

ОТЧЕТ ПО ЛАБОРАТОРНОЙ РАБОТЕ № 1

Выполнил студент группы.....КС-30 Лихолат Полина Николаевна

Ссылка на репозиторий:.....https://github.com/MUCTR-IKT-CPP/Likholat_algorithms

Приняли:Пысин Максим Дмитриевич

.....Краснов Дмитрий Олегович

Дата сдачи: 13.03.2023

Оглавление

Описание задачи.....	2
Описание метода/модели.....	4
Выполнение задачи.	6
Заключение.	12

Описание задачи.

Вариант 3

Необходимо изучить и реализовать очередь, при этом, структура должна:

- Использовать шаблонный подход, обеспечивая работу контейнера с произвольными данными.
- Реализовывать свой итератор предоставляющий стандартный для языка механизм работы с ним
- Обеспечивать работу стандартных библиотек и конструкции `for each` если она есть в языке, если их нет, то реализовать собственную функцию использующую итератор.
- Проверку на пустоту и подсчет количества элементов.
- Операцию сортировки с использованием стандартной библиотеки.

Очередь, операции:

- добавление в конец
- взятие с начала

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 заполнения 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 используя структуру из стандартной библиотеки и сравнив результат.
- Инверсировать содержимое контейнера заполненного отсортированными по возрастанию элементами не используя операцию перемещения при помощи итератора, а только операторы изъятия и вставки.

Описание метода/модели.

Вариант 3

Очередь – структура данных типа «список», позволяющая добавлять элементы лишь в конец списка, и извлекать их из его начала.

Очередь подчиняется принципу FIFO — Firts In First Out («первый пришел — первый вышел»). Первый элемент в очереди выходит первым.



Рисунок 1.0.1 Добавление и удаление элементов в очереди

Элемент 1 поступает в очередь до 2 и удаляется первым (рис. 1.1) — это и есть принцип FIFO.

Enqueue — метод, который добавляет элемент в очередь. Dequeue, наоборот, удаляет его.

Очередь работает следующим образом:

- Реализуется два указателя — **FRONT** и **REAR**.
- **FRONT** — указатель на первый элемент очереди.
- **REAR** — указатель на последний элемент очереди.
- Значения **FRONT** и **REAR** изначально должны быть равны -1.

Очередь — абстрактный тип данных. Он поддерживает такие операции:

- **Enqueue** — позволяет добавить элемент в конец очереди.
- **Pop** — позволяет удалить элемент из начала очереди.
- **IsEmpty** — проверяет, пуста ли очередь.
- **IsFull** — проверяет, заполнена ли очередь.
- **Peek** — позволяет получить элемент в начале очереди без его удаления.

Недостатки очереди

На рисунке 1.2 можно заметить, что после нескольких операций удаления и добавления элементов размер очереди увеличивается.

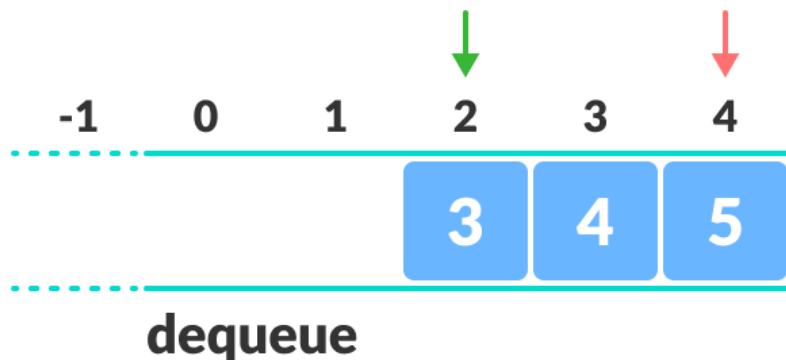


Рисунок 1.0.1. Изменение размера очереди

Теперь использовать индексы 0 и 1 мы не можем — очередь нужно сбросить в изначальное состояние, т. е. удалить все элементы из очереди.

Существует модификация очереди, исправляющая этот недостаток — круговая очередь. Она позволяет сместить указатель `REAR`, когда он достигает конца очереди. Благодаря этому мы можем вновь использовать освободившееся пространство.

Временная сложность очередей

Сложность операций `enqueue` и `dequeue` очереди, реализованной с помощью массивов, — $O(1)$.

Выполнение задачи.

Для реализации очереди использовался язык C++. Для начала подключаем нужные библиотеки.

```
#include <iostream> // ввод вывод
#include <vector> // шаблон класса для контейнеров последовательностей
#include <algorithm> // определяет функции шаблона контейнера стандартной библиотеки C++, кото-
рые выполняют алгоритмы
#include <random> // определяет средства для генерации случайных чисел
#include <string> // для организации работы со строками
```

Далее реализуем шаблонный класс с необходимыми методами:

```
template <typename T>
class Queue {
private:
    struct Node {
        T data;
        Node* next;
        Node(const T& d) : data(d), next(nullptr) {}
    };

    Node* head_;
    Node* tail_;
    size_t size_;

public:
    Queue() : head_(nullptr), tail_(nullptr), size_(0) {}

    // Add an element to the back of the queue
    void push(const T& element) {
        Node* new_node = new Node(element);
        if (tail_ == nullptr) {
            head_ = new_node;
            tail_ = new_node;
        }
        else {
            tail_>next = new_node;
            tail_ = new_node;
        }
        size_++;
    }

    // Remove the first element from the queue
    void pop() {
        if (head_ == nullptr) {
            throw std::out_of_range("Queue is empty");
        }
        Node* old_head = head_;
        head_ = old_head->next;
        delete old_head;
        size_--;
        if (head_ == nullptr) {
            tail_ = nullptr;
        }
    }

    // Get the first element from the queue
    T& front() const {
        if (head_ == nullptr) {
            throw std::out_of_range("Queue is empty");
        }
        return head_>data;
    }

    // Check if the queue is empty
    bool isEmpty() const {
        return head_ == nullptr;
    }
};
```

```

}

// Get the number of elements in the queue
size_t size() const {
    return size_;
}

// Sort using std::sort
void sortqueue() {
    std::vector<T> temp;
    while (!isEmpty()) {
        temp.push_back(front());
        pop();
    }
    std::sort(temp.begin(), temp.end());
    for (const T& t : temp) {
        push(t);
    }
}

// Reverse using only pop() and push() operations
void reverse() {
    std::stack<T> s;
    // push elements from the queue onto the stack
    while (!isEmpty()) {
        s.push(front());
        pop();
    }

    // pop elements from the stack and push them back onto the queue
    while (!s.empty()) {
        push(s.top());
        s.pop();
    }
}

```

Затем пишем класс итератора:

```

// Iterator
class Iterator {
private:
    Node* current_;

public:
    Iterator(Node* current = nullptr) : current_(current) {}

    T& operator*() {
        if (current_ == nullptr) {
            throw std::out_of_range("Iterator is out of range");
        }
        return current_>data;
    }

    Iterator& operator++() {
        if (current_ != nullptr) {
            current_ = current_>next;
        }
        return *this;
    }

    bool operator!=(const Iterator& other) const {
        return current_ != other.current_;
    }
};

Iterator begin() {
    return Iterator(head_);
}

Iterator end() {
    return Iterator(nullptr);
}

```

```
}
```

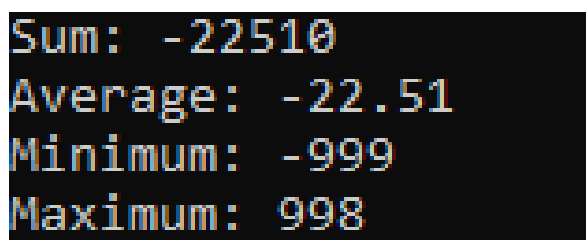
```
};
```

Мы реализовали структуру очереди и можем приступить к тестам.

1. Заполнение контейнера 1000 целыми числами в диапазоне от -1000 до 1000 и подсчет их суммы, среднего, минимального и максимального.

```
void testQueue(int N) {  
    // Initialize the random number generator  
    std::random_device rd;  
    std::mt19937 gen(rd());  
    std::uniform_int_distribution<int> distr(-1000, 1000);  
  
    // Create a new queue  
    Queue<int> myqueue;  
  
    // Fill the list with 1000 random integers  
    for (int i = 0; i < N; ++i) {  
        myqueue.push(distr(gen));  
    }  
  
    // Compute the sum, average, minimum, and maximum of the list  
    int sum = 0;  
    int min = std::numeric_limits<int>::max();  
    int max = std::numeric_limits<int>::min();  
  
    for (auto it : myqueue) {  
        sum += it;  
        if (it < min) {  
            min = it;  
        }  
        if (it > max) {  
            max = it;  
        }  
    }  
  
    double avg = static_cast<double>(sum) / static_cast<double>(myqueue.size());  
  
    // Output the results  
    std::cout << "Sum: " << sum << std::endl;  
    std::cout << "Average: " << avg << std::endl;  
    std::cout << "Minimum: " << min << std::endl;  
    std::cout << "Maximum: " << max << std::endl;  
}
```

Вызываем функцию в main и получаем следующие результаты (рис.2.1):



```
Sum: -22510  
Average: -22.51  
Minimum: -999  
Maximum: 998
```

Рисунок 2.1. Результаты теста 1

2. Провести проверку работы операций вставки и изъятия элементов на коллекции из 10 строковых элементов.

```
void testString(int N)
{
    Queue<std::string> myqueue;

    for (int i = 0; i < N; i++) {
        myqueue.push("old");
    }

    std::cout << "Queue before adding element: ";
    for (auto it : myqueue) {
        std::cout << it << " ";
    }
    std::cout << std::endl;

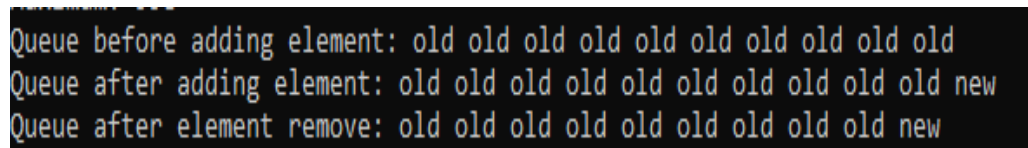
    myqueue.push("new");
    std::cout << "Queue after adding element: ";
    for (auto it : myqueue) {
        std::cout << it << " ";
    }

    std::cout << std::endl;

    myqueue.pop();
    std::cout << "Queue after element remove: ";
    for (auto it : myqueue) {
        std::cout << it << " ";
    }

    std::cout << std::endl;
}
```

Вызываем функцию в main и получаем следующие результаты (рис.2.2):



```
Queue before adding element: old old old old old old old old old old
Queue after adding element: old old old old old old old old old old new
Queue after element remove: old old old old old old old old old new
```

Рисунок 2.2. Результаты теста 2

3. 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 заполнения 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.

```
std::vector<std::string> surnames = { "Kazanova", "Petrova", "Sidorova", "Kalich", "Tokareva",
    "Zubareva", "Maksimova", "Pupkina", "Petushkova", "Kuricina" };
std::vector<std::string> names = { "Polina", "Anna", "Elizaveta", "Alice", "Maria",
    "Katerina", "Julia", "Anastasia", "Agata", "Olga" };
std::vector<std::string> patronymics = { "Nikolaevna", "Vadimovna", "Dmitrievna",
    "Maksimovna", "Ivanovna", "Vitalievna", "Semenovna", "Vladimirovna", "Egorovna", "Artemovna"
};
```

```

// Structure for holding personal data
struct Person {
    std::string surname;
    std::string name;
    std::string patronymic;
    std::string dob;
};

int randomInt(int min, int max) {
    return rand() % (max - min + 1) + min;
}

std::string randomDob() {
    int day = randomInt(1, 31);
    int month = randomInt(1, 12);
    int year = randomInt(1980, 2020);
    return std::to_string(day) + "/" + std::to_string(month) + "/" + std::to_string(year);
}

std::string randomName(const std::vector<std::string>& names) {
    return names[randomInt(0, names.size() - 1)];
}

void testStructure()
{
    srand(time(0));
    Queue<Person> people;
    for (int i = 0; i < 100; i++) {
        Person person;
        person.surname = randomName(surnames);
        person.name = randomName(names);
        person.patronymic = randomName(patronymics);
        person.dob = randomDob();
        people.push(person);
    }

    Queue<Person> under20;
    Queue<Person> over30;
    Queue<Person> sortedpeople;
    for (const auto& person : people) {
        int year = std::stoi(person.dob.substr(6, 4));
        int age = 2023 - year;
        if (age < 20) {
            under20.push(person);
            sortedpeople.push(person);
        }
        else if (age > 30) {
            over30.push(person);
            sortedpeople.push(person);
        }
    }

    int mistakesCount = 0;
    for (const auto& person : sortedpeople) {
        int year = std::stoi(person.dob.substr(6, 4));
        int age = 2023 - year;
        if ((age > 20) && (age < 30)) {
            mistakesCount++;
        }
    }

    std::cout << "Number of people under 20: " << under20.size() << std::endl;
    std::cout << "Number of people over 30: " << over30.size() << std::endl;
    std::cout << "Number of people under 20 and over 30: " << sortedpeople.size() <<
std::endl;
    std::cout << "Number of people between 20 and 30: " << mistakesCount << std::endl;
}

```

Вызываем функцию в main и получаем следующие результаты (рис.2.3):

```

Number of people under 20: 11
Number of people over 30: 86
Number of people under 20 and over 30: 97
Number of people between 20 and 30: 0

```

Рисунок 2.3. Результаты теста 3

4. Заполнить структуру 1000 элементов и отсортировать ее, проверить правильность используя структуру из стандартной библиотеки и сравнив результат.

Для наглядности количество элементов уменьшим до 10

```

void testSort(int N)
{
    std::random_device rd;
    std::mt19937 gen(rd());
    std::uniform_int_distribution<int> distr(-1000, 1000);

    // Create a new Queue
    Queue<int> myqueue;
    std::vector<int> vec(N);

    // Fill the list with 1000 random integers
    for (int i = 0; i < N; ++i) {
        myqueue.push(distr(gen));
    }

    int i = 0;
    for (auto l : myqueue) {
        vec[i] = l;
        i++;
    }

    std::sort(vec.begin(), vec.end());

    myqueue.sortqueue();

    std::cout << "Sorted queue" << std::endl;
    i = 0;
    int mistakes = 0;
    for (auto l : myqueue) {
        if (l != vec[i]) {
            mistakes++;
        }
        std::cout << l << std::endl;
        i++;
    }

    std::cout << "Sorted vector" << std::endl;
    for (auto l : vec) {
        std::cout << l << std::endl;
    }

    std::cout << "Number of mistakes: " << mistakes << std::endl;
}

```

Вызываем функцию в main и получаем следующие результаты (рис.2.4):

```
Sorted queue
-856
-443
-26
-22
31
499
577
657
885
922
Sorted vector
-856
-443
-26
-22
31
499
577
657
885
922
Number of mistakes: 0
```

Рисунок 2.4. Результаты теста 4

5. Инвертировать содержимое контейнера заполненного отсортированными по возрастанию элементами не используя операцию перемещения при помощи итератора, а только операторы изъятия и вставки.

Дополним функцию testSort следующими строками и получим инвертированную очередь (рис.2.5):

```
std::cout << "Inverted queue" << std::endl;
myqueue.reverse();
for (auto l : myqueue) {
    std::cout << l << std::endl;
}
```

```
Inverted queue
922
885
657
577
499
31
-22
-26
-443
-856
```

Рисунок 2.5. Результаты теста 5

Заключение.

Очередь – простейшая структура данных с чётко определённым набором поддерживаемых операций.

Использование данной структуры будет удобно в следующих примерах:

- Планирование процессов и работы жесткого диска.
- Для синхронизации данных, перемещаемых между двумя процессами. Например: буферы ввода-вывода, конвейеры, файловые буферы ввода-вывода и т. д.
- Обработка прерываний в системах реального времени .
- Телефоны в колл-центрах используют очереди, чтобы обслуживать клиентов по порядку.

