|  |  |
| --- | --- |
| **Российский университет транспорта (МИИТ) Институт транспортной техники и систем управления Кафедра «Управление и защита информации»** | |
| **Отчет по практическому заданию по теме «Структуры данных» по дисциплине «Системы управления базами данных»** | |
|  | Выполнил:  Студенты группы ТКИ-441  Бабич И.С.  Кирюшин Н.А. |
|  | Проверил:  Доцент кафедры УиЗИ к.т.н., доцент  Васильева М. А. |
| Москва 2023 | |

**Оглавление**

[1. Условие задачи 3](#_Toc154330301)

[2. Текст программы на языке С++ 4](#_Toc154330302)

[2.1. DL.cpp 4](#_Toc154330306)

[2.2. DoubleList.h 4](#_Toc154330307)

[2.3. Test.cpp 10](#_Toc154330308)

[3. Результат работы программы 14](#_Toc154330309)

[4. UML диаграмма классов 15](#_Toc154330310)

1. Условие задачи

**ДВУСВЯЗНЫЙ ЦИКЛИЧЕСКИЙ СПИСОК**

Определить класс «DoubleList», включающий следующую информацию:

* Данные любого типа: Элементы списка могут содержать данные любого типа, например, числа, строки, объекты и т.д.
* Указатели на узлы: Каждый элемент списка содержит указатели на предыдущий и следующий узлы, образуя двухсвязный циклический список.
* Size: Метод, позволяющий получить размер списка.
* PushF, PushB: Методы, позволяющие добавить элементы в начало или конец списка (соответственно).
* Get: Метод, позволяющий получить данные о местонахождении элемента в списке.
* GetF, GetB: Методы, позволяющие информацию о первом или последнем элементе списка (соответственно).
* PopF, PopB: Методы, позволяющие удалить первый или последний элемент списка (соответственно).
* Remove: Метод позволяющий удалить узел списка по ключу (место элемента в списке).
* Add: Метод, позволяющий добавить элемент в список по ключу.
* Clean: Метод, позволяющий удалить все элементы списка.

1. Текст программы на языке С++

4. 1. DL.cpp

#include "DoubleList.h"

using namespace std;

/\*

\* @brief Точка входа в программу.

\*/

int main()

{

DoubleList <int> DL1;

DL1.PushB(15);

DL1.PushB(23);

DL1.Add(67, 1);

cout << DL1;

return 0;

}

* 1. DoubleList.h

#pragma once

#include <iostream>

using namespace std;

/\*

\* @brief Структура для реализации класса DoubleList, в которой хранится информации об элементе, а также ссылки на предыдущий и следующий элементы.

\*/

template <typename T>

struct Node

{

Node\* prev;

Node\* next;

T data;

Node() : data(), prev(nullptr), next(nullptr) {};

};

/\*

\* @brief Класс DoubleList - двусвязный циклический список.

\*/

template <typename T>

class DoubleList

{

private:

Node<T>\* head;

Node<T>\* tail;

size\_t size;

public:

/\*

\* @breif Конструктор по умолчанию.

\*/

DoubleList();

/\*

\* @brief Метод, позволяющий получить размер списка.

\* @return Размер списка.

\*/

size\_t Size() const;

/\*

\* @brief Метод, позволяющий добавить элемент в конец списка.

\* @param NewData - информация о новом элементе

\*/

void PushB(T NewData);

/\*

\* @brief Метод, позволяющий добавить элемент в начало списка.

\* @param NewData - информация о новом элементе

\*/

void PushF(T NewData);

/\*

\* @brief Метод, позволяющий получить данные об элементе списка.

\* @param item - определение места элемента в списке.

\* @return Информация об элементе = его размер.

\*/

T Get(size\_t item) const;

/\*

\* @brief Метод, позволяющий получить информацию о первом элементе списка.

\* @return Информация о первом элементе.

\*/

T GetF() const;

/\*

\* @brief Метод, позволяющий получить информацию о последнем элементе списка.

\* @return Информация о последнем элементе.

\*/

T GetB() const;

/\*

\* @brief Метод, удаляющий первый элемент списка.

\*/

void PopF();

/\*

\* @brief Метод, удаляющий последний элемент списка.

\*/

void PopB();

/\*

\* @brief Метод, удаляющий элемент списка по ключу.

\* @param item - ключ (место элемента в списке).

\*/

void Remove(size\_t item);

/\*

\* @brief Метод, позволяющий вставить элемент в список.

\* @param data - информация об элементе (его размер).

\* @param item - место, на которое надо поставить элемент.

\*/

void Add(T data, size\_t item);

/\*

\* @brief Метод, удаляющий все элементы списка.

\*/

void Clean();

/\*

\* @brief Конструктор копирования.

\*/

DoubleList(const DoubleList& another);

/\*

\* @brief Оператор копирования.

\*/

DoubleList& operator = (const DoubleList& another);

/\*

\* @brief Конструктор перемещения.

\*/

DoubleList(DoubleList&& another) noexcept;

/\*

\* @brief Оператор перемещения

\*/

DoubleList& operator = (DoubleList&& another) noexcept;

/\*

\* @brief Деструктор

\*/

~DoubleList();

};

template <typename T>

DoubleList<T>::DoubleList() : head(nullptr), tail(nullptr), size(0) {};

template <typename T>

DoubleList<T>::DoubleList(const DoubleList& another) : DoubleList()

{

Node<T>\* temp = another.head;

for (size\_t i = 0; i < another.Size(); i++)

{

this->PushB(temp->data);

temp = temp->next;

}

this->size = another.size;

}

template <typename T>

DoubleList<T>& DoubleList<T>::operator = (const DoubleList& another)

{

if (this == &another)

return \*this;

DoubleList<T> copy(another);

std::swap(this->tail, copy.tail);

std::swap(this->head, copy.head);

std::swap(this->size, copy.size);

return \*this;

}

template <typename T>

size\_t DoubleList<T>::Size() const

{

return this->size;

}

template <typename T>

void DoubleList<T>::PushB(T NewData)

{

Node<T>\* temp = new Node<T>();

temp->data = NewData;

if (this->Size() == 0)

{

temp->prev = temp;

temp->next = temp;

this->head = temp;

this->tail = temp;

}

else if (this->Size() == 1)

{

temp->prev = this->head;

temp->next = this->head;

this->tail = temp;

this->head->next = temp;

this->head->prev = temp;

}

else

{

temp->prev = this->tail;

temp->next = this->head;

tail->next = temp;

this->tail = temp;

this->head->prev = this->tail;

}

++this->size;

}

template <typename T>

void DoubleList<T>::PushF(T NewData)

{

Node<T>\* temp = new Node<T>();

temp->data = NewData;

if (this->Size() == 0)

{

temp->next = temp;

temp->prev = temp;

this->head = temp;

this->tail = temp;

}

else if (this->Size() == 1)

{

temp->prev = this->tail;

temp->next = this->tail;

this->head = temp;

this->tail->next = temp;

this->tail->prev = temp;

}

else

{

temp->next = this->head;

temp->prev = this->tail;

head->prev = temp;

this->head = temp;

this->tail->next = this->head;

}

++this->size;

}

template <typename T>

T DoubleList<T>::GetF() const

{

return this->head->data;

}

template <typename T>

T DoubleList<T>::GetB() const

{

return this->tail->data;

}

template <typename T>

void DoubleList<T>::PopB()

{

if (this->Size() == 0)

throw std::logic\_error("Невозможно удалить элемент, список пуст!");

if (this->Size() == 1)

{

delete this->tail;

this->head = nullptr;

this->tail = nullptr;

--this->size;

return;

}

else if (this->Size() == 2)

{

delete this->tail;

this->head->next = this->tail;

this->tail = this->head;

--this->size;

return;

}

auto temp = this->tail;

this->tail = this->tail->prev;

this->tail->next = this->head;

this->head->prev = this->tail;

delete temp;

--this->size;

}

template <typename T>

void DoubleList<T>::PopF()

{

if (this->Size() == 0)

throw std::logic\_error("Невозможно удалить элемент, список пуст!");

if (this->Size() == 1)

{

delete this->head;

this->head = nullptr;

this->tail = nullptr;

--this->size;

return;

}

else if (this->Size() == 2)

{

delete this->head;

this->tail->prev = nullptr;

this->head = this->tail;

--this->size;

return;

}

auto temp = this->head;

this->head = this->head->next;

this->head->prev = this->tail;

this->tail->next = this->head;

delete temp;

--this->size;

}

template <typename T>

void DoubleList<T>::Remove(size\_t item)

{

if (item >= this->Size())

throw std::out\_of\_range("Позиция больше размера списка");

if (item == 0)

{

this->PopF();

return;

}

else if (item == this->Size() - 1)

{

this->PopB();

return;

}

auto temp = this->head;

for (size\_t i = 0; i < item; i++)

{

temp = temp->next;

}

temp->prev->next = temp->next;

temp->next->prev = temp->prev;

delete temp;

--this->size;

}

template <typename T>

void DoubleList<T>::Add(T data, size\_t item)

{

if (item > this->Size())

throw std::out\_of\_range("Введенное значение больше размера списка");

if (item == 0) {

this->PushF(data);

return;

}

else if (item == this->Size()) {

this->PushB(data);

return;

}

auto add = this->head;

for (size\_t i = 0; i < item; i++)

{

add = add->next;

}

Node<T>\* temp = new Node<T>();

temp->data = data;

if (add->prev != nullptr && this->Size() != 1)

add->prev->next = temp;

temp->next = add;

temp->prev = add->prev;

add->prev = temp;

++this->size;

}

template <typename T>

T DoubleList<T>::Get(size\_t item) const

{

if (item >= this->Size())

throw std::out\_of\_range("Введенное значение больше размера списка");

auto get = this->head;

for (size\_t i = 0; i < item; i++)

{

get = get->next;

}

return get->data;

}

template <typename T>

void DoubleList<T>::Clean()

{

while (this->Size() != 0)

this->PopB();

}

template <typename T>

DoubleList<T>::~DoubleList()

{

this->Clean();

}

template <typename T>

DoubleList<T>::DoubleList(DoubleList&& another) noexcept : DoubleList()

{

\*this = another;

}

template <typename T>

DoubleList<T>& DoubleList<T>::operator = (DoubleList&& another) noexcept

{

std::swap(this->tail, another.tail);

std::swap(this->head, another.head);

std::swap(this->size, another.size);

return \*this;

}

template <typename T>

std::ostream& operator << (std::ostream& os, const DoubleList<T>& DoubleList)

{

for (size\_t i = 0; i < DoubleList.Size(); i++)

{

os << DoubleList.Get(i) << " ";

}

return os;

* 1. Test.cpp

#include "pch.h"

#include "../DL/DoubleList.h"

#include <string>

using namespace std;

TEST(TestString, TestPushB)

{

DoubleList <string> DL2;

DL2.PushB("Hello");

DL2.PushB("World");

EXPECT\_EQ(DL2.Size(), 2);

EXPECT\_EQ(DL2.GetB(), "World");

}

TEST(TestString, TestPushF)

{

DoubleList <string> DL2;

DL2.PushF("World");

DL2.PushF("Hello");

EXPECT\_EQ(DL2.Size(), 2);

EXPECT\_EQ(DL2.GetF(), "Hello");

}

TEST(TestInt, TestPushB)

{

DoubleList <int> DL1;

DL1.PushB(34);

DL1.PushB(57);

EXPECT\_EQ(DL1.Size(), 2);

EXPECT\_EQ(DL1.GetB(), 57);

}

TEST(TestInt, TestPushF)

{

DoubleList <int> DL1;

DL1.PushF(65);

DL1.PushF(17);

EXPECT\_EQ(DL1.Size(), 2);

EXPECT\_EQ(DL1.GetF(), 17);

}

TEST(TestInt, TestGet)

{

DoubleList <int> DL1;

DL1.PushB(10);

DL1.PushB(15);

DL1.PushB(48);

DL1.PushB(-4);

EXPECT\_EQ(DL1.Get(2), 48);

EXPECT\_EQ(DL1.Get(3), -4);

EXPECT\_EQ(DL1.Get(0), 10);

}

TEST(TestString, TestGet)

{

DoubleList <string> DL2;

DL2.PushB("Hello");

DL2.PushB("Perfect");

DL2.PushB("World");

EXPECT\_EQ(DL2.Get(0), "Hello");

}

TEST(TestInt, TestPopFLogicError)

{

DoubleList <int> DL1;

EXPECT\_THROW(DL1.PopF(), logic\_error);

}

TEST(TestString, TestPopFLogicError)

{

DoubleList <string> DL2;

EXPECT\_THROW(DL2.PopF(), logic\_error);

}

TEST(TestInt, TestPopF)

{

DoubleList <int> DL1;

DL1.PushB(23);

DL1.PushB(37);

DL1.PopF();

EXPECT\_EQ(DL1.Size(), 1);

}

TEST(TestString, TestPopF)

{

DoubleList <string> DL2;

DL2.PushB("Dreams");

DL2.PushB("Come");

DL2.PushB("True");

DL2.PopF();

EXPECT\_EQ(DL2.Size(), 2);

}

TEST(TestInt, TestPopBLogicError)

{

DoubleList <int> DL1;

EXPECT\_THROW(DL1.PopB(), logic\_error);

}

TEST(TestString, TestPopBLogicError)

{

DoubleList <string> DL2;

EXPECT\_THROW(DL2.PopB(), logic\_error);

}

TEST(TestInt, TestPopB)

{

DoubleList <int> DL1;

DL1.PushB(45);

DL1.PushB(78);

DL1.PopB();

EXPECT\_EQ(DL1.Size(), 1);

}

TEST(TestString, TestPopB)

{

DoubleList <string> DL2;

DL2.PushB("Dreams");

DL2.PushB("Come");

DL2.PushB("True");

DL2.PopB();

EXPECT\_EQ(DL2.Size(), 2);

}

TEST(TestInt, TestRemove)

{

DoubleList <int> DL1;

DL1.PushB(15);

DL1.PushB(23);

DL1.PushB(67);

DL1.Remove(1);

EXPECT\_EQ(DL1.Size(), 2);

}

TEST(TestString, TestRemove)

{

DoubleList <string> DL2;

DL2.PushB("Dreams");

DL2.PushB("Come");

DL2.PushB("True");

DL2.Remove(0);

EXPECT\_EQ(DL2.Size(), 2);

}

TEST(TestInt, TestAddOutOfRange)

{

DoubleList <int> DL1;

EXPECT\_THROW(DL1.Add(14, 4), out\_of\_range);

}

TEST(TestString, TestAddOutOfRange)

{

DoubleList <string> DL2;

EXPECT\_THROW(DL2.Add("Worry", 4), out\_of\_range);

}

TEST(TestInt, TestAdd)

{

DoubleList <int> DL1;

DL1.PushB(15);

DL1.PushB(23);

DL1.Add(67, 1);

EXPECT\_EQ(DL1.Size(), 3);

}

TEST(TestString, TestAdd)

{

DoubleList <string> DL2;

DL2.PushB("Dreams");

DL2.PushB("True");

DL2.Add("Come", 1);

std::vector<string> a1 = { "Dreams","Come","True"};

for (size\_t i = 0; i < a1.size(); i++)

{

EXPECT\_TRUE(DL2.Get(i).compare(a1[i]) == 0);

}

}

TEST(TestInt, TestClean)

{

DoubleList <int> DL1;

DL1.PushB(15);

DL1.PushB(23);

DL1.PushB(67);

EXPECT\_NO\_FATAL\_FAILURE(DL1.Clean());

EXPECT\_EQ(DL1.Size(), 0);

}

TEST(TestString, TestClean)

{

DoubleList <string> DL2;

DL2.PushB("Dreams");

DL2.PushB("True");

DL2.PushB("Come");

EXPECT\_NO\_FATAL\_FAILURE(DL2.Clean());

EXPECT\_EQ(DL2.Size(), 0);

}

int main(int argc, char\*\* argv)

{

::testing::InitGoogleTest(&argc, argv);

return RUN\_ALL\_TESTS();

}

1. Результат работы программы

|  |
| --- |
|  |
| 1. Результат тестов |

1. UML диаграмма классов

|  |
| --- |
|  |
| 1. UML диаграмма классов |