АЛГОРИТМЫ И СТРУКТУРЫ ДАННЫХ

Лекция 1

Мацкевич С.Е.

Позвольте представиться

Мацкевич Степан Евгеньевич

- Учился в 57 школе в 1998-2001гг,
- Окончил мех-мат МГУ в 2006г,
- Кандидат физ.-мат. наук в 2010г,
- Работаю в «ABBYY» с 2006 по н.в. Участвовал в разработке:
  - ABBYY Compreno (анализ текста, перевод),
  - ABBYY FactExtractor (система хранения фактов),
  - ABBYY InfoExtractor (извлечение информации из текста)
- Преподаю в МФТИ с 2009 по н.в. на факультете Инноваций и Высоких Технологий (доцент):
  - Алгоритмы и структуры данных (лекции и семинары),
  - Программирование под Windows (лекции и семинары).

План занятий

1. Базовые алгоритмы и структуры данных.
   - Лекция 1. Введение в курс. Массивы.
   - Семинар 1. Базовые алгоритмы. Поиск элемента в массиве. Бинарный поиск.
   - Лекция 2. Базовые структуры данных. Динамическое программирование и жадные алгоритмы.
   - Семинар 2. Списки, стек, очередь, дек.
   - Семинар 3. Динамическое программирование и жадные алгоритмы.
   - Рубежный контроль 1.

План занятий

2. Сортировки.
   - Лекция 3. Сортировки 1.
   - Семинар 4. Сортировки с использованием операции сравнения.
   - Лекция 4. Сортировки 2.
   - Семинар 5. Порядковые статистики и сортировка слиянием.
   - Семинар 6. Поразрядные сортировки. Соревнование.
   - Рубежный контроль 2.

План занятий

3. Хеш-таблицы. Деревья.

- Лекция 5. Хеш-таблицы.
- Семинар 7. Хеш-таблицы.
- Лекция 6. Деревья.
- Семинар 8. Двоичные деревья поиска. Декартовы деревья.
- Семинар 9. АВЛ деревья. Сплей-деревья.
- Рубежный контроль 3.

План лекции 1 «Введение в курс. Элементарные алгоритмы»

МГТУ им. Н.Э. Баумана ТЕХНОПАРК Mail.Ru Group

- Понятие алгоритма и структуры данных.
- Обзор алгоритмов и структур данных. Литература.
- Понятие вычислительной сложности. O-нотация.
- Вычисление n-ого числа Фибоначчи.
- Проверка числа на простоту.
- Быстрое возведение числа в целую степень (за log(n)).
- Массивы. Однопроходные алгоритмы.
- Линейный поиск. Поиск минимального элемента.
- Бинарный поиск. Рекурсивный и нерекурсивный алгоритмы.
- Динамический массив.

Определения

- **Алгоритм —** это формально описанная вычислительная процедура, получающая исходные данные (input), называемые также входом алгоритма или его аргументом, и выдающая результат вычисления на выход (output).

```
output Функция( input )
{
    процедура;
}
```

Алгоритм определяет функцию (отображение) $F : X \rightarrow Y$.
X – множество исходных данных, Y – множество значений.

Определения

- **Структура данных —** программная единица, позволяющая хранить и обрабатывать множество однотипных и/или логически связанных данных.

Типичные операции:

- добавление данных,
- изменение данных,
- удаление данных,
- поиск данных.

Базовые алгоритмы с массивами

- Стандартные операции с массивом.
  - Доступ к элементам по индексу. Изменение элементов.
  - Линейный (последовательный) поиск.
  - Бинарный поиск.
- Сортировка массива.
  - Сортировка вставками, сортировка пузырьком, сортировка выбором.
  - Сортировка слиянием,
  - Быстрая сортировка,
  - Пирамидальная сортировка,
  - Поразрядные сортировки,
  - TimSort.

Обзор алгоритмов и структур данных

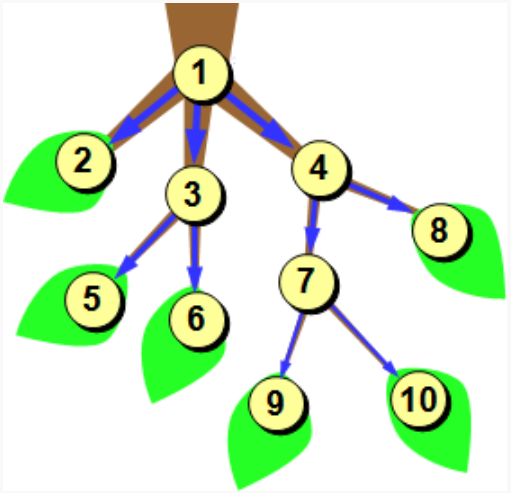
Базовые структуры данных

- СД «Динамический массив»
- СД «Однонаправленный список» и «Двунаправленный список»
- СД «Стек»
- СД «Очередь»
- СД «Дек»
- СД «Двоичная куча»,
- АТД «Очередь с приоритетом»

Обзор алгоритмов и структур данных

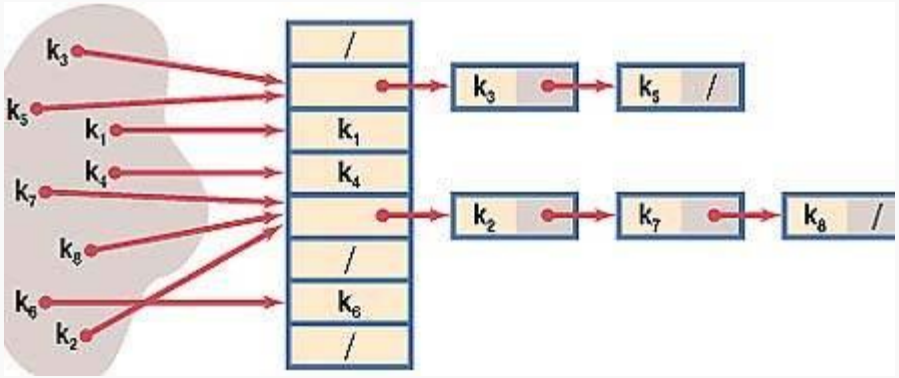
Деревья

- СД «Двоичное дерево»
  - Алгоритмы обхода дерева в ширину и в глубину.
- СД «Дерево поиска». Сбалансированные деревья
  - СД «Декартово дерево»,
  - СД «АВЛ-дерево»,
  - СД «Красно-черное дерево»,
  - СД «Сплей дерево»,
  - АТД «Ассоциативный массив».
- СД «Дерево отрезков»
- СД «B-дерево»



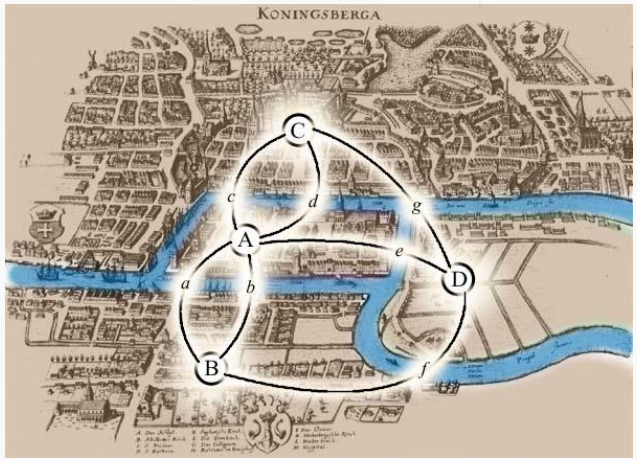
Хеширование

- Алгоритмы вычисления хеш-функций.
  - Хеш-функции для строк.
- СД «Хеш-таблица»
  - Реализация хеш-таблицы методом цепочек,
  - Реализация хеш-таблицы методом открытой адресации.
- Блюм-фильтр



Обзор алгоритмов и структур данных

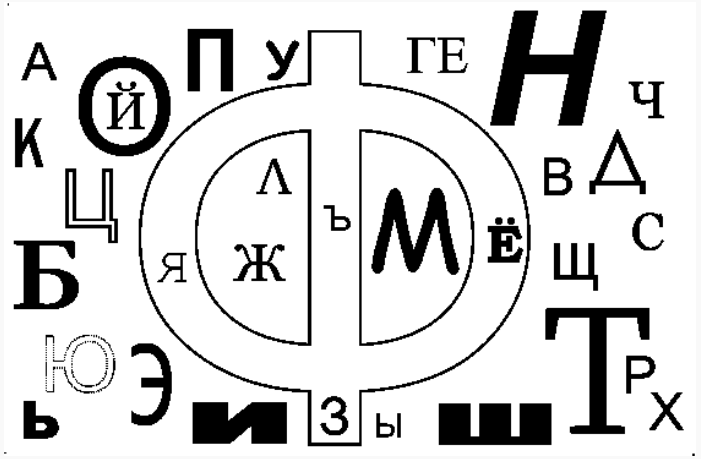
Графы



- Обходы в ширину и глубину.
- Топологическая сортировка.
- Поиск сильносвязных компонент.
- Поиск кратчайших путей между вершинами.
- Поиск Эйлерова пути.
- Поиск Гамильтонова пути минимального веса. Задача коммивояжера.
- Нахождение остовного дерева минимального веса.
- Вычисление максимального потока в сети.
- Нахождение наибольшего паросочетания в двудольном графе.
- Вычисление хроматического числа графа.

Обзор алгоритмов и структур данных

Строки



- Поиск подстрок
  - Алгоритм Кнута-Морриса-Пратта,
  - Алгоритм Ахо-Корасик
- Индексирование текста
  - Бор,
  - Суффиксный массив,
  - Суффиксное дерево,
  - Суффиксный автомат.
- Регулярные выражения.
- Вычисление редакторского расстояния между строками.

Обзор алгоритмов и структур данных

- Вычислительная геометрия
- Теория игр
- Полиномы и быстрое преобразование Фурье
- Матрицы
- Алгоритмы сжатия
- Численные методы решения уравнений
- Машинное обучение

Литература

- **Кормен Т., Лейзерсон Ч., Ривест Р., Штайн К, Алгоритмы. Построение и анализ**., 2-е изд, М.: Издательский дом «Вильямс», 2005.
- **Седжвик Р., Фундаментальные алгоритмы на C++, части 1-4, анализ, структуры данных, сортировка, поиск**, М.: ДиаСофт, 2001.
- **Шень А., Программирование: теоремы и задачи**, 2-е изд., испр. и доп., М.: МЦНМО, 2004.
- **Викиконспекты,** http://neerc.ifmo.ru/wiki/index.php?title=Дискретная_математика,_алгоритмы_и_структуры_данных.

Анализ алгоритмов

Эффективность алгоритма определяется:

- Временем работы,
- Объемом дополнительно используемой памяти,
- Другими характеристики.
  Например, количеством операций сравнения или количеством обращений к диску.

Анализ алгоритмов

Часто исходные данные характеризуются натуральным числом **n**.

Тогда время работы алгоритма – **T(n)**.

Объем доп. памяти – **M(n)**.

Пример 1. Сортировка массива. Важен размер исходного массива – **n**.

Исходные данные могут характеризоваться несколькими числами.

Пример 2. Поиск всех вхождений строки-шаблона T длины **k** в строку S длины n. В этой задаче время работы алгоритма **T(n, k)** может зависеть от двух чисел **n** и **k**.

Асимптотические обозначения

Для обозначения асимптотического поведения времени работы алгоритма (или объема памяти) используются Θ, O и Ω – обозначения.

Определение. Для функции g(n) записи Θ(g(n)), O(g(n)) и Ω(g(n)) означают следующие множества функций:

$\Theta(g(n)) = \{f(n)$: существуют положительные константы c_1, c_2 и n_0, такие что $0 \le c_1 g(n) \le f(n) \le c_2 g(n)$ для всех $n \ge n_0\}$,

$O(g(n)) = \{f(n)$: существуют положительные константы c и n_0, такие что $0 \le f(n) \le cg(n)$ для всех $n \ge n_0\}$,

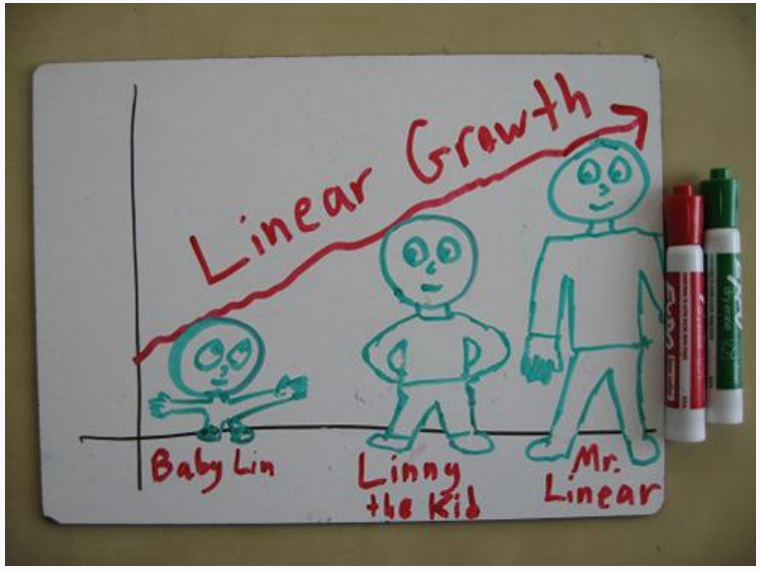
$\Omega(g(n)) = \{f(n)$: существуют положительные константы c и n_0, такие что $0 \le cg(n) \le f(n)$ для всех $n \ge n_0\}$.

Обозначение. Вместо записи «$T(n) \in \Theta(g(n))$» часто используют запись «$T(n) = \Theta(g(n))$».

Асимптотические обозначения

Примеры.

- $T(n) = \Theta(n)$.
  Линейное время.
  разг. «за линию».
- $T(n) = \Theta(n^2)$.
  Квадратичное время.
  разг. «за квадрат».
- $T(n) = \Theta(\log n)$.
  Логарифм.
- $T(n) = \Theta(n \log n)$.
- $T(n) = \Theta(e^n)$.



Числа Фибоначчи

Задача. Найти n-ое число Фибоначчи.

$F(0) = 1; F(1) = 1;$
$F(n) = F(n-1) + F(n-2);$

1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, …

Числа Фибоначчи в цветках тысячелистника.

Есть рекурсивный и нерекурсивный алгоритмы.

Числа Фибоначчи


```
// Рекурсивный алгоритм.
int Fibonacci1( int n )
{
    if( n == 0 || n == 1 ) {
        return 1;
    }
    return Fibonacci1( n - 1 ) + Fibonacci1( n - 2 );
}
```

Числа Фибоначчи


```
// Нерекурсивный алгоритм.
int Fibonacci2( int n )
{
    if( n == 0 ) {
        return 1;
    }
    int prev = 1; // F(0).
    int current = 1; // F(1).
    for( int i = 2; i <= n; ++i ) {
        int temp = current;
        current += prev; // Вычисление F(i).
        prev = temp; // Запоминаем F(i-1).
    }
    return current;
}
```

Числа Фибоначчи

МГТУ им. Н.Э. Баумана ТЕХНОПАРК Mail.Ru Group

Рекурсивный алгоритм

Время работы T(n) больше, чем количество вызовов Fibonacci1(1), которое равно F(n-1).

Формула Бине: $\boldsymbol{F(n) = \frac{\varphi^n - (-\varphi)^{-n}}{2\varphi - 1}}$, где $\boldsymbol{\varphi} = 1{,}6180339887$ – золотое сечение.

Таким образом, $\boldsymbol{T(n) = \Omega(\varphi^n)}$.

Объем доп. памяти $\boldsymbol{M(n) = O(n)}$ – максимальная глубина рекурсии.

Нерекурсивный алгоритм

Время работы $\boldsymbol{T(n) = O(n)}$ – количество итераций в цикле.

Объем доп. памяти $\boldsymbol{M(n) = O(1)}$.

Проверка числа на простоту

Задача. Проверить, является ли заданное натуральное число n простым.

Можем быстро определять, делится ли одно натуральное число (n) на другое (k), проверив остаток от деления:

```
n % k == 0
```

Будем перебирать все числа от 1 до $\sqrt{n}$, проверяя, делит ли какое-нибудь из них n.

Проверка числа на простоту


```
bool IsPrime( int n )
{
    if( n == 1 ) {
        return false;
    }
    for( int i = 2; i * i <= n; ++i ) {
        if( n % i == 0 ) {
            return false;
        }
    }
    return true;
}
```

Проверка числа на простоту

Время работы $\boldsymbol{T(n) = O(\sqrt{n})}$.

Объем доп. памяти $\boldsymbol{M(n) = O(1)}$.

- Существует алгоритм, проверяющий число на простоту за полиномиальное время (от log) – Тест Агравала – Каяла – Саксены (2002 год). Время работы его улучшенной версии (2011год) $\boldsymbol{T(n) = O(\log^3 n)}$.

Быстрое возведение в степень

Задача. Дано число a и неотрицательное целое число n. Найти $\boldsymbol{a^n}$.

Тривиальный алгоритм: перемножить n-1 раз число a:

$$\boldsymbol{a \cdot a \cdot \cdots \cdot a}$$

Время работы тривиального алгоритма $\boldsymbol{T(n) = O(n)}$.

- Воспользуемся тем, что $\boldsymbol{a^{2^k} = \left(\left(a^2\right)^2\right)^{\cdots 2}}$ ($\boldsymbol{k}$ раз).

  Если $\boldsymbol{n = 2^{k_1} + 2^{k_2} + \cdots + 2^{k_s}}$, $\boldsymbol{k_1, k_2, \ldots, k_s}$ **– различны, то**

$$\boldsymbol{a^n = a^{2^{k_1}} a^{2^{k_2}} \cdots a^{2^{k_s}}}.$$

Быстрое возведение в степень

- Как получить разложение $n = 2^{k_1} + 2^{k_2} + \cdots + 2^{k_s}$, где $k_1, k_2, \ldots, k_s$ – различны?

`n = 10011010` – в двоичной системе счисления.
`7 43 1` – степени двоек в разложении n.

Начнем с младших степеней.
`result = 1, aInDegreeOf2 = a`.

`n = 10011010` – пусть пройдено 3 шага алгоритма.
Если следующий бит == 1, то домножим `result` на `aInDegreeOf2` $= a^{2^3}$.
Вне зависимости от бита возводим `aInDegreeOf2` в квадрат.

Быстрое возведение в степень

- Как извлекать очередной бит из n?

Будем на каждом шаге сдвигать n вправо на 1 бит. Т.е. делить n пополам.

```
n = n >> 1;
```

Тогда интересующий бит будет всегда располагаться последним. Последний бит соответствует четности числа, достаточно проверить остаток от деления на 2:

```
n % 2 == 1.
```

Или с помощью битовых операций:

```
n & 1 == 1.
```

Быстрое возведение в степень


```
double Power( double a, int n )
{
    double result = 1; // Для хранения результата.
    double aInDegreeOf2 = a; // Текущее значение ((a^2)^2…)^2
    while( n > 0 ) {
        // Добавляем нужную степень двойки к результату,
        // если она есть в разложении n.
        if( n & 1 == 1 ) {
            result *= aInDegreeOf2;
        }
        aInDegreeOf2 *= aInDegreeOf2;
        n = n >> 1; // Можно писать n /= 2.
    }
    return result;
}
```

Быстрое возведение в степень

Количество итераций цикла = степень двойки, не превышающая n, т.е. $\log(n)$.

Каждая итерация цикла требует ограниченное количество операций, $O(1)$.

Доп. память не используется.

$T(n) = O(\log n)$,
$M(n) = O(1)$.

Массивы

Определение 1. Массив – набор однотипных компонентов (элементов), расположенных в памяти непосредственно друг за другом, доступ к которым осуществляется по индексу (индексам).

Традиционно индексирование элементов массивов начинают с 0.

Определение 2. Размерность массива – количество индексов, необходимое для однозначного доступа к элементу массива.

- **Одномерный массив целых чисел:**

20	34	11	563	23	-1	2	0	-33	7
0	1	2	3	4	5	6	7	8	9

Массивы

Для передачи массива в функцию можно передать указатель на начало массива и количество элементов.

```
void Function1( int* arr, int count );
void Function2( const int* arr, int count );
```

Массивы

- **Строка – массив символов.**

Часто после последнего символа ставят символ с кодом 0. Чтобы во время передачи строки не передавать число элементов.

Символы могут быть однобайтными (char) или двухбайтными (wchar_t).

```
void Function1( char* str );
void Function2( const char* str );
void Function3( wchar_t* str );
void Function4( const wchar_t* str );
```

Т	е	х	н	о	п	а	р	к	\0
0	1	2	3	4	5	6	7	8	9

СД «Массив»


```
// Создание массивов в C++:

// Массив из 10 целых чисел. Создается на стеке
потока.
int intArray1[10];

// Массив из заранее неизвестного количества целых
чисел.
// Создается в куче процесса.
int n = 0;
cin >> n;
int* intArray2 = new int[n];
delete[] intArray2;
```

Массивы. Линейный поиск

Задача 1. Проверить, есть ли заданный элемент в массиве.

Решение. Последовательно проверяем все элементы массива, пока не найдем заданный элемент, либо пока не закончится массив.

Время работы в худшем случае $T(n) = O(n)$, где n – количество элементов в массиве.

Массивы. Линейный поиск.


```
// Проверка наличия элемента.
bool HasElement( const double* arr, int count,
     double element )
{
     for( int i = 0; i < count; ++i ) {
          if( arr[i] == element ) { // Нашли.
               return true;
          }
     }
     return false;
}
```

Массивы. Линейный поиск

Задача 2. Проверить, есть ли заданный элемент в массиве. Если он есть, вернуть позицию его первого вхождения. Если его нет, вернуть -1.

Решение. Последовательно проверяем все элементы массива, пока не найдем заданный элемент. Если нашли, возвращаем его позицию. Если не нашли, возвращаем -1.

Время работы в худшем случае $T(n) = O(n)$, где n – количество элементов в массиве.

Массивы. Линейный поиск.


```
// Возвращает позицию элемента, если он есть
// в массиве. Возвращает -1, если его нет.
int FindElement( const double* arr, int count,
    double element )
{
    for( int i = 0; i < count; ++i ) {
        if( arr[i] == element ) { // Нашли.
            return i;
        }
    }
    return -1;
}
```

Массивы. Линейный поиск

Задача 3. Найти максимальный элемент в массиве. Вернуть его значение.

Решение. Последовательно проверяем все элементы массива, запоминаем текущее значение максимума.

Время работы $T(n) = O(n)$, где n – количество элементов в массиве.

Массивы. Линейный поиск.


```
// Возвращает максимальный элемент в массиве.
double MaxElement( const double* arr, int count )
{
    // Число элементов должно быть больше 0.
    assert( count > 0 );
    double currentMax = arr[0];
    for( int i = 1; i < count; i++ ) {
        if( arr[i] > currentMax ) { // Нашли новый.
            currentMax = arr[i];
        }
    }
    return currentMax;
}
```

Массивы. Бинарный поиск.

Определение. Упорядоченный по возрастанию массив – массив A, элементы которого сравнимы, и для любых индексов k и l, $k < l$:

$$A[k] \leq A[l].$$

Упорядоченный по убыванию массив определяется аналогично.

-40	-12	0	1	2	6	22	54	343	711
0	1	2	3	4	5	6	7	8	9

Массивы. Бинарный поиск

Задача (Бинарный поиск = Двоичный поиск).
Проверить, есть ли заданный элемент в упорядоченном массиве. Если он есть, вернуть позицию его первого вхождения. Если его нет, вернуть -1.

Решение. Шаг. Сравниваем элемент в середине массива (медиану) с заданным элементом. Выбираем нужную половинку массива в зависимости результата сравнения. Повторяем этот шаг до тех пор, пока размер массива не уменьшится до 1.

Массивы. Бинарный поиск.


```
// Возвращает позицию вставки элемента на отрезке [first, last).
int FindInsertionPoint( const double* arr, int first,
    int last, double element )
{
    if( last - first == 1 )
        return element <= arr[first] ? first : last;

    int mid = ( first + last ) / 2;
    if( element <= arr[mid] )
        return FindInsertionPoint( arr, first, mid, element );
    else
        return FindInsertionPoint( arr, mid, last, element );
}


// Возвращает позицию элемента в упорядоченном массиве, если он есть.
// Возвращает -1, если его нет.
int BinarySearch( const double* arr, int count, double element )
{
    if( count == 0 ) return -1;
    int point = FindInsertionPoint( arr, 0, count, element );
    return ( point == count || arr[point] != element ) ? -1 : point;
}
```

Массивы. Бинарный поиск


```
// Бинарный поиск без рекурсии.
int BinarySearch2( double* arr, int count, double element )
{
    int first = 0;
    int last = count; // Элемент в last не учитывается.
    while( first < last ) {
        int mid = ( first + last ) / 2;
        if( element <= arr[mid] )
            last = mid;
        else
            first = mid + 1; // Здесь отличие от BinarySearch1.
    }
    // Все элементы слева от first строго больше искомого.
    return ( first == count || arr[first] != element ) ? -1 : first;
}
```

Массивы. Бинарный поиск

Время работы $T(n) = O(\log n)$, где n – количество элементов в массиве.

Объем дополнительной памяти:
В нерекурсивном алгоритме $M(n) = O(1)$.
В рекурсивном алгоритме $M(n) = O(\log n)$, так как максимальная глубина рекурсии – $\log n$.

Абстрактные типы данных и структуры данных

Определение. Абстрактный тип данных (**АТД**) — это тип данных, который предоставляет для работы с элементами этого типа определённый набор функций, а также возможность создавать элементы этого типа при помощи специальных функций.

Вся внутренняя структура такого типа спрятана – в этом и заключается суть абстракции.

АТД = Интерфейс

АТД «Динамический массив»

<u>Определение.</u> АТД «Динамический массив» — интерфейс с операциями

- Добавление элемента в конец массива «Add» (или PushBack),
- Доступ к элементу массива по индексу за $O(1)$ «GetAt» (или оператор []).

СД «Динамический массив»

Динамический массив содержит внутренний массив фиксированной длины для хранения элементов. Внутренний массив называется **буфером**.

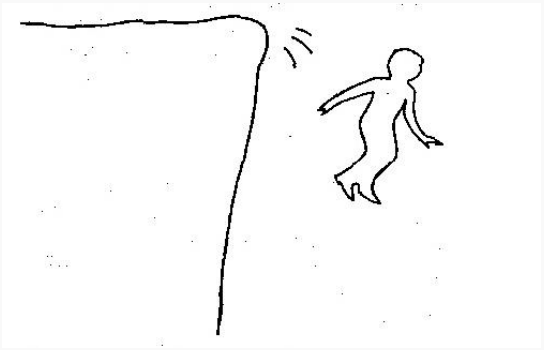
Помнит текущее количество добавленных элементов.

Размер буфера имеет некоторый запас для возможности добавления новых элементов.

Пример. Буфер размера 14 заполнен 10 элементами.

Т	**е**	**х**	**н**	**о**	**п**	**а**	**р**	**к**	**!**				
0	1	2	3	4	5	6	7	8	9	10	11	12	13

СД «Динамический массив»



Буфер может закончиться…

Если буфер закончился, то
при добавлении нового элемента:

- выделим новый буфер, больший исходного;
- скопируем содержимое старого буфера в новый;
- добавим новый элемент.

Т	е	х	н	о	п	а
0	1	2	3	4	5	6

Т	е	х	н	о	п	а	р						
0	1	2	3	4	5	6	7	8	9	10	11	12	13

СД «Динамический массив»


```
// Класс «Динамический массив».
class CArray {
public:
    CArray() : buffer( 0 ), bufferSize( 0 ), realSize( 0 ) {}
    ~CArray() { delete[] buffer; }

    // Размер.
    int Size() const { return realSize; }
    // Доступ по индексу.
    double GetAt( int index ) const;
    double operator[]( int index ) const { return GetAt( index ); }
    double& operator[]( int index );

    // Добавление нового элемента.
    void Add( double element );

private:
    double* buffer; // Буфер.
    int bufferSize; // Размер буфера.
    int realSize; // Количество элементов в массиве.

    void grow();
};
```

СД «Динамический массив»


```
// Доступ к элементу.
double CArray::GetAt( int index )
{
    assert( index >= 0 && index < realSize && buffer != 0 );
    return buffer[index];
}
// Увеличение буфера.
void CArray::grow()
{
    int newBufferSize = std::max( bufferSize * 2, DefaultInitialSize );
    double* newBuffer = new double[newBufferSize];
    for( int i = 0; i < realSize; ++i )
        newBuffer[i] = buffer[i];
    delete[] buffer;
    buffer = newBuffer;
    bufferSize = newBufferSize;
}
// Добавление элемента.
void CArray::Add( double element )
{
    if( realSize == bufferSize )
        grow( arr );
    assert( realSize < bufferSize && buffer != 0 );
    buffer[realSize++] = element;
}
```

СД «Динамический массив»

Как долго работает функция Add добавления элемента?

- В лучшем случае $= O(1)$
- В худшем случае $= O(n)$
- В среднем?

Имеет смысл рассматривать несколько операций добавления и оценить среднее время в контексте последовательности операций.

Подобный анализ называется **амортизационным**.

Амортизационный анализ

Определение (по Кормену…). При **амортизационном анализе** время, требуемое для выполнения последовательности операций над структурой данных, усредняется по всем выполняемым операциям.

Этот анализ можно использовать, например, чтобы показать, что даже если одна из операций последовательности является дорогостоящей, то при усреднении по всей последовательности средняя стоимость операций будет небольшой.

Амортизационный анализ

Амортизационный анализ отличается от анализа средних величин тем, что в нем не учитывается вероятность.

При амортизационном анализе гарантируется **средняя производительность операций в наихудшем случае**.

Амортизационный анализ

Определение. Пусть $S(n)$ – время выполнения последовательности всех n операций в наихудшем случае. **Амортизированной стоимостью (временем)** $AC(n)$ называется среднее время, приходящееся на одну операцию $S(n)/n$.

Оценим амортизированную стоимость операций Add динамического массива.

СД «Динамический массив»

Утверждение. Пусть в реализации функции *grow()* буфер удваивается. Тогда амортизированная стоимость функции *Add* составляет $O(1)$.

Доказательство. Рассмотрим последовательность из n операций *Add*.
Обозначим $P(k)$ - время выполнения *Add* в случае, когда $RealSize = k$.

- $P(k) \leq c_1 k$, если $k = 2^m$.
- $P(k) \leq c_2$, если $k \neq 2^m$.

$S(n) = \sum_{k=0}^{n-1} P(k) \leq c_1 \sum_{m:2^m<n} 2^m + c_2 \sum_{k:k\neq 2^m} 1 \leq$

$2c_1 n + c_2 n = (2c_1 + c_2)n.$

Амортизированное время $AC(n) = {}^{S(n)}/_{n} \leq 2c_1 + c_2 = O(1).$

Вопросы?

Спасибо за внимание!