ПНИПУ

Федеральное государственное автономное образовательное учреждение высшего образования «Пермский национальный исследовательский политехнический университет»

Лабораторная работа №6.

АТД. Контейнеры.

Выполнил студент группы РИС-23-3Б

Буковский Денис Владимирович

Проверила доцент кафедры ИТАС О.А. Полякова

2024

1. 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.

Класс- контейнер СПИСОК с ключевыми значениями типа int.

Реализовать операции:

[] - доступа по индексу;

int() — определение размера списка;

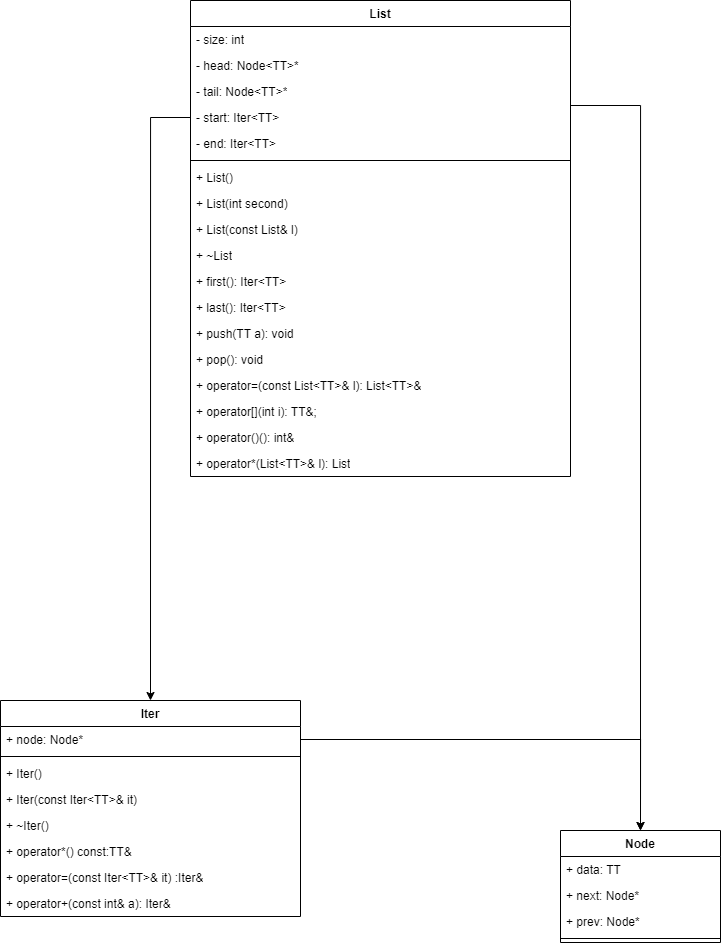
\* вектор — умножение элементов списков a[i]\*b[i];

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

1. Анализ задачи

Создадим структуру – Node (элемент списка), классы Iter – итератор списка и сам список – List. Опишем стандартные для этой структуры данных методы, а также методы из задания. Протестируем программу в функции main().

1. Блок-схема



1. Код

List.h

#pragma once

const int MAX\_SIZE = 25;

template <typename TT>

class Node {

public:

TT data;

Node\* next = nullptr;

Node\* prev = nullptr;

};

template <typename TT>

class Iter

{

public:

Node<TT>\* node;

Iter();

Iter(const Iter<TT>& it);

~Iter() { }

Iter& operator=(const Iter<TT>& a);

TT& operator\*() const;

Iter& operator+ (const int& a);

};

template <typename TT>

class List

{

protected:

int size;

Node<TT>\* head = nullptr;

Node<TT>\* tail = nullptr;

Iter<TT> start;

Iter<TT> end;

public:

List() { size = 0; head = nullptr; tail = nullptr; };

List(int second);

List(const List<TT>& l);

~List();

Iter<TT> first();

Iter<TT> last();

void push(TT a);

void pop();

List& operator= (const List<TT>& l);

TT& operator[] (int i);

int& operator()();

List operator\*(List<TT>& l);

};

template <typename TT>

List<TT>::List(int second)

{

size = second;

Node<TT>\* new\_elem = new Node<TT>;

head = new\_elem;

tail = new\_elem;

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

{

Node<TT>\* new\_elem = new Node<TT>;

tail->next = new\_elem;

new\_elem->prev = tail;

tail = new\_elem;

}

start.node = head;

end.node = tail;

}

template <typename TT>

List<TT>::List(const List<TT>& l)

{

Node<TT>\* node = l.head;

while (node != nullptr) {

push(node->data);

node = node->next;

}

start.node = head;

end.node = tail;

}

template <typename TT>

List<TT>::~List()

{

Node<TT>\* curr = head;

while (curr != nullptr)

{

head = curr->next;

delete curr;

curr = head;

}

}

template <typename TT>

List<TT>& List<TT>::operator= (const List<TT>& l)

{

if (this == &l)return \*this;

if (head != nullptr)

{

Node<TT>\* curr = head;

while (curr != nullptr)

{

head = curr->next;

delete curr;

curr = head;

}

}

Node<TT>\* node = l.head;

while (node != nullptr) {

push(node->data);

node = node->next;

}

start.node = head;

end.node = tail;

return \*this;

}

template <typename TT>

TT& List<TT>::operator[] (int i)

{

Node<TT>\* curr = head;

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

curr = curr->next;

}

return curr->data;

}

template <typename TT>

List<TT> List<TT>::operator\*(List<TT>& l)

{

List base = \*this;

int second = this->size;

List temp;

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

{

temp.push(base[i] \* l[i]);

}

return temp;

}

template <typename TT>

int& List<TT>::operator()() {

return size;

}

template <typename TT>

Iter<TT> List<TT>::first()

{

return start;

}

template <typename TT>

Iter<TT> List<TT>::last()

{

return end;

}

template <typename TT>

void List<TT>::push(TT a)

{

Node<TT>\* new\_elem = new Node<TT>;

new\_elem->data = a;

if (size == 0)

{

head = new\_elem;

tail = new\_elem;

size++;

start.node = head;

end.node = tail;

}

else

{

tail->next = new\_elem;

new\_elem->prev = tail;

tail = new\_elem;

size++;

end.node = tail;

}

}

template <typename TT>

void List<TT>::pop()

{

Node<TT>\* curr = tail;

tail = curr->prev;

delete curr;

tail->next = nullptr;

size--;

end.node = tail;

}

template <typename TT>

Iter<TT>::Iter() {

node = nullptr;

}

template <typename TT>

Iter<TT>::Iter(const Iter& it) {

node = it.node;

}

template <typename TT>

Iter<TT>& Iter<TT>::operator=(const Iter<TT>& a)

{

node = a.node;

return \*this;

}

template <typename TT>

TT& Iter<TT>::operator\*() const

{

return node->data;

}

template <typename TT>

Iter<TT>& Iter<TT>::operator+ (const int& a)

{

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

node = node->next;

}

return \*this;

}

Main.cpp

#include <iostream>

#include "List.h"

using namespace std;

int main()

{

int count, in\_value, index;

List<int> list1;

Iter<int> iter;

cout << "Enter quantity of elements: ";

cin >> count;

cout << endl;

cout << "Enter elements:" << endl;

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

{

cin >> in\_value;

list1.push(in\_value);

}

cout << endl;

iter = list1.first();

cout << "Enter element's index: ";

cin >> index;

cout << list1[index] << endl;

cout << endl;

cout << "Enter number to move for from first element: ";

cin >> index;

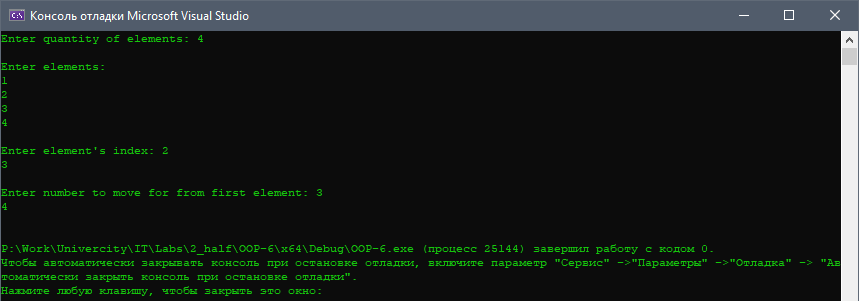
iter + index;

cout << \*iter << endl;

cout << endl;

}

1. Вывод программы



1. Ответы на контрольные вопросы.

1. АТД — это тип данных, определяемый только через его функционал. АТД может быть список, вектор, дерево и т.д.

2. Элемент списка – узел.

3. Класс списка.

4. Контейнер – это набор однотипных элементов, пример – список.

5. В контейнерах выделяют следующие операции: изменение содержимого, поиск, сортировка.

6. Доступ к элементам может быть прямым и ассоциативным (по содержимому), а также последовательным (через итератор). Примеры – arr[4] – прямой доступ.

7. Итератор – объект, который позволяет хранить в себе один элемент контейнера и перемещаться между элементами.

8. Итератор может быть реализован как класс или структура.

9. Объединение контейнеров можно организовать помещением всех элементов двух контейнеров в один новый.

10. Контейнер, содержащий пары “ключ-значение” предоставляет ассоциативный доступ по ключу.

11. Контейнер, в котором вставка и удаление элементов выполняется на одном конце контейнера называется стеком.

12. d.

13. d.

14. Прямым.

15. Последовательным.