Министерство науки и высшего образования Российской Федерации

Федеральное государственное бюджетное образовательное учреждение высшего образования «Российский химико-технологический университет имени Д.И. Менделеева»

Факультет цифровых технологий и химического инжиниринга

Кафедра информационных компьютерных технологий

ОТЧЕТ ПО ЛАБОРАТОРНОЙ РАБОТЕ

№ 3 ПО КУРСУ

«АЛГОРИТМЫ И СТРУКТУРЫ ДАННЫХ»:

СТУДЕНТ группы КС-33 Костяева К.С.

Москва
2024

ОГЛАВЛЕНИЕ

ТЕОРИЯ

ListQueue — это реализация очереди, основанная на односвязном списке. Очередь — это структура данных, работающая по принципу FIFO (First In, First Out — "первым пришел, первым ушел"), где элементы добавляются в конец (tail), а извлекаются с начала (head).

StackQueue — это реализация очереди, использующая два стека из стандартной библиотеки STL (std::stack). Стек — это структура данных типа LIFO (Last In, First Out — "последним пришел, первым ушел"), но с помощью двух стеков можно эмулировать поведение FIFO.

Алгоритм:

ListQueue(Односвязный список):

1. Структура узла: Каждый элемент очереди представлен структурой Node, содержащей данные (data) и указатель на следующий узел (next).
2. Указатели:
   - head указывает на первый узел (начало очереди).
   - tail указывает на последний узел (конец очереди).
3. Операции:
   - Добавление (enqueue): Новый узел создается с помощью new Node(value), добавляется в конец списка, обновляется tail. Если очередь пуста, head тоже указывает на новый узел.
   - Извлечение (dequeue): Удаляется первый узел (head), возвращается его значение, head перемещается на следующий узел. Если очередь становится пустой, tail обнуляется.
4. Подсчет размера: Переменная size увеличивается при добавлении и уменьшается при извлечении.
5. Итератор: Реализован для обхода элементов с помощью операций ++ (переход к следующему узлу) и != (сравнение текущего положения).

StackQueue(Очередь через два стека):

1. Два стека:
   - inStack: Служит для добавления новых элементов.
   - outStack: Служит для извлечения элементов.
2. Операции:
   - Добавление (enqueue): Элемент просто помещается в inStack через push.
   - Извлечение (dequeue):
     - Если outStack пуст, все элементы из inStack переносятся в outStack (в обратном порядке) с помощью функции transfer().
     - Извлекается верхний элемент из outStack через pop.

    - Перенос (transfer): Пока inStack не пуст, элементы извлекаются (pop) и помещаются в outStack (push), меняя порядок на противоположный.
- Принцип инверсии: Стек меняет порядок элементов на обратный. Два стека вместе позволяют получить FIFO:
    - Первый стек (inStack) принимает элементы в порядке поступления (например, 1, 2, 3).
    - При переносе в outStack порядок становится обратным (3, 2, 1).
    - Извлечение из outStack дает элементы в порядке FIFO (1, 2, 3).

ЗАДАНИЕ

Написать две реализации очереди (Очередь через односвязный список и очередь через два стека(стек из библиотеки))

- Добавление в конец
- Взятие с начала

В рамках лабораторной работы необходимо изучить и реализовать очередь, при этом, все структуры должны:

- Использовать шаблонный подход, обеспечивая работу контейнера с произвольными данными.
- Реализовывать свой итератор предоставляющий стандартный для языка механизм работы с ним(для C++ это операции ++ и операция !=, для python это)
- 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.

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. Тесты по вариантам:

2. Очередь

- Инверсировать содержимое контейнера заполненного отсортированными по возрастанию элементами не используя операцию перемещения при помощи итератора, а только операторы изъятия и вставки.
- Сравнить две реализации между собой (Сравнить на основании скорости выполнения операции вставки и изъятия на контейнере, использования памяти на все элементы), тестировать для коллекции состоящей из 10000 элементов.

ПРАКТИЧЕСКАЯ ЧАСТЬ

Код:

язык программирования C++

```
#include<iostream>
#include<stack>
#include<string>
#include<random>
#include<chrono>
#include<vector>

usingnamespace std;

// Структура для теста с ФИО и датой рождения
structPerson{
string surname, name, patronymic;
int birthYear;
};

// 1. Очередь через односвязный список
template<typenameT>
classListQueue{
private:
structNode {
T data;
Node* next;
        Node(constT& value) : data(value), next(nullptr) {}
    };

Node* head;
Node* tail;
size_t size;

public:
    ListQueue() : head(nullptr), tail(nullptr), size(0) {}

    ~ListQueue() {
while (head) {
Node* temp = head;
            head = head->next;
delete temp;
        }
    }

void enqueue(constT& value) {
Node* newNode = newNode(value);
if (!head) {
            head = tail = newNode;
        }
else {
            tail->next = newNode;
            tail = newNode;
        }
        size++;
    }

T dequeue() {
if (!head) throwruntime_error("Queue is empty");
Node* temp = head;
T value = temp->data;
        head = head->next;
delete temp;
        size--;
if (!head) tail = nullptr;
return value;
    }

bool empty() const { return size == 0; }
size_t getSize() const { return size; }
```

```
classIterator {
private:
Node * current;
public:
        Iterator(Node* node) : current(node) {}
T& operator*() { return current->data; }
Iterator& operator++() { current = current->next; return *this; }
booloperator != (constIterator & other) const { return current != other.current; }
    };

Iterator begin() { returnIterator(head); }
Iterator end() { returnIterator(nullptr); }
};

// 2. Очередь через два стека
template<typenameT>
classStackQueue{
private:
stack<T> inStack;
stack<T> outStack;
size_t size;

void transfer() {
while (!inStack.empty()) {
            outStack.push(inStack.top());
            inStack.pop();
        }
    }

public:
    StackQueue() : size(0) {}

void enqueue(constT& value) {
        inStack.push(value);
        size++;
    }

T dequeue() {
if (outStack.empty()) transfer();
if (outStack.empty()) throwruntime_error("Queue is empty");
T value = outStack.top();
        outStack.pop();
        size--;
return value;
    }

bool empty() const { return size == 0; }
size_t getSize() const { return size; }
};

// Тест 1: 1000 чисел
template<typenameQueueType>
void testNumbers() {
    QueueType q;
    random_device rd;
    mt19937 gen(rd());
    uniform_int_distribution<> dis(-1000, 1000);

    int sum = 0, minVal = 1000, maxVal = -1000;
    for (int i = 0; i < 1000; i++) {
        int val = dis(gen);
        q.enqueue(val);
        sum += val;
        minVal = min(minVal, val);
        maxVal = max(maxVal, val);
    }

    double avg = static_cast<double>(sum) / 1000;
```

```
    cout << "Sum: " << sum << ", Average: " << avg
        << ", Min: " << minVal << ", Max: " << maxVal << endl;
}

// Тест 2: 10 строк
template<typenameQueueType>
void testStrings() {
    QueueType q;
    vector<string> strings = { "one", "two", "three", "four", "five",
    "six", "seven", "eight", "nine", "ten" };

    for (constauto& s : strings) q.enqueue(s);
    cout << "Dequeued strings: ";
    while (!q.empty()) cout << q.dequeue() << " ";
    cout << endl;
}

// Тест 3: Люди с фильтрацией по возрасту
template<typenameQueueType>
void testPeople() {
    QueueType q;
    vector<string> surnames = { "Ivanov", "Petrov", "Sidorov" };
    vector<string> names = { "Ivan", "Petr", "Alexey" };
    vector<string> patronymics = { "Ivanovich", "Petrovich", "Alexeevich" };

    random device rd;
    mt19937 gen(rd());
    uniform_int_distribution<> yearDist(1980, 2020);

    for (int i = 0; i < 100; i++) {
        Person p;
        p.surname = surnames[gen() % 3];
        p.name = names[gen() % 3];
        p.patronymic = patronymics[gen() % 3];
        p.birthYear = yearDist(gen);
        q.enqueue(p);
    }

    QueueType young, old;
    while (!q.empty()) {
        Person p = q.dequeue();
        int age = 2025 - p.birthYear;
        if (age < 20) young.enqueue(p);
        elseif(age > 30) old.enqueue(p);
    }

    cout << "People < 20: " << young.getSize()
        << ", People > 30: " << old.getSize() << endl;
}

// Тест 4: Инверсия (исправленный)
template<typenameQueueType>
void testInversion() {
    QueueType q;
    for (int i = 1; i <= 10; i++) q.enqueue(i);
    stack<int> s;
    while (!q.empty()) s.push(q.dequeue());
    while (!s.empty()) q.enqueue(s.top()), s.pop();

    cout << "Inverted: ";
    while (!q.empty()) cout << q.dequeue() << " ";
    cout << endl;
}

// Тест 5: Сравнение производительности
void testPerformance() {
    ListQueue<int> lq;
    StackQueue<int> sq;
```

```
    auto start = chrono::high_resolution_clock::now();
    for (int i = 0; i < 10000; i++) lq.enqueue(i);
    while (!lq.empty()) lq.dequeue();
    auto end = chrono::high_resolution_clock::now();
    auto listTime = chrono::duration_cast<chrono::microseconds>(end - start);

    start = chrono::high_resolution_clock::now();
    for (int i = 0; i < 10000; i++) sq.enqueue(i);
    while (!sq.empty()) sq.dequeue();
    end = chrono::high_resolution_clock::now();
    auto stackTime = chrono::duration_cast<chrono::microseconds>(end - start);

    cout << "List Queue time: " << listTime.count() << "us" << endl;
    cout << "Stack Queue time: " << stackTime.count() << "us" << endl;
}

int main() {
    cout << "List Queue Tests:" << endl;
    testNumbers<ListQueue<int>>();
    testStrings<ListQueue<string>>();
    testPeople<ListQueue<Person>>();
    testInversion<ListQueue<int>>();

    cout << "\nStack Queue Tests:" << endl;
    testNumbers<StackQueue<int>>();
    testStrings<StackQueue<string>>();
    testPeople<StackQueue<Person>>();
    testInversion<StackQueue<int>>();

    cout << "\nPerformance Comparison:" << endl;
    testPerformance();

    return 0;
}
```

1. ListQueue: Очередь через односвязный список

Структура и специфические элементы

- Класс Node: Внутренняя структура, представляющая узел списка:
  - T data — данные произвольного типа.
  - Node* next — указатель на следующий узел.
  - Конструктор: Node(const T& value) инициализирует данные и обнуляет next.
- Переменные:
  - Node* head — указатель на начало очереди.
  - Node* tail — указатель на конец очереди.
  - size_t size — счетчик элементов.
- Методы:
  - enqueue(const T& value): Добавляет элемент в конец:
    - Создает новый узел через new Node(value).
    - Если очередь пуста (head == nullptr), устанавливает head и tail на новый узел.
    - Иначе присоединяет узел к tail->next и обновляет tail.
    - Увеличивает size.
  - dequeue(): Извлекает элемент с начала:
    - Проверяет пустоту, бросает исключение при head == nullptr.

        - Сохраняет данные первого узла, обновляет head, освобождает память (delete).
        - Уменьшает size, обнуляет tail, если очередь опустела.
    - empty(): Возвращает size == 0.
    - getSize(): Возвращает size.
- Итератор:
    - Вложенный класс Iterator:
        - Node* current — указатель на текущий узел.
        - Оператор * возвращает данные узла.
        - Оператор ++ перемещает current на следующий узел.
        - Оператор != сравнивает текущие позиции.
    - Методы begin() и end() возвращают итераторы на начало (head) и конец (nullptr).
- Деструктор: Освобождает память всех узлов через цикл с delete.

2. StackQueue: Очередь через два стека

Структура и специфические элементы

- Переменные:
    - stack<T> inStack — стек для добавления элементов.
    - stack<T> outStack — стек для извлечения элементов.
    - size_t size — общее количество элементов.
- Методы:
    - enqueue(const T& value): Добавляет элемент в inStack через push, увеличивает size.
    - dequeue(): Извлекает элемент с начала:
        - Если outStack пуст, вызывает transfer().
        - Проверяет пустоту, бросает исключение, если оба стека пусты.
        - Извлекает верхний элемент outStack через pop, уменьшает size.
    - transfer(): Переносит элементы из inStack в outStack:
        - Пока inStack не пуст, извлекает элементы (pop) и добавляет в outStack (push).
    - empty(): Возвращает size == 0.
    - getSize(): Возвращает size.
- Отсутствие итератора: Не требуется по условию для обеих реализаций.

Результат работы программы

вывод в консоль

```
List Queue Tests:
Sum: 19789, Average: 19.789, Min: -998, Max: 999
Dequeued strings: one two three four five six seven eight nine ten
People < 20: 32, People > 30: 42, Sum of filtered people: 74
People 20-30 (expected): 26, People 20-30 (actual): 26
Inverted: 10 9 8 7 6 5 4 3 2 1

Stack Queue Tests:
Sum: -7836, Average: -7.836, Min: -1000, Max: 996
Dequeued strings: one two three four five six seven eight nine ten
People < 20: 34, People > 30: 36, Sum of filtered people: 70
People 20-30 (expected): 30, People 20-30 (actual): 30
Inverted: 10 9 8 7 6 5 4 3 2 1

Performance Comparison:
List Queue time: 1028us
Stack Queue time: 257us
```

1. Тест с числами (1000 случайных чисел)

- ListQueue:
  - Сумма: 19789
  - Среднее: 19.789
  - Минимум: -998
  - Максимум: 999
- StackQueue:
  - Сумма: -7836
  - Среднее: -7.836
  - Минимум: -1000
  - Максимум: 996
- Сумма и среднее для ListQueue положительные, а для StackQueue отрицательные, что допустимо, так как числа генерируются случайно в диапазоне [-1000, 1000]. Разница между ListQueue и StackQueue обусловлена независимой генерацией случайных чисел для каждого теста.
- Минимум и максимум близки к границам диапазона, что подтверждает корректность работы генератора случайных чисел и обеих реализаций.
- Обе очереди правильно обрабатывают вставку и извлечение чисел, сохраняя порядок FIFO.

2. Тест со строками (10 строк)

- ListQueue: "one two three four five six seven eight nine ten"
- StackQueue: "one two three four five six seven eight nine ten"
- Результат идентичен для обеих реализаций. Это подтверждает, что операции enqueue (вставка в конец) и dequeue (извлечение с начала) работают корректно, сохраняя порядок FIFO.
- Тест демонстрирует стабильность работы с шаблонными типами (string).

3. Тест с людьми (100 человек, фильтрация по возрасту)

- ListQueue:
  - Люди < 20 лет: 32
  - Люди > 30 лет: 42
  - Сумма отфильтрованных людей: 74
  - Люди 20–30 (ожидаемое): 26
  - Люди 20–30 (реальное): 26
- StackQueue:

    - Люди < 20 лет: 34
    - Люди > 30 лет: 36
    - Сумма отфильтрованных людей: 70
    - Люди 20–30 (ожидаемое): 30
    - Люди 20–30 (реальное): 30
- Сумма отфильтрованных людей:
    - ListQueue: 32 + 42 = 74 (26 человек от 20 до 30 лет исключены из фильтрации <20 и >30).
    - StackQueue: 34 + 36 = 70 (30 человек от 20 до 30 лет исключены из фильтрации <20 и >30).
- Проверка 20–30 лет:
    - ListQueue: Ожидаемое количество (100 - 74 = 26) совпадает с реальным (26), что подтверждает корректность фильтрации.
    - StackQueue: Ожидаемое количество (100 - 70 = 30) совпадает с реальным (30), что также подтверждает корректность.
- Различия в числах между ListQueue и StackQueue обусловлены случайной генерацией годов рождения в диапазоне 1980–2020, что является ожидаемым поведением.
- Обе очереди корректно работают с пользовательскими структурами (Person) и выполняют фильтрацию по возрасту (2025 - год рождения).

4. Тест инверсии

- ListQueue: "10 9 8 7 6 5 4 3 2 1"
- StackQueue: "10 9 8 7 6 5 4 3 2 1"
- Инверсия выполнена с использованием только операций вставки (enqueue, push) и изъятия (dequeue, pop), без применения итераторов.
- Обе реализации успешно прошли тест, что подтверждает их способность работать с алгоритмом разворота через вспомогательный стек.

5. Сравнение производительности

- ListQueue: 1028 мкс
- StackQueue: 257 мкс
- StackQueue быстрее ListQueue примерно на 75% (257 мкс против 1028 мкс).
- Причины разницы:
    - ListQueue: динамическое выделение памяти для каждого узла (new Node) и управление указателями увеличивают накладные расходы.
    - StackQueue: использует std::stack (обычно основанный на std::deque), что обеспечивает более эффективное управление памятью и операции доступа.

ВЫВОД

В ходе тестирования на корректность и производительность были исследованы две реализации

очередей: ListQueue (на основе связного списка) и StackQueue (на основе стека). Обе очереди продемонстрировали корректную работу в качестве FIFO (первым пришел — первым ушел), успешно обрабатывая числовые, строковые данные и более сложные структуры, такие как информация о людях с фильтрацией по возрасту. Они сохранили порядок элементов и корректно выполнили операцию инверсии. При этом реализация StackQueue (на основе стека) показала значительно более высокую производительность, оказавшись на 75% быстрее, что обусловлено меньшими накладными расходами на управление динамической памятью, характерными для ListQueue (на основе связного списка). Различия в результатах тестов с использованием случайных чисел и данных о людях объясняются случайным характером этих входных данных.