I) Conteneurs associatifs

1) Définition de principales commandes pour chaque conteneur pour quelques éléments

Les conteneurs associatifs contiennent 4 types de conteneur différents qui sont :

- Les maps
- Les multimaps
- Les sets
- Les multisets

Cette famille de container a la particularité de représenter des données sous la forme d'arbre ordonnés constamment trié. En effet à chaque insertion d'élément celui-ci va automatiquement arrangé ses éléments pour être trié. Il est à noter que l'arrangement se fait par comparaison des valeurs comme par exemple dans l'ordre croissant pour les entiers, ordre alphabétique pour les string… Cependant il est tout à fait possible de donner en argument au constructeur comment comparer les valeurs.

Par exemple avec une fonction qui peut permettre de trier des entiers par ordre décroissant avec la fonction :

```
struct classcomp {
  bool operator() (const int& lhs, const int& rhs) const
  {return lhs>rhs;}
};
```

Et en exécutant le code suivant :

```
int myints[]= {100,20,300,40,50,18,56};
std::set<int,classcomp> fifth (myints,myints+8);                    // class
as Compare
```

On obtient le set suivant :

```
300,100,56,50,40,20,18
```

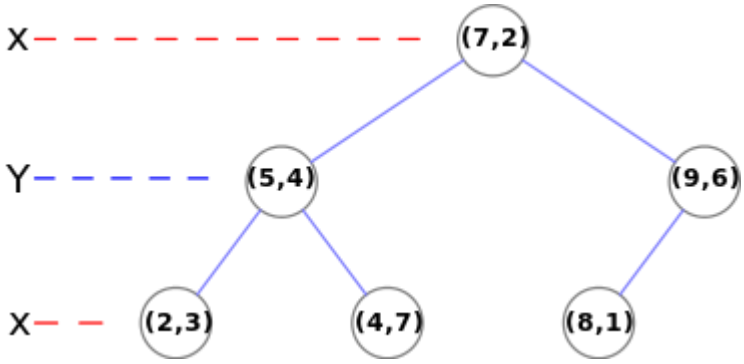
Pour tous les tests effectués, ce sera le tri par default des entiers qui sera utilisé (par ordre croissant).

Les associatives container possèdent des itérateurs bidirectionnels de 4 types :

- `Begin(), end()` : parcourt le container dans le sens croissant.
- `Rbegin(), Rend()` : parcourt le container dans le sens inverse(décroissant).
- `CBegin(), Cend()` : parcourt le container dans le sens croissant avec le paramètre const qui permet d'éviter de modifier et d'éviter des copies qui peuvent être lourde.
- `Crbegin(), Crend()` : parcourt le container dans le sens inverse(décroissant) avec le paramètre const qui permet d'éviter de modifier et d'éviter des copies qui peuvent être lourde.

1) LES MAPS

Les maps sont des collections ordonnées ou chaque élément correspond à un couple (clé, valeurs). En effet ces couples de valeurs sont ordonnés selon les clés qui sont généralement assimilé à des identifiant sachant qu'il ne peut y avoir de doublons.



La représentation des arbres est faite tel que pour tout nœud « i » :

- Les clés du sous arbre gauche sont strictement inferieur à la clé de « i »
- Et les clés du sous arbre droite sont strictement supérieur à la clé de « i »

Les maps possèdent quelques fonctions propres utiles à leurs utilisations par exemple l'operateur [] qui prend en argument une clé et renvoie la référence de la valeur associée à celle-ci. Si la clé n'est pas présente l'élément est créé.

```
std::map<int,int> first_map;
//insert les cases (1,10) et (2, 30)
first_map[1]=10;
first_map[2]=30;
```

Les conteneurs map possèdent aussi la fonction `at` qui prend en argument la clé et renvoie la référence de la valeur associée à celle-ci.

```
//modifie les cases
first_map.at[1]=70;
first_map.at[2]=50;
```

Pour insérer des éléments dans la map il y a aussi la fonction `insert` qui prend en argument une paire (clé, valeur) et l'ajoute dans la map. Si la clé est déjà présente la fonction ne fera rien du fait que les doublons ne sont pas autorisés.

```
std::map<int,int> first_map2;
//insert les cases (1,10) et (2, 30)
first_map2.insert(std::pair<int,int>(1,10));
first_map2.insert(std::pair<int,int>(2,30));
```

Pour construire une map de taille N il est nécessaire de faire une boucle for et d'utiliser soit [] ou insert.

```
std::map<int,int>test;
for (int i=1; i<=N; i++) test.insert(std::pair<int,int>(i,i));
```

A l'aide des constructeurs il est possible de réaliser une copie d'une map vers une autre map :

```
//on crée la map test2 identique à la map test
std::map<int,int> test2 (test.begin(),test.end());
```

Pour lire les conteneurs nous allons utiliser des « based range loop » avec l'utilisation des méthodes .first et .second qui permettent de sélectionner respectivement la clé et la valeur associé.

```
for(auto const&value: map)
   std::cout<<"("<< value.first << " associé " << value.second <<")"<<" ";
std::cout<<std::endl;
```

Il existe une fonction qui permet de supprimer un élément de la map qui est `erase` et qui prend en argument la clé de la case.

```
test.erase(n);
```

De même il y a une fonction `clear` qui supprime tous les éléments d'une map

```
test.clear();
```

Une fonction qui renvoie la taille de la map, c'est-à-dire le nombre d'élément qu'elle possède.

```
test.size();
```

2) LES MULTIMAPS

Les multimaps sont très semblables au map à la différence que les clés doubles sont autorisées, il peut donc y avoir plusieurs cases avec des clés semblables.

Du fait de la possibilité de clé multiple l'opérateur [] ainsi que la méthode .at n'existe plus. Il est donc nécessaire d'utiliser la fonction insert pour insérer des éléments.

```
std::multimap<int,int>test;
```

```
for (int i=1; i<=N; i++) test.insert(std::pair<int,int>(i,i));
```

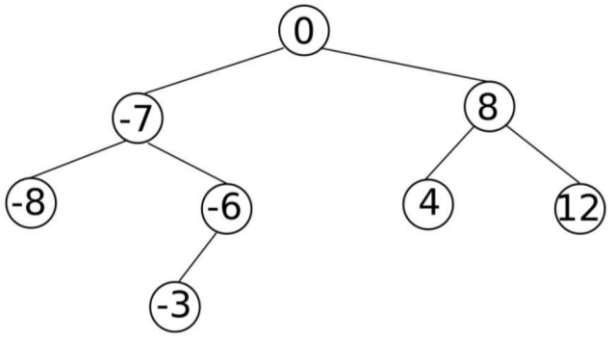
De même il est nécessaire d'utiliser les itérateurs pour lire dans la multimap.

Les fonctions erase, clear, size existent toujours pour les multisets, cependant pour la fonction erase il est possible que la clé donnée en argument désigne plusieurs cases, cela a pour conséquence de supprimer plusieurs cases.

Il est à savoir que si la clé est présente plusieurs fois tous les éléments associés avec celle-ci seront supprimé.

3) LES SETS

L'ensemble des sets peut être assimilés à une map mais avec une valeur associé égale à sa clé (on a donc un seul élément)



.

Tout comme les maps, les sets n'acceptent pas les doublons, de plus n'ayant qu'une seule valeur les operateurs [] et at n'existe pas non plus, on utilise donc des `insert` ainsi que des itérateurs pour la lecture.

```
std::set<int>test;
for (int i=1; i<=N; i++) test.insert(i));
```

Les fonctions `erase`, `clear`, size les copie ont exactement la même application que pour les map.

4) LES MULTISETS

Les ensembles multiset sont, analogiquement aux multimap, des set qui acceptent les doublons et sont construits avec des `insert` comme précédemment.

```
std::multiset<int>test;
for (int i=1; i<=N; i++) test.insert(i));
```

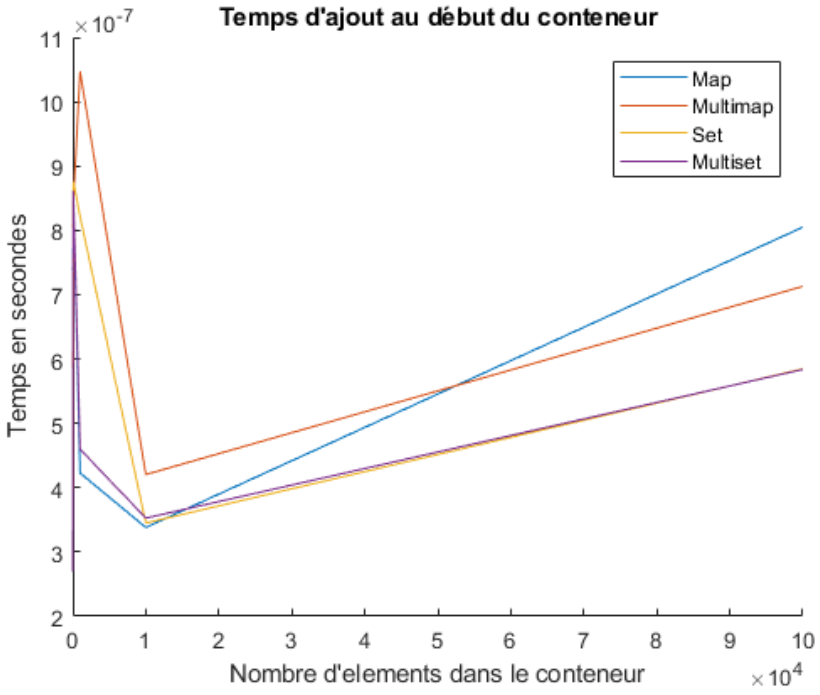
Les fonctions `erase`, `clear`, size les copie ont exactement la même application que pour les multimap.

2) Mesures des temps d'exécution pour chaque conteneur de séquence

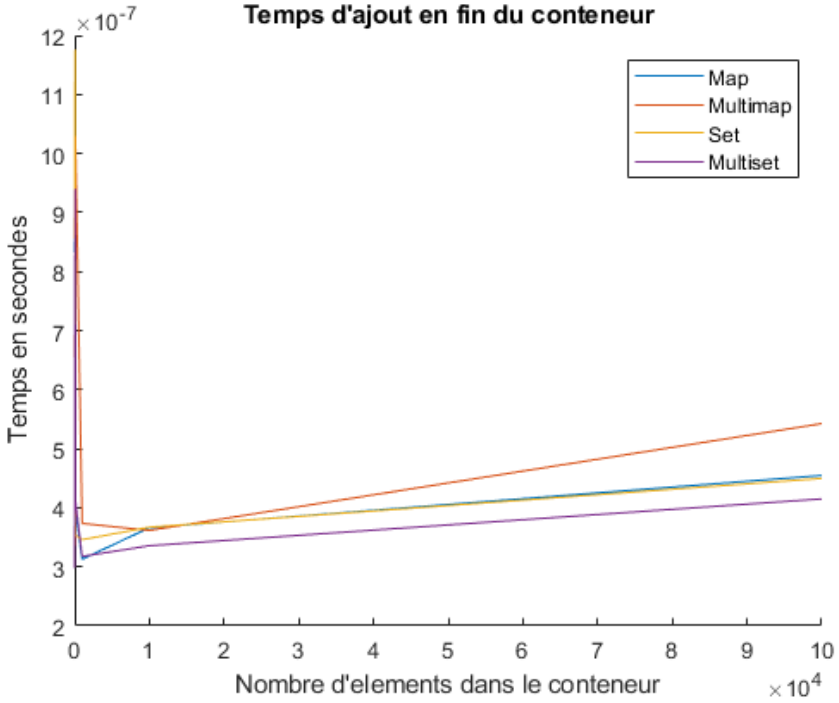
Avec le code de la partie 1 on peut réaliser les courbes d'exécution des différentes fonctions afin de comparer le comportement des différents conteneurs.

Pour différentes insertions d'éléments, il est intéressant de tracer les courbes de temps d'exécution.

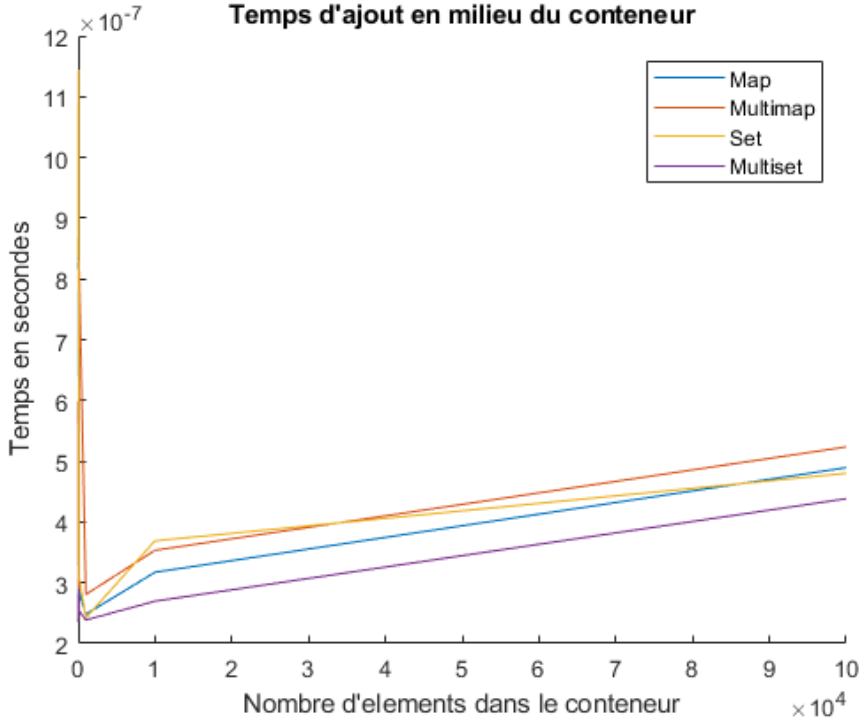
En insérant une petite valeur :



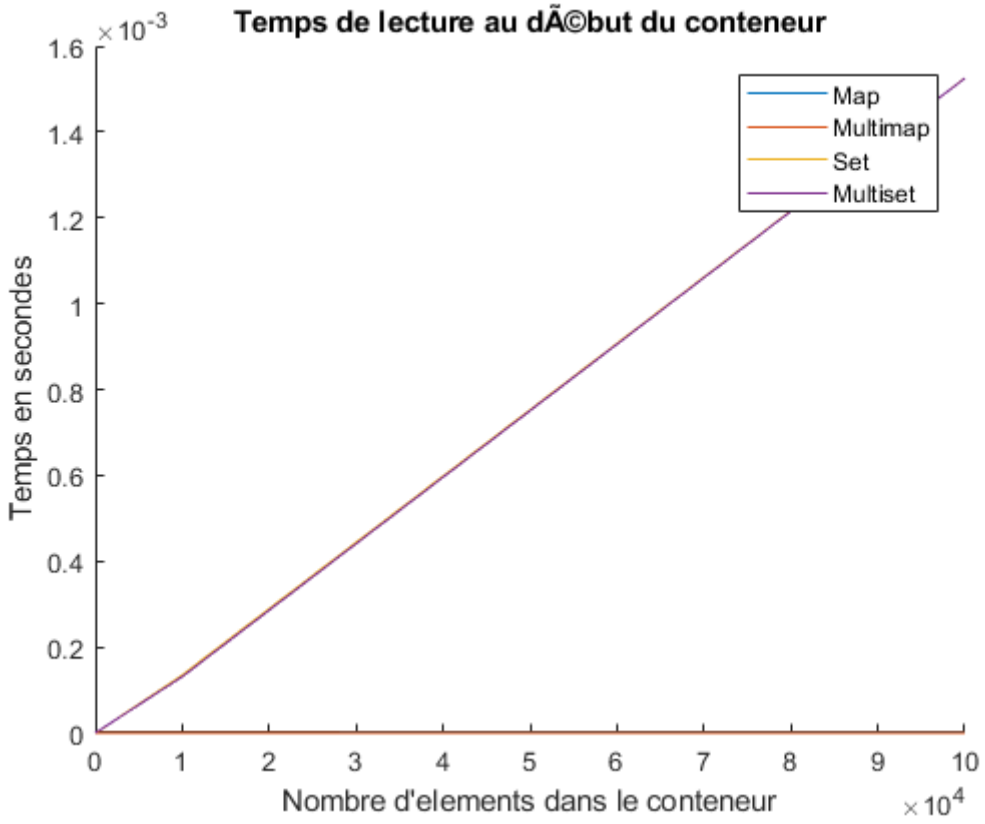
En insérant une grande valeur :



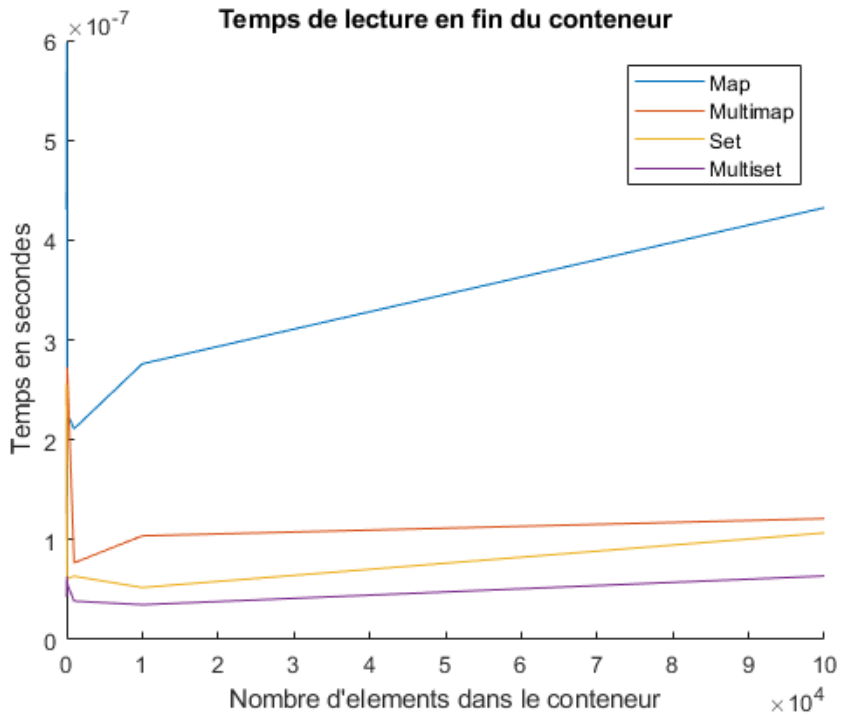
En insérant une valeur moyenne :



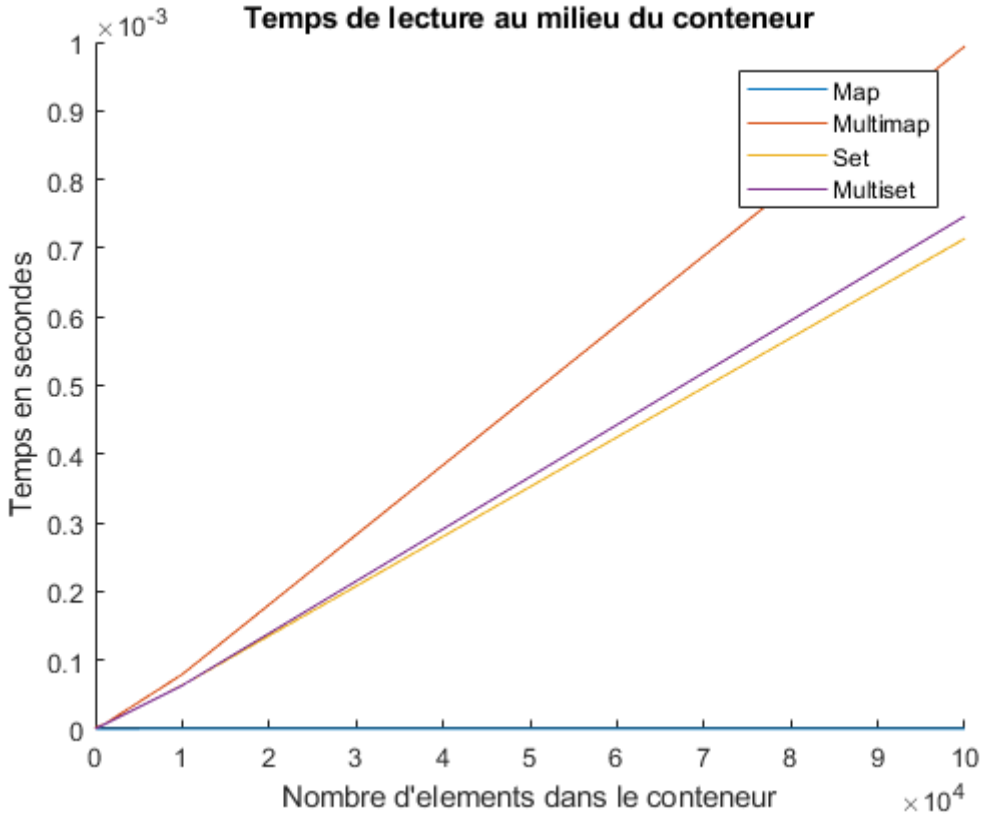
L'étude du temps d'ajout montre que le fonction `insert` utilisé pour les 4 container est relativement équivalente. De plus bien qu'il peut sembler qu'il y a une complexité en O(N) il y a en réalité une complexité en O(log(N)) car l'élément insérer est inséré dans l'arbre.



Il semble que pour la lecture en fin des conteneurs set et multiset soit en O(n) et en O(1) pour les maps et multimap. En effet le premier élément du conteneur correspond à l'utilisation de begin `()` sur l'itérateur associé ce qui explique la complexité en O(1).



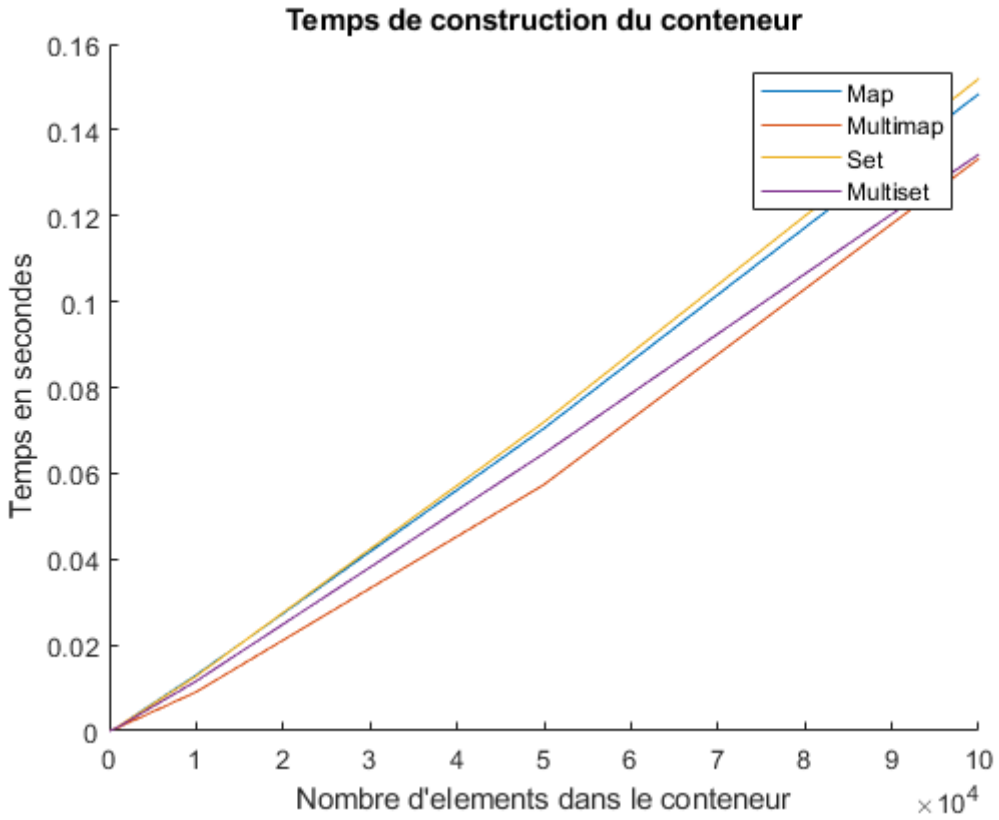
Il semble que pour la lecture en fin du conteneur map est moins rapide bien qu'elles soient toute en O(1). En effet le premier élément du conteneur correspond à l'utilisation de `.rbegin()` sur l'itérateur associé, qui est un peu plus rapide que l'operateur [].



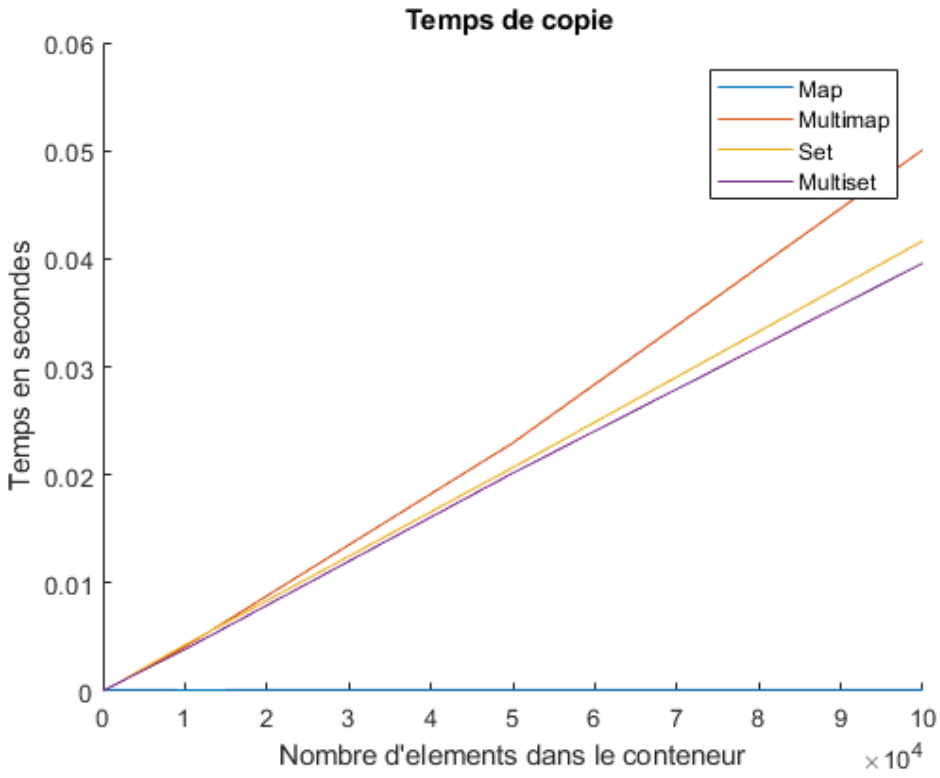
Le résultat montre que les multimap,multiset et set de complexité O(N) alors que la map a une complexité constant en O(1) pour la lecture au milieu du container.

Cela est du au fait que la map possède l'operateur [] et peut donc immédiatement avoir la valeur de l'élément voulue. A l'inverse les autres conteneurs utilisent des itérateurs qui parcourt le conteneur, il est donc naturel d'avoir une complexité en O(N).

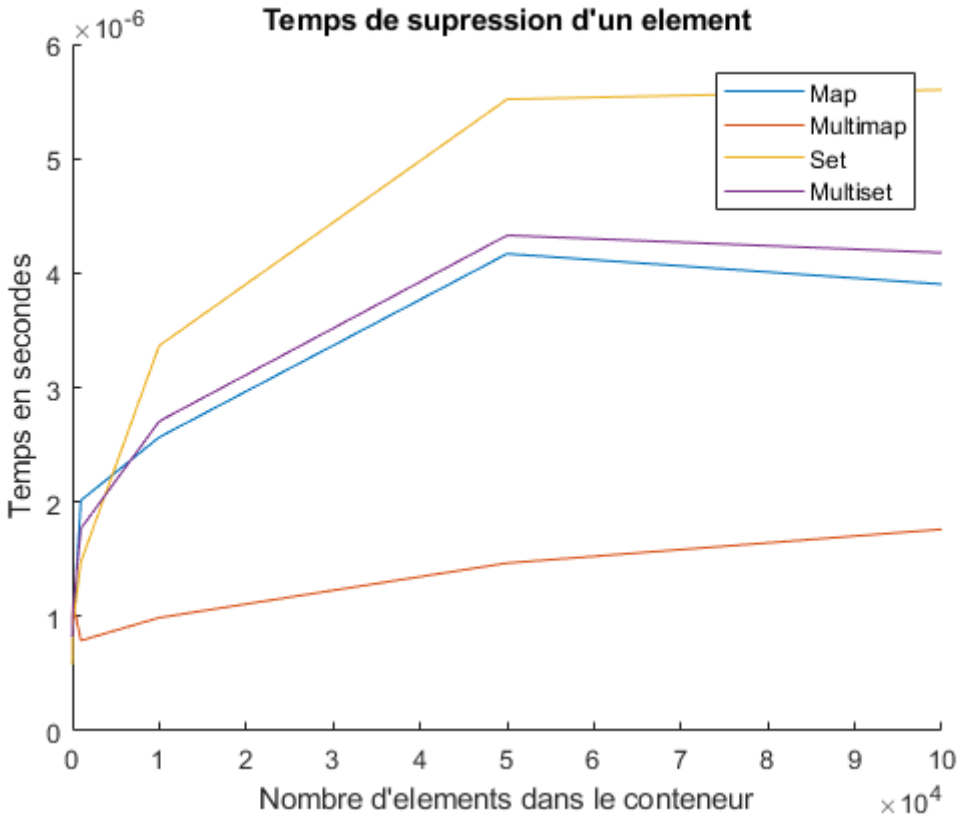
Le calculs du temps d'exécution des fonctions des conteneurs associatifs peut permettre d'évaluer la complexité de celles-ci



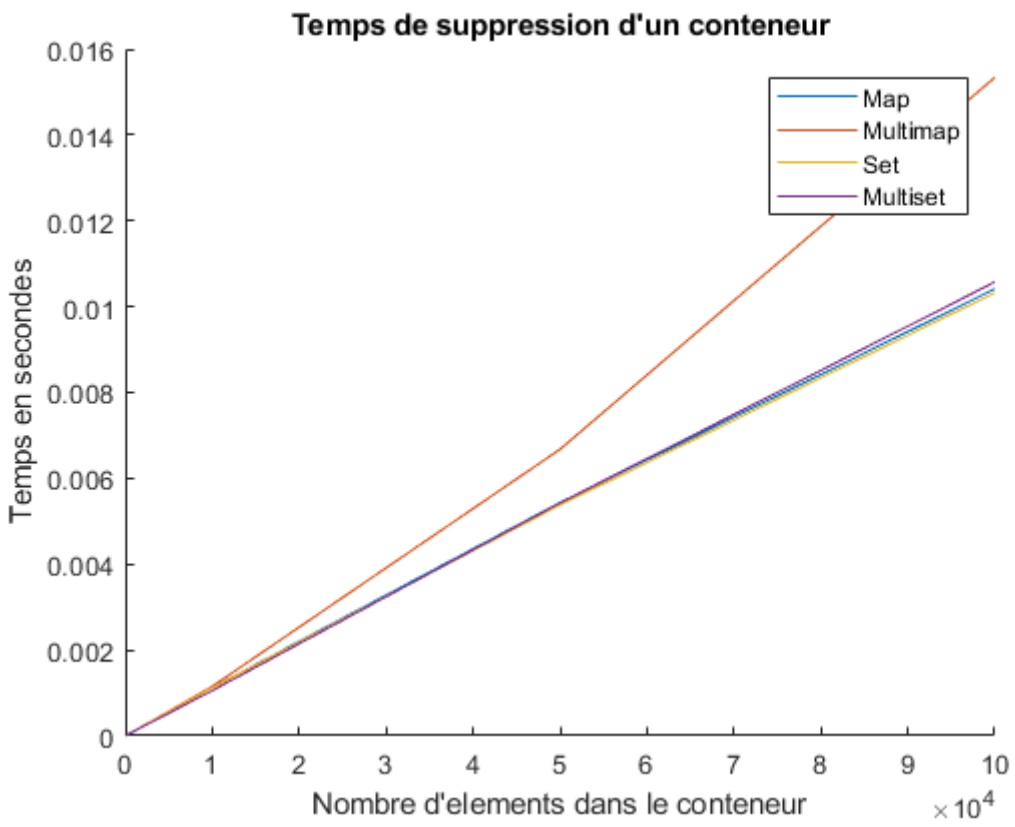
La construction d'un conteneur à l'aide de la fonction `insert` est de complexité O(Nlog(N)). En effet la fonction insert est de complexité O(log(N)) donc étant donné que l'on fait un boucle for nous avons donc bien la complexité voulue.



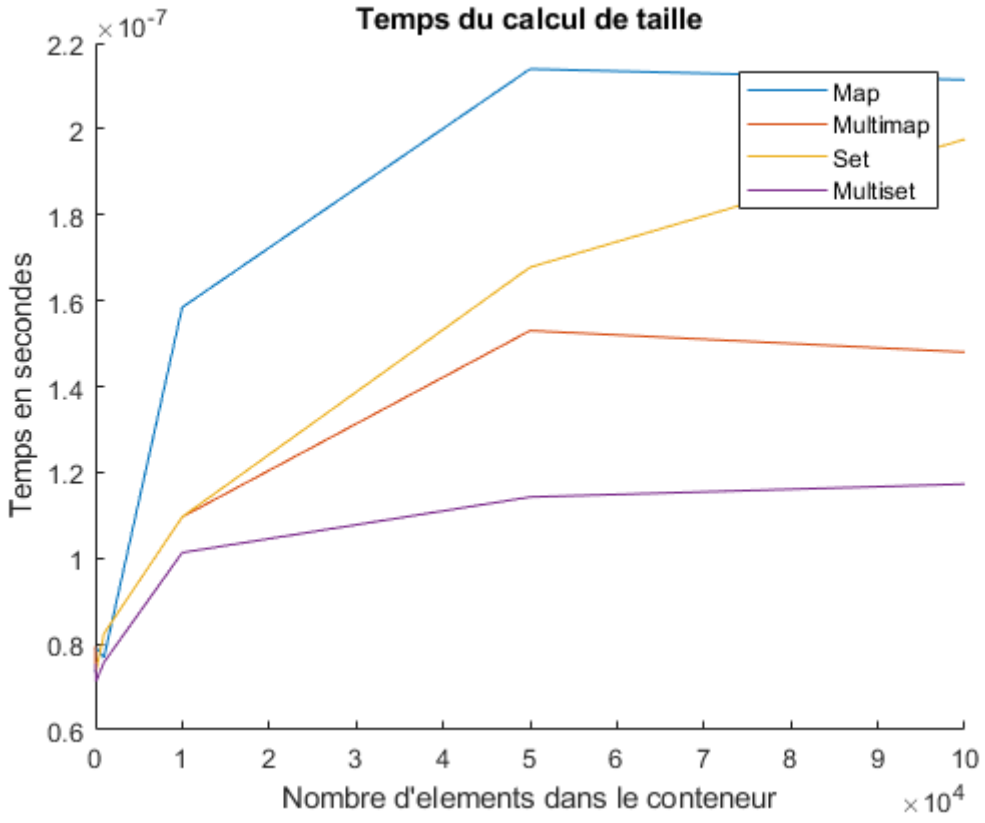
La copie d'un conteneur à l'aide du constructeur a une complexité en O(N) car celui-ci parcourt le conteneur qu'il copie.



La fonction `erase` semble de complexité O(log(N)) mais en fait du fait des faibles variations de valeurs celle-ci est en fait de complexité constante en O(1)



La suppression d'un conteneur est équivalente parmi les conteneurs associatifs, en effet la fonction `erase` est elle-même équivalente en O(1). Etant donné que l'on supprime chaque élément du conteneur il y une complexité en O(N).



La fonction size est plus efficace sur les multiset que sur les map et la complexité semble être en O(log(N)), cependant les valeurs étant très éloigné il y a une complexité constante en O(1).

Au sein des conteneurs associatif il y des fonctions qui sont aussi défini dans la librairie algorithme et étudiés dans la partie algorithm (partie V) tel que :

- `Find` : qui prend en argument une clé et renvoie une position d'itérateur de l'élément associé à la clé.

Cette fonction a une complexité en O(log(N)) du fait de la construction en forme d'arbre.

- `Count` : qui prend en argument une clé et compte combien de fois cet élément est présent dans le conteneur

Cette fonction peut être utile dans des set et map pour savoir si un élément est présent ou non. Celle-ci a une complexité en O(N).

- `Swap` : qui prend en argument 2 conteneurs et échange les valeurs de deux conteneurs.

Elle a une complexité constante du fait qu'il suffit d'échanger la référence de conteneur.

3) Interprétation des résultats

Les conteneurs associatifs ont la particularité d'être très efficace pour rechercher des valeurs dans des grosse base de données (log(N)). ON peut par exemple facilement imaginer l'application de ce conteur pour un dictionnaire ou les clefs peuvent être les mots classés par ordre alphabétique et la valeur associé peut être sa définition sous forme de string.

La vitesse de lecture est par contre au détriment de la rapidité de création du conteneur, en effet la création se fait en O(n*log(N)) ou la plupart des conteneurs sont en O(N). Ces conteneurs son particulièrement adapté à un ensemble qui nécessite d'être trié.