Министерство науки и высшего образования Российской Федерации

Федеральное государственное бюджетное образовательное учреждение высшего образования

«Российский химико-технологический университет имени Д.И. Менделеева»

Кафедра информационных компьютерных технологий

ОТЧЕТ ПО ЛАБОРАТОРНОЙ РАБОТЕ № 2

Выполнил студент группы КС-30 (Ноль Эльвира Гарриевна)

Ссылка на репозиторий: (https://github.com/Elviranng/Nol-Elvira-KS-30/tree/master/lab2.sem2)

Приняли: Пысин Максим Дмитриевич

Краснов Дмитрий Олегович

Дата сдачи: (05.06.21)

Оглавление

[Описание задачи. 2](#_Toc68080086)

[Описание модели. 3](#_Toc68080087)

[Выполнение задачи. 4](#_Toc68080088)

[Заключение. 12](#_Toc68080089)

# Описание задачи.

В рамках лабораторной работы необходимо изучить и реализовать очередь, в соответствии со своим вариантом, при этом:

* Использовать шаблонный подход, обеспечивая работу контейнера с произвольными данными.
* Реализовывать свой итератор с реализацией операторов ++ и !=
* Обеспечивать работу стандартных c++ библиотек и конструкции for(auto el: struct), для чего реализовывать функции begin, end, rbegin, rend.
* Проверку на пустоту и подсчет количества элементов.
* Операцию сортировки с использованием стандартной библиотеки.

Список должен реализовывать операции:

* добавления элемента после произвольного элемента
* удаление произвольного элемента из списка

Для демонстрации работы структуры необходимо создать набор тестов (под тестом понимается функция, которая создаёт структуру, проводит операцию или операции над структурой и удаляет структуру):

* заполнение контейнера 1000 целыми числами в диапазоне от -1000 до 1000 и подсчет их суммы, среднего, минимального и максимального.
* Провести проверку работы операций вставки и изъятия элементов на коллекции из 10 строковых элементов.
* заполнение контейнера 100 структур, содержащих фамилию, имя, отчество и дату рождения (от 01.01.1980 до 01.01.2020) значения каждого поля генерируются случайно из набора заранее заданных. После заполнение необходимо найти всех людей младше 20 лет и старше 30 и создать новые структуры, содержащие результат фильтрации, проверить выполнение на правильность, подсчётом кол-ва элементов, не подходящих под условие в новых структурах.
* Заполнить структуру 1000 элементов и отсортировать ее, проверить правильность использую структуру из стандартной библиотеки и сравнив результат.
* (Список) Перемешать все элементы отсортированного списка в случайном порядке.

# Описание модели.

Список это структура данных, в которой каждый элемент связан с одним из своих соседних элементов, для однонаправленного списка это следующий элемент:

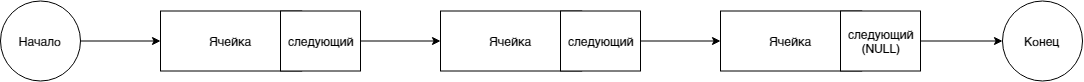


Рисунок 1 "Однонаправленный список"

Или для двунаправленного списка:



Рисунок 2 "Двунаправленный список"

Структура одного элемента списка:

Двунаправленный – поле указателя на предыдущий, поле значения, поле указателя на последующий

Однонаправленный – поле значения элемента, поле указателя на последующий

Имея такую структуру список позволяет легко проводить операции вставки и удаления, так же в зависимости от реализации списки могут поддерживать обращения по индексу, хотя в базовой модели такого не предполагается

# Выполнение задачи.

Полный код программы можно посмотреть в репозитории. Здесь же, будут подробно рассмотрены основные части программы.

Сперва, необходимо было создать шаблонный контейнер, чтобы поддерживать работу с разными типами данных. Как раз с этого момента и начинается реализация списка.

Стоит отметить, что в данной работе был выполнен список по принципу двунаправленного списка, т.е. каждый элемент списка – это указатель на предыдущий элемент, значение и указатель на следующий элемент. Сделано это в связи с необходимостью иметь обратный итератор.

Реализация узла:

// Узел списка (состоит из значения, указателя на след. и пред. элемент списка)

    class Node {

    public:

        T\* value = nullptr;  // Указатель на значение узла (т.к. шаблонный контейнер)

        Node\* prev = nullptr;  // Указатель на пред. элемент

        Node\* next = nullptr;  // Указатель на след. элемент

        // Конструктор по-умолчанию (для пустых элементов)

        Node() {}

        // Конструктор с параметрами

        Node(T value) {

            this->value = new T(value);

        }

        ~Node() {

            if (this->value != nullptr) {

                delete value;

            }

        }

    };

Сам узел состоит из 3 членов класса, и является в свою очередь вложенным элементом класса списка. У каждого узла имеются два указателя на предыдущий и следующий элементы, а также указатель на значение этого узла.

Выбор указателя на значение вместо обычного значения был обусловлен тем, что мы имеем дело с шаблонным классом, т.е. неизвестно какой тип данных окажется в нем. В процессе разработки программы стало понятно, что хранение в узле непосредственного значения в некоторых случаях могло вызывать неопределенное поведение.

Члены класса очереди:

Node\* first = nullptr;  // Указатель на начало списка

    Node\* last = nullptr;  // Указатель на конец списка

    int elems = 0; // Кол-во элементов

Сам список имеет 3 члена: указатель на первый узел, на последний узел и счетчик кол-ва значений.

Для работы со структурами for auto и sort() необходимо иметь итератор, причем sort требует итератор с тегом random\_access\_iterator.

// Итератор списка

    class Iterator {

    public:

        using difference\_type = std::ptrdiff\_t;

        using value\_type = T;

        using pointer = T \* ;

        using reference = T & ;

        using iterator\_category = std::random\_access\_iterator\_tag;

Это стандартное объявление итератора такого типа.

protected:

        Node\* pointer\_node;  // Указатель на узел

        Node\* pointer\_last;

В связи со сложностями, связанными с методами begin, end и т.п., в итераторе имеются 2 члена класса – указатель на текущий узел и указатель на последний узел. Такая структура итератора позволяет избежать неправильной работы функции sort.

Конструктор итератора принимает 2 аргумента – указатель на текущий узел и указатель на последний узел структуры.

Iterator(Node\* pointer\_node, Node\* pointer\_last) :

            pointer\_node(pointer\_node),

            pointer\_last(pointer\_last)

        {}

Далее рассматриваются операторы итератора. Рассмотрены будут не все из них, так как некоторые работают по аналогии и не требуют дополнительных пояснений.

// Перегрузка постфиксного инкремента (с фиктивным параметром)

        Iterator operator++(int) {

            Iterator tmp\_it(this->pointer\_node, this->pointer\_last);

            this->pointer\_node = this->pointer\_node->next;

            return tmp\_it;

        }

Стандартная перегрузка оператора постфиксного инкремента для итератора заключается в том, что она должна поменять текущий узел в итераторе на следующий, но вернуть новый итератор, который указывает всё ещё на текущий узел.

Отдельное внимание стоит уделить оператору постфиксного декремента.

// Перегрузка постфиксного декремента (с фиктивным параметром)

        Iterator operator--(int) {

            Iterator tmp\_it(this->pointer\_node, this->pointer\_last);

            if (this->pointer\_node == nullptr) {

                this->pointer\_node = this->pointer\_last;

            }

            else {

                this->pointer\_node = this->pointer\_node->prev;

            }

            return tmp\_it;

        }

Работает он аналогично инкременту итератора (переходит на предыдущий элемент), однако в данной реализации есть следующий момент: из-за структуры списка, его первый и последний узлы будут иметь в своих указателях на предыдущий и последующий узлы соответственно, нулевые указатели. Однако метод end и rend должны возвращать итератор на элемент после (перед) последнего (первого) элемента структуры. А сам этот итератор должен иметь возможность вернуть предыдущий (следующий) узел. В связи с этим, реализация оператора декремента имеет особенность – если текущим элементом итератора является нулевой указатель, то будет произведена замена нулевого узла на последний (для обычного итератора) или на первый (для обратного итератора). Такое решение позволяет работать одновременно и for auto и sort.

Оператор разыменования просто возвращает ссылку на значение текущего узла (если оно существует).

// Перегрузка оператора разыменования

        T& operator\*() const {

            if (this->pointer\_node != nullptr) {

                return \*(this->pointer\_node->value);

            }

            else {

                throw std::runtime\_error("Exception\_runtime: empty node has no value");

            }

        }

Операторы равенства и неравенства просто проверяют являются ли два узла одним и тем же (указывают ли они на одно значение в памяти)

// Перегрузка оператора равенства

        bool operator==(const Iterator& b) const {

            return this->pointer\_node == b.pointer\_node;

        }

Оператор сравнения работает следующим образом: он сравнивает два узла вычитая один итератор из другого, и сравнивая ответ с нулем. Данный оператор используется в методе быстрой сортировки в функции sort.

// Перегрузка оператора меньше

        bool operator<(const Iterator& b) const {

            return ((b - (\*this)) > 0);

        }

// Оператор вычитания итераторов

        difference\_type operator-(const Iterator& b) const {

            difference\_type i = 0;

            Iterator tmp\_it = \*this;

            while (tmp\_it.pointer\_node != b.pointer\_node) {

                (tmp\_it)--;

                i++;

            }

            return i;

        }

Оператор вычитания итераторов находит разницу в положении между двумя узлами и возвращает некое значение разностного типа, которое указывает сколько элементов может быть помещено между этими узлами.

Так же имеется обратный итератор ReverseIterator – он полностью аналогичен обычному за исключением операторов инкремента и декремента (они работают в обратную сторону):

// Перегрузка постфиксного декремента (с фиктивным параметром)

        ReverseIterator operator--(int) {

            ReverseIterator tmp\_it(this->pointer\_node, this->pointer\_last);

            if (this->pointer\_node == nullptr) {

                this->pointer\_node = this->pointer\_last;

            }

            else {

                this->pointer\_node = this->pointer\_node->next;

            }

            return tmp\_it;

        }

То есть, происходит переход не на предыдущий узел, а на следующий – таковы условия работы итератора для rbegin, rend.

У самого класса списка имеются 2 конструктора и 1 деструктор, который предотвращает утечку памяти:

// Конструктор по-умолчанию

    MyList() {}

    // Конструктор списка инициализации

    MyList(initializer\_list<T> init) {

        for (auto el : init)

            this->push(el);

    }

    // Деструктор

    ~MyList() {

        int my\_size = this->size();

        if (my\_size > 0) {

            for (int i = 0; i < my\_size; i++) {

                this->pop();

            }

        }

    }

Инициализация возможна с использованием списка значений.

Метод добавления в конец списка элементов реализуется созданием нового узла, и переадресацией указателей на него:

// Добавление элемента в конец

    void push(T value) {

        Node\* node = new Node(value);

        if (this->first == nullptr) {

            this->first = node;

        }

        if (this->last == nullptr) {

            this->last = node;

        }

        else {

            node->prev = this->last;

            this->last->next = node;

            this->last = node;

        }

        this->elems++;

    }

Метод добавления элемента после определенного элемента работает аналогично, но вставляет элемент на определенную позицию (после проверки валидности переданных данных). Этот метод принимает два параметра – новое значение и позиция, за которой надо вставить элемент с этим значением.

// Добавление после произвольного элемента

    void insert(T value, int position) {

        if((this->size() <= position) || (position < 0)){

            throw std::runtime\_error("Exception\_runtime: no such element in list");

        }

        Node\* node = new Node(value);

        if(this->size() == 0){

            if (this->first == nullptr) {

                this->first = node;

            }

            if (this->last == nullptr) {

                this->last = node;

            }

        }else if(position < this->size() - 1){

            Node\* tmp\_node = this->first;

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

                tmp\_node = tmp\_node->next;

            }

            node->prev = tmp\_node;

            node->next = tmp\_node->next;

            tmp\_node->next->prev = node;

            tmp\_node->next = node;

        }else{

            node->prev = this->last;

            node->next = nullptr;

            this->last->next = node;

            this->last = node;

        }

        this->elems++;

    }

Метод взятие из начала реализуется обратным образом – значение элемента копируется, он сам удаляется, после чего происходит переадресация указателей с учетом измененной структуры, а само значение возвращается из метода:

// Взятие элемента из начала

    T pop() {

        if (this->first != nullptr) {

            this->elems--;

            T val = \*(this->first->value);

            if (this->first == this->last) {

                delete this->first;

                this->last = nullptr;

                this->first = nullptr;

            }

            else {

                Node\* tmp\_node = this->first->next;

                delete this->first;

                this->first = tmp\_node;

                this->first->prev = nullptr;

            }

            return val;

        }

        else {

            throw std::logic\_error("Exception\_logic: empty node has no value");

        }

    }

Метод взятия определенного элемента работает аналогично, но принимает параметром позицию, элемент на которой надо удалить и вернуть значение которого.

// Взятие произвольного элемента

    T erase(int position) {

        if (position < this->size() && position >= 0) {

            Node\* tmp\_node\_main = this->first;

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

                tmp\_node\_main = tmp\_node\_main->next;

            }

            T val = \*(tmp\_node\_main->value);

            if (this->first == this->last) {

                delete this->first;

                this->last = nullptr;

                this->first = nullptr;

            }

            else {

                if(position == 0){

                    Node\* tmp\_node = this->first->next;

                    delete this->first;

                    this->first = tmp\_node;

                    this->first->prev = nullptr;

                }else if(position == this->size() - 1){

                    Node\* tmp\_node = this->last->prev;

                    delete this->last;

                    this->last = tmp\_node;

                    this->last->next = nullptr;

                }else{

                    tmp\_node\_main->prev->next = tmp\_node\_main->next;

                    tmp\_node\_main->next->prev = tmp\_node\_main->prev;

                    tmp\_node\_main->prev = nullptr;

                    tmp\_node\_main->next = nullptr;

                    delete tmp\_node\_main;

                }

            }

            this->elems--;

            return val;

        }

        else {

            throw std::logic\_error("Exception\_logic: empty node has no value");

        }

    }

В конце класса имеем методы возврата итераторов на конец списка, на начало и их реверсивные реализации:

// Метод возврата итератора на начало

    Iterator begin() {

        return Iterator(this->first, this->last);

    }

    // Метод возврата итератора на конец

    Iterator end() {

        return Iterator(nullptr, this->last);

    }

Метод end возвращает итератор с указателем на nullptr, который дает понять sort, что дальше элементов нет, после чего такой итератор способен вернуть последний элемент структуры благодаря реализации оператора декремента.

// Метод подсчета кол-ва элементов

    int size() {

        return this->elems;

    }

    // Метод проверки на пустоту

    bool empty() {

        return this->first ? false : true;

    }

Методы, проверяющие список на пустоту и на размер.

На этом описание структуры заканчивается и в коде идут её тесты.

# Заключение.

В ходе данной лабораторной работы была реализована и разобрана структура двусвязного списка. Данная структура является очень удобной в обращении и популярной в программировании из-за своего удобства использования.

К данной структуре были написаны два итератора – прямой и обратный, также была создана поддержка sort и for auto для написанной структуры, что является очень полезным и нужным опытом.

Структура прошла все тесты без ошибок в Visual Studio 2019