Министерство образования и науки Российской Федерации

Федеральное государственное образовательное автономное учреждение высшего образования

"Пермский национальный исследовательский политехнический университет"

ОТЧЕТ

ПО ЛАБОРАТОРНОЙ РАБОТЕ №11

Дисциплина: Основы алгоритмизации и программирования

Тема: Информационные динамические структуры данных

|  |  |
| --- | --- |
| 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**

Записи в линейном списке содержат ключевое поле типа \*char(строка символов). Сформировать двунаправленный список, однонаправленный список, стек и очередь. Удалить элемент с заданным номером. Добавить по К элементов перед и после элемента с заданным ключом.

**Анализ задачи**

Для реализации данной задачи необходимо подключить библиотеки iostream – для работы с консолью, fstream - для работы с файлами. Для реализации каждой структуры данных необходимо реализовать классы двунаправленных и однонаправленных списков, стека и очереди. Для этого необходимо реализовать поля и методы:

Вложенный класс, Node, который хранить указатель на предыдущий и на следующий элемент, указатель на первый элемент, и количество элементов в структуре данных – это поля, которые необходимо реализовать для каждой структуры данных. Также для двусвязного списка и очереди необходимо реализовать поле, которое будет хранить указатель на последний элемент.

Однонаправленный список:

push\_back – добавление элемента в конец списка

GetSize – возврат количества элементов в списке, так как прямой доступ нам запрещают принципы инкапсуляции.

operator[] – перегрузить оператор [] для того, чтобы мы могли взаимодействовать со списком как с массивом, а точнее индексироваться по списку

pop\_front - удаление элемента из начала списка

void clear - удаление всех элементов списка

push\_front - добавление элемента в начало списка

insert - добавление элемента по индексу

removeA - удаление элемента по индексу

pop\_back - удаление из элемента из конца списка

writeFile - запись списка в файл

readFile - чтение списка из файла

removeFewAt - удаление нескольких элементов из списка

Для двусвязного списка необходимо реализовать точно такие же методы.

Стек:

GetSize – возврат количества элементов в списке, так как прямой доступ нам запрещают принципы инкапсуляции.

Del - удаление элемента из стека

getData – получение данных с верхушки стека

clear - удаление всех элементов их стека

add - добавление элемента в стек

writeFile - запись стека в файл

readFile - чтение стека из файла

Очередь:

GetSize – возврат количества элементов в списке, так как прямой доступ нам запрещают принципы инкапсуляции.

Clear – удаление всех элементов очереди

add - добавление элемента в очередь

del - удаление элемента из очереди

writeFile - запись очереди в файл

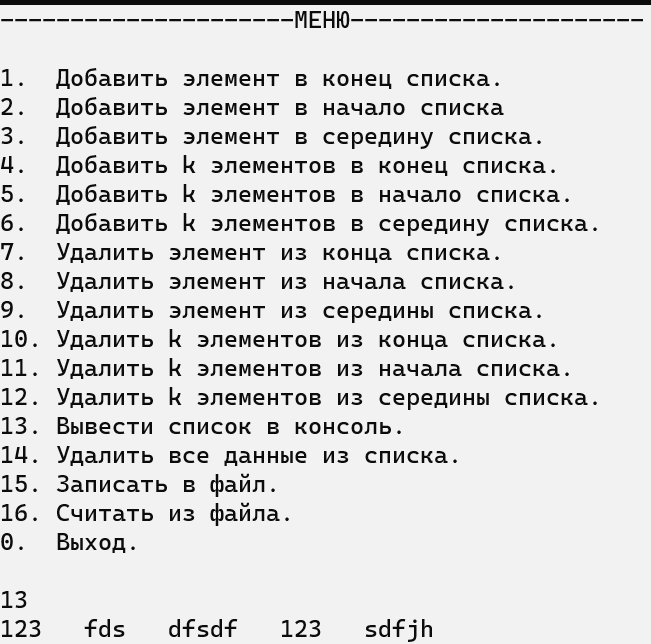
readFile - чтение очереди из файла

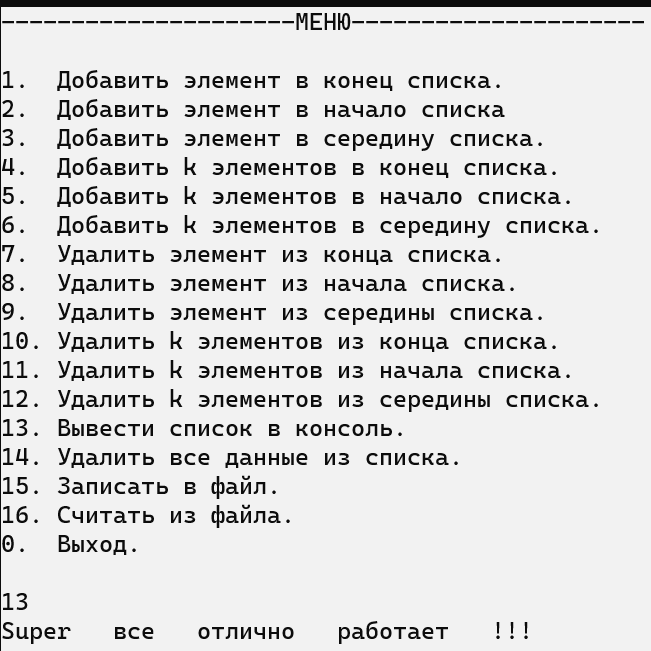
getDataEnd - получение данных из конца очереди

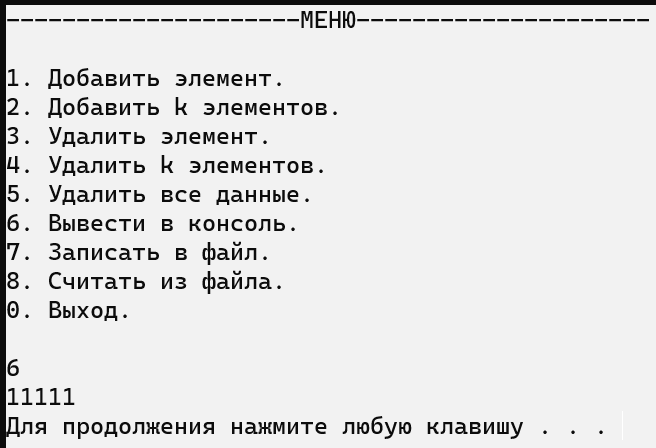
getDataStart - получение данных из начала очереди

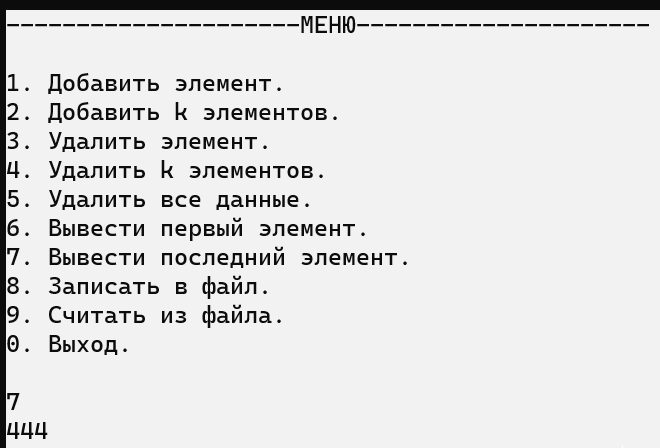
**Тестирование программы**

Для того, чтобы протестировать программу, необходимо протестировать работу всех методов для каждой структуры данных.

*Рис. 1 – Тестирование методов для односвязного списка.*

*Рис. 2 – Тестирование методов двусвязного списка.*

*Рис 3. – Тестирование методов стека.*

*Рис. 4 – Тестирование методов очереди.*

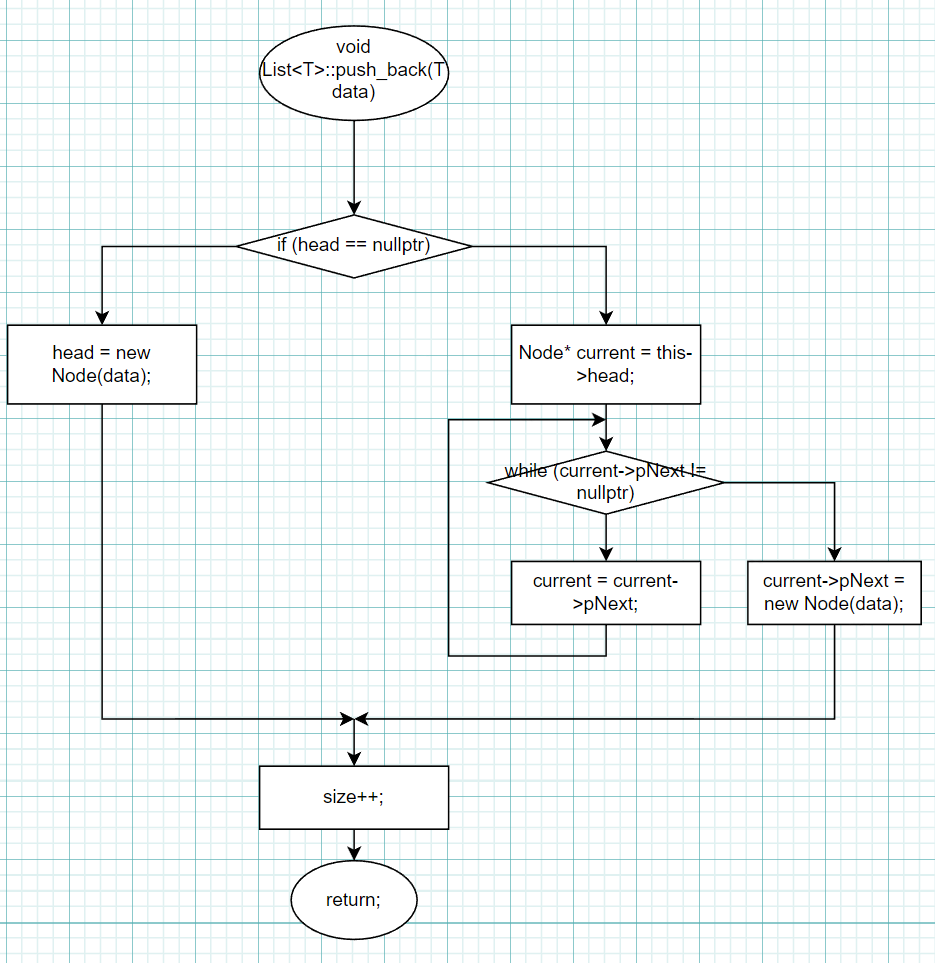
**Заключение**

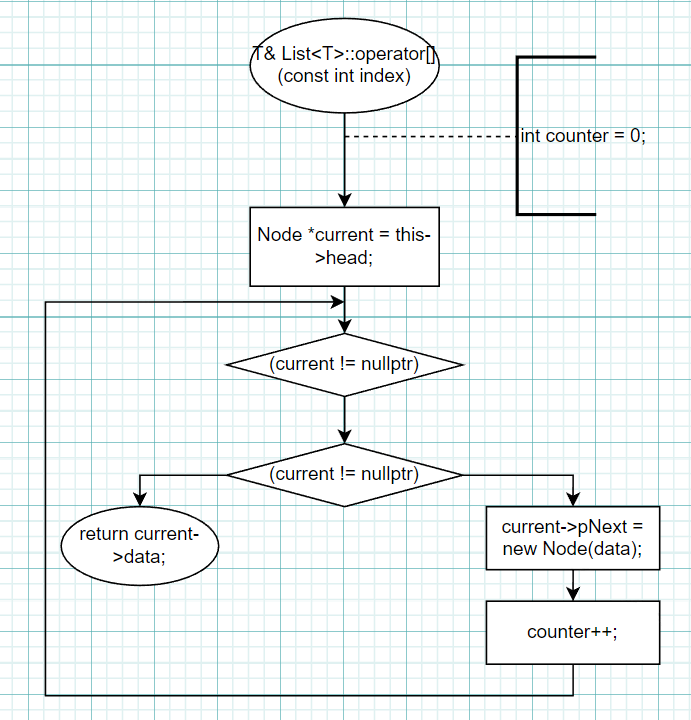
Были разработаны структуры данных, которые могут взаимодействовать с любыми типами данных, так как являются шаблонными. Данные структуры данных могут широко использоваться в различных программах для хранения данных. Односвязный список удобен тем, что при большом количестве операций, по увеличению и уменьшению длины они работают гораздо быстрее, двунаправленный список улучшает однонаправленный и благодаря этому можно быстрее получать значения, которые находятся ближе к концу. Стек удобен для хранения данных, если нужен доступ только к последнему элементу, а очередь если нужен доступ к первому и последнему элементу. Нет наилучшего способа организации данных, для каждой отдельной задачи, может лучше или хуже подходить определенная структура данных.

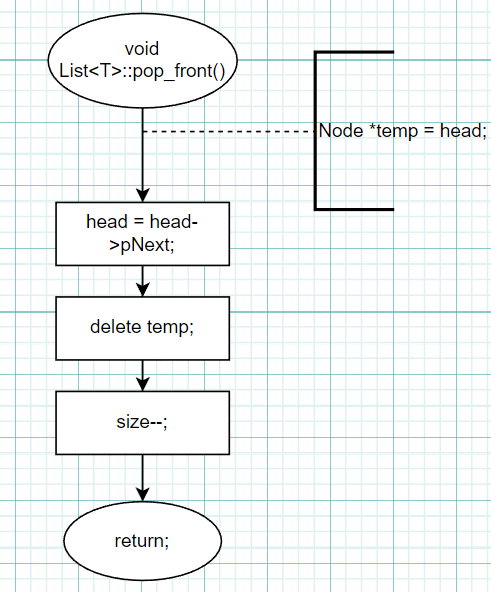
**Приложения**

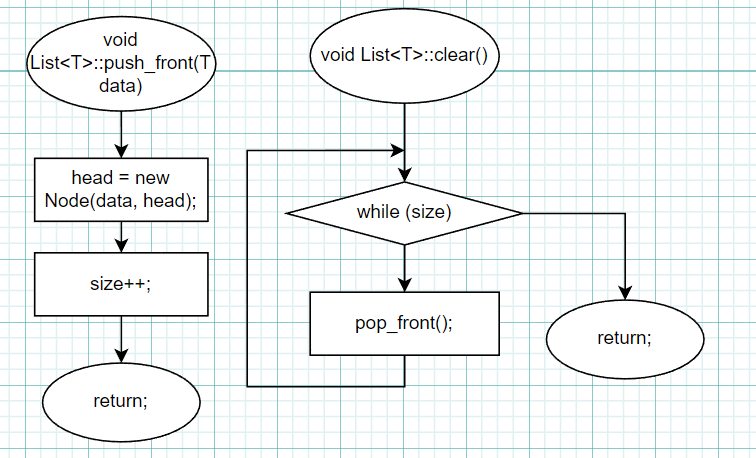
Приложение А – блок схема программы.

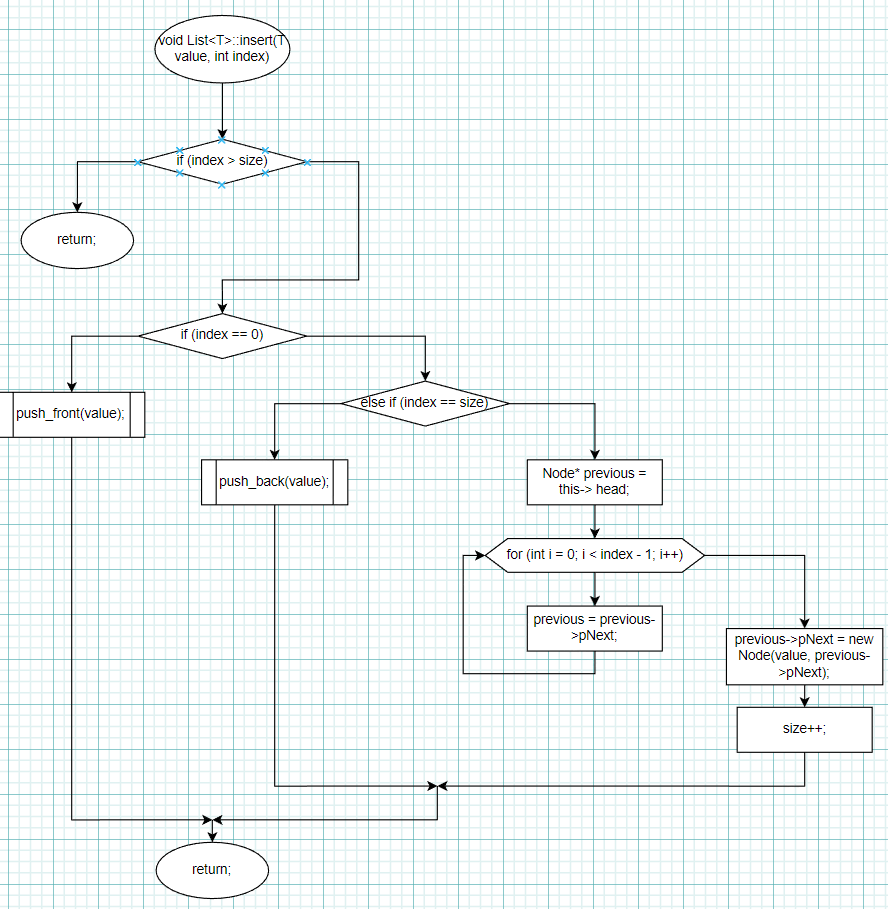
Односвязный список:

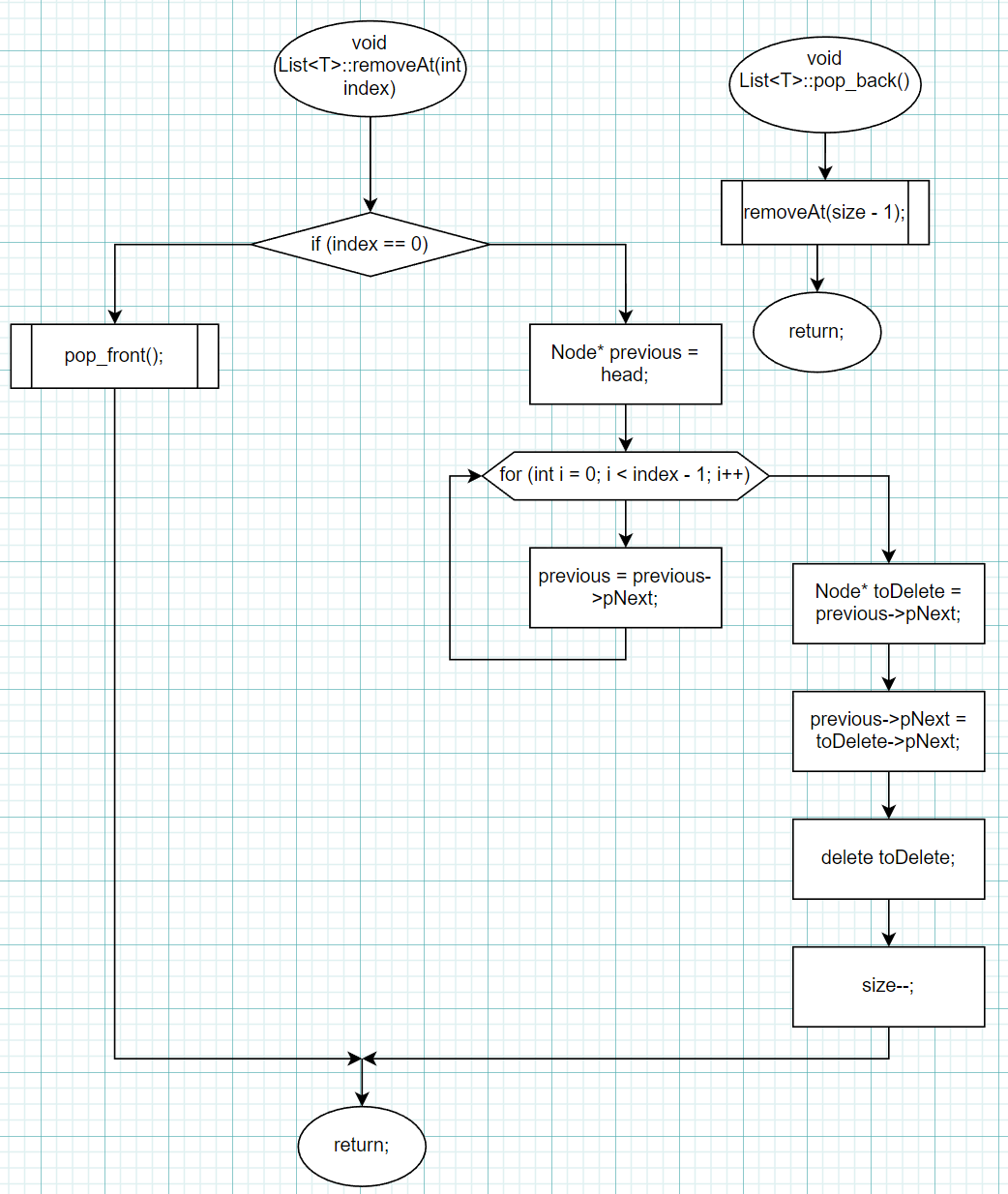


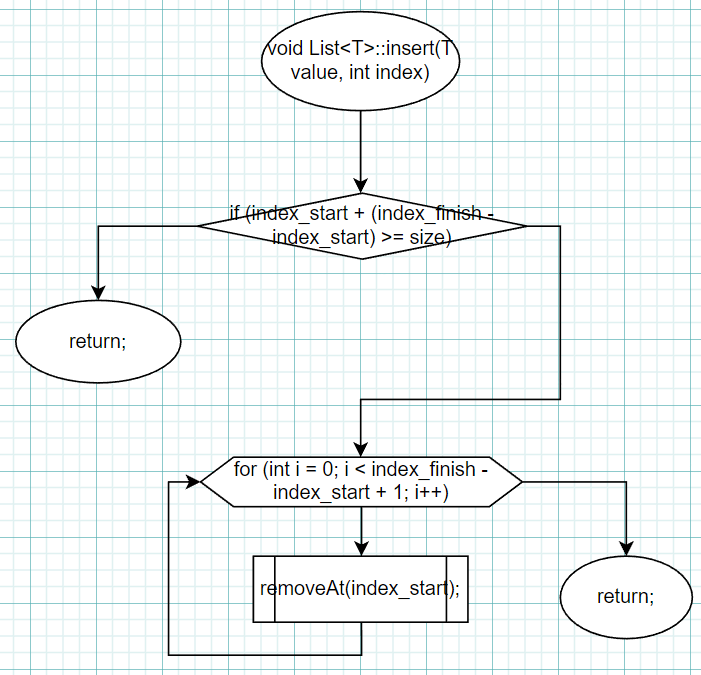


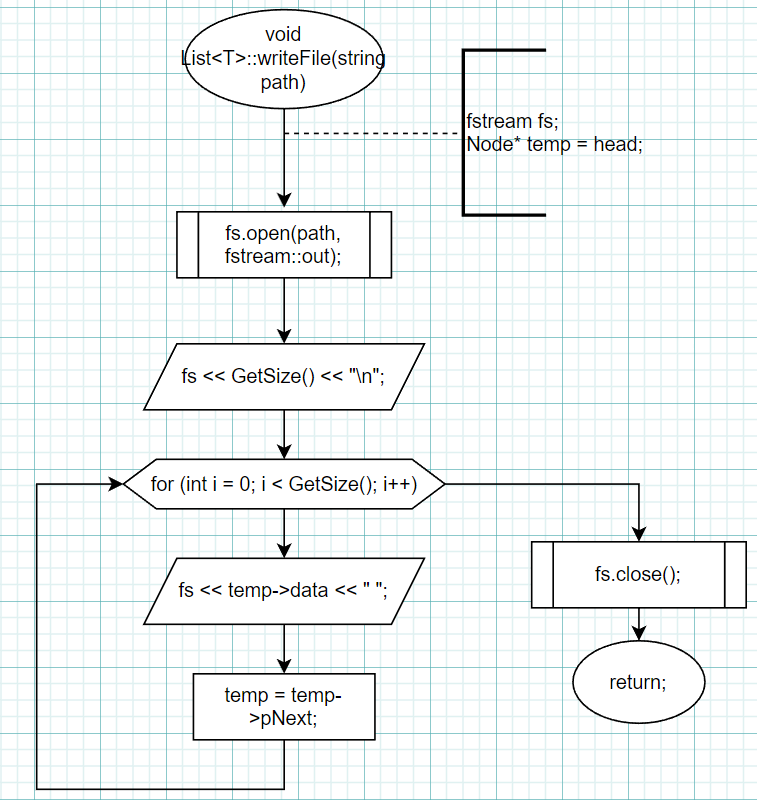


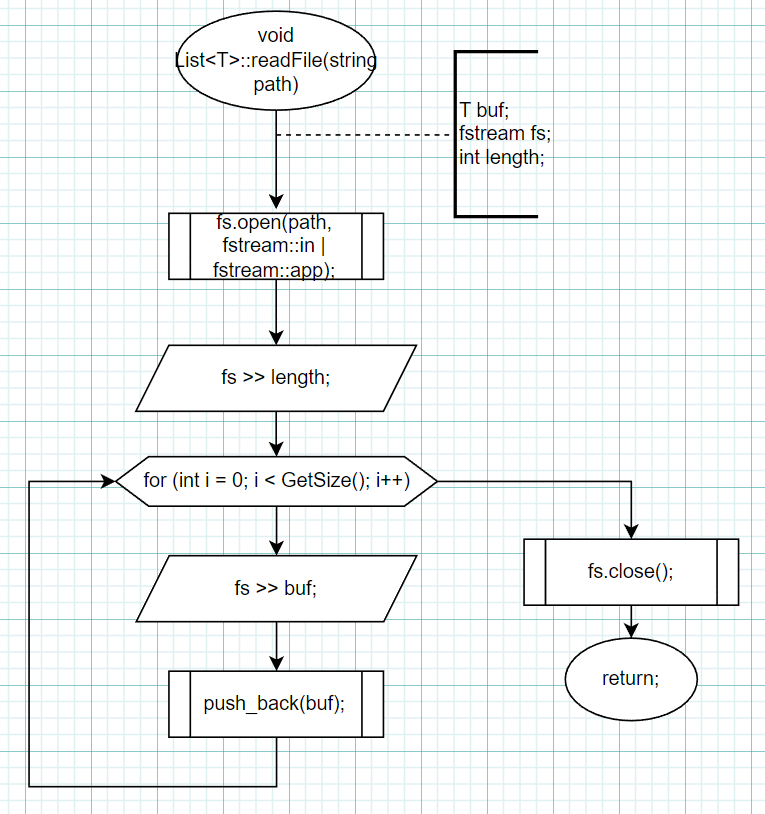




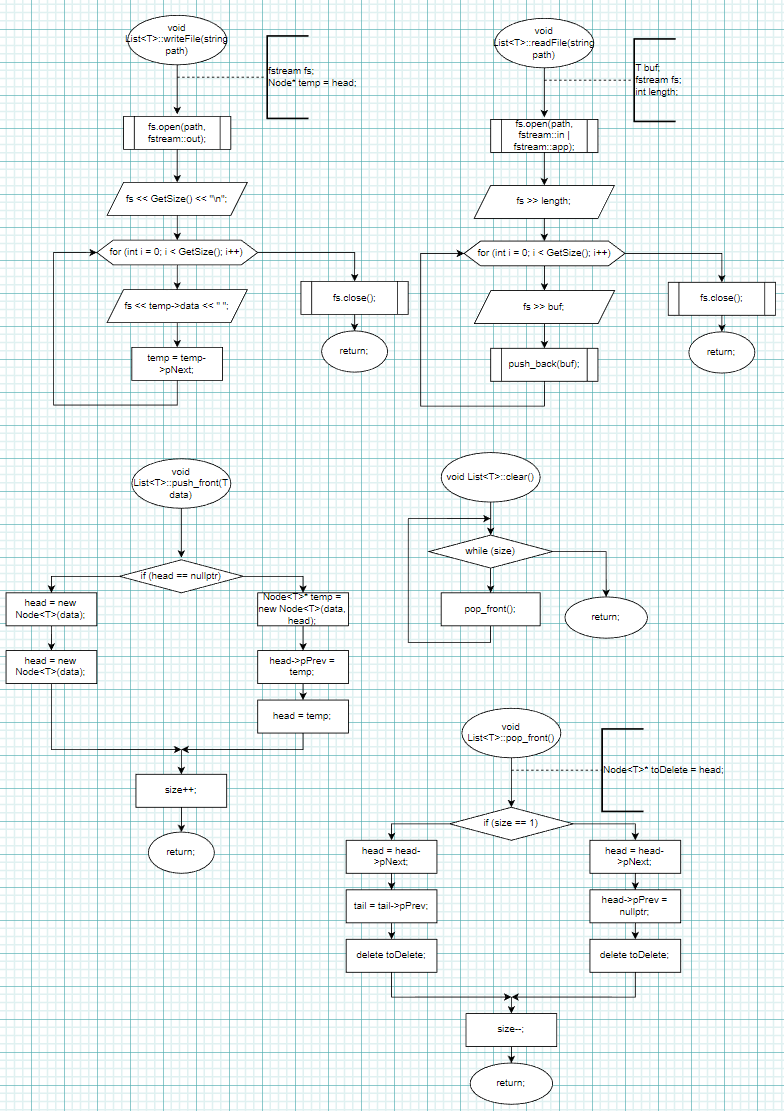


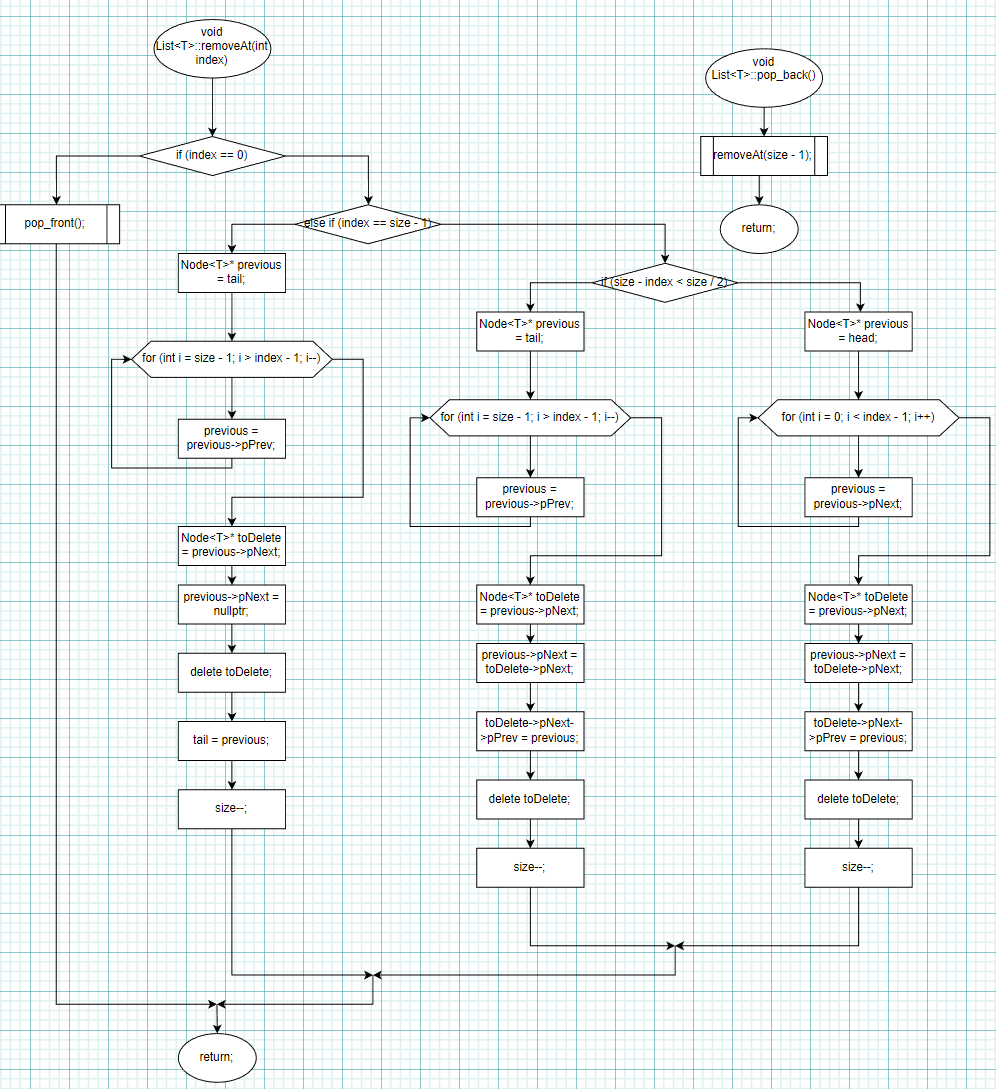


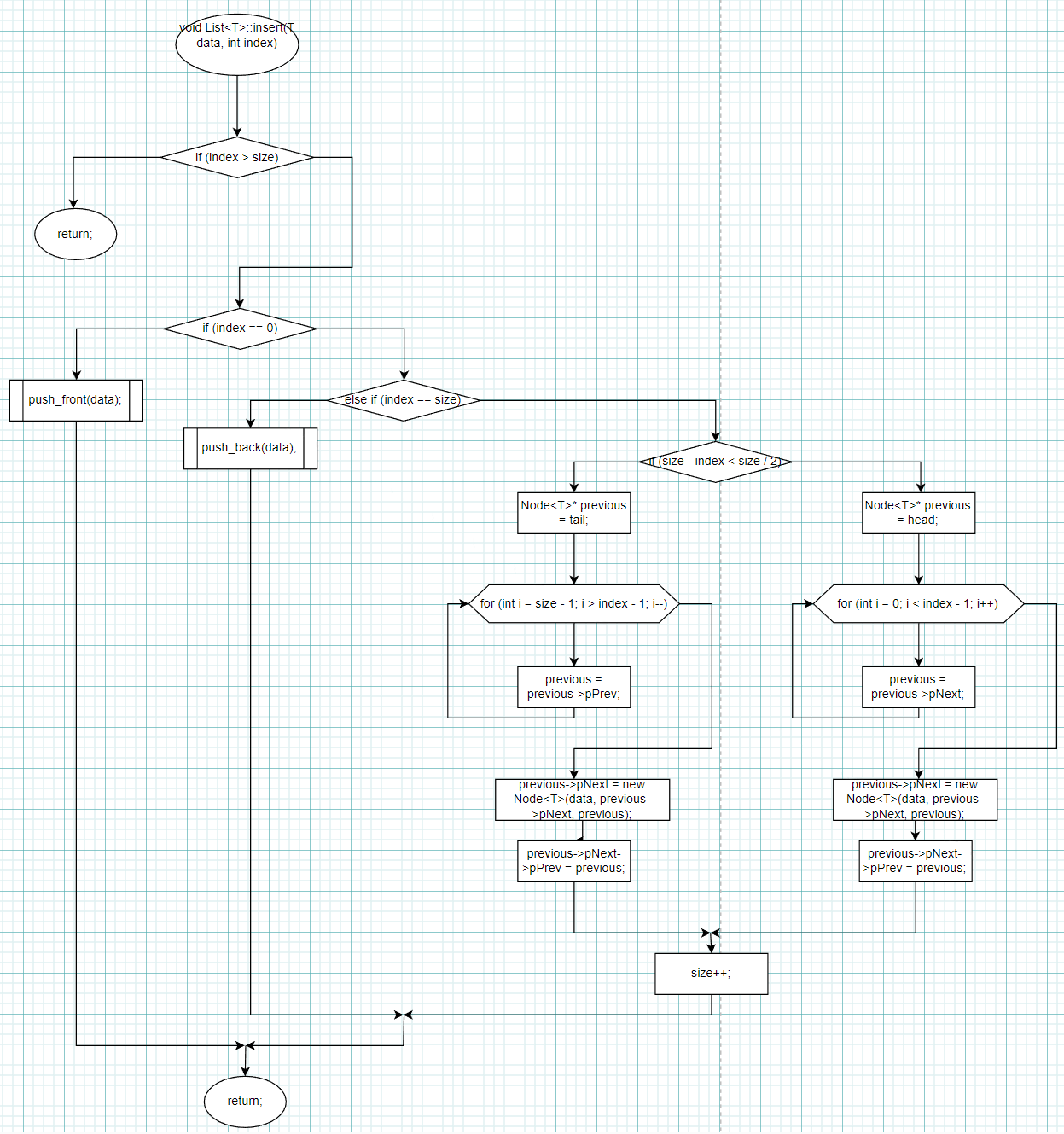


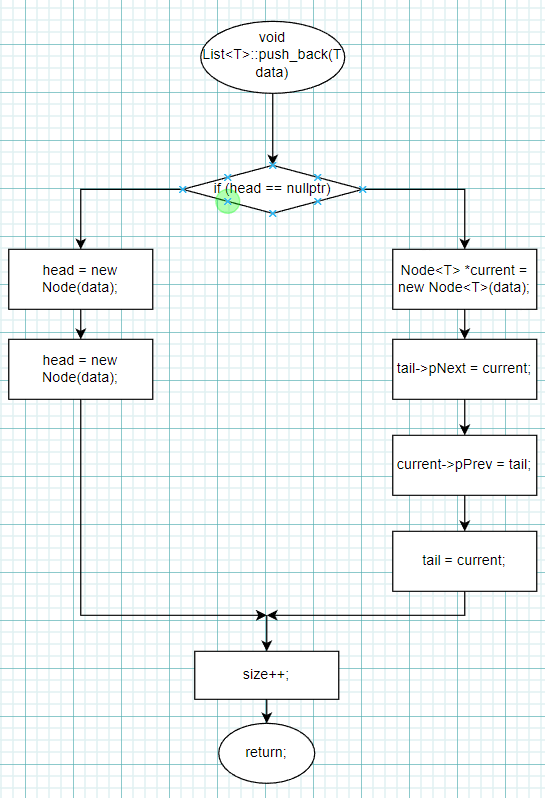


Двунаправленный список:

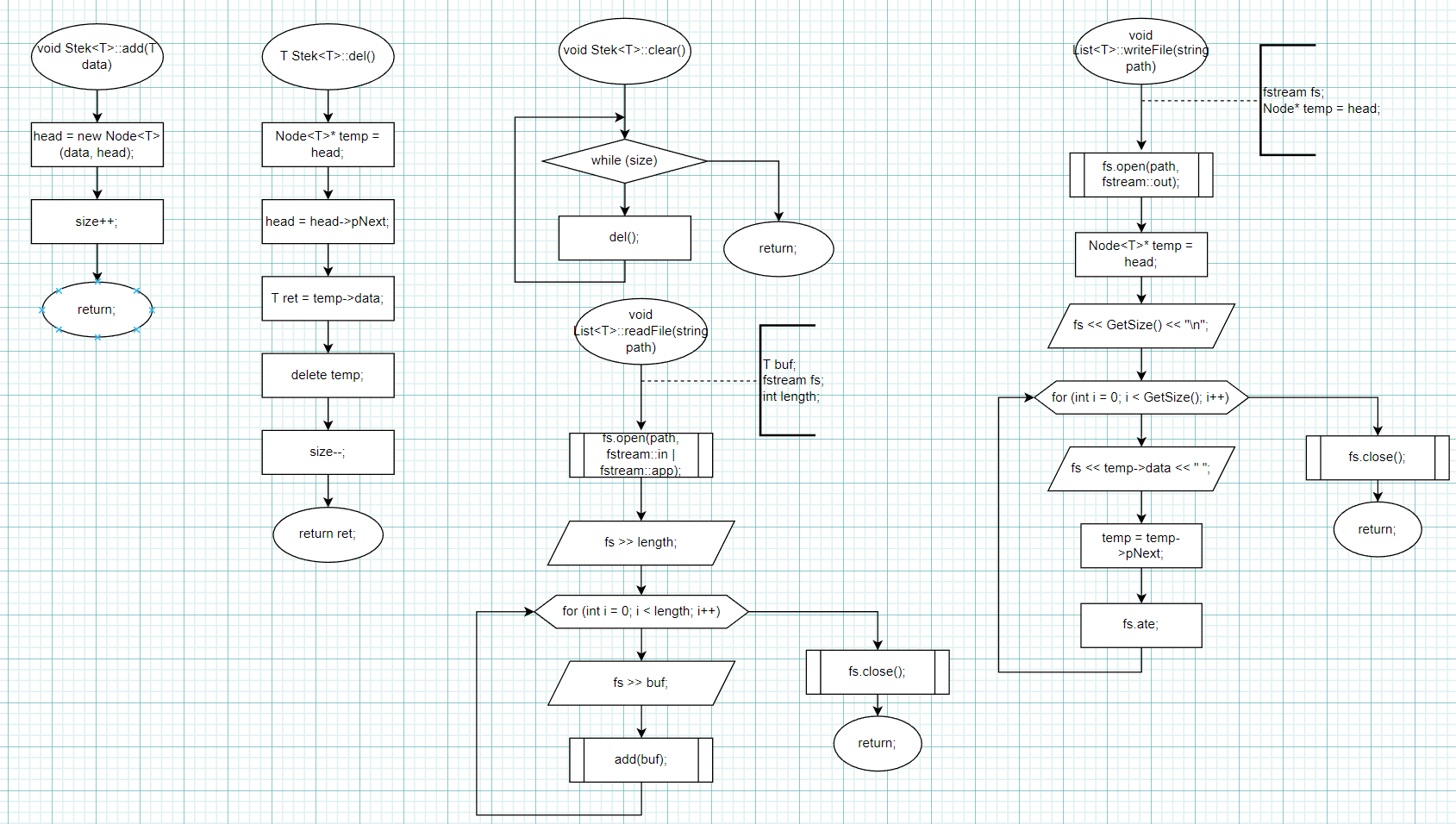




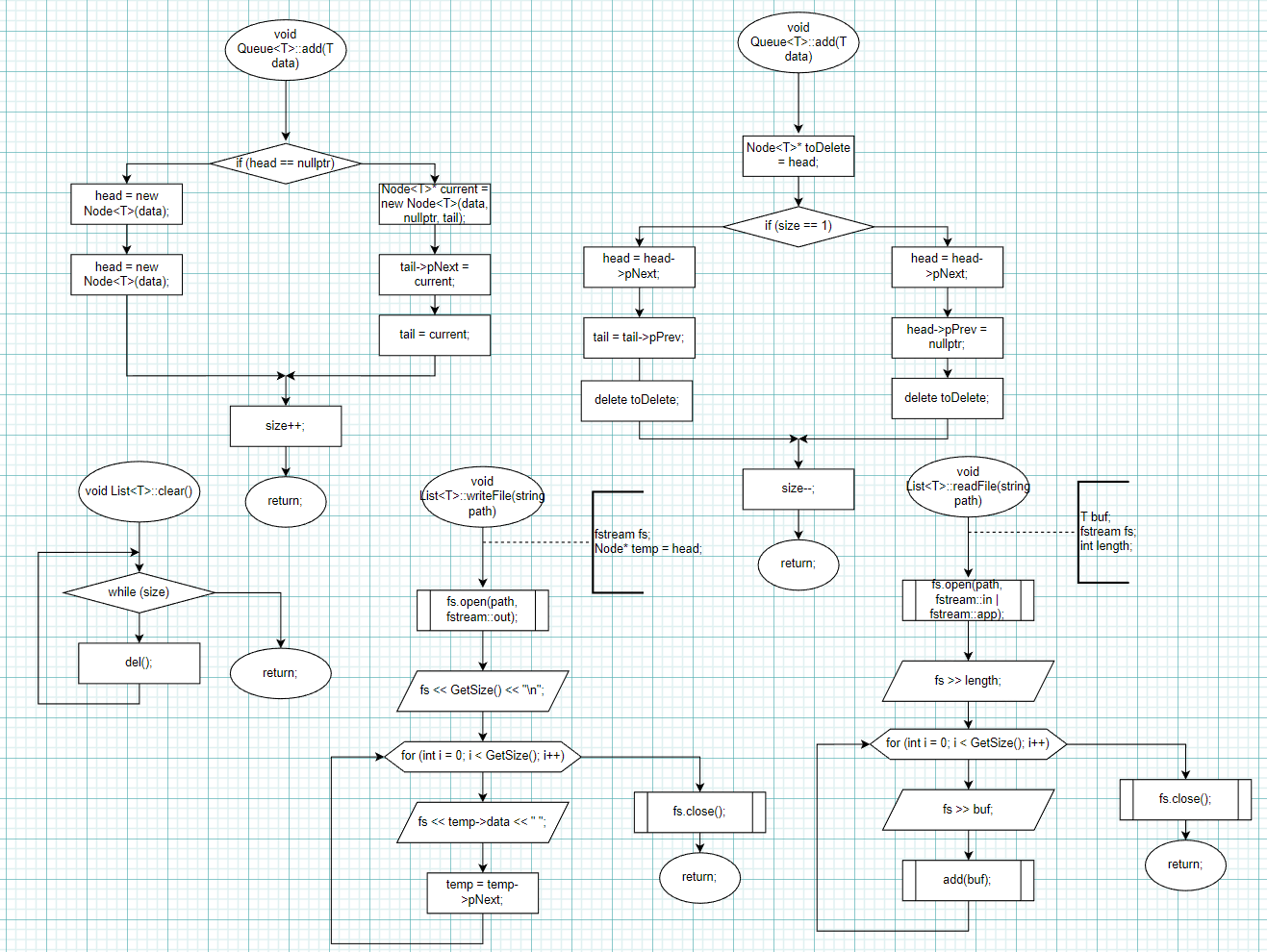




Стек:



Очередь:



Приложение Б – код программы.

Односвязный список:

template <typename T>

class List

{

private:

//template<typename T> //минимальная ячейка памяти

class Node //вложенный класс - ячейка памяти

{

public:

Node(T data = T(), 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

T data; //хранит наши данные

};

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

int size; //количество элементов в списке.

public:

List(); //Конструктор

~List(); //Деструктор

void push\_back(T 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

T& 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(T data); //добавляет элемент в начало списка

void insert(T value, int index); //добавление элемента по индексу

void removeAt(int index); //удаление элемента по указаному индексу

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

void writeFile(string path); //запись в файл

void readFile(string path); //чтение из файла

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

};

template<typename T>

void List<T>::writeFile(string path) //запись в файл

{

fstream fs;

fs.open(path, fstream::out);

Node\* temp = head;

fs << GetSize() << "\n"; //запись в файл размера списка

for (int i = 0; i < GetSize(); i++) //запись данных в файл

{

fs << temp->data << " ";

temp = temp->pNext;

}

fs.close();

}

template<typename T>

void List<T>::readFile(string path) //чтение из файла

{

T buf;

fstream fs;

fs.open(path, fstream::in | fstream::app);

int length;

fs >> length; //чтение длины (сколько данных в файле)

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

{

fs >> buf;

push\_back(buf); //полученные данные добавляем в конец списка

}

fs.close();

}

int main()

{

setlocale(LC\_ALL, "ru");

system("color F0");

SetConsoleCP(1251);

int a = 2;

MyClass mc;

List<MyClass> lst;

bool f = true;

string path;

while (f)

{

cout << "---------------------МЕНЮ---------------------" << endl << endl;

cout << "1. Добавить элемент в конец списка." << endl

<< "2. Добавить элемент в начало списка" << endl

<< "3. Добавить элемент в середину списка." << endl

<< "4. Добавить k элементов в конец списка." << endl

<< "5. Добавить k элементов в начало списка." << endl

<< "6. Добавить k элементов в середину списка." << endl

<< "7. Удалить элемент из конца списка." << endl

<< "8. Удалить элемент из начала списка." << endl

<< "9. Удалить элемент из середины списка." << endl

<< "10. Удалить k элементов из конца списка." << endl

<< "11. Удалить k элементов из начала списка." << endl

<< "12. Удалить k элементов из середины списка." << endl

<< "13. Вывести список в консоль." << endl

<< "14. Удалить все данные из списка." << endl

<< "15. Записать в файл." << endl

<< "16. Считать из файла." << endl

<< "0. Выход." << endl << endl;

cin >> a;

switch (a)

{

case 1:

{

cout << "Введите данные: " << endl;

mc.setData();

lst.push\_back(mc);

cout << "DONE" << endl;

break;

}

case 2:

{

cout << "Введите данные: " << endl;

mc.setData();

lst.push\_front(mc);

cout << "DONE" << endl;

break;

}

case 3:

{

int b;

cout << "Введите данные: " << endl;

mc.setData();

do

{

cout << "На какую позицию записать данные? (Введите индекс) "; cin >> b;

} while (b > lst.GetSize());

lst.insert(mc, b);

cout << "DONE" << endl;

break;

}

case 4:

{

int b;

cout << "Введите количество элементов для добавления: "; cin >> b;

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

{

cout << "Введите данные: " << endl;

mc.setData();

lst.push\_back(mc);

cout << "DONE" << endl;

}

break;

}

case 5:

{

int b;

cout << "Введите количество элементов для добавления: "; cin >> b;

cout << "Введите данные: " << endl;

mc.setData();

lst.push\_front(mc);

cout << "DONE" << endl;

for (int i = 1; i < b; i++)

{

cout << "Введите данные: " << endl;

mc.setData();

lst.insert(mc, i);

cout << "DONE" << endl;

}

break;

}

case 6:

{

int b, c;

cout << "Введите количество элементов для добавления: "; cin >> b;

do

{

cout << "На какую позицию записать данные? (Введите индекс) "; cin >> c;

} while (c > lst.GetSize());

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

{

cout << "Введите данные: " << endl;

mc.setData();

lst.insert(mc, c + i);

cout << "DONE" << endl;

}

break;

}

case 7:

{

lst.pop\_back();

cout << "DONE" << endl;

break;

}

case 8:

{

lst.pop\_front();

cout << "DONE" << endl;

break;

}

case 9:

{

int b;

cout << "Введите индекс элемента для удаления: "; cin >> b;

lst.removeAt(b);

cout << "DONE" << endl;

break;

}

case 10:

{

int b;

cout << "Введите количество элементов для удаления: "; cin >> b;

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

{

lst.pop\_back();

}

cout << "DONE" << endl;

break;

}

case 11:

{

int b;

cout << "Введите количество элементов для удаления: "; cin >> b;

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

{

lst.pop\_front();

}

cout << "DONE" << endl;

break;

}

case 12:

{

int b, c;

cout << "Введите начальный индекс: "; cin >> b;

cout << "Введите конечный индекс: "; cin >> c;

lst.removeFewAt(b, c);

cout << "DONE" << endl;

break;

}

case 13:

{

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

{

cout << lst[i] << " ";

}

cout << endl << endl;

system("pause");

break;

}

case 14:

{

lst.clear();

cout << "DONE" << endl;

break;

}

case 15:

{

if (path == "")

{

cout << "Введите название файла для записи: ";

cin >> path;

path = path + ".txt";

}

lst.writeFile(path);

break;

}

case 16:

{

if (path == "")

{

cout << "Введите название файла для чтения: ";

cin >> path;

path = path + ".txt";

}

lst.readFile(path);

break;

}

case 0: {f = false; break;}

default:

{

cout << "Неверный параметр, попробуйте снова." << endl;

break;

}

}

cout << "\033[2J\033[1;1H";

}

SetConsoleCP(866);

return 0;

}

template <typename T>

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

{

size = 0;

head = nullptr;

}

template <typename T>

List<T>::~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--;

}

}

template<typename T>

void List<T>::push\_back(T data)

{

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

{

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

}

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++; //количество элементов увеличиваем на 1

}

template<typename T>

T& List<T>::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++;

}

}

template<typename T>

void List<T>::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

}

template<typename T>

void List<T>::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();

}

}

template<typename T>

void List<T>::push\_front(T 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

}

template<typename T>

void List<T>::insert(T value, int index) //вставка элемента по индексу

{

if (index > size) return; //если индекс больше size, то вставить невозможно

if (index == 0) //если индекс 0 то вызываем вставку в начало списка

{

push\_front(value);

}

else if (index == size) //если индекс равен длине, вызываем вставку в конец списка

{

push\_back(value);

}

else

{

Node\* previous = this-> head; //создаем указатель на головной элемент для поиска нужного элемента

for (int i = 0; i < index - 1; i++) //поиск предыдущего элемента, перед индексом, чтобы поменять в нем указатель

{

previous = previous->pNext;

}

previous->pNext = new Node(value, previous->pNext); //создание нового элемента с переданным значением и установка указателя на элемент с переданным индексом

size++; //Предыдущая строка "вклинивает" новый элемент между старыми

}

}

template<typename T>

void List<T>::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--;

}

}

template<typename T>

void List<T>::pop\_back()

{

removeAt(size - 1);

}

template<typename T>

void List<T>::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);

}

}

Двунаправленный список:

#include <iostream>

#include <fstream>

#include <Windows.h>

using namespace std;

class MyClass

{

public:

void setData() { cin >> str; }

string getData() { return str; }

private:

string str;

friend ostream& operator << (std::ostream& os, MyClass& cl);

friend istream& operator >> (std::istream& in, MyClass& cl);

};

ostream& operator << (std::ostream& os, MyClass& cl) //перегруженный оператор вывода для моего класса

{

return os << cl.str; //запись данных в поток

}

istream& operator >> (std::istream& in, MyClass& cl) //перегруженный оператор ввода для моего класса

{

in >> cl.str; //что и куда читать из потока

return in;

}

template <typename T>

class List

{

private:

template<typename T> //минимальная ячейка памяти

class Node //вложенный класс - ячейка памяти

{

public:

Node(T data = T(), Node\* pNext = nullptr, Node\* pPrev = 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;

this->pPrev = pPrev;

}

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

Node\* pPrev; //хранит указатель на предыдущий элемент

T data; //хранит наши данные

};

Node<T>\* head; //указатель на первый элемент в списке

Node<T>\* tail; //указатель на последний элемент в списке

int size; //количество элементов в списке.

public:

List(); //Конструктор

~List(); //Деструктор

void push\_back(T 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

T& 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(T data); //добавляет элемент в начало списка

void insert(T value, int index); //добавление элемента по индексу

void removeAt(int index); //удаление элемента по указаному индексу

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

void writeFile(string path); //запись в файл

void readFile(string path); //чтение из файла

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

//Node<T>\*& searchPriveousIndex(const int index); //поиск предыдущего элемента для реализации методов insert и removeAt

//void insertFewElem(int index)

};

int main()

{

setlocale(LC\_ALL, "ru");

system("color F0");

SetConsoleCP(1251);

int a = 2;

MyClass mc;

List<MyClass> lst;

bool f = true;

string path;

while (f)

{

cout << "---------------------МЕНЮ---------------------" << endl << endl;

cout << "1. Добавить элемент в конец списка." << endl

<< "2. Добавить элемент в начало списка" << endl

<< "3. Добавить элемент в середину списка." << endl

<< "4. Добавить k элементов в конец списка." << endl

<< "5. Добавить k элементов в начало списка." << endl

<< "6. Добавить k элементов в середину списка." << endl

<< "7. Удалить элемент из конца списка." << endl

<< "8. Удалить элемент из начала списка." << endl

<< "9. Удалить элемент из середины списка." << endl

<< "10. Удалить k элементов из конца списка." << endl

<< "11. Удалить k элементов из начала списка." << endl

<< "12. Удалить k элементов из середины списка." << endl

<< "13. Вывести список в консоль." << endl

<< "14. Удалить все данные из списка." << endl

<< "15. Записать в файл." << endl

<< "16. Считать из файла." << endl

<< "0. Выход." << endl << endl;

cin >> a;

switch (a)

{

case 1:

{

cout << "Введите данные: " << endl;

mc.setData();

lst.push\_back(mc);

cout << "DONE" << endl;

break;

}

case 2:

{

cout << "Введите данные: " << endl;

mc.setData();

lst.push\_front(mc);

cout << "DONE" << endl;

break;

}

case 3:

{

int b;

cout << "Введите данные: " << endl;

mc.setData();

do

{

cout << "На какую позицию записать данные? (Введите индекс) "; cin >> b;

} while (b > lst.GetSize());

lst.insert(mc, b);

cout << "DONE" << endl;

break;

}

case 4:

{

int b;

cout << "Введите количество элементов для добавления: "; cin >> b;

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

{

cout << "Введите данные: " << endl;

mc.setData();

lst.push\_back(mc);

cout << "DONE" << endl;

}

break;

}

case 5:

{

int b;

cout << "Введите количество элементов для добавления: "; cin >> b;

cout << "Введите данные: " << endl;

mc.setData();

lst.push\_front(mc);

cout << "DONE" << endl;

for (int i = 1; i < b; i++)

{

cout << "Введите данные: " << endl;

mc.setData();

lst.insert(mc, i);

cout << "DONE" << endl;

}

break;

}

case 6:

{

int b, c;

cout << "Введите количество элементов для добавления: "; cin >> b;

do

{

cout << "На какую позицию записать данные? (Введите индекс) "; cin >> c;

} while (c > lst.GetSize());

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

{

cout << "Введите данные: " << endl;

mc.setData();

lst.insert(mc, c + i);

cout << "DONE" << endl;

}

break;

}

case 7:

{

lst.pop\_back();

cout << "DONE" << endl;

break;

}

case 8:

{

lst.pop\_front();

cout << "DONE" << endl;

break;

}

case 9:

{

int b;

cout << "Введите индекс элемента для удаления: "; cin >> b;

lst.removeAt(b);

cout << "DONE" << endl;

break;

}

case 10:

{

int b;

cout << "Введите количество элементов для удаления: "; cin >> b;

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

{

lst.pop\_back();

}

cout << "DONE" << endl;

break;

}

case 11:

{

int b;

cout << "Введите количество элементов для удаления: "; cin >> b;

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

{

lst.pop\_front();

}

cout << "DONE" << endl;

break;

}

case 12:

{

int b, c;

cout << "Введите начальный индекс: "; cin >> b;

cout << "Введите конечный индекс: "; cin >> c;

lst.removeFewAt(b, c);

cout << "DONE" << endl;

break;

}

case 13:

{

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

{

cout << lst[i] << " ";

}

cout << endl << endl;

system("pause");

break;

}

case 14:

{

lst.clear();

cout << "DONE" << endl;

break;

}

case 15:

{

if (path == "")

{

cout << "Введите название файла для записи: ";

cin >> path;

path = path + ".txt";

}

lst.writeFile(path);

break;

}

case 16:

{

if (path == "")

{

cout << "Введите название файла для чтения: ";

cin >> path;

path = path + ".txt";

}

lst.readFile(path);

break;

}

case 0: {f = false; break;}

default:

{

cout << "Неверный параметр, попробуйте снова." << endl;

break;

}

}

cout << "\033[2J\033[1;1H";

}

SetConsoleCP(866);

return 0;

}

/\*

Двусвязный список / двунаправленный список

Имеет head и tail.

\* хранит в каждой ячейке указатель на следующий и предыдущий элемент.

\* более быстрый по сравнению с односвязным списком, при выполнении действий ближе к концу списку.

\*/

template<typename T>

void List<T>::writeFile(string path) //запись в файл

{

fstream fs;

fs.open(path, fstream::out);

Node<T>\* temp = head;

fs << GetSize() << "\n"; //запись в файл размера списка

for (int i = 0; i < GetSize(); i++) //запись данных в файл

{

fs << temp->data << " ";

temp = temp->pNext;

}

fs.close();

}

template<typename T>

void List<T>::readFile(string path) //чтение из файла

{

T buf;

fstream fs;

fs.open(path, fstream::in | fstream::app);

int length;

fs >> length; //чтение длины (сколько данных в файле)

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

{

fs >> buf;

push\_back(buf); //полученные данные добавляем в конец списка

}

fs.close();

}

template<typename T>

void List<T>::pop\_back() //удаление с конца файла

{

removeAt(size - 1); //вызов функции удаления по индексу

}

template<typename T>

void List<T>::removeAt(int index) //удаление по индксу

{

if (index == 0) pop\_front(); //если 0 индекс, то удаляем с конца

else if (index == size - 1) //если последний элемент

{

Node<T>\* previous = tail; //устанавливаем указатель на хвост

for (int i = size - 1; i > index - 1; i--) //идем на предыдущий элемент

{ //(относительно удаляемого элемента

previous = previous->pPrev; //итерируемся по списку

}

Node<T>\* toDelete = previous->pNext; //создаем указатель на ячейку для удаления

previous->pNext = nullptr; //меняем указатели на следующий элемент у предшествющего

delete toDelete; //удаляем элемент

tail = previous; //меняем уазатель на хвост

size--;

}

else

{

if (size - index < size / 2) //условие с какой стороны меньше итерироваться до предыдущего элемента

{

Node<T>\* previous = tail;

for (int i = size - 1; i > index - 1; i--)

{

previous = previous->pPrev;

}

Node<T>\* toDelete = previous->pNext;

previous->pNext = toDelete->pNext; //замена указателя на следующий элемент

toDelete->pNext->pPrev = previous; //замена указателя на предыдущий элемент

delete toDelete;

size--;

}

else

{

Node<T>\* previous = head;

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

{

previous = previous->pNext;

}

Node<T>\* toDelete = previous->pNext;

previous->pNext = toDelete->pNext;

toDelete->pNext->pPrev = previous;

delete toDelete;

size--;

}

}

}

template<typename T>

void List<T>::insert(T data, int index) //добавление элемента по индексу

{

if (index > size) return; //проверки на исключения

if (index == 0) push\_front(data);

else if (index == size) push\_back(data);

else

{

if (size - index < size / 2) //условие с какой стороны меньше итерироваться до предыдущего элемента

{

Node<T>\* previous = this->tail;

for (int i = size - 1; i > index - 1; i--)

{

previous = previous->pPrev; //нахождение предшествующего элемента

}

previous->pNext = new Node<T>(data, previous->pNext, previous); //добавление элемента

previous->pNext->pPrev = previous; //смена указателей

}

else

{

Node<T>\* previous = this->head;

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

{

previous = previous->pNext; //нахождение предшествующего элемента

}

previous->pNext = new Node<T>(data, previous->pNext, previous); //добавление элемента

previous->pNext->pPrev = previous;//смена указателей

}

size++;

}

}

template<typename T>

void List<T>::push\_front(T data) //добавление в начало

{

if (head == nullptr) //исключение

{

head = new Node<T>(data); //создание объекта

tail = head; //установка хвоста

}

else

{

Node<T>\* temp = new Node<T>(data, head); //создание объекта с параметрами, чтобы сразу установить предществущий

head->pPrev = temp; //смена указателей

head = temp;

}

size++;

}

template<typename T>

List<T>::List() //конструктор

{

head = tail = nullptr;

size = 0;

}

template<typename T>

List<T>::~List() //деструктор

{

clear();

}

//вот тут я остановился делать блоксхему

template<typename T>

void List<T>::clear() //очищение всего списка

{

while (size) //пока длина != 0

{

pop\_front();

}

}

template<typename T>

void List<T>::pop\_front() //удаление первого элемента

{

Node<T>\* toDelete = head; //установка указателя на голову

if (size == 1) //проверка исключения

{

head = head->pNext; //смена указателей

tail = tail->pPrev;

delete toDelete;

}

else

{

head = head->pNext; //смена указателей

head->pPrev = nullptr;

delete toDelete;

}

size--;

}

template<typename T>

T& List<T>::operator[](const int index)

{

int counter\_prev = size - 1; //счетчик для того, чтобы идти с конца

int counter\_next = 0; //счетчик для того, чтобы идти с начала

Node<T>\* elem; //указатель для итерации по списку

if (size - index <= size / 2) //условие с какой стороны лучще идти по списку

{

elem = this->tail; //установка отсчета

while (elem != nullptr) //цикл с условием окончания списка

{

if (counter\_prev == index)

{

return elem->data; //возврат данных, когда находимся в нужно ячейке

}

elem = elem->pPrev; //переход на следующую ячейку

counter\_prev--;

}

}

else

{

elem = this->head;

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

{

if (counter\_next == index)

{

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

}

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

counter\_next++;

}

}

}

template<typename T>

void List<T>::push\_back(T data) //добавление в конец

{

if (head == nullptr) //исключения

{

head = new Node<T>(data); //создание объекта

tail = head;

}

else

{

Node<T> \*current = new Node<T>(data); //создание объекта

tail->pNext = current; //смена указателей

current->pPrev = tail;

tail = current;

}

size++;

}

template<typename T>

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

{

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

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

{

removeAt(index\_start);

}

}

Стек:

#include <iostream>

#include <Windows.h>

#include <fstream>

using namespace std;

class MyClass

{

private:

string str;

friend ostream& operator << (std::ostream& os, const MyClass& cl);

friend istream& operator >> (std::istream& in, MyClass& cl);

public:

MyClass()

{

this->str = "";

}

MyClass(string str)

{

this->str = str;

}

MyClass& operator =(const MyClass& other);

void setData()

{

cin >> str;

}

string getData() { return str; }

};

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

{

this->str = other.str;

return \*this;

}

ostream& operator << (std::ostream& os, const MyClass& cl) //перегруженный оператор вывода для моего класса

{

return os << cl.str; //запись данных в поток

}

istream& operator >> (std::istream& in, MyClass& cl) //перегруженный оператор ввода для моего класса

{

in >> cl.str; //что и куда читать из потока

return in;

}

template <typename T>

class Stek

{

private:

template<typename T> //минимальная ячейка памяти

class Node //вложенный класс - ячейка памяти

{

public:

Node(T data = T(), 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

T data; //хранит наши данные

};

Node<T>\* head; //указатель на заглавный элемент в стеке

int size; //количество элементов в стеке.

public:

Stek(); //Конструктор

~Stek(); //Деструктор

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

T del(); //удаляет элемент из стека

T getData() { return head->data; }

void clear(); //удаляет все элементы их стека

void add(T data); //добавляет элемент в стек

void writeFile(string path); //запись в файл

void readFile(string path); //чтение из файла

};

int main()

{

setlocale(LC\_ALL, "ru");

system("color F0");

SetConsoleCP(1251);

int a = 2;

MyClass mc;

Stek<MyClass> stk;

bool f = true;

string path;

while (f)

{

cout << "---------------------МЕНЮ---------------------" << endl << endl;

cout << "1. Добавить элемент." << endl

<< "2. Добавить k элементов." << endl

<< "3. Удалить элемент." << endl

<< "4. Удалить k элементов." << endl

<< "5. Удалить все данные." << endl

<< "6. Вывести в консоль." << endl

<< "7. Записать в файл." << endl

<< "8. Считать из файла." << endl

<< "0. Выход." << endl << endl;

cin >> a;

switch (a)

{

case 1:

{

cout << "Введите данные: " << endl;

mc.setData();

stk.add(mc);

cout << "DONE" << endl;

break;

}

case 2:

{

int b;

cout << "Введите количество элементов для добавления: "; cin >> b;

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

{

cout << "Введите данные: " << endl;

mc.setData();

stk.add(mc);

cout << "DONE" << endl;

}

break;

}

case 3:

{

cout << "Удаленный элемент: " << stk.del() << endl;

cout << "DONE" << endl;

break;

}

case 4:

{

int b, c;

do

{

cout << "Введите количество элементов для удаления: "; cin >> c;

} while (c > stk.GetSize());

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

{

cout << "Удаленный элемент" << stk.del();

cout << "DONE" << endl;

}

break;

}

case 5:

{

stk.clear();

break;

}

case 6:

{

cout << stk.getData() << endl;

system("pause");

break;

}

case 7:

{

if (path == "")

{

cout << "Введите название файла для записи: ";

cin >> path;

path = path + ".txt";

}

stk.writeFile(path);

cout << "DONE" << endl;

break;

}

case 8:

{

if (path == "")

{

cout << "Введите название файла для записи: ";

cin >> path;

path = path + ".txt";

}

stk.readFile(path);

cout << "DONE" << endl;

break;

}

case 0: {f = false; break;}

default:

{

cout << "Неверный параметр, попробуйте снова." << endl;

break;

}

}

cout << "\033[2J\033[1;1H";

}

SetConsoleCP(866);

return 0;

}

/\*

СТЕК

Динамическая структура данных, работающая по принципу последний вошел, первый вышел. (LIFO).

Представить можно в виде стопки тарелок.

По сути стек - урезанная версия односвзного списка (подходит под метод - LIFO), нам необходимо оставить только операции, которые подходят под это.

\* Удаление первого элемента, при удалении (извлечении из стека, нам необходимо получать данные, которые там хранились)

\* Добавление первого элемента

\* Метод просмотра элемента из вершины стека

\*

\*/

template <typename T>

Stek<T>::Stek() //мы создаем пустой стек, а следствено, в нем лежит 0 элементов, а следственно указывать некуда.

{

size = 0;

head = nullptr;

}

template <typename T>

Stek<T>::~Stek() //деструктор освобождает динамическую память по приципу метода clear,

{ //мы можем делегировать ему эти обязаности

clear();

}

template<typename T>

void Stek<T>::add(T data)

{

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

size++; //увеличиваем длину на 1

}

template<typename T>

T Stek<T>::del() //удаление из начала

{

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

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

T ret = temp->data;

delete temp; //удаляем первый объект

size--; //уменьшаем длину на 1

return ret;

}

template<typename T>

void Stek<T>::clear()

{

while (size)

{ //вызываем pop\_front столько раз, пока size != 0

del(); //логика основывается на pop\_front поэтому мы делегируем ей эти обязанности

}

}

template<typename T>

void Stek<T>::writeFile(string path) //запись в файл

{

fstream fs;

fs.open(path, fstream::out);

Node<T>\* temp = head;

fs << GetSize() << "\n"; //запись в файл размера списка

for (int i = 0; i < GetSize(); i++) //запись данных в файл

{

fs << temp->data << " ";

temp = temp->pNext;

fs.ate;

}

fs.close();

}

template<typename T>

void Stek<T>::readFile(string path) //чтение из файла

{

T buf;

fstream fs;

fs.open(path, fstream::in | fstream::app);

int length;

fs >> length; //чтение длины (сколько данных в файле)

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

{

fs >> buf;

add(buf); //полученные данные добавляем в конец списка

}

fs.close();

}

Очередь:

#include <iostream>

#include <fstream>

#include <Windows.h>

using namespace std;

class MyClass

{

private:

string str;

friend ostream& operator << (std::ostream& os, MyClass& cl);

friend istream& operator >> (std::istream& in, MyClass& cl);

public:

MyClass()

{

this->str = "";

}

MyClass(string str)

{

this->str = str;

}

MyClass& operator =(const MyClass& other);

void setData() { cin >> str; }

string getData() { return str; }

};

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

{

this->str = other.str;

return \*this;

}

ostream& operator << (std::ostream& os, MyClass& cl) //перегруженный оператор вывода для моего класса

{

return os << cl.str; //запись данных в поток

}

istream& operator >> (std::istream& in, MyClass& cl) //перегруженный оператор ввода для моего класса

{

in >> cl.str; //что и куда читать из потока

return in;

}

template <typename T>

class Queue

{

private:

template<typename T> //минимальная ячейка памяти

class Node //вложенный класс - ячейка памяти

{

public:

Node(T data = T(), Node\* pNext = nullptr, Node\* pPrev = 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;

this->pPrev = pPrev;

}

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

Node\* pPrev; //хранит указатель на предыдущий элемент

T data; //хранит наши данные

};

Node<T>\* head; //указатель на первый элемент в очереди

Node<T>\* tail; //указатель на последний элемент в очереди

int size; //количество элементов в очереди.

public:

Queue(); //Конструктор

~Queue(); //Деструктор

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

void clear(); //удаляет все элементы очереди

void add(T data); //добавляет элемент в очередь

void del(); //удаление элемента из очереди

void writeFile(string path); //запись в файл

void readFile(string path); //чтение из файла

T& getDataEnd() { return tail->data; } //получение данных из конца очереди

T& getDataStart() { return head->data; } //получение данных из начала очереди

};

int main()

{

setlocale(LC\_ALL, "ru");

setlocale(LC\_ALL, "ru");

system("color F0");

SetConsoleCP(1251);

int a = 2;

MyClass mc;

Queue<MyClass> que;

bool f = true;

string path;

while (f)

{

cout << "---------------------МЕНЮ---------------------" << endl << endl;

cout << "1. Добавить элемент." << endl

<< "2. Добавить k элементов." << endl

<< "3. Удалить элемент." << endl

<< "4. Удалить k элементов." << endl

<< "5. Удалить все данные." << endl

<< "6. Вывести первый элемент." << endl

<< "7. Вывести последний элемент." << endl

<< "8. Записать в файл." << endl

<< "9. Считать из файла." << endl

<< "0. Выход." << endl << endl;

cin >> a;

switch (a)

{

case 1:

{

cout << "Введите данные: " << endl;

mc.setData();

que.add(mc);

cout << "DONE" << endl;

break;

}

case 2:

{

int b;

cout << "Введите количество элементов для добавления: "; cin >> b;

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

{

cout << "Введите данные: " << endl;

mc.setData();

que.add(mc);

cout << "DONE" << endl;

}

break;

}

case 3:

{

que.del();

cout << "DONE" << endl;

break;

}

case 4:

{

int b, c;

do

{

cout << "Введите количество элементов для удаления: "; cin >> c;

} while (c > que.GetSize());

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

{

que.del();

cout << "DONE" << endl;

}

break;

}

case 5:

{

que.clear();

break;

}

case 6:

{

cout << que.getDataStart() << endl;

system("pause");

break;

}

case 7:

{

cout << que.getDataEnd() << endl;

system("pause");

break;

}

case 8:

{

if (path == "")

{

cout << "Введите название файла для записи: ";

cin >> path;

path = path + ".txt";

}

que.writeFile(path);

cout << "DONE" << endl;

break;

}

case 9:

{

if (path == "")

{

cout << "Введите название файла для записи: ";

cin >> path;

path = path + ".txt";

}

que.readFile(path);

cout << "DONE" << endl;

break;

}

case 0: {f = false; break;}

default:

{

cout << "Неверный параметр, попробуйте снова." << endl;

break;

}

}

cout << "\033[2J\033[1;1H";

}

SetConsoleCP(866);

return 0;

return 0;

}

/\*

Очередь как динамическая структура данных

При взаимодействии с очередья мы можем взаимодействовать толко с 1 и последним элементами, остальные нам не доступны.

Не можем:

\* Перебрать все элементы очереди

\* Не можем вставить или убрать элемент из середины очереди

\* Не можем добавить элемент в начало очереди, только в конец можно

\* Нельзя удалить элемент из конца очереди

Можем:

\* Посмотреть что находится в начале очереди и в конце очереди

\* Добавить элемент в начало очереди.

\* Можно удалить/извлечь элемент из начала очереди

\*/

template<typename T>

Queue<T>::Queue() //конструктор, который создает пустую очередь

{

head = tail = nullptr;

size = 0;

}

template<typename T>

Queue<T>::~Queue() //деструктор, при выходе из области видимости уничтожает все объекты списка

{

clear();

}

template<typename T>

void Queue<T>::add(T data) //добавление элемента

{

if (head == nullptr) //исключения

{

head = new Node<T>(data); //создание объекта

tail = head;

}

else

{

Node<T>\* current = new Node<T>(data, nullptr, tail); //создание объекта с указанием предыдущего элемента

tail->pNext = current; //смена указателей

tail = current; //смена хвоста

}

size++;

}

template<typename T>

void Queue<T>::del()

{

Node<T>\* toDelete = head; //установка указателя на голову

if (size == 1) //проверка исключения

{

head = head->pNext; //смена указателей

tail = tail->pPrev;

delete toDelete;

}

else

{

head = head->pNext; //смена указателей

head->pPrev = nullptr;

delete toDelete;

}

size--;

}

template<typename T>

void Queue<T>::clear() //полное очищение очереди

{

while (size) //пока длина !=0

{

del(); //удаляем элемент

}

}

template<typename T>

void Queue<T>::writeFile(string path) //запись в файл

{

fstream fs;

fs.open(path, fstream::out);

Node<T>\* temp = head;

fs << GetSize() << "\n"; //запись в файл размера списка

for (int i = 0; i < GetSize(); i++) //запись данных в файл

{

fs << temp->data << " ";

temp = temp->pNext;

}

fs.close();

}

template<typename T>

void Queue<T>::readFile(string path) //чтение из файла

{

T buf;

fstream fs;

fs.open(path, fstream::in | fstream::app);

int length;

fs >> length; //чтение длины (сколько данных в файле)

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

{

fs >> buf;

add(buf); //полученные данные добавляем в конец списка

}

fs.close();

}