КОНСПЕКТ ЛЕКЦИЙ ДИСЦИПЛИНЫ

Алгоритмы и структуры данных

Раздел 1. Введение в предмет. Понятие алгоритма и структуры данных. Классификация структур данных. Вычислительная сложность алгоритмов. Анализ верхней и средней оценок сложности алгоритмов; сравнение наилучших, средних и наихудших оценок.

Для работы с любым видом данных необходимо понимать основные принципы работы с ними. Условно можно выделить два направления: *алгоритмы* и *структура*. Машинное обучение и искусственный интеллект требуют глубокого понимания вышеописанных понятий для полноценного использования в решении задач.

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

Свойства алгоритма:

Выделяют следующие основные свойства для алгоритмов в общем виде:

- **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 - 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.

Линейная структура – упорядоченные по порядковому номеру элементы.

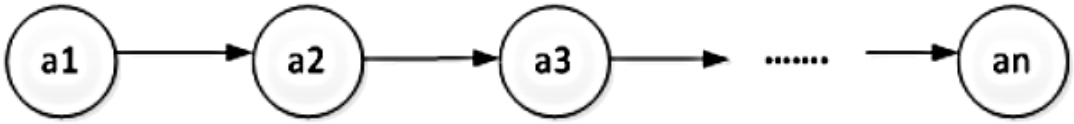


Рис.1 – Линейная структура

Нелинейные структуры – в них порядок определяется по некоторым правилам. В них входят: иерархические(древовидные) и графовые структуры.

Иерархические структуры – в них входят деревья (узлы и ребра). Первый элемент на первом уровне называется корнем. Далее элементы соединяются ребрами, находясь каждый на определённом уровне (может быть несколько на 1 уровень).

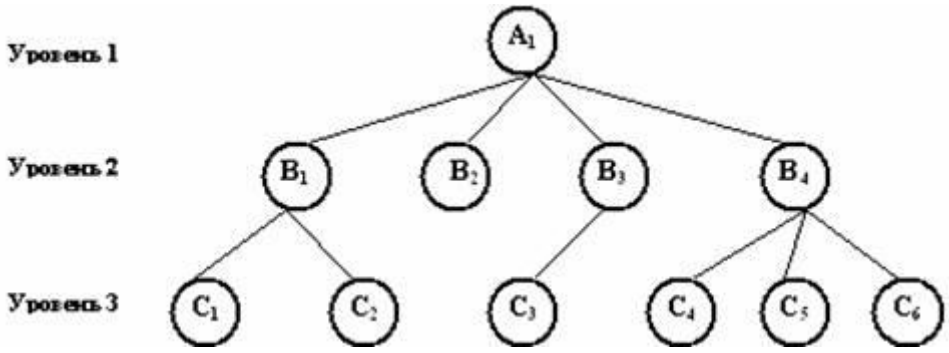


Рис. 2 – Иерархическая структура древовидная

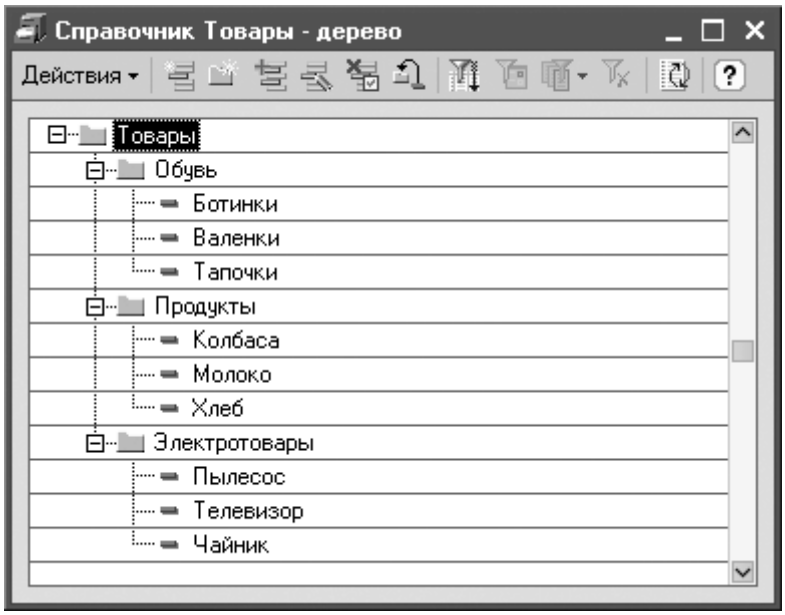


Рис.3 – Иерархический список

Сетевая структура представляется графом: ориентированным(имеется направление) и неориентированным.

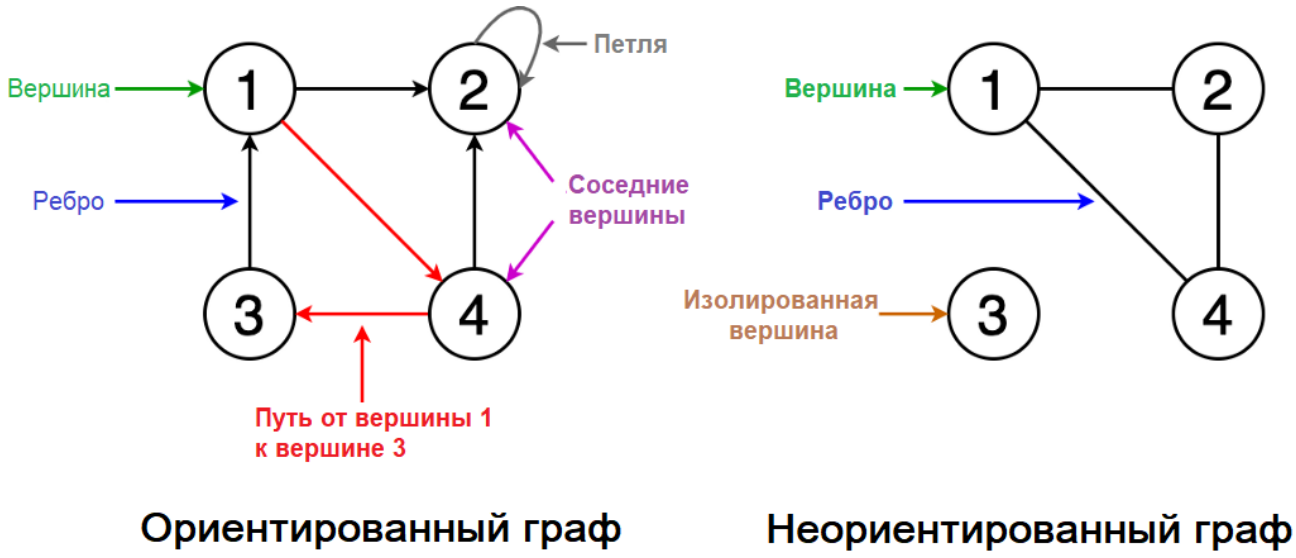


Рис.4 – Графы

Подробнее остальные виды будут рассмотрены в последующих лекциях.

Далее стоит перейти параметрам оценки алгоритмов между собой, к которым относятся: вычислительная сложность алгоритмов, анализ верхней и средней оценок сложности алгоритмов; сравнение наилучших, средних и наихудших оценок.

Вычислительная сложность алгоритмов

Для решения определённого круга задач может быть использовано несколько алгоритмов, но как их сравнивать между собой? Какие требования применяются к алгоритмам?

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**. Поэтому они, как правило, неэффективны при работе с большими множествами. Примером является задача поиска выхода из лабиринта, временные затраты на которую экспоненциально растут с увеличением числа разветвлений.

Рассмотрим более подробно с практической точки зрения. При решении какой-то задачи часто возникает вопрос: время или память? Иными словами, задачу можно решить быстро, при это занимая большое количество памяти для хранения данных. Или же медленно, но задействуя меньший объем. Решить этот вопрос можно, определив приоритет. В случае, если вычисление происходит на машине с ограниченным объемом памяти, теряем в скорости. Если же важна скорость обработки и(!) позволяет память, то используем вариант с использованием памяти. Но, также многое зависит как от задачи, так и от исходных данных.

Пример задачи из сети[1]:

«*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*

[1] Оценка сложности алгоритмов - https://habr.com/ru/post/104219/

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***.

Для реального кода вычислительная сложность может быть крайне сложной функцией. Кроме того, легко ошибиться в расчетах и забыть какое-нибудь из слагаемых.

С другой стороны, есть множество причин, почему нет смысла считать точный вид 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.)

Для оценки сложности инженеры обычно пользуются так называемой O-нотацией. Например, говорят «сложность этого алгоритма $O(n^2)$ (произносится: о от эн квадрат), а этого $\Theta(n \log(n))$ (произносится: тэта от эн лог эн).

Если очень кратко — $O(n^2)$ означает, что самое быстро растущее слагаемое в функции сложности — это n в степени не более 2, а $\Theta(n^2)$ — что степень в точности 2.

Формальные определения:

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 $g(n) = O(f(n))$

Ниже представлен 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 на примере пузырьковой сортировки

При использовани 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.

А что, если подать на вход массив, отсортированный в обратном порядке? Алгоритм этому точно не обрадуется. 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.

Зная это, мы можем примерно понять, какие объемы данных еще можно обрабатывать этим алгоритмом и для нас это будет приемлемо, а какие нет. Так как время работы, чаще всего, напрямую зависит от количества операций, рассмотрим абстрактный пример.

Например, для сортировки 1000 элементов алгоритму в худшем случае потребуется 1 секунда, тогда на сортировку 10000 элементов у алгоритма уйдет уже 100 секунд. Если мы захотим отсортировать 100000 элементов ждать придется уже почти 3 часа.

Как мы выяснили, O-нотация показывает худший случай, но что, если потребуется оценить среднее значение сложности или лучший случай?

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(n2) - в худшем случае, например, когда входные данные отсортированы в обратном порядке, вычислительная сложность алгоритма будет квадратично зависеть от количества элементов.

Θ(n2) - в среднем случае, когда данные перемешаны в случайном порядке вычислительная сложность алгоритма так же будет квадратично зависеть от количества элементов.

Ω(n) - в лучшем случае, когда входные данные уже отсортированы, нам все равно потребуется выполнить один проход по массиву, то есть произвести n операций.

Ниже приведены таблицы сложности алгоритмов

Алгоритмы поиска

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|)

Алгоритмы сортировки

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)

Структуры данных

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)

Кучи

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)

Представление графов

Пусть дан граф с |V| вершинами и |E| ребрами, тогда

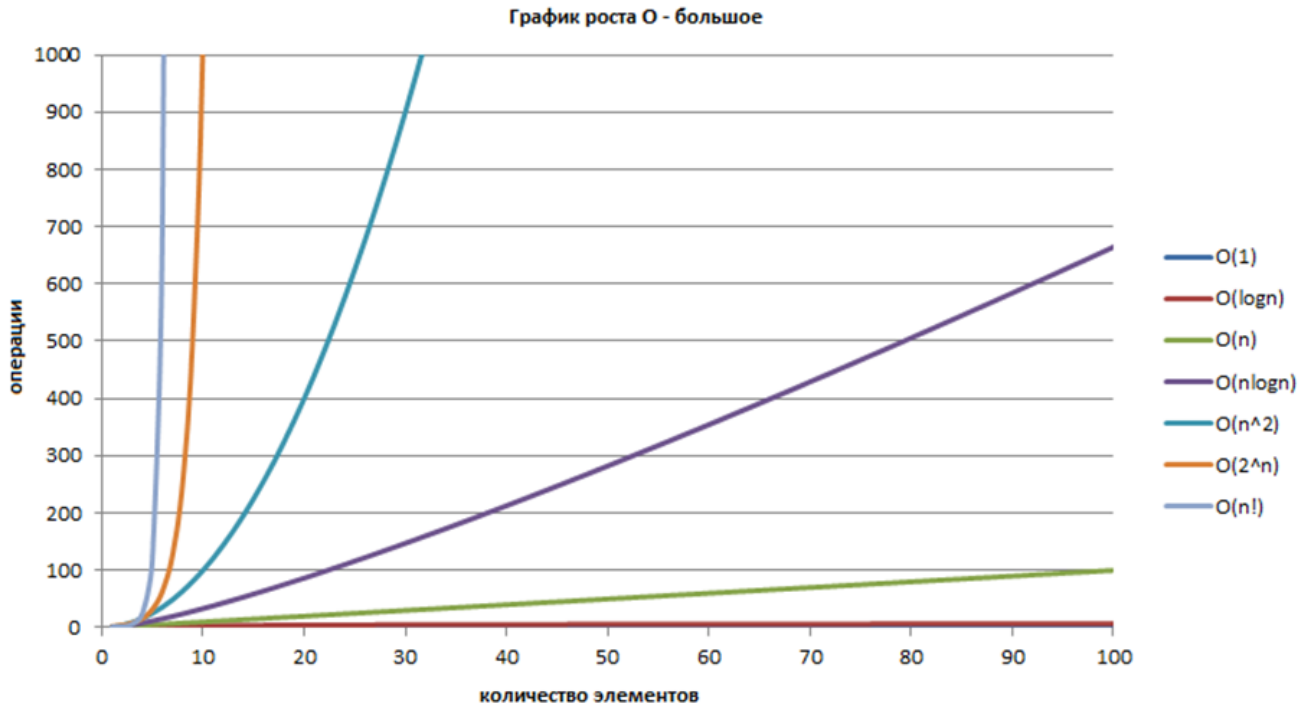
Способ представления	Память	Добавление вершины	Добавление ребра	Удаление вершины	Удаление ребра	Проверка смежности вершин
Список смежности	O(|E| + |V|)	O(1)	O(1)	O(|E| + |V|)	O(|E|)	O(|V|)
Список инцидентности	O(|E| + |V|)	O(1)	O(1)	O(|E|)	O(|E|)	O(|E|)
Матрица смежности	O(|V|^2)	O(|V|^2)	O(1)	O(|V|^2)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

Нотация асимптотического роста

Обозначение	Граница	Рост
(Тета) Θ	Нижняя и верхняя границы, точная оценка	Равно
(О - большое) O	Верхняя граница, точная оценка неизвестна	Меньше или равно
(о - малое) o	Верхняя граница, не точная оценка	Меньше
(Омега - большое) Ω	Нижняя граница, точная оценка неизвестна	Больше или равно
(Омега - малое) ω	Нижняя граница, не точная оценка	Больше

1. (O — большое) — верхняя граница, в то время как (Омега — большое) — нижняя граница. Тета требует как (O — большое), так и (Омега — большое), поэтому она является точной оценкой (она должна быть ограничена как сверху, так и снизу). К примеру, алгоритм требующий Ω (n logn) требует не менее n logn времени, но верхняя граница не известна. Алгоритм требующий Θ (n logn) предпочтительнее потому, что он требует не менее n logn (Ω (n logn)) и не более чем n logn (O(n logn)).

2. f(x)=Θ(g(n)) означает, что f растет так же как и g когда n стремится к бесконечности. Другими словами, скорость роста f(x) асимптотически пропорциональна скорости роста g(n).
3. f(x)=O(g(n)). Здесь темпы роста не быстрее, чем g (n). O большое является наиболее полезной, поскольку представляет наихудший случай.



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.

В предыдущей лекции мы ознакомились с понятием алгоритма, структуры данных, рассмотрели виды структур, проанализировали способы оценки сложности алгоритмов. В данном разделе подробно рассмотрим списки, стек и очередь. Но их понимание невозможно без определения динамических структур данных в целом.

Список однонаправленный и двунаправленный.

Зачастую заранее неизвестно, сколько памяти будет необходимо выделить для выполнения задачи, т.к. количество и вес объектов может меняться в процессе выполнения программы. Для этого существуют **динамические структуры данных**. Из вышесказанного становится понятно, что это такие структуры, под которые память выделяется и освобождается по мере необходимости.

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. Такая особенность динамических структур, как непостоянство их размера и характера отношений между элементами, приводит к тому, что на этапе создания машинного кода программа-компилятор не может выделить для всей структуры в целом участок памяти фиксированного размера, а также не может сопоставить с отдельными компонентами структуры конкретные адреса. Для решения проблемы адресации динамических структур данных используется метод, называемый ***динамическим распределением памяти***, то есть память под отдельные элементы выделяется в момент, когда они "начинают существовать" в процессе выполнения программы, а не во время компиляции. Компилятор в этом случае выделяет фиксированный объем памяти для хранения ***адреса*** динамически размещаемого элемента, а не самого элемента.

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 ***указатель (ее значение – адрес этого объекта)***, посредством которой осуществляется доступ к динамической структуре.

Сами динамические величины не требуют описания в программе, поскольку во время компиляции память под них не выделяется. Во время компиляции

память выделяется только под статические величины. **Указатели** – это статические величины, поэтому они требуют описания.

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.

Поскольку элементы динамической структуры располагаются по непредсказуемым адресам памяти, адрес элемента такой структуры не может быть вычислен из адреса начального или предыдущего элемента. 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.

Последний недостаток является наиболее серьезным и именно им ограничивается применимость связного представления данных. Если в смежном представлении данных для вычисления адреса любого элемента нам во всех случаях достаточно было номера элемента и информации, содержащейся в дескрипторе структуры, то для связного представления адрес элемента не может быть вычислен из исходных данных. Дескриптор связной структуры содержит один или несколько указателей, позволяющих войти в структуру, далее поиск требуемого элемента выполняется следованием по цепочке указателей от элемента к элементу. Поэтому связное представление практически никогда не применяется в задачах, где логическая структура данных имеет вид вектора или массива – с доступом по номеру элемента, но часто применяется в задачах, где логическая структура требует другой исходной информации доступа (таблицы, списки, деревья и т.д.).

Порядок работы с динамическими структурами данных следующий:

- создать (отвести место в динамической памяти);
- работать при помощи указателя;
- удалить (освободить занятое структурой место).

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.

Они отличаются способом связи отдельных элементов и/или допустимыми операциями. Динамическая структура может занимать несмежные участки оперативной памяти.

Динамические структуры широко применяют и для более эффективной работы с данными, размер которых известен, особенно для решения задач сортировки.

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

Длина списка равна числу элементов, содержащихся в списке, список нулевой длины называется пустым списком. Списки представляют собой способ организации структуры данных, при которой элементы некоторого типа образуют цепочку. Для связывания элементов в списке используют систему указателей. В минимальном случае, любой элемент линейного списка имеет один указатель, который указывает на следующий элемент в списке или является пустым указателем, что интерпретируется как конец списка.

Структура, элементами которой служат записи с одним и тем же форматом, связанные друг с другом с помощью указателей, хранящихся в самих элементах, называют ***связанным списком***. В связанном списке элементы линейно упорядочены, но порядок определяется не номерами, как в массиве, а указателями, входящими в состав элементов списка. Каждый список имеет особый элемент, называемый указателем начала списка (головой списка), который обычно по содержанию отличен от остальных элементов. В поле указателя последнего элемента списка находится специальный признак NULL, свидетельствующий о конце списка.

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.

Однонаправленные (односвязные) и двунаправленные связные списки

Наиболее простой динамической структурой является однонаправленный список, элементами которого служат объекты структурного типа.

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.

Рис1. Односвязный линейный список

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

Рис.2. Односвязный циклический список

Двусвязный линейный список (ДЛС)- каждый узел ДЛС содержит два поля указателей: на следующий и на предыдущий узел. Поле указателя на следующий узел последнего узла содержит нулевое значение (указывает на NULL). Поле указателя на предыдущий узел первого узла (корня списка) также содержит

нулевое значение (указывает на NULL).

Рис.3. Двусвязный линейный список

Двусвязный циклический список (ДЦС) - каждый узел ДЦС содержит два поля указателей: на следующий и на предыдущий узел. Поле указателя на следующий узел последнего узла содержит адрес первого узла (корня списка). Поле указателя на предыдущий узел первого узла (корня списка) содержит адрес последнего узла.

Рис.4. Двусвязный циклический список (ДЦС)

Разобравшись с теоретическим описанием списков, перейдем к **программной реализации**.

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.

Особое внимание следует обратить на то, что при выполнении любых операций с линейным однонаправленным списком необходимо обеспечивать позиционирование какого-либо указателя на первый элемент. В противном случае часть или весь список будет недоступен.

Создание однонаправленного списка

Для того, чтобы создать *список*, нужно создать сначала первый *элемент списка*, а затем при помощи функции добавить к нему остальные элементы. При относительно небольших размерах списка наиболее изящно и красиво использование *рекурсивной функции*. Добавление может выполняться как в начало, так и в конец 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

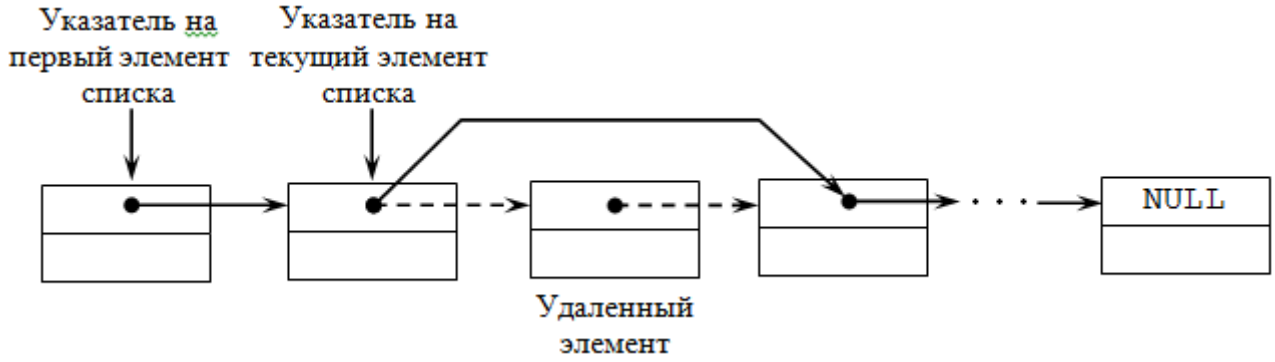
Операция печати списка заключается в последовательном просмотре всех элементов списка и выводе их значений на экран. Для обработки списка организуется *функция*, в которой нужно переставлять *указатель* на следующий *элемент списка* до тех пор, пока *указатель* не станет равен NULL, то есть будет достигнут конец списка. Реализуем данную функцию рекурсивно.

```
//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";
}
```

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;
  }
}
```

Таким образом, однонаправленный список имеет только один указатель в каждом элементе. Это позволяет минимизировать расход памяти на организацию такого списка. Одновременно это позволяет осуществлять переходы между элементами только в одном направлении, что зачастую увеличивает время, затрачиваемое на обработку списка. Например, для перехода к предыдущему элементу необходимо осуществить просмотр списка с начала до элемента, указатель которого установлен на текущий элемент.

Двунаправленные (двусвязные) списки

Для ускорения многих операций целесообразно применять переходы между элементами списка в обоих направлениях. Это реализуется с помощью *двунаправленных* списков, которые являются сложной динамической структурой.

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.

В таком списке каждый элемент (кроме первого и последнего) связан с предыдущим и следующим за ним элементами. Каждый элемент *двунаправленного* списка имеет два поля с указателями: одно *поле* содержит ссылку на следующий элемент, другое *поле* – ссылку на

предыдущий элемент и третье *поле* – информационное. Наличие ссылок на следующее звено и на предыдущее позволяет двигаться по списку от каждого звена в любом направлении: от звена к концу списка или от звена к началу списка, поэтому такой *список* называют двунаправленным.

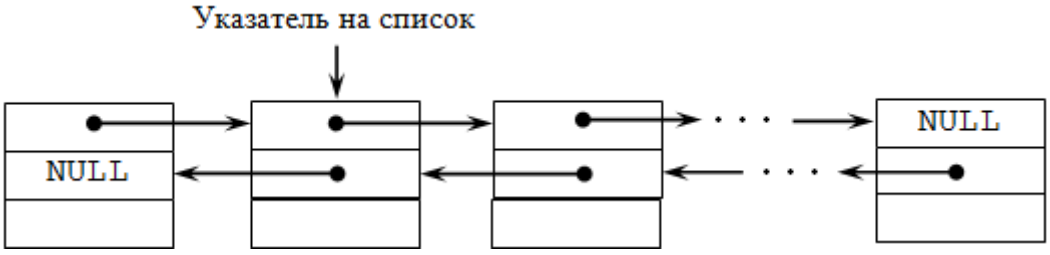


Рис. 29.4. Двунаправленный список

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*.

Например:

```
struct list {
              type elem ;
              list *next, *pred ;
            }
list *headlist ;
```

где type – тип информационного поля элемента списка;

*next, *pred – указатели на следующий и предыдущий элементы этой структуры соответственно.

Переменная-указатель headlist задает *список* как единый программный *объект*, ее *значение – указатель* на первый (или заглавный) *элемент списка*.

Основные *операции*, выполняемые над двунаправленным списком, те же, что и для однонаправленного списка. Так как *двунаправленный список* более гибкий, чем однонаправленный, то при включении элемента в *список*, нужно использовать *указатель* как на элемент, за которым происходит включение, так и *указатель* на элемент, перед которым происходит включение. При исключении элемента из списка нужно использовать как *указатель* на сам исключаемый элемент, так и указатели на предшествующий или следующий за исключаемым элементы. Но так как элемент *двунаправленного* списка имеет два

указателя, то при выполнении *операций включения*/исключения элемента надо изменять больше связей, чем в однонаправленном списке.

Рассмотрим 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.

Особое внимание следует обратить на то, что в отличие от однонаправленного списка здесь нет необходимости обеспечивать *позиционирование* какого-либо указателя именно на первый *элемент списка*, так как благодаря двум указателям в элементах можно получить *доступ* к любому элементу списка из любого другого элемента, осуществляя переходы в прямом или обратном направлении. Однако по правилам хорошего тона программирования *указатель* желательно ставить на заголовок 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;
```

```
}
```

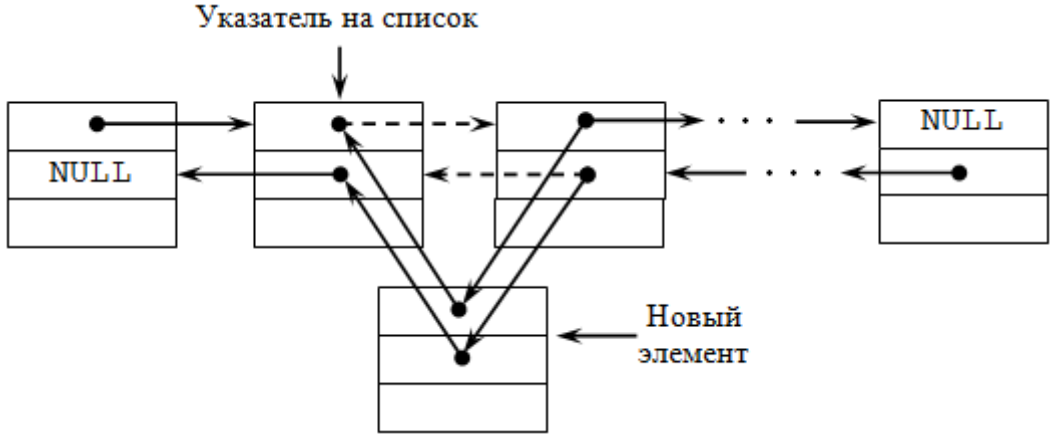
Печать (просмотр) двунаправленного списка

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.



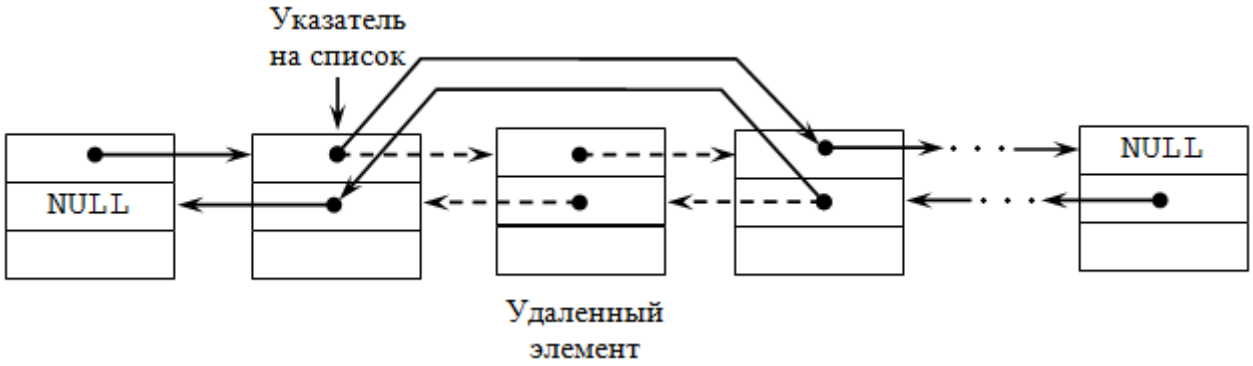
Добавление элемента в двунаправленный список

```
//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

Из динамических структур можно удалять элементы, так как для этого достаточно изменить значения адресных полей. *Операция удаления элемента* из *двунаправленного* списка осуществляется во многом аналогично удалению из однонаправленного списка .




```
/*удаление элемента с заданным номером 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 и очередь. Способы программной организации стека и очереди, и обработка данных.

Стек и очередь – это частные случаи линейного списка. Рассмотри подробнее.

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.

Стек – это список, у которого доступен один элемент (одна позиция). Этот элемент называется вершиной стека. Взять элемент можно только из вершины стека, добавить элемент можно только в вершину стека. Например, если записаны в стек числа 1, 2, 3, то при последующем извлечении получим 3,2,1.

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 как *динамическую структуру данных* легко организовать на основе линейного списка. Поскольку работа всегда идет с заголовком стека, то есть не требуется осуществлять просмотр элементов, удаление и вставку элементов в середину или конец списка, то достаточно использовать экономичный по памяти линейный однонаправленный *список*. Для такого списка достаточно хранить *указатель вершины стека*, который указывает на первый *элемент списка*. Если *стек* пуст, то списка не существует, и *указатель* принимает *значение* NULL.

Описание элементов стека аналогично описанию элементов линейного однонаправленного списка. Поэтому объявим *стек* через объявление линейного однонаправленного списка:

```
struct Stack {
              Single_List *Top;//вершина стека
             };
. . . . . .
```

```
Stack *Top_Stack;//указатель на вершину стека
```

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);
}

//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;
}

//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);
```

```
}
```

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.

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 ;
           }
```

Очередь как *динамическую структуру данных* легко организовать на основе линейного списка. Поскольку работа идет с обоими концами очереди, то предпочтительно будет использовать линейный двунаправленный *список*. Хотя для работы с таким списком достаточно иметь один *указатель* на любой *элемент списка*, здесь целесообразно хранить два указателя – один на начало списка (откуда извлекаем элементы) и один на конец списка (куда добавляем элементы). Если *очередь* пуста, то списка не существует, и указатели принимают *значение* NULL.

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.

Реализацию этих операций приведем в виде соответствующих функций, которые, в свою *очередь*, используют функции операций с линейным двунаправленным списком.

```
//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.

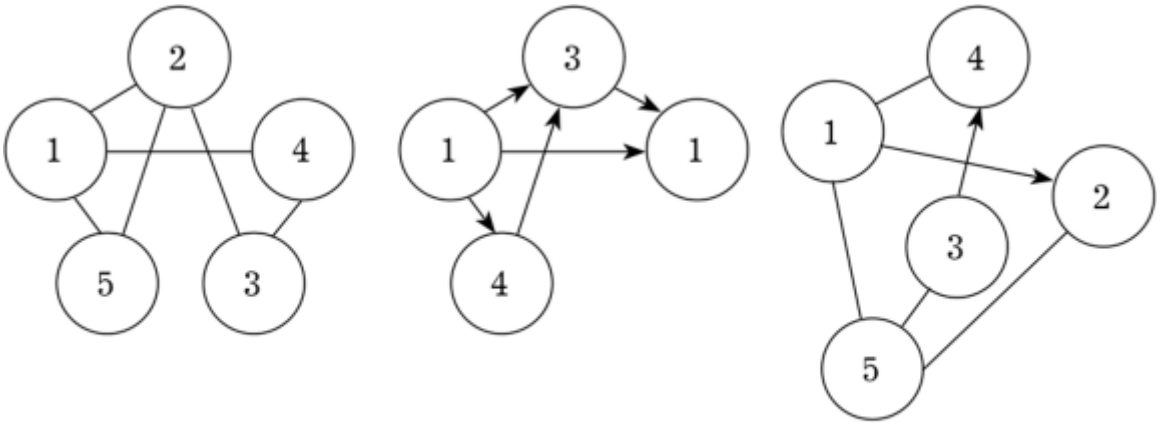
Способы задания графов. Матрица инцидентности, матрица смежности, матрица весов, список ребер, список смежности.

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.

Продолжаем изучение структур данных. Следующий вид, который мы рассмотрим – графы. Данная структура широко используется во множестве оптимизационных задач, теории принятия решений. В когнитивном моделировании пример использования графовой формы – когнитивная карта. Также, в интеллектуальных системах знания могут быть представлены в виде онтологии – отображать связь между понятиями в предметной области и их отношения.

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.



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

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.***

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.

Граф H, множество вершин V’ которого является подмножеством вершин V данного графа G и множество рёбер которого является подмножеством рёбер графа G соединяющими вершины из V’ называется подграфом графа G.

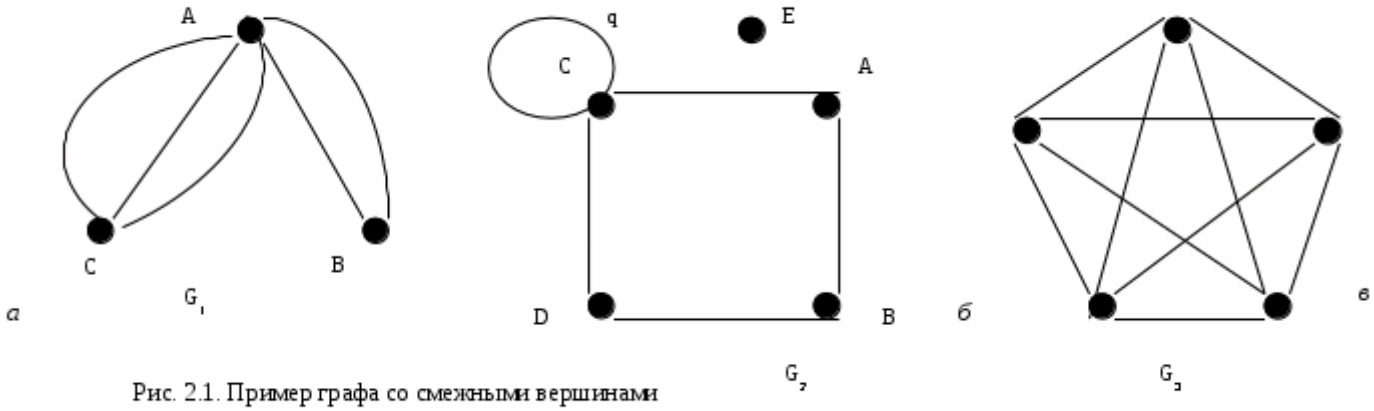


Рис. 2.1. Пример графа со смежными вершинами
а – со смежными вершинами; б – с петлёй, в - полный

На рисунке показаны: граф со смежными вершинами(соединены ребром С и А, А и В.), граф с петлей(С) и изолированной вершиной (Е), полный граф(Все вершины соединены ребрами между собой).

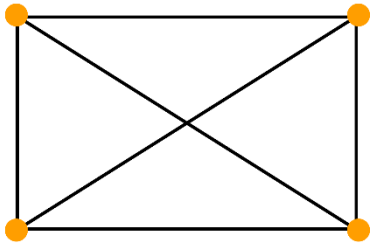
Эйлеров граф отличен тем, что в нем можно обойти все вершины и при этом пройти одно ребро только один раз. В нём каждая вершина должна иметь только чётное число рёбер.

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

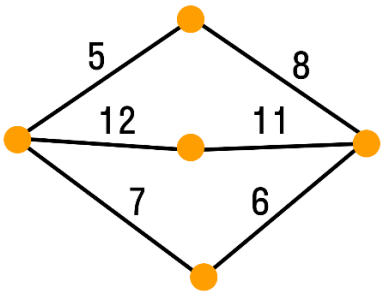
Ответ. Если n — нечётное число, то каждая вершина инцидентна n - 1 ребрам. В таком случае наш граф — эйлеровый.

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**.**

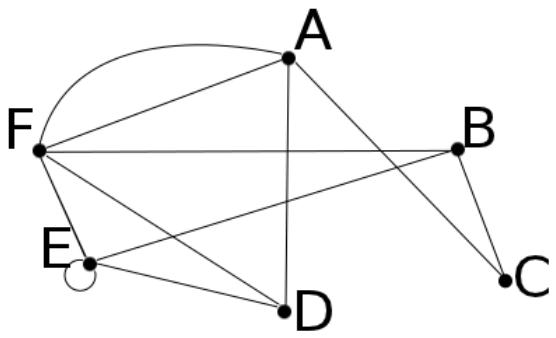


Способы задания графов. Матрица инцидентности, матрица смежности, матрица весов, список ребер, список смежности.

Существует несколько способов задать граф. Графический мы рассмотрели выше, с помощью вершин и ребер. Также, задать граф можно с помощью матрицы.

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:

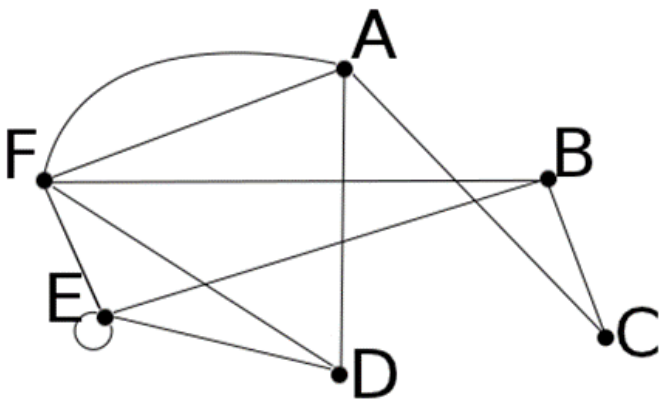
$$b_{ij} = \begin{cases} 1, & \text{если } e_j \text{ инцидентно } V_i \\ 0, & \text{если } e_j \text{ не инцидентно } V_i \end{cases}$$

У матрицы инцидентности характеризуются:

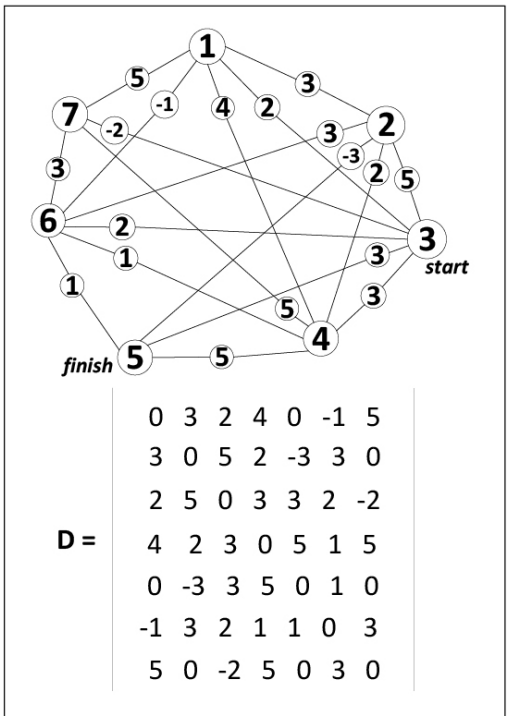
- в каждой строке должны стоять две единицы, а все остальные символы — нули;
- она не используется для графов с петлями.

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

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

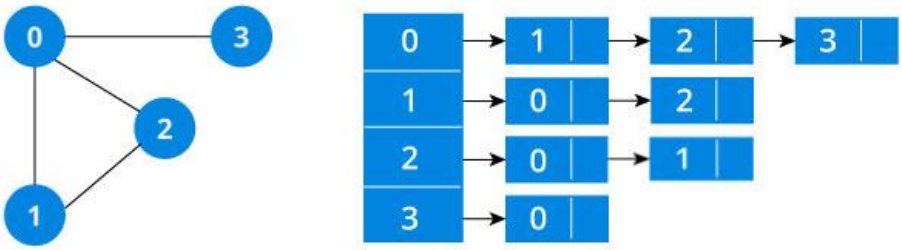


Матрица весов строится для взвешенного графа аналогично матрице смежности, однако указывается вес.



$$D = \begin{pmatrix} 0 & 3 & 2 & 4 & 0 & -1 & 5 \\ 3 & 0 & 5 & 2 & -3 & 3 & 0 \\ 2 & 5 & 0 & 3 & 3 & 2 & -2 \\ 4 & 2 & 3 & 0 & 5 & 1 & 5 \\ 0 & -3 & 3 & 5 & 0 & 1 & 0 \\ -1 & 3 & 2 & 1 & 1 & 0 & 3 \\ 5 & 0 & -2 & 5 & 0 & 3 & 0 \end{pmatrix}$$

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



Как видно из рисунка слева указываются вершины по порядку, а справа перечисляются все, что с ними связаны. Если граф ориентированный, то указываются справа те вершины, в которые входит дуга.

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.

Самому простому списку смежности требуется структура данных узла для хранения вершины и структура данных графа для организации узлов.
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++

```
#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.

```
#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
          */
}
```

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).

Эти методы чаще всего рассматриваются на ориентированных графах, но они применимы и для неориентированных, ребра которых считаются двунаправленными. Алгоритмы обхода в глубину и в ширину лежат в основе решения различных задач обработки графов, например, построения остовного леса, проверки связности, ацикличности, вычисления расстояний между вершинами и других.

Поиск в глубину

При *поиске в глубину* посещается первая *вершина*, затем необходимо идти вдоль ребер графа, до попадания в *тупик*. *Вершина* графа является *тупиком*, если все смежные с ней вершины уже посещены. После попадания в *тупик* нужно возвращаться назад вдоль пройденного пути, пока не будет обнаружена *вершина*, у которой есть еще не посещенная *вершина*, а затем необходимо двигаться в этом новом направлении. Процесс оказывается завершенным при возвращении в начальную вершину, причем все смежные с ней вершины уже должны быть посещены.

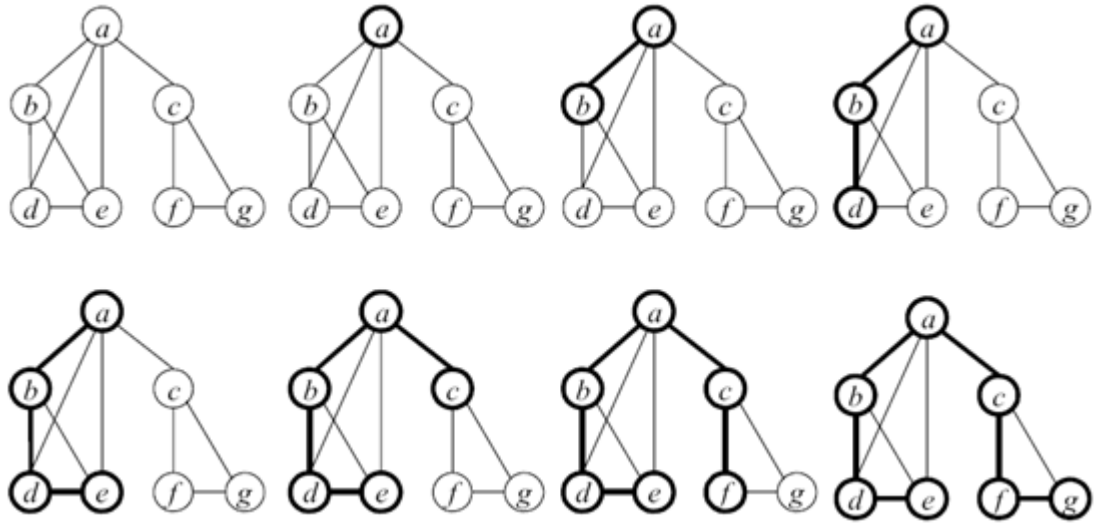
Таким образом, основная идея поиска в глубину – когда возможные пути по *ребрам*, выходящим из вершин, разветвляются, нужно сначала полностью исследовать одну ветку и только потом переходить к другим веткам (если они останутся нерассмотренными).

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:

```
//Описание функции 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

При *поиске в ширину*, после посещения первой вершины, посещаются все соседние с ней вершины. Потом посещаются все вершины, находящиеся на расстоянии двух ребер от начальной. При каждом новом шаге посещаются вершины, *расстояние* от которых до начальной на единицу больше предыдущего. Чтобы предотвратить повторное посещение вершин, необходимо вести *список* посещенных вершин. Для хранения временных данных, необходимых для работы алгоритма, используется *очередь* – упорядоченная последовательность элементов, в которой новые элементы добавляются в конец, а старые удаляются из начала.

Таким образом, основная 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

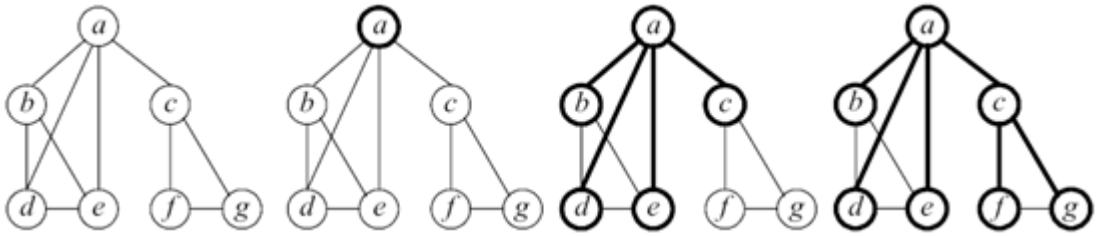
3 и т.д. Обратим внимание, что при этом для каждой вершины сразу находятся *длина* кратчайшего маршрута от начальной вершины.

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

```
//Описание функции 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.

Нахождение кратчайшего пути является практически главной задачей использование графов. Задача кратчайшего пути используется для оптимизации процессов, планирования перевозок, коммутации информационного пакета в сетях и т.п.

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

Данный *алгоритм* является алгоритмом на графах, который изобретен нидерландским ученым Э. Дейкстрой в 1959 году. *Алгоритм* находит кратчайшее *расстояние* от одной из *вершин графа* до всех остальных и работает только для графов без ребер отрицательного веса.

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 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.

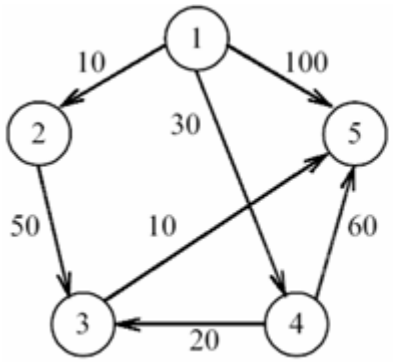
Шаг 6. Если текущей вершиной оказывается конечная, то *путь* найден, и его *вес* есть *вес* конечной вершины.

Шаг 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}

```
//Описание функции алгоритма Дейкстры
void Dijkstra(int n, int **Graph, int Node){
  bool *S = new bool[n];
  int *D = new int[n];
  int *P = new int[n];
  int i, j;
  int Max_Sum = 0;
  for (i = 0 ; i < n ; i++)
    for (j = 0 ; j < n ; j++)
      Max_Sum += Graph[i][j];
  for (i = 0 ; i < n ; i++)
    for (j = 0 ; j < n ; j++)
      if (Graph[i][j] == 0)
        Graph[i][j] = Max_Sum;
  for (i = 0 ; i < n ; i++){
```

```
    S[i] = false;
    P[i] = Node;
    D[i] = Graph[Node][i];
  }
  S[Node] = true;
  P[Node] = -1;
  for ( i = 0 ; i < n - 1 ; i++ ){
    int w = 0;
    for ( j = 1 ; j < n ; j++ ){
      if (!S[w]){
        if (!S[j] && D[j] <= D[w])
          w = j;
      }
      else w++;
    }
    S[w] = true;
    for ( j = 1 ; j < n ; j++ )
      if (!S[j])
        if (D[w] + Graph[w][j] < D[j]){
          D[j] = D[w] + Graph[w][j];
          P[j] = w;
        }
  }
  for ( i = 0 ; i < n ; i++ )
    printf("%5d",D[i]);
  cout << endl;
  for ( i = 0 ; i < n ; i++ )
    printf("%5d",P[i]+1);
  cout << endl;
  delete [] P;
  delete [] D;
  delete [] S;
}
```

Сложность *алгоритма Дейкстры* зависит от способа нахождения вершины, а также способа хранения *множества* непосещенных вершин и способа обновления длин.

Если для представления графа использовать *матрицу смежности*, то *время выполнения* этого алгоритма имеет порядок $O(n^2)$, где n – количество *вершин графа*.

Алгоритм Флойда

Рассматриваемый *алгоритм* иногда называют *алгоритмом Флойда*-Уоршелла. *Алгоритм Флойда*-Уоршелла является алгоритмом на графах, который разработан в 1962 году Робертом Флойдом и Стивеном Уоршеллом. Он служит для нахождения *кратчайших путей* между всеми парами *вершин графа*.

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.

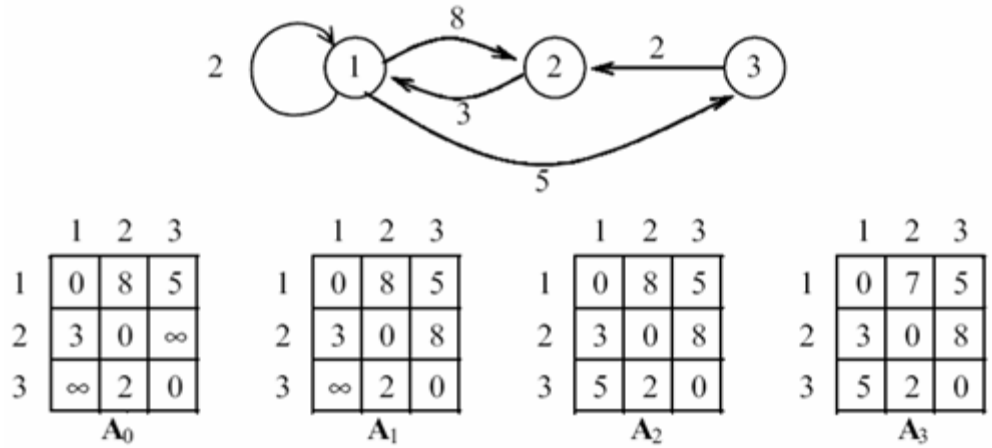
Основной шаг k. Задаем строку k и столбец k как ведущую строку и ведущий столбец. Рассматриваем возможность применения замены описанной выше, ко всем элементам A[i,j] матрицы A_{k-1}. Если выполняется *неравенство*

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

тогда выполняем следующие действия:

1. создаем матрицу A_k путем замены в матрице A_{k-1} элемента A[i,j] на сумму A[i,k]+A[k,j] ;
2. 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.




```
//Описание функции алгоритма Флойда
void Floyd(int n, int **Graph, int **ShortestPath){
  int i, j, k;
  int Max_Sum = 0;
  for ( i = 0 ; i < n ; i++ )
    for ( j = 0 ; j < n ; j++ )
      Max_Sum += ShortestPath[i][j];
  for ( i = 0 ; i < n ; i++ )
    for ( j = 0 ; j < n ; j++ )
      if ( ShortestPath[i][j] == 0 && i != j )
        ShortestPath[i][j] = Max_Sum;
  for ( k = 0 ; k < n; k++ )
    for ( i = 0 ; i < n; i++ )
```

```
      for ( j = 0 ; j < n ; j++ )
        if ((ShortestPath[i][k] + ShortestPath[k][j]) <
             ShortestPath[i][j])
          ShortestPath[i][j] = ShortestPath[i][k] +
            ShortestPath[k][j];
}
```

Заметим, что если *граф* неориентированный, то все матрицы, получаемые в результате преобразований симметричны и, следовательно, достаточно вычислять только элементы, расположенные выше главной диагонали.

Если *граф* представлен *матрицей смежности*, то *время выполнения* этого алгоритма имеет порядок $O(n^3)$, поскольку в нем присутствуют вложенные друг в друга три *цикла*.

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

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 (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. Существует несколько частных случаев общей постановки задачи, в частности, геометрическая задача коммивояжёра (также называемая планарной или евклидовой, когда матрица расстояний отражает расстояния между точками на плоскости), метрическая задача коммивояжёра (когда на матрице стоимостей выполняется неравенство треугольника), симметричная и асимметричная задачи коммивояжёра. Также существует обобщение задачи, так называемая обобщённая задача коммивояжёра.

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.

Приведём небольшую классификацию вариантов задачи:

- 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. При поиске оптимального решения строится бинарное дерево (в каждом узле порождаются 2 ветви: коммивояжёр идёт в некоторый город или не идёт в него).
- 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.

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 граф мы рассмотрели ранее. Существует такое понятие, как **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).*

Давайте смотреть исходный код Эйлерова цикла. Ввод данных с клавиатуры.

```
#include <iostream.h>
#include <stdlib.h>

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 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.

Параллельная обработка. Если устройство выполняет одну операцию за единицу времени, то тысячу операций оно выполнит за тысячу единиц. Если предположить, что есть пять таких же независимых устройств, способных работать одновременно, то ту же тысячу операций система из пяти устройств может выполнить уже не за тысячу, а за двести единиц времени. Аналогично система из N устройств ту же работу выполнит за 1000/N единиц времени. Подобные аналогии можно найти и в жизни: если один солдат вскопает огород за 10 часов, то рота солдат из пятидесяти человек с такими же способностями, работая одновременно, справятся с той же работой за 12 минут.

Конвейерная обработка. Что необходимо для сложения двух вещественных чисел, представленных в форме с плавающей запятой? Целое множество мелких операций таких, как сравнение порядков, выравнивание порядков, сложение мантисс, нормализация и т.п. Процессоры первых компьютеров выполняли все эти "микрооперации" для каждой пары аргументов последовательно одна за одной до тех пор, пока не доходили до окончательного результата, и лишь после этого переходили к обработке следующей пары слагаемых.

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. Предположим, что в операции можно выделить пять микроопераций, каждая из которых выполняется за одну единицу времени. Если есть одно неделимое последовательное устройство, то 100 пар аргументов оно обработает за 500 единиц. Если каждую микрооперацию выделить в отдельный этап (или иначе говорят - ступень) конвейерного устройства, то на пятой единице времени на разной стадии обработки такого устройства будут находится первые пять пар аргументов, а весь набор из ста пар будет обработан за 5+99=104 единицы времени - ускорение по сравнению с последовательным устройством почти в пять раз (по числу ступеней конвейера).

Казалось бы конвейерную обработку можно с успехом заменить обычным параллелизмом, для чего продублировать основное устройство столько раз, сколько ступеней конвейера предполагается выделить. В самом деле, пять устройств предыдущего примера обработают 100 пар аргументов за 100 единиц времени, что быстрее времени работы конвейерного устройства! В чем же дело? Ответ прост, увеличив в пять раз число устройств, мы значительно увеличиваем

как объем аппаратуры, так и ее стоимость. Представьте себе, что на автозаводе решили убрать конвейер, сохранив темпы выпуска автомобилей. Если раньше на конвейере одновременно находилась тысяча автомобилей, то действуя по аналогии с предыдущим примером надо набрать тысячу бригад, каждая из которых (1) в состоянии полностью собрать автомобиль от начала до конца, выполнив сотни разного рода операций, и (2) сделать это за то же время, что машина прежде находилась на конвейере. Представили себестоимость такого автомобиля?

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.

Параллельный алгоритм – алгоритм, предназначенный для реализации на параллельной вычислительной системе.

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. Тем самым, библиотека PFX упрощает написание *параллельных программ*, обеспечивая *увеличение производительности* при увеличении числа ядер или числа процессоров, исключая многие сложности современных моделей *параллельного программирования*. Первая версия библиотеки была представлена 29 ноября 2007 года, впоследствии выходили обновления в декабре 2007 и июне 2008 годов. На момент написания данных лекций, библиотека PFX вошла в состав *.NET* 4 *CTP* и *Visual* Studio 2008/2010.

Parallel **Extensions** обеспечивает несколько новых способов организации *параллелизма*:

- *Параллелизм* при декларативной обработке данных. Реализуется при помощи параллельного интегрированного языка запросов (PLINQ) - параллельной реализации *LINQ*. Отличие от *LINQ* заключается в том, что запросы выполняются параллельно, обеспечивая *масштабируемость* и загрузку доступных ядер и процессоров.
- *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*.

Parallel Extensions - это управляемая (managed) библиотека и для своей работы она требует установленный *.NET* Framework 3.5.

Лучший способ использования Parallel Extensions

Библиотека *Parallel* Extensions была разработана в целях обеспечения наибольшей производительности при использовании многоядерных процессоров или *многопроцессорных* машин. Вместо того чтобы принимать решение о *верхней границе* количества распараллеливаемых задач во время разработки, библиотека позволяет автоматически масштабировать выполняемые задачи, динамически вовлекая в работу все большее количество ядер, по мере того как они становятся доступными. Прирост производительности достигается при использовании *Parallel* Extensions на многоядерных процессорах или *многопроцессорных* машинах. Вместе с этим *Parallel* Extensions разработана так, чтобы минимизировать издержки и при выполнении на однопроцессорных машинах.

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.

Parallel Extensions содержит менеджер задач, который планирует выполнение задач. Другое название менеджера задач

- *планировщик*. *Планировщик* управляет множеством рабочих потоков, на которых происходит выполнение задач. По умолчанию, создается число потоков равное числу процессоров (ядер). Каждый поток связан со своей очередью задач. По завершении выполнения очередной задачи, поток извлекает следующую задачу из своей *локальной очереди*. Если же она пуста, то он может взять для исполнения задачу, находящуюся в *локальной очереди* другого рабочего потока. Задачи выполняются независимо друг от друга. Поэтому при использовании ими *разделяемых ресурсов* требуется выполнять синхронизацию при помощи блокировок или других конструкций.

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 на сайте 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.

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.

Кроме того, исполнением задач управляет планировщик задач, который работает с пулом потоков. Это, например, означает, что несколько задач могут разделять один и тот же поток. Класс Task (и вся остальная библиотека TPL) определены в пространстве имен System.Threading.Tasks.

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.

BackgroundWorker

Выполнение многих часто выполняемых операций может занимать длительное время. Пример:

- Загрузка изображений
- Вызовы веб-служб
- Скачивание и загрузка файлов (в т. ч. через одноранговые приложения)
- Сложные локальные вычисления
- Транзакции баз данных
- Обращение к локальному диску в случае низкой скорости по сравнению с доступом к памяти

Такие операции могут привести к блокировке пользовательского интерфейса во время их выполнения. Если вы хотите получить отзывчивый пользовательский интерфейс, но столкнулись с длительными задержками в результате выполнения таких операций, удобным решением станет компонент BackgroundWorker.

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.

BackgroundWorkerКомпонент доступен в BackgroundWorkerна вкладке компоненты . Чтобы добавить в форму, перетащите BackgroundWorker компонент на форму. Он отображается в области компонентов, и его свойства отображаются в окне Свойства .

Для запуска асинхронной работы используйте метод RunWorkerAsync. RunWorkerAsync принимает необязательный object параметр, который можно использовать для передачи аргументов в рабочий метод. Класс BackgroundWorker показывает событие DoWork, к которому обработчик событий DoWork прикрепляет рабочий поток.

Обработчик событий DoWork задействует параметр DoWorkEventArgs со свойством Argument. Данное свойство получает параметр из RunWorkerAsync и может быть передано в рабочий метод, который будет вызываться в обработчике событий DoWork. В следующем примере показан способ назначения результата из рабочего метода, который называется ComputeFibonacci. Он является частью большого примера, который можно найти в разделе как реализовать форму, использующую фоновую операцию.

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

```
public class BackgroundWorker : System.ComponentModel.Component
```

BackgroundWorkerКласс позволяет выполнять операцию в отдельном выделенном потоке. Длительные операции, такие как загрузка и транзакции базы данных, могут привести к тому, что пользовательский интерфейс будет выглядеть так, будто он перестает отвечать на запросы во время работы. Если вам нужен пользовательский интерфейс для реагирования и вы столкнулись с длительными задержками, связанными с такими операциями, BackgroundWorker класс предоставляет удобное решение.

Чтобы выполнить трудоемкую операцию в фоновом режиме, создайте BackgroundWorker и прослушайте события, которые сообщают о ходе выполнения операции и подают сигнал о завершении операции. Можно создать BackgroundWorker программно или перетащить его на форму с вкладки компоненты панели элементов. если создать BackgroundWorker в конструктор Windows Forms, он появится в области компонентов, а его свойства будут отображаться в окно свойств.

Базовые принципы разработки распараллеливания алгоритмов на центральном процессоре

Разработка параллельных алгоритмов состоит из следующих этапов:

1. Декомпозиция задачи на подзадачи, которые реализуются независимо. 2. Определение для сформированного набора подзадач информационных взаимодействий.

3. Масштабирование подзадач, определение количества процессоров.

4. Определение архитектуры системы, закрепление подзадач за процессорами, составление расписания.

Этапы 1-4 могут повторяться, в случае необходимости, например для улучшения эффективности алгоритма. Если желаемые показатели не достигаются, то следует изменить математическую модель задачи. Приведенная схема является общей, в этой связи в каждом конкретном случае последовательность этапов может меняться. Например, если заранее не известно точное число процессоров, но известны границы решающего поля, разработку алгоритма можно начать с масштабирования базового набора задач и только потом перейти к декомпозиции и выявление связей по информации. Схема взаимосвязи типовых этапов разработки алгоритмов параллельных вычислений приведена на рисунке.

На этапе декомпозиции осуществляется выделение базовых подзадач. В процессе декомпозиции предъявляются минимальные требования обеспечить:

- примерно равный объем вычислений в выделяемых подзадачах;
- минимальный информационный обмен данными между процессорами. На этапе анализа информационных зависимостей между подзадачами различают следующие типы этих зависимостей:
- локальные (на соседних процессорах) и глобальные (в которых принимают участие все процессоры) схемы передачи данных;
- структурные (соответствующие типовым топологиям коммуникаций) и произвольные способы взаимодействия;
- статические (задаваемые на этапе проектирования) или динамические (определяемые в ходе выполняемых вычислений);
- синхронные (следующая операция выполняется после выполнения предыдущей операции всеми процессорами) и асинхронные способы взаимодействия (процессы могут не дожидаться полного завершения действий по передаче данных) подзадачи обладают высокой степенью информационной взаимозависимости.

Этап масштабирования параллельного алгоритма выполняется в случае, если количество подзадач (областей данных) отличается от числа процессоров. Тогда осуществляется переход на этап декомпозиции. При этом, число подзадач уменьшают за счет укрупнения области исходных данных. В первую очередь следует объединить области, для которых подзадачи обладают высокой степенью информационной взаимозависимости.

На этапе закрепления задач за процессорами следует учитывать информационные взаимодействия между областями данных этих задач. Такие задачи целесообразно размещать на процессорах, связанных прямыми линиями передачи данных.

Приведенная схема может использоваться для построения параллельного алгоритма, который характеризуется параллелизмом данных и параллелизмом задач. Если имеет место параллелизм данных, то задача сводится к разбиению массива исходных данных на фрагменты, обработка которых ведется независимо

на различных процессорах. Должно соблюдаться требование равномерная загрузка процессоров.

При этом следует учитывать возможность различной производительности процессоров. 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.

В современном обществе особое место занимают социальные сети. Они уже интегрированы в повседневную жизнь. Соцсети уже не развлечение и инструмент для общения, это - огромная площадка, используемая для торговли, рекламы, анализа рынка и предпочтений различных групп потенциальных или действующих потребителей. Иными словами, это огромный источник гетерогенной информации, полезной для аналитиков разных сфер. Потому, рассмотрим отдельно анализ социальных сетей.

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.;

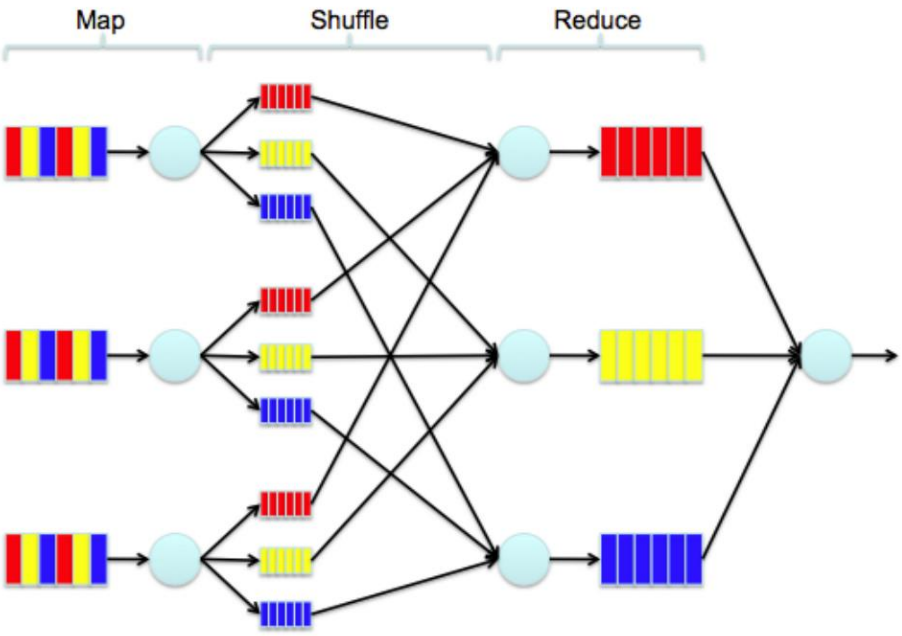
– содержащие данные смешанного типа видео – FaceBook, ВКонтакте и т.д.

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

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.

При этом 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. 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.



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). Аналогично множество рабочих узлов могут осуществлять свёртку – для этого необходимо, чтобы все результаты предварительной обработки с одним конкретным значением ключа обрабатывались одним рабочим узлом в один момент времени.

Хотя этот процесс может быть менее эффективным по сравнению с более последовательными алгоритмами, технология MapReduce может быть применена к большим объёмам данных, которые могут обрабатываться большим количеством серверов. Так, MapReduce может быть использована для сортировки петабайта данных, что займёт всего лишь несколько часов. Параллелизм также даёт некоторые возможности восстановления после частичных сбоев серверов: если в рабочем узле, производящем операцию предварительной обработки или свёртки, возникает сбой, то его работа может быть передана другому рабочему узлу (при условии, что входные данные для проводимой операции доступны). Обработанные и структурированные данные анализируются в соответствие с их типом. Отдельные примеры подобной аналитики представлены в следующих разделах данного учебного пособия.

Для проведения анализа любых данных, в том числе полученных из социальных сетей, необходимо организовать их сбор и дальнейшую обработку. 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 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.

Важно, что аккуратность меры центральности зависит от топологии сети, а сложные сети имеют неоднородную топологию. Поэтому меры центральности, пригодные для выявления узлов с высокой степенью влиятельности, скорей всего, будут непригодны для остатка сети. Из-за этого для вычисления влиятельности в сетях со сложной топологией используют показатель ***доступность***, который измеряет разброс невозвратных блужданий (последовательность смежных вершин, посещаемых однократно), начинающихся с данного узла.

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. В FOAF описание социальных связей между людьми основывается на транзитивности доверия. Описание алгоритма вычисления уровня доверия (TrustRank) приведено ниже. Можно выделить три типа графовых моделей .

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.

Графовые модели социальных сетей используются для моделирования экономических и коммуникационных связей людей, анализа процессов

распространения информации, нахождения сообществ и связанных подгрупп, на которые можно разбить всю социальную сеть.

Анализ центральности и других локальных свойств.

Чтобы определить относительную важность (вес) вершин графа (т. е. насколько участник в рамках конкретной сети является влиятельным), вводят понятие центральности – меры близости к центру графа. Центральность можно определить разными способами, поэтому существуют различные меры центральности. Следует отметить, что речь идет не о геометрической центральности при визуализации графа отношений. ***Центральность по степени (Degree centrality)*** определяется как количество связей, инцидентных вершине:

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

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

Другими словами, центральность по степени предполагает, что среди участников сети более влиятельным является тот, у кого больше друзей, либо тот, кто входит в большее количество сообществ. Тем не менее участник сети, имеющий большое количество друзей, может быть связан с остальным графом маленьким количеством ребер. Поэтому вводится следующее понятие.

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;

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

Центральность можно вычислить при помощи алгоритма ссылочного ранжирования (PageRank), который используется в поисковой системе Google. В основу положен принцип «важности» веб-страницы: чем больше ссылок на страницу, тем она «важнее». Кроме того, вес самой страницы определяется весом ссылки передаваемой на нее страницы. Таким образом, 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 - 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:

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

где $\alpha \in (0,1)$ – доля участия удаленных вершин, называемая *коэффициентом затухания*.

Центральность Каца можно представить как разновидность центральности по собственному вектору:

$$C_{Katz}(i) = x_i = \alpha \sum_{j=1}^{N} a_{ij} \left(x_j + 1\right).$$

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.

Как только силы на узлах и рёбрах определены, поведение всего графа под действием этих сил может быть итеративно промоделировано, как если бы это была физическая система. В такой ситуации силы, действующие на узлы, пытаются стянуть их ближе или оттолкнуть их друг от друга подальше. Это продолжается, пока система не придёт в состояние механического равновесия, то есть положение узлов не меняется от итерации к итерации. Положение узлов в этом состоянии равновесия используется для генерации рисунка графа.

Для сил, определённых из пружин, идеальная длина которых пропорциональна расстоянию в графе, мажорирование стресса даёт очень хорошее поведение (то есть монотонную сходимость) и математически элегантный путь минимизации этой разницы и, следовательно, к хорошему размещению вершин графа.

Можно также использовать механизмы, которые ищут минимум энергии более прямо, а не по физической модели. Такие механизмы, являющиеся примерами общих методов глобальной оптимизации, включают имитацию отжига и генетические алгоритмы.

Следующие свойства являются наиболее важными ***преимуществами*** силовых алгоритмов:

- Результаты хорошего качества

По меньшей мере для графов среднего размера (до 50—500 вершин), полученные результаты обычно имеют очень хорошие рисунки графов по следующим критериям: однородность длин рёбер, равномерное распределение вершин и симметрия. Последний критерий наиболее важен и трудно достижим в других типах алгоритмов.

- Гибкость

Силовые алгоритмы могут быть легко приспособлены и расширены для дополнительных эстетических критериев. Это делает алгоритмы более универсальными классами алгоритмов визуализации графов. Примерами существующих расширений являются алгоритмы для ориентированных графов, визуализация трёхмерных графов, кластерная визуализация графов, визуализация графов с ограничениями и динамическая визуализация графов.

- Интуитивность

Поскольку алгоритмы основаны на физических аналогах привычных объектов, наподобие пружин, поведение алгоритмов относительно просто предсказать и понять. Этого нет в других типах алгоритмов визуализации графов.

- Простота

Типичные силовые алгоритмы просты и могут быть реализованы в несколько строк кода. Другие классы алгоритмов визуализации, такие как алгоритмы на основе ортогональных размещений, обычно требуют куда больше работы.

- Интерактивность

Ещё одним преимуществом этого класса алгоритмов является аспект интерактивности. При рисовании промежуточных этапов графа пользователь может проследить, как меняется граф, прослеживая эволюцию от беспорядочного месива в хорошо выглядящую конфигурацию. В некоторых средствах интерактивного рисования графа пользователь может отбросить один или несколько узлов из состояния равновесия и наблюдать миграцию узлов в новое состояние равновесия. Это даёт алгоритмам преимущество для динамических и онлайновых систем визуализации графов.

- Строгая теоретическая поддержка

В то время как простые силовые алгоритмы часто появляются в литературе и на практике (поскольку они относительно просты и понятны), начинает

возрастать число более обоснованных подходов. Статистики решали подобные задачи в многомерном шкалировании (англ. multidimensional scaling, MDS) с 1930-х годов, а физики также имеют длинную историю работы со связанными задачами моделирования движения n тел, так что существуют вполне вызревшие подходы. Как пример, подход мажорирования стресса к метрическим MDS может быть применен для визуализации графа и в этом случае можно доказать монотонную сходимость. Монотонная сходимость, свойство, что алгоритм будет на каждой итерации уменьшать напряжение или цену размещения вершин, важно, поскольку это гарантирует, что размещение, в конечном счёте, достигнет локального минимума и алгоритм остановится. Глушение колебаний приводит к остановке алгоритма, но не гарантирует, что будет достигнут истинный локальный минимум.

Программные приложения для анализа социальных сетей

Для анализа социальных сетей существует множество приложений для моделирования взаимодействий и процессов в сети, для вычисления определенных параметров сети и для визуализации графа сети.

Например, приложения по визуализации сети ВКонтакте (см. http://www.yasiv.com/vk) или Facebook (http://www.touchgraph.com/facebook). В них используются различные методы и алгоритмы, которые описаны ранее в данной работе. К наиболее известным средствам автоматического анализа социальных взаимодействий относятся: NetMiner (http://www.netminer.com/index.php), NetworkX (http://networkx.lanl.gov), SNAP (http://snap.stanford.edu), UCINet (http://www.analytictech.com/ucinet), Pajek (http:// vlado.fmf.uni-lj.si/pub/networks/pajek), ORA (см. http://www.casos.cs.cmu.edu/projects/ora), Cytoscape (http://www.cytoscape.org) и др. Для подобных приложений важным требованием является возможность обрабатывать очень большое количество данных. В связи с этим процесс обработки часто распараллеливают. Существуют приложения, которые моделируют «теорию шести рукопожатий», которые выстраивают цепочку из связей (друзей) между двумя пользователями сети: для русскоязычной сети ВКонтакте (http://ienot.ru/hand), для англоязычных сетей (http://www. sixdegrees.org, http://sixdegrees.com). Эти цепочки, как правило, действительно получаются небольшой длины.

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

На примере фрагмента решения задачи анализа тональности для социальных сетей путем переноса событий Twitter в Центры событий Azure в режиме реального времени разберем алгоритмы обработки данных и процесс анализа.

Надо составить запрос Azure Stream Analytics для анализа данных, а затем сохранить результаты для последующего использования или создать панель мониторинга Power BI для получения аналитической информации в режиме реального времени.

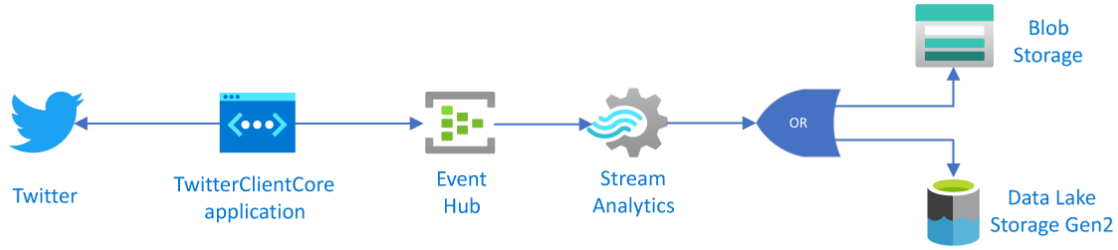
Средства аналитики для социальных сетей помогают организациям определить популярные темы, то есть темы и отношения с большим количеством записей в социальных сетях. Анализ тональности, также называемый интеллектуальным анализом мнений, использует инструменты аналитики для социальных сетей, чтобы определить отношение к продуктам или идеям.

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: Анализ тональности в социальной сети в режиме реального времени

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.



Начнем с тональности. Для решения этого класса задач в основном используется обработка естественного языка. С этой целью необходимо применять словари.

Целью анализа тональности является определение отношения говорящего, пишущего или другого субъекта к какой-либо теме, либо общей контекстуальной полярности или эмоциональной реакции на документ, действие или событие.

Полярность является основной метрикой анализа тональности. Большинство сервисов представляют ее с помощью числа из некоторого диапазона (обычно [-1,1] или [0,1]), соответствующего диапазону между негативной и позитивной тональностью. Альтернативный подход заключается в присвоении тексту меток возможных тональностей с оценкой уверенности в каждой метке. Различные сервисы предоставляют метки различной степени детальности: тональность может быть просто «позитивной» или «негативной», или же принимать промежуточные значения, например, «слегка негативная».

Особую важность для присвоения значения тональности, означающей мнение, имеют используемые в предложении прилагательные. Отрицатели и усилители в данном контексте не менее важны.

Одним из важных различий между сервисами оценки тональности является способность определять смешанную полярность, что в случае подхода с присвоением тексту меток возможных тональностей с оценкой уверенности в них использует отдельную метку (например, «mixed»), но обычно не может быть выражено с помощью подхода с числами.

Аспектный анализ тональности (aspect-based sentiment analysis) — это подвид анализа тональности, чья задача заключается в определении отношения к конкретному аспекту основного предмета обсуждения. Все подходы к анализу тональности можно разделить на три группы.

Первая — ***подходы на основе правил (rule-based).*** Чаще всего в них используются вручную заданные правила классификации и эмоционально размеченные словари. Эти правила обычно на основе эмоциональности ключевых слов и их совместного использования с другими ключевыми словами рассчитывают класс текста. Несмотря на высокую эффективность в текстах из какой-то определенной тематики, методы на основе правил плохо обобщают. Кроме того, они крайне трудоёмки в создании, особенно когда нет доступа к подходящему словарю настроений. Последнее особенно характерно для русского языка, потому что на нем не так много источников, как на английском, особенно в сфере анализа тональности. Крупнейшие русскоязычные словари настроений — RuSentiment и LINIS Crowd. Но в них есть только информация о тональности от позитивной до негативной, без характеристик эмоций. Таким образом, не существует альтернатив таким мощным англоязычным подборкам с обширными эмоциональными характеристиками, как SenticNet, SentiWordNet и SentiWords.

Вторая группа — подходы на основе машинного обучения. Они используют автоматическое извлечение признаков из текста и применение алгоритмов машинного обучения. Классическими алгоритмами классификации полярности являются ***наивный байесовский классификатор, дерево решений, логистическая регрессия и метод опорных векторов***. В последние годы внимание исследователей привлекают методы глубокого обучения, которые значительно превосходят традиционные методы в анализе тональности.

В простых подходах для представления текста в векторном пространстве обычно используется модель «мешок слов» (bag of words). В более сложных системах для генерирования эмбеддингов слов применяются модели дистрибутивной семантики, например, Word2Vec, GloVe или FastText. Также есть алгоритмы генерирования эмбеддингов на уровне предложений или параграфов, которые предназначены для переноса обучения в разных задачах обработки естественного языка. К таким алгоритмам относятся ELMo, Universal Sentence Encoder, BERT, ERNIE и XLNet. Одним из их главных недостатков с точки зрения генерирования эмбеддингов является потребность в больших массивах текстов для обучения. Это справедливо для всех методов машинного

обучения, потому что всем алгоритмам обучения с учителем нужны для обучения размеченные наборы данных.

Третья группа — гибридные подходы. Они объединяют в себе подходы двух предыдущих видов. Например, гибридный фреймворк для анализа тональности персидского языка, в котором сочетаются лингвистические правила, а также модули свёрточных нейросетей и LSTM для классификации настроений. В гибридной модели аспектного анализа ALDONAr сочетаются онтология настроений для захвата информации о настроениях, BERT для получения эмбеддингов слов и два слоя CNN для расширенной классификации тональности.

Языковые модели часто применяются в гибридных алгоритмах, как и решения на основе правил. С одной стороны, комбинация методов на основе правил и машинного обучения обычно позволяет добиться более точных результатов. А с другой — гибридные подходы наследуют трудности и ограничения составляющих их алгоритмов.

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

Устройство хранения информации на компьютере изучалось еще на школьных курсах информатике, затем, более углубленно в университете. В контексте задач машинного обучения необходимо рассмотреть отдельные моменты, связанные с данными.

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.

Когда компьютер запрашивает данные, хранящиеся на диске, пластины начинают вращаться, а головки – двигаться, пока не будет найдена область с запрашиваемой информацией. Головка пассивно "парит" над поверхностью диска, и, когда микроскопические магниты, образующие магнитные домены, проходят под ней, они влияют на магнитное поле головки. Электроника дисковода многократно усиливает эти слабые возмущения, превращая их в последовательности нулей и единиц, которые затем поступают в микросхемы памяти компьютера.

Магнитные жесткие диски

Довольно долгое время магнитные жесткие диски были основными устройствами хранения данных на компьютерах и ноутбуках. С годами появились более совершенные жесткие диски, которые намного быстрее работают и более надежны в эксплуатации, но принцип их работы остается неизменным до сих пор. В основе их работы лежат алюминиевые диски с магнитным покрытием, над поверхностью которых находятся магнитные головки, осуществляющие запись и чтение информации. Магнитные диски постоянно вращаются с большой скоростью, а магнитные головки могут спокойно перемещаться вдоль поверхности диска.

Эта технология довольно надежна и относительна недорогая по сравнению с другими вариантами хранения данных. На рынке есть огромный выбор различных жестких дисков от многих производителей, они могут хранить огромные массивы данных и не требуют специального программного обеспечения при подключении к компьютеру. Магнитные жесткие диски значительно уступают по производительности твердотельным накопителям или даже гибридным жестким дискам. На сегодняшний день самые быстрые магнитные жесткие диски со временем доступа к информации 8 мс могут

записать или прочитать не более 200 МБ данных за одну секунду. Скорость работы таких жестких дисков определяется скоростью вращения дисков, соответственно магнитные жесткие диски со скоростью вращения 7200 об/мин будут значительно быстрее работать, чем жесткие диски со скоростью вращения 5400 об/мин. В основном такие диски подходят для пользователей, которые хранят на компьютере огромные объемы информации и не нуждаются в особой скорости, просматривая интернет и редактируя текстовые файлы.

Твердотельные жесткие диски

Не так много производителей предлагают твердотельные жесткие диски SSD, поэтому выбор тут намного меньше. Выполняют они точно такие же функции, как и магнитные жесткие диски, только в основе их работы заложен совершенно другой принцип. По сути, твердотельные жесткие диски представляют собой большие флешки с энергонезависимой памятью. Подключаются они к компьютеру таким же способом, как и обычные жесткие диски, только за счет отсутствия подвижных частей потребляют гораздо меньше электроэнергии и значительно быстрее работают. 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. Есть диски и большего номинала, но они сильно дорогие для использования в домашнем компьютере. Другим недостатком твердотельных накопителей является их мгновенный выход из строя. Если магнитные жесткие диски всячески показывают приближение поломки появлением разных ошибок и битых секторов, то SSD накопители просто отключаются. Твердотельные жесткие диски подходят для опытных пользователей, которым требуется высокая эффективность работы всей системы. Наилучшим вариантом считается одновременное использование SSD накопителей и стандартных жестких дисков. Твердотельный накопитель может применяться для хранения файлов операционной системы и наиболее часто используемых приложений, а основной объем хранения данных может приходиться на обычный магнитный жесткий диск.

Гибридные жесткие диски

Гибридные жесткие диски сочетают в себе достоинства обоих типов жестких дисков. Они содержат в одном корпусе магнитные диски и твердотельные элементы, за счет чего может достигаться значительные прирост в скорости работы при незначительном увеличении стоимости. В большинстве случаев пользователь не может выбирать, в какой тип памяти производить запись данных, так как операционная система видит только общий доступный объем жесткого диска. Во время работы гибридного жесткого диска производится анализ наиболее часто используемых файлов, которые впоследствии переносятся во флеш память жесткого диска автоматически.

Облачные хранилища — модель онлайн-хранилища, в котором данные хранятся на многочисленных распределённых в сети серверах, предоставляемых в пользование клиентам, в основном, третьей стороной. В отличие от модели хранения данных на собственных выделенных серверах, приобретаемых или арендуемых специально для подобных целей, количество или какая-либо внутренняя структура серверов клиенту, в общем случае, не видна. Данные хранятся и обрабатываются в так называемом «облаке», которое представляет собой, с точки зрения клиента, один большой виртуальный сервер. Физически же такие серверы могут располагаться удалённо друг от друга географически.

Облачными хранилищами являются такие интернет-сервисы, как: Dropbox, OneDrive, Google Drive, iCloud, Яндекс.Диск, Облако Mail.Ru, МегаДиск, Mega, BOX, TeraBox, pCloud, Files.fm и др.

Актуальными стали такие хранилища в эпоху Больших данных.

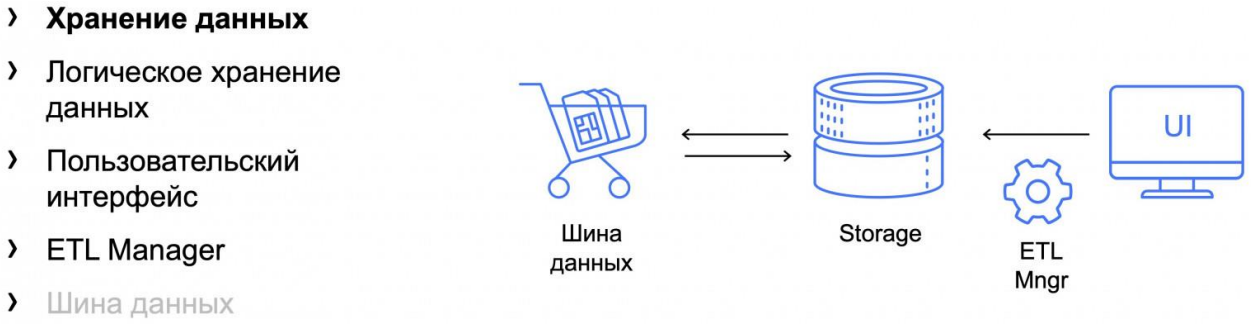
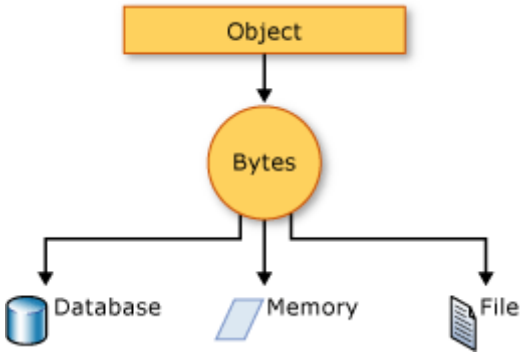


Схема аналитического хранилища данных

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



Объект сериализуется в поток, который служит для передачи данных. Поток также может содержать сведения о типе объекта, в том числе о его версии, языке и региональных параметрах, а также имени сборки. В этом формате потока объект можно сохранить в базе данных, файле или памяти.

Сериализация используется для передачи объектов по сети и для сохранения их в файлы. Например, нужно создать распределённое приложение, разные части которого должны обмениваться данными со сложной структурой.

В таком случае для типов данных, которые предполагается передавать, пишется код, который осуществляет ***сериализацию и десериализацию***. Объект заполняется нужными данными, затем вызывается код сериализации, в результате получается, например, XML-документ. Результат сериализации передаётся принимающей стороне по, скажем, электронной почте или HTTP. Приложение-получатель создаёт объект того же типа и вызывает код десериализации, в результате получая объект с теми же данными, что были в объекте приложения-отправителя. По такой схеме работает, например, сериализация объектов через SOAP(простой протокол доступа к объектам) в Microsoft .NET.

JSON — один из популярных форматов для сериализации, он текстовый, легковесный и легко читается человеком.

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" }
```

Саму сериализацию (и десериализацию) можно производить вручную, или пользоваться соответствующими библиотеками/фреймворками.

Нотация JSON.

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

Несмотря на происхождение от JavaScript (точнее, от подмножества языка стандарта ECMA-262 1999 года), формат считается независимым от языка и может использоваться практически с любым языком программирования. Для многих языков существует готовый код для создания и обработки данных в формате JSON.

JSON (нотация объектов JavaScript) – это облегченный открытый формат обмена данными, который использует читаемый человеком текст для передачи объектов данных, состоящих из пар атрибут – значение. Это подмножество JavaScript. Он использует соглашения, знакомые программистам семейства языков C, таких как C, C ++, C #, Java, JavaScript, Perl, Python.

Тип файла для файлов JSON – «.json», а тип MIME – «application/json».

Зачем использовать JSON поверх XML?

JSON легче, так как требует меньше тегов, чем XML. **Он более компактен, чем XML,** и поэтому быстро загружается. **Читать и писать** JSON быстрее, чем XML. **Объекты JSON являются типизированными** (строка, число, массив, логическое значение, объект), тогда как данные XML не содержат типов (только строка).

Данные легко доступны в виде объектов JSON для кода JavaScript, тогда как данные XML необходимо анализировать и присваивать переменным через утомительные API DOM.

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. Как правило, регистр учитывается программами — имена с буквами в разных регистрах считаются разными, например[5]), значением — любая форма. Повторяющиеся имена ключей допустимы, но не рекомендуются стандартом; обработка таких ситуаций происходит на усмотрение программного обеспечения, возможные варианты — учитывать только первый такой ключ, учитывать только последний такой ключ, генерировать ошибку.
- Упорядоченный набор значений. Во многих языках это реализовано как массив, вектор, список или последовательность.

Структуры данных, используемые JSON, поддерживаются любым современным языком программирования, что и позволяет применять JSON для обмена данными между различными языками программирования и программными системами.

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.

Строка очень похожа на литерал одноимённого типа данных в языке Javascript. Число тоже очень похоже на Javascript-число, за исключением того, что используется только десятичный формат (с точкой в качестве разделителя). Пробелы могут быть вставлены между любыми двумя синтаксическими элементами.

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"
    ]
}
```

JSON Schema — один из языков описания структуры JSON-документа. Использует синтаксис JSON. Базируется на концепциях XML Schema, RelaxNG, Kwalify. JSON Schema — самоописательный язык: при его использовании для обработки данных и описания их допустимости могут использоваться одни и те же инструменты сериализации/десериализации.

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.

Альтернативные варианты языка XML: XAML, JSON, XF.

Структура XML

Язык программирования XML может использоваться для группирования любых данных, чтобы создать иерархию или разметку. Структура XML представлена простым самописным синтаксисом:

```
<?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>
```

Теперь разберем этот пример детальнее.

Первая строка документа — это XML декларация. Здесь определяется версия XML (version="1.0") и тип кодировки документа (encoding="UTF-8")

Далее описывается корневой элемент документа. Корневой элемент в документе может быть только один, и он будет содержать все ваши данные. Поскольку мы можем сами давать названия тегам, мы использовали <marvel>...</marvel>. Наш документ будет содержать список героев <hero>...</hero>.

Обратите внимание, что в документе можно писать комментарии, но это не обязательное требование. Пример комментария:

<!-- ЗДЕСЬ ВАШ КОММЕНТАРИЙ К КОДУ -->

Далее внутри тега <hero>...</hero> мы описываем каждого героя. У нас есть тег <nickname>...</nickname> в который мы записали псевдоним героя, <realname>...</realname> — в который мы записали реальное имя героя, <abilities>...</abilities> — в который мы записали суперспособности героя.

Также у тега hero присутствует атрибут (id="positive_character"). Атрибуты предоставляют дополнительную информацию об элементе. Эта информация может быть важна для приложений, которые будут манипулировать этим элементом. Значение атрибута всегда должно заключаться в кавычки. Название атрибута вы можете придумать сами, поскольку язык — расширяемый.

Синтаксис XML выглядит просто, но не приемлет ошибок. Например, если вы пропишите значение атрибута без кавычек, это вызовет синтаксическую ошибку, поэтому обязательно валидируйте свой файл. Проверка XML на валидность может быть выполнена с помощью Яндекс.Вебмастер или другого онлайн-сервиса, например, XML Validator.

XML и HTML не заменяют друг друга. Можно преобразовать код из одного формата в другой. Вывод XML в HTML выполняется с помощью онлайн-конвертеров. XML предназначен для хранения и отправки данных, а HTML служит для их отображения на веб-странице.

К тому же XML отличается расширенной разметкой и может быть дополнен самописными тегами. Этот язык используется во всех сферах программирования и очень популярен как метод преобразования объемной информации в форму иерархии для ее удобного хранения.

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.

Пример

Используйте 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").innerHTML = 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").innerHTML = obj.name + ", " + obj.age();
```

Парсинг XML

Все современные браузеры имеют встроенный XML парсер.

Этот 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();
```

В строке **var xhttp = new XMLHttpRequest();** создается объект XMLHttpRequest.

В строке **xhttp.onreadystatechange = function()** свойство **onreadystatechange** определяет функцию, которая будет выполняться каждый раз, когда статус объекта XMLHttpRequest изменится.

Строка **if (this.readyState == 4 && this.status == 200)**. Когда свойство **readyState** равно 4, и свойство **status** равно 200, ответ готов.

Свойство **responseText** возвращает ответ сервера в виде текстовой строки.

Эта текстовая строка может использоваться для изменения кода веб-страницы. Строка **document.getElementById("demo").innerHTML = xhttp.responseText;**.

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);
}
```