Алгоритмы и структуры данных

Понятие алгоритма и структуры данных.
Классификация структур данных.
Вычислительная сложность алгоритмов.
Анализ верхней и средней оценок сложности алгоритмов; сравнение наилучших, средних и наихудших оценок.

Понятие алгоритма

Алгоритм – конечная последовательность команд для исполнителя, преобразовывающая входные данные в выходные. Направлено решение, чаще всего, на определённый круг задач.

Свойства алгоритма:

Дискретность — алгоритм должен представлять процесс решения задачи как упорядоченное выполнение некоторых простых шагов. При этом для выполнения каждого шага алгоритма требуется конечный отрезок времени, то есть преобразование исходных данных в результат осуществляется во времени дискретно.

Детерминированность (определённость). В каждый момент времени следующий шаг работы однозначно определяется состоянием системы. Таким образом, алгоритм выдаёт один и тот же результат (ответ) для одних и тех же исходных данных.

Понятность — алгоритм должен включать только те команды, которые доступны исполнителю и входят в его систему команд.

Массовость (универсальность). Алгоритм должен быть применим к разным наборам начальных данных.

Результативность — завершение алгоритма определёнными результатами или выдача сообщение о невозможности получения результата.

Корректность – для любого набора данных должен быть правильный результат.

Структуры данных

Структуры данных - это совокупность логически связанных элементов данных и отношений между ними. Под отношениями между данными понимают функциональные связи между ними и указатели на то, где находятся эти данные. Структурой может быть измерения температуры за определенный период, оценки студента за семестр.

Виды структур.

Структуры классифицируются в зависимости от отношений между элементами.

Классификация структур данных.

Линейные:

- массив;
- список;
- связанный список;
- стек;
- очередь;
- хэш-таблица.

Иерархические:

- двоичные деревья;
- n-арные деревья;
- иерархический список.

Сетевые:

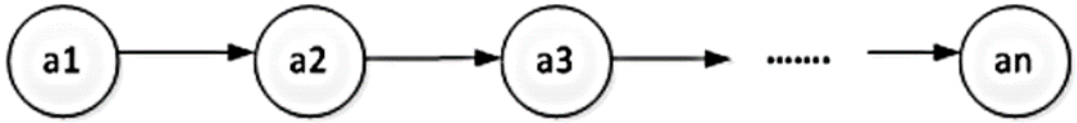
- неориентированный граф;
- ориентированный граф.

Табличные:

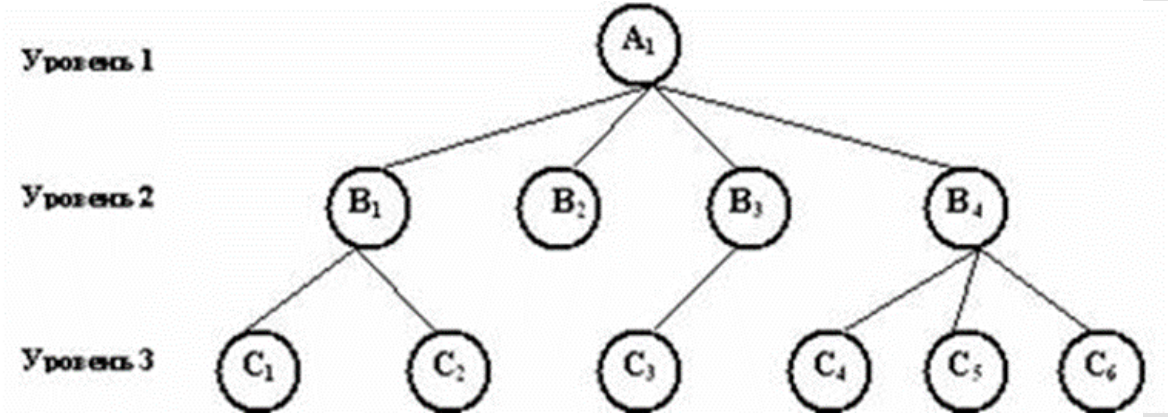
- таблица реляционной базы данных;
- двумерный массив.

Структуры данных

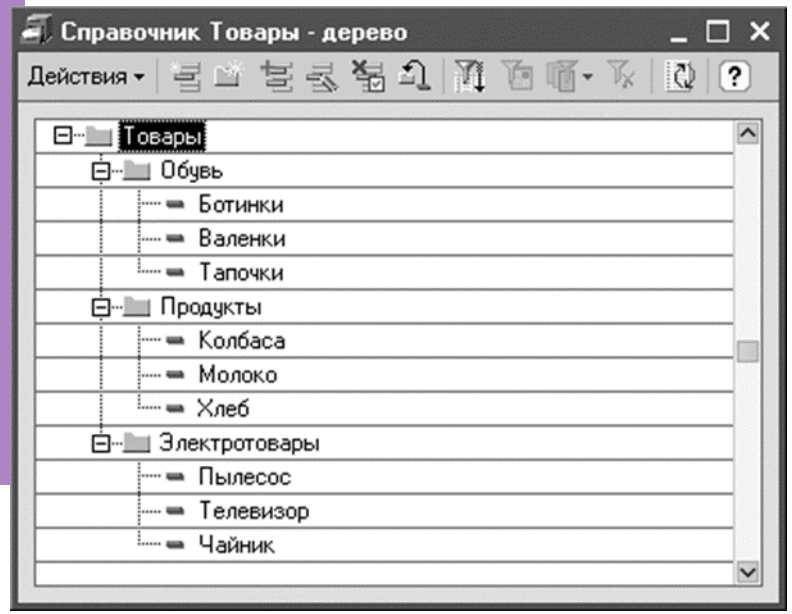
Линейная структура



Иерархическая структура древовидная



Иерархический список



Массивы

- **Массив** — это самая простая и широко используемая структура данных. Другие структуры данных, такие как стеки и очереди, являются производными от массивов.

Изображение простого массива размера 4, содержащего элементы (1, 2, 3 и 4).

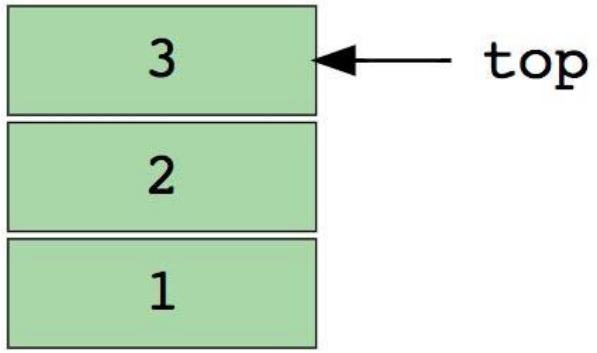
Каждому элементу данных присваивается положительное числовое значение (индекс), который соответствует позиции элемента в массиве. Большинство языков определяют начальный индекс массива как 0.

Стеки

Основные операции

Push-вставляет элемент сверху
Pop-возвращает верхний элемент после удаления из стека
isEmpty-возвращает true, если стек пуст
Top-возвращает верхний элемент без удаления из стека

- **Стек** — абстрактный тип данных, представляющий собой список элементов, организованных по принципу LIFO (англ. last in — first out, «последним пришёл — первым вышел»).
- Это не массивы. Это очередь. Примером стека может быть куча книг, расположенных в вертикальном порядке. Для того, чтобы получить книгу, которая где-то посередине, вам нужно будет удалить все книги, размещенные на ней. Так работает метод LIFO (Last In First Out). Функция «Отменить» в приложениях работает по LIFO.
- Изображение стека, в три элемента (1, 2 и 3), где 3 находится наверху и будет удален первым.

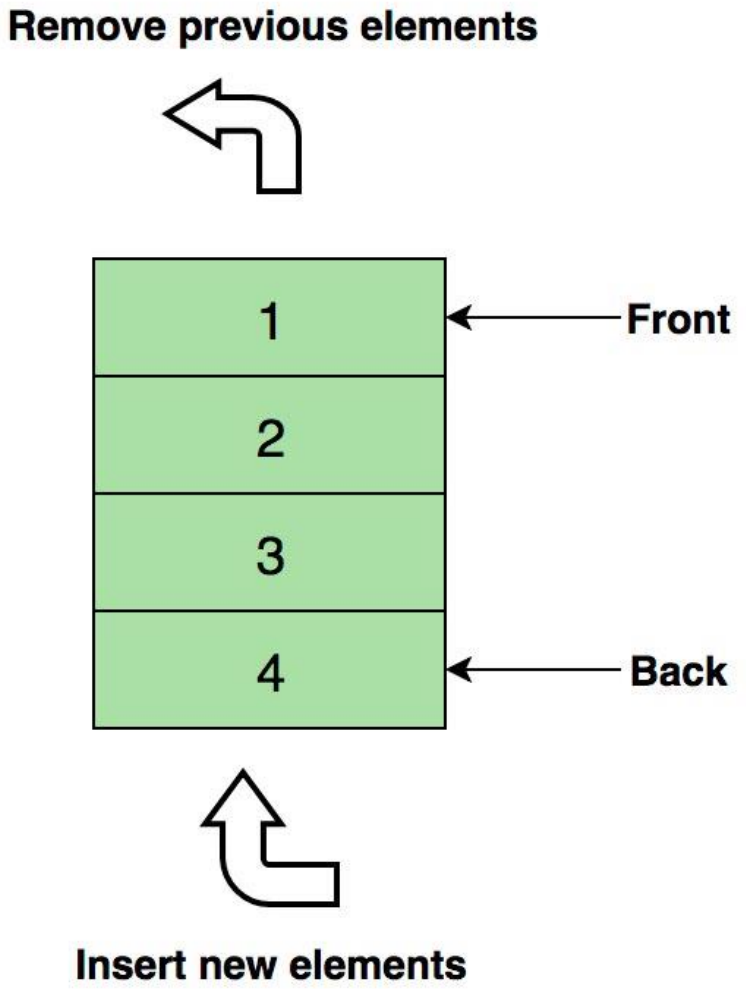


Пример реализации стека на c++

```
#include <iostream>
#include <stack>  // подключаем библиотеку для использования стека
using namespace std;

int main() {
 setlocale(LC_ALL,"rus");
 stack <int> steck;  // создаем стек
 int i = 0;
 cout << "Введите шесть любых целых чисел: " << endl; // предлагаем пользователю ввести 6 чисел
   while (i != 6) {
   int a;
   cin >> a;

   steck.push(a);  // добавляем введенные числа
   i++;
 }

 if (steck.empty()) cout << "Стек не пуст";  // проверяем пуст ли стек (нет)
 cout << "Верхний элемент стека: " << steck.top() << endl; // выводим верхний элемент
 cout << "Давайте удалим верхний элемент " << endl;
 steck.pop();  // удаляем верхний элемент
 cout << "А это новый верхний элемент: " << steck.top(); // выводим уже новый
                                                         // верхний элемент
 system("pause");
 return 0;
}
```

stack.cpp

```
Введите шесть любых целых чисел:
9 5 2 1 5 6
Верхний элемент стека: 6
Давайте удалим верхний элемент
А это новый верхний элемент: 5
Process returned 0 (0x0) execution time : 0.010 s
Press any key to continue.
```



Очереди

Подобно стекам, очередь — хранит элемент последовательным образом. Существенное отличие от стека – использование FIFO (First in First Out) вместо LIFO.

Пример очереди – очередь людей. Последний занял последним и будешь, а первый первым ее и покинет.

Изображение очереди, в четыре элемента (1, 2, 3 и 4), где 1 находится наверху и будет удален первым

Основные операции

Enqueue() — вставляет элемент в конец очереди
Dequeue () — удаляет элемент из начала очереди
isEmpty () — возвращает значение true, если очередь пуста
Top () — возвращает первый элемент очереди

Графы

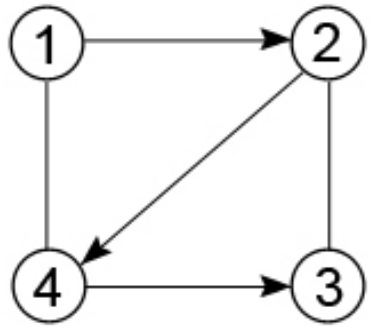
Граф - это набор узлов (вершин), которые соединены друг с другом в виде сети ребрами (дугами).

- **Ориентированный**, ребра являются направленными, т.е. существует только одно доступное направление между двумя связными вершинами.
- **Неориентированные**, к каждому из ребер можно осуществлять переход в обоих направлениях.
- **Смешанные**

Встречаются в таких формах как:

- Матрица смежности
- Список смежности

	1	2	3	4
1	0	1	0	1
2	0	0	1	1
3	0	1	0	0
4	1	0	1	0



Матрица смежности | Рисунок 2 | Соответствующий граф

Вычислительная сложность алгоритмов

- ***Вычислительной сложностью*** называют количество элементарных операций, затрачиваемых алгоритмом для решения конкретной задачи. Сложность зависит не только от размерности входных данных, но и от самих данных. Очевидно, что чем сложнее алгоритм в вычислительном плане, тем больше времени и вычислительных ресурсов потребует его выполнение.
- Различают ***временную и пространственную*** сложность. Первая определяет время, требуемое на решение задачи заданной размерности с помощью данного алгоритма, а вторая — количество требуемых ресурсов (памяти) при тех же условиях.
- Каждый вычислительный алгоритм может быть отнесен к одному из двух классов сложности. В данном случае это множество задач, для решения которых известны алгоритмы, схожие по трудоемкости.
- В **классе P** вычислительные затраты **линейно растут** с увеличением размерности. Например, время, требуемое на уборку придомовой территории, прямо пропорционально площади. Если ее увеличить вдвое, то и временные затраты возрастут в два раза.
- **Класс NP** включает задачи, для решения которых известны только алгоритмы, сложность которых **экспоненциально зависит** от размерности данных.

Пример задачи из сети

- «Типичным примером в данном случае служит алгоритм поиска кратчайшего пути. Представив карту города в виде сети, можно написать алгоритм для определения кратчайшего расстояния между двумя любыми точками этой сети. Чтобы не вычислять эти расстояния всякий раз, когда они нам нужны, мы можем вывести кратчайшие расстояния между всеми точками и сохранить результаты в таблице. Когда нам понадобится узнать кратчайшее расстояние между двумя заданными точками, мы можем просто взять готовое расстояние из таблицы.
- Результат будет получен мгновенно, но это потребует огромного объёма памяти. Карта большого города может содержать десятки тысяч точек. Тогда, описанная выше таблица, должна содержать более 10 млрд. ячеек. Т.е. для того, чтобы повысить быстродействие алгоритма, необходимо использовать дополнительные 10 Гб памяти.
- Из этой зависимости проистекает идея объёмно-временной сложности. При таком подходе алгоритм оценивается, как с точки зрении скорости выполнения, так и с точки зрения потреблённой памяти.
- Мы будем уделять основное внимание временной сложности, но, тем не менее, обязательно будем оговаривать и объём потребляемой памяти.»

Вычислительная временная сложность (time complexity) — это максимальное возможное количество выполненных алгоритмом элементарных операций, как функция от размера входных данных.

Вычислительная ёмкостная сложность (space complexity) — это максимальный возможный размер занятой алгоритмом дополнительной памяти, как функция от размера входных данных.

Эта самая функция возвращает максимум операций/памяти на всех входах такого размера. То есть для худшего, самого затратного случая.

Пример:

```
for (int i=0; i < n; i++)
    count++;
```

Посчитаем количество элементарных операций:

- *1 для int i = 0*
- *n+1 для i < n*
- *2n для i++ (что эквивалентно i = i + 1, а это две операции: присваивание и сложение)*
- *2n для count++*

Получаем, что временную сложность алгоритма ***C(n) = 2 + 5nC(n)=2+5n***.

Минусы точного расчёта функции сложности

- Некоторые «элементарные» операции, такие как sin или квадратный корень более дорогие, а функция сложности это не учитывает.
- Стоимость элементарных операций разная на разных процессорах.
- Стоимость элементарной операции зависит от контекста выполнения (от кэширования, переключения контекстов, работы конвейера команд у процессора, ...).
- Когда код написан на языке высокого уровня (C#, Python, JavaScript, ...) есть скрытая стоимость: выделение памяти, сборка мусора, JIT-компиляция и т.п.)

О-нотация

Определение 1. Говорят «$f(n) = O(g(n))$» тогда и только тогда, когда $\exists C > 0, \exists n_0 : \forall n > n_0 \to f(n) < Cg(n)$

Определение 2. Говорят «$f(n) = \Theta(g(n))$» тогда и только тогда, когда $f(n) = O(g(n))$ и одновременно $g(n) = O(f(n))$

Пример оценки:

1. $n = O(n)$
2. $n = O(n^2)$
3. $n = O(n^3)$
4. $2n + 5 = O(n)$
5. $n \log n = O(n^2)$
6. $2n^2 + 20n + 1 = O(n^2)$
7. $n^3 = \Theta(n^3)$
8. $n^3 + n^2 \log n = \Theta(n^3)$
9. $5n^2 + 3n + \log n + 1000 = \Theta(n^2)$

Оценка лучшего и среднего случая

При использовании алгоритма пузырьковой сортировки при каждом проходе по массиву попарно сравниваются элементы и, если нужно, меняются местами так, чтобы слева всегда был меньший элемент. Оценим сложность этого алгоритма.

- Допустим, на вход подается полностью отсортированный массив из 10 элементов, то есть n=10. Мы сравниваем все пары элементов от начала до конца, менять ничего не нужно. Таким образом за 10 операций сравнения на выходе мы получаем отсортированный массив.

- Если подать на вход массив, отсортированный в обратном порядке, то ему придется выполнить 10 проходов по массиву и выполнить по 10 перестановок на каждом проходе, то есть выполнить 100 операций сравнения и перестановок.
- Это худший случай для этого алгоритма, и мы видим, что в этом случае придется выполнить n^2 операций. Получается, что сложность этого алгоритма может быть оценена как $O(n^2)$, то есть в этом случае наша искомая функция будет выглядеть вот так: $g(n)=n^2$. Запись $O(n^2)$ как раз и говорит о том, что в худшем случае, для сортировки массива, состоящего из n элементов, нашему алгоритму придется выполнить n^2 операций.

Для обозначения оценки лучшего случая используется Ω-нотация, а для оценки среднего случая используется Θ-нотация.

Таким образом, для алгоритма сортировки пузырьком(для примера) верна будет запись:

- **$O(n^2)$** - в худшем случае, например, когда входные данные отсортированы в обратном порядке, вычислительная сложность алгоритма будет квадратично зависеть от количества элементов.
- **$\Theta(n^2)$** - в среднем случае, когда данные перемешаны в случайном порядке вычислительная сложность алгоритма так же будет квадратично зависеть от количества элементов.
- **$\Omega(n)$** - в лучшем случае, когда входные данные уже отсортированы, нам все равно потребуется выполнить один проход по массиву, то есть произвести n операций.

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|)

Куча	Временная сложность						
	Преобразование к куче	Поиск максимума	Извлечение максимума	Увеличить ключ	Вставить	Удалить	Слияние
Связный список (отсортированный)	-	O(1)	O(1)	O(n)	O(n)	O(1)	O(m+n)
Связный список (не отсортированный)	-	O(n)	O(n)	O(1)	O(1)	O(1)	O(1)
Бинарная куча	O(n)	O(1)	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(m+n)
Биномиальная куча	-	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))
Фибоначчева куча	-	O(1)	O(log(n))	O(1)*	O(1)	O(log(n))*	O(1)

Алгоритм	Структура данных	Временная сложность			Вспомогательные данные
		Лучшее	В среднем	В худшем	В худшем
Быстрая сортировка	Массив	O(n log(n))	O(n log(n))	O(n^2)	O(n)
Сортировка слиянием	Массив	O(n log(n))	O(n log(n))	O(n log(n))	O(n)
Пирамидальная сортировка	Массив	O(n log(n))	O(n log(n))	O(n log(n))	O(1)
Пузырьковая сортировка	Массив	O(n)	O(n^2)	O(n^2)	O(1)
Сортировка вставками	Массив	O(n)	O(n^2)	O(n^2)	O(1)
Сортировка выбором	Массив	O(n^2)	O(n^2)	O(n^2)	O(1)
Блочная сортировка	Массив	O(n+k)	O(n+k)	O(n^2)	O(nk)
Поразрядная сортировка	Массив	O(nk)	O(nk)	O(nk)	O(n+k)

Способ представления	Память	Добавление вершины	Добавление ребра	Удаление вершины	Удаление ребра	Проверка смежности вершин
Список смежности	O(|E| + |V|)	O(1)	O(1)	O(|E| + |V|)	O(|E|)	O(|V|)
Список инцидентности	O(|E| + |V|)	O(1)	O(1)	O(|E|)	O(|E|)	O(|E|)
Матрица смежности	O(|V|^2)	O(|V|^2)	O(1)	O(|V|^2)	O(1)	O(1)
Матрица инцидентности	O(|V||E|)	O(|V||E|)	O(|V||E|)	O(|V||E|)	O(|V||E|)	O(|E|)

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)

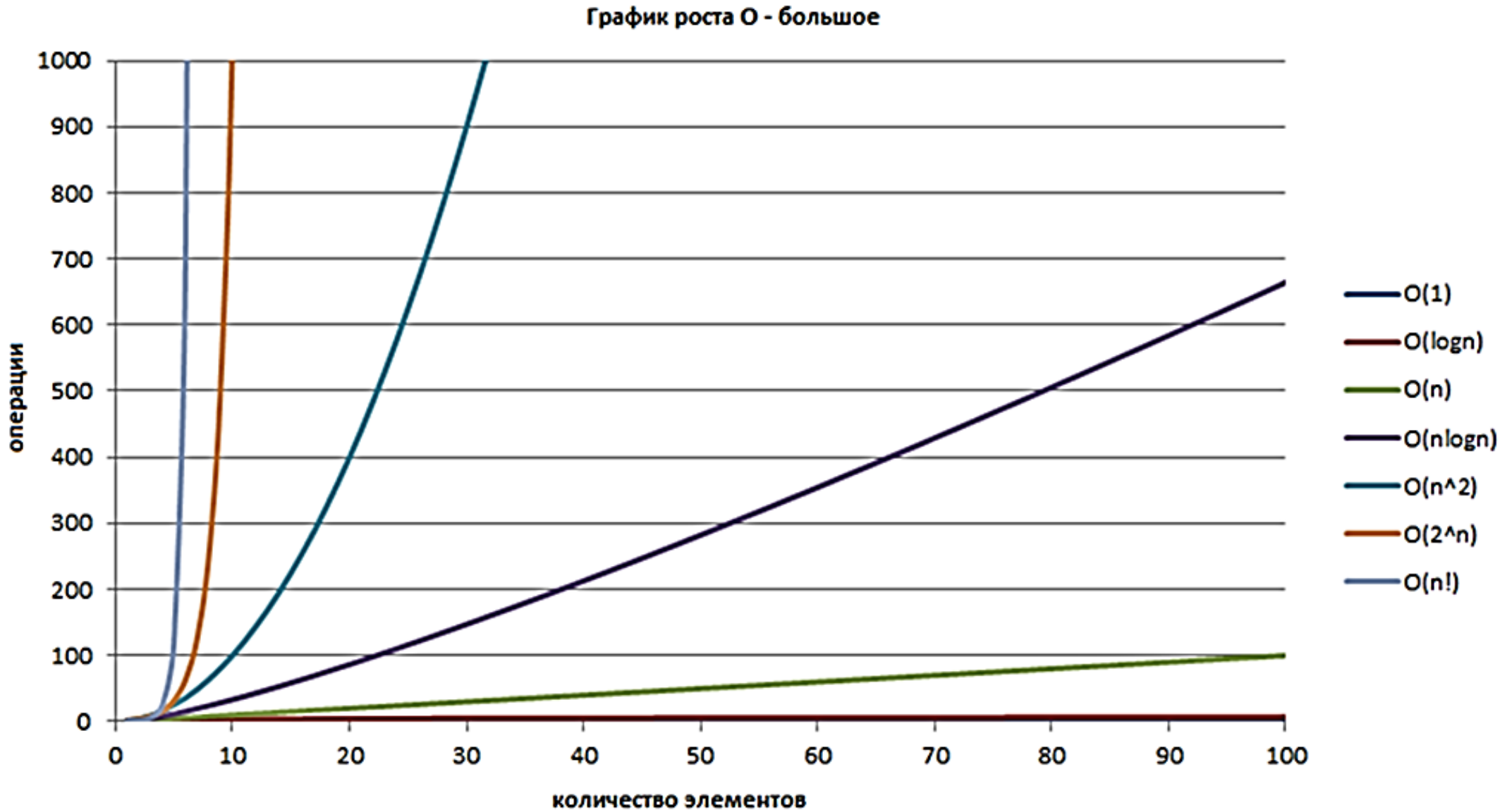


График роста O - большое
операции
1000
900
800
700
600
500
400
300
200
100
0
0
10
20
30
40
50
60
70
80
90
100
количество элементов
O(1)
O(logn)
O(n)
O(nlogn)
O(n^2)
O(2^n)
O(n!)

Раздел 2. Список однонаправленный и двунаправленный. Способы организации и обработки данных списка на программном языке высокого уровня, на примере языка C#/C++. Понятие стек и очередь. Способы программной организации стека и очереди, и обработка данных.

Динамические структуры данных

Динамические структуры данных в процессе существования в памяти могут изменять не только число составляющих их элементов, но и характер связей между элементами. При этом не учитывается изменение содержимого самих элементов данных.

Динамическая структура данных имеет следующие характеристики:

- она не имеет имени;
- ей выделяется память в процессе выполнения программы;
- количество элементов структуры может не фиксироваться;
- размерность структуры может меняться в процессе выполнения программы;
- в процессе выполнения программы может меняться характер взаимосвязи между элементами структуры.

Каждой динамической структуре данных сопоставляется статическая переменная ***типа указат***ель (ее значение – адрес этого объекта), посредством которой осуществляется доступ к динамической структуре.

Динамические структуры данных

Указатели – это статические величины, поэтому они требуют описания. Его значение – адрес объекта.

Применяют динамические структуры данных в следующих случаях:

- Используются переменные большого размера только для одной части задачи, а для других они уже не нужны. Например, массив большой размерности.
- В процессе работы программы нужен массив, список или иная структура, размер которой изменяется в широких пределах и трудно предсказуем.
- Когда размер данных, обрабатываемых в программе, превышает объем сегмента данных.

Динамические структуры данных

Для установления связи между элементами динамической структуры используются указатели, через которые устанавливаются явные связи между элементами. Такое представление данных в памяти называется связным.

Достоинства связного представления данных – в возможности обеспечения значительной изменчивости структур:

- размер структуры ограничивается только доступным объемом машинной памяти;
- при изменении логической последовательности элементов структуры требуется не перемещение данных в памяти, а только коррекция указателей;
- большая гибкость структуры.

Вместе с тем, связное представление не лишено и недостатков, основными из которых являются следующие:

- на поля, содержащие указатели для связывания элементов друг с другом, расходуется дополнительная память;
- доступ к элементам связной структуры может быть менее эффективным по времени.

Классификация динамических структур данных

Во многих задачах требуется использовать данные, у которых конфигурация, размеры и состав могут меняться в процессе выполнения программы. Для их представления используют динамические информационные структуры. К таким структурам относят:

- однонаправленные (односвязные) списки;
- двунаправленные (двусвязные) списки;
- циклические списки;
- стек;
- дек;
- очередь;
- бинарные деревья.

Списки

Списком называется упорядоченное множество, состоящее из переменного числа элементов, к которым применимы операции включения, исключения. Список, отражающий отношения соседства между элементами, называется **линейным**.

Линейные связные списки являются простейшими динамическими структурами данных. Из всего многообразия связанных списков можно выделить следующие основные:

- однонаправленные (односвязные) списки;
- двунаправленные (двусвязные) списки;
- циклические (кольцевые) списки.

В основном они отличаются видом взаимосвязи элементов и/или допустимыми операциями.

Однонаправленный (односвязный) список

- **Однонаправленный (односвязный) список** – это структура данных, представляющая собой последовательность элементов, в каждом из которых хранится значение и указатель на следующий элемент списка. В последнем элементе указатель на следующий элемент равен NULL.

Односвязный циклический список (ОЦС) - каждый узел ОЦС содержит 1 поле указателя на следующий узел. Поле указателя последнего узла содержит адрес первого узла (корня списка).

Двусвязный список

Двусвязный линейный список (ДЛС)- каждый узел ДЛС содержит два поля указателей: на следующий и на предыдущий узел.

Двусвязный циклический список (ДЦС) - каждый узел ДЦС содержит два поля указателей: на следующий и на предыдущий узел.

Программная реализация

Описание простейшего элемента ОЛС такого списка выглядит следующим образом:

struct имя_типа { информационное *поле*; адресное *поле*; };

где информационное поле – это *поле* любого, ранее объявленного или стандартного, типа;

адресное поле – это *указатель* на *объект* того же типа, что и определяемая структура, в него записывается *адрес* следующего элемента списка.

Например:

```
struct Node {
        int key;//информационное поле
        Node*next;//адресное поле
       };
```

Информационных полей может быть несколько.

Например:

```
struct point {
         char*name;//информационное поле
         int age;//информационное поле
         point*next;//адресное поле
        };
```

Каждый *элемент списка* содержит *ключ*, который идентифицирует этот элемент. *Ключ* обычно бывает либо целым числом, либо строкой.

Создание однонаправленного списка

//создание однонаправленного списка (добавления в конец)

```
void Make_Single_List(int n,Single_List** Head){
 if (n > 0) {
   (*Head) = new Single_List();
   //выделяем память под новый элемент
   cout << "Введите значение ";
   cin >> (*Head)->Data;
   //вводим значение информационного поля
   (*Head)->Next=NULL;//обнуление адресного поля
    Make_Single_List(n-1,&((*Head)->Next));
 }
}
```

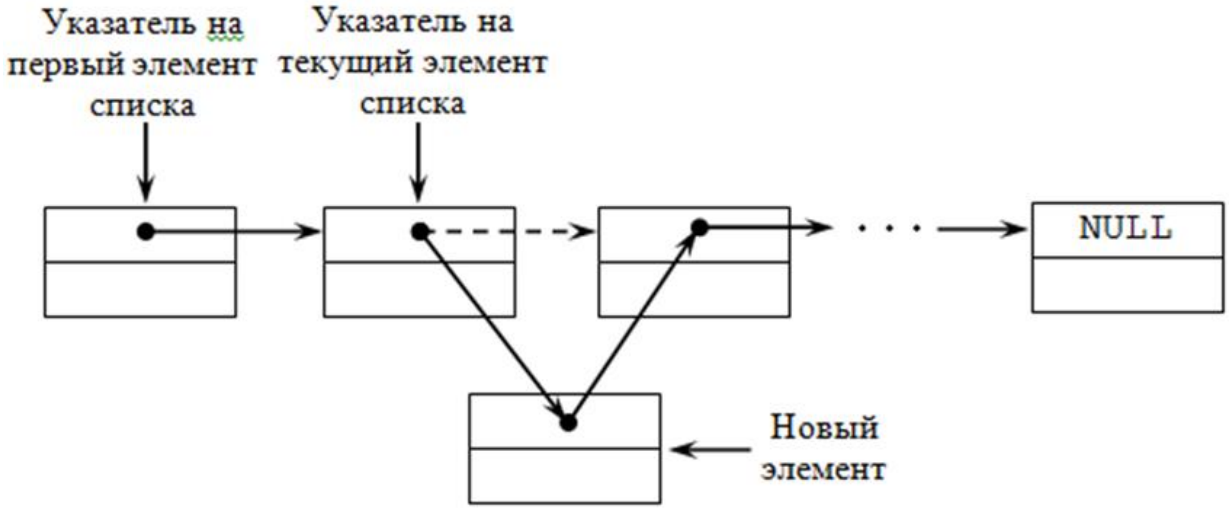
Печать (просмотр) однонаправленного списка

//печать однонаправленного списка

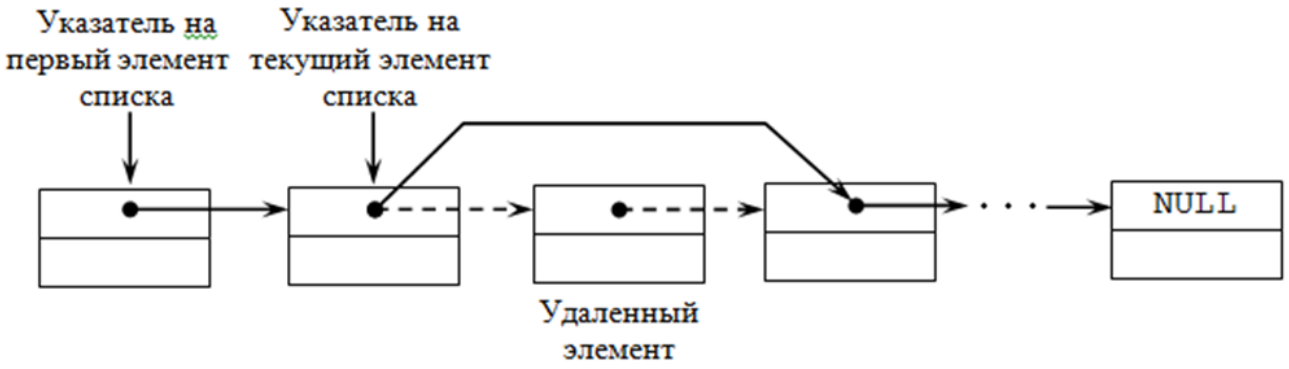
```
void Print_Single_List(Single_List* Head) {
 if (Head != NULL) {
   cout << Head->Data << "\t";
   Print_Single_List(Head->Next);
   //переход к следующему элементу
 }
 else cout << "\n";
}
```

Вставка элемента в однонаправленный список

В динамические структуры легко добавлять элементы, так как для этого достаточно изменить значения адресных полей. Вставка первого и последующих элементов списка отличаются друг от друга. Поэтому в функции, реализующей данную операцию, сначала осуществляется проверка, на какое место вставляется элемент. Далее реализуется соответствующий алгоритм добавления



Удаление элемента из однонаправленного списка



Из динамических структур можно удалять элементы, так как для этого достаточно изменить значения адресных полей. Операция удаления элемента однонаправленного списка осуществляет удаление элемента, на который установлен указатель текущего элемента. После удаления указатель текущего элемента устанавливается на предшествующий элемент списка или на новое начало списка, если удаляется первый.

Алгоритмы удаления первого и последующих элементов списка отличаются друг от друга. Поэтому в функции, реализующей данную операцию, осуществляется проверка, какой элемент удаляется. Далее реализуется соответствующий алгоритм удаления.

```
/*удаление элемента с заданным номером из однонаправленного списка*/

Single_List*
Delete_Item_Single_List(Single_List* Head,
      int Number){
  Single_List *ptr;//вспомогательный указатель
  Single_List *Current = Head;
  for (int i = 1; i < Number && Current != NULL;
i++)
    Current = Current->Next;
  if    (Current    !=    NULL){//проверка    на
корректность
    if   (Current   ==   Head){//удаляем   первый
элемент
      Head = Head->Next;
      delete(Current);
      Current = Head;
    }
    else {//удаляем непервый элемент
      ptr = Head;
      while (ptr->Next != Current)
        ptr = ptr->Next;
      ptr->Next = Current->Next;
      delete(Current);
      Current=ptr;
    }
  }
  return Head;
}
```

Поиск элемента в однонаправленном списке

Операция поиска элемента в списке заключается в последовательном просмотре всех элементов списка до тех пор, пока текущий элемент не будет содержать заданное *значение* или пока не будет достигнут конец списка. В последнем случае фиксируется отсутствие искомого элемента в списке (*функция* принимает *значение* false).

```
//поиск элемента в однонаправленном списке
bool  Find_Item_Single_List(Single_List*  Head, int DataItem){
  Single_List *ptr; //вспомогательным указатель
  ptr = Head;
  while (ptr != NULL){//пока не конец списка
    if (DataItem == ptr->Data) return true;
    else ptr = ptr->Next;
  }
  return false;
}
```

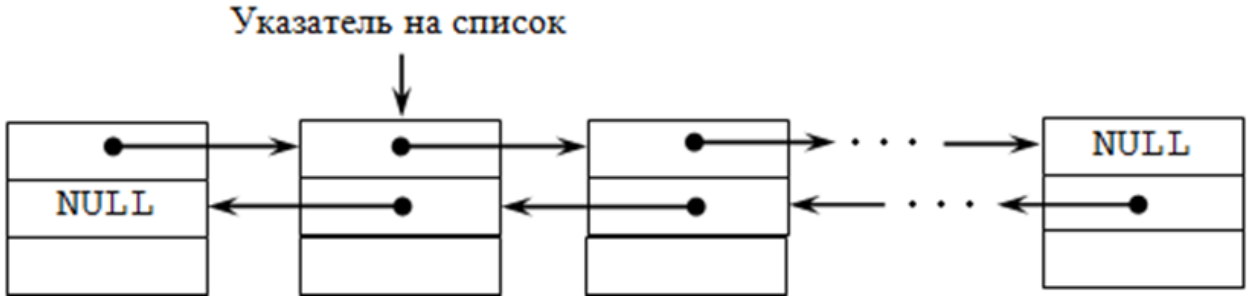
Удаление однонаправленного списка

Операция удаления списка заключается в освобождении динамической памяти. Для данной *операции* организуется *функция*, в которой нужно переставлять *указатель* на следующий *элемент списка* до тех пор, пока *указатель* не станет равен NULL, то есть не будет достигнут конец списка. Реализуем *рекурсивную функцию*.

```
/*освобождение памяти, выделенной под однонаправленный список*/
void Delete_Single_List(Single_List* Head){
  if (Head != NULL){
    Delete_Single_List(Head->Next);
    delete Head;
  }
}
```

Двунаправленные (двусвязные) списки

- **Двунаправленный (двусвязный) список** – это структура данных, состоящая из последовательности элементов, каждый из которых содержит информационную часть и два указателя на соседние элементы. При этом два соседних элемента должны содержать взаимные ссылки друг на друга.



Описание простейшего элемента такого списка выглядит следующим образом:

```
struct имя_типа {
        информационное поле;
        адресное поле 1;
        адресное поле 2;
       };
```

где информационное поле – это поле любого, ранее объявленного или стандартного, типа;

адресное поле 1 – это указатель на объект того же типа, что и определяемая структура, в него записывается адрес следующего элемента списка ;

адресное поле 2 – это указатель на объект того же типа, что и определяемая структура, в него записывается адрес предыдущего элемента списка.

Двунаправленные (двусвязные) списки

Основные операции, осуществляемые с двунаправленными списками, такие как:

- создание списка;
- печать (просмотр) списка;
- вставка элемента в список;
- удаление элемента из списка;
- поиск элемента в списке;
- проверка пустоты списка;
- удаление списка.

Для описания алгоритмов этих основных операций используется следующее объявление:

```
struct Double_List {//структура данных
                    int Data; //информационное поле
                    Double_List *Next, //адресное поле
                                *Prior; //адресное поле
                   };
. . . . . .
Double_List *Head; //указатель на первый элемент списка
. . . . . .
Double_List *Current;
//указатель на текущий элемент списка (при необходимости)
```

Создание двунаправленного списка

Для того, чтобы создать *список*, нужно создать сначала первый *элемент списка*, а затем при помощи функции добавить к нему остальные элементы. Добавление может выполняться как в начало, так и в конец списка. Реализуем *рекурсивную функцию*.

```
//создание двунаправленного списка (добавления в конец)
void Make_Double_List(int n,Double_List** Head,
          Double_List* Prior){
  if (n > 0) {
    (*Head) = new Double_List();
     //выделяем память под новый элемент
    cout << "Введите значение ";
    cin >> (*Head)->Data;
    //вводим значение информационного поля
    (*Head)->Prior = Prior;
    (*Head)->Next=NULL;//обнуление адресного поля
    Make_Double_List(n-1,&((*Head)->Next),(*Head));
  }
  else (*Head) = NULL;
}
```

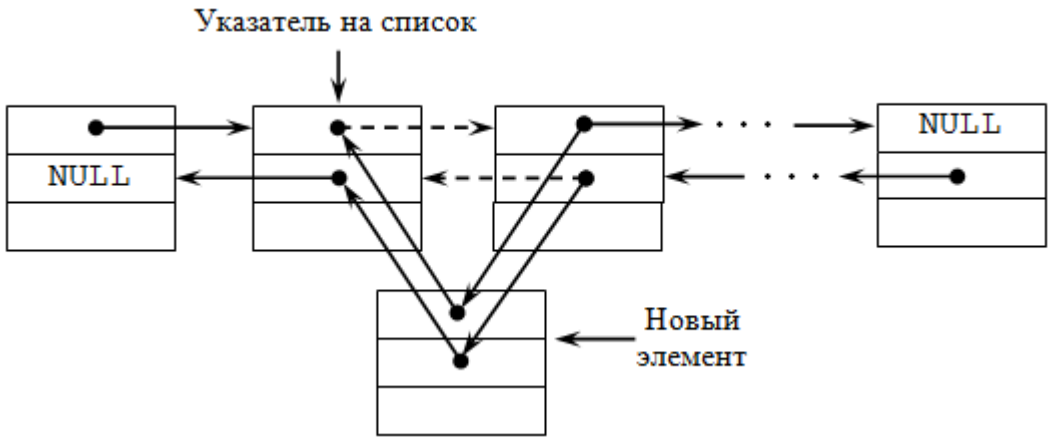
Печать (просмотр) двунаправленного списка

Операция печати списка для *двунаправленного* списка реализуется абсолютно аналогично соответствующей функции для однонаправленного списка. Просматривать *двунаправленный список* можно в обоих направлениях.

```
//печать двунаправленного списка

void Print_Double_List(Double_List* Head) {
  if (Head != NULL) {
    cout << Head->Data << "\t";
    Print_Double_List(Head->Next);
    //переход к следующему элементу
  }
  else cout << "\n";
}
```

Вставка элемента в двунаправленный список

В динамические структуры легко добавлять элементы, так как для этого достаточно изменить значения адресных полей. Операция вставки реализовывается аналогично функции вставки для однонаправленного списка, только с учетом особенностей *двунаправленного* списка.



Добавление элемента в двунаправленный список

```
//вставка элемента с заданным номером в двунаправленный список
Double_List* Insert_Item_Double_List(Double_List* Head,
      int Number, int DataItem){
  Number--;
  Double_List *NewItem=new(Double_List);
  NewItem->Data=DataItem;
  NewItem->Prior=NULL;
  NewItem->Next = NULL;
  if (Head == NULL) {//список пуст
    Head = NewItem;
  }
  else {//список не пуст
    Double_List *Current=Head;
    for(int i=1; i < Number && Current->Next!=NULL; i++)
    Current=Current->Next;
    if (Number == 0){
    //вставляем новый элемент на первое место
      NewItem->Next = Head;
      Head->Prior = NewItem;
      Head = NewItem;
    }
    else {//вставляем новый элемент на непервое место
      if (Current->Next != NULL) Current->Next->Prior =
NewItem;
      NewItem->Next = Current->Next;
      Current->Next = NewItem;
        NewItem->Prior = Current;
      Current = NewItem;
    }
  }
  return Head;
}
```

Удаление элемента из двунаправленного списка

```
Double_List* Delete_Item_Double_List(Double_List* Head,
      int Number){
  Double_List *ptr;//вспомогательный указатель
  Double_List *Current = Head;
  for (int i = 1; i < Number && Current != NULL; i++)
    Current = Current->Next;
  if (Current != NULL){//проверка на корректность
    if (Current->Prior == NULL){//удаляем первый элемент
      Head = Head->Next;
      delete(Current);
      Head->Prior = NULL;
      Current = Head;
    }
    else {//удаляем непервый элемент
      if (Current->Next == NULL) {
      //удаляем последний элемент
        Current->Prior->Next = NULL;
        delete(Current);
        Current = Head;
      }
      else {//удаляем непервый и непоследний элемент
        ptr = Current->Next;
        Current->Prior->Next =Current->Next;
        Current->Next->Prior =Current->Prior;
        delete(Current);
        Current = ptr;
      }
    }
  }
  return Head;
}
```

Поиск элемента в двунаправленном списке

Операция поиска элемента в двунаправленном списке реализуется абсолютно аналогично соответствующей функции для однонаправленного списка. *Поиск* элемента в двунаправленном списке можно вести:

а) просматривая элементы от начала к концу списка;

б) просматривая элементы от конца списка к началу;

в) просматривая *список* в обоих направлениях одновременно: от начала к середине списка и от конца к середине (учитывая, что элементов в списке может быть четное или нечетное количество).

```
//поиск элемента в двунаправленном списке
bool Find_Item_Double_List(Double_List* Head,
    int DataItem){
  Double_List *ptr; //вспомогательный указатель
  ptr = Head;
  while (ptr != NULL){//пока не конец списка
    if (DataItem == ptr->Data) return true;
    else ptr = ptr->Next;
    }
  return false;
}
```

Проверка пустоты двунаправленного списка

Операция проверки *двунаправленного* списка на пустоту осуществляется аналогично проверки однонаправленного списка.

```
//проверка пустоты двунаправленого списка
bool Empty_Double_List(Double_List* Head){
  if (Head!=NULL)  return false;
  else  return true;
}
```

Удаление двунаправленного списка

Операция удаления *двунаправленного* списка реализуется аналогично удалению однонаправленного списка.

```
//освобождение памяти, выделенной под двунаправленный список
void Delete_Double_List(Double_List* Head){
  if (Head != NULL){
    Delete_Double_List(Head->Next);
    delete Head;
  }
}
}
```

Понятие стек Способы программной организации стека и очереди, и обработка данных.

Стек (англ. stack – стопка) – это структура данных, в которой новый элемент всегда записывается в ее начало (вершину) и очередной читаемый элемент также всегда выбирается из ее начала. В стеках используется метод доступа к элементам LIFO (Last Input – First Output, "последним пришел – первым вышел"). Чаще всего принцип работы стека сравнивают со стопкой тарелок: чтобы взять вторую сверху, нужно сначала взять верхнюю.

Описание стека выглядит следующим образом:

```
struct имя_типа {
        информационное поле;
        адресное поле;
       };
```

где информационное поле – это поле любого ранее объявленного или стандартного типа;
адресное поле – это указатель на объект того же типа, что и определяемая структура, в него записывается адрес следующего элемента стека.
Например:

```
struct list {
        type  pole1;
        list *pole2;
       } stack;
```

Стек

Основные операции, производимые со стеком:

- создание стека;
- печать (просмотр) стека;
- добавление элемента в вершину стека;
- извлечение элемента из вершины стека;
- проверка пустоты стека;
- очистка стека.

Реализацию этих операций рассмотрим в виде соответствующих функций, которые, в свою очередь, используют функции операций с линейным однонаправленным списком. Обратим внимание, что в функции создания стека используется функция добавления элемента в вершину стека.

```
//создание стека
void Make_Stack(int n, Stack* Top_Stack){
  if (n > 0) {
    int tmp;//вспомогательная переменная
    cout << "Введите значение ";
    cin >> tmp; //вводим значение информационного поля
    Push_Stack(tmp, Top_Stack);
    Make_Stack(n-1,Top_Stack);
  }
}
```

```
//печать стека
void Print_Stack(Stack* Top_Stack){
  Print_Single_List(Top_Stack->Top);
}

//добавление элемента в вершину стека
void Push_Stack(int NewElem, Stack* Top_Stack){
  Top_Stack->Top =Insert_Item_Single_List(Top_Stack-
>Top,1,NewElem);
}
```

```
//проверка пустоты стека
bool Empty_Stack(Stack* Top_Stack){
  return Empty_Single_List(Top_Stack->Top);
}
```

```
//очистка стека
void Clear_Stack(Stack* Top_Stack){
  Delete_Single_List(Top_Stack->Top);
}
```

```
//извлечение элемента из вершины стека
int Pop_Stack(Stack* Top_Stack){
  int NewElem = NULL;
  if (Top_Stack->Top != NULL) {
    NewElem = Top_Stack->Top->Data;
    Top_Stack->Top =
Delete_Item_Single_List(Top_Stack-
>Top,0);
    //удаляем вершину
  }
  return NewElem;
}
```

Очередь

- **Очередь** – это структура данных, представляющая собой последовательность элементов, образованная в порядке их поступления. Каждый новый элемент размещается в конце очереди; элемент, стоящий в начале очереди, выбирается из нее первым. В очереди используется принцип доступа к элементам FIFO (First Input – First Output, "первый пришёл – первый вышел") .В очереди доступны два элемента (две позиции): начало очереди и конец очереди. Поместить элемент можно только в конец очереди, а взять элемент только из ее начала. Примером может служить обыкновенная очередь в магазине.

Описание очереди выглядит следующим образом:

```
struct имя_типа {
        информационное поле;
        адресное поле1;
        адресное поле2;
       };
```

где информационное поле – это *поле* любого, ранее объявленного или стандартного, типа;

адресное поле1, адресное поле2 – это указатели на объекты того же типа, что и определяемая структура, в них записываются адреса первого и следующего элементов очереди.

Описание очереди

Например:

1 способ: адресное *поле* ссылается на объявляемую структуру.

```
struct list2 {
        type  pole1;
        list2 *pole1, *pole2;
       }
```

2 способ: адресное *поле* ссылается на ранее объявленную структуру.

```
struct list1 {
        type  pole1;
        list1  *pole2;
       }
struct ch3 {
       list1 *beg, *next ;
      }
```

Описание элементов очереди

Описание элементов очереди аналогично описанию элементов линейного двунаправленного списка. Поэтому объявим очередь через объявление линейного двунаправленного списка:

```
struct Queue {
        Double_List *Begin;//начало очереди
        Double_List *End; //конец очереди
       };
. . . . . .
Queue *My_Queue;//указатель на очередь
```

Основные операции, производимые с очередью:

- создание очереди;
- печать (просмотр) очереди;
- добавление элемента в конец очереди;
- извлечение элемента из начала очереди;
- проверка пустоты очереди;
- очистка очереди.

```
//создание очереди
void Make_Queue(int n, Queue* End_Queue){
  Make_Double_List(n,&(End_Queue->Begin),NULL);
  Double_List *ptr; //вспомогательный указатель
  ptr = End_Queue->Begin;
  while (ptr->Next != NULL)
    ptr = ptr->Next;
  End_Queue->End = ptr;
}

//печать очереди
void Print_Queue(Queue* Begin_Queue){
  Print_Double_List(Begin_Queue->Begin);
}

//добавление элемента в конец очереди
void Add_Item_Queue(int NewElem, Queue* End_Queue){
  End_Queue->End = Insert_Item_Double_List(End_Queue->End,
     0, NewElem)->Next;
}
```

```
//извлечение элемента из начала очереди
int Extract_Item_Queue(Queue* Begin_Queue){
  int NewElem = NULL;
  if (Begin_Queue->Begin != NULL) {
   NewElem = Begin_Queue->Begin->Data;
   Begin_Queue-
>Begin=Delete_Item_Double_List(Begin_Queue-
>Begin,0);
   //удаляем вершину
  }
  return NewElem;
}

//проверка пустоты очереди
bool Empty_Queue(Queue* Begin_Queue){
  return Empty_Double_List(Begin_Queue-
>Begin);
}

//очистка очереди
void Clear_Queue(Queue* Begin_Queue){
  return Delete_Double_List(Begin_Queue-
>Begin);
}
```

Раздел 3. Графы. Основные понятия и определения: граф, ориентированный, неориентированный граф, петля, путь в графе, ребра в графе.
Способы задания графов. Матрица инцидентности, матрица смежности, матрица весов, список ребер, список смежности.
Какие структуры можно использовать для программной работы с графами. Поиск в графе. Поиск в ширину, поиск в глубину.
Нахождение кратчайших путей. Алгоритм Дейкстры, алгоритм Флойда. Нахождение центра графа. Задача коммивояжера.
Эйлеровы пути и циклы

Графы. Основные понятия и определения.

Граф состоит из вершин(точек) и ребер, соединяющих вершины. Выделяют три основных вида графов:

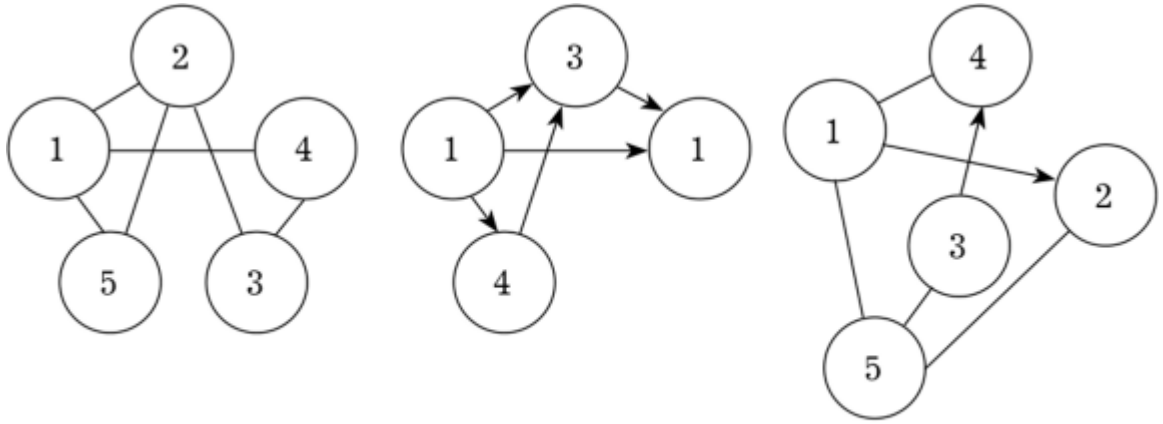
- ориентированный – ребра имеют вид дуги, направление имеет значение.
- неориентированный – все ребра являются звеньями.
- смешанный.

Степенью вершины называется количество ребер, входящих в вершину.

Путем или цепью в графе называют конечную последовательность вершин, в которой каждая вершина (кроме последней) соединена со следующей в последовательности вершин ребром.

Циклом называют путь, в котором первая и последняя вершины совпадают. Путь или цикл называют простым, если ребра в нем не повторяются.

Если в графе любые две вершины соединены путем, то такой ***граф называется связным.***



а) неориентированный граф б) ориентированный граф в) смешанный граф

Два графа называются ***изоморфными***, если у них поровну вершин. При этом вершины каждого графа можно занумеровать числами так, чтобы вершины первого графа были соединены ребром тогда и только тогда, когда соединены ребром соответствующие занумерованные теми же числами вершины второго графа.

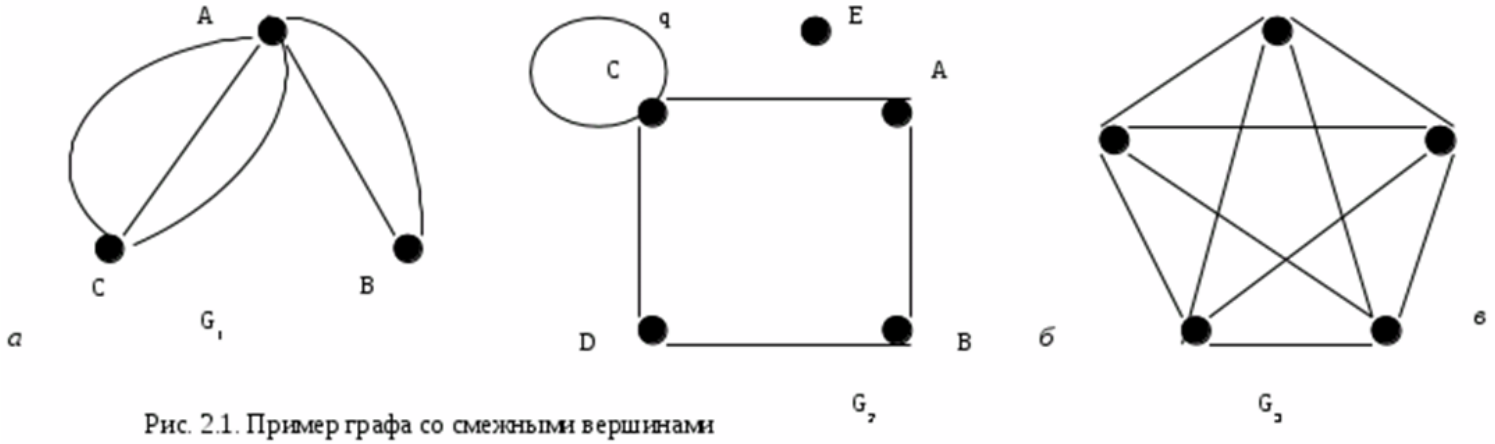


Рис. 2.1. Пример графа со смежными вершинами
а – со смежными вершинами; б – с петлёй, в - полный

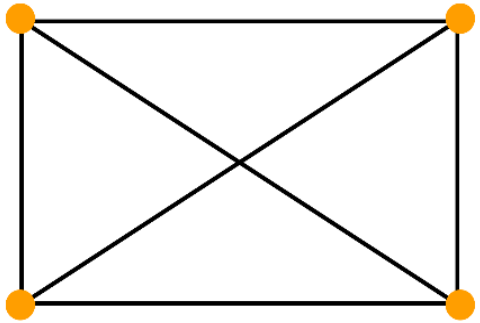
Эйлеров граф отличен тем, что в нем можно обойти все вершины и при этом пройти одно ребро только один раз. В нём каждая вершина должна иметь только чётное число рёбер.

Пример. Является ли полный граф с одинаковым числом n рёбер, которым инцидентна каждая вершина, эйлеровым графом?

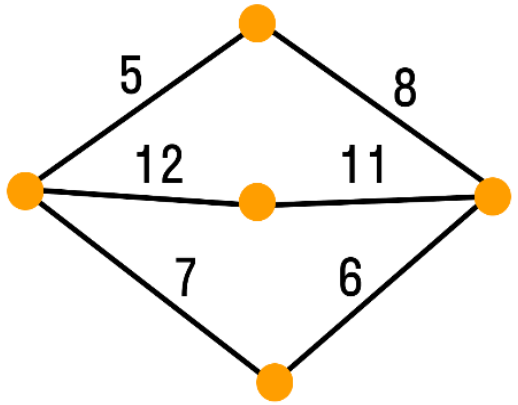
Ответ. Если n — нечётное число, то каждая вершина инцидентна n - 1 ребрам. В таком случае наш граф — эйлеровый.

Гамильтоновым графом называется граф, содержащий гамильтонов цикл. **Гамильтоновым циклом** называется простой цикл, который проходит через все вершины рассматриваемого графа.

Говоря проще, гамильтонов граф — это такой граф, в котором можно обойти все вершины, и каждая вершина при обходе повторяется лишь один раз.



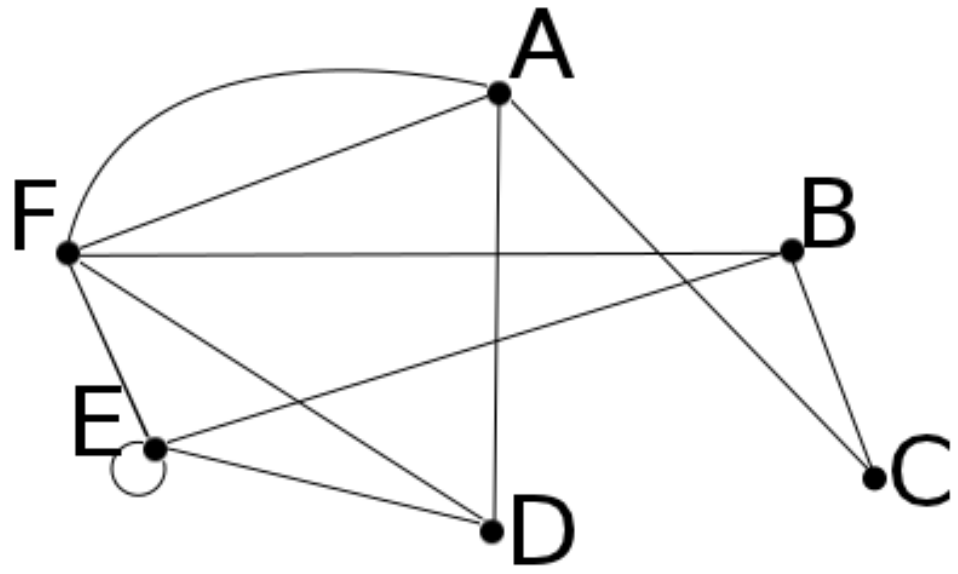
Взвешенным графом называется граф, вершинам и/или ребрам которого присвоены «весы» — обычно некоторые числа. Пример взвешенного графа — транспортная сеть, в которой ребрам присвоены весы: они показывают стоимость перевозки груза по ребру и пропускные способности дуг.



Способы задания графов. Матрица инцидентности, матрица смежности, матрица весов, список ребер, список смежности.

Матрицей смежности для графа G=(V,E) называется квадратная матрица A=(a_{ij}), строкам и столбцам которой соответствуют вершины графа. Для неориентированного графа число a_{ij} равно числу ребер, инцидентных Vi и Vj. Для орграфа число a_{ij} равно числу ребер с началом в Vi и концом в Vj.

$$A = \begin{pmatrix} 0 & 0 & 1 & 1 & 0 & 2 \\ 0 & 0 & 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 1 & 1 & 1 \\ 2 & 1 & 0 & 1 & 1 & 0 \end{pmatrix}$$



Списком ребер графа называется таблица, состоящая из двух столбцов, в которой на пересечении i-й строки и первого (левого) столбца записывается ребро e_i, а на пересечении i -й строки и второго (правого) столбца записываются вершины, инцидентные ребру e_i. Для того, чтобы список ребер однозначно задавал граф, необходимо помимо списка ребер указать множество всех изолированных вершин этого графа.

1	B, C
2	A, C
3	B, F
4	A, F
5	A, F
6	A, D
7	D, F
8	D, E
9	E, F
10	B, E
11	E, E

Матрицей инцидентности для графа G=(V,E) называется матрица B=(b_{ij}), столбцам которой соответствуют вершины графа, а строкам — ребра. Число орграфа b_{ij} равно:

$$b_{ij} = \begin{cases} 1, \text{ если } e_j \text{ исходит из } V_i \\ -1, \text{ если } e_j \text{ заходит в } V_i \\ 0, \text{ если } e_j \text{ не инцидентно } V_i \end{cases}$$

Для неориентированного графа b_{ij} равно:

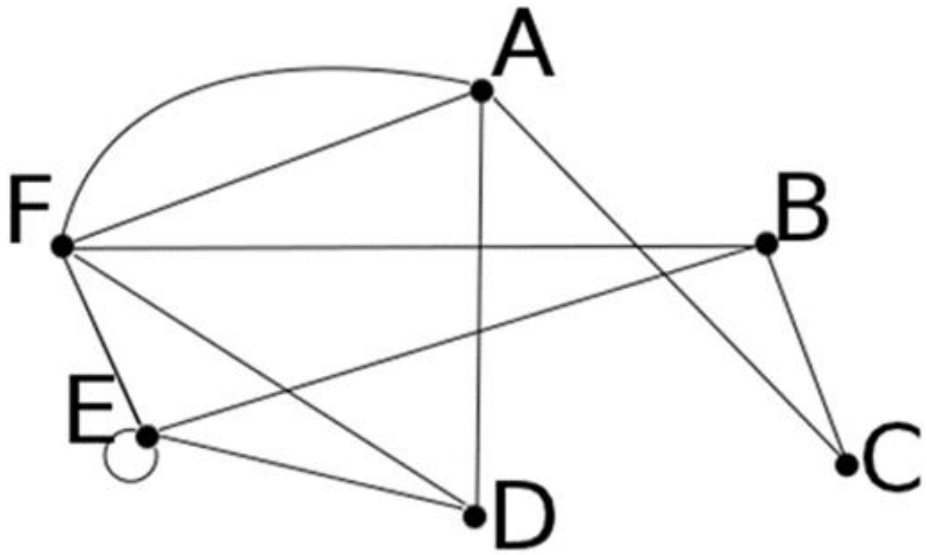
$$b_{ij} = \begin{cases} 1, \text{ если } e_j \text{ инцидентно } V_i \\ 0, \text{ если } e_j \text{ не инцидентно } V_i \end{cases}$$

У матрицы инцидентности характеризуются:

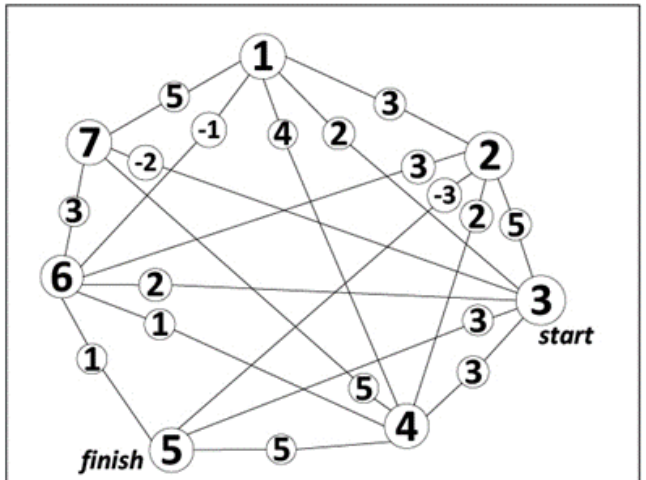
- в каждой строке должны стоять две единицы, а все остальные символы — нули;
- она не используется для графов с петлями.

На примере ниже такая матрица после удаления петли E графа рядом.

$$B = \begin{pmatrix} 0 & 1 & 1 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 0 & 1 & 0 \end{pmatrix}$$



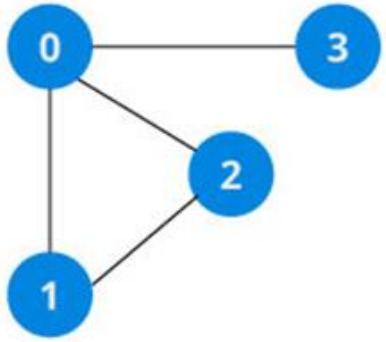
Матрица весов строится для взвешенного графа аналогично матрице смежности, однако указывается вес

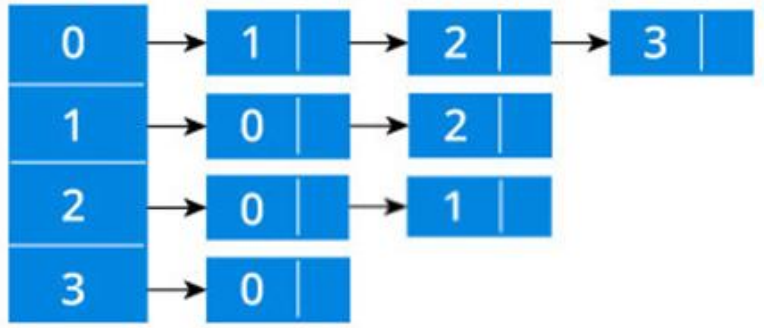


D =

0	3	2	4	0	-1	5
3	0	5	2	-3	3	0
2	5	0	3	3	2	-2
4	2	3	0	5	1	5
0	-3	3	5	0	1	0
-1	3	2	1	1	0	3
5	0	-2	5	0	3	0

Граф и его эквивалентное ***представление списка смежности*** показаны ниже.





Какие структуры можно использовать для программной работы с графами. Поиск в графе. Поиск в ширину, поиск в глубину. Нахождение кратчайших путей. Алгоритм Дейкстры, алгоритм Флойда. Нахождение центра графа. Задача коммивояжера. Эйлеровы пути и циклы.

Структуры данных для работы с графами

- Мы приближаемся к основному определению графа - совокупности вершин и ребер {V, E}. Для простоты мы используем немаркированный граф, а не помеченный, то есть вершины идентифицируются по их индексам 0,1,2,3.
- Давайте рассмотрим структуру данных ниже.

```
struct node
{
    int vertex;
    struct node* next;
};
struct Graph
{
    int numVertices;
    struct node** adjLists;
};
```

- Нам нужно сохранить указатель struct node . Потому, как мы не знаем, сколько вершин будет в графе, и поэтому не можем создать массив связанных списков во время компиляции.

Код списка смежности в C++

```cpp
#include <iostream>
#include <list>
using namespace std;

class Graph
{
    int numVertices;
    list *adjLists;

public:
    Graph(int V);
    void addEdge(int src, int dest);
};

Graph::Graph(int vertices)
{
    numVertices = vertices;
    adjLists = new list[vertices];
}

void Graph::addEdge(int src, int dest)
{
    adjLists[src].push_front(dest);
}


int main()
{
    Graph g(4);
    g.addEdge(0, 1);
    g.addEdge(0, 2);
    g.addEdge(1, 2);
    g.addEdge(2, 3);

    return 0;
}
```

Матрица смежности в C++ представляется через двумерный массив.

```cpp
#include <iostream>
using namespace std;
class Graph {
private:
	bool** adjMatrix;
	int numVertices;
public:
	Graph(int numVertices) {
		this->numVertices = numVertices;
		adjMatrix = new bool*[numVertices];
		for (int i = 0; i < numVertices; i++) {
			adjMatrix[i] = new bool[numVertices];
			for (int j = 0; j < numVertices; j++)
				adjMatrix[i][j] = false;
		}
	}

	void addEdge(int i, int j) {
			adjMatrix[i][j] = true;
			adjMatrix[j][i] = true;
	}

	void removeEdge(int i, int j) {
			adjMatrix[i][j] = false;
			adjMatrix[j][i] = false;
	}

	bool isEdge(int i, int j) {
			return adjMatrix[i][j];
	}

	void toString() {
		for (int i = 0; i < numVertices; i++) {
				cout << i << " : ";
				for (int j = 0; j < numVertices; j++)
					cout << adjMatrix[i][j] << " ";
				cout << "\n";
		}
	}

	~Graph() {
		for (int i = 0; i < numVertices; i++)
			delete[] adjMatrix[i];
		delete[] adjMatrix;
	}
};
int main(){
	Graph g(4);

	g.addEdge(0, 1);
	g.addEdge(0, 2);
	g.addEdge(1, 2);
	g.addEdge(2, 0);
	g.addEdge(2, 3);
	g.toString();
	/* Outputs
	   0: 0 1 1 0
	   1: 1 0 1 0
	   2: 1 1 0 1
	   3: 0 0 1 0
	  */
}
```

Поиск в графе

Существует много алгоритмов на графах, в основе которых лежит систематический перебор вершин графа, такой что каждая вершина просматривается (посещается) в точности один раз. Поэтому важной задачей является нахождение хороших **методов поиска в графе**.

Под обходом графов (**поиском на графах**) понимается процесс систематического просмотра всех ребер или вершин графа с целью отыскания ребер или вершин, удовлетворяющих некоторому условию.

При решении многих задач, использующих графы, необходимы эффективные методы регулярного обхода вершин и ребер графов. К стандартным и наиболее распространенным методам относятся:

- поиск в глубину (Depth First Search, DFS);
- поиск в ширину (Breadth First Search, BFS).

Поиск в глубину

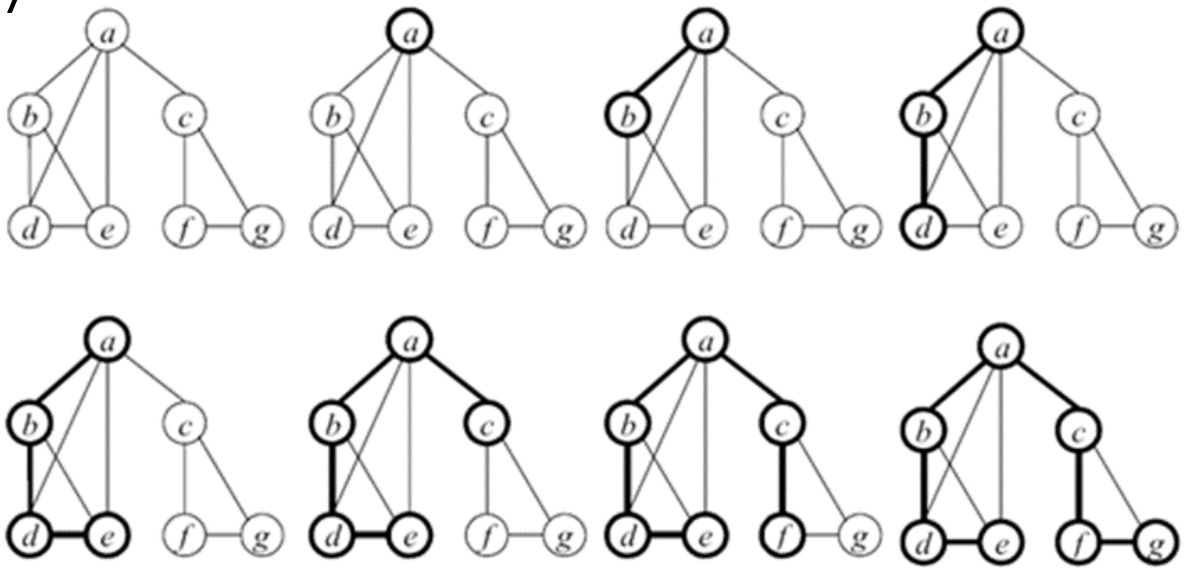
Основная идея поиска в глубину – когда возможные пути по ребрам, выходящим из вершин, разветвляются, нужно сначала полностью исследовать одну ветку и только потом переходить к другим веткам (если они останутся нерассмотренными).

Алгоритм поиска в глубину

- Шаг 1. Всем вершинам графа присваивается значение не посещенная. Выбирается первая вершина и помечается как посещенная.
- Шаг 2. Для последней помеченной как посещенная вершины выбирается смежная вершина, являющаяся первой помеченной как не посещенная, и ей присваивается значение посещенная. Если таких вершин нет, то берется предыдущая помеченная вершина.
- Шаг 3. Повторить шаг 2 до тех пор, пока все вершины не будут помечены как посещенные.

Демонстрация алгоритма поиска в глубину:

```
void Depth_First_Search(int n, int **Graph, bool *Visited,
                        int Node){
  Visited[Node] = true;
  cout << Node + 1 << endl;
  for (int i = 0 ; i < n ; i++)
    if (Graph[Node][i] && !Visited[i])
      Depth_First_Search(n,Graph,Visited,i);
}
```



Также часто используется нерекурсивный *алгоритм* поиска в глубину. В этом случае *рекурсия* заменяется на *стек*. Как только *вершина* просмотрена, она помещается в *стек*, а использованной она становится, когда больше нет новых вершин, смежных с ней.

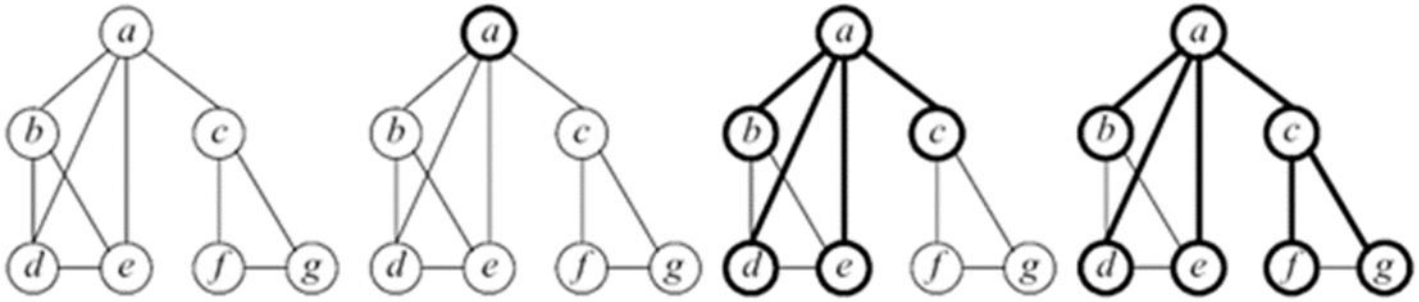
Временная сложность зависит от представления графа. Если применена *матрица смежности*, то временная сложность равна $O(n^2)$, а если нематричное *представление* – $O(n+m)$: рассматриваются все вершины и все *ребра*.

Поиск в ширину

- Идея **поиска в ширину** заключается в том, что сначала исследуются все вершины, смежные с начальной вершиной (*вершина* с которой начинается обход). Эти вершины находятся на расстоянии 1 от начальной. Затем исследуются все вершины на расстоянии 2 от начальной, затем все на расстоянии 3 и т.д. Обратим внимание, что при этом для каждой вершины сразу находятся *длина* кратчайшего маршрута от начальной вершины.

Алгоритм поиска в ширину

- Шаг 1. Всем вершинам графа присваивается *значение* не посещенная. Выбирается первая *вершина* и помечается как посещенная (и заносится в *очередь*).
- Шаг 2. Посещается первая *вершина* из очереди (если она не помечена как посещенная). Все ее соседние вершины заносятся в *очередь*. После этого она удаляется из очереди.
- Шаг 3. Повторяется шаг 2 до тех пор, пока *очередь* не пуста



Демонстрация алгоритма поиска в ширину

```
void Breadth_First_Search(int n, int **Graph,
                          bool *Visited, int Node){
  int *List = new int[n]; //очередь
  int Count, Head;        // указатели очереди
  int i;
  // начальная инициализация
  for (i = 0; i < n ; i++)
    List[i] = 0;
  Count = Head = 0;
  // помещение в очередь вершины Node
  List[Count++] = Node;
  Visited[Node] = true;
  while ( Head < Count ) {
    //взятие вершины из очереди
    Node = List[Head++];
    cout << Node + 1 << endl;
    // просмотр всех вершин, связанных с вершиной Node
    for (i = 0 ; i < n ; i++)
      // если вершина ранее не просмотрена
      if (Graph[Node][i] && !Visited[i]){
        // заносим ее в очередь
        List[Count++] = i;
        Visited[i] = true;
      }
  }
}
```

Нахождение кратчайших путей. Алгоритм Дейкстры, алгоритм Флойда.

Поиск кратчайшего пути ведется между двумя заданными вершинами в графе. Результатом является *путь*, то есть последовательность вершин и ребер, *инцидентных* двум соседним вершинам, и его *длина*.

Рассмотрим три наиболее *эффективных алгоритма* нахождения *кратчайшего пути*:

- *алгоритм Дейкстры*;
- *алгоритм Флойда*;

переборные алгоритмы.

Указанные алгоритмы легко выполняются при малом количестве вершин в графе. При увеличении их количества задача поиска *кратчайшего пути* усложняется.

Алгоритм Дейкстры

Каждой вершине приписывается *вес* – это *вес* пути от начальной вершины до данной. Также каждая *вершина* может быть выделена. Если *вершина* выделена, то *путь* от нее до начальной вершины кратчайший, если нет – то временный. *Обходя граф*, *алгоритм* считает для каждой вершины *маршрут*, и, если он оказывается кратчайшим, выделяет вершину. Весом данной вершины становится *вес* пути. Для всех соседей данной вершины *алгоритм* также рассчитывает *вес*, при этом ни при каких условиях не выделяя их. *Алгоритм* заканчивает свою работу, дойдя до конечной вершины, и весом *кратчайшего пути* становится *вес* конечной вершины.

Алгоритм

Алгоритм Дейкстры

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.

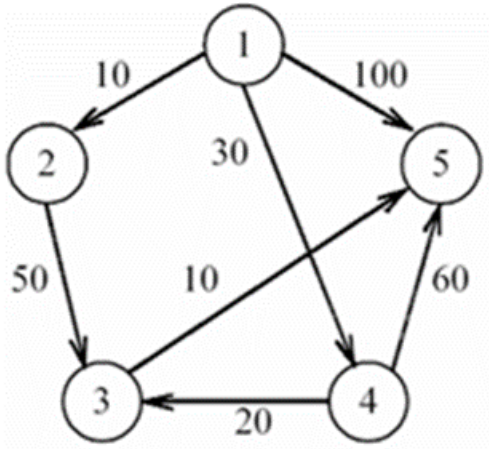
Шаг 6. Если текущей вершиной оказывается конечная, то *путь* найден, и его *вес* есть *вес* конечной вершины.

Ш**аг 7**. Переход на шаг 4.

В программной реализации *алгоритма Дейкстры* построим множество S вершин, для которых кратчайшие пути от начальной вершины уже известны. На каждом шаге к множеству S добавляется та из оставшихся вершин, *расстояние* до которой от начальной вершины меньше, чем для других оставшихся вершин. При этом будем использовать *массив* D, в который записываются длины *кратчайших путей* для каждой вершины. Когда множество S будет содержать все *вершины графа*, тогда *массив* D будет содержать длины *кратчайших путей* от начальной вершины к каждой вершине.

Помимо указанных массивов будем использовать *матрицу длин* C, где элемент C[i,j] *–длина ребра* (i,j), если *ребра* нет, то ее *длина* полагается равной бесконечности, то есть больше любой фактической длины ребер. Фактически *матрица* C представляет собой *матрицу смежности*, в которой все нулевые элементы заменены на бесконечность.

Для определения самого *кратчайшего пути* введем *массив* P вершин, где P[v] будет содержать вершину, непосредственно предшествующую вершине v в кратчайшем пути.



Итерация	*S*	*w*	D[2]	D[3]	D[4]	D[5]
начало	{1}	–	10	∞	30	100
1	{1, 2}	2	10	60	30	100
2	{1, 2, 4}	4	10	50	30	90
3	{1, 2, 4, 3}	3	10	50	30	60
4	{1, 2, 4, 3, 5}	5	10	50	30	60

Массив **P**:

	1	4	1	3

Кратчайший путь из 1 в 5: {1, 4, 3, 5}

Алгоритм Флойда

Метод Флойда непосредственно основывается на том факте, что в графе с положительными весами ребер всякий неэлементарный (содержащий более 1 *ребра*) *кратчайший путь* состоит из других *кратчайших путей*.

Этот *алгоритм* более общий по сравнению с алгоритмом Дейкстры, так как он находит кратчайшие пути между любыми двумя *вершинами графа*.

В алгоритме Флойда используется *матрица* A размером nxn, в которой вычисляются длины *кратчайших путей*. Элемент A[i,j] равен расстоянию от вершины i к вершине j, которое имеет конечное *значение*, если существует *ребро* (i,j), и равен бесконечности в противном случае.

Основная идея алгоритма. Пусть есть три вершины i, j, k и заданы расстояния между ними. Если выполняется *неравенство* A[i,k]+A[k,j]<A[i,j], то целесообразно заменить *путь* i->j путем i->k->j. Такая замена выполняется *систематически* в процессе выполнения данного алгоритма.

Шаг 0. Определяем начальную матрицу расстояния A_0 и матрицу последовательности вершин S_0. Каждый диагональный элемент обеих матриц равен 0, таким образом, показывая, что эти элементы в вычислениях не участвуют. Полагаем k = 1.

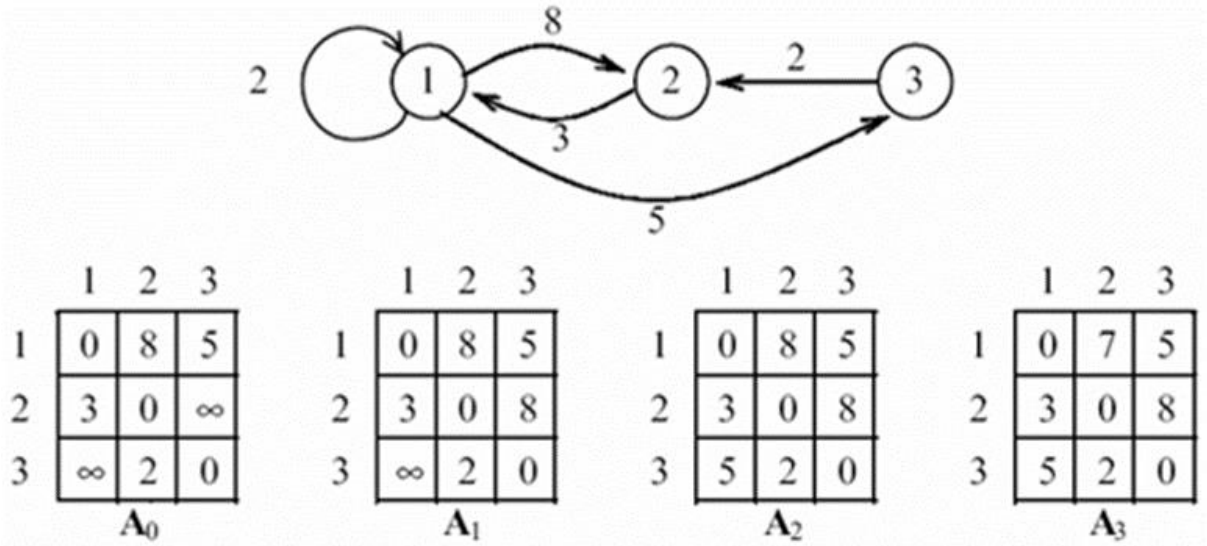
Основной шаг k. Задаем строку k и столбец k как ведущую строку и ведущий столбец. Рассматриваем возможность применения замены описанной выше, ко всем элементам A[i,j] матрицы A_{k-1}. Если выполняется *неравенство*

$$A[i,k] + A[k,j] < A[i,j], (i \neq k, j \neq k, i \neq j)$$

, тогда выполняем следующие действия:

- создаем матрицу A_k путем замены в матрице A_{k-1} элемента A[i,j] на сумму A[i,k]+A[k,j] ;
- создаем матрицу S_k путем замены в матрице S_{k-1} элемента S[i,j] на k. Полагаем k = k + 1 и повторяем шаг k.

Таким образом, *алгоритм Флойда* делает n итераций, после i -й итерации *матрица* A будет содержать длины *кратчайших путей* между любыми двумя парами вершин при условии, что эти пути проходят через вершины от первой до i -й. На каждой итерации перебираются все пары вершин и *путь* между ними сокращается при помощи i -й вершины.



Нахождение центра графа. Задача коммивояжера. Эйлеровы пути и циклы.

Центр графа

Центр (или центр Жордана) графа — это множество всех вершин с минимальным эксцентриситетом. То есть множество всех вершин A, для которой максимальное расстояние d(A,B) до других вершин B минимально. Эквивалентно, это множество вершин с эксцентриситетом, равным радиусу графа.

Нахождение центра графа полезно для задач размещения предприятий, целью которых является минимизация наиболее дальних расстояний до предприятия. Например, размещение госпиталя в центре объекта уменьшает максимальное расстояние, которое приходится преодолевать машинам медицинской помощи.

Концепция центра графа связана с измерением центральности по близости в анализе социальных сетей, которая равна обратной величине к среднему расстояний d(A,B)

Задача коммивояжёра

Задача коммивояжёра (или TSP от англ. travelling salesman problem) — одна из самых известных задач комбинаторной оптимизации, заключающаяся в поиске самого выгодного маршрута, проходящего через указанные города хотя бы по одному разу с последующим возвратом в исходный город. В условиях задачи указываются критерий выгодности маршрута (кратчайший, самый дешёвый, совокупный критерий и тому подобное) и соответствующие матрицы расстояний, стоимости и тому подобного. Как правило, указывается, что маршрут должен проходить через каждый город только один раз — в таком случае выбор осуществляется среди гамильтоновых циклов.

Оптимизационная постановка задачи относится к классу NP-трудных задач, впрочем, как и большинство её частных случаев. Версия «decision problem» (то есть такая, в которой ставится вопрос, существует ли маршрут не длиннее, чем заданное значение k) относится к классу NP-полных задач. Задача коммивояжёра относится к числу трансвычислительных: уже при относительно небольшом числе городов (66 и более) она не может быть решена методом перебора вариантов никакими теоретически мыслимыми компьютерами за время, меньшее нескольких миллиардов лет.

Классификация вариантов задач

- Симметричная проблема коммивояжёра (TSP = traveling salesman problem) в которой расстояния заданы между любыми двумя городами и матрица расстояний симметрична: Dji = Dij.
- Асимметричная проблема коммивояжёра (ATSP) допускает несимметричность матрицы Dji ≠ Dij. В ещё более общем случае, пути между некоторыми городами могут отсутствать (== иметь бесконечную длину).
- Задача с частичным упорядочиванием (SOP = sequential ordering problem) требующая, чтобы определённый город i был посещён до города j (таких условий может быть несколько).
- Поиск цикла Гамильтона (HCP = hamiltonian cycle problem) - обнаружение в произвольном графе замкнутых путей, проходящих через каждую вершину в точности один раз.

В TSP (симметричной задаче коммивояжёра) путь замкнут и стартовать можно с любого города (и в любую сторону). Поэтому для N городов существует (N-1)!/2 различных путей. Факториал растёт очень быстро: N! ~ NN и пространство в котором ищется оптимальное решение оказывается огромным. Именно поэтому задача коммивояжёра интересна для тестирования различных алгоритмов.

Методы решения

1. ***Точные методы*** не только находят некоторое решение, но и при *окончании* своей работы доказывают, что это решение - наилучшее. Отметим следующие из них:

- *Полный перебор* перестановок N-1 чисел (стартовый город фиксирован). Практически бесполезен при N > 15.
- *Направленный поиск с возвратами* - перебор вариантов "вокруг" некоторого решения с отсечением путей, имеющих длину большую, чем лучший к текущему моменту путь.
- *Метод ветвей и границ* - наиболее эффективный из известных метод отсечения "неперспективных" узлов, за счёт анализа матрицы расстояний. Линейное программирование применяется для минимизации (с ограничениями) линейной формы **d**·**x**, где **x** - искомый бинарный вектор размерности N(N-1)/2, компоненты которого x_i равны 1 или 0, в зависимости от того, входит i-е ребро в путь или нет. Вектор **d** (той же размерности) равен длинам рёбер.

2. **Эвристические методы**, обычно, существенно быстрее точных, однако они не гарантируют оптимальности найденного решения. Результат их комбинации может далее использоваться как первое приближение для последующего улучшения, например, при помощи поиска с возвратом:

- *Жадный алгоритм* при выборе очередного города берёт ближайший не посещённый до этого город.
- *Метод шнурка* - геометрическая вариация жадного алгоритма, в которой города охватываются замкнутым контуром. Он постепенно растягивается, стараясь пройти через все города, минимальным образов увеличив свою длину.
- *Скользящий перебор* переставляет местами города из небольшой части пути. Затем такое "окно перебора" скользит вдоль всего пути. Метод имеет различные вариации и оказывается эффективным способом улучшения решения, найденного предыдущими двумя эвристическими методами.

Методы решения

3. ***Вероятностные методы*** фактически ни когда не останавливаются, совершая случайные изменения пути, в ожидании получения более короткого. Из этого класса методов отметим:

- *Метод отжига* в котором происходят перестановки городов с постепенно "затухающей" интенсивностью. При этом постоянно сохраняется наилучшее найденное решение.
- *Генетический алгоритм* - более "продвинутый" вариант, при котором создаётся большое количество различных путей. Они постоянно "мутируют" и "скрещиваются" друг с другом, обмениваясь отдельным участками.

Эйлеров путь – путь в графе, проходящий через все точки графа. **Эйлеров цикл** - это Эйлеров путь, являющийся циклом.

Чтобы проверить, существует ли эйлеров путь, нужно воспользоваться следующей теоремой.

*Пусть дан неориентированный **связный** цикл. Эйлеров цикл существует тогда и только тогда, когда степени всех вершин чётны. Эйлеров путь существует тогда и только тогда, когда количество вершин с нечётными степенями равно двум (или нулю, в случае существования эйлерова цикла).*

```
struct Node
{
   int inf;
   Node *next;
};

//=================Stack=================

void push(Node *&st,int dat)
{  // Загрузка числа в стек

   Node *el = new Node;
   el->inf = dat;
   el->next = st;
   st=el;
}
int pop(Node *&st)
{       // Извлечение из стека

   int value = st->inf;
   Node *temp = st;
   st = st->next;
   delete temp;

   return value;
}
int peek(Node *st)
{     // Получение числа без его извлечения
   return st->inf;
}
```

```
Node **graph;    // Массив списков смежности
const int vertex = 1; // Первая вершина

void add(Node*& list,int data)
{  //Добавление смежной вершины

   if(!list){list=new Node;list->inf=data;list->next=0;return;}

   Node *temp=list;
   while(temp->next)temp=temp->next;
   Node *elem=new Node;
   elem->inf=data;
   elem->next=NULL;
   temp->next=elem;
}

void del(Node* &l,int key)
{ // Удаление вершины key из списка

   if(l->inf==key){Node *tmp=l; l=l->next; delete tmp;}
   else
        {
      Node *tmp=l;
      while(tmp)
         {
         if(tmp->next) // есть следующая вершина
            if(tmp->next->inf==key)
                  {  // и она искомая
               Node *tmp2=tmp->next;
                  tmp->next=tmp->next->next;
                  delete tmp2;
                }
            tmp=tmp->next;
      }
   }
}
```

```
int eiler(Node **gr,int num)
{ // Определение эйлеровости графа
   int count;
   for(int i=0;i<num;i++)
   {  //проходим все вершины

      count=0;
      Node *tmp=gr[i];

       while(tmp)
          {       // считаем степень
             count++;
          tmp=tmp->next;
          }
      if(count%2==1)return 0; // степень нечетная
   }
   return 1;   // все степени четные
}
void eiler_path(Node **gr)
{ //Построение цикла

   Node *S = NULL;// Стек для  пройденных вершин
   int v=vertex;// 1я вершина (произвольная)
   int u;

   push(S,v); //сохраняем ее в стек
   while(S)
      {  //пока стек не пуст
      v = peek(S); // текущая вершина
      if(!gr[v]){ // если нет инцидентных ребер
      v=pop(S); cout<<v+1<<"   "; //выводим вершину  (у нас отсчет от
1, поэтому +1)
      }
     else
      {
      u=gr[v]->inf; push(S,u);  //проходим в следующую вершину
      del(gr[v],u); del(gr[u],v); //удаляем пройденное ребро
      }
   }
}
```

```
int main()
{
system("CLS");
     cout<<"Количество вершин:  "; int n; cin>>n; // Количество
вершин
   int zn;// Текущее значение
   graph=new Node*[n];
   for(int i=0;i<n;i++)graph[i]=NULL;
   for(i=0;i<n;i++)    // заполняем массив списков
      for(int j=0;j<n;j++)
           {
         cin>>zn;
         if (zn) add(graph[i],j);
       }
          cout<<"\n\nРЕЗУЛЬТАТ  ";
   if(eiler(graph,n))eiler_path(graph);
   else cout<<"Граф не является эйлеровым.";
   cout<<endl;
   cin.get();
   cin.get();
   return(0);
}
```

Раздел 4. Параллельная обработка данных на CPU. Распараллеливание алгоритмов. Библиотека ParallelExtention в MS VisualStudio. Класс Thread, Task., BackgroundWorker. Базовые принципы разработки распараллеливания алгоритмов на центральном процессоре.

Параллельная и конвейерная обработка

Параллельная обработка.

- Параллельная обработка данных, воплощая идею одновременного выполнения нескольких действий, имеет две разновидности: конвейерность и параллельность. Оба вида параллельной обработки интуитивно понятны, поэтому сделаем лишь небольшие пояснения.

Конвейерная обработка.

- Идея конвейерной обработки заключается в выделении отдельных этапов выполнения общей операции, причем каждый этап, выполнив свою работу, передавал бы результат следующему, одновременно принимая новую порцию входных данных. Получаем очевидный выигрыш в скорости обработки за счет совмещения прежде разнесенных во времени операций.

Параллелизм на уровне алгоритмов

- Данный вид параллелизма предполагает замену последовательных алгоритмов некоторых вычислений на параллельные. Это касается алгоритмов поиска, сортировки и т.п. Организация процесса распараллеливания осуществляется за счет использования различных средств параллельного программирования, таких как, специальные библиотеки, переменные окружения, директивы компилятора и т.п. Примером может служить технология OpenMP.

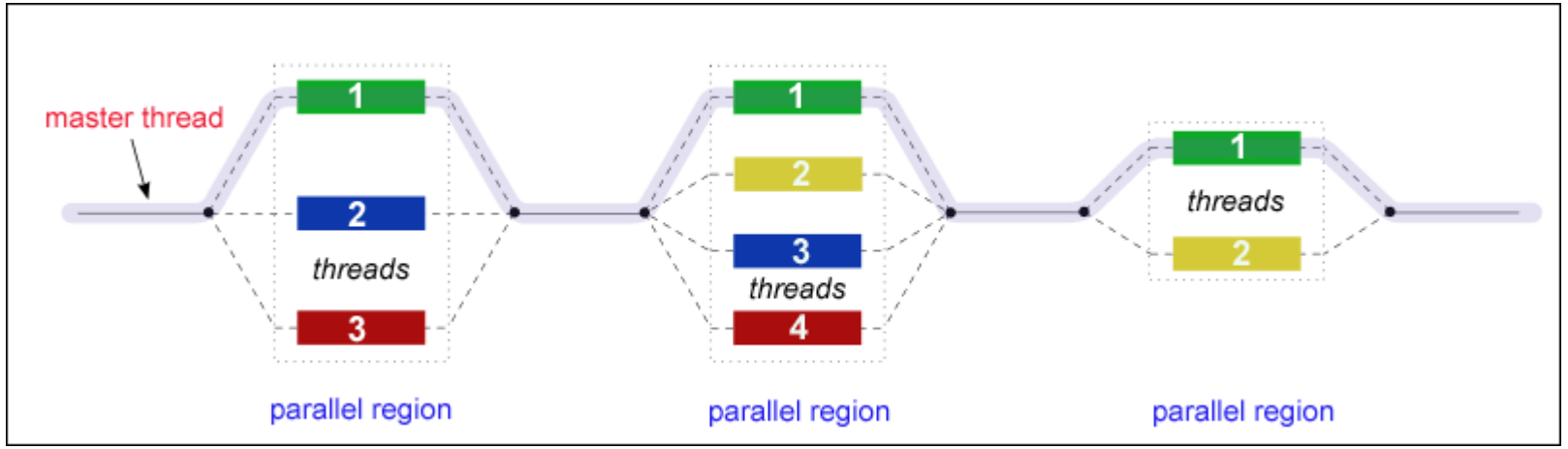
OpenMP для C/C++

OpenMP — открытый стандарт для распараллеливания программ на языках Си, Си++ и Фортран. Даёт описание совокупности директив компилятора, библиотечных процедур и переменных окружения, которые предназначены для программирования многопоточных приложений на многопроцессорных системах с общей памятью.

Спецификация OpenMP для C/C++, выпущенная на год позже фортранной, содержит в основном аналогичную функциональность.

Пример распараллеливания for-цикла в C

```
#pragma omp parallel for private(i)
#pragma omp shared(x, y, n) reduction(+: a, b)
for (i=o; i<n; i++)
{
   a = a + x[i];
   b = b + y[i];
}
```



Parallel Extensions

- **Parallel Extensions** to the .NET Framework (другие названия - Parallel FX Library, PFX) - это библиотека, разработанная фирмой Microsoft, для использования в программах на базе управляемого (managed) кода. Она позволяет распараллеливать задачи, в которых могут использоваться специальные - координирующие (coordinating) - структуры данных.

Parallel Extensions обеспечивает несколько новых способов организации параллелизма:

- ***Параллелизм при декларативной обработке данных.*** Реализуется при помощи параллельного интегрированного языка запросов (PLINQ) - параллельной реализации LINQ
- ***Параллелизм при императивной обработке данных.*** Реализуется при помощи библиотечных реализаций параллельных вариантов основных итеративных операций над данными, таких как циклы for и foreach. Их выполнение автоматически распределяется на все доступные ядра/процессоры вычислительной системы.
- ***Параллелизм на уровне задач.*** Библиотека Parallel Extensions обеспечивает высокоуровневую работу с пулом рабочих потоков, позволяя явно структурировать параллельно исполняющийся код с помощью легковесных задач. Планировщик библиотеки Parallel Extensions выполняет диспетчеризацию и управление исполнением этих задач, а также единообразный механизм обработки исключительных ситуаций.

Как начать программировать с использованием Parallel Extensions

Для того чтобы начать разрабатывать программы с применением Parallel Extensions необходимо выполнить следующие шаги:

1. Установить Parallel Extensions to the .Net Framework
2. Запустить Visual Studio
3. Создать новый проект
4. Добавить в проект ссылку на библиотеку System.Threading.dll

Microsoft Parallel Extensions for .Net состоит из двух компонент:

- TPL (Task Parallel Library);
- PLINQ (Parallel Language-Integrated Query).

А также содержит набор координирующих структур данных, используемых для синхронизации и координации выполнения параллельных задач.

Основная, базовая концепция Parallel Extensions - это задача (Task) - небольшой участок кода, представленный лямбда-функцией, который может выполняться независимо от остальных задач. И PLINQ, и TPL API предоставляют методы для создания задач - PLINQ разбивает запрос на небольшие задачи, а методы TPL API - Parallel.For, Parallel.ForEach и Parallel.Invoke - разбивают цикл или последовательность блоков кода на задачи.

Класс Thread

- **Класс Thread** является самым элементарным из всех типов пространства имен System.Threading. Этот класс представляет объектно-ориентированную оболочку вокруг заданного пути выполнения внутри определенного AppDomain. Этот тип также определяет набор методов (как статических, так и уровня экземпляра), которые позволяют создавать новые потоки внутри текущего AppDomain, а также приостанавливать, останавливать и уничтожать определенный поток. Список основных статических членов приведен по ссылке: https://professorweb.ru/my/csharp/thread_and_files/1/1_4.php
- **Класс Thread** также поддерживает несколько методов уровня экземпляра, часть из которых описана в таблице ниже. Отмена или приостановка активного потока обычно считается плохой идеей. Когда вы делаете это, есть шанс (хотя и небольшой), что поток может допустить "утечку" своей рабочей нагрузки, когда его беспокоят или прерывают.

Task класс

- В основу TPL положен **класс Task.** Элементарная единица исполнения инкапсулируется в TPL средствами класса Task, а не Thread. Класс Task отличается от класса Thread тем, что он является абстракцией, представляющей асинхронную операцию. А в классе Thread инкапсулируется поток исполнения. Разумеется, на системном уровне поток по-прежнему остается элементарной единицей исполнения, которую можно планировать средствами операционной системы. Но соответствие экземпляра объекта класса Task и потока исполнения не обязательно оказывается взаимно-однозначным.
- **TaskКласс** представляет отдельную операцию, которая не возвращает значение и обычно выполняется асинхронно. Taskобъекты — это один из центральных компонентов асинхронной модели на основе задач , впервые представленный в платформа .NET Framework 4. Так как работа, выполняемая Task объектом, обычно выполняется асинхронно в потоке пула потоков, а не синхронно в основном потоке приложения, можно использовать Status свойство, а также IsCanceled IsCompleted свойства, и IsFaulted , чтобы определить состояние задачи. Чаще всего лямбда-выражение используется для указания работы, которую должна выполнить задача.

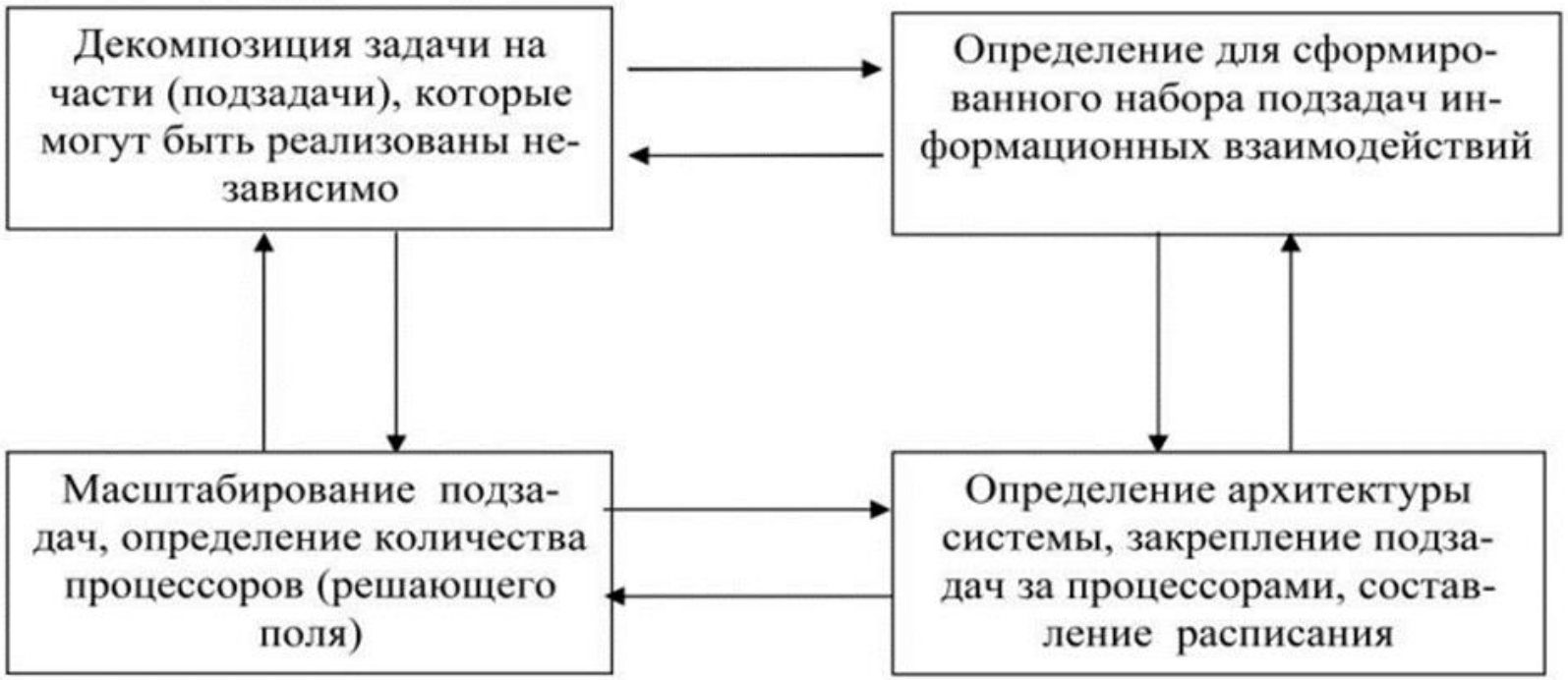
Background Worker

- ***Компонент BackgroundWorker*** позволяет выполнять длительные операции асинхронно (в фоновом режиме), т. е. в потоке, отличающемся от основного потока пользовательского интерфейса. Для использование компонента BackgroundWorker необходимо только указать, какой рабочий метод обработки длительных операций будет выполняться в фоновом режиме, а затем вызвать метод RunWorkerAsync. Вызывающий поток продолжает работать нормально, в то время как рабочий метод работает асинхронно. Когда метод закончит работу, компонент BackgroundWorker предупредит вызывающий поток событием RunWorkerCompleted, которое может содержать результаты операции.

- BackgroundWorker Класс - выполняет операцию в отдельном потоке. На C# это будет выглядеть:

- **public class BackgroundWorker : System.ComponentModel.Component**

Общая схема разработки параллельных алгоритмов



Эффективность

Эффективность параллельной программы зависит от соотношения временных затрат на проведение вычислений на фрагментах исходных данных и пересылку данных. Вычислительная задача разбивается на несколько самостоятельных подзадач в том случае если в ней отсутствует параллелизм по данным. Каждый процессор занимается решением отдельной подзадачи. В данном случае имеет место параллелизм задач. Количество задач влияет на количество процессоров. При обеспечении равномерной загрузки процессоров и минимизации обмена данными между ними можно ожидать значительного ускорения. Эффективность кода предполагает анализ затрачиваемого времени разными частями программы с целью выявления наиболее ресурсоемких частей

Раздел 5. Введение в анализ социальных сетей. Меры центральности. Алгоритмы вычисления показателей центральности. Алгоритмы визуализации социальных сетей. Алгоритмы обработки данных из социальных сетей, на примере социальной сети Twitter

Контент социальных сетей

В части контента имеет место следующее. Данные социальных сетей в относят к неструктурированным и гетерогенным. Социальные сети можно условно классифицировать по типу данных следующим образом:

- содержащие преимущественно текстовые данные – Twitter, Telegram, LiveJournal и т.д.;
- содержащие преимущественно изображения – Instagram, Snapchat и т.д.;
- содержащие преимущественно видео – YouTube, Vimeo и т.д.;
- содержащие данные смешанного типа видео – FaceBook, ВКонтакте и т.д.

Для анализа различных типов данных существуют различные подходы и методы. Зачастую неструктурированные «сырые» данные подлежат предварительной обработке.

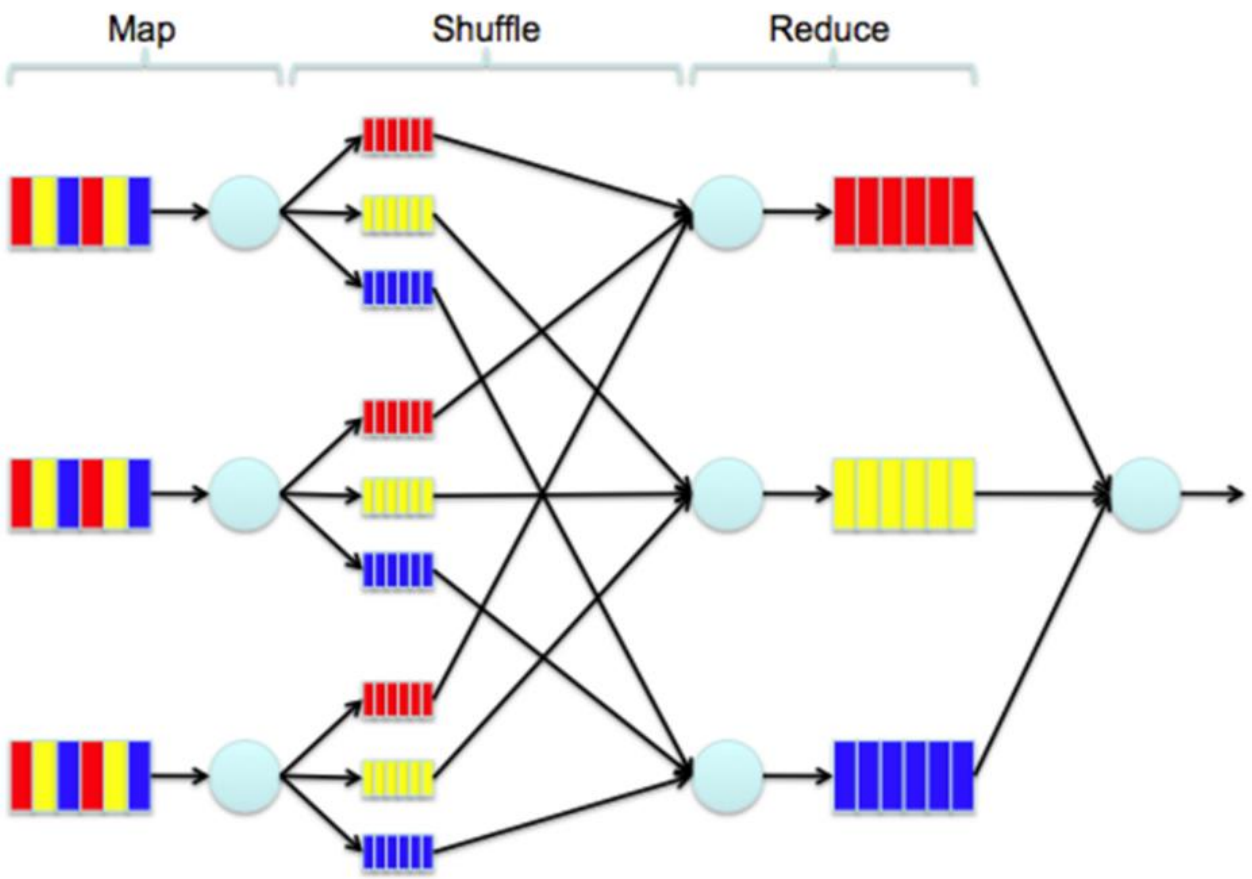
Введение в анализ

- Первый этап работы с данными социальных сетей обусловлен, прежде всего, наличием необходимого инструментария, а также возможностью предоставления сбора (открытого API) самой социальной сетью. Сбор может осуществляться как уже размещенных в социальных сетях данных, так и в режиме on-line. При этом еще на этапе сбора можно ставить определенные фильтры, к примеру, собрать лишь те сообщения, в которых присутствует то или иное слово или хештег.
- Для дальнейшего анализа бывают важны лишь те данные, которые представляют интерес, поэтому необходимо отделить служебную информацию от нужной. В платформах, как правило, используется технология, основанная на модели распределенных вычислений MapReduce.

Технология MapReduce

- С помощью этой технологии производится структуризация путем компоновки и исключения служебных и не представляющих практический интерес данных. В подходе, основанном на данной модели, производятся вычисления некоторых наборов распределенных задач с использованием большого количества компьютеров (называемых «нодами»), образующих кластер.
- Работа MapReduce состоит из двух шагов: Map и Reduce. На Map-шаге происходит предварительная обработка входных данных. Для этого один из компьютеров (называемый главным узлом — master node) получает входные данные задачи, разделяет их на части и передает другим компьютерам (рабочим узлам — worker node) для 10 предварительной обработки. Название данный шаг получил от одноименной функции высшего порядка. На Reduce-шаге происходит свёртка предварительно обработанных данных. Главный узел получает ответы от рабочих узлов и на их основе формирует результат – решение задачи, которая изначально формулировалась. Пример работы технологии MapReduce показан на рисунке.

Технология MapReduce



Преимущество MapReduce заключается в том, что она позволяет распределено производить операции предварительной обработки и свёртки. Операции предварительной обработки работают независимо друг от друга и могут производиться параллельно (хотя на практике это ограничено источником входных данных и/или количеством используемых процессоров).

Инструменты сбора данных

В настоящее время существует множество программных инструментов для сбора данных. Многие ведущие компании мира, такие как IBM, Oracle, Microsoft и другие имеют свои решения в этой области.
Из свободных и сравнительно недорогих чаще всего используются следующие продукты:

- **1010data;**
- **Apache Hadoop;**
- **Apache Ambari;**
- **Jaspersoft;**
- **LixisNexis Risj Sokutions HPCC Systems;**
- **Revolution Analytics (на базе языка R для мат. статистики);**
- **Hortonworks;**
- **Biginsights;**
- **Cloudera;**
- **Teradata и др.**

Многие из них имеют модули для работы с данными социальных сетей. К примеру, используя решения от Apache Ambari, IBM Biginsights, Hortonworks, либо Cloudera можно довольно просто организовать потоковый on-line сбор данных социальной сети Twitter в режиме реального времени по любой выбранной геолокации.

Показатель центральности

Показатель центральности или близости к центру, который определяет наиболее важные вершины графа, т.е. наиболее влиятельных лиц. Чем центральнее узел, тем ближе он ко все остальным узлам. С понятием центральности связаны следующие меры:

- степень посредничества – мера центральности вершины в графе, число раз, когда узел служит мостом в кратчайшем пути между двумя другими узлами, показывает меру количественного выражения взаимодействия человека с другими людьми в социальной сети.
- степень влиятельности – мера влияния узла в сети в зависимости от его связей с другими узлами – связи с узлами с высоким показателем влиятельности вкладывают больше в показатель рассматриваемого узла, чем такая же связь с узлом с низким показателем.

Алгоритмы вычисления показателей центральности

Centrality – характеристика узла, а можно посчитать ещё Centralization – это характеристика всей сети, которая показывает насколько равномерно распределение Degree centrality. Меры для измерения централизации сети:

- Формула Фримена: — максимальное значение центральности в сети.

$$C_d = \frac{\sum_{i=1}^{g} \max(C_d) - C_d(i)}{(N-1)(N-2)}$$

Где $\max C_d$ - максимальное значение центральности в сети.

- Коэффициент Джини
- SD = standard deviation

В графе социальной сети вершинами являются участники, а ребра означают наличие отношений между ними. Отношения могут быть как направленными, так и ненаправленными. Как правило, выделяют два типа отношений: «дружба» (люди знакомы друг с другом) и «интересы» (есть общие интересы, люди входят в одну группу по интересам). Эти отношения используются, например, в FOAF (Friend of a fried) – онтологии описания людей, их активности и отношений к другим людям и объектам

Графовые модели

- ***Стохастические блоковые модели*** задаются матрицей A размера N × N, где N – число групп (блоков) участников. Элемент $a_{ij} \in [0,1]$ показывает плотность связей между участниками сети, принадлежащими к группе v_i, и участниками, принадлежащими к группе v_j При этом граф не содержит дополнительных ребер и вершин, соответствующих связям участников внутри одной группы.

- ***Вероятностные графовые модели*** задаются матрицей A размера N × N, где N – число участников сети. Элемент $a_{ij} \in [0,1]$ показывает вероятность взаимодействия участника vi и участника vj в течение определенного периода времени.

- ***Обычные графовые модели*** задаются матрицей связности A размера N × N. Для анализа графовых моделей социальных сетей иногда удобно использовать *коэффициент плотности*, определенный как отношение числа ребер в анализируемом графе к числу ребер в полном графе с тем же числом вершин (полный граф – это граф, в котором все вершины соединены между собой). Кроме этого, сеть могут характеризовать такие величины, как число путей заданной длины (путь – последовательность вершин, связанных между собой), минимальное число ребер, удаление которых разбивает граф на несколько частей.

Центральность по степени

- Центральность по степени (Degree centrality) определяется как количество связей, инцидентных вершине:

$$C_D(v) = \deg(v).$$

- Выделяют ***входящие и исходя***щие связи. Входящие связи характеризуют популярность человека, выходящие – его общительность. Полученную величину можно нормировать, разделив на общее число участников в сети.

Центральность по близости

- ***Центральность по близости*** (Closeness centrality) является показателем, насколько быстро распространяется информация в сети от одного участника к остальным. В качестве меры расстояния между двумя участниками используется кратчайший путь по графу (геодезическое расстояние). Так, непосредственные друзья участника находятся на расстоянии 1, друзья друзей – на расстоянии 2, друзья друзей друзей – на расстоянии 3 и т. д. Далее берется сумма всех расстояний и нормируется. Полученная величина называется удаленностью вершины v от других вершин. Близость определяется как величина, обратная удаленности:

$$C_C(v) = \frac{N-1}{\sum_{t \in V \setminus v} d_G(v,t)},$$

где σ_{st} – общее количество кратчайших путей из вершины s к вершине t;

$\sigma_{st}(v)$ – количество кратчайших путей из вершины s к вершине t, проходящих через вершину v .

PageRank

- PageRank – это метод вычисления веса страницы путем подсчета важности ссылок на нее, т. е. вершина, ссылающаяся на другую вершину с большим весом, сама получает большой вес:

$$C_{PageRank}(i) = x_i = \alpha \sum_{j=1}^{N} a_{ji} \frac{x_j}{L(j)} + \frac{1-\alpha}{N},$$

Где $L(j) = \sum_j a_{ji}$ – количество вершин, соседних с вершиной j (или количество выходящих связей в ориентированном графе).

Центральность Каца

Центральность Каца - вычисляет относительное влияние узла в сети путём измерения числа ближайших соседей (узлы первой степени), а также всех других узлов в сети, которые соединяются через этих ближайших соседей. Любому пути или связи между парой узлов назначается вес, определённый значением α и расстоянием между узлами как α^d. При этом вес соединений с удалёнными соседями уменьшаются на множитель α.

Обобщением центральности по степени является центральность Каца (Katz centrality). Отличие в том, что центральность по степени учитывает количество непосредственных соседей вершины, а центральность Каца учитывает количество всех вершин, которые могут быть соединены путем:

$$C_{Katz}(i) = \sum_{k=1}^{\infty} \sum_{j=1}^{N} \alpha^k \left(a^k\right)_{ji},$$

Алгоритмы визуализации социальных сетей

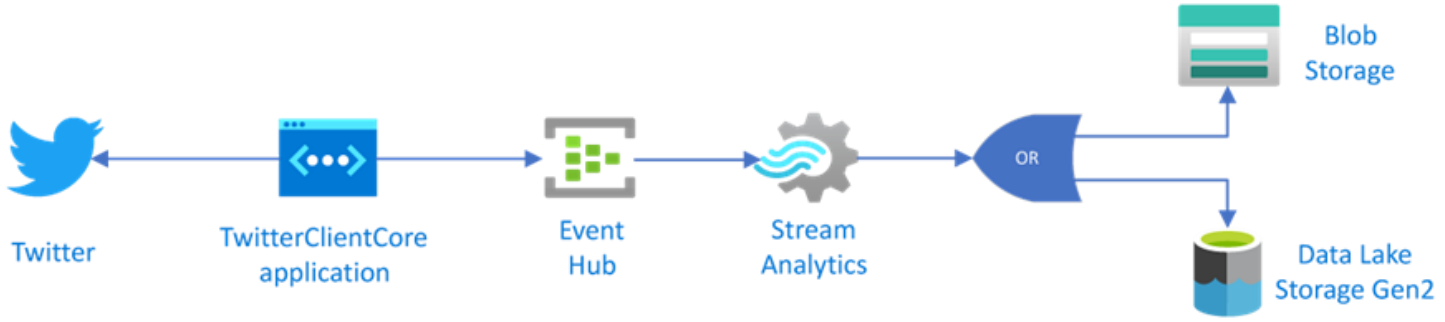
- **Силовые алгоритмы визуализации графов** — класс алгоритмов визуализации графов в эстетически приятном виде. Их цель — расположить узлы графа в двумерном или трёхмерном пространстве так, что все рёбра имели бы более-менее одинаковую длину, и свести к минимуму число пересечений рёбер путём назначения сил для множества рёбер и узлов основываясь на их относительных положениях, а затем путём использования этих сил либо для моделирования движения рёбер и узлов, либо для минимизации их энергии.

Следующие свойства являются наиболее важными преимуществами силовых алгоритмов:

- Результаты хорошего качества
- Гибкость
- Интуитивность
- Простота
- Интерактивность
- Строгая теоретическая поддержка

Алгоритмы обработки данных из социальных сетей, на примере социальной сети Twitter.

- Хороший пример средства прогнозной аналитики — анализ тенденций Twitter в режиме реального времени. Модель подписки с использованием хэштегов позволяет ожидать передачи определенных ключевых слов и выполнять анализ тональности веб-канала.



Сценарий:

У организации есть новостной веб-сайт. Она хочет получить конкурентное преимущество, мгновенно предлагая читателям содержимое, которое будет им интересно. Организация выполняет анализ социальных сетей по темам, которые интересуют их читателей, анализируя тональность данных Twitter в режиме реального времени.

Чтобы определять наиболее популярные темы в Twitter в режиме реального времени, организации необходимо анализировать количество твитов и тональность по ключевым темам.

Раздел 6. Хранение данных на жестком диске: форматы и нотации. Нотация JSON. Язык XML. Сериализация и десериализация в c#. Парсинг данных JSON и XML.

Хранение данных на жестком диске

- Данные на жестком диске записываются в виде последовательности двоичных (бинарных) битов (бит – цифра двоичной системы счисления, т.е. “0” или “1”). Каждый бит хранится как магнитный заряд (положительный или отрицательный) на магнитном слое пластины. При записи информации, данные посылаются к жесткому диску в виде последовательности битов. После получения диском данных, используются головки для магнитной записи. В этот момент головка генерирует поток магнитных импульсов, кодирующих данные на поверхности диска. Изменение полярности отвечает значению “1”, а отсутствие изменения – значению “0”. Информация не обязательно хранятся последовательно; например, данные одного файла могут быть записаны в разные места на разных пластинах.

Виды жестких дисков

Магнитные жесткие диски

Внутри диска находятся несколько алюминиевых пластин.

Операции чтения и записи происходят за счет вращения пластин и расположенной в нескольких нанометрах считывающей головки. Скорость пластин достигает 15 000 оборотов в минуту, отсюда и привычный шум, и высокая температура при работе дисков.

Такие диски стали популярными за счет большого объема дискового пространства (до 16 ТБ на одном HDD-диске), высокой степени надежности, устойчивости к операциям чтения и записи.

Твердотельные жесткие диски

Самые быстрые твердотельные жесткие диски могут записывать и считывать данные со скоростью свыше 500 МБ в секунду, при этом доступ к данным получается практически мгновенным. Такие накопители совершенно не боятся вибраций, что делает их очень устойчивыми к механическим повреждениям. Но у твердотельных жестких дисков на сегодняшний день есть один существенный недостаток, это их цена. Они значительно дороже обычных магнитных жестких дисков и выпускаются только небольшой емкости. Наибольшей популярностью пользуются SSD диски объемом 120 ГБ и 256 ГБ.

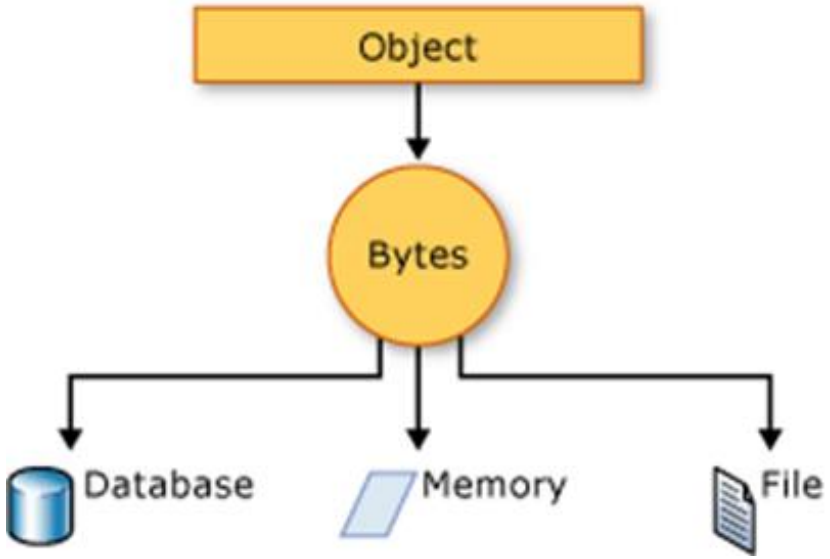
- ***Сериализация*** (в программировании) — процесс перевода структуры данных в последовательность байтов. Обратной к операции сериализации является операция десериализации (структуризации) — создание структуры данных из битовой последовательности.

Пример: если у вас есть класс

```
class Test
{
   int length;
   String name;

   public Test(int length, String name)
   {
      this.length = length;
      this.name = name;
   }
}
```

Объект этого класса в сериализованной форме может иметь вид:

{ "length": 25, "name": "Имя" }



Нотация JSON

JSON (англ. JavaScript Object Notation) — текстовый формат обмена данными, основанный на JavaScript. Как и многие другие текстовые форматы, JSON легко читается людьми.

Синтаксис JSON

JSON-текст представляет собой (в закодированном виде) одну из двух структур:

- Набор пар ключ: значение. В различных языках это реализовано как запись, структура, словарь, хеш-таблица, список с ключом или ассоциативный массив. Ключом может быть только строка (регистрозависимость не регулируется стандартом, это остаётся на усмотрение программного обеспечения, значением — любая форма.
- Упорядоченный набор значений. Во многих языках это реализовано как массив, вектор, список или последовательность.

В качестве значений в JSON могут быть использованы:

- **запись** — это неупорядоченное множество пар ключ:значение, заключённое в фигурные скобки «{ }». Ключ описывается строкой, между ним и значением стоит символ «:». Пары ключ-значение отделяются друг от друга запятыми.
- **массив** (одномерный) — это упорядоченное множество значений. Массив заключается в квадратные скобки «[ ]». Значения разделяются запятыми. Массив может быть пустым, то есть не содержать ни одного значения. Значения в пределах одного массива могут иметь разный тип.
- **число (целое или вещественное).**
- **литералы *true (логическое значение «истина»)*, *false*** (логическое значение «ложь») и null.
- ***строка*** — это упорядоченное множество из нуля или более символов юникода, заключённое в двойные кавычки. Символы могут быть указаны с использованием escape-последовательностей, начинающихся с обратной косой черты «\» (поддерживаются варианты \", \\, \/, \t, \n, \r, \f и \b), или записаны шестнадцатеричным кодом в кодировке Unicode в виде \uFFFF.

JSON-представление данных

Следующий пример показывает JSON-представление данных об объекте, описывающем человека. В данных присутствуют строковые поля имени и фамилии, информация об адресе и массив, содержащий список телефонов. Как видно из примера, значение может представлять собой вложенную структуру.

```
{
  "firstName": "Иван",
  "lastName": "Иванов",
  "address": {
    "streetAddress": "Московское ш., 101, кв.101",
    "city": "Ленинград",
    "postalCode": 101101
  },
  "phoneNumbers": [
    "812 123-1234",
    "916 123-4567"
  ]
}
```

Язык XML

XML (расширяемый язык разметки) — это язык программирования, который состоит из объявлений в виде информации и определяющих тегов. С его помощью удобно хранить и передавать любые данные.

Язык не зависит от операционной системы и среды обработки. XML служит для представления неких данных в виде структуры, которую вы можете сами разработать или подстроить под программу или сервис.

Именно поэтому данный язык называют расширяемым, и в этом его главное достоинство, за которое его так ценят.

Плюсы языка XML

- Легкость чтения, подача в простой форме;
- стандартный вид кодировки;
- возможность создания разных структур (списков, схем, деревьев);
- возможность восстановить данные, которые были сохранены в XML;
- возможность обмена данными между любыми платформами;
- популярность в разных сферах программирования.

Минусы языка XML

- Чрезмерный синтаксис, большое количество сущностей и тегов;
- один объект может быть представлен в разных описаниях;
- отсутствуют стандартные указания типа объекта.

Структура XML

```
<?xml version="1.0" encoding="UTF-8"?>
<marvel>
<!-- this is a good man -->
<hero id="positive_character">
<nickname>Captain America</nickname>
<realname>Steven Rogers</realname>
<abilities>Superhuman strength</abilities>
</hero>
<!-- this is a bad man -->
<hero id="negative_character">
<nickname>Red Skull</nickname>
<realname>Johann Schmidt</realname>
<abilities>Superhuman strength</abilities>
</hero>
</marvel>
```

Парсинг данных JSON и XML.

Обычно JSON используется для обмена данными с сервером. При получении с сервера данные всегда передаются в виде строки. Если обработать эти данные при помощи функции JSON.parse(), то они станут объектом JavaScript.

Пример - JSON.parse()

Представьте, что мы получили этот текст с веб сервера:

'{"name":"Андрей", "age":50, "city":"Пермь"}'

Используйте функцию JavaScript JSON.parse() для преобразования текста в объект JavaScript:

```
var obj = JSON.parse('{"name":"Андрей", "age":50, "city":"Пермь"}');
```

JSON с сервера

Вы можете запросить JSON с сервера, используя AJAX запрос

Пока ответ от сервера записан в формате JSON, вы можете проанализировать строку в объект JavaScript.

(XMLHttpRequest для получения данных с сервера)

```
var xmlhttp = new XMLHttpRequest();
xmlhttp.onreadystatechange = function() {
if (this.readyState == 4 && this.status == 200) {
var myObj = JSON.parse(this.responseText);
document.getElementById("demo").innerHTML =
myObj.name;
}
};
xmlhttp.open("GET", "json_demo.txt", true);
xmlhttp.send();
```

JSON массив

При использовании JSON.parse() на JSON, производном от массива, метод вернет массив JavaScript, а не объект JavaScript.

JSON, возвращенный с сервера, представляет собой массив:

```
var xmlhttp = new XMLHttpRequest();
xmlhttp.onreadystatechange = function() {
if (this.readyState == 4 && this.status == 200) {
var myArr = JSON.parse(this.responseText);
document.getElementById("demo").innerHTML =
myArr[0];
}
};
xmlhttp.open("GET", "json_demo_array.txt", true);
xmlhttp.send();
```

Исключения

Объекты дат не допускаются в JSON.

Если вам нужно включить дату, запишите ее в виде строки.

Вы можете преобразовать его обратно в объект даты позже:

Преобразование строки в дату:

```
var text = '{"name":"Андрей",
"birth":"1969-07-15",
"city":"Пермь"}';
var obj = JSON.parse(text);
obj.birth = new Date(obj.birth);

document.getElementById("demo").inn
erHTML = obj.name + ", " +
obj.birth;
```

Или вы можете использовать второй параметр JSON.parse(), называемая reviver.

Параметр reviver - это функция, которая проверяет каждое свойство перед возвращением значения.

```
var text = '{"name":"Андрей",
"birth":"1969-07-15", "city":"Пермь"}';
var obj = JSON.parse(text, function (key,
value) {
if (key == "birth") {
return new Date(value);
} else {
return value;
}
});

document.getElementById("demo").innerHTML =
obj.name + ", " + obj.birth;
```

Парсинг функции

Функции не разрешены в JSON.

Если вам нужно включить функцию, запишите ее в виде строки.

```
var text = '{"name":"Андрей",
"age":"function () {return 50;}",
"city":"Пермь"}';
var obj = JSON.parse(text);
obj.age = eval("(" +
obj.age + ")");

document.getElementById("demo").inn
erHTML = obj.name + ", " +
obj.age();
```

Парсинг XML

XML парсер преобразует XML документ в объект XML DOM, которым затем можно манипулировать при помощи JavaScript.

Объект XMLHttpRequest позволяет обмениваться данными в фоновом режиме.

Это настоящая сбывшаяся мечта разработчика, потому что вы можете:

- Обновлять содержимое веб-страницы не перезагружая веб-страницу
- Запрашивать данные с сервера, когда веб-страница уже загружена
- Получать данные с сервера, когда веб-страница уже загружена
- Посылать данные на сервер в фоновом режиме

Создание объекта XMLHttpRequest

Все современные браузеры (IE7+, Firefox, Chrome, Safari, Opera) уже имеют встроенный объект XMLHttpRequest.

Объект XMLHttpRequest создается следующим образом:

```
xmlhttp = new XMLHttpRequest();
```

Работа с объектом XMLHttpRequest

Типичный синтаксис JavaScript для работы с объектом XMLHttpRequest выглядит следующим образом:

```
var xhttp = new XMLHttpRequest();
xhttp.onreadystatechange = function() {
    if (this.readyState == 4 && this.status == 200) {
        // Typical action to be performed when the document is
ready:
        document.getElementById("demo").innerHTML =
xhttp.responseText;
    }
};
xhttp.open("GET", "filename", true);
xhttp.send();
```

Парсинг XML документа

Следующий фрагмент кода парсит XML документ в объект XML DOM:

```
if (window.XMLHttpRequest)
{
   // для IE7+, Firefox, Chrome, Opera, Safari
   xmlhttp = new XMLHttpRequest();
}
else
{
   // для IE6, IE5
   xmlhttp = new ActiveXObject("Microsoft.XMLHTTP");
}

xmlhttp.open("GET","books.xml",false);
xmlhttp.send();
xmlDoc = xmlhttp.responseXML;
```

Парсинг XML строки

Следующий фрагмент кода парсит XML строку в объект XML DOM:

```
txt = "<bookstore><book>";
txt = txt + "<title>Everyday Italian</title>";
txt = txt + "<author>Giada De Laurentiis</author>";
txt = txt + "<year>2005</year>";
txt = txt + "</book></bookstore>";

if (window.DOMParser)
{
   parser = new DOMParser();
   xmlDoc = parser.parseFromString(txt,"text/xml");
}
else // Internet Explorer
{
   xmlDoc = new ActiveXObject("Microsoft.XMLDOM");
   xmlDoc.async = false;
   xmlDoc.loadXML(txt);
}
```