Министерство образования и науки Российской Федерации

Федеральное государственное бюджетное образовательное учреждение высшего образования

**«Пермский национальный исследовательский политехнический университет»**

Кафедра «Информационные технологии и автоматизированные системы»

**ОТЧЕТ**

Тема: «АТД. Контейнеры»

Выполнил работу

Студент группы РИС-22-2Б

Вахрушева А.В.

Проверил

Доцент кафедры ИТАС

Полякова О.А.

Г. Пермь-2023

**Постановка задачи**

1. Определить класс-контейнер.

2. Реализовать конструкторы, деструктор, операции ввода-вывода, операцию присваивания.

3. Перегрузить операции, указанные в варианте.

4. Реализовать класс-итератор. Реализовать с его помощью операции последовательного доступа.

5. Написать тестирующую программу, иллюстрирующую выполнение операций.

**Вариант 15**

Класс-контейнер СПИСОК с ключевыми значениями типа int.   
Реализовать операции: [] – доступа по индексу;   
int() – определение размера списка;   
\* вектор – умножение элементов списков a[i]\*b[i];   
+n - переход вправо к элементу с номером n ( с помощью класса-итератора).

**Ответы на контрольные вопросы**

1. Абстрактный тип данных (АТД) - это математическая модель, описывающая поведение некоторой абстрактной структуры данных и набор операций, которые могут быть выполнены над этой структурой. Примеры АТД: классы, стек, очередь, дерево, граф.
2. Абстракция через параметризацию - это создание обобщенной структуры данных, которая может работать с разными типами данных. Примеры: шаблоны классов и функций в C++.
3. Абстракция через спецификацию - это определение интерфейса для работы со структурой данных, скрывающее детали реализации. Примеры: абстрактные классы в С++.
4. Контейнер - это структура данных, которая хранит и управляет коллекцией элементов определенного типа. Примеры: вектор, список, множество, словарь.
5. Группы операций, выделяемых в контейнерах, включают вставку, удаление, поиск, доступ к элементам, сортировку и обход элементов.
6. Существуют два вида доступа к элементам контейнера: доступ по индексу (например, в векторе) и доступ через итераторы (например, в списке).
7. Итератор - это объект, который позволяет последовательно перебирать элементы контейнера. Он предоставляет интерфейс для доступа к элементам контейнера и управления их обходом.
8. Итератор может быть реализован как указатель на элемент контейнера, как объект с методами next() и hasNext(), как указатель на функцию и т.д.
9. Объединение контейнеров можно организовать, используя операции объединения, пересечения и разности множеств, которые определены для соответствующих контейнеров.
10. Контейнер, состоящий из элементов "ключ-значение", предоставляет доступ к элементам по ключу.
11. Контейнер, в котором вставка и удаление элементов выполняется на одном конце контейнера, называется стеком.
12. Из объектов (a,b,c,d) только c. struct {char name[30]; int age;} mas; является контейнером.
13. Из объектов (a,b,c,d) только d. int mas; не является контейнером.
14. Если контейнер реализован как динамический массив и имеет операцию доступа по индексу, то доступ к элементам осуществляется путем указания индекса элемента. Например, чтобы получить значение элемента с индексом i, нужно написать container[i].
15. Если контейнер реализован как линейный список, то доступ к элементам осуществляется путем перебора всех элементов списка от начала до нужного элемента.

**Диаграмма класса**

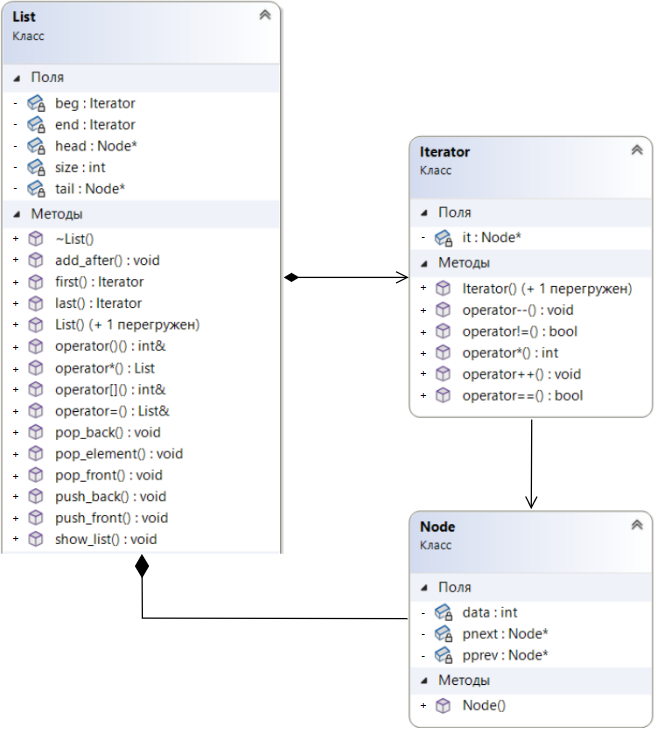


Рисунок 1 – классы List, Iterator и Node

**Описание классов**

Class List со вложенными классами Node и Iterator

#pragma once

#include <iostream>

#include <string>

using namespace std;

class List

{

private:

class Node

{

friend class List;

private:

int data;

Node\* pnext = nullptr;

Node\* pprev = nullptr;

public:

Node(int data);

};

int size = 0;

Node\* head = nullptr;

Node\* tail = nullptr;

class Iterator

{

friend class List;

private:

Node\* it = nullptr;

public:

Iterator();

Iterator(const Iterator& it);

bool operator==(const Iterator& it);

bool operator!=(const Iterator& it);

void operator++();

void operator--();

int operator\*();

};

Iterator beg;

Iterator end;

public:

List();

List(List& object);

~List();

void push\_front(int data);

void pop\_front();

void push\_back(int data);

void pop\_back();

void show\_list();

void add\_after(int number, int data);

void pop\_element(int number);

int& operator[](int index) const;

int& operator()();

List& operator=(const List& object);

List operator\*(List& object);

Iterator first()

{

if (head != nullptr)

beg.it = head;

return beg;

}

Iterator last()

{

if (tail != nullptr)

end.it = tail->pnext;

return end;

}

};

**Определение методов**

Class List и вложенный Node

#include "List.h"

List::List()

{

size = 0;

head = nullptr;

tail = nullptr;

}

List::List(List& object)

{

for (int i = 0; i < object.size; i++)

(\*this).push\_back(object[i]);

}

List::~List()

{

if (head != nullptr)

{

Node\* current = head;

while (current != nullptr)

{

Node\* next = current->pnext;

delete current;

current = next;

}

head = nullptr;

tail = nullptr;

size = 0;

}

}

List::Node::Node(int data)

{

this->data = data;

}

void List::push\_front(int data)

{

if (head == nullptr)

{

head = new Node(data);

tail = head;

}

else

{

Node\* temp = head;

head = new Node(data);

temp->pprev = head;

head->pnext = temp;

}

++size;

}

void List::pop\_front()

{

if (head != nullptr)

{

Node\* temp = head->pnext;

delete head;

head = temp;

if (head != nullptr)

head->pprev = 0;

else

tail = nullptr;

--size;

}

else

{

system("cls"); cout << "Список пуст!\n\n";

}

}

void List::push\_back(int data)

{

if (head == nullptr)

{

head = new Node(data);

tail = head;

}

else

{

Node\* current = tail;

current->pnext = new Node(data);

tail = current->pnext;

tail->pprev = current;

tail->pnext = 0;

}

++size;

}

void List::pop\_back()

{

if (head != nullptr)

{

Node\* current = tail->pprev;

delete tail;

tail = current;

if (tail != nullptr)

current->pnext = nullptr;

else

head = nullptr;

--size;

}

else { system("cls"); cout << "Список пуст!\n\n"; }

}

void List::show\_list()

{

/\*system("cls");\*/

if (head != nullptr)

{

int choice = 1;

/\*cout << "Введите 1 - вывод списка с начала\nВведите 2 - вывод списка с конца\n\n";

cin >> choice;

system("cls");\*/

switch (choice)

{

case 1:

{

/\*cout << "Список: ";\*/

Node\* current = head;

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

{

cout << current->data << " ";

current = current->pnext;

}

cout << endl;

break;

}

case 2:

{

cout << "Список: ";

Node\* current = tail;

for (size\_t i = size; i > 0; i--)

{

cout << current->data << " ";

current = current->pprev;

}

cout << endl;

break;

}

default:

system("cls"); cout << "Ошибка ввода!\n";

break;

}

}

else { system("cls"); cout << "Список пуст!\n"; }

}

void List::add\_after(int number, int data)

{

if (head != nullptr)

{

if (number >= 0 && number <= size)

{

if (number == size)

push\_back(data);

else if (number == 0)

push\_front(data);

else

{

if (number <= size / 2)

{

Node\* current = head;

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

current = current->pnext;

Node\* temp = current->pnext;

current->pnext = new Node(data);

current->pprev = current;

current = current->pnext;

current->pnext = temp;

++size;

}

else

{

Node\* current = tail;

for (int i = size; i > number; i--)

current = current->pprev;

Node\* temp = current->pnext;

Node\* temp2 = current;

current->pnext = new Node(data);

temp->pprev = current->pnext;

current = current->pnext;

current->pprev = temp2;

current->pnext = temp;

++size;

}

}

}

else { system("cls"); cout << "Ошибка ввода!\n\n"; }

}

else { system("cls"); cout << "Список пуст!\n\n"; }

}

void List::pop\_element(int number)

{

if (head != nullptr)

{

if (number >= 1 && number <= size)

{

if (number == size)

pop\_back();

else if (number == 1)

pop\_front();

else

{

if (number <= size / 2)

{

Node\* current = head;

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

current = current->pnext;

Node\* temp = current->pprev;

Node\* temp2 = current->pnext;

delete current;

temp->pnext = temp2;

temp2->pprev = temp;

}

else

{

Node\* current = tail;

for (int i = size; i > number; i--)

current = current->pprev;

Node\* temp = current->pprev;

Node\* temp2 = current->pnext;

delete current;

temp->pnext = temp2;

temp2->pprev = temp;

}

--size;

}

}

else { system("cls"); cout << "Ошибка ввода!\n\n"; }

}

else { system("cls"); cout << "Список пуст!\n\n"; }

}

int& List::operator[](const int index) const

{

if (index >= 0 && index <= size)

{

if (index <= size / 2)

{

Node\* current = head;

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

current = current->pnext;

return current->data;

}

else

{

Node\* current = tail;

for (int i = index; i > index; --i)

current = current->pprev;

return current->data;

}

}

}

int& List::operator()()

{

return size;

}

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

{

if (this != &object)

if (this->head != nullptr)

{

this->~List();

for (int i = 0; i < object.size; i++)

this->push\_back(object[i]);

}

else

for (int i = 0; i < object.size; i++)

this->push\_back(object[i]);

return \*this;

}

List List::operator\*(List& object)

{

if (object.size > size)

{

List temp;

Node\* current = (\*this).head;

for (int i = 0; i < object.size; i++)

{

if (current != nullptr)

temp.push\_back((\*this)[i] \* object[i]);

else

temp.push\_back(0);

if (current != 0)

current = current->pnext;

}

return temp;

}

else

{

List temp;

Node\* current = object.head;

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

{

if (current != nullptr)

temp.push\_back((\*this)[i] \* object[i]);

else

temp.push\_back(0);

if (current != 0)

current = current->pnext;

}

return temp;

}

}

Class Iterator

#include "List.h"

List::Iterator::Iterator()

{

it = nullptr;

}

List::Iterator::Iterator(const Iterator& it)

{

this->it = it.it;

}

bool List::Iterator::operator==(const Iterator& it)

{

return this->it = it.it;

}

bool List::Iterator::operator!=(const Iterator& it)

{

return this->it != it.it;

}

void List::Iterator::operator++()

{

it = it->pnext;

}

void List::Iterator::operator--()

{

it = it->pprev;

}

int List::Iterator::operator\*()

{

return this->it->data;

}