Лекция 1. Структуры данных. Стек. Очередь. Дек

Контейнерные классы. Стандартная библиотека шаблонов STL. Последовательные и ассоциативные контейнеры. Итераторы. Последовательные контейнеры: векторы, двусторонние очереди, списки, очереди, стеки, очереди с приоритетами. Примеры задач.

Общее представление

Перед написанием программы важно правильно выбрать структуру данных, обеспечивающую эффективное решение задачи. Одни и те же данные можно сохранить в структурах, требующих различного объема памяти, а алгоритмы работы с каждой структурой могут иметь различную эффективность. Если выбрана наиболее подходящая структура и вы в совершенстве владеете методами работы с ней, то разработка алгоритма уже не вызовет затруднений, а сам алгоритм решения будет оптимален и по объему занимаемой памяти, и по времени работы алгоритма. Структуры данных определяют *объекты*, организованные определенным образом и *операции*, которые можно выполнять над объектами. Доступ к объектам и все операции с ними осуществляются только через интерфейс. Интерфейс отделяет реализацию структуры данных от клиента и является «непрозрачным» для клиента. Структура данных может быть представлена как некий «черный ящик» с интерфейсами.

Контейнерные классы предназначены для хранения данных, организованных определенным образом. Для каждого типа контейнера определены методы работы с его элементами, не зависящие от конкретного типа данных, которые хранятся в контейнере. Поэтому один и тот же вид контейнера можно использовать для хранения данных различных типов. Возможность работы с контейнерными классами реализована с помощью стандартной библиотеки шаблонов (STL Standard Template Library), в которую входят контейнерные классы, алгоритмы и итераторы. Использование STL позволяет повысить надежность программ и уменьшить сроки их реализации.

Контейнеры можно разделить на два типа: *последовательные* и *ассоциативные*. *Последовательные контейнеры* обеспечивают хранение конечного количества однотипных элементов в виде непрерывной последовательности. К последовательным контейнерам относятся: векторы (*vector*), двусторонние очереди (*deque*), списки (*list*), а также так называемые адаптеры (варианты контейнеров): стеки (*stack*), очереди (*queue*) и очереди с приоритетами (*priority_queue*). Каждый вид контейнера обеспечивает свой набор действий над данными, и выбор того или иного контейнера зависит от того, что именно требуется делать с данными в программе. Ассоциативные контейнеры обеспечивают быстрый доступ к данным по ключу. Такие контейнеры построены на основе сбалансированных деревьев. К ассоциативным контейнерам относятся: словари (map), словари с дубликатами (multimap), множества (set), множества с дубликатами (multiset) и битовые множества (bitset).

Каждая структура предоставляет алгоритмы, работающие с данными с той или иной сложностью. **Сложность** понимается как количество элементарных операций, выполняемых вычислительной машиной для решения задачи в зависимости от размера задачи. Размер задачи определяется входными данными. Так, например, для простейшей операции сложения двух натуральных чисел, размером задачи можно считать максимальную длину в битах одного из слагаемых. Для оценки сложности алгоритмов применяется верхняя оценка сложности, выражаемая в O – символике. Алгоритм имеет сложность $O(n)$, если время его работы (количество элементарных операций) есть величина <= cn (меньшая или равная cn), где c – некоторая постоянная. Например, поиск элементов в одномерном массиве имеет сложность $O(n)$ – линейную сложность. «Пузырьковая сортировка» массива имеет сложность $O(n^2)$ – квадратичную

сложность. $O(1)$ – сложность, представляющая собой константу. Чем меньше сложность, тем быстрее работает алгоритм.

Двусторонняя очередь (дек) – последовательный контейнер, поддерживающий доступ к произвольным элементам и обеспечивающий вставку и удаление из обоих концов очереди за постоянное время. Операции с элементами внутри очереди занимают время, пропорциональное количеству перемещаемых элементов. Доступ к элементам очереди осуществляется за постоянное время (оно несколько больше, чем для вектора).

Список – последовательный контейнер, обеспечивающий вставку и удаление элементов за постоянное время. Не предоставляет произвольный доступ к своим элементам. Операции с элементами внутри списка (вставка элемента, удаление элемента) занимают постоянное время.

Стек – последовательный контейнер, обеспечивающий вставку элемента в вершину стека и удаление элемента из вершины стека.

Очередь – последовательный контейнер, обеспечивающий добавление элементов в конец очереди и извлечение элементов с начала очереди.

Очередь с приоритетом – структура, реализованная при помощи очереди на основе контейнера, допускающего произвольный доступ к элементам (например, вектора или двусторонней очереди). Первым параметром при описании очереди с приоритетами является тип ключа, вторым последовательный контейнер, третьим – функция определения приоритета.

STL определяется в следующих заголовочных файлах: algorithm, deque, functional, iterator, list, map, memory, numeric, queue, set, stack, utility, vector.

Итераторы

Итератор является аналогом указателя на элемент и используется для просмотра контейнера в прямом и обратном направлении. Итератор ссылается на элемент и реализует операцию перехода к следующему элементу. Константные итераторы используются в том случае, когда значения соответствующих элементов контейнера не изменяются. В таблице приведены варианты итераторов и методы доступа к элементов при помощи итераторов.

Поле	**Пояснение**
iterator	Итератор
const_iterator	Константный итератор
reverse_iterator	Обратный итератор
const_reverse_iterator	Константный обратный итератор
Методы доступа	
iterator begin() const_iterator begin()	Указывает на первый элемент
iterator end() const_iterator end()	Указывает на элемент, следующий за последним
reverse iterator rbegin() const_reverse_iterator rbegin()	Указывает на первый элемент в обратной последовательности
reverse_iterator rend() const_reverse_iterator rend()	Указывает на элемент, следующий за последним в обратной последовательности

В каждом контейнере эти типы и методы определяются способом, зависящим от их реализации. Для итераторов реализованы операции ++, -- после которых итератор указывает на следующий, предыдущий элемент в контейнере в порядке обхода. Доступ к элементу осуществляется при помощи операции разыменования *.

Общие методы и алгоритмы для последовательных контейнеров

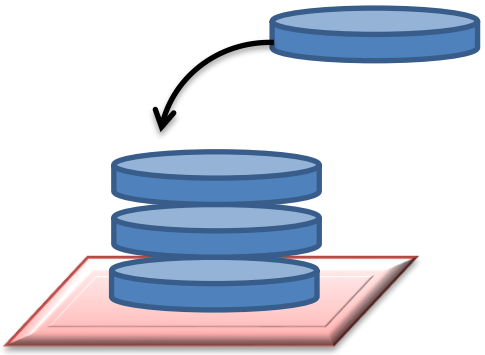
Для работы с последовательными контейнерами используется ряд общих для них методов, предназначенных для получения сведений о размере контейнера или выполнения тех или иных операций с контейнером.

Методы получения сведений о размере контейнера	
size()	Получение сведений о размере контейнера (число элементов)
max_size()	Максимальный размер контейнера
empty()	Логическая функция, показывающая, пуст ли контейнер
Методы работы с элементами контейнера	
push_front()	Вставка в начало дека
pop_front()	Удаление из начала дека
push_back()	Вставка в конец дека или вектора
pop_back()	Удаление из конца дека или вектора
push()	Вставить элемент в вершину стека или очереди
pop()	Удалить элемент из вершины стека или очереди
top()	Посмотреть элемент в вершине стека
insert()	Вставка элемента во множество или словарь
erase()	Удаление элемента из множества или словаря. Удаление элементов в диапазоне итераторов
at []	Произвольный доступ к элементу в векторе или двусторонней очереди

Алгоримы	
clear()	Очищает контейнер
count (it1, it2, value)	Считает количество вхождений элемента value в указанное множество, задаваемое двумя итераторами.
find (it1, it2, value)	Находит итератор, соответствующий элементу, равному value. Возвращает итератор на элемент или end() если элемент не найден
min_element(it1, it2) max_element(it1, it2)	Находит итератор, соответствующий минимальному (максимальному) элементу
binary_search(it1, it2, value)	Возвращает true, если элемент найден в указанном множестве и false если такого элемента нет
lower_bound(it1, it2, value) upper_bound(it1, it2, value)	Возвращает итератор, соответствующий первому/последнему элементу в множестве [it1, it2), перед которым можно вставить элемент value, сохраняя отсортированность множества.
sort(it1, it2) stable_sort(it1, it2)	Сортирует элементы Сортирует элементы, сохраняя относительный порядок элементов
replace(it1, it2,value1,value2)	Заменяет все вхождения value1 на value2 в указанном множестве, задаваемым итераторами it1, it2
reverse(it1, it2)	Меняет порядок на противоположный
merge(it11, it12, it21,it22, out_it)	Сливает два отсортированных множества, задаваемых итераторами первого и второго множества, копируя элементы в новое множество. Возвращает итератор end() нового множества a[3]={2,3,4}; b[3]={3,4,5}; merge(a,a+3,b,b+3,c); Получим c={2,3,3,4,4,5}
unique(it11, it12)	Сжимает отсортированное множество, удаляя одинаковые элементы
random_shuffle(it11, it12)	Случайным образом перемешивает элементы множества
next_permutation(it11, it12) prev_permutation(it11, it12)	Генерирует следующую (предыдущую) лексикографическую перестановку множества. Возвращает true, если такая нашлась и false в противном случае.

Для понимания логики построения той или иной структуры полезным упражнением является самостоятельная реализации структур, например стека при помощи массива.

Реализация структуры стек при помощи статического массива



Стек можно представить себе как стопку дисков (см. рисунок), на которую сверху можно класть диски, и брать их можно тоже только сверху.

Стек (*stack)* является структурой данных, поддерживающей две операции: добавление элемента в вершину стека push (value) и удаление элемента из вершины стека pop (value). Дисциплина работы стека обозначается LIFO, последним пришел — первым ушел (Last In First Out). Стек можно реализовать при помощи массива.

```
#include <iostream>
using namespace std;
const int MAX_SIZE = 10; // Количество элементов в стеке
struct my_stack // Описываем структуру стек
{
     int data[MAX_SIZE]; // Массив для хранения элементов стека
     int last;           // Указатель на элемент после вершины
};
void push(my_stack &s, int x) // Процедура добавления элемента
{
     if (s.last == MAX_SIZE) // Сообщение о переполнении стека
     {
          cout << "Stack Overflow";
          exit(-1);
     }
     s.data[s.last++] = x;
}
int pop(my_stack &s) // Получение элемента с вершины стека
{
     return s.data[--s.last];
     // Возвращаем элемент с вершины стека
}
int main()
{
     my_stack a;
     a.last = 0;
     push(a, 3);
     push(a, 6);
     push(a, 2);
     cout << pop(a) << " "; // Программа выведет 2
     cout << pop(a) << " "; // Программа выведет 6
     cout << pop(a) << " "; // Программа выведет 3
     return 0;
}
```

Как мы видим, заполняя стек методом push, мы помещаем каждый новый элемент в конец массива (Вершину стека). Указателем на вершину стека служит поле last в структуре стека. Снимая элемент с вершины стека методом pop в нашей реализации, мы возвращаем значение элемента, а затем удаляем этот элемент (уменьшая указатель last на единицу). Сложность каждой

описанной операции O(1). Таким образом, обращаясь к вершине стека после его заполнения, мы разворачиваем последовательность элементов. *Стек можно применять для разворота последовательности элементов*. Приведенную программу логично дополнить проверкой стека на наличие в нем элементов – методом empty(). Функция empty() вернет true, если стек пуст и false в противном случае.

```
bool empty(my_stack &s)
{
     return s.last == 0;
}
```

Метод size() позволяет определить количество элементов в стеке.

```
int size(my_stack &s)
{
     return s.last;
}
```

Метод top() позволяет обращаться к вершине стека – возвращать элемент, находящийся в вершине стека, но при этом не удалять его (в отличие от метода pop()).

```
int top(my_stack&s)
{
     int i=s.last-1;
     return s.data[i];
}
```

Полезным упражнением может быть также реализация алгоритмов работы со стеком, например, count или find (перечень алгоритмов работы с контейнерами приведены в таблице в начале лекции).

Задачи, решаемые при помощи стека

1. Грамматика правильной скобочной последовательности.

Правильная скобочная последовательность – последовательность, состоящая из символов – «скобок», в которой каждой отрывающей скобке соответствует закрывающая скобка такого же типа, что и открывающая скобка. Например, правильными будут следующие последовательности: [([])((([[[]]])))]{()}, ()((()))[[]]. Не будут являться правильными скобочные последовательности [[]) (несоответствие типа закрывающих скобок типу открывающих), }{ (закрывающая скобка стоит раньше открывающей), [[{{}}] (не каждой открывающей скобке соответствует закрывающая).

Решение задачи для скобочной последовательности, состоящей из одного типа скобок, заключается в вычислении баланса скобочной последовательности. Каждой открывающей скобке ставим в соответствие число +1, каждой закрывающей скобке число – 1. Считаем баланс последовательности balance, двигаясь слева направо. Если баланс в процессе подсчета станет равным отрицательному числу – balance < 0, то это означает, что не для всех закрывающих скобок в последовательности имелись открывающие и последовательность не является правильной. По достижении конца строки баланс должен быть равным нулю – balance = 0. В этом случае последовательность правильная, иначе – неправильная.

Решение задачи для скобочной последовательности, состоящей из скобок различного типа, удобно выполнить при помощи структуры стек. При движении слева направо по строке в стек заносятся открывающие скобки. При добавлении в стек закрывающей скобки проверяем наличие в вершине стека открывающей скобки такого же типа. Если таковая скобка в вершине стека есть, то очередная скобка не добавляется, а имеющаяся в вершине удаляется. Рассмотрим пример для правильной скобочной последовательности [([])(([])[])].

Последовательность	[	(	[	]	)	(	(	[	]	)	)	]
Состояние стека	[	[(	[([	[(	[	[(	[((	[(([	[((	[(	[	

В конце работы программы стек оказывается пустым – в этом случае скобочная последовательность правильная. Как видим, задача проверки скобочной последовательности на правильность имеет линейную сложность $O(n)$. Приведем пример программы, определяющей, является ли правильной скобочная последовательность. Программа использует стек STL.

```
#include <iostream>
#include <stack>
#include <string>
using namespace std;
int main()
{
	string s;
	stack<char> b;
	cin >> s;
	for (auto c : s)
	{
		if (c == ')' && b.top() == '(' ||
			c == ']' && b.top() == '[' ||
			c == '}' && b.top() == '{')
		{
			b.pop();
		}
		else if (c == ')' || c == ']' || c == '}')
		{
			cout<<"NO";
			return 0;
		}
		else
		{
			b.push(c);
		}
	}
	cout << b.empty() ? "YES" : "NO";
	return 0;
}
```

2. Вычисления арифметических выражений.

Арифметическое выражение состоит из чисел, знаков арифметических действий и скобок. Например, рассмотрим арифметическое выражение (7 + 6) / (26 - 13). Его можно записать и других формах. *Префиксная форма записи* арифметического выражения представляет выражение таким образом, что знаки арифметических операций предшествуют операндам, над которыми эти операции выполняются. А *постфиксная форма записи (обратная польская нотация)* представляет выражение таким образом, что знаки арифметических операций указываются после операндов.

Арифметическое выражение	**(7+6)/(26-13)**
Префиксная форма записи	/ **+ 7 6 – 26 13**
Постфиксная форма записи	**7 6 + 26 13 -** /

Алгоритм вычисления значения арифметического выражения, записанного в постфиксной форме, имеет линейную сложность – O(n). Алгоритм использует стек. При чтении выражения слева направо в вершину стека помещаются операнды. Как только при чтении встречается знак

арифметической операции, из стека извлекаются два последних операнда, к ним применяется текущая операция, и результат записывается обратно в вершину стека. По завершении работы алгоритма в стеке оказывается один элемент – значение арифметического выражения.

Рассмотрим алгоритм вычисления арифметического выражения, использующий в неявном виде обратную польскую нотацию. Заводим два стека. Один – для чисел, второй для знаков арифметических операций и скобок. Читаем выражение слева направо, и, встретив число, помещаем его в первый стек. Если текущий символ – закрывающаяся скобка, то выполняем вычисления до тех пор, пока не встретим парную ей открывающуюся скобку. Если текущий символ – знак арифметической операции и на вершине стека операции с таким же или большим приоритетом, то выполняем все необходимые для нее действия. По завершении алгоритма в стеке операций могут остаться еще не выполненные операции, их надо выполнить, как было описано выше. Рассмотрим некоторые идеи для написания программы, вычисляющей значение арифметического выражения при условии, что арифметическое выражение имеет правильный формат записи. Прежде всего, договоримся, что исходная строка – арифметическое выражение, текущая позиция строки, два стека **num** - для хранения чисел и **op** - знаков арифметических операций и скобок будут объявлены как глобальные переменные.

```
string s;
int cur_pos=0;
stack <int> num;
stack <char> op;
```

При проходе по строке слева направо нам требуется уметь определять – является ли текущий символ цифрой или знаком операции, скобкой. В этом помогут следующие функции.

```
bool LT_END() // Функция определяет тип лексемы - конец строки
     { return cur_pos >= s.size(); }
bool LT_N()    // Тим лексемы - цифра
     { return s[cur_pos] >= '0' && s[cur_pos] <= '9'; }
bool LT_OB(char c) // Тип лексемы - открывающаяся скобка
     { return c == '('; }
bool LT_CB(char c) // Тип лексемы - закрывающаяся скобка
     { return c == ')'; }
bool LT_OP(char c) // Тип лексемы - знак арифметической операции
     { return c == '+' || c == '-' || c == '*' || c == '/'; }
```

Как только в строке встретится символ – цифра, необходимо «собрать» все число целиком. Для этого предусмотрим функцию `get_num()`.

```
int get_num()
{
     int value = 0;
     while (!LT_END() && LT_N())
     {
          value = value * 10 + (s[cur_pos] - '0');
          // Собираем число
          cur_pos++;
     }
     cur_pos--;
     return value;
}
```

При выполнении арифметических действий необходимо учитывать приоритет арифметических операций. Более высокий приоритет имеют операции * и /.

```
int pr(char c)
```

```
{
     if (c == '+' || c == '-') return 1;
     if (c == '*' || c == '/') return 2;
}
```

Для вычисления простейших арифметических выражений вида: <число> <знак арифметической операции> <число> реализуем функцию get_res(), получающую на вход операцию, которая будет произведена с двумя числами, находящимися в вершине стека **num** с числами. После этого эти числа удалятся, а полученный результат будет добавлен в вершину стека **num**.

```
void get_res(char op)
{
     int r = num.top(); num.pop();
     int l = num.top(); num.pop();
     switch (op)
     {
     case '+': num.push(l + r); break;
     case '-': num.push(l - r); break;
     case '*': num.push(l * r); break;
     case '/': num.push(l / r); break;
     }
}
```

Наконец, функция calc() принимает на вход исходную стоку – арифметическое выражение, и, анализируя символы этой строки, делает все необходимые действия алгоритма.

```
int calc()
{
     while(!LT_END())
     {
          // Если встретили закрывающуюся скобку, то вычисляем
          // выражение между скобками, пока не встретим
          // открывающуюся скобку
          if (LT_CB(s[cur_pos]))
          {
               while (!LT_OB(op.top()))
               {
                    get_res(op.top());
                    op.pop();
               }
               op.pop();
          }
          // Открывающуюся скобку сразу заносим в стек op
          else if (LT_OB(s[cur_pos]))
               op.push(s[cur_pos]);
          // Если встретили арифметическую операцию, то пока в
          // стеке операций находится операция с большим или
          // равным ей приоритетом, выполняем арифметические
          // действия с числами стека num
          else if (LT_OP(s[cur_pos]))
          {
               char cur_op = s[cur_pos];
               int cur_pr = pr(s[cur_pos]);
```

```
                while (!op.empty() && LT_OP(op.top()) &&
                      pr(op.top()) >= cur_pr)
                {
                      get_res(op.top()); op.pop();
                }
                op.push(cur_op);
          }
          else if (LT_N())
                num.push(get_num());
          cur_pos++;
     }
     while (!op.empty())
     {
          // В конце в стеках могут остаться данные – необходимо
          // «довычислить» выражение.
          get_res(op.top()); op.pop();
     }
     return num.top();
}
```

3. Ближайший меньший слева и справа.

Дан массив чисел. Требуется вывести ближайший меньший слева и справа для данного элемента. Например, для массива a[9]={6 5 9 8 7 1 2 3 5} и числа 7 ближайшим меньшим слева будет 6 с индексом 2, а ближайший меньший справа будет 1 с индексом 6. Будем искать ближайший меньший справа. Начинаем последовательность брать элементы с конца массива и записывать их индексы в стек.

На первой итерации добавим в стек (9) – индекс последнего элемента в массиве. На второй итерации запоминаем (8) – индекс предпоследнего элемента в массиве. Обращаемся к вершине стека. Пока на вершине стека индексы элементов, которые больше, чем текущий или равны ему, пропускаем эти элементы, удаляя их из стека. Запоминаем ответ (вершину стека) и добавляем новое число (индекс просматриваемого элемента) в стек. Продолжаем алгоритм далее, пока не получим ответ для последнего числа. Таким образом, сложность алгоритма линейная – $\boldsymbol{O(n)}$.

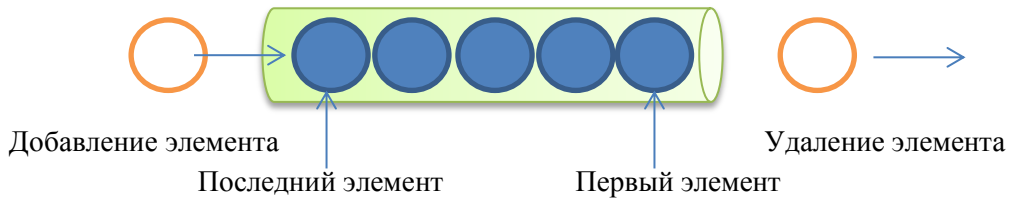
1 итерация. Стек	9	Добавляем в стек индекс последнего элемента массива. В ответ для него записываем -1.								
2 итерация. Стек	8		Берем из массива элемент 3 с индексом 8. В вершине стека индекс элемента 5, который больше, чем 3. Поэтому удаляем из стека индекс 9, и записываем в стек индекс 8. В ответ записываем – 1.							
3 итерация. Стек	7			Берем из массива элемент 2 с индексом 7. Просматриваем стек. На вершине стека индекс 8 (элемента 3. 3 больше 2), поэтому удаляем из стека 8, записываем 7. В ответ записываем – 1.						
4 итерация. Стек	6				Для элемента 1 с индексом 6 ответом также является -1					
5 итерация. Стек	6	5				Ответ для текущего элемента 7 с индексом 5 – индекс 6.				
6 итерация. Стек	6	5	4				Ответ для элемента 8 - индекс 5			
7 итерация. Стек	6	5	4	3				Ответ для элемента 9 с индексом 3, индекс 4.		
8 итерация. Стек	6	2							Ответ 6	
9 итерация. Стек	6	2	1							2

Исходный массив	6	5	9	8	7	1	2	3	5	
ans	2	6	4	5	6	-1	-1	-1	-1	

По полученному ответу легко восстановить значения ближайших минимальных элементов ко всем элементам исходного массива. Индексы ближайших минимальных справа к каждому из элементов массива будут храниться в массиве ans[9]. Аналогично решается задача поиска ближайших минимальных слева элементов.

Реализация структуры очередь при помощи статического массива

Очередь **queue** является контейнером позволяющим добавлять элементы в конец (хвост tail) очереди - push(), удалять элементы в начале (голове head) очереди - pop(). Очередь можно представить себе как очередь людей, в которую каждый новый приходящий попадает в конец очереди, а покидают очередь стоящие в ней из ее начала. Дисциплина работы очереди обозначается FIFO, первым пришел — первым уйдешь (First In First Out).



Очередь, как и стек, можно реализовать при помощи статического массива. Если ее реализовывать так, как показано на рисунке выше, то возникают следующие проблемы логики работы очереди: при удалении первого элемента необходимо будет передвигать всю очередь на место первого элемента – на это потребуется время, пропорциональное количеству элементов в очереди. Удобнее будет «закольцевать очередь» *по модулю MAXSIZE*.

```
const int MAX_SIZE = 10;
```

Определим класс queue, и опишем во внутреннем разделе класса очередь как статистический массив, имеющий указатели на первый - first элемент и последний – last элемент, которые указывают на голову и хвост очереди.

```
class queue
{
private:
     int a[MAX_SIZE]; // Статический массив для хранения
                      // элементов очереди
     int first;       // Указатель на первый элемент очереди
     int last;        // Указатель на элемент после конца
```

В открытом разделе класса инициализируем очередь, поместив указатели first и last на нулевой индекс массива.

```
public:
     queue()
     {
          last = 0;
          first = 0;
     }
```

Реализуем метод push() для добавления элементов в конец очереди. Новые элементы добавляются в позицию last по модулю MAXSIZE, после этого указатель last перемещается на

следующую позицию. Защита от ошибок предусмотрена в том случае, если количество элементов в очереди больше максимально возможного.

```
void push(int x)
{
     if (last + 1 == first)
     {
          cout << "Queue overflow";
          exit(-1);
     }
     else
          a[(last++) % MAX_SIZE] = x;
}
```

Метод pop() реализован так, что он возвращает первый элемент, а затем указатель first передвигается на следующий элемент.

```
int pop()
{
     return a[(first++) % MAX_SIZE];
}
```

Размер очереди можно определить при помощи метода size(), которая возвращает значение (last-first)%(MAX_SIZE+1). Очередь пустая, если ее размер равен нулю.

```
int size()
{
     return (last - first + MAX_SIZE + 1) % (MAX_SIZE + 1);
}
```

Для очереди предусмотрены две операции просмотра элементов – элемента в начале очереди front() и элемента в конце очереди back().

```
int front() { return a[first]; }
int back() { return a[last - 1]; }
```

В STL очередь реализована при помощи структуры queue, к которой можно применить стандартные методы, описанные выше и алгоритмы, представленные в начале лекции. Для работы с очередью STL необходимо подключить заголовок #include <queue>. Сложность операции добавления элементов в очередь и их извлечения – $O(1)$.

Задача нахождения минимального элемента на фиксированном отрезке. Реализация очереди для нахождения минимума при помощи двух стеков.

Постановка задачи. Дан массив **A**, состоящий из **n** элементов. Необходимо найти минимум на всех подотрезках массива фиксированной длины **K** за $O(n)$. Например, для массива A[10]={1, 8, 4, 3, 5, 7, 4, 5, 6, 7} при K=3 получим ответ ans={1, 3, 3, 3, 4, 4, 4, 5}.

Для начала рассмотрим идеи по реализации очереди при помощи двух стеков. Первый стек head отвечает за уход элементов из очереди и представляет собой начало очереди. Второй стек tail отвечает за приход элементов в очередь и представляет собой хвост очереди. Например, в очередь постепенно добавляются элементы: 1, 8, 4, 3, а некоторые из добавленных удаляются. Если при попытке извлечь элемент стек head оказался пустым, то переносим все элементы из tail в head (при этом элементы будут перенесены в обратном порядке, что и нужно для извлечение первого элемента из очереди).

Операции	Состояние стека tail	Состояние стека head
Добавление элемента 1 в очередь	1	Стек пустой
Добавление элемента 8 в очередь	8 1	Стек пустой

Добавление элемента 4 в очередь	4 8 1	Стек пустой
Удаление элемента из очереди. Удаляется первый элемент – 1. Так как стек head пустой – переносим все элементы из стека tail в стек head. Затем удаляем 1	Все элементы переносим в стек head	1 – удаляем элемент 1 8 4
Добавляем элемент 3 в очередь	3	8 4
Удаляем элемент из очереди	3	8 – удаляем элемент 8 4
Удаляем элемент из очереди	3	4 – удаляем элемент из очереди
Удаляем элемент из очереди	Элементы переместили во второй стек. Стек пустой	3 – удаляем элемент из очереди

Как мы видим, каждый элемент один раз помещается в стек, один раз перекладывается в другой стек, и один раз удаляется из стека. Операция добавления выполняется за $O(1)$. Операция удаления в худшем случае выполняется за $O(n)$.

Модифицируем реализацию очереди при помощи двух стеков для нахождения минимума на отрезке за $O(1)$. Будем хранить в стеках пары <элемент, текущий минимум стека>.

В алгоритме реализуется идея скользящего окна – по массиву движется окно фиксированного размера К, в котором определяется минимум. В первый стек помещаем К элементов, записываем в ответ текущий минимум. Затем в первый стек добавляем (К + 1) элемент. Извлекаем первый элемент очереди – для этого перекладываем все элементы из стека tail в стек head с новым минимумом для второго стека. Удаляем элемент из вершины второго стека. На вершине второго стека будет находиться минимум для второго подотрезка – записываем ответ. Добавим новый элемент в очередь, выведем ответ – минимум из двух вершин стека. Удаляем элемент из очереди. Продолжаем алгоритм далее. Иллюстрацию приведем на массиве A[10]={1, 8, 4, 3, 5, 7, 4, 5, 6, 7}. К = 3

Первый стек **tail**	Ответ **ans**	Второй стек **head**
(4,1) Стек заполнили К элементами. (8,1) **Записали** ответ из вершины стека. (1,1)	1	
(3,1) **Добавили** в стек следующий элемент. (4,1) (8,1) (1,1)	1	
Перекладываем элементы во второй стек с новым минимумом для второго стека и удалили элемент с вершины стека	1 3	(1,1) – **удалили** элемент (8,3) – **записали** ответ (4,3) (3,3)
(5,5) **добавили** элемент в очередь	1 3	(8,3) – **удаляем** элемент (4,3) (3,3)
(5,5) **Записали** ответ – минимум из двух вершин стеков	1 3 3	(4,3) (3,3)
(7,5) **Добавили** элемент в очередь (5,5)	1 3 3 3	(4,3) – **удаляем** элемент (3,3)
Продолжаем алгоритм далее.		

Таким образом, извлечение минимума происходит с вершин стеков если они оба не пустые, или из вершины непустого стека если имеется один пустой стек.

```
if (tail.empty() || head.empty())
     min = tail.empty() ? head.top().second : tail.top().second;
else
     min = min(tail.top().second, head.top().second);
```

При добавлении элементов в виде пары <элемент, минимум>, нужно определять текущий минимум стека, который становится вторым элементом пары.

```
int m = tail.empty() ? new_el : min(new_el, tail.top().second);
tail.push(make_pair(new_el, m));
```

Извлечение элемента происходит из стека head если он не пустой, в противном случае элементы их первого стека перекладываются в стек head, а затем элемент извлекается из вершины стека.

```
if (head.empty())
     while (!tail.empty())
     {
       int el = tail.top().first;
       tail.pop();
       int m = head.empty() ? el : min(el, head.top().second);
       head.push(make_pair(el, m));
     }
res = head.top().first;
head.pop();
```

Минимум на одном подотрезке модификацией очереди в виде двух стеков описанным способом определяется за $O(1)$. Таким образом, сложность работы всего алгоритма – $O(n)$.

Дек STL. Задача нахождения минимального элемента на фиксированном отрезке при помощи деков.

Дек **deque** (двусторонняя очередь) является контейнером позволяющим добавлять элементы в конец push_back() и начало очереди push_front() и удалять элементы в начале pop_back() и конце pop_front() дека. deque поддерживает доступ к произвольному элементу.

Решим задачу нахождения минимального на фиксированном отрезке при помощи деков. Рассмотрим сначала ситуацию, когда нам нужно найти минимальный на одном подотрезке.

В деке нужно хранить не все элементы, а только нужные для определения минимума. В этом случае очередь представляет собой неубывающую последовательность чисел.

```
deque<int> q; // Дек для хранения неубывающей последовательности
```

В вершине такого дека хранится минимум последовательности.

```
min = q.front();
```

Добавление элементов в дек происходит следующим образом: пока в конце очереди элементы, большие или равные данному, то удаляем их, и добавляем новый элемент в конец дека.

```
while (!q.empty() && q.back() > new_el)
     q.pop_back();
q.push_back(new_el);
```

Тем самым мы, с одной стороны, не нарушим порядка, а с другой стороны, не потеряем текущий элемент, если он на каком-либо последующем шаге окажется минимумом. Но при извлечении элемента из головы дека его там может и не оказаться. Это происходит потому, что модифицированная очередь могла удалить этот элемент в процессе перестроения. Поэтому при удалении элемента необходимо значение извлекаемого элемента - если элемент с этим значением находится в голове дека, то он извлекается, а в противном случае никаких действий не производится.

```
if (!q.empty() && q.front() == rem_el)
     q.pop_front();
```

Минимум на одном подотрезке описанным алгоритмом определяется за $O(1)$.

Для применения этого алгоритма в случае определения минимума на всех подотрезках фиксированной длины заданного массива необходимо, как и в случае очереди на двух стеках,

делать следующие действия. Сначала записать указанным способом в дек К элементов, **записать** ответ – текущий минимум на подотрезке, **добавить** новый элемент в дек, **удалить** (если это возможно) первый элемент из дека, **записать** новый ответ из начала дека, **добавить** элемент в конец дека, **удалить** (если это возможно) элемент из начала очереди и так далее.

Сложность работы всего алгоритма – $O(n)$.

Реализация задачи для нахождения минимума на фиксированном подотрезке при помощи дека проще, чем при помощи модификации очереди двумя стеками. Но для способа с деком придется хранить весь массив, а в случае с двумя стеками весь массив хранить не нужно – нужно лишь знать значение очередного i элемента.

Задания

1. Составьте таблицу сложности выполнения основных операций (добавление элементов, удаление элементов для структурами: вектор, стек, очередь, дек).

2. По заданному текстовому файлу получите новый текстовый файл, в котором слова из первого файла расположены в обратном порядке.

3. Для данной матрицы прямоугольности растрового изображения, содержащей номера цветов пикселей, реализуйте функцию закраски связной области. На вход программа получает размеры матрицы, элементы матрицы в виде чисел [0;255] – номера цветов точек, координаты точки (i,j), с которой начинается заливка связной области и цвет заливки. Программа должна вывести матрицу после заливки.

4. Реализуйте дек и методы работы с ним при помощи статического массива.

5. Реализуйте при помощи векторов метод Ван Херка определения минимального на всех подотрезках фиксированной длины. Суть метода заключается в том, что для данного массива заполняются два вспомогательных массива – массив префиксных минимумов на блоках длины К и максим постфиксных минимумов для блоков длины К. Для массива A[10]={1, 8, 4, 3, 5, 7, 4, 5, 6, 7} при K=3 получим prefix[10]={1,1,1,3,3,3,4,4,4,7} и suffix[10]={1,4,4,3,5,7,4,5,6,7}. Результирующий массив ответов получаем по следующему закону: минимум на отрезке [i, i+K-1] равен min(suffix[i], prefix[i+K-1]). ans={1, 3, 3, 3, 4, 4, 4, 5}.

Литература

Ворожцов А.В., Винокуров Н.А. Лекции «Алгоритмы: построение, анализ и реализация на языке программирования Си». – М.: Издательство МФТИ, 2007.

С/С++. Программирование на языке высокого уровня /Т.А. Павловская. – СПб.: Питер, 2003.

А.Шень. Программирование: теоремы и задачи. – М.: МЦНМО, 2004. Второе издание, исправленное и дополненное.

Зимняя школа по программированию. – Харьков: ХНУРЭ, 2013.

Густакашин М. Стеки, очереди, деки. http://informatics.mccme.ru/file.php/18/stacks-etc.pdf

Модификация стека и очереди для нахождения минимума за O (1). http://e-maxx.ru/algo/stacks_for_minima