**МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ**

**ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ**

**УЧРЕЖДЕНИЕ**

**ВЫСШЕГО ПРОФЕССИОНАЛЬНОГО ОБРАЗОВАНИЯ**

**«ЮЖНЫЙ ФЕДЕРАЛЬНЫЙ УНИВЕРСИТЕТ»**

**ИНЖЕНЕРНО-ТЕХНОЛОГИЧЕСКАЯ АКАДЕМИЯ**

**Институт компьютерных технологий и информационной безопасности**

**Кафедра математического обеспечения и применения ЭВМ**

**Отчет по лабораторной работе №1**

**по курсу «Структуры и алгоритмы обработки данных»**

**на тему**

**«Представление связных списков»**

**Вариант №3**

Выполнил: студент

группы КТбо2-1: Самардак А.В

Проверил:

Доцент каф. МОП ЭВМ Дроздов С.Н

**Цель работы**

* Разработать и отладить программу для работы со связными списками;
* Реализация в коде программы одного из видов связных списков, согласно номеру варианта:
* В соответствии с требованиями, реализовать в коде программы минимальный набор действий со списками;
* Реализовать две дополнительные операции, согласно номеру варианта;

**Формулировка задачи**

Для всех вариантов нужно реализовать меню со следующим минимальным наборомопераций со списком:

**-** Инициализация пустого списка

- Уничтожение списка с освобождением памяти

- Добавление узла в голову списка

- Добавление узла в хвост списка

- Удаление узла из головы списка

- Удаление узла из хвоста списка

- Выдача текущего списка на экран

Кроме того, в каждом варианте следует реализовать две дополнительные операции. Двусвязный линейный список. Дополнительные операции: a) Добавить новый узел в указанную позицию; б) Поменять местами первый и последний узлы(требуется поменять именно узлы, а не их значения).

**Листинг программы на языке С++**

**Main.cpp**

#include <vld.h>

#include "Console.h"

int main()

{

setlocale(LC\_ALL, "Russian");

Console().start();

return 0;

}

**Console.h**

#pragma once

#include <iostream>

#include "List.h"

class Console

{

public:

Console() = default;

~Console() = default;

void start();

private:

List \_list;

int dialog\_get\_data();

void dialog\_push\_front();

void dialog\_push\_back();

void dialog\_pop\_front();

void dialog\_pop\_back();

void dialog\_print\_list();

void dialog\_insert();

void dialog\_swap();

void print\_menu() const;

};

**Console.cpp**

#include "Console.h"

void Console::start()

{

bool operation = true;

while (operation)

{

print\_menu();

int check = dialog\_get\_data();

switch (check)

{

case 1:

dialog\_push\_front();

break;

case 2:

dialog\_push\_back();

break;

case 3:

dialog\_pop\_front();

break;

case 4:

dialog\_pop\_back();

break;

case 5:

dialog\_print\_list();

break;

case 6:

dialog\_insert();

break;

case 7:

dialog\_swap();

break;

case 8:

operation = false;

}

}

}

int Console::dialog\_get\_data()

{

int data;

std::cin >> data;

return data;

}

void Console::dialog\_push\_front()

{

std::cout << "Введите число для добавления в голову: ";

\_list.push\_front(dialog\_get\_data());

}

void Console::dialog\_push\_back()

{

std::cout << "Введите число для добавления в хвост: ";

\_list.push\_back(dialog\_get\_data());

}

void Console::dialog\_pop\_front()

{

\_list.pop\_front();

}

void Console::dialog\_pop\_back()

{

\_list.pop\_back();

}

void Console::dialog\_print\_list()

{

for (auto itr = \_list.begin(); itr != \_list.end(); itr++)

{

std::cout << " | " << \*itr << " | " << std::endl;

}

}

void Console::dialog\_insert()

{

std::cout << "Введите место куда добавить число: ";

int count = dialog\_get\_data();

std::cout << "Введите число для добавления: ";

int data = dialog\_get\_data();

try

{

auto it = \_list.begin();

for (int i = 0; i < count - 1; i++)

{

it++;

}

\_list.insert(it, data);

}

catch (const std::exception& ex)

{

std::cout << ex.what() << std::endl;

}

}

void Console::dialog\_swap()

{

\_list.swap();

}

void Console::print\_menu() const

{

std::cout << "-----------------------------" << std::endl;

std::cout << "Что вы хотите сделать?:" << std::endl;

std::cout << "1. Добавить узел в голову списка" << std::endl;

std::cout << "2. Добавить узел в хвост списка" << std::endl;

std::cout << "3. Удалить узел из головы списка" << std::endl;

std::cout << "4. Удалить узел из хвоста списка" << std::endl;

std::cout << "5. Вывод спика" << std::endl;

std::cout << "6. Добавить новый узел в указанную позицию" << std::endl;

std::cout << "7. Поменять местами первый и последний узлы" << std::endl;

std::cout << "8. Выход" << std::endl;

std::cout << "-----------------------------" << std::endl;

}

**List.h**

#pragma once

#include <iostream>

class List {

private:

struct Node;

class iterator;

public:

List() = default;

~List();

void push\_front(int data);

void push\_back(int data);

void pop\_front();

void pop\_back();

void insert(const iterator& itr, int data);

void swap();

class iterator

{

public:

iterator() = default;

iterator(Node\* node);

iterator(const iterator& itr);

~iterator() = default;

const iterator& operator=(const iterator& itr);

int operator\*() const;

const iterator& operator++();

const iterator& operator++(int);

bool operator==(const iterator& itr) const;

bool operator!=(const iterator& itr) const;

private:

friend class List;

Node\* \_currentElement = nullptr;

};

iterator begin();

iterator end();

private:

void create\_first\_node(int data);

struct Node {

int \_data;

Node\* \_next = nullptr;

Node\* \_prev = nullptr;

};

int \_count = 0;

Node\* \_default\_node = new Node();

Node\* \_head = \_default\_node;

Node\* \_tail = nullptr;

};

**List.cpp**

#include "List.h"

List::~List()

{

Node\* currentElement = \_head;

for (int i = 0; i < \_count; i++)

{

currentElement = currentElement->\_next;

delete currentElement->\_prev;

}

delete \_default\_node;

}

void List::push\_front(int data) {

if (!\_count)

{

create\_first\_node(data);

}

else

{

\_head->\_prev = new Node();

\_head->\_prev->\_next = \_head;

\_head = \_head->\_prev;

\_head->\_data = data;

}

\_count++;

}

void List::push\_back(int data) {

if (!\_count)

{

create\_first\_node(data);

}

else

{

\_tail->\_next = new Node();

\_tail->\_next->\_prev = \_tail;

\_tail = \_tail->\_next;

\_tail->\_next = \_default\_node;

\_tail->\_data = data;

}

\_count++;

}

void List::pop\_front()

{

if (\_count > 1)

{

\_head = \_head->\_next;

delete \_head->\_prev;

\_count--;

}

else if (\_count)

{

delete \_head;

\_head = \_default\_node;

\_count--;

}

}

void List::pop\_back()

{

if (\_count > 1)

{

\_tail = \_tail->\_prev;

delete \_tail->\_next;

\_tail->\_next = \_default\_node;

\_count--;

}

else if (\_count)

{

delete \_tail;

\_head = \_default\_node;

\_count--;

}

}

void List::insert(const iterator& itr, int data)

{

if (!\_count)

{

create\_first\_node(data);

}

else if (itr == begin())

{

push\_front(data);

}

else if (itr == end())

{

push\_back(data);

}

else

{

Node\* new\_node = new Node();

new\_node->\_data = data;

new\_node->\_next = itr.\_currentElement;

new\_node->\_prev = itr.\_currentElement->\_prev;

new\_node->\_prev->\_next = new\_node;

itr.\_currentElement->\_prev = new\_node;

}

\_count++;

}

void List::swap()

{

if (\_count > 2)

{

Node\* temp\_first = \_head->\_next;

Node\* temp\_second = \_tail->\_prev;

\_head->\_next = \_default\_node;

\_head->\_prev = temp\_second;

\_tail->\_next = temp\_first;

\_tail->\_prev = nullptr;

temp\_first->\_prev = \_tail;

temp\_second->\_next = \_head;

Node\* temp\_swap = \_head;

\_head = \_tail;

\_tail = temp\_swap;

}

else if (\_count == 2)

{

\_head->\_next = \_default\_node;

\_head->\_prev = \_tail;

\_tail->\_next = \_head;

\_tail->\_prev = nullptr;

\_head = \_tail;

\_tail = \_tail->\_next;

}

}

List::iterator List::begin()

{

return iterator(\_head);

}

List::iterator List::end()

{

return iterator(\_default\_node);

}

void List::create\_first\_node(int data)

{

\_head = new Node();

\_head->\_next = \_default\_node;

\_head->\_data = data;

\_tail = \_head;

}

// Определение класса "iterator"

List::iterator::iterator(Node\* node)

{

\_currentElement = node;

}

List::iterator::iterator(const iterator& itr) : \_currentElement(itr.\_currentElement) {}

const List::iterator& List::iterator::operator=(const iterator& itr)

{

if (this != &itr)

{

\_currentElement = itr.\_currentElement;

return \*this;

}

return \*this;

}

int List::iterator::operator\*() const

{

return \_currentElement->\_data;

}

const List::iterator& List::iterator::operator++()

{

if (\_currentElement->\_next)

{

\_currentElement = \_currentElement->\_next;

return \*this;

}

else

{

throw std::exception("---Нельзя сделать смещение на такое расстояние!---");

}

}

const List::iterator& List::iterator::operator++(int)

{

if (\_currentElement->\_next)

{

List::iterator temp = \*this;

\_currentElement = \_currentElement->\_next;

return temp;

}

else

{

throw std::exception("---Нельзя сделать смещение на такое расстояние!---");

}

}

bool List::iterator::operator==(const iterator& itr) const

{

return this->\_currentElement == itr.\_currentElement;

}

bool List::iterator::operator!=(const iterator& itr) const

{

return this->\_currentElement != itr.\_currentElement;

}

**Заключение**

Список имеет несколько преимуществ перед обычным массивом. Основными преимуществами списков над массивами являются:

1) Легкое добавление элемента в любое место последовательности (в том числе в середину последовательности)

2) Легкое удаление элемента из любой позиции последовательности (в том числе из середины последовательности)

3) Неограниченная размерность последовательности (программисту не нужно изначально указывать размер последовательности на момент написания программы)

4) Легкое объединение нескольких списков (в отличие от массивов)

Но также связанный список имеет и ряд недостатков в сравнении с массивами:

1) Узлы не имеют собственного адреса. Хранится только адрес первого узла, и для того, чтобы добраться до любого узла, нам нужно пройти весь список от начала до нужного узла.

2) Поскольку все узлы не имеют своего конкретного адреса, двоичный поиск не может быть выполнен.

Конкретно в моём варианте был двусвязный линейный список. Его преимущества:

1) Возможность «идти» по списку в обе стороны

2) Легко вставить один или несколько элементов в последовательность

3) Легко удалить необходимые узлы (как по значению, так и по индексу)

4) Всегда можно перейти к началу списка, т. к. есть указатель на голову

Одним из основных недостатков двусвязного линейного списка является память, т. к. структура содержит два указателя, помимо необходимых данных.