Aula 13:

STL – Parte 2
Standard Template Library

ECOP13A - Programação Orientada a Objetos
Prof. André Bernardi
andrebernardi@unifei.edu.br

Universidade Federal de Itajubá

Rev. Junho 22

Estruturas não-lineares

Para alguns problemas, existem maneiras melhores de se representar os dados do que uma simples sequência. Com as implementações da STL das estruturas não-lineares que iremos discutir a seguir, podemos aumentar nossa eficiência na aplicação de algoritmos sempre que o problema em questão apresentar condições favoráveis para sua utilização.

De que estruturas estamos falando?

- Balanced Binary Search Tree (BST): <map> e <set>
- Heap: priority_queue <queue>

Árvores Binárias de Busca (BST)

Estrutura que organiza dados em forma de árvore, ou seja, grafos acíclicos e conexos.

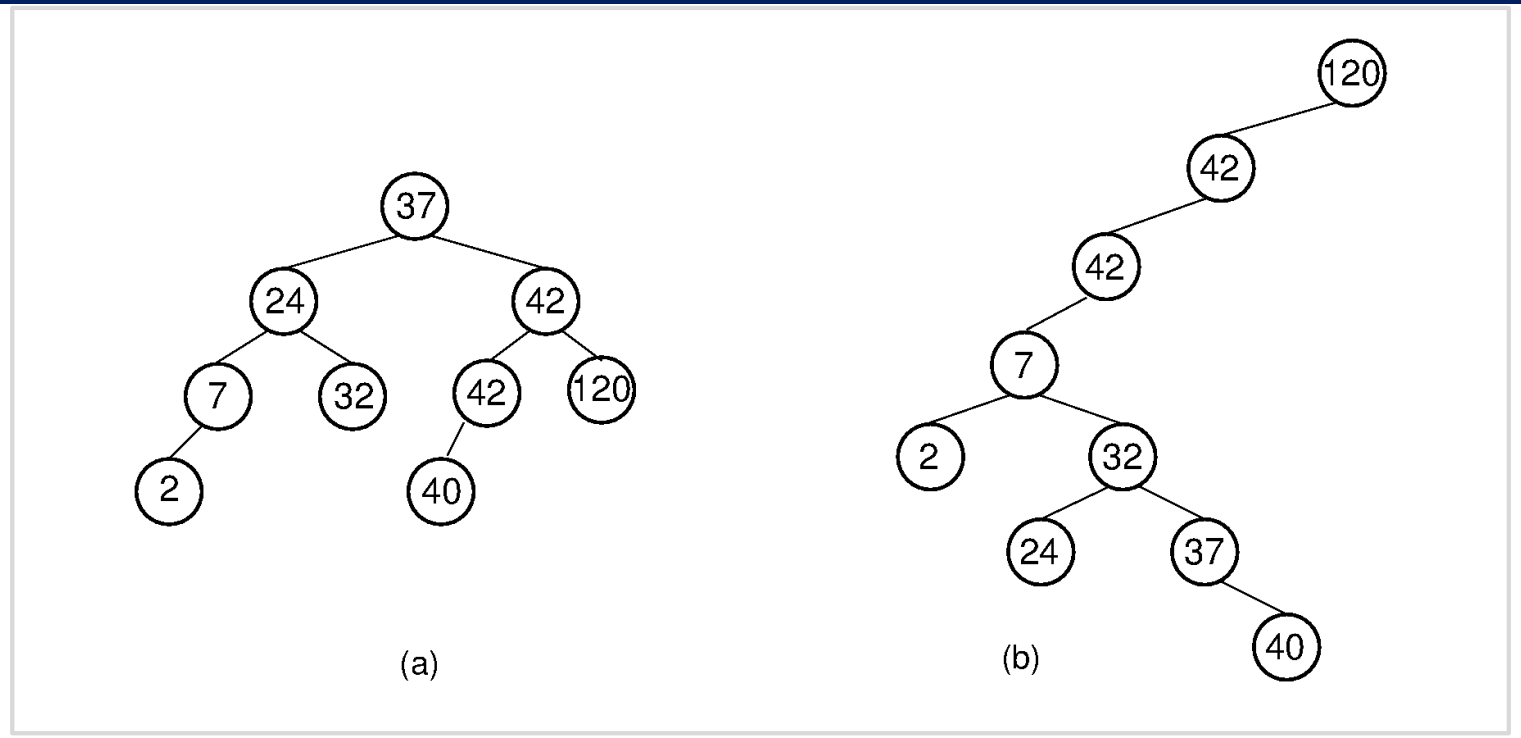
Neste tipo de árvore, em especial, cada um de seus nós possui no máximo **dois** filhos, um à esquerda e outro à direita. Em cada sub-árvore com raiz em um nó *x*, os itens na sub-árvore à esquerda de *x* são menores que *x* e os itens na sub-árvore à direita são maiores ou iguais a *x*.

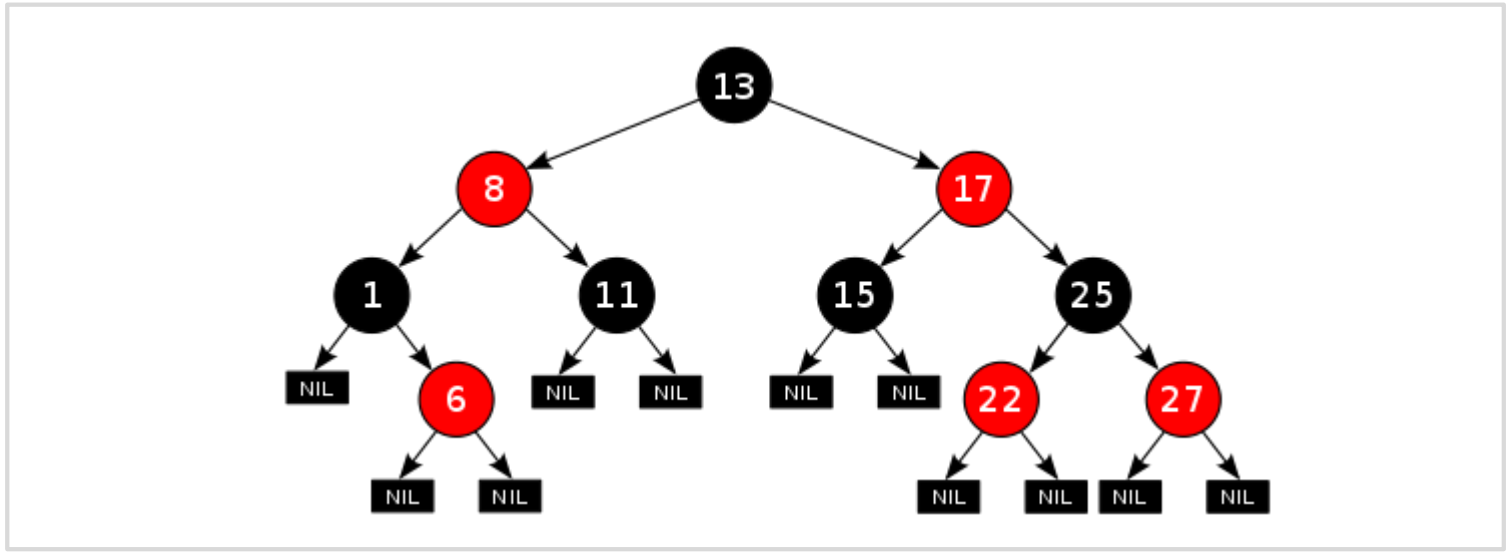
Árvores Binárias de Busca (BST)

Este tipo de estrutura de dados permite que operações de inserção, busca e remoção de um determinado valor sejam realizadas de maneira muito eficiente, em O(log*n*), caso sejam balanceadas (Árvore AVL, RB-Tree).

A altura de uma árvore AVL, por exemplo, está no intervalo abaixo, o que explica a quantidade de computação necessária (O(logn)) para as operações descritas acima.

$$\log_2(n+1) \leq h \leq c\log_2(n+2) + b$$


37
24
42
7
32
42
120
2
40
(a)
120
42
42
7
2
32
24
37
40
(b)


13
8
17
1
11
15
25
NIL
6
NIL
NIL
NIL
NIL
22
27
NIL
NIL
NIL
NIL

Árvores Binárias de Busca (BST)

STL <map> e <set> são implementações de um tipo de árvore binária de busca balanceada chamada **Red-Black Tree**, ou Árvore Rubro-Negra. Portanto, nelas, todas as operações são realizadas em O(log*n*).

Qual a diferença entre as duas?

<map> armazena **pares** (chave, dado)

<set> armazena apenas a **chave**.

Exemplo set<>

```cpp
#include <iostream>
#include <set>

using namespace std;

int main()
{
    set<int> conjunto;
    set<int>::iterator it;

    // Em um set, as chaves não podem ser duplicadas.
    // para isso, utilize um multiset (também de <set>)
    conjunto.insert(30);
    conjunto.insert(20);
    conjunto.insert(10);
    conjunto.insert(10); // não será inserido
    conjunto.insert(20); // não será inserido
    conjunto.insert(40);
    conjunto.erase(10); // apaga um item, baseado no valor.

    cout << "Quantidade de elementos: " << conjunto.size() << endl;
    cout << "Elementos: ";
    // perceba que os elementos em um set estão sempre ordenados
    for(it = conjunto.begin(); it != conjunto.end(); ++it)
        cout << *it << " ";
    cout << endl;

    return 0;
}
```

```
Quantidade de elementos: 3
Elementos: 20 30 40
```

Exemplo multiset<>

```cpp
#include <iostream>
#include <set>

using namespace std;

int main()
{
    multiset<int> conjunto;
    multiset<int>::iterator it;

    // Em um multiset, as chaves podem ser duplicadas.
    conjunto.insert(30);
    conjunto.insert(20);
    conjunto.insert(10);
    conjunto.insert(10); // será inserido
    conjunto.insert(20); // será inserido
    conjunto.insert(40);
    conjunto.erase(10); // apaga os itens com valor 10, baseado no valor.

    cout << "Quantidade de elementos: " << conjunto.size() << endl;
    cout << "Elementos: ";
    // perceba que os elementos em um multiset tb estão sempre ordenados
    for(it = conjunto.begin(); it != conjunto.end(); ++it)
        cout << *it << " ";
    cout << endl;

    return 0;
}
```

```
Quantidade de elementos: 4
Elementos: 20 20 30 40
```

Os **maps** guardam pares ordenados compostos por uma chave e um valor. Talvez essa seja a sua única diferença significativa com relação aos sets. Mas como fazemos para inserir um par de dados em um map? Através do tipo **pair** da STL, presente no cabeçalho **<utility>**.

Um ***pair*** é composto por dois itens, chamados de *first* e *second* (Que podem ser de tipos distintos T1 e T2). Veja alguns exemplos:

```
pair<int, double> par{1,1.9};
cout << "par: " << par.first << " e " << par.second;
```

Ou, ainda:

```
pair<int, int> par2 = make_pair(10, 100);
par2.first = 11; // muda valor da chave
cout << "par2: " << par2.first << " e " << par2.second << endl;
```

Exemplo map<>

```cpp
#include <map>
#include <iostream>

using namespace std;

int main()
{
    map<int, string> alunos;
    map<int, string>::iterator it;

    // inserindo quatro alunos
    // o primeiro item do pair é a chave, o segundo é o valor
    alunos.insert(make_pair(11984, "Joao"));
    alunos.insert(pair<int, string>{23456, "Jose"});
    alunos.insert(make_pair(8541, "Carlos"));
    alunos.insert(make_pair(8541, "Edmilson")); // Não é inserido (multimap)
    alunos.insert(pair<int, string>{29546, "Maria"});

    // Removendo José
    alunos.erase(23456);

    // Imprimindo lista de alunos
    // Repare que o map está ordenado de acordo com a chave
    cout << "Lista de Alunos: " << endl;
    for(it = alunos.begin(); it != alunos.end(); ++it)
        cout << it->first << " - " << it->second << endl;

    return 0;
}
```

Não é necessário incluir **<utility>**. Map já a inclui em seu próprio cabeçalho.

```
Lista de Alunos:
8541 - Carlos
11984 - Joao
29546 - Maria
```

Exemplo multimap<>

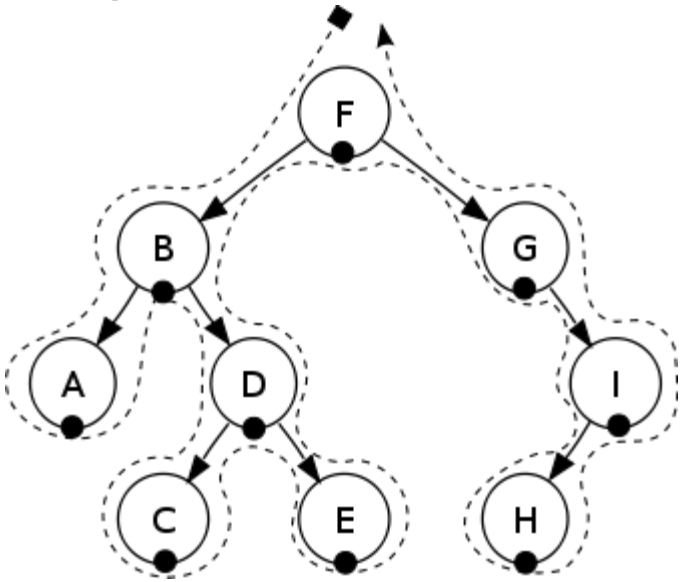
```cpp
#include <map>
#include <iostream>

using namespace std;

int main()
{
    multimap<int, string> alunos;
    multimap<int, string>::iterator it;

    // inserindo cinco alunos
    alunos.insert(make_pair(11984, "Joao"));
    alunos.insert(pair<int, string>{23456, "Jose"});
    alunos.insert(make_pair(8541, "Carlos"));
    alunos.insert(make_pair(8541, "Edmilson")); // É inserido!
    alunos.insert(pair<int, string>{29546, "Maria"});

    // Removendo José
    alunos.erase(23456);

    // Imprimindo lista de alunos
    // Repare que o map está ordenado de acordo com a chave
    cout << "Lista de Alunos: " << endl;
    for(it = alunos.begin(); it != alunos.end(); ++it)
        cout << it->first << " - " << it->second << endl;

    return 0;
}
```

```
Lista de Alunos:
8541 - Carlos
8541 - Edmilson
11984 - Joao
29546 - Maria
```

Uma importante propriedade dos ***maps*** e ***sets*** é que, ao se iterar pelos itens do contêiner, eles são **percorridos em ordem**. Isso se deve à própria estrutura da BST balanceada, onde o caminhamento "em ordem" leva à visitação dos itens de maneira ordenada. Veja:

Em-ordem:

1. Percorra a sub-árvore à esquerda recursivamente;
2. Mostre o valor do nó atual;
3. Percorra a sub-árvore direita recursivamente.



Em-ordem:
A, B, C, D, E, F, G, H, I.

Este tipo de contêiner é chamado de **associativo**, e provê acesso direto para armazenamento e recuperação de elementos via chaves. Possuem as seguintes funções (além de **insert** e **erase**):

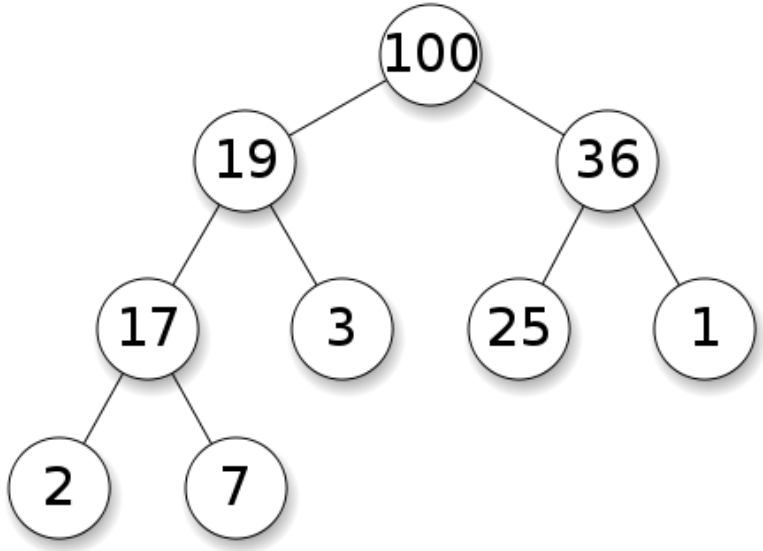
- **find**: retorna um iterador para um elemento.
- **lower_bound**: Retorna um iterador para um limite inferior.
- **upper_bound**: Retorna o iterador para o limite superior.
- **count**: conta elementos com um valor específico.

```
std::set<int> myset;
std::set<int>::iterator itlow,itup;

for (int i=1; i<10; i++) myset.insert(i*10); // 10 20 30 40 50 60 70 80 90

itlow=myset.lower_bound (30);                //       ^
itup=myset.upper_bound (60);                 //                   ^

myset.erase(itlow,itup);                     // 10 20 70 80 90
```

Heap

É uma estrutura de dados organizada como uma **árvore binária completa**, ou seja, uma árvore binária onde cada nível é completado da esquerda para a direita e deve estar cheio antes que o próximo nível seja iniciado.

Além disso, há uma restrição especial: cada nó da heap deve conter um valor maior (ou menor) do que todos os valores contidos por nós descendentes dele.



Heap

Normalmente não se trabalha com busca na heap, mas inserção e remoção podem ser feitas em O(logn), o que está também relacionado a sua altura máxima.

Pode ser modelada como uma fila de prioridade.

Na STL, está em <queue> implementada como **priority_queue**.

É importante em uma série de algoritmos como **Dijkstra** (caminho mínimo), **Kruskal** (árvore geradora mínima) e na ordenação **heap sort**, implementada em partial_sort (C++) e realizada em O(k logn) quando ordenamos k elementos.

Priority queue (STL)

Filas de prioridade são um tipo de **adaptador de container** (portanto, também não são de primeira classe) projetado especificamente para que seu **primeiro elemento seja sempre o maior entre todos os elementos**, de acordo com um critério de ordenação.

O contexto é, portanto, similar ao de uma heap, onde elementos podem ser inseridos a qualquer momento, e somente o elemento máximo da heap pode ser obtido (aquele no topo da fila de prioridade, pop_back).

Funções membro:

- **empty**
- **size**
- **top**
- **push** (push_back)
- **pop** (pop_back)

Exemplo priority_queue<>

```cpp
#include <iostream>
#include <queue>

using namespace std;

int main()
{
    priority_queue<float> distancias;

    // insere valores de distancias
    distancias.push(1000.0);
    distancias.push(100.0);
    distancias.push(10.0);
    distancias.push(1001.0);
    distancias.push(900.0);

    cout << "Imprimindo na ordem de prioridade: " << endl;
    while(!distancias.empty())
    {
        cout << distancias.top() << endl;
        distancias.pop();
    }

    return 0;
}
```

```
Imprimindo na ordem de prioridade:
1001
1000
900
100
10
```

Algoritmos

Ordenação

Na prática, não é crucial conhecer TODOS os métodos passo a passo de cabeça. Em geral, o que precisamos é apenas utilizar a função de ordenação O(n log n) presente na STL.

Lembre-se de que a operação de ordenação normalmente é apenas um passo preliminar para um algoritmo mais complexo, ou um último passo, para organizar os dados da saída. Dificilmente será o objetivo do programa, mas a familiaridade com algoritmos de ordenação é muito importante.

Na STL,

Temos três algoritmos prontos (biblioteca <algorithm>):

- **sort**: O algoritmo específico não é fixo e pode variar dependendo da implementação. No entanto, a complexidade no pior caso é, obrigatoriamente, O(n log n).
  - Bastante rápido. Normalmente *quick sort*;
  - Ordena tanto dados básicos quanto tipos definidos pelo usuário.

- **partial_sort**: Implementa a *heap sort* e pode ser utilizado para ordenar apenas uma parte da estrutura. Se for necessário ordenar k itens, sua complexidade no tempo será de O(k logn).

- **stable_sort**: Preserva ordem de elementos com o mesmo valor, se necessário.

```cpp
#include <iostream>
#include <algorithm>
#include <vector>
#include <ctime>
#include <cstdlib>

using namespace std;

int main()
{
    vector<int> numeros;
    srand(time(NULL));

    // Sorteando numeros aleatorios e imprimindo
    cout << "Vetor sorteado: ";
    for(int i = 0; i < 10; i++) {
        int x = rand()%100;
        numeros.push_back(x);
        cout << x << " ";
    }
    cout << endl;

    // Ordenando parcialmente vetor
    // Apenas 5 primeiros ficarão em seus próprios lugares
    partial_sort(numeros.begin(), numeros.begin()+5, numeros.end());
    cout << "Primeira metade ordenada: ";
    for(int i = 0; i < 10; i++)
        cout << numeros[i] << " ";
    cout << endl;
```

```cpp
    // Ordenando segunda metade (sort normal)
    sort(numeros.begin() + 5, numeros.end());
    cout << "Segunda metade ordenada: ";
    for(int i = 0; i < 10; i++)
        cout << numeros[i] << " ";
    cout << endl;

    // Ordenando tudo (com função de comparação e ordenação estável)
    stable_sort(numeros.begin(), numeros.end());
    cout << "Vetor ordenado: ";
    for(int i = 0; i < 10; i++)
        cout << numeros[i] << " ";
    cout << endl;

    return 0;
}
```

```
Vetor sorteado: 33 95 90 3 18 93 2 12 31 30
Primeira metade ordenada: 2 3 12 18 30 95 93 90 33 31
Segunda metade ordenada: 2 3 12 18 30 31 33 90 93 95
Vetor ordenado: 2 3 12 18 30 31 33 90 93 95
```

Algoritmos

Busca

- Existem dois casos:
  1. Quando o vetor já se encontra ordenado (logarítmica).
  2. Caso contrário (linear).

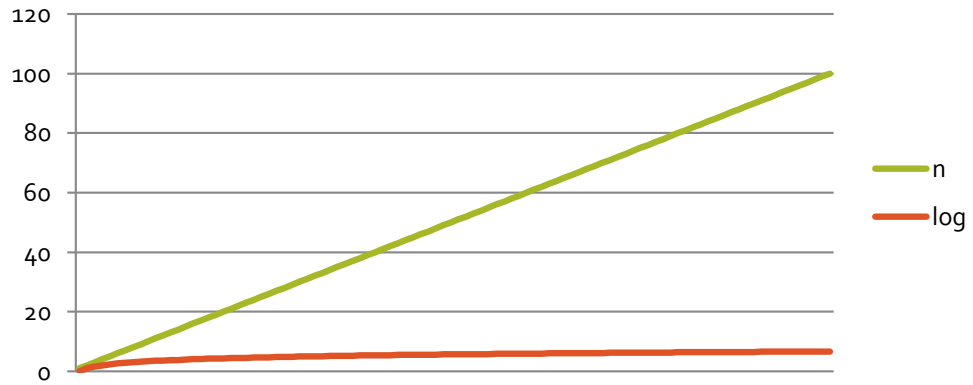
A busca pode ser feita **linearmente** quando o vetor não estiver ordenado. Trivial, com complexidade linear O(n).

Quando ordenado, podemos utilizar a **busca binária**, que possui complexidade O(logn). Não é necessário implementar do zero, há implementação na STL.

Busca Binária:

Primeiro comparamos a chave de pesquisa k com o elemento na posição n/2 de nosso vetor e, dependendo do resultado, fazemos **recursão** ou na primeira **metade** [0, ..., n/2 – 1], ou na segunda **metade** [n/2 + 1, ..., n-1]. Procuramos somente na primeira metade caso k < v[n/2]. Caso k > v[n/2], procuramos na segunda metade.

É obrigatório que o vetor esteja previamente ordenado. Possui uma eficiência superior à do algoritmo trivial, pois a quantidade de operações não aumenta muito com o aumento do tamanho do vetor. Aumenta proporcionalmente a **logn**.



Na STL,

Temos dois algoritmos prontos (biblioteca <algorithm>). Nos dois casos, é necessário que o vetor esteja previamente ordenado. Veja:

- **binary_search**: Recebe o intervalo de busca e o valor a ser buscado. Retorna *true* se encontrar e *false* caso contrário.
  Desvantagem: Não retorna uma referência para o elemento encontrado (iterador).

- **lower_bound**: Recebe os mesmos parâmetros da binary_search. No entanto, retorna um iterador apontando para o primeiro elemento no intervalo que não é menor do que o valor buscado.
  Vantagem: Retorna uma referência para o próprio elemento, caso seja necessário processá-lo ou imprimi-lo na tela.

```cpp
#include <iostream>
#include <algorithm>
#include <vector>
#include <ctime>
#include <cstdlib>

using namespace std;

int main()
{
    vector<int> numeros;
    srand(time(NULL));

    // Sorteando numeros aleatorios
    for(int i = 0; i < 30; i++){
        int x = rand()%100;
        numeros.push_back(x);
    }

    // Precisam estar ordenados para que a busca binária funcione
    sort(numeros.begin(), numeros.end());

    int valor = -1;
    while(true) {
        cout << "Entre com um valor a ser buscado: ";
        cin >> valor;

        if(binary_search(numeros.begin(), numeros.end(), valor)){
            vector<int>::iterator it = lower_bound(numeros.begin(), numeros.end(), valor);
            cout << "Numero encontrado: " << *it << endl;
            break;
        }
        else
            cout << "Nao encontrado." << endl;
    }
}
```

```
Entre com um valor a ser buscado: 1
Nao encontrado.
Entre com um valor a ser buscado: 11
Nao encontrado.
Entre com um valor a ser buscado: 12
Nao encontrado.
Entre com um valor a ser buscado: 23
Numero encontrado: 23
```

Referências

- https://cplusplus.com/reference/
  - https://cplusplus.com/reference/queue/priority_queue/
  - https://cplusplus.com/reference/map/
    - https://cplusplus.com/reference/map/map/
    - https://cplusplus.com/reference/map/multimap/
  - https://cplusplus.com/reference/set/
    - https://cplusplus.com/reference/set/set/
    - https://cplusplus.com/reference/set/multiset/
  - https://cplusplus.com/reference/algorithm/
  - https://cplusplus.com/reference/utility/

- Material de aula ECOP03 - Prof. João Paulo R. R. Leite - 2021