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

Descrição | Enviar | Editar | Visualizar envios

Nota

Revisado em domingo, 23 Set 2018, 14:04 por Atribuição automática de nota
Nota 100 / 100
Relatório de avaliação
[+]**Summary of tests**
Enviado em domingo, 23 Set 2018, 14:03 (Baixar)

doubly_linked_list.h

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

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

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

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

//! Inserts an element at the list's leftmost part
template<typename T>
void structures::DoublyLinkedList<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;
    if (size_ == 0) {
        tail = node;
    }
    size_++;
}

//! Inserts an element at the given index
template<typename T>
void structures::DoublyLinkedList<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.");
        }
        if (index == size_) {
            node -> next(tail);
            tail -> next(node);
            tail = node;
        } else {
            std::size_t i = 1;
            while (i < index) {
                previous_node = previous_node -> next();
                i++;
            }
            node -> next(previous_node -> next());
            node -> prev(previous_node);
            previous_node -> next() -> prev(node);
            previous_node -> next(node);
        }
        size_++;
    }
}

//! Inserts an element sorted by data
template<typename T>
void structures::DoublyLinkedList<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::DoublyLinkedList<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::DoublyLinkedList<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++;
        }
        if (index == size_) {
            node = tail;
            tail = node -> prev();
        } else {
            node = previous_node -> next();
```

```
            previous_node -> next(node -> next());
            // node -> next() -> prev(previous_node);
        }
        T deleted_data = node -> data();
        delete node;
        size_--;
        return deleted_data;
    }
}

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

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

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

#endif

```

VPL 3.1.5