АЛГОРИТМЫ И СТРУКТУРЫ ДАННЫХ

Лекция 1

Мацкевич С.Е.

С чего начать?

Не забудьте отметиться на портале:

https://park.mail.ru/

Позвольте представиться

Мацкевич Степан Евгеньевич

- 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).

План занятий

1. Базовые алгоритмы и структуры данных.
   - Лекция 1. Массивы. Базовые структуры данных. Динамическое программирование и жадные алгоритмы.
   - Семинар 1. Бинарный поиск. Списки, стек, очередь, дек.
   - Лекция 2. Сортировки. Порядковые статистики.
   - Семинар 2. Динамическое программирование и жадные алгоритмы. Квадратичные сортировки.
   - Семинар 3. Сортировки, порядковые статистики. Числовые сортировки.
   - Рубежный контроль 1.

План занятий

2. Хеш-таблицы. Деревья.

- Лекция 3. Хеш-таблицы.
- Семинар 4. Хеш-таблицы.
- Лекция 4. Деревья.
- Семинар 5. Двоичные деревья поиска. Декартовы деревья.
- Семинар 6. АВЛ деревья. Сплей-деревья.
- Рубежный контроль 2.

План занятий

3. Алгоритмы на графах.

- Лекция 5. Обходы графов. Поиск циклов, поиск компонент связности, топологическая сортировка.
- Семинар 7. Реализация графа. Обходы графов.
- Лекция 6. Кратчайшие пути. Остовные деревья.
- Семинар 8. Алгоритм Дейкстры.
- Семинар 9. Остовные 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.
- Понятие вычислительной сложности. O-нотация.
- Проверка числа на простоту.
- Быстрое возведение числа в целую степень (за log(n)).
- Массивы. Однопроходные алгоритмы. Бинарный поиск.
- Динамический массив.
- Списки.
- Стек, очередь, дек.
- Динамическое программирование.
- Жадные алгоритмы.
- Обзор алгоритмов и структур данных. Литература.

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) $F : X \to Y$.
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.

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:

$\Theta\big(g(n)\big)$
$= \{f(n)$: 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 $n \ge n_0\}$,

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

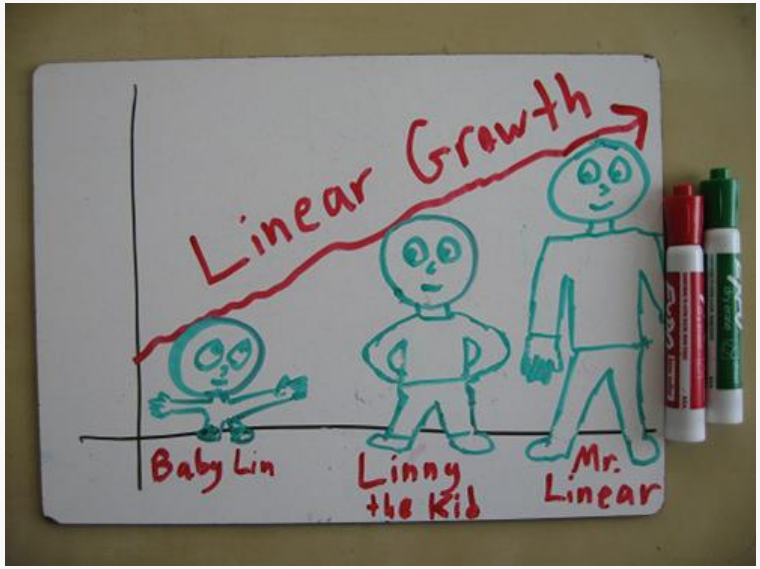
$\Omega\big(g(n)\big) = \{f(n)$: существуют положительные константы c и n_0, такие что $0 \le cg(n) \le f(n)$ для всех $n \ge n_0\}$.

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.
- $T(n) = \Theta(n \log n)$.
- $T(n) = \Theta(e^n)$.



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 a^n.

Тривиальный алгоритм: перемножить n-1 раз число a:

$$a \cdot a \cdot \dots \cdot a$$

Время работы тривиального алгоритма $T(n) = O(n)$.

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

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

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

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, aInPowerOf2 = a`.

`n = 10011010` - пусть пройдено 3 шага алгоритма.
Если следующий бит == 1, то домножим `result` на `aInPowerOf2` = a^{2^3}.
Вне зависимости от бита возводим `aInPowerOf2` 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 aInPowerOf2 = 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 *= aInPowerOf2;
        }
        aInPowerOf2 *= aInPowerOf2;
        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. 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.

```
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. 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


```
// Бинарный поиск без рекурсии. Ищем первое вхождение.
int BinarySearch( const 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( arr[mid] < element )
            first = mid + 1;
        else // В случае равенства arr[mid] останется справа.
            last = mid;
    }
    // Все элементы слева от first строго меньше искомого.
    return ( first == count || arr[first] != element ) ? -1 : first;
}
```

Массивы. Бинарный поиск


```
// Возвращает позицию вставки элемента на отрезке [first, last).
// Равные элементы располагаются после. (std::lower_bound)
int FindInsertionPoint( const 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( arr[mid] < element )
            first = mid + 1;
        else // В случае равенства arr[mid] останется справа.
            last = mid;
    }
    return first;
}
```

Массивы. Бинарный поиск

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

Объем дополнительной памяти:
$M(n) = O(1)$.

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»

Определение. 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**.

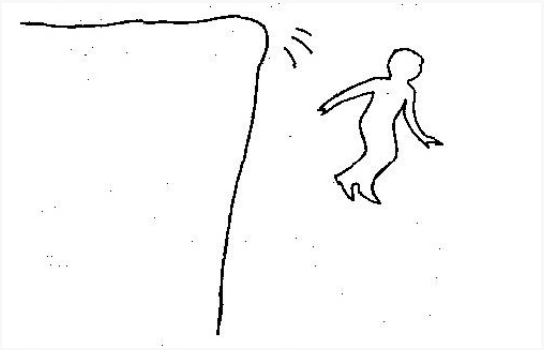
Помнит текущее количество добавленных элементов.

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();
    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

<u>Определение (по Кормену…).</u> 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.

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

Связные списки

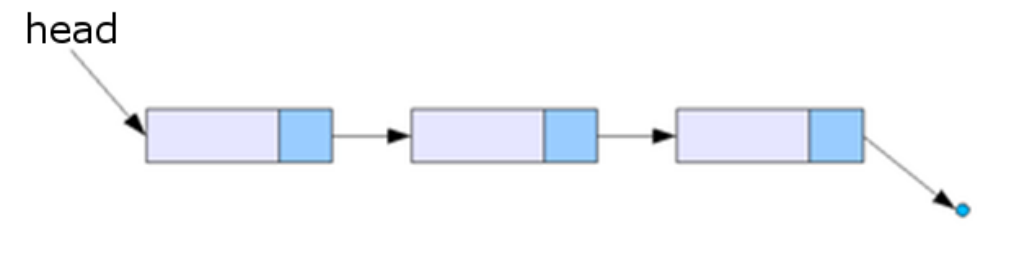
Определение. Связный список — динамическая структура данных, состоящая из узлов, каждый из которых содержит как собственно данные, так и одну или две ссылки («связки») на следующий и/или предыдущий узел списка.

Преимущество перед массивом:

- Порядок элементов списка может не совпадать с порядком расположения элементов данных в памяти, а порядок обхода списка всегда явно задаётся его внутренними связями.

Связные списки

Односвязный список (однонаправленный связный список)

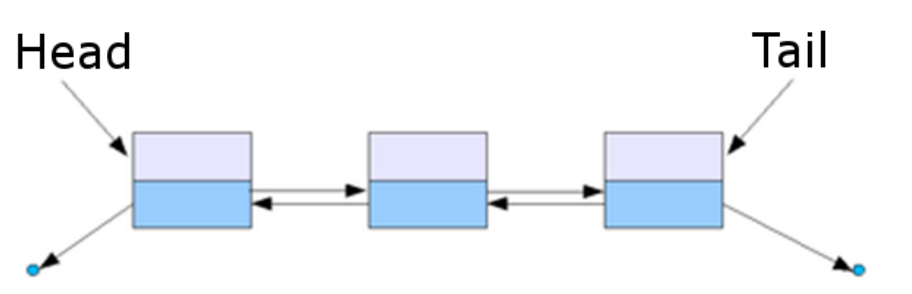


Ссылка в каждом узле одна.

Указывает на следующий узел в списке.

Узнать адрес предыдущего элемента, опираясь на содержимое текущего узла, невозможно.

Двусвязный список (Двунаправленный связный список)



Ссылки в каждом узле указывают на предыдущий и на последующий узел в списке.

Связные списки

Операции со списками:

- Поиск элемента,
- Вставка элемента,
- Удаление элемента,
- Объединение списков,
- …

Связные списки. Узел.


```
// Элемент двусвязного списка с целочисленными
// значениями.
struct CNode {
    int Data;
    CNode* Next;
    CNode* Prev;

    CNode() : Data( 0 ), Next( 0 ), Prev( 0 ) {}
};
```

Связные списки. Поиск.


```
// Линейный поиск элемента «a» в списке.
// Возвращает 0, если элемент не найден.
CNode* Search( CNode* head, int a )
{
    CNode* current = head;
    while( current != 0 ) {
        if( current->Data == a )
            return current;
        current = current->Next;
    }
    return 0;
}
```

Время работы в худшем случае = O(n), где n – длина списка.

Связные списки. Вставка.


```
// Вставка элемента «a» после текущего.
CNode* InsertAfter( CNode* node, int a )
{
    assert( node != 0 );
    // Создаем новый элемент.
    CNode* newNode = new CNode();
    newNode->Data = a;
    newNode->Next = node->Next;
    newNode->Prev = node;
    // Обновляем Prev следующего элемента, если он есть.
    if( node->Next != 0 ) {
        node->Next->Prev = newNode;
    }
    // Обновляем Next текущего элемента.
    node->Next = newNode;
    return newNode;
}
```

Время работы = O(1).

Связные списки. Удаление.


```
// Удаление элемента.
void DeleteAt( CNode* node )
{
    assert( node != 0 );
    // Обновляем Prev следующего элемента, если он есть.
    if( node->Next != 0 ) {
        node->Next->Prev = node->Prev;
    }
    // Обновляем Next предыдущего элемента, если он есть.
    if( node->Prev != 0 ) {
        node->Prev->Next = node->Next;
    }
    delete node;
}
```

Время работы = O(1).

Связные списки. Объединение.


```
// Объединение односвязных списков. К списку 1 подцепляем
список 2.
// Возвращает указатель на начало объединенного списка.
CNode* Union( CNode* head1, CNode* head2 ) {
    if( head1 == 0 ) {
        return head2;
    }
    if( head2 == 0 ) {
        return head1;
    }
    // Идем в хвост списка 1.
    CNode* tail1 = head1;
    for( ; tail1->Next != 0; tail1 = tail1->Next );
    // Обновляем Next хвоста.
    tail1->Next = head2;
    return head1;
}
```

Время работы = O(n), где n – длина первого списка.

Сравнение списков с массивами.

Недостатки списков:

- Нет быстрого доступа по индексу.
- Расходуется доп. память.
- Узлы могут располагаться в памяти разреженно, что не позволяет использовать кэширование процессора.

Преимущества списков перед массивом:

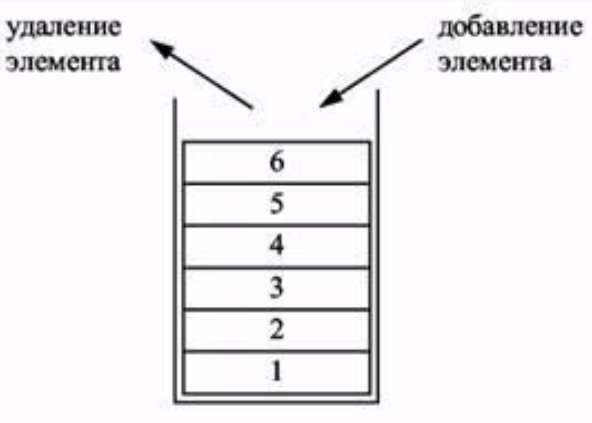
- Быстрая вставка узла.
- Быстрое удаление узла.

АТД «Стек»

Определение. Стек – абстрактный тип данных (или структура данных), представляющий из себя список элементов, организованный по принципу
LIFO = Last In First Out, «последним пришел, первым вышел».

Операции:

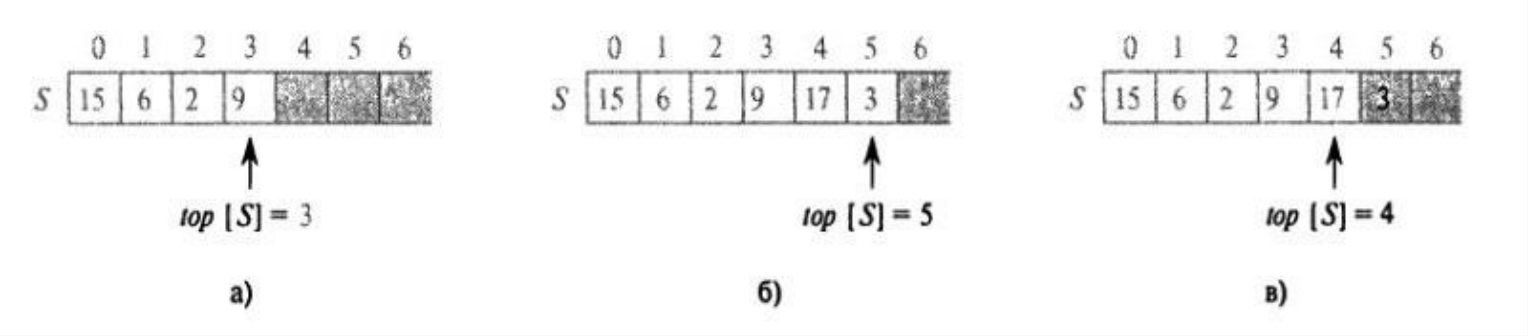
1. Вставка (Push)
2. Извлечение (Pop) – извлечение элемента, добавленного последним.



СД «Стек»

Стек можно реализовать с помощью массива или с помощью списка.

Реализация с помощью массива.



Храним указатель на массив и текущее количество элементов в стеке.

Можно использовать динамический массив.

СД «Стек»


```
// Стек целых чисел, реализованный с помощью массива.
class CStack {
public:
    explicit CStack( int _bufferSize );
    ~CStack();

    // Добавление и извлечение элемента из стека.
    void Push( int a );
    int Pop();

    // Проверка на пустоту.
    bool IsEmpty() const { return top == -1; }

private:
    int* buffer;
    int bufferSize;
    int top; // Индекс верхнего элемента.
};
```

СД «Стек»

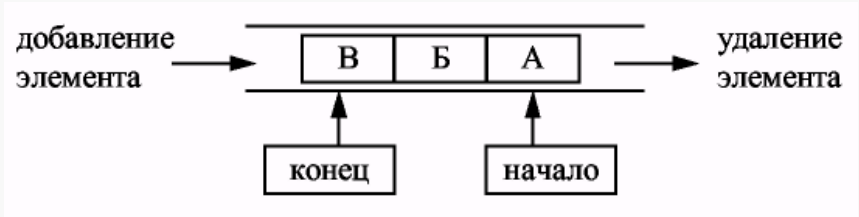

```
CStack::CStack( int _bufferSize ) :
    bufferSize( _bufferSize ),
    top( -1 )
{
    buffer = new int[bufferSize]; // Создаем буфер.
}
CStack::~CStack()
{
    delete[] buffer; // Удаляем буфер.
}
// Добавление элемента.
void CStack::Push( int a )
{
    assert( top + 1 < bufferSize );
    buffer[++top] = a;
}
// Извлечение элемента.
int CStack::Pop()
{
    assert( top != -1 );
    return buffer[top--];
}
```

АТД «Очередь»

Определение. Очередь – абстрактный тип данных (или структура данных), представляющий из себя список элементов, организованный по принципу
FIFO = First In First Out, «первым пришел, первым вышел».

Операции:

1. Вставка (Enqueue)
2. Извлечение (Dequeue) – извлечение элемента, добавленного первым.



СД «Очередь»

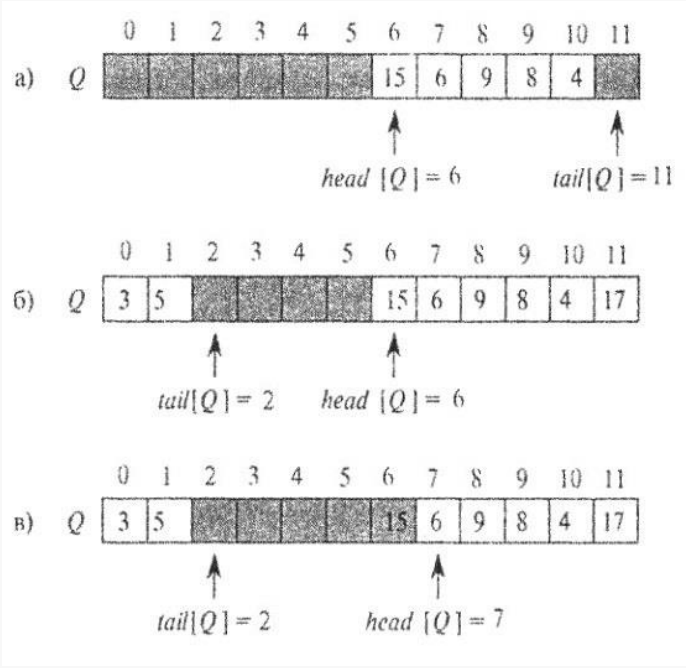
Очередь также как и стек можно реализовать с помощью массива или с помощью списка.

Реализация с помощью массива.

Храним указатель на массив, текущее начало и конец очереди.

Считаем массив зацикленным.

Можно использовать динамически растущий буфер.



СД «Очередь»


```
// Очередь целых чисел, реализованная с помощью массива.
class CQueue {
public:
    explicit CQueue( int size );
    ~CQueue() { delete[] buffer; }

    // Добавление и извлечение элемента из очереди.
    void Enqueue( int a );
    int Dequeue();

    // Проверка на пустоту.
    bool IsEmpty() const { return head == tail; }

private:
    int* buffer;
    int bufferSize;
    int head; // Указывает на первый элемент очереди.
    int tail; // Указывает на следующий после последнего.
};
```

СД «Очередь»


```
CQueue::CQueue( int size ) :
    bufferSize( size ),
    head( 0 ),
    tail( 0 )
{
    buffer = new int[bufferSize]; // Создаем буфер.
}

// Добавление элемента.
void CQueue::Enqueue( int a )
{
    assert( ( tail + 1 ) % bufferSize != head );
    buffer[tail] = a;
    tail = ( tail + 1 ) % bufferSize;
}
// Извлечение элемента.
int CQueue::Dequeue()
{
    assert( head != tail );
    int result = buffer[head];
    head = ( head + 1 ) % bufferSize;
    return result;
}
```

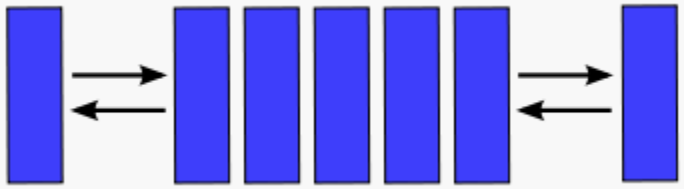
АТД «Дэк»

Определение. Двусвязная очередь – абстрактный тип данных (структура данных), в которой элементы можно добавлять и удалять как в начало, так и в конец, то есть принципами обслуживания являются одновременно FIFO и LIFO.

Операции:

1. Вставка в конец (PushBack),
2. Вставка в начало (PushFront),
3. Извлечение из конца (PopBack),
4. Извлечение из начала (PopFront).

Дек, также как стек или очередь, можно реализовать с помощью массива или с помощью списка.



Динамическое программирование

Динамическое программирование (ДП) – способ решения сложных задач путём разбиения их на более простые подзадачи.

Подход динамического программирования состоит в том, чтобы решить каждую подзадачу только один раз, сократив тем самым количество вычислений.

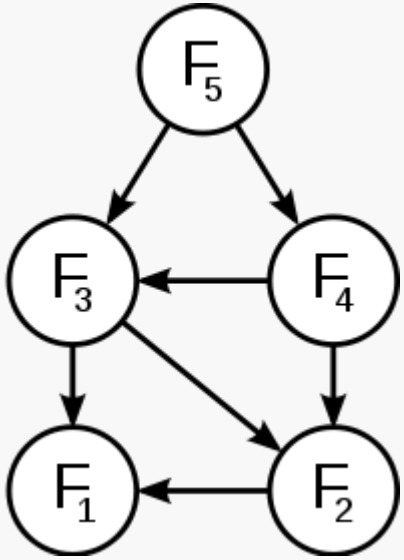
Термин «динамическое программирование» также встречается в теории управления в смысле «динамической оптимизации». Основным методом ДП является сведение общей задачи к ряду более простых экстремальных задач.

Лит.: Беллман Р., Динамическое программирование, пер. с англ., М., 1960.

Динамическое программирование

Пример. Числа Фибоначчи. $F_n = F_{n-1} + F_{n-2}$. $F_1 = 1, F_0 = 1$.

Можно вычислять рекурсивно:
$F_5 = F_4 + F_3$,
$F_4 = F_3 + F_2$,
$F_3 = F_2 + F_1$,



Многие значения могут вычисляться несколько раз.

Решение – сохранять результаты решения подзадач.
Этот подход называется кэшированием.

Динамическое программирование

Пример. Вычисление рекуррентных функций нескольких аргументов.

$$F(x,y) = 3 \cdot F(x-1,y) - 2 \cdot F^2(x,y-1),$$
$$F(x,0) = x, F(0,y) = 0.$$

Вычисление $F(x,y)$ сводится к вычислению двух $F(\cdot,\cdot)$ от меньших аргументов.

Есть перекрывающиеся подзадачи.

$F(x-1,y-1)$ в рекурсивном решении вычисляется дважды.

$F(x-2,y-1)$ в рекурсивном решении вычисляется три раза.

$F(x-n,y-m)$ в рекурсивном решении вычисляется C_{n+m}^{n} раз.

Снова будем использовать кэширование – сохранять результаты.

Вычисления будем выполнять от меньших аргументов к большим.

Динамическое программирование


```
// Вычисление рекуррентного выражения от двух переменных.
int F( int x, int y )
{
    vector<vector<int>> values( x + 1 );
    for( int i = 0; i <= x; ++i ) {
        values[i].resize( y + 1 );
        values[i][0] = i; // F( x, 0 ) = x;
    }
    for( int i = 1; i <= y; ++i ) {
        values[0][i] = 0; // F( 0, y ) = 0;
    }
    // Вычисляем по столбцам для каждого x.
    for( int i = 1; i <= x; ++i ) {
        for( int j = 1; j <= y; ++j ) {
            values[i][j] = 3 * values[i - 1][j] -
                2 * values[i][j - 1] * values[i][j - 1];
        }
    }
    return value[x][y];
}
```

Динамическое программирование

При вычисление рекуррентной функции $F(x, y)$ можно было не хранить значения на всех рядах.
Для вычисления очередного ряда достаточно иметь значения предыдущего ряда.

Важная оптимизация ДП: Запоминать только те значения, которые будут использоваться для последующих вычислений.

Для вычисления числа Фибоначчи F_i также достаточно хранить лишь два предыдущих значения: F_{i-1} и F_{i-2}.

Динамическое программирование

Принципы ДП:

1. Разбить задачу на подзадачи.
2. Кэшировать результаты решения подзадач.
3. Удалять более неиспользуемые результаты решения подзадач (опционально).

Динамическое программирование

Два подхода динамического программирования:

1. *Нисходящее динамическое программирование*: задача разбивается на подзадачи меньшего размера, они решаются и затем комбинируются для решения исходной задачи. Используется запоминание для решений часто встречающихся подзадач.
2. *Восходящее динамическое программирование*: все подзадачи, которые впоследствии понадобятся для решения исходной задачи просчитываются заранее и затем используются для построения решения исходной задачи.

Второй способ лучше нисходящего программирования в смысле размера необходимого стека и количества вызова функций, но иногда бывает нелегко заранее выяснить, решение каких подзадач нам потребуется в дальнейшем.

Динамическое программирование

Иногда бывает нелегко заранее выяснить, решение каких подзадач нам потребуется в дальнейшем.

Пример. $F(n) = F\left(\frac{n}{2}\right) + F\left(\frac{2 \cdot n}{3}\right), F(0) = F(1) = 1$, деление целочисленное.

Восходящее динамическое программирование применить можно, но будут вычислены все значения F от 1 до n. Сложно определить, для каких k требуется вычислять $F(k)$.

Нисходящее динамическое программирование позволит вычислить только те $F(k)$, которые требуются, но рекурсивно. Максимальная глубина рекурсии $= \log_{3/2}(n)$.

Динамическое программирование


```
// Вычисление рекуррентного выражения методом нисходящего ДП.
int F( int n )
{
    // Разумнее использовать std::map, а не std::vector.
    std::vector<int> values( n, -1 );
    values[0] = 1;
    values[1] = 1;
    return calcF( n, values );
}

// Рекурсивное вычисление с запоминанием.
int calcF( int n, std::vector<int>& values )
{
    int& cached = values[n];
    if( cached == -1 )
        cached = calcF( n / 2, values ) + calcF( 2 * n / 3, values );
    return cached;
}
```

Наибольшая общая подпоследовательность

Последовательность X является подпоследовательностью Y, если из Y можно удалить несколько элементов так, что получится последовательность X.

Задача. Наибольшая общая подпоследовательность (англ. longest common subsequence, LCS) – задача поиска последовательности, которая является подпоследовательностью нескольких последовательностей (обычно двух).

Элементами подпоследовательности могут быть числа, символы…

X = ABCAB,
Y = DCBA,
LCS(X, Y) = BA, CA, CB.

Наибольшая общая подпоследовательность

Будем решать задачу нахождения наибольшей общей подпоследовательности с помощью ДП.

Сведем задачу к подзадачам меньшего размера:

$f(n_1, n_2)$ – длина наибольшей общей подпоследовательности строк $s_1[0..n_1]$, $s_2[0..n_2]$.

$$f(n_1, n_2) = \begin{cases} 0, & n_1 = 0 \vee n_2 = 0 \\ f(n_1 - 1, n_2 - 1) + 1, & s[n_1] = s[n_2] \\ max(f(n_1 - 1, n_2), f(n_1, n_2 - 1)), & s[n_1] \neq s[n_2] \end{cases}$$

Наибольшая общая подпоследовательность

		A	B	C	A	B
	0	0	0	0	0	0
D	0	0	0	0	0	0
C	0	0	0	1	1	1
B	0	0	1	1	1	2
A	0	1	1	1	2	2

В каждой ячейке – длина наибольшей общей подстроки соответствующих начал строк.

Наибольшая общая подпоследовательность

$f(n_1, n_2)$ – длина НОП.

Как восстановить саму подпоследовательность?

Можно хранить в каждой ячейке таблицы «направление» перехода. Переход по диагонали означает, что очередной символ присутствует в обоих последовательностях.

Начинаем проход от правого нижнего угла.
Идем по стрелкам, на каждый переход по диагонали добавляем символ в начало строящейся подпоследовательности.

«Направления» можно не хранить, а вычислять по значениям в таблице.

Наибольшая общая подпоследовательность

		A	B	C	A	B
	0	0	0	0	0	0
D	0	←↑ 0	←↑ 0	←↑ 0	←↑ 0	←↑ 0
C	0	←↑ 0	←↑ 0	↖ 1	← 1	← 1
B	0	←↑ 0	↖ 1	←↑ 1	←↑ 1	↖ 2
A	0	↖ 1	←↑ 1	←↑ 1	↖ 2	←↑ 2

Желтым выделены ячейки, лежащие на лучшем пути.
Переход по диагонали соответствует совпадению символа.
Возможно существованию несколько лучших путей – по одному на каждую Наибольшую Общую Подпоследовательность.

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 выбором, пузырьком.
  - Сортировка слиянием,
  - Быстрая сортировка,
  - Пирамидальная сортировка,
  - Поразрядные сортировки,
  - TimSort.
- Вычисление порядковой статистики.

Обзор алгоритмов и структур данных

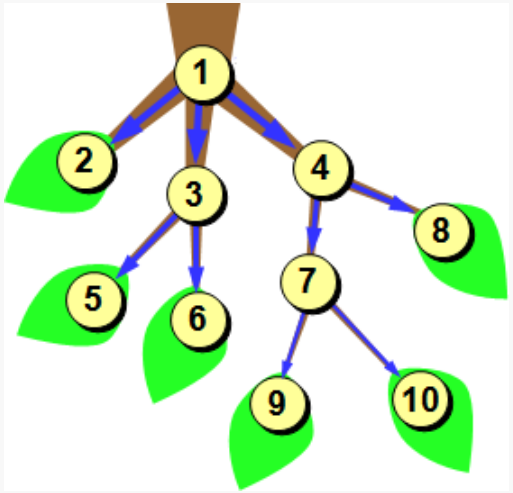
Базовые структуры данных

- 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

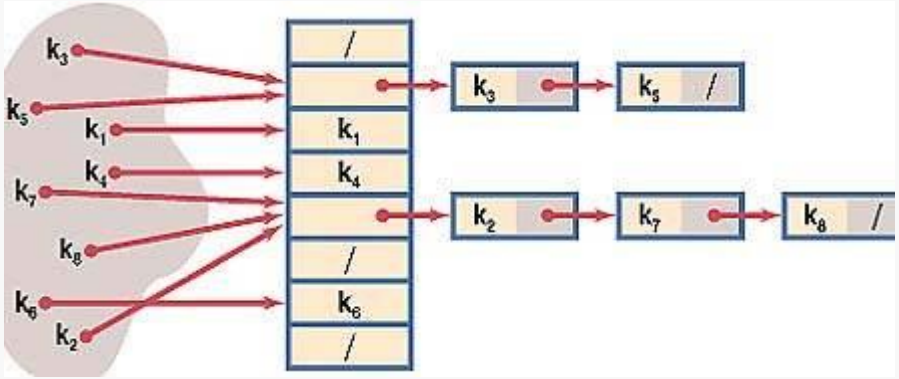
- СД «Двоичное дерево»
  - Алгоритмы обхода дерева в ширину и в глубину.
- СД «Дерево поиска». Сбалансированные деревья
  - СД «Декартово дерево»,
  - СД «АВЛ-дерево»,
  - СД «Красно-черное дерево»,
  - СД «Сплей дерево»,
  - АТД «Ассоциативный массив».
- СД «Дерево отрезков»
- СД «B-дерево»



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

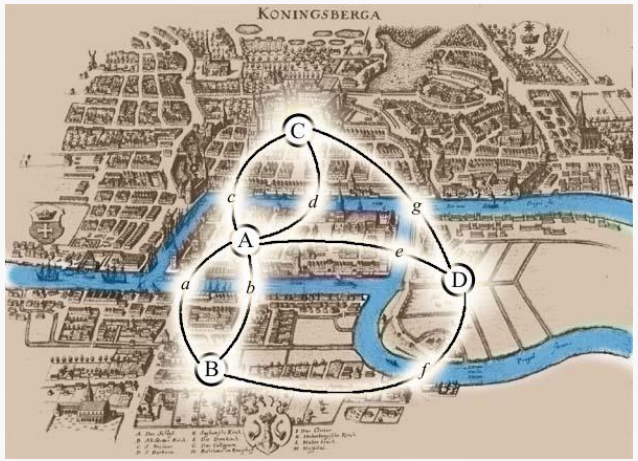
Хеширование

- 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

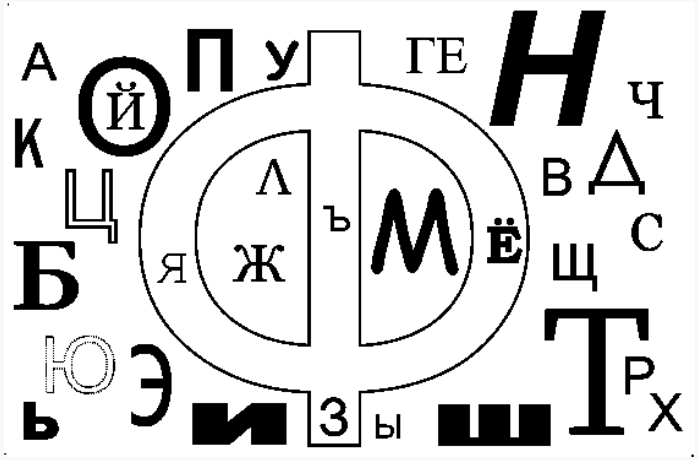
Графы



- 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.

Обратная связь

Почта smatsk@yandex.ru

Пишите письма с темой, содержащей «[ТР]».

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

Вопросы?

Спасибо за внимание!