CONTAINERS

C++

Mustapha Benbrikho
Antoine Bralet
Valentin Burgevin
Sébastien Courmaceul
Victor Ducray
Guillaume Duret
Benjamin Gédéon
Paul Méteyer
Léo Pallas

CPE LYON

Table des matières

Introduction

Qu'est-ce qu'un conteneur ?

Un conteneur est un objet permettant de stocker d'autres objets. C'est l'élément de base de la STL qui est la bibliothèque standard de C++ basée sur des `templates` et qui apporte un canevas de programmation plus ou moins efficace et simple d'utilisation. Les conteneurs sont des objets qui peuvent proposer des méthodes permettant de manipuler les autres objets qu'ils stockent.

Pour créer un conteneur en C++, nous devons utiliser la syntaxe suivante : `std::nom_conteneur<typename>` avec `nom_conteneur` qui prend le nom d'un conteneur comme `list`, `array`, etc… .

Ce n'est pas la seule démarche pour utiliser un conteneur. En effet, il faut rajouter le fichier d'en-tête correspondant. Par exemple, si nous voulons utiliser un `vector`, il faut rajouter en haut du code la ligne `#include <vector>`. Il en est de même pour les autres types de conteneurs. Le constructeur s'écrit de la manière suivante pour tous les conteneurs sauf exception qui sera notifié dans ce cas :

```
std::nom_conteneur<typename> first; // empty nom_conteneur of typename
```

Les itérateurs

Les itérateurs servent à rendre plus flexible l'utilisation des collections et des algorithmes. En effet, un itérateur représente un élément dans n'importe quelle collection donc il est possible d'utiliser les algorithmes sur n'importe quelle collection. Pour utiliser un itérateur, qui est une sorte de pointeur (se déplace dans la mémoire), il faut mettre en œuvre la syntaxe suivante :

```
std::conteneur<typename>::iterator it;
```

Avec conteneur le type de conteneur souhaité (`vector`, `list`, etc…).

On doit aussi le relier à un conteneur pour que l'itérateur puisse le parcourir. Généralement, la référence prise est le début du conteneur pour l'itérateur `it = nom_conteneur.begin();` mais il est possible de lui ajouter une valeur n pour avoir une référence non pas à la première valeur mais à la $n^{ième}$.
Ceci permet de parcourir « simplement » le conteneur du début jusqu'à la fin en laissant la possibilité de modifier ses valeurs.

Il y également d'autres types d'itérateurs :

- Les itérateurs reverse qui permettent de parcourir une collection de la fin vers le début et qui s'écrivent de la manière suivante :
  ```
  std::conteneur<typename>::reverse_iterator rit = nom_conteneur.rbegin();
  ```

- Les itérateurs constants qui interdisent de modifier les éléments d'une collection (`const_iterator` ou `const_reverse_iterator`):

```
std::conteneur<typename>::const_iterator rit =
nom_conteneur.cbegin();
```

```
std::conteneur<typename>::const_reverse_iterator rit =
nom_conteneur.rcbegin();
```

Les benchmarks

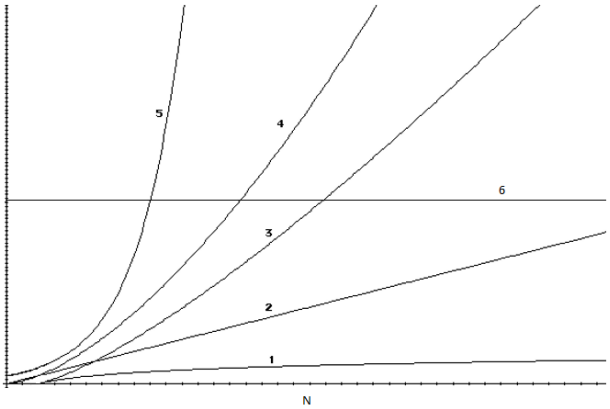
Pour comparer l'optimisation entre les conteneurs, il faut calculer les temps de réponse pour chaque commande des conteneurs (ajout au début, au milieu, à la fin, lecture au début, au milieu, à la fin et bibliothèque `algorithm`) en fonction du nombre d'éléments. Le tracé des courbes des temps de réponse en fonction du nombre d'éléments est appelé un benchmark et elle utilise la bibliothèque chrono du c++. Le programme C++ s'appuiera sur Matlab pour le tracé des courbes et il sera globalement construit sur le squelette de code suivant :

```
// copy algorithm example
#include <iostream>     // std::cout
#include <algorithm>    // std::copy
#include <vector>       // std::vector
#include <chrono>          //Libraire chron

std::chrono::time_point<std::chrono::system_clock> stard,end;
stard = std::chrono::system_clock::now();

          ---- Insérez commande à tester ----

end=std::chrono::system_clock::now();
float elapsed_seconds =
std::chrono::duration_cast<std::chrono::nanoseconds>(end-
stard).count();

  std::cout << "elapsed time: " << elapsed_seconds << "ns\n";
```

Il faudra juste insérer la commande à tester (dont on veut calculer le temps de réponse) à la ligne prévue à cet effet.

La complexité

La complexité est une grandeur qui permet d'interpréter la rapidité d'exécution d'une fonction. Elle représentée par le temps de réponse d'une fonction. On l'exprime en O. Voici quelques exemples de complexité en fonction de la longueur d'un conteneur :



1. Complexité logarithmique en O(ln(N))
2. Complexité linéaire en O(N)
3. Complexité quasi-linéaire en O(N ln(N))
4. Complexité polynômiale en $O(N^2)$
5. Complexité exponentielle en $O(a^N)$
6. Complexité instantanée en O(1)

Figure 1 : représentation de plusieurs types de complexité en fonction de la longueur d'un conteneur

Les objectifs de l'étude

Notre étude des conteneurs de la librairie STL a plusieurs objectifs :

- Comprendre le fonctionnement de chaque conteneur et mettre en évidence les particularités qui leur sont associées (lecture, ajout, modification …).
- Mettre en pratique la notion d'itérateur et analyser leur rôle selon les différents conteneurs.
- Etudier les temps de réponses associées aux fonctions de chaque conteneur. Cette étude permettra de déduire la complexité de ces fonctions mais aussi de pouvoir comparer les différents conteneurs.

I. Les conteneurs de séquence

Dans cette partie, nous allons comparer les performances des différents conteneurs de séquence entre eux. Les conteneurs de séquence font, en informatique, référence à un groupe de modèles de classe de conteneur dans la bibliothèque standard du langage C++ qui implémente le stockage d'éléments de données. Ils peuvent être utiliser pour stocker des éléments arbitraires comme des entiers ou des classes personnalisées. Avec ce type de conteneur, il est possible d'accéder aux éléments de manière séquentielle. Tous les conteneurs résident dans l'espace des noms std (cf. https://en.wikipedia.org/wiki/Sequence_container_(C%2B%2B)). Les conteneurs de séquence sont au nombre de cinq :

- Vector (le plus courant)
- Deque
- Array
- List
- Forward_list

Passons directement à la comparaison des performances des différents conteneurs de séquence. Tout d'abord, nous allons voir les différentes commandes principales pour différents conteneurs (un élément à ajouter, à enlever, etc…) puis faire, dans une deuxième partie, une comparaison des différents temps de réponse en fonction du nombre d'éléments que nous ajoutons, enlevons, etc… Nous allons enfin discuter grâce à ces courbes sur les notions de complexité des conteneurs pour chaque commande.

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

Cette partie a pour but de décrire les différentes opérations principales pour les différents conteneurs.

a. Les `vector`

Les `vectors` sont très simples à utiliser. En fait, les éléments du `vector` sont contigus en mémoire, c'est-à-dire qu'une valeur a un espace mémoire qui lui est alloué et sa valeur suivante dans le `vector` est également voisin dans la mémoire. Ceci permet d'optimiser la copie des éléments en copiant directement un espace mémoire. Cette copie est beaucoup plus rapide que de copier chaque élément les uns après les autres. On peut accéder aux valeurs présentes dans le `vector` de façon indicielle, c'est-à-dire avec les crochets [ ]. Ainsi, on peut modifier une valeur située à la n-ième position en écrivant `vector[n] = …` et pareillement pour chercher une valeur dans le `vector`, ce qui est très pratique et ce n'est pas le cas pour d'autres conteneurs que nous verrons plus tard. Nous pouvons utiliser, pour optimiser, la fonction `std::copy` de la partie `algorithm` pour copier le contenu d'un `vector` sur un autre `vector` plutôt que d'affecter un `vector` à un autre par une égalité.

- Pour les `vector`, il y a une fonction faite exprès pour ajouter un élément à la fin. Cette fonction est la fonction `push_back()`.
- La méthode `pop_back()` permet de supprimer la dernière case.
- La méthode `front()` et `back()` permettent respectivement d'accéder (lecture) à la première et la dernière case du `vector`.
- Pour ajouter un élément aléatoirement (au début ou au milieu) dans un `vector`, il faut utiliser la fonction `std::insert` mais le `vector` n'est pas vraiment un conteneur optimisé pour faire ce genre d'opération. Il faut également utiliser un itérateur :

```
std::vector<int> myvector (3,10); //Création d'un vector avec 3 fois
la valeur 10
std::vector<int>::iterator it;

it = myvector.begin(); //On référence
it = myvector.insert ( it , 20 ); //On insère 20 à la position de l
iterator (begin)

myvector.insert (it,2,30); //On insère deux fois 30 a la position de
l iterator (begin)
for (it=myvector.begin(); it<myvector.end(); it++)
    std::cout << ' ' << *it;
std::cout << '\n';
```

Ce qui donne en sortie :

```
30 30 20 10 10 10
```

Pour ajouter un élément aléatoirement dans le `vector`, il faut utiliser un itérateur et le référencer là où on veut ajouter un ou des éléments. A chaque fois, la référence de l'itérateur était au `begin()` du `vector` donc c'est pour cela qu'au début on a inséré 20 puis deux fois 30 dans le `vector`. Cependant, le `vector` n'est pas du tout optimisé pour faire ces opérations car le temps de calcul est premièrement assez long. Deuxièmement, si nous voulons rajouter des valeurs à différentes positions, il faudra appeler de nombreuses fois la fonction `insert` qui est déjà assez longue en termes de temps de réponse pour le `vector` et troisièmement, nous avons déplacé chaque valeur présente dans la liste à chaque nouvel appel de la fonction `insert`, ce qui a pour effet de les déplacer également dans la mémoire (perte de performances).

- Pour échanger le contenu d'un `vector` avec un autre, il est possible d'utiliser la fonction `std::swap` qui s'utilise de la manière suivante :

```
std::vector<int> foo (3,100);   // three ints with a value of 100
std::vector<int> bar (5,200);   // five ints with a value of 200
foo.swap(bar);
```

- Il existe deux fonctions qui sont spécifiques uniquement aux `vectors` pour la famille des conteneurs de séquence : `capacity` et `reserve`. La fonction `capacity` correspond au nombre de places dans l'espace mémoire allouées (réservées) pour ce vecteur. Elle correspond à la première puissance de deux située au-dessus ou égale de la taille du `vector`. En effet, si un vector est remplit de 128 valeurs, la fonction capacity s'utilise de la manière suivante :

```
    std::cout << "size: " << (int) myvector.size() << '\n';
    std::cout << "capacity: " << (int) myvector.capacity() << '\n';
```

Ce qui nous renvoit :

```
size: 128
capacity: 128
```

Alors que si on remplit le `vector` de 129 valeurs, on obtient le résultat suivant :

```
size: 129
capacity: 256
```

La fonction `reserve` correspond au nombre de places dans l'espace mémoire que l'on veut réservées pour ce vecteur, c'est nous qui définissons le nombre de places. En effet, en affectant 100 places dans l'espace mémoire, il est possible d'avoir 100 éléments dans le `vector`. Si le `vector` a plus de 100 éléments alors qu'il n'y avait que 100 places allouées, le programme allouera tout seul avec la fonction `capacity` et prendra la puissance de deux supérieure ou égale au nombre d'éléments. Le fonctionnement de cette fonction se fait de la manière suivante :

```
std::vector<int> myvector;
myvector.reserve(128);
// set some content in the vector:
for (int i=0; i<128; i++) myvector.push_back(i);
std::cout << "size: " << (int) myvector.size() << '\n';
std::cout << "capacity: " << (int) myvector.capacity() << '\n';
```

Ces fonctions ne sont utilisables que par les `vectors` qui, comme dit au début, ont ses éléments contigus en mémoire.

b. *Les Arrays*

Le principe des Arrays repose sur un Vector limité. En effet, afin d'instancier un tel conteneur, il faut spécifier un second paramètre template : un entier permettant de spécifier la taille de cet array :

```
std::array<int,N> test; //Création d'un array d'entiers de taille N
```

Cette particularité rend ce conteneur incompressible et inextensible ceci explique donc l'impossibilité d'insérer un élément dans ce conteneur (i.e. aucune fonction insert, push_back ni push_front disponible).

Les fonctions propres à ce type de conteneur sont les suivantes :

- empty qui permet de tester si un array est plein ou non
- swap qui permet d'échanger les valeurs de deux arrays distincts
- l'opérateur [] qui permet d'accéder rapidement à un élément

c. Les `deque`

Les `deque` sont tout aussi faciles à utiliser que les `vector`. En effet, toutes les fonctions que l'on a vues pour les `vector` sont utilisables de la même manière que pour les `deque`. Cependant, ils ont une particularité supplémentaire qui les distingue des `vector` : on peut facilement accéder aux éléments situés à l'avant (en première position du conteneur). Les `deque` sont des conteneurs optimisés pour une structure FIFO (First In, First Out). On peut ajouter une valeur au début du conteneur grâce à la fonction `push_front` et on peut enlever une même valeur en début de conteneur grâce à `pop_front`.

d. Les lists.

Ce sont des conteneurs qui prennent en charge le parcours complet de ces éléments ainsi que l'insertion intégrale (avant, arrière et milieu). Contrairement aux `deque` qui gèrent leurs éléments avec un tableau dynamique et fournit un accès aléatoire, les `lists` gère leurs éléments comme une liste à double liaison, c'est-à-dire qu'il est possible de la parcourir dans les deux sens. Elle ne fournit pas d'accès aléatoire à ses valeurs. Les `lists` sont également plus optimisés que les `deque` pour effectuer des opérations d'insertion et de retrait d'éléments dans le conteneur. Concernant les pointeurs, les `deque` invalident tous les pointeurs, références et itérateurs qui font référence à des éléments du `deque` quand il s'agit d'insérer des éléments au milieu, ce qui n'est pas le cas pour les `lists` (donc plus optimisé).

(Source : https://stackoverflow.com/questions/1436020/whats-the-difference-between-deque-and-list-stl-containers)

Mis à part les différences citées ci-dessus, les `lists` se comportent globalement comme les `deques` à l'exception près qu'il n'existe pas l'opérateur [] pour accéder de manière indicielle aux éléments du conteneur `list`.
Les autres fonctions de tri, de copie et d'insertion sont similaires aux `deques` et aux `vectors`.

e. Les Forward_Lists

Le principe d'une forward_list est exactement le même qu'une liste à l'exception prêt que les itération d'un élément à l'autre ne sont disponibles que dans un sens : la forward_list conserve uniquement en mémoire le pointeur de l'élément suivant mais ne connais pas l'élément qui le précède. Aussi, ceci implique une contrainte supplémentaire mais correspond également à un gain en mémoire non négligeable.

Cette particularité prive donc ce conteneur de la possibilité d'utilisation des itérateurs reverses. De plus les fonctions push_back et pop_back ne sont pas non plus définies : ces fonctions sont importantes afin d'insérer un élément à la fin ou le retirer facilement, or ici, le principe est de toujours se ramener à l'itérateur de début, de telles fonctions auraient donc contredit le principe même de la forward_list.

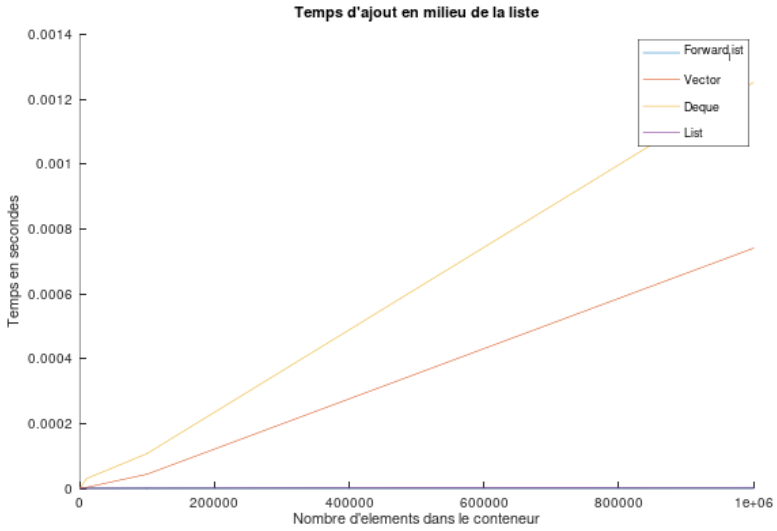
Les fonctions propres à ce type de conteneur sont les suivantes :

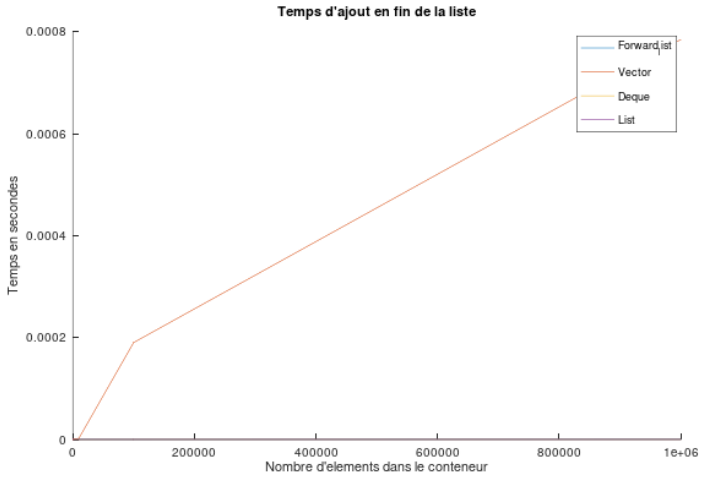
- push_front et pop_front définies préalablement
- emplace_after, insert_after et erase_after permettant respectivement de construire, insérer ou supprimer un élément dans la liste après un itérateur donné.

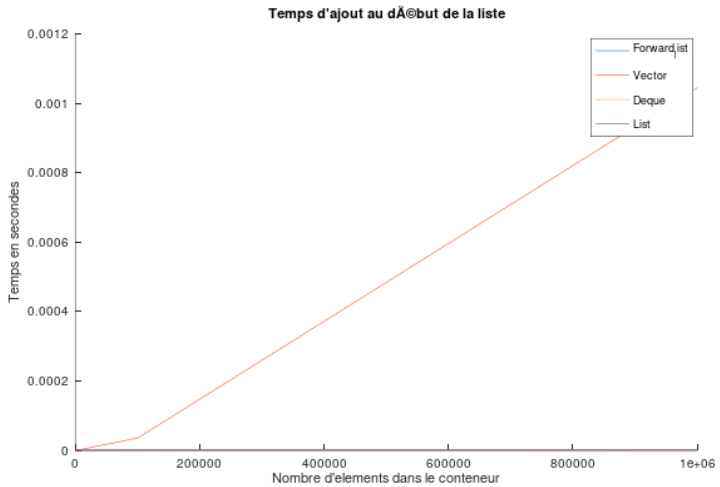
splice_after permettant de transférer des éléments depuis une autre forward_list

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

Avec le code utilisé dans la partie 1, il est possible de tracer les courbes sur Matlab des temps d'exécution pour chaque commande d'une famille de conteneurs en fonction du nombre d'éléments. Les courbes correspondantes au temps d'ajout d'éléments dans une liste sont les suivantes :





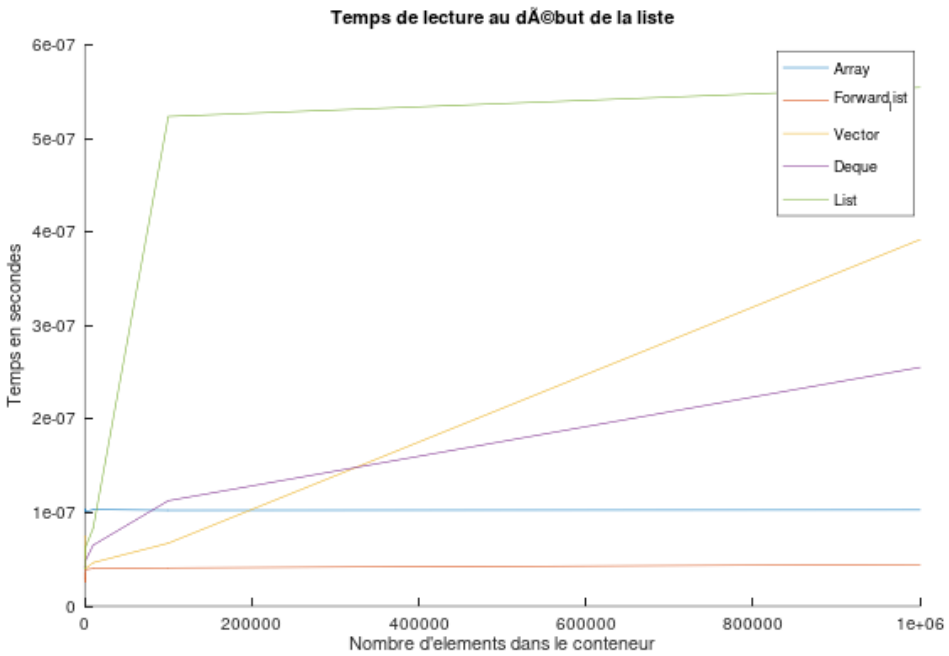


- Il est alors constatable que le `vector` est le moins optimisé pour ajouter des éléments en début de liste, ce qui était prévisible puisqu'il n'y a pas de fonction propre pour cela. L'étude montre que la complexité est linéaire O(N). Il est également observable que les autres conteneurs ont une complexité constante et sont bien plus optimisés que le `vector` puisqu'ils ont tous une fonction pour ajouter des éléments au début du conteneur.

- Pour le temps d'ajout en fin de conteneur, le code montre que le `vector` a une complexité linéaire alors que les autres conteneurs ont une complexité constante. Ce résultat paraît étrange car, d'après la théorie, c'est le conteneur `list` qui devrait avoir cette complexité et le `vector` devrait avoir une complexité constante. Les autres conteneurs, en revanche, ont une complexité constante, ce qui est conforme à la théorie. Nous supposons une erreur dans le code C++ pour le calcul du temps de réponse du `vector` de l'ajout en fin de conteneur en fonction du nombre d'itérations.
- Pour le temps en milieu de conteneur, la théorie dit que le conteneur `list` et le conteneur `forward_list` ont une complexité constante, que le `deque` et le `vector` ont une complexité linéaire et c'est bien ce qui est retrouvé à travers la courbe du temps de réponse. Les `deque` et les `vector` sont bien moins optimisés pour insérer et faire des fusions (modifier la taille) que les `lists`.

Comme précédemment, il est possible de définir les temps lors de la lecture dans un des conteneurs présentés. De nouveau, trois types de lecture sont possibles : la lecture en début de conteneur, la lecture en fin de conteneur et la lecture aléatoire dans le conteneur (ici pour prendre le pire des cas, il s'agit de la lecture au milieu du conteneur).

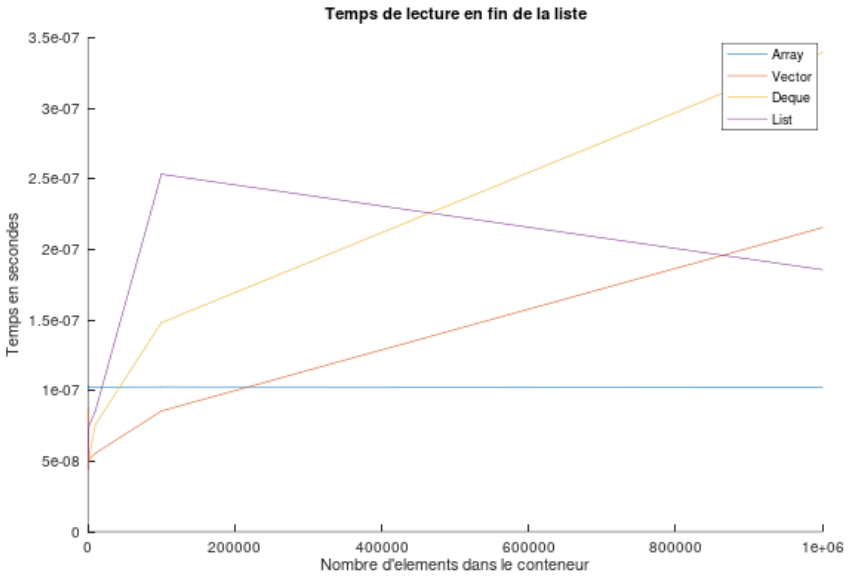
- Lecture en début de conteneur



Cette lecture concerne tous les conteneurs, en effet, chacun des conteneurs possède une fonction permettant d'accéder au début du conteneur, à savoir, la fonction "`begin()`". Aussi il est alors constatable que la lecture en début de conteneur est particulièrement rapide : de l'ordre de 0.1µs et que ce temps est relativement constant pour l'ensemble des conteneurs, ce qui permet donc d'affirmer que cette méthode a une complexité constante (soit en O(1)). De plus, il est remarquable que pour accéder

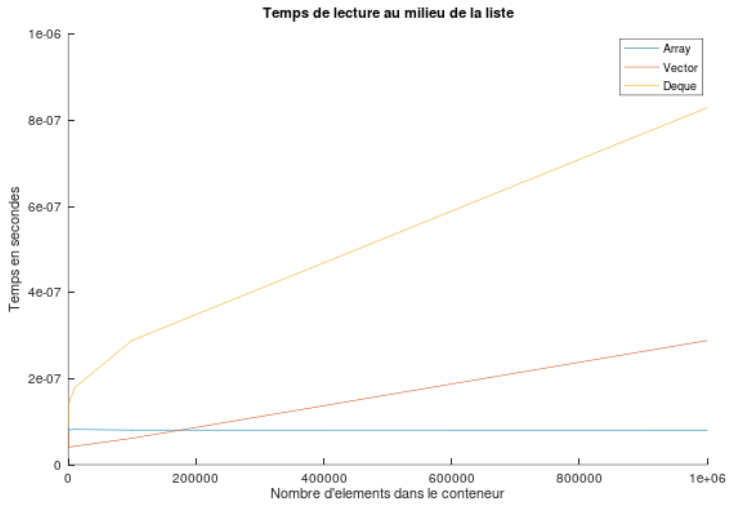
le plus rapidement au premier élément d'un conteneur, la forward_list paraît donc la plus optimisée. Ce qui était intuitif parce que cette liste à pour particularité de n'avoir d'itérateur que dans le sens "classique", soit un itérateur plus optimisé que les autres.

- Lecture en fin de conteneur



Cette lecture n'est pas possible pour tous les conteneurs, en effet, la forward_list ne possédant pas d'itérateur reverse, celle ci ne figure donc pas dans les tracés : Afin de déterminer l'élément de fin de ce conteneur, il faudrait parcourir l'ensemble de la liste pour pouvoir lire le dernier. Il apparaît comme évident que cette opération sera linéaire selon la taille de la forward_list (soit en O(N)). Pour l'ensemble des autres conteneurs, la fonction "`end()`" existe et permet ainsi d'accéder rapidement à l'itérateur de fin du conteneur. De nouveau, il est constatable qu'il est très rapide d'accéder à un tel élément grâce à cette fonction (de l'ordre de 0.1µs). Enfin, il semble que dans le cas où il faut traiter de grandes listes de nombre, le conteneur Array est plus optimisé que les autres tandis que pour de petites listes de nombre, le vector est particulièrement efficace.

- Lecture en milieu de conteneur



Le dernier cas de figure est possible seulement par 3 des conteneurs initiaux, en effet, les conteneurs list et forward_list ne possède que des références à leurs itérateur de début et/ou de fin, mais la lecture des autres valeurs doit alors se faire à l'aide d'une boucle (typiquement une boucle "`for`") ce qui a pour principale répercussion que cette opération agit avec une complexité linéaire (soit en O(n)). Au contraire, les trois autres conteneurs possèdent des méthodes spécifiques afin d'accéder rapidement à chacun de leurs éléments, en effet, deux cas sont possibles (et pas incompatibles) : soit la méthode "`at()`" a été implémentée et permet d'accéder directement à la valeur dont l'indice est donné en argument, soit l'opérateur "`[]`" a été redéfini afin d'accéder directement à ces valeurs. C'est ainsi que de la même manière que précédemment, les courbes indiquent bien que la complexité d'une telle fonction reste constante et que les conteneurs array et vector restent les plus efficaces lorsqu'il s'agit d'accéder à des éléments.

Il est tout de même notable que, le temps de lecture dans les conteneurs deque et vector semblent tout de même augmenter en fonction du nombre d'éléments qu'ils contiennent. Cette augmentation reste suffisamment faible pour pouvoir la considérer comme étant à complexité constante mais il s'agit tout de même d'un paramètre à surveiller dans le cas où l'optimisation de code est requise.

3) Utilisation type des différents conteneurs

On utilise les conteneurs de séquence lorsque les éléments ne doivent pas être triés et qu'ils ne sont pas associés à des clés. Le choix des conteneurs va se faire sur le fait de savoir si nous voulons supprimer des éléments au milieu ou au début (pas à la fin puisque tous les conteneurs de séquence peuvent accéder aux éléments de fin).

- Une `list` est un conteneur optimisé lorsque les fusions entre les conteneurs sont fréquents et lorsque nous voulons supprimer, insérer, lire ou même trier des éléments n'importe où dans le conteneur. Comme la `list` fournit une structure de listes doublement chaînées, il est possible de la parcourir dans les deux sens et donc d'utiliser `reverse_iterator` pour lire la liste en inverse. Un exemple type d'utilisation d'une `list` est le suivant (mettre `using namespace std` au début) :

```
std::list<int> first, second; // une liste vide
first.push_back( 10 );
first.push_back( 5 );
first.push_back( 7 );
first.push_back( 4 );
first.push_back( 1 );
first.push_back( 9 );
first.push_back( 7 );

second.push_back( 3 );
second.push_back( 10 );
second.push_back( 2 );

cout << "La liste first contient " << first.size() << " entiers \n";
cout << "La liste second contient " << second.size() << " entiers \n";
// utilisation d'un itérateur pour parcourir la liste first
first.sort();
for (list<int>::iterator it1 = first.begin(); it1 != first.end();
++it1)
```

```
cout << " " << *it1;
cout << "\n";
second.sort();
for (list<int>::iterator it2 = second.begin(); it2 != second.end();
++it2)
cout << " " << *it2;
cout << "\n";
// afficher le premier élément
cout << "Premier element liste first : " << first.front() << "\n";
// afficher le dernier élément
cout << "Dernier element liste first : " << first.back() << "\n";

first.merge(second);
cout << "Fusion des deux listes triees : ";
for (list<int>::iterator it=first.begin(); it!=first.end(); ++it)
cout << ' ' << *it;
cout << '\n';


cout << "Parcours de la liste first en inverse : " << "\n" ;
// parcours avec un itérateur en inverse
for ( list<int>::reverse_iterator rit = first.rbegin(); rit !=
first.rend(); ++rit )
{
cout <<" " << *rit;
}
cout << "\n";
```

Ce qui nous donne :

```
La liste first contient 7 entiers
La liste second contient 3 entiers
 1 4 5 7 7 9 10
 2 3 10
Premier element liste first : 1
Dernier element liste first : 10
Fusion des deux listes triees :  1 2 3 4 5 7 7 9 10 10
Parcours de la liste first en inverse :
 10 10 9 7 7 5 4 3 2 1
```

- Un `vector` est un conteneur optimisé pour insérer ou supprimer des éléments à la fin. Ce conteneur peut aussi être utilisé pour accéder aux valeurs aléatoirement dans celui-ci puisqu'il est possible d'accéder aux éléments par l'opérateur [] sans qu'il n'y ait de trop gros changements de taille. En effet, vu que les éléments d'un `vector` sont contigus en mémoire, si nous voulons ajouter des éléments (augmenter la taille du `vector`) autrement qu'en partant de la fin, il faut déplacer tous les éléments situés après la dernière valeur rajoutée sur la droite, ce qui prend énormément de temps. Un exemple simple d'utilisation du `vector` (mettre `using namespace std` au début) :

```
 vector<int> vector1(1);
 vector<int>::iterator it;
```

```
vector1.push_back(1);
vector1.push_back(2);
vector1.push_back(3);
vector1.push_back(4);
vector1.push_back(5);
//Ligne à insérer
cout << "Le vector contient " << vector1.size() << " entiers \n";
cout << "Le vector contient les valeurs suivantes (facon indicielle) : ";
for(int unsigned i=0;i<vector1.size();i++) //Parcours les différents éléments
du vecteur de façon indicielle
        cout << vector1[i] << " " ;
cout << "\n";

cout << "Le vector contient les valeurs suivantes (facon iterateur) : ";
for (it=vector1.begin(); it<vector1.end(); it++) //Parcours les différents
éléments du vecteur avec itérateurs
        cout << *it << " " ;
        cout << "\n";
// afficher le premier élément
cout << "Premier element vector : " << vector1.front() << "\n";
// afficher le dernier élément
cout << "Dernier element vector : " << vector1.back() << "\n";
```

Ce qui donne en sortie :

```
Le vector contient 6 entiers
Le vector contient les valeurs suivantes (facon indicielle) : 0 1 2 3 4 5
Le vector contient les valeurs suivantes (facon iterateur) : 0 1 2 3 4 5
Premier element vector : 0
Dernier element vector : 5
```

En fait, ce conteneur est multitâche et peut convenir pour faire l'ensemble des autres conteneurs : il n'a pas d'utilisation spécifique. Si on ne doit pas faire de choses spécifiques à d'autres conteneurs comme ajouter un élément au début, trier des éléments ou faire des fusions fréquentes entre conteneurs, par défaut on prend le `vector`.

- Un `deque` fonctionne de la même manière que le `vector`. Il fait tout comme un vector (de façon un peu moins optimisé) à l'exception près que le `deque` est plus optimisé pour ajouter ou supprimer des éléments au début et au milieu. Il est alors plus optimisé pour gérer les changements de taille. Un exemple d'utilisation type des `deques` :

```
deque<double> mydeque;
mydeque.push_back (100);
mydeque.push_back (200);
mydeque.push_back (300);

while (!mydeque.empty())
{
  mydeque.pop_front();
}
mydeque.push_front(1);
mydeque.push_back(2);
mydeque.push_front(0);
for (unsigned int i = 1; i<10; i++){
      deque<double>::iterator it = mydeque.begin()+1+i;
      mydeque.insert(it, 1, 1 + i*0.1);
}
```

```
cout << ”Mon deque contient : “
for (deque<double>::iterator it1=mydeque.begin(); it1!=mydeque.end();
++it1)
    cout << ' ' << *it1;
  cout << '\n';
```

Ce qui donne en sortie :

```
Mon deque contient :  0 1 1.1 1.2 1.3 1.4 1.5 1.6 1.7 1.8 1.9 2
```

- Les `Arrays`

Les `arrays` ayant la particularité d'avoir une taille fixe et inchangeable, de nombreuses utilisations peuvent en être faites. La première d'entre elle tient en la création de vecteurs coordonnées d'un point en dimension N (avec N la taille d'un `array` fixé). Aussi pour déterminer les coordonnées d'un point dans un plan, il suffit de créer un `array<double,2>` et pour décrire un point en trois dimensions, la création d'un `array<double,3>` convient très bien.
Aussi par extension, il est possible de créer des matrices de taille fixe et inextensible comme l'illustre le code suivant :

```
     /*Création de la matrice :
     *    [ 0  1  2  3  4
     *      5  6  7  8  9
     *     10 11 12 13 14]   */
     std::array<std::array<int,5>,3> matrix;

     //Création des lignes de la matrice
     std::array<int, 5> ligne1 = {0,1,2,3,4};
     std::array<int, 5> ligne2 = {5,6,7,8,9};
     std::array<int, 5> ligne3 = {10,11,12,13,14};

     matrix = {ligne1, ligne2, ligne3};
```

Afin de vérifier que la matrice est bien effective, il suffit alors d'afficher quelques termes de celle-ci tel que `matrix[0][1]` ou `matrix[2][4]` qui permettront en théorie d'obtenir 1 et 14, lors de l'affichage du test, le résultat est alors :

```
la valeur de matrix[0][1] vaut : 1
la valeur de matrix[2][4] vaut : 14
```

À partir de ces deux informations, il est alors aisément déductible que des classes `matrix` et `vect`, ayant pour argument leur taille et leur termes, peuvent être aisément implémentées ainsi que toutes les fonctions propres à ces deux outils (multiplication matricielle, produit vectoriel, produit tensoriel, …) à l'aide de la surcharge d'opérateur ou simplement de fonctions nouvelles. Il est donc possible de traiter toutes les applications qui découlent de ces outils :

traitement d'image, traitement de signaux, cartographie, statistiques, et ce en réservant toujours le minimum de place en mémoire avec un accès relativement rapide.

- Les `forward_list`

Ces listes peuvent avoir la même utilité qu'une `list` simple à ceci près que l'itérateur reverse n'est pas nécessaire. Aussi utiliser une `forward_list` plutôt qu'une `list` permettrait alors d'avoir exactement le même résultat mais de prendre moins de temps en mémoire et avec un temps d'accès à la première valeur beaucoup plus rapide que les autres.

II. Container adaptors

Les container adaptors sont construits de tel sorte à adapter un containeur séquentiel à un usage spécifique (file, pile …). Pour cela, l'accès aux éléments est restreint et le nombre de méthodes associées est par conséquent limitée. Cette famille se divise en trois conteneurs : `queue`, `stack` et `priority_queue`.

1) Présentation des conteneurs

a. Le conteneur « `queue` »

La librairie `std` contient un conteneur `queue` qui s'apparente à une file d'attente. En effet, elle fonctionne selon le principe du FIFO (First In, First Out). Cela signifie que l'on ne peut ajouter des éléments que par la fin et en retirer par le début.

Figure 2 : schéma de principe de la file d'attente

Pour construire une queue, il existe un constructeur que l'on utilise comme ceci :

```
std::queue test_queue;
```

Ceci permet de construire une queue vide. Cependant, il est aussi possible de construire une `queue` déjà remplie grâce à une surcharge du constructeur. Par exemple avec le code suivant :

```
std::deque<int> D {10, 2, 3, 8, 1, 5};
std::queue<int> test_queue2 (D);
```

Ce code permet de construire une `queue` à partir d'un `deque`. La queue alors construite aura les valeurs du `deque`, dans le même ordre.

Il existe trois méthodes permettant de modifier une file :

- `Push` : cette méthode permet d'ajouter un élément en fin de `queue`. `Push` prend l'élément à ajouter en paramètre. Exemple :

```
test_queue.push(1); // test_queue = {1}
test_queue.push(2); // test_queue = {1, 2}
```

- `Emplace` : il s'agit, comme push, d'une méthode permettant d'ajouter un élément en fin de queue, mais `emplace` construit un élément en-place dans la mémoire. Cela signifie que la queue n'est pas copiée puis recréée avec un élément de plus. On lui ajoute simplement une case mémoire contenant l'élément à ajouter. Exemple :

```
test_queue.emplace(3); // test_queue = {1, 2, 3}
```

- Pop : cette méthode permet de retirer le premier élément de la file. Cette fonction peut être utilisée de la manière suivante :

```
test_queue.pop(); // test_queue = {2, 3}
```

Ces trois méthodes n'ont pas de type de retour. Elles permettent seulement de modifier la file.

Il existe aussi une méthode `swap` permettant d'échanger le contenue de deux files. Voici un exemple d'utilisation de cette méthode :

```
test_queue.swap(test_queue2);
// test_queue = {2, 3}
// test_queue2 = {10, 2, 3, 8, 1, 5}
```

Pour lire les éléments de la file, il n'existe que deux méthodes `front` et `back` qui permettent de lire le premier et le dernier élément de la file. Ces deux méthodes ne prennent pas de paramètre mais renvoie une valeur dont le type est celui des éléments de la queue. Ces méthodes s'utilisent de la manière suivante.

```
int a = test_queue.front(); // a = 2 ;
int b = test_queue.back(); // a = 3 ;
```

Enfin, les files ont à leur disposition une méthode `size` ne prenant pas de paramètre et renvoyant un entier valant sa taille, c'est-à-dire le nombre d'élément qu'elle contient. Voici comment cette méthode peut être utilisée.

```
int taille = test_queue.size(); // taille = 2
```

Dans la mémoire, les éléments de la file ne sont pas ranger les uns à la suite des autres. Ceci permet de ne pas avoir à réserver un espace mémoire au moment de sa construction.

b. Le conteneur « stack »

Après avoir traité les containers queue, il est intéressant d'aborder un autre type de container de la famille des containers adaptors, les stacks. Le container stack caractérise une liste de type LIFO (Last In, First Out). Plus précisément, il s'agit donc d'une pile où on ne peut donc interagir que par le sommet de celle-ci.

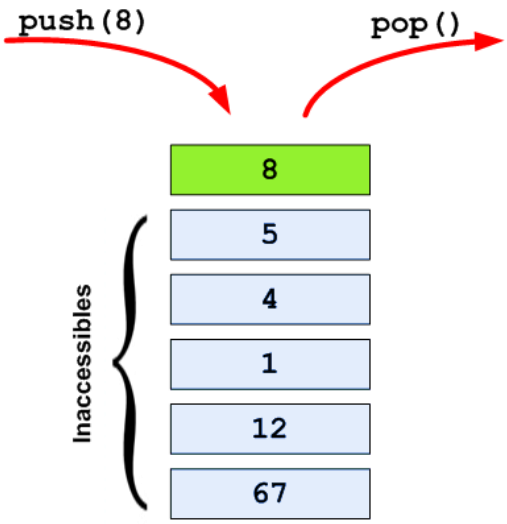


Figure 3 : schéma du principe de la pile

Son comportement au sein de l'espace mémoire est similaire à celui d'une queue. Il est là aussi possible d'initialiser un stack à partir d'un deque :

```
std::deque<int> test_deque;
test_deque = {67,12,1,4,5,8};
std::stack<int> test_stack(test_deque);
```

Les fonctions membres du container *stack* sont quasi semblables à celles du container queue à quelques différences. Plus précisément :

- Les fonctions empty, size, push, pop, emplace et swap sont identiques à celle utilisées pour le container queue.
- La fonction top permet d'accéder à l'élément au sommet de la pile et il n'y a donc plus de fonction front et back comme ce fut le cas pour la queue.

Exemple d'utilisation de la fonction top :

```
std::cout << test_stack.top() << std::endl;
//returns 8
```

c. Le conteneur priority_queue

Ce conteneur se rapproche beaucoup du conteneur `std::stack`. En effet, il possède les mêmes méthodes que se dernier, à la différence que les éléments aoutés par les méthodes `push` et `emplace` ne sont pas ajouter en fin du conteneur. Ils sont en fait triés par priorité.

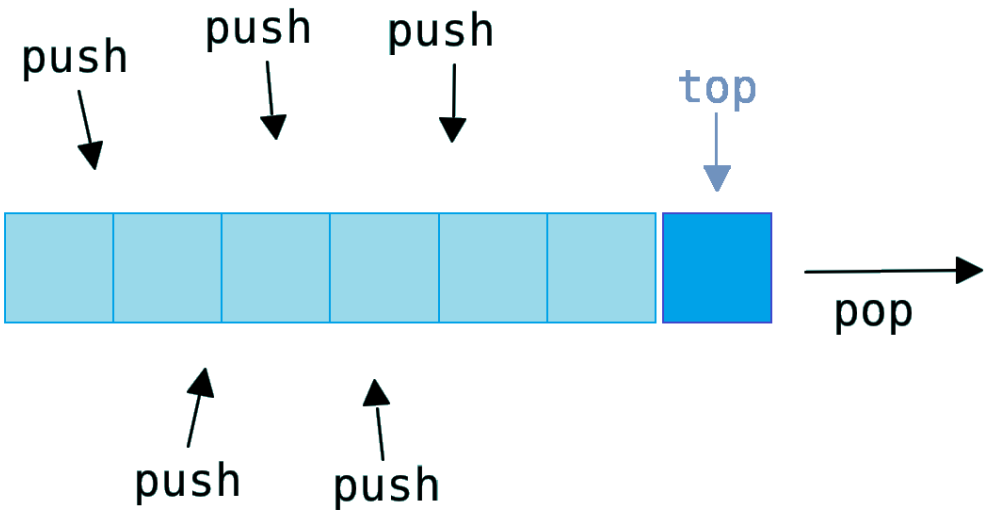


Figure 4 : schéma de principe de la `priority_queue`

Par exemple, nous pouvons prendre le code suivant :

```
#include <queue>
#include <deque>
#include <iostream>

int main()
{
	// Construction d'une priority_queue
	std::priority_queue<int> test_priority_queue;
	test_priority_queue.push(1);
	test_priority_queue.push(20);
	test_priority_queue.push(46);
	test_priority_queue.push(13);
	test_priority_queue.push(5);

	int taille = test_priority_queue.size();
	std::cout << "Conteneur avant ajout : " << std::endl;

	// Affichage de la priority_queue
	for(int i = 0; i < taille ; i++)
	{
		std::cout << i << " : " << test_priority_queue.top();
		std::cout << std::endl;
		test_priority_queue.pop();
	}
	std::cout << std::endl;


	// On re-remplit la priority_queue
	test_priority_queue.push(1);
	test_priority_queue.push(20);
```

```
	test_priority_queue.push(46);
	test_priority_queue.push(13);
	test_priority_queue.push(5);

	test_priority_queue.push(12);
	int taille2 = test_priority_queue.size();

	std::cout << "Conteneur après ajout : " << std::endl;

	// Affichage de la priority_queue
	for(int i = 0; i < taille2; i++)
	{
		std::cout << i << " : " << test_priority_queue.top();
		std::cout << std::endl;
		test_priority_queue.pop();
	}

	return 0;
}
```

Ce code donne l'affichage suivant :

```
Conteneur avant ajout :
0 : 46
1 : 20
2 : 13
3 : 5
4 : 1

Conteneur après ajout :
0 : 46
1 : 20
2 : 13
3 : 12
4 : 5
5 : 1
```

Ces résultats montrent bien que les valeurs sont triées par ordre décroissant dans la pile. La valeur qui est ajoutée n'est ni au début, ni à la fin.

2) Mesures de temps d'exécution

a. Operator=

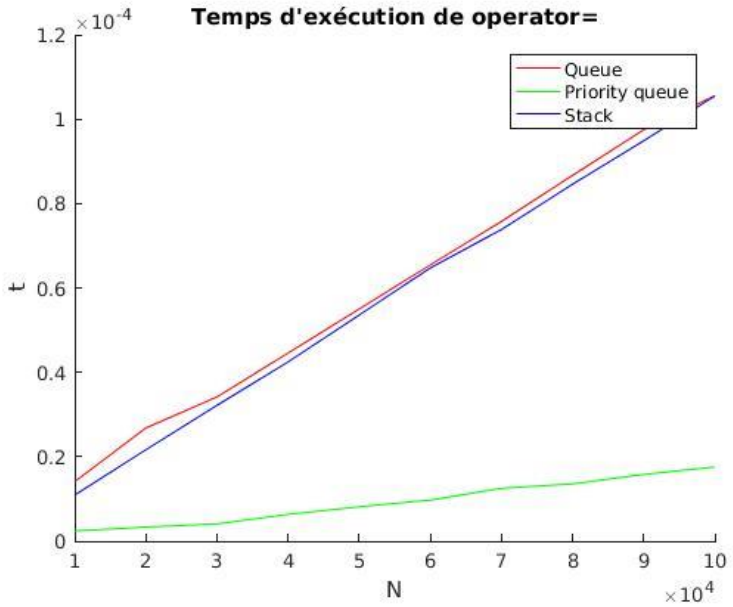


Figure 5 : temps d'exécution de l'opérateur « = » en fonction de la longueur du conteneur

L'évolution des temps de réponses est linéaire. La complexité d'`operator=` est donc en O(N). Pour cet opérateur, `stack` et `queue` sont très similaires, tandis que `priority_queue` est plus rapide.

b. Méthodes de lecture dans les conteneurs

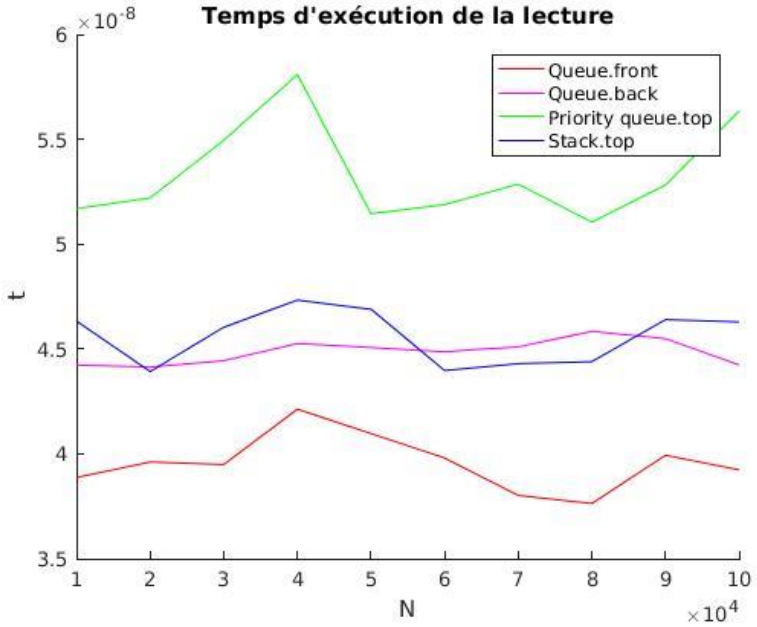


Figure 5 : temps d'exécution des méthodes de lecture en fonction de la longueur du conteneur

Etant donné que les trois conteneurs ne possèdent pas les mêmes fonctions en matière de lecture, il a fallu adapter les tests en conséquence. Pouvant accéder au premier et au dernier élément de la

queue, nous avons testé la lecture en début et fin de file. Quant aux conteneurs `stack` et `priority_queue`, seul la fonction top a été testé.

En observant les différents courbes, nous constatons un comportement relativement constant de chacune d'elle (complexité en O(1)). La précision de la mesure (à 10^{-8} près) entraine certes parfois d'importantes variations de temps mais l'évolution tourne autour d'une même moyenne.

En comparant ces moyennes, un écart entre le temps de réponse d'une lecture en début de file et en fin est observable pour la queue. Il serait donc plus rapide de lire une valeur en début de queue qu'en fin.

De plus, nous pouvons constater que le temps de réponse de la lecture pour la `priority_queue` est plus grand que pour les deux autres conteneurs.

c. Méthode `empty`

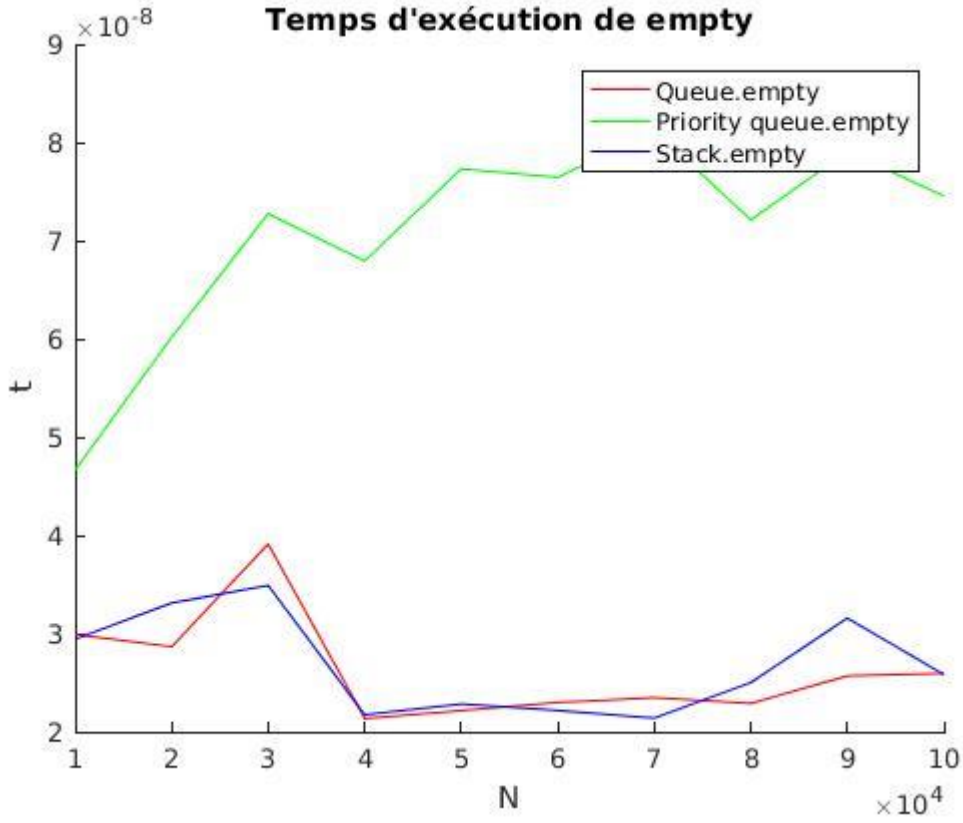


Figure 6 : temps d'exécution de la méthode « `empty` » en fonction de la longueur du conteneur

A partir de la figure précédente, un comportement similaire est observable entre les conteneurs `stack` et `queue`. Le temps de réponse de la fonction empty demeure constant (complexité en O(1)) pour chacune d'elle et relativement faible par rapport à celui du conteneur `priority_queue`.

En effet, le temps de réponse de ce conteneur est assez grand par rapport aux deux autres. Etant donné que la précision des temps est là aussi à relativiser, nous pouvons conclure sur un comportement quasi constant de la fonction empty au sein de ce conteneur (complexité en O(1)).

d. Méthode size

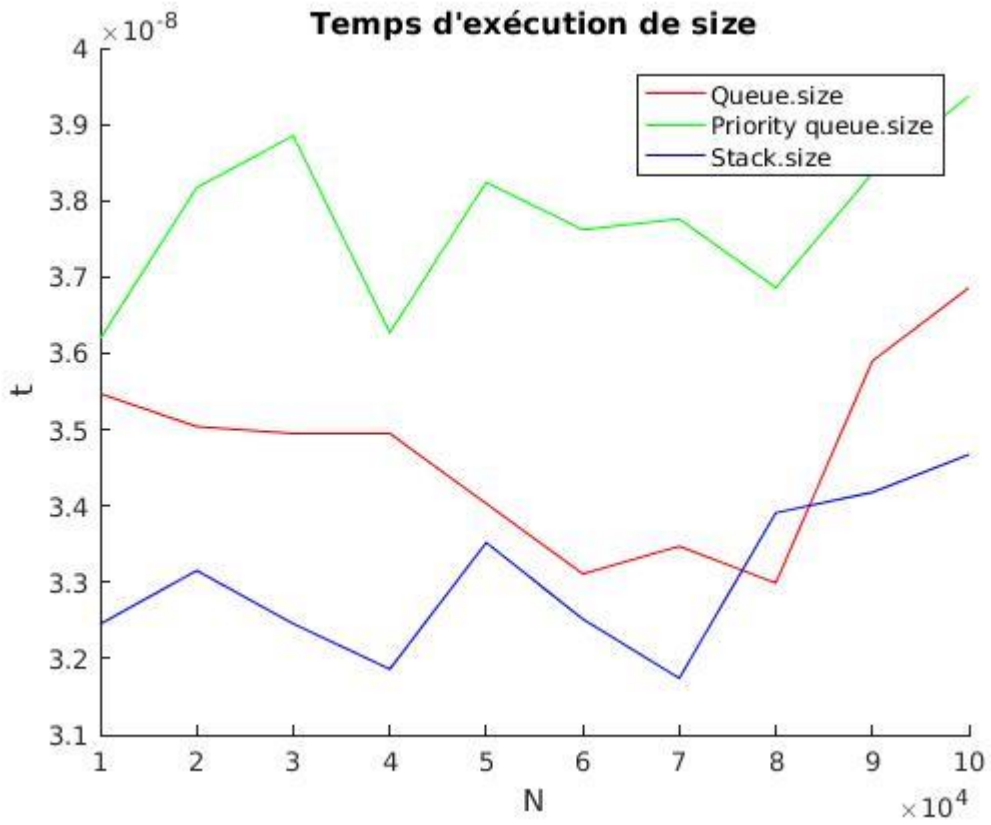


Figure 7 : temps d'exécution de la méthode « `size` » en fonction de la longueur du conteneur

Il est facile de voir que les temps de réponse de la fonction `size` pour chaque conteneur présentent d'importantes variations. Les temps de réponses étaient très petits, de faibles variations d'exécutions ou internes à l'ordinateur provoque une instabilité des mesures. Cependant, malgré cela, nous pouvons envisager un comportement relativement constant de cette fonction au sein de chaque conteneur (complexité en O(1)).

De plus, nous observons là aussi un temps de réponse plus élevé pour le conteneur `priority_queue` que pour les deux autres.

e. Méthodes d'ajout d'un élément dans le conteneur

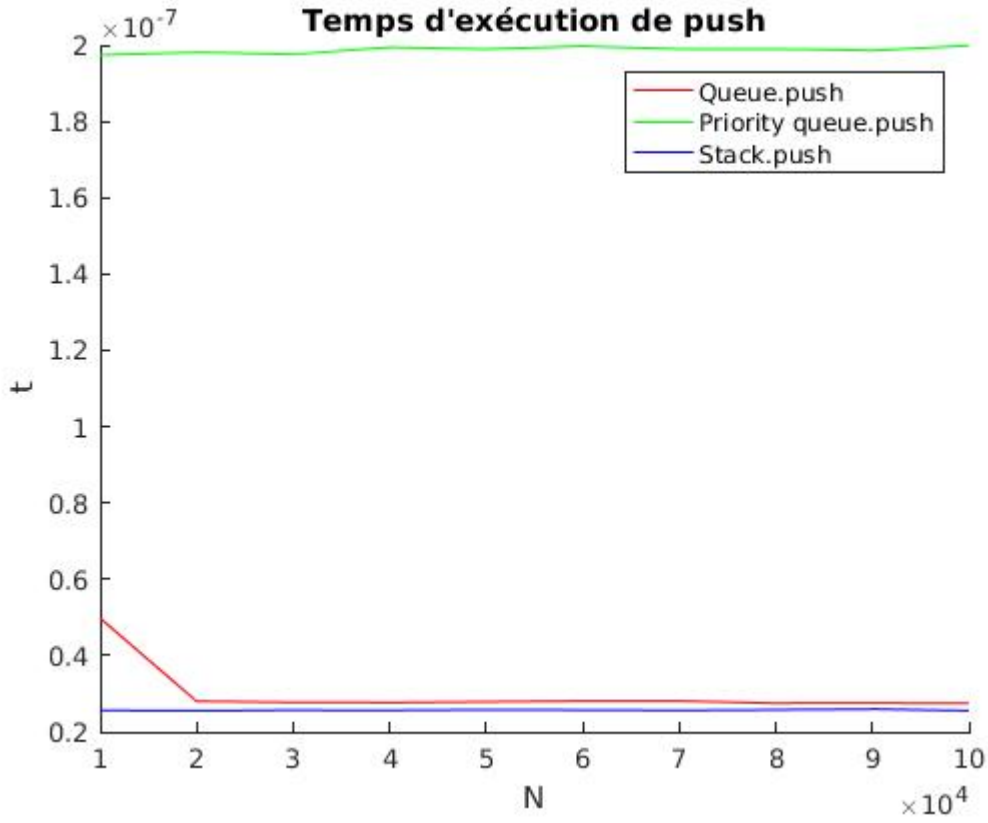


Figure 8 : temps d'exécution des méthodes « `push` » en fonction de la longueur du conteneur

Contrairement aux précédents tests, le caractère constant (complexité en O(1)) de la fonction `push` est clairement observable pour chaque conteneur. De plus, nous constatons là aussi une importante similarité entre les temps de réponses pour queue et stack. De son côté, `priority_queue` demeure le conteneur donc le temps de réponse associé à une fonction est le plus grand.

f. Méthode d'ajout d'un élément du conteneur

En observant la figure (ci-dessous), nous constater une forte ressemblance entre les courbes de la fonction `push` et celles de la fonction `emplace`. En effet, nous retrouvons là aussi un comportement constant des courbes correspondants à chaque conteneur (complexité en 0(1)). De plus, nous retrouvons encore une fois l'écart entre les temps de réponse de `priority_queue` par rapport aux deux autres conteneurs.

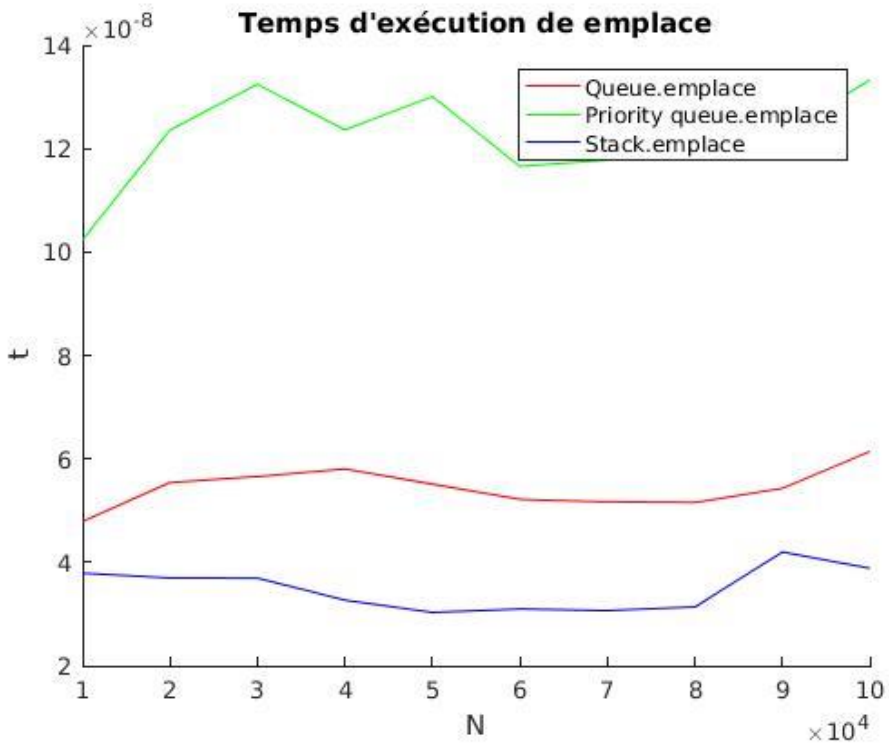


Figure 9 : temps d'exécution des méthodes « `emplace` » en fonction de la longueur du conteneur

g. Méthode de retrait d'un élément du conteneur

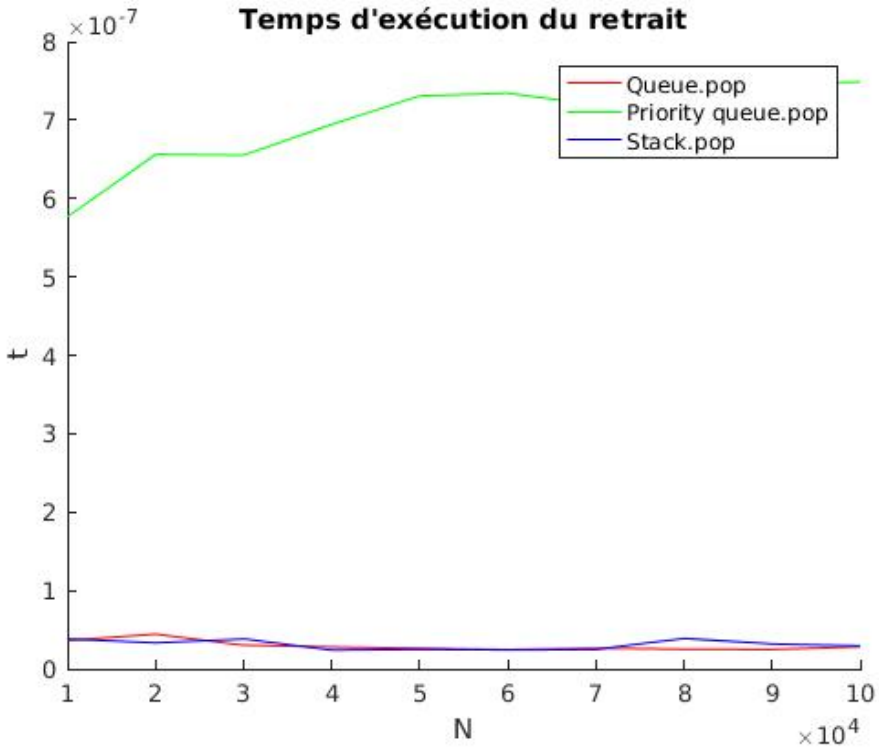


Figure 9 : temps d'exécution de la méthode « `pop` » en fonction de la longueur du conteneur

Nous remarquons encore une fois un comportement similaire de la fonction `pop` pour chaque conteneur :

- Un comportement globalement constant du temps de réponse (complexité O(1)).
- Un temps de réponse plus grand pour la `priority_queue` et similaire pour les deux autres.

h. Méthode `swap`

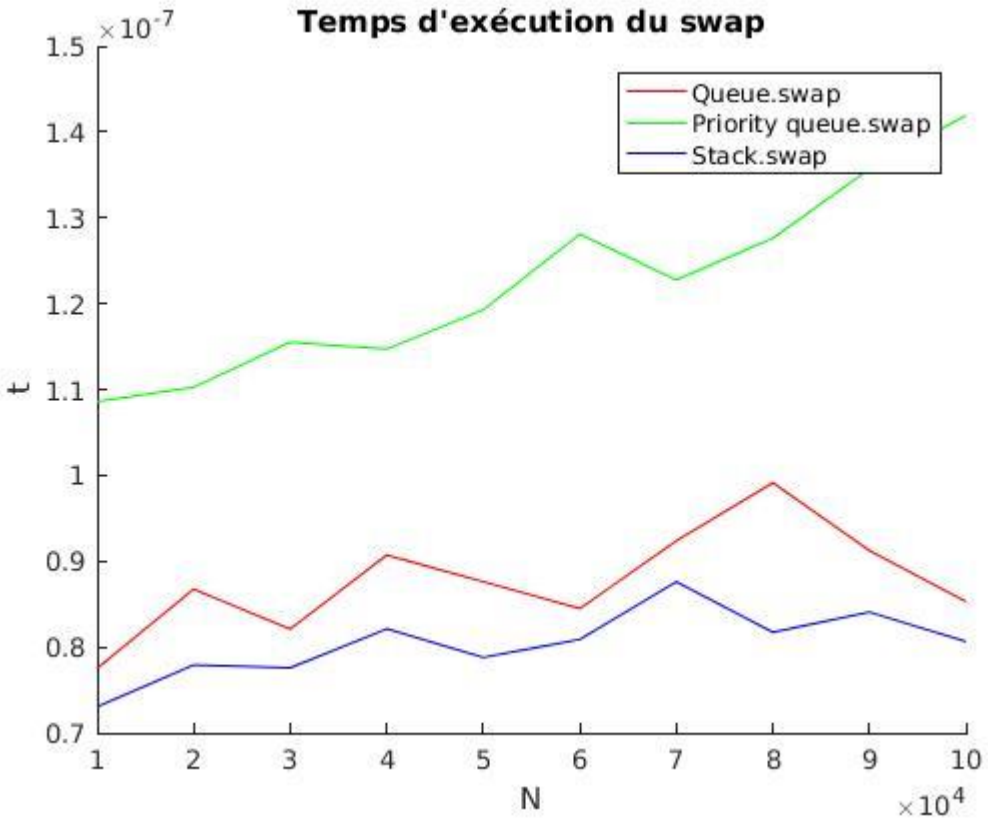


Figure 10 : temps d'exécution de la méthode « `swap` » en fonction de la longueur du conteneur

Pour cette dernière fonction, les variations sont plus importantes que pour les précédents tests. Cependant nous pouvons là aussi conclure à une complexité en O(1). De plus, l'écart entre les temps d'execution demeure identique aux précédents tests.

3) Interprétation des résultats

Les résultats de la partie 2) b. montrent que les temps de réponse de la fonction `operator=` avaient un comportement linéaire. Copier une liste de N éléments prendraient donc N fois plus de temps que pour 1 élément. En effet, nous avons pu voir dans les parties 2) f. et 2) g. que les fonctions d'ajouts étaient de complexité O(1). Une copie de chaque élément par la fonction `operator=` entraine donc N appels à des fonctions de complexité constantes.

Pour les autres fonctions, nous avons globalement observé une complexité constante. En effet, l'accès aux éléments de la liste ne s'effectue que par les extrémités. Le nombre d'éléments N n'influe donc pas sur l'exécution de ces fonctions.

De plus, les expériences 2) f. et 2) g. ont mis en évidence le fait que la méthode `push` est plus lente que la méthode `emplace`. Ceci est cohérent avec le fonctionnement de ces deux méthodes. En effet, la fonction push crée un nouveau conteneur de longueur N+1 tandis que la méthode `emplace` se contente d'ajouter un élément au conteneur existant.

Enfin, les méthodes associées au conteneur `priority_queue`, notamment les fonctions d'ajouts, sont plus lentes que pour les deux autres conteneurs. Ceci s'explique par le fait que l'on doive trier le conteneur à chaque ajout d'élément.

III. Unoredered containers

Les unordered associative containers sont une famille de conteneurs dans lequel la notion d'ordre n'existe pas. Donc toutes les valeurs deviennent des constantes à l'intérieur du conteneur. Comme la notion d'ordre n'existe pas, alors pour ranger chaque élément du conteneur, le conteneur possède des objets appelés bucket. Un bucket (seau en anglais) est un emplacement permettant de ranger des éléments à partir de leur hashcode (valeur intrinsèque à chaque valeur qui permet de les différencier les unes des autres).

Il n'y a pas de notion d'ordre donc le conteneur ne possède pas de reverse_iterator mais peut utiliser des const_iterator.

1) Présentation des conteneurs

a. Unordered Set

Tout comme les set, les unordered set sont des conteneurs associatif dont tous les éléments sont uniques. Cependant, les valeurs contenues dans les unordered set ne sont pas modifiables. Donc toutes les valeurs deviennent des constantes à l'intérieur du conteneur. Un élément souhaité ne peut être accédé directement car les éléments ne sont pas rangés dans un ordre précis.

Les méthodes les plus importantes de ce conteneur sont : empty, size, max_size, clear, insert, emplace, erase, swap, find.

Il utilise aussi count qui, dans le cas de ce conteneur, retourne 1 si l'élément en paramètre de la fonction se trouve à l'intérieur du conteneur et 0 sinon.

b. Unordered Multiset

Les unordered multiset sont identiques aux unordered set en presque tout point. La seule différence réside dans le fait que deux éléments du conteneur peuvent avoir la même valeur.

Les fonctions restent les mêmes à l'exception de la méthode count qui compte le nombre d'éléments qui ont la même valeur que le paramètre de la fonction. Elle peut donc, contrairement au cas précédent, retourner une valeur supérieure à 1.

c. Unordered Map

Tout comme les map, les unordered map sont des conteneurs associatifs contenant des paires d'éléments clé-valeur dont les clés sont uniques.

Ce conteneur utilise les mêmes méthodes que les unordered set avec deux méthodes en plus : at, operator[]. Les deux méthodes permettent d'accéder aux valeurs des éléments de la map en

mettant la clé en paramètre. La différence entre les deux méthodes est que la méthode `at` retourne une erreur si la clé n'existe pas alors que la fonction `opérator[]` crée un nouvel élément. La méthode count retourne le nombre d'éléments avec la clé spécifiée. Comme la clé est unique, alors cette méthode retourne 1 si la clé est à l'intérieur du conteneur et 0 sinon.

d. Unordered Multimap

Les `unordered multimap` fonctionnent de manière similaire que les `unordered map` excepté que deux valeurs différentes peuvent avoir la même clé.

2) Mesure des temps

Cette partie consiste en la mesure du temps d'exécution d'une liste de fonctions que l'on exécute pour les conteneurs `unordered set` et `unordered map`. Parmi ces fonctions, on y trouve : `insert`, `find`, `count`, `swap`, `copy`, `min_element`, `max_element`, `set_union` et `set_intersection`.

i. Fonction Insert

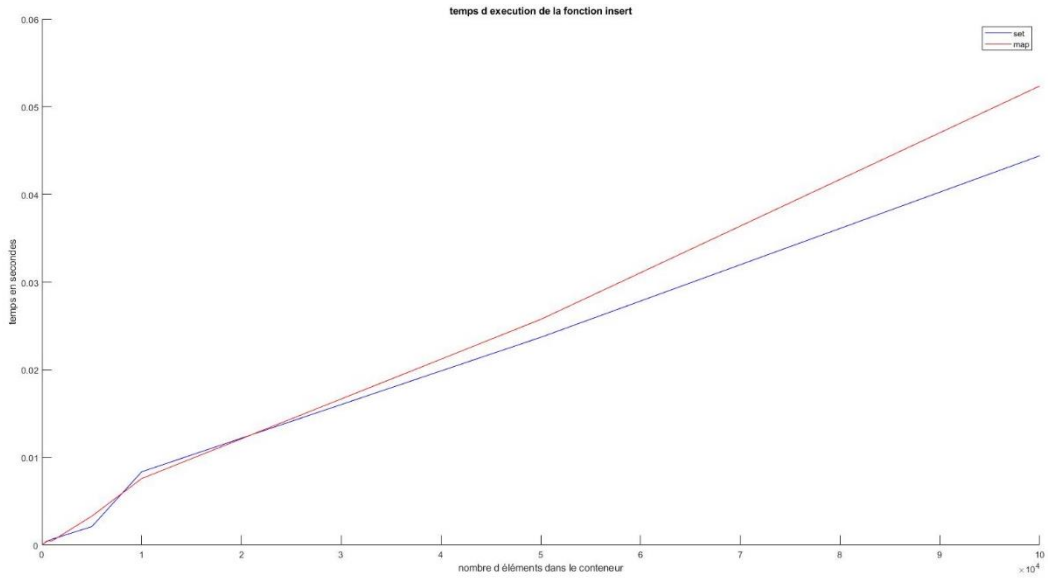


Figure 6: comparaison du temps d'exécution sur la fonction *insert*

Etudions les différences entre les deux conteneurs pour la fonction `insert` : Nous pouvons voir une évolution linéaire pour les `unordered_map` alors que les `unordered_set`, le temps d'exécution est légèrement plus faible. Les courbes montrent que le temps d'exécution de la fonction est environ le même jusqu'à un nombre d'éléments de 20 000, puis les `unordered_map` sont un peu plus rapide au-delà de ce nombre d'éléments dans le conteneur.

ii. Fonctions `Find`

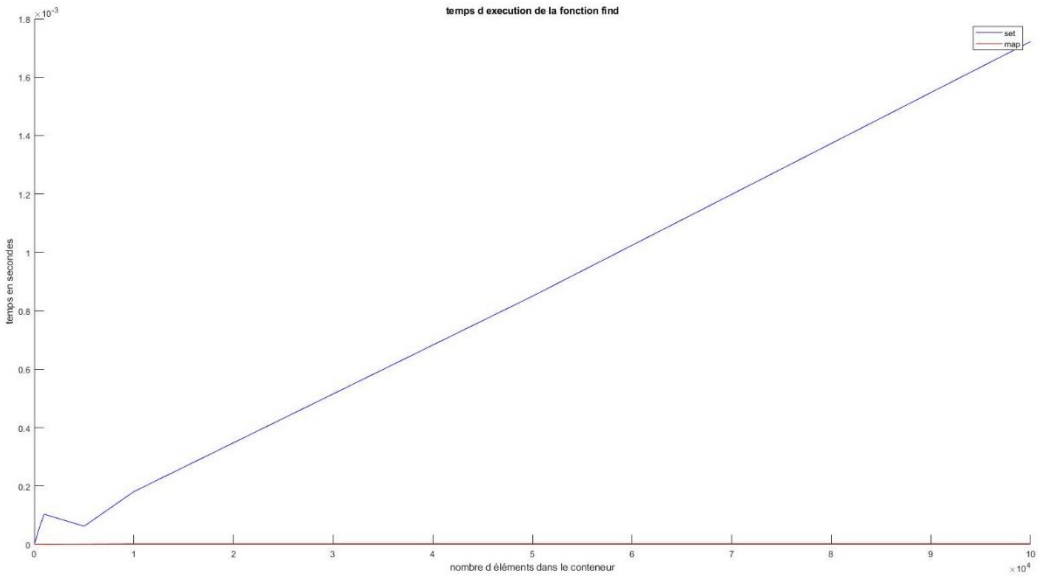
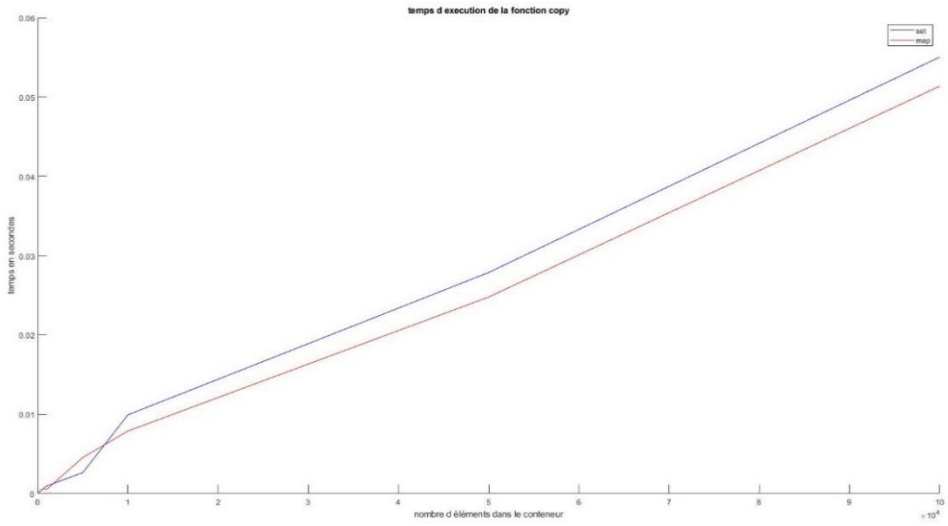


Figure 7 : comparaison du temps d'exécution sur la fonction *find*

Comparons nos deux conteneurs pour la fonction `find` : Nous pouvons voir que le temps d'exécution pour le conteneur `unordered_set` augmente linéairement (complexité O(n)) en fonction du nombre d'éléments dans le conteneur là où il reste presque constant pour `l'unordered_map` (complexité O (1)).

iii. Fonction `Copy`

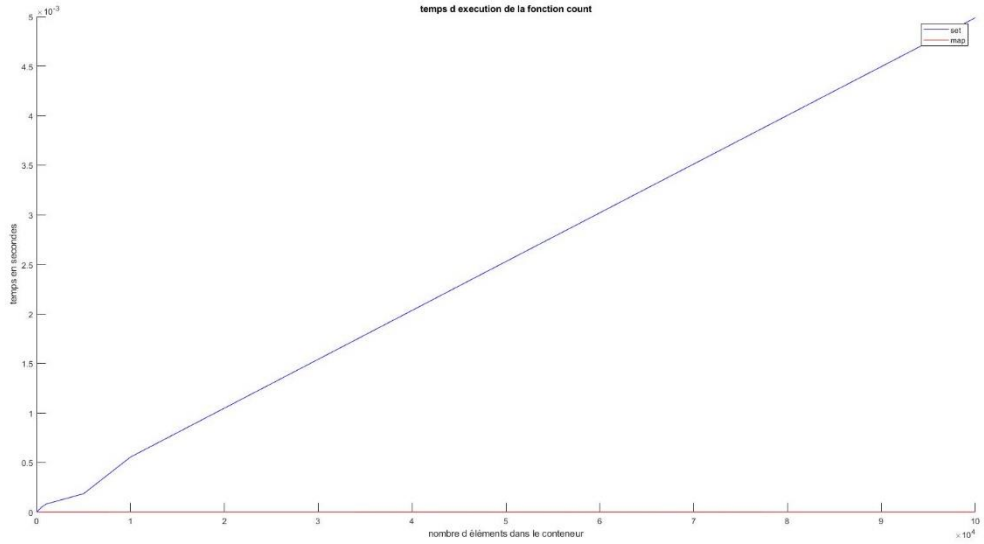
La fonction copy est une fonction qui permet de copier tous les éléments d'un conteneur dans un autre.



La figure ci-dessus compare les temps d'exécution de la fonction copy de la librairie `algorithm` pour les conteneurs `unordered_set` et `unordered_map` en fonction du nombre d'élément dans le conteneur. Les deux courbes ont des comportements très similaires. En effet, le temps d'exécution de la fonction évolue à peu près linéairement selon le nombre d'éléments du conteneur. Ainsi la méthode `copy` des conteneurs `unordered set` et `unordered map` a une complexité plutôt linéaire en o(n) dans les deux cas.

iv. Fonction `Count`

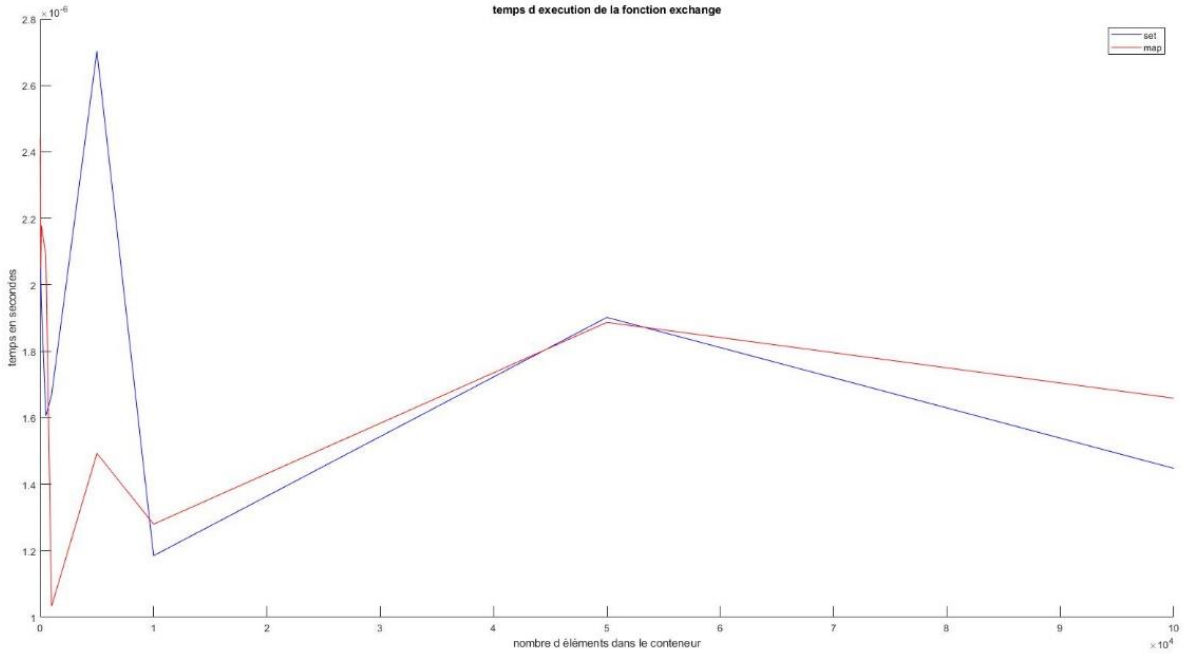
La fonction `count` permet de compter le nombre d'occurrence d'un élément dans un conteneur.



Nous comparons cette fois-ci les fonctions *count* des deux conteneurs. D'après la figure ci-dessus, le temps d'exécution des deux fonctions est très différent. La courbe est constante pour le `unordered map` alors qu'elle évolue linéairement pour le `unordered set`. La fonction a donc une complexité de o(1) pour les `unordered map` et o(n) pour les `unordered set`.

v. Fonction `Swap`

La fonction `swap` permet d'échanger le contenu d'un conteneur avec un autre.



La figure montre qu'à partir d'un certain nombre d'éléments, la fonction `swap` évolue de façon similaire. Malgré certaines variations locales, globalement, la fonction `swap` évolue de manière constante (entre 1 et 3×10^{-6} sec). Ainsi la complexité pour les deux conteneurs est de o(1).

vi. Fonction `Min_element`

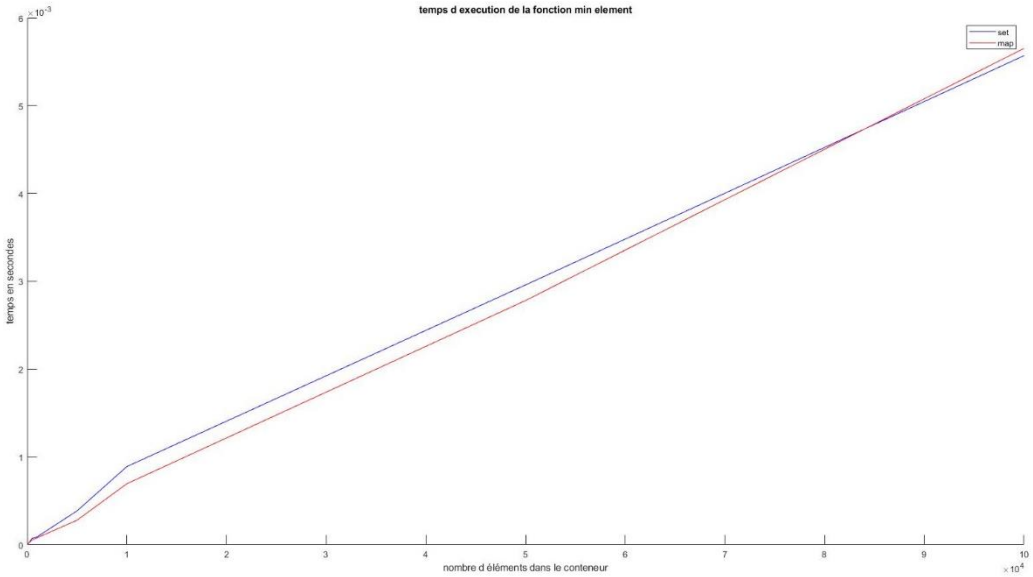


Figure 8:comparaison du temps d'exécution sur la fonction *min_element*

Nous comparons maintenant les temps d'exécution des deux conteneurs pour la fonction `min_element` : ils évoluent linéairement (complexité O(n)) pour les deux conteneurs, ils sont aussi rapides pour exécuter la fonction.

vii. Fonction `max_element`

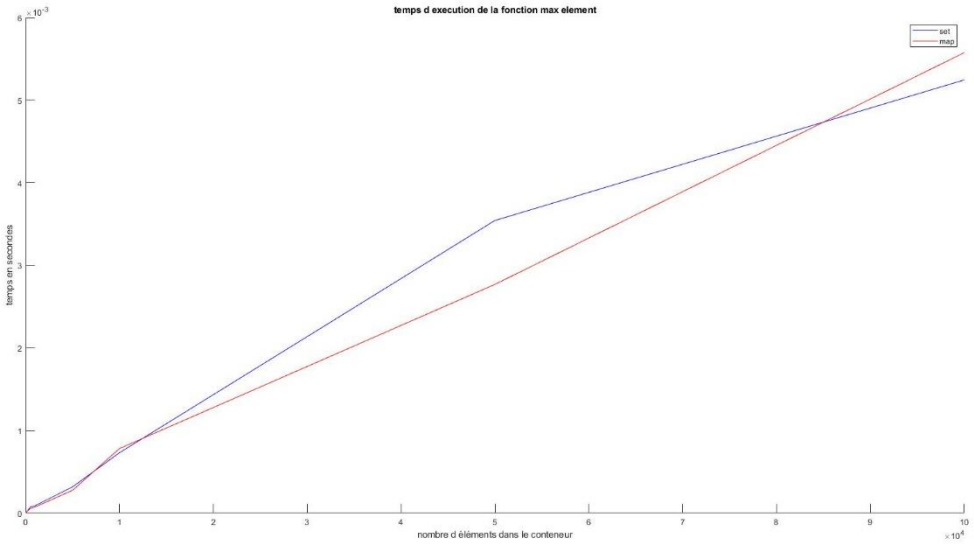


Figure 9: comparaison pour la fonction max_element

On souhaite comparer à l'aide du graphique précédent le temps d'exécution pour la recherche de l'élément maximum de nos conteneurs (`unordered_map` et `unordered_set`). Globalement, les courbes des deux conteneurs évoluent de la même manière c'est-à-dire de manière linéaire.

viii. Fonction intersection

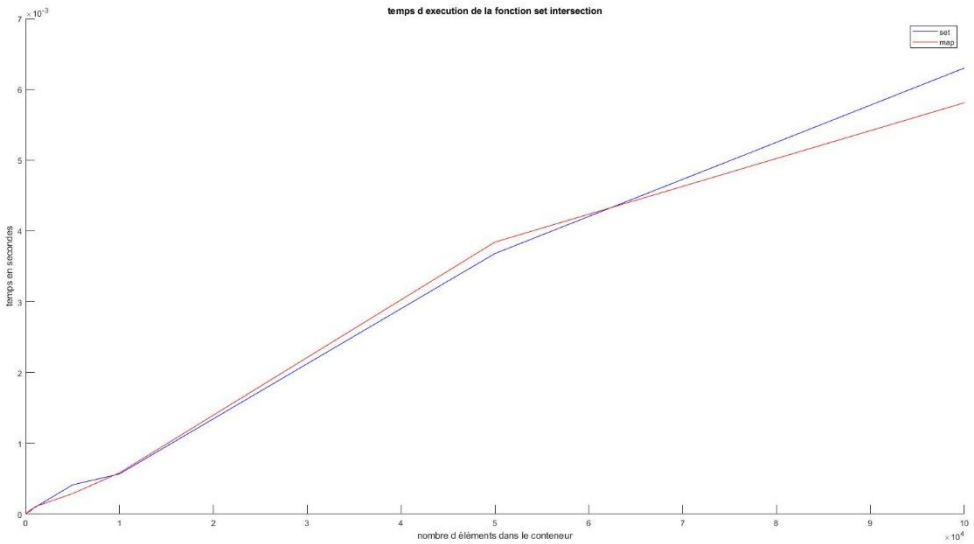


Figure 10: comparaison du temps d'exécution pour la fonction d'intersection

De même, on souhaite comparer les temps d'exécution de l'`unordered_map` et l'`unordered_set` pour la fonction d'intersection. D'après les courbes obtenues avec Matlab, pour

un faible nombre d'éléments il sera plus rapide pour l'algorithme d'utiliser une `unordered_map`. Si on a un grand nombre d'éléments situés entre 1000 et 6000 éléments, les deux conteneurs semblent aussi rapides l'un que l'autre.

ix. Fonction `Union`

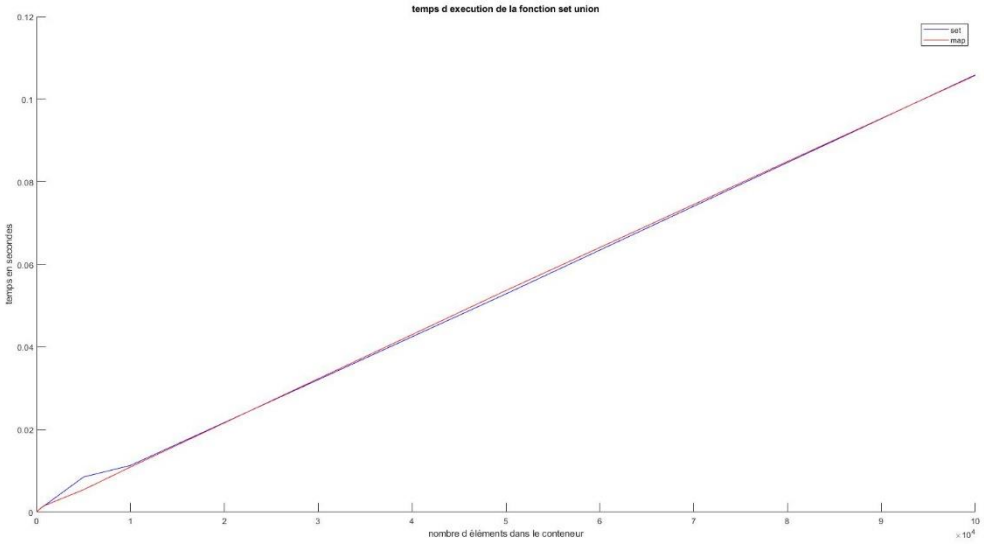


Figure 11: comparaison du temps d'exécution sur la fonction d'union

Enfin, on souhaite comparer le temps d'exécution entre nos deux conteneurs pour la fonction set union. L'`unordered_map` s'avère plus rapide pour un faible nombre d'élément pour exécuter la fonction d'union (nombre d'éléments <1000). Enfin si nos deux conteneurs contiennent plus de 1000 éléments, la fonction d'union requerra le même temps d'exécution, aussi bien que pour `l'unordered_map` que pour `l'unordered _set`.

3) Discussion des résultats

Globalement, les résultats des fonctions sont plutôt similaires pour les deux conteneurs malgré quelques différences. De manière générale, les fonctions évoluent de manière linéaire excepté les fonctions `count`, `swap` et `find`.

Les variations dans les résultats sont probablement matérielles. Autrement dit, des imprécisions provenant de la machine.

Ces types de conteneurs peuvent avoir différentes utilités. Les `unordered set` peuvent servir pour représenter des ensembles mathématiques dans lesquels il n'y a pas d'ordre par exemple. Les `unordered multiset` peuvent remplacer les vecteurs et les tableaux pour leur performance. Les `unordered map` et `unordered multimap` sont des équivalents aux dictionnaires en python.

IV. Associative containers

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
```

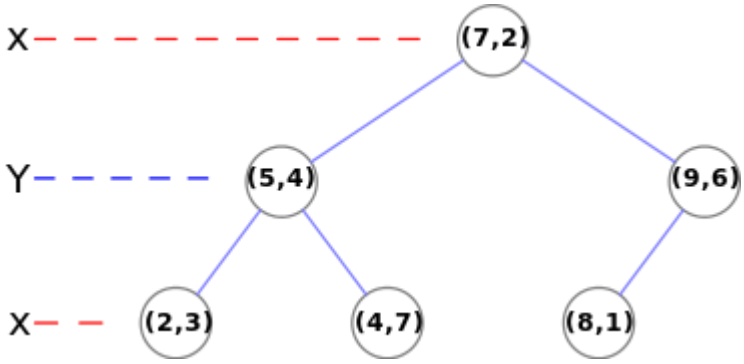
Pour tous les tests effectués, ce sera le tri par default des entiers qui sera utilisé (par ordre croissant).

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));
```

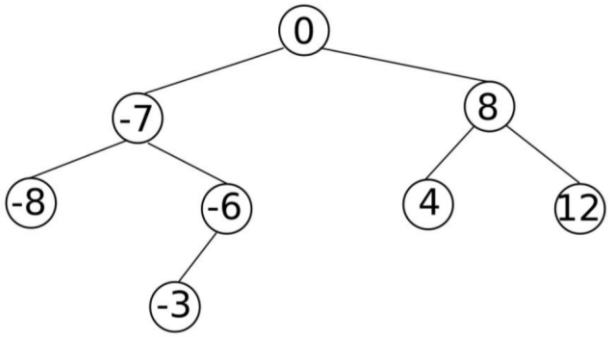
De même il est nécessaire d'utiliser les itérateurs pour lire dans la multimap.

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));
```

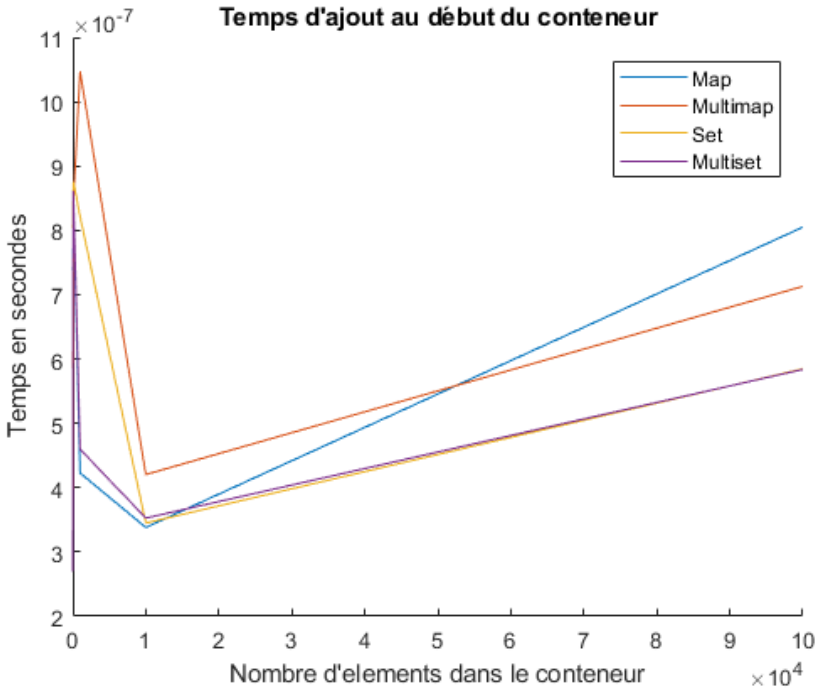
Les fonctions `erase`, `clear`, size les copie ont exactement la même application que pour les multimap.

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

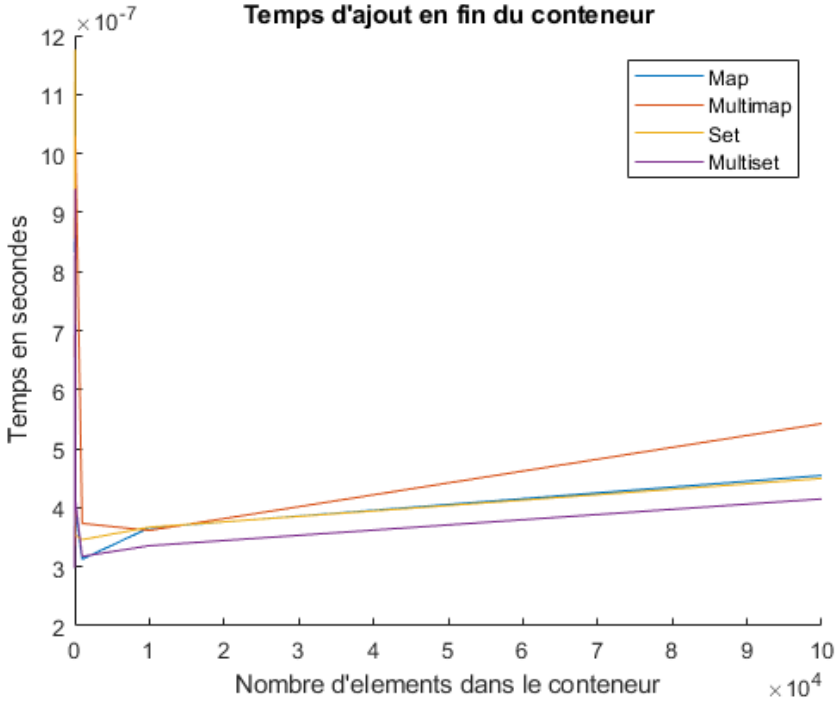
Avec le code de la partie 1 on peut réaliser les courbes d'exécution des différentes fonctions afin de comparer le comportement des différents conteneurs.

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

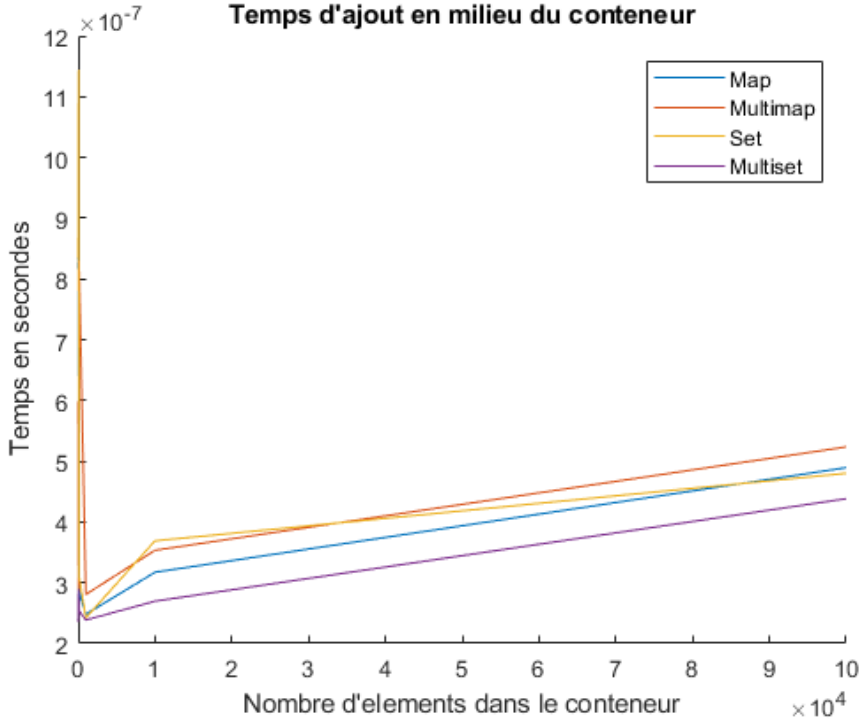
En insérant une petite valeur :



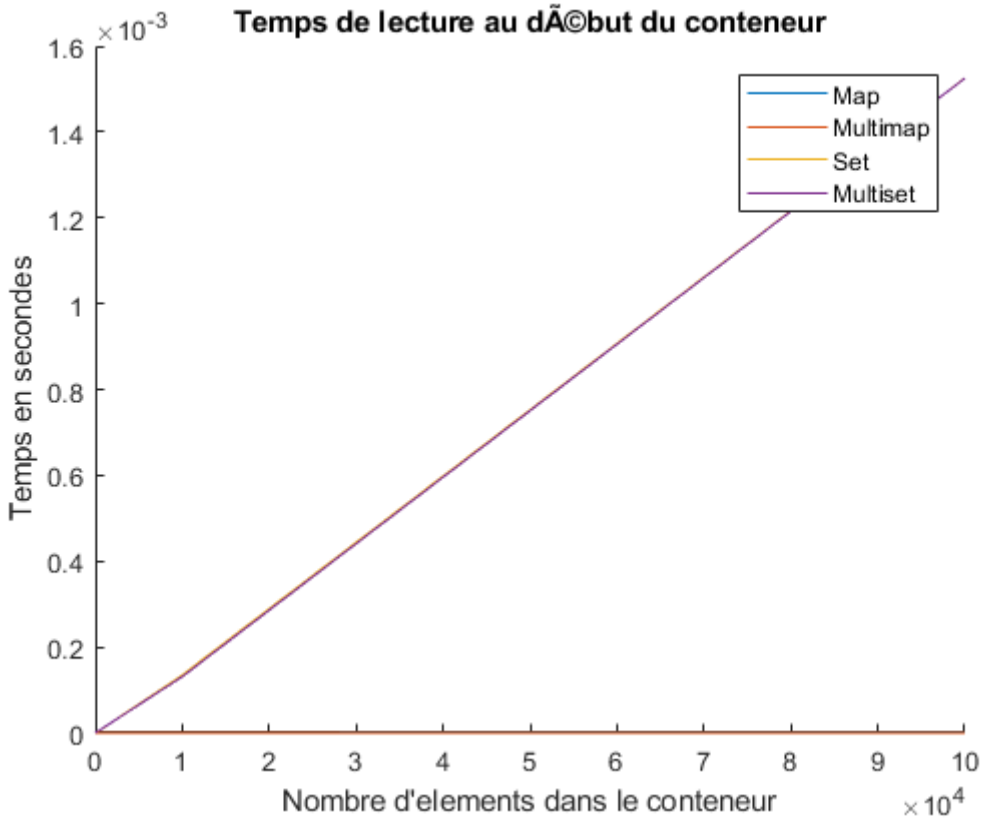
En insérant une grande valeur :



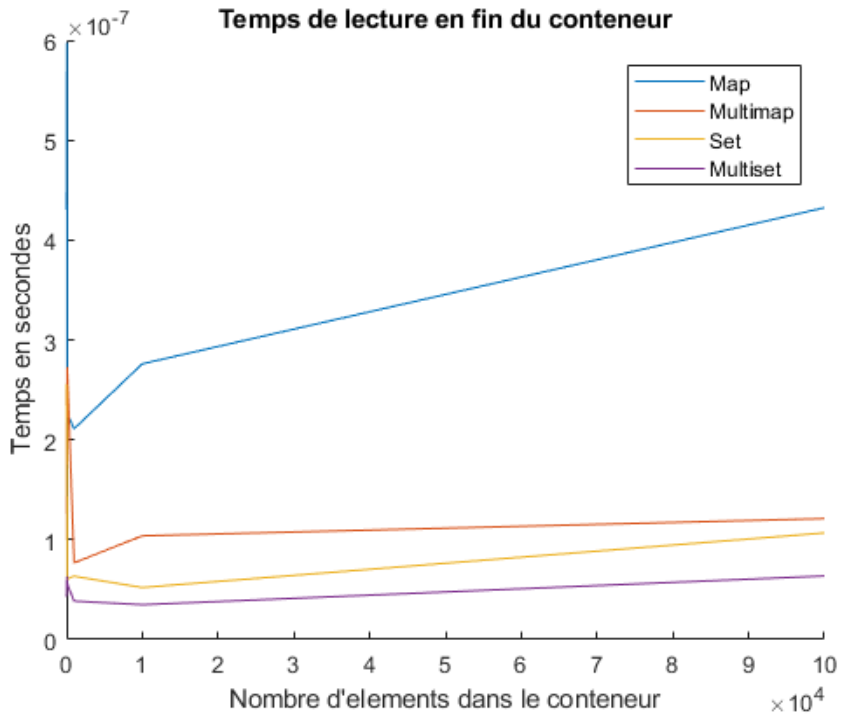
En insérant une valeur moyenne :



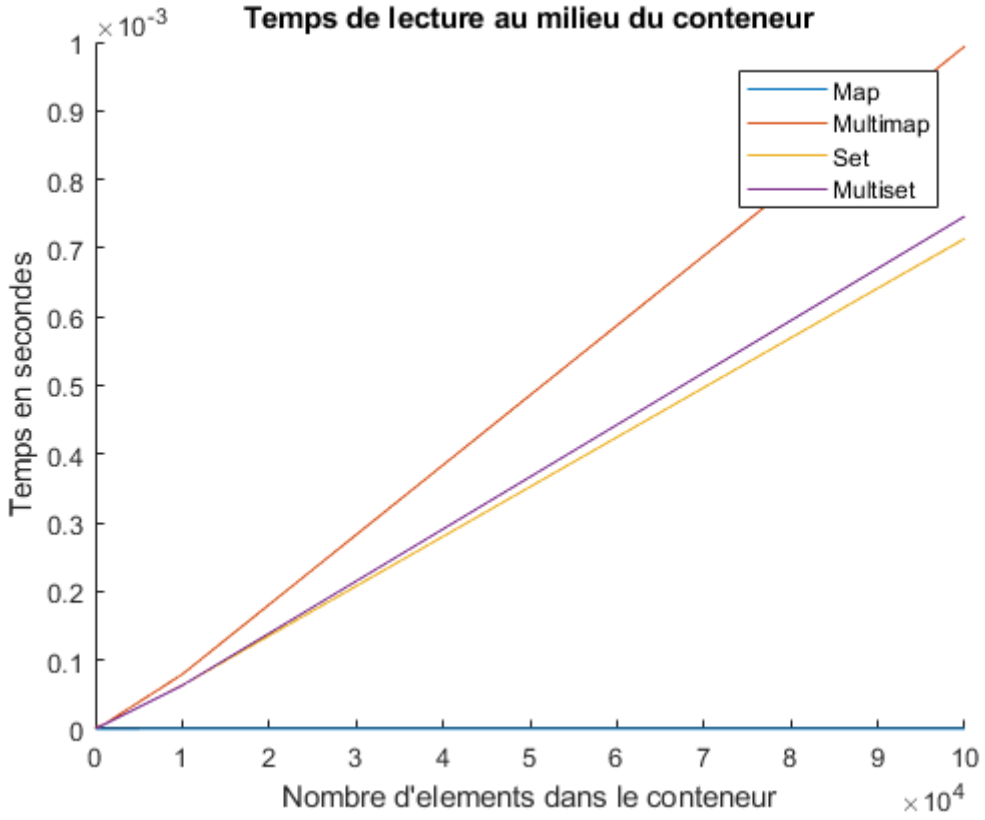
L'étude du temps d'ajout montre que le fonction `insert` utilisé pour les 4 container est relativement équivalente. De plus bien qu'il peut sembler qu'il y a une complexité en O(N) il y a en réalité une complexité en O(log(N)) car l'élément insérer est inséré dans l'arbre.



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).



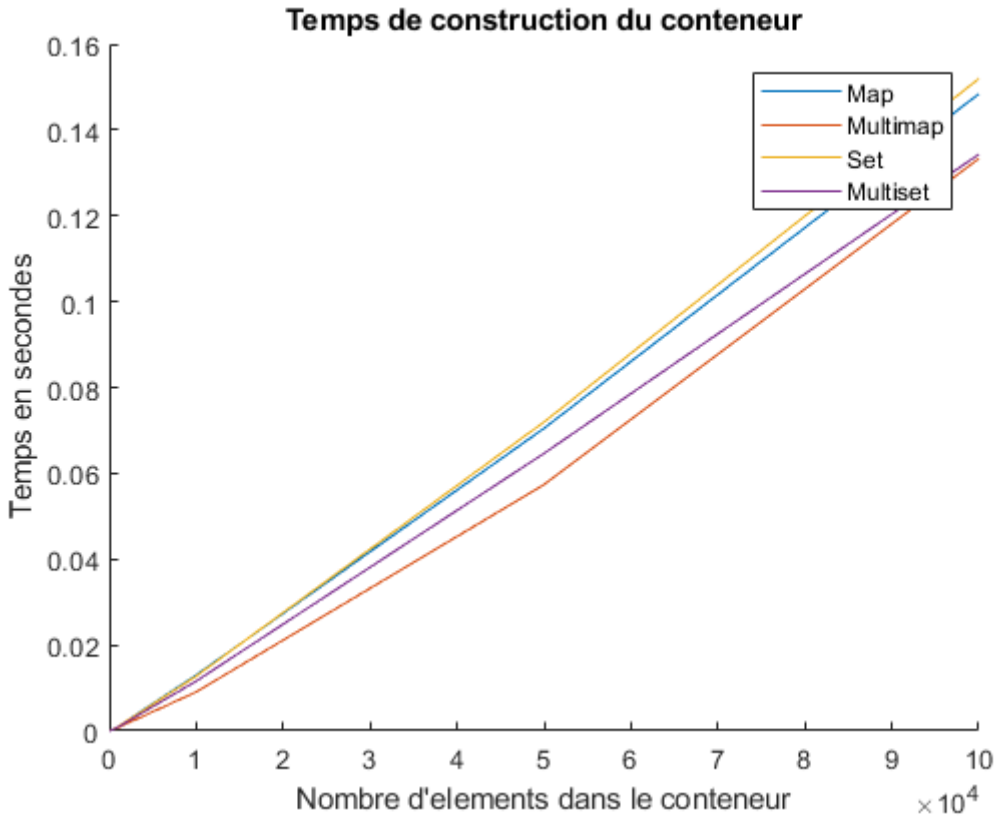
Il semble que pour la lecture en fin du conteneur map est moins rapide bien qu'elles soient toute en O(1). En effet le premier élément du conteneur correspond à l'utilisation de `.rbegin()` sur l'itérateur associé, qui est un peu plus rapide que l'operateur [].



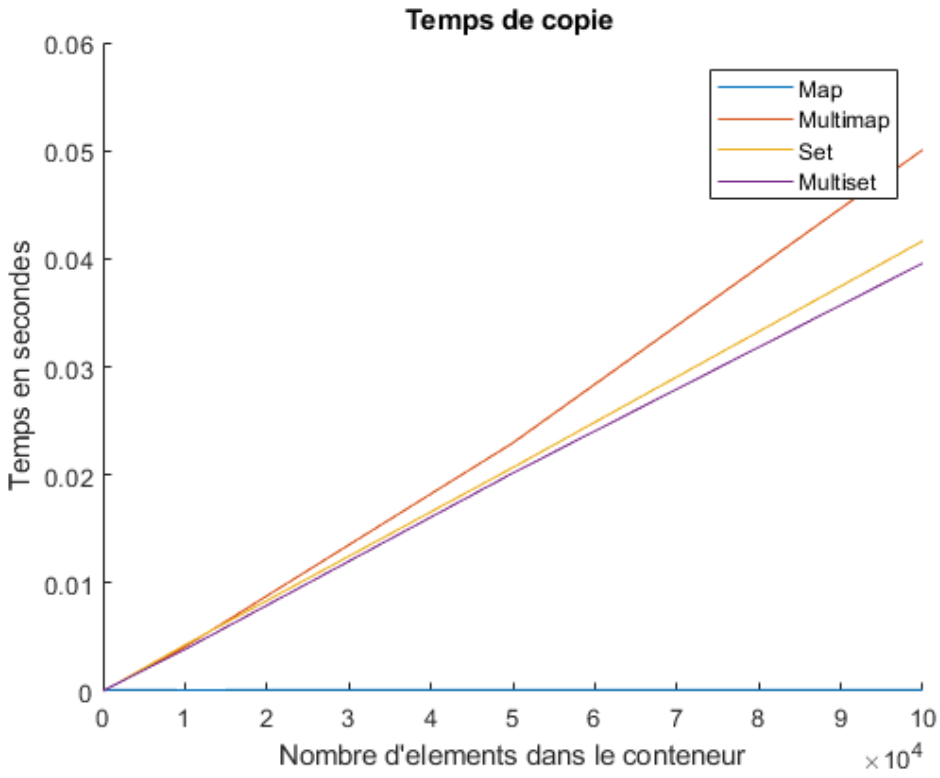
Le résultat montre que les multimap,multiset et set de complexité O(N) alors que la map a une complexité constant en O(1) pour la lecture au milieu du container.

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).

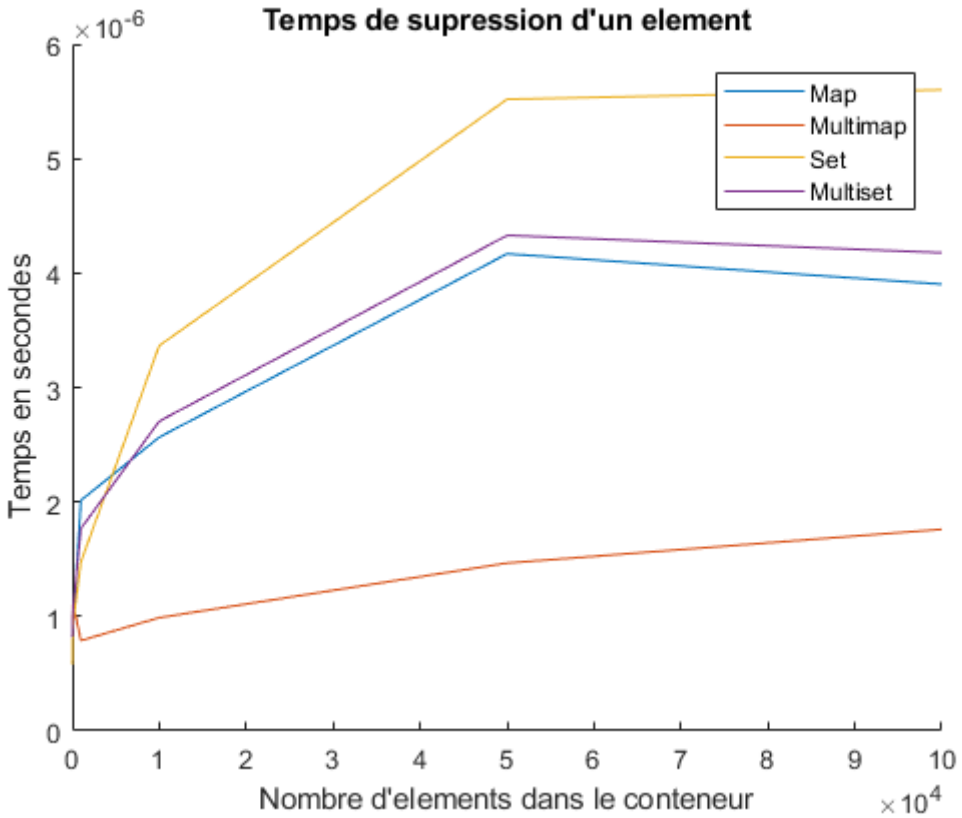
Le calculs du temps d'exécution des fonctions des conteneurs associatifs peut permettre d'évaluer la complexité de celles-ci



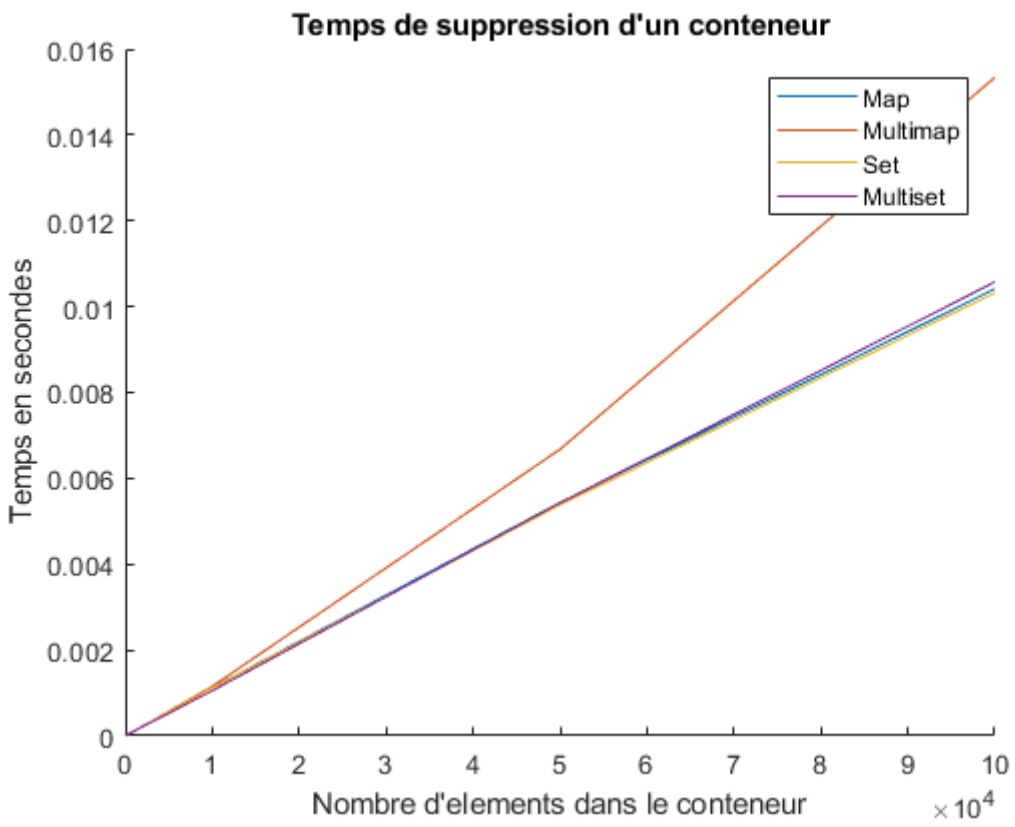
La construction d'un conteneur à l'aide de la fonction `insert` est de complexité O(Nlog(N)). En effet la fonction insert est de complexité O(log(N)) donc étant donné que l'on fait un boucle for nous avons donc bien la complexité voulue.



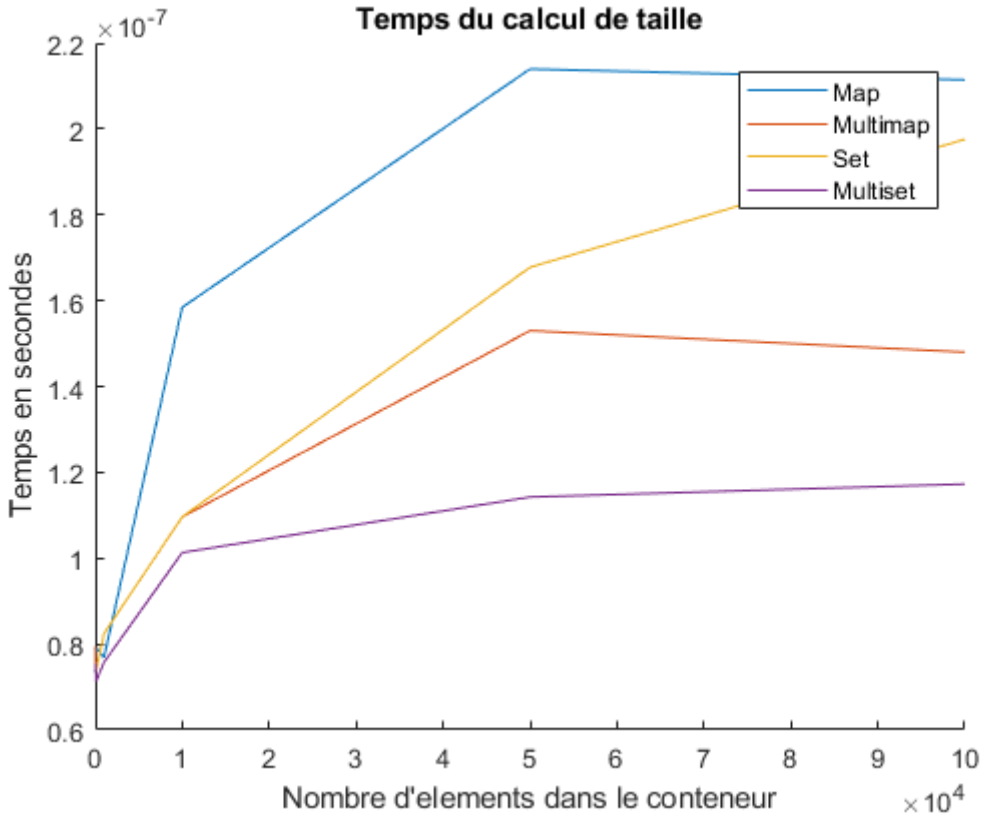
La copie d'un conteneur à l'aide du constructeur a une complexité en O(N) car celui-ci parcourt le conteneur qu'il copie.



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.

5) Interprétation des résultats

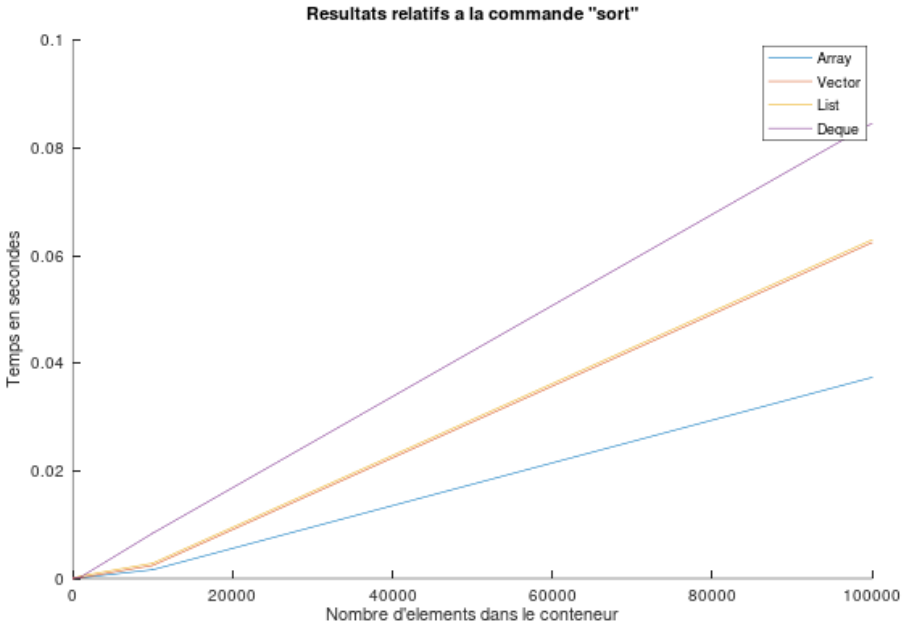
Les conteneurs associatifs ont la particularité d'être très efficace pour rechercher des valeurs dans des grosse base de données (log(N)). ON peut par exemple facilement imaginer l'application de ce conteur pour un dictionnaire ou les clefs peuvent être les mots classés par ordre alphabétique et la valeur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

V. Partie `algorithm` (comparaison inter-familles)

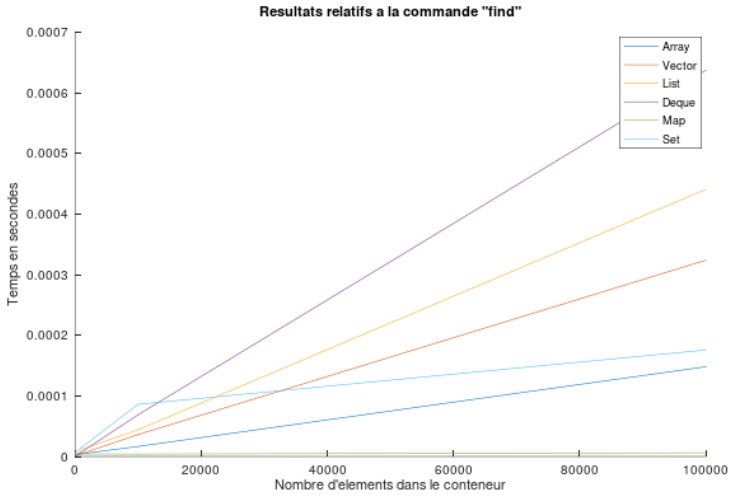
Il est également possible de définir les temps des fonctions de la librairie `algorithm` dans des conteneurs présentés. Les courbes qui vont suivre permettront de comparer les principaux conteneurs et de voir quelles commandes sont les plus optimisées pour les différentes familles. La librairie `algorithm` est composée de nombreuses fonctions :

- Sort : Cette fonction permet à l'utilisateur de trier la portion de conteneur qu'il lui a référencé en argument dans l'ordre croissant.
  Cette fonction n'est pas disponible pour tous les conteneurs de la STL mais essentiellement pour les conteneurs séquentiels. En effet, les autres conteneurs ont déjà un ordre préétabli : les stacks sont en LIFO (Last In First Out), les queues sont en FIFO (First in First Out), les set et les map possèdent des fonctions de triage propres, etc….



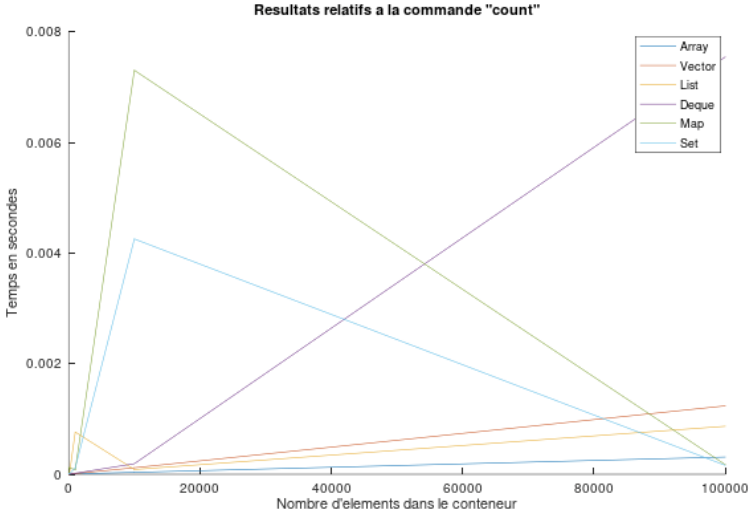
Aussi, il est alors notable que cette méthode possède une complexité en O(N*log(N)). Il existe effectivement de nombreux algorithm permettant de trier des lists entre autres, et l'un des plus connu est le tri fusion, qui lui-même est en O(N*log(N)). Il est particulièrement difficile d'avoir une complexité meilleure que celle-ci dans le sens où pour trier un conteneur, il faut forcément le parcourir au moins une fois. Dans le cas des conteneurs présents, cette complexité est bien visible avec l'augmentation du nombre d'éléments dans le conteneur, car cette méthode met un temps compris entre quelques µs à quelques centaines de ms avant de donner un résultat lorsque le nombre d'élément est particulièrement important, et ce pour tous les conteneurs.

- find : Cette fonction permet de trouver l'itérateur de l'élément spécifié en argument de la fonction et compris dans une portion de conteneur spécifiée elle aussi en argument.



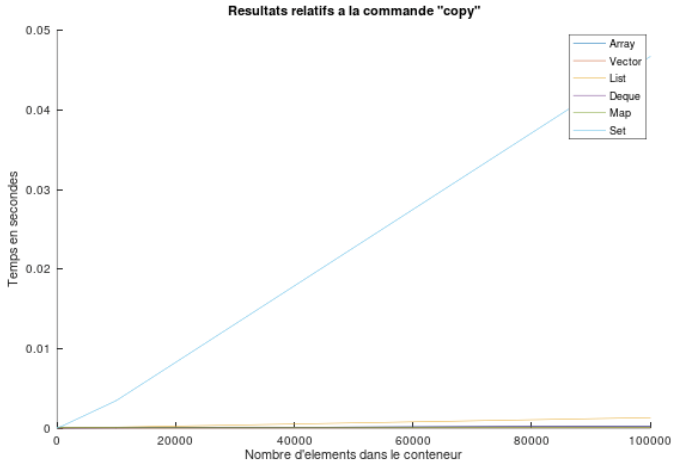
Cette fonction est de complexité linéaire (soit en O(N)) ce qui est particulièrement intuitif car il est nécessaire de parcourir au moins une fois la liste afin de savoir si l'élément lui appartient ou non. Il est alors à noter que cette complexité est vraie pour tous les conteneurs exceptés pour les sets. En effet ceux-ci ayant leur contenu déjà trié, il est beaucoup plus aisé de trouver si un élément appartient ou non à ce conteneur. C'est pourquoi, pour ce conteneur en particulier la complexité de la fonction apparait comme étant constante (en O(1)).

- count : Cette fonction permet de compter le nombre d'éléments dans une plage spécifiée qui est égale à un nombre donné.



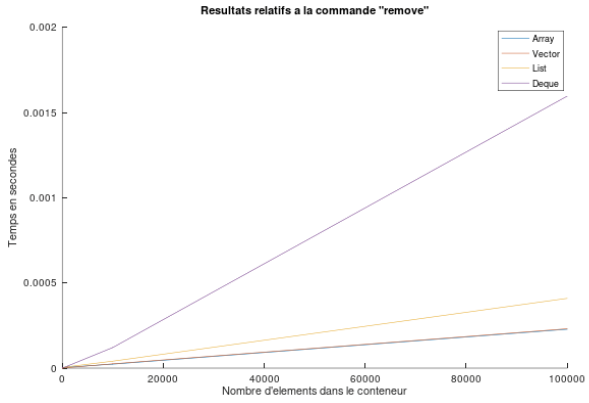
De nouveau il paraît très intuitif que cette fonction soit de complexité linéaire (en O(1)) car il est nécessaire de parcourir l'ensemble du conteneur pour pouvoir compter le nombre d'éléments présents dans la liste. Ceci est bien vérifié par l'ensemble de nos conteneurs ci dessus. Il est tout de même notable qu'une forte diminution apparaît pour les set et les map pour 100 000 éléments. Ceci peut être dû au fait qu'aucun élément n'était à compter ce qui se solde alors par une diminution très vive du temps : en effet il est à rappeler que les set et les map associent un identifiant à chaque élément, donc ils savent si l'élément recherché existe ou non.

- copy : Cette fonction permet de copier les éléments d'un conteneur dans un autre conteneur.



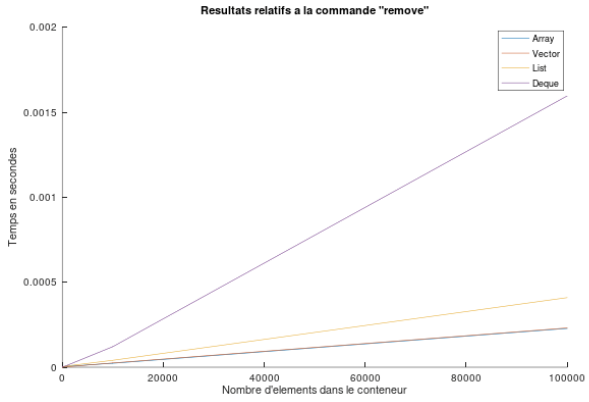
Comme pour les autres fonctions, celle-ci est aussi en complexité linéaire (en O(N)) car pour pouvoir copier l'ensemble d'un conteneur, il faut le parcourir au moins une fois. Aussi il est notable que les sets prennent beaucoup plus de temps lors de la copie car ce conteneur trie chaque élément qui lui est ajouté. Cette fonction n'échappe pas à la règle et le temps de tri peut prendre un certain temps qui implique donc que cette fonction en est rallongée.

- remove : Cette fonction a pour but de transformer une plage spécifiée en une autre plage avec des éléments supprimés lorsque ceux-ci sont égaux à une valeur donnée.



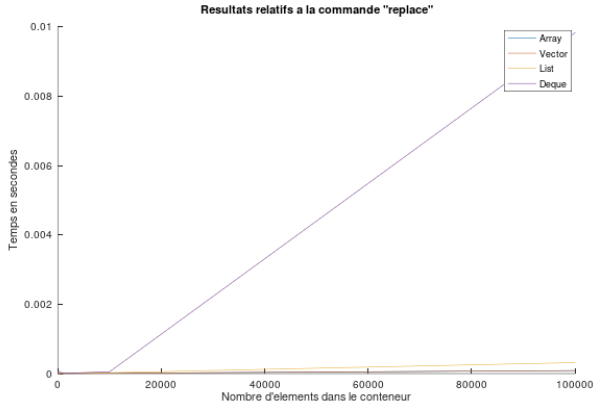
Il est à noter que cette fonction n'est pas déterminée pour l'ensemble des conteneurs, il s'agit là essentiellement des conteneurs séquentiels ; les stacks et les queues ont formellement interdiction de l'utiliser : cela irait à l'encontre de leur principe même et les map et set ont des méthodes propres.
Sachant que la fonction retire tous les éléments spécifiés en argument, il est alors normal de nouveau que sa complexité soit linéaire.

- remove : Cette fonction a pour but de transformer une plage spécifiée en une autre plage avec des éléments supprimés lorsque ceux-ci sont égaux à une valeur donnée.



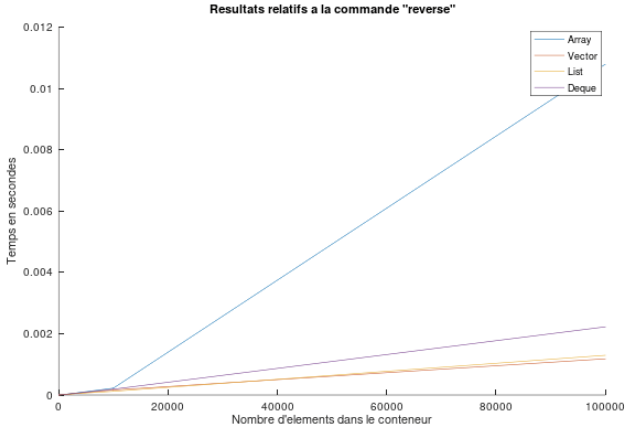
Il est à noter que cette fonction n'est pas déterminée pour l'ensemble des conteneurs, il s'agit là essentiellement des conteneurs séquentiels ; les stacks et les queues ont formellement interdiction de l'utiliser : cela irait à l'encontre de leur principe même et les map et set ont des méthodes propres.
Sachant que la fonction retire tous les éléments spécifiés en argument, il est alors normal de nouveau que sa complexité soit linéaire.

- replace : Cette fonction assigne, dans une plage spécifiée, une nouvelle valeur à toutes les valeurs égales à une valeur donnée.



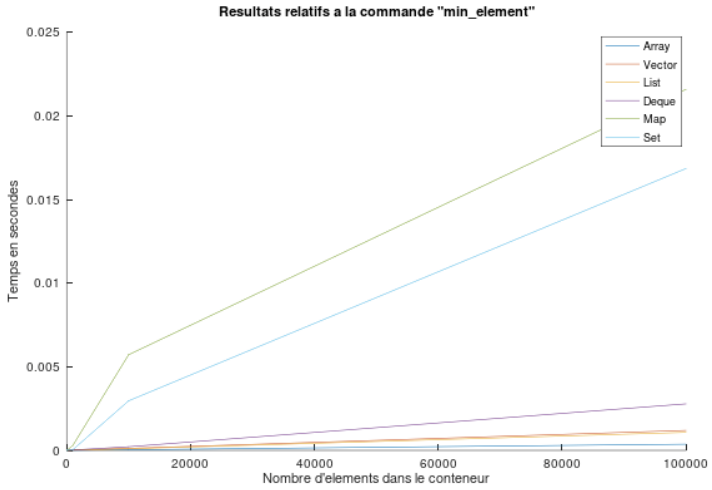
Cette fonction possède une complexité en O(N) car combine essentiellement deux actions : trouver l'élément à remplacer et le remplacer. Dans le cas présent où seuls les conteneurs séquentiels sont étudiés, puisqu'aucun d'entre eux n'a d'opération supplémentaire à effectuer avant d'ajouter un élément, la complexité de replace est donc commune à la complexité de find soit en O(N)

- reverse : Cette fonction permet d'inverser tous les éléments contenus dans un plage donné.



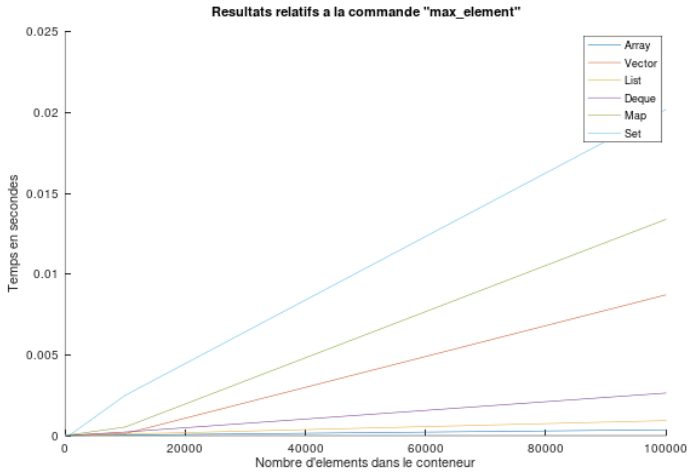
De nouveau, cette fonction n'est pas adaptable pour les autres conteneurs que les conteneurs séquentiels, pour des raisons de tri ou encore pour des raison d'ordre d'entrée imposé (type stack ou file). La complexité de cette fonction est également en O(N) car il suffit simplement d'utiliser un itérateur reverse pour pouvoir lire le conteneur dans le sens inverse et donc parcourir l'ensemble de la liste, ainsi la complexité est la même que la fonction copy. Il est notable que tous les conteneurs ne possédant pas d'itérateur reverse mettra beaucoup plus de temps avant de pouvoir s'effectuer.

- min_element : Cette fonction renvoie la plus petite des valeurs du conteneur



Cette fonction nécessite de nouveau le parcours entier du conteneur pour pouvoir s'exécuter. C'est pourquoi il est tout aussi intuitif de confirmer que cette fonction s'exécute avec une complexité en O(N). Il est tout de même important de noter que les set qui pourtant sont déjà ordonnés par ordre croissant mettent un temps plus élevé que les conteneurs séquentiels avant de trouver leur minimum.

- max_element : Cette fonction renvoie la plus grande des valeurs du conteneur.



Cette fonction fonctionne exactement selon le même principe que la fonction précédente.

Conclusion

À la suite de cette large étude, l'ensemble des principaux conteneurs de la STL ont pu être étudié, comparé, testé, mais que retirer de cette étude ?
Le principal point à retenir pour cette étude est qu'un conteneur ne se choisit pas au hasard. En effet, lors du choix d'un conteneur, il est nécessaire au préalable de se poser les questions suivantes :

- Ai-je besoin d'une taille limite ?
- Ai-je besoin de trier mon conteneur ?
- Ai-je besoin d'un accès rapide au début, à la fin, n'importe où ?
- Ai-je besoin de faire correspondre à chaque élément un identifiant propre ?
- L'ordre de mes éléments est-il important ?

La réponse à chacune de ces questions permet alors de s'orienter plus aisément sur un conteneur précis et ainsi être sûr de faire le bon choix. Et attention au surplus de libertés ! En effet, avoir la possibilité d'utiliser des fonctions est très agréable pour le programmeur, mais ceci coûte aussi beaucoup en mémoire. Aussi, dans un souci d'optimisation il sera préférable de choisir un conteneur en remplaçant tous les "Ai-je besoin" des questions précédentes pas des "Est-ce fondamental d'avoir". Ceci fait toujours revenir le programmeur à l'éternel compromis optimisation en mémoire - optimisation en temps de calcul.

Répartition des tâches :

Conteneurs de séquence : BRALET Antoine, METEYER Paul
Adaptateurs de conteneurs : BENBRIKHO Mustapha, COURMACEUL Sébastien
Conteneurs associatifs : DURET Guillaume, PALLAS Léo
Conteneurs associatifs non ordonnés : GEDEON Benjamin, BURGEVIN Valentin, DUCRAY Victor,