UNIVERSIDADE FEDERAL DE VIÇOSA
CENTRO DE CIÊNCIAS EXATAS
DEPARTAMENTO DE INFORMÁTICA

# IMPLEMENTAÇÃO DE MULTICONJUNTOS UTILIZANDO ESTRUTURAS LINEARES

Gabriel Vita Silva Franco - 79208

VIÇOSA
MINAS GERAIS - BRASIL
09/2018

# Sumário

# 1 Implementação

A ideia geral do trabalho é implementar a estrutura multiconjunto utilizando estruturas lineares, compara-las entre si e com a implementação da biblioteca STL/C++. A classe `multiset`[1] de C++ é implementada utilizando árvore binária de pesquisa, logo, ela é mais eficiente em relação as implementações utilizando estruturas lineares. Quando a ordem dos elementos não importa, o multiconjunto é implementado de forma eficiente utilizando Tabela Hash, como foi feito na classe `unordered_multiset`[2] de C++. Os multiconjuntos foram implementados de duas formas, são elas:

1. Utilizando lista duplamente encadeada
2. Utilizando Vector

Os detalhes de cada implementação estão nas subseções abaixo. Foi utilizada para as implementações a linguagem C++ na versão 17. Todas as duas implementações foram feitas de forma genérica, utilizando `template`[3]. O código fonte está disponível no GitHub.

## 1.1 Implementação utilizando lista duplamente encadeada

A ideia dessa implementação é usar a estrutura de lista encadeada, com cada nó guardando os ponteiros para os nós anterior e próximo, o valor e o número de ocorrências desse valor. Com isso, para inserir ou remover um elemento repetido no multiconjunto, é só incrementar ou decrementar o valor dessa variável. A lista guarda os elementos de forma ordenada. A estrutura desse multiconjunto é mostrada na Figura 1.

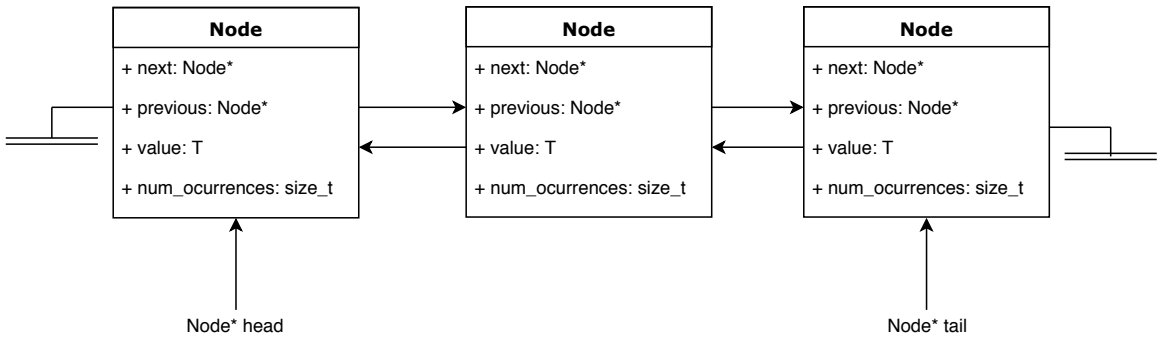


Figura 1: Estrutura do multiconjunto utilizando lista duplamente encadeada. Fonte: Autor

A ideia geral das implementações de cada operação é a seguinte:

[1] http://www.cplusplus.com/reference/set/multiset/
[2] http://www.cplusplus.com/reference/unordered_set/unordered_multiset/
[3] http://www.cplusplus.com/doc/oldtutorial/templates/

- INSERE - Procura a posição para a inserção. Se o elemento já existe na lista, faz `num_ocurrences++`, senão, insere o elemento de forma que a lista continue ordenada, atualizando os ponteiros `next` e `previous` dos nós vizinhos.

- PERTENCE - É feita uma busca linear procurando o elemento. Se ele existe, retorna `true`, senão, retorna `false`.

- FREQUÊNCIA - É feita uma busca linear procurando o elemento. Se ele existe, retorna `num_ocurrences`, senão, retorna `0`.

- REMOVE - É feita uma busca linear procurando o elemento. Se ele existe e `num_ocurrences` $> 1$, `num_ocurrences--`. Se ele existe e `num_ocurrences` $== 1$, remove o Nó, atualizando os ponteiros `next` e `previous` dos nós vizinhos. Se ele não existe, nada é feito.

- APAGA - É feita uma busca linear procurando o elemento. Se ele existe, remove o Nó, atualizando os ponteiros `next` e `previous` dos nós vizinhos. Se ele não existe, nada é feito.

- UNIÃO - São utilizados dois ponteiros `p1` e `p2` que iteram sobre os multiconjuntos $M_1$ e $M_2$. Se `p1->value` $<$ `p2->value`, a união recebe `p1->value` `p1->num_ocurrences` vezes e `p1 = p1->next`. Se `p2->value` $<$ `p1->value`, a união recebe `p2->value` `p2->num_ocurrences` vezes e `p2 = p2->next`. Se `p1->value` $==$ `p2->value`, a união recebe `p1->value max(p1->num_ocurrences, p2->num_ocurrences)` vezes, `p1 = p1->next` e `p2 = p2->next`. A união acaba quando os dois multiconjuntos são percorridos completamente.

- INTERSEÇÃO - São utilizados dois ponteiros `p1` e `p2` que iteram sobre os multiconjuntos $M_1$ e $M_2$. Se `p1->value` $<$ `p2->value`, `p1 = p1->next`. Se `p2->value` $<$ `p1->value`, `p2 = p2->next`. Se `p1->value` $==$ `p2->value`, a interseção recebe `p1->value min(p1->num_ocurrences, p2->num_ocurrences)` vezes, `p1 = p1->next` e `p2 = p2->next`. A interseção acaba quando os dois multiconjuntos são percorridos completamente.

- DIFERENÇA - São utilizados dois ponteiros `p1` e `p2` que iteram sobre os multiconjuntos $M_1$ e $M_2$. Se `p1->value` $<$ `p2->value`, a diferença recebe `p1->value` `p1->num_ocurrences` vezes e `p1 = p1->next`. Se `p2->value` $<$ `p1->value`, `p2 = p2->next`. Se `p1->value` $==$ `p2->value` e `p1->num_ocurrence` $>$ `p2->num_ocurrences`, a diferença recebe `p1->value (p1->num_ocurrence - p2->num_ocurrences)` vezes, `p1 = p1->next` e `p2 = p2->next`. A diferença acaba quando os dois multiconjuntos são percorridos completamente.

## 1.2 Implementação utilizando vector

A ideia dessa implementação é usar Vector, ou seja, um *array* comum, contíguo em memória, alocado dinamicamente a medida que o tamanho cresce. Esse mecanismo foi implementado da seguinte forma: quando *length* == *capacity*, *capacity* *= *2* e aloca-se um espaço novo de memória com a função `realloc`[4] utilizando a nova capacidade. O Vector guarda os elementos de forma ordenada. A estrutura desse multiconjunto é mostrada na Figura 2.

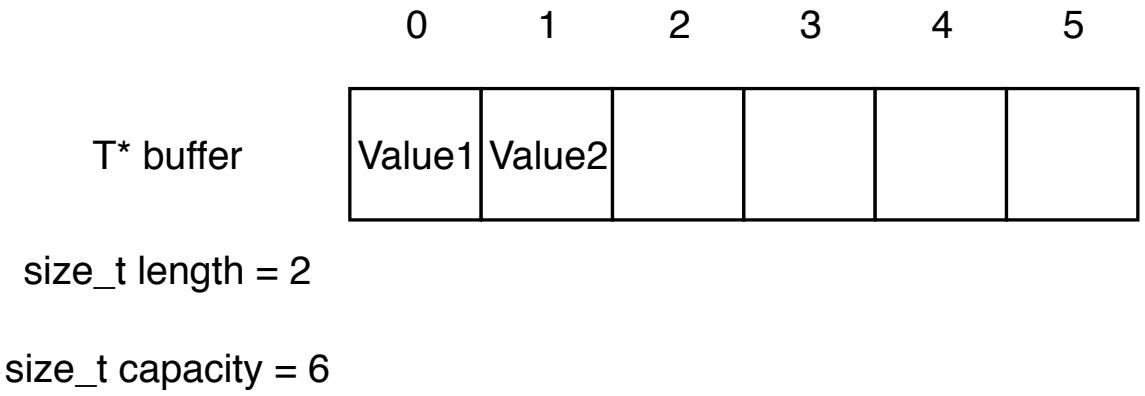


Figura 2: Estrutura do multiconjunto utilizando vector. Fonte: Autor

A ideia geral das implementações de cada operação é a seguinte:

- INSERE - Procura a posição para a inserção. Insere de forma a deixar o Vector ordenado, fazendo *shift* se necessário.
- PERTENCE - É feita uma busca binária procurando o elemento. Se ele existe, retorna `true`, senão, retorna `false`.
- FREQUÊNCIA - É feita uma busca binária procurando o elemento. Se ele existe, é feita uma busca linear a partir daquela posição com dois contadores, um para frente (`lower_pos`) e um para trás (`upper_pos`), para saber quantas vezes esse elemento aparece. O valor retornado é `upper_pos` $-$ `lower_pos` $+1$. Se o elemento não existe, retorna `0`.
- REMOVE - É feita uma busca binária procurando o elemento. Se ele existe, remove, fazendo *shift* se necessário. Se ele não existe, nada é feito.
- APAGA - É feita uma busca binária procurando o elemento. Se ele existe, é feita uma busca linear a partir daquela posição com dois contadores, um para frente (`lower_pos`) e um para trás (`upper_pos`), para saber quantas vezes esse elemento aparece. Os valores dentro do intervalo [ `lower_pos` , `upper_pos`] são removidos, fazendo *shift* se necessário. Se ele não existe, nada é feito.

[4] http://www.cplusplus.com/reference/cstdlib/realloc/

- UNIÃO - São utilizados dois contadores `i1` e `i2` que iteram sobre os multiconjuntos $M_1$ e $M_2$. Quando $M_1[\texttt{i1}] < M_2[\texttt{i2}]$, $M_1[\texttt{i1}]$ é inserido na união e $\texttt{i1}{+}{+}$. Quando $M_2[\texttt{i2}] < M_1[\texttt{i1}]$, $M_2[\texttt{i2}]$ é inserido na união e $\texttt{i2}{+}{+}$. Quando $M_1[\texttt{i1}] == M_2[\texttt{i2}]$, $M_1[\texttt{i1}]$ é inserido na união, $\texttt{i1}{+}{+}$ e $\texttt{i2}{+}{+}$. A união acaba quando os dois multiconjuntos são totalmente percorridos.

- INTERSEÇÃO - São utilizados dois contadores `i1` e `i2` que iteram sobre os multiconjuntos $M_1$ e $M_2$. Quando $M_1[\texttt{i1}] < M_2[\texttt{i2}]$, $\texttt{i1}{+}{+}$. Quando $M_2[\texttt{i2}] < M_1[\texttt{i1}]$, $\texttt{i2}{+}{+}$. Quando $M_1[\texttt{i1}] == M_2[\texttt{i2}]$, $M_1[\texttt{i1}]$ é inserido na interseção, $\texttt{i1}{+}{+}$ e $\texttt{i2}{+}{+}$. A interseção acaba quando os dois multiconjuntos são totalmente percorridos.

- DIFERENÇA - São utilizados dois contadores `i1` e `i2` que iteram sobre os multiconjuntos $M_1$ e $M_2$. Quando $M_1[\texttt{i1}] < M_2[\texttt{i2}]$, $M_1[\texttt{i1}]$ é inserido na diferença e $\texttt{i1}{+}{+}$. Quando $M_2[\texttt{i2}] < M_1[\texttt{i1}]$, $\texttt{i2}{+}{+}$. Quando $M_1[\texttt{i1}] == M_2[\texttt{i2}]$, $\texttt{i1}{+}{+}$ e $\texttt{i2}{+}{+}$. A diferença acaba quando os dois multiconjuntos são totalmente percorridos.

Na próxima seção, será apresentada uma comparação teórica entre as duas estruturas implementadas, mostrando as principais diferenças entre as duas implementações.

# 2 Comparação teórica

A comparação teórica entre as estruturas é importante para levantas suas vantagens e desvantagens, e consequentemente, utiliza-las da forma correta. As Tabelas 1 e 2 mostram a comparação das estruturas no melhor e pior caso, respectivamente. Neste caso, $n$ e $m$ são o número de elementos dos conjuntos $A$ e $B$, respectivamente.

Tabela 1: Comparação teórica de melhor caso das implementações

| Operação | Utilizando lista duplamente encadeada | Utilizando Vector |
|---|---|---|
| Insere | $O(1)$ | $O(1)$ |
| Pertence | $O(1)$ | $O(1)$ |
| Frequência | $O(1)$ | $O(1)$ |
| Remove | $O(1)$ | $O(1)$ |
| Apaga | $O(1)$ | $O(1)$ |
| União | $O(n+m)$ | $O(n+m)$ |
| Interseção | $O(n+m)$ | $O(n+m)$ |
| Diferença | $O(n+m)$ | $O(n+m)$ |

Tabela 2: Comparação teórica de pior caso das implementações

| Operação | Utilizando lista duplamente encadeada | Utilizando Vector |
|---|---|---|
| Insere | $O(n)$ | $O(n)$ |
| Pertence | $O(n)$ | $O(logn)$ |
| Frequência | $O(n)$ | $O(n)$ |
| Remove | $O(n)$ | $O(n)$ |
| Apaga | $O(n)$ | $O(n)$ |
| União | $O(n+m)$ | $O(n+m)$ |
| Interseção | $O(n+m)$ | $O(n+m)$ |
| Diferença | $O(n+m)$ | $O(n+m)$ |

Em relação a eficiência de espaço, as duas implementações são $O(n)$ no pior caso. Mas como se trata de um multiconjunto, e pode existir elemento repetido, a implementação utilizando lista duplamente encadeada se aproveita dessa situação, por utilizar a variável $num\_ocurrences$ para contar o número de ocorrências de um certo valor. Por exemplo, dado o multiconjunto $A = \{1, 1, 1, 1, 1\}$, a lista utilizará apenas um nó para guardar $A$, com $num\_ocurrences = 5$. Já a implementação utilizando Vector utilizará as 5 posições do array. Em um "caso médio", a lista é mais eficiente em espaço.

# 3 Análise experimental

Foi realizada uma análise experimental para comparar as duas implementações realizadas. Na implementação com Vector, foi escolhida $capacity = 1024$. Os experimentos foram divididos em três partes, e cada uma foi realizada com $Numero\ de\ Elementos = [100000, 50000, 100000]$. Os experimentos são:

- *Experimento* 1: Utilizando os elementos do intervalo $[0, Numero\ de\ Elementos - 1]$
- *Experimento* 2: Utilizando apenas um valor, no caso, 0.
- *Experimento* 3: Utilizando elementos aleatórios no intervalo de $[0, Numero\ de\ Elementos - 1]$ gerados pela função `rand`[1] do C++.

Nos experimentos 1 e 2, os conjuntos $M_1$ e $M_2$ são iguais, diferente do experimento 3, no qual eles são aleatórios. Foi utilizada uma máquina com o processador AMD FX 8-Core Black Edition FX-8350 @ 4.20Ghz e 16GB DDR3 1866Mhz de RAM para rodar os experimentos. Os códigos foram compilados usando o g++, usando as *flags* `-std=c++17 e -O3`. Os resultados dos experimentos 1, 2 e 3 estão nas Tabelas 3, 4 e 5, respectivamente.

Tabela 3: *Experimento* 1 entre as duas implementações

| Operação | Tempo (ms) | | | | | |
|---|---|---|---|---|---|---|
| | Lista duplamente encadeada | | | Vector | | |
| | 10000 Elementos | 50000 Elementos | 100000 Elementos | 10000 Elementos | 50000 Elementos | 100000 Elementos |
| Inserção em $M_1$ | 315,000000 | 13.761,000000 | 89.343,000000 | 36,000000 | 943,000000 | 3.666,000000 |
| Inserção em $M_2$ | 317,000000 | 12.435,000000 | 89.505,000000 | 35,000000 | 987,000000 | 3.770,000000 |
| $M_1 \cup M_2$ | 332,000000 | 13.081,000000 | 87.258,000000 | 37,000000 | 1.212,000000 | 4.345,000000 |
| $M_1 \cap M_2$ | 332,000000 | 14.158,000000 | 87.578,000000 | 56,000000 | 1.175,000000 | 4.054,000000 |
| $M_1 - M_2$ | 333,000000 | 13.403,000000 | 86.386,000000 | 38,000000 | 1.170,000000 | 4.172,000000 |
| Limpar $M_1$ utilizando `erase` | 0,242554 | 1,208876 | 1,871562 | 25,779531 | 726,802176 | 2.565,562320 |
| Limpar $M_2$ utilizando `remove` | 0,235470 | 0,922962 | 2,277383 | 8,561659 | 212,524088 | 843,456441 |

Tabela 4: *Experimento* 2 entre as duas implementações

| Operação | Tempo (ms) | | | | | |
|---|---|---|---|---|---|---|
| | Lista duplamente encadeada | | | Vector | | |
| | 10000 Elementos | 50000 Elementos | 100000 Elementos | 10000 Elementos | 50000 Elementos | 100000 Elementos |
| Inserção em $M_1$ | 0,019323 | 0,058846 | 0,229790 | 38,000000 | 942,000000 | 3.995,000000 |
| Inserção em $M_2$ | 0,018890 | 0,092641 | 0,1156660 | 37,000000 | 964,000000 | 3.779,000000 |
| $M_1 \cup M_2$ | 0,000398 | 0,000544 | 0,000344 | 46,000000 | 956,000000 | 3.923,000000 |
| $M_1 \cap M_2$ | 0,000185 | 0,000227 | 0,000129 | 38,000000 | 941,000000 | 3.905,000000 |
| $M_1 - M_2$ | 0,000186 | 0,000174 | 0,000122 | 40,000000 | 950,000000 | 3.845,000000 |
| Limpar $M_1$ utilizando `erase` | 0,019653 | 0,089312 | 0,116133 | 0,016444 | 0,092482 | 0,189590 |
| Limpar $M_2$ utilizando `remove` | 0,021142 | 0,104366 | 0,207788 | 4,137101 | 104,423589 | 419,812390 |

[1] http://www.cplusplus.com/reference/cstdlib/rand/

Tabela 5: *Experimento* 3 entre as duas implementações

| Operação | Tempo (ms) | | | | | |
|---|---|---|---|---|---|---|
| | Lista duplamente encadeada | | | Vector | | |
| | 10000 Elementos | 50000 Elementos | 100000 Elementos | 10000 Elementos | 50000 Elementos | 100000 Elementos |
| Inserção em $M_1$ | 217,000000 | 8.587,000000 | 50.159,000000 | 35,000000 | 883,000000 | 3.579,000000 |
| Inserção em $M_2$ | 222,000000 | 8.347,000000 | 49.685,000000 | 48,000000 | 1.172,000000 | 4.677,000000 |
| $M_1 \cup M_2$ | 252,000000 | 7.691,000000 | 63.227,000000 | 90,000000 | 2.352,000000 | 8.907,000000 |
| $M_1 \cap M_2$ | 255,000000 | 7.776,000000 | 64.222,000000 | 88,000000 | 2.301,000000 | 8.864,000000 |
| $M_1 - M_2$ | 252,000000 | 7.018,000000 | 63.476,000000 | 87,000000 | 2.297,000000 | 8.856,000000 |
| Limpar $M_1$ utilizando `erase` | 90,721067 | 3408,465680 | 18440,091614 | 7,689044 | 188,855760 | 745,930100 |
| Limpar $M_2$ utilizando `remove` | 116,317147 | 4151,871200 | 24441,940456 | 4,762278 | 108,974491 | 433,727526 |

Quando o multiconjunto não tem nenhum elemento repetido, a implementação utilizando Vector leva vantagem em boa parte dos casos. Isso se dá pelo fato da lista não conseguir aproveitar o atributo $num\_ocurrences$, e como toda busca gasta $O(n)$, as operações ficam bem mais lentas. Já o Vector, como está ordenado, utiliza busca binária em todas as operações, que tem o custo de $log(n)$. As operações de Limpar foram feitas em ordem crescente no Vector, então ele remove o menor e faz *shift* dos elementos. Este é o pior caso da remoção no Vector, por esse motivo essa operação ficou muito lenta. Este caso é também o pior caso da complexidade da lista em espaço, ou seja, $O(n)$.

No caso em que o multiconjunto só possui um elemento, a implementação com lista é extremamente superior em relação a do Vector. Este é o melhor caso da lista, onde é alocado apenas um nó para armazenar todo o multiconjunto. No caso onde os elementos são aleatórios, o Vector leva vantagem até com o número de elementos maior, mesmo precisando fazer *shift*. A lista provavelmente levará vantagem se o número de elementos crescer e o intervalo de valores diminuir, pois nesses casos, o atributo $num\_ocurrences$ irá dar muita vantagem pra lista, além do Vector precisar fazer *shifts* com muitos elementos.

Comparando as duas implementações, a utilizando lista leva vantagem na eficiência de espaço. Outra vantagem é, que ao contrário do Vector, a lista não precisa fazer *shift* quando insere um elemento. Já a implementação utilizando Vector aloca memória menos vezes, e utiliza funções eficientes para alocação (`malloc`[2] e `realloc`), ao invés de `new`[3], que é o que a lista usa, já que esta precisa alocar um novo `Node` para cada elemento que não está na lista. O `buffer` do Vector está em posição contígua de memória, e dependendo do número de elementos, isso dá uma vantagem absurda pela localidade de cache. No Vector também é possível buscar utilizando busca binária, que é $O(logn)$ ao invés do $O(n)$ na lista.

Todas essas comparações entre estruturas lineares são para efeito de estudo. Na prática, para implementar um multiconjunto ordenado de forma eficiente, é utilizada uma árvore binária de pesquisa balanceada. Na próxima seção, será feita uma comparação teórica e prática com a classe `multiset` do C++, que implementa um multiconjunto ordenado de forma eficiente.

[2] http://www.cplusplus.com/reference/cstdlib/malloc/
[3] http://www.cplusplus.com/reference/new/operator%20new/

# 4 Comparação com a classe `multiset` da STL/C++

Como dito em seções anteriores, a forma eficiente para implementar um multiconjunto ordenado é utilizando árvore binária de pesquisa balanceada, como é feito na classe `multiset` de C++. A tabela 6 mostra a comparação teórica entre as três formas de implementação. Já a tabela 7 mostra a comparação prática.

Tabela 6: Comparação teórica de pior caso das implementações em relação ao `multiset` de C++

| Operação | Utilizando lista duplamente encadeada | Utilizando Vector | `multiset` C++ |
|---|---|---|---|
| Insere | $O(n)$ | $O(n)$ | $O(logn)$ |
| Pertence | $O(n)$ | $O(logn)$ | $O(logn)$ |
| Frequência | $O(n)$ | $O(n)$ | $O(logn)$ |
| Remove | $O(n)$ | $O(n)$ | $O(logn)$ |
| Apaga | $O(n)$ | $O(n)$ | $O(logn)$ |
| União | $O(n+m)$ | $O(n+m)$ | $O(n+m)$[1] |
| Interseção | $O(n+m)$ | $O(n+m)$ | $O(n+m)$[2] |
| Diferença | $O(n+m)$ | $O(n+m)$ | $O(n+m)$[3] |

Tabela 7: Experimentos 1, 2 e 3 com 100.000 elementos utilizando `multiset` de C++

| Operação | Tempo (ms) | | |
|---|---|---|---|
| | Experimento 1 | Experimento 2 | Experimento 3 |
| Inserção em $M_1$ | 17,2 | 17,4 | 24,5 |
| Inserção em $M_2$ | 17,1 | 17,3 | 24,1 |
| $M_1 \cup M_2$ | 4,1 | 4,3 | 10,4 |
| $M_1 \cap M_2$ | 3,3 | 3,3 | 8,7 |
| $M_1 - M_2$ | 2,6 | 2,9 | 8,0 |
| Limpar $M_1$ utilizando `erase` | 10,4 | 2,6 | 23,7 |
| Limpar $M_2$ utilizando `clear` | 2,2 | 2,3 | 5,6 |

.

A complexidade da união, inserção e diferença, segundo a documentação oficial, são dadas por: "Linear em $2(count1 + count2) - 1$, (onde $countX$ é a distância entre $firstX$ e $lastX$)". Isto acaba sendo $O(n)$ no pior caso, por isso essas operações são dadas por $O(n+m)$. As comparações acima mostram a discrepância entre as implementações com estruturas de dados lineares e a implementação utilizando árvore binária de pesquisa balanceada. Ou seja, nunca implemente um multiconjunto ordenado utilizando estruturas lineares.

[1] http://www.cplusplus.com/reference/algorithm/set_union/
[2] http://www.cplusplus.com/reference/algorithm/set_intersection/
[3] http://www.cplusplus.com/reference/algorithm/set_difference/