C Programming

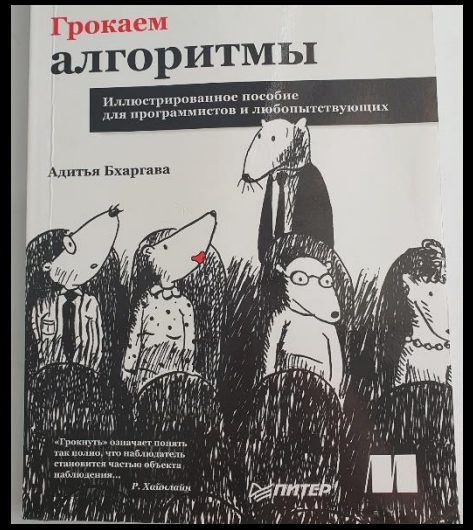
1_algorithms

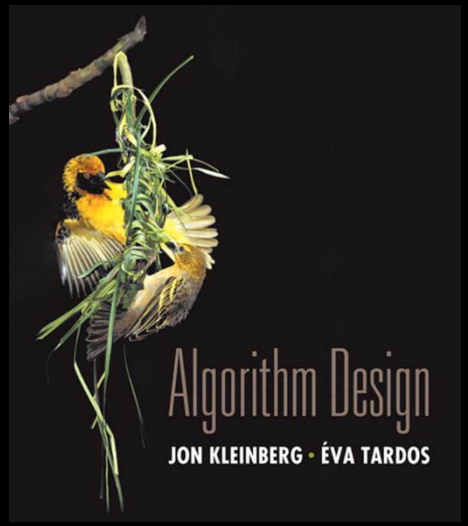
Алгоритм

Алгоритм — есть последовательность действий (или инструкций) для выполнения задачи и получения некоторого результата.

Алгоритмы можно представить как черную коробку, которой мы отдаем что-то на вход и получаем что-то ожидаемое на выходе, например какой-нибудь алгоритм сортировки, на вход набор чисел, а на выходе отсортированный набор чисел. Здесь черная коробка и есть алгоритм.

Зная алгоритм его легко можно перенести в программу написанную на любом языке, тут понадобится лишь немного погуглить синтаксис языка, а алгоритм от языка к языку остается прежним



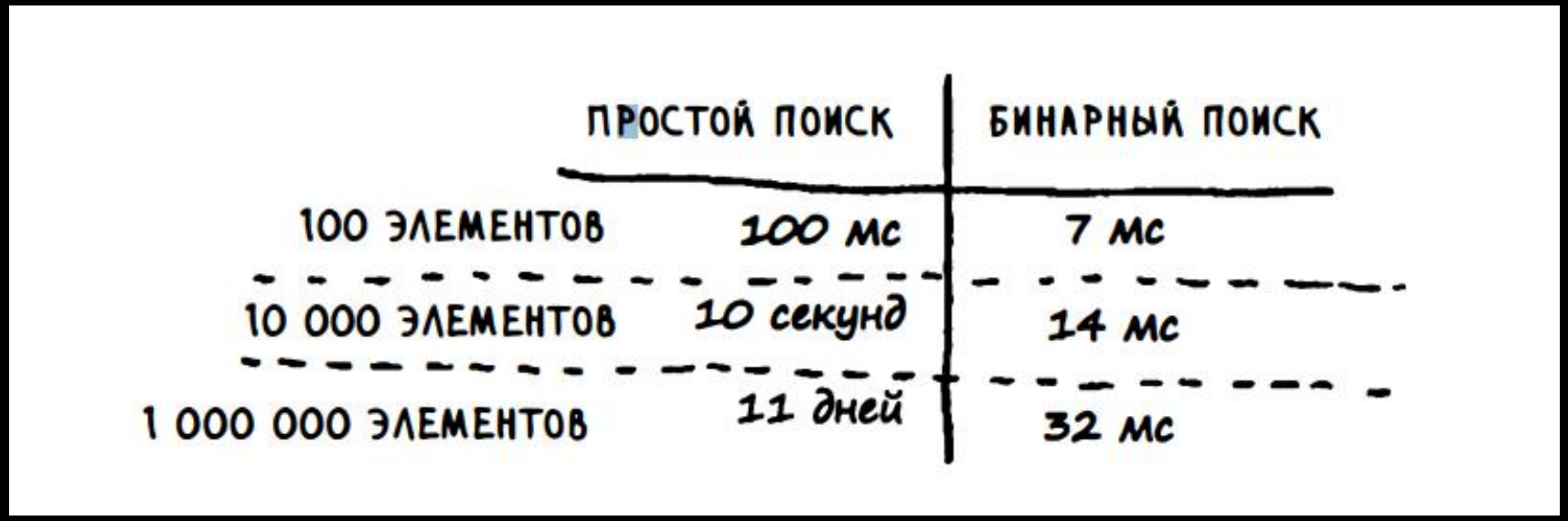


Зачем алгоритмы?

- Хорошо подобранные алгоритмы для задачи позволяют выполнять эту задачу быстро

Пример: нам нужно срочно в ближайшие пару дней найти в массиве размером в 1 млрд какой-то один особенный элемент. У нас есть два варианта поиска - линейный поиск и бинарный

Зачем алгоритмы?



Зачем алгоритмы?

- Алгоритмы повторяемы и выдают предсказуемые результаты каждый раз, когда они выполняются.
- Алгоритмы можно стандартизировать и совместно использовать скажем в разных устройствах.
- Алгоритм не зависит от используемого языка. Он сообщает программисту логику, используемую для решения проблемы. Таким образом, алгоритм - это логическая пошаговая процедура, которая служит программистам своего рода планом.

Типы алгоритмов

- Алгоритмы сортировки - алгоритмы для упорядочивания каких-либо данных в определенном порядке. В интернет-магазине включаем фильтр по количеству отзывов. Пузырьковая сортировка, сортировка вставками, быстрая сортировка, сортировка подсчетом и много-много других..
- Алгоритмы поиска - алгоритмы поиска значения в наборе данных. Линейный поиск, бинарный и т.д.
- Алгоритмы с графами - поиск кратчайших путей. Алгоритм Дейкстра, BFS, DFS и т.д. Пример в 2gis ищем как пройти от какого-нибудь отеля до ТЦ. Социальные сети работают на алгоритмах с графами

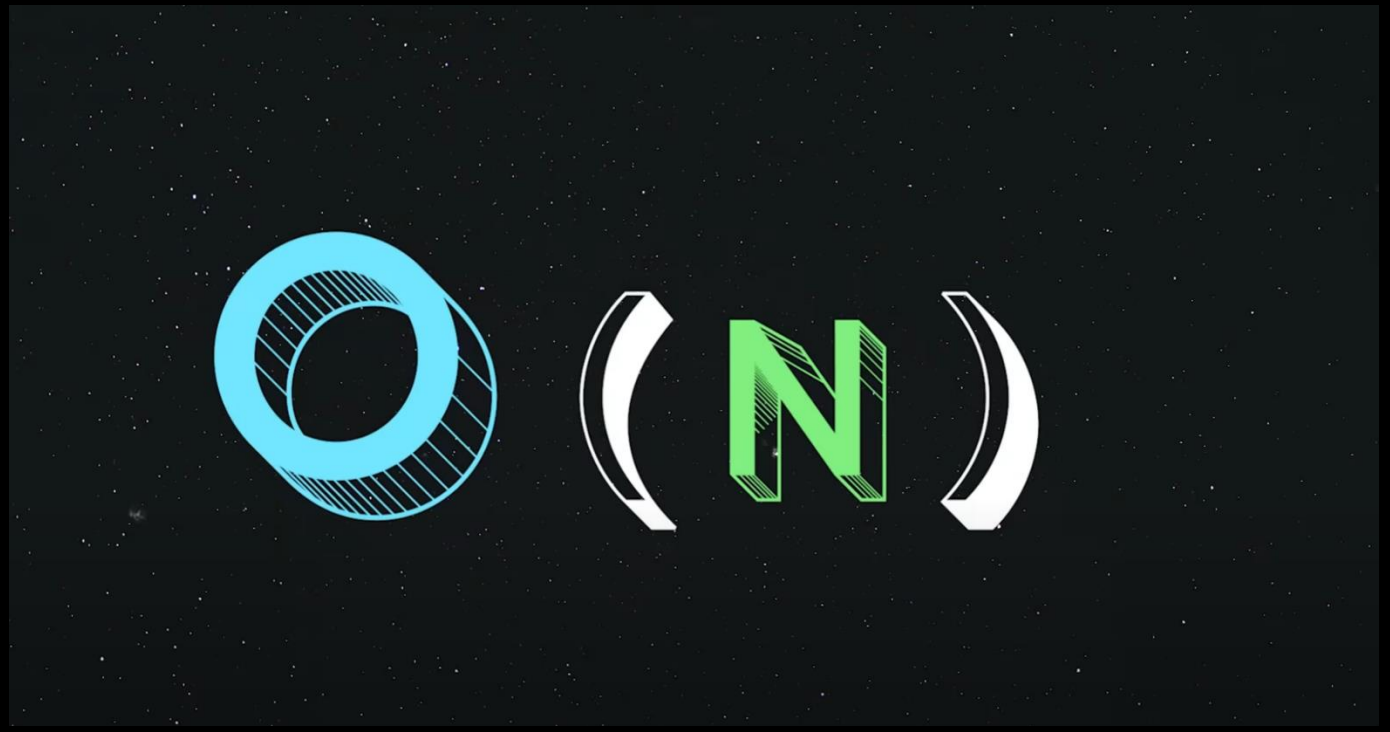
Сложность алгоритмов

Сложность в алгоритмах относится к количеству ресурсов, необходимых для решения проблемы или выполнения задачи. К таким ресурсам относят время и память. (Time and Space complexity)

Память говорит сколько оперативной памяти занимает программа, переменные и т.д.
Время – время выполнения алгоритма

Big O notation

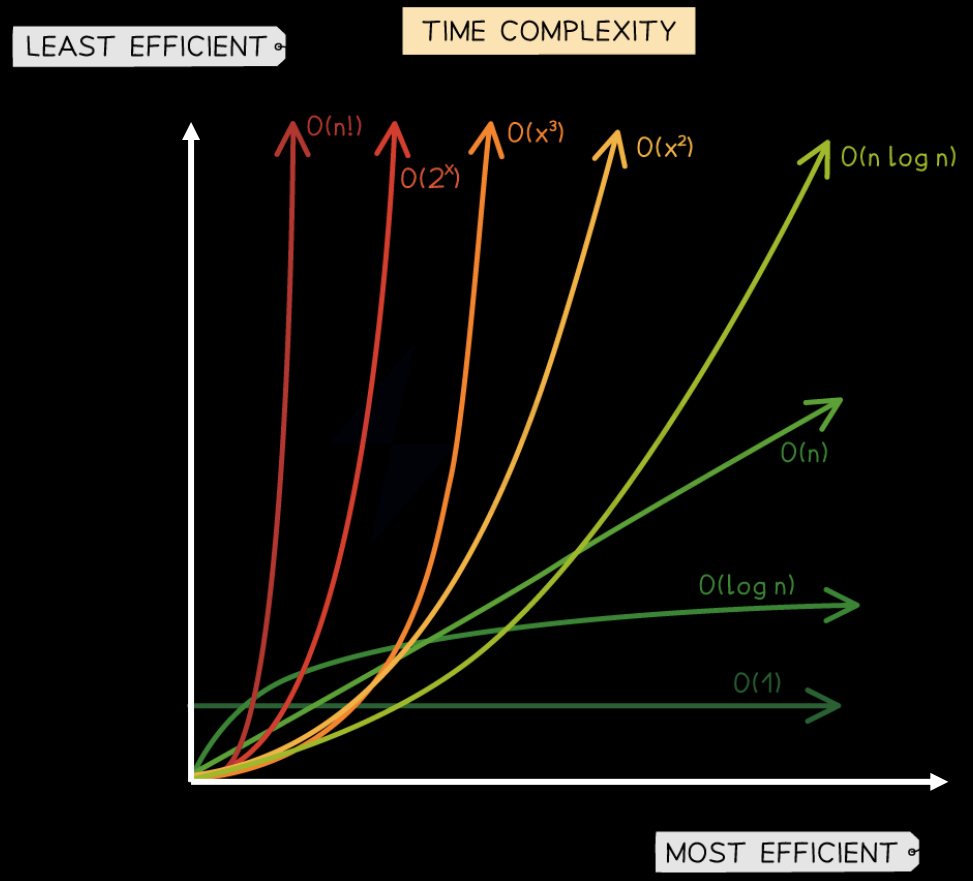
Простыми словами - Big O нотация используется для оценки наихудшего случая временной сложности алгоритма, т.е. "O" покажет как поменяется производительность алгоритма. Записывается обычно следующим образом: O(n) - читается как "O от n", где n - размер входных данных.



Виды сложности

Сложность:

- O(1) - Константная ~
- O(log n) - Логарифмическая ~
- O(n) - Линейная ~
- O(n log n) - Линейно-логарифмическая ~
- O(n^2) - Квадратичная ~
- O(n^3) - Кубическая ~
- O(2^n) - Экспоненциальная ~
- O(n!) - Факториальная ~



O(1) Константная сложность

Такая сложность можно сказать является идеальной, т.к. производительность алгоритмов с такой сложностью не зависит от размера входных данных. Популярный пример такой сложности - получение элемента массива.

```
int get_elem(int idx) {
    return some_int_array[idx];
}
```

O(n) Линейная сложность

```
for (int s = 0; s < n; s++) {
  if (recipes[s] == 'M'){
    // get recipe
  }
}
```

O(log n) Логарифмическая сложность

1. Поделим алфавит на 2 части А-Я: А-О и П-Я -> буква М в первой половине алфавита, соответственно вторая половина нам не нужна
2. Делим первую половину еще на две половины А-О: А-Ж и З-О -> буква М во второй половине, первую отбрасываем А-Ж
3. З-О -> З-К и Л-О -> Л-О
4. Л-О -> Л-М и Н-О -> Л-М
5. Л и М -> М ответ

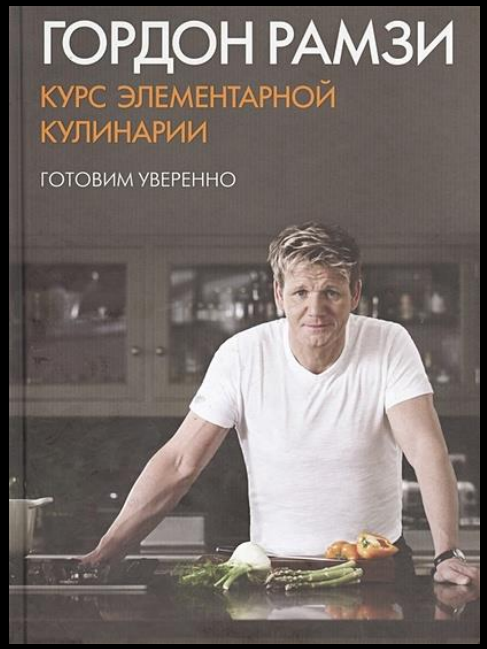
5 операций чтобы найти ответ, где n = 33, а log(33) = 5

А	Б	В	Г	Д	Е	Ё	Ж	З	И	Й
1	2	3	4	5	6	7	8	9	10	11
К	Л	М	Н	О	П	Р	С	Т	У	Ф
12	13	14	15	16	17	18	19	20	21	22
Х	Ц	Ч	Ш	Щ	Ь	Ы	Ъ	Э	Ю	Я
23	24	25	26	27	28	29	30	31	32	33

O(n log n) - Линейно-логарифмическая сложность

Она схожа с O (log n), но добавляется еще одна n - например мы ищем не в одной книге, а среди n книг рецептов Гордона Рамзи рецепт на какую-то букву.

Итого: есть цикл O(n), внутри которого используются алгоритмы "разделяй и властвуй" O(log n) -> O(n * log n)



O(n^2) Квадратичная сложность

Сюда относятся все вложенные циклы, т.е. цикл в цикле. Самый простой пример - пузырьковая сортировка.

```
for (int i = 0; i < n; i++) {
  for (int j = 0; j < n - 1; j++) {
    if (a[j] > a[j + 1]) {
      swap(&a[j], &a[j + 1]);
    }
  }
}
```

O(n^3) Кубическая сложность

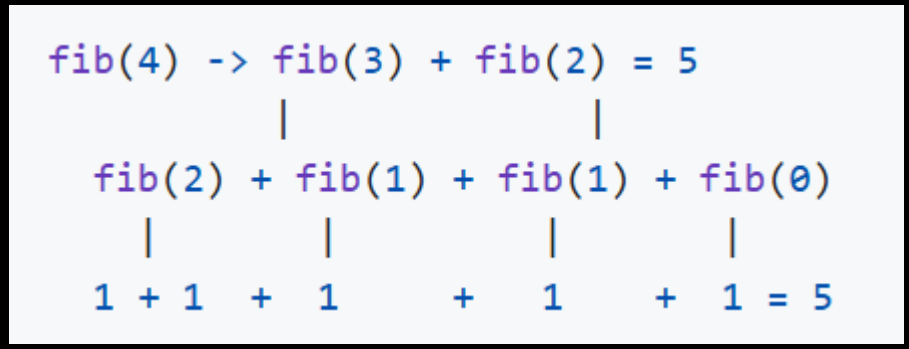
Если O(n) - один проход по набору данных (один цикл), O(n^2) - n*n проходом по данным (два цикла - цикл в цикле), то O(n^3) три цикла, пример:

```
for (int i = 0; i < n; i++) {
  for (int j = 0; j < n; j++) {
    for (int k = 0; k < n; k++) {
      work()
    }
  }
}
```

O(2^n) Экспоненциальная сложность

Такую сложность можно встретить в рекурсиях, например при расчете чисел Фибоначчи. В данном коде используется рекурсия - функция, вызывающая саму себя. Однако каждый раз, когда вызывается функция fib(), она порождает за собой два дополнительных вызова, что приводит к экспоненциальному увеличению количества вызовов функций с увеличением n.

```
int fib(int n) {
  if (n <= 1) {
    return n;
  } else {
    return fib(n - 1) + fib(n - 2);
  }
}
```



O(n!) Факториальная сложность

Здесь время выполнения алгоритма растет пропорционально факториалу размера входных данных. Самый простой пример такой трудозатратной фукции:

```
void f(int n) {
  for(int i = 0; i < n; i++) {
    f(n - 1);
  }
}
```

Вызов функции f() от n единожды порождает n функций, вызывающих n-1 функций, вызывающих n-2 функций..

Нюансы Big O

При этом при O - нотации принято не указывать константы и упрощать по возможности: т.е. O(n) == O(2*n), так же все младшие степени "поглощаются" старшими O(n^2 + n) == O(n^2). Исключение может быть только при смешивании полиномиальной и экспоненциальной части (для малых n полином может быть больше).

O(n^2) + O(log n) = O(n^2)
O(3n) = O(n)
O(10000 * n^2) = O(n^2)
O(2n * log n) = O(n * log n)
O(n^2 + n) = O(n^2)
O(n^3 + 100n * log n) = O(n^3)
O(n! + 999) = O(n!)

Пытаемся оценить сложность алгоритма

```
int sumOfPairs(int arr[], int length) {
    int sum = 0;
    for (int i = 0; i < length; i++) {
        for (int j = 0; j < length; j++) {
            sum += arr[i] + arr[j];
        }
    }
    return sum;
}
```

```
int findMax(int arr[], int length) {
    int max = arr[0];
    for (int i = 1; i < length; i++) {
        if (arr[i] > max) {
            max = arr[i];
        }
    }
    return max;
}
```

Пытаемся оценить сложность алгоритма

```
int test(int arr[], int length) {
  int sum = 0;
  for (int i = 0; i < length; i++) {
    for (int j = 0; j < length; j++) {
        sum += arr[i] + arr[j];
    }
  }
  if (length < 3) return sum;
  int max = arr[0];
  for (int i = 1; i < 3; i++) {
      if (arr[i] > max) {
          max = arr[i];
      }
  }
  return sum + max;
}
```

```
const int keys[] = {0x0,
0xDEADBEEF, 0xC0FFEE, 0xFFFF};

int get_key(int n) {
  return keys[n];
}
```

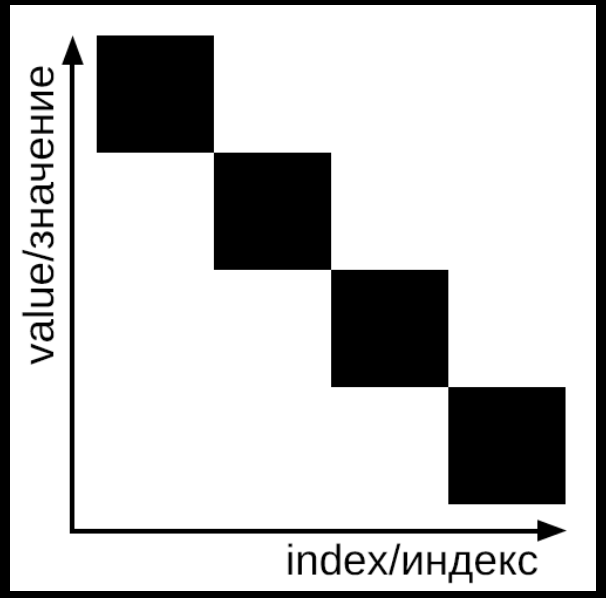
Сортировки

MergeSort, QuickSort, Counting, Selection, Shell, Bubble, Insertion и прочие алгоритмы сортировок

Залазим на wiki с сортировками

Непрактичная сортировка

Есть такая сортировка, которая в может отсортировать массив любого размера за всего один проход по массиву, но если немного повезет.



Bogo Sort

Ее суть заключается в том, что проходя по элементам массива мы элементы массива кладем в случайные места массива. И с очень-очень маленьким шансом (но он все же есть) мы можем упорядочить элементы в правильном порядке. Но вообще ее сложность оценивается что-то типа O(n*n!)

При работе 4-ядерного процессора на частоте 2,4 ГГц (9,6 млрд операций в секунду):

Кол-во элементов	Среднее время
10	0,0037 с
11	0,045 с
12	0,59 с
13	8,4 с
14	2,1 мин
15	33,6 мин
16	9,7 ч
17	7,29 сут
18	139 сут
19	7,6 лет
20	160 лет

Таким образом, колода в 32 карты будет сортироваться этим компьютером в среднем $2{,}7 \cdot 10^{19}$ лет.

Списки

Список — это структура данных, которая хранит элементы в линейном порядке и позволяет вставлять и удалять элементы в любом месте последовательности. Списки могут быть односвязными, двусвязными или круговыми (циклическими).

Таблица временной сложности

Параметр	Массив (статический)	Массив (динамический)	Односвязный список	Двусвязный список
Доступ	O(1)	O(1)	O(n)	O(n)
Вставка/ Удаление в начало	--- (размер фиксированный)	O(n) (нужно двигать элементы)	O(1)	O(1)
Вставка/ удаление в конец	--- (размер фиксированный)	O(1) (если есть место в массиве), O(n) (если места нет - копирование); удаление O(1)	O(n)	O(1)
Вставка/ удаление в середину	--- (размер фиксированный)	O(n) (нужно двигать элементы)	O(1) (но добираться до середины O(n))	O(1) (но добираться до середины O(n))
Поиск	O(n)	O(n)	O(n)	O(n)

Связанные списки (Linked List)

Связанный список представляет собой последовательность узлов, где каждый узел состоит из двух компонентов:

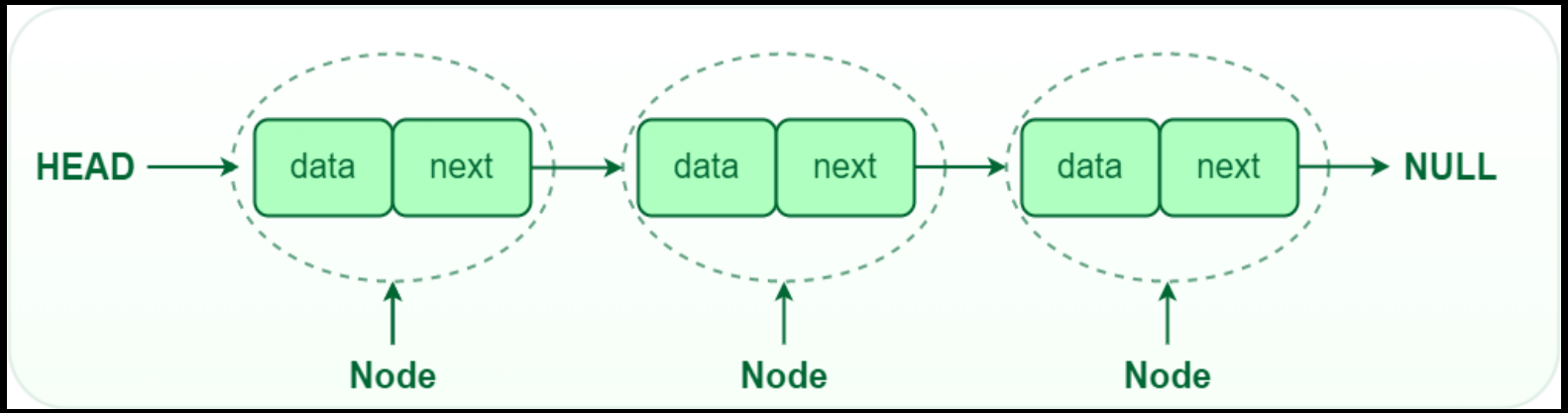
1. Данные: любые значения, например, число, структуры и т.д.
2. Указатель: ссылка на следующий узел в списке.

В отличие от массивов, связанные списки не хранят элементы в смежных ячейках памяти. Каждый узел указывает на следующий, образуя цепочечную структуру. Чтобы получить доступ к любому элементу (узлу), необходимо последовательно пройтись по всем узлам перед ним.

Односвязные списки

Односвязный список состоит из узлов. В каждом узле хранится указатель на следующий узел и сами данные. Указатели последнего узла (хвост - tail) указывают на NULL, что указывает на конец связанного списка.

```
struct node_s
{
  int data;
  struct node_s *next;
};
```



Создадим свой односвязный список

Здесь начинаем live code (программировать в живую):
Создадим односвязный список и напишем функции вставка/удаление, поиск, длина списка и вывод списка

Github

https://github.com/kruffka/C-Programming