R5.A.04 - Qualité algo. (Systèmes de recherche) – Cours

Le but de ce cours est de présenter, pour des problèmes courants en programmation, les techniques permettant d'estimer leur complexité et de trouver l'algorithme le plus efficace pour résoudre ces problèmes. Nous verrons également comment le choix des structures de données utilisées pourra être déterminant pour améliorer grandement l'efficacité de ces algorithmes.

La première étape présentera la complexité algorithmique, en expliquant son intérêt dans la programmation.

La deuxième détaillera les différentes structures de données proposées par la STL disponible avec le langage de programmation C++, et la complexité de chacune par rapport aux opérations courantes (recherche, insertion, suppression).

La troisième donnera un tableau comparatif des différentes structures de données.

Remarque : la première partie contenant uniquement du cours est longue : environ 1h30 avant le 1er exercice !

Complexité d'un problème

En informatique, on cherche toujours à écrire un algorithme qui soit le plus efficace possible pour résoudre un problème.

Cette efficacité se mesure avec la quantité de ressources utilisées en temps ou en mémoire. La plupart des cas, ce sont les ressources en temps qui sont analysées, car les mémoires modernes sont suffisamment grandes pour éviter des problèmes de stockage.

Dans ce module, nous nous intéresserons à ce que les algorithmes que nous écrirons consomment le moins de temps possible.

Comment mesurer cela ?

La technique consiste à calculer, pour un algorithme, le nombre d'instructions élémentaires (addition, multiplication, affectation, comparaison…) dans le « pire des cas », c'est-à-dire le nombre d'instructions maximal que pourrait générer cet algorithme.

On notera « C(P) » la complexité du problème « P ». Cette complexité sera le minimum des complexités des différents algorithmes résolvant le problème. C(P) = min {C(A1), C(A2), … , C(AN)}.

Exemple : Recherche d'un élément dans une liste triée d'entiers

Considérons une liste de N entiers sous forme de tableau, triée du plus petit élément au plus grand. L'accès à tout entier du tableau nécessite 1 instruction élémentaire. On dit que l'ordre de grandeur d'accès à toute entrée est « O(1) » (complexité constante).

Algorithme « A1 »

```
bool A1(int list[], int size, int x)
{
    for(int i= 0; i < size; i++)
    {
        if(list[i] == x)
        {
            return true;
        }
    }

    return false;
}
```

Question : Calculer le nombre d'instructions élémentaires de ce programme.

Calcul du nombre d'instructions élémentaires dans le pire des cas :

C(A1) = 1 + N * (1 + 2 + 2) + 1 = 5N +2

Si on a 1000 éléments dans la liste, on voit qu'il faudra au pire 5002 instructions.

On parle dans ce cas de complexité est de l'ordre de N, autrement dit "O(N)". On ne prend pas en compte les constantes multiplicatives (ici le facteur 5) pour déterminer un ordre de grandeur car la différence entre N et 5N est négligeable par rapport à la différence entre N et N^2 , surtout avec de très grandes valeurs de N.

C(A1) = O(N)

Algorithme « A2 »

Nous pouvons proposer un algorithme plus efficace étant donné que le tableau est trié :

```
bool A2Recursive(int list[], int iBeg, int iEnd, int x)
{
    if(iBeg > iEnd)
    {
        return false;
    }

    int iMid = (iBeg + iEnd) / 2;
    if(list[iMid] == x)
    {
        return true;
    }

    if(list[iMid] > x)
    {
        return A2Recursive(list, iBeg, iMid - 1, x);
    }

    return A2Recursive (list, iMid + 1, iEnd, x);
}
bool A2(int list[], int size, int x)
{
    return A2Recursive(list, 0, size - 1, x);
}
```

Calcul du nombre d'instructions élémentaires dans le pire des cas :

C(A2) = 1 + (1 + 3 + 2 + 2) + (1 + 3 + 2 + 2) + ... (au total, "log2(N)" fois)

Nous voyons qu'à chaque itération, nous réduisons la taille de la liste de moitié, ce qui réduit automatiquement le nombre d'itérations.

On parle dans ce cas de complexité est de l'ordre de log(N), autrement dit "O(log(N))".

$$C(A2) = O(\log(N))$$

Si on a 1000 éléments dans la liste, il faudra environ 10 itérations car log2(1000) ~= 9.97.

Conclusion de l'exemple : complexité du problème « Recherche d'un élément dans une liste triée »

Comme expliqué dans la définition précédemment, on considère donc que la complexité de ce problème est de l'ordre de log (N).

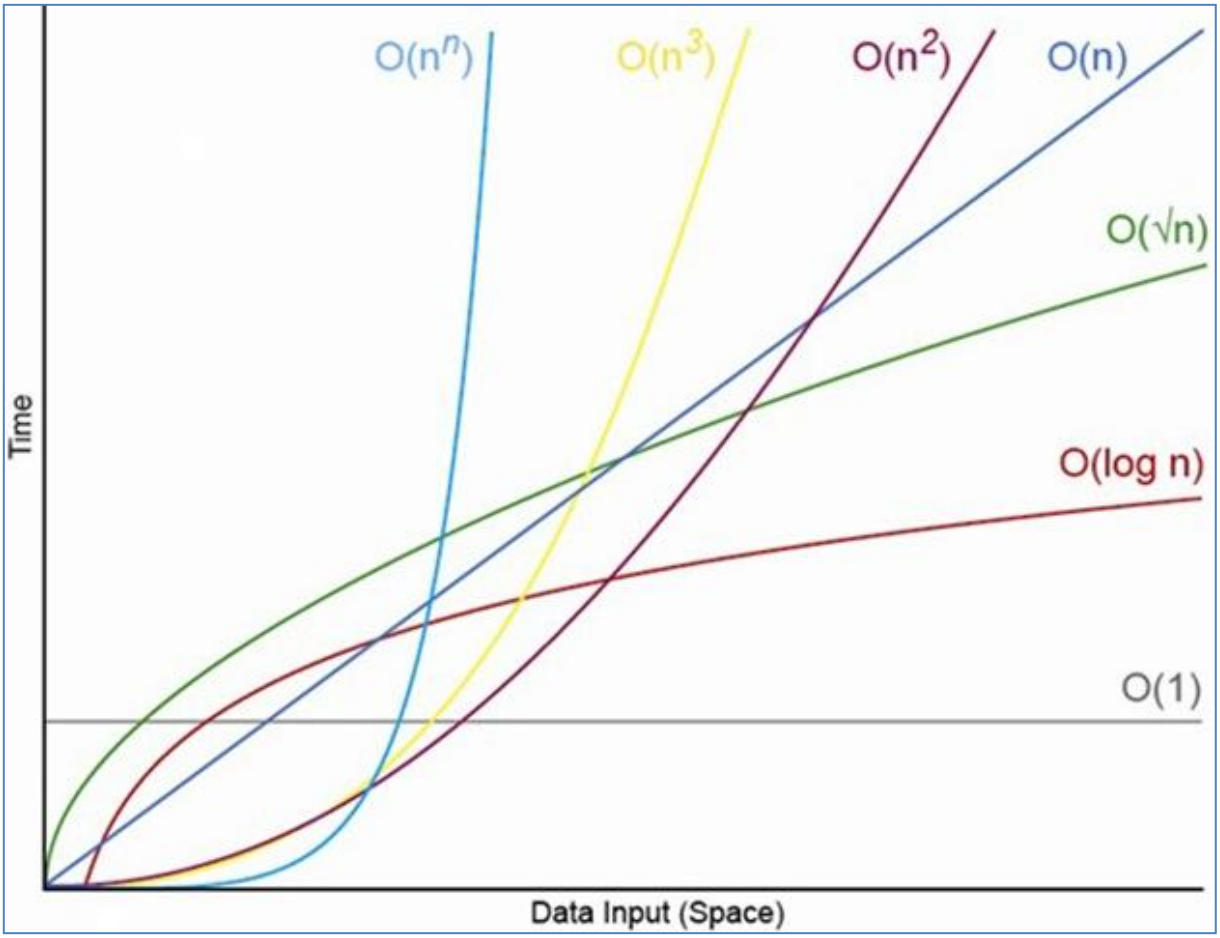
$$C(P) = O(\log(N))$$

Remarque importante à propos des performances d'un programme

La question des performances nécessite de définir avant tout ce que l'on entend par programme performant en informatique. En effet, la seule rapidité du code compilé résultant n'est pas un critère unique pour évaluer un programme. Il est nécessaire de prendre également en compte la qualité du code C++, c'est-à-dire sa facilité de compréhension, de maintenance et de réutilisation, ainsi que la compatibilité du code avec les différents compilateurs disponibles.

La STL que nous étudions dans ce module est un code C++ standard et efficace, permettant ainsi un code source facilement lisible et réutilisable, et un code compilé rapide. L'utiliser contribue donc à de bonnes performances du programme en termes de qualité de code, en plus du bon choix de ses structures de données.

Graphique comparant les différents ordres de grandeur



Complexité	Appellation
O(1)	Constante
O(N)	Linéaire
O(log(N))	Logarithmique
O(N^k)	Polynomiale
O(e^N)	Exponentielle

C++, STL (Standard Template Library) et complexité

Nous avons vu dans les exemples ci-dessus l'utilisation de listes sous forme de tableaux d'entiers. Cette structure de données est la plus basique pour manipuler des listes, mais nous nous rendons compte qu'il est fastidieux de l'utiliser car toute opération comme l'augmentation de la capacité de la liste, l'insertion ou la suppression d'un élément au milieu de la liste demande d'écrire de nombreuses lignes de code, et nécessite un nombre important d'instructions.

La Standard Template Library (STL), disponible avec le langage C++, propose un large choix de structures de données. Nous verrons également des éléments proposées par Boost qui complètent STL dans le même esprit. Nous allons étudier les principales structures, et donner pour chacune la complexité des opérations courantes (recherche, insertion, suppression…).

L'aide en ligne du langage C++, incluant la STL, est disponible ici : https://en.cppreference.com/w/cpp. Elle est très bien faite, et propose de nombreux exemples. Privilégiez la version anglaise qui est plus complète et propose plus d'exemples par rapport à la version française.

Pourquoi utiliser C++ ?

C++ est un langage de programmation compilé permettant la programmation sous de multiples paradigmes (comme la programmation procédurale, orientée objet ou générique). Ses bonnes performances, et sa compatibilité avec le C en font un des langages de programmation les plus utilisés dans les applications où la performance est importante.

Modules de l'IUT liés au langage C++ et à ce module

- C++ : Notion de classe, constructeurs, chaînes de caractères et allocation dynamique, héritage, méthodes et classes abstraites (pas de programmation générique) : en première année avec Yon Dourisboure (certaines diapos de ce cours sont extraites de son cours).
- STL : en deuxième année avec Philippe Lopistéguy : utilisation de « std::map » et programmation générique (pas sous le spectre de la complexité).
- Principes de programmation SOLID : en deuxième année avec Philippe Lopistéguy.
- Complexité dans les modules précédents de l'IUT : Pantxika Dagorret et Patrick Etcheverry.

Programmation générique

Le « T » de STL correspond à « Template », ce qui indique qu'elle utilise la programmation générique.

Programmation générique

- Considérons les deux fonctions suivantes :

```
void swap(int& a, int& b) {int tmp = a ; a = b ; b = tmp; }
void swap(string& a, string& b) {string tmp = a ; a = b; b = tmp; }
```

- Elles utilisent le même principe, seul le type des deux paramètres (et celui de tmp) changent.
- Cette double écriture est une perte de temps, une source d'erreur, et révèle que la méthode d'échange ne dépend pas du type de ce qu'on échange.
- Il est possible d'écrire une seule fois la fonction swap de manière à ce qu'elle soit utilisable pour échanger toute paire de variables pour lesquelles l'opérateur d'affectation a été défini grâce aux patrons de fonction.

Remarque : les diapos intégrées dans ce cours sont extraites du cours du module « R2.01 Développement objet en C++ créé par Yon Dourisboure. Celles concernant la programmation générique et STL ne sont plus au programme en première année et sont donc extraites d'une ancienne version de ce cours.

Patrons/Modèles/Templates de fonctions

Swap.h

```
template <typename T>
void swap(T& a, T& b){
      T tmp=a;
      a=b;
      b=tmp;
}
```

Main.cpp

```
#include "swap.h"
[...]
int m=10, n=12;
swap<int>(m, n);
string s("toto"), t("tata");
swap<string>(s, t);
Personne p1("Tom"), p2("Luc");
swap<Personne>(p1, p2);
```

- **Définitions** (et pas seulement déclaration) dans le fichier ".h".
- L'exemple est valable pour tout type T où T& T::operator=(const T&) existe.
- Le compilateur crée (instancie) autant de fonctions différentes que nécessaire. Dans l'exemple il crée 3 fonctions swap<int>, swap<string> et swap<Personne>.
- Attention la partie <---> fait partie du nom de la fonction !

Remarque : Nous parlerons en détail de la surcharge d'opérateurs dans une section spécifique de ce cours.

Patrons de classes

Tablo.h

```
template < typename T, int n >
class Tablo {
private:
      T elements[n];
      [...]
};
// Définition des méthodes
[...]
```

Main.cpp

```
#include "Tablo.h"
[...]
Tablo<int,10> t1 ;            // t1 OK
const int max=10 ;
Tablo<Personne,max> t2 ; // t2 OK
int max2=10 ;
Tablo<char,max2> t3 ;  // t3 ERREUR
```

- Même idée, mais pour générer des classes et non des fonctions
- Dans l'exemple, on crée des « tableaux » de dimension fixe mais quelconque. Le cellules de ces vecteurs peuvent être de type quelconque.
- **Définition** du patron dans le ".h" et instanciation dans les ".cpp"
- Le compilateur crée/instancie autant de classes différentes que nécessaire.
- Ici aussi la partie <---> fait partie du nom de la classe : Tablo n'est pas un nom de classe mais Tablo<int,5> oui !
- Les paramètres de types doivent être connus au moment de la compilation.
- Les paramètres qui ne sont pas des types (ici n) doivent être des expressions constantes dont la valeur est connue au moment de la compilation.

Question : Pourquoi la dernière ligne de code affichée génère-t-elle une erreur de compilation ?

Patrons de classes : définition des méthodes

Tablo.h

```
template < typename T, int n >
class Tablo {
private:
      T elements[n];
public:
      T getElt(int i) {
            return this->elements[i];
      }
};
```

Tablo.h

```
template < typename T, int n >
class Tablo {
private:
      T elements[n];
public:
      T getElt(int);
};
template < typename T , int n >
T Tablo<T,n>::getElt(int i) {
      return this->elements[i];
}
```

- La définition des méthodes doit toujours être dans le ".h"
  - Soit directement dans la classe.
  - Soit après la classe en spécifiant le contexte d'application avec le nom complet de la classe.

Bibliothèque de modèles standards

Standard Templates Library (STL)

- Contient une série de patrons prêts à l'emploi.
- Conteneurs de séquence :
  - vector<> : Tableau "autoredimensionnable" avec insertions et suppressions rapides à l'arrière, accès direct à tout élément.
  - deque<> : Insertions et suppressions rapides à l'arrière et à l'avant, accès direct à n'importe quel élément.
  - list<> : Liste doublement chaînée, insertions et suppresions rapides n'importe où.
- Conteneurs associatifs :
  - set<> : ensemble ordonné, recherche rapide par clef.
  - multiset<> : comme set mais avec des doublons.
- Adaptateur de conteneur :
  - stack<>, queue<> : Lifo, Fifo.
  - priority_queue<> : L'élément de plus grande priorité est le 1er sorti.
- **Et bien d'autres choses encore…**

Questions :

- Que signifient « Lifo » et « Fifo » ? Donnez un exemple de chaque, et les dessiner au tableau.
- Connaissez-vous d'autres structures STL ?

Toutes les structures de données proposées par la STL commencent par le préfixe « std:: » (par exemple « std::vector »), « std » est le nom de l'espace de noms dans lequel ces structures sont définies.

Si vous souhaitez éviter d'écrire "std::" devant tout objet de la STL, écrivez au début de votre fichier C++ :

```
using namespace std;    // Pour éviter d'écrire "std::" devant tout objet de la STL.
```

Les structures linéaires

Une structure est dite linéaire si l'on peut concevoir une fonction (dite "successeur") qui à chacun de ses éléments sauf le dernier associe un unique autre élément de la même structure (l'élément suivant). De plus, on doit pouvoir parcourir l'intégralité de la structure à l'aide de cette fonction sans visiter deux fois le même élément.

Une conséquence directe de cette définition est que l'on peut s'imaginer et représenter le contenu d'une telle structure de façon linéaire. Commençons par la structure linéaire la plus simple.

Les tableaux

Eléments contigus en mémoire. Il suffit de connaître l'adresse du 1er élément pour accéder directement à tous les éléments. Par exemple, les tableaux d'entiers que nous avons vus au début de cours.

std::vector

La structure STL la plus proche des tableaux utilisé précédemment est « std::vector ». Il est défini dans le fichier STL « vector ».

Le patron vector<>

Tableau homogène "autoredimensionnable"

- Exemple :

```
vector<int> tab(8); // semblable à int tab[8]
```

- Comme pour un tableau, accès possible avec "[ ]" :

```
tab[2] = 10;
```

- Accès au nombre d'éléments :

```
tab.size();
```

- Ajout en fin de tableau :

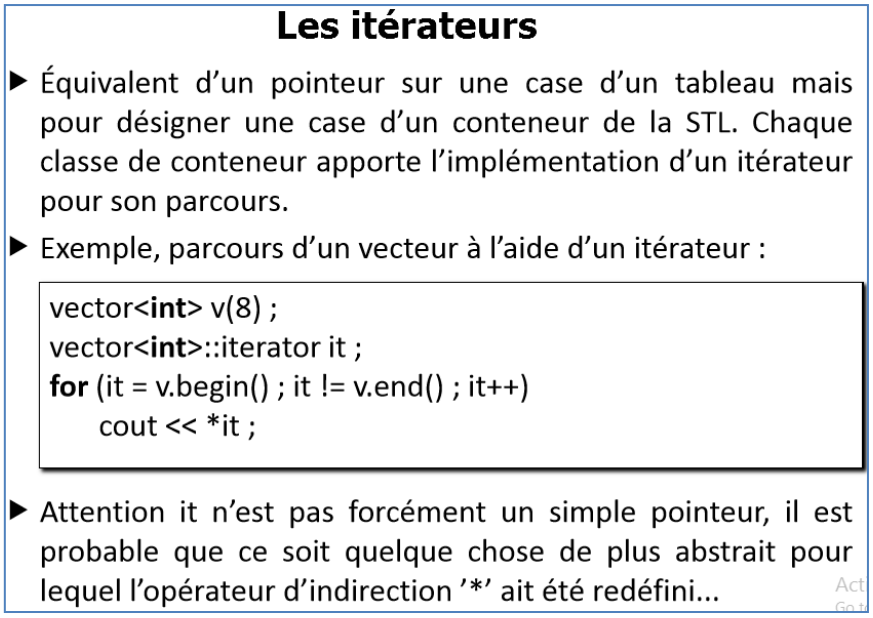
```
tab.push_back(11);
```

- Suppression du dernier élément :

```
tab.pop_back();
```

Les itérateurs

Pour aller plus loin dans la présentation de cette structure, nous aurons besoin de présenter la notion d'itérateur fournie également par STL.



Les itérateurs sont utilisables avec toutes les structures de la STL.

Il existe 4 types d'itérateurs :

- `iterator`
- `const_iterator`
- `reverse_iterator`
- `const_reverse_iterator`

Un `const_iterator`, contrairement à un `iterator`, donne un accès uniquement en lecture à l'élément "pointé".

Avec une structure "non const" (accessible en lecture et en écriture), on peut utiliser tout itérateur.

Avec une structure "const" (accessible en lecture seule), on peut utiliser seulement un des deux itérateurs « const ».

`reverse_iterator` et `const_reverse_iterator` sont l'équivalent des 2 premiers, mais en parcourant les éléments de la fin vers le début (au lieu du début vers la fin). Ils sont rarement utilisés, alors que les deux premiers le sont beaucoup.

Méthodes, disponibles sur chaque structure STL, liées aux itérateurs
« begin () » retourne un itérateur vers le 1er élément.

« end() » retourne un itérateur vers l'élément « après » le dernier élément.

Exemple : parcours d'un vecteur à l'aide d'un d'itérateur (voir diapo précédente).

Méthodes, disponibles sur chaque structure STL, liées aux reverse itérateurs
« rbegin () » retourne un itérateur vers le dernier élément.

« rend () » retourne un itérateur vers l'élément « avant » le 1er élément.

Exemple d'utilisation d'un « const_reverse_iterator » :

```
#include <vector>           // std::vector ()
#include <iostream>         // std::cout
using namespace std;        // Pour éviter d'écrire "std::" devant tout objet STL.
vector<int> v(3);
v[0] = 30;
v[1] = 15;
v[2] = 40;
for(vector<int>::const_reverse_iterator crit = v.rbegin (); crit != v.rend (); crit++)
{
    cout << * crit << endl;
}
// Affiche : "40", "15", "30"
```

Avertissement en cas de modification d'une structure lors de son parcours via itérateur

Lors du parcours d'une structure STL en utilisant un itérateur, dans une boucle « for » par exemple, faites très attention si vous appelez des méthodes d'ajout/suppression d'éléments, car cela pourrait entraîner l'invalidité d'un itérateur en cours d'utilisation.

```
vector<int> v (3);
v[0] = 30;
v[1] = 15;
v[2] = 40;
for(vector<int>::const_iterator cit = v.begin(); cit != v.end(); )
{
    const int & iRef = * cit;
    if(iRef == 15)
    {
        cit = v.erase(cit);
    }
    else
    {
        cit++;
    }
}
for(vector<int>::const_iterator cit = v.begin (); cit != v.end (); cit++)
{
    cout << *cit << endl;
}
// Affiche : "30", "40"
```

Question : Demander aux étudiants les différences entre les deux parcours de boucle dans le code ci-dessus.

Initialisation d'un « std::vector »

Si un container est de taille connue avant utilisation et fixe, ce qui est rare, initialiser la taille du vecteur dans le constructeur (voir diapo précédente).

Si cette taille est connue pendant l'utilisation et fixe, il est possible d'utiliser les méthodes « resize() » ou « reserve() », mais sachez qu'elles génèrent de la réallocation mémoire, couteuse en termes de performances.

Dans les autres cas, ne pas définir de taille et profiter du fait que cette structure est « auto redimensionnable ».

Parcours d'un « std::vector »

Le parcours se fait en utilisant les itérateurs (voir diapo précédente), ou en utilisant les crochets comme pour un tableau en langage C.

Recherche d'un élément dans un « std::vector » trié d'entiers

Question : Demander aux étudiants des propositions pour une telle recherche.

Algorithme « A3 »

```
bool A3(vector<int> & v, int x)
{
    int iSize = v.size();
    for(int i = 0; i < iSize; i++)
    {
        if(v[i] == x)
        {
            return true;
        }
    }

    return false;
}
```

Question : Pourquoi utilise-t-on « `vector <int> &` » comme type de paramètre de cette fonction ?

Algorithme « A4 »

```
bool A4(vector<int> & v, int x)
{
    for(vector<int>::iterator it = v.begin(); it != v.end(); it++)
    {
        if((*it) == x)
        {
            return true;
        }
    }

    return false;
}
```

Algorithme « A5 »

```
bool A5(const vector<int> & cv, int x)
{
    for(vector<int>::const_iterator cit = cv.begin (); cit != cv.end(); cit++)
    {
        if((*cit) == x)
        {
            return true;
        }
    }

    return false;
}
```

Algorithme « A5Bis »

```
bool A5Bis(const vector<int> & cv, int x)
{
    for(auto cit = cv.begin(); cit != cv.end(); cit++)
    {
        if((*cit) == x)
        {
            return true;
        }
    }

    return false;
}
```

Algorithme « A5Ter »

```
bool A5Ter(const vector<int> & cv, int x)
{
    for(auto i : cv)
    {
        if(i == x)
        {
            return true;
        }
    }

    return false;
}
```

Algorithme « A6 »

```
#include <algorithm>        // std::find()
bool A6(vector<int> & v, int x)
{
    vector<int>::iterator itFind = find(v.begin(), v.end(), x);
    return(itFind != v.end());
}
```

Algorithme « A7 »

```
bool A7(const vector<int> & cv, int x)
{
    vector<int>::const_iterator citFind = find(cv.begin(), cv.end(), x);
    return(citFind != cv.end());
}
```

Algorithme « A8 »

```
bool A8(const vector<int> & cv, int x)
{
    return(find(cv.begin(), cv.end(), x) != cv.end());
}
```

Algorithme « A8bis »

```
bool A8bis(const vector<int> & cv, int x)
{
    return(ranges::find(cv, x) != cv.end());
}
```

Algorithme « A9 »

```
bool A9(const vector<int> & cv, int x)
{
    return(binary_search(cv.begin(), cv.end(), x));
}
```

Algorithme « A9bis »

```
bool A9bis(const vector<int> & cv, int x)
{
    return(ranges::binary_search(cv, x));
}
```

Synthèse

Avec ces différents algorithmes, nous n'avons pas amélioré la complexité du problème de recherche d'un élément dans une liste triée d'entiers :

- "O(N)" pour les algorithmes de A3 jusqu'à A8.
- "O(log(N))" pour les algorithmes A9 et A9bis.

Mais nous avons amélioré la qualité du code à travers les points suivants, ce qui est loin d'être négligeable :

- Le nombre d'instructions du code a été considérablement réduit (ce qui ne réduit pas forcément le nombre d'instructions élémentaires).
- L'intelligibilité du code a été améliorée.
- Nous utilisons une structure « auto redimensionnable » et qui accepte des types de données complexes comme des classes.
- Nous effectuons désormais un parcours en lecture seule.

Pause, puis rappel de tout ce que nous venons de voir.

Tri d'un « std::vector »

Utilisation de la méthode « std::sort() » présente dans le fichier « algorithm ».

Exemple d'un tri du plus petit au plus grand:

```
#include <algorithm>        // std::find(), std::sort()...
vector<int> v(3);
v[0] = 30;
v[1] = 15;
v[2] = 40;
sort(v.begin(), v.end());
for(vector<int>::const_iterator cit = v.begin(); cit != v.end(); cit++)
{
      cout << * cit << endl;
}
// Affiche : "15", "30", "40".
```

Exemple d'un tri du plus grand au plus petit :

```
#include <functional>       // std::greater
vector<int> v;
v.push_back(30);
v.push_back(15);
v.push_back(40);
sort(v.begin(), v.end(), greater<int>());
for(vector<int>::const_iterator cit = v.begin(); cit != v.end(); cit++)
{
      cout << * cit << endl;
}
// Affiche : "40", "30", "15".
```

Cette fonction propose également de trier par une fonction personnalisable, ou en utilisant une expression lambda.

```
struct myIntCmp
{
    bool operator()(int a, int b) const
    {
        string sa = to_string(a);
        string sb = to_string(b);
        return sa < sb;
    }
};
vector<int> v;
v.push_back(2);
v.push_back(10);
v.push_back(11);
sort(v.begin(), v.end(), myIntCmp());
for(vector<int>::const_iterator cit = v.begin(); cit != v.end(); cit++)
{
      cout << * cit << endl;
}
// Affiche : "10", "11", "2".
```

Liste d'opérations et complexité

Accès arbitraire	O(1)
Insertion / Suppression	O(N)
Insertion / Suppression à la fin	O(1)
Recherche	O(N)

Exemple :

```
vector<int> v;
v.push_back(2);
v.insert(v.begin(), 1);
v.push_back(3);
v.push_back(4);
v.pop_back();
v.insert(v.begin(), 0);
v.erase(v.begin());
v.insert(v.begin() + 2, 50);
v.insert(v.end() - 2, 60);
v.erase(v.begin() + 1, v.begin() + 3);
for(vector<int>::const_iterator cit = v.begin(); cit != v.end(); cit++)
{
	cout << * cit << endl;
}
// Affiche : "1", "50", "3".
```

Déboguer ces lignes une par une pour montrer l'évolution du vecteur jusqu'à son contenu final.

Liste, File, Pile

Quand on veut favoriser l'insertion ou la suppression.

Structures proposées par la STL :

- std::list : Liste doublement chaînée. Définie dans le fichier STL « list ».
- std::stack : Pratique pour manipuler des piles.
- std::queue : Idéal pour les files.
- std::deque : Liste optimisée pour l'accès arbitraire.
- std::forward_list : Liste simplement chaînée.

Liste d'opérations et complexité

Accès arbitraire	O(N)
Insertion / Suppression	O(1)
Recherche	O(N)

Avant le TD1 : Rappels sur les constructeurs et la liste d'initialisation

Constructeurs pas liste initialisation

- Les **listes d'initialisation** permettent, dans un constructeur, d'utiliser le constructeur de chaque donnée membre, plutôt que d'effectuer une affectation après coup :

Ratio.cpp :

```
Ratio::Ratio(int n, int d) :
      m_num (n),
      m_den (d) {
}
```

- Les constructeurs de m_num et m_den sont appelés juste avant l'appel au constructeur de l'instance de la classe Ratio.
- Remarque : Le corps d'un constructeur est donc souvent "vide".

Vous devez toujours privilégier cette écriture !

TD1

Cas où il est nécessaire de surcharger des opérateurs

Nous aurons l'occasion de surcharger plusieurs opérateurs dans ce module.

Rappels :

Surcharge des opérateurs

A utiliser avec discernement !!

- But : écrire simplement des opérations sur des classes rajoutées au C++. Exemple : addition de deux rationnels.
- 45 opérations : + - * / = += -= *= /= [] () == != <= < >= > << >> etc. (cf. tableau des priorités). Presque toutes ces opérations peuvent être (re)définies sur les nouvelles classes afin d'étendre les possibilités du langage C++.
- Le nom de la fonction à surcharger est **operatorxx(…)** où **xx** est le symbole d'usage de l'opérateur.
- La **signature** de l'opérateur change suivant de quel opérateur il s'agit. **Il est impératif de la respecter**. (Il faut par exemple chercher dans un manuel pour obtenir son type exact).
- Surcharge avec fonction membre ou fonction non-membre (mais amie) suivant les cas.

Pour illustrer cela, considérons la classe "MyClass" basique suivante :

```
class MyClass
{
    protected:

    int m_i;

    public:

    MyClass(int i = 0);
    void display() const;
};
```

Si besoin de rechercher via « std::find() » un élément dans un "std::vector" contenant des instances d'une classe "maison", surcharger l'opérateur "==" de cette classe

```
vector<MyClass> v;
v.push_back(MyClass(2));
v.push_back(MyClass(1));

int i = 2;
vector<MyClass>::const_iterator citFind = find(v.begin(), v.end(), MyClass(i));
if(citFind != v.end())
{
	cout << i << " est dans la liste." << endl;
}
else
{
	cout << i << " n'est pas dans la liste." << endl;
}
// Affiche : "2 est dans la liste.".

i = 4;
citFind = find(v.begin(), v.end(), MyClass(i));
if (citFind != v.end())
{
	cout << i << " est dans la liste." << endl;
}
else
{
	cout << i << " n'est pas dans la liste." << endl;
}
// Affiche : "4 n'est pas dans la liste.".
```

Pour pouvoir compiler ce code, il faut donc ajouter à la classe "MyClass":

```
    bool operator==(const MyClass & mc) const {return (this->m_i == mc.m_i);}
```

Si besoin de trier via « std::sort() » un "std::vector" contenant des instances d'une classe "maison", surcharger l'opérateur correspondant au tri souhaité

Pour que le compilateur accepte qu'on effectue un tri croissant sur une liste contenant des instances d'une classe "maison", il faut redéfinir (surcharger) l'opérateur « < » entre 2 instances de cette classe.

```
vector<MyClass> v;
v.push_back(MyClass(2));
v.push_back(MyClass(1));
sort(v.begin(), v.end());
for(vector<MyClass>::const_iterator cit = v.begin(); cit != v.end(); cit++)
{
	cit->display();
}
// Affiche : "1", "2".
```

Pour pouvoir compiler ce code, il faut donc ajouter à la classe "MyClass":

```
    bool operator<(const MyClass & mc) const {return (this->m_i < mc.m_i);}
```

Si on souhaite trier dans un autre ordre, surcharger l'opérateur correspondant (« > » par exemple) ou définir une fonction prédicat, et préciser cet opérateur ou cette fonction prédicat en 3ème paramètre de « std::sort() » (comme fait précédemment dans ce cours).

Si besoin d'utiliser "std::cout" avec des classes "maison", surcharger l'opérateur "<<" par fonction amie

Surcharge d'opérateurs par fonction amie

Ratio.h

```
class Ratio {
friend ostream& operator<< (ostream& , const Ratio& );
friend Ratio operator+ (int, const Ratio&);
    ...
};
```

Ratio.cpp

```
ostream& operator<< (ostream& os, const Ratio& r){
    os << r.num << '/' << r.den ;
    return os;
}
Ratio operator+ (int entier, const Ratio& r){
    return r + entier;
}
```

➢ Observez qu'une fonction amie a accès aux données membres privées

Pour que le compilateur accepte le code suivant :

```
vector<MyClass> v;
v.push_back(MyClass(2));
v.push_back(MyClass(1));
for(vector<MyClass>::const_iterator cit = v.begin(); cit != v.end(); cit++)
{
      cout << (*cit) << endl;
}
// Affiche : "2", "1".
```

Il faut ajouter ceci à la déclaration de la classe "MyClass":

```
    friend ostream & operator<<(ostream &, const MyClass &);
```

Puis ajouter ceci en dehors de la classe "MyClass":

```
ostream & operator<<(ostream & os, const MyClass & mc)
{
    os << mc.m_i;
    return os;
}
```

Remarque : à partir de ce moment-là, la méthode "`display ()`" de la classe "MyClass" peut être supprimée.

Méthodes et classes abstraites

- Dans une classe il est possible de définir une (ou plusieurs) **méthode(s) abstraite(s)**. Pour cela on utilise **virtual** au début et "**=0**" à la fin de son prototype dans la définition de la classe.

```
class A {
public :
    virtual void f ( ) = 0 ;
    [...]
};
```

- Une classe possédant une telle méthode est une **classe abstraite**, elle ne peut pas être instanciée.
- Toute classe dérivée d'une classe abstraite doit redéclarer toutes les méthodes abstraites, elle peut cependant ne pas apporter de définition à toutes les méthodes abstraites, dans ce cas elle est aussi abstraite.
- Pourquoi utiliser une classe abstraite ?
  - Afin de permettre des traitements génériques sur une hiérarchie de classes.
  - Afin d'imposer une spécification que les sous-classes doivent obligatoirement implémenter.
  - Afin de séparer la spécification et l'implémentation.
- Exemple classique :
  - Une classe abstraite "Forme" (pour forme géométrique) **déclarant** des méthodes aire(), perimetre(), dessiner(), etc.
  - Des classes concrètes dérivant de la classe "Forme" telles que Triangle, Rectangle, Cercle, etc. **définissant** les méthodes aire(), perimetre(), dessiner(), etc.

Rappels sur les conversions automatiques

L'héritage : conversions automatiques

- Si B hérite de A, alors toutes les instances de B sont aussi des instances de A, il est donc possible de faire :

```
A a ; B b ; a = b;                    // ok
```

- Propriété conservée lorsqu'on utilise des pointeurs :

```
A* pa ; B* pb ; pa = pb;              // ok
```

- Évidement, l'inverse n'est pas vrai :

```
A a ; B b ; b = a;                    // Erreur !
```

- Pareil pour les pointeurs :

```
A* pa ; B* pb ; pb = pa;              // Erreur !
```

TD2

Tableaux associatifs et ensembles

Utiles pour créer des dictionnaires, ou toute autre association d'un élément à un autre.

Voici les structures correspondantes proposées par le langage C++ :

- `std::unordered_map` : Adapté pour les tableaux associatifs.
- `std::unordered_set` : Pour les ensembles d'éléments.
- `std::unordered_multimap` : Tableaux associatifs où une clé peut référer à plusieurs éléments.
- `std::unordered_multiset` : Ensembles dans lesquels une valeur peut être présente plus d'une fois.

Exemple avec « std::unordered_set »

```
cout << endl << "std::unordered_set:" << endl;
unordered_set<string> stringSet;
stringSet.insert("code");
stringSet.insert("in");
stringSet.insert("c++");
stringSet.insert("is");
stringSet.insert("fast");

string key = "slow";
if(stringSet.find(key) == stringSet.end())
{
      cout << key << " not found" << endl << endl ;
}
else
```

```
{
	cout << "Found " << key << endl << endl ;
}

key = "c++";
if(stringSet.find(key) == stringSet.end())
{
	cout << key << " not found" << endl << endl ;
}
else
{
	cout << "Found " << key << endl << endl ;
}

cout << endl << "All elements:" << endl;
for(unordered_set<string>::const_iterator cit = stringSet.begin(); cit !=
stringSet.end(); cit ++)
{
	cout << *cit << endl ;
}
```

Liste d'opérations et complexité

Accès arbitraire	O(1)
Insertion / Suppression	O(1)
Recherche	O(1)

Les arbres

Un arbre est un graphe acyclique orienté où l'on a défini un nœud central appelé racine et dans lequel chaque nœud (sauf la racine) admet exactement un parent. Les nœuds n'admettant pas de fils sont dits feuilles. Concrètement, une telle structure peut se représenter sous une forme arborescente.

Les tas

Un tas est un arbre binaire, c'est-à-dire que chaque nœud comporte au plus deux nœuds fils. La particularité du tas réside dans l'ordre qu'il impose à ses éléments : chaque parent doit avoir une valeur plus grande (ou plus petite, mais c'est plus rare) que ses fils. Le maximum (respectivement le minimum) correspond donc à la racine. Un tas est conventionnellement équilibré, c'est-à-dire que sa profondeur en chaque point varie au plus d'une unité.

Les opérations que l'on peut effectuer sur un tas sont typiquement les suivantes : récupération du maximum et insertion d'un élément.

Le champ d'application des tas est très vaste : on peut s'en servir pour classer des tâches, de la plus prioritaire à la moins importante, mais cette structure est également le pilier d'un algorithme de tri appelé tri par tas. Certains langages s'en servent aussi pour les allocations dynamiques de mémoire (les allocations statiques étant effectuées sur une pile). Voir la documentation en ligne de la fonction « std::make_heap » pour plus de détails.

Liste d'opérations et complexité

Lecture du maximum	O(1)
Insertion / Suppression	O(log (N))

Les arbres binaires de recherche

Une deuxième catégorie d'arbre est également souvent sollicitée : les arbres binaires de recherche, ou ABR. Dans un tel arbre, tous les éléments du sous-arbre gauche à un nœud N sont inférieurs à N et tous les éléments du sous-arbre droit lui sont supérieurs. On doit retrouver cette propriété de manière récursive au niveau de chaque nœud.

Un ABR est donc une manière de représenter une collection triée d'éléments. C'est pour cette raison qu'elle est uniquement envisageable quand les éléments sont comparables entre eux (si a et b sont deux éléments, on doit

pouvoir dire si a < b est vrai ou faux). Généralement la fonction de comparaison est paramétrable, permettant n'importe quelle arborescence triée souhaitée. L'intérêt d'un ABR réside dans la recherche et l'insertion d'éléments dans un ensemble trié : celles-ci doivent être optimales tout en conservant l'ordre entre les éléments.

C'est pour cette raison qu'un ABR est idéalement équilibré, sinon on pourrait très bien envisager un arbre balancé uniquement d'un côté qui ne serait en fait nullement différent d'une simple liste chaînée triée. Dans ce cas l'insertion et la recherche d'un élément tout en conservant l'ordre ne seraient plus optimales du tout. Il existe différents types d'ABR équilibré : les arbres rouge-noir, les AVL, etc.

Il est possible d'utiliser un ABR de manière naïve où chaque élément ne représente qu'une information. Cependant, on peut l'utiliser d'une deuxième manière : en l'ordonnant selon des clés et en ajoutant à chaque nœud un attribut supplémentaire correspondant à une valeur, on arrive à reproduire le type abstrait d'un tableau associatif (pour rappel : `std::unordered_map`, `std::unordered_set`, `std::unordered_multimap` et `std::unordered_multiset`). La seule différence réside alors dans le fait que l'on aurait alors une notion d'ordre, qui induit évidemment des performances moins spectaculaires que les tables de hachage.

Il est intéressant de noter que la valeur la plus à gauche de l'arborescence se trouve être le minimum de l'ensemble. Symétriquement, la valeur la plus à droite est le maximum.

Dispos (dans "std::" ou "boost::" selon les versions de compilateur ; pour rappel, Boost complète STL dans le même esprit) :

- `std::set` : Idéal pour les ABR équilibrés et les ensembles ordonnés.
- `std::map` : Tableau associatif ordonné.
- `std::multiset` : Ensemble ordonné pouvant contenir plusieurs fois la même valeur.
- `std::multimap` : Tableau associatif ordonné pouvant indexer plusieurs fois la même clé.
- `boost::bimap` : Tableau associatif ordonné avec association dans les deux sens.

Liste d'opérations et complexité

Lecture du maximum ou minimum	O(1)
Accès arbitraire	O(log(N))
Insertion / Suppression	O(log(N))
Recherche	O(log(N))

Remarque : cette section sur les arbres recoupe une partie de la section graphes vue avec Yon Dourisboure.

Avant de voir « std::map » en détail, autre structure complémentaire utilisée par la STL : « std::pair »

« std::pair<T1,T2> » permet de stocker 2 éléments dans une même structure.

Exemple :

```
#include <utility>          // std::pair
#include <string>           // std::string

pair<int, string> p = make_pair(5, "chaine 1");
cout << p.first << " " << p.second << endl;
```

C'est aussi le type retourné par certaines opérations effectuées sur des structures STL, par exemple :

- Les itérateurs de « map <T1, T2> » pointent vers des objets de type "`pair <T1, T2>`".
- « set <T>::insert (T) » a pour retour un objet de type "`pair <T &,bool>`".

Liste d'opérations et complexité

Insertion	O(1)
Accès	O(1)

Focus sur « std::set » et « std::map »

« set<T…> » et « map<T1, T2…> » sont très pratiques à utiliser si on a besoin d'une liste sans doublon et triée. « set » est défini dans le fichier « set », et « map » dans le fichier « map ».

Le 1er prérequis pour les utiliser est que le type (« T » pour « set <T…> », « T2 » pour « map <T1, T2…> ») ait un constructeur par défaut qui soit défini.

Le 2ème prérequis est de définir une relation d'ordre entre les éléments du type (« T » pour « set <T…> », « T1 » pour « map <T1, T2…> ») afin de pouvoir les trier. Par défaut, l'ordre utilisé pour le tri est le « foncteur » `std::less <T>` (basé sur l'opérateur "<"). Pour que le compilateur accepte qu'on utilise un « set » ou un « map » avec une classe que nous avons écrite, il faut définir (surcharger) l'opérateur « < » de cette classe, ce qui permettra au compilateur de savoir comment trier les objets.

Le parcours d'un « set » se fait en utilisant les itérateurs.

Le parcours d'un « map » se fait en utilisant les itérateurs, ou en utilisant les crochets comme pour un tableau en langage C, mais attention car l'utilisation des crochets entraîne la création de l'élément si cet élément n'est pas encore présent (pas seulement une recherche), puis retourne une référence sur l'élément. Si une simple recherche est nécessaire, utiliser la méthode « find () ».

STL propose également de définir un autre ordre de tri que l'opérateur « < ». Si c'est le cas, il faudra préciser cet ordre au moment de déclarer la variable concernée. Par exemple, on peut déterminer que l'ordre soit du plus grand au plus petit.

Les structures `std::set` et `std::map` tirent parti de l'ordre pour construire une structure d'arbre rouge noir, ce qui permet une recherche logarithmique d'un élément.

Exemple d'utilisation de `std::set` et `std::map` avec l'ordre par défaut (croissant) :

```
#include <functional>       // std::greater
#include <set>              // std::set
#include <map>              // std::map

set<int> si1;
si1.insert(3);
si1.insert(3);
si1.insert(2);
si1.insert(4);
si1.erase(4);
set<int>::const_iterator citFind = si1.find(2);
if(citFind == si1.end())
{
      cout << "2 n'est pas dans si1." << endl;
}
else
{
      cout << "2 est dans si1." << endl;
}
for(set<int>::const_iterator cit = si1.begin(); cit != si1.end(); cit++)
{
      cout << * cit << endl;
}
// Affiche : "2", "3".

map<int, int> mii1;
mii1[2] = 21;
mii1[2] = 22;
mii1.insert(make_pair(1, 11));
for(map<int, int>::const_iterator cit = mii1.begin(); cit != mii1.end(); cit++)
{
      const pair<int, int> & cpRef = * cit;
      cout << cpRef.first << " " << cpRef.second << endl;
}
// Affiche : "1", "11", "2", "22".
```

Exemple d'utilisation de `std::set` et `std::map` avec l'ordre décroissant :

```
set<int, greater<int>> siMaxToMin1;
siMaxToMin1.insert(3);
siMaxToMin1.insert(3);
siMaxToMin1.insert(2);
for(set<int, greater<int>>::const_iterator cit = siMaxToMin1.begin(); cit !=
siMaxToMin1.end(); cit++)
{
      cout << * cit << endl;
}
// Affiche : "3", "2".

map<int, int, greater<int>> miiMaxToMin1;
miiMaxToMin1[2] = 21;
miiMaxToMin1[1] = 11;
for(map<int, int, greater<int>>::const_iterator cit = miiMaxToMin1.begin(); cit !=
miiMaxToMin1.end(); cit++)
{
      const pair<int, int> & cpRef = * cit;
      cout << cpRef.first << " " << cpRef.second << endl;
}
// Affiche : "2", "21", "1", "11".
```

Exemple avec une classe "maison" où il est nécessaire de surcharger des opérateurs

```
class MyClass
```

```
{
    protected:

    int m_i;

    public:

    MyClass(int i = 0);
    void display() const;
    bool operator<(const MyClass & mc) const;
    bool operator>(const MyClass & mc) const;
};

MyClass::MyClass(int i) : m_i(i)
{}

void MyClass::display() const
{
    cout << this->m_i << endl;
}

bool MyClass::operator<(const MyClass & mc) const
{
    return (this->m_i < mc.m_i);
}

bool MyClass::operator>(const MyClass & mc) const
{
    return (this->m_i > mc.m_i);
}

typedef set<MyClass> sMC;
sMC smc1;
smc1.insert(MyClass(4));
smc1.insert(MyClass(4));
smc1.insert(MyClass(5));
for(sMC::const_iterator cit = smc1.begin(); cit != smc1.end(); cit++)
{
      const MyClass & cmcRef = * cit;
      cmcRef.display();
      cout << endl;
}
// Affiche : "4", "5".

typedef map<MyClass, int, greater<MyClass>> mMCiGr;
mMCiGr mmcMaxToMin1;
mmcMaxToMin1[MyClass(4)] = 1;
mmcMaxToMin1.insert(make_pair(MyClass(9), 2));
for(mMCiGr::const_iterator cit = mmcMaxToMin1.begin(); cit != mmcMaxToMin1.end(); cit++)
{
      const pair<MyClass, int> & cpRef = * cit;
      const MyClass & cmciRef = cpRef.first;
      cmciRef.display();
      cout << cpRef.second << endl;
}
// Affiche : "9", "2", 4", "1".
```

Exemples d'utilisation de "std::multiset" et "std::multimap"

```
const multiset<string> words =
{
      "some", "not", "sorted", "words",
      "will", "come", "out", "sorted",
};
for(auto it = words.begin(); it != words.end(); )
{
```

```
        auto cnt = words.count(*it);
        cout << *it << ":\t" << cnt << '\n';
        advance(it, cnt); // all cnt elements have equivalent keys
}
// "come:      1 \n not:      1 \n out:      1 \n some:     1 \n sorted:   2 \n will:     1 \n words:    1".

cout << endl;
multimap <size_t, string> mmap;
mmap.insert({ sizeof(long), "long" });
mmap.insert({ sizeof(int), "int" });
mmap.insert({ sizeof(short), "short" });
mmap.insert({ sizeof(char), "char" });
mmap.insert({{ sizeof(float), "float" }, { sizeof(double), "double" }});
for(multimap<size_t, string>::const_iterator cit = mmap.begin(); cit != mmap.end();
cit++)
{
        const pair<size_t, string> & cpRef = * cit;
        cout << cpRef.first << " " << cpRef.second << endl;
}
// Affiche : "1 char \n 2 short \n 4 long \n 4 int \n 4 float \n 8 double
```

TD3

Récapitulatif

Nous avons vu que la STL propose un large choix de structures de données. Nous avons détaillé les structures les plus couramment utilisées. Il est important de savoir que chacune de ces structures existe, afin d'utiliser la plus adaptée à votre besoin spécifique.

Tableau récapitulatif des complexités en fonction des structures et des opérations courantes :

<table>
<tr><th></th><th>Accès arbitraire</th><th>Insertion
Suppression</th><th>Recherche</th><th>Lecture du minimum</th><th>Lecture du maximum</th><th>Suppression du maximum</th><th>Doublons autorisés</th></tr>
<tr><td>Tableau
std::vector<T...></td><td>O(1)</td><td>O(N)
O(1) fin</td><td>O(N)</td><td>O(N)</td><td>O(N)</td><td></td><td>Oui</td></tr>
<tr><td>std::deque<T...></td><td>O(1)</td><td>O(N)
O(1) début/fin</td><td>O(N)</td><td>O(N)</td><td>O(N)</td><td></td><td>Oui</td></tr>
<tr><td>std::pair<T1, T2></td><td>O(1)</td><td>O(1)</td><td></td><td></td><td></td><td></td><td></td></tr>
<tr><td>std::list <T...></td><td>O(N)</td><td>O(1)</td><td>O(N)</td><td>O(N)</td><td>O(N)</td><td></td><td>Oui</td></tr>
<tr><td>std::stack <T...></td><td>O(N)</td><td>O(1) début</td><td>O(N)</td><td>O(N)</td><td>O(N)</td><td></td><td></td></tr>
<tr><td>std::priority_queue<T...></td><td></td><td>O(log(N))</td><td>O(N)</td><td></td><td>O(1)</td><td>O(log(N))</td><td>Oui</td></tr>
<tr><td>std::unordered_map<T1, T2...></td><td rowspan="4">O(1)</td><td rowspan="4">O(1)</td><td rowspan="4">O(1)</td><td rowspan="4">O(N)</td><td rowspan="4">O(N)</td><td rowspan="4"></td><td rowspan="2">Non</td></tr>
<tr><td>std::unordered_set<T...></td></tr>
<tr><td>std::unordered_multimap<T1, T2...></td><td rowspan="2">Oui</td></tr>
<tr><td>std::unordered_multiset<T...></td></tr>
<tr><td>std::set<T...></td><td rowspan="4">O(log(N))</td><td rowspan="4">O(log(N))</td><td rowspan="4">O(log(N))</td><td rowspan="4">O(1)</td><td rowspan="4">O(1)</td><td rowspan="4"></td><td rowspan="2">Non</td></tr>
<tr><td>std::map<T1, T2...></td></tr>
<tr><td>std::multiset<T...></td><td rowspan="2">Oui</td></tr>
<tr><td>std::multimap<T1, T2...></td></tr>
</table>

Tableau récapitulatif des différentes structures de données STL en fonction des besoins :

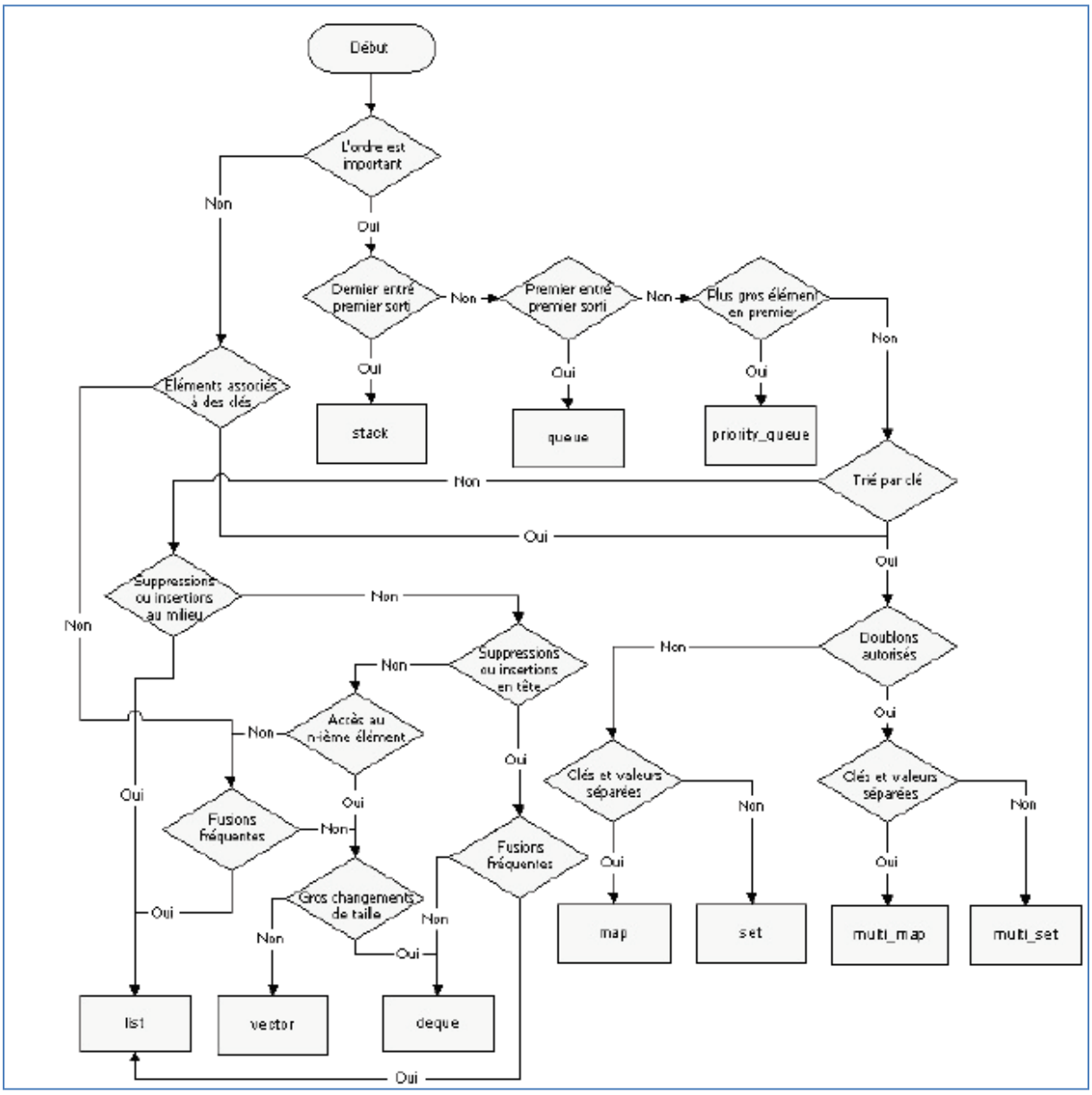
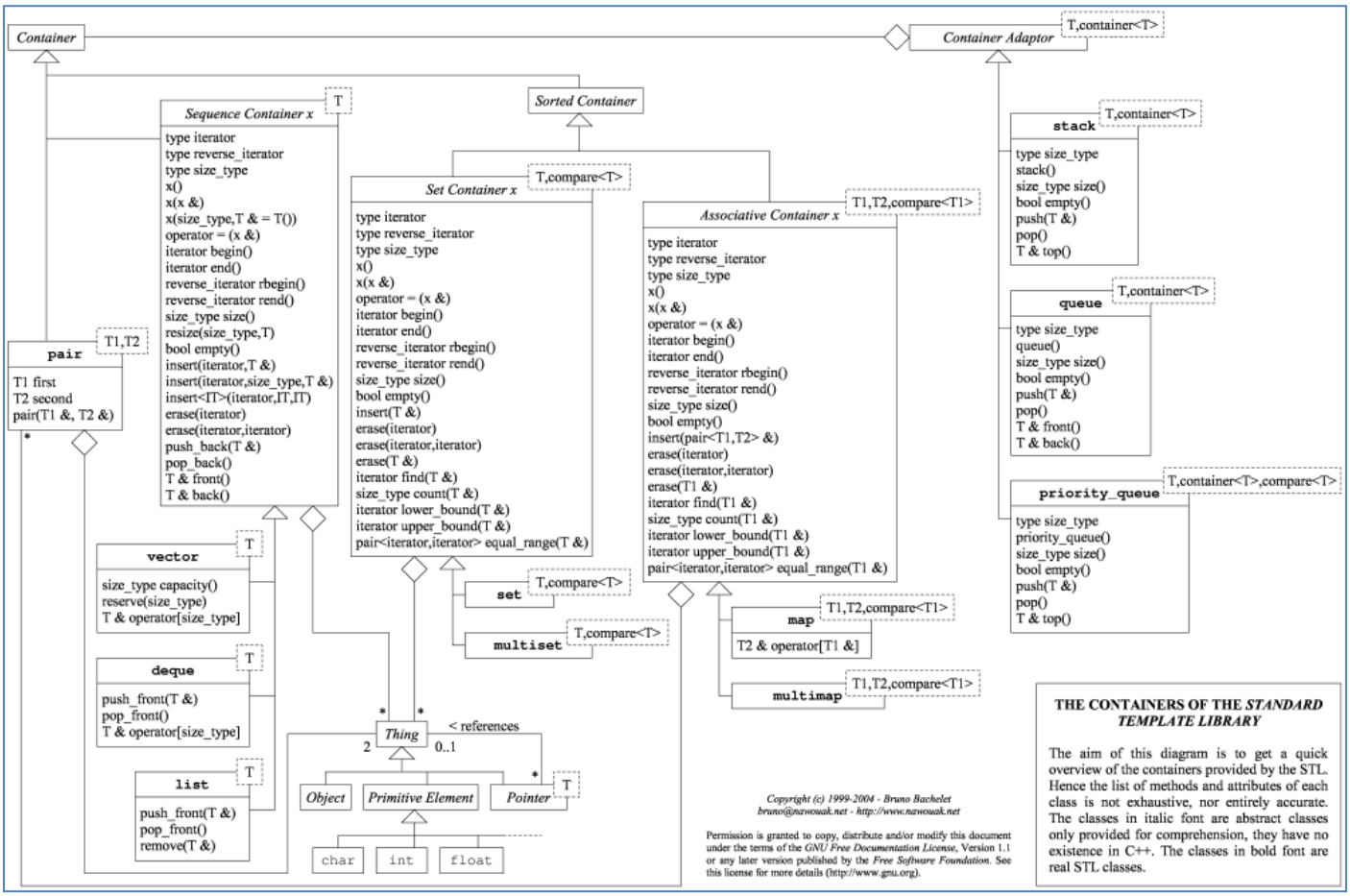


Tableau récapitulatif des différentes structures de données STL :



Références

Complexité d'un problème

Exemples : http://frederic-junier.org/ISN/Cours/CoursAlgoDicho14V1.pdf.

Complexité en temps : https://fr.wikipedia.org/wiki/Complexit%C3%A9_en_temps.

Méthode de calcul de la complexité d'un algorithme | Rachid Guerraoui : https://www.youtube.com/watch?v=clZ4q5zPBlE.

Recherche dichotomique : https://fr.wikipedia.org/wiki/Recherche_dichotomique

C++

Cours du module « Prog Objet » (M2103, 2ème semestre de l'IUT) créé par Yon Dourisboure.

STL

Aide en ligne du langage C++, incluant la STL : https://en.cppreference.com/w/cpp. Elle est très bien faite, et propose de nombreux exemples. Privilégiez la version anglaise qui est plus complète et propose plus d'exemples.

« Tutoriel : C++ : Bien choisir ses structures de données » : http://sdz.tdct.org/sdz/c-bien-choisir-ses-structures-de-donnees.html

« Introduction à la STL en C++ (Standard Template Library) » : https://www.commentcamarche.net/faq/11255-introduction-a-la-stl-en-c-standard-template-library.