|  |  |
| --- | --- |
| **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»** | |
|  | Выполнили:  Студенты группы ТКИ-442  Мацнев В. А.  Шуриков Д. А.  Проверил:  Доцент кафедры УиЗи, к.т.н., с.н.с.  Васильева М. А. |
| Москва 2023 | |

Оглавление

[**Задание** 3](#_Toc154609748)

[**1. Текст программы на языке C++** 3](#_Toc154609749)

[**1.1.** **Код файла CMakeLists.txt** 3](#_Toc154609750)

[**1.2.** **Код файла List.cpp** 3](#_Toc154609751)

[**1.3.** **Код файлов List\_test.cpp** 6](#_Toc154609752)

[**1.4.** **Код файлa main.cpp** 8](#_Toc154609753)

[**1.5.** **Код файлa List.h** 9](#_Toc154609754)

[**1.6.** **Код файлa Node.h** 11](#_Toc154609755)

[**2. Результат работы программы** 11](#_Toc154609756)

[**3. UML диаграмма классов** 12](#_Toc154609757)

[**Заключение** 13](#_Toc154609758)

**Задание**

Разработать структуру данных на языке программирования С++ в ООП парадигме. В нашем случае структура данных – это односвязный список, основные операции которого будут вставка элементов, их удаление, доступ к определенным элементам по значению или позиции и обход списка для выполнения операций над каждым элементом.

**1. Текст программы на языке C++**

* 1. **Код файла CMakeLists.txt**

cmake\_minimum\_required(VERSION 3.14)

project(List)

SET(CMAKE\_CXX\_STANDARD 14)

set(CMAKE\_CXX\_FLAGS

"${CMAKE\_CXX\_FLAGS} -Wall -pedantic-errors")

add\_subdirectory(test)

add\_executable(main)

target\_sources(main PRIVATE main.cpp)

* 1. **Код файла List.cpp**

template<typename T>

List<T>::List() : head(nullptr), size(0) {}

template<typename T>

List<T>::List(const List<T>& ll) : head(nullptr), size(0)

{

Node<T>\* head = ll.getHead();

while (head != nullptr) {

push\_back(head->data);

head = head->next;

}

}

template<typename T>

List<T>::List(List<T>&& ll) : List()

{

head = ll.getHead();

ll.head = nullptr;

size = ll.getSize();

ll.size = 0;

}

template<typename T>

List<T>& List<T>::operator=(List<T> const& ll)

{

if (this == &ll)

return \*this;

List<T> tmp(ll);

size = tmp.getSize();

head = tmp.getHead();

tmp.head = nullptr;

return \*this;

}

template<typename T>

List<T>& List<T>::operator=(List<T>&& ll)

{

if (this != &ll)

return \*this;

head = std::move(ll.head);

size = std::move(ll.size);

return \*this;

}

template<typename T>

List<T>::~List()

{

Node<T>\* tmp = nullptr;

while (head)

{

tmp = head;

head = head->next;

delete tmp;

}

head = nullptr;

}

template<typename T>

Node<T>\* List<T>::getHead() const

{

return head;

}

template<typename T>

size\_t List<T>::getSize() const

{

return size;

}

template<typename T>

void List<T>::display(std::ostream& out) const

{

Node<T>\* node = head;

while (node != nullptr)

{

out << node->data << " ";

node = node->next;

}

}

template<typename T>

void List<T>::push\_back(T data)

{

Node<T>\* t = new Node<T>(data);

Node<T>\* tmp = head;

if (tmp == nullptr)

{

head = t;

}

else

{

while (tmp->next != nullptr)

{

tmp = tmp->next;

}

tmp->next = t;

}

size++;

}

template<typename T>

void List<T>::push\_front(T data)

{

Node<T>\* t = new Node<T>(data);

if (head == nullptr)

{

head = t;

}

else

{

t->next = head;

head = t;

}

size++;

}

template<typename T>

void List<T>::push(T data, size\_t index)

{

Node<T>\* cur = head;

Node<T>\* prev = nullptr;

size\_t cur\_index = 0;

while (cur\_index != index && cur != nullptr)

{

prev = cur;

cur = cur->next;

++cur\_index;

}

if (cur\_index == index)

{

Node<T>\* t = new Node<T>(data);

if (prev != nullptr) prev->next = t;

else head = t;

t->next = cur;

size++;

}

}

template<typename T>

Node<T>\* List<T>::get\_by\_index(size\_t index)

{

Node<T>\* cur = head;

size\_t cur\_index = 0;

while (cur\_index != index && cur != nullptr)

{

cur = cur->next;

++cur\_index;

}

if (cur\_index == index)

{

return cur;

}

return nullptr;

}

template<typename T>

void List<T>::pop\_front()

{

Node<T>\* tmp = head;

if (tmp != nullptr)

{

head = head->next;

delete tmp;

size--;

}

}

template<typename T>

void List<T>::pop\_back()

{

Node<T>\* tmp = head;

Node<T>\* prev = nullptr;

if (tmp != nullptr)

{

while (tmp->next != nullptr)

{

prev = tmp;

tmp = tmp->next;

}

if (prev != nullptr)

{

prev->next = nullptr;

}

else

{

head = nullptr;

}

delete tmp;

size--;

}

}

template<typename U>

std::ostream& operator<<(std::ostream& os, const List<U>& ll)

{

ll.display(os);

return os;

}

* 1. **Код файлов List\_test.cpp**

#include <gtest/gtest.h>

#include "../List.h"

TEST(ListTest, DefaultConstructorTest) {

auto ll1 = List<int>();

ASSERT\_EQ(ll1.getSize(), 0);

ASSERT\_EQ(ll1.getHead(), nullptr);

}

TEST(ListTest, ListGrowsAndShrinks) {

List<int> ll1;

ll1.push\_back(5);

ll1.push\_front(6);

ll1.push\_back(7);

ll1.push\_front(8);

ll1.push(9,2);

ASSERT\_EQ(ll1.getSize(), 5);

ll1.pop\_back();

ll1.pop\_front();

ASSERT\_EQ(ll1.getSize(), 3);

}

TEST(ListTest, CopyConstructorTest) {

List<int> ll1;

ll1.push\_back(5);

ll1.push\_front(6);

ll1.push\_back(7);

ll1.push\_front(8);

ll1.push(9,2);

ASSERT\_EQ(ll1.getSize(), 5);

List<int> ll2 = List<int>(ll1);

ASSERT\_EQ(ll1.getHead()->data, ll1.getHead()->data);

ASSERT\_EQ(ll1.getSize(), ll1.getSize());

}

TEST(ListTest, CopyAssignTest) {

List<int> ll1;

ll1.push\_back(5);

ll1.push\_front(6);

ll1.push\_back(7);

ll1.push\_front(8);

ll1.push(9,2);

ASSERT\_EQ(ll1.getSize(), 5);

List<int> ll2 = ll1;

ASSERT\_EQ(ll1.getHead()->data, ll1.getHead()->data);

ASSERT\_EQ(ll1.getSize(), ll1.getSize());

}

TEST(ListTest, MoveConstructorTest) {

List<int> ll1;

ll1.push\_back(5);

ll1.push\_front(6);

ll1.push\_back(7);

ll1.push\_front(8);

ll1.push(9,2);

const auto ll1Size = ll1.getSize();

ASSERT\_EQ(ll1Size, 5);

List<int> ll2 = List<int>(std::move(ll1));

ASSERT\_EQ(ll2.getSize(), ll1Size);

ASSERT\_NE(ll2.getHead(), nullptr);

}

TEST(ListTest, MoveAssignmentTest) {

List<int> ll1;

ll1.push\_back(5);

ll1.push\_front(6);

ll1.push\_back(7);

ll1.push\_front(8);

ll1.push(9,2);

const auto ll1Size = ll1.getSize();

ASSERT\_EQ(ll1Size, 5);

List<int> ll2 = std::move(ll1);

ASSERT\_EQ(ll2.getSize(), ll1Size);

ASSERT\_NE(ll2.getHead(), nullptr);

}

TEST(ListTest, OverloadOutOperator) {

List<int> ll1;

ll1.push\_back(5);

ASSERT\_EQ(ll1.getSize(), 1);

std::stringstream stream;

stream << ll1;

ASSERT\_EQ(stream.str().compare("5"), 1);

}

TEST(ListTest, TestGetByIndex) {

List<int> ll1;

int found1 = 5;

int found2 = 5;

ll1.push\_back(found1);

ll1.push\_back(1);

ll1.push\_back(found2);

ll1.push\_back(3);

ASSERT\_EQ(ll1.getSize(), 4);

ASSERT\_EQ(ll1.get\_by\_index(0)->data, found1);

ASSERT\_EQ(ll1.get\_by\_index(INT32\_MAX), nullptr);

}

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

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

return RUN\_ALL\_TESTS();

}

* 1. **Код файлa main.cpp**

#include <iostream>

#include "List.h"

int main()

{

List<int> ll1;

ll1.push\_back(5);

std::cout << ll1 << '\n';

std::cout << "ll1 Size: " << ll1.getSize() << "\n\n";

ll1.push\_front(6);

std::cout << ll1 << '\n';

std::cout << "ll1 Size: " << ll1.getSize() << "\n\n";

ll1.push\_back(7);

std::cout << ll1 << '\n';

std::cout << "ll1 Size: " << ll1.getSize() << "\n\n";

ll1.push\_front(8);

std::cout << ll1 << '\n';

std::cout << "ll1 Size: " << ll1.getSize() << "\n\n";

ll1.push(9, 2);

std::cout << ll1 << '\n';

std::cout << "ll1 Size: " << ll1.getSize() << "\n\n";

ll1.pop\_back();

std::cout << ll1 << '\n';

std::cout << "ll1 Size: " << ll1.getSize() << "\n\n";

ll1.pop\_front();

std::cout << ll1 << '\n';

std::cout << "ll1 Size: " << ll1.getSize() << "\n\n";

List<int> ll2(ll1);

std::cout << "ll2 " << ll2 << '\n';

List<int> ll3;

ll3 = ll1;

std::cout << "ll3 " << ll3 << '\n';

List<int> ll4 = List<int>(std::move(ll3));

std::cout << "ll4 " << ll4 << "\n";

List<int> ll5 = std::move(ll4);

std::cout << "ll5 " << ll5 << "\n";

std::cout << "ll4 " << ll4 << "\n";

return 0;

}

* 1. **Код файлa List.h**

#include "Node.h"

/\*

\* Односвязный список.

\* @tparam T - тип данных элементов, хранимых в списке.

\*/

template <typename T>

class List

{

private:

Node<T>\* head;

size\_t size;

public:

List();

List(const List<T>& ll);

List(List<T>&& ll);

~List();

List<T>& operator=(List<T> const& ll);

List<T>& operator=(List<T>&& ll);

/\*

\* @brief получение указателя на первый элемент в списке.

\* @return указатель на первый элемент в списке.

\*/

Node<T>\* getHead() const;

/\*

\* @brief получить количество элементов в списке.

\* @return количество элементов в списке.

\*/

size\_t getSize() const;

/\*

\* @brief вывод списка в поток.

\* @param out поток для вывода.

\*/

void display(std::ostream& out = std::cout) const;

/\*

\* @brief добавление элемента в конец списка.

\* @param data информация об элементе.

\*/

void push\_back(T data);

/\*

\* @brief добавление элемента в начало списка.

\* @param data информация об элементе.

\*/

void push\_front(T data);

/\*

\* @brief добавление элемента на заданную позицию.

\* @param data информация об элементе.

\* @param index позиция элемента.

\*/

void push(T data, size\_t index);

/\*

\* @brief удаление первого элемента в списке.

\*/

void pop\_front();

/\*

\* @brief удаление последнего элемента в списке.

\*/

void pop\_back();

/\*

\* @brief получение элемента на заданной позиции.

\* @param index позиция элемента.

\* @return указатель на найденый элемент в списке или nullptr.

\*/

Node<T>\* get\_by\_index(size\_t index);

/\*

\* @brief перегузка оператора вывода в поток.

\*/

template <typename U>

friend std::ostream& operator<<(std::ostream& os, const List<U>& ll);

};

#include "List.cpp"

* 1. **Код файлa Node.h**

template<typename T>

struct Node

{

T data;

Node\* next;

Node(T val) : data(val), next(nullptr) {}

};

**2. Результат работы программы**

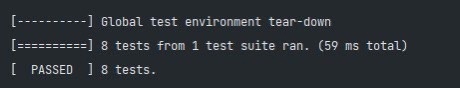


Рисунок 1 – Результат тестов

**3. UML диаграмма классов**

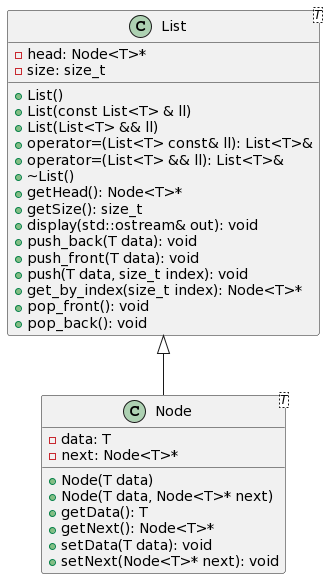


Рисунок 2 – UML диаграмма классов Node и List

**Заключение**

В результате выполнения практического задания была разработана структура данных - односвязный список. Этот односвязный список стал эффективным инструментом для хранения данных в удобной форме. Его гибкость позволяет эффективно добавлять и удалять элементы, а также осуществлять поиск необходимых данных. Разработка такой структуры данных дает возможность улучшить производительность программы или приложения, использующего её, и сделать работу с данными более эффективной. Дальнейшие улучшения могут включать в себя оптимизацию работы с памятью или добавление дополнительных методов для обработки информации в списке.