



## INE5408-03208A | INE5609-03238B (20182) - Estruturas de Dados

Painel ► Agrupamentos de Turmas ► INE5408-03208A | INE5609-03238B (20182) ► Tópico 9 ► Implementação de Lista Encadeada

## NAVEGAÇÃO



## Painel

- Página inicial do site
- Moodle UFSC
- ▼ Curso atual
  - ▼ INE5408-03208A | INE5609-03238B (20182)
    - Participantes
    - Emblemas
    - Geral
    - Tópico 1
    - Tópico 2
    - Tópico 3
    - Tópico 4
    - Tópico 5
    - Tópico 6
    - Tópico 7
    - Tópico 8
    - ▼ Tópico 9
      -  VisualAlgo de Lista Encadeada
        - ▼  Implementação de Lista Encadeada
          - Descrição
          - Enviar
          - Editar
          - Visualizar envios
      - Tópico 10
    - Meus cursos

## ADMINISTRAÇÃO



- Administração do curso

Descrição Enviar Editar Visualizar envios

## Nota

Revisado em domingo, 9 Set 2018, 14:14 por Atribuição automática de nota  
Nota 100 / 100

## Relatório de avaliação

[\*]Summary of tests

Enviado em domingo, 9 Set 2018, 14:13 (Baixar)

## linked\_list.h

```
1  //! Copyright 2018 Matheus Henrique Schaly
2
3  #ifndef STRUCTURES_LINKED_LIST_H
4  #define STRUCTURES_LINKED_LIST_H
5
6  #include <cstdlib>
7  #include <stdexcept>
8
9
10 namespace structures {
11
12     //! Dynamic Simple Linked List
13     template<typename T>
14     class LinkedList {
15     public:
16         //! Constructor
17         LinkedList();
18         //! Destructor
19         ~LinkedList();
20         //! Removes list's elements
21         void clear();
22         //! Inserts an element at the list's leftmost part of the list
23         void push_back(const T& data);
24         //! Inserts an element at the list's leftmost part
25         void push_front(const T& data);
26         //! Inserts an element at the given index
27         void insert(const T& data, std::size_t index);
28         //! Inserts an element sorted by data
29         void insert_sorted(const T& data);
30         //! Returns an element's data at index
31         T& at(std::size_t index);
32         //! Removes an element from index
33         T pop(std::size_t index);
34         //! Removes an element from the rightmost part of the list
35         T pop_back();
36         //! Removes an element from the leftmost part of the list
37         T pop_front();
38         //! Removes an element with the given data
39         void remove(const T& data);
40         //! Returns true if list is empty and false otherwise
41         bool empty() const;
42         //! Checks if the list contains the node with the given data
43         bool contains(const T& data) const;
44         //! Returns the index of the given data
45         std::size_t find(const T& data) const;
46         //! Returns the current size of the list
47         std::size_t size() const;
48
49     private:
50         class Node { // Elemento
51         public:
52             //! Constructor with 1 parameter
53             explicit Node(const T& data):
54                 data_{data}
55             {}
56
57             //! Constructor with 2 parameters
58             Node(const T& data, Node* next):
59                 data_{data},
60                 next_{next}
61             {}
62
63             //! Data's getter
64             T& data() {
65                 return data_;
66             }
67
68             //! Data's constant getter
69             const T& data() const {
70                 return data_;
71             }
72
73             //! Next's getter
74             Node* next() {
75                 return next_;
76             }
77
78             //! Next's constant getter
79             const Node* next() const {
80                 return next_;
81             }
82
83             //! Next's setter
84             void next(Node* node) {
85                 next_ = node;
86             }
87
88         private:
89             //! Node's data
90             T data_;
91
92             //! Node's next node
93             Node* next_{nullptr};
94         };
95
96         //! Returns the list's last node
97         Node* end() { // último nodo da lista
98             auto it = head;
99             for (auto i = 1; i < size(); ++i) {
100                 it = it->next();
101             }
102             return it;
103         }
104
105         //! List's leftmost node
106         Node* head{nullptr};
107
108         //! List's current size
109         std::size_t size_{0u};
110     };
111
112 } // namespace structures
113
114 //! Constructor
115 template<typename T>
116 structures::LinkedList<T>::LinkedList() {}
```

```

117
118 //! Destructor
119 template<typename T>
120 struct LinkedList<T>::~LinkedList() {
121     clear();
122 }
123
124 //! Removes list's elements
125 template<typename T>
126 void LinkedList<T>::clear() {
127     while (!empty()) {
128         pop_front();
129     }
130 }
131
132 //! Inserts an element at the list's leftmost part of the list
133 template<typename T>
134 void LinkedList<T>::push_back(const T& data) {
135     insert(data, size_);
136 }
137
138 //! Inserts an element at the list's leftmost part
139 template<typename T>
140 void LinkedList<T>::push_front(const T& data) {
141     Node* node = new Node(data, head);
142     if (node == nullptr) {
143         throw std::out_of_range("A lista esta cheia.");
144     }
145     head = node;
146     size_++;
147 }
148
149 //! Inserts an element at the given index
150 template<typename T>
151 void LinkedList<T>::insert(const T& data, std::size_t index) {
152     if (index > size_ || index < 0) {
153         throw std::out_of_range("Indice invalido.");
154     }
155     if (index == 0) {
156         push_front(data);
157     } else {
158         Node* node = new Node(data);
159         Node* previous_node = head;
160         if (node == nullptr) {
161             throw std::out_of_range("A lista esta cheia.");
162         }
163         std::size_t i = 1;
164         while (i < index) {
165             previous_node = previous_node->next();
166             i++;
167         }
168         node->next(previous_node->next());
169         previous_node->next(node);
170         size_++;
171     }
172 }
173
174 //! Inserts an element sorted by data
175 template<typename T>
176 void LinkedList<T>::insert_sorted(const T& data) {
177     if (head == nullptr) {
178         push_front(data);
179     } else {
180         Node* previous_node = head;
181         std::size_t i = 0;
182         while (previous_node != nullptr && data > previous_node->data()) {
183             previous_node = previous_node->next();
184             i++;
185         }
186         try {
187             insert(data, i);
188         } catch (std::out_of_range error) {
189             throw error;
190         }
191     }
192 }
193
194 //! Returns an element's data at index
195 template<typename T>
196 T& LinkedList<T>::at(std::size_t index) {
197     if (index >= size_ || index < 0) {
198         throw std::out_of_range("Indice invalido.");
199     }
200     Node* node = head;
201     std::size_t i = 1;
202     while (i <= index) {
203         node = node->next();
204         i++;
205     }
206     return node->data();
207 }
208
209 //! Removes an element from index
210 template<typename T>
211 T LinkedList<T>::pop(std::size_t index) {
212     if (empty() || index >= size_ || index < 0) {
213         throw std::out_of_range("Indice invalido.");
214     }
215     if (index == 0) {
216         return pop_front();
217     } else {
218         Node* node;
219         Node* previous_node = head;
220         std::size_t i = 1;
221         while (i < index) {
222             previous_node = previous_node->next();
223             i++;
224         }
225         node = previous_node->next();
226         previous_node->next(node->next());
227         T deleted_data = node->data();
228         delete node;
229         size_--;
230         return deleted_data;
231     }
232 }
233
234 //! Removes an element from the rightmost part of the list
235 template<typename T>
236 T LinkedList<T>::pop_back() {
237     try {
238         return pop(size_ - 1);
239     } catch (std::out_of_range error) {
240         throw error;
241     }
242 }
243
244 //! Removes an element from the leftmost part of the list
245 template<typename T>
246 T LinkedList<T>::pop_front() {
247     if (empty()) {
248         throw std::out_of_range("A lista esta vazia");
249     }
250     Node* node = head;
251     T deleted_data = node->data();
252     head = node->next();
253     delete node;
254     size_--;
255     return deleted_data;
256 }
257
258 //! Removes an element with the given data
259 template<typename T>
260 void LinkedList<T>::remove(const T& data) {
261     pop(find(data));
262 }
263
264 //! Returns true if list is empty and false otherwise
265 template<typename T>
266 bool LinkedList<T>::empty() const {
267     return size_ == 0;
268 }

```

```

267     return size_ == 0;
268 }
269
270 /// Checks if the list contains the node with the given data
271 template<typename T>
272 bool structures::LinkedList<T>::contains(const T& data) const {
273     if (empty()) {
274         throw std::out_of_range("A lista esta vazia.");
275     } else {
276         Node* node = head;
277         std::size_t i = 1;
278         while (i < size_) {
279             if (node->data() == data) {
280                 return true;
281             }
282             node = node->next();
283             i++;
284         }
285         return false;
286     }
287 }
288
289 /// Returns the index of the given data
290 template<typename T>
291 std::size_t structures::LinkedList<T>::find(const T& data) const {
292     if (empty()) {
293         throw std::out_of_range("A lista esta vazia.");
294     } else {
295         Node* node = head;
296         std::size_t i = 0;
297         while (i < size_ - 1) {
298             if (node->data() == data) {
299                 return i;
300             }
301             node = node->next();
302             i++;
303         }
304         if (node->data() == data) {
305             return size_ - 1;
306         }
307     }
308     return size_;
309 }
310
311 /// Returns the current size of the list
312 template<typename T>
313 std::size_t structures::LinkedList<T>::size() const {
314     return size_;
315 }
316
317 #endif
318

```