



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

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

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

ADMINISTRAÇÃO



- Administração do curso

Descrição Enviar Editar Visualizar envios

Nota

Revisado em sábado, 29 Set 2018, 22:18 por Atribuição automática de nota
Nota 100 / 100

Relatório de avaliação

[+]Summary of tests

Enviado em sábado, 29 Set 2018, 22:18 (Baixar)

doubly_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 Doubly Circular Linked List
12  template<typename T>
13  class DoublyCircularList {
14  public:
15      //! Constructor
16      DoublyCircularList();
17      //! Destructor
18      ~DoublyCircularList();
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          //! Constructor with 2 parameters
65          Node(const T& data, Node* next, Node* prev):
66              data_{data},
67              next_{next},
68              prev_{prev}
69          {}
70
71          //! Data's getter
72          T& data() {
73              return data_;
74          }
75
76          //! Data's constant getter
77          const T& data() const {
78              return data_;
79          }
80
81          //! Prev's constant
82          Node* prev() {
83              return prev_;
84          }
85
86          //! Prev's constant getter
87          const Node* prev() const {
88              return prev_;
89          }
90
91          //! Prev's setter
92          void prev(Node* node) {
93              prev_ = node;
94          }
95
96          //! Next's getter
97          Node* next() {
98              return next_;
99          }
100
101          //! Next's constant getter
102          const Node* next() const {
103              return next_;
104          }
105
106          //! Next's setter
107          void next(Node* node) {
108              next_ = node;
109          }
110
111      private:
112          //! Node's data
113          T data_;
114
115          //! Node's next
116          Node* next_;
```

```

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

```

```

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

```