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

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>
```

```
class BinaryTree {
public:
    //! Constructor
    BinaryTree() = default;

    //! Destructor
    virtual ~BinaryTree() {
        delete root;
        size_ = 0u;
    }

    //! Inserts an element
    void insert(const T& data) {
        if (empty()) {
            root = new Node(data);
        } else {
            root -> insert(data);
        }
        size_++;
    }

    //! Remvoes an element
    void remove(const T& data) {
        if (!empty()) {
            bool removed;
            removed = root -> remove(data);
            if (removed) {
                size_--;
            }
        }
    }

    //! True if tree contains the data, false otherwise
    bool contains(const T& data) const {
        if (!empty()) {
            return root -> contains(data);
        } else {
            return false;
        }
    }

    //! True is tree is empty, false otherwise
    bool empty() const {
        return size_ == 0;
    }

    //! Tree's current size
    std::size_t size() const {
        return size_;
    }

    //! Orders the elements as middle left right
    ArrayList<T> pre_order() const {
        structures::ArrayList<T> v{};
        if (!empty()) {
            root -> pre_order(v);
        }
        return v;
    }

    //! Orders the elements as left middle right
    ArrayList<T> in_order() const {
        structures::ArrayList<T> v{};
        if (!empty()) {
```

```
            root -> in_order(v);
        }
        return v;
    }

    //! Orders the elements as left right middle
    ArrayList<T> post_order() const {
        structures::ArrayList<T> v{};
        if (!empty()) {
            root -> post_order(v);
        }
        return v;
    }

private:
    struct Node {
        explicit Node(const T& data):
            data{data},
            left{nullptr},
            right{nullptr}
        {}

        T data;
        Node* left;
        Node* right;

        void insert(const T& data_) {
            Node *new_node;
            if (data_ < data) {
                if (left == nullptr) {
                    new_node = new Node(data_);
                    left = new_node;
                } else {
                    left -> insert(data_);
                }
            } else {
                if (right == nullptr) {
                    new_node = new Node(data_);
                    right = new_node;
                } else {
                    right -> insert(data_);
                }
            }
        }

        bool remove(const T& data_) {
            if (data_ == data) {
                if ((left != nullptr) && (right != nullptr)) {
                    Node *temp = right;
                    while (temp -> left != nullptr) {
                        temp = temp -> left;
                    }
                    data = temp -> data;
                    return right -> remove(data);
                } else {
                    if (right != nullptr) {
                        data = right -> data;
                        return right -> remove(data);
                    } else {
                        if (left != nullptr) {
                            data = left -> data;
                            return left -> remove(data);
                        } else {
                            delete this;
```

```
                            return true;
                        }
                    }
                }
            } else {
                if (right != nullptr && data < data_) {
                    return right -> remove(data_);
                } else if (left != nullptr && data > data_) {
                    return left -> remove(data_);
                }
            }
            return false;
        }

        bool contains(const T& data_) const {
            if (data_ == data) {
                return true;
            } else {
                if ((left != nullptr) && (data_ < data)) {
                    return left -> contains(data_);
                } else if ((right != nullptr) && (data_ > data)) {
                    return right -> contains(data_);
                }
            }
            return false;
        }

        void pre_order(ArrayList<T>& v) const {
            v.push_back(data);
            if (left != nullptr) {
                left -> pre_order(v);
            }
            if (right != nullptr) {
                right -> pre_order(v);
            }
        }

        void in_order(ArrayList<T>& v) const {
            if (left != nullptr) {
                left -> in_order(v);
            }
            v.push_back(data);
            if (right != nullptr) {
                right -> in_order(v);
            }
        }

        void post_order(ArrayList<T>& v) const {
            if (left != nullptr) {
                left -> post_order(v);
            }
            if (right != nullptr) {
                right -> post_order(v);
            }
            v.push_back(data);
        }
    };

    Node* root{nullptr};
    std::size_t size_{0u};
};

}  // namespace structures

```

```
206 #endif
207
```

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 specfic index
35 void insert(const T& data, std::size_t index);
36
37 //! Inserts an element in a sorted possition
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 if 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 containg 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     }
```

```cpp
}

template<typename T>
void structures::ArrayList<T>::push_front(const T& data) {
    if (full()) {
        throw std::out_of_range("A lista esta cheia.");
    } else {
        for (int i = 0; i < size_; i++) {
            contents[size_ - i] = contents[size_ - i - 1];
        }
        size_++;
        contents[0] = data;
    }
}

template<typename T>
void structures::ArrayList<T>::insert(const T& data, std::size_t index) {
    if (full() || (index < 0 || index >= size_)) {
        throw std::out_of_range("A lista esta cheia.");
    } else {
        if (index == 0) {
            push_front(data);
            return;
        }
        if (index == size_) {
            push_back(data);
            return;
        }
        for (int i = 0; i < size_ - index; i++) {
            contents[size_ - i] = contents[size_ - i - 1];
        }
        size_++;
        contents[index] = data;
    }
}

template<typename T>
void structures::ArrayList<T>::insert_sorted(const T& data) {
    if (full()) {
        throw std::out_of_range("A lista esta cheia.");
    } else {
        for (int i = 0; i < size_; i++) {
            if (contents[i] >= data) {
                insert(data, i);
                return;
            }
        }
        push_back(data);
    }
}

template<typename T>
T structures::ArrayList<T>::pop(std::size_t index) {
    if (empty() || (index < 0 || index >= size_)) {
        throw std::out_of_range("A lista esta vazia.");
    } else {
        T removed_element = contents[index];
        for (int i = index; i < size_ - 1; i++) {
            contents[i] = contents[i + 1];
        }
        size_--;
        return removed_element;
    }
}
```

```cpp

template<typename T>
T structures::ArrayList<T>::pop_back() {
    if (empty()) {
        throw std::out_of_range("A lista esta vazia");
    } else {
        size_--;
        return contents[size_];
    }
}

template<typename T>
T structures::ArrayList<T>::pop_front() {
    if (empty()) {
        throw std::out_of_range("A lista esta vazia");
    } else {
        T removed_element = contents[0];
        for (int i = 0; i < size_ - 1; i++) {
            contents[i] = contents[i + 1];
        }
        size_--;
        return removed_element;
    }
}

template<typename T>
void structures::ArrayList<T>::remove(const T& data) {
    if (empty()) {
        throw std::out_of_range("A lista esta vazia");
    } else {
        for (int i = 0; i < size_; i++) {
            if (contents[i] == data) {
                pop(i);
            }
        }
    }
}

template<typename T>
bool structures::ArrayList<T>::full() const {
    return (size_ == max_size_);
}

template<typename T>
bool structures::ArrayList<T>::empty() const {
    return (size_ == 0);
}

template<typename T>
bool structures::ArrayList<T>::contains(const T& data) const {
    for (int i = 0; i < size_; i++) {
        if (contents[i] == data) {
            return true;
        }
    }
    return false;
}

template<typename T>
std::size_t structures::ArrayList<T>::find(const T& data) const {
    for (int i = 0; i < size_; i++) {
        if (contents[i] == data) {
            return i;
        }
```

```
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

- 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?
  - Video 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
- Tópico 29
- Prova Teórica II

Meus cursos

ADMINISTRAÇÃO

- Administração do curso

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