



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

[Painel](#) ▶ [Agrupamentos de Turmas](#) ▶ [INE5408-03208A | INE5609-03238B \(20182\)](#) ▶  
[Tópico 16](#) ▶ [Árvore AVL](#)

[Descrição](#)

[Enviar](#)

[Editar](#)

[Visualizar envios](#)

## Nota

Revisado em quarta, 7 Nov 2018, 16:34 por Atribuição automática de nota

Nota 84 / 100

**Relatório de avaliação**

[\[+\]Summary of tests](#)

Enviado em domingo, 4 Nov 2018, 10:40 ([Baixar](#))

## avl\_tree.h

```
1  //! Copyright 2018 Matheus Henrique Schaly
2
3  #ifndef AVL_TREE_H
4  #define AVL_TREE_H
5
6  #include <stdexcept>
7  #include <algorithm> // std::max
8  #include "array_list.h"
9
10
11 namespace structures {
12
13 //! AVLTree implementation
14 template<typename T>
```

```

14 template<typename T>
15 class AVLTree {
16 public:
17     ///! Destructor
18     ~AVLTree();
19
20     ///! Inserts an element
21     void insert(const T& data);
22
23     ///! Removes an element
24     void remove(const T& data);
25
26     ///! True if tree contains the data, false otherwise
27     bool contains(const T& data) const;
28
29     ///! True if tree is empty, false otherwise
30     bool empty() const;
31
32     ///! Tree's current size
33     std::size_t size() const;
34
35     ///! Orders the elements as middle left right
36     ArrayList<T> pre_order() const;
37
38     ///! Orders the elements as left middle right
39     ArrayList<T> in_order() const;
40
41     ///! Orders the elements as left right middle
42     ArrayList<T> post_order() const;
43
44     ///! Prints the tree
45     void print_tree();
46
47 private:
48     struct Node {
49         T data;
50         std::size_t height{0u};
51         Node* left{nullptr};
52         Node* right{nullptr};
53
54         ///! Constructor
55         explicit Node(const T& data) {
56             this->data = data;
57         }
58
59         ///! Inserts a node
60         Node* insert(const T& data) {
61             if (data < this->data) {
62                 if (left == nullptr) {
63                     left = new Node(data);
64                 } else {
65                     left = left->insert(data);
66                 }
67             } else {
68                 if (right == nullptr) {
69                     right = new Node(data);
70                 } else {
71                     right = right->insert(data);
72                 }
73             }
74
75             updateHeight();
76
77             int bf = this->bf();
78

```

```

18
79         if (bf > 1) { // Left is heavier, rot. right is needed
80             if (this -> left -> bf() >= 0) {
81                 return this -> simpleRight();
82             } else {
83                 return this -> doubleRight();
84             }
85         } else if (bf < -1) { // Right is heavier, rot. left is needed
86             if (this -> right -> bf() <= 0) {
87                 return this -> simpleLeft();
88             } else {
89                 return this -> doubleLeft();
90             }
91         } else {
92             return this;
93         }
94     }
95
96     Node* remove(const T& data) {
97         if ((data > this -> data) && (right != nullptr)) {
98             right = right -> remove(data);
99         } else if ((data < this -> data) && (left != nullptr)) {
100             left = left -> remove(data);
101         } else {
102             if ((right != nullptr) && (left != nullptr)) {
103                 this -> data = right -> find_minimum();
104                 right = right -> remove(data);
105             } else if (right != nullptr) {
106                 this -> data = right -> data;
107                 right = right -> remove(this -> data);
108             } else if (left != nullptr) {
109                 this -> data = left -> data;
110                 left = left -> remove(this -> data);
111             } else {
112                 delete this;
113                 return nullptr;
114             }
115         }
116
117         updateHeight();
118
119         int bf = this -> bf();
120
121         if (bf > 1) { // Left is heavier, rot. right is needed
122             if (this -> left -> bf() >= 0) {
123                 return this -> simpleRight();
124             } else {
125                 return this -> doubleRight();
126             }
127         } else if (bf < -1) { // Right is heavier, rot. left is needed
128             if (this -> right -> bf() <= 0) {
129                 return this -> simpleLeft();
130             } else {
131                 return this -> doubleLeft();
132             }
133         } else {
134             return this;
135         }
136     }
137
138     T find_minimum() {
139         if (left == nullptr) {
140             return this -> data;
141         } else {

```

```

142         return left -> find_minimum();
143     }
144 }
145
146 bool contains(const T& data) const {
147     if (data == this -> data) {
148         return true;
149     } else {
150         if ((data < this -> data) && (left != nullptr)) {
151             return left -> contains(data);
152         } else if (right != nullptr) {
153             return right -> contains(data);
154         } else {
155             return false;
156         }
157     }
158 }
159
160 ///! Updates the node's height
161 void updateHeight() {
162     if ((left != nullptr) && (right != nullptr)) {
163         height = std::max(left -> height, right -> height) + 1;
164     } else if (left != nullptr) {
165         height = left -> height + 1;
166     } else if (right != nullptr) {
167         height = right -> height + 1;
168     } else {
169         height = 0;
170     }
171 }
172
173 ///! Left rotation (RR rotation)
174 Node* simpleLeft() {
175     Node *temp = right;
176     right = right -> left;
177     temp -> left = this;
178     temp -> left -> updateHeight();
179     if (temp -> right != nullptr) {
180         temp -> right -> updateHeight();
181     }
182     temp -> left -> updateHeight();
183     temp -> updateHeight();
184     return temp;
185 }
186
187 ///! Right rotation (LL rotation)
188 Node* simpleRight() {
189     Node *temp = left;
190     left = left -> right;
191     temp -> right = this;
192     if (temp -> left != nullptr) {
193         temp -> left -> updateHeight();
194     }
195     temp -> right -> updateHeight();
196     temp -> updateHeight();
197     return temp;
198 }
199
200 ///! Big left rotation (RL rotation)
201 Node* doubleLeft() {
202     right = right -> simpleRight();
203     return this -> simpleLeft();
204 }
205

```

```

206     ///! Big right rotation (LR rotation)
207     Node* doubleRight() {
208         left = left -> simpleLeft();
209         return this -> simpleRight();
210     }
211
212     void pre_order(ArrayList<T>& v) const {
213         v.push_back(data);
214         if (left != nullptr) {
215             left -> pre_order(v);
216         }
217         if (right != nullptr) {
218             right -> pre_order(v);
219         }
220     }
221
222     void in_order(ArrayList<T>& v) const {
223         if (left != nullptr) {
224             left -> in_order(v);
225         }
226         v.push_back(data);
227         if (right != nullptr) {
228             right -> in_order(v);
229         }
230     }
231
232     void post_order(ArrayList<T>& v) const {
233         if (left != nullptr) {
234             left -> post_order(v);
235         }
236         if (right != nullptr) {
237             right -> post_order(v);
238         }
239         v.push_back(data);
240     }
241
242     int bf() {
243         if ((left != nullptr) && (right != nullptr)) {
244             return (left -> height + 1) - (right -> height + 1);
245         } else if (left != nullptr) {
246             return left -> height + 1;
247         } else if (right != nullptr) {
248             return -(right -> height + 1);
249         } else {
250             return 0;
251         }
252     }
253
254     void print_tree() {
255         std::cout << data << " ";
256         if (left != nullptr) {
257             left -> print_tree();
258         }
259         if (right != nullptr) {
260             right -> print_tree();
261         }
262     }
263 };
264
265 Node* root{nullptr};
266 std::size_t size_{0u};
267 };
268
269 } // namespace structures

```

```

270
271 ///! Destructor
272 template <typename T>
273 structures::AVLTree<T>::~~AVLTree() {
274     delete root;
275     size_ = 0u;
276 }
277
278 ///! Inserts an element
279 template <typename T>
280 void structures::AVLTree<T>::insert(const T& data) {
281     if (empty()) {
282         root = new Node(data);
283         size_++;
284     } else if (!contains(data)) {
285         root = root -> insert(data);
286         size_++;
287     }
288     // print_tree();
289 }
290
291 ///! Removes an element
292 template <typename T>
293 void structures::AVLTree<T>::remove(const T& data) {
294     std::cout << "Data: " << data << std::endl;
295     print_tree();
296     if (empty() || !contains(data)) {
297         return;
298     } else {
299         root = root -> remove(data);
300         size_--;
301     }
302     print_tree();
303 }
304
305
306 ///! True if tree contains the data, false otherwise
307 template <typename T>
308 bool structures::AVLTree<T>::contains(const T& data) const {
309     if (empty()) {
310         return false;
311     } else {
312         return root -> contains(data);
313     }
314 }
315
316 ///! True is tree is empty, false otherwise
317 template <typename T>
318 bool structures::AVLTree<T>::empty() const {
319     return size_ == 0;
320 }
321
322 ///! Tree's current size
323 template <typename T>
324 std::size_t structures::AVLTree<T>::size() const {
325     return size_;
326 }
327
328 ///! Orders the elements as middle left right
329 template <typename T>
330 structures::ArrayList<T> structures::AVLTree<T>::pre_order() const {
331     structures::ArrayList<T> v{};
332     if (!empty()) {
333         root -> pre_order(v);

```



```

334     }
335     return v;
336 }
337
338 ///! Orders the elements as left middle right
339 template <typename T>
340 structures::ArrayList<T> structures::AVLTree<T>::in_order() const {
341     structures::ArrayList<T> v{};
342     if (!empty()) {
343         root -> in_order(v);
344     }
345     return v;
346 }
347
348 ///! Orders the elements as left right middle
349 template <typename T>
350 structures::ArrayList<T> structures::AVLTree<T>::post_order() const {
351     structures::ArrayList<T> v{};
352     if (!empty()) {
353         root -> post_order(v);
354     }
355     return v;
356 }
357
358 template <typename T>
359 void structures::AVLTree<T>::print_tree() {
360     std::cout << "Tree size: " << size_ << std::endl;
361     if (!empty()) {
362         root -> print_tree();
363     }
364     std::cout << std::endl << std::endl;
365 }
366
367 #endif
368

```

## array\_list.h

```

1 ///! Copyright 2018 Matheus Henrique Schaly
2
3 #ifndef STRUCTURES_ARRAY_LIST_H
4 #define STRUCTURES_ARRAY_LIST_H
5
6 #include <cstdint>
7 #include <stdexcept>
8
9
10 namespace structures {
11
12 ///! Static List
13 template<typename T>
14 class ArrayList {
15 public:
16 ///! Constructor
17 ArrayList();
18
19 ///! Constructor with parameter
20 explicit ArrayList(std::size_t max_size);
21
22 ///! Destructor
23 ~ArrayList();

```

```

24
25 ///! Clears the list
26 void clear();
27
28 ///! Pushes an element to the back of the list
29 void push_back(const T& data);
30
31 ///! Pushes an element to the front of the list
32 void push_front(const T& data);
33
34 ///! Inserts an element at a specific index
35 void insert(const T& data, std::size_t index);
36
37 ///! Inserts an element in a sorted position
38 void insert_sorted(const T& data);
39
40 ///! Removes an element from a specific index
41 T pop(std::size_t index);
42
43 ///! Removes an element from the back of the list
44 T pop_back();
45
46 ///! Removes an element from the front of the list
47 T pop_front();
48
49 ///! Removes the first element containing the data
50 void remove(const T& data);
51
52 ///! Verifies if the list is full
53 bool full() const;
54
55 ///! Verifies if the list is empty
56 bool empty() const;
57
58 ///! Verifies if the list contains the data
59 bool contains(const T& data) const;
60
61 ///! Returns the index of the first element containing the data, else return size
62 std::size_t find(const T& data) const;
63
64 ///! Returns the current size of the list
65 std::size_t size() const;
66
67 ///! Returns the maximum size of the list
68 std::size_t max_size() const;
69
70 ///! Returns the element at index
71 T& at(std::size_t index);
72
73 ///! Overloads the [] operator
74 T& operator[](std::size_t index);
75
76 ///! Returns the element at index as constant
77 const T& at(std::size_t index) const;
78
79 ///! Overloads the [] operator, but returns it as a constant
80 const T& operator[](std::size_t index) const;
81
82 private:
83 T* contents;
84 std::size_t size_;
85 std::size_t max_size_;
86 static const auto DEFAULT_MAX = 10u;
87 };

```



```

88 } // namespace structures
89
90 template<typename T>
91 structures::ArrayList<T>::ArrayList() {
92     contents = new T[DEFAULT_MAX];
93     size_ = 0;
94 }
95
96 template<typename T>
97 structures::ArrayList<T>::ArrayList(std::size_t max_size) {
98     size_ = 0;
99     max_size_ = max_size;
100     contents = new T[max_size_];
101 }
102
103 template<typename T>
104 structures::ArrayList<T>::~~ArrayList() {
105     delete[] contents;
106 }
107
108 template<typename T>
109 void structures::ArrayList<T>::clear() {
110     size_ = 0;
111 }
112
113 template<typename T>
114 void structures::ArrayList<T>::push_back(const T& data) {
115     if (full()) {
116         throw std::out_of_range("A lista esta cheia.");
117     } else {
118         contents[size_] = data;
119         size_++;
120     }
121 }
122
123 template<typename T>
124 void structures::ArrayList<T>::push_front(const T& data) {
125     if (full()) {
126         throw std::out_of_range("A lista esta cheia.");
127     } else {
128         for (int i = 0; i < size_; i++) {
129             contents[size_ - i] = contents[size_ - i - 1];
130         }
131         size_++;
132         contents[0] = data;
133     }
134 }
135
136 template<typename T>
137 void structures::ArrayList<T>::insert(const T& data, std::size_t index) {
138     if (full() || (index < 0 || index >= size_)) {
139         throw std::out_of_range("A lista esta cheia.");
140     } else {
141         if (index == 0) {
142             push_front(data);
143             return;
144         }
145         if (index == size_) {
146             push_back(data);
147             return;
148         }
149         for (int i = 0; i < size_ - index; i++) {
150             contents[size_ - i] = contents[size_ - i - 1];
151         }

```

```

152     size_++;
153     contents[index] = data;
154 }
155 }
156
157 template<typename T>
158 void structures::ArrayList<T>::insert_sorted(const T& data) {
159     if (full()) {
160         throw std::out_of_range("A lista esta cheia.");
161     } else {
162         for (int i = 0; i < size_; i++) {
163             if (contents[i] >= data) {
164                 insert(data, i);
165                 return;
166             }
167         }
168         push_back(data);
169     }
170 }
171
172 template<typename T>
173 T structures::ArrayList<T>::pop(std::size_t index) {
174     if (empty() || (index < 0 || index >= size_)) {
175         throw std::out_of_range("A lista esta vazia.");
176     } else {
177         T removed_element = contents[index];
178         for (int i = index; i < size_ - 1; i++) {
179             contents[i] = contents[i + 1];
180         }
181         size_--;
182         return removed_element;
183     }
184 }
185
186 template<typename T>
187 T structures::ArrayList<T>::pop_back() {
188     if (empty()) {
189         throw std::out_of_range("A lista esta vazia");
190     } else {
191         size_--;
192         return contents[size_];
193     }
194 }
195
196 template<typename T>
197 T structures::ArrayList<T>::pop_front() {
198     if (empty()) {
199         throw std::out_of_range("A lista esta vazia");
200     } else {
201         T removed_element = contents[0];
202         for (int i = 0; i < size_ - 1; i++) {
203             contents[i] = contents[i + 1];
204         }
205         size_--;
206         return removed_element;
207     }
208 }
209
210 template<typename T>
211 void structures::ArrayList<T>::remove(const T& data) {
212     if (empty()) {
213         throw std::out_of_range("A lista esta vazia");
214     } else {
215         for (int i = 0; i < size_; i++) {

```

```

216         if (contents[i] == data) {
217             pop(i);
218         }
219     }
220 }
221 }
222
223 template<typename T>
224 bool structures::ArrayList<T>::full() const {
225     return (size_ == max_size_);
226 }
227
228 template<typename T>
229 bool structures::ArrayList<T>::empty() const {
230     return (size_ == 0);
231 }
232
233 template<typename T>
234 bool structures::ArrayList<T>::contains(const T& data) const {
235     for (int i = 0; i < size_; i++) {
236         if (contents[i] == data) {
237             return true;
238         }
239     }
240     return false;
241 }
242
243 template<typename T>
244 std::size_t structures::ArrayList<T>::find(const T& data) const {
245     for (int i = 0; i < size_; i++) {
246         if (contents[i] == data) {
247             return i;
248         }
249     }
250     return size_;
251 }
252
253 template<typename T>
254 std::size_t structures::ArrayList<T>::size() const {
255     return size_;
256 }
257
258 template<typename T>
259 std::size_t structures::ArrayList<T>::max_size() const {
260     return max_size_;
261 }
262
263 template<typename T>
264 T& structures::ArrayList<T>::at(std::size_t index) {
265     if (empty() || (index < 0 || index >= size_)) {
266         throw std::out_of_range("Index invalido");
267     }
268     return contents[index];
269 }
270
271 template<typename T>
272 T& structures::ArrayList<T>::operator[](std::size_t index) {
273     return contents[index];
274 }
275
276 template<typename T>
277 const T& structures::ArrayList<T>::at(std::size_t index) const {
278     return contents[index];
279 }

```

```

280
281 template<typename T>
282 const T& structures::ArrayList<T>::operator[](std::size_t index) const {
283     return contents[index];
284 }
285
286 #endif
287


```

VPL 3.1.5

## 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](#)
    - ▶ [Tópico 12](#)
    - ▶ [Tópico 13](#)
    - ▶ [Tópico 14](#)
    - ▶ [Prova Teórica I](#)
    - ▶ [Prova Prática I](#)
    - ▶ [Tópico 15](#)
    - ▼ [Tópico 16](#)
      - ▼  [Árvore AVL](#)
        - [Descrição](#)
        - [Enviar](#)
        - [Editar](#)

- [Visualizar envios](#)

-  [Testes \(Árvore AVL\)](#)

- ▶ [Tópico 20](#)
- ▶ [Tópico 21](#)
- ▶ [Tópico 22](#)
- ▶ [Tópico 23](#)
- ▶ [Tópico 24](#)
- ▶ [Tópico 25](#)
- ▶ [Tópico 26](#)
- ▶ [Tópico 27](#)
- ▶ [Prova Prática II](#)
- ▶ [Tópico 29](#)
- ▶ [Prova Teórica II](#)

- ▶ [Meus cursos](#)

## ADMINISTRAÇÃO



- ▶ [Administração do curso](#)

---

Você acessou como [Matheus Henrique Schaly \(18200436\) \(Sair\)](#)  
[INE5408-03208A](#) | [INE5609-03238B \(20182\)](#)