UNIVERSIDADE FEDERAL
DE SANTA CATARINA

Moodle UFSC

Matheus Henrique Schaly (18200436)

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

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

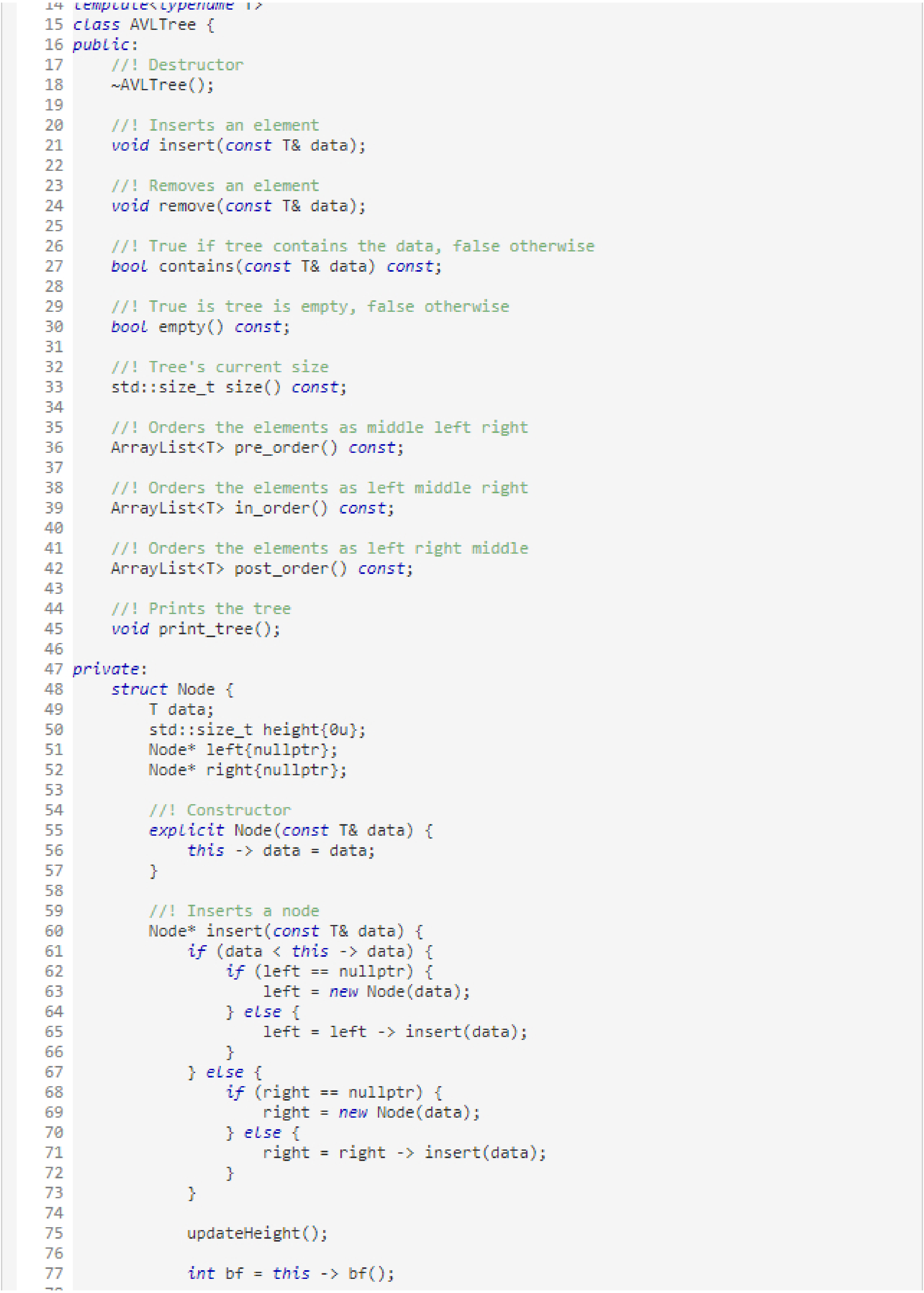
Descrição Enviar Editar Visualizar envios

Nota

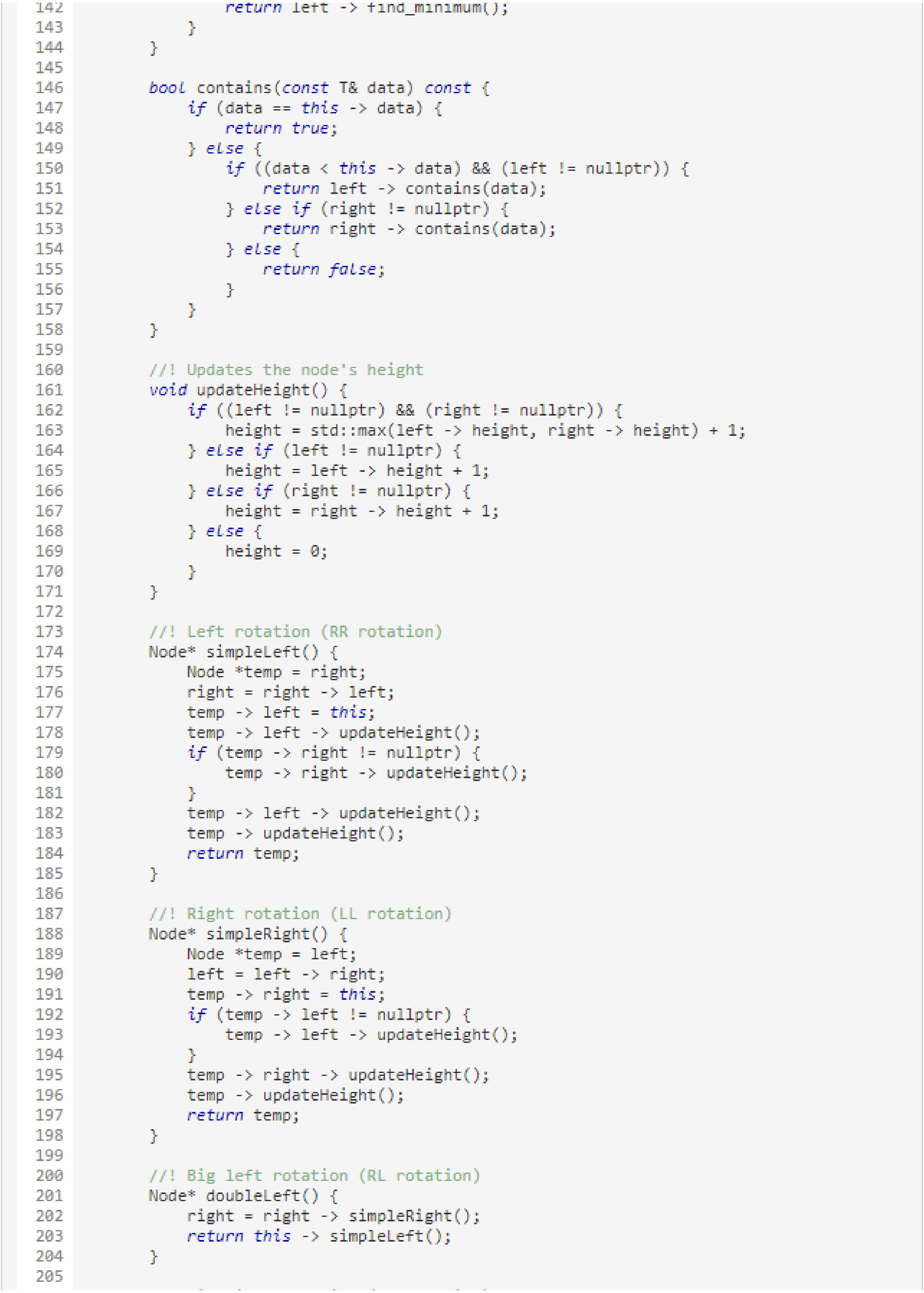
Revisado em quarta, 7 Nov 2018, 16:34 por Atribuição automática de nota
Nota 84 / 100
Relatório de avaliação
[+]**Summary of tests**
Enviado em domingo, 4 Nov 2018, 10:40 (Baixar)

avl_tree.h

```
//! Copyright 2018 Matheus Henrique Schaly

#ifndef AVL_TREE_H
#define AVL_TREE_H

#include <stdexcept>
#include <algorithm>  // std::max
#include "array_list.h"


namespace structures {

//! AVLTree implementation
template<typename T>
```



```cpp
template<typename T>
class AVLTree {
public:
    //! Destructor
    ~AVLTree();

    //! Inserts an element
    void insert(const T& data);

    //! Removes an element
    void remove(const T& data);

    //! True if tree contains the data, false otherwise
    bool contains(const T& data) const;

    //! True is tree is empty, false otherwise
    bool empty() const;

    //! Tree's current size
    std::size_t size() const;

    //! Orders the elements as middle left right
    ArrayList<T> pre_order() const;

    //! Orders the elements as left middle right
    ArrayList<T> in_order() const;

    //! Orders the elements as left right middle
    ArrayList<T> post_order() const;

    //! Prints the tree
    void print_tree();

private:
    struct Node {
        T data;
        std::size_t height{0u};
        Node* left{nullptr};
        Node* right{nullptr};

        //! Constructor
        explicit Node(const T& data) {
            this -> data = data;
        }

        //! Inserts a node
        Node* insert(const T& data) {
            if (data < this -> data) {
                if (left == nullptr) {
                    left = new Node(data);
                } else {
                    left = left -> insert(data);
                }
            } else {
                if (right == nullptr) {
                    right = new Node(data);
                } else {
                    right = right -> insert(data);
                }
            }

            updateHeight();

            int bf = this -> bf();
            if (bf > 1) {  // Left is heavier, rot. right is needed
                if (this -> left -> bf() >= 0) {
                    return this -> simpleRight();
                } else {
                    return this -> doubleRight();
                }
            } else if (bf < -1) {  // Right is heavier, rot. left is needed
                if (this -> right -> bf() <= 0) {
                    return this -> simpleLeft();
                } else {
                    return this -> doubleLeft();
                }
            } else {
                return this;
            }
        }

        Node* remove(const T& data) {
            if ((data > this -> data) && (right != nullptr)) {
                right = right -> remove(data);
            } else if ((data < this -> data) && (left != nullptr)) {
                left = left -> remove(data);
            } else {
                if ((right != nullptr) && (left != nullptr)) {
                    this -> data = right -> find_minimum();
                    right = right -> remove(data);
                } else if (right != nullptr) {
                    this -> data = right -> data;
                    right = right -> remove(this -> data);
                } else if (left != nullptr) {
                    this -> data = left -> data;
                    left = left -> remove(this -> data);
                } else {
                    delete this;
                    return nullptr;
                }
            }

            updateHeight();

            int bf = this -> bf();

            if (bf > 1) {  // Left is heavier, rot. right is needed
                if (this -> left -> bf() >= 0) {
                    return this -> simpleRight();
                } else {
                    return this -> doubleRight();
                }
            } else if (bf < -1) {  // Right is heavier, rot. left is needed
                if (this -> right -> bf() <= 0) {
                    return this -> simpleLeft();
                } else {
                    return this -> doubleLeft();
                }
            } else {
                return this;
            }
        }

        T find_minimum() {
            if (left == nullptr) {
                return this -> data;
            } else {
```



```
                return left -> find_minimum();
            }
        }

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

        //! Updates the node's height
        void updateHeight() {
            if ((left != nullptr) && (right != nullptr)) {
                height = std::max(left -> height, right -> height) + 1;
            } else if (left != nullptr) {
                height = left -> height + 1;
            } else if (right != nullptr) {
                height = right -> height + 1;
            } else {
                height = 0;
            }
        }

        //! Left rotation (RR rotation)
        Node* simpleLeft() {
            Node *temp = right;
            right = right -> left;
            temp -> left = this;
            temp -> left -> updateHeight();
            if (temp -> right != nullptr) {
                temp -> right -> updateHeight();
            }
            temp -> left -> updateHeight();
            temp -> updateHeight();
            return temp;
        }

        //! Right rotation (LL rotation)
        Node* simpleRight() {
            Node *temp = left;
            left = left -> right;
            temp -> right = this;
            if (temp -> left != nullptr) {
                temp -> left -> updateHeight();
            }
            temp -> right -> updateHeight();
            temp -> updateHeight();
            return temp;
        }

        //! Big left rotation (RL rotation)
        Node* doubleLeft() {
            right = right -> simpleRight();
            return this -> simpleLeft();
        }

        //! Big right rotation (LR rotation)
        Node* doubleRight() {
            left = left -> simpleLeft();
            return this -> simpleRight();
        }

        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);
        }

        int bf() {
            if ((left != nullptr) && (right != nullptr)) {
                return (left -> height + 1) - (right -> height + 1);
            } else if (left != nullptr) {
                return left -> height + 1;
            } else if (right != nullptr) {
                return -(right -> height + 1);
            } else {
                return 0;
            }
        }

        void print_tree() {
            std::cout << data << "  ";
            if (left != nullptr) {
                left -> print_tree();
            }
            if (right != nullptr) {
                right -> print_tree();
            }
        }
    };

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

}  // namespace structures
```

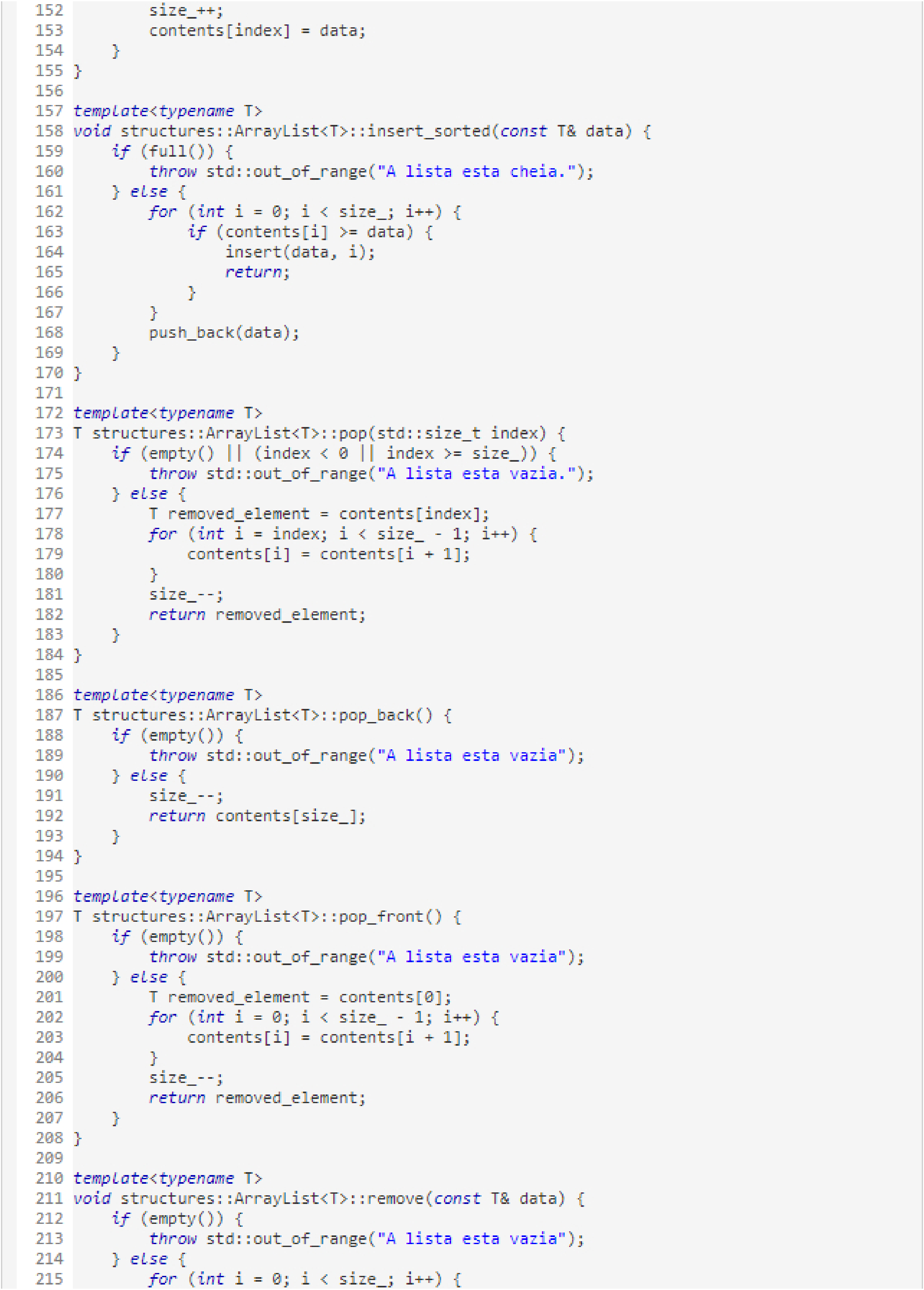
```cpp

//! Destructor
template <typename T>
structures::AVLTree<T>::~AVLTree() {
    delete root;
    size_ = 0u;
}

//! Inserts an element
template <typename T>
void structures::AVLTree<T>::insert(const T& data) {
    if (empty()) {
        root = new Node(data);
        size_++;
    } else if (!contains(data)) {
        root = root -> insert(data);
        size_++;
    }
    // print_tree();
}

//! Removes an element
template <typename T>
void structures::AVLTree<T>::remove(const T& data) {
    std::cout << "Data: " << data << std::endl;
    print_tree();
    if (empty() || !contains(data)) {
        return;
    } else {
        root = root -> remove(data);
        size_--;
    }
    print_tree();
}


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

//! True is tree is empty, false otherwise
template <typename T>
bool structures::AVLTree<T>::empty() const {
    return size_ == 0;
}

//! Tree's current size
template <typename T>
std::size_t structures::AVLTree<T>::size() const {
    return size_;
}

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

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

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

template <typename T>
void structures::AVLTree<T>::print_tree() {
    std::cout << "Tree size: " << size_ << std::endl;
    if (!empty()) {
        root -> print_tree();
    }
    std::cout << std::endl << std::endl;;
}

#endif

```

array_list.h

```
//! Copyright 2018 Matheus Henrique Schaly

#ifndef STRUCTURES_ARRAY_LIST_H
#define STRUCTURES_ARRAY_LIST_H

#include <cstdint>
#include <stdexcept>


namespace structures {

//! Static List
template<typename T>
class ArrayList {
 public:
//! Constructor
ArrayList();

//! Constructor with parameter
explicit ArrayList(std::size_t max_size);

//! Destructor
~ArrayList();
```

```cpp

//! Clears the list
void clear();

//! Pushes an element to the back of the list
void push_back(const T& data);

//! Pushes an element to the front of the list
void push_front(const T& data);

//! Inserts an element at a specfic index
void insert(const T& data, std::size_t index);

//! Inserts an element in a sorted possition
void insert_sorted(const T& data);

//! Removes an element from a specific index
T pop(std::size_t index);

//! Removes an element from the back of the list
T pop_back();

//! Removes an element from the front of the list
T pop_front();

//! Removes the first element containing the data
void remove(const T& data);

//! Verifies if the list if full
bool full() const;

//! Verifies if the list is empty
bool empty() const;

//! Verifies if the list contains the data
bool contains(const T& data) const;

//! Returns the index of the first element containg the data, else return size
std::size_t find(const T& data) const;

//! Returns the current size of the list
std::size_t size() const;

//! Returns the maximum size of the list
std::size_t max_size() const;

//! Returns the element at index
T& at(std::size_t index);

//! Overloads the [] operator
T& operator[](std::size_t index);

//! Returns the element at index as constant
const T& at(std::size_t index) const;

//! Overloads the [] operator, but returns it as a constant
const T& operator[](std::size_t index) const;

 private:
T* contents;
std::size_t size_;
std::size_t max_size_;
static const auto DEFAULT_MAX = 10u;
};
```

```cpp
}  // namespace structures

template<typename T>
structures::ArrayList<T>::ArrayList() {
    contents = new T[DEFAULT_MAX];
    size_ = 0;
}

template<typename T>
structures::ArrayList<T>::ArrayList(std::size_t max_size) {
    size_ = 0;
    max_size_ = max_size;
    contents = new T[max_size_];
}

template<typename T>
structures::ArrayList<T>::~ArrayList() {
    delete[] contents;
}

template<typename T>
void structures::ArrayList<T>::clear() {
    size_ = 0;
}

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

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];
        }
```



```cpp
        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;
    }
}

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;
        }
    }
    return size_;
}

template<typename T>
std::size_t structures::ArrayList<T>::size() const {
    return size_;
}

template<typename T>
std::size_t structures::ArrayList<T>::max_size() const {
    return max_size_;
}

template<typename T>
T& structures::ArrayList<T>::at(std::size_t index) {
    if (empty() || (index < 0 || index >= size_)) {
        throw std::out_of_range("Index invalido");
    }
    return contents[index];
}

template<typename T>
T& structures::ArrayList<T>::operator[](std::size_t index) {
    return contents[index];
}

template<typename T>
const T& structures::ArrayList<T>::at(std::size_t index) const {
    return contents[index];
}
```

```
template<typename T>
const T& structures::ArrayList<T>::operator[](std::size_t index) const {
    return contents[index];
}

#endif

```

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)