*Решение задачи на компьютере*

1. Постановка задачи – точная формулировка условий задачи с описанием её входных и выходных данных

2. Разработка алгоритма решения задачи

3. Доказательство корректности алгоритма и анализ его эффективности

4. Реализация алгоритма на языке программирования

5. Выполнение программы для получения требуемого результата

7.Этапы решения задач на ЭВМ:

|  |  |
| --- | --- |
| **1. Постановка задачи:**  • сбор информации о задаче;  • формулировка условия задачи;  • определение конечных целей решения задачи;  • определение формы выдачи результатов;  • описание данных (их типов, диапазонов величин, структуры и т. п.).  **2. Анализ и исследование задачи, модели:**  • анализ существующих аналогов;  • анализ технических и программных средств;  • разработка математической модели;  • разработка структур данных.  **3. Разработка алгоритма:**  • выбор метода проектирования алгоритма;  • выбор формы записи алгоритма (блок-схемы, псевдокод и др.);  • выбор тестов и метода тестирования;  • проектирование алгоритма. | **4. Программирование:**  • выбор языка программирования;  • уточнение способов организации данных;  • запись алгоритма на выбранном языке программирования.  **5. Тестирование и отладка:**  • синтаксическая отладка;  • отладка семантики и логической структуры;  • тестовые расчеты и анализ результатов тестирования;  • совершенствование программы.  **6. Анализ результатов решения** задачи и уточнение в случае  необходимости математической модели с повторным выполнением этапов 2-5.  **7. Сопровождение программы:**  • доработка программы для решения конкретных задач;  • составление документации к решенной задаче, к математической модели, к алгоритму, к программе, к набору тестов, к использованию. |

**АЛГОРИТМЫ**

Алгоритм – система последовательных шагов, выполнение которых приводит к решению задачи.

Теория алгоритмов (основатели: Алан Тьюринг и Эмиль Пост) – наука, изучающая общие свойства и закономерности алгоритмов и модели их представления.

Алгоритмизация – процесс подготовки решения задачи на ЭВМ.

Запись алгоритма на формальном языке – **программа**.

**Классы алгоритмов**:

1. Вычислительные – исходные данные простые и их немного; процесс вычислений долгий и сложный. *Матем. и физич. задачи.*

2. Информационные – не очень сложные вычисления, большой объем данных. Требуют организации данных. *Эконом. задачи.*

3.Управляющие – исходные данные поступают от процесса или объекта, которым управляют. Результат – воздействие на процесс. Автоматизированное управление технолог. процессами (участие человека на определенных этапах; автоматическое – без участия).

4.Реального времени – результат должен быть получен к определенному времени.

|  |  |
| --- | --- |
| **Способы записи алгоритма:**  • словесно-формульный;  • структурный или блок-схемный;  • с помощью граф-схем;  • с помощью сетей Петри. | **Формы записи алгоритмов:**   * На естественном языке (словесная); * На языке формул; * На спец.языке (f.e.на языке прогр-я); * В граф.форме (блок-схемы) |

Графическое описание алгоритма

Представление алгоритма в виде схемы, состоящей из последовательности блоков (гeометрических фигур), каждый из которых отображает содержание очередного шага алгоритма, называется графическим описанием алгоритма.

Внутри фигур кратко записывают действие, выполняемое в этом блоке. Такую схему называют блок-схемой алгоритма.

Правила изображения фигур сведены в единой документации (ГОСТ 19.701-90). По данному ГОСТу - это схема данных.

Схема данных содержит:

* символы данных;
* символы процесса, который нужно выполнить над данными;
* символы линий или стрелок;
* специальные символы, которые используются для удобства чтения схемы.

Кроме словесного (на естественном языке) и графического описания алгоритма также иногда используют:

* псевдокод
* алгоритмический язык - запись на конкретном формальном языке (программа)

Псевдокод - представляет собой систему обозначений правил, предназначенную для единообразной записи алгоритмов. Псевдокод занимает промежуточное место между естественными и формальными языками.

**Алгоритм** – это точное предписание о выполнении в определенном порядке некоторых операций, приводящих к решению всех задач данного класса.

**Свойства** **алгоритмов**

- Дискретность ( алгоритм представляется как последовательность инструкций исполнителя.)

- Конечность (алгоритм должен заканчиваться после выполнения конечного числа инструкций)

- Массовость

- Детерминированность (определенность)(получение однозначного результата ) (каждый шаг алгоритма должен быть точно определен – записан на формальном языке исполнителя.)

- Результативность

///////////////////////////////////////////////////////////////////////////////////////

Важнейшей характеристикой алгоритма и соответствующей ему программы является их **сложность**, которая может оцениваться:

* временем решения задачи (трудоемкостью алгоритма - количеством выполняемых элементарных операций) – показывает, насколько быстро работает алгоритм – это временная эффективность;
* требуемой емкостью памяти – отражает максимальное количество памяти, требуемой для выполнения алгоритма – это пространственная эффективность.

Эти характеристики позволяют оценивать потребности алгоритма в вычислительных ресурсах: процессорном времени, памяти, пропускной способности сети и сравнивать алгоритмы между собой.

В общем случае сложность алгоритма можно оценить по порядку величины. Этот метод применим как к временной, так и к емкостной сложности.

**(2)Показатели эффективности алгоритмов: (курносов)**

 Количество выполняемых операций – временная эффективность (time efficiency), показывает на сколько быстро работает алгоритм

 Объем потребляемой памяти – пространственная эффективность (space efficiency), отражает максимальное количество памяти требуемой для выполнения алгоритма

 Показатели эффективности позволяют:

*  Оценивать потребности алгоритма в вычислительных ресурсах: процессорном времени, памяти, пропускной способности сети
*  Сравнивать алгоритмы между собой

В общем случае сложность алгоритма можно оценить по порядку величины. Этот метод применим как к временной, так и к емкостной сложности.

**Анализ времени выполнения алгоритмов**

Что влияет на время выполнения алгоритма (программы)?

1. Размер входных данных

2. Качество реализации алгоритма на языке программирования

3. Качество скомпилированного кода

4. Производительность вычислительной машины

Для большинства алгоритмов количество выполняемых операций напрямую зависит от размера входных данных (например, время выполнения алгоритма поиска максимального элемента зависит от длины массива, а не от значений в нем).

У каждого алгоритма есть параметры, определяющие размер его входных данных:

 Поиск наименьшего элемента в массиве: n – количество элементов в массиве

 Алгоритм умножения двух матриц: количества строк m и столбцов n в матрицах

 Сравнения двух строк: s1, s2 – длина первой и второй строк

 Поиск кратчайшего пути в графе между парой вершин: n, m – количество вершин и ребер в графе

Количество операций алгоритма

 Количество операций алгоритма можно выразить как функцию от размера его входных данных: T(n), T(s1, s2), T(n, m)

 В качестве исполнителя будем использовать модель однопроцессорной вычислительной машины с произвольным доступом к памяти (Random Access Machine – RAM):

*  Машина обладает неограниченной памятью
*  Для выполнения арифметических и логических операций (+, ‒, \*, /, %) требуется один временной шаг – такт процессора
*  Обращение к оперативной памяти для чтения или записи занимает один временной шаг
*  Выполнение условного перехода (if-then-else) требует вычисления логического выражения и выполнения одной из ветвей if then-else
*  Выполнение цикла (for, while, do) подразумевает выполнение всех его итераций

Классификация алгоритмов по виду функции трудоемкости:

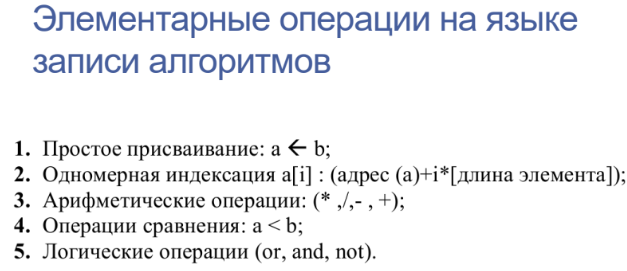
1. **Количественно-зависимые** по трудоемкости алгоритмы. Их функция трудоемкости зависит только от размерности входных данных и не зависит от их значений. Пример: умножение матриц, умножение матрицы на вектор и т.д.
2. **Параметрически-зависимые** по трудоемкости алгоритмы. Их функция трудоемкости определяется конкретными значениями обрабатываемых слов памяти. Пример: алгоритмы вычисления стандартных функций с заданной точностью путем вычисления соответствующих степенных рядов.
3. **Количественно-параметрические** по трудоемкости алгоритмы. Функция трудоемкости зависит и от количества данных на входе, так и от их значений. Пример: алгоритмы численных методов, в которых существует параметрически-зависимый цикл по точности и цикл, количественно-зависимый по размерности.

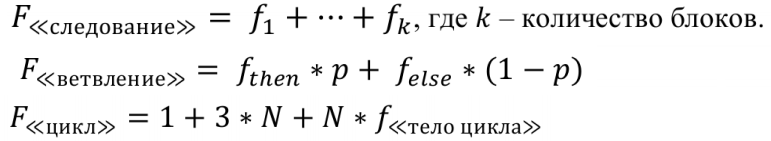
**Оценки**

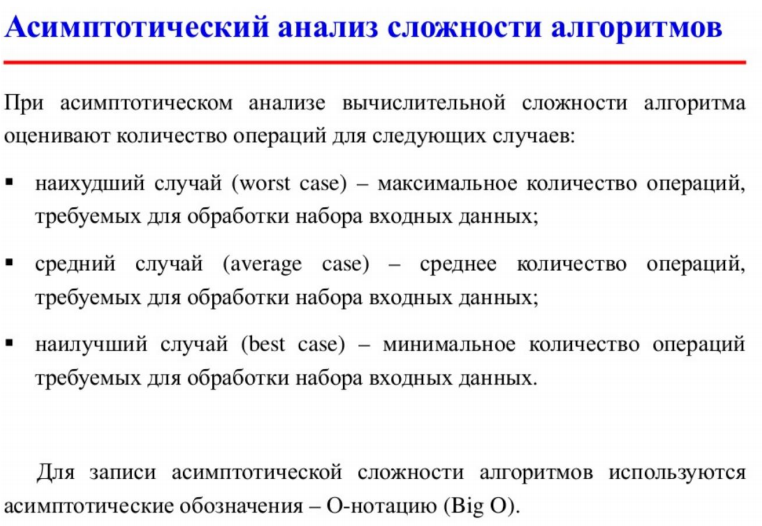
временная оценка

• Эта оценка считается наиболее существенной. Для определения трудоемкости алгоритма обычно используют следующий набор «элементарных» операций, характерных для языков высокого уровня.

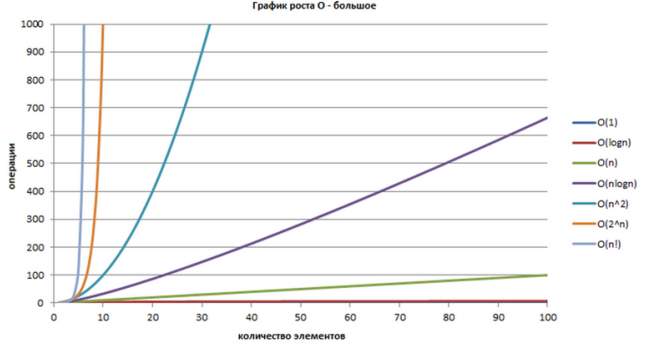
T(n) – **временная** **оценка** сложности **алгоритма**. Самый простой способ **оценки** сложности **алгоритма** – экспериментальный, т.е. запрограммировать **алгоритм** и выполнять полученную программу на нескольких задачах, оценивая время выполнения программы.

****

****

****

****

****

Оценка пространственной эффективности:

• Такая оценка выполняется аналогично временной.

• При этом если для реализации алгоритма требуется дополнительная память для одной, двух или трех переменных, то емкостная сложность алгоритма будет константной, т.е. О(1). Такую сложность имеет, например, рассмотренный выше алгоритм поиска максимума. Если дополнительная память пропорциональна n, то пространственная сложность равна О(n) и т.д.

**Сложность пузырька:**

****

**Лекция 2**

**РЕКУРСИВНЫЕ АЛГОРИТМЫ**

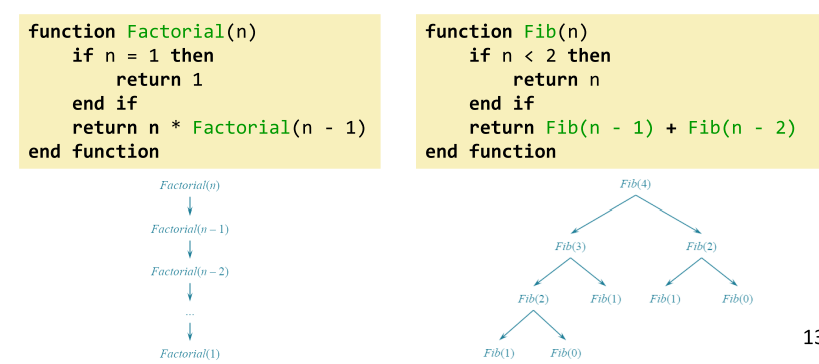
**Рекурсивная функция (**recursive function) – функция, в теле которой присутствует вызов самой себя. Алгоритм, основанный на таких функциях, называется рекурсивным алгоритмом.

**Системный стек (stack) –** память, предназначенная для хранения адресов возврата из функций, локальных переменных и передачи аргументов в функции.

* Рекурсивные функции могут занимать значительную часть стековой памяти для хранения адресов возврата!
* Стек имеет конечный размер: $ ulimit –s 8192

**Виды рекурсии:**

* - Линейная рекурсия – в функции присутствует единственный рекурсивный вызов самой себя
* - Древовидная рекурсия – в функции присутствует несколько рекурсивных вызовов



**Лекция 2 часть сортировок Лекция 3**

Сортировка слиянием (merge sort) – асимптотически оптимальный алгоритм сортировки сравнением, основанный на методе декомпозиции («разделяй и властвуй», decomposition).

Алгоритм включает две фазы:

1) Разделение (partition) – рекурсивное разбиение массива на меньшие подмассивы, их сортировка

2) Слияние (merge) – объединение упорядоченных массивов в один

Фаза разделения: Подмассив A[low, high] делится на две части:

* A[low..mid] и A[mid + 1, high]
* mid = floor((low + high) / 2)

Фаза слияния:

* Функция Merge сливает упорядоченные подмассивы A[low..mid] и A[mid+1..high] в один отсортированный массив, элементы которого занимают позиции A[low..high]
* Сортируемый массив A[low..high] разделяется (partition) на две максимально равные по длине части
*  Первая часть содержит n/2 элементов, вторая: n/2 элементов
*  Подмассивы рекурсивно сортируются

Функция Merge требует порядка Θ(n) ячеек памяти для хранения копии B сортируемого массива

 Сравнение и перенос элементов из массива B в массив A требует Θ(n)

* На каждом уровне i находится 2^i узлов
* Каждый узел требует выполнения n/2^I операций

Просматриваем элементы, начиная с первого, и сравниваем ключи

*  В худшем случае искомый элемент находится в конце массива или отсутствует
*  Количество операций в худшем случае (worst case) T(n) = O(n)

Бинарный поиск (Binary Search)

 Имеется упорядоченная последовательность ключей

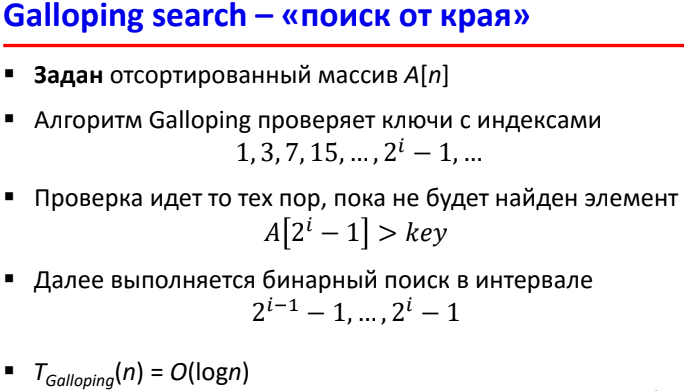
 Бинарный поиск (Binary search)

1. Если центральный элемент равен искомому, конец

2. Если центральный меньше, делаем текущей правую половину массива

3. Если центральный больше, делаем текущей левую половину массива

Бинарный поиск неэффективно использует кеш-память процессора – доступ к элементам массива непоследовательный (прыжки по массиву).

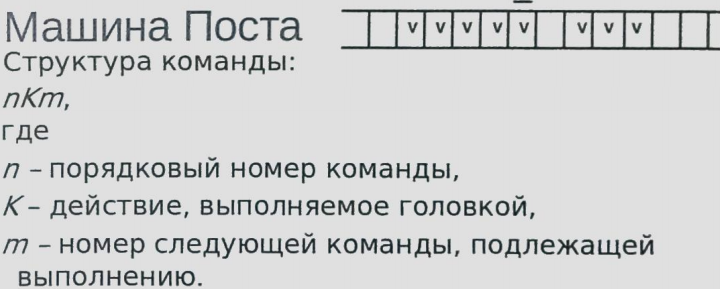
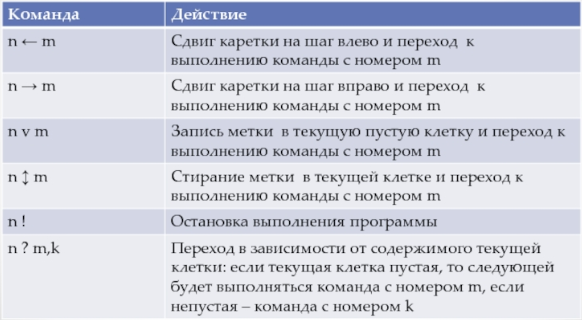


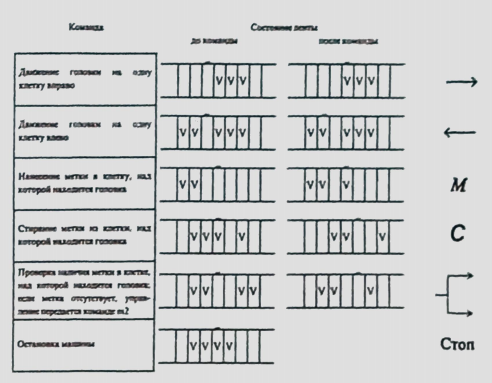
**Машина Поста, НАМ, ТЬюринг**

Машина Поста – это абстрактная (несуществующая реально) вычислительная машина, созданная для уточнения (формализации) понятия алгоритма.

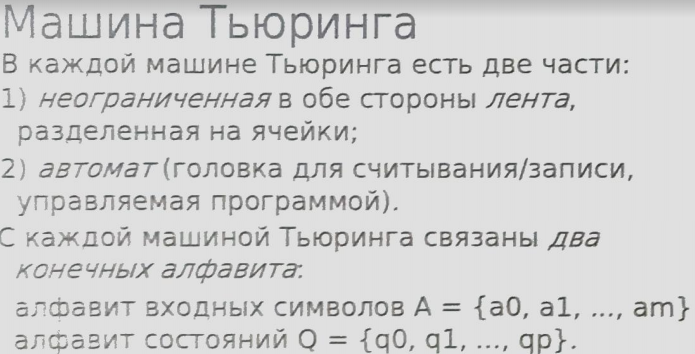
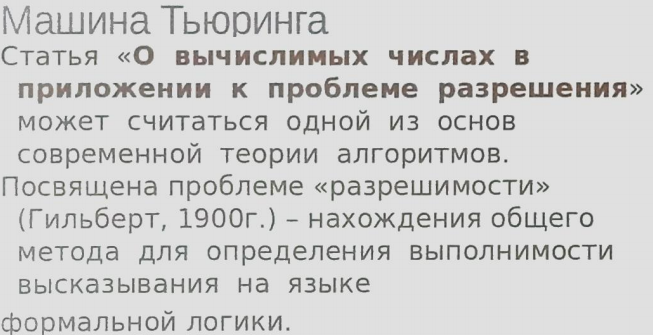
Машина Поста состоит из …

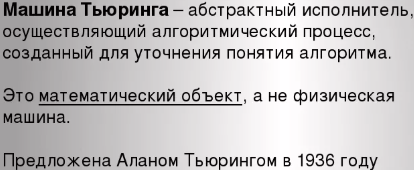
* бесконечной ленты, поделенной на одинаковые ячейки (секции). Ячейка может быть пустой (0 или пустота) или содержать метку (1 или любой другой знак),
* головки (каретки), способной передвигаться по ленте на одну ячейку в ту или иную сторону, а также способной проверять наличие метки, стирать и записывать метку

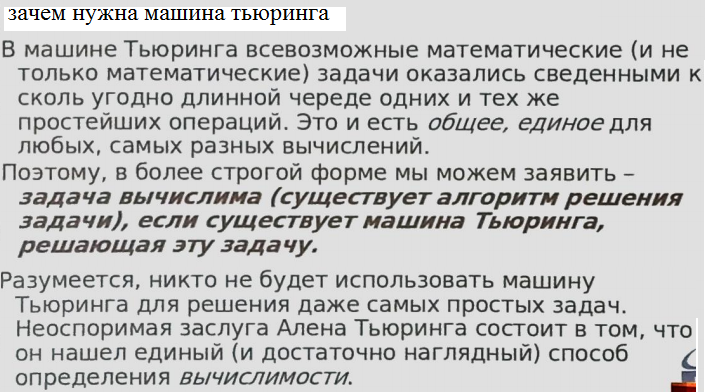
****



**Машина Тьюринга:**

****

****

****

# Машина Тьюринга

В 1936 г. Аланом Тьюрингом для уточнения понятия алгоритма был предложен **абстрактный универсальный исполнитель**. Его абстрактность заключается в том, что он представляет собой логическую вычислительную конструкцию, а не реальную вычислительную машину. Термин «универсальный исполнитель» говорит о том, что данный исполнитель может имитировать любой другой исполнитель. Например, операции, которые выполняют реальные вычислительные машины можно имитировать на универсальном исполнителе. В последствие, придуманная Тьюрингом вычислительная конструкция была названа **машиной Тьюринга**.  
**Что собой представляет машина Тьюринга?**

Машина Тьюринга состоит из бесконечной в обе стороны ленты, разделенной на ячейки, и автомата (головки), которая управляется программой.  
Программы для машин Тьюринга записываются в виде таблицы, где первые столбец и строка содержат буквы внешнего алфавита и возможные внутренние состояния автомата (внутренний алфавит). Содержимое таблицы представляет собой команды для машины Тьюринга. Команда определяется пересечением символов внешнего и внутреннего алфавитов в таблице.

**Чтобы задать конкретную машину Тьюринга, требуется описать для нее следующие составляющие:**

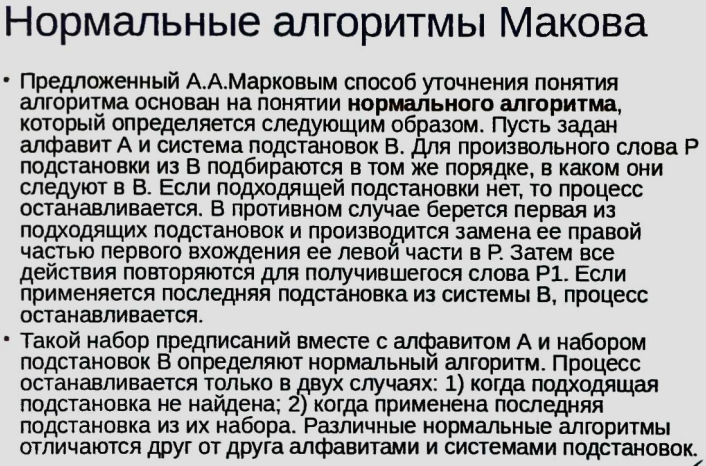
* **Внешний алфавит.** Конечное множество (например, А), элементы которого называются буквами (символами). Одна из букв этого алфавита (например, а0) должна представлять собой пустой символ.
* **Внутренний алфавит.** Конечное множество состояний головки (автомата). Одно из состояний (например, q1) должно быть начальным. Еще одно из состояний (q0) должно быть конечным – состояние останова.
* **Таблица переходов.** Описание поведения автомата (головки) в зависимости от состояния и считанного символа.

**Автомат машины Тьюринга в процессе своей работы может выполнять следующие действия:**

* Записывать символ внешнего алфавита в ячейку (в том числе и пустой), заменяя находившийся в ней (в том числе и пустой).
* Передвигаться на одну ячейку влево или вправо.
* Менять свое внутреннее состояние.

///////

**Нормальные алгоритмы Маркова**. Теория нормальных алгоритмов была разработана советским математиком А. А. Марковым в конце 1940-х — начале 1950-х гг. Эти алгоритмы представляют собой некоторые правила по переработке слов в алфавите, так что и исходные данные, и результаты работы алгоритмов являются словами в этом алфавите

****

**Понятие типа данных (data type)**

 Язык программирования позволяет оперировать с величинами различного вида: строки, числа, логические значения

 Тип данных (data type) – это атрибут любого значения, которое встречается в программе

 Тип данных определяет две характеристики:

*  множество допустимых значений, которые могут принимать данные, принадлежащие к этому типу
*  набор операций, которые можно выполнять над данными этого типа

Типы данных (data type)

*  Базовые (примитивные) типы данных: int, char, bool, string
*  Cоставные типы данных (агрегатные, структурные, композитные типы) – это типы данных, которые формируются на основе базовых (например, массивы, структуры и классы в языках С и C++)
*  Структура данных (data structure) – программная единица, реализующая хранение и выполнение операций над совокупностью однотипных элементов
*  Набор функций для выполнения операций над структурой данных называется её интерфейсом (interface)

**Абстрактные типы данных**

 Абстрактный тип данных (АТД, abstract data type) – это тип данных, который задан описанием своего интерфейса.

 Способ хранения данных в памяти компьютера и алгоритмы работы с ними сокрыты внутри функций интерфейса (в этом суть абстракции).

 Если описать реализацию функций АТД, получим конкретную структуру данных.

Абстрактный тип данных «список»:

 Интерфейс АТД список (list)

 Insert

 Delete

 Lookup

 Next

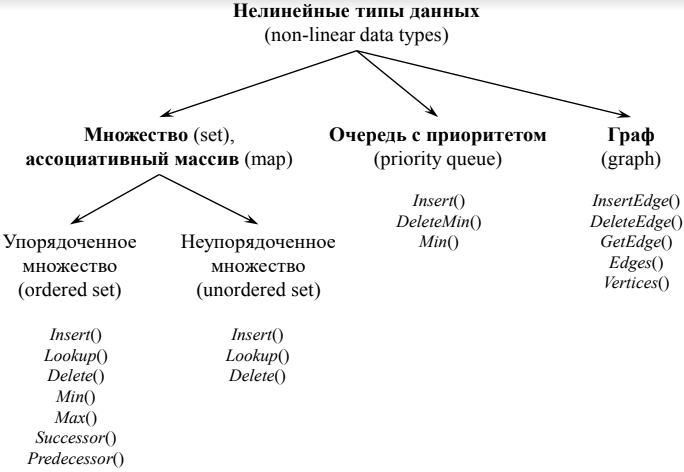
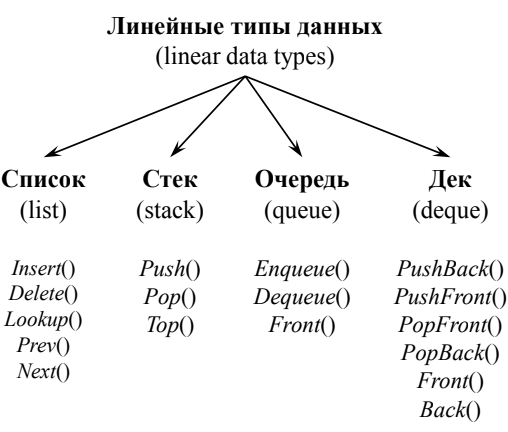
 Prev

Возможные реализации АТД список

 Статический массив

 Связный список

Вычислительная сложность и сложность по памяти функций разных реализаций могут быть различными.



**Связные списки**

Связный список (linked list) – динамическая структура данных для хранения информации, в которой каждый элемент хранит указатели на один или несколько других элементов.



 Размер списка заранее не известен – элементы добавляются во время работы программы (динамически)

 Память под элементы выделяется динамически (функции: malloc, calloc, free)

Поиск элемента в списке (Lookup)

 Начиная с головы списка просматриваем все узлы и сравниваем ключи

 В худшем случае требуется просмотреть все узлы, это требует O(n) операций

**Двусвязные списки**

 Каждый узел двусвязного списка имеет три поля: value, next и prev.

 Поле value – это некоторые данные, ассоциированные с узлом

 В поле next хранится адрес узла, следующего за текущим, а в поле prev – адрес предшествующего узла

Добавление узла в начало списка:

 Функция создает в памяти новый узел с заданным значением поля value

 В поле next нового узла заносится адрес головы списка

 Если список не пуст, то необходим записать в указатель prev первого узла адрес нового элемента

**Стек**

 Стек (Stack) – структура данных для хранения элементов

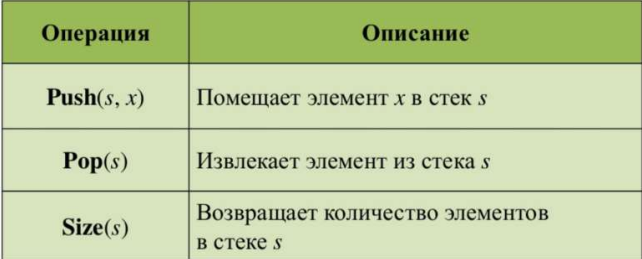
 Дисциплина доступа к элементам: “последним пришел – первым вышел” (Last In – First Out, LIFO)

 Элементы помещаются и извлекаются с головы стека (top)

Подходы к реализации стека:

1. На основе связных списков (linked lists). Длина стека ограничена объемом доступной памяти

2. На основе статических массивов. Длина стека фиксирована (задана его максимальная длина – количество элементов в массиве)



Реализация стека на основе связных списков:

*  Элементы стека хранятся в односвязном списке (singly linked list)
*  Добавление элемент и удаление выполняется за время O(1)

Стек состоит из ячеек (в примере — это книги), которые представлены в виде структуры, содержащей какие-либо данные и указатель типа данной структуры на следующий элемент.

При добавлении элемента у нас возникнет две ситуации:

•Стек пуст, и нужно создать его

•Стек уже есть и нужно лишь добавить в него новый элемент

Реализация стека на основе массива:

*  Элементы стека хранятся в массиве фиксированной длины L
*  Добавление элемент и удаление выполняется за время O(1)

**Очередь (Queue)**

 Очередь (Queue) – структура данных для хранения элементов (контейнер)

 Дисциплина доступа к элементам: “первым пришел – первым вышел” (First In – First Out, FIFO)

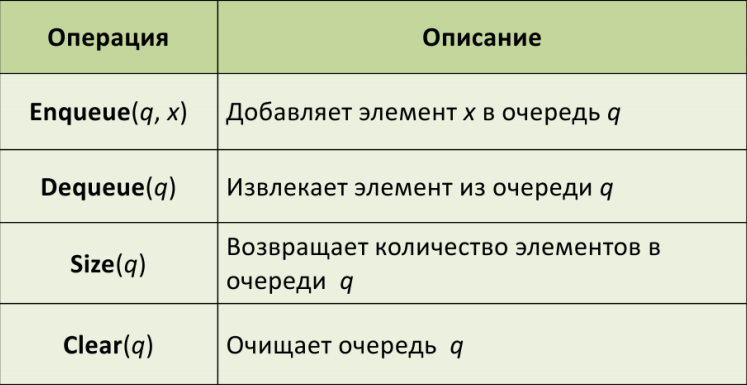
 Элементы добавляются в хвост (tail), извлекаются с головы (head)

 Очереди широко используется в алгоритмах обработки данных:

*  очереди печати
*  буфер ввода с клавиатуры
*  алгоритмы работы с графами

Примеры применения очередей:

* Наиболее известные примеры применения очередей в программировании – очередь событий системы Windows и ей подобных.
* Очереди используются также для моделирования в задачах массового обслуживания (например, обслуживания клиентов в банке).



Подходы к реализации очереди:

1. На основе связных списков. Длина очереди ограничена лишь объемом доступной памяти.

2. На основе статических массивов. Длина очереди фиксирована (задана максимальная длина).

Реализация очереди на основе связных списков:

 Элементы очереди хранятся в односвязном списке (singly linked list)

 Для быстрого (за время O(1)) добавления и извлечения элементов из списка поддерживается указатель на последний элемент (tail)

 Новые элементы добавляются в конец списка

+/-:

 Преимущества: длина очереди ограничена лишь объемом доступной памяти

 Недостатки (по сравнению с реализацией на основе массивов): работа с очередью немного медленнее, требуется больше памяти для хранения одного элемента

Реализация очереди на основе циклических массивов:

• Если у стека один конец «закреплен» (не двигается), то у очереди «подвижны» оба конца. Чтобы не сдвигать все элементы в массиве при удалении или добавлении элемента, обычно используют две переменные head и tail – первая из них обозначает номер первого элемента в очереди, а вторая – номер последнего. Если они равны, то в очереди всего один элемент. Массив как бы замыкается в кольцо – если массив закончился, но в начале массива есть свободные места, то новый элемент добавляется в начало массива, как показано на рисунках.

**Дек**

• Дек (deque) – это упорядоченный набор элементов, в котором добавление новых и удаление существующих элементов допустимо с любого конца.

• Дек может быть реализован на основе массива или двусвязного списка. Для дека разрешены четыре операции:

1) добавление элемента в начало;

2) добавление элемента в конец;

3) удаление элемента сначала;

4) удаление элемента с конца.

Их можно реализовать, используя написанные выше функции для стека и очереди.

**Бинарные деревья поиска**

АТД «Словарь» (dictionary)

 Словарь (dictionary) – структура данных для хранения пар вида «ключ» – «значение» (key – value)

 Альтернативные название – ассоциативный массив (associative array, map)

 В словаре может быть только одна пара с заданным ключом



Реализация АТД «Словарь»

 Реализации словарей отличаются вычислительной сложностью операций и объемом требуемой памяти для хранения пар «ключ-значение»

 Распространение получили следующие реализации:

1. Деревья поиска (Search trees)

2. Хэш-таблицы (Hash tables)

3. Списки с пропусками (Skip lists)

4. Связные списки, массивы



**Бинарное дерево** – это дерево (структура данных), в котором каждый узел (node) имеет не более двух дочерних узлов

**Бинарные деревья поиска (binary search trees)**

 Двоичное дерево поиска (binary search tree, BST) – это двоичное дерево, в котором:

1) каждый узел x (node) имеет не более двух дочерних узлов (child nodes) и содержит ключ (key) и значение (value)

2) ключи всех узлов левого поддерева узла x меньше значения его ключа

3) ключи всех узлов правого поддерева узла x больше значения его ключа

* Упорядоченный словарь (ordered map) – словарь, обеспечивающий перебор элементов в упорядоченной последовательности
* Операции Prev(key), Next(key)

 При добавлении элемента необходимо спуститься от корня дерева до листа – это требует количества операций порядка высоты h дерева

 Поиск листа – O(h), создание элемента и корректировка указателей – O(1)

Поиск элемента в BST:

1. Сравниваем ключ корневого узла с искомым. Если совпали, то элемент найден.

2. Переходим к левому или правому дочернему узлу и повторяем шаг 1.

Возможны рекурсивная и не рекурсивная реализации.

Поиск минимального элемента в BST:

 Минимальный элемент всегда расположен в левом поддереве корневого узла

 Требуется найти самого левого потомка корневого узла

Поиск максимального элемента в BST:

 Максимальный элемент всегда расположен в правом поддереве корневого узла

 Требуется найти самого правого потомка корневого узла

Удаление элемента из BST:

1. Находим узел z с заданным ключом – O(n)

2. Возможны 3 ситуации:

1. узел z не имеет дочерних узлов
2. узел z имеет 1 дочерний узел
3. узел z имеет 2 дочерних узла

Анализ эффективность BST:

1. Операции имеют трудоемкость пропорциональную высоте h дерева

2. В худшем случае высота дерева O(n) (вставка элементов в отсортированной последовательности)

3. В среднем случае высота дерева O(logn)



**Сбалансированные деревья поиска**

 Сбалансированное по высоте дерево поиска (self-balancing binary search tree) – дерево поиска, в котором высоты поддеревьев узла различаются не более чем на заданную константу k.

 Баланс высоты поддерживается при выполнении операций добавления и удаления элементов.

 Типы сбалансированных деревьев поиска:

*  Красно-черные деревья (Red-black tree): h ≤ 2 log2(n + 1)
*  АВЛ-деревья (AVL-tree): h < 1.4405 ∙ log2(n + 2) − 0.3277
*  B-деревья
*  Деревья Ван Эмде Боаса и тд

Все операции на красно-черном дереве и АВЛ-дереве в худшем случае выполняются за время O(logn)

**Хэш-таблицы**

АТД «Словарь» (dictionary)

 Словарь (ассоциативный массив) – структура данных (контейнер) для хранения пар вида «ключ – значение» (key – value)

 Реализации словарей отличаются вычислительной сложностью операций добавления, поиска и удаления элементов

 Наибольшее распространение получили следующие реализации:

1. Деревья поиска (search trees)

2. Хэш-таблицы (hash tables)

3. Связные списки

4. Массивы

В словаре не могут присутствовать пары с одинаковыми ключами.

**Хеш-таблицы (Hash tables)**

** Хеш-таблица (hash table) – структура данных для хранения пар «ключ – значение»**

 Доступ к элементам осуществляется по ключу (key)

 Ключи могут быть строками, числами, указателями, ...

 Хеш-таблицы позволяют в среднем за время О(1) выполнять добавление, поиски и удаление элементов

Основная идея:

 Чем хороши статические массивы int v[100]?

 Быстрый доступ O(1) к элементу массива по его ключу (индексу): v[17] = 45

 Ограничение – ключи (индексы) это целые числа

 Можно ли как-то использовать типы float, double, строки (char []) в качестве индексов в массиве?

 Пример: массив анкет пользователей Twitter (словарь), ключ – login, значение – анкета (профиль с данными пользователя)

 Массив структур: struct twitter\_user users[MAX\_USERS];

 Хеш-функция – отображает (преобразует) ключ (key) в номер элемента (index) массива (в целое число от 0 до h – 1)

 Время вычисления хеш-функции зависит от длины ключа и не зависит от количества элементов в массиве

 Ячейки массива называются buckets, slots

 На практике, обычно известна информация о диапазоне значений ключей

 На основе этого выбирается размер h хеш-таблицы и выбирается хеш-функция

 Коэффициент АЛЬФА заполнения хеш-таблицы (load factor, fill factor) – отношение числа n хранимых элементов в хеш-таблицы к размеру h массива (среднее число элементов на одну ячейку): α =n/h

 Пример: h = 128, в хеш-таблицу добавили 50 элементов, тогда  = 50 / 128 == 0.39

 От этого коэффициента зависит среднее время выполнения операций добавления, поиска и удаления элементов

 Хеш-функция (Hash function) – это функция, преобразующая значения ключа (например: строки, числа, файла) в целое число.

 Значение, возвращаемое хеш-функцией, называется хеш-кодом (hash code), контрольной суммой (hash sum) или хешем (hash)

unsigned int hash(char \*s) {

unsigned int h = 0;

char \*p;

for (p = s; \*p != '\0'; p++)

**h = h \* HASH\_MUL + (unsigned int)\*p;**

return h % HASH\_SIZE;

}

Thash = O(|s|)

h = 105 \* HASH\_MUL + 118

Требования к хеш-функциям:

 Быстрое вычисление хэш-кода по значению ключа

 Сложность вычисления хэш-кода не должна зависеть от количества n элементов в хеш-таблице

 Детерминированность – для заданного значения ключа хэш-функция всегда должна возвращать одно и то же значение

 Равномерность – хеш-функция должна равномерно заполнять индексы массива возвращаемыми номерами

 Желательно, чтобы все хэш-коды формировались с одинаковой равномерной распределенной вероятностью

**Коллизии**

**Коллизия** (Collision) – это совпадение значений хеш-функции для двух разных ключей.

Существуют хеш-функции без коллизий – совершенные хеш-функции (perfect hash function).

Метод цепочек (Chaining) – закрытая адресация.

Элементы с одинаковым значением хеш-функции объединяются в связный список. Указатель на список хранится в советующей ячейке хеш-таблицы.

 При коллизии элемент добавляется в начало списка.

 Поиск и удаление элемента требуют просмотра всего списка.

Открытая адресация (Open addressing).

В каждой ячейке хеш-таблицы хранится не указатель на связный список, а один элемент (ключ, значение).

Если ячейка с индексом hash(key) занята, то осуществляется поиск свободной ячейки в следующих позициях таблицы.

Линейное хеширование (linear probing) – проверяются позиции:

hash(key) + 1, hash(key) + 2, ..., (hash(key) + i) mod h, ...

Если свободных ячеек нет, то таблица заполнена.

Пример:

 hash(D) = 3, но ячейка с индексом 3 занята

 Просматриваем ячейки: 4 – занята, 5 – свободна

Хеш-таблица требует предварительной инициализации ячеек значениями NULL – трудоемкость O(h)



Пример хэш-функции для чисел:

 Ключи – размер файла (int)

 Значение, хранимое в словаре – название файла

 Требуется разработать хеш-функцию

Хеш-таблицы (Hash table):

 Длину h хеш-таблицы выбирают как простое число

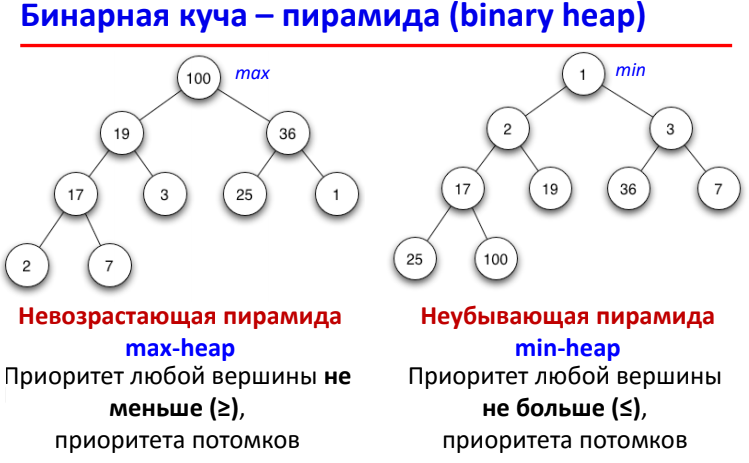
 Для такой таблицы модульная хеш-функция дает равномерное распределение значений ключей

hash(key) = key % h

**Бинарные кучи**

**Бинарная куча** (пирамида, сортирующее дерево, binary heap) – это двоичное дерево, удовлетворяющее следующим условиям:

* Приоритет любой вершины не меньше приоритета ее потомков.
* Дерево является полным двоичным деревом – все уровни заполнены слева направо (возможно, за исключением последнего)

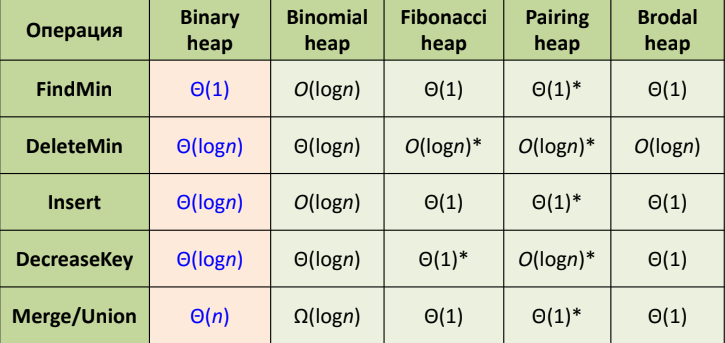


Сортировка на базе бинарной кучи:

 На основе бинарной кучи можно реализовать алгоритм сортировки с вычислительной сложностью O(nlogn) в худшем случае

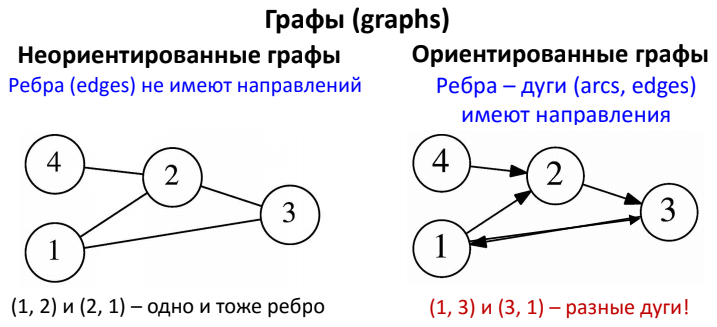
В таблице приведены трудоемкости операций очереди с приоритетом (в худшем случае, worst case)

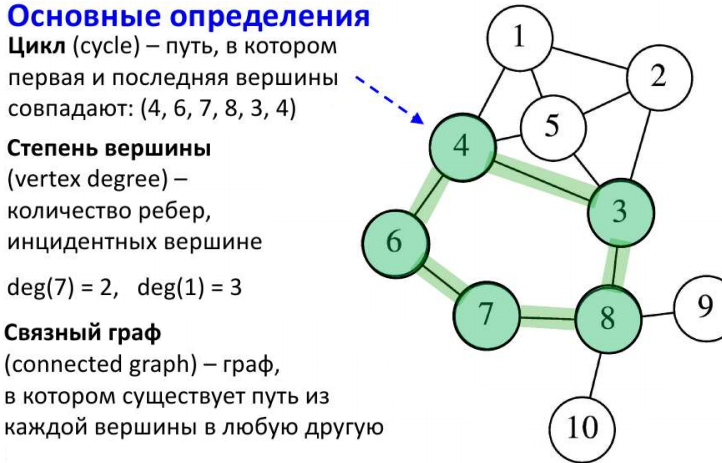
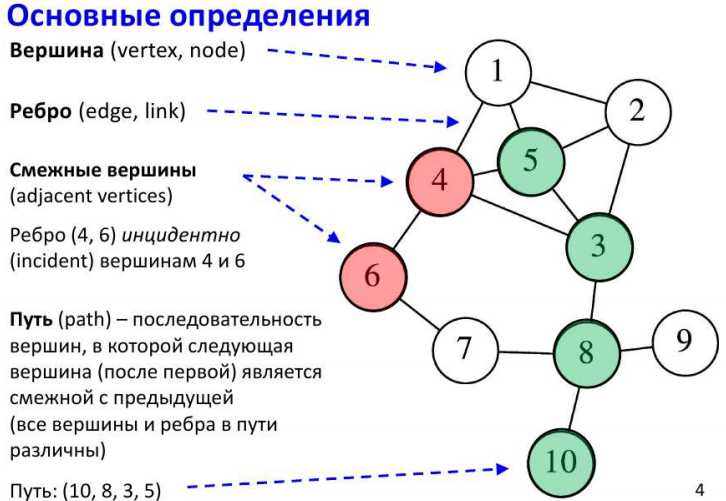
 Символом ‘\*’ отмечена амортизированная сложность операций



**Графы**

Граф (graph) – это совокупность непустого множества V вершин и множества E ребер.





Основные определения:

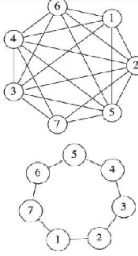
 Взвешенный граф (weighted graph) – это граф, ребрами (дугам) которого назначены веса

 Вес ребра (i,j) обозначим как wij

5

Вес 5

**Полный граф** – это граф, в котором каждая пара различных вершин смежна (каждая вершина соединена со всеми)

Количество ребер в полном неориентированном графе: m = n(n − 1)/2

Насыщенность D графа (density): D = 2m/n(n − 1)

У полного графа насыщенность D = 1

Насыщенный граф (dense graph) – это граф, в котором количество ребер близко к максимально возможному.

Разреженный граф (sparse graph) – граф, в котором количество ребер близко к количеству вершин в графе.

Представление графов в памяти:

 Представление графа в памяти определяет вычислительную сложность операций над графом и объем требуемой памяти.

 Основные способы представления графов в памяти:

*  Матрица смежности (adjacency matrix) – эффективна для насыщенных графов
*  Списки смежных вершин (adjacency list) – эффективен для разреженных графов

Матрица A смежности (adjacency matrix) – это матрица n × n элементов, в которой aij = 1, если i,j ∈ E, 0, иначе.

 Объем требуемой памяти O( V2)

 Быстрое определение присутствия ребра (i,j) в графе

 За время O(1) получаем доступ к элементу aij матрицы

Списки смежных вершин:

 Списки смежных вершин – это массив A[n], каждый элемент A[i] которого содержит список узлов смежных с вершиной i.

 **Обход графа** (graph traversal) – это процедура перебора (посещения) всех вершин графа, начиная с заданной.

** Поиск в глубину** (depth-first search – DFS) – процедура посещения всех вершин графа начиная с заданного узла v

 Сперва посещаем (обрабатываем) все самые “глубокие” вершины

1.  Обход в глубину графа, представленного матрицей смежности, имеет трудоемкость O(|V|2)
2.  Обход в глубину графа, представленного списком смежности, имеет трудоемкость O(|V| + |E|)

** Поиск в ширину** (BFS, обход в ширину) – процедура посещения всех вершин графа начиная с заданного узла v

 Сперва посещаем (обрабатываем) свои дочерние вершины

1.  Обход в ширину графа, представленного матрицей смежности, имеет трудоемкость O(|V|2)
2.  Обход в ширину графа, представленного списком смежности, имеет трудоемкость O(|V| + |E|)

**Длина пути** (path length, path cost, path weight) – это сумма весов ребер, входящих в него

Постановки задачи о кратчайшем пути:

*  Задача о кратчайшем пути между парой вершин (single-pair shortest path problem)

Требуется найти кратчайший путь из заданной вершины s в заданную вершину d

*  Задача о кратчайших путях из заданной вершины во все (single-source shortest path problem)

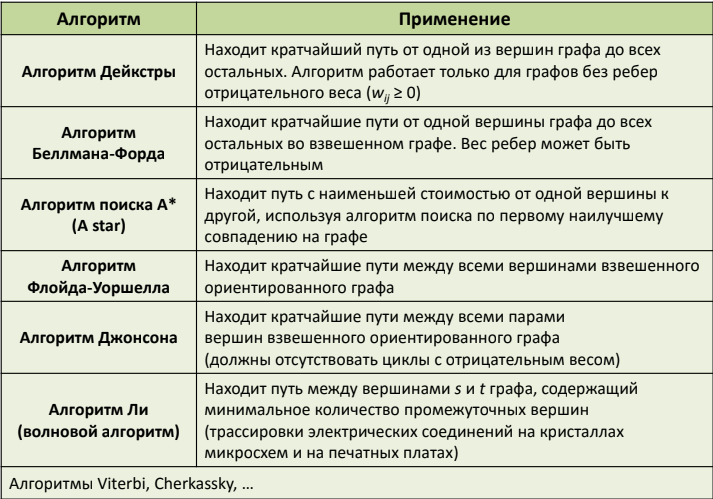
Найти кратчайшие пути из заданной вершины s во все

*  Задача о кратчайшем пути в заданный пункт назначения (single-destination shortest path problem)

Требуется найти кратчайшие пути в заданную вершину v из всех вершин графа

*  Задача о кратчайшем пути между всеми парами вершин (all-pairs shortest path problem)

Требуется найти кратчайший путь из каждой вершины u в каждую вершину v



**Алгоритм Дейкстры**

 Алгоритм Дейкстры (1959) – алгоритм поиска кратчайшего пути в графе из заданной вершины во все остальные.

 Находит кратчайшее расстояние от одной из вершин графа до всех остальных

 Применим только для графов без ребер отрицательного веса и петель (wij ≥ 0)

 Эдсгер Дейкстра (Edsger Wybe Dijkstra) – нидерландский ученый (структурное программирование, язык Алгол, семафоры, распределенные вычисления)

 Лауреат премии Тьюринга (ACM A.M. Turing Award)

 **Вычислительная сложность** алгоритма Дейкстры определяется следующими факторами:

1) выбором структуры данных для хранения графа (матрица смежности, списки смежных вершин)

2) способом поиска вершины с минимальным расстоянием D[i]:

*  Очередь с приоритетом: Binary heap – O(logn), Fibonacchi heap, ...
*  Сбалансированное дерево поиска: Red-black tree – O(logn), AVL-tree, ...
*  Линейный поиск – O(n)

**Методы разработки алгоритмов**

Основные методы разработки алгоритмов:

 Метод грубой силы (brute force, исчерпывающий поиск – полный перебор)

 Декомпозиция (decomposition, “разделяй и властвуй”)

 Уменьшение размера задачи (“уменьшай и властвуй”)

 Преобразование (“преобразуй и властвуй”)

 Жадные алгоритмы (greedy algorithms)

 Динамическое программирование (dynamic programming)

 Поиск с возвратом (backtracking)

 Локальный поиск (local search)

-------

 Метод грубой силы (brute force) – решение “в лоб”

*  Основан на прямом подходе к решению задачи
*  Опирается на определения понятий, используемых в постановке задачи

Примеры алгоритмов, основанных на методе грубой силы:

*  Умножение матриц по определению O(n3)
*  Линейный поиск наибольшего/наименьшего элемента в списке
*  Сортировка выбором (Selection sort, O(n2))
*  Пузырьковая сортировка (Bubble sort, O(n2))
*  Поиск подстроки в строке методом грубой силы
*  Поиск перебором пары ближайших точек на плоскости

**Метод декомпозиции (Decomposition)**

Метод декомпозиции (decomposition method, метод “разделяй и властвуй” – “divide and conquer”)

 Структура алгоритмов, основанных на этом методе:

1. Задача разбивается на несколько меньших экземпляров той же задачи

2. Решаются сформированные меньшие экземпляры задачи (обычно рекурсивно)

3. При необходимости решение исходной задачи формируется как комбинация решений меньших экземпляров задачи

ПРИМЕР: Задача. Вычислить сумму чисел a0 , a1 , .., an – 1 .

Метод декомпозиции (Decomposition):

 В общем случае задача размера n делится на экземпляры задачи размера n / b, из которых a требуется решить (b > 1, a ≥ 0)

 Время T(n) работы алгоритмы, основанного на методе декомпозиции, равно T(n) = aT(n / b) + f(n), (\*), где f(n) – функция, учитывающая затраты времени на разделение задачи на экземпляры и комбинирование их решений

 Рекуррентное соотношение (\*) – это обобщённое рекуррентное уравнение декомпозиции.

Примеры алгоритмов, основанных на методе декомпозиции:

*  Сортировка слиянием (MergeSort)
*  Быстрая сортировка (QuickSort)
*  Бинарный поиск (Binary Search)
*  Обход бинарного дерева (Tree traverse)
*  Решение задачи о поиске пары ближайших точек
*  Решение задачи о поиске выпуклой оболочки
*  Умножение матриц алгоритмом Штрассена

**Динамическое программирование**

 **Динамическое программирование** – метод решения задач путем разбиения их на более простые подзадачи

 Решение задачи идет от простых подзадач к сложным, периодически используя ответы для уже решенных подзадач (как правило, через рекуррентные соотношения)

 Основная идея – запоминать решения встречающихся подзадач на случай, если та же подзадача встретится вновь

 Теория динамического программирования разработана Р. Беллманом в 1940 – 50-х годах

В динамическом программировании используются таблицы, в которых сохраняются решения подзадач (жертвуем памятью ради времени).

Жадные алгоритмы:

 “Жадный” алгоритм (Greedy algorithms) – алгоритм, принимающий на каждом шаге локально-оптимальное решение

 Предполагается, что конечное решение окажется оптимальным

 Примеры “жадных” алгоритмов:

* алгоритм Прима
* алгоритм Курскала
* алгоритм Дейкстры
* алгоритм Хаффмана (кодирования)

ПРИМЕР: Пример: Имеются монеты достоинством 1, 2, 5 и 10 рублей. Выдать сумму S = 27 рублей

♣ “Жадное” решение (алгоритм): 2 монеты по 10 руб., 1 по 5, 1 по 2

♣ На каждом шаге берётся наибольшее возможное количество монет достоинства an (от большего к меньшему)

**Код Хаффмана**

 Деревья Хаффмана (Huffman) и коды Хаффмана используются для сжатия информации путем кодирования часто встречающихся символов короткими последовательностями битов.

 Предложен Д. А. Хаффманом в 1952 году (США, MIT).

Код Хаффмана:

Задано множество символов и известны вероятности их появления в тексте (в файле).

Требуется каждому символу сопоставить код – последовательность битов.

Шаг 1. Создается n одноузловых деревьев

♣ В каждом узле записан символ алфавита и вероятность его повеления в тексте.

Шаг 2. Находим два дерева с наименьшими вероятностями и делаем их левым и правым поддеревьями нового дерева – создаем родительский узел.

♣ В созданном узле записываем сумму вероятностей поддеревьев.

♣ Повторяем шаг 2 пока не получим одно дерево.

На каждом шаге осуществляется “жадный выбор” – выбираем два узла с наименьшими вероятностями.

 Часто встречающиеся символы получили короткие коды

 Ни один код не является префиксом другого

**Поиск с возвратом (Backtracking)**

 **Поиск с возвратом** (Backtracking) – метод решения задач, в которых необходим полный перебор всех возможных вариантов в некотором множестве М.

 “Построить все возможные варианты ...”, “Сколько существует способов ...”, “Есть ли способ ...”.

 Термин Backtrack введен в 1950 г. D. H. Lehmer

 Примеры задач:

* Задача коммивояжёра
* Подбор пароля
* Задача о восьми ферзях
* Задача о ранце
* Раскраска карты

Локальный поиск (local search)

 Локальный поиск (Local search) – это метод приближенного решения оптимизационных задач.

 Жертвуем точностью решения для сокращения времени работы алгоритма.

 Примеры методов локального поиска:

* Имитация отжига (Simulated annealing)
* Генетические алгоритмы (Genetic algorithms)
* Поиск с запретами (Tabu search)

**АСИМПТОТИЧЕСКИЙ АНАЛИЗ**

Асимптотический анализ позволяет оценивать скорость роста функций T(n) при стремлении размера входных данных к бесконечности (при n → ∞).

Асимптотическая оценка времени считается наиболее существенной. Для определения трудоемкости алгоритма обычно используют следующий набор «элементарных» операций, характерных для языков высокого уровня (простое присваивание, арифметические, сравнения и логические операции – 1, одномерная индексация – 2).

С их помощью трудоемкость основных алгоритмических конструкций можно представить так:

1. **Линейная** конструкция из k последовательных блоков. Трудоемкость конструкции есть сумма трудоемкостей блоков, следующих друг за другом. 

2. конструкция **ветвление** 

3. конструкция **цикл** 

* *Лучший случай* – это экземпляр задачи (набор входных данных), на котором алгоритм выполняет наименьшее число операций
* *Худший случай* – это экземпляр задачи, на котором алгоритм выполняет наибольшее число операций
* *Средний случай* – это “средний” экземпляр задачи, набор “усреднённых” входных данных

**Верхняя оценка** сложности алгоритма f(n) обозначается O(n) (омикрон). Считается, что она пропорциональна максимальному элементу в формуле Θ(n)

**Нижняя оценка** сложности Ω(омега большое). Иначе говоря, 𝜔(𝑔(𝑛)) – множество всех функций, значения которых при больших 𝑛 больше значения 𝑐𝑔 (𝑛) для любых 𝑐.

**ПО-3**

**СОРТИРОВКИ**

Алгоритм сортировки не меняющий относительный порядок следования равных ключей называется **устойчивым**.

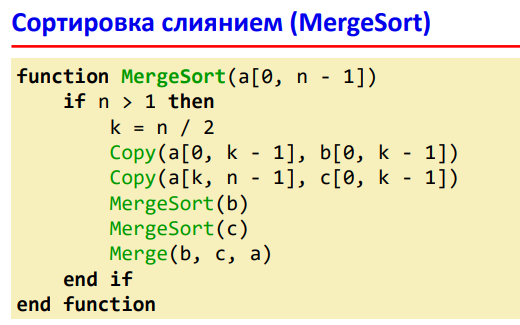
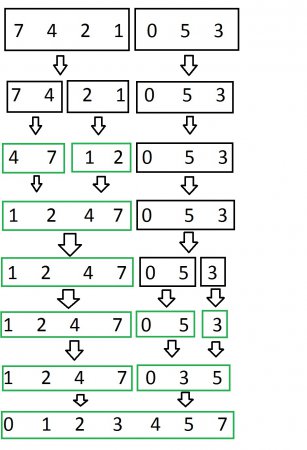
Внутренние методы сортировки – сортируемые элементы полностью размещены в оперативной памяти компьютера.

Внешняя сортировка – элементы размещены на внешней памяти (жесткий диск, USB-флеш).

Алгоритм сортировки не использующий дополнительной памяти (кроме сортируемого массива) называется алгоритмом сортировки на месте (in-place sort).

**Сортировка слиянием** – рекурсивный алгоритм сортировки сравнением, основанный на методе декомпозиции.

1. Массив A[n] разделяем пополам, и отсортировываем каждый из подмассивов попарно по возрастанию (если предыдущий больше текущего элемента, меняем местами)
2. У нас есть 2 массива b [n/2] и a[n/2]
3. Устанавливаем указатель на первые элементы тих массивов, и сравниваем их, меньший записываем в новый слитый массив
4. Затем передвигаем указатель, и сравниваем с тем элементом, что оказался большим на предыдущем этапе
5. Продолжаем рукурсивно до те пор, пока в одном из массивов указатель не достигнет конца массива, тогда остальных элементы из оставшегося массива перезаписываем в новый отсортированный

**Целочисленная сортировка** — это задача сортировки коллекции значений данных при помощи целочисленных ключей (корзинная сортировка, поразрядная сортировка, сортировка подсчётом)

**Сортировка подсчетом: (пример целочисленной сортировки)**

Пусть у нас есть массив source из n десятичных цифр ( m = 10 ).

Например, source[7] = { 7, 9, 8, 5, 4, 7, 7 }, n=7. Здесь положим const k=1.

1. Создать массив count из m элементов(счетчиков).

2. Присвоить count[i] количество элементов source, равных i. Для этого:

a. проинициализовать count[] нулями,

b. пройти по source от начала до конца, для каждого числа увеличивая элемент count с соответствующим номером.

c. for( i=0; i<n; i++) count [ source[i] ]++

В нашем примере count[] = { 0, 0, 0, 0, 1, 1, 0, 3, 1, 1 }

3. Присвоить count[i] значение, равное сумме всех элементов до данного:

count[i] = count[0]+count[1]+...count[i-1].

В нашем примере count[] = { 0, 0, 0, 0, 1, 2, 2, 2, 5, 6 }

Эта сумма является количеством чисел исходного массива, меньших i.

4. Произвести окончательную расстановку.

Для каждого числа source[i] мы знаем, сколько чисел меньше него - это значение хранится в count[source[i] ]. Таким образом, нам известно окончательное место числа в упорядоченном массиве: если есть K чисел меньше данного, то оно должно стоять на позиции K+1.

Осуществляем проход по массиву source слева направо, одновременно заполняя выходной массив dest:

for ( i=0; i<n; i++ ) {

c = source[i];

dest[ count[c] ] = c;

count[c]++; // для повторяющихся чисел

}

Таким образом, число c=source[i] ставится на место count[c]. На этот случай, если числа повторяются в массиве, предусмотрен оператор count[c]++, который увеличивает значение позиции для следующего числа c, если таковое будет.

## Быстрая сортировка

Этот метод был разработан более 40 лет назад. Это наиболее широко применяемый и один из самых эффективных алгоритмов.

Метод основан на подходе "разделяй-и-властвуй". Общая схема такова:

1. из массива выбирается некоторый опорный элемент *a[i];*
2. запускается процедура разделения массива, которая перемещает все ключи, меньшие либо равные *a[i],* влево от него, а все ключи, большие либо равные *a[i]* - вправо,
3. теперь массив состоит из двух подмножеств, причем левое меньше либо равно правому (рис.21);

|  |  |  |
| --- | --- | --- |
| <= *a[i]* | *a[i]* | *>= a[i]* |

Рис. 21

1. для обоих подмассивов: если в подмассиве более двух элементов, рекурсивно запускаем для него ту же процедуру.

**Пирамидальная сортировка** основана на алгоритме построения пирамиды.

Пирамидальная сортировка производится в два этапа. Сначала строится пирамида из элементов массива. По свойству пирамиды правая часть массива является (*n/2+1, n*)-пирамидой. Будем добавлять по одному элементу слева, расширяя пирамиду, пока в неё не войдут все элементы массива. Тогда свойству пирамиды первый элемент последовательности – минимальный.

Произведём двустороннее усечение: уберём элементы a1,an. По свойству (1) оставшаяся последовательность является (2, n-1)-пирамидой. Элемент a1 поставим на последнее место, а элемент an добавим к пирамиде a2,…,an-1 слева. Получим новую (1, *n-1*)-пирамиду. В ней первый элемент является минимальным. Поставим первый элемент пирамиды на позицию *n-1*, а элемент an-1 добавим к пирамиде a2,…,an-1, и т.д. В результате получим обратно отсортированный массив.

**Поиск**

Наиболее распространенными являются три алгоритма поиска по совпадению:

1) Линейный;

2) ~~Дихотомический~~(бинарный):

3) Интерполирующий.

**1. Алгоритм линейного поиска**

В соответствии с этим алгоритмом массив просматривается последовательно от первого до последнего элемента. Основной недостаток алгоритма линейного поиска – большое время. В худшем случае искомый элемент находится в конце массива или отсутствует.(O(n))**2. Алгоритм ~~дихотомического~~ (бинарного) поиска**

Этот алгоритм является более быстрым, чем линейный, но применяется только к упорядоченным массивам. Асимптотическая оценка алгоритма О= log2n. Применим для массива упорядоченного по возрастанию.

Метод основан на последовательном делении на 2 диапазона поиска. На каждом шаге осуществляется поиск середины отрезка по формуле:mid = (left + right)/2

Если искомый элемент равен элементу с индексом mid, поиск завершается. В случае если искомый элемент меньше элемента с индексом mid, на место mid перемещается правая граница рассматриваемого отрезка, в противном случае — левая граница.

Бинарный поиск неэффективно использует кэш-память процессора – доступ к элементам массива непоследовательный (прыжки по массиву).

Бинарный поиск (Binary search)

1. Если центральный элемент равен искомому, конец

2. Если центральный меньше, делаем текущей правую половину массива

3. Если центральный больше, делаем текущей левую половину массива

**3. Алгоритм интерполирующего поиска**

Интерполирующий поиск основан на принципе поиска в телефонной книге или, например, в словаре. Вместо сравнения каждого элемента с искомым как при линейном поиске, производится предсказание местонахождения элемента.

В среднем, интерполирующий поиск производит O=log(log(n)) операций, где n - число элементов в массиве.

Ключом может быть не только номер, число, но и, например, текстовая строка.

Общий алгоритм интерполирующего поиска без задания исходного массива может быть таким:

На каждой стадии рассчитывается позиция mid (средняя) для следующей проверки, по формуле:

mid = low + (аргумент - ключ [low])\*(high - low) / (ключ [high] - ключ[low])

где low – нижняя граница диапазона, high – его верхняя граница.

Затем в зависимости от результата сравнения переносится верхняя или нижняя граница, определяя новую область поиска. Процесс заканчивается, если элемент с индексом mid равен искомому ключу или нижняя и верхняя границы поиска совпали.

**Словари. Двоичные деревья поиска**

Словарь – структура данных для хранения пар вида «ключ» – «значение» (key – value) (ассоциативный массив )

Бинарное дерево – это дерево (структура данных), в котором каждый узел имеет не более двух дочерних узлов

Двоичное дерево поиска – это двоичное дерево, в котором:

1) каждый узел x (node) имеет не более двух дочерних узлов и содержит ключ (key) и значение (value)

2) ключи всех узлов левого поддерева узла x меньше значения его ключа

3) ключи всех узлов правого поддерева узла x больше значения его ключа

**ОБХОДЫ БИНАРНОГО ДЕРЕВА**

//прямой обход бинарного дерева

void PreOrder\_BinaryTree(BinaryTree\* Node){

if (Node != NULL) {

printf ("%3ld",Node->Data);

PreOrder\_BinaryTree(Node->Left);

PreOrder\_BinaryTree(Node->Right);

}

}

//обратный обход бинарного дерева

void PostOrder\_BinaryTree(BinaryTree\* Node){

if (Node != NULL) {

PostOrder\_BinaryTree(Node->Left);

PostOrder\_BinaryTree(Node->Right);

printf ("%3ld",Node->Data);

}

}

//симметричный обход бинарного дерева

void SymmetricOrder\_BinaryTree(BinaryTree\* Node){

if (Node != NULL) {

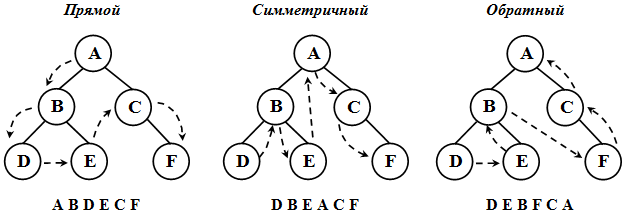
PostOrder\_BinaryTree(Node->Left);

printf ("%3ld",Node->Data);

PostOrder\_BinaryTree(Node->Right);

}

}



**СЛОВАРЬ. ХЕШ-ТАБЛИЦЫ**

**Словарь** (ассоциативный масив) – структура данных (контейнер) для хранения пар вида «ключ – значение».

Наибольшее распространение получили следующие реализации: Деревья поиска, Хэш-таблицы, Списки с пропусками, Связные списки, Массивы

**Хеш-таблицы:**

- Хеш-таблица (hash table) – структура данных для хранения пар «ключ – значение»

- Доступ к элементам осуществляется по ключу (key)

- Ключи могут быть строками, числами, указателями, …

- Хеш-таблицы позволяют в среднем за время О(1) выполнять добавление, поиски и удаление элементов

Коэффициент α заполнения хеш-таблицы (load factor, fill factor) – отношение числа n хранимых элементов в хеш-таблицы к размеру h массива (среднее число элементов на одну ячейку).

Хеш-функция (Hash function) – это функция преобразующая значения ключа (например: строки, числа, файла) в целое число.

Значение возвращаемое хеш-функцией называется хеш-кодом, контрольной суммой или хешем.

**Коллизия** – это совпадение значений хеш-функции для двух разных ключей.

**Разрешение коллизий:**

1. метод цепочек – закрытая адресация:

Элементы с одинаковым значением хеш-функции объединяются в связный список. Указатель на список хранится в советующей ячейке хеш-таблицы.

- При коллизии элемент добавляется в начало списка.

- Поиск и удаление элемента требуют просмотра всего списка.

**2)** Открытая адресация

В каждой ячейке хеш-таблицы хранится не указатель на связный список, а один элемент (ключ, значение)

Если ячейка с индексом hash(key) занята, то осуществляется поиск свободной ячейки в следующих позициях таблицы.

**Требования к хеш-функциям:**

- Быстрое вычисление хэш-кода по значению ключа

- Сложность вычисления хэш-кода не должна зависеть от количества n элементов в хеш-таблице

- Детерминированность – для заданного значения ключа хэш-функция всегда должна возвращать одно и то же значение

- равномерность

**БИНАРНЫЕ КУЧИ**

Очередь с приоритетом – очередь, в которой элементы имеют приоритет (вес); первым извлекается элемент с наибольшим приоритетом (ключом).

Бинарная куча (пирамида, сортирующее дерево, binary heap) – это двоичное дерево, удовлетворяющее следующим условиям:

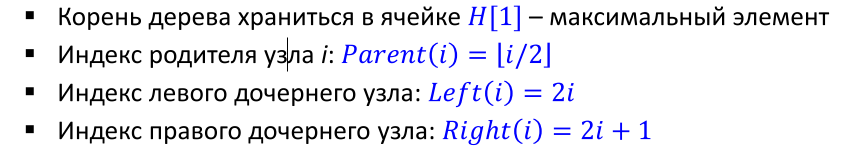
- Приоритет любой вершины не меньше ( ≥ , невозрастающая пирамида max-heap) / не больше (≤, неубывающая пирамида min-heap), приоритета ее потомков

- Дерево является полным двоичным деревом – все уровни заполнены слева направо.

**Реализация бинарной кучи на основе массива:**

Новый элемент добавляется на последнее место в массиве, то есть позицию с максимальным индексом. Возможно, что при этом будет нарушено основное свойство кучи, так как новый элемент может быть больше родителя. В таком случае новый элемент «поднимается» на один уровень (менять с вершиной-родителем) до тех пор, пока не будет соблюдено основное свойство кучи.

В упорядоченном max-heap максимальный элемент всегда хранится в корне. Восстановить упорядоченность двоичной кучи после удаления максимального элемента можно, поставив на его место последний элемент и вызвав метод упорядочения для корня, то есть упорядочив все дерево.



**Графы. Способы представления графов в памяти.**

Граф – совокупность точек, соединенных линиями. Точки называются вершинами, или узлами, а линии – ребрами, или дугами.

Графы, ребрам которых поставлено в соответствие конкретное числовое значение, они называются *взвешенными графами,* а это значение – весом ребра.

*Связный граф*– граф, в котором существует путь из каждой вершины в любую другую

*Полный граф* – это граф, в котором каждая пара различных вершин смежна (каждая вершина соединена со всеми)

**Основные способы представления графов в памяти:**

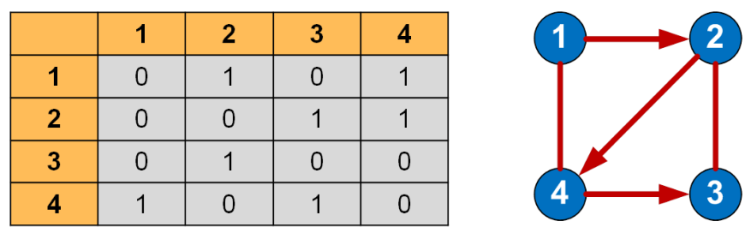
- Матрица смежности – эффективна для насыщенных графов (это граф, в котором количество ребер близко к максимально возможному)

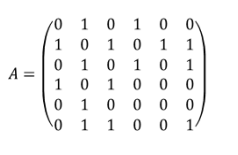
- Списки смежных вершин– эффективен для разреженных графов (граф, в котором количество ребер близко к количеству вершин в графе)

*Матрица смежности графа* — это квадратная матрица, в которой каждый элемент принимает одно из двух значений: 0 или 1.

Число строк матрицы смежности равно числу столбцов и соответствует количеству вершин графа.

0 – соответствует отсутствию ребра,

1 – соответствует наличию ребра.

Списки смежных вершин– это массив A[n], каждый элемент A[i] которого содержит список узлов смежных с вершиной i

**Обходы графов: поиск в глубину, поиск в ширину.**

Обход графа – это процедура перебора (посещения) всех вершин графа начиная с заданной

Поиск в глубину предполагает продвижение вглубь до тех пор, пока это возможно. Невозможность продвижения означает, что следующим шагом будет переход на последний, имеющий несколько вариантов движения (один из которых исследован полностью), ранее посещенный узел (вершина).

Поиск в ширину подразумевает поуровневое исследование графа:

-вначале посещается корень – произвольно выбранный узел,

-затем – все потомки данного узла,

-после этого посещаются потомки потомков и т.д.

Вершины просматриваются в порядке возрастания их расстояния от корня.

Алгоритм прекращает свою работу после обхода всех вершин графа, либо в случае выполнения требуемого условия.

(функции остальные те же)

**Алгоритм Дейкстры поиска кратчайшего пути.**

Находит кратчайшее расстояние от одной из вершин графа до всех остальных. Работает только для графов без рёбер отрицательного веса.

Шаг 1. Всем вершинам, за исключением первой, присваивается *вес* равный бесконечности, а первой вершине – 0.

Шаг 2. Все вершины не выделены.

Шаг 3. Первая *вершина* объявляется текущей.

Шаг 4. *Вес* всех невыделенных вершин пересчитывается по формуле: *вес* невыделенной вершины есть минимальное число из старого веса данной вершины, суммы веса текущей вершины и веса *ребра*, соединяющего текущую вершину с невыделенной.

Шаг 5. Среди невыделенных вершин ищется *вершина* с минимальным весом. Если таковая не найдена, то есть *вес* всех вершин равен бесконечности, то *маршрут* не существует. Следовательно, *выход*. Иначе, текущей становится найденная *вершина*. Она же выделяется.

Шаг 6. Если текущей вершиной оказывается конечная, то *путь* найден, и его *вес* есть *вес* конечной вершины.

Шаг 7. Переход на шаг 4.

1. Выбираем вершину(текущая), приравниваем к нулю( мы ее посетили)
2. Остальные точки приравниваем к макс сумме всех элементов
3. Смотрим на вес, ведущий к ближайшим узлам
4. *Вес* всех невыделенных вершин пересчитывается по формуле: *вес* невыделенной вершины есть минимальное число из старого веса данной вершины, суммы веса текущей вершины и веса *ребра*, соединяющего текущую вершину с невыделенной.
5. Всегда считаем сколько стоит попасть в точку начиная от вершины
6. Среди невыделенных вершин ищется *вершина* с минимальным весом
7. Если текущей вершиной оказывается конечная, то *путь* найден,
8. Переход на шаг 4.