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
        - Descrição
        - Enviar
        - Editar
        - Visualizar envios
      - Testes (Lista Circular)
      - Implementação de Lista Circular Dupla
      - 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, 22:10 por Atribuição automática de nota
Nota 100 / 100
Relatório de avaliação
[+]**Summary of tests**
Enviado em sábado, 29 Set 2018, 22:10 (Baixar)

circular_list.h

```
1  //! Copyright 2018 Matheus Henrique Schaly
2
3  #ifndef STRUCTURES_CIRCULAR_LIST_H
4  #define STRUCTURES_CIRCULAR_LIST_H
5
6  #include <cstdint>
7  #include <stdexcept>
8
9  namespace structures {
10
11 //! Dynamic Simple Circular Linked List
12 template<typename T>
13 class CircularList {
14  public:
15     //! Constructor
16     CircularList();
17     //! Destructor
18     ~CircularList();
19     //! Removes list's elements
20     void clear();
21     //! Inserts an element at the list's rightmost part
22     void push_back(const T& data);
23     //! Inserts an element at the list's leftmost part
24     void push_front(const T& data);
25     //! Inserts an element at the given index
26     void insert(const T& data, std::size_t index);
27     //! Inserts an element sorted by data
28     void insert_sorted(const T& data);
29     //! Returns the element's data at index (checks limits)
30     T& at(std::size_t index);
31     //! Returns the constant element's data at index (checks limits)
32     const T& at(std::size_t index) const;
33     //! Removes an element from index
34     T pop(std::size_t index);
35     //! Removes an element from the rightmost part
36     T pop_back();
37     //! Removes an element from the leftmost part
38     T pop_front();
39     //! Removes an element with the given data
40     void remove(const T& data);
41     //! Returns true if the list is empty and false otherwise
42     bool empty() const;
43     //! Checks if the list contains the node with the given data
44     bool contains(const T& data) const;
45     //! Returns the index of the given data
46     std::size_t find(const T& data) const;
47     //! Returns the current size of the list
48     std::size_t size() const;
49
50 private:
51     class Node {  // Elemento
52      public:
53         //! Constructor with 1 parameter
54         explicit Node(const T& data):
55             data_{data}
56         {}
57
58         //! Constructor with 2 parameters
59         Node(const T& data, Node* next):
60             data_{data},
61             next_{next}
62         {}
63
64         //! Data's getter
65         T& data() {
66             return data_;
67         }
68
69         //! Data's constant getter
70         const T& data() const {
71             return data_;
72         }
73
74         //! Next's getter
75         Node* next() {
76             return next_;
77         }
78
79         //! Next's constant getter
80         const Node* next() const {
81             return next_;
82         }
83
84         //! Next's setter
85         void next(Node* node) {
86             next_ = node;
87         }
88
89      private:
90         //! Node's data
91         T data_;
92
93         //! Nodes next node
94         Node* next_{nullptr};
95     };
96
97     //! Returns the list's last node
98     Node* end() {  // último nodo da lista
99         auto it = head;
100        for (auto i = 1u; i < size(); ++i) {
101            it = it->next();
102        }
103        return it;
104    }
105
106    //! List's leftmost node
107    Node* head{nullptr};
108
109    //! List's current size
110    std::size_t size_{0u};
111 };
112
113 }  // namespace structures
114
115 //! Constructor
116 template<typename T>
```

```cpp
structures::CircularList<T>::CircularList() {}

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

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

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

//! Inserts an element at the list's leftmost part
template<typename T>
void structures::CircularList<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::CircularList<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?
        }
    }
}

//! Inserts an element sorted by data
template<typename T>
void structures::CircularList<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::CircularList<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::CircularList<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?
        }
        return deleted_data;
    }
}

//! Removes an element from the rightmost part of the list
template<typename T>
T structures::CircularList<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::CircularList<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::CircularList<T>::remove(const T& data) {
    pop(find(data));
}

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

//! Checks if the list contais the node with the given data
template<typename T>
bool structures::CircularList<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::CircularList<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::CircularList<T>::size() const {
    return size_;
}

#endif

```