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

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
      - Implementação de Lista Circular Simples
      - Testes (Lista Circular)
      - Implementação de Lista Circular Dupla
        - Descrição
        - Enviar
        - Editar
        - Visualizar envios
      - Testes (Lista Circular Duplamente Encadeada)
      - Video Aula sobre Listas Circulares
      - Lecture 9
    - Tópico 12
    - Tópico 13
    - Tópico 14
    - Prova Teorica I
- Meus cursos

ADMINISTRAÇÃO
- Administração do curso

Descrição | Enviar | Editar | Visualizar envios

Nota

Revisado em sábado, 29 Set 2018, 22:18 por Atribuição automática de nota
Nota 100 / 100
Relatório de avaliação
[+]**Summary of tests**
Enviado em sábado, 29 Set 2018, 22:18 (Baixar)

doubly_circular_list.h

```cpp
//! Copyright 2018 Matheus Henrique Schaly

#ifndef STRUCTURES_CIRCULAR_LIST_H
#define STRUCTURES_CIRCULAR_LIST_H

#include <cstdint>
#include <stdexcept>

namespace structures {

//! Dynamic Doubly Circular Linked List
template<typename T>
class DoublyCircularList {
 public:
    //! Constructor
    DoublyCircularList();
    //! Destructor
    ~DoublyCircularList();
    //! Removes list's elements
    void clear();
    //! Inserts an element at the list's rightmost part
    void push_back(const T& data);
    //! Inserts an element at the list's leftmost part
    void push_front(const T& data);
    //! Inserts an element at the given index
    void insert(const T& data, std::size_t index);
    //! Inserts an element sorted by data
    void insert_sorted(const T& data);
    //! Returns the element's data at index (checks limits)
    T& at(std::size_t index);
    //! Returns the constant element's data at index (checks limits)
    const T& at(std::size_t index) const;
    //! Removes an element from index
    T pop(std::size_t index);
    //! Removes an element from the rightmost part
    T pop_back();
    //! Removes an element from the leftmost part
    T pop_front();
    //! Removes an element with the given data
    void remove(const T& data);
    //! Returns true if the list is empty and false otherwise
    bool empty() const;
    //! Checks if the list contains the node with the given data
    bool contains(const T& data) const;
    //! Returns the index of the given data
    std::size_t find(const T& data) const;
    //! Returns the current size of the list
    std::size_t size() const;

 private:
    class Node {  // Elemento
     public:
        //! Constructor with 1 parameter
        explicit Node(const T& data):
            data_{data}
        {}

        //! Constructor with 2 parameters
        Node(const T& data, Node* next):
            data_{data},
            next_{next}
        {}

        //! Constructor with 2 parameters
        Node(const T& data, Node* next, Node* prev):
            data_{data},
            next_{next},
            prev_{prev}
        {}

        //! Data's getter
        T& data() {
            return data_;
        }

        //! Data's constant getter
        const T& data() const {
            return data_;
        }

        //! Prev's constant
        Node *prev() {
            return prev_;
        }


        //! Prev's constant getter
        const Node* prev() const {
            return prev_;
        }

        //! Prev's setter
        void prev(Node* node) {
            prev_ = node;
        }


        //! Next's getter
        Node* next() {
            return next_;
        }

        //! Next's constant getter
        const Node* next() const {
            return next_;
        }

        //! Next's setter
        void next(Node* node) {
            next_ = node;
        }

     private:
        //! Node's data
        T data_;
```

```
        //! Node's next node
        Node* next_{nullptr};

        //! Node's previous node
        Node* prev_{nullptr};
    };

    //! Returns the list's last node
    Node* end() {  // último nodo da lista
        auto it = head;
        for (auto i = 1u; i < size(); ++i) {
            it = it->next();
        }
        return it;
    }

    //! List's leftmost node
    Node* head{nullptr};

    //! List's rightmost node
    Node* tail{nullptr};

    //! List's current size
    std::size_t size_{0u};
};

}  // namespace structures

//! Constructor
template<typename T>
structures::DoublyCircularList<T>::DoublyCircularList() {}

//! Destructor
template<typename T>
structures::DoublyCircularList<T>::~DoublyCircularList() {
    clear();
}

//! Removes list's elements
template<typename T>
void structures::DoublyCircularList<T>::clear() {
    while (!empty()) {
        pop_front();
    }
}

//! Inserts an element at the list's rightmost part
template<typename T>
void structures::DoublyCircularList<T>::push_back(const T& data) {
    insert(data, size_);
}

//! Inserts an element at the list's leftmost part
template<typename T>
void structures::DoublyCircularList<T>::push_front(const T& data) {
    Node* node = new Node(data, head);
    if (node == nullptr) {
        throw std::out_of_range("A lista esta cheia.");
    }
    head = node;
    size_++;
}

//! Inserts an element at the given index
template<typename T>
void structures::DoublyCircularList<T>::insert(const T& data,
                                               std::size_t index) {
    if (index > size_ || index < 0) {
        throw std::out_of_range("Indice invalido");
    }
    if (index == 0) {
        push_front(data);
    } else {
        Node* node = new Node(data);
        Node* previous_node = head;
        if (node == nullptr) {
            throw std::out_of_range("A lista esta cheia.");
        }
        std::size_t i = 1;
        while (i < index) {
            previous_node = previous_node -> next();
            i++;
        }
        node -> next(previous_node -> next());
        previous_node -> next(node);
        size_++;
        if (index == size_) {
            node -> next(head);  // Fastest way to build circular list?
            head -> prev(node);  // Fastest way to build circular list?
        }
    }
}

//! Inserts an element sorted by data
template<typename T>
void structures::DoublyCircularList<T>::insert_sorted(const T& data) {
    if (head == nullptr) {
        push_front(data);
    } else {
        Node* previous_node = head;
        std::size_t i = 0;
        while (previous_node != nullptr && data > previous_node -> data()) {
            previous_node = previous_node -> next();
            i++;
        }
        try {
            insert(data, i);
        } catch (std::out_of_range error) {
            throw error;
        }
    }
}

//! Returns an element's data at index
template<typename T>
T& structures::DoublyCircularList<T>::at(std::size_t index) {
    if (index >= size_ || index < 0) {
        throw std::out_of_range("Indice invalido.");
    }
    Node* node = head;
    std::size_t i = 1;
    while (i <= index) {
        node = node -> next();
        i++;
    }
    return node -> data();
}

//! Removes an element from index
template<typename T>
T structures::DoublyCircularList<T>::pop(std::size_t index) {
    if (empty() || index >= size_ || index < 0) {
        throw std::out_of_range("Indice invalido.");
    }
    if (index == 0) {
        return pop_front();
    } else {
        Node* node;
        Node* previous_node = head;
        std::size_t i = 1;
        while (i < index) {
            previous_node = previous_node -> next();
            i++;
        }
        node = previous_node -> next();
        previous_node -> next(node -> next());
        T deleted_data = node -> data();
        delete node;
        size_--;
        if (index == size_) {
```

```
            previous_node -> next(head);  // Fastest way to build circular list?
            head -> prev(previous_node);  // Fastest way to build circular list?
        }
        return deleted_data;
    }
}

//! Removes an element from the rightmost part of the list
template<typename T>
T structures::DoublyCircularList<T>::pop_back() {
    try {
        return pop(size_ - 1);
    } catch (std::out_of_range error) {
        throw error;
    }
}

//! Removes an element from the leftmost part of the list
template<typename T>
T structures::DoublyCircularList<T>::pop_front() {
    if (empty()) {
        throw std::out_of_range("A lista esta vazia");
    }
    Node* node = head;
    T deleted_data = node -> data();
    head = node -> next();
    delete node;
    size_--;
    return deleted_data;
}

//! Removes an element with the given data
template<typename T>
void structures::DoublyCircularList<T>::remove(const T& data) {
    pop(find(data));
}

//! Returns true if list is empty and false otherwise
template<typename T>
bool structures::DoublyCircularList<T>::empty() const {
    return size_ == 0;
}

//! Checks if the list contais the node with the given data
template<typename T>
bool structures::DoublyCircularList<T>::contains(const T& data) const {
    if (empty()) {
        throw std::out_of_range("A lista esta vazia.");
    } else {
        Node* node = head;
        std::size_t i = 1;
        while (i < size_) {
            if (node -> data() == data) {
                return true;
            }
            node = node -> next();
            i++;
        }
        return false;
    }
}

//! Returns the index of the given data
template<typename T>
std::size_t structures::DoublyCircularList<T>::find(const T& data) const {
    if (empty()) {
        throw std::out_of_range("A lista esta vazia.");
    } else {
        Node* node = head;
        std::size_t i = 0;
        while (i < size_ - 1) {
            if (node -> data() == data) {
                return i;
            }
            node = node -> next();
            i++;
        }
        if (node -> data() == data) {
            return size_ - 1;
        }
    }
    return size_;
}

//! Returns the current size of the list
template<typename T>
std::size_t structures::DoublyCircularList<T>::size() const {
    return size_;
}

#endif

```