Министерство образования и науки Российской Федерации

Федеральное государственное образовательное автономное учреждение высшего образования

"Пермский национальный исследовательский политехнический университет"

ОТЧЕТ

ПО ЛАБОРАТОРНОЙ РАБОТЕ №6

Дисциплина: Основы алгоритмизации и программирования

Тема: АТД. Контейнеры.

|  |  |
| --- | --- |
| Выполнил работу | |
| Студент группы РИС-22-1б | |
| Деревнин И.В. | |
|  | |
| Проверил работу | |
| Доцент кафедры ИТАС | |
| Полякова О.А. | |
|  | |

Пермь – 2023

**Анализ предметной области**

**Постановка задачи**

1. Определить класс контейнер.
2. Реализовать конструкторы, деструктор, операции ввода-вывода, операцию присваивания.
3. Реализовать итератор. Реализовать с его помощью операции последовательного доступа.
4. Написать тестирующую программу, иллюстрирующую выполнение операций.

Вариант 15: Класс-контейнер список с ключевыми значениями типа int.

Реализовать операции:

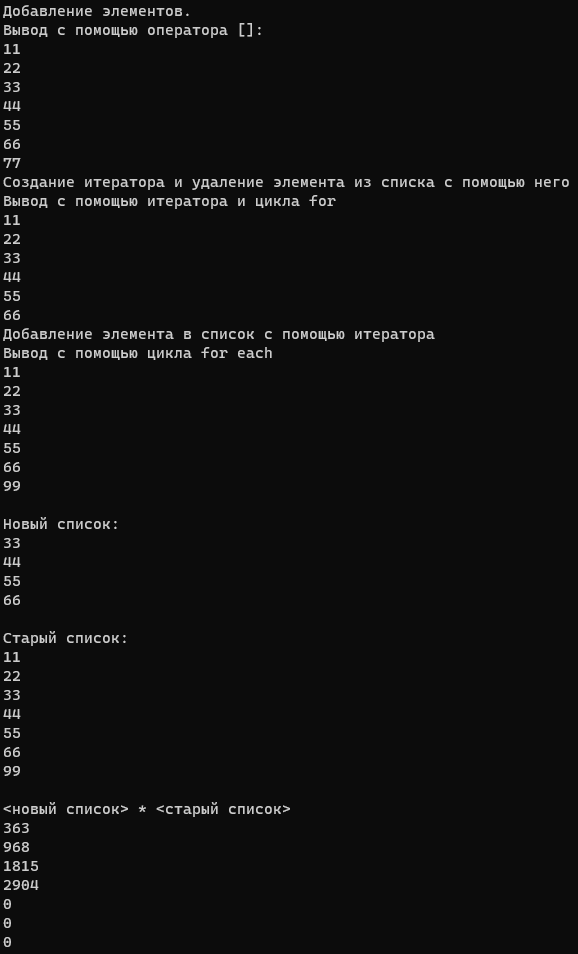
[] – доступ по индексу;

Int() – определение размера списка;  
\* список – умножение элементов списков a[i] \* b[i];

+n – переход вправо к элементу с номером n (с помощью класса-итератора).

**Анализ задачи**

Необходимо реализовать класс односвязный список и итератор. Для этого необходимо: реализовать вложенный класс Node – который будет являться минимальной ячейкой памяти списка, в Node будет хранится data и указатель на следующий элемент. Далее необходимо вести подсчет элементов в списке, для этого необходимо создать поле size и в данное поле подсчитывать изменения, вносимые в список. Также для списка необходимо перегрузить операторы: [] – для доступа по индексу и «\*» для умножения одного списка на другой. Для быстродействия данной структуры данных необходимо реализовать итератор, который также будет являться вложенным классом, так как данный итератор никуда больше не сможет подойти. Итератор должен указывать на какой-либо элемент, следовательно поле его класса – минимальная ячейка памяти списка – указатель на node. Для итератора необходимо реализовать операторы: +n, префиксный и постфикксный инкремент, оператор разыменования, для того, чтобы можно было получать данные, оператор == и != для возможности перебирать данные с помощью цикла и итератора. А также необходимо для списка реализовать методы begin и end, которые будут возвращать итераторы, которые указывают на первый и последний элемент соответственно.

****Тестирование программы**

*Рис. 1 – Тестирование программы.*

**Заключение**

Была разработана программа, в которой реализована динамическая структура данных – односвязный список. В данном односвязном списке реализовано 2 вложенных класса для его нужд: класс node – минимальная ячейка данных и класс iterator – для более быстрого и прямого доступа к данным. Также для односвязного списка и итератора были реализованы методы, благодаря которым список может работать с range-based циклами.

# Ответы на контрольные вопросы

1. Что такое абстрактный тип данных? Привести примеры АТД

АТД - тип данных, определяемый только через операции, которые могут

выполняться над соответствующими объектами безотносительно к способу представления этих объектов.

template <typename T>

T & Stack<T>::push(){ //операция добавления элементов в АТД типа стек

return head->data;

}

2. Привести примеры абстракции через параметризацию.

template <typename T>

class node {

T data;

node\* next, \*prev;

}

3. Привести примеры абстракции через спецификацию.

class complex {

double re, im;

friend complex operator+(complex, complex);

public:

complex() { re = im = 0;}

complex(double r) { re = r; im = 0; }

complex(double r, double i) { re = r; im = i; }

};

complex operator+(complex a1, complex a2) {

return complex(a1.re + a2.re, a1.im + a2.im);

}

Для поддержки смешанной арифметики комплексных и действительных чисел (Re x, In y) + double A требуется специфицировать перегруженную функцию.

complex operator+(complex,complex);

complex operator+(complex,double);

complex operator+(double,complex);

Абстракция через спецификацию достигается за счет того, что операции

представляются как часть типа (абстракция вычислений, перегрузка операторов).

4. Что такое контейнер? Привести примеры.

Контейнер – это объект. Имя контейнера – это имя переменной. Контейнер, так же как и другие объекты, обладает временем жизни. Время жизни контейнера в общем случае не зависит от времени жизни его элементов. Элементами контейнера могут любые объекты, в том числе, и другие контейнеры.

class queue() {

int size;

node \*head, \*tail;

…

};

//в основной функции

queue\* q; //контейнер-очередь

float a = 7.5;

q->head = a; //головой очереди q является float переменная

queue \*c;

c->tail = q; //хвостом очереди c является контейнер queue

5. Какие группы операций выделяют в контейнерах?

Операции доступа к элементам

template <typename T>

class queue() {

int size;

node \*head, \*tail;

node & getHead() { return head->data; }

…

};

* Операции замены значений элементов;

void queue::setHead(node head){ this->head = head; }

* Операции добавления и удаления элементов или групп элементов;

T queue::pop(){

T data = tail->data;

size–;

tail = tail->prev;

tail->next = nullptr;

return data;

}

* Операции поиска элементов и групп элементов;

int queue::find\_index(T data) {

int counter = 0;

node \* curr = this->head;

while (counter < size) {

if (curr->data = data) return counter;

curr = curr->next;

counter++;

}

}

* Операции объединения контейнеров;

queue queue::merge(queue q1, queue q2) {

q1->tail->next = q2->head;

q1->tail = q2->tail;

delete q2;

return q1;

}

* Специальные операции, которые зависят от вида контейнера.

bool queue::isCircular(queue q) {

return (q->tail->next == q->head); }

6. Какие виды доступа к элементам контейнера существуют? Привести примеры.

Доступ к элементам контейнера бывает: последовательный, прямой и

ассоциативный.

Прямой доступ – это доступ по индексу. Например, a[10] – требуется найти

элемент контейнера с номером 10 (с учетом индексации с нуля)

Ассоциативный доступ также выполняется по индексу, но индексом будет являться не номер элемента, а его содержимое. Пусть имеется контейнер–словарь, в котором хранится информация, состоящая, как минимум из двух полей: слово и его перевод.

Индексом будут данные поля, например, q[T data]. Поле, с содержимым которого ассоциируется элемент контейнера, называется ключом или полем доступа. Элемент, с которым ассоциируется ключ, называется

значением. Контейнер, который представляет ассоциативный доступ, состоит из пар «ключ-значение» (ключ - T data, значение - node name).

При последовательном доступе осуществляется перемещение от элемента к элементу контейнера.

Например, дана очередь, и для того, чтобы найти индекс элемента, требуется поэлементно пройти по каждому узлу.

int queue::find\_index(T data) {

int counter = 0;

node \* curr = this->head;

while (counter < size) {

if (curr->data = data) return counter;

curr = curr->next;

counter++;

}

}

7. Что такое итератор?

Итератор (перечислитель) — интерфейс, предоставляющий доступ к элементам контейнера и навигацию по ним. Главное предназначение итераторов заключается в предоставлении возможности пользователю обращаться к любому элементу контейнера при сокрытии внутренней структуры контейнера от пользователя. Это позволяет контейнеру хранить элементы любым способом при допустимости работы пользователя с ним как с простой последовательностью или списком. Итератор можно реализовать как класс, представляющий такой же набор операций.

В С++ итератор реализуется как класс, который имеет такой же интерфейс, как и указатель для совместимости с массивами.

8. Каким образом может быть реализован итератор?

class iterator() {

friend class queue; //поддержка работы с классом очередей

public:

iterator() { elem = nullptr; }

…

iterator & operator++() {  
 elem = elem->next; //реализация части кода с переходом на следующий элемент

}

…

};

9. Каким образом можно организовать объединение контейнеров?

queue queue::merge(queue q1, queue q2) {

q1->tail->next = q2->head;

q1->tail = q2->tail;

delete q2;

return q1;

}

10. Какой доступ к элементам предоставляет контейнер, состоящий из элементов «ключ-значение»?

Ассоциативный доступ (индексом будут данные поля, например, q[T data]. Поле, с содержимым которого ассоциируется элемент контейнера, называется ключом или полем доступа. Элемент, с которым ассоциируется ключ, называется значением)

11. Как называется контейнер, в котором вставка и удаление элементов выполняется на одном конце контейнера?

Стек

12. Какой из объектов (a,b,c,d) является контейнером?

a. int mas=10; //переменная типа int

b. 2. int mas; //переменная типа int

c. 3. struct {char name[30]; int age;} mas; //структура

d. 4. int mas[100]; //массив переменных int

Ответ: d

13. Какой из объектов (a,b,c,d) не является контейнером?

a. int a[]={1,2,3,4,5}; //заполненный массив переменных int

b. 2. int mas[30]; //пустой массив переменных int

c. 3. struct {char name[30]; int age;} mas[30]; //массив структур

d. 4. int mas; //переменная типа int

Ответ: d

14. Контейнер реализован как динамический массив, в нем определена операция доступ по индексу. Каким будет доступ к элементам контейнера?

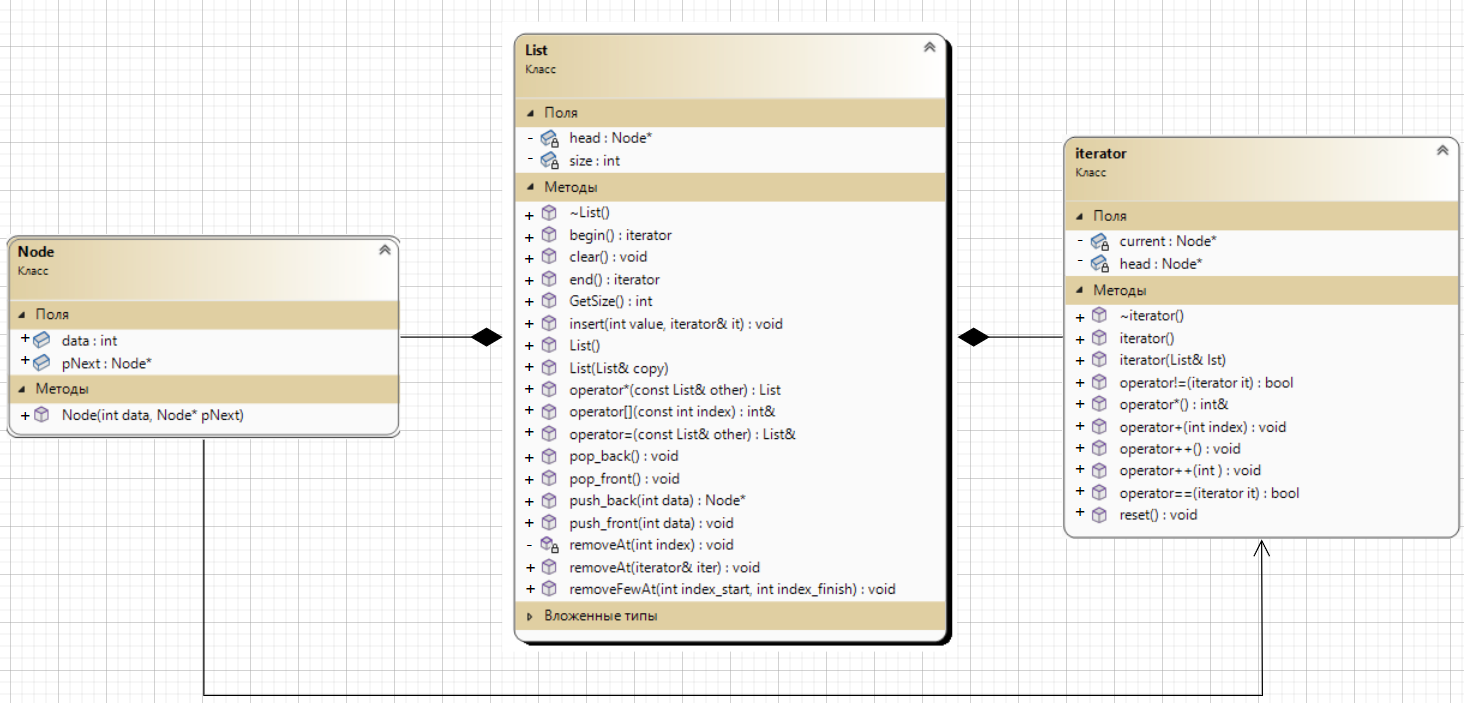
В С++ массивы поддерживают доступ по индексу. Доступ по индексу - это прямой доступ.

15. Контейнер реализован как линейный список. Каким будет доступ к элементам контейнера?

В С++ в линейной списке нет доступа по индексу, и требуется пройти по каждому элементу. Это последовательный доступ.

**Приложения**

Приложение UML-диаграмма



Приложение Б – код программы

Main.cpp:

#include <iostream>

#include <Windows.h>

#include <fstream>

#include "List.h"

using namespace std;

int main()

{

system("chcp 1251 >> null");

//system("color F0");

List lst;

cout << "Добавление элементов." << endl;

lst.push\_back(22);

lst.push\_front(11);

lst.push\_back(33);

lst.push\_back(44);

lst.push\_back(55);

lst.push\_back(66);

lst.push\_back(77);

cout << "Вывод с помощью оператора []:" << endl;

for (int i = 0; i < lst.GetSize(); i++)

{

cout << lst[i] << endl;

}

cout << "Создание итератора и удаление элемента из списка с помощью него" << endl;

auto i = lst.begin();

i + 6;

lst.removeAt(i);

cout << "Вывод с помощью итератора и цикла for" << endl;

for (auto it = lst.begin(); it != lst.end(); it++)

{

cout << \*it << endl;

}

cout << "Добавление элемента в список с помощью итератора" << endl;

lst.insert(99, i);

cout << "Вывод с помощью цикла for each\n";

for (auto& it : lst)

{

cout << it << endl;

}

List list2;

list2 = lst;

list2.pop\_front();

list2.pop\_back();

list2.pop\_front();

cout << endl << "Новый список:" << endl;

for (auto& it : list2)

{

cout << it << endl;

}

cout << endl << "Старый список:" << endl;

for (auto& it : lst)

{

cout << it << endl;

}

cout << "\n<новый список> \* <старый список>" << endl;

list2 = (list2 \* lst);

for (auto& it : list2)

{

cout << it << endl;

}

return 0;

}

list.h:

#pragma once

class List

{

private:

void removeAt(int index);

//минимальная ячейка памяти

class Node //вложенный класс - ячейка памяти

{

public:

Node(int data = int(), Node\* pNext = nullptr) //используется параметр по умолчанию, то есть если мы не передаем указатель, то он автоматически присваивается nullptr

{ //для поля data также используется параметр по умолчанию, выглядит он как вызов конструктора по умолчанию

this->data = data;

this->pNext = pNext;

}

Node\* pNext; //хранит указатель на следующий элемент

int data; //хранит наши данные

};

Node\* head; //указатель на первый элемент в спискеint

int size; //количество элементов в списке.

public:

class iterator //вложнный итератор

{

private:

friend class List;

Node\* head; //указатель на node

Node\* current;

public:

iterator(); //конструкторы итератора / деструкторы

~iterator();

iterator(List& lst);

void reset();

void operator +(int index); //операторы итератора

void operator ++(int);

void operator ++();

int& operator \*();

bool operator ==(iterator it);

bool operator !=(iterator it);

};

List(); //Конструктор

List(List& copy);//!!!!!!

~List(); //Деструктор

List& operator=(const List& other);//!!!!!!!!!!

Node\* push\_back(int data); //добавляет элемент в конец списка

int GetSize() { return size; } //так как инкапсуляция не дает нам доступ к количеству элементов в списке, то нам необходимо происать геттер

int& operator[](const int index); //перегрузка оператора [] для того, чтобы мы могли взаимодействовать со списком как с массивом

void pop\_front(); //удаляет элемент из начала списка

void clear(); //удаляет все элементы списка (логика как у pop\_front, деструктор основан на этой штуке)

void push\_front(int data); //добавляет элемент в начало списка

void removeAt(List::iterator& iter); //удаление элемента по указаному индексу

void pop\_back(); //удаление из конца списка

iterator begin();

iterator end();

void removeFewAt(int index\_start, int index\_finish); //удаление нескольких элементов

friend class iterator;

void insert(int value, List::iterator& it);

List operator\*(const List& other);

};

list.cpp:

#include"List.h"

#include <iostream>

List::List() //мы создаем пустой список, а следствено, в нем лежит 0 элементов, а следственно указывать некуда.

{

size = 0;

head = nullptr;

}

List::List(List& copy)

{

this->size = 0;

this->head = nullptr;

Node\* temp = copy.head;

while (temp != nullptr) {

push\_back(temp->data);

temp = temp->pNext;

}

}

List::~List() //деструктор освобождает динамическую память по приципу метода clear,

{ //мы можем делегировать ему эти обязаности

Node\* temp; //но здесь прописана логика удаления

while (size)

{

temp = head;

head = head->pNext;

delete temp;

size--;

}

}

List::Node\* List::push\_back(int data)

{

if (head == nullptr) //создание первого элемента

{

head = new Node(data); //создаем элемент и помещаем его в голову

size++;

return head;

}

else //создание последующих элементов, создаем указатель на первый элемент

{ //и идем по элементам, пока не наткнемся на тот, что указывает на nullptr

Node\* current = this->head;

while (current->pNext != nullptr)

{

current = current->pNext;

}

current->pNext = new Node(data);

++size;

return current->pNext;

}

//количество элементов увеличиваем на 1

}

int& List::operator[](const int index) //мы не можем просто вернуть данные из списка, для этого сначала необходимо перейти в нужную ячейку

{

int counter = 0; //счетчик

Node\* current = this->head; //указатель, для того, чтобы итерироваться по нашему списку

while (current != nullptr) //цикл с условием, что мы не вылетим за границы нашего списка

{

if (counter == index)

{

return current->data; //если мы находимся в нужной ячейке то получаем оттуда данные

}

current = current->pNext; //меняем наш временный указатель на следующую ячейку

counter++;

}

}

void List::pop\_front() //удаление из начала

{

Node\* temp = head; //делай временный указатель на объект head

head = head->pNext; //Указательь head переносим на следующую ячейку

delete temp; //удаляем первый объект

size--; //уменьшаем длину на 1

}

void List::clear()

{

//Node<T>\* temp;

while (size)

{

//temp = head; //вызываем pop\_front столько раз, пока size != 0

//head = head->pNext; //логика основывается на pop\_front поэтому мы делегируем ей эти обязанности

//delete temp;

//size--;

pop\_front();

}

this->head = nullptr;

}

void List::push\_front(int data) //добавление элемента в начало

{

head = new Node(data, head); //вместо головного элемента создаем новый элемент, указатель в котором будет на старый головной элемент

size++; //увеличиваем длину на 1

}

void List::insert(int value, List::iterator& it)

{

if (it.current == nullptr) { it.current = push\_back(value); }

else if (it.head == it.current) { push\_front(value); it.current = it.head = this->head; }

else

{

Node\* temp = it.head;

while (temp->pNext != it.current)

temp = temp->pNext;

temp->pNext = new Node(value, temp->pNext);

++size;

}

}

void List::removeAt(List::iterator& iter)

{

if (iter.current == nullptr) return;

Node\* temp = iter.head;

while (temp->pNext != iter.current)

temp = temp->pNext;

temp->pNext = iter.current->pNext;

delete iter.current;

iter.current = temp->pNext;

--size;

}

void List::removeAt(int index)

{

if (index == 0)

{

pop\_front();

}

else

{

Node\* previous = head;

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

{

previous = previous->pNext;

}

Node\* toDelete = previous->pNext;

previous->pNext = toDelete->pNext;

delete toDelete;

size--;

}

}

void List::pop\_back()

{

removeAt(size - 1);

}

void List::removeFewAt(int index\_start, int index\_finish) //удаление нескольких элементов

{

if (index\_start + (index\_finish - index\_start) >= size) return; //условие невыхода за границы

for (int i = 0; i < index\_finish - index\_start + 1; i++)

{

removeAt(index\_start);

}

}

List::iterator List::begin()

{

iterator it;

it.head = List::head;

it.current = List::head;

return it;

}

List::iterator List::end()

{

iterator temp;

return temp;

}

List List::operator\*(const List& other)

{

List temp;

Node\* l1 = this->head;

Node\* l2 = other.head;

while (l1 != nullptr && l2 != nullptr)

{

temp.push\_back(l1->data \* l2->data);

l1 = l1->pNext;

l2 = l2->pNext;

}

int sz = abs(this->size - other.size);

while (sz)

{

temp.push\_back(0);

--sz;

}

return temp;

}

List& List::operator=(const List& other)

{

this->clear();

Node\* temp = other.head;

while (temp != nullptr)

{

this->push\_back(temp->data);

temp = temp->pNext;

}

return \*this;

}

Iterator.cpp:

#include "List.h"

List::iterator::iterator()

{

this->head = nullptr;

this->current = nullptr;

}

List::iterator::~iterator()

{

this->head = nullptr;

this->current = nullptr;

}

List::iterator::iterator(List& lst)

{

this->head = lst.head;

this->current = lst.head;

}

void List::iterator::operator+(int index)

{

for (int i = 0; i < index; i++)

{

this->current = this->current->pNext;

}

}

void List::iterator::operator++(int)

{

this->current = this->current->pNext;

}

void List::iterator::operator++()

{

this->current = this->current->pNext;

}

void List::iterator::reset()

{

this->current = this->head;

}

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

{

return this->current->data;

}

bool List::iterator::operator==(iterator it)

{

if (this->current == it.current) return true;

return false;

}

bool List::iterator::operator!=(iterator it)

{

return !(\*this == it);

}