Московский Авиационный институт
(Национальный исследовательский университет)

Факультет №8
«Компьютерные науки и прикладная математика»

Кафедра 806
«Вычислительная математика и программирование»

Курсовая работа

по теме

«Линейные списки»

Студент:	Ефименко К. И.
Группа:	М8О-109Б-22
Преподаватель:	Сысоев М. А.
Подпись:	
Оценка:	

Москва, 2023

Постановка задачи

Составить и отладить программу на языке C++ для обработки линейного списка заданной организации с отображением списка на динамические структуры. Навигацию по списку следует реализовать с применением итераторов. Предусмотреть выполнение одного нестандартного действия и четырёх стандартных действий:

Нестандартное действие:

удалить элементы списка со значениями, находящимися в заданном диапазоне;

Стандартные действия:

- Печать списка
- Вставка нового элемента в список Удаление элемента из списка
- Подсчёт длины списка

Вид списка:

Линейный однонаправленный с барьерным элементом

Тип элемента списка:

Целый

Основная часть

Необходимая теория

Список - структура данных, состоящая из элементов одного типа, связанных с помощью указателей. Двунаправленный означает, что двигаться можно в обе стороны. Барьерный элемент позволяет получить быстрый доступ к первому и последнему элементам списка.

Описание структуры

Создаётся структура списка и записи в списке. Структура списка содержит указатель на барьерный элемент списка, структура записи содержит само значение элемента списка, а также указатель следующий элемент.

После создания структуры списка создаётся структура итератора. Благодаря ней можно писать одинаковые функции для вариантов на указателях и на векторах. Пишутся функции для печати, удаления, ввода и подсчёта длины списка.

Алгоритм нестандартного действия

Алгоритм достаточно простой. Длина списка нам уже известна, поэтому доходим до середины списка. Ставим один указатель на

первый элемент списка, другой на первый элемент второй половины. И начинаем движение, пока второй указатель не дойдет до конца и обмениваем значения.

Функциональное назначение

Программа предназначена для демонстрации использования метода хранения линейного списка на указателях и работы с ним. Также она демонстрирует использование итераторов.

Код программы

List.hpp

```
#ifndef LIST_HPP_INCLUDED
#define LIST_HPP_INCLUDED

#include <iostream>

template <typename T, typename Allocator = std::allocator<T>>
class List {
private:
    class Node {
    public:
        T value;
        Node* next = nullptr;

        Node(const T& value);
        Node(const T& value, Node* next);
    };

    using Alloc = typename std::allocator_traits<Allocator>::
template rebind_alloc<Node>;
    Alloc alloc;
    using AllocTraits = std::allocator_traits<Alloc>;

    Node* head_ptr = nullptr;
    Node* const block_element_ptr = AllocTraits::allocate(all
oc, 1);
    size_t size = 0;

public:
    class Iterator;

    size_t get_size() const;

    List() = default;
    List(const T& value);
    ~List();

    void push_back(const T& value);
    void push_front(const T& value);
    void pop_back();
    void pop_front();
    void insert(const T& value, Iterator& it);
    void erase(Iterator& iter);
    void erase(Iterator& start, Iterator& end);

    template <typename... Args>
    void emplace_back(const Args&... args);
```

```
    std::ostream& operator<<(std::ostream& stream);

    class Iterator {
    public:
        Node* current_node_ptr = nullptr;

        Iterator(Node& node);

        Iterator& operator=(Node* node_ptr);
        Iterator& operator++();
        Iterator& operator++(int);

        T& operator*();
        const T& operator*() const ;

        bool operator!=(const Iterator& iterator) const;
        bool operator==(const Iterator& iterator) const;
    };

    Iterator begin() const;
    Iterator end() const;
};

#include "../src/List.cpp"

#endif
```

List.cpp

```
#include "../include/List.hpp"

template <typename T, typename Allocator>
List<T, Allocator>::List(const T& value) {
    Node* ptr = AllocTraits::allocate(alloc, 1);
    AllocTraits::construct(alloc, ptr, value);
    head_ptr = ptr;
    head_ptr->next = block_element_ptr;
    ++size;
}

template <typename T, typename Allocator>
List<T, Allocator>::~List() {
    while (head_ptr) {
        if (head_ptr == block_element_ptr) {
            AllocTraits::deallocate(alloc, head_ptr, 1);
            return;
        }

        Node* new_head = head_ptr->next;
        AllocTraits::destroy(alloc, head_ptr);
        AllocTraits::deallocate(alloc, head_ptr, 1);
        head_ptr = new_head;
    }
}

template <typename T, typename Allocator>
size_t List<T, Allocator>::get_size() const {
    return size;
}

template <typename T, typename Allocator>
void List<T, Allocator>::push_back(const T& value) {
    if (size == 0) {
        push_front(value);
```

```
        return;
    }

    Node* temp_ptr = head_ptr;

    while (temp_ptr->next != block_element_ptr) {
        temp_ptr = temp_ptr->next;
    }

    Node* new_node_ptr = AllocTraits::allocate(alloc, 1);
    AllocTraits::construct(alloc, new_node_ptr, value);

    temp_ptr->next = new_node_ptr;
    new_node_ptr->next = block_element_ptr;
    ++size;
}

template <typename T, typename Allocator>
void List<T, Allocator>::push_front(const T& value) {
    Node* swap_ptr = nullptr;
    Node* ptr = AllocTraits::allocate(alloc, 1);
    AllocTraits::construct(alloc, ptr, value);

    if (head_ptr == nullptr) {
        head_ptr = ptr;
        ptr->next = block_element_ptr;
    } else {
        swap_ptr = head_ptr;
        this->head_ptr = ptr;
        this->head_ptr->next = swap_ptr;
    }
    ++size;
}

template <typename T, typename Allocator>
void List<T, Allocator>::pop_back() {
    if (size == 0) {
        return;
    }

    if (size == 1) {
        pop_front();
        return;
    }

    Iterator it = begin();

    while (it.current_node_ptr->next-
>next != block_element_ptr) {
        ++it;
    }


    AllocTraits::destroy(alloc, it.current_node_ptr->next);
    AllocTraits::deallocate(alloc, it.current_node_ptr-
>next, 1);

    it.current_node_ptr->next = block_element_ptr;
    --size;
}

template <typename T, typename Allocator>
void List<T, Allocator>::pop_front() {
    if (head_ptr == nullptr) {
        return;
    }
```

```
    Node* temp_ptr = nullptr;
    temp_ptr = head_ptr;
    head_ptr = head_ptr->next;
    --size;

    if (head_ptr == block_element_ptr) {
        head_ptr = nullptr;
    }

    AllocTraits::destroy(alloc, temp_ptr);
    AllocTraits::deallocate(alloc, temp_ptr, 1);
}

template <typename T, typename Allocator>
template <typename... Args>
void List<T, Allocator>::emplace_back(const Args&... args) {
    push_back(T(args...));
}

template <typename T, typename Allocator>
void List<T, Allocator>::insert(const T& value, Iterator& iter
ator) {
    if (iterator.current_node_ptr == block_element_ptr) {
        return;
    }

    Node* ptr = iterator.current_node_ptr->next;
    Node* new_object = AllocTraits::allocate(alloc, 1);
    AllocTraits::construct(alloc, new_object, value);

    iterator.current_node_ptr->next = new_object;
    new_object->next = ptr;

    ++size;
}

template <typename T, typename Allocator>
void List<T, Allocator>::erase(Iterator& iter) {
    if (size == 0) {
        return;
    }

    Iterator it = this->begin();

    while (it.current_node_ptr-
>next != iter.current_node_ptr) {
        ++it;
    }

    Node* temp_ptr = iter.current_node_ptr->next;
    AllocTraits::destroy(alloc, it.current_node_ptr->next);
    AllocTraits::deallocate(alloc, it.current_node_ptr-
>next, 1);
    it.current_node_ptr->next = temp_ptr;
}

template <typename T, typename Allocator>
void List<T, Allocator>::erase(Iterator& start, Iterator& end)
 {
    Iterator it = this->begin();

    if (it != start) {
        while (it.current_node_ptr-
>next != start.current_node_ptr) {
            ++it;
```

```
        }
        it.current_node_ptr->next = end.current_node_ptr;

    } else {
        head_ptr = end.current_node_ptr;
    }

    while (start != end) {
        ++it;
        AllocTraits::destroy(alloc, start.current_node_ptr);
        AllocTraits::deallocate(alloc, start.current_node_ptr,
 1);
        --size;
        start = it;
    }
}

template <typename T, typename Allocator>
std::ostream& List<T, Allocator>::operator<<(std::ostream& str
eam) {
    for (Iterator it = begin(); it != end(); ++it) {
        stream << *it;
    }

    return stream;
}

template <typename T, typename Allocator>
List<T, Allocator>::Node::Node(const T& value) {
    this->value = value;
    this->next = nullptr;
}

template <typename T, typename Allocator>
List<T, Allocator>::Node::Node(const T& value, Node* next) {
    this->value = value;
    this->next = next;
}

template <typename T, typename Allocator>
List<T, Allocator>::Iterator::Iterator(Node& node): current_no
de_ptr(std::addressof(node)) {}

template <typename T, typename Allocator>
typename List<T, Allocator>::Iterator& List<T, Allocator>::Ite
rator::operator=(Node* node_ptr) {
    current_node_ptr = node_ptr;
    return *this;
}

template <typename T, typename Allocator>
typename List<T, Allocator>::Iterator& List<T, Allocator>::Ite
rator::operator++() {
    if (current_node_ptr) {
        current_node_ptr = current_node_ptr->next;
    }
    return *this;
}

template <typename T, typename Allocator>
typename List<T, Allocator>::Iterator& List<T, Allocator>::Ite
rator::operator++(int) {
    Iterator iterator = *this;
    ++*this;
    return iterator;
}
```

```
template <typename T, typename Allocator>
bool List<T, Allocator>::Iterator::operator!=(const Iterator&
iterator) const {
    return current_node_ptr != iterator.current_node_ptr;
}

template <typename T, typename Allocator>
bool List<T, Allocator>::Iterator::operator==(const Iterator&
iterator) const {
    return current_node_ptr == iterator.current_node_ptr;
}

template <typename T, typename Allocator>
T& List<T, Allocator>::Iterator::operator*() {
    return current_node_ptr->value;
}

template <typename T, typename Allocator>
const T& List<T, Allocator>::Iterator::operator*() const {
    return current_node_ptr->value;
}

template <typename T, typename Allocator>
typename List<T, Allocator>::Iterator List<T, Allocator>::begi
n() const {
    Iterator iter(*head_ptr);
    return iter;
}

template <typename T, typename Allocator>
typename List<T, Allocator>::Iterator List<T, Allocator>::end(
) const {
    Iterator iter(*block_element_ptr);
    return iter;
}
```

Benchmark.cpp

```
#include "../include/List.hpp"
#include <forward_list>
#include <chrono>

void benchmark() {
    std::ios_base::sync_with_stdio(false);
    std::cin.tie(nullptr);
    std::cout.tie(nullptr);

    List<int> myLs;
    std::forward_list<int> stdLs;

    std::cout << "Push_front:" << std::endl;
    std::chrono::steady_clock::time_point begin = std::chrono
::steady_clock::now();
    for(size_t i = 0; i < 1000000; ++i){
        stdLs.push_front(i);
    }
    std::chrono::steady_clock::time_point end = std::chrono::
steady_clock::now();
    std::cout << std::chrono::duration_cast<std::chrono::mill
iseconds>(end - begin).count() << "[ms]" << std::endl;

    begin = std::chrono::steady_clock::now();
```

```
    for(size_t i = 0; i < 1000000; ++i){
        myLs.push_front(i);
    }
    end = std::chrono::steady_clock::now();
    std::cout << std::chrono::duration_cast<std::chrono::mill
iseconds>(end - begin).count() << "[ms]" << std::endl;
    std::cout << std::endl;

    std::cout << "Pop_front:" << std::endl;
    begin = std::chrono::steady_clock::now();
    for(size_t i = 0; i < 1000000; ++i){
        stdLs.pop_front();
    }
    end = std::chrono::steady_clock::now();
    std::cout << std::chrono::duration_cast<std::chrono::mill
iseconds>(end - begin).count() << "[ms]" << std::endl;

    begin = std::chrono::steady_clock::now();
    for(size_t i = 0; i < 1000000; ++i){
        myLs.pop_front();
    }
    end = std::chrono::steady_clock::now();
    std::cout << std::chrono::duration_cast<std::chrono::mill
iseconds>(end - begin).count() << "[ms]" << std::endl;

    std::cout << std::endl;
    std::cout << "Push_back(1000):" << std::endl;

    begin = std::chrono::steady_clock::now();
    for(size_t i = 0; i < 1000; ++i){
        myLs.push_back(i);
    }
    end = std::chrono::steady_clock::now();
    std::cout << std::chrono::duration_cast<std::chrono::mill
iseconds>(end - begin).count() << "[ms]" << std::endl;
    std::cout << std::endl;

    std::cout << "pop_back(1000):" << std::endl;
    begin = std::chrono::steady_clock::now();
    for(size_t i = 0; i < 1000; ++i){
        myLs.pop_back();
    }
    end = std::chrono::steady_clock::now();
    std::cout << std::chrono::duration_cast<std::chrono::mill
iseconds>(end - begin).count() << "[ms]" << std::endl;


}
```

main.cpp

```
#include <iostream>
#include "benchmark.cpp"
#include "../include/List.hpp"

int main() {
    benchmark();

    List<int> list;
    int left_border = 0;
    int right_border = 0;
    size_t list_size = 0;

    std::cout << "Enter list size: ";
```

```
    std::cin >> list_size;

    int array[list_size];

    std::cout << "\nEnter values of list: ";
    for (int i = list_size - 1; i >= 0; --i) {
        std::cin >> array[i];
    }
    std::cout << "\nList sucessfully created!\n";

    std::cout << "\nPlease enter left border: ";
    std::cin >> left_border;

    std::cout << "\nRight border: ";
    std::cin >> right_border;

    for (size_t i = 0; i < list_size; ++i) {
        list.push_front(array[i]);
    }

    List<int>::Iterator begin = list.begin();
    List<int>::Iterator end = list.end();

    std::cout << "\nResult: ";
    for (; begin != end; ++begin) {
        if (*begin >= left_border && *begin <= right_border)
{
            std::cout << *begin << ' ';
        }
    }
    std::cout << '\n';
}
```

Тесты производительности

Push_front:

148[ms]

86[ms]

Pop_front:

71[ms]

63[ms]

Исходя из результатов тестирования, видим, что скорость работы моего однонаправленного списка ниже, чем у стандартного списка. Я думаю, что это связано с мув семантикой, так как в своей реализации я ее не использовал. Но использовал Аллокаторы и надстройку allocator_traits. Видимо они довольно сильно замедляют работу контейнера. Также работу списка можно очень сильно ускорить передав в шаблонный параметр вместо стандартного аллокатора нетривиальный, который выделит сразу много памяти под контейнер, в следствие чего уменьшится кол-во запросов к операционной системе и выйгрыш по скорости будет довольно большимю

Заключение

В задании №8 курсовой работы я познакомился с линейными списками. Списки – классическая структура данных. Прикольно, хоть и немного лень, было написать свои итераторы для этого списка. Смысл в том, чтобы никто не мучался с тем, как работает этот список, а просто брал и юзал готовые итераторы для работы с ним. Это даёт представление о том, что надо заботиться о юзерах и о том, как мы будем писать в командной разработке.

Список литературы

1. Методические указания к выполнению курсовых работ. Зайцев В. Е.
2. https://prog-cpp.ru/data-dls/
3. https://ru.wikipedia.org/wiki/%D0%A1%D0%B2%D1%8F%D0%B7%D0%BD%D1%8B%D0%B9_%D1%81%D0%BF%D0%B8%D1%81%D0%BE%D0%BA