Estrutura de Dados em C++ e STL

...

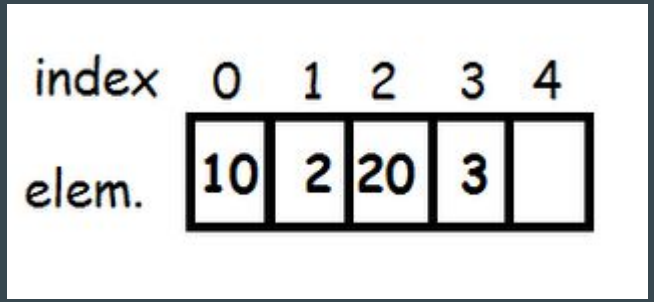
Tópicos abordados

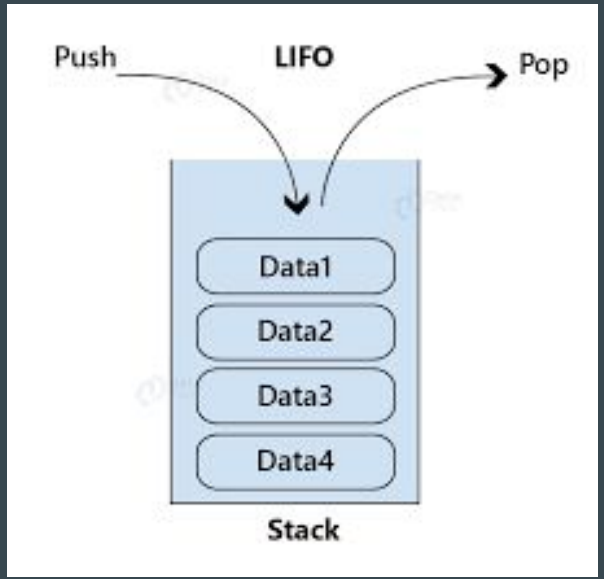
- O que são estruturas de dados
- O que é STL
- Estruturas de dados em C++
  - Array e Vector
  - Stack (Pilha)
  - Queue (Fila)
  - Priority Queue (Fila de prioridade)
  - Set (Conjunto)
  - Map (Dicionário)

Estruturas de Dados

Estruturas de dados

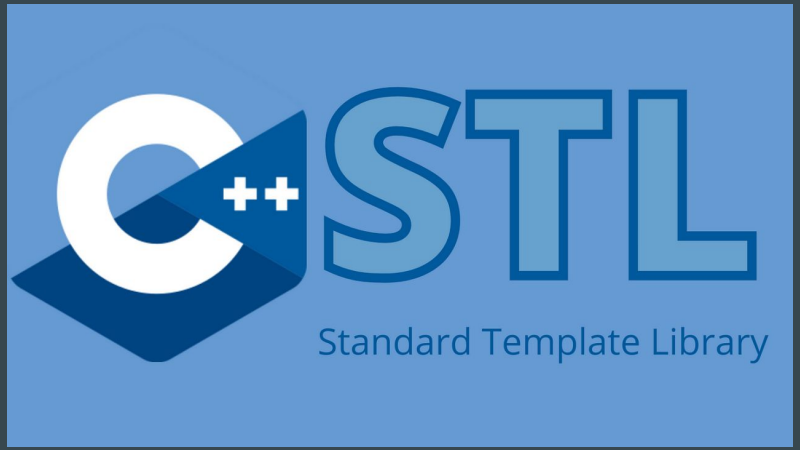
- Usados para guardar e organizar... dados!
- Existem diferentes tipos de estruturas de dados, usadas em diferentes tipos contexto
- Extremamente relevantes para modelar problemas de maratona :)





Estruturas de dados em C++

- Em C++, temos a **S**tandard **T**emplate **L**ibrary
  - Biblioteca que possui diferentes estruturas de dados e algoritmos implementados para manipulá-las.
  - Tá tudo pronto, é só usar!



Pair e Tuple

Pair

- Guarda dois valores de tipo arbitrário
- Acesso direto pelos membros first e second

```
pair<int,int> pii = {1,2};
cout << pii.first << endl; // Imprime 1 (primeiro valor)
cout << pii.second << endl; //Imprime 2 (segundo valor)
```

Tuple

- Quantidade fixa de valores
- Podem ser de tipos diferentes
- Acesso usando tie(), get() ou structured binding

```
tuple<int,int,int>tup = {1,2,3}; //pode ter mais de 3
// Acesso com Structured binding
auto [x,y,z] = tup;
// Acesso com tie
int a,b,c;
tie(a,b,c) = tup;
// Acesso com get
int i,j,k;
i = get<0>(tup);
j = get<1>(tup);
k = get<1>(tup);
```

Array e Vector

Array

- Usado para armazenar mais de um valor em uma única variável
- Consiste de uma coleção de elementos de um mesmo tipo
- Já apresentado na aula passada!

```
1  int arr[50]; // declaração
2
3  arr[0]; // <-- primeiro elemento da lista
4  arr[1]; // <-- segundo elemento
5  arr[49]; // <-- ultimo elemento
6  arr[50]; // <-- ILEGAL:
```

Array

- Problema: depois de declarados, arrays **NÃO** podem mais mudar de tamanho.
- Solução: no próximo slide...

Vector

- Similares às arrays, é uma coleção de elementos de um mesmo tipo.
- Maaaaas seu tamanho **PODE** mudar durante a execução do programa :)

Vector - Inicialização

- Existem várias formas de declarar um Vector:

```cpp
vector<int> v1; // Vetor vazio

vector<int> v2 = {1, 2, 3, 4}; // Vetor com 4 elementos

vector<int> v3(5); // Vetor com tamanho 5

vector<int> v4(5, 2); // Vetor com 5 elementos de valor 2
```

Vector - Operações de Acesso

- Acesso: similar às arrays, uso do operador []
- Posições acessíveis vão de 0 até TAMANHO-1

```
vector<int> v = {1, 2, 3, 4};
v[0]; // <--- primeiro elemento
v[3]; // <--- último elemento
v[4]; // <--- ILEGAL
```

Vector - Operações de Acesso

- Estruturas de dados da STL normalmente oferecem diversas funções úteis para manipular dados
- Exemplo: uso das funções back() e front()
- Complexidade: O(1)

```
vector<int> v = {1, 2, 3, 4};
v.front(); // primeiro elemento
v.back(); // último elemento
```

Vector - Tamanho

- v.empty(): verifica se o vetor está vazio
- v.size(): retorna o tamanho do vetor
- Complexidade: O(1)

```
vector<int> v = {2, 8, 1, 6};
if(v.empty()) {
    cout << "Tá vazio!";
}
else {
    cout << v.size(); // <--- 4
}
```

Vector - Inserção e Remoção

- Normalmente, se inserem ou removem elementos do FINAL de um vetor (O(1))
- v.push_back(i): adiciona o elemento “i” no final do vetor
- v.pop_back(): remove o último elemento do vetor “v”

```
vector<int> v = {10, 20, 30};

v.push_back(40); // Agora v é {10, 20, 30, 40}

v.push_back(50); // Agora v é {10, 20, 30, 40, 50}

v.pop_back();    // Agora v é {10, 20, 30, 40};
```

Vector - Inserção e Remoção

- Também é possível adicionar e remover sem ser no final do vetor.
- Mas a complexidade piora bastante... (O(n))

Vector - Iterando sobre elementos

```cpp
vector<int> fib = {1, 2, 3, 5, 8};
int n = fib.size();
// Opção 1
for(int i = 0; i < n; i++) {
    cout << fib[i] << " ";
}
```

Vector - Iterando sobre elementos

```
vector<int> fib = {1, 2, 3, 5, 8};
// Opção 2 (for each)
for(int e : fib) {
    cout << e << " ";
}
```

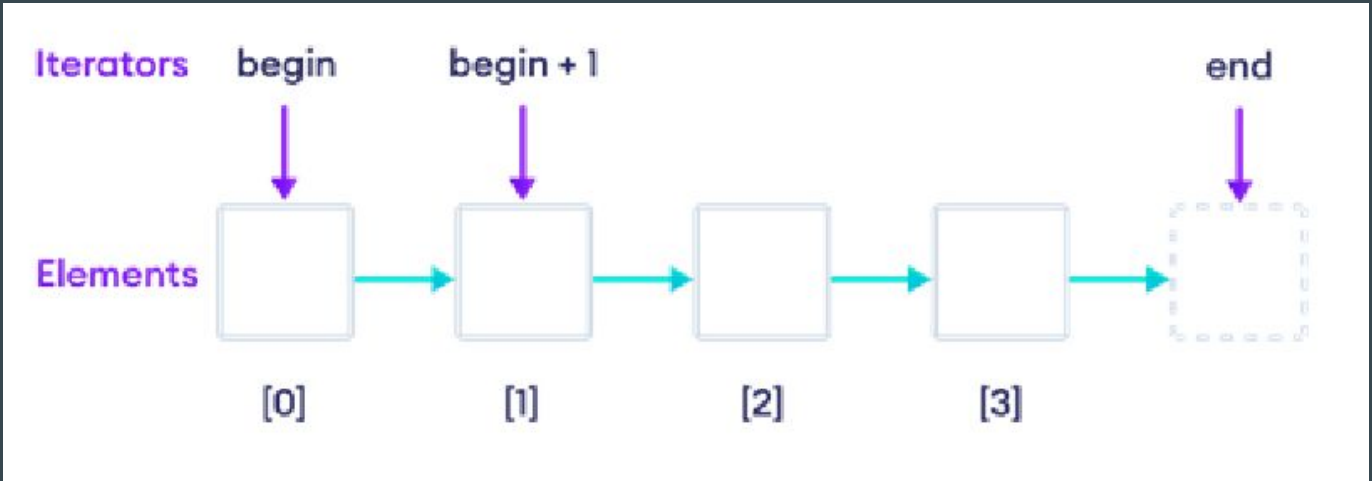
Vector - Iterando sobre elementos

- Outra forma de iterar: usando **ITERATORS** !

Iterators

Iterators

- Usados para acessar ou iterar sobre elementos de estruturas de dados
- begin(): retorna um iterador apontando para o primeiro elemento da estrutura
- end(): retorna um iterador apontando para a posição logo após o último elemento



Iterators

- O elemento apontado por um iterador pode ser acessado usando “*”
- Cuidado para não acessar uma posição inválida!

```
vector<int> v = {1, 2, 3, 4, 5};
vector<int>::iterator it = v.begin();
cout << *it << endl; // <--- 1
vector<int>::iterator it = v.end();
cout << *it << endl; // ERRO!!!!
```

Iterators

- Em versões do C++11 e em diante, é possível utilizar a palavra chave “auto” para detectar o tipo de uma variável
- Útil para o caso dos iterators

```
vector<int> v = {1, 2, 3, 4, 5};
auto it = v.begin();
cout << *it << endl; // <--- 1
auto it = v.end();
it--;
cout << *it << endl; // <--- 5
```

Iterators

- Exemplo: iterando sobre um vector

```
vector<int> fib = {1, 2, 3, 5, 8};
for(auto it = fib.begin(); it < fib.end(); it++) {
    cout << *it << " ";
}
```

Iterators

- Várias funções fornecidas pela STL utilizam de iteradores em suas declarações.
- Por exemplo...

sort, lower_bound e upper_bound

sort

- Ordena o container em O(n log n)
- Realisticamente? Só é usado pra ordenar vector e array

```cpp
vector<int> v = {4, 2, 8, 9, 1};
sort(v.begin(), v.end());
// Vai imprimir: 1 2 4 8 9
for(int e : v) {
    cout << e << " ";
}
```

lower_bound

- Encontra o primeiro elemento **maior ou igual** ao valor especificado
- Usa busca binária para achar esse elemento em O(log n)

```
vector<int> v = {1, 4, 5, 9, 12};
cout << *lower_bound(v.begin(), v.end(), 3) << endl;  // --> 4
cout << *lower_bound(v.begin(), v.end(), 9) << endl;  // --> 9
cout << *lower_bound(v.begin(), v.end(), 13) << endl; // ERRO!
```

upper_bound

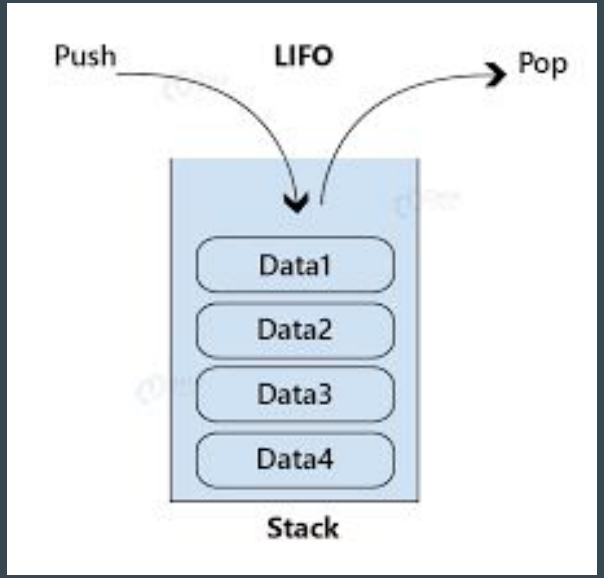
- Encontra o primeiro elemento **maior** ao valor especificado
- Usa busca binária para achar esse elemento em O(log n)

```
vector<int> v = {1, 4, 5, 9, 12};
cout << *upper_bound(v.begin(), v.end(), 3) << endl;  // --> 4
cout << *upper_bound(v.begin(), v.end(), 9) << endl;  // --> 12
cout << *upper_bound(v.begin(), v.end(), 13) << endl; // ERRO!
```

Stack, Queue e Priority Queue

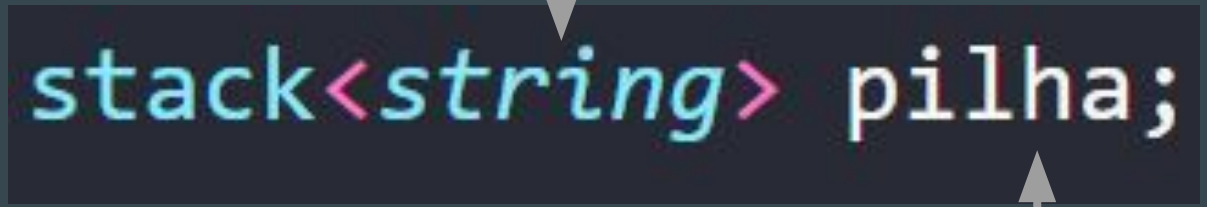
Stack

- Uma stack (pilha) armazena dados em uma ordem específica
  - **LIFO**: **L**ast **I**n **F**irst **O**ut
  - “O último que entra é o primeiro que sai”



Stack - Inicialização

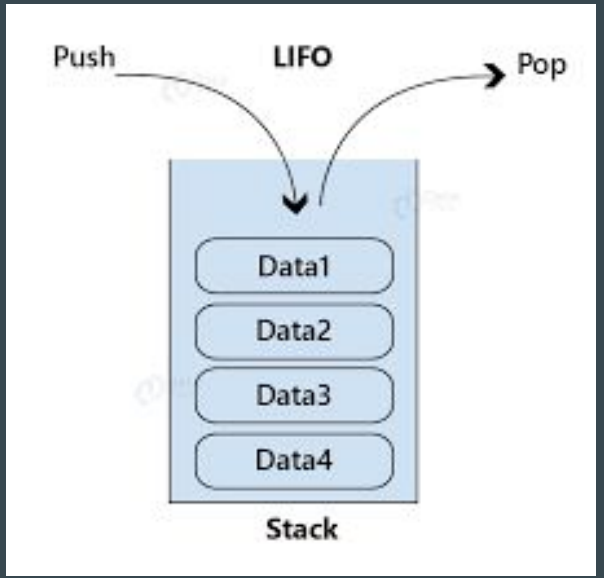
Tipo de dado armazenado



Nome da pilha

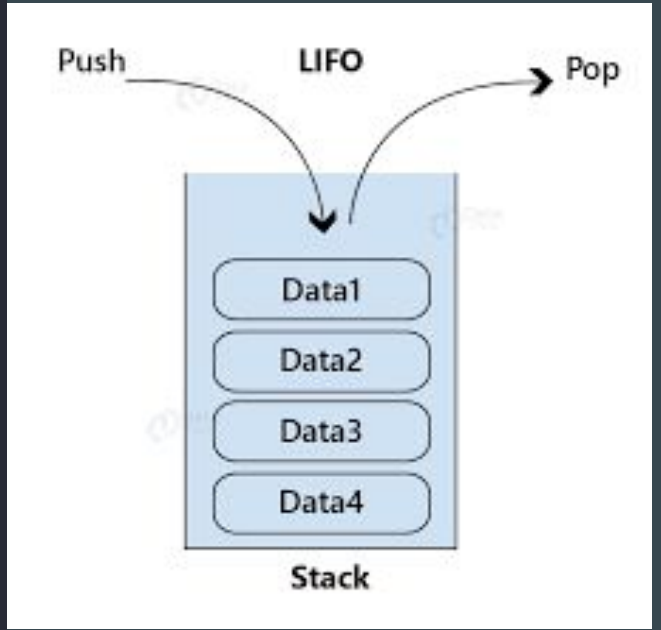
Stack - Principais Operações

- push(): adiciona um elemento no topo da pilha
- pop(): remove um elemento do topo da pilha (cuidado!)
- top(): vê qual é o elemento no topo da pilha (cuidado!)
- size(): vê o tamanho atual da pilha
- empty(): verifica se a pilha está vazia



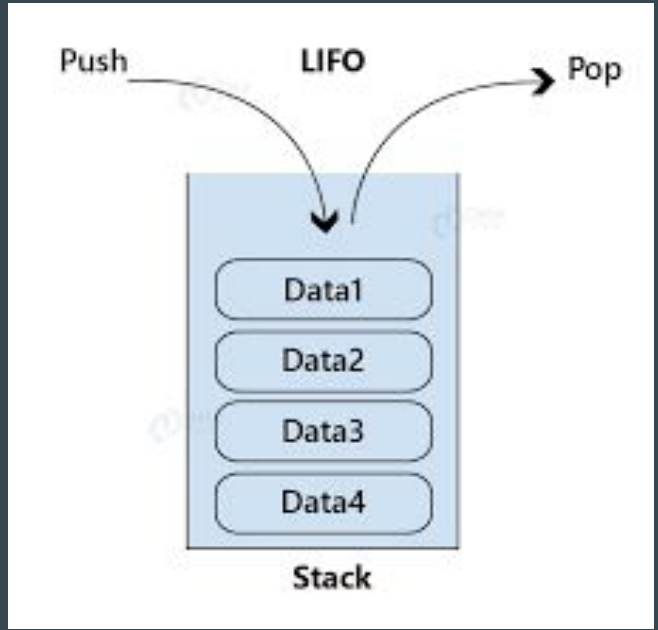
Stack - Inserindo Elementos

```
stack<string> pilha;
pilha.push("Data4");
pilha.push("Data3");
pilha.push("Data2");
pilha.push("Data1");

cout << pilha.top(); // Data1
```



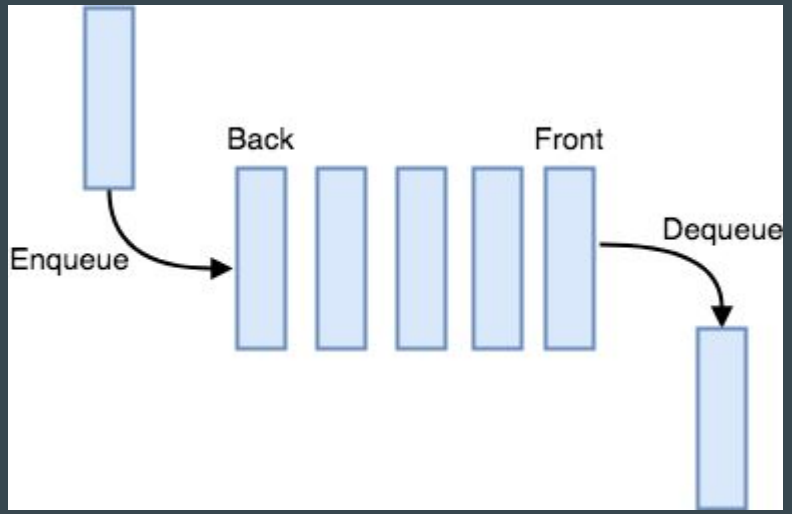
Stack - Removendo Elementos

```
stack<string> pilha;
pilha.push("Data4");
pilha.push("Data3");
pilha.push("Data2");
pilha.push("Data1");

// Remove todos os elementos da pilha
while(!pilha.empty()) {
    pilha.pop();
}
```



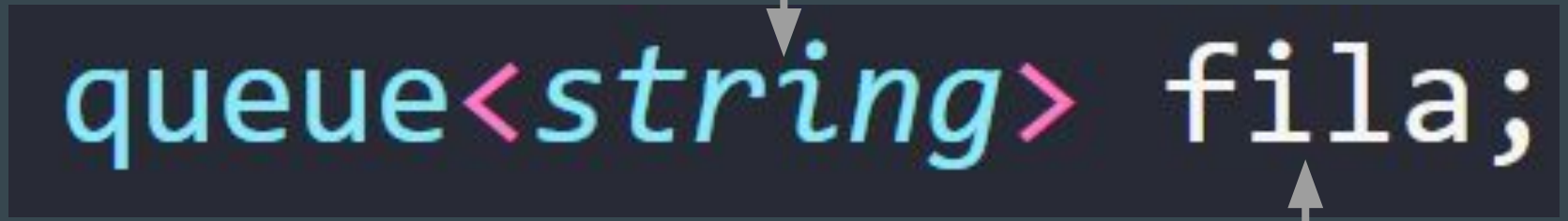
Queue

- Uma queue (fila) armazena dados em uma ordem específica
  - **FIFO**: **F**irst **I**n **F**irst **O**ut
  - “O primeiro que entra é o primeiro que sai”



Queue - Inicialização

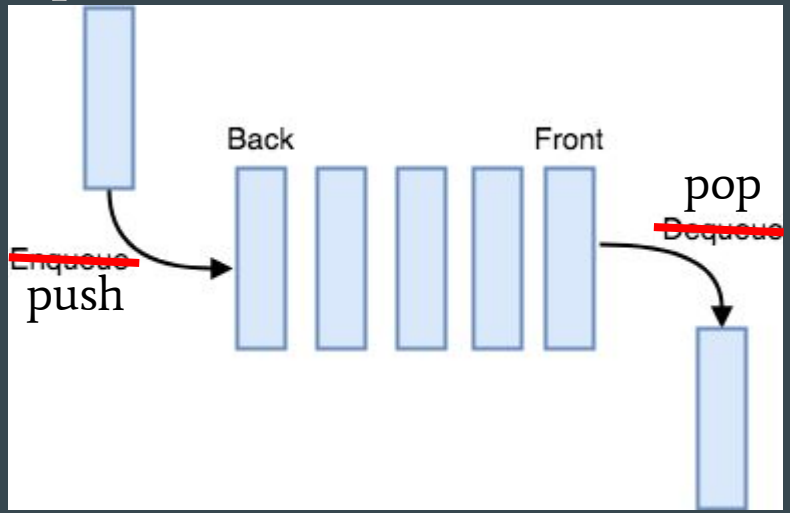
Tipo de dado armazenado



Nome da fila

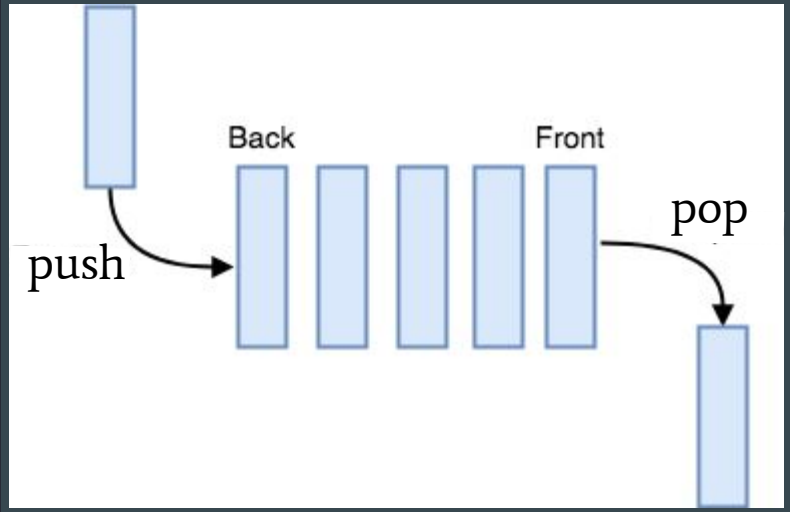
Queue - Principais Operações

- push(): adiciona um elemento no final da fila
- pop(): remove um elemento da frente da fila (cuidado!)
- back(): vê qual é o último elemento da fila (cuidado!)
- front(): vê qual é primeiro elemento da fila (cuidado!)
- size()
- empty()



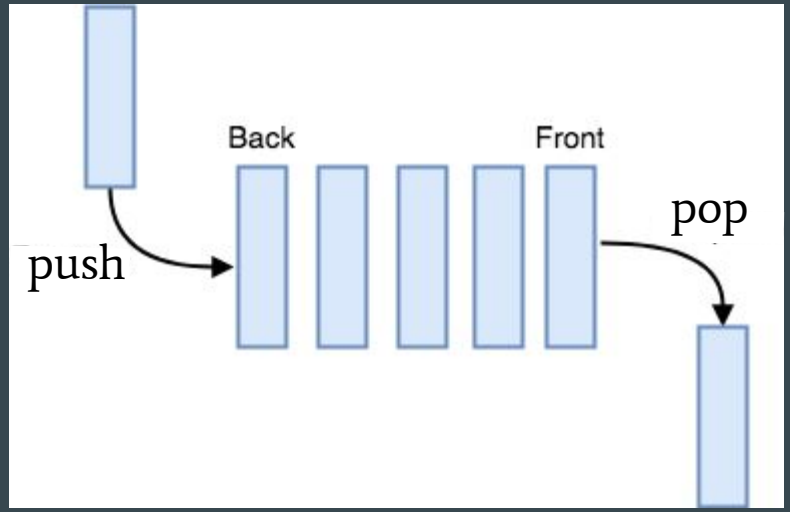
Queue - Inserindo elementos

```
queue<string> fila;
fila.push("Clara");
fila.push("Pedro");
fila.push("Ana");
fila.push("João");

cout << fila.front(); // Clara
cout << fila.back();  // João
```



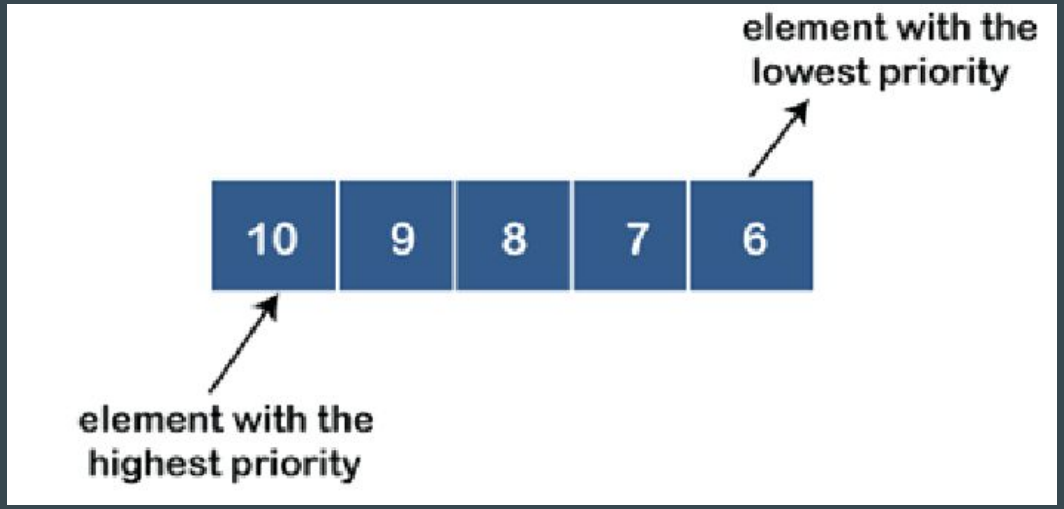
Queue - Removendo elementos

```
queue<string> fila;
fila.push("Clara");
fila.push("Pedro");
fila.push("Ana");
fila.push("João");

while(!fila.empty()) {
    fila.pop();
}
```



Priority Queue

- Uma priority queue (fila de prioridade) armazena dados em uma ordem específica
- O elemento de **maior prioridade** sempre se mantém no topo
- Em C++, por padrão, o **maior** elemento é mantido no topo

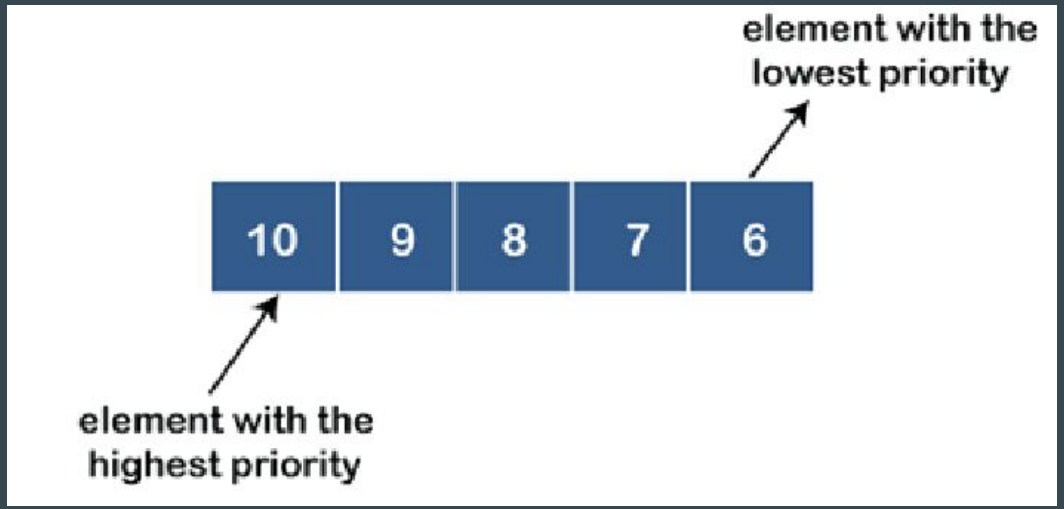


Priority Queue - Inicialização

- Obs: Por padrão, o **maior** elemento é mantido no topo

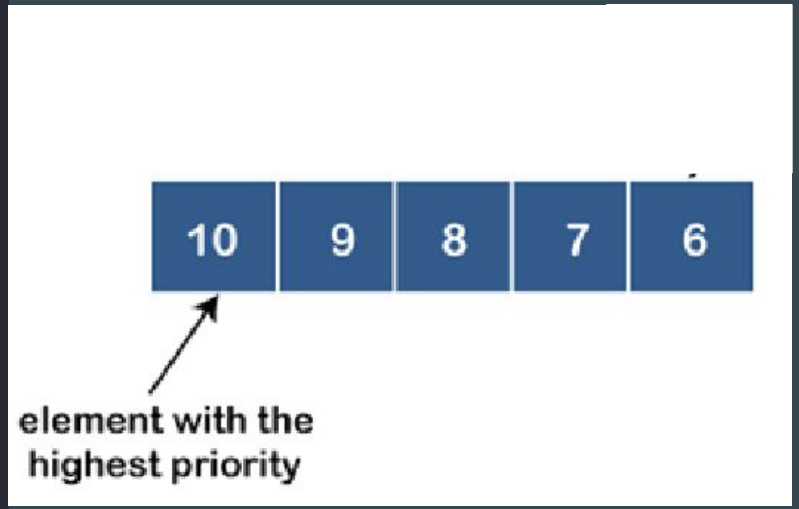
Priority Queue - Principais Operações

- push(): adiciona um elemento na fila de prioridade (O(log n))
- pop(): remove o elemento de maior prioridade da fila(O(log n))
- top(): retorna o elemento de maior prioridade da fila (O(1))
- size()
- empty()



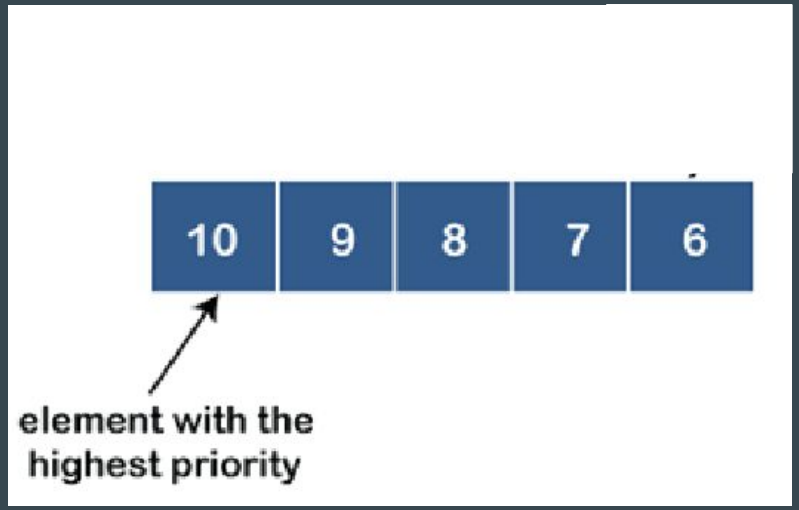
Priority Queue - Inserindo elementos

```cpp
priority_queue<int> pq;
pq.push(7);
pq.push(9);
pq.push(8);
pq.push(10);
pq.push(6);
cout << pq.top(); // <-- 10
```



Priority Queue - Removendo elementos

```cpp
priority_queue<int> pq;
pq.push(7);
pq.push(9);
pq.push(8);
pq.push(10);
pq.push(6);
// Vai imprimir 10 9 8 7 6
while(!pq.empty()) {
    cout << pq.top() << " ";
    pq.pop();
}
```



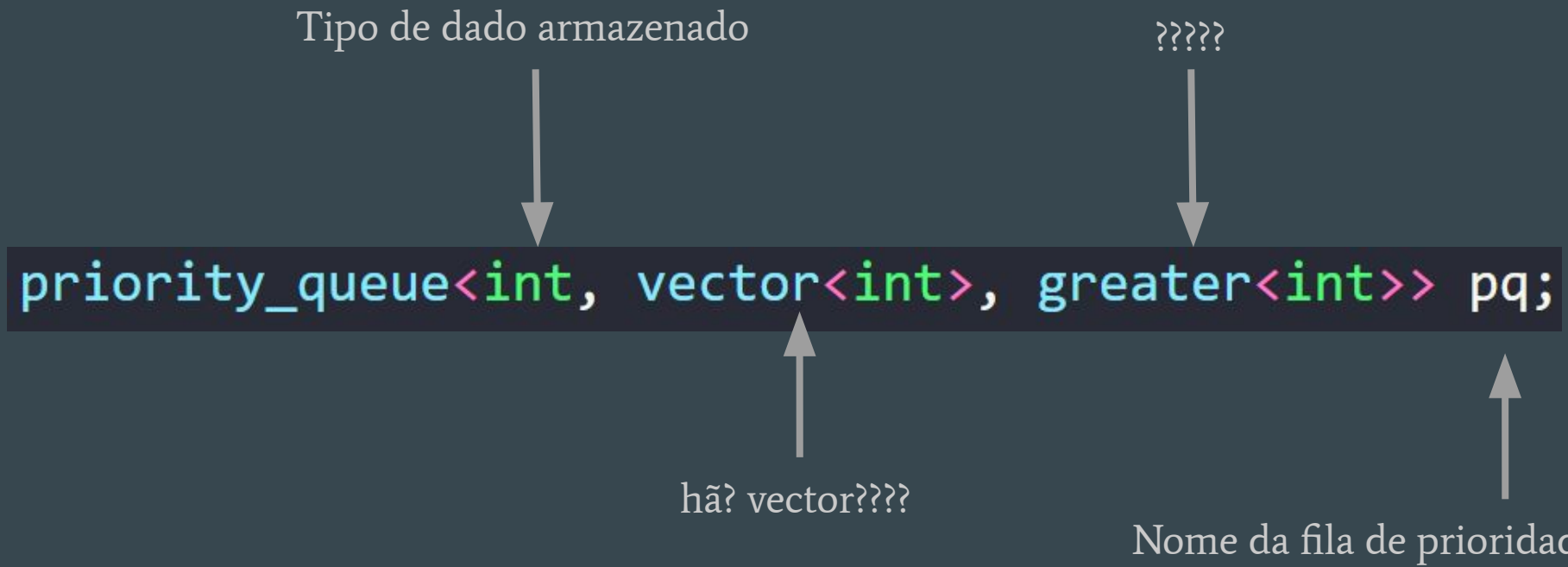
Priority Queue

- Tem como mudar a regra de prioridade?

Priority Queue

- Tem como mudar a regra de prioridade?
  - Tem.....

Priority Queue - Inicialização 2



Priority Queue - Inicialização 2

- Assim, temos uma pq de **inteiros** que mantém o **menor** elemento no topo.

Set e Map

Set

- Gerencia uma coleção de elementos (chaves) ordenadas
- Permite busca por qualquer Elemento em O(log n)
- Inserções e Remoções em O(log n)

Set - Inicialização

Existem duas formas principais de declarar um set:

```
// Declaração de um set vazio
set<int> s;
// Declaração com as chaves especificadas
set<int> s = {5,7,1,3,2};
```

Set - Inserção e Remoção

- Inserção feita com insert()
- Remoção com erase()
- Ambos O(log n)

```
set<int> s = {1, 2, 3};
s.insert(-1); // Insere a chave -1
s.erase(2); // Remove a chave 2
```

Set - Busca por Elementos

Podemos procurar por uma chave usando dois métodos: o find() e o count()

```cpp
set<int> s = {1, 2, 3};
// Busca usando count()
if (s.count(2) > 0) cout << "Sim" << endl; // Sim
else cout << "Não" << endl;
// Busca usando find()
if (s.find(2) != s.end()) cout << "Sim" << endl; // Sim
else cout << "Não" << endl;
```

Set - Métodos lower_bound e upper_bound

```cpp
set<int> s = {-1, 1, 3};

cout << *s.lower_bound(1) << endl; // 1
cout << *s.upper_bound(1) << endl; // 3
```

Set - Métodos lower_bound e upper_bound

Por quê não usar as funções lower_bound e upper_bound normais?

```cpp
set<int> s = {-1, 1, 3};

cout << *s.lower_bound(1) << endl; // 1
cout << *s.upper_bound(1) << endl; // 3
```

Set - Métodos lower_bound e upper_bound

Por quê não usar as funções lower_bound e upper_bound normais?

```
set<int> s = {-1, 1, 3};

cout << *s.lower_bound(1) << endl; // 1
cout << *s.upper_bound(1) << endl; // 3
```

Devido à forma como elas funcionam, sua complexidade no set é O(n) e não O(log n)

Set - unordered_set

- Inserção, Remoção, Busca e Acesso em O(1)
- Não tem lower_bound nem upper_bound

```cpp
unordered_set<int> us = {1, 2, 3};

//Busca por elementos - O(1)
if (us.count(2) > 0) cout << "Sim" << endl; // Sim
else cout << "Não" << endl;

//Inserção e Remoção de Elementos - O(1)
us.insert(4);
us.erase(2);
```

Set - Multiset

- Aceita Chaves repetidas
- Inserção e Remoção um pouco diferentes

```cpp
multiset<int> ms = {1, 1, 2, 2, 3};

// Remoção de Chave
ms.erase(2);

// Todas as ocorrências da chave 2 foram removidas
cout << ms.count(2) << endl; // 0

// Remove somente um elemento com a chave especificada
ms.erase(ms.find(1));

// Ainda existe um '1'
cout << ms.count(1) << endl; // 1
```

Map

- Gerencia uma coleção de Pares (Chave-Valor) ordenadas pelo primeiro elemento do par. Também é chamado de Dicionário.

- Permite busca por qualquer Elemento com a chave correspondente em O(log n)

- Inserções e Remoções em O(log n)

Map - Inicialização

Existem duas formas principais de declarar um map:

```
//Declaração Vazia
map<string, int> m;
//Declaração com os elementos especificados
map<string,int> m = {{"Bixiga",100},{"Maratona",200}};
```

Map - Inserção e Remoção

- No map, além do insert(), podemos inserir elementos usando o operador []
- No erase() só especificamos a chave do elemento

```
//Inserção usando insert()
m.insert({"Maratona",200});
//Inserção usando o operador []
m["Bixiga"] = 100;
// Removendo o elemento com a chave "Maratona"
m.erase("Maratona");
```

Map - Busca por Elementos e Acesso

- A busca ainda pode ser feita usando find() ou count()
- Caso o Elemento já esteja presente, podemos consultar seu valor usando o operador []

```
map<string,int> m = {{"Bixiga",100},{"Maratona",200}};
//Busca usando count()
if (m.count("Bixiga") > 0) cout << m["Bixiga"] << endl; // 100
else cout << "Não tem Bixiga" << endl;
//Busca usando find()
if (m.find("Bixiga") != m.end()) cout << m["Bixiga"] << endl; // 100
else cout << "Não tem Bixiga" << endl;
```

Map - Métodos lower_bound e upper_bound

Situação similar ao set

```cpp
map<string, int> m = {{"Bixiga",100},{"Maratona",200}};

auto [chave_lb, valor_lb] = *m.upper_bound("Bixiga");
auto [chave_ub, valor_ub] = *m.upper_bound("Bixiga");

cout << chave_lb << " " << valor_lb << endl; // Bixiga 100
cout << chave_ub << " " << valor_ub << endl; // Maratona 200
```

Map - unordered_map

- Inserção, Remoção, Busca e Acesso em O(1)
- Não tem lower_bound nem upper_bound

```cpp
unordered_map<string, int> um;
um["Bixiga"] = 200; // Consulta em O(1)
cout << um["Bixiga"] << endl; // Bixiga 200
```

Map - Multimap

- Aceita Chaves repetidas
- Não tem operador []
- Inserção e Remoção um pouco diferentes

```
multimap<string, int> mm;

// Insere um elemento com a chave "Bixiga" e o valor 100
mm.insert({"Bixiga", 100});
// Insere outro elemento também com a chave "Bixiga" e o valor 200
mm.insert({"Bixiga", 200});
```

Referências

Onde eu olho tudo isso?

- https://cplusplus.com/reference/
- https://en.cppreference.com/w/