



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

Painel ► Agrupamentos de Turmas ► INE5408-03208A | INE5609-03238B (20182) ► Tópico 11 ► Implementação de Lista Circular Simples

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
 - ▼ Tópico 11
 - Implementação de Lista Circular Simples
 - Descrição
 - Enviar
 - Editar
 - Visualizar envios
 - Testes (Lista Circular)
 - Implementação de Lista Circular Dupla
 - Testes (Lista Circular Duplamente Encadeada)
 - Vídeo Aula sobre Listas Circulares
 - Lecture 9
 - Tópico 12
 - Tópico 13
 - Tópico 14
 - Prova Teórica I

ADMINISTRAÇÃO



- Administração do curso

Descrição Enviar Editar Visualizar envios

Nota

Revisado em sábado, 29 Set 2018, 22:10 por Atribuição automática de nota

Nota 100 / 100

Relatório de avaliação

[+]Summary of tests

Enviado em sábado, 29 Set 2018, 22:10 (Baixar)

circular_list.h

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

```

117 structures::CircularList<T>::CircularList() {}
118 //! Destructor
119 template<typename T>
120 structures::CircularList<T>::~CircularList() {
121     clear();
122 }
123 }
124 //! Removes list's elements
125 template<typename T>
126 void structures::CircularList<T>::clear() {
127     while (!empty()) {
128         pop_front();
129     }
130 }
131 }
132 //! Inserts an element at the list's rightmost part
133 template<typename T>
134 void structures::CircularList<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 structures::CircularList<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 structures::CircularList<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         if (index == size_) {
172             node -> next(head); // Fastest way to build circular list?
173         }
174     }
175 }
176 }
177 //! Inserts an element sorted by data
178 template<typename T>
179 void structures::CircularList<T>::insert_sorted(const T& data) {
180     if (head == nullptr) {
181         push_front(data);
182     } else {
183         Node* previous_node = head;
184         std::size_t i = 0;
185         while (previous_node != nullptr && data > previous_node -> data()) {
186             previous_node = previous_node -> next();
187             i++;
188         }
189         try {
190             insert(data, i);
191         } catch (std::out_of_range error) {
192             throw error;
193         }
194     }
195 }
196 }
197 //! Returns an element's data at index
198 template<typename T>
199 T& structures::CircularList<T>::at(std::size_t index) {
200     if (index >= size_ || index < 0) {
201         throw std::out_of_range("Indice invalido.");
202     }
203     Node* node = head;
204     std::size_t i = 1;
205     while (i <= index) {
206         node = node -> next();
207         i++;
208     }
209     return node -> data();
210 }
211 }
212 //! Removes an element from index
213 template<typename T>
214 T structures::CircularList<T>::pop(std::size_t index) {
215     if (empty() || index >= size_ || index < 0) {
216         throw std::out_of_range("Indice invalido.");
217     }
218     if (index == 0) {
219         return pop_front();
220     } else {
221         Node* node;
222         Node* previous_node = head;
223         std::size_t i = 1;
224         while (i < index) {
225             previous_node = previous_node -> next();
226             i++;
227         }
228         node = previous_node -> next();
229         previous_node -> next(node -> next());
230         T deleted_data = node -> data();
231         delete node;
232         size_--;
233         if (index == size_) {
234             previous_node -> next(head); // Fastest way to build circular list?
235         }
236         return deleted_data;
237     }
238 }
239 }
240 //! Removes an element from the rightmost part of the list
241 template<typename T>
242 T structures::CircularList<T>::pop_back() {
243     try {
244         return pop(size_ - 1);
245     } catch (std::out_of_range error) {
246         throw error;
247     }
248 }
249 }
250 //! Removes an element from the leftmost part of the list
251 template<typename T>
252 T structures::CircularList<T>::pop_front() {
253     if (empty()) {
254         throw std::out_of_range("A lista esta vazia");
255     }
256     Node* node = head;
257     T deleted_data = node -> data();
258     head = node -> next();
259     delete node;
260     size_--;
261     return deleted_data;
262 }
263 }
264 //! Removes an element with the given data
265 template<typename T>
266 void structures::CircularList<T>::remove(const T& data) {

```

```

267 void structures::CircularList<T>::remove(const T& data) {
268     pop(find(data));
269 }
270
271 //! Returns true if list is empty and false otherwise
272 template<typename T>
273 bool structures::CircularList<T>::empty() const {
274     return size_ == 0;
275 }
276
277 //! Checks if the list contains the node with the given data
278 template<typename T>
279 bool structures::CircularList<T>::contains(const T& data) const {
280     if (empty()) {
281         throw std::out_of_range("A lista este vazia.");
282     } else {
283         Node* node = head;
284         std::size_t i = 1;
285         while (i < size_) {
286             if (node->data() == data) {
287                 return true;
288             }
289             node = node->next();
290             i++;
291         }
292         return false;
293     }
294 }
295
296 //! Returns the index of the given data
297 template<typename T>
298 std::size_t structures::CircularList<T>::find(const T& data) const {
299     if (empty()) {
300         throw std::out_of_range("A lista este vazia.");
301     } else {
302         Node* node = head;
303         std::size_t i = 0;
304         while (i < size_ - 1) {
305             if (node->data() == data) {
306                 return i;
307             }
308             node = node->next();
309             i++;
310         }
311         if (node->data() == data) {
312             return size_ - 1;
313         }
314     }
315     return size_;
316 }
317
318 //! Returns the current size of the list
319 template<typename T>
320 std::size_t structures::CircularList<T>::size() const {
321     return size_;
322 }
323
324 #endif
325

```