Отчет по лабораторной работе № 26 по курсу

«Практикум программирования»

Студент группы М8О-109Б-22 Ефименко Кирилл Игоревич

Контакты: telegram @vivichv9

Работа выполнена: 3.06.2023

Преподаватель: каф.806 Сысоев Максим Алексеевич

Отчет сдан «25» июня 2023 г., итоговая оценка ____

Подпись преподавателя _____

1. Тема: Абстрактные типы данных. Рекурсия. Модульное программирование на C++

2. Цель работы: Реализовать свою структуру данных и написать сортировку для нее

3. Задание (вариант № 3, 2):

Структура: Deque

2.	Процедура:	Вставка элемента в стек, дек, список или очередь, упорядоченные по возрастанию, с сохранением порядка
	Метод:	сортировка простой вставкой

4. Оборудование (студента):

Процессор AMD Ryzen 5 5600H with Radeon Graphics 3.30 GHz, ОП 16,0 Гб, SSD 512 Гб. Монитор 1920x1080 144 Hz

5. Программное обеспечение (студента):

Операционная система семейства Linux, наименование Ubuntu, версия 18.10

Интерпретатор команд: bash, версия 4.4.19

Система программирования – версия --, редактор текстов Emacs, версия 25.2.2

Утилиты операционной системы –

Прикладные системы и программы –

Местонахождение и имена файлов программ и данных на домашнем компьютере –

6. Идея, метод, алгоритм решения задачи *(в формах: словесной, псевдокода, графической [блок-схема, диаграмма, рисунок, таблица] или формальные спецификации с пред- и постусловиями)*

Все просто. Пишу дек и сортировку для него

7. Сценарий выполнения работы *(план работы, первоначальный текст программы в черновике [можно на отдельном листе] и тесты, либо соображения по тестированию)*

1. Пишу свой дек
2. Пишу сортировку
3. Тестирую

8. Распечатка протокола *(подклеить листинг окончательного варианта программы с тестовыми примерами, подписанный преподавателем)*

Node.hpp

```
#ifndef NODE_HPP_INCLUDED
#define NODE_HPP_INCLUDED

template <typename T>
class Node {
public:
    T value;
    Node<T>* next = nullptr;
    Node<T>* prev = nullptr;

public:
    Node() = default;
    Node(const T& value);
    Node(const T& value, Node<T>* prev, Node<T>* current_node);

    void set_value(const T& value);

};

#include "../src/Node.cpp"

#endif
```

Node.cpp

```
#include "../include/Node.hpp"

template <typename T>
Node<T>::Node(const T& value): value(value) {}

template <typename T>
Node<T>::Node(const T& value, Node<T>* prev, Node<T>* current_node): v
alue(value), next(current_node), prev(prev) {}

template <typename T>
void Node<T>::set_value(const T& value) {
    this->value = value;
}
```

Deque.hpp

```
#ifndef DEQUE_HPP_INCLUDED
#define DEQUE_HPP_INCLUDED

#include "Node.hpp"

template <typename T, typename Allocator = std::allocator<T>>
class Deque {
private:
    Node<T>* head_ptr = nullptr;
    Node<T>* block_ptr = nullptr;
    size_t size = 0;
```

```
public:
    class Iterator;

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

    Deque();
    Deque(const std::initializer_list<T>& list);
    ~Deque();

    bool empty() const;
    size_t get_size() const;
    void insert(Iterator& it, const T& value);
    void erase(Iterator& it);
    void erase(Iterator& start, Iterator& end);
    void clear();
    void push_front(const T& value);
    void pop_front();
    void push_back(const T& value);
    void pop_back();
    void swap_nodes(Iterator& it_1, Iterator& it_2);
    void set_val(Iterator& it, T& value);
    void task_procedure(Iterator& it, bool& flag, Deque<T, Allocator>&
 deque);
    void insert_sort();

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

    T& operator[](size_t index);
    Iterator& at(size_t index);
    Iterator begin();
    Iterator end();

    class Iterator {
        friend class Deque<T, Allocator>;
    private:
        Node<T>* current_node_ptr = nullptr;

    public:
        Iterator(Node<T>* ptr);
        T& operator*();
        Iterator& operator=(const Iterator& it);
        Iterator& operator++();
        Iterator& operator++(int);
        Iterator& operator--();
        Iterator& operator--(int);
        Iterator& operator+=(size_t n);
        Iterator& operator-=(size_t n);

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

#include "../src/Deque.cpp"

#endif
```

Deque.cpp

```
#include <iostream>
#include "../include/Deque.hpp"

template <typename T, typename Allocator>
Deque<T, Allocator>::Deque() {
    block_ptr = AllocTraits::allocate(alloc, 1);
    AllocTraits::construct(alloc, block_ptr);
}

template <typename T, typename Allocator>
Deque<T, Allocator>::Deque(const std::initializer_list<T>& list) {
    block_ptr = AllocTraits::allocate(alloc, 1);
    AllocTraits::construct(alloc, block_ptr);

    for (auto it = list.begin(); it != list.end(); ++it) {
        push_back(*it);
    }
}

template <typename T, typename Allocator>
Deque<T, Allocator>::~Deque() {
    while (head_ptr) {
        Node<T>* temp_ptr = head_ptr;
        head_ptr = head_ptr->next;

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

template <typename T, typename Allocator>
bool Deque<T, Allocator>::empty() const {
    return size == 0;
}

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

template <typename T, typename Allocator>
void Deque<T, Allocator>::swap_nodes(Iterator& it_1, Iterator& it_2) {
    if (head_ptr == nullptr || head_ptr-
>next == nullptr || it_1 == it_2)
        return;

    Node<T>* Node1 = it_1.current_node_ptr;
    Node<T>* Node2 = it_2.current_node_ptr;

    if (Node1 == head_ptr)
        head_ptr = Node2;
    else if (Node2 == head_ptr)
        head_ptr = Node1;
    if (Node1 == block_ptr->prev)
        block_ptr->prev = Node2;
    else if (Node2 == block_ptr->prev)
        block_ptr->prev = Node1;

    Node<T>* temp;
```

```
    temp = Node1->next;
    Node1->next = Node2->next;
    Node2->next = temp;

    if (Node1->next != nullptr)
        Node1->next->prev = Node1;
    if (Node2->next != nullptr)
        Node2->next->prev = Node2;

    temp = Node1->prev;
    Node1->prev = Node2->prev;
    Node2->prev = temp;

    if (Node1->prev != nullptr)
        Node1->prev->next = Node1;
    if (Node2->prev != nullptr)
        Node2->prev->next = Node2;

}

template <typename T, typename Allocator>
void Deque<T, Allocator>::set_val(Iterator& it, T& val) {
    it.current_node_ptr->set_value(val);
}

template <typename T, typename Allocator>
void Deque<T, Allocator>::insert(Iterator& it, const T& value) {
    Iterator it_begin = begin();

    if (size == 0 || it_begin == it) {
        push_front(value);
        return;
    }

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

    Node<T>* ptr = AllocTraits::allocate(alloc, 1);
    AllocTraits::construct(alloc, ptr, value, it_begin.current_node_pt
r, it.current_node_ptr);

    it_begin.current_node_ptr->next = ptr;
    it.current_node_ptr->prev = ptr;
    ++size;
}

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

    Iterator it = begin();

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

    Node<T>* 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;
    temp_ptr->prev = it.current_node_ptr;
}

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

    Iterator it = begin();

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

        it.current_node_ptr->next = end.current_node_ptr;
        end.current_node_ptr->prev = it.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>
void Deque<T, Allocator>::clear() {
    while (head_ptr != nullptr) {
        Node<T>* tmp = head_ptr;
        head_ptr = head_ptr->next;
        AllocTraits::destroy(alloc, tmp);
        AllocTraits::deallocate(alloc, tmp, 1);
    }
    size = 0;
}

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

    if (head_ptr == nullptr) {
        head_ptr = ptr;
        ptr->next = block_ptr;
        block_ptr->prev = ptr;
    } else {
        swap_ptr = head_ptr;
        head_ptr = ptr;
        head_ptr->next = swap_ptr;
        swap_ptr->prev = head_ptr;
    }
```

```
    ++size;
}

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

    if (size == 1) {
        AllocTraits::destroy(alloc, head_ptr);
        AllocTraits::deallocate(alloc, head_ptr, 1);
        block_ptr->prev = nullptr;
        head_ptr = nullptr;
        --size;
        return;
    }

    head_ptr = head_ptr->next;
    AllocTraits::destroy(alloc, head_ptr->prev);
    AllocTraits::deallocate(alloc, head_ptr->prev, 1);
    head_ptr->prev = nullptr;
    --size;
}

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

    Node<T>* prev = block_ptr->prev;
    Node<T>* ptr = AllocTraits::allocate(alloc, 1);
    AllocTraits::construct(alloc, ptr, value, prev, block_ptr);
    prev->next = ptr;
    block_ptr->prev = ptr;
    ++size;
}

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

    if (size == 1) {
        AllocTraits::destroy(alloc, head_ptr);
        AllocTraits::deallocate(alloc, head_ptr, 1);
        block_ptr->prev = nullptr;
        head_ptr = nullptr;
        --size;
        return;
    }

    Node<T>* ptr = block_ptr->prev;
    Node<T>* ptr_prev = ptr->prev;

    AllocTraits::destroy(alloc, ptr);
    AllocTraits::deallocate(alloc, ptr, 1);
```

```
    ptr_prev->next = block_ptr;
    block_ptr->prev = ptr_prev;
    --size;
}

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

template <typename T, typename Allocator>
T& Deque<T, Allocator>::operator[](size_t index) {
    if (index >= size) {
        throw std::range_error("index out of range");
    }

    if (index < size / 2) {
        Iterator it = begin();
        for (size_t i = 0; i < index; ++i, ++it) {}
        return *it;
    } else {
        Iterator it = end();
        for (size_t i = size - 1; i >= index; --i, --it) {}
        return *it;
    }
}

template <typename T, typename Allocator>
typename Deque<T, Allocator>::Iterator& Deque<T, Allocator>::at(size_t
 index) {
    if (index >= size) {
        throw std::range_error("index out of range");
    }


    if (index < size / 2) {
        Iterator it = begin();
        for(size_t i = 0; i < index; ++i, ++it) {}
        return it;
    } else {
        Iterator it = end();
        for (size_t i = size - 1; i >= index; --i, --it) {}
        return it;
    }
}

template <typename T, typename Allocator>
typename Deque<T, Allocator>::Iterator Deque<T, Allocator>::begin() {
    Iterator iter(head_ptr);
    return iter;
}

template <typename T, typename Allocator>
typename Deque<T, Allocator>::Iterator Deque<T, Allocator>::end() {
    Iterator iter(block_ptr);
    return iter;
}

template <typename T, typename Allocator>
Deque<T, Allocator>::Iterator::Iterator(Node<T>* ptr): current_node_pt
```

```
r(ptr) {}

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

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

template <typename T, typename Allocator>
typename Deque<T, Allocator>::Iterator& Deque<T, Allocator>::Iterator:
:operator--() {
    if (current_node_ptr) {
        current_node_ptr = current_node_ptr->prev;
    }
    return *this;
}

template <typename T, typename Allocator>
typename Deque<T, Allocator>::Iterator& Deque<T, Allocator>::Iterator:
:operator--(int) {
    Iterator iterator = *this;
    --*this;
    return iterator;
}

template <typename T, typename Allocator>
typename Deque<T, Allocator>::Iterator& Deque<T, Allocator>::Iterator:
:operator=(const Iterator& it) {
    current_node_ptr = it.current_node_ptr;;
    return *this;
}

template <typename T, typename Allocator>
bool Deque<T, Allocator>::Iterator::operator==(const Iterator& it) con
st {
    return current_node_ptr == it.current_node_ptr;
}

template <typename T, typename Allocator>
bool Deque<T, Allocator>::Iterator::operator!=(const Iterator& it) con
st {
    return current_node_ptr != it.current_node_ptr;
}

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

template <typename T, typename Allocator>
```

```
typename Deque<T, Allocator>::Iterator& Deque<T, Allocator>::Iterator:
:operator+=(size_t n) {
    Iterator& it = *this;
    for (size_t i = 0; i < n; ++i) {
        ++it;
    }
    return it;
}

template <typename T, typename Allocator>
typename Deque<T, Allocator>::Iterator& Deque<T, Allocator>::Iterator:
:operator-=(size_t n) {
    Iterator& it = *this;
    for (size_t i = 0; i < n; ++i, --it) {}
    return it;
}

template <typename T, typename Allocator>
void Deque<T, Allocator>::task_procedure(typename Deque<T, Allocator>:
:Iterator& it, bool& flag, Deque<T, Allocator>& deque) {
    auto temp_iter = deque.begin();
    while (temp_iter != it) {
        if (*it < *temp_iter) {
            deque.insert(temp_iter, *it);
            auto it_copy = it;
            flag = true;
            ++it;
            deque.erase(it_copy);
            break;
        }
        ++temp_iter;
    }
}

template <typename T, typename Allocator>
void Deque<T, Allocator>::insert_sort() {
    auto it = begin();
    bool flag = false;
    ++it;
    for (; it != end();) {
        flag = false;
        task_procedure(it, flag, *this);
        if (!flag) {
            ++it;
        }
    }
}
```

Run.cpp

```
#include <iostream>
#include "../include/Deque.hpp"

int main() {
    Deque<int> deque = {123, 23, 454, 654, 32, 42, 1, 5, 8};

    deque.insert_sort();

    auto ite = deque.begin();
    while (ite != deque.end()) {
        std::cout << *ite << ' ';
        ++ite;
    }
}
```

9. Дневник отладки *(дата и время сеансов отладки и основные события [ошибки в сценарии и программе, нестандартные ситуации] и краткие комментарии к ним. В дневнике отладки приводятся сведения об использовании других ЭВМ, существенном участии преподавателя и других лиц в написании и отладке программы)*

№	*Лаб. или дом*	*Дата*	*Время*	*Событие*	*Действие по исправлению*	*Примечания*

Особых проблем при выполнении лабы не возникло

10. Замечания автора *(по существу работы)*

Замечания отсутствуют

11. Вывод

Благодаря данной лабе изучил новую для себя структуры дэк. Также улучшил свои навыки в написании сортировки. Лаба понравилась, была очень интересная.

Подпись студента _____