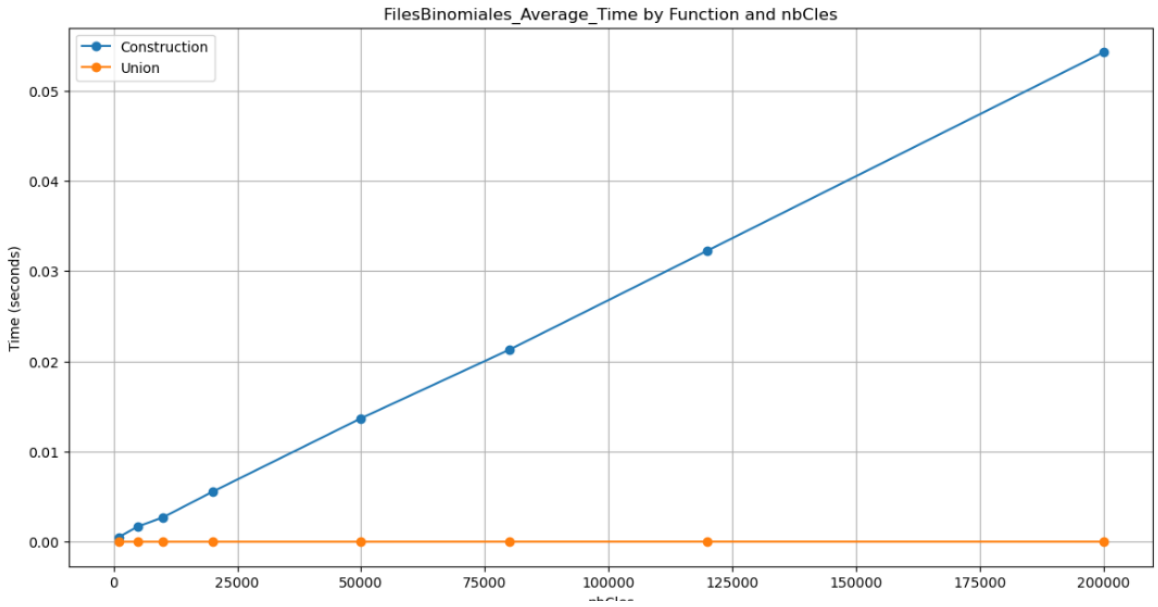
*fb_Construction* construit un tas binomial à partir d'un tableau de clés en insérant successivement chaque clé.La complexité globale de la construction est O(n log n), car chaque insertion a une complexité de O(log n) et il y a n insertions.

## Analyse

*fb_Construction* construit une file binomiale en ajoutant des éléments de manière itérative. Théoriquement, ce processus devrait avoir une complexité temporelle de O(n log n), car chaque insertion pourrait impliquer une opération de fusion de O(log n). Cependant, la croissance linéaire du temps observée dans les expériences indique que les performances réelles surpassent les attentes théoriques. Cela peut être dû au fait que, dans la pratique, chaque insertion n'entraîne pas systématiquement une fusion. En raison de la structure de la file binomiale, les fusions d'arbres binomiaux de petits degrés sont très rapides et, si la distribution des données le permet, de nombreuses insertions peuvent ne pas conduire à des fusions entre arbres, réduisant ainsi le

nombre total de fusions. De plus, des optimisations dans l'implémentation, telles que le stockage en cache des opérations de fusion, peuvent également contribuer à améliorer l'efficacité. Par conséquent, même si le pire cas d'insertion individuelle est O(log n), la complexité moyenne du processus de construction est proche de O(n), ce qui explique pourquoi vos tests de performance montrent une complexité temporelle croissante de manière linéaire.

*fb_union* effectue la fusion de deux files binomiales. Théoriquement, ce processus devrait avoir une complexité temporelle de O(log n), car il nécessite de parcourir les arbres binomiaux dans la file et de fusionner ceux ayant le même degré. Cependant, les résultats des tests montrent que le temps de l'opération Union ne varie presque pas avec l'augmentation du nombre d'éléments, ce qui suggère que les opérations de fusion réelles peuvent être bien plus efficaces que ce que prévoit le modèle théorique. Des optimisations pendant le processus de fusion, comme par exemple lorsque les deux files ont des tailles très différentes, peuvent réduire le nombre d'étapes de fusion nécessaires, ou en raison d'une distribution particulière des données, les opérations de fusion peuvent être plus simples que prévu. De plus, les coûts fixes dans l'opération de fusion, tels que l'allocation de mémoire, peuvent dominer l'ensemble du processus et donner lieu à une complexité temporelle apparaissant constante.



# 6 Fonction de hachage

## Pseudo code

```
md5(message):

      Conversion du message en tableau d'octets

      Padding(remplissage de bit: bit 1 puis bit 0 jusqu'à ce que le
      message modulo 512 est égal 448 et taille du message initial en 64
      bits Little Endian)

      Découpage en blocs de 512 bits

      Subdivision de chaque bloc en 16 mots de 32 bits en Little Endian

      Boucle principal qui fait les opérations
```

## Exemple d'utilisation

On a un exemple d'utilisation sur la chaîne de caractère "Hello World!". Voici le résultat de chaque étape.

```
Convertion : 48 65 6c 6c 6f 20 57 6f 72 6c 64 21
Padding : 48 65 6c 6c 6f 20 57 6f 72 6c 64 21 80 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 60 00 00
 00 00 00 00 00
Decoupage :
Bloc 0 : 48 65 6c 6c 6f 20 57 6f 72 6c 64 21 80 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 60 00 00 00 00 00 00 00
 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 60 00 00 00 00 00 00 00
 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 60 00 00 00 00 00 00 00
Mots :
w[ 0] = 6c6c6548
w[ 1] = 6f57206f
w[ 2] = 21646c72
w[ 3] = 00000080
w[ 4] = 00000000
w[ 5] = 00000000
w[ 6] = 00000000
w[ 7] = 00000000
w[ 8] = 00000000
w[ 9] = 00000000
w[10] = 00000000
w[11] = 00000000
w[12] = 00000000
w[13] = 00000000
w[14] = 00000060
w[15] = 00000000
md5 : 876207ed36862e53921e845e8c0dc5bf
```

Chaque étape semble être correcte, mais le résultat  trouvé est différent de la bonne réponse : ed076287532e86365e841e92bfc50d8c.

# 7 Arbre de recherche

## Structure

L’arbre binaire de recherche ou la clé du nœud courant est supérieure à son fils gauche et inférieur à son fils droit est implémenté.

```
typedef struct ABR{
    uint128_t cle;
    struct ABR *gauche;
    struct ABR *droite;
}ABR;
```

## Recherche de mot unique

Pour générer la liste des mots uniques du livre de Shakespeare, une approche consiste à utiliser un arbre binaire de recherche (ABR). Initialement, l'ABR est vide. En parcourant la liste de mots du livre, chaque mot est recherché dans l'ABR. Si le mot n'est pas présent, il est ajouté à l'arbre ; sinon, on passe au mot suivant. À la fin du processus, l'arbre contient l'ensemble des mots distincts du livre de Shakespeare.

# 8 Organisation des fichiers

Les fichiers associés à la manipulation d'une structure de données sont organisés en quatre catégories distinctes. Tout d'abord, il y a le fichier qui contient la définition de la structure. Ensuite, il y a le fichier qui implémente les fonctionnalités de cette structure. Un autre type de fichier est dédié aux tests des fonctionnalités de la

structure, où les opérations sont exécutées, et les résultats ou éventuelles erreurs sont affichés dans le terminal. Enfin, il y a des fichiers qui évaluent les performances des fonctions implémentées en utilisant des jeux de données et génèrent des fichiers csv.

Pour pouvoir reproduire les tests, il suffit de suivre les commandes de la colonne make du tableau suivant.

| **En tête** | **fonctionnalité** | **test** | **calcul performance** | **csv** | **make** |
|---|---|---|---|---|---|
| cle.h | cle.c | testCle.c | | | make testCle |
| liste.h | liste.c | | | | |
| tasTableau.h | tasTableau.c | testTasTableau.c | calculPerformanceTasTableau.c | performance_tas_tableau.csv | make testTasTableau |
| tasArbre.h | tasArbre.c | testTasArbre.c | calculPerformanceTasArbreAjoutIt.c<br>calculPerformanceTasArbreCons.c<br>calculPerformanceTasArbreUnion.c | performance_tas_arbre_AjoutIt.csv<br>performance_tas_arbre_cons.csv<br>performance_tas_arbre_Union.csv | make testTasArbre |
| filesBinomiales.h | filesBinomiales.c | testfilesBinomiales.c | calculPerformanceFB.c | performance_files_binomiales.csv | make testFilesBinomiales |
| arb.h | abr.c | testAbr.c | | | make testAbr |
| md5.h | md5.c | testMd5.c | | | make testMd5 |