**Правительство Российской Федерации**

**Федеральное государственное автономное образовательное учреждение высшего профессионального образования   
"Национальный исследовательский университет   
"Высшая школа экономики"**

Московский институт электроники и математики Национального

исследовательского университета "Высшая школа экономики"

Департамент прикладной математики

**ОТЧЕТ**

**По лабораторной работе № 3**

**По курсу «Компьютерный практикум»**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| |  |  | | --- | --- | |  | ФИО студента | | Номер группы | Дата |
| Рыбаков Владислав Евгеньевич | БПМ-204 | 26.02.2022 |
|  |
|  |
|  |

**Москва – 2022 г.**

**ЗАДАНИЕ**

**Односвязный список элементов.** Создать класс Итератор. Добавить в класс контейнера указанные методы.

**ОПИСАНИЕ КЛАССОВ**

*Для удобства код будет приведен в конце.*

1. **Класс Time**

Представляет собой тип данных, в котором хранится время в формате “мм:сс”. Имеет два поля, в которых хранятся минуты и секунды, а так же сеттер, геттер, операторы ввода-вывода, присваивания и оператор сравнения == с типом данных int (для сравнения с нулем).

1. **Класс Node** (шаблонный класс)

Является одним элементом списка, имеющим два поля: data – в котором хранится информация, а так же next – в котором хранится указатель на следующий элемент списка. Список методов: два конструктора, операторы ввода-вывода, оператор присваивания.

1. **Класс Iterator** (шаблонный класс)

Является отдельным классом. List – дружественный класс. Содержит в себе указатель на Node. Список методов:

* Конструктор с аргументом типа Node\*
* Операторы сравнения == и !=

Сравнивают то, на какую область памяти указывают итераторы, а не на значения, хранящихся в этой памяти.

* Оператор \*

Возвращает значение, хранящееся в области памяти, на которую указывает итератор.

* Постфиксный и префиксный инкременты

Передвигают итератор на следующий элемент списка (если это возможно, а если нет – ничего не делают, чтобы не было ошибки разыменования пустого указателя)

* Оператор присваивания

1. **Класс List** (шаблонный класс)

Является основным классом-контейнером, содержит в себе единственное поле head – указатель Node\* на начало списка. Все узлы списка хранятся в динамической памяти, поэтому в классе List происходит полное очищение всей выделенной памяти. Итак, список методов класса:

* createIterator() – создает итератор на первый элемент списка.
* begin() – возвращает итератор на первый элемент списка
* end() – возвращает итератор на конец списка (ВАЖНО: не на последний элемент, а именно на конец списка, то есть имеющий указатель равный NULL)
* insert\_after(Iterator, abstracted) – получает на вход итератор и значение, вставляет это значение в список после элемента, на который указывает итератор. Возвращает итератор, указывающий на вставленный элемент.
* erase\_after(Iterator) - получает на вход итератор, удаляет элемент в списке после элемента, на который указывает итератор. Делает проверку на то, есть ли что удалять.
* erase\_after(Iterator first, Iterator second) - удаляет диапазон элементов, на начало и конец которого указывают соответственно итераторы first и second. Возвращает итератор на элемент после последнего удаленного. Делает проверку на то, есть ли что удалять.
* empty() – возвращает true, если список пустой
* swap(List, List) – статический метода класса, обменивает содержимое двух списков.
* add\_head(abstracted) – добавляет значение в начало списка
* add\_tail(abstracted) – добавляет значение в конец списка
* printList() – печатает список
* createList() – метод создания списка
* delete\_head() – удаляет первое значение списка
* delete\_tail() – удаляет последнее значение списка
* деструктор – при завершении области видимости списка, очищается вся выделенная ранее динамическая память

**ОТВЕТЫ НА ВОПРОСЫ**

1. Контейнер поддерживает все операции итератора, которые для него написаны. У итератора отсутствуют операции постфиксного и префиксного декремента (потому что в односвязном списке невозможно двигаться в обратном направлении), операции ввода-вывода (потому что они не нужны, для вывода значения используется оператор \*), операции умножения, деления (поскольку они не нужны для итератора), += (потому что вместо этого есть инкремент). Почти все операции могут подойти и двусвязному списку, однако для использования обратного обхода нужно будет добавить операции. Помимо осуществляющих доступ к данным определённого контейнера в библиотеке STL имеются ещё несколько итераторов, подходящих к списку: istream\_iterator и ostream\_iterator. Они осуществляют считывание данных из потока и запись данных в поток. В конструкторы этих итераторов необходимо передать входной или же выходной поток.
2. Итератор является отдельным классом, что позволяет ему существовать отдельно от списка. Так же это дает возможность для наследования. Преимущества реализации в том, что итератор не создается как указатель на область в динамической памяти. Это позволяет гораздо проще с ним работать, а так же предотвращает возможную утечку памяти. Недостатков нет. Итератор не является другом для класса контейнера, однако сам класс контейнер является другом итератора (для доступа к единственному закрытому полю итератора). Итератор внешний, это дает больше возможностей клиенту. Управляет обходом агрегат, поэтому итератор является по сути просто курсором.

**КОД ПРОГРАММЫ**

main:

#include "Time.h"

#include "List.h"

#include "Iterator.h"

int main()

{

List<Time> myList;

Time one(1, 1);

myList.createList();

Iterator<Time> it = myList.createIterator();

std::cout << "\nYour list:\n";

for (it = myList.begin(); it != myList.end(); ++it) {

std::cout << \*it << '\n';

}

it = myList.begin(); // после первого элемента вставим другой

myList.insert\_after(it, one);

std::cout << "\nYour new list:\n";

for (it = myList.begin(); it != myList.end(); ++it) {

std::cout << \*it << '\n';

}

it = myList.begin(); // удалим второй элемент

myList.erase\_after(it);

std::cout << "\nYour new list:\n";

for (it = myList.begin(); it != myList.end(); ++it) {

std::cout << \*it << '\n';

}

it = myList.end(); // попробуем удалить конец списка

std::cout << "\nTry to erase after end of list: ";

myList.erase\_after(it);

it = myList.begin();

Iterator<Time> it2 = myList.createIterator();

it++; // поставим первый итератор на 2-ой элемент

it2++;

it2++;

it2++; // второй итератор на 4-й элемент

myList.erase\_after(it, it2);

std::cout << "\nYour new list:\n";

for (it = myList.begin(); it != myList.end(); ++it) {

std::cout << \*it << '\n';

}

//List<Time> myNewList;

//myNewList.createList();

//List<Time>::swap(myList, myNewList);

//myList.printList();

//myNewList.printList();

}

Time:

#pragma once

#include <iostream>

class Time

{ //Time class

public:

Time() = default; // конструктор по умолчанию

Time(int, int); // конструктор со значениями минут и секунд

void setTime(int, int); // сеттер

void getTime(int&, int&) const; // геттер

friend std::ostream& operator << (std::ostream& out, const Time& a);

friend std::istream& operator >> (std::istream& in, Time& a);

friend bool operator==(const Time& a, const int& i);

Time& operator=(const Time& a);

Time& operator=(const int& a);

private:

int min, sec; // минуты и секунды соответственно

};

Time::Time(int mi, int se) {

min = (mi + se / 60) % 60; // проверка если секунд ввелось больше 60 и минут получается больше 60, то высчитывается правильное время

sec = se % 60; // высчитывается правильное кол-во секунд

}

void Time::setTime(int mi, int se) {

min = (mi + se / 60) % 60;

sec = se % 60;

}

void Time::getTime(int& one, int& two) const { // в две ссылки кладутся значения полей объекта

one = min;

two = sec;

}

std::ostream& operator << (std::ostream& out, const Time& a) { // оператор вывода

if (a.sec > 9) { // если число секунд двузначно

out << a.min << ":" << a.sec;

}

else {

out << a.min << ":" << "0" << a.sec; // если число секунд однозначно, то добавляем после двоеточия 0

}

return out;

}

std::istream& operator >> (std::istream& in, Time& a) { // оператор ввода

char o; // в переменную о должнен ввестись символ ':'

int mi, se;

in >> mi >> o >> se; // подразумевается, что ввод будет в виде mm:ss

a.setTime(mi, se);

return in;

}

Time& Time::operator=(const Time& a) { // оператор присваивания такого же типа

min = a.min;

sec = a.sec;

return \*this;

}

Time& Time::operator=(const int& a) { // оператор присваивания переменной типа Time значения int (для работы с NULL)

min = (a / 60) % 60;

sec = a % 60;

return \*this;

}

bool operator==(const Time& a, const int& i) { // оператор сравнения переменной типа Time и типа int (для сравнения с NULL)

if (a.min == (i / 60) % 60 && a.sec == i % 60) return true;

return false;

}

Node:

#pragma once

#include <iostream>

template <typename abstracted>

class Node

{

public:

Node\* next;

abstracted data;

Node() : next(NULL) {}

explicit Node(const abstracted& smth) : data(smth), next(NULL) {}

friend std::ostream& operator << (std::ostream& out,const Node& a) {

out << a.data << " "; // у типа abstracted должен быть перегружен оператор вывода

return out;

}

friend std::istream& operator >> (std::istream& in, Node& a) { // оператор ввода

abstracted smt;

in >> smt; // у типа abstracted должен быть перегружен оператор ввода

a.data = smt;

return in;

}

Node& operator=(const Node& a) { // оператор присваивания

data = a.data;

next = a.next;

return \*this;

}

};

List:

#pragma once

#include "Node.h"

#include "Iterator.h"

template <typename abstracted>

class List

{

Node<abstracted>\* head;

public:

List() : head(NULL) {}

Iterator<abstracted>& createIterator() const {

Iterator<abstracted> it(head);

return it;

}

Iterator<abstracted> begin() const {

Iterator<abstracted> it(head);

return it;

}

Iterator<abstracted> end() const {

Iterator<abstracted> it(NULL);

return it;

}

Iterator<abstracted> insert\_after(const Iterator<abstracted>& it, const abstracted& val) const {

if (it.container == NULL) {

std::cout << "You can not insert here!\n";

return Iterator<abstracted>(NULL);

}

Node<abstracted>\* tmp = new Node<abstracted>(val);

tmp->next = it.container->next;

it.container->next = tmp;

return Iterator<abstracted>(tmp);

}

Iterator<abstracted> erase\_after(const Iterator<abstracted>& it) const {

if (it.container == NULL || it.container->next == NULL) {

std::cout << "Nothing to delete!\n";

return Iterator<abstracted>(NULL);

}

else {

Node<abstracted>\* tmp = it.container->next;

it.container->next = tmp->next;

delete tmp;

return Iterator<abstracted>(it.container->next);

}

}

Iterator<abstracted> erase\_after(const Iterator<abstracted>& first, const Iterator<abstracted>& last) const {

Node<abstracted>\* tmp = first.container->next;

if (first.container == NULL || first.container->next == NULL) {

std::cout << "Nothing to delete!\n";

return Iterator<abstracted>(NULL);

}

first.container->next = last.container->next;

last.container->next = NULL;

Node<abstracted>\* p;

for (; tmp != NULL;) {

p = tmp->next;

delete tmp;

tmp = p;

}

return first;

}

bool empty() const {

if (begin() == end()) return true;

return false;

}

static void swap(List<abstracted>& L1, List<abstracted>& L2) {

Node<abstracted>\* tmp = L1.head;

L1.head = L2.head;

L2.head = tmp;

}

void add\_head(const abstracted& one) {

if (head == NULL) {

head = new Node<abstracted>(one);

return;

}

Node<abstracted>\* tmp = new Node<abstracted>(one);

tmp->next = head;

head = tmp;

}

void add\_tail(const abstracted &one) {

if (head == NULL) {

head = new Node<abstracted>(one);

return;

}

Node<abstracted>\* tmp = head;

for (; tmp->next != NULL; tmp = tmp->next);

tmp->next = new Node<abstracted>(one);

}

void printList() const {

Node<abstracted>\* tmp = head;

std::cout << "\nYour list:\n";

if (tmp == NULL || tmp->data == NULL && tmp->next == NULL) { // проверка на пустоты списка

std::cout << "\nList is empty!\n";

return;

}

for (tmp = head; tmp != NULL; tmp = tmp->next) std::cout << tmp->data << '\n';

}

void createList() {

int n;

abstracted data;

Node<abstracted> \*tmp;

std::cout << "Enter number of elements: ";

std::cin >> n;

if (n <= 0) { // проверка на некорректный ввод

std::cout << "\nWrong enter!\n";

return;

}

std::cout << "Enter " << n << " elements:\n";

for (int i = 0; i < n; i++) {

std::cin >> data;

add\_tail(data);

}

}

void delete\_head() {

if (head->next == NULL) {

delete head;

head = NULL;

}

else {

Node<abstracted>\* tmp = head;

head = head->next;

delete tmp;

}

}

void delete\_tail() {

if (head->next == NULL) {

delete head;

head = NULL;

}

else {

Node<abstracted>\* tmp = head;

for (; (tmp->next)->next != NULL; tmp = tmp->next);

delete (tmp->next);

tmp->next = NULL;

}

}

~List() {

while (head != NULL) delete\_head();

}

};

Iterator:

#pragma once

#include "List.h"

template<typename abstracted>

class List;

template<typename abstracted>

class Iterator

{

Node<abstracted>\* container;

public:

Iterator(Node<abstracted>\* ptr): container(ptr) {}

friend class List<abstracted>;

bool operator==(const Iterator a) const {

return this->container == a.container;

}

bool operator!=(const Iterator a) const {

return this->container != a.container;

}

abstracted& operator\*() {

return container->data;

}

Iterator& operator++()

{

if (container != NULL) container = container->next;

return \*this;

}

Iterator operator++(int)

{

Iterator\* prev = this;

if(container != NULL) container = container->next;

return \*prev;

}

Iterator& operator=(const Iterator& a) {

container = a.container;

return \*this;

}

};

**ТЕСТЫ**