ITIVITI

LET IT TRADE

Алгоритмы и структуры данных

Косяков Михаил Сергеевич
к.т.н., доцент кафедры ВТ

УНИВЕРСИТЕТ ИТМО

Содержание курса

- Введение в теорию алгоритмов
- Алгоритмы сортировок
- **Структуры данных**
  - **Линейные структуры**
  - Бинарные деревья поиска
  - Хеши и хеш-функции
- Алгоритмы на графах
  - Обходы графов в ширину и глубину
  - Минимальные остовные деревья
  - Поиск кратчайших путей в графе

Структуры данных

ITIVITI

- Определенный вид организации и хранения данных, который может быть использован для **эффективного** решения конкретной задачи
- Рассмотрим пример зоопарка:
  - Слон, лошадь, попугай, медведь, верблюд, собака и т.д.
  - Есть ли в зоопарке медведь?
  - Как нам лучше организовать данные для ответа на этот вопрос?
- Примеры: массивы, списки, пирамиды, деревья поиска, хеш-таблицы, блум-фильтры, матрицы смежности и др.

LET IT TRADE

УНИВЕРСИТЕТ ИТМО

Типы данных

ITIVITI

- Каждый тип данных определяет:
  - множество допустимых значений данных этого типа
  - набор допустимых операций над данными этого типа
- Примитивные типы: символьный, целый, вещественный, логический, void
- Пользовательские типы: структуры, классы
  - операции над пользовательскими типами реализуются программистом через набор функций (методов) класса
  - интерфейс - объявление функций (методов), через которые осуществляется работа с данными

LET IT TRADE

УНИВЕРСИТЕТ ИТМО

Абстрактные типы данных

ITIVITI

- Абстрактный тип данных (АТД, abstract data type) – это тип данных, который задан только описанием своего интерфейса
- Цель: скрыть способ хранения данных и алгоритмы работы с ними внутри функций интерфейса
- Если детально описать реализацию функций АТД, то получим конкретную структуру данных
- АТД удобно использовать при описании алгоритмов, не вдаваясь в подробности реализации операций АТД

LET IT TRADE

АТД приоритетная очередь

- Поддерживаемые операции:
  - `Insert(S, x)`
  - `Maximum(S) / Minimum(S)`
  - `Extract_Max(S) / Extract_Min(S)`
  - `Increase_Key(S, x, key) / Decrease_Key(S, x, key)`
- Реализация на (не)отсортированном массиве
- Реализация на бинарной пирамиде
- Каждая реализация поддерживает один и тот же набор операций:
  - Результат работы разных реализаций совпадает
  - Но эффективность м.б. разная!!!

Линейные типы данных

- С прямым доступом: массив (array)
- С последовательным доступом:
- Список (list)
  - `Insert(), Delete(), Prev(), Next(), Lookup()`
- Стек (stack)
  - `Push(), Pop(), Top()`
- Очередь (queue)
  - `Enqueue(), Dequeue(), Front()`
- Дек (deque)
  - `PushBack(), PopBack(), PushFront(), PopFront(), Back(), Front()`

Нелинейные типы данных

ITIVITI

- Множество (set): совокупность элементов одного типа
- Ассоциативный массив (map): мн-во пар <ключ, значение>
- Упорядоченное множество (ordered set / map):
  - `Insert()`, `Delete()`, `Lookup()`, `Min()`, `Max()`, `Successor()`, `Predecessor()`, т.к. задано отношение порядка
- Неупорядоченное множество (unordered set / map):
  - `Insert()`, `Delete()`, `Lookup()`
- Приоритетная очередь (priority queue)
  - `Insert()`, `Maximum()`, `Extract_Max()`, `Increase_Key()`
- Граф (graph)
  - `InsertEdge()`, `DeleteEdge()`, `InsertVertex()`, `DeleteVertex()`, `Edges()`, `Vertices()`

Нелинейные типы данных

ITIVITI

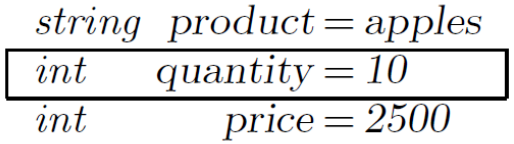
- Основные структуры данных для реализации нелинейных типов:
- Упорядоченное множество (ordered set / map):
  - бинарное дерево поиска, красно-черное дерево, АВЛ-дерево, B-дерево, префиксное дерево (если ключ - строка)
- Неупорядоченное множество (unordered set / map):
  - хеш-таблица
- Приоритетная очередь (priority queue)
  - бинарная пирамида, пирамида Фибоначчи
- Граф (graph)
  - матрица смежности, списки смежности

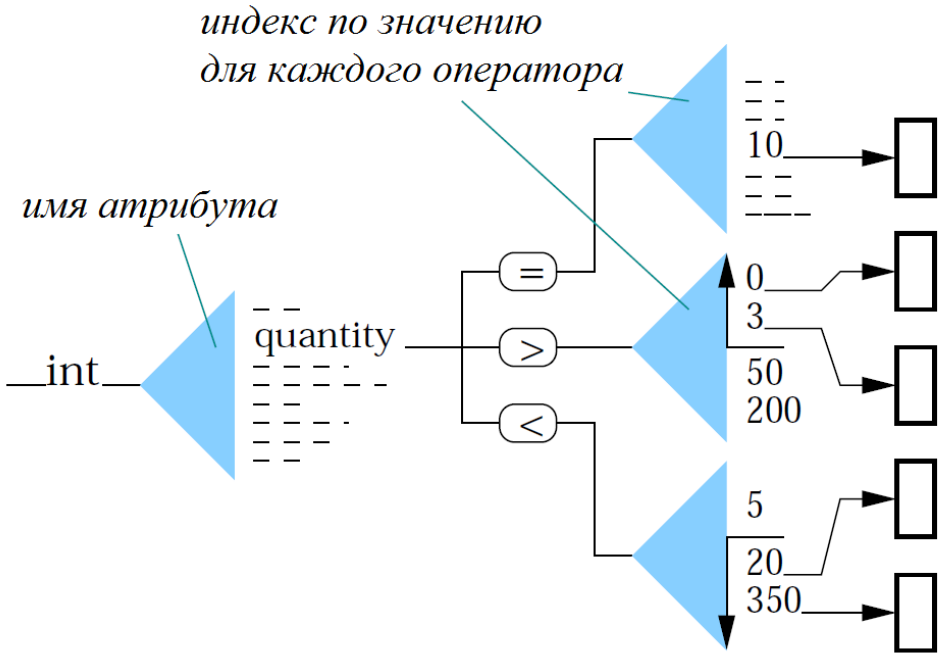
LET IT TRADE

УНИВЕРСИТЕТ ИТМО

Выбор АТД
Организация подписок

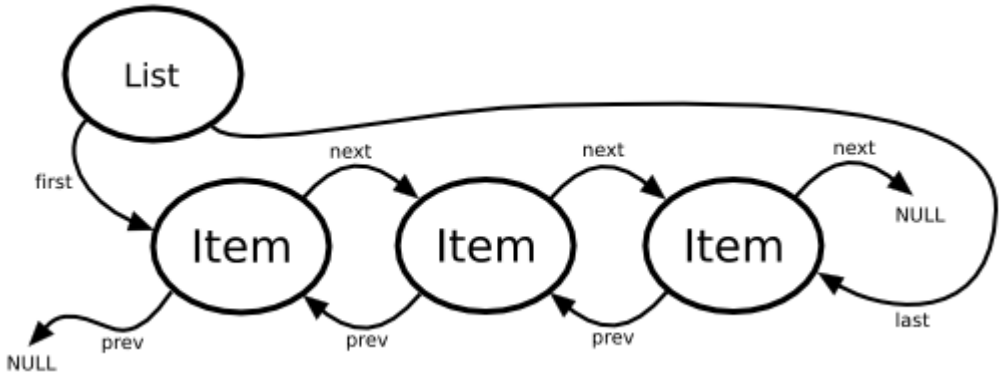
- [*product* = *apples*, *quantity* = 10, *price* = 3000 RUR]
- [*product* = *apples*, *quantity* < 20, *price* > 2500 RUR]
- [*product* = *oranges*, *quantity* > 50, *price* < 1000 RUR]
- …





Список

- Хранит последовательность объектов
- Последовательный доступ
- Эффективная вставка / удаление в середину списка
- Одно- и двусвязный список:



Список

```
1.  list<string> names;
2.
3.  names.push_back("Misha");
4.  names.push_back("Masha");
5.  names.push_back("Vasya");
6.  // обратиться напрямую names[2] нельзя, используем итератор
7.  list<string>::iterator pos;
8.
9.  // Итератор на третьей позиции
10. // Поставим Аню на позицию, куда смотрит итератор
11. pos = names.begin(); pos++; pos++;
12. names.insert(pos, "Anya");
13.
14. // Удалим Машу со второй позиции (куда смотрит итератор)
15. pos = names.begin(); pos++;
16. names.erase(pos);
17. // Misha, Anya, Vasya
```

Список / реализация

```
1. class Node {
2. public:
3.     Node(const string& s);
4. private:
5.     string m_data;
6.     Node* m_prev;
7.     Node* m_next;
8. friend class List;
9. friend class Iterator;
10.};
```

```
1. class List {
2. public:
3.     List()
4.     { m_first = NULL;
5.       m_last = NULL; }
6.     void push_back(const string&
7.                                 data);
8.     void insert(Iterator pos,
9.                 const string& s);
10.    Iterator erase(Iterator pos);
11.    Iterator begin();
12.    Iterator end();
13. private:
14.    Node* m_first;
15.    Node* m_last;
16.};
```

Список / реализация

```
1. void List::push_back(const string& data) {
2.     Node* new_node = new Node(data);
3.     if (m_last == NULL) { // Список пуст
4.         m_first = new_node;
5.         m_last = new_node;
6.     }
7.     else {
8.         new_node->m_prev = m_last;
9.         m_last->m_next = new_node;
10.        m_last = new_node;
11.     }
12.}
```

Список / реализация

```
1. // Вставляем перед iter.position
2. void List::insert(Iterator iter, const string& s) {
3.     if (iter.position == NULL) { // Вставляем в конец списка
4.         push_back(s);
5.         return;
6.     }
7.     Node* after = iter.position;
8.     Node* before = after->m_prev;
9.     Node* new_node = new Node(s);
10.    new_node->m_prev = before;
11.    new_node->m_next = after;
12.    after->m_prev = new_node;
13.    if (before == NULL) // Вставляем в начало списка
14.        m_first = new_node;
15.    else
16.        before->m_next = new_node;
17.}
```

Список / реализация

ITIVITI

LET IT TRADE

```
1. List() {
2.     Node* dummy_head = new Node("DUMMY_HEAD");
3.     m_first = dummy_head;
4.     m_last = dummy_head; // dummy_tail ?
5. }
```

```
1. void List::push_back(const string& data) {
2.     Node* new_node = new Node(data);
3.     new_node->m_prev = m_last;
4.     m_last->m_next = new_node;
5.     m_last = new_node;
6. }
```


УНИВЕРСИТЕТ ИТМО

Список / реализация

ITIVITI

LET IT TRADE

```
1. class List {
2. ...
3. private:
4.   string m_nodes[100];
5.   size_t m_prev[100];
6.   size_t m_next[100];
7.   size_t m_first;
8.   size_t m_last;
9.   size_t m_free;
10.};
```

```
1. List() {
2.   m_nodes[0] = "DUMMY_HEAD";
3.   m_first = 0; m_last = 0; m_free = 1;
4. }
```


УНИВЕРСИТЕТ ИТМО

Список / реализация

ITIVITI

```
1. void List::push_back(const string& data) {
2.   m_nodes[m_free] = data;
3.   m_prev[m_free] = m_last;
4.   m_next[m_last] = m_free;
5.   m_last = m_free;
6.   m_free++;
7.}
```

- Что в такой реализации не очень хорошо?

LET IT TRADE

УНИВЕРСИТЕТ ИТМО

Список / реализация

```
1. void List::push_back(const string& data) {
2.   m_nodes[m_free] = data;
3.   m_prev[m_free] = m_last;
4.   m_next[m_last] = m_free;
5.   m_last = m_free;
6.   m_free++;
7.}
```

- Как нам учитывать освободившиеся ячейки?
- Можно поддержать `list<size_t> m_freelist;`
  - `m_free = m_freelist.back(); m_freelist.pop_back();`
  - `m_freelist.push_back(m_free);`
- Аналогично для динамически выделяемой памяти

Мультисписок / реализация

ITIVITI

LET IT TRADE

```
1. class List {
2. ...
3. private:
4.   static string s_nodes[100];
5.   static size_t s_prev[100];
6.   static size_t s_next[100];
7.   size_t m_first;
8.   size_t m_last;
9.   static list<size_t> s_free;
10.};
```

```
1. List() {
2.   s_nodes[s_free.back()] = "DUMMY_HEAD";
3.   m_first = s_free.back(); m_last = s_free.back();
4.   s_free.pop_back();
5. }
```


УНИВЕРСИТЕТ ИТМО

Список vs std::vector

ITIVITI

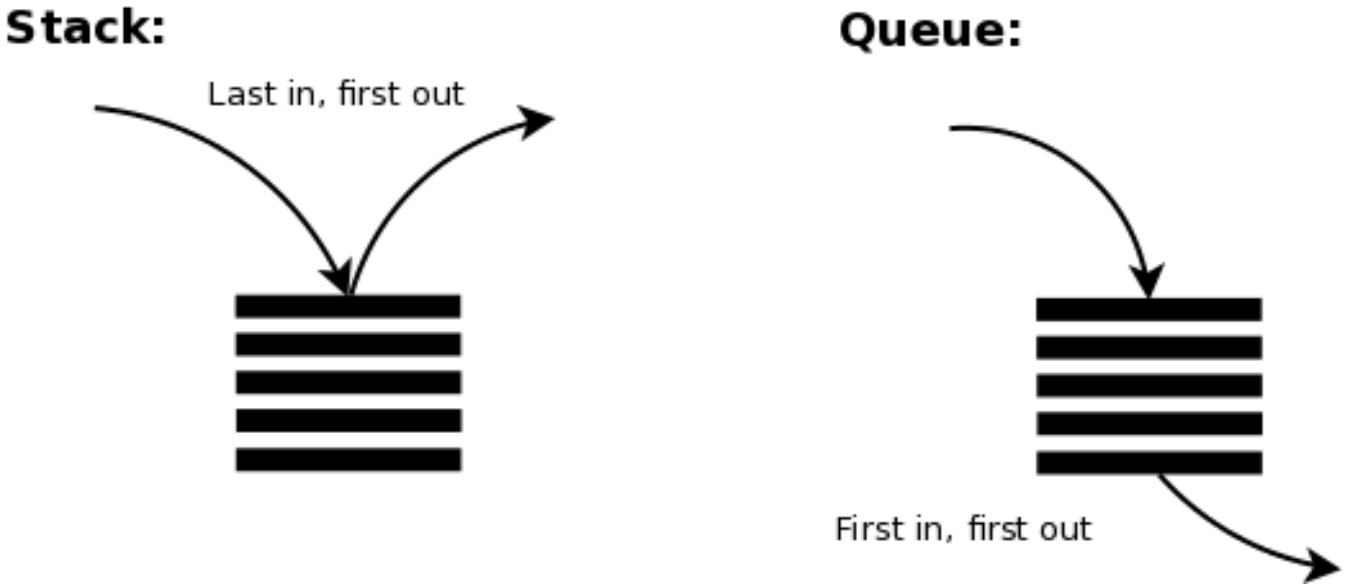
LET IT TRADE

- Получение k-го элемента: $O(k)$ vs $O(1)$
- Вставка / удаление в заданную позицию (через итератор или по индексу): $O(1)$ vs $O(N)$
- Вставка / удаление в конец: $O(1)$ vs $O(1)+$
- Последовательно вставим в конец 1280 элементов:
  - $T = 1280\,T_1 + 10\,T_2 + 20\,T_2 + 40\,T_2 + \ldots + 1280\,T_2$ $< 1280 \cdot T_1 + 1280 \cdot 2\,T_2 = 1280\,(T_1 + 2\,T_2)$
  - реаллокация std::vector инвалидирует все итераторы / указатели на элементы

УНИВЕРСИТЕТ ИТМО

Стек и очередь

- Поддерживают только добавление / удаление элементов в множество
- Не позволяют вставлять / удалять из середины



Стек и очередь

```
1.  stack<int>
2.  stack<int, vector <int> >
3.  stack<int, list <int> >
4.
5.  stack<string> s;
6.  s.push("Misha");
7.  s.push("Masha");
8.  s.push("Petya");
9.  while (s.size() > 0) {
10. cout << s.top() << "\n";
11. s.pop();
12. }
13. // Petya, Masha, Misha
```

```
1.  queue<int>
2.  //? queue<int, vector<int> >
3.  queue<int, list <int> >
4.
5.  queue<string> q;
6.  q.push("Misha");
7.  q.push("Masha");
8.  q.push("Petya");
9.  while (q.size() > 0) {
10. cout << q.front() << "\n";
11. q.pop();
12. }
13. // Misha, Masha, Petya
```

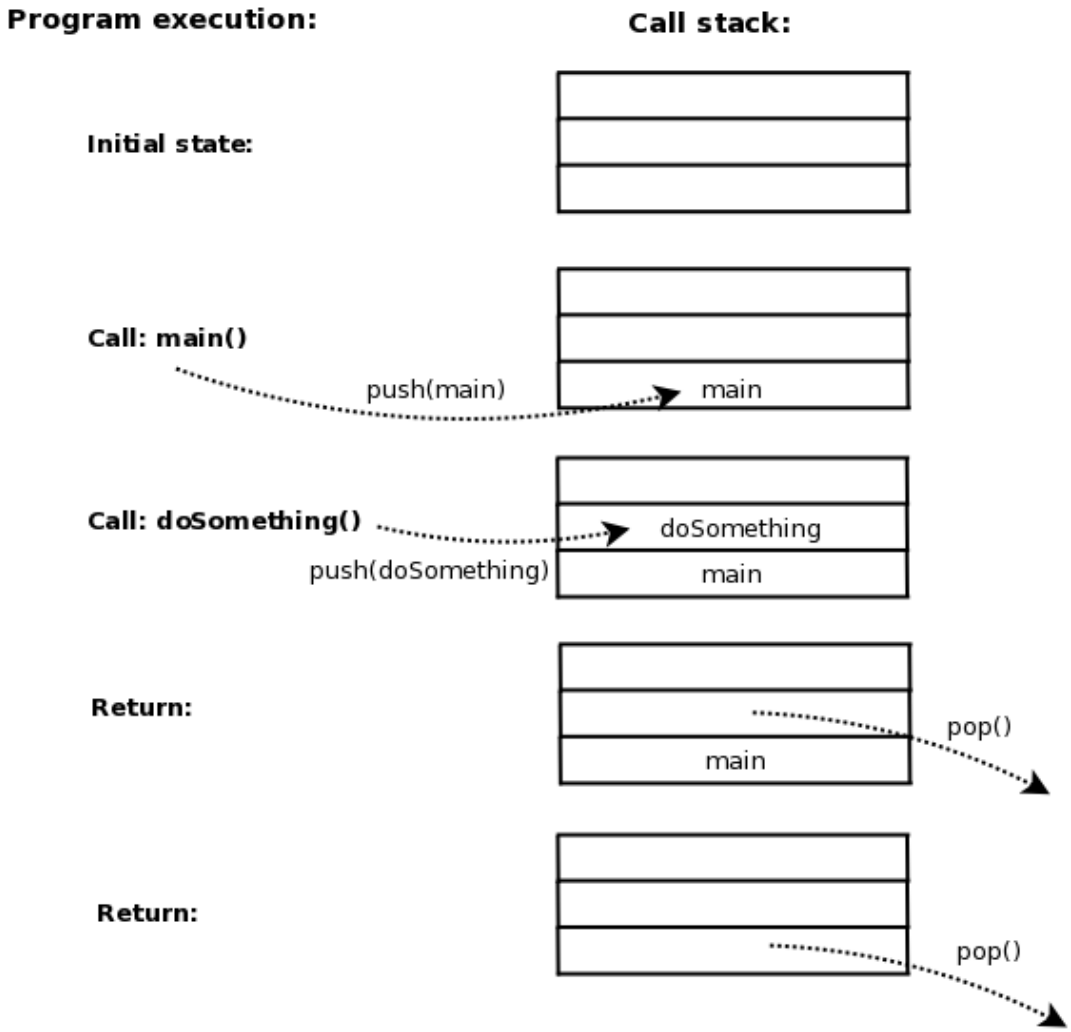
Стек и очередь

ITIVITI

- Удобно хранить набор задач, ожидающих выполнения
- Стек: откладываем выполнение текущей задачи, переключаемся на новую, после окончания возвращаемся к отложенной задаче
- Очередь: организуем последовательное выполнение задач согласно порядку их поступления
- Обобщенная очередь: выбираем задачи на выполнение согласно дисциплине обслуживания очереди (например, по приоритетам)
  - Теория массового обслуживания (queueing systems theory)
  - Система массового обслуживания (queueing system)

LET IT TRADE

Стек



Стек / быстрая сортировка

ITIVITI

LET IT TRADE

```
1. void quicksort(int * a, int l, int r)
2. {
3.     if (r <= l) return;
4.     int p = partition(a, l, r);
5.     quicksort(a, l, p-1);
6.     quicksort(a, p+1, r);
7. }
```

Стек / быстрая сортировка

ITIVITI

LET IT TRADE

```
1. void quicksort(int * a, int l, int r)
2. {
3.     if (r <= l) return;
4.     int p = partition(a, l, r);
5.     quicksort(a, l, p-1);
6.     quicksort(a, p+1, r);
7. }
```

УНИВЕРСИТЕТ ИТМО

Стек / быстрая сортировка

```
1. void quicksort(int * a, int l, int r)
2. {
3.     std::stack< std::pair<int, int> > s;
4.     s.push({l, r});
5.     while (!s.empty())
6.     {
7.         l = s.top().first;
8.         r = s.top().second;
9.         s.pop();
10.        if (r <= l) continue;
11.        int p = partition(a, l, r);
12.        if (p-l > r-p)
13.            { s.push({l, p-1}); s.push({p+1, r}); }
14.        else
15.            { s.push({p+1, r}); s.push({l, p-1}); }
16.    }
17.}
```

Стек / обратная польская нотация

ITIVITI

- Что неудобно в этой записи?
  - 3 + (4 + 5) * 6 + (7 + 8) * 9
- Читаем слева-направо без скобок:
  - (4 + 5) * 6 => 4 5 + 6 *
  - 1 + 2 * 3 => 1 2 3 * +
  - 1 + 2 + 3 => 1 2 + 3 +
  - 3 4 5 + 6 * + 7 8 + 9 * +
- Каждый операнд кладем в стек
- Каждая операция достает соответствующее число операндов из стека, вычисляет значение, и кладет его обратно в стек

LET IT TRADE

Стек / обратная польская нотация

```
1. stack<int> values;
2. string input;
3. while (cin >> input) {
4.     if (input == "+" || input ==  "*" || ...
5.     {
6.         int second = values.top(); values.pop();
7.         int first = values.top(); values.pop();
8.         if (input == "+")
9.             values.push(first + second);
10.        else if (input == "*")
11.            values.push(first * second);
12.        ...
13.    }
14.    else
15.        values.push(atoi(input.c_str()));
16.}
```

Стек / реализация

```
1. class Stack {
2. public:
3.     Stack() { m_top = 0; }
4.     void push(const string& val) { m_stack[m_top++] = val; }
5.     string pop() { return m_stack[--m_top]; }
6.     string top() { return m_stack[m_top-1]; }
7.     size_t size() { return m_top; }
8. private:
9.     string m_stack[100];
10.    size_t m_top;
11.};
```

Мультистек / реализация

ITIVITI

LET IT TRADE

```
1. class Stack {
2. public:
3.     Stack() { m_top = 0; }
4.     ...
5.     void push(const string& val) { s_next[s_free] = m_top; \
6.             m_top = s_free; s_free++; s_stack[m_top] = val; }
7.     ...
8.     bool empty() { return m_top == 0; }
9.
10.private:
11.    size_t m_top;
12.    static string s_stack[100];
13.    static size_t s_next[100];
14.    static size_t s_free;  // static list<size_t> s_free;
15.};
16.size_t Stack::s_free = 1;
```


УНИВЕРСИТЕТ ИТМО

Мультистек / реализация

```
1. class Stack {
2. public:
3.     Stack() { }
4.     ...
5.     void push(const string& val) { m_mlist.push_back(val); }
6.     void pop() { return m_mlist.pop_back(); }
7.     ...
8.     string top() { return m_mlist.back(); }
9.     size_t size() { return m_mlist.size(); }
10.private:
11.    List m_mlist;
12.};
```

Очередь / реализация

ITIVITI

LET IT TRADE

```
1. class Queue {
2. public:
3.     Queue() { m_head = 100; m_tail = 0; }
4.     bool empty() const { return m_head % 100 == m_tail; }
5.     void enqueue(const string& val) {
6.         m_queue[m_tail++] = val; m_tail = m_tail % 100;
7.     }
8.     string dequeue() {
9.         m_head = m_head % 100; return m_queue[m_head++];
10.    }
11.    string front() { return m_queue[m_head % 100]; }
12.    size_t size() { return (m_tail+100 - m_head) % 100; }
13.private:
14.    size_t m_head;
15.    size_t m_tail;
16.    string m_queue[100];
17.};
```


УНИВЕРСИТЕТ ИТМО

ITIVITI

LET IT TRADE

Спасибо за внимание!

УНИВЕРСИТЕТ ИТМО