Институт приборостроения, автоматизации и информационных технологий

Кафедра информационных систем и цифровых технологий

Дисциплина «Алгоритмы и структуры данных»

Отчет к лабораторной работе № 2

«Многосвязный список»

Выполнил:

Василения Иван Валерьевич

Мельниченко Артём Олегович

Принял:

Рыженков Д.В.

Орёл, 2024г

Листинг файла main.cpp:

#include <iostream>

#include "List.h"

using namespace std;

class Student {

public:

    string family = "Ivanov";

    int ex1 = 5;

    int ex2 = 5;

    int ex3 = 5;

    bool hasHonorCertificate = false;

    string homeCity = "Moscow";

    bool needDormitory = false;

    string getKey() {

        return family;

    }

};

enum Descriptor {

    all = 0,

    exams = 1,

    certificate = 2,

    city = 3,

    dorimotory = 4

};

void showMenu() {

    cout << "1. Ввод записи об абитуриенте\n";

    cout << "2. Вывод подсписка\n";

    cout << "3. Вывод полного списка\n";

    cout << "4. Удаление записи об абитуриенте\n";

    cout << "5. Удаление всех записей\n";

    cout << "0. Выход\n";

}

void inputStudent(List<Student>& list) {

    // Ввод и заполнение Student, обработка ошибок всего такого

    // Для теста использую дефолтного

    Student st;

    // Сохраняю указатель на ноду для подсписков

    // Вставка в подсписки

    cout << "Введите фамилию студента" << endl;

    cin >> st.family;

    cout << "Введите оценки за три экзамена" << endl;

    cin >> st.ex1;

    cin >> st.ex2;

    cin >> st.ex3;

    cout << "Есть ли у студента аттестат с отличием (Д/Н)" << endl;

    char cert;

    cin >> cert;

    cout << "Введите название населенного пункта, в котором проживает студент" << endl;

    cin >> st.homeCity;

    if (cert == 'д' || cert == 'Д' || cert == 'Y' || cert == 'y') {

        st.hasHonorCertificate = true;

    }

    else {

        st.hasHonorCertificate = false;

    }

    Node<Student>\* node = list.insert(st); //Вставка в главный список

    if (st.ex1 == 5 && st.ex2 == 5 && st.ex3 == 5) { // Отличник

        list.insertByDescriptor(node, Descriptor::exams);

    }

    if (st.hasHonorCertificate) { // Аттестат с отличием

        list.insertByDescriptor(node, Descriptor::certificate);

    }

    if (st.homeCity != "Orel" && st.homeCity != "Орёл") { // Иногородний

        list.insertByDescriptor(node, Descriptor::city);

    }

    if (st.needDormitory) { // Нуждается в общаге

        list.insertByDescriptor(node, Descriptor::dorimotory);

    }

}

void insertStudent(List<Student>& list, Student st) { //для отладки нужна была

    Node<Student>\* node = list.insert(st); //Вставка в главный список

    if (st.ex1 == 5 && st.ex2 == 5 && st.ex3 == 5) { // Отличник

        list.insertByDescriptor(node, Descriptor::exams);

    }

    if (st.hasHonorCertificate) { // Аттестат с отличием

        list.insertByDescriptor(node, Descriptor::certificate);

    }

    if (st.homeCity != "Orel" && st.homeCity != "Орёл") { // Иногородний

        list.insertByDescriptor(node, Descriptor::city);

    }

    if (st.needDormitory) { // Нуждается в общаге

        list.insertByDescriptor(node, Descriptor::dorimotory);

    }

}

void showSubList(List<Student>& list) {

    char cmd;

    vector<Student> students;

    cout << "1. Вывести отличников" << endl;

    cout << "2. Вывести владельцев красного аттестата" << endl;

    cout << "3. Вывести иногородних" << endl;

    cout << "4. Вывести нуждающихся в общежитии" << endl;

    cin >> cmd;

    switch (cmd) {

    case '1':

        students = list.getAll(Descriptor::exams);

        break;

    case '2':

        students = list.getAll(Descriptor::certificate);

        break;

    case '3':

        students = list.getAll(Descriptor::city);

        break;

    case '4':

        students = list.getAll(Descriptor::dorimotory);

        break;

    default:

        cout << "Неверный номер подсписка" << endl;

    }

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

    {

        cout << students[i].family << endl;

    }

}

void showList(List<Student>& list) {

    for (auto i : list.getAll()) {

        cout << i.family << endl;

    }

}

void deleteNode(List<Student>& list) {

    string st;

    cout << "Введите ключ записи, которую надо удалить" << endl;

    cin >> st;

    list.deleteNode(st);

}

void deleteAll(List<Student>& list) {

    list.clear();

}

int main() {

    setlocale(LC\_ALL, "Russian");

    List<Student> list;

    // Вставка дескрипторов подсписков

    list.insertDescriptor(); // exams

    list.insertDescriptor(); // certificate

    list.insertDescriptor(); // city

    list.insertDescriptor(); // dorimotory

    char cmd;

    do {

        system("cls");

        showMenu();

        cin >> cmd;

        switch (cmd) {

        case '1':

            // Добавить информацию о студенте

            inputStudent(list);

            break;

        case '2':

            // Вывод подсписков

            showSubList(list);

            break;

        case '3':

            // Вывод полного списка

            showList(list);

            break;

        case '4':

            // Удаление записи

            deleteNode(list);

            break;

        case '5':

            // Удаление всех записей

            deleteAll(list);

            break;

        case '0':

            cout << "Выход из программы.\n";

            break;

        default:

            cout << "Неверный выбор. Попробуйте еще раз.\n";

            break;

        }

        system("pause");

    } while (cmd != '0');

}

Листинг файла List.h:

#pragma once

#include <vector>

using namespace std;

template <typename T>

struct Node {

    T data;

    std::vector<Node<T>\*> links;

    Node(T val) : data(val) {}

    Node(){}

};

template <typename T>

class ListDescriptor {

public:

    Node<T>\* first = nullptr;

    Node<T>\* last = nullptr;

    void clear() {

        first = nullptr;

        last = nullptr;

    }

};

template <typename T>

class List {

private:

    std::vector<ListDescriptor<T>> descriptors;

public:

    List() {

        ListDescriptor<T> ls;

        descriptors.push\_back(ls);

    }

    // Вставка значения в список

    Node<T>\* insert(T val) {

        Node<T>\* newNode = new Node<T>(val);

        if (isEmpty()) {

            descriptors[0].first = newNode;

            descriptors[0].last = newNode;

            return newNode;

        }

        Node<T>\* current = descriptors[0].last;

        current->links.push\_back(newNode);

        descriptors[0].last = newNode;

        return newNode;

    }

    // enter desc

    void insertDescriptor() {

        ListDescriptor<T> desc;

        descriptors.push\_back(desc);

    }

    // Вставка узла в конкретный подсписок

    bool insertByDescriptor(Node<T>\* newNode, int descriptorIndex) {

        if (descriptorIndex >= descriptors.size()) {

            return false;

        }

        if (descriptors[descriptorIndex].first == nullptr) {

            descriptors[descriptorIndex].first = newNode;

            descriptors[descriptorIndex].last = newNode;

            return true;

        }

        Node<T>\* last = descriptors[descriptorIndex].last;

        while (last->links.size() <= descriptorIndex) {

            last->links.push\_back(nullptr);

        }

        last->links[descriptorIndex] = newNode;

        descriptors[descriptorIndex].last = newNode;

    }

    // Нужны комментарии?

    bool isEmpty() {

        return descriptors[0].first == nullptr;

    }

    // Найти узел по значению key. Опционально - указатель на предыдущий узел и дескрпитор подсписка

    Node<T>\* findNode(string key, Node<T>\*\* prevNode = nullptr, int descriptorIndex = 0) {

        if (isEmpty()) { return nullptr; }

        Node<T>\* tmp = descriptors[descriptorIndex].first;

        if (!tmp) { return nullptr; }

        while (tmp->links.size() >= descriptorIndex+1 && key != tmp->data.getKey())

        {

            if (prevNode) {

                \*prevNode = tmp;

            }

            if (tmp->links[descriptorIndex] == nullptr) {

                break;

            }

            tmp = tmp->links[descriptorIndex];

        }

        if (tmp->data.getKey() == key) {

            return tmp;

        }

        return nullptr;

    }

    // Удалить по значению key из всех подсписков

    bool deleteNode(string key) {

        bool success = false;

        for (int i = descriptors.size()-1; i >= 0 ; i--)

        {

            if (deleteNodeByDescriptor(key, i)) {

                success = true; // if any deleted

            }

        }

        return success;

    }

    // Удалить по значению key из подсписка

    bool deleteNodeByDescriptor(string key, int descriptorIndex = 0) {

        Node<T>\* prevNode = nullptr;

        Node<T>\* node = findNode(key, &prevNode, descriptorIndex);

        if (!node) { return false; }

        if (prevNode) { // node is not first

            if (node->links.size() >= descriptorIndex + 1) { // node has links forward

                prevNode->links[descriptorIndex] = node->links[descriptorIndex];

                node->links[descriptorIndex] = nullptr;

            }

            else {

                prevNode->links[descriptorIndex] = nullptr;

            }

            if (descriptors[descriptorIndex].last = node) { // prevNode now must be last

                descriptors[descriptorIndex].last = prevNode;

            }

            while (node->links.size() > 0) // while last link is null -> delete it

            {

                if (node->links[node->links.size() - 1] != nullptr) { break; }

                node->links.pop\_back();

            }

            if (descriptorIndex == 0) {// delete from main List

                delete node;

            }

            return true;

        }

        else { // node is first

            Node<T>\* next = nullptr;

            if (node->links.size() >= descriptorIndex + 1) { //if has links forward

                next = node->links[descriptorIndex];

            }

            descriptors[descriptorIndex].first = next;

            if (descriptors[descriptorIndex].first == nullptr) { // if first null -> then last also must be null

                descriptors[descriptorIndex].last = nullptr;

            }

            if (descriptorIndex == 0) {

                delete node;

            }

            return true;

        }

    }

    // Получить вектор элементов подсписка

    vector<T> getAll(int descriptorIndex = 0) {

        vector<T> resp;

        Node<T>\* tmp = descriptors[descriptorIndex].first;

        if (!tmp) { return {}; }

        while (tmp->links.size() >= descriptorIndex+1)

        {

            resp.push\_back(tmp->data);

            if (tmp->links[descriptorIndex] == nullptr) {

                return resp;

            }

            tmp = tmp->links[descriptorIndex];

        }

        resp.push\_back(tmp->data);

        return resp;

    }

    // Очистка

    void clear() {

        if (isEmpty()) { return; }

        Node<T>\* head = descriptors[0].first;

        while (head->links.size() > 0) {

            Node<T>\* tmp = head->links[0];

            delete head;

            head = tmp;

        }

        delete head;

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

        {

            descriptors[i].clear();

        }

    }

};