

Listas Simplesmente Encadeadas

Estrutura de Dados — QXD0010



UNIVERSIDADE
FEDERAL DO CEARÁ
CAMPUS QUIXADÁ

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

Universidade Federal do Ceará

2º semestre/2021

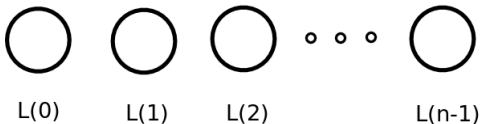


Introdução



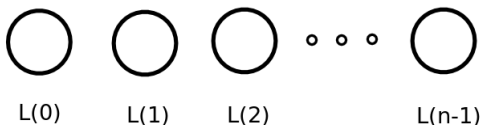
Estrutura de dado: Lista linear

- Uma **lista linear** L é um conjunto de $n \geq 0$ **nós** (ou **células**) $L[0], L[1], \dots, L[n-1]$ tais que suas propriedades estruturais decorrem, unicamente, da posição relativa dos nós dentro da sequência linear:

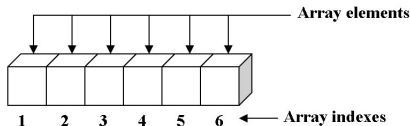


Estrutura de dado: Lista linear

- Uma **lista linear** L é um conjunto de $n \geq 0$ **nós** (ou **células**) $L[0], L[1], \dots, L[n-1]$ tais que suas propriedades estruturais decorrem, unicamente, da posição relativa dos nós dentro da sequência linear:



- Vimos que uma lista linear pode ser implementada usando alocação dinâmica de memória usando um vetor (**alocação sequencial**).



One-dimensional array with six elements

Alocação sequencial

Vantagens do uso de vetores:

- operações de acesso aos elementos são rápidas: $O(1)$
- poucos ponteiros: maior parte do espaço é utilizada para dados

Alocação sequencial

Vantagens do uso de vetores:

- operações de acesso aos elementos são rápidas: $O(1)$
- poucos ponteiros: maior parte do espaço é utilizada para dados

Desvantagens do uso de vetores:

- estão alocados contiguamente na memória
 - pode ser que tenhamos espaço na memória
 - mas não para alocar um vetor do tamanho desejado

Alocação sequencial

Vantagens do uso de vetores:

- operações de acesso aos elementos são rápidas: $O(1)$
- poucos ponteiros: maior parte do espaço é utilizada para dados

Desvantagens do uso de vetores:

- estão alocados contiguamente na memória
 - pode ser que tenhamos espaço na memória
 - mas não para alocar um vetor do tamanho desejado
- têm um tamanho fixo
 - ou alocamos um vetor pequeno e o espaço pode acabar
 - ou alocamos um vetor grande e desperdiçamos memória
 - **Solução:** criar lista sequencial redimensionável

Alocação sequencial

Vantagens do uso de vetores:

- operações de acesso aos elementos são rápidas: $O(1)$
- poucos ponteiros: maior parte do espaço é utilizada para dados

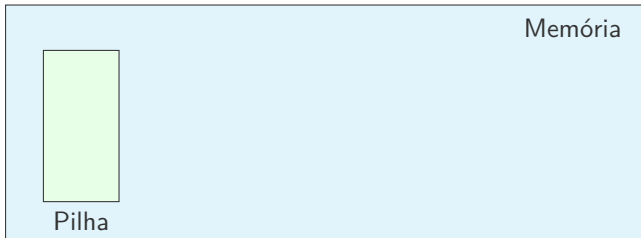
Desvantagens do uso de vetores:

- estão alocados contiguamente na memória
 - pode ser que tenhamos espaço na memória
 - mas não para alocar um vetor do tamanho desejado
- têm um tamanho fixo
 - ou alocamos um vetor pequeno e o espaço pode acabar
 - ou alocamos um vetor grande e desperdiçamos memória
 - **Solução:** criar lista sequencial redimensionável
- operações de inserção e remoção de elementos são custosas: $O(n)$

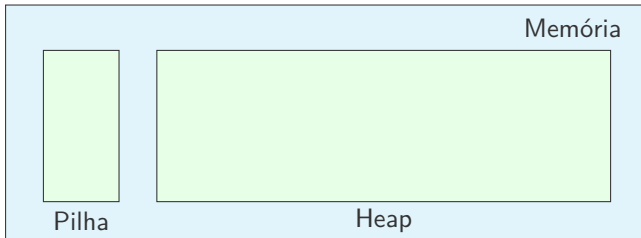
Listas Simplesmente Encadeadas



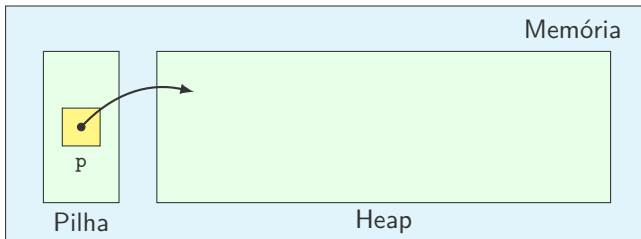
Alternativa - Lista Simplesmente Encadeada



Alternativa - Lista Simplesmente Encadeada

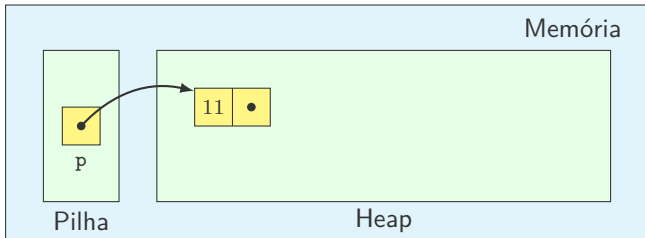


Alternativa - Lista Simplesmente Encadeada



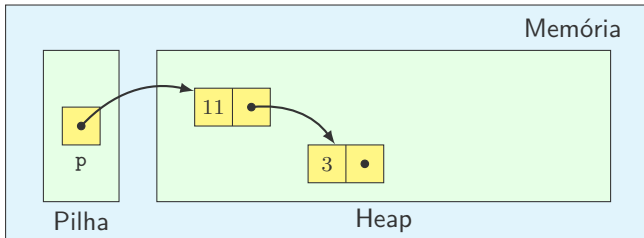
- declaramos um ponteiro para a lista no nosso programa

Alternativa - Lista Simplesmente Encadeada



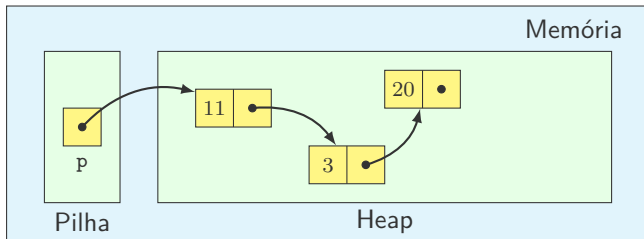
- declaramos um ponteiro para a lista no nosso programa
- alocamos memória conforme o necessário

Alternativa - Lista Simplesmente Encadeada



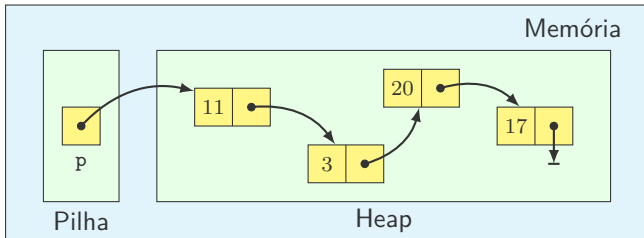
- declaramos um ponteiro para a lista no nosso programa
- alocamos memória conforme o necessário
- o primeiro **nó** aponta para o segundo

Alternativa - Lista Simplesmente Encadeada



- declaramos um ponteiro para a lista no nosso programa
- alocamos memória conforme o necessário
- o primeiro **nó** aponta para o segundo
- o segundo nó aponta para o terceiro

Alternativa - Lista Simplesmente Encadeada



- declaramos um ponteiro para a lista no nosso programa
- alocamos memória conforme o necessário
- o primeiro **nó** aponta para o segundo
- o segundo nó aponta para o terceiro
- o último nó aponta para `nullptr`

Lista Simplesmente Encadeada

- O TAD **Lista Linear** pode ser implementado usando **alocação encadeada** como uma **lista simplesmente encadeada**.

Lista Simplesmente Encadeada

- O TAD **Lista Linear** pode ser implementado usando **alocação encadeada** como uma **lista simplesmente encadeada**.
- A Lista Simplesmente Encadeada contém dois dados:
 - um ponteiro para o primeiro nó (**head**).
 - o número de elementos atualmente lista (**size**).

Lista Simplesmente Encadeada

- O TAD **Lista Linear** pode ser implementado usando **alocação encadeada** como uma **lista simplesmente encadeada**.
- A Lista Simplesmente Encadeada contém dois dados:
 - um ponteiro para o primeiro nó (**head**).
 - o número de elementos atualmente lista (**size**).
- Operações que podemos querer realizar numa lista:
 - Criar uma nova lista vazia.
 - Deixar a lista vazia.
 - Destruir a lista.
 - Adicionar um elemento em qualquer posição da lista.
 - Remover da lista um elemento em certa posição.
 - Percorrer a lista acessando os elementos.
 - Consultar o tamanho atual da lista.
 - Saber se lista está vazia.

Detalhes de Implementação



Listas Encadeadas – Detalhes de Implementação

É formada por um conjunto de objetos chamados nós.

Nó é um elemento alocado dinamicamente que contém:

- o dado armazenado
- um ponteiro para o nó seguinte na lista

Listas Encadeadas – Detalhes de Implementação

É formada por um conjunto de objetos chamados nós.

Nó é um elemento alocado dinamicamente que contém:

- o dado armazenado
 - um ponteiro para o nó seguinte na lista
-
- Um nó pode ser implementado como um `struct` ou como uma `class`.

Listas Encadeadas – Detalhes de Implementação

É formada por um conjunto de objetos chamados nós.

Nó é um elemento alocado dinamicamente que contém:

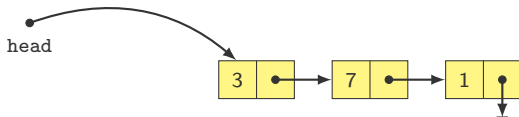
- o dado armazenado
 - um ponteiro para o nó seguinte na lista
-
- Um nó pode ser implementado como um `struct` ou como uma `class`.
 - Vou implementar como um `struct`

Arquivo Node.h

```
1 #ifndef NODE_H
2 #define NODE_H
3
4 typedef int Item;
5
6 struct Node {
7     Item data;    // data
8     Node *next;  // pointer to the next node
9
10    // Constructor: initializes node's data
11    Node(const Item& k, Node *nextnode = nullptr) {
12        data = k;
13        next = nextnode;
14    }
15 };
16
17 #endif
```

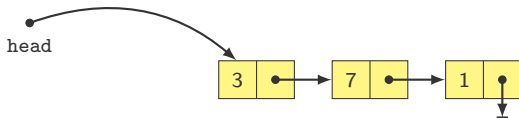

Listas Encadeadas – Detalhes da Implementação

- Conjunto de nós ligados entre si de maneira sequencial



Listas Encadeadas – Detalhes da Implementação

- Conjunto de nós ligados entre si de maneira sequencial



Observações:

- a lista encadeada é acessada a partir de um ponteiro (**head**)
- o campo **next** do último nó aponta para **nullptr**
- Assim que a lista é criada, ela está vazia e não tem nenhum elemento. Logo, o ponteiro **head** inicia apontando para **nullptr**.

Arquivo LinkedList.h

- Este arquivo contém a declaração da classe `LinkedList`, que contém a lógica da lista simplesmente encadeada discutida anteriormente.

Arquivo LinkedList.h

```
1 #include <iostream>
2 #include "Node.h"
3
4 class LinkedList {
5 private:
6     Node* m_head; // ponteiro para o primeiro elemento
7     int m_size;    // número de elementos na lista
8 public:
9     LinkedList();
10    bool empty();
11    void clear();
12    int size();
13    void push_back(Item data);
14    void pop_back();
15    void insert(int index, Item data);
16    void remove(int index);
17    ~LinkedList();
18    Item& operator[](int index);
19    friend std::ostream& operator<<(std::ostream& out,
20                                    const LinkedList& l);
21 };
22
23 #endif
```

Arquivo main.cpp

```
1 #include <iostream>
2 #include "LinkedList.h"
3 using namespace std;
4
5 int main() {
6     LinkedList list; // cria lista vazia
7
8     for(int i = 1; i <= 10; ++i) // insere 1..10
9         list.push_back(i);
10
11     cout << list << endl; // imprime lista na tela
12
13     for(int i = 0; i < list.size(); ++i)
14         list[i] *= 2; // dobra cada valor
15
16     cout << list << endl; // imprime lista na tela
17
18     list.clear(); // esvazia a lista
19     cout << list << endl; // imprime lista na tela
20 }
```

Arquivo LinkedList.cpp (1)

```
1 #include <stdexcept>
2 #include <iostream>
3 #include "Node.h"
4 #include "LinkedList.h"
5 using namespace std;
6
7 // Constructor: the linked list
8 // initializes empty
9 LinkedList::LinkedList() {
10     m_size = 0;
11     m_head = nullptr;
12 }
13
14 // Returns true if and only if the
15 // list is empty
16 bool LinkedList::empty() {
17     return m_head == nullptr;
18 }
```

Arquivo LinkedList.cpp (2)

```
19 // Empty the list and frees memory
20 void LinkedList::clear() {
21     while(m_head != nullptr) {
22         Node *temp = m_head;
23         m_head = m_head->next;
24         delete temp;
25     }
26     m_size = 0;
27 }
28
29 // Returns the size of the list
30 int LinkedList::size() {
31     return m_size;
32 }
```

Arquivo LinkedList.cpp (3)

```
33 // Adds an element at the end of the list
34 void LinkedList::push_back(Item data) {
35     Node *newnode = new Node(data);
36     if(m_head == nullptr) {
37         m_head = newnode;
38     } else {
39         Node *current = m_head;
40         while(current->next != nullptr) {
41             current = current->next;
42         }
43         current->next = newnode;
44     }
45     m_size++;
46 }
```


Arquivo LinkedList.cpp (4)

```
47 // Deletes an element from the end of the list
48 void LinkedList::pop_back() {
49     if(m_head == nullptr) {
50         throw std::underflow_error("empty list");
51     }
52     if(m_head->next == nullptr) {
53         delete m_head;
54         m_head = nullptr;
55         m_size = 0;
56         return;
57     }
58     Node *current = m_head;
59     while(current->next->next != nullptr) {
60         current = current->next;
61     }
62     delete current->next;
63     current->next = nullptr;
64     m_size--;
65 }
```

Arquivo LinkedList.cpp (5)

```
66 // Inserts data at any position in the range [0..size()]
67 void LinkedList::insert(int index, Item data) {
68     if(index < 0 || index > m_size) {
69         throw std::out_of_range("index out of range");
70     }
71     if(index == 0) {
72         Node *newnode = new Node(data, m_head);
73         m_head = newnode;
74         m_size++;
75         return;
76     }
77     int counter = 0;
78     Node *current = m_head;
79     while(counter < index-1) {
80         counter++;
81         current = current->next;
82     }
83     Node *newnode = new Node(data, current->next);
84     current->next = newnode;
85     m_size++;
86 }
```

Arquivo LinkedList.cpp (6)

```
87 // Deletes data at any position in the range [0..size()-1]
88 void LinkedList::remove(int index) {
89     if(index < 0 || index >= m_size) {
90         throw std::out_of_range("index out of range");
91     }
92     if(index == 0) {
93         Node *temp = m_head;
94         m_head = m_head->next;
95         delete temp;
96         m_size--;
97         return;
98     }
99     int counter = 0;
100     Node *current = m_head;
101     while(counter < index-1) {
102         counter++;
103         current = current->next;
104     }
105     Node *temp = current->next;
106     current->next = current->next->next;
107     delete temp;
108     m_size--;
109 }
```

Arquivo LinkedList.cpp (7)

```
110 // Destructor
111 LinkedList::~~LinkedList() {
112     clear();
113 }
114
115 // operator[] overloaded
116 Item& LinkedList::operator[](int index) {
117     if(index < 0 || index >= m_size) {
118         throw std::out_of_range("index out of range");
119     }
120     int counter = 0;
121     Node *current = m_head;
122     while(counter < index) {
123         counter++;
124         current = current->next;
125     }
126     return current->data;
127 }
```

Arquivo LinkedList.cpp (8)

```
128 // operator<< overloaded
129 std::ostream& operator<<(std::ostream& out, const LinkedList&
    list) {
130     Node *current = list.m_head;
131     out << "[ ";
132     while(current != nullptr) {
133         out << "(" << current->data << ") ";
134         current = current->next;
135     }
136     out << "]";
137     return out;
138 }
```

Listas Sequenciais × Encadeadas



Listas sequenciais × Listas encadeadas

- Acesso à posição k :
 - Vetor: $O(1)$
 - Lista: $O(k)$ (precisa percorrer a lista)

Listas sequenciais × Listas encadeadas

- Acesso à posição k :
 - Vetor: $O(1)$
 - Lista: $O(k)$ (precisa percorrer a lista)
- Inserção na posição 0:
 - Vetor: $O(n)$ (precisa mover itens para a direita)
 - Lista: $O(1)$

Listas sequenciais × Listas encadeadas

- Acesso à posição k :
 - Vetor: $O(1)$
 - Lista: $O(k)$ (precisa percorrer a lista)
- Inserção na posição 0:
 - Vetor: $O(n)$ (precisa mover itens para a direita)
 - Lista: $O(1)$
- Remoção da posição 0:
 - Vetor: $O(n)$ (precisa mover itens para a esquerda)
 - Lista: $O(1)$

Listas sequenciais × Listas encadeadas

- Acesso à posição k :
 - Vetor: $O(1)$
 - Lista: $O(k)$ (precisa percorrer a lista)
- Inserção na posição 0:
 - Vetor: $O(n)$ (precisa mover itens para a direita)
 - Lista: $O(1)$
- Remoção da posição 0:
 - Vetor: $O(n)$ (precisa mover itens para a esquerda)
 - Lista: $O(1)$
- Uso de espaço:
 - Vetor: provavelmente desperdiçará memória
 - Lista: não desperdiça memória, mas cada elemento consome mais memória por causa do ponteiro

Listas sequenciais × Listas encadeadas

- Acesso à posição k :
 - Vetor: $O(1)$
 - Lista: $O(k)$ (precisa percorrer a lista)
- Inserção na posição 0:
 - Vetor: $O(n)$ (precisa mover itens para a direita)
 - Lista: $O(1)$
- Remoção da posição 0:
 - Vetor: $O(n)$ (precisa mover itens para a esquerda)
 - Lista: $O(1)$
- Uso de espaço:
 - Vetor: provavelmente desperdiçará memória
 - Lista: não desperdiça memória, mas cada elemento consome mais memória por causa do ponteiro

Qual é melhor?

- depende do problema, do algoritmo e da implementação

Exercício 1



Funções Adicionais

Exercício: Implemente as seguintes funções adicionais na LinkedList.

- `LinkedList(int v[], int n)`
Construtor que recebe um array `v` com n inteiros e inicializa a lista com os n elementos do array `v`.
- `LinkedList(const LinkedList& list)`
Construtor de cópia, que recebe uma referência para uma `LinkedList list` e inicializa a nova lista com os elementos de `list`.
- `const LinkedList& operator=(const LinkedList& l)`
Implemente uma versão sobrecarregada do operador de atribuição para a `LinkedList`. O operador de atribuição permite atribuir uma lista a outra.
Exemplo: `list2 = list1;`
Após esta atribuição, `list2` e `list1` são duas listas distintas que possuem **o mesmo** conteúdo.

Funções Adicionais

- `bool equals(const LinkedList& lst)`
Determina se a lista `lst`, passada por parâmetro, é igual a lista em questão. Duas listas são iguais se têm o mesmo tamanho e o valor do k -ésimo elemento da primeira lista é igual ao k -ésimo valor da segunda.
- `void concat(LinkedList& lst)`
Concatena a lista atual com a lista `lst`. A lista `lst` não é modificada nessa operação.
- `void reverse()`: Inverte a ordem dos nós (o primeiro nó passa a ser o último, o segundo passa a ser o penúltimo, etc.) Essa operação faz isso sem criar novos nós, apenas altera os ponteiros.

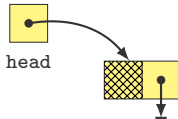
Exercício 2



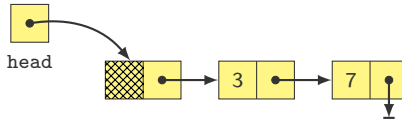
LinkedList usando nó sentinela

Exercício: Implemente a lista simplesmente encadeada **LinkedList** usando um **nó sentinela**, que é um nó auxiliar (sem conteúdo de valor) que serve apenas para marcar o início da lista.

- O ponteiro **head** **sempre** aponta para o nó sentinela.
- Quando a lista está vazia, o nó sentinela é o único nó na lista e seu campo **next** aponta para **nullptr**.



Lista vazia



Lista com 2 elementos

- Neste contexto, **reimplemente todas as operações vistas nesta aula.**

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 vetor permite **acesso randômico** dos seus elementos individuais e pode aumentar dinamicamente. O gerenciamento de memória é feito automaticamente pelo container.
- Definido na biblioteca: `<vector>`
 - Necessário `#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> primeiro;
8
9     // cria um vector com 30 números zero
10    vector<float> segundo(30);
11
12    // um vector contendo 4 inteiros com valor 100
13    // -----
14    // | 100 | 100 | 100 | 100 |
15    // -----
16    vector<int> terceiro(4, 100);
17
18    // Uma cópia do terceiro
19    vector<int> quarto(terceiro);
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, seus dados não podem ser acessados randomicamente.
- Para conseguirmos acessar elementos de todos os tipos de container, precisamos de `iteradores`.

Iteradores

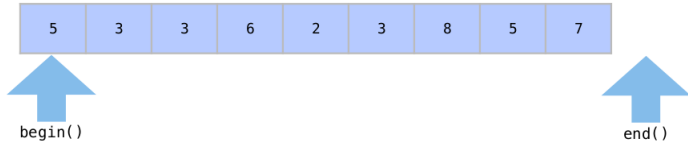
- A opção de se acessar elementos de um container através do `operator[]` é restrita apenas a containers de acesso aleatório, como o `vector`.
- Os containers `list` e `forward_list`, por exemplo, não possuem o `operator[]`. Como são implementados com listas, seus dados não podem ser acessados randomicamente.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

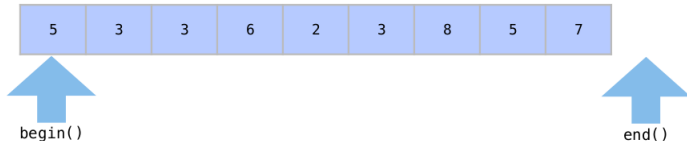
Iteradores

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



Iteradores

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



- 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);
7
8     vector<int>::iterator it; // definição de um iterador
9     int val = 1;
10
11     for(it = vec.begin(); it != vec.end(); ++it) {
12         *it = val;
13         val++;
14     }
15
16     for(it = vec.begin(); it != vec.end(); ++it)
17         cout << *it << " ";
18
19     cout << endl;
20 }
```

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     int val = 1;
9
10    for(auto it = vec.begin(); it != vec.end(); ++it) {
11        *it = val;
12        val++;
13    }
14
15    for(auto it = vec.begin(); it != vec.end(); ++it)
16        cout << *it << " ";
17
18    cout << endl;
19 }
```

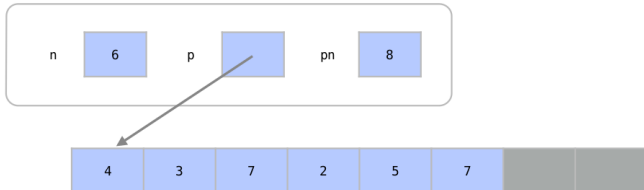
Funções de vector que usam iteradores

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;



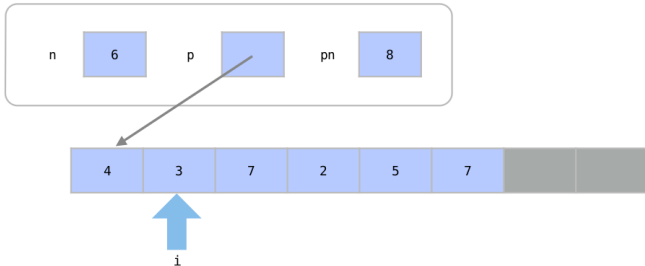
Funções de vector que usam iteradores

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;
```

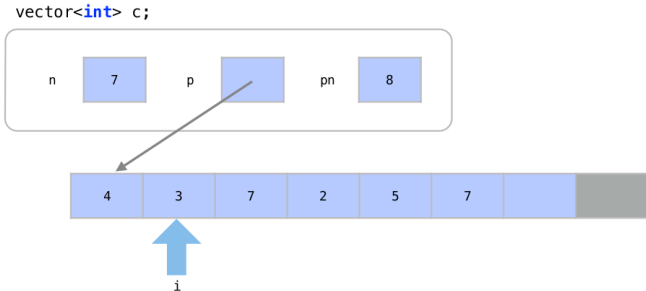
vector<int> c;



Funções de vector que usam iteradores

insert

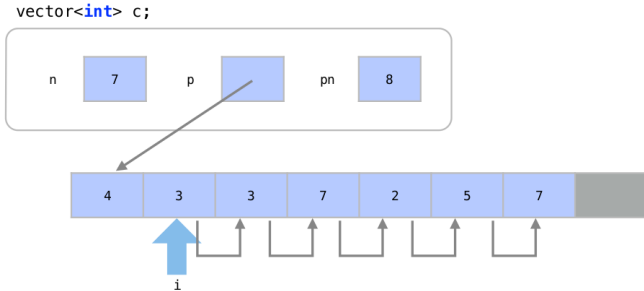
- Queremos inserir 11 na posição 2 do vector. Uma sequência de passos deve ser executada.
1. Primeiro, a variável n é incrementada. Em alguns casos, pode ser que um novo array precise ser alocado, levando a um custo $O(n)$.



Funções de vector que usam iteradores

insert

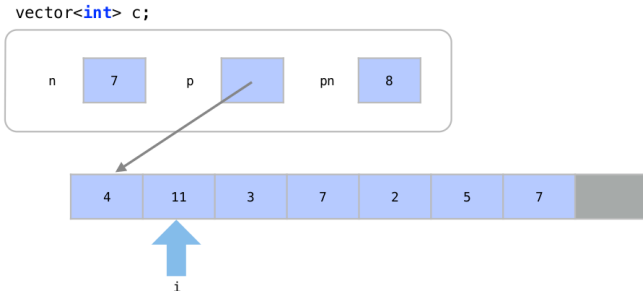
2. Todos os elementos entre i e o penúltimo elemento precisam ser deslocados para a próxima posição do arranjo. Esta operação tem um custo $O(n - i)$. Se i for o último elemento, temos um melhor caso $O(1)$. Se i for o primeiro elemento, temos um pior caso $O(n)$.



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:

- Nos livros
- Na internet
- <https://www.cplusplus.com/reference/vector/vector/>
- <https://www.learncpp.com/>
- <https://www.geeksforgeeks.org/vector-in-cpp-stl/>

FIM

