



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

Painel ▸ Agrupamentos de Turmas ▸ INE5408-03208A | INE5609-03238B (20182) ▸ Tópico 7 ▸ Lista em vetor - aplicação de lista de ponteiros p...

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
 - ▀ Lista em vetor - aplicação de lista de ponteiros p...
 - ▀ Descrição
 - ▀ Enviar
 - ▀ Editar
 - ▀ Visualizar envios
 - 🔍 Testes (lista de strings)
 - Tópico 8
 - Tópico 9
 - Tópico 10
- ▀ Meus cursos

ADMINISTRAÇÃO



- Administração do curso

Descrição Enviar Editar Visualizar envios

Nota

Revisado em quarta, 5 Set 2018, 21:13 por Atribuição automática de nota

Nota 100 / 100

Relatório de avaliação

[+] Summary of tests

Enviado em quarta, 5 Set 2018, 21:13 (Baixar)

string_list.h

```
1  //! Copyright 2018 Matheus Henrique Schaly
2
3  #ifndef STRUCTURES_STRING_LIST_H
4  #define STRUCTURES_STRING_LIST_H
5
6  #include <stdint>
7  #include <stdexcept> // C++ exceptions
8  #include <string>
9
10
11 namespace structures {
12
13     //! Static list with pointers
14     template<typename T>
15     class ArrayList {
16     public:
17         //! Constructor
18         ArrayList();
19         //! Constructor with size parameter
20         explicit ArrayList(std::size_t max_size);
21         //! Destructor
22         ~ArrayList();
23         //! Clears the list
24         void clear();
25         //! Pushes an element to the back of the list
26         void push_back(const T& data);
27         //! Pushes an element to the front of the list
28         void push_front(const T& data);
29         //! Inserts an element at a specific index
30         void insert(const T& data, std::size_t index);
31         //! Inserts an element in a sorted position
32         void insert_sorted(const T& data);
33         //! Removes an element from a specific index
34         T pop(std::size_t index);
35         //! Removes an element from the back of the list
36         T pop_back();
37         //! Removes an element from the front of the list
38         T pop_front();
39         //! Removes the first element containing the data
40         void remove(const T& data);
41         //! Verifies if the list is full
42         bool full() const;
43         //! Verifies if the list is empty
44         bool empty() const;
45         //! Verifies if the list contains the data
46         bool contains(const T& data) const;
47         //! Returns the index of the first element containing the data, else size
48         std::size_t find(const T& data) const;
49         //! Returns the current size of the list
50         std::size_t size() const;
51         //! Returns the maximum size of the list
52         std::size_t max_size() const;
53         //! Returns the element at index
54         T& at(std::size_t index);
55         //! Overloads the [] operator
56         T& operator[](std::size_t index);
57         //! Returns the element at index as constant
58         const T& at(std::size_t index) const;
59         //! Overloads the [] operator, but returns it as a constant
60         const T& operator[](std::size_t index) const;
61
62     private:
63         T* contents;
64         std::size_t size_;
65         std::size_t max_size_;
66
67         static const auto DEFAULT_MAX = 10u;
68     };
69
70     //! ArrayListString e' uma especializacao da classe ArrayList
71     class ArrayListString : public ArrayList<char*> {
72     public:
73         //! Constructor
74         ArrayListString() : ArrayList() {}
75         //! Constructor with parameter
76         explicit ArrayListString(std::size_t max_size) : ArrayList(max_size) {}
77         //! Destructor
78         ~ArrayListString();
79         //! Clears the list
80         void clear();
81         //! Inserts an element to the back of the list
82         void push_back(const char* data);
83         //! Inserts an element to the front of the list
84         void push_front(const char* data);
85         //! Inserts an element at a specific index
86         void insert(const char* data, std::size_t index);
87         //! Inserts an element in a sorted position
88         void insert_sorted(const char* data);
89         //! Removes an element from a specific index
90         char* pop(std::size_t index);
91         //! Removes an element from the back of the list
92         char* pop_back();
93         //! Removes an element from the front of the list
94         char* pop_front();
```

```

95  //! Removes an element containing the data
96  void remove(const char *data);
97  //! Verifies if the list contains the data
98  bool contains(const char *data);
99  //! Returns the index of the first element containing the data, else size
100 std::size_t find(const char *data);
101 };
102
103 // namespace structures
104
105 structures::ArrayListString::~ArrayListString() {
106     clear();
107 }
108
109 void structures::ArrayListString::clear() {
110     ArrayList::clear();
111 }
112
113 void structures::ArrayListString::push_back(const char *data) {
114     if (full()) {
115         throw std::out_of_range("A lista esta cheia.");
116     } else {
117         char *datanew = new char[strlen(data) + 1];
118         snprintf(datanew, strlen(data)+1, "%s", data);
119         ArrayList::push_back(datanew);
120     }
121 }
122
123 void structures::ArrayListString::push_front(const char *data) {
124     if (full()) {
125         throw std::out_of_range("A lista esta cheia.");
126     } else {
127         char *datanew = new char[strlen(data) + 1];
128         snprintf(datanew, strlen(data)+1, "%s", data);
129         ArrayList::push_front(datanew);
130     }
131 }
132
133 void structures::ArrayListString::insert(const char *data,
134                                         std::size_t index) {
135     if (full() || (index < 0 || index >= ArrayList::size())) {
136         throw std::out_of_range("A lista esta cheia.");
137     } else {
138         char *datanew = new char[strlen(data) + 1];
139         snprintf(datanew, strlen(data)+1, "%s", data);
140         ArrayList::insert(datanew, index);
141     }
142 }
143
144 void structures::ArrayListString::insert_sorted(const char *data) {
145     if (full()) {
146         throw std::out_of_range("A lista esta cheia.");
147     } else {
148         char *datanew = new char[strlen(data) + 1];
149         snprintf(datanew, strlen(data)+1, "%s", data);
150         for (std::size_t i = 0; i < ArrayList::size(); i++) {
151             if (strcmp(ArrayList::at(i), data) > 0) {
152                 ArrayList::insert(datanew, i);
153                 return;
154             }
155         }
156         ArrayList::push_back(datanew);
157     }
158 }
159
160 char* structures::ArrayListString::pop(std::size_t index) {
161     if (empty() || (index < 0 || index >= ArrayList::size())) {
162         throw std::out_of_range("A lista esta vazia.");
163     } else {
164         char* removed_element = ArrayList::pop(index);
165         return removed_element;
166     }
167 }
168
169 char* structures::ArrayListString::pop_back() {
170     if (empty()) {
171         throw std::out_of_range("A lista esta vazia");
172     } else {
173         char* removed_element = ArrayList::pop_back();
174         return removed_element;
175     }
176 }
177
178 char* structures::ArrayListString::pop_front() {
179     if (empty()) {
180         throw std::out_of_range("A lista esta vazia");
181     } else {
182         char* removed_element = ArrayList::pop_front();
183         return removed_element;
184     }
185 }
186
187 void structures::ArrayListString::remove(const char *data) {
188     if (empty()) {
189         throw std::out_of_range("A lista esta vazia");
190     } else {
191         for (std::size_t i = 0; i < ArrayList::size(); i++) {
192             if (strcmp(ArrayList::at(i), data) == 0) {
193                 pop(i);
194             }
195         }
196     }
197 }
198
199 bool structures::ArrayListString::contains(const char *data) {
200     for (std::size_t i = 0; i < ArrayList::size(); i++) {
201         if (strcmp(ArrayList::at(i), data) == 0) {
202             return true;
203         }
204     }
205     return false;
206 }
207
208 std::size_t structures::ArrayListString::find(const char *data) {
209     for (std::size_t i = 0; i < ArrayList::size(); i++) {
210         if (strcmp(ArrayList::at(i), data) == 0) {
211             return i;
212         }
213     }
214     return ArrayList::size();
215 }
216
217 // Super Class
218
219 template<typename T>
220 structures::ArrayList<T>::ArrayList() {
221     ArrayList(DEFAULT_MAX);
222 }

```

```

224 template<typename T>
225 structures::ArrayList<T>::ArrayList(std::size_t max_size) {
226     size_ = 0;
227     max_size_ = max_size;
228     contents = new T[max_size_];
229 }
230
231 template<typename T>
232 structures::ArrayList<T>::~ArrayList() {
233     delete[] contents;
234 }
235
236 template<typename T>
237 void structures::ArrayList<T>::clear() {
238     size_ = 0;
239 }
240
241 template<typename T>
242 void structures::ArrayList<T>::push_back(const T& data) {
243     if (full()) {
244         throw std::out_of_range("A lista esta cheia.");
245     } else {
246         contents[size_] = data;
247         size_++;
248     }
249 }
250
251 template<typename T>
252 void structures::ArrayList<T>::push_front(const T& data) {
253     if (full()) {
254         throw std::out_of_range("A lista esta cheia.");
255     } else {
256         for (int i = 0; i < size_; i++) {
257             contents[size_ - i] = contents[size_ - i - 1];
258         }
259         size_++;
260         contents[0] = data;
261     }
262 }
263
264 template<typename T>
265 void structures::ArrayList<T>::insert(const T& data, std::size_t index) {
266     if (full() || (index < 0 || index >= size_)) {
267         throw std::out_of_range("A lista esta cheia.");
268     } else {
269         if (index == 0) {
270             push_front(data);
271             return;
272         }
273         if (index == size_) {
274             push_back(data);
275             return;
276         }
277         for (int i = 0; i < size_ - index; i++) {
278             contents[size_ - i] = contents[size_ - i - 1];
279         }
280         size_++;
281         contents[index] = data;
282     }
283 }
284
285 template<typename T>
286 void structures::ArrayList<T>::insert_sorted(const T& data) {
287     if (full()) {
288         throw std::out_of_range("A lista esta cheia.");
289     } else {
290         for (int i = 0; i < size_; i++) {
291             if (contents[i] >= data) {
292                 insert(data, i);
293                 return;
294             }
295         }
296         push_back(data);
297     }
298 }
299
300 template<typename T>
301 T structures::ArrayList<T>::pop(std::size_t index) {
302     if (empty() || (index < 0 || index >= size_)) {
303         throw std::out_of_range("A lista esta vazia.");
304     } else {
305         T removed_element = contents[index];
306         for (int i = index; i < size_ - 1; i++) {
307             contents[i] = contents[i + 1];
308         }
309         size_--;
310         return removed_element;
311     }
312 }
313
314 template<typename T>
315 T structures::ArrayList<T>::pop_back() {
316     if (empty()) {
317         throw std::out_of_range("A lista esta vazia.");
318     } else {
319         size_--;
320         return contents[size_];
321     }
322 }
323
324 template<typename T>
325 T structures::ArrayList<T>::pop_front() {
326     if (empty()) {
327         throw std::out_of_range("A lista esta vazia.");
328     } else {
329         T removed_element = contents[0];
330         for (int i = 0; i < size_ - 1; i++) {
331             contents[i] = contents[i + 1];
332         }
333         size_--;
334         return removed_element;
335     }
336 }
337
338 template<typename T>
339 void structures::ArrayList<T>::remove(const T& data) {
340     if (empty()) {
341         throw std::out_of_range("A lista esta vazia.");
342     } else {
343         for (int i = 0; i < size_; i++) {
344             if (contents[i] == data) {
345                 pop(i);
346             }
347         }
348     }
349 }
350
351 template<typename T>

```

```

352 bool structures::ArrayList<T>::full() const {
353     return (size_ == max_size_);
354 }
355
356 template<typename T>
357 bool structures::ArrayList<T>::empty() const {
358     return (size_ == 0);
359 }
360
361 template<typename T>
362 bool structures::ArrayList<T>::contains(const T& data) const {
363     for (int i = 0; i < size_; i++) {
364         if (contents[i] == data) {
365             return true;
366         }
367     }
368     return false;
369 }
370
371 template<typename T>
372 std::size_t structures::ArrayList<T>::find(const T& data) const {
373     for (int i = 0; i < size_; i++) {
374         if (contents[i] == data) {
375             return i;
376         }
377     }
378     return size_;
379 }
380
381 template<typename T>
382 std::size_t structures::ArrayList<T>::size() const {
383     return size_;
384 }
385
386 template<typename T>
387 std::size_t structures::ArrayList<T>::max_size() const {
388     return max_size_;
389 }
390
391 template<typename T>
392 T& structures::ArrayList<T>::at(std::size_t index) {
393     if (empty() || (index < 0 || index >= size_)) {
394         throw std::out_of_range("Index invalido");
395     }
396     return contents[index];
397 }
398
399 template<typename T>
400 T& structures::ArrayList<T>::operator[](std::size_t index) {
401     return contents[index];
402 }
403
404 template<typename T>
405 const T& structures::ArrayList<T>::at(std::size_t index) const {
406     return contents[index];
407 }
408
409 template<typename T>
410 const T& structures::ArrayList<T>::operator[](std::size_t index) const {
411     return contents[index];
412 }
413
414 #endif
415

```