

## **Moodle UFSC**



Matheus Henrique Schaly (18200436)



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

### - Estruturas de Dados

```
Painel ► Agrupamentos de Turmas ► INE5408-03208A | INE5609-03238B (20182) ► Tópico 14 ► Arvore Binária de Busca com Percursos
```

Descrição

Enviar

Editar

Visualizar envios

### Nota

Revisado em sexta, 12 Out 2018, 20:09 por Atribuição automática de nota

Nota 100 / 100

Relatório de avaliação

[+]Summary of tests

Enviado em sexta, 12 Out 2018, 20:09 (Baixar)

### binary\_tree.h

```
1 //! Copyright 2018 Matheus Henrique Schaly
2
3 #ifndef BINARY_TREE_H
4 #define BINARY_TREE_H
5
6 #include <stdexcept>
7 #include "array_list.h"
8
9
10 namespace structures {
11
12 //! BinaryTree implementation
13 template<typename T>
14
15
16
17
18
19
19
10 respect to the state of th
```

```
14 CLUSS DinaryTree {
15 public:
16
      //! Constructor
17
       BinaryTree() = default;
18
       //! Destructor
19
20
      virtual ~BinaryTree() {
           delete root;
21
22
           size_ = 0u;
23
24
25
      //! Inserts an element
       void insert(const T& data) {
26
27
           if (empty()) {
               root = new Node(data);
28
29
           } else {
               root -> insert(data);
30
31
32
           size_++;
33
       }
34
35
       //! Remvoes an element
36
       void remove(const T& data) {
           if (!empty()) {
37
38
               bool removed;
39
               removed = root -> remove(data);
               if (removed) {
40
41
                   size_--;
42
43
           }
44
       }
45
46
       //! True if tree contains the data, false otherwise
47
       bool contains(const T& data) const {
48
           if (!empty()) {
               return root -> contains(data);
49
50
           } else {
51
               return false;
52
           }
53
54
55
       //! True is tree is empty, false otherwise
56
       bool empty() const {
57
           return size_ == 0;
58
59
60
       //! Tree's current size
61
       std::size_t size() const {
62
           return size_;
63
64
       //! Orders the elements as middle left right
65
       ArrayList<T> pre_order() const {
66
67
           structures::ArrayList<T> v{};
68
           if (!empty()) {
69
               root -> pre_order(v);
70
           }
71
           return v;
72
       }
73
74
       //! Orders the elements as left middle right
75
       ArrayList<T> in_order() const {
76
           structures::ArrayList<T> v{};
77
           if (!empty()) {
```

```
root -> in_order(v);
 78
 79
 80
            return v;
 81
 82
 83
        //! Orders the elements as left right middle
        ArrayList<T> post_order() const {
            structures::ArrayList<T> v{};
 85
 86
            if (!empty()) {
 87
                root -> post_order(v);
 88
 89
            return v;
 90
 91
 92 private:
        struct Node {
 94
            explicit Node(const T& data):
                data{data},
 95
 96
                left{nullptr},
 97
                right{nullptr}
 98
            {}
 99
100
            T data;
            Node* left;
101
            Node* right;
102
103
104
            void insert(const T& data_) {
                Node *new_node;
105
106
                if (data_ < data) {</pre>
                     if (left == nullptr) {
107
108
                         new_node = new Node(data_);
109
                         left = new_node;
110
                     } else {
111
                         left -> insert(data_);
112
113
                } else {
114
                     if (right == nullptr) {
115
                         new_node = new Node(data_);
116
                         right = new_node;
117
                     } else {
118
                         right -> insert(data_);
119
                     }
120
                }
            }
121
122
            bool remove(const T& data_) {
123
124
                 if (data_ == data) {
125
                     if ((left != nullptr) && (right != nullptr)) {
126
                         Node *temp = right;
127
                         while (temp -> left != nullptr) {
128
                             temp = temp -> left;
129
130
                         data = temp -> data;
131
                         return right -> remove(data);
132
                     } else {
133
                         if (right != nullptr) {
134
                             data = right -> data;
135
                             return right -> remove(data);
136
                         } else {
137
                             if (left != nullptr) {
138
                                 data = left -> data;
139
                                 return left -> remove(data);
140
                             } else {
141
                                 delete this;
```

```
142
                                 return true;
143
                             }
144
                         }
145
146
                 } else {
147
                     if (right != nullptr && data < data_) {</pre>
148
                         return right -> remove(data_);
149
                     } else if (left != nullptr && data > data_) {
150
                         return left -> remove(data_);
151
152
                 }
153
                 return false;
154
155
156
            bool contains(const T& data_) const {
157
                 if (data_ == data) {
158
                     return true;
159
                 } else {
                     if ((left != nullptr) && (data_ < data)) {</pre>
160
161
                         return left -> contains(data_);
                     } else if ((right != nullptr) && (data_ > data)) {
162
163
                         return right -> contains(data_);
164
                     }
165
                 }
166
                 return false;
167
168
169
            void pre_order(ArrayList<T>& v) const {
170
                 v.push_back(data);
171
                 if (left != nullptr) {
172
                     left -> pre_order(v);
173
174
                 if (right != nullptr) {
175
                     right -> pre_order(v);
176
177
            }
178
179
            void in_order(ArrayList<T>& v) const {
                 if (left != nullptr) {
180
181
                     left -> in_order(v);
182
183
                 v.push_back(data);
184
                 if (right != nullptr) {
185
                     right -> in_order(v);
186
187
            }
188
189
            void post_order(ArrayList<T>& v) const {
190
                 if (left != nullptr) {
191
                     left -> post_order(v);
192
                 if (right != nullptr) {
193
194
                     right -> post_order(v);
195
196
                 v.push_back(data);
197
198
        };
199
200
        Node* root{nullptr};
        std::size_t size_{0u};
201
202 };
203
204 } // namespace structures
205
```

### array\_list.h

```
1 //! Copyright 2018 Matheus Henrique Schaly
 3 #ifndef STRUCTURES_ARRAY_LIST_H
 4 #define STRUCTURES_ARRAY_LIST_H
 6 #include <cstdint>
 7 #include <stdexcept>
 8
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);
22 //! Destructor
23 ~ArrayList();
24
25 //! Clears the list
26 void clear();
28 //! Pushes an element to the back of the list
29 void push_back(const T& data);
31 //! Pushes an element to the front of the list
32 void push_front(const T& data);
33
34 //! Inserts an element at a specfic index
35 void insert(const T& data, std::size_t index);
37 //! Inserts an element in a sorted possition
38 void insert_sorted(const T& data);
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();
46 //! Removes an element from the front of the list
47 T pop_front();
49 //! Removes the first element containing the data
50 void remove(const T& data);
51
52 //! Verifies if the list if full
53 bool full() const;
55 //! Verifies if the list is empty
56 bool empty() const;
```

```
58 //! Verifies if the list contains the data
59 bool contains(const T& data) const;
 61 //! Returns the index of the first element containg the data, else return size
62 std::size_t find(const T& data) const;
 64 //! Returns the current size of the list
 65 std::size_t size() const;
 67 //! Returns the maximum size of the list
 68 std::size_t max_size() const;
 70 //! Returns the element at index
 71 T& at(std::size_t index);
 73 //! Overloads the [] operator
 74 T& operator[](std::size_t index);
 76 //! Returns the element at index as constant
 77 const T& at(std::size_t index) const;
 79 //! Overloads the [] operator, but returns it as a constant
 80 const T& operator[](std::size_t index) const;
 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() {
      contents = new T[DEFAULT_MAX];
 93
       size_{=} = 0;
 94 }
95
 96 template<typename T>
 97 structures::ArrayList<T>::ArrayList(std::size_t max_size) {
       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 {
           contents[size_] = data;
118
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 {
            for (int i = 0; i < size_; i++) {
128
129
                contents[size_ - i] = contents[size_ - i - 1];
130
131
            size_++;
            contents[0] = data;
132
133
134 }
135
136 template<typename T>
137 void structures::ArrayList<T>::insert(const T& data, std::size t index) {
        if (full() || (index < 0 || index >= size_)) {
138
            throw std::out_of_range("A lista esta cheia.");
139
140
        } else {
            if (index == 0) {
141
142
                push_front(data);
143
                return;
144
            if (index == size_) {
145
                push_back(data);
146
147
                return;
148
149
            for (int i = 0; i < size_ - index; i++) {
                contents[size_ - i] = contents[size_ - i - 1];
150
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) {
        if (empty() || (index < 0 || index >= size_)) {
174
175
            throw std::out_of_range("A lista esta vazia.");
176
        } else {
            T removed_element = contents[index];
177
178
            for (int i = index; i < size_ - 1; i++) {
179
                contents[i] = contents[i + 1];
180
            size_--;
181
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()) {
            throw std::out_of_range("A lista esta vazia");
213
214
        } else {
           for (int i = 0; i < size_; i++) {
215
216
                if (contents[i] == data) {
217
                    pop(i);
218
219
            }
220
        }
221 }
222
223 template<typename T>
224 bool structures::ArrayList<T>::full() const {
       return (size_ == max_size_);
226 }
227
228 template<typename T>
229 bool structures::ArrayList<T>::empty() const {
        return (size_ == 0);
230
231 }
232
233 template<typename T>
234 bool structures::ArrayList<T>::contains(const T& data) const {
       for (int i = 0; i < size_; i++) {
235
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_)) {
           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 {
      return contents[index];
278
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

# 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 Por que não há Sistemas Operacionais com Coletor de Lixo? Provideo Aula sobre Árvores Binárias de Busca Animação de Árvore de Busca Binária Arvore Binária de Busca com Percursos Descrição Enviar Editar Visualizar envios Testes (Árvore binária de busca) Lecture 11 Prova Teórica I Prova Prática I

Tópico 15
Tópico 16
Tópico 20
Tópico 21
Tópico 22
Tópico 23
Tópico 24
Tópico 25
Tópico 26
Tópico 27

Prova Prática II

Prova Teórica II

Tópico 29

Meus cursos

### ADMINISTRAÇÃO

Administração do curso

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