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:	Концебалов О.С.
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:

Переставить первую и вторую половины списка

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:

Ссылочный

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

Node.hpp

```
#ifndef NODE_HPP
#define NODE_HPP

#include <iostream>

template <typename T>
class Node{
private:
    T val;
    Node<T>* next;

public:
    Node() = default;
    Node(const T& val);
    Node(const T& val, Node<T>* next);

    T& get_val() const;
    Node<T>* get_next() const;

    void set_val(const T& val);
    void set_next(Node<T>* next);
};

#include "Node.cpp"

#endif
```

Node.cpp

```
#include "Node.hpp"

template <typename T>
```

```cpp
Node<T>::Node(const T& value): val(value){}

template <typename T>
Node<T>::Node(const T& value, Node<T>* next): val(value),
next(next){}

template <typename T>
T& Node<T>::get_val() const{
    return val;
}

template <typename T>
Node<T>* Node<T>::get_next() const{
    return next;
}

template <typename T>
void Node<T>::set_val(const T& value){
    val = value;
}

template <typename T>
void Node<T>::set_next(Node<T>* next_node){
    next = next_node;
}
```

Iterator.hpp

```cpp
#ifndef ITERATOR_HPP
#define ITERATOR_HPP

#include "Node.hpp"

template <typename T>
class List;

template <typename T>
class Iterator{
    friend class List<T>;
private:
    Node<T>* node;
```

```
public:
    Iterator() = default;
    Iterator(Node<T>* node);
    Iterator(const List<T>& lst);
    ~Iterator() = default;

    Iterator<T>& operator++();
    Iterator<T>& operator+(size_t value);

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

    bool operator==(const Iterator<T>& other) const;
    bool operator!=(const Iterator<T>& other) const;
};

#endif
```

Iterator.cpp

```
#include "Iterator.hpp"
#include "List.hpp"

template <typename T>
Iterator<T>::Iterator(Node<T>* node): node(node){}

template <typename T>
Iterator<T>::Iterator(const List<T>& lst){
    node = lst.barrier;
}

template <typename T>
Iterator<T>& Iterator<T>::operator++(){
    if (node != nullptr){
        node = node->get_next;
    }

    return *this;
}
```

```cpp
template <typename T>
Iterator<T>& Iterator<T>::operator+(size_t value){
    while (value != 0 && node != nullptr){
        node = node->get_next;
        --value;
    }

    return *this;
}

template <typename T>
T& Iterator<T>::operator*(){
    return node->get_val;
}

template <typename T>
const T& Iterator<T>::operator*() const{
    return node->get_val;
}

template <typename T>
bool Iterator<T>::operator==(const Iterator<T>& other) const{
    return this->node == other.node;
}

template <typename T>
bool Iterator<T>::operator!=(const Iterator<T>& other) const{
    return !(this->node == other.node);
}
```

List.hpp

```cpp
#ifndef LIST_HPP
#define LIST_HPP

#include "Node.hpp"
#include "Iterator.hpp"
#include <iostream>

template <typename T>
class Iterator;
```

```
template <typename T>
class List{
    friend class Iterator<T>;

private:
    Node<T>* barrier;
    size_t list_size;

public:
    List();
    ~List();

    Iterator<T> begin() const;
    Iterator<T> end() const;

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

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

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

    void swap_halfs();
    std::ostream& operator<<(std::ostream& os);
};

#endif
```

List.cpp

```
#include "List.hpp"
#include "Node.hpp"
#include "Node.cpp"
```

```cpp
#include "Iterator.hpp"
#include "Iterator.cpp"
#include <iostream>

template <typename T>
List<T>::List(){
    barrier = new Node<T>();
    barrier->set_next(barrier);
    list_size = 0;
}

template <typename T>
List<T>::~List(){
    Iterator<T> cur = ++begin();
    Iterator<T> to_delet = cur;

    while (cur.node != barrier){
        ++cur;
        delete to_delet.node;
        to_delet = cur;
    }

    delete cur.node;
}

template <typename T>
Iterator<T> List<T>::begin() const{
    return Iterator<T>(*this);
}

template <typename T>
Iterator<T> List<T>::end() const{
    Node<T>* end = barrier;

    while (end->get_next != barrier){
        end = end->get_next;
    }
    return Iterator<T>(end);
}
```

```
template <typename T>
void List<T>::push_front(const T& value){
    if (begin().node->get_next == barrier){
        begin().node->set_next(new Node<T>(value, barrier));
        barrier = begin().node->get_next;

        ++list_size;
        return;
    }

    begin().node->set_next(new Node<T>(*(begin()),
begin().node->get_next));
    *(begin()) = value;
    ++list_size;
}

template <typename T>
void List<T>::push_back(const T& value){
    if (begin() == end()){
        begin().node->set_next(new Node<T>(value, barrier));
        barrier = begin().node->get_next;

        ++list_size;
        return;
    }

    Iterator<T> iterator = begin();
    while (iterator.node->next->next != barrier){
        ++iterator;
    }

    iterator.node->set_next(new Node<T>(value, iterator.node-
>next));
    ++list_size;
}

template <typename T>
void List<T>::pop_front(){
    if (begin().node->get_next == barrier) return;
```

```
    Iterator<T> it = ++begin();
    begin().node->set_val((++begin()).node->get_val);
    begin().node->set_next((++begin()).node->get_next);

    delete it.node;
    --list_size;
}

template <typename T>
void List<T>::pop_back(){
    if (begin().node->get_next == barrier) return;

    Iterator<T> it = ++end();
    while (it.node->next->next != barrier){
        ++it;
    }

    delete it.node->get_next;
    it.node->set_next(barrier);
    --list_size;
}

template <typename T>
void List<T>::insert(const Iterator<T>& it, const T& value){
    it.node->set_next(new Node<T>(value, it.node->next));
    ++list_size;
}

template <typename T>
void List<T>::erase(const Iterator<T>& start, const
Iterator<T>& end){
    Iterator<T> it = start;
    ++it;
    start.node->set_next(end.node);

    while (it != end){
        if (it == begin()) start = end.node;

        Node<T>* to_delet = it.node;
        ++it;
```

```
        delete to_delet;
        --list_size;
    }
}

template <typename T>
template <typename... Args>
void List<T>::emplace_front(const Args&... args){
    push_front(T(args...));
}

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

template <typename T>
void List<T>::swap_halfs(){
    if (barrier == nullptr) return;

    Iterator<T> first_half = begin();
    Iterator<T> second_half = begin();
    for (size_t i = 0; i != list_size / 2; ++i){
        ++second_half;
    }

    while (second_half.node->get_next != barrier){
        T swap = first_half.node->get_val;
        first_half.node->set_val(second_half.node->get_val);
        second_half.node->set_val(swap);

        ++first_half;
        ++second_half;
    }
}

template <typename T>
std::ostream& List<T>::operator<<(std::ostream& os){
    Node<T>* cur = barrier;
```

```
    while (cur->get_next != barrier){
        os << cur->val << " ";
        cur = cur->get_next;
    }

    return os;
}
```

Benchmark.cpp

Run.cpp

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

int main(){
    List<int> lst;

    lst.push_back(10);
    lst.push_back(15);
    lst.push_back(16);
    lst.push_back(12);
    lst.push_back(18);
    lst.push_front(5);
    lst.push_front(20);

    lst.swap_halfs();

    bencmark();
}
```

Тесты производительности

Функция push_front

	10^4	10^5	10^6
myList	1[ms]	5[ms]	57[ms]
Std::forward_list	1[ms]	7[ms]	76[ms]

Функция pop_front

	10^4	10^5	10^6
myList	0[ms]	5[ms]	58[ms]
Std::forward_list	0[ms]	5[ms]	43[ms]

Исходя из результатов тестирования, видим, что скорость работы моего однонаправленного списка примерно в 1,5 раза выше, чем у списка из STL при добавлении элемента и немного ниже (примерно в 1,1 раза) при удалении элемента. Предполагаю, что это связано со схожей реализацией этих функций, а также с оптимизацией удаления и каких-то дополнительных процедурах и методах при добавлении элемента.

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