









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

Painel ► Agrupamentos de Turmas ► INE5408-03208A | INE5609-03238B (20182) ► Tópico 10 ► Implementação de Lista Duplamente 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
 - ▼ Tópico 10
 -  Implementação de Pilha Encadeada
 -  Testes (Pilha Encadeada)
 -  Implementação de Fila Encadeada
 -  Testes (Fila Encadeada)
 - ▼  Implementação de Lista Duplamente Encadeada
 - Descrição
 - Enviar
 - Editar
 - Visualizar envios
 -  Testes (Lista Duplamente Encadeada)
 -  Videoaula de Pilha e Fila Encadeada
 -  Lecture 8 (rev. 04-set)
 - Tópico 11
 - Tópico 12
 - Tópico 13
- Meus cursos

ADMINISTRAÇÃO



- Administração do curso

Descrição Enviar Editar Visualizar envios

Nota

Revisado em domingo, 23 Set 2018, 14:04 por Atribuição automática de nota

Nota 100 / 100

Relatório de avaliação

[*] Summary of tests

Enviado em domingo, 23 Set 2018, 14:03 (Baixar)

doubly_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 DoublyLinkedList {
15     public:
16         //! Constructor
17         DoublyLinkedList();
18         //! Destructor
19         ~DoublyLinkedList();
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             //! Constructor with 3 parameters
64             Node(const T& data, Node* next, Node* prev):
65                 data_{data},
66                 next_{next},
67                 prev_{prev}
68             {}
69
70             //! Data's getter
71             T& data() {
72                 return data_;
73             }
74
75             //! Data's constant getter
76             const T& data() const {
77                 return data_;
78             }
79
80             //! Next's getter
81             Node* next() {
82                 return next_;
83             }
84
85             //! Next's constant getter
86             const Node* next() const {
87                 return next_;
88             }
89
90             //! Prev's constant
91             Node* prev() {
92                 return prev_;
93             }
94
95             //! Prev's constant getter
96             const Node* prev() const {
97                 return prev_;
98             }
99
100             //! Prev's setter
101             void prev(Node* node) {
102                 prev_ = node;
103             }
104
105             //! Next's setter
106             void next(Node* node) {
107                 next_ = node;
108             }
109
110     private:
111         //! Node's data
112         T data_;
113
114         //! Nodes next node
115         Node* next_{nullptr};
116         Node* prev_{nullptr};
```

```

117 };
118
119 // Returns the list's last node
120 Node* end() { // último nodo da lista
121     auto it = head;
122     for (auto i = 1u; i < size(); ++i) {
123         it = it->next();
124     }
125     return it;
126 }
127
128 // List's leftmost node
129 Node* head(nullptr);
130
131 // List's rightmost node
132 Node* tail(nullptr);
133
134 // List's current size
135 std::size_t size_{0u};
136 };
137
138 } // namespace structures
139
140 // Constructor
141 template<typename T>
142 structures::DoublyLinkedList<T>::DoublyLinkedList() {}
143
144 // Destructor
145 template<typename T>
146 structures::DoublyLinkedList<T>::~DoublyLinkedList() {
147     clear();
148 }
149
150 // Removes list's elements
151 template<typename T>
152 void structures::DoublyLinkedList<T>::clear() {
153     while (!empty()) {
154         pop_front();
155     }
156 }
157
158 // Inserts an element at the list's leftmost part of the list
159 template<typename T>
160 void structures::DoublyLinkedList<T>::push_back(const T& data) {
161     insert(data, size_);
162 }
163
164 // Inserts an element at the list's leftmost part
165 template<typename T>
166 void structures::DoublyLinkedList<T>::push_front(const T& data) {
167     Node* node = new Node(data, head);
168     if (node == nullptr) {
169         throw std::out_of_range("A lista esta cheia.");
170     }
171     head = node;
172     if (size_ == 0) {
173         tail = node;
174     }
175     size_++;
176 }
177
178 // Inserts an element at the given index
179 template<typename T>
180 void structures::DoublyLinkedList<T>::insert(const T& data, std::size_t index) {
181     if (index > size_ || index < 0) {
182         throw std::out_of_range("Índice inválido");
183     }
184     if (index == 0) {
185         push_front(data);
186     } else {
187         Node* node = new Node(data);
188         Node* previous_node = head;
189         if (node == nullptr) {
190             throw std::out_of_range("A lista esta cheia.");
191         }
192         if (index == size_) {
193             node->next(tail);
194             tail->next(node);
195             tail = node;
196         } else {
197             std::size_t i = 1;
198             while (i < index) {
199                 previous_node = previous_node->next();
200                 i++;
201             }
202             node->next(previous_node->next());
203             node->prev(previous_node);
204             previous_node->next()->prev(node);
205             previous_node->next(node);
206         }
207         size_++;
208     }
209 }
210
211 // Inserts an element sorted by data
212 template<typename T>
213 void structures::DoublyLinkedList<T>::insert_sorted(const T& data) {
214     if (head == nullptr) {
215         push_front(data);
216     } else {
217         Node* previous_node = head;
218         std::size_t i = 0;
219         while (previous_node != nullptr && data > previous_node->data()) {
220             previous_node = previous_node->next();
221             i++;
222         }
223         try {
224             insert(data, i);
225         } catch (std::out_of_range error) {
226             throw error;
227         }
228     }
229 }
230
231 // Returns an element's data at index
232 template<typename T>
233 T& structures::DoublyLinkedList<T>::at(std::size_t index) {
234     if (index >= size_ || index < 0) {
235         throw std::out_of_range("Índice inválido.");
236     }
237     Node* node = head;
238     std::size_t i = 1;
239     while (i <= index) {
240         node = node->next();
241         i++;
242     }
243     return node->data();
244 }
245
246 // Removes an element from index
247 template<typename T>
248 void structures::DoublyLinkedList<T>::pop(std::size_t index) {
249     if (empty() || index >= size_ || index < 0) {
250         throw std::out_of_range("Índice inválido.");
251     }
252     if (index == 0) {
253         return pop_front();
254     } else {
255         Node* node;
256         Node* previous_node = head;
257         std::size_t i = 1;
258         while (i < index) {
259             previous_node = previous_node->next();
260             i++;
261         }
262         if (index == size_) {
263             node = tail;
264             tail = node->prev();
265         } else {
266             node = previous_node->next();
267             previous_node->next(previous_node->next());
268         }
269         size_--;
270     }
271 }

```

```

267         previous_node -> next() -> next();
268         // node -> next() -> prev(previous_node);
269     }
270     T deleted_data = node -> data();
271     delete node;
272     size_--;
273     return deleted_data;
274 }
275 }
276
277 /// Removes an element from the rightmost part of the list
278 template<typename T>
279 T structures::DoublyLinkedList<T>::pop_back() {
280     try {
281         return pop(size_ - 1);
282     } catch (std::out_of_range error) {
283         throw error;
284     }
285 }
286
287 /// Removes an element from the leftmost part of the list
288 template<typename T>
289 T structures::DoublyLinkedList<T>::pop_front() {
290     if (empty()) {
291         throw std::out_of_range("A lista esta vazia");
292     }
293     Node* node = head;
294     T deleted_data = node -> data();
295     head = node -> next();
296     delete node;
297     size_--;
298     return deleted_data;
299 }
300
301 /// Removes an element with the given data
302 template<typename T>
303 void structures::DoublyLinkedList<T>::remove(const T& data) {
304     pop(find(data));
305 }
306
307 /// Returns true if list is empty and false otherwise
308 template<typename T>
309 bool structures::DoublyLinkedList<T>::empty() const {
310     return size_ == 0;
311 }
312
313 /// Checks if the list contains the node with the given data
314 template<typename T>
315 bool structures::DoublyLinkedList<T>::contains(const T& data) const {
316     if (empty()) {
317         throw std::out_of_range("A lista esta vazia.");
318     } else {
319         Node* node = head;
320         std::size_t i = 1;
321         while (i < size_) {
322             if (node -> data() == data) {
323                 return true;
324             }
325             node = node -> next();
326             i++;
327         }
328         return false;
329     }
330 }
331
332 /// Returns the index of the given data
333 template<typename T>
334 std::size_t structures::DoublyLinkedList<T>::find(const T& data) const {
335     if (empty()) {
336         throw std::out_of_range("A lista esta vazia.");
337     } else {
338         Node* node = head;
339         std::size_t i = 0;
340         while (i < size_ - 1) {
341             if (node -> data() == data) {
342                 return i;
343             }
344             node = node -> next();
345             i++;
346         }
347         if (node -> data() == data) {
348             return size_ - 1;
349         }
350     }
351     return size_;
352 }
353
354 /// Returns the current size of the list
355 template<typename T>
356 std::size_t structures::DoublyLinkedList<T>::size() const {
357     return size_;
358 }
359
360 #endif
361

```