

# Lista Linear com Alocação Sequencial

Estrutura de Dados — QXD0010



UNIVERSIDADE  
FEDERAL DO CEARÁ  
CAMPUS QUIXADÁ

Prof. Atílio Gomes Luiz  
gomes.atilio@ufc.br

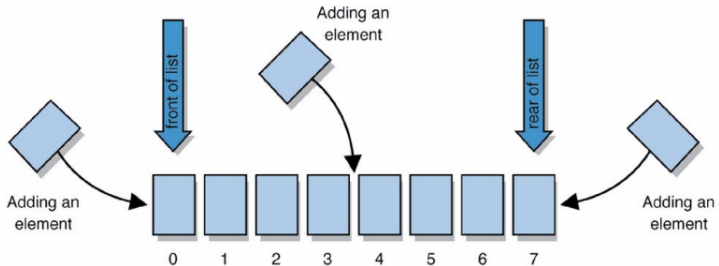
Universidade Federal do Ceará

2º semestre/2022



# Lista Linear — Definição

- Uma **lista linear**  $L$  é um conjunto de  $n$  **nós**  $L_0, L_1, \dots, L_{n-1}$ , com  $n \geq 0$ , tais que suas propriedades estruturais decorrem, unicamente, da posição relativa dos nós dentro da sequência linear:
  - (a) Se  $n > 0$ ,  $L_0$  é o primeiro nó;
  - (b) Para  $0 < k \leq n - 1$ , o nó  $L_k$  é precedido por  $L_{k-1}$ .



# Lista linear – Operações

Algumas operações que podemos querer realizar sobre listas lineares:

- Inserir um elemento novo antes ou depois de  $L_k$ .

# Lista linear – Operações

Algumas operações que podemos querer realizar sobre listas lineares:

- Inserir um elemento novo antes ou depois de  $L_k$ .
- Remover  $L_k$ .

# Lista linear – Operações

Algumas operações que podemos querer realizar sobre listas lineares:

- Inserir um elemento novo antes ou depois de  $L_k$ .
- Remover  $L_k$ .
- Acessar  $L_k$

# Lista linear – Operações

Algumas operações que podemos querer realizar sobre listas lineares:

- Inserir um elemento novo antes ou depois de  $L_k$ .
- Remover  $L_k$ .
- Acessar  $L_k$
- Colocar todos os elementos da lista em ordem.

# Lista linear – Operações

Algumas operações que podemos querer realizar sobre listas lineares:

- Inserir um elemento novo antes ou depois de  $L_k$ .
- Remover  $L_k$ .
- Acessar  $L_k$
- Colocar todos os elementos da lista em ordem.
  - Estudaremos algoritmos de ordenação no final do curso.

# Lista linear – Operações

Algumas operações que podemos querer realizar sobre listas lineares:

- Inserir um elemento novo antes ou depois de  $L_k$ .
- Remover  $L_k$ .
- Acessar  $L_k$
- Colocar todos os elementos da lista em ordem.
  - Estudaremos algoritmos de ordenação no final do curso.
- Combinar duas ou mais listas lineares em uma só.



# Lista linear – Operações

Algumas operações que podemos querer realizar sobre listas lineares:

- Inserir um elemento novo antes ou depois de  $L_k$ .
- Remover  $L_k$ .
- Acessar  $L_k$
- Colocar todos os elementos da lista em ordem.
  - Estudaremos algoritmos de ordenação no final do curso.
- Combinar duas ou mais listas lineares em uma só.
- Quebrar uma lista linear em duas ou mais.

# Lista linear – Operações

Algumas operações que podemos querer realizar sobre listas lineares:

- Inserir um elemento novo antes ou depois de  $L_k$ .
- Remover  $L_k$ .
- Acessar  $L_k$
- Colocar todos os elementos da lista em ordem.
  - Estudaremos algoritmos de ordenação no final do curso.
- Combinar duas ou mais listas lineares em uma só.
- Quebrar uma lista linear em duas ou mais.
- Copiar uma lista linear em um outro espaço.

# Implementação de uma lista linear

- O modo de implementar listas lineares depende da classe de operações mais frequentes. Não existe, em geral, uma única implementação para a qual todas as operações são eficientes.

# Implementação de uma lista linear

- O modo de implementar listas lineares depende da classe de operações mais frequentes. Não existe, em geral, uma única implementação para a qual todas as operações são eficientes.
- Por exemplo, não existe uma implementação para atender às seguintes duas operações de maneira eficiente:
  - (1) ter acesso rápido ao elemento  $L_k$ , para  $k$  qualquer.
  - (2) inserir ou remover elementos em qualquer posição da lista linear.

# Implementação de uma lista linear

- O modo de implementar listas lineares depende da classe de operações mais frequentes. Não existe, em geral, uma única implementação para a qual todas as operações são eficientes.
- Por exemplo, não existe uma implementação para atender às seguintes duas operações de maneira eficiente:
  - (1) ter acesso rápido ao elemento  $L_k$ , para  $k$  qualquer.
  - (2) inserir ou remover elementos em qualquer posição da lista linear.

A operação (1) fica eficiente se a lista é implementada em um vetor (array) em alocação sequencial na memória.

Para a operação (2) é mais adequada a alocação encadeada, com o uso de ponteiros.

# Tipos de alocação

O tipo de armazenamento de uma lista linear pode ser classificado de acordo com a posição relativa na memória de dois nós consecutivos na lista.

- **Alocação sequencial:** dois nós consecutivos na lista estão em **posições contíguas** de memória.
- **Alocação encadeada:** dois nós consecutivos na lista podem estar em **posições não contíguas** da memória.

# Listas Sequenciais



# Listas sequenciais (Vetores)

- Nós em posições contíguas da memória.

$L$	$L+c$	$L+2c$	$L+3c$	$L+4c$	$L+5c$	$L+6c$
nó 1	nó 2	nó 3	nó 4	nó 5	nó 6	nó 7



# Listas sequenciais (Vetores)

- Nós em posições contíguas da memória.

L	L+c	L+2c	L+3c	L+4c	L+5c	L+6c
nó 1	nó 2	nó 3	nó 4	nó 5	nó 6	nó 7

- Neste caso, o endereço real do  $(j + 1)$ -ésimo nó da lista se encontra  $c$  unidades adiante daquele correspondente ao  $j$ -ésimo. A constante  $c$  é o número de bytes que cada nó ocupa.

# Listas sequenciais (Vetores)

- Nós em posições contíguas da memória.

L	L+c	L+2c	L+3c	L+4c	L+5c	L+6c
nó 1	nó 2	nó 3	nó 4	nó 5	nó 6	nó 7

- Neste caso, o endereço real do  $(j + 1)$ -ésimo nó da lista se encontra  $c$  unidades adiante daquele correspondente ao  $j$ -ésimo. A constante  $c$  é o número de bytes que cada nó ocupa.
- A correspondência entre o índice da lista e o endereço real é feita automaticamente pela linguagem de programação quando da compilação do programa.

## TAD Lista Sequencial (Vector)

- O TAD Lista Sequencial tem os seguintes **atributos**:
  - a lista de elementos (implementada como um vetor).
  - a quantidade de elementos atualmente na lista.
  - a capacidade total da lista (essa capacidade pode aumentar)

# TAD Lista Sequencial (Vector)

- O TAD Lista Sequencial tem os seguintes **atributos**:
  - a lista de elementos (implementada como um vetor).
  - a quantidade de elementos atualmente na lista.
  - a capacidade total da lista (essa capacidade pode aumentar)
- Exemplos de **operações** que esse TAD pode ter:
  - Criar lista vazia.
  - Criar lista a partir de uma lista prévia.
  - Liberar lista.
  - Retornar número de elementos da lista.
  - Retornar a capacidade atual da lista.
  - Retornar se lista está vazia.
  - Retornar uma referência para o elemento na posição  $k$ .
  - Inserir elemento em uma dada posição da lista.
  - Remover elemento em certa posição.
  - Buscar primeira ocorrência de um elemento e retorna índice.
  - Limpar a lista, deixando-a vazia.
  - Retornar uma sublista da lista.

## Interface do nosso Vector



# Vector.h

```
1 #ifndef VECTOR_H
2 #define VECTOR_H
3
4 class Vector {
5 private:
6     int* m_list {nullptr}; // ponteiro para a lista
7     int m_size {0};        // numero de elementos na lista
8     int m_capacity {0};    // capacidade total da lista
9
10 public:
11     // Construtor default: aloca uma lista com
12     // capacidade inicial igual a 16 e size = 0
13     Vector(); // 0(1)
14
15     // Copy constructor: cria uma nova lista com os
16     // mesmos elementos da lista passada como argumento
17     Vector(const Vector& vector); // 0(n)
18
19     // Destrutor: libera memoria alocada
20     ~Vector(); // 0(1)
```

## Vector.h (cont.)

```
21 // Retorna a capacidade atual da lista
22 int capacity() const; // 0(1)
23
24 // Retorna o numero de elementos na lista
25 int size() const; // 0(1)
26
27 // Retorna true se e somente se a lista estiver vazia
28 bool empty() const; // 0(1)
29
30 // Retorna uma referencia para o elemento na posicao k.
31 // A funcao verifica automaticamente se n esta dentro dos
32 // limites de elementos validos no vetor, lancando uma
33 // excecao 'out_of_range' se nao estiver.
34 int& at(int k); // 0(1)
35 const int& at(int k) const; // 0(1)
36
37 // Retorna uma referencia para o elemento na posicao k.
38 // Essas funcoes nao verificam se o indice eh valido.
39 int& operator[](int index); // 0(1)
40 const int& operator[](int index) const; // 0(1)
```

## Vector.h (cont.)

```
41 // Solicita que a capacidade do vetor seja >= n.
42 // Se n for maior que a capacidade atual do vetor, a
43 // funcao faz com que a lista aumente sua capacidade
44 // realocando os elementos para o novo vetor. Em todos
45 // os outros casos, a chamada da funcao nao causa uma
46 // realocacao e a capacidade do vetor nao eh afetada.
47 void reserve(int n); // O(n)
48
49 // Recebe um inteiro como argumento e o adiciona
50 // logo apos o ultimo elemento da lista.
51 void push_back(const int& value); // tempo medio O(1)
52
53 // Remove o ultimo elemento da lista se a lista nao
54 // estiver vazia. Caso contrario, faz nada
55 void pop_back(); // O(1)
56
57 };
58
59 #endif
```



# main.cpp

```
1 #include <iostream>
2 #include "Vector.h"
3 using namespace std;
4
5 void print(const Vector& list) {
6     for(int i = 0; i < list.size(); i++)
7         cout << list.at(i) << " ";
8     cout << endl;
9 }
10
11 int main() {
12     Vector list1, list2;
13     for(int i = 0; i < 150; ++i) {
14         list1.push_back(i);
15         list2.push_back(i * 2);
16     }
17     print(list1);
18     print(list2);
19 }
```

# Exercício



# Exercício

**Exercício:** Implementar as funções-membro da classe **Vector**.

Você pode implementar as funções-membro dentro da própria classe ou pode implementá-las em um arquivo-fonte separado chamado `Vector.cpp`.

# Exercício

Implemente as seguintes operações adicionais na Lista Sequencial:

- `void replaceAt(int value, int k)`: Troca o elemento no índice  $k$  pelo elemento `value` (somente se  $0 \leq k \leq size\_vec - 1$ )
- `void removeAt(int k)`: Remove o elemento com índice  $k$  na lista. Deve-se ter  $0 \leq k \leq size\_vec - 1$ ; caso contrário, a remoção não é realizada.
- `bool insert(int value, int k)`: Adiciona o elemento `value` no índice  $k$  (somente se  $0 \leq k \leq size\_vec$ ). Antes de fazer a inserção, todos os elementos do índice  $k$  em diante são deslocados uma posição para a direita.
- `void removeAll(int value)`: Remove todas as ocorrências do elemento `value` na lista.

# Introdução à STL



# Standard Template Library

- A STL é uma parte do padrão C++ aprovada em 1997/1998 e estende o núcleo do C++ fornecendo componentes gerais.
  - A STL fornece o tipo de dado `std::string`, diferentes estruturas para armazenamento de dados, classes para entrada/saída e algoritmos utilizados frequentemente por programadores.

Parte lógica	Descrição
Containers	Gerenciam coleções de objetos
Iteradores	Percorrem elementos das coleções de objetos
Algoritmos	Processam elementos da coleção

# Containers

- **Containers** são como a STL chama suas estruturas de dados. Dentro dos containers podemos guardar vários dados de um mesmo tipo.
  - Containers armazenam qualquer tipo de dado válido.


# Containers

- **Containers** são como a STL chama suas estruturas de dados. Dentro dos containers podemos guardar vários dados de um mesmo tipo.
  - Containers armazenam qualquer tipo de dado válido.
- Cada tipo de container possui uma estratégia diferente para organização interna de seus dados.
  - Por isso, cada container possui vantagens e desvantagens em diferentes situações.



# Containers

- **Containers** são como a STL chama suas estruturas de dados. Dentro dos containers podemos guardar vários dados de um mesmo tipo.
  - Containers armazenam qualquer tipo de dado válido.
- Cada tipo de container possui uma estratégia diferente para organização interna de seus dados.
  - Por isso, cada container possui vantagens e desvantagens em diferentes situações.
- **Containers sequenciais:** são aqueles utilizados para representar sequências de elementos. Em uma sequência de elementos, **cada elemento deve ter uma posição específica**.  
**Exemplos:** `vector`, `list`, `deque`, `forward_list`



`std::vector`



# std::vector

- `std::vector` é um **container sequencial** implementado como um array redimensionável.
  - O vector permite **acesso randômico** aos seus elementos individuais e pode aumentar dinamicamente. O gerenciamento de memória é feito automaticamente pelo container.
- Definido na biblioteca: `<vector>`
  - `#include <vector>`

# Construindo um std::vector

```
1 #include <iostream> // vector01.cpp
2 #include <vector>
3 using namespace std;
4
5 int main() {
6     // cria um vector vazio
7     vector<int> v1;
8     // cria vector com 7 elementos zeros
9     vector<int> v2 (7);
10    // 4 inteiros com valor 100
11    vector<int> v3 (4, 100);
12    // cria um vector que é uma copia do v3
13    vector<int> v4 (v3);
14    // cria um vector com uma lista inicializadora
15    vector<int> v5 {1,2,3,4,5,6};
16
17    for(int elemento : v5)
18        cout << elemento << " ";
19    cout << endl;
20 }
```

## Inserindo e removendo no final

- `void push_back(const value_type& val)`  
Adiciona um novo elemento ao final do vector, depois do seu último elemento atual.

```
vector<int> myVector; // um vetor vazio  
myVector.push_back(30);  
myVector.push_back(25);  
myVector.push_back(80);
```

## Inserindo e removendo no final

- `void push_back(const value_type& val)`

Adiciona um novo elemento ao final do vector, depois do seu último elemento atual.

```
vector<int> myVector; // um vetor vazio  
myVector.push_back(30);  
myVector.push_back(25);  
myVector.push_back(80);
```

- `void pop_back()`

Remove o último elemento no vector, decrementando seu tamanho em 1.

```
myVector.pop_back();  
myVector.pop_back();
```

## size e resize

- `size_type size()`

Retorna o número de elementos no vector.

Exemplo: `myVector.size()`

- `void resize(size_type n)`

Modifica o vector de modo que ele contenha `n` elementos.

- Se `n` for menor que o `size()` atual, o conteúdo é reduzido aos primeiros `n` elementos, removendo os demais.
- Se `n` for maior que o `size()` atual, o conteúdo é expandido inserindo quantos elementos forem necessários até atingir o tamanho `n`. O elemento a ser inserido pode ser especificado na função (`resize(n, val)`), caso contrário ele é um valor *default*.

## resize — exemplo

```
1 #include <iostream> // vector02.cpp
2 #include <vector>
3 using namespace std;
4
5 void print(vector<int>& vec) {
6     for(int e : vec) cout << e << " ";
7     cout << endl;
8 }
9
10 int main () {
11     vector<int> myVector;
12
13     for(int i = 1; i <= 8; i++)
14         myVector.push_back(i);
15
16     myVector.resize(10);
17     print(myVector); // 1 2 3 4 5 6 7 8 0 0
18     myVector.resize(5);
19     print(myVector); // 1 2 3 4 5
20     myVector.resize(8,100); // 1 2 3 4 5 100 100 100
21     print(myVector);
22 }
```



# Acesso randômico aos elementos

- O `operator[]` permite acesso randômico do elemento na posição  $i$  do vector ( $0 \leq i \leq \text{size()}-1$ ).  
Ele devolve uma referência para o elemento requerido.
  - Se o índice  $i$  requisitado estiver fora do intervalo válido, o comportamento será indefinido. **Nunca faça isso.**
- `value_type& at(size_type i)`  
Retorna uma referência para o elemento na posição  $i$  do vector. Esta função checa se  $i$  está dentro do intervalo  $0..\text{size()}-1$  e, caso não esteja, lança uma exceção.

# Acesso randômico aos elementos

```
1 #include <iostream> // vector03.cpp
2 #include <vector>
3 using namespace std;
4
5 int main () {
6     vector<int> myVector(7);
7
8     for(int i = 0; i < 7; ++i)
9         myVector[i] = i+1;
10
11     for(size_t i = 0; i < myVector.size(); ++i)
12         cout << myVector[i] << " ";
13     cout << endl;
14 }
```

# Iteradores



# Iteradores

- A opção de se acessar elementos de um container através do `operator[]` é restrita apenas a containers de acesso aleatório, como o `vector`.
- Os containers `list` e `forward_list`, por exemplo, não possuem o `operator[]`. Como são implementados com listas encadeadas, seus dados não podem ser acessados randomicamente.
- Para conseguirmos acessar elementos de todos os tipos de container, precisamos de `iteradores`.

# Iteradores

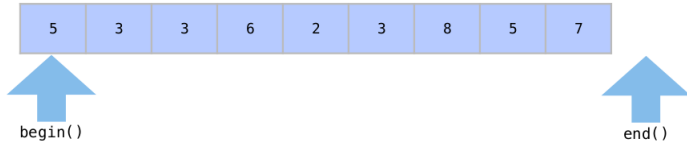
- A opção de se acessar elementos de um container através do `operator[]` é restrita apenas a containers de acesso aleatório, como o `vector`.
- Os containers `list` e `forward_list`, por exemplo, não possuem o `operator[]`. Como são implementados com listas encadeadas, seus dados não podem ser acessados randomicamente.
- Para conseguirmos acessar elementos de todos os tipos de container, precisamos de `iteradores`.

**Definição:** Os `iteradores` são objetos que caminham (iteram) sobre elementos de containers.

- Eles funcionam como um ponteiro especial para elementos de containers: enquanto um ponteiro representa uma posição na memória, um iterador representa uma posição em um container.

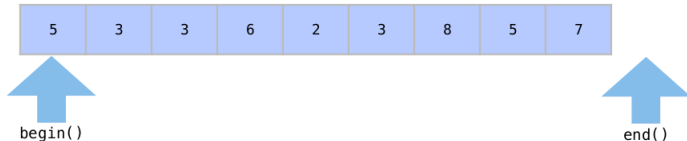
# Iteradores

- Os iteradores podem ser gerados através de funções de um container.
  - Suponha um container chamado `i`
  - O comando `i.begin()` retorna um iterador para o primeiro elemento deste container `c`.
  - O comando `i.end()` retorna um iterador para uma posição após o último elemento do container `i`.



# Iteradores

- Os iteradores podem ser gerados através de funções de um container.
  - Suponha um container chamado `i`
  - O comando `i.begin()` retorna um iterador para o primeiro elemento deste container `c`.
  - O comando `i.end()` retorna um iterador para uma posição após o último elemento do container `i`.



- Por exemplo, o container `vector` possui funções `begin()` e `end()` que retornam iteradores, para o primeiro elemento e para uma posição após o último elemento, respectivamente.

# Operações comuns com iteradores

- Supondo um iterador chamado `i`, a tabela abaixo apresenta algumas funções que são comuns a iteradores.

Função	Retorna
<code>*i</code>	Retorna o elemento na posição do iterador
<code>++i</code>	Avança o iterador para o próximo elemento
<code>==</code>	Confere se dois iteradores apontam para mesma posição
<code>!=</code>	Confere se dois iteradores apontam para posições diferentes



## Exemplo de uso de iteradores – vector

```
1 #include <iostream> // vector04.cpp
2 #include <vector>
3 using namespace std;
4
5 int main () {
6     vector<int> vec(7); // vector com size = 7
7
8     vector<int>::iterator it; // definição de um iterador
9
10    for(it = vec.begin(); it != vec.end(); ++it) {
11        *it = 33;
12    }
13
14    for(it = vec.begin(); it != vec.end(); ++it)
15        cout << *it << " ";
16
17    cout << endl;
18 }
```

# Exemplo de uso de iteradores – vector

## Uso da palavra-chave auto

```
1 #include <iostream> // vector05.cpp
2 #include <vector>
3 using namespace std;
4
5 int main () {
6     vector<int> vec(7);
7
8     for(auto it = vec.begin(); it != vec.end(); ++it) {
9         *it = 33;
10    }
11
12    for(auto it = vec.begin(); it != vec.end(); ++it)
13        cout << *it << " ";
14
15    cout << endl;
16 }
```

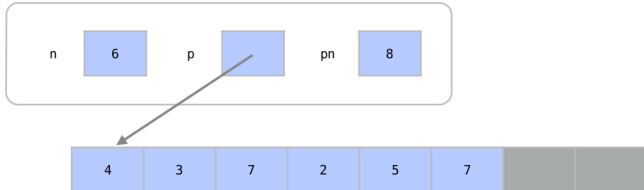
# Funções de vector que usam iteradores

## insert

- Considere um **vector** gerado pelo código abaixo.

```
1 vector<int> c;  
2 c.push_back(4);  
3 c.push_back(3);  
4 c.push_back(7);  
5 c.push_back(2);  
6 c.push_back(5);  
7 c.push_back(7);
```

vector<int> c;



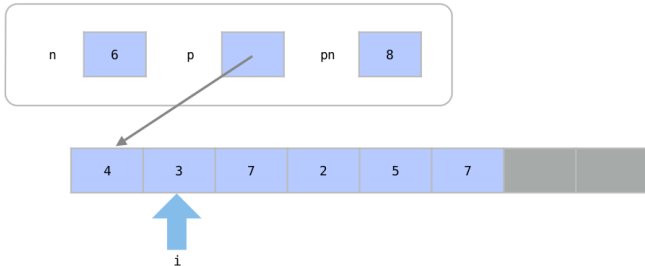
# Funções de vector que usam iteradores

## insert

- Considere também que geramos um iterador *i* apontando para o segundo elemento da sequência.

```
1 vector<int>::iterator i;  
2 i = c.begin();  
3 ++i;
```

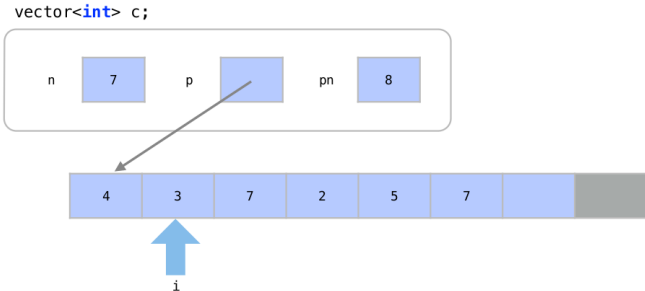
vector<int> c;



# Funções de vector que usam iteradores

## insert

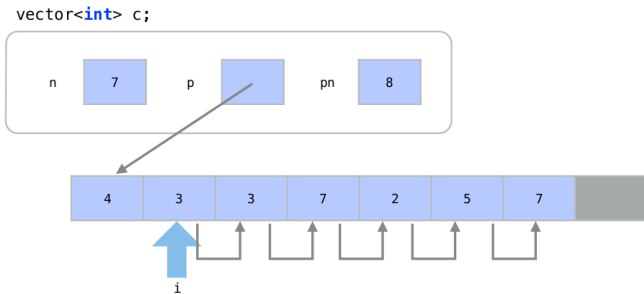
- Queremos inserir 11 na segunda posição do vector. Uma sequência de passos deve ser executada.
1. Primeiro, a variável  $n$  é incrementada. Em alguns casos, pode ser que um novo array precise ser alocado, levando a um custo  $O(n)$ .



# Funções de vector que usam iteradores

## insert

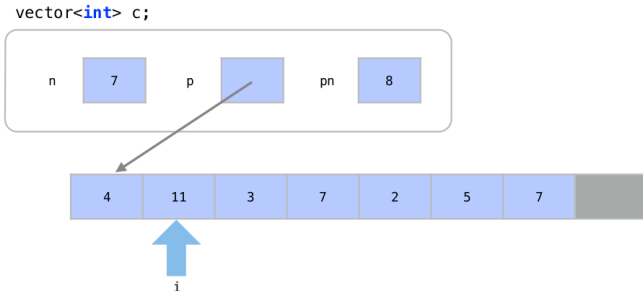
2. Todos os elementos entre  $i$  e o penúltimo elemento precisam ser deslocados para a próxima posição do arranjo. Esta operação tem um custo  $O(n - i)$ . Se  $i$  for o último elemento, temos um melhor caso  $O(1)$ . Se  $i$  for o primeiro elemento, temos um pior caso  $O(n)$ .



# Funções de vector que usam iteradores

## insert

3. O elemento 11 pode ser inserido na posição  $i$ , com custo  $O(1)$ .



Devido a todas as movimentações, esta operação completa de inserção no meio da sequência tem um custo  $O(n)$ .

# Funções de vector que usam iteradores

## insert

- `iterator insert(iterator it, const value_type& val)`  
Insere o valor `val` na posição indicada pelo iterador `it`.  
Esta função retorna um iterador apontando para o objeto recém inserido.



# Funções de vector que usam iteradores

## insert

- `iterator insert(iterator it, const value_type& val)`  
Insere o valor `val` na posição indicada pelo iterador `it`.  
Esta função retorna um iterador apontando para o objeto recém inserido.
- `iterator insert(iterator it, int n, const value_type& val)`  
Insere o valor `val` um total de `n` vezes na posição indicada pelo iterador `it`. Esta função retorna um iterador apontando para o objeto recém inserido.

# Funções de vector que usam iteradores

## insert

- `iterator insert(iterator it, const value_type& val)`  
Insere o valor `val` na posição indicada pelo iterador `it`.  
Esta função retorna um iterador apontando para o objeto recém inserido.
- `iterator insert(iterator it, int n, const value_type& val)`  
Insere o valor `val` um total de `n` vezes na posição indicada pelo iterador `it`. Esta função retorna um iterador apontando para o objeto recém inserido.
- `iterator insert(iterator it, InputIterator first, InputIterator last)`  
Essa operação insere todos os elementos entre o iterador `first` (inclusive) e o iterador `last` (exclusive) no vector, a partir da posição indicada pelo iterador `it`. Os iteradores `first` e `last` pertencem a um outro container.

# Funções de vector que usam iteradores

## insert

```
1 #include <iostream>
2 #include <vector>
3 using namespace std;
4
5 int main() {
6     vector<int> vec(3, 10); //vec: 10,10,10
7     vector<int>::iterator it;
8
9     it = vec.begin();
10    it = vec.insert(it, 20); //vec: 20,10,10,10
11
12    vec.insert(it, 2, 30); //vec: 30,30,20,10,10,10
13
14    // "it" no longer valid, get a new one:
15    it = vec.begin();
16
17    vector<int> z(2, 40); //z: 40,40
18    vec.insert(it+2, z.begin(), z.end());
19    //vec: 30,30,40,40,20,10,10,10
20 }
```

# Funções de vector que usam iteradores

## erase

- `iterator erase (iterator position)`

Remove do `vector` um único elemento, que é o elemento apontado pelo iterador `position`.

- Essa função decrementa o tamanho do vector em 1 unidade.
- Essa função devolve um iterador apontando para a nova localização do elemento que seguia o último elemento apagado pela chamada de função. Este elemento é o `end()` se a operação apagou o último elemento na sequência.

# Funções de vector que usam iteradores

## erase

```
1 #include <iostream> // vector07.cpp
2 #include <vector>
3 using namespace std;
4
5 int main() {
6     vector<int> vec;
7
8     // set some values (from 1 to 10)
9     for(int i = 1; i <= 10; i++)
10         vec.push_back(i);
11
12     // erase the 6th element
13     auto it = vec.erase(vec.begin() + 5);
14     vec.erase(it); // erase the number 7
15
16     cout << "vec contains: ";
17     for(size_t i = 0; i < vec.size(); ++i)
18         cout << " " << vec[i];
19     cout << endl;
20 }
```

# Mais informações

Sobre vector e outros containers:

- <https://www.cplusplus.com/reference/vector/vector/>
- <https://www.geeksforgeeks.org/vector-in-cpp-stl/>

FIM

