Подготовка к магистратуре МГТУ ИУ7 by Winterpuma

Направление: 09.04.04

Дисциплина 2.

# Типы и структуры данных

**Оглавление**

**1. Понятие списка, стека, дека, очереди. Основные алгоритмы включения, исключения элементов в этих структурах. Сравнительный анализ эффективности конкретных реализаций данных структур. Алгоритмы включения и исключения элемента из двусвязного списка.**

**Список** — это абстрактный тип данных, представляющий собой упорядоченный набор значений, в котором некоторое значение может встречаться более одного раза.

**Стек** — это линейный список с переменной длиной, в котором все операции вставки, удаления и доступа к данным выполняются только на одном из концов списка, называемом вершиной стека. (функционирует по принципу LIFO)

**Очередь** — это такой линейный список с переменной длиной, в котором включение элементов выполняется только с одной стороны списка (эту сторону часто называют концом или хвостом очереди), а исключение — с другой стороны (называемой началом или головой очереди). (FIFO)

**Дек** (double ended queue, т.е. очередь с двумя концами), представляет собой структуру данных, в которой можно добавлять и удалять элементы с двух сторон.

TODO **Сравнительный анализ эффективности конкретных реализаций данных структур**

**Алгоритмы включения и исключения элемента из двусвязного списка.**


Структура линейного двусвязного списка (есть еще кольцевой)

**Узел ДЛС:**

```
struct list
{
  int field; // поле данных
  struct list *next; // указатель на следующий элемент
  struct list *prev; // указатель на предыдущий элемент
};
```

**Добавление корня:**

хм)

**Добавление узла** (возвращает адрес добавленного узла):

```
struct list * addelem(list *lst, int number)
{
  struct list *temp, *p;
  temp = (struct list*)malloc(sizeof(list));
  p = lst->next; // сохранение указателя на следующий узел
  lst->next = temp; // предыдущий узел указывает на создаваемый
  temp->field = number; // сохранение поля данных добавляемого узла
  temp->next = p; // созданный узел указывает на следующий узел
  temp->prev = lst; // созданный узел указывает на предыдущий узел
  if (p != NULL)
    p->prev = temp;
  return(temp);
}
```

**Удаление узла:**

```
struct list * deletelem(list *lst)
{
  struct list *prev, *next;
```

```
  prev = lst->prev; // узел, предшествующий lst
  next = lst->next; // узел, следующий за lst
  if (prev != NULL)
    prev->next = lst->next; // переставляем указатель
  if (next != NULL)
    next->prev = lst->prev; // переставляем указатель
  free(lst); // освобождаем память удаляемого элемента
  return(prev);
}
```

**Удаление корня**:

```
struct list * deletehead(list *root)
{
  struct list *temp;
  temp = root->next;
  temp->prev = NULL;
  free(root);   // освобождение памяти текущего корня
  return(temp); // новый корень списка
}
```

**2. Понятие рекурсии. Рекурсивные типы данных. Рекурсивные процедуры и функции. Критерии выбора для разработки рекурсивных или итеративных алгоритмов.**

Рекурсивное объявление объекта подразумевает определение объекта частично или полностью через (с помощью) этот объект.

Пример-факториал: $n! = (n-1)! * n$

**Рекурсия** — это такой способ организации процесса, при котором подпрограмма, в ходе выполнения составляющих её операторов, обращается к самой себе. Обращение может быть прямое или косвенное.

**Рекурсивные типы данных** - представляют собой тип данных для значений, которые могут содержать другие значения одного и тот же типа.

- **Список** — это либо пустая структура, либо состоящая из двух элементов: головы и хвоста, который является списком.
- **Дерево** — это либо пустая структура, либо узел, связанный дугами с конечным числом деревьев (поддеревьев).

**Хвостовая рекурсия** — частный случай рекурсии, при котором любой рекурсивный вызов является последней операцией перед возвратом из функции.

Хвостовую рекурсию можно заменить на **итерацию**.

Хвостовая рекурсия C:

```
int fact (int n, int acc) {
    return (n==0) ? acc : fact(n - 1, acc * n);
}
int factorial (int n) {
    return fact(n, 1);
}
```

Не хвостовая рекурсия C:

```
int factorial (int n) {
    return (n==0) ? 1 : n*factorial(n-1);
}
```

**Критерии выбора для разработки рекурсивных или итеративных алгоритмов.**

Точных правил для выбора между рекурсивной и нерекурсивной версиями алгоритма решения задачи не существует. Краткость и выразительность большинства рекурсивных процедур упрощает их чтение и сопровождение. С другой стороны, выполнение рекурсивных процедур требует больших затрат и памяти, и времени процессора нежели их итерационные аналоги.

**3. Оценка эффективности алгоритмов. Принципы выбора различных по эффективности алгоритмов для решения конкретных задач.**

**Эффективность алгоритма** — это свойство алгоритма, которое связано с вычислительными ресурсами, используемыми алгоритмом.

Различные ресурсы (такие как время и память) нельзя сравнить напрямую, так что какой из двух алгоритмов считать более эффективным часто зависит от того, какой фактор более важен.

В теоретическом анализе алгоритмов обычной практикой является оценка сложности алгоритма в его асимптотическом поведении, то есть для отражения сложности алгоритма как функции от размера входа n используется **нотация «О» большое**. Эта оценка, в основном, достаточно точна при большом n, но может привести к неправильным выводам при малых значениях n (так, сортировка пузырьком, считающаяся медленной, может оказаться быстрее «быстрой сортировки», если нужно отсортировать лишь несколько элементов).

**Примеры нотации О большое:**

- O(1) постоянное - использование хеш-функции для выбора элемента
- O(log n) логарифмическое - нахождение элемента в отсортированном массиве с помощью двоичного поиска.
- O(n) линейное - поиск элемента в несортированном списке (худший случай)
- O(n log n) квазилинейное - пирамидальная сортировка, быстрая сортировка (лучший и средний случай), сортировка слиянием
- O(n^2) квадратное - сортировка пузырьком (худший случай), сортировка Шелла, быстрая сортировка (худший случай), сортировка выбором, сортировка вставками
- O(c^n), c>1 экспоненциальное - нахождение (точного) решения задачи коммивояжёра с помощью динамического программирования.

**Оценим сортировку вставками**

Будем считать, что срока псевдокода требует не более чем фиксированного количества элементарных операций, если строка не является формулировкой сложных действий. При анализе около каждой строки будем отмечать ее стоимость – число элементарных операций Ci и число раз, которые эта строка выполняется. Выведем основной параметр, используемый при анализе – размерность массива n, т.е. число элементов.

| | Insertion_Sort (A, n) | Стоимость | Число раз |
|---|---|---|---|
| 1. | for j ← 2 to n | C1 | n |
| 2. | do key ← A[j] | C2 | n-1 |
| 3. | ▷ добавить A[j] к отсортированной части A[1... j -1] | | |
| 4. | i←j -1 | C4 | n-1 |
| 5. | while i>0 and A[i]>key | C5 | $\sum_{j=2}^{n} t_j$ |
| 6. | do A[i+1] ← A[i] | C6 | $\sum_{j=2}^{n} (t_j - 1)$ |
| 7. | i←i-1 | C7 | $\sum_{j=2}^{n} (t_j - 1)$ |
| 8. | A[i+1] ←key | C8 | n-1 |

Для каждого j от 2 до n подсчитаем, сколько раз будет выполнена строка 5 и обозначим это число через tj. Строка стоимостью C, повторенная n раз, вносит в общее время c*n операций. Сложив все вклады строк, получим показатель:

$$T(n) = C_1 n + C_2(n-1) + C_4(n-1) +$$
$$C_5 \sum_{j=2}^{n} t_j + C_6 \sum_{j=2}^{n} (t_j - 1) + C_7 \sum_{j=2}^{n} (t_j - 1) + C_8(n-1)$$

Эффективность зависит не только от размерности массива, но и от его упорядоченности.

**Лучший случай**, когда массив уже отсортирован. Тогда цикл в строке 5 завершается после первой проверки, т. к. A[i] <=key при i= j-1 и **tj =1**. Тогда общее время равно: **T(n)=a*n+b**

$$T(n) = C_1(n) + C_2(n-1) + C_4(n-1) + C_5(n-1) + C_8(n-1) =$$
$$(C_1 + C_2 + C_4 + C_5 + C_8)n - (C_2 + C_4 + C_5 + C_8)$$

**Худший случай**, когда массив расположен в обратном порядке, убывающем порядке. время работы функции будет максимальным, т.к. каждый элемент A[j] придется сравнивать со всеми элементами A[1]…A[j-1]. При этом **tj =j**.

Таким образом, **T(n)=a*n^2+b*n+c**. Константы a, b, c определяются значениями Ci.

Асимптотическая оценка a*n2+b*n+c ⇔ a*n2 – порядок роста – определяет нотацию BigO=O(n2) и C1, …. , C8.

**Принципы выбора различных по эффективности алгоритмов для решения конкретных задач.**

Для выбора подходящего алгоритма необходимо определить использование каких ресурсов является критическим моментом: память или время. Зачастую оптимизации использования памяти можно достичь за счет увеличения времени выполнения и наоборот. Также нужно учесть с какими данными ведется работа: сортировка пузырьком, считающаяся медленной, может оказаться быстрее «быстрой сортировки», если нужно отсортировать лишь несколько элементов.

**4. Разреженные матрицы. Методы хранения и расчета разреженных матриц. Использование разреженных матриц.**

**Разрежённая матрица** — это матрица с преимущественно нулевыми элементами. В противном случае, если большая часть элементов матрицы ненулевые, матрица считается плотной.

Огромные разрежённые матрицы часто возникают при решении таких задач, как дифференциальное уравнение в частных производных.

Операции и алгоритмы, применяемые для работы с обычными, плотными матрицами, применительно к большим разрежённым матрицам работают относительно медленно и требуют значительных объёмов памяти. Однако разрежённые матрицы **могут быть легко сжаты** путём записи только своих ненулевых элементов, что снижает требования к компьютерной памяти.

**Методы хранения и расчета разреженных матриц:**

- **координатный формат** - элементы матрицы и ее структура хранятся в трех массивах, содержащих значения, и X и Y координаты.

- **схема Кнута** - ненулевые элементы хранятся в массиве структур (Data, I, J, NR, NC), где Data != 0; I,Jj строчный и столбцовый индексы; NR - указатель на следующий ненулевой элемент строки, NC - -||- столбца. Также есть два массива JR и JC, содержащие указатели входа для строк и столбцов.
    - такая схема **неэкономна** по памяти (пять ячеек для каждого ненулевого + указатели входа).
    - в любом месте можно включить или исключить элемент, эффективное сканирование строк и столбцов.
    - идеально приспособлена для случаев, когда матрица строится каким-то алгоритмом, где нельзя предсказать конечное число и позиции ненулевых элементов.
- **разреженный строчный формат** - для хранения требуется 3 одномерных массива:
    1. массив ненулевых элементов матрицы A, в котором они перечислены по строкам от первой до последней.
    2. массив номеров столбцов для соответствующих элементов массива 1.
    3. массив указателей позиций, с которых начинается описание очередной строки.
    - наиболее широко используемый;
    - минимальные требования к памяти, но при этом удобен для операций над разреженными матрицами (сложение, умножение, перестановки, транспонирование, итд)

$$A = \begin{bmatrix} 1 & -1 & 0 & -3 & 0 \\ -2 & 5 & 0 & 0 & 0 \\ 0 & 0 & 4 & 6 & 4 \\ -4 & 0 & 2 & 7 & 0 \\ 0 & 8 & 0 & 0 & -5 \end{bmatrix}$$

7.40 в разреженном строчном формате:

values=(1, -1, -3, -2, 5, 4, 6, 4, -4, 2, 7, 8, -5);

cols=(1, 2, 4, 1, 2, 3, 4, 5, 1, 3, 4, 2, 5);

pointer=(1, 4, 6, 9, 12, 14).

- **разреженный столбцовый формат** - почти как строчный только элементы хранятся по столбцам. Столбцовые представления могут рассматриваться как строчные представления транспонированных матриц.
- сжатие по Шерману.

**5. Понятие абстрактных типов данных. Принципы создание алгоритмов с использованием абстрактных типов данных.**

**Абстрактный тип данных** (АТД) — математическая модель для типов данных, где тип данных определяется в терминах возможных значений, возможных операций над данными этого типа и поведения этих операций.

Абстрактный тип данных определяет набор функций, независимых от конкретной реализации типа, для оперирования его значениями. Конкретные реализации АТД называются структурами данных.

В программировании абстрактные типы данных обычно представляются в виде интерфейсов, которые скрывают соответствующие реализации типов. Программисты работают с абстрактными типами данных исключительно через их интерфейсы, поскольку реализация может в будущем измениться. Такой подход соответствует **принципу инкапсуляции** в объектно-ориентированном программировании.

Различие между АТД и структурами данных, которые их реализуют: АТД список может быть реализован при помощи массива или линейного списка с использованием различных методов динамического выделения памяти. Однако каждая реализация определяет один и тот же набор функций, который должен работать одинаково (по результату, а не по скорости) для всех реализаций.

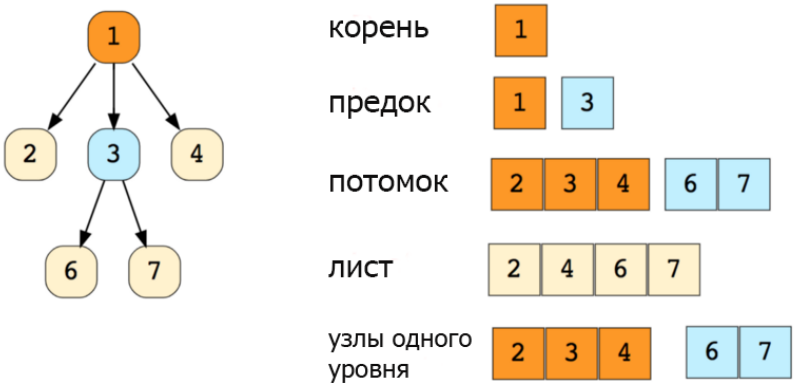
Абстрактные типы данных позволяют достичь модульности программных продуктов и иметь несколько альтернативных взаимозаменяемых реализаций отдельного модуля.

**6. Деревья. Виды деревьев. Остовное дерево. Использование различных видов деревьев для поиска и сортировки. Сравнение различных методов поиска в массивах, деревьях, хэш-таблицах**

**Дерево** - это либо пустая структура, либо узел, связанный дугами с конечным числом деревьев (поддеревьев).

Вершина Y, находящаяся непосредственно ниже вершины X, называется **потомком** X, а вершина X - **предком** Y.

Считается, что корень дерева находится на 0 **уровне**, непосредственные потомки корня на 1 уровне и т.д. максимальный уровень какой-нибудь из вершин дерева - **глубина дерева**.



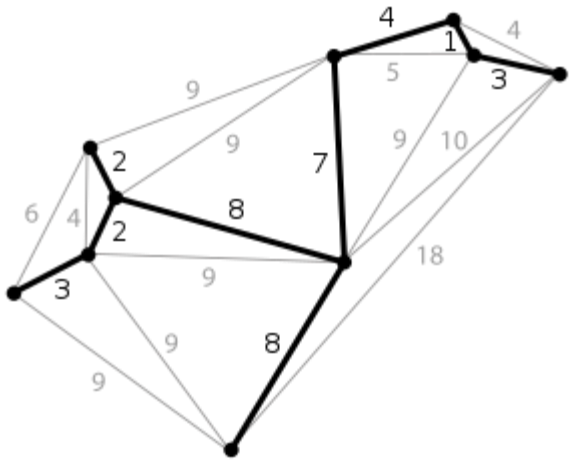

**Типы деревьев (определения для ориентированных):**

- N-арное дерево — это ориентированное дерево, в котором исходящие степени вершин (число исходящих рёбер) не превосходят N.;
- сбалансированное дерево;
- бинарное дерево - ориентированное дерево, в котором исходящие степени вершин (число исходящих рёбер) не превосходят 2;
- бинарное дерево поиска;

- оба поддерева — левое и правое — являются двоичными деревьями поиска;
- у всех узлов левого поддерева произвольного узла X значения ключей данных меньше, нежели значение ключа данных самого узла X;
- у всех узлов правого поддерева произвольного узла X значения ключей данных больше либо равны, нежели значение ключа данных самого узла X.

- дерево AVL - сбалансированное по высоте двоичное дерево поиска: для каждой его вершины высота её двух поддеревьев различается не более чем на 1;
- красно-черное дерево;
- 2-3 дерево.

**Остовное дерево** графа — получается из исходного графа удалением максимального числа рёбер, входящих в циклы, но без нарушения связности графа. Остовное дерево включает в себя все n вершин исходного графа и содержит n-1 ребро.




**Использование различных видов деревьев для поиска и сортировки**.

Для сортировки при помощи **бинарного дерева поиска** нужно:

1. разместить все элементы в структуре дерево (первый корневой, далее располагаем слева если меньше или справа, если больше)
2. вывод при помощи инфиксной формы обхода.

Обход дерева в инфиксной форме:

```
void treeprint(tnode *tree) {
  if (tree!=NULL) { //Пока не встретится пустой узел
    treeprint(tree->left); //Рекурсивная функция для левого поддерева
    cout << tree->field; //Отображаем корень дерева
    treeprint(tree->right); //Рекурсивная функция для правого поддерева
  }
}
```

**Heapsort** (сортировка с помощью дерева):

1. построение дерева: листья - все элементы массива, предок соседней пары заполняется наименьшим значением из пары;
2. текущий наименьший лист заменяется бесконечно большим значением, его предки предки пересчитываются. Процесс продолжается до тех пор, пока все узлы дерева не будут заполнены фиктивными ключами.

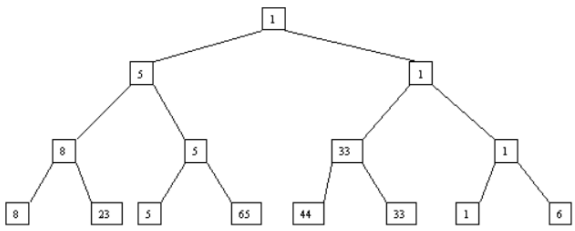



*Рис. 2.1.*

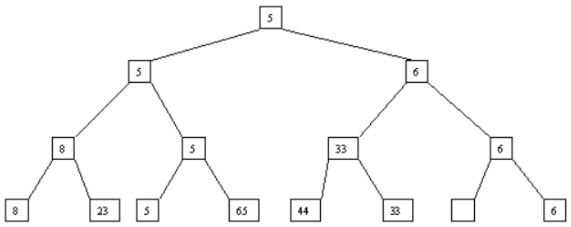



*Рис. 2.2. Второй шаг*

**Поиск:**

Поиск узла в **двоичном дереве поиска** можно осуществить, двигаясь от корня в левое или правое поддерево в зависимости от значения ключа поддерева. Желательно, чтобы все пути в дереве от корня до листьев имели примерно одинаковую длину, то есть чтобы глубина и левого, и правого поддеревьев была примерно одинакова в любом узле. В противном случае теряется производительность.

**В вырожденном случае** может оказаться, что все левое дерево пусто на каждом уровне, есть только правые деревья, и в таком случае дерево

вырождается в список (идущий вправо). Поиск (а значит, и удаление и добавление) в таком дереве по скорости равен поиску в списке и намного медленнее поиска в сбалансированном дереве.

**Дерево АВЛ** будет иметь высоту не более Log2 N , поэтому для поиска среди N элементов может потребоваться не больше Log2 N сравнений вместо N.

## Сравнение различных методов поиска в массивах, деревьях, хэш-таблицах

| Хорошо | Приемлемо | Плохо |
|---|---|---|

| Алгоритм | Структура данных | Временная сложность | | Сложность по памяти |
|---|---|---|---|---|
| | | В среднем | В худшем | В худшем |
| Поиск в глубину (DFS) | Граф с \|V\| вершинами и \|E\| ребрами | - | O(\|E\| + \|V\|) | O(\|V\|) |
| Поиск в ширину (BFS) | Граф с \|V\| вершинами и \|E\| ребрами | - | O(\|E\| + \|V\|) | O(\|V\|) |
| Бинарный поиск | Отсортированный массив из n элементов | O(log(n)) | O(log(n)) | O(1) |
| Линейный поиск | Массив | O(n) | O(n) | O(1) |
| Кратчайшее расстояние по алгоритму Дейкстры используя двоичную кучу как очередь с приоритетом | Граф с \|V\| вершинами и \|E\| ребрами | O((\|V\| + \|E\|) log \|V\|) | O((\|V\| + \|E\|) log \|V\|) | O(\|V\|) |
| Кратчайшее расстояние по алгоритму Дейкстры используя массив как очередь с приоритетом | Граф с \|V\| вершинами и \|E\| ребрами | O(\|V\|^2) | O(\|V\|^2) | O(\|V\|) |
| Кратчайшее расстояние используя алгоритм Беллмана–Форда | Граф с \|V\| вершинами и \|E\| ребрами | O(\|V\|\|E\|) | O(\|V\|\|E\|) | O(\|V\|) |

| Структура данных | Временная сложность | | | | | | | | Сложность по памяти |
|---|---|---|---|---|---|---|---|---|---|
| | В среднем | | | | В худшем | | | | В худшем |
| | Индексация | Поиск | Вставка | Удаление | Индексация | Поиск | Вставка | Удаление | |
| Обычный массив | O(1) | O(n) | - | - | O(1) | O(n) | - | - | O(n) |
| Динамический массив | O(1) | O(n) | O(n) | O(n) | O(1) | O(n) | O(n) | O(n) | O(n) |
| Односвязный список | O(n) | O(n) | O(1) | O(1) | O(n) | O(n) | O(1) | O(1) | O(n) |
| Двусвязный список | O(n) | O(n) | O(1) | O(1) | O(n) | O(n) | O(1) | O(1) | O(n) |
| Список с пропусками | O(log(n)) | O(log(n)) | O(log(n)) | O(log(n)) | O(n) | O(n) | O(n) | O(n) | O(n log(n)) |
| Хеш таблица | - | O(1) | O(1) | O(1) | - | O(n) | O(n) | O(n) | O(n) |
| Бинарное дерево поиска | O(log(n)) | O(log(n)) | O(log(n)) | O(log(n)) | O(n) | O(n) | O(n) | O(n) | O(n) |
| Декартово дерево | - | O(log(n)) | O(log(n)) | O(log(n)) | - | O(n) | O(n) | O(n) | O(n) |
| Б-дерево | O(log(n)) | O(log(n)) | O(log(n)) | O(log(n)) | O(log(n)) | O(log(n)) | O(log(n)) | O(log(n)) | O(n) |
| Красно-черное дерево | O(log(n)) | O(log(n)) | O(log(n)) | O(log(n)) | O(log(n)) | O(log(n)) | O(log(n)) | O(log(n)) | O(n) |
| Расширяющееся дерево | - | O(log(n)) | O(log(n)) | O(log(n)) | - | O(log(n)) | O(log(n)) | O(log(n)) | O(n) |
| АВЛ-дерево | O(log(n)) | O(log(n)) | O(log(n)) | O(log(n)) | O(log(n)) | O(log(n)) | O(log(n)) | O(log(n)) | O(n) |

**7. Базовые понятия теории графов. Поиск в графах. Алгоритмы поиска в ширину и глубину, построение каркасов графа. Пути в графах. Алгоритмы поиска Эйлерова и Гамильтонова пути. Алгоритмы поиска минимальных путей. Раскраска графов. Планарность графов.**

**Базовые понятия теории графов.**

**Граф** — это геометрическая фигура, которая состоит из точек и линий, которые их соединяют. Точки называют вершинами графа, а линии — ребрами.

Два ребра называются **смежными**, если у них есть общая вершина.

Два ребра называются **кратными**, если они соединяют одну и ту же пару вершин.

Ребро называется **петлей**, если его концы совпадают.

**Степенью вершины** называют количество ребер, для которых она является концевой (при этом петли считают дважды).

Вершина называется **изолированной**, если она не является концом ни для одного ребра.

Вершина называется **висячей**, если из неё выходит ровно одно ребро.

Граф без кратных ребер и петель называется **обыкновенным**.

**Поиск в графах.**

Во многих задачах поиск решения сводится к поиску кратчайшего расстояния между двумя узлами на некотором графе. Один узел обычно является стартовым (начальное состояние системы), а второй - целевым.

**Алгоритм поиска в ширину**

```
D = [None] * (n + 1)
 D[start] = 0
 Q = [start]
 Qstart = 0
 while Qstart < len(Q):
    u = Q[Qstart]
    Qstart += 1
    for v in V[u]:
        if D[v] is None:
            D[v] = D[u] + 1
            Q.append(v)
```

**Алгоритм поиска в глубину**

```
visited = [False] * (n + 1)
prev = [None] * (n + 1)
def dfs(start, visited, prev, g):
    visited[start] = True
    for u in g[start]:
        if not visited[u]:
            prev[u] = start
            dfs(u)
dfs(start, visited, prev, g)
```

**Построение каркасов графа.** Алгоритм Прима.

Введем массив `used[N]`, первоначально заполненный нулями. Пусть наш каркас первоначально пуст.

1. Пометим первую вершину как использованную: `used[0] = 1`.
2. Найдем ребро `(i, j)`, соединяющее использованную вершину (`used[i] == 1`) с неиспользованной (`used[j] == 0`). Если таких ребер несколько, выберем ребро с минимальной стоимостью. Если таких ребер нет, закончим выполнение алгоритма.
3. Добавим это ребро к каркасу.
4. Пометим `used[j] = 1`.
5. Перейдем к шагу 2.

**Пути в графах.**

**Путь в графе** — последовательность вершин, в которой каждая вершина соединена со следующей ребром.

**Индуцированный путь** - путь, для которого никакие рёбра графа не соединяют две вершины пути.

**Гамильтонов путь** - простая цепь, содержащая все вершины графа без повторений.

**Гамильтонов цикл** - простой цикл, содержащий все вершины графа без повторений.

**Фундаментальный цикл** - цикл, получаемый добавлением ребра графа к остовному дереву исходного графа.

Алгоритмы поиска Эйлерова и Гамильтонова пути. TODO

Алгоритмы поиска минимальных путей. TODO

**Раскраска графов**.

**Раскраска графа** — теоретико-графовая конструкция, частный случай разметки графа. При раскраске элементам графа ставятся в соответствие метки с учётом определённых ограничений; эти метки традиционно называются «цветами».

**Раскраска вершин** - простейший случай окраски, при котором любым двум смежным вершинам соответствуют разные цвета.

**Раскраска рёбер** присваивает цвет каждому ребру так, чтобы любые два смежных ребра имели разные цвета.

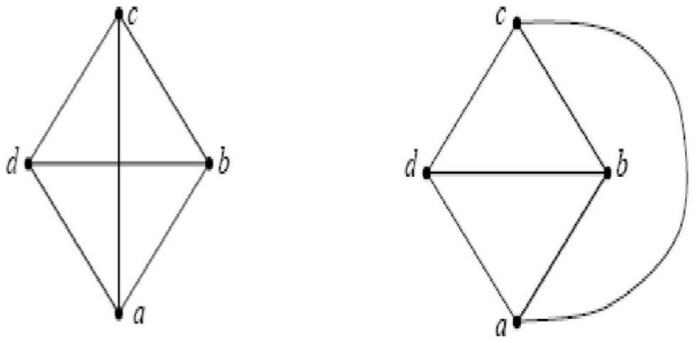
**Раскраска областей** планарного графа назначает цвет каждой области, так, что каждые две области, имеющие общую границу, не могут иметь одинаковый цвет.

**Раскраска вершин** — главная задача раскраски графов, все остальные задачи в этой области могут быть сведены к ней.

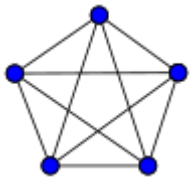
**Планарность графов.**

**Планарный граф** — граф, который можно изобразить на плоскости без пересечений рёбер не по вершинам. Какое-либо конкретное изображение планарного графа на плоскости без пересечения рёбер не по вершинам называется **плоским графом**.

Пример изображения графа как планарного




Полный граф с пятью вершинами (K5) нельзя уложить на плоскость:



**Источники вдохновения**


1: Стек, очередь, дек (studfile.net)

1: 5 Нелинейные связные структуры (studfile.net)

1: Двусвязный линейный список (prog-cpp.ru)

2: Рекурсия (studfile.net)

2: Рекурсивный тип данных (hmong.wiki)

2: Хвостовая рекурсия · Winterpuma/bmstu_FaLP Wiki (github.com)

3: Эффективность алгоритма — Википедия (wikipedia.org)

4: Разреженная матрица — Википедия (wikipedia.org)

4: jun05033.pdf (window.edu.ru)

4: Лабораторная работа Разреженное матричное умножение - PDF Free Download (docplayer.ru)

5: Абстрактный тип данных — Википедия (wikipedia.org)

6: 8 известных структур данных, о которых спросят на собеседовании (proglib.io)

6: Рекурсия (studfile.net)

6: Сортировка с помощью дерева (prog-cpp.ru) (бинарное поиска)

6: Методы сортировки и поиска (citforum.ru) (heapsort)

6: Знай сложности алгоритмов / Хабр (habr.com)

7: Основные понятия Теории Графов (skysmart.ru)

7: Онлайн-школа Фоксфорд (foxford.ru)

7: Онлайн-школа Фоксфорд (foxford.ru)

7: Методы программирования :: Практика :: Задание 7 (fenster.name) (построение каркасов)

7: