Г.И. Атаева, У.Н. Хамроев, Ш.Ж. Сейтов, Д.Н. Хамроева

АЛГОРИТМЫ И СТРУКТУРЫ ДАННЫХ

УЧЕБНИК

Издательство «КАМОЛОТ»
Бухара-2024

УДК: 004.421(075.8)
ББК: 32.97я73
А92

Г.И. Атаева, У.Н. Хамроев, Ш.Ж. Сейтов, Д.Н. Хамроева, Алгоритмы и структуры данных [текст]: учебник - БУХАРА:. ООО “БУХОРО ДЕТЕРМИНАНТИ” издательство КАМОЛОТ, 2024 г. 184 ст

Данный учебник предназначен для студентов технических специальностей, изучающих программирование, а также для преподавателей дисциплины Алгоритмы и структуры данных. Учебник разработан по типовой программе дисциплины Алгоритмы и структуры данных для студентов, занимающихся на факультете информационных технологий Бухарского государственного университета.

Учебник содержит теоретический материал по теме лекционных занятий, также в учебнике содежатся материалы практических занятий для преподавателей, которые позволяют организовать практические занятия. Материалы практических занятий содержат примеры программных кодов, направленных на развитие знаний студентов по программированию.

Рецензенты:
Н.С. Сайидова – БухГУ, кандидат физико-математических наук, доцент кафедры информационных систем и цифровых технологий.
Ф.Р. Мурадова – Бухарский инженерно-технологический институт, профессор кафедры информационно-коммуникационных технологий

ISBN: 978-9910-761-75-1

Рекомендовано к печати Министерством высшего образования, науки и инноваций Республики Узбекистан в качестве учебник для студентов и преподавателей профильных вузов (приказ № 438 от 29 сентябрь 2024 г.)

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Алгоритмы и структуры данных — фундаментальные концепции, лежащие в основе эффективного программирования и разработки программного обеспечения. Алгоритмы представляют собой последовательности действий, направленных на решение конкретных задач, тогда как структуры данных обеспечивают способ организации и хранения информации, что позволяет эффективно работать с данными в процессе выполнения алгоритмов.

С развитием технологий и увеличением объемов обрабатываемой информации, потребность в быстрых и эффективных решениях возрастает. Знание алгоритмов и понимание, как выбрать подходящую структуру данных для конкретной задачи, становится критически важным для программистов. В учебнике рассматриваются как базовые алгоритмы, такие как сортировка и поиск, так и более сложные, применяемые в реальных проектах.

Учебник ориентирован на студентов и начинающих программистов, желающих углубить свои знания в области алгоритмов и структур данных. Изучение этого материала поможет не только понять основы программирования, но и улучшить навыки анализа и оптимизации кода.

В каждой теме подробно рассматриваются конкретные алгоритмы, а также различные типы структур данных, начиная с простейших, таких как массивы и списки, и заканчивая более сложными — деревьями и графами. Включены многочисленные примеры и упражнения, которые помогут закрепить теоретические знания на практике.

Надеемся, что данный учебник станет надежным проводником в мир алгоритмов и структур данных и поможет освоить ключевые концепции, которые станут основой для дальнейшего профессионального роста в области разработки программного обеспечения.

ТЕМА 1. ВВЕДЕНИЕ В АЛГОРИТМЫ И СТРУКТУРЫ ДАННЫХ

Алгоритмы и структуры данных — основополагающие темы в области информатики и программирования. Они играют ключевую роль в разработке эффективных программных решений, а также в понимании работы различных компьютерных систем. Данный раздел поможет читателю познакомиться с базовыми понятиями и принципами, лежащими в основе этих тем.

Алгоритм — это упорядоченный набор инструкций или шагов, которые приводят к решению задачи или достижению определенной цели. Важно, чтобы алгоритм был четко определенным, понятным и выполнимым за конечное количество шагов.

Свойства алгоритмов:

- **Определенность** — каждый шаг алгоритма должен быть четко описан.
- **Конечность** — алгоритм должен завершаться за конечное количество шагов.
- **Массовость** — алгоритм применим для широкого класса задач.
- **Результативность** — выполнение алгоритма должно приводить к результату.

Пример простого алгоритма: нахождение наибольшего числа среди двух.

- Сравнить два числа, A и B.
- Если A больше B, вывести A как результат.
- Если B больше или равно A, вывести B как результат.

Типы алгоритмов

- **Линейные алгоритмы** — последовательное выполнение действий. Пример: вычисление суммы элементов массива.
- **Разветвляющиеся алгоритмы** — выполнение различных действий в зависимости от условий. Пример: выбор действий в зависимости от значения переменной.
- **Циклические алгоритмы** — повторяющиеся действия до тех пор, пока выполняется определенное условие. Пример: суммирование элементов массива до тех пор, пока не достигнута определенная граница.

Структура данных — это способ организации и хранения данных, который позволяет эффективно использовать их для решения задач. Основные структуры данных включают массивы, списки, очереди, стеки, деревья и графы.

Важность структур данных:

- **Эффективность** — правильный выбор структуры данных позволяет ускорить выполнение программ.
- **Упрощение** — структуры данных помогают упростить работу с большими объемами данных.
- **Масштабируемость** — структуры данных позволяют строить программы, которые легко масштабировать на большие задачи.

Основные структуры данных

- **Массивы** — это хранилище элементов одного типа, записанных в памяти последовательно. Каждый элемент массива имеет свой индекс, что позволяет быстро получать доступ к данным. Недостатком является фиксированный размер массива.
- **Связные списки** — это динамическая структура данных, где каждый элемент (узел) содержит ссылку на следующий элемент. Преимущество списка в том, что его можно легко изменять (добавлять и удалять элементы). Однако для доступа к элементам необходимо проходить по всему списку.
- **Стек** — структура данных, работающая по принципу LIFO (Last In, First Out — последним пришел, первым ушел). Элементы добавляются и удаляются только с одного конца стека. Примером использования является отмена операций в текстовых редакторах.
- **Очередь** — структура данных, работающая по принципу FIFO (First In, First Out — первым пришел, первым ушел). Элементы добавляются с одного конца и удаляются с другого. Пример использования: управление задачами в процессоре.
- **Деревья** — иерархическая структура данных, где каждый элемент (узел) имеет подчиненные элементы (потомки). Наиболее известным примером является двоичное дерево, где у каждого узла может быть не более двух потомков.
- **Графы** — это структура данных, представляющая собой множество узлов (вершин) и связей между ними (ребра). Примеры графов включают социальные сети, транспортные сети и компьютерные сети.

Алгоритмы работы со структурами данных

Алгоритмы поиска:

- **Линейный поиск** — алгоритм, который проходит по всем элементам структуры данных и сравнивает их с искомым значением. Время выполнения пропорционально количеству элементов.
- **Бинарный поиск** — более эффективный алгоритм, применимый к отсортированным данным. Он делит массив пополам и сравнивает

искомое значение с центральным элементом. Время выполнения*: O(log N).*

Алгоритмы сортировки:

- **Сортировка пузырьком** — простой алгоритм, который попарно сравнивает соседние элементы и меняет их местами, если они находятся в неправильном порядке.
- **Быстрая сортировка** — один из самых эффективных алгоритмов сортировки. Он выбирает опорный элемент и делит массив на две части: элементы меньше опорного и больше опорного, затем рекурсивно сортирует каждую часть.

Контрольные вопросы

1. Что такое алгоритм? Приведите пример.
2. Какие основные характеристики алгоритма?
3. В чем разница между линейной и нелинейной структурой данных?
4. Каковы основные типы структур данных?
5. Что означает термин "сложность алгоритма"? Какие существуют виды сложности?
6. Чем отличаются статические и динамические структуры данных?
7. Что такое массив, и в чем его ключевая особенность?
8. Объясните разницу между стеком и очередью.
9. Каков процесс выбора подходящей структуры данных для конкретной задачи?
10. Какие этапы включает в себя разработка алгоритма?

ТЕМА 2. ПОНЯТИЕ АЛГОРИТМА. РЕКУРСИВНЫЕ АЛГОРИТМЫ

Алгоритм — это набор инструкций, который определяет порядок действий для выполнения задачи. В фпрограммировании алгоритм представляет собой программу или её часть, которая определяет, какие операции выполнять и в какой последовательности.

Основные свойства алгоритмов:

- **Дискретность**: Алгоритм состоит из отдельных шагов.
- **Понятность**: Каждый шаг должен быть четко определен.
- **Определенность**: Одна и та же последовательность действий должна приводить к одинаковому результату.
- **Конечность**: Алгоритм должен завершаться через определённое количество шагов.
- **Результативность**: Алгоритм должен давать определённый результат.

Пример простого алгоритма

Представим, что мы решаем задачу сложения двух чисел:

1. Вводим два числа.
2. Складываем эти два числа.
3. Выводим результат сложения.

Алгоритмы можно классифицировать на несколько типов: линейные, разветвляющиеся, циклические и рекурсивные.

Рекурсия — это один из ключевых концептов в программировании, когда функция вызывает саму себя. Рекурсивные алгоритмы очень мощны и часто используются для решения задач, которые могут быть разделены на подобные подзадачи. Например, вычисление факториала числа или решение задачи о ханойских башнях.

Головоломка *Ханойские башни* состоит из трёх стержней, пронумеруем их слева направо: 1, 2 и 3. Также в головоломке используется стопка дисков с отверстием посередине. Радиус дисков уменьшается снизу вверх. Изначально диски расположены на левом стержне (стержень 1), самый большой диск находится внизу. Диски в игре перемещаются по одному со стержня на стержень. Диск можно надеть на стержень, только если он пустой или верхний диск на нём большего размера, чем перемещаемый.

Цель головоломки — перенести все диски со стержня 1 на стержень 3. Попробуйте нашу интерактивную версию Ханойских башен и узнайте, как переместить все диски с одного стержня на другой.

Ханойские башни: *Вывод списка действий, необходимых для решения головоломки «Ханойские башни».*

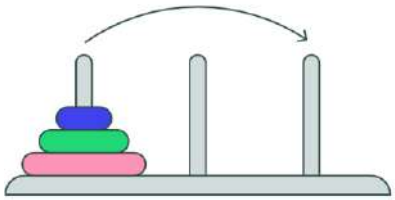
- Входные данные: Целое число n*n*.

- Выходные данные: Последовательность ходов для решения головоломки «Ханойские башни» из n*n* дисков.

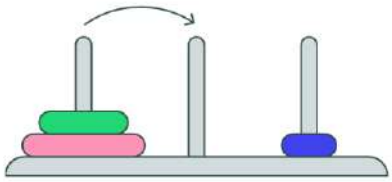
Решить головоломку с одним диском легко — просто переместите его на правый стержень. Головоломка на два диска ненамного сложнее. Сначала нужно переместить маленький диск на стержень посередине, а большой — на стержень справа. Затем переместить маленький диск на большой на правом стержне.

Версия на три диска чуть сложнее, но и ее можно решить с помощью следующих семи шагов:

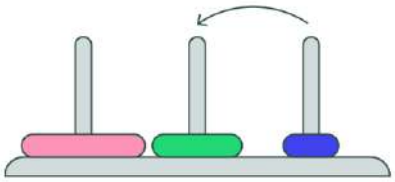
- Переместить диск со стержня 1 на стержень 3



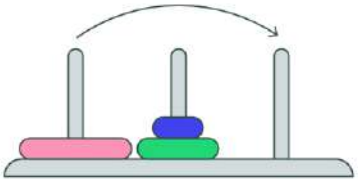
- Переместить диск со стержня 1 на стержень 2



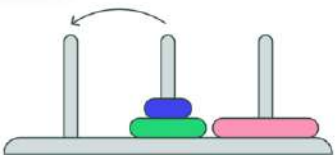
- Переместить диск со стержня 3 на стержень 2



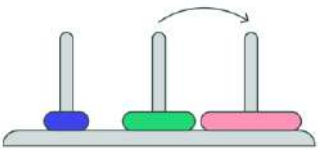
- Переместить диск со стержня 1 на стержень 3



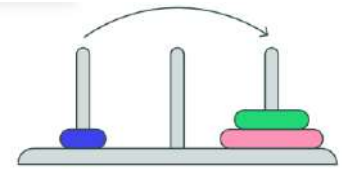
- Переместить диск со стержня 2 на стержень 1



- Переместить диск со стержня 2 на стержень 3



- Переместить диск со стержня 1 на стержень 3



Теперь давайте посчитаем, сколько шагов потребуется для решения версии на четыре диска. Нам нужно обязательно переместить самый большой диск, но для этого придётся сперва поместить все остальные диски на пустой стержень. Если у нас не три диска, а четыре, то нужно переложить три верхних диска на пустой стержень (7 действий), а затем переместить самый большой диск (1 действие). Теперь нужно снова переместить три диска с «временного» стержня на самый большой диск (еще 7 действий). Весь процесс будет состоять из 7+1+7=15 действий.

Обобщим. Чтобы переместить n дисков с левого стержня на правый, сначала необходимо переместить n−1 дисков на стержень посередине. Затем, когда диск под номером *n*, самый большой, оказывается на правом стержне, нужно переместить на него оставшиеся диски со стержня посередине. Чтобы переместить n−1 дисков со стержня посередине направо, нужно сначала переместить n−2 дисков на стержень слева, затем переместить (n−1)-й диск вправо, потом переместить n−2 дисков с левого стержня на правый и так далее.

На первый взгляд задача «Ханойские башни» может показаться сложной. Тем не менее данный ***рекурсивный алгоритм*** находит нужные перемещения дисков всего за 8 строк.

```
HanoiTowers(n,fromPeg,toPeg)
    if n = 1:
        output “Move disk from peg fromPeg to peg
toPeg”
        return
    unusedPeg = 6 - fromPeg - toPeg
    HanoiTowers(n-1,fromPeg,unusedPeg)
    output “Move disk from peg fromPeg to peg toPeg”
    HanoiTowers(n-1,unusedPeg,toPeg)
```

Переменные fromPeg,toPeg и unusedPeg указывают на три разных стержня. Таким образом, HanoiTowers(n, 1, 3) перемещает диски (n шт.) с первого стержня на третий.

Переменная unusedPeg указывает, какой из трёх стержней можно использовать для временного хранения первых (n−1) дисков. Обратите внимание, что fromPeg+toPeg+unusedPeg всегда равняется 1+2+3=61+2+3=6.

Таким образом, значение переменной unusedPeg можно определить как *6*−fromPeg−toPeg. Представленная таблица показывает

результаты 6−fromPeg−toPeg для всех возможных переменных fromPeg и toPeg.

fromPeg	toPeg	unusedPeg
1	2	3
1	3	2
2	1	3
2	3	1
3	1	2
3	2	1

Определив unusedPeg как 6−fromPeg−toPeg, операторы

```
HanoiTowers(n-1,fromPeg,unusedPeg)
output "Move disk from peg fromPeg to peg toPeg"
HanoiTowers(n-1,unusedPeg,toPeg)
```

выполняют более простую задачу: они сначала перемещают n−1 дисков на временный стержень, затем перекладывают большой диск, а потом складывают на него оставшиеся n−1 дисков. Обратите внимание, что нет необходимости указывать, какой диск игрок должен переложить с fromPeg на toPeg: перемещается всегда тот диск, что является верхним на fromPeg.

Остановитесь и подумайте: *Сколько нужно действий, чтобы переместить 66 дисков?*

Хотя решение Ханойских башен можно уложить в 9 строк псевдокода, его выполнение займет на удивление много времени. Решение головоломки на пять дисков состоит из 31-го действия. А в решении башни из сотни дисков количество действий будет исчисляться очень большой цифрой. Такое резкое увеличение числа действий для HanoiTowers неудивительно. Заметим, что каждый раз, когда вызывается HanoiTowers(n, 1, 3), алгоритм дважды вызывает сам себя для перемещения n−1 дисков, что запускает четыре вызова для перемещения n−2 дисков и так далее.

Чтобы вычислить время выполнения HanoiTowers размера n, мы введём в рассмотрение функцию T(n) — количество перемещений дисков, которые выполняет HanoiTowers(n). Получается следующее уравнение:

$T(n)=2\cdot T(n-1)+1$.

Начиная с T(1)=1, это рекуррентное соотношение задаёт последовательность:

1,3,7,15,31,63 и так далее. Мы можем вычислить T(n), прибавив 1 с обеих сторон и обнаружив, что

$T(n)+1=2\cdot T(n-1)+1+1=2\cdot(T(n-1)+1)$.

Если мы введём новое обозначение, $U(n)=T(n)+1$, то $U(n)=2\cdot U(n-1)$. Таким образом, нужно решить следующее рекуррентное соотношение:

$U(n)=2\cdot U(n-1)$.

Начиная с $U(1)=2$, получаем последовательность

2,4,8,16,32,64,…

То есть, $U(n)=2n$ и $T(n)=U(n)-1=2n-1$.

Следовательно, HanoiTowers(n) — экспоненциальный алгоритм.

Контрольные вопросы

1. Что такое алгоритм? Дайте точное определение.
2. Какие основные этапы разработки алгоритма?
3. В чем заключается принцип работы рекурсивного алгоритма?
4. Какие преимущества и недостатки имеют рекурсивные алгоритмы?
5. Приведите пример рекурсивного алгоритма.
6. В чем разница между рекурсией и итерацией?
7. Какие типы задач лучше всего решать с использованием рекурсии?
8. Какова роль базового условия в рекурсивных алгоритмах?
9. Почему важно избегать бесконечной рекурсии?
10. Как можно оптимизировать рекурсивные алгоритмы?

ТЕМА 3. ПРОСТЫЕ СТРУКТУРЫ ДАННЫХ. СТЕК

Стек (англ. stack — стопка; читается стэк) — абстрактный тип данных, представляющий собой список элементов, организованных по принципу LIFO (англ. last in — first out, «последним пришёл — первым вышел»).

В стеках мы не можем получить доступ к произвольному элементу. Мы можем только добавлять или удалять элементы с помощью специальных методов. У стека нет также метода Contains, как у списков. Кроме того, у стека нет итератора. Для того, чтобы понимать, почему на стек накладываются такие ограничения, давайте посмотрим на то, как он работает и как используется.

Наиболее часто встречающаяся аналогия для объяснения стека — стопка тарелок (рис.1). Вне зависимости от того, сколько тарелок в стопке, мы всегда можем снять верхнюю. Чистые тарелки точно так же кладутся на верх стопки, и мы всегда будем первой брать ту тарелку, которая была положена последней.

Рисунок 1. Стопка тарелок.

Если мы положим, например, красную тарелку, затем синюю, а затем зеленую, то сначала надо будет снять зеленую, потом синюю, и, наконец, красную. Главное, что надо запомнить — тарелки всегда ставятся и на верх стопки. Когда кто-то берет тарелку, он также снимает ее сверху. Получается, что тарелки разбираются в порядке, обратном тому, в котором ставились.

Теперь, когда мы понимаем, как работает стек, введем несколько терминов. Операция добавления элемента на стек называется «push», удаления — «pop». Последний добавленный элемент называется верхушкой стека, или «top», и его можно посмотреть с помощью операции «peek». Давайте теперь посмотрим на заготовку класса, реализующего стек.

Класс Stack

Класс Stack определяет методы Push, Pop, Peek для доступа к элементам и поле Count. В реализации мы будем использовать LinkedList<T> для хранения элементов.

```
public class Stack
{
    LinkedList _items = new LinkedList();
```

```
    public void Push(T value)
    {
        throw new NotImplementedException();
    }
    public T Pop()
    {
        throw new NotImplementedException();
    }
    public T Peek()
    {
        throw new NotImplementedException();
    }
    public int Count
    {
        get;
    }
}
```

Метод Push добавляет элемент на вершину стека.

Поскольку мы используем связный список для хранения элементов, можно просто добавить новый в конец списка.

```
public void Push(T value)
{
    _items.AddLast(value);
}
```

Метод Pop удаляет элемент с вершины стека и возвращает его. Если стек пустой, то выдаётся сообщение: «InvalidOperationException».

Push добавляет элементы в конец списка, поэтому забирать их будет также с конца. В случае, если список пуст, будет выбрасываться исключение.

```
public T Pop()
{
    if (_items.Count == 0)
    {
        throw new InvalidOperationException("The
stack is empty");
    }
    T result = _items.Tail.Value;
    _items.RemoveLast();
    return result;
}
```

Метод Peek возвращает верхний элемент стека, но не удаляет его. Если стек пустой, кидает InvalidOperationException.

```
public T Peek()
{
    if (_items.Count == 0)
    {
        throw new InvalidOperationException("The
stack is empty");
    }
    return _items.Tail.Value;
}
```

Метод Count возвращает количество элементов в стеке.

Зачем нам знать, сколько элементов находится в стеке, если мы все равно не имеем к ним доступа? С помощью этого поля мы можем проверить, есть ли элементы в стеке или он пуст. Это очень полезно, учитывая, что метод Pop выдаёт исключение.

```
public int Count{
    get{
        return _items.Count;
    }
}
```

Пример: калькулятор в обратной польской записи

Классический пример использования стека — калькулятор в обратной польской, или постфиксной, записи. В ней оператор записывается после своих операндов. То есть, мы пишем:

<операнд> <операнд> <оператор>

вместо традиционного:

<операнд> <оператор> <операнд>

Другими словами, вместо «4 + 2» мы запишем «4 2 +». Если вам интересно происхождение обратной польской записи и ее названия, то:

Краткий исторический очерк: «Что такое обратная польская нотация?»

В 1920-х годах Ян Лукасевич разработал формальную логическую систему, которая позволяла задавать математические выражения без круглых скобок, помещая операторы перед (префиксная запись) или после (постфиксная запись) операндов. Например, выражение (инфиксное обозначение)

(4 + 5) × 6

может быть выражено в префиксной записи как

× 6 + 4 5 или × + 4 5 6

и может быть выражено в постфиксной записи как

4 5 + 6 × или 6 4 5 + ×

Префиксная нотация также стала известна как польская нотация в честь Лукасевича. Компания HP скорректировала постфиксную запись

для использования с помощью клавиатуры калькулятора, добавила стек для хранения операндов и функции для изменения порядка записей в стеке. Полученную систему, компания HP назвала обратной польской нотацией (RPN) также в честь Лукасевича.

То, как вычисляется обратная польская запись и почему стек так полезен при ее использовании, хорошо видно из следующего алгоритма:

```
for each input value
    if the value is an integer
        push the value on to the operand stack
    else if the value is an operator
        pop the left and right values from the
stack
        evaluate the operator
        push the result on to the stack
pop answer from stack
```

То есть, для выражения «4 2 +» действия будут следующие:

```
push(4)
push(2)
push(pop() + pop())
```

В конце в стеке окажется одно значение-6.

Далее приводится полный код простого калькулятора, который считывает выражение (например, 4 2 +) из консоли, разбивает входные данные по пробелам (["4", "2", "+"]) и выполняет алгоритм вычисления. Вычисление продолжается до тех пор, пока не будет встречено слово quit.

```
void RpnLoop()
{
    while (true)
    {
        Console.Write("> ");
        string input = Console.ReadLine();
        if (input.Trim().ToLower() == "quit")
        {
            break;
        }
        // Стек еще не обработанных значений.
        Stack values = new Stack();
        foreach (string token in input.Split(new
char[] { ' ' }))
        {
            // Если значение - целое число...
            int value;
```

```
            if (int.TryParse(token, out value))
            {
                // ... положить его на стек.
                values.Push(value);
            }
            else
            {
                // в противном случае выполнить
операцию...
                int rhs = values.Pop();
                int lhs = values.Pop();
                // ... и положить результат
обратно.
                switch (token)
                {
                    case "+":
                        values.Push(lhs + rhs);
                        break;
                    case "-":
                        values.Push(lhs - rhs);
                        break;
                    case "*":
                        values.Push(lhs * rhs);
                        break;
                    case "/":
                        values.Push(lhs / rhs);
                        break;
                    case "%":
                        values.Push(lhs % rhs);
                        break;
                    default:
                        // Если операция не +, -, *
или /
                        throw new
ArgumentException(

string.Format("Unrecognized token: {0}", token));
                }
            }
        }
        // Последний элемент на стеке и есть
результат.
```

```
        Console.WriteLine(values.Pop());
    }
}
```

Пример реализации стека на языке C# с использованием класса `Stack<T>`, предоставляемого библиотекой `System.Collections.Generic`. Также показано, как реализовать собственный стек вручную, если нужно контролировать структуру данных более детально.

1. Использование встроенного класса Stack<T>

```
class Program
{
    static void Main()
    {
        // Создаем стек
        Stack<int> stack = new Stack<int>();

        // Добавляем элементы в стек
        stack.Push(10);
        stack.Push(20);
        stack.Push(30);

        // Извлекаем и выводим верхний элемент
стека
        Console.WriteLine("Верхний элемент: " +
stack.Peek());  // 30

        // Извлекаем элементы из стека и выводим их
        while (stack.Count > 0)
        {
            Console.WriteLine("Извлечен элемент: "
+ stack.Pop());
        }
    }
}
```

2. Реализация стека с помощью массива

Если нужно реализовать стек вручную, можно использовать массив или список для хранения элементов. Вот пример с использованием массива:

```
class MyStack
{
```

```
    private int[] stack;
    private int top;
    private int capacity;

    // Конструктор
    public MyStack(int size)
    {
        stack = new int[size];
        capacity = size;
        top = -1;
    }

    // Добавление элемента в стек
    public void Push(int item)
    {
        if (top == capacity - 1)
        {
            Console.WriteLine("Стек переполнен.");
            return;
        }
        stack[++top] = item;
        Console.WriteLine("Элемент " + item + "
добавлен в стек.");
    }
    // Удаление элемента из стека
    public int Pop()
    {
        if (IsEmpty())
        {
            Console.WriteLine("Стек пуст.");
            return -1;
        }
        return stack[top--];
    }
    // Возвращает верхний элемент стека без
удаления
    public int Peek()
    {
        if (IsEmpty())
        {
            Console.WriteLine("Стек пуст.");
            return -1;
```

```
        }
        return stack[top];
    }
    // Проверка, пуст ли стек
    public bool IsEmpty()
    {
        return top == -1;
    }
    // Получение текущего размера стека
    public int Size()
    {
        return top + 1;
    }
}

class Program
{
    static void Main()
    {
        MyStack stack = new MyStack(5);
        stack.Push(10);
        stack.Push(20);
        stack.Push(30);

        Console.WriteLine("Верхний элемент: " +
stack.Peek());  // 30
        Console.WriteLine("Размер стека: " +
stack.Size());     // 3

        Console.WriteLine("Извлечен элемент: " +
stack.Pop());  // 30
        Console.WriteLine("Извлечен элемент: " +
stack.Pop());  // 20
        Console.WriteLine("Размер стека после
извлечения: " + stack.Size()); // 1
    }
}
```

В этом примере работаю следующие методы:

Push(int item) — добавляет элемент в стек, если есть место.

Pop() — извлекает элемент из стека, если стек не пуст.

Peek() — возвращает верхний элемент стека без его удаления.

IsEmpty() — проверяет, пуст ли стек.

Size() — возвращает текущее количество элементов в стеке.

Контрольные вопросы

1. Что такое стек, и какова его основная функция?
2. Как работает принцип LIFO в стеке?
3. Какие основные операции выполняются со стеком?
4. В чем различие между операциями push и pop в стеке?
5. Какие области применения стека в программировании?
6. Как стек используется в рекурсивных вызовах функций?
7. Чем отличается стек от очереди?
8. Что произойдет, если попытаться извлечь элемент из пустого стека?
9. Как можно реализовать стек с помощью массива?
10. В чем заключается проблема переполнения стека, и как ее можно решить?

ТЕМА 4. СТРУКТУРА ОЧЕРЕДИ

Очередь — это линейная динамическая структура данных, для которой выполняется правило: добавление новых данных возможно только в конец этой структуры, а удаление (извлечение) — только с начала. В англоязычной литературе этот принцип называется **FIFO** (First Input — First Output, т.е. первый пришёл — первый ушёл).

Примером из реальной жизни может быть очередь из покупателей к кассе в магазине.

Как не трудно понять, очередь — это линейный список, для которого определены всего две основные операции: добавление в конец и извлечение с начала. Значит, удобно иметь два указателя: на начало и конец этой динамической структуры. Но списки бывают односвязные и двухсвязные. Какой использовать? Подойдёт только двухсвязный список. В этом можно будет убедиться при рассмотрении основных алгоритмов для работы с очередью.

На рисунке 1 ниже показано графическое представление очереди. Как и в предыдущих темах, очередь будем строить из целых чисел, например: **3, 5, 1.**

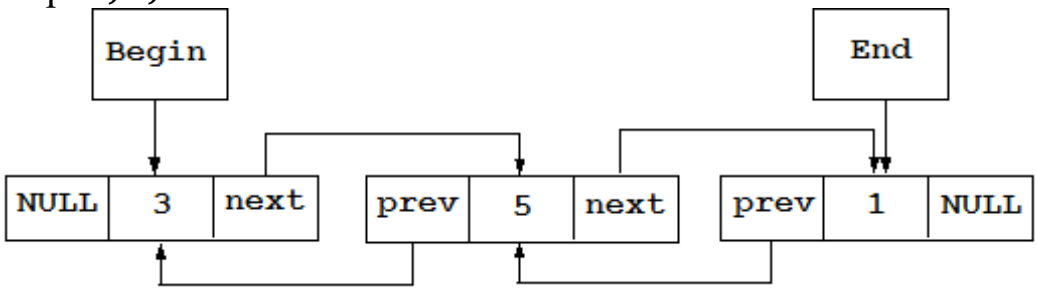


Рисунок 1. Графическое представление очереди.

Очередь - это структура данных (как было сказано выше), которая построена по принципу FIFO (first in - first out: первым пришел - первым вышел). В C++ уже есть готовый STL контейнер - queue.

В очереди, если вы добавите элемент, который вошел первым, то он выйдет тоже самым первым. Получается, если вы добавите 4 элемента, то первый добавленный элемент выйдет первым.

Чтобы понять принцип работы очереди вы можете представить себе магазинную очередь. И вы стоите посреди нее, чтобы вы оказались напротив кассы, сначала понадобится всех впереди стоящих людей обслужить. А вот для последнего человека в очереди нужно, чтобы кассир обслужил всех людей кроме него самого.

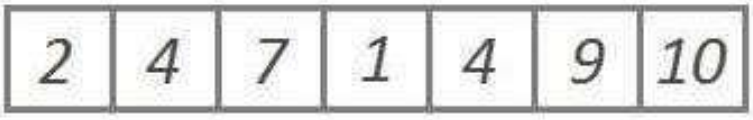


Рисунок 2. Очередь из чисел.

На рисунке слева находятся 7 чисел: 2, 4, 7, 1, 4, 9, 10. Если нам понадобится их извлечь, то извлекать мы будем в таком же порядке как они находятся на рисунке!

Так например чтобы извлечь число 4 нам понадобится сначала обслужить число 2, а потом уже и само число 4.

Хотя в стеке присутствует функция peek() в шаблоне очереди невозможно обратится к определенному элементу.

Но если вам нужно иметь доступ ко всем элементам очереди, то можете реализовать очередь через массив. Чуть ниже мы покажем как это сделать.

Создание очереди в C++

Если вы хотите использовать шаблон очереди в C++, то вам сначала нужно подключить библиотеку - <queue>.

Дальше для объявления очереди нужно воспользоваться конструкцией ниже.

```
queue <тип данных> <имя>;
```

Сначала нам нужно написать слова queue. Дальше в <тип данных> мы должны указать тот тип, которым будем заполнять нашу очередь.

И в конце нам остается только указать название очереди.

Вот пример правильного объявления:

```
queue <int> q;
```

Методы очереди

Метод - это та же самая функция, но она работает только с контейнерами STL. Примерами таких контейнеров могут служить очередь и стек.

Для работы с очередью вам понадобится знать функции: push(), pop(), front(), back(), empty(). Кстати, если хотите узнать, как в C++ работают функции и как их правильно использовать в проекте, то можете узнать все это здесь.

Для добавления в очередь нового элемента нужно воспользоваться функцией - push(). В круглых скобках должно находится значение, которое мы хотим добавить.

Если нам понадобилось удалить первый элемент нужно оперировать функцией pop(). В круглых скобках уже не чего не нужно указывать, но по правилам они в обязательном порядке должны присутствовать! Эти функции тоже не нуждаются в указании аргумента: empty(), back() и front().

Если вам понадобилось обратиться к первому элементу очереди, то вам понадобится функция front().

Чтобы обратиться к последнему элементу в очереди вам поможет функция back().

Чтобы узнать пуста ли очередь нужно воспользоваться функцией empty(). Если ваша очередь пуста - возвратит true. Если же в ней что-то есть - возвратит false.
Ниже мы использовали все выше перечисленные методы:

```
#include <iostream>
#include <queue>  // подключили библиотеку queue
using namespace std;
int main() {
  setlocale(LC_ALL,"rus");
  queue <int> q;  // создали очередь q
    cout << "Пользователь, пожалуйста введите 7
чисел: " << endl;
  for (int h = 0; h < 7; h++) {
    int a;
        cin >> a;
          q.push(a);  // добавляем в очередь
элементы
  }

  cout << endl;
  cout << "Первый элемент в очереди: " << q.front()
<< endl;  // выводим первый

// элемент очереди
  q.pop();  // удаляем элемент из очереди

  cout << "Новый первый элемент (после удаления): "
<< q.front() << endl;

  if (!q.empty()) cout << "Очередь не пуста!";  //
проверяем пуста ли очередь (нет)

  system("pause");
  return 0;
}
```

Очередь можно реализовать через массив. Обычно, если кто-то создает такую очередь, то массив называют queue. Мы бы также назвали массив, но это уже зарезервированное слова в C++. Поэтому его назовем так, как назвали выше шаблон очереди - q.

Для реализации нам понадобится создать две дополнительные переменные start и ends. start будет указывать на первый элемент очереди, а ends на последний элемент. Чтобы обратится к последнему

элементу нам придется использовать эту конструкцию - queue[ends]. Обращение к первому элементу будет выглядеть аналогично queue[start].

Если понадобится удалить элемент из очереди, то придется всего лишь уменьшить переменную start на один.

“А как же проверить пуста ли очередь?” - спросите вы. Для этого мы просто проверим условие start == ends:

Если результатом условия будет true, то очередь пуста. Если же результат условия false, значит очередь чем-то заполнена.

Ниже приведён пример создания такой очереди:

```
  setlocale(LC_ALL,"rus");
  int q[7];  // создали массив q
  int start = 0, ends = 0;  // создали переменные
начала и конца очереди

  cout << "Пользователь, пожалуйста введите 7
чисел: " << endl;

  for (int h = 0; h < 7; h++) { int a; cin >> a;
    int a;

    cin >> a;

    q[ends++] = a;  // добавляем элементы в
очередь(массив)
  }

  cout << "Первый элемент в очереди: " << q[start]
<< endl;

  start++;  // удаляем первый элемент(увеличиваем
индекс начала очереди на 1)

  cout << "Новый первый элемент (после удаления): "
<< q[start] << endl;

  cout << "Последний элемент в очереди: " << q[ends
- 1];  // выводим последний

// элемент очереди
  if (start != ends) cout << "Очередь заполнена!";
// проверяем пуста ли
```

Очередь с приоритетом

Очередь с приоритетом (priority_queue) - это обычная очередь, но в ней новый элемент добавляется в то место, чтобы очередь была отсортирована по убыванию.

Так самый большой элемент в приоритетной очереди будет стоять на первом месте.

Для объявления шаблона очереди с приоритетом нужно использовать конструкцию ниже:

```
priority_queue <тип данных> <имя>;
```

В начале нужно написать priority_queue. Потом в скобках указать тип данных, который будет находится в очереди.

И конечно же в самом конце мы должны дать ей имя.

Для добавления элемента в очередь с приоритетом мы должны использовать функцию push(). Но чтобы обратится к первому элементу должны использоваться именно функция - top() (как и в стеке). А не функция - front().

Также нельзя использовать функцию back() для обращения к последнему элементу. Для приоритетной очереди она также не работает, как функция front().

Вот пример использования очереди с приоритетом в программе:

```
 setlocale(LC_ALL,"rus");
    priority_queue <int> priority_q;  // объявляем
приоритетную очередь
    cout << "Введите 7 чисел: " << endl;
   for (int j = 0; j < 7; j++) { int a; cin >> a;
      priority_q.push(a);  // добавляем элементы в
очередь
  }

// выводим первый
  cout << "Первый элемент очереди: " <<
priority_q.top();  // элемент
```

Лично мы всегда используем первый способ реализации очереди. Он работает быстрее чем второй способ и реализация совсем простая.

Контрольные вопросы

1. Что такое очередь, и как она функционирует?
2. В чем заключается принцип FIFO в очереди?
3. Какие основные операции выполняются с очередью?
4. Как отличаются операции enqueue и dequeue в очереди?
5. Приведите пример реальной ситуации, где используется структура очереди.

6. Чем очередь отличается от стека?
7. Какие существуют типы очередей (например, циклическая очередь, приоритетная очередь)?
8. В чем заключается проблема переполнения очереди и как ее можно избежать?
9. Как очередь используется в системах управления задачами?
10. Каким образом можно реализовать очередь с использованием связного списка?

ТЕМА 5. СТРУКТУРА СТРОК ДАННЫХ

Строка — одна из самых распространенных структур данных в программировании. Они используются для хранения текстовой информации и являются основой работы с данными в большинстве приложений. Понимание строковых структур данных и их особенностей позволяет разработчикам эффективно управлять текстовой информацией.

Строка — это последовательность символов, которая может содержать буквы, цифры, знаки препинания и пробелы. В большинстве языков программирования строки представляются как массивы символов, где каждый символ имеет свой индекс. Например, строка "Привет" состоит из 6 символов:

Индекс: 0 1 2 3 4 5

Символ: 'П' 'р' 'и' 'в' 'е' 'т'

Основные операции со строками

Строковые структуры данных поддерживают множество операций. К основным из них относятся:

1. **Создание строки** — инициализация строки с помощью литералов или функций.
2. **Конкатенация** — объединение двух строк в одну.
3. **Извлечение подстроки** — получение части строки по заданным индексам.
4. **Поиск** — нахождение позиции подстроки внутри строки.
5. **Замена** — изменение одной подстроки на другую в строке.
6. **Разделение** — деление строки на части по определенному разделителю.

Реализация строковых структур данных

Строки могут быть реализованы различными способами в зависимости от языка программирования. В языках, таких как Python, строки являются встроенными типами данных, а в языках, таких как C, строки представляют собой массивы символов, заканчивающиеся нулевым символом ('\0').

Пример на языке C

```
#include <stdio.h>
int main() {
    char str1[] = "Привет";
    char str2[] = " мир";
    char result[20];

    // Конкатенация строк
    strcpy(result, str1);
```

```
    strcat(result, str2);

    printf("%s\n", result); // Вывод: Привет мир
    return 0;
}
```

Особенности строковых структур данных

1. **Иммутабельность** — в некоторых языках программирования (например, Python, Java) строки являются неизменяемыми. Это означает, что при изменении строки создается новая строка, а старая остается неизменной.
2. **Управление памятью** — строковые структуры могут занимать значительное количество памяти, особенно при работе с большими текстами. Эффективное управление памятью имеет важное значение для производительности приложения.
3. **Кодировки** — строки могут быть представлены в различных кодировках, таких как UTF-8, ASCII и другие. Понимание кодировок позволяет избежать проблем с совместимостью при работе с текстовыми данными.

Строковые структуры данных играют ключевую роль в обработке текстовой информации. Понимание их особенностей и операций позволяет разработчикам эффективно работать с данными. Применение строковых структур данных в различных алгоритмах и приложениях помогает решить множество задач, связанных с текстом.

Контрольные вопросы

1. Что такое строка данных, и как она представляется в памяти компьютера?
2. Какие операции можно выполнять со строками?
3. В чем разница между изменяемыми и неизменяемыми строками?
4. Как происходит объединение строк, и какие методы используются для этого?
5. Как можно сравнивать строки в программировании?
6. Что такое длина строки, и как ее можно определить?
7. Какие структуры данных могут быть использованы для хранения строк?
8. Как строки кодируются в различных кодировках (например, ASCII, Unicode)?
9. Какие методы поиска подстроки в строке существуют?
10. Какие задачи лучше всего решать с использованием строк?

ТЕМА 6. МАССИВЫ

Массив — это тип структуры данных, в которой элементы или данные хранятся в смежных местах. Всякий раз, когда у пользователя есть набор данных с одним и тем же типом данных, структура данных массива является вариантом для организации этих данных**.** Размер массива зависит от размера данных. Прежде чем элементы будут сохранены в массиве, необходимо определить размер массива, чтобы он эффективно включал все элементы. Каждому элементу, хранящемуся в массиве, присвоено значение индекса, которое помогает определить местоположение этого элемента в массиве. Первый элемент массива имеет значение индекса, равное нулю.

Важные термины, связанные со структурой данных массива:

- **Элемент.** Элемент представляет каждый объект или элемент, хранящийся в структуре данных.
- **Индекс.** Индекс представляет расположение элемента в массиве. Он имеет числовое значение.

Размер массива меняется в зависимости от языка программирования. В зависимости от размера массив может быть двух типов: статический и динамический.

Концепция массива

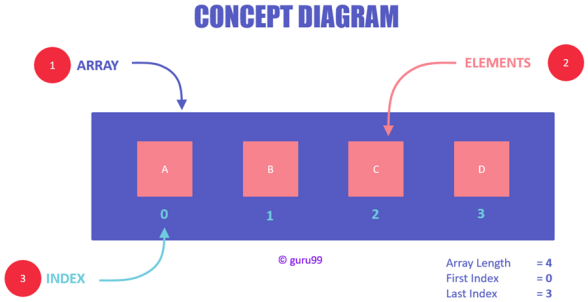


Рисунок 1. Концептуальная диаграмма массивов.

Приведенная выше диаграмма иллюстрирует это:

1. Массив — это контейнер элементов.
2. Элементы имеют определенное значение и тип данных, например «ABC», TRUE или FALSE и т. д.
3. Каждый элемент также имеет свой индекс, который используется для доступа к элементу.

Примечание:

- Элементы хранятся в смежных ячейках памяти.
- Индекс всегда меньше общего количества элементов массива.
- С точки зрения синтаксиса любая переменная, объявленная как массив, может хранить несколько значений.
- Почти все языки одинаково понимают массивы, но имеют разные способы их объявления и инициализации.

- Однако три части всегда будут оставаться общими при всех инициализациях, т. е. имя массива, элементы и тип данных элементов.

Следующая диаграмма иллюстрирует синтаксис объявления массив в Python и C++ чтобы представить, что понимание остается тем же, хотя синтаксис может немного различаться в разных языках.

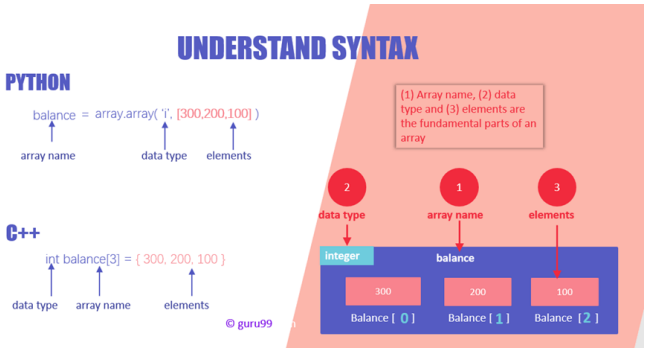


Рисунок 2. Понимание синтаксиса массивов.

- **Имя массива:** необходимо для удобного обращения к коллекции элементов
- **Тип данных:** необходимо для проверки типов и целостности данных
- **Элементы:** это значения данных, присутствующие в массиве

Зачем нужны массивы?

Вот несколько причин использования массивов в структуре данных:

- Массивы лучше всего подходят для хранения нескольких значений в одной переменной.
- Массивы лучше и быстрее обрабатывают множество значений.
- Сортировка и поиск значений проще в массивах.

Создание массива в Python

In Python, массивы отличаются от списков; списки могут содержать элементы массива типов данных, тогда как массивы могут содержать только элементы одного и того же типа данных.

Python имеет отдельный модуль для работы с массивами, называемый array, который необходимо импортировать, прежде чем начать с ними работать.

Примечание: Массив должен содержать действительные числа, такие как целые числа и числа с плавающей запятой, строки не допускаются.

Следующий код показывает, как можно создать целочисленный массив в Python для хранения баланса счета:

```
import array
```

```
balance = array.array('i', [300,200,100])
print(balance)
```

Способы объявления массива в Python

Вы можете объявить массив в Python при инициализации с использованием следующего синтаксиса.

```
arrayName = array.array(type code for data type,
[array,items])
```

Следующее изображение объясняет синтаксис.

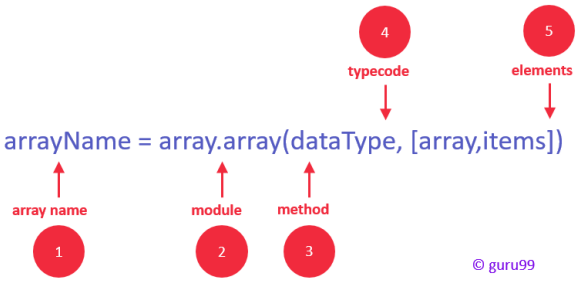


Рисунок 3. Синтаксис массива в Python.

1. **Идентификатор**: имя массива.
2. **Модули**: Python имеет специальный модуль для создания массивов, который называется «array» — перед использованием его необходимо импортировать. Модуль массива имеет метод инициализации массива. Он принимает два аргумента: тип и элементы.
3. **Тип**: тип данных массива.
4. **Элементы**: элементы массива в квадратных скобках, например [130,450,103]

Как получить доступ к определенному значению массива?

Вы можете получить доступ к любому элементу массива, используя его индекс, индексация элементов массива начинается с нуля.

Синтаксис

arrayName[indexNum]

Это критически важно для анализа и выбора наиболее эффективных ключевых слов для улучшения рейтинга вашего сайта.

Следующее изображение иллюстрирует базовую концепцию доступа к элементам массива по их индексу.

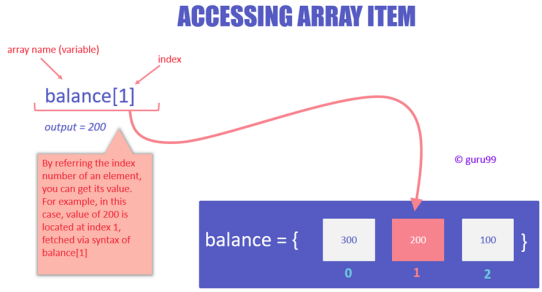


Рисунок 4. Доступ к элементу массива.

В примере мы получили доступ ко второму значению массива, используя его индекс, который равен 1. Результатом будет 200.

```
import array
balance = array.array('i', [300,200,100])
print(balance[1])
```

Вывод

200

Модуль массива Python имеет отдельные функции для выполнения операций с массивами. Рассмотрим деструктивные метод работы с массивами, что означает, что изменение будет сохранено в переменной массива без создания его копии.

Рассмотрим следующие методы:

1) Вставка

С помощью этой операции вы можете вставить один или несколько элементов в массив в начало, конец или любой заданный индекс массива. Этот метод ожидает два аргумента: индекс и значение.

Синтаксис метода - *arrayName.insert(index, value)*

Добавим новое значение сразу после второго элемента массива. В настоящее время наш массив баланса содержит три элемента: 300, 200 и 100. Итак, каков будет индекс второго элемента массива со значением 200, если вы выбрали индекс 1.

Чтобы вставить новое значение сразу после индекса 1, вам нужно сослаться на индекс 2 в методе вставки, например:

```
import array
balance = array.array('i', [300,200,100])
balance.insert(2, 150)
```

Теперь, чтобы проверить, вставлено ли новое значение, введите имя массива и нажмите Enter на клавиатуре:

```
import array
balance = array.array('i', [300,200,100])
balance.insert(2, 150)
print(balance)
```

2) Удаление

С помощью этой операции вы можете удалить один элемент из массива по значению. Этот метод принимает только один аргумент — значение. После запуска этого метода элементы массива переупорядочиваются и индексы переназначаются.

Синтаксис метода - *arrayName.remove(value)*

Давайте удалим значение 150 из массива. В настоящее время наш массив баланса содержит четыре элемента: 300, 200, 150 и 100. Таким

образом, чтобы удалить 150 из массива, нам нужно всего лишь ввести 150 внутри аргумента метода.

```
import array
balance = array.array('i', [300,200,100])
balance.insert(2, 150)
print(balance)
balance.remove(150)
```

Теперь, чтобы проверить, было ли удалено значение, введите имя массива и нажмите Enter на клавиатуре:

```
import array
balance = array.array('i', [300,200,100])
balance.insert(2, 150)
print(balance)
balance.remove(150)
print(balance)
```

3) Поиск

С помощью этой операции вы можете искать элемент в массиве по его значению. Этот метод принимает только один аргумент — значение. Это неразрушающий метод, то есть он не влияет на значения массива.

Синтаксис метода - *arrayName.index(value)*

Давайте найдем значение 150 в массиве. В настоящее время наш массив баланса содержит четыре элемента: 300, 200, 150 и 100. Таким образом, чтобы найти 150 в массиве, нам нужно всего лишь ввести 150 внутри аргумента метода. Это довольно легко. Этот метод возвращает индекс искомого значения.

```
import array
balance = array.array('i', [300,200,150,100])
print(balance.index(150))
```

4) Обновление

Эта операция очень похожа на метод вставки, за исключением того, что она заменит существующее значение по заданному индексу. Это означает, что он просто присвоит новое значение по данному индексу. Этот метод ожидает два аргумента: индекс и значение.

Синтаксис метода - *arrayName.udpate(index, value)*

Предположим, что в нашем массиве четыре элемента: 300, 200, 150 и 100, и мы хотим заменить 150 на 145.

Чтобы заменить значение 150 с индексом 2, вам нужно сослаться на индекс 2, используя простой оператор присваивания, например этот:

```
import array
balance = array.array('i', [300,200,150,100])
```

```
balance[2] = 145
```

Теперь, чтобы проверить, обновилось ли значение, введите имя массива и нажмите Enter на клавиатуре:

```
import array
balance = array.array('i', [300,200,150,100])
balance[2] = 145
print(balance)
```

Вывод

array('i', [300,200,145,100])

Создание массива в C++

C++ язык более гибкий, чем Python когда дело доходит до создания массивов. Вы можете создать C++ массивы тремя способами, упомянутыми выше.

Следующий код иллюстрирует, как можно создать целочисленный массив в C++ для хранения баланса счета:

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

int main()
{
  int balance[3] = { 300, 200, 100 };
  for (int i = 0; i < 3; i++)
    {
      cout << "value of i: " << balance[i] << endl;
    }
  return 0;
}
```

Способы объявления массива в C++

Вы можете объявить массив в трех вариантах синтаксиса. Какой из них подходит вашей программе; этот выбор зависит от требований вашей программы.

1) *dataType arrayName[arraySize];*

int balance[3];

2) *dataType arrayName[] = {array, items};*

int balance[] = { 300, 200, 100 };

3) *dataType arrayName[arraySize] = {array, items};*

int balance[3] = { 300, 200, 100 };

Доступ к определенному значению массива

Вы можете получить доступ к любому элементу массива, используя его индекс.

arrayName[indexNum]

balance[1]

Следующее изображение иллюстрирует базовую концепцию доступа к элементам массива по их индексу.

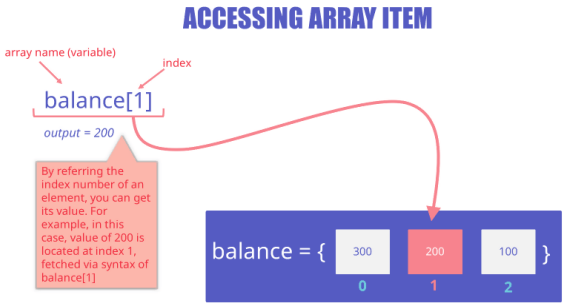


Рисунок 5. Доступ к элементу массива C++.

Здесь мы получили доступ ко второму значению массива, используя его индекс, который равен 1. Результатом будет 200.

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

int main()
{
  int balance[3] = { 300, 200, 100 };
  cout << balance[1];

  return 0;
}
```

В отличие от Python в C++ вам необходимо самостоятельно запрограммировать логику для выполнения операций вставки, удаления, поиска, обновления и перебора элементов массива.

Методы в C++.

1) Вставка

Алгоритм операции вставки выглядит следующим образом:

1. перебирать элементы массива
2. сдвинуть их к большему индексу
3. добавить новый элемент массива по заданному индексу

В следующем примере у нас есть 5 элементов в массиве баланса, и мы хотим добавить новый элемент сразу после значения 200. Это означает, что нам нужно сдвинуть все элементы после 200 к большему индексу, а затем вставить наш новый элемент со значением 150.

```
#include <iostream>
#include <stdio.h>
main() {
   int pos = 2;
   int size = 4;
```

```
int balance[] = {300,200,100,50,0};

printf("BEFORE INCREMENT: \n");
for(int i = 0; i<5; i++) {
   printf("%d\t",balance[i]);
}

/* FOR SHIFTING ITEMS TO A GREATER INDEX */
for(int i = size; i >= pos; i--) {
    balance[i+1]=balance[i];
}

/* FOR INSERTING VALUE AT OUR DESIRED INDEX */
balance[pos] = 150;
   printf("AFTER INCREMENT: \n");
   /* FOR PRINTING THE NEW ARRAY */
for(int i = 0; i<6; i++) {
   printf("%d\t",balance[i]);
} }
```

Результат

Перед вставкой	После вставки
BEFORE INCREMENT 300 200 100 50 0	AFTER INCREMENT 300 200 150 100 50 0

Заключение

Массив — это структура данных для хранения нескольких элементов данных, имеющих одинаковый тип данных.

Идентификатор, тип данных, длина массива, элементы и индекс являются основными частями массива.

Индекс используется для обработки значений элементов массива.

Массивы имеют отличную возможность сохранения типа данных без изменений.

В большинстве языков массив создается путем указания идентификатора, типа данных и элементов массива.

Массивы лучше всего подходят для обработки большого количества значений, а также для быстрой сортировки и поиска.

Python имеет модули и встроенные методы для выполнения основных операций с массивами, таких как вставка, удаление, поиск, обновление и перебор значений.

В C++ необходимо определить программы для базовых операций с массивами, таких как вставка, удаление, поиск, обновление и перебор значений.

Контрольные вопросы

1. Что такое массив, и как он используется в программировании?
2. Какие типы массивов существуют (одномерные, многомерные)?
3. Как создается и инициализируется массив в различных языках программирования?
4. Как осуществляется доступ к элементам массива?
5. В чем преимущество использования массива по сравнению с другими структурами данных?
6. Как можно изменить размер массива? Можно ли это сделать динамически?
7. В чем разница между статическими и динамическими массивами?
8. Как осуществляется поиск элементов в массиве? Какие существуют алгоритмы поиска?
9. Как можно сортировать массив? Приведите примеры методов сортировки.
10. Какие проблемы могут возникнуть при работе с массивами (например, выход за пределы массива)?

ТЕМА 7. СТРУКТУРА СПИСКА ДАННЫХ

Список данных является одним из фундаментальных понятий в программировании, который широко применяется для структурирования и организации информации. Структура списка данных определяет способ хранения, доступа и обработки данных, что делает ее важным инструментом в разработке программного обеспечения. Давайте рассмотрим основные принципы и виды структур списка данных.

1. Односвязный список представляет собой структуру данных, состоящую из узлов, каждый из которых содержит данные и ссылку на следующий узел. Это позволяет эффективно добавлять и удалять элементы в списке, обеспечивая линейный доступ к данным.

2. Двусвязный список расширяет концепцию односвязного списка, добавляя каждому узлу ссылку на предыдущий узел. Это обеспечивает возможность двунаправленного обхода списка, что упрощает операции вставки и удаления элементов.

3. Кольцевой список - это список данных, в котором последний элемент ссылается на первый элемент, образуя замкнутую структуру. Это обеспечивает циклическое обращение к элементам списка и может быть полезно в некоторых сценариях.

4. Стек - это структура данных, в которой элементы добавляются и удаляются в соответствии с принципом "последний вошел, первый вышел" (LIFO). Он реализуется часто через односвязный список и используется для реализации операций вроде отката или обработки вызовов функций.

5. Очередь - это структура данных, где элементы добавляются в конец очереди и удаляются из ее начала в соответствии с принципом "первый вошел, первый вышел" (FIFO). Она также может быть реализована через список, обеспечивая удобное управление элементами.

Структура списка данных играет важную роль в эффективной организации и обработке информации в программировании. Понимание различных типов структур списка данных поможет разработчикам выбирать наиболее подходящий подход для решения конкретных задач и оптимизации работы программ.

Структура списка может быть реализована как в статическом, так и в динамическом виде. Эти две реализации отличаются способом хранения данных и управлением памятью. Рассмотрим обе структуры.

Статическая структура списка использует фиксированный массив для хранения элементов. Размер массива задается заранее, и его нельзя изменить во время выполнения программы. Это означает, что если список заполнен, то добавление новых элементов невозможно без перераспределения памяти и копирования данных в новый массив.

Пример статической структуры списка на C++:

```
#include <iostream>
class StaticList {
private:
    int arr[100];  // Массив фиксированного размера
    int size;      // Текущий размер списка
public:
    StaticList() : size(0) {}
    void add(int value) {
        if (size < 100) { // Проверяем, есть ли
место для нового элемента
            arr[size] = value;
            size++;
        } else {
            std::cout << "List is full!" <<
std::endl;
        }
    }
    void print() const {
        for (int i = 0; i < size; i++) {
            std::cout << arr[i] << " ";
        }
        std::cout << std::endl;
    }
};
int main() {
    StaticList list;
    list.add(10);
    list.add(20);
    list.add(30);
    list.print();  // Output: 10 20 30
    return 0;
}
```

*Особенности статической структуры списка**:***

- **Плюсы**:
  - Простота реализации.
  - Нет необходимости в управлении динамической памятью.
  - Более высокая производительность за счет отсутствия выделения и освобождения памяти во время выполнения.
- **Минусы**:
  - Ограниченность по размеру (если массив переполнен, нужно пересоздавать его или ограничить добавление элементов).

- Память под массив выделяется сразу и может быть неиспользованной, что приводит к неэффективному расходу памяти.

Динамическая структура списка, в отличие от статической, не имеет фиксированного размера. Элементы списка размещаются в динамической памяти, что позволяет списку расти или уменьшаться по мере необходимости. Для реализации динамических списков часто используют структуры данных, такие как связные списки или векторы (в стандартной библиотеке C++).

Пример динамической структуры списка (односвязный список) на C++:

```
#include <iostream>
class Node {
public:
    int data;
    Node* next;
    Node(int value) : data(value), next(nullptr) {}
};
class DynamicList {
private:
    Node* head;
public:
    DynamicList() : head(nullptr) {}
    ~DynamicList() {
        // Освобождаем память, когда список больше
не нужен
        Node* current = head;
        while (current != nullptr) {
            Node* next = current->next;
            delete current;
            current = next;
        }
    }

    void add(int value) {
        Node* newNode = new Node(value);
        if (head == nullptr) {
            head = newNode;
        } else {
            Node* temp = head;
            while (temp->next != nullptr) {
                temp = temp->next;
```

```
            }
            temp->next = newNode;
        }
    }

    void print() const {
        Node* temp = head;
        while (temp != nullptr) {
            std::cout << temp->data << " ";
            temp = temp->next;
        }
        std::cout << std::endl;
    }
};

int main() {
    DynamicList list;
    list.add(10);
    list.add(20);
    list.add(30);
    list.print();  // Output: 10 20 30

    return 0;
}
```

*Особенности динамической структуры списка**:***

- ***Плюсы***:
  - Гибкость в плане размера: список может динамически увеличиваться и уменьшаться.
  - Память выделяется по мере необходимости, что позволяет более эффективно использовать ресурсы.
  - Нет ограничений по количеству элементов, если есть доступная память.
- ***Минусы***:
  - Сложность реализации: нужно управлять динамической памятью, не забывая освобождать её.
  - Более высокая нагрузка на систему, так как требуется выделение и освобождение памяти во время выполнения программы.

- В связных списках сложнее получать доступ к элементам (в отличие от массивов, где доступ осуществляется по индексу).

Основные отличия:

- Статическая структура списка**:**
  - Фиксированный размер.
  - Простота и высокая производительность, но ограниченные возможности по изменению размера.
  - Использование заранее выделенной памяти.
- Динамическая структура списка**:**
  - Переменный размер, который можно увеличивать и уменьшать по мере необходимости.
  - Более сложная реализация с управлением динамической памятью, но позволяет работать с большими объемами данных, если это необходимо.

Примеры использования:

- Статические списки подходят для ситуаций, где заранее известен максимальный размер данных, например, работа с ограниченным количеством данных (счётчики, настройки).
- Динамические списки полезны в тех случаях, когда данные неизвестны заранее или могут сильно варьироваться по объему, например, работа с потоками данных, коллекциями в приложениях и т.д.

Основные методы работы со списками в C++ зависят от того, какой тип списка вы используете: статический (массив) или динамический (например, односвязный или двусвязный список, либо использование контейнеров из стандартной библиотеки STL). Рассмотрим методы для обоих подходов:

1. Методы для статического списка (на основе массива)

Если список реализован на основе массива, набор методов обычно ограничен операциями вставки, удаления и доступа к элементам.

Основные методы для статического списка:

- add(value): добавление элемента в конец списка.
- insert(value, index): вставка элемента на определенную позицию.
- remove(): удаление элемента с конца.
- removeAt(index): удаление элемента с определенной позиции.
- get(index): получение элемента по индексу.
- print(): печать всех элементов списка.

Пример методов для статического списка:

```
#include <iostream>
class StaticList {
```

```
private:
    int arr[100];  // Массив фиксированного размера
    int size;      // Текущий размер списка
public:
    StaticList() : size(0) {}
    // Добавление элемента в конец списка
    void add(int value) {
        if (size < 100) {
            arr[size] = value;
            size++;
        } else {
            std::cout << "List is full!" <<
std::endl;
        }
    }
    // Вставка элемента на определенную позицию
    void insert(int value, int index) {
        if (index >= 0 && index <= size && size <
100) {
            for (int i = size; i > index; i--) {
                arr[i] = arr[i - 1];
            }
            arr[index] = value;
            size++;
        } else {
            std::cout << "Invalid index or list is
full!" << std::endl;
        }
    }
    // Удаление элемента с конца
    void remove() {
        if (size > 0) {
            size--;
        } else {
            std::cout << "List is empty!" <<
std::endl;
        }
    }

    // Удаление элемента с конкретной позиции
    void removeAt(int index) {
        if (index >= 0 && index < size) {
```

```
            for (int i = index; i < size - 1; i++)
{
                arr[i] = arr[i + 1];
            }
            size--;
        } else {
            std::cout << "Invalid index!" <<
std::endl;
        }
    }
    // Доступ к элементу по индексу
    int get(int index) const {
        if (index >= 0 && index < size) {
            return arr[index];
        } else {
            std::cout << "Invalid index!" <<
std::endl;
            return -1;  // Возвращаем значение по
умолчанию
        }
    }
    // Печать списка
    void print() const {
        for (int i = 0; i < size; i++) {
            std::cout << arr[i] << " ";
        }
        std::cout << std::endl;
    }
};

int main() {
    StaticList list;
    list.add(10);+
    list.add(20);
    list.insert(15, 1);  // Вставка 15 на позицию 1
    list.print();        // Output: 10 15 20

    list.removeAt(1);    // Удаление элемента на
позиции 1
    list.print();        // Output: 10 20
```

```
    std::cout << list.get(0) << std::endl;  // 
Output: 10

    return 0;
}
```

Методы для динамического списка (связный список). Для динамических списков, таких как односвязные или двусвязные списки, существуют более сложные методы, которые включают управление узлами (элементами списка), добавление, удаление и итерацию по списку.

Основные методы для динамического списка:

- add(value): добавление элемента в конец списка.
- insertAtBeginning(value): вставка элемента в начало списка.
- remove(): удаление элемента с конца списка.
- removeByValue(value): удаление элемента по значению.
- get(index): получение значения элемента по индексу.
- print(): печать всех элементов списка.

Пример методов для односвязного списка:

```
#include <iostream>
class Node {
public:
    int data;
    Node* next;
    Node(int value) : data(value), next(nullptr) {}
};

class LinkedList {
private:
    Node* head;

public:
    LinkedList() : head(nullptr) {}
    ~LinkedList() {
        // Удаляем все узлы
        Node* current = head;
        while (current != nullptr) {
            Node* next = current->next;
            delete current;
            current = next;
        }
```

```
}

// Добавление элемента в конец
void add(int value) {
    Node* newNode = new Node(value);
    if (head == nullptr) {
        head = newNode;
    } else {
        Node* temp = head;
        while (temp->next != nullptr) {
            temp = temp->next;
        }
        temp->next = newNode;
    }
}
// Вставка элемента в начало
void insertAtBeginning(int value) {
    Node* newNode = new Node(value);
    newNode->next = head;
    head = newNode;
}
// Удаление элемента с конца
void remove() {
    if (head == nullptr) return;

    if (head->next == nullptr) {
        delete head;
        head = nullptr;
    } else {
        Node* temp = head;
        while (temp->next->next != nullptr) {
            temp = temp->next;
        }
        delete temp->next;
        temp->next = nullptr;
    }
}

// Удаление элемента по значению
void removeByValue(int value) {
    if (head == nullptr) return;
    if (head->data == value) {
```

```
            Node* temp = head;
            head = head->next;
            delete temp;
            return;
        }

        Node* temp = head;
        while (temp->next != nullptr && temp->next-
>data != value) {
            temp = temp->next;
        }

        if (temp->next != nullptr) {
            Node* nodeToDelete = temp->next;
            temp->next = temp->next->next;
            delete nodeToDelete;
        }
    }
    // Получение значения по индексу
    int get(int index) const {
        Node* temp = head;
        int count = 0;
        while (temp != nullptr) {
            if (count == index) {
                return temp->data;
            }
            count++;
            temp = temp->next;
        }
        return -1;  // Если индекс вне границ
    }
    // Печать всех элементов списка
    void print() const {
        Node* temp = head;
        while (temp != nullptr) {
            std::cout << temp->data << " ";
            temp = temp->next;
        }
        std::cout << std::endl;
    }
};
int main() {
```

```
    LinkedList list;
    list.add(10);
    list.add(20);
    list.insertAtBeginning(5);  // Вставляем в
начало
    list.print();  // Output: 5 10 20

    list.removeByValue(10);     // Удаляем элемент
со значением 10
    list.print();  // Output: 5 20

    std::cout << list.get(1) << std::endl;  //
Output: 20

    return 0;
}
```

Контрольные вопросы

1. Что такое односвязный список и каковы его основные характеристики?
2. В чем отличие двусвязного списка от односвязного списка и какие преимущества он предоставляет?
3. Что представляет собой кольцевой список данных и в каких случаях он может быть полезен?
4. Как работает стек как структура данных и для каких задач его применяют? 5. Чем отличается очередь от стека и какую роль она играет в обработке данных?
6. Какие основные операции могут выполняться с элементами списка данных? 7. Какие преимущества и недостатки имеют различные типы структур списка данных?
8. Какие алгоритмы используются для работы с различными типами списка данных?
9. Какова сложность операций добавления, удаления и доступа к элементам в различных типах списков данных?
10. Как выбрать наиболее подходящую структуру списка данных для конкретной задачи в программировании?

ТЕМА 8. СВЯЗАННЫЕ СПИСКИ

Связанные списки являются одной из основных структур данных в программировании. Они представляют собой упорядоченную коллекцию элементов, где каждый элемент содержит ссылку на следующий элемент в списке. Такая связь между элементами обеспечивает гибкость в добавлении, удалении и доступе к данным. Одним из основных преимуществ связанных списков является возможность динамического изменения их размера. Это означает, что список может быть легко изменен путем добавления новых элементов или удаления существующих, без необходимости заранее выделенного фиксированного размера памяти.

Существует несколько типов связанных списков, таких как односвязный список, двусвязный список и кольцевой список. Односвязный список состоит из узлов, каждый из которых содержит данные и ссылку на следующий узел. Двусвязный список отличается тем, что каждый узел содержит ссылки как на предыдущий, так и на следующий узел. Кольцевой список представляет собой список, в котором последний элемент ссылается на первый, образуя замкнутую цепочку.

Операции над связанными списками включают добавление элемента в начало или конец списка, удаление элемента, поиск элемента по значению, обход списка и многие другие. Для эффективной работы с данными в списке необходимо уметь правильно выбирать подходящую структуру и алгоритмы для каждой конкретной задачи.

Связанные списки являются важным инструментом в разработке программного обеспечения, обеспечивая эффективное хранение и управление данными. Понимание принципов работы связанных списков позволяет разработчикам создавать более эффективные и масштабируемые программы.

Это структура, которая решает проблему производительности, если нам приходится часто добавлять и удалять данные. Данные в связном списке хранятся не подряд, а вразброс.

Каждое значение хранится в отдельном объекте, который называется **узлом**. Помимо значения, объект хранит ссылку на следующий узел списка. В самом последнем узле вместо ссылки на следующий элемент хранится значение **null**.

Опишем класс, содержащий значение (value) и ссылку на следующий узел (next):

```
class LinkedListNode {
  constructor(value, next) {
    this.value = value;
    this.next = next;
```

```
  }
}
```

Список из элементов 5, 8, 12 и 3 может быть создан так:

```
const head = new LinkedListNode(5,
    new LinkedListNode(8,
        new LinkedListNode(12,
            new LinkedListNode(3, null)
        )
    )
);
```

Оператор new не только создает объект, но и выделяет для него место в памяти. Самый первый элемент односвязного списка часто называют головой (head), поэтому и переменную со ссылкой на голову мы назвали head.

В поле next самого последнего узла находится null — значит, узлов больше нет. В отличие от массива, узлы списка не размещаются в памяти подряд:

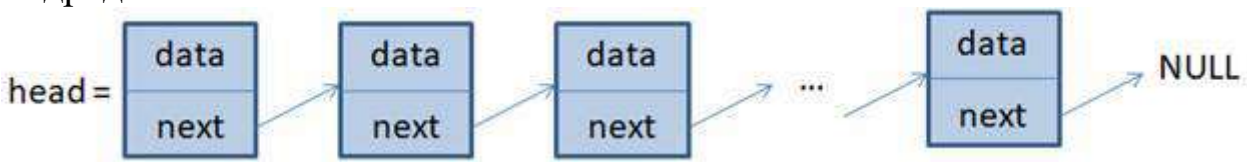


Рисунок 1. Структура связного списка.

На рисунке узлы списка занимают две ячейки. В первой хранится значение, а во второй — адрес следующего узла или null. Иногда узлы могут располагаться рядом, но в общем случае это не так.

Вся работа со списком производится через ссылку на его первый элемент, так что переменная head — единственное, что нам нужно для реализации алгоритмов.

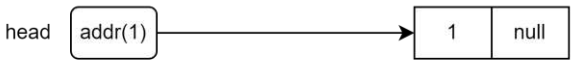
Поскольку мы пишем на языке JavaScript, который поддерживает объектно-ориентированное программирование, мы объединим поле head и все наши функции в один класс. Он будет называться LinkedList, что в переводе с английского означает связный список. При создании нового списка в поле head хранится значение null, что означает, что список пустой:

```
class LinkedList {
  head = null;
}
```

Вставка элемента в начало списка

В простейшем случае, мы вставляем элемент в пустой список. В самом начале значение поля head равно null:

После вставки head указывает на единственный элемент списка:



Мы не рисуем кучу целиком, потому что в реальной программе трудно предугадать, по какому адресу разместится то или иное значение. Конкретные адреса могут сбить с толку.

Поэтому мы просто отмечаем факт, что для нового узла списка в куче были выделены две ячейки, и head указывает на этот узел. Адрес первого узла мы запишем, как addr(1), это позволит нам отличать адреса друг от друга.

После вставки второго узла в начало списка (рис.2), картина примет такой вид:

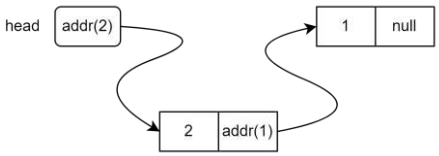


Рисунок 2. Вставка второго узла в список.

Теперь addr(1), который находился в head, переместился в узел 2, а в head попал адрес второго узла addr(2).

В обоих случаях (вставка в пустой и непустой список), сначала мы создаем узел, куда, в качестве указателя на следующий элемент, записываем текущее значение head.

Взглянем на код:

```
class LinkedList {
  head = null;
  add(value) {
    this.head = new LinkedListNode(value,
this.head);
  }
}
```

Метод add принимает параметр value (значение). Он создает новый узел, помещает туда значение и вставляет узел в начало списка.

Метод состоит из единственной строки:

```
this.head = new LinkedListNode(value, this.head);
```

Здесь мы совместили две операции, которые можно было бы записать в две строки:

```
let node = new LinkedListNode(value, this.head);
this.head = node;
```

Если первый вариант кода кажется вам сложным, вы можете остановиться на втором. Обычно программисты, по мере чтения кода, привыкают к часто встречающимся конструкциям. Знакомые конструкции уже не кажутся сложными.

Насколько быстро работает метод add? Создание объекта может занимать большое время, но у нас простой конструктор с двумя присваиваниями, поэтому работать он будет быстро.

Время добавления узла в начало всегда одно и то же и не зависит от размера списка, поэтому в данном случае речь идет об алгоритмической сложности .

Вставка элемента в середину или конец списка

Вставка в середину или конец — сложная операция. В отличие от массива, мы не можем сразу получить второй или пятый узел списка. Мы должны перебрать все узлы с начала, чтобы понять, куда вставить новое значение:

```
insert(index, value) {
  if (this.head === null) {
    this.head = new LinkedListNode(value, null);
  } else if (index === 0) {
    this.add(value);
  } else {
    let current = this.head;
    while (current.next !== null && index > 1) {
      current = current.next;
      index = index - 1;
    }

    current.next = new LinkedListNode(value,
current.next);
  }
}
```

Вызов insert(index, value) означает, что узел со значением value будет вставлен в середину списка в позицию index. Если index равен 0, то значение вставится перед первым элементом — так же, как при вызове add:

```
if (index === 0) {
  add(value);
}
```

Если список пустой, index не может быть больше нуля, потому что мы можем вставить значение только в начало списка. Как должна поступать в этом случае функция решает ее автор. Мы можем

генерировать ошибку или попытаться выполнить какое-то разумное действие.

Пойдем вторым путем — если список пустой, при больших значениях index будем вставлять элемент в самое начало:

```
if (this.head === null) {
  this.head = new LinkedListNode(value, null);
}
```

А в случае, если index оказывается за концом списка, будем вставлять элемент в самый конец. Для этого будем проверять два условия: либо мы нашли элемент с номером index, и он не в конце, либо мы достигли последнего элемента:

```
while (current.next !== null && index > 1) {
  current = current.next;
  index = index - 1;
}
```

Нарисуем, что должно происходить при вставке. Пусть в начале у нас есть список из элементов A и C. Мы хотим вставить элемент B после A, но перед C:

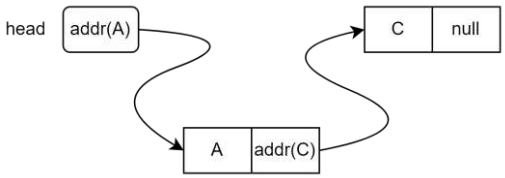


Рисунок 3. Процесс вставки элемента в список.

Когда цикл заканчивается, переменная current указывается на узел A:

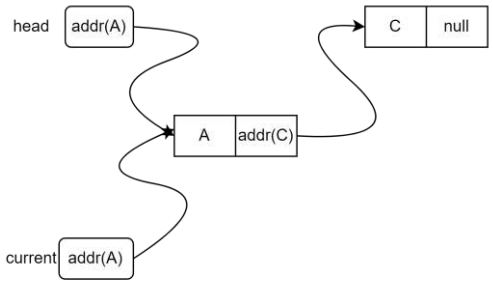


Рисунок 4. Конец цикла.

После вставки мы получим такую картину:

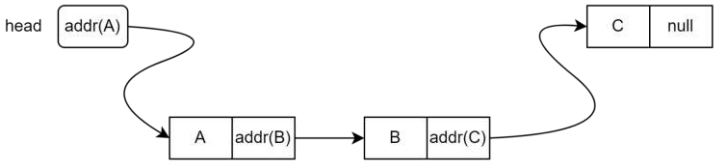


Рисунок 5. Окончание вставки элемента.

Перед вставкой в current.next хранится ссылка на C, но теперь она должна переехать в узел B, а в current.next мы запишем ссылку на новый узел B:

```
current.next = new LinkedListNode(value,
current.next);
```

Поиск элемента

При работе с массивом важное значение имеет индекс элементов. При работе со списком индекс практически не используется, хотя формально мы можем найти третий или семнадцатый узел списка.

Если при поиске элемента в массиве, функции часто возвращают индекс элемента, то для списка — логическое значение true или false.

Вместо ответа на вопрос «если такой элемент в массиве есть, где он находится», функция поиска в списке отвечает на вопрос «есть ли такой элемент в списке»:

```
contains(value) {
  let current = this.head;
  while (current !== null) {
    if (current.value === value) {
      return true;
    }

    current = current.next;
  }
  return false;
}
```

Здесь у нас простой цикл. Мы пробегаем по всем узлам, пока не натыкаемся на null. Если мы встретили null и не нашли значения, значит, в списке его нет — в конце цикла мы возвращаем false.

Если значение находится в одном из узлов, мы прерываем цикл и сразу возвращаем true:

```
if (current.value === value) {
  return true;
}
```

В конце цикла важно переходить к следующему узлу, иначе цикл станет бесконечным:

```
current = current.next;
```

Определение длины списка

В отличие от массива, длина списка нам неизвестна, ее нужно вычислить. Мы заводим счетчик и пробегаем по всем узлам списка, увеличивая его на каждой итерации. Длина пустого списка считается равной нулю.

В целом, код получается простой:

```
length() {
```

```
  let result = 0;
  let current = this.head;
  while (current !== null) {
    result = result + 1;
    current = current.next;
  }
  return result;
}
```

Переменная result — это наш счетчик. Переменная current на каждой итерации указывает на текущий узел списка. Когда она становится равной null, список пройден до конца. Перейдя к следующему узлу, мы увеличиваем счетчик, поэтому в конце цикла его значение равно количеству узлов.

Удаление элемента из начала списка

Удаление элемента из начала списка такое же простое, как и вставка. Мы будем возвращать значение из удаленного узла в качестве результата.

Если список пустой, мы не будем ничего удалять, и в качестве результата вернем undefined:

```
remove() {
  if (this.head === null) {
    return undefined;
  }
  const value = this.head.value;
  this.head = this.head.next;

  return value;
}
```

При удалении в this.head попадает ссылка на следующий узел из первого узла. Когда в списке остается один узел, там находится null. После удаления последнего узла this.head становится равным null, что для нас означает пустой список.

Нам не приходится как-то по-особому описывать этот случай в коде — наш код работает для списков любой длины.

Рисунок поможет разобраться, как удаляются узлы из списка:

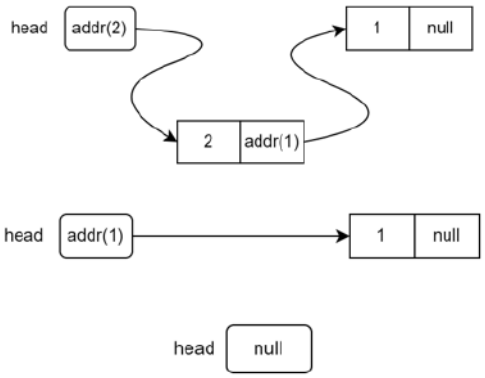


Рисунок 6. Процесс удаления узла из списка.

Удаление элемента из середины или конца списка

Теперь, когда мы умеем вставлять элементы в начало и середину списка, и удалять их из начала списка, удаление узлов из середины не должно представлять для нас проблемы.

Мы просто скомбинируем написанный нами ранее код:

```
removeAt(index) {
  if (this.head === null) {
    return undefined;
  } else if (index === 0 || this.head.next ===
null) {
    return this.remove();
  } else {
    let current = this.head;
    while (current.next.next !== null && index > 1)
{
      current = current.next;
      index = index - 1;
    }
    const value = current.next.value;
    current.next = current.next.next;
    return value;
  }
}
```

Единственное нововведение по сравнению с предыдущим кодом заклю в том, что при удалении узла из середины, нам нужно просматривать список на два элемента вперед. Если мы хотим удалить последний узел в списке, нам придется вносить изменения в предпоследний — менять там значение ссылки next.

Поэтому нам приходится, как особый случай, обрабатывать список из одного элемента. Впрочем, эту проверку мы можем совместить с проверкой на удаление из начала:

```
if (index === 0 || this.head.next === null) {
  return remove();
}
```

Меняется и условие в цикле. Если раньше мы проверяли, что current.next не равен null, то теперь мы проверяем current.next.next. Иными словами, вместо ответа на вопрос «есть ли следующий узел в списке», мы отвечаем на вопрос «есть ли узел за следующим узлом».

В остальном код остается прежним.

Контрольные вопросы

1. Что такое связный список?
2. Чем связный список отличается от массива?
3. Какие основные виды связных списков существуют?
4. Какие основные элементы содержит узел в односвязном списке?
5. Что такое голова (head) и хвост (tail) списка?
6. Как добавить элемент в начало односвязного списка?
7. Как добавить элемент в конец односвязного списка?
8. Как происходит удаление элемента из начала и конца связного списка?
9. Как осуществляется поиск элемента в связном списке?
10. Как вставить элемент в середину связного списка?

ТЕМА 9. ПРИНЦИП РЕКУРСИИ

Рекурсия является одним из наиболее мощных подходов в программировании. С ее помощью можно решать чрезвычайно сложные задачи, печатая при этом невероятно малый объем кода. Тем не менее понимание данного принципа нередко вызывает сложности, поскольку для этого требуется нестандартный взгляд на процесс выполнения команд.

Сложность примеров, описанных ниже, будет возрастать постепенно, начиная от самых простых и заканчивая достаточно трудными. При этом каждый из них будет сопровождаться подробной диаграммой:

- Рекурсивный способ мышления.
- Рекуррентные соотношения.
- Мемоизация.
- Стратегия “разделяй и властвуй”.

Рекурсия используется для решения задач, в которых функция вызывает сама себя в рамках собственного определения. В каждой реализации рекурсии должно присутствовать два элемента:

- Заключительный базовый кейс или кейсы, в которых рекурсивный поиск ответа более не производится.
- Набор правил (рекуррентных соотношений), который инициирует очередные циклы рекурсии, сводя таким образом общие кейсы к базовому.

В качестве примера давайте рассмотрим задачу по выводу развернутого варианта строки. В этом случае на выходе из вводного слова ‘hello‘ должно получиться ‘olleh’. Итеративный метод решения этой задачи — применение цикла for и вывод каждого знака, начиная с последнего и заканчивая первым.

Рисунок 1. Развёрнутая строка.

В рекурсивном же подходе мы сначала создаем функцию reverseString, получающую интересующую нас строку в качестве параметра. Если длина ввода не равна 0, то такой кейс является базовым или заключительным, и мы выводим последнюю букву, а затем инициируем еще один экземпляр reverseString для той же строки, исключив последнюю букву, поскольку ее мы уже вывели.

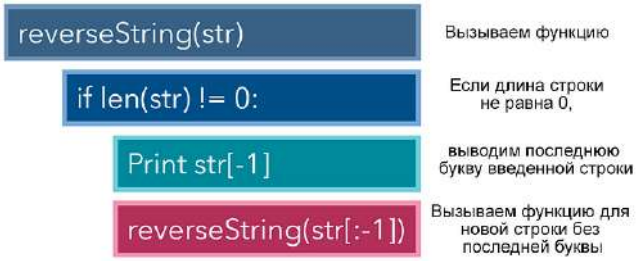


Рисунок 2. Рекурсивный подход вывода строки.

Обратите внимание: поскольку функция вызывается изнутри себя, она сама создает цикл for. Кроме того, наличие инструкции if перед вызовом другого экземпляра функции обязательно, в противном случае будет выброшена ошибка RecursionError или RuntimeError, поскольку скрипт не увидит конца бесконечного цикла. Данная ситуация аналогична бесконечному циклу While True.

Давайте посмотрим, как эта рекурсивная функция работает с ‘hello‘:

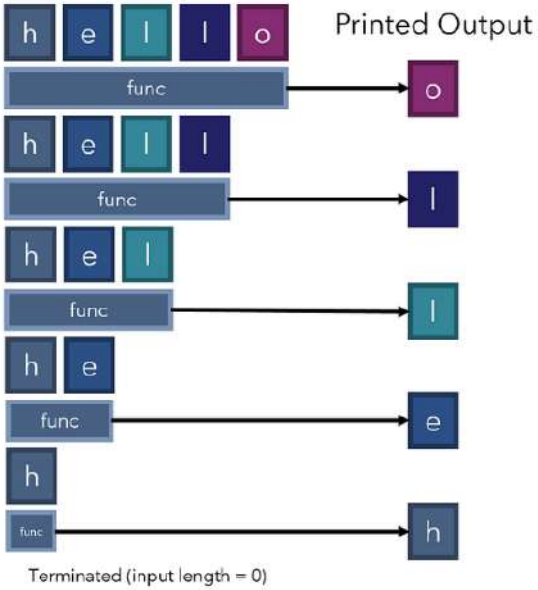


Рисунок 3. Принцип работы рекурсивной функции.

Рекуррентность в более сложных задачах может вызвать трудности при определении двух ее компонентов — рекуррентного соотношения, т.е. соотношения между результатом задачи и результатом ее подзадач и базовым кейсом, представляющим кейс, который можно вычислить напрямую без дополнительных рекурсивных вызовов. Иногда базовые кейсы называются “нижними кейсами”, потому что они являются кейсами, в которых задача уменьшена до наименьшего масштаба.

Рассмотрите в качестве примера треугольник Паскаля, в котором каждое число является суммой двух, находящихся над ним, и который имеет по сторонам единицы. Как можно использовать рекурсию для нахождения значения любого значения в точке (i, j)? Каково будет рекуррентное соотношение и базовый/заключительный кейс?

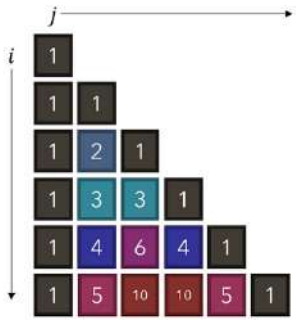


Рисунок 4. Треугольник Паскаля.

Рекуррентное соотношение можно выразить следующим уравнением:

$$v(i,j) = v(i-1,j-1) + v(i-1,j)$$

Это очевидно, когда смотришь на график треугольника (рис.4). Еще более примечательно в этой формуле то, что если мы продолжим с ее помощью разбивать любую точку (i, j) на сумму двух других точек, то в итоге неизбежно получим базовый кейс — 1. Треугольник Паскаля начинается с единиц, и из их сумм строится весь сложный паттерн.

Реализация программы

Для начала давайте найдем набор правил, чтобы определять, когда выполняется базовый кейс, при котором значение ячейки равняется 1. Обратите внимание, что 1-цы появляются при выполнении одного из двух условий: когда располагаются либо в первом столбце (j=0), либо по диагонали (i=j) смотри рисунок 4.

Теперь выполнить реализацию просто (рис.5). Если условия для базового кейса выполнены, мы возвращаем базовое значение (1). В противном случае мы продолжаем уменьшать задачу до тех пор, пока не достигнем базового кейса, до которого, как мы определили, будет неизбежно разбит любой ввод.

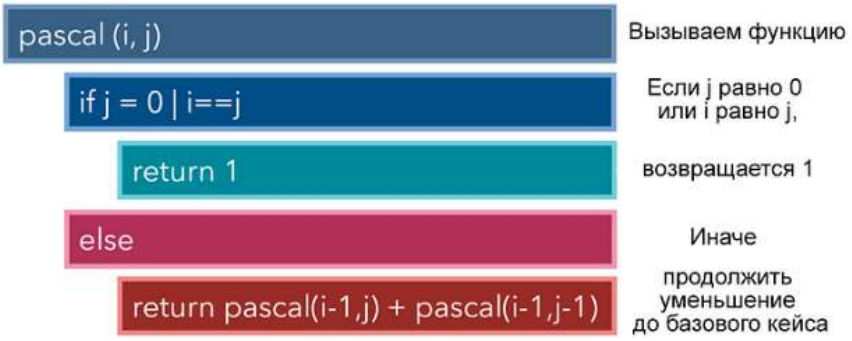


Рисунок 5. Программная реализация треугольника Паскаля.

Теперь рекурсия раскрылась для вас во всей красе. Здесь, с помощью всего пяти строк кода, мы, по сути, создаем саморазвивающееся дерево. При желании число строк кода можно даже уменьшить до трех. Вызывая функцию pascal дважды, мы инициируем две ветви поиска,

каждая из которых также инициирует еще две, предполагая, что базовый кейс достигнут не был.

Такая эффективность рекурсии может показаться магической и даже может запутать. Давайте рассмотрим работу ее алгоритма на примере (рис.6).

В соответствии с нашим рекуррентным соотношением (4, 2) разбивается на (3, 1) и (3, 2), которые являются двумя числами, стоящими над ней. Обратите внимание, что алгоритм на деле не знает, что значения этих ячеек представлены 3. Он просто учитывает их местоположение. Мы не знаем или даже не интересуемся никакими значениями, пока выполняются базовые условия. На основе базовых кейсов (1) мы можем вычислить другие не базовые точки, но для начала должны быть найдены все базовые.

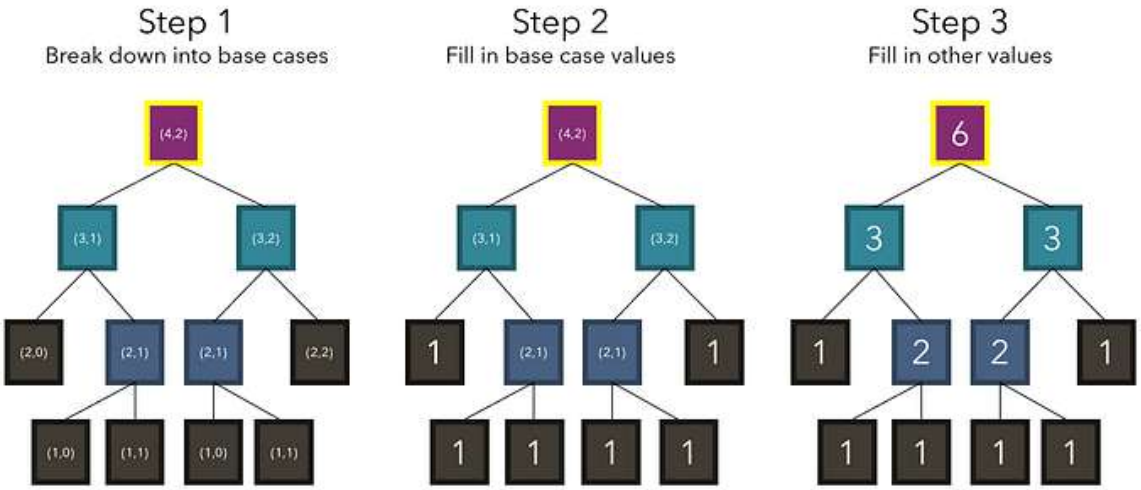


Рисунок 6. Пример работы рекурсивного алгоритма.

Согласно нашему рекуррентному соотношению, кейсы итеративно разбиваются до тех пор, пока не достигается базовый (j = 0 или i = j). Поскольку нам известны значения этих базовых кейсов (1), мы можем заполнить и другие значения, зависимые от базового кейса (рис.6).

Это, конечно же, очень, а возможно и чересчур подробное представление принципа работы рекурсивного алгоритма. В действительности ни один из этих трех шагов не нужно программировать, так как они выполняются скриптом автоматически. Все, что вам нужно, — это вызвать функцию внутри ее самой и обеспечить ее завершение в некоторых точках при достижении базового случая.

При вызове return pascal(i-1, j) + pascal(i-1, j-1) мы рассматриваем возврат не как процесс, а как число. Поскольку pascal(i-1, j) инициирует собственные процессы ветвления, а в итоге возвращает другое число (например, 3), будет правильным воспринимать его именно как число, а

не как процесс, что может вызвать ненужную сложность и затруднение в понимании.

С некоторой точки зрения этот рекурсивный алгоритм можно назвать неэффективным. В конце концов ‘6' разбивается на ‘3', которое, с позиции значения, имеет идентичные разбивки, но при этом зачем-то вычисляется второй раз. Это стандартный случай в рекурсии, когда базовые кейсы одного кейса, будучи вычисленными ранее, вычисляются повторно. Для устранения этой проблемы мы используем мемоизацию.

Возьмите в качестве примера последовательность Фибоначчи (рис.7), в которой первые два числа представлены 0 и 1, а последующие числа являются суммой двух, им предшествующих. На основе уже сформированного знания мы понимаем, что базовыми кейсами здесь будут 0 и 1, а рекуррентное соотношение будет выглядеть как $v(i) = v(i\text{-}2) + v(i\text{-}1)$. В таком случае мы можем построить простую рекурсивную функцию для нахождения значения последовательности Фибоначчи в любом индексе, начиная с 0.

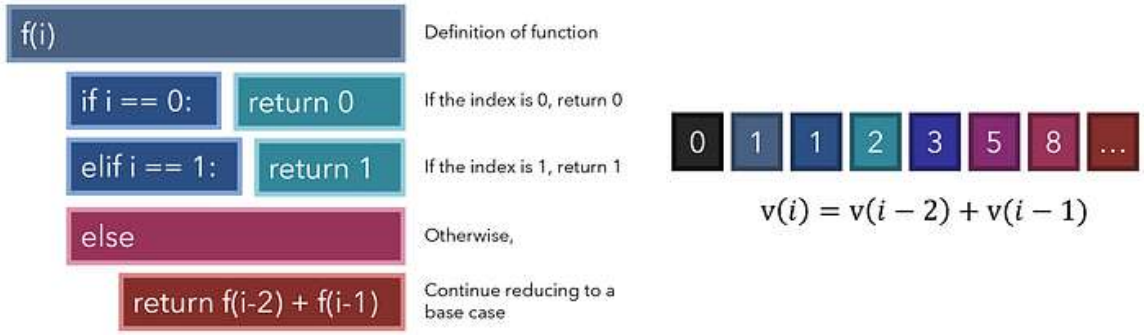


Рисунок 7. Алгоритм последовательности Фибоначчи.

Обратите внимание, что хоть это и грамотное применение рекурсии, оно весьма неэффективно. 8 разбивается на 3 и 5, а 5, в свою очередь, разбивается на 3 и 2. Мы вычисляем все с самого начала и строим завершенное дерево поиска со множеством повторений.

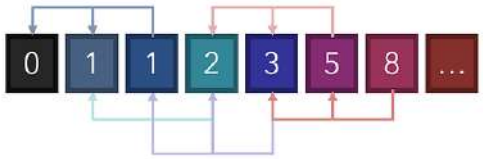


Рисунок 8. Принцип мемоизации.

С помощью мемоизации мы можем решить эту проблему, создав кэш. Это можно реализовать, используя словарь и сохраняя значения, заданные ранее. Например, когда индекс 6 (значение 8) разбивается на индекс 4 и индекс 5 (значения 3 и 5), мы можем сохранить значение индекса 4 в кэше. При вычислении индекса 5 как индекса 3 плюс индекс 4 мы можем извлечь индекс 4 из кэша вместо того, чтобы повторно вычислять его, создавая еще одно обширное дерево, ведущее к базовым кейсам (рис.8).

Чтобы включить в нашу функцию мемоизацию, мы добавим две функциональности: во-первых, если текущий индекс был сохранен в кэше, мы будем просто возвращать его сохраненное значение; во-вторых, прежде чем продолжать уменьшать значение, мы будем добавлять это значение к кэшу,чтобы ускорить дальнейшие операции. Обратите внимание, что кэш должен быть либо глобальной переменной, либо такой, которую можно извлечь и изменить независимо от области вызова команды.

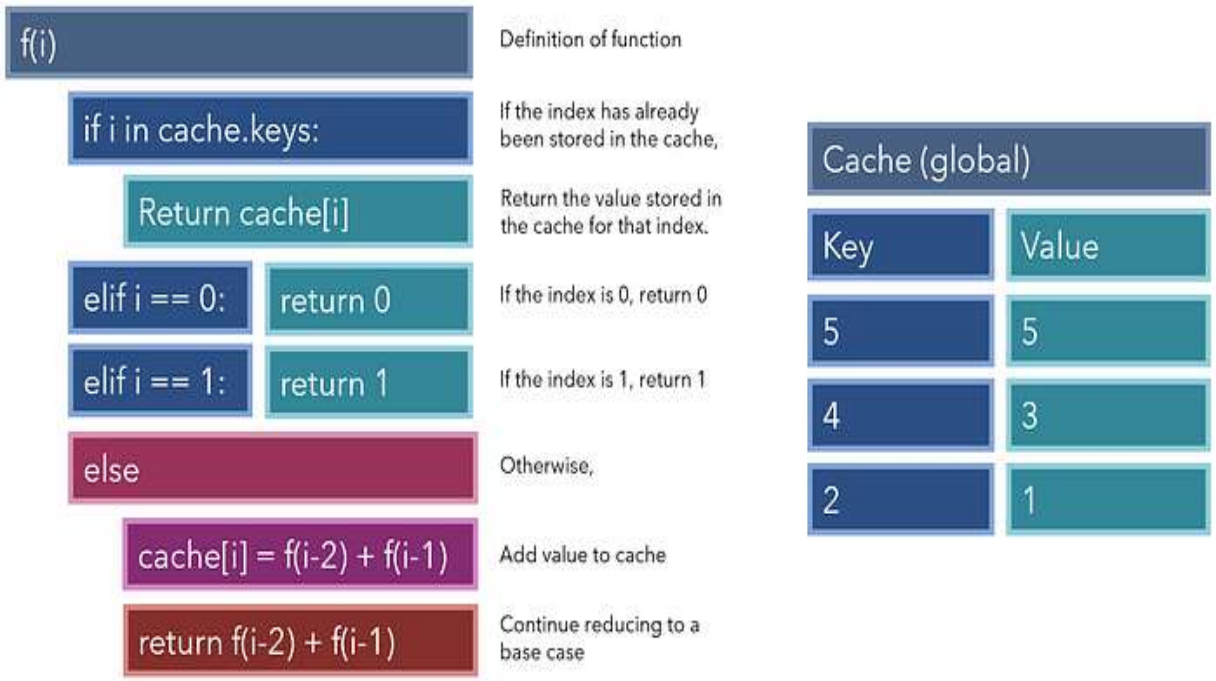


Рисунок 9. Мемоизация рекурсивной функции.

После добавления мемоизации наша рекурсивная функция стала намного быстрее и мощнее (рис.9).

Рекурсия занимает центральное место среди многих наибыстрейших алгоритмов сортировки. Цель этих алгоритмов — получить список чисел и вернуть их в порядке от меньшего к большему. Поскольку очень много приложений опираются на быструю сортировку, весьма важно найти такой, который сможет сортировать списки максимально быстро. При этом некоторые из самых быстрых алгоритмов используют рекурсивный подход, называемый “разделяй и властвуй”.

В стратегии “разделяй и властвуй” изначальная задача рекурсивно разбивается на несколько подзадач (рис.10). После того, как каждая подзадача достигает размера единицы (аналог базового кейса), выявляются их подрешения, которые вновь рекурсивно совмещаются для формирования окончательного решения.

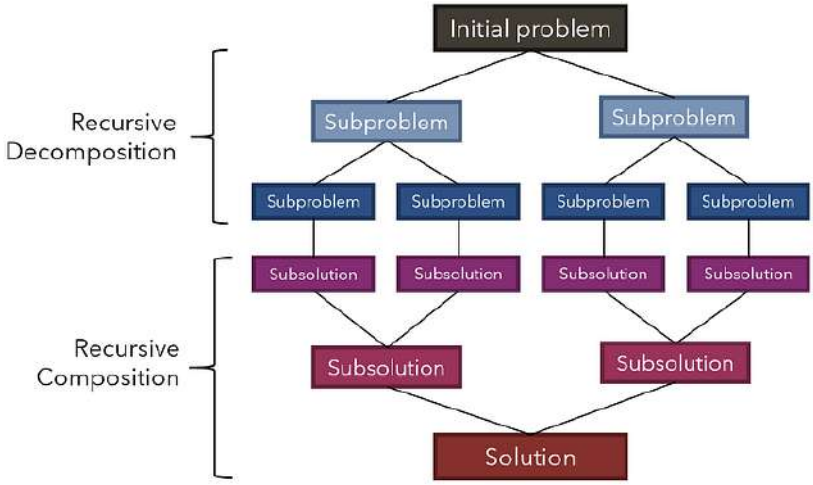


Рисунок 10. Стратегия “разделяй и властвуй”.

Рассмотрите, к примеру, QuickSort — один из наиболее быстрых алгоритмов сортировки, который при грамотной реализации может выполняться в 2–3 раза быстрее своих соперников и предшественников.

QuickSort начинает выполнение с выбора “опоры” (рис.11). В простейших реализациях и для наших целей такую опору можно выбрать произвольно. Однако в более специализированных реализациях к ее выбору уже стоит подходить осторожно.

Рисунок 11. Выбор опоры списка.

Все элементы со значениями меньше опоры переносятся левее нее, а все, чьи значения больше — правее. Выполнить это можно, пройдя по каждому элементу и переместив те, что меньше опоры, в один список, а те, что больше нее — в другой. Заметьте, что эта операция справляется с задачей только частично.

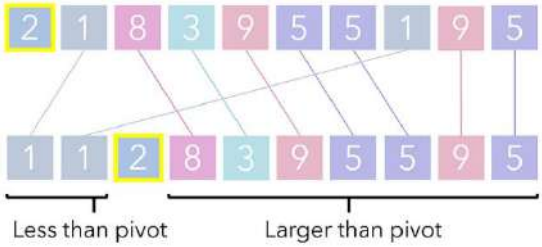


Рисунок 12. Разделение списка.

Процесс сортировки списка на основе его опорной точки называется разделением (рис.12), так как опора разделяет этот список на два раздела, иначе говоря, на две стороны. Каждый их этих разделов вызывает очередную итерацию разделения сам на себя и так продолжается до тех пор, пока раздел не достигнет базового кейса (1 единицы, просто одного числа) рисунок 13.

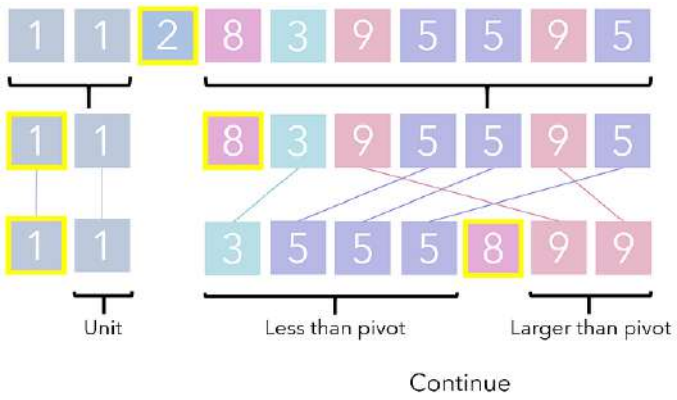


Рисунок 13. Получение базового кейса.

После достаточного числа рекурсивных вызовов исходный список будет разделен до точки, когда эти вызовы продолжить уже невозможно. В этой точке композиция подрешений выполняется простым составлением их в горизонтальном порядке. Результат такой композиции — отсортированный список.

Обратите внимание, что QuickSort следует многим из перечисленных основ рекурсии, рассмотренных нами ранее, только на более высоком уровне. Рекуррентное соотношение является разделением компонента, а базовый кейс — это разделение размера 1. Начиная с оригинального списка, те же процессы (выбор опоры и разделение) вызываются рекурсивно до тех пор, пока результат не будет состоять только из базовых кейсов.

Контрольные вопросы

1. Что такое рекурсия и как она работает?
2. Какие основные компоненты рекурсивного алгоритма?
3. Что такое базовый случай в рекурсии, и почему он важен?
4. Как рекурсивные вызовы работают в стеке вызовов?
5. Как избежать бесконечной рекурсии?
6. В чем разница между прямой и косвенной рекурсией?
7. Какие задачи лучше всего решаются с использованием рекурсии?
8. Что такое мемоизация?
9. Какова временная сложность рекурсивного алгоритма по сравнению с итеративным?
10. Приведите пример рекурсивного алгоритма (например, факториал или последовательность Фибоначчи).

ТЕМА 10. СТРУКТУРА ДАННЫХ ДЕРЕВО

Организация хранения данных в виде **дерева** позволяет обойти ограничения линейной структуры данных. Например, в последней нельзя организовать быстрыми поиск и вставку элементов одновременно. При этом в иерархической структуре данных можно эффективно выбирать и обновлять большие объемы данных.

Один из наиболее часто используемых и простых в реализации подвидов деревьев — бинарные деревья. Помимо организации поиска, бинарные деревья используют, когда разбирают математические выражения и компьютерные программы. Еще их используют, чтобы хранить данные для алгоритмов сжатия, а также они лежат в основе других структур данных, например, очереди с приоритетом, кучи и словари.

Эта тема познакомит нас с устройством и особенностями бинарных деревьев и разберем основные операции с его узлами.

Бинарное дерево или **двоичное дерево** — это дерево, в котором у каждого из его узлов не более двух дочерних узлов. При этом каждый дочерний узел тоже представляет собой бинарное дерево.

Рассмотрим примеры деревьев на рисунке 1:

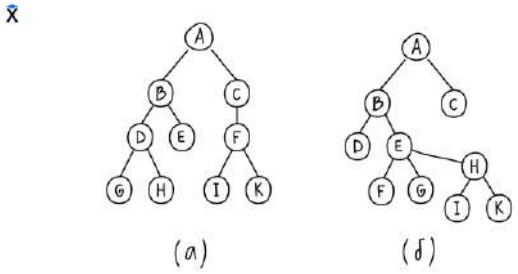


Рисунок 1. а) бинарное дерево; б) небинарное дерево.

Дерево (а) — бинарное. У каждого его узла не более двух дочерних узлов, у каждого из которых тоже не более двух дочерних. Например, узлы E, G, H, I и K — листовые, значит, у них ноль дочерних узлов. У узла C только один дочерний узел, а у узлов A, B, D и F по два дочерних.

Как только правило двух дочерних нарушается, то дерево перестает относиться к классу бинарных. Так, дерево (б) не является бинарным, так как у узла E три дочерних узла.

Благодаря тому, что дочерних узлов всегда не больше двух, их называют **правый** и **левый** дочерние узлы.

Напомним, что есть завершенное и полное деревья. Для бинарных деревьев они приобретают следующий вид:

- **Завершенное бинарное дерево** — это бинарное дерево, в котором каждый уровень, кроме последнего, полностью заполнен, а заполнение последнего уровня производится слева направо

- **Полное бинарное дерево** — это бинарное дерево, в котором у каждого узла ноль или два дочерних узла

На практике чаще применяются два подвида бинарных деревьев: бинарные деревья поиска и бинарные кучи.

Что такое бинарные деревья поиска

Бинарные деревья поиска отличаются от обычных бинарных деревьев тем, что хранят данные в отсортированном виде. Хранение значений внутри бинарного дерева поиска (рис.2) организовано в следующем виде:

- Все значения в узлах левого дочернего поддерева меньше значения родительского узла
- Все значения в узлах правого дочернего поддерева больше значения родительского узла
- Каждый дочерний узел тоже является бинарным деревом поиска

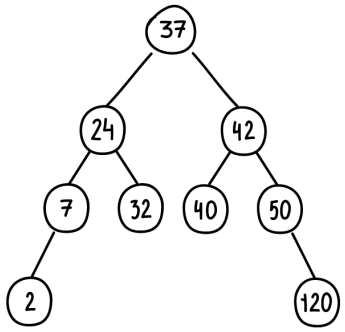


Рисунок 2. Бинарное дерево поиска.

Благодаря такой структуре хранения данных поиск узла в бинарном дереве поиска занимает меньше времени. Это значительно меньше, если хранить значения в списках.

Если использовать отсортированный массив для хранения данных, скорость поиска элементов сравняется. Но при оценке времени вставки хранение в массиве значительно проигрывает работе с деревьями - против соответственно.

Такая высокая эффективность поиска в бинарном дереве поиска наблюдается только при сохранении его в сбалансированном состоянии — когда все уровни, кроме последнего полностью заполнены. Это значит, что любое добавление или удаление вершины может потребовать полное перестроение дерева.

Рассмотрим на примере поиска элемента (10) сравнение операций поиска в отсортированном массиве, списке и бинарном дереве поиска:

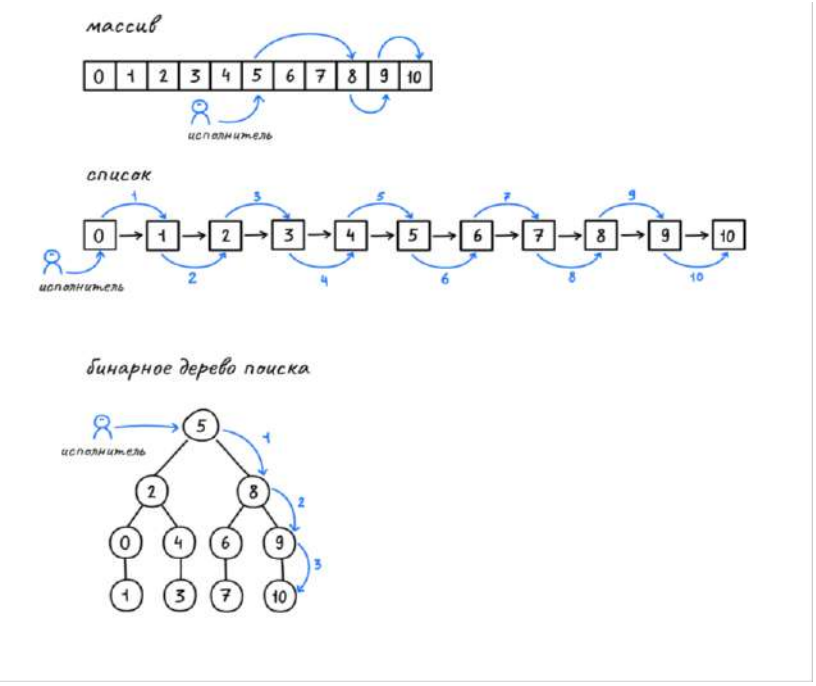


Рисунок 3. Пример поиска элемента (10) в списке с помощью бинарного дерева поиска.

Для поиска в массиве применяется традиционный подход на основе половинного деления — метод дихотомии. На схеме можно видеть что аналогичная операция и на массиве, и на бинарном дереве поиска будет выполнена за три шага. В то же время поиск этого элемента на списке займет десять шагов.

Далее поговорим о структуре бинарных деревьев и начнем реализовывать бинарное дерево в коде.

Код реализации бинарного дерева поиска

Напомним свойства бинарных деревьев:

- Должно быть не более двух дочерних узлов
- Дочерние узлы тоже должны быть бинарными деревьями
- Дочерние узлы называют левыми и правыми

В этом случае структура узла принимает следующий вид:

```
class BinaryTreeNode {
  constructor(value, parent) {
    this.left = null; //ссылка на левый дочерний
узел
    this.right = null; //ссылка на правый дочерний
    this.parent = parent; //ссылка на родителя
    this.value = value; //полезная нагрузка
  }
}
```

Теперь нам необходимо расширить наш класс и реализовать необходимые операции, чтобы взаимодействовать с проектируемым классом. Начнем с операции поиска узла.

С бинарными деревьями поиска можно выполнять следующие операции:

- Искать узел
- Вставлять узел
- Удалять узел
- Выполнять обход дерева

Поиск узла

Если искомое значение бинарного дерева поиска меньше значения узла, то оно может находиться только в левом поддереве. Искомое значение, которое больше значения узла, может быть только в правом поддереве. В таком случае мы можем применить рекурсивный подход и операция поиска будет выглядеть так:

```
class BinaryTreeNode {
    // ...

    findNode(value){
        let node = this;
        while (node){
            if (value == node.value) return node;
            if (value < node.value) node =
node.left;
            if (value > node.value) node =
node.right;
        }

        return null;
    }
}
```

Вставка узла

Все значения меньше текущего значения узла надо размещать в левом поддереве, а большие — в правом. Чтобы вставить новый узел, нужно проверить, что текущий узел не пуст. Далее может быть два пути:

- Если это так, сравниваем значение со вставляемым. По результату сравнения проводим проверку для правого или левого поддеревьев
- Если узел пуст, создаем новый и заполняем ссылку на текущий узел в качестве родителя

Операция вставки использует рекурсивный подход аналогично операции поиска. Переведем данный алгоритм на язык JavaScript и получим следующий код метода вставки:

```
class BinaryTreeNode {
    // ...

    insertNode(value){
        return this.#insertNode(value, this)
      }

      #insertNode(value, parentNode){
        if (value < parentNode.value){
          if (parentNode.left == null){
            parentNode.left = new
BinaryTreeNode(value, parentNode);
          } else {
            this.#insertNode(value,
parentNode.left);
          }
        }
        if (value > parentNode.value){
          if (parentNode.right == null){
            parentNode.right = new
BinaryTreeNode(value, parentNode);
          } else {
            this.#insertNode(value,
parentNode.right);
          }
        }
      }
}
```

Удаление узла

Чтобы удалить элемент в связном списке, нужно найти его и ссылку на следующий элемент перенести в поле ссылки на предыдущем элементе.

Если необходимо удалить корневой узел или промежуточные вершины и сохранить структуру бинарного дерева поиска, выбирают один из следующих двух способов:

- Находим и удаляем максимальный элемент левого поддерева и используем его значение в качестве корневого или промежуточного узла
- Находим и удаляем минимальный элемент правого поддерева и используем его значение в качестве корневого или промежуточного узла

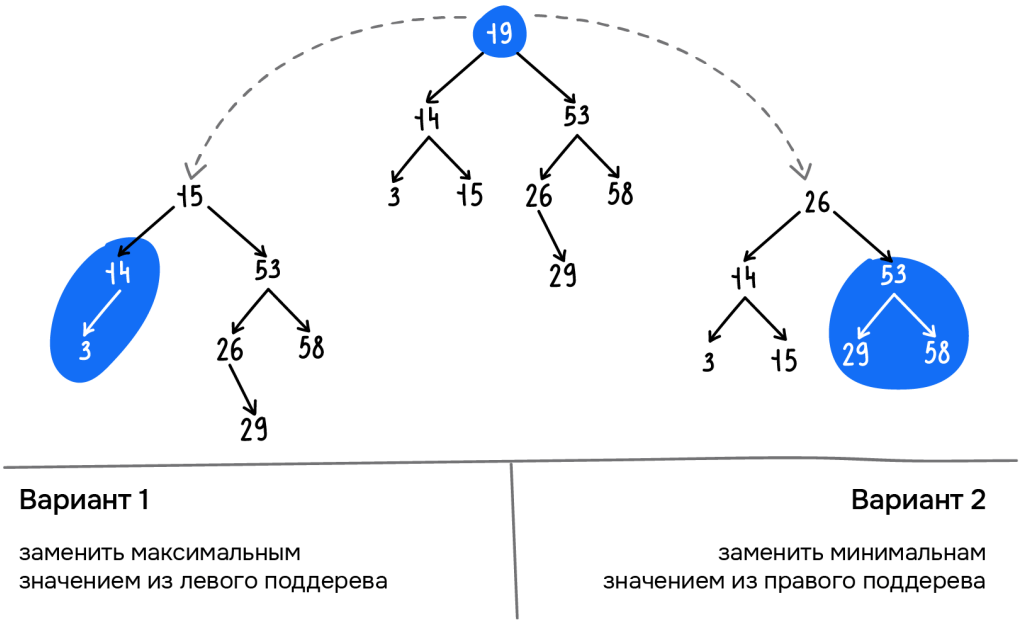


Рисунок 4. Удаление узлов бинарного дерева.

Оба варианта приемлемы для нашего дерева. Реализуем в коде второй вариант:

```
class BinaryTreeNode {
    // ...

    removeNode(value) {
        return this.#removeNode(value, this);
    }
    #removeNode(value, node) {
        if (node == null) return null;
        if (value < node.value) {
            node.left = this.#removeNode(value,
node.left);
        } else if (value > node.value) {
            node.right = this.#removeNode(value,
node.right);
        } else {
            if (node.left == null) return
node.right;
            if (node.right == null) return
node.left;
        }

        let original = node;
```

```
            node = node.right;
            while (node.left) {
                node = node.left;
            }

            node.right =
this.#removeMin(original.right);
            node.left = original.left;
        }
        #removeMin(node) {
            if (node.left === null) {
                return node.right;
            }
            node.left = this.#removeMin(node.left);
            return node;
        }
}
```

Реализация первого варианта будет выглядеть практически идентично. Только есть исключение: мы будем обходить правое поддерево и искать максимальное значение узла вместо минимального.

Для деревьев также существуют специфические операции, важнейшая из которых — обход дерева. Рассмотрим эту операцию подробнее.

Обход деревьев

Когда мы поработали с деревом и нам нужно его сохранить в файл или вывести в печать, нам больше не нужен древовидный формат. Здесь мы прибегаем к **обходу дерева** — последовательное единоразовое посещение всех вершин дерева.

Существуют такие три варианта обхода деревьев:

- **Прямой обход (КЛП)**: корень → левое поддерево → правое поддерево
- **Центрированный обход (ЛКП)**: левое поддерево → корень → правое поддерево
- **Обратный обход (ЛПК)**: левое поддерево → правое поддерево → корень

Такие обходы называются **поиском в глубину**. На каждом шаге итератор пытается продвинуться вертикально вниз по дереву перед тем, как перейти к родственному узлу — узлу на том же уровне. Еще есть **поиск в ширину** — обход узлов дерева по уровням: от корня и далее:

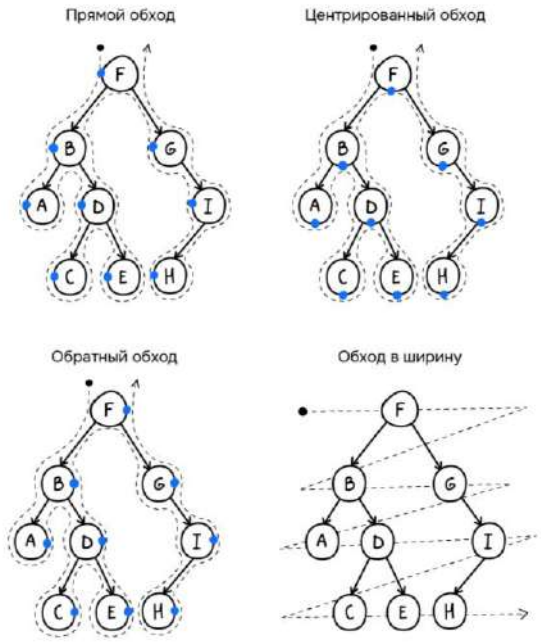


Рисунок 5. Варианты обхода бинарного дерева.

Реализация поиска в глубину может осуществляться или с использованием рекурсии, или с использованием стека. А поиск в ширину реализуется за счет использования очереди:

```
class BinaryTreeNode {
    // ...

    traverseRecursive(node) {
        if (node != null) {
            console.log(`node = ${node.val}`);
            traverseRecursive(node.left);
            traverseRecursive(node.right);
        }
    }
    traverseWithStack() {
        let stack = [];
        stack.push(this);
        while (stack.length > 0) {
            let currentNode = stack.pop();

            console.log(`node =
${currentNode.val}`);
            if (currentNode.right != null) {
                stack.push(currentNode.right);
            }
```

```
            if (currentNode.left != null) {
                stack.push(currentNode.left);
            }
        }
    }

    traverseWithQueue() {
        let queue = [];
        queue.push(this.root);
        while (queue.length > 0) {
            let currentNode = queue.shift();
            console.log(`node =
${currentNode.val}`);
            if (currentNode.left) {
                queue.push(currentNode.left);
            }
            if (currentNode.right) {
                queue.push(currentNode.right);
            }
        }
    }
}
```

Контрольные вопросы

1. Что такое дерево в контексте структур данных и какова его основная структура?
2. Какие основные термины используются при описании дерева (узел, корень, лист, высота)?
3. В чем разница между бинарным деревом и деревом с более чем двумя дочерними узлами?
4. Что такое бинарное дерево поиска и как оно устроено?
5. Какие операции можно выполнять с деревьями (добавление, удаление, обход)?
6. Каковы различные способы обхода дерева (прямой, обратный, уровень)?
7. Что такое сбалансированное дерево, и почему важно поддерживать баланс?
8. Как можно реализовать дерево с помощью связных списков?
9. В чем различия между деревьями AVL и красно-черными деревьями?
10. Какие области применения имеют деревья в программировании и компьютерных науках?

ТЕМА 11. ОЦЕНКА СЛОЖНОСТИ АЛГОРИТМА. ПОНЯТИЕ BIG «O»

«О» большое (Big O notation) — это математическая нотация, которая позволяет оценить, как изменяется время выполнения алгоритма или объем используемой памяти в зависимости от размера входных данных. Есть еще «о» малое — эта нотация дает более строгую верхнюю границу для сложности алгоритма, но часто ее труднее вычислить, чем «О» большое. На практике «О» большое используется чаще, поскольку эта концепция проще для анализа и дает достаточно хорошую оценку сложности в большинстве случаев.

Зачем нужно «О» большое

Понимание этой нотации позволяет:

- Сравнивать эффективность различных алгоритмов для решения одной и той же задачи.
- Прогнозировать поведение алгоритма при увеличении размера входных данных.
- Оптимизировать код путем идентификации и улучшения сложных алгоритмов.
- Выбирать оптимальные структуры данных и алгоритмы при решении ресурсоемких задач.

Эффективность алгоритмов: временная и пространственная сложность

Обычно эффективность алгоритмов оценивается по двум критериям:

1. **Временная сложность** — это количество операций, которые должен выполнить алгоритм. Она показывает, как растет время выполнения алгоритма при увеличении входных данных.
2. **Пространственная сложность** алгоритма измеряет объем памяти, которую он использует в зависимости от размера входных данных. На пространственную сложность влияют несколько факторов, включая количество переменных, тип и размер структуры данных, вызовы функций и способ выделения памяти.

Как «О» большое описывает эффективность алгоритмов

Итак, «О» большое — это способ оценить, как время выполнения алгоритма или требуемый объем памяти растут с увеличением размера входных данных.

Пример: предположим, что стоматолог тратит 30 минут на прием одного пациента. По мере увеличения очереди пациентов, время, необходимое для приема всех пациентов, будет расти линейно пропорционально количеству пациентов в очереди.

Учитывая это, мы можем охарактеризовать эффективность стоматолага как линейную (рис.1). В нотации большого «О» эта эффективность выглядит как **O(n),** где **n** равно количеству пациентов, а

время, необходимое стоматологу для завершения работы, растет линейно или пропорционально количеству пациентов:

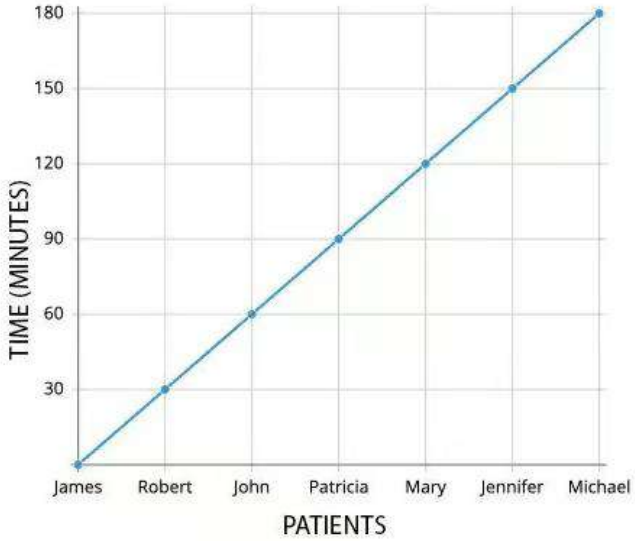


Рисунок 1. Линейная функция эффективности стоматолога.

Эффективность стоматолога можно описать этой функцией:

```
        var time = 0;
const patients = ['James', 'Robert', 'John',
'Patricia', 'Mary', 'Jennifer', 'Michael'];

function calculateTime() {
  for(let i = 0; i < patients.length; i++){
    time += 30;
  }
  console.log("Total Time 👉 ", time);
}
```

Здесь функция calculateTimeслужит простым примером для иллюстрации концепции большого «O» и потому не выполняет вычисления, умножая пациентов на затраченное на них время. Временная сложность этой функции **O(n)**, а сложность вывода в консоль **O(1)**:

```
        function calculateTime() {
  for(let i = 0; i < patients.length; i++){ // 👉
O(n)
    time += 30; // 👉 O(1)
  }
  console.log("Total Time 👉 ", time); // 👉 O(1)
}
```

Разберемся, как именно можно определить сложность любого алгоритма.

Основные правила вычисления большого «О»

Три главных правила вычисления большого «О» включают:

1. Определение худшего случая.
2. Исключение постоянных величин.
3. Игнорирование всех процессов, которые выполняются быстрее (или затрачивают меньше памяти), чем самая медленная и тяжелая часть алгоритма.

- **Лучший случай** показывает, как быстро алгоритм выполняется для определенного входного значения. В примере с функцией findJoey, которая ищет Joey в массиве друзей, лучший случай происходит, когда Joey находится в начале массива. В этом случае алгоритм выполнится всего один раз, поскольку сразу же найдет имя и прервет цикл с помощью оператора break. Таким образом, время выполнения функции в лучшем случае составляет **O(1)**, что означает константное время выполнения.

```
        let friends = ['Joey', 'Rachel', 'Phoebe',
'Ross', 'Monica', 'Chandler'];

function findJoey(){
  for(let i = 0; i < friends.length; i++){
    if(friends[i] === 'Joey'){
      console.log("Found!");
      break;
    }
  }
}
```

- **Средний случай** представляет собой прогноз о времени выполнения алгоритма, когда входные данные являются случайными. В функции findJoey, где Joey может находиться в любом месте массива, средний случай будет зависеть от того, где именно находится Joey. Если Joey находится в середине массива, алгоритм должен будет пройти половину массива, чтобы найти его, что приведет к времени выполнения **O(n/2)**, где **n** — количество элементов в массиве.

```
        let friends = ['Rachel', 'Phoebe', 'Ross',
'Joey', 'Monica', 'Chandler'];

function findJoey(){
  for(let i = 0; i < friends.length; i++){
    if(friends[i] === 'Joey'){
      console.log("Found!");
      break;
```

```
    }
  }
}
```

- **Наихудший случай** означает, как долго алгоритм может выполниться для предоставленного входного значения. В примере с функцией findJoey, где Joey находится в конце массива, алгоритму придется пройти через весь массив, чтобы найти имя. В этом случае время выполнения алгоритма составляет **O(n)**, где **n** — количество элементов в массиве.

Анализ худшего случая позволяет оценить максимальное время выполнения алгоритма.

Исключение постоянных величин

При расчете сложности алгоритма нет необходимости учитывать константы. Для примера снова вернемся к функции findJoey, которая ищет Joey в массиве друзей:

```
let friends = ['Rachel', 'Phoebe', 'Ross', 'Monica', 'Chandler', 'Joey'];

function findJoey(){
  for(let i = 0; i < friends.length; i++){ // O(n)
    console.log("Step 👉", i + 1); // O(1)
    if(friends[i] === 'Joey'){ // O(1)
      console.log("Found!"); // O(1)
    }
  }
}
```

Просуммируем сложность выполнения отдельных шагов:

```
Сложность = O(n) + O(1) + O(1) + O(1)
          = O(n) + O(3)
          = O(n + 3)
```

После упрощения сложность равна **O(n+3)**. Однако, при расчете эффективности функции, время выполнения константных операций не имеет значения. Это связано с тем, что даже если массив будет очень большим, например, содержать 2 миллиона элементов, эти константные операции не повлияют на общую эффективность функции, так как основной объем работы все равно будет заключаться в итерации по массиву:

```
Реальная сложность = O(n + 3)
                   = O(n)
```

Примечание: если функция состоит только из операций с константами, то ее сложность будет константной. Например,

функциясteps, которая выводит три уведомления, имеет реальную сложность **O(3)**. Но по правилам большого «O» мы определяем ее сложность как **O(1)**, так как предполагаем, что она всегда будет выполняться за одинаковое время:

```
        function steps(){
  console.log("Step 1"); // O(1)
  console.log("Step 2"); // O(1)
  console.log("Step 3"); // O(1)
}
```

Игнорирование всех недоминирующих процессов

С увеличением времени выполнения и/или объема потребления памяти эффективность алгоритмов снижается. В нотации большого «O» эта иерархия выглядит так:

Рисунок 2. Иерархия алгоритмов в нотации Big «O».

Возрастание сложности алгоритмов

При оценке алгоритма его итоговой сложностью является сложность самого медленного шага. Поэтому при оценке этой функции мы считаем ее сложностью **O(n)**, и игнорируем процессы **O(1)**:

```
        let friends = ['Rachel', 'Phoebe', 'Ross',
'Monica', 'Chandler', 'Joey'];
function findJoey(){
  for(let i = 0; i < friends.length; i++){ // O(n)
    console.log("Step 👉", i + 1); // O(1)
    if(friends[i] === 'Joey'){ // O(1)
      console.log("Found!"); // O(1)
    }
  }
}
```

Основные классы сложности алгоритмов

Классов сложности алгоритмов много, здесь мы рассмотрим только самые основные:

- **O(1) — константное время**. Это означает, что время выполнения алгоритма остается постоянным, независимо от размера входных данных.
  Пример: доступ к элементу массива по индексу.
- **O(log n) — логарифмическое время**. Время выполнения алгоритма увеличивается логарифмически по отношению к размеру входных данных.
  Пример: бинарный поиск в отсортированном массиве.
- **O(n) — линейное время**. Время выполнения алгоритма прямо пропорционально размеру входных данных.
  Пример: итерация по массиву.
- **O(n log n) — линейно-логарифмическое время**. Время выполнения алгоритма растет растет пропорционально произведению размера входных данных на логарифм этого размера.
  Пример: сортировка слиянием.
- **O(n^2) — квадратичное время**. Время выполнения алгоритма растет как квадрат размера входных данных.
  Пример: вложенный цикл, где каждый элемент массива сравнивается с каждым другим элементом.
- **O(2^n) — экспоненциальное время.** Время выполнения алгоритма растет экспоненциально по отношению к размеру входных данных.
  Пример: рекурсивный алгоритм, который решает задачу путем разделения ее на подзадачи.
- **O(n!) — факториальное время.** Время выполнения алгоритма растет как факториал размера входных данных.
  Пример: задача коммивояжера, где необходимо найти самый короткий маршрут, проходящий через все города.
- **O(m+n) и O(mn)**. В первом случае время выполнения алгоритма растет линейно по отношению к размеру входных данных, а во втором случае время выполнения будет квадратичным.
  Пример: линейный поиск в двумерном массиве имеет сложность **O(mn),** а объединение двух отсортированных списков в общий отсортированный список выполняется за **O(m + n)**.

Эффективность различных классов сложности

А теперь рассмотрим основные примеры временной и пространственной сложности подробнее.

Временная сложность. O(1) — константное время. Константное время выполнения означает, что время выполнения алгоритма не зависит от размера входных данных. Время выполнения остается постоянным,

независимо от того, сколько элементов содержит массив или как много данных обрабатывает алгоритм. Это происходит потому, что алгоритм выполняет фиксированное количество операций, не зависящих от размера входных данных:

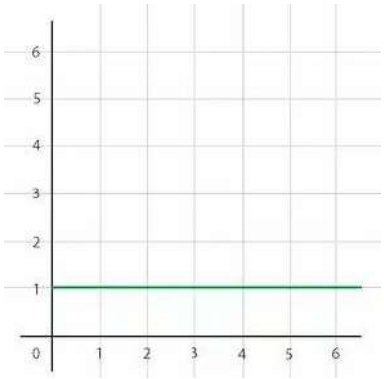


Рисунок 3. Константная сложность.

Примером константного времени выполнения может служить операция доступа к элементу массива по индексу. Например, если у вас есть массив friends и вы хотите вывести четвертый элемент массива, независимо от того, сколько элементов содержит массив, время выполнения этой операции останется постоянным — потому, что для доступа к элементу массива по индексу требуется выполнить только одну операцию, которая не зависит от размера массива:

```
        let friends = ['Rachel', 'Phoebe', 'Ross',
'Monica', 'Chandler', 'Joey'];
function printFourthIndex(){
  console.log(friends[3]); // O(1)
}
```

O(log n) — логарифмическое время. Логарифмическое время выполнения означает, что время выполнения алгоритма увеличивается пропорционально логарифму размера входных данных, то есть очень медленно:

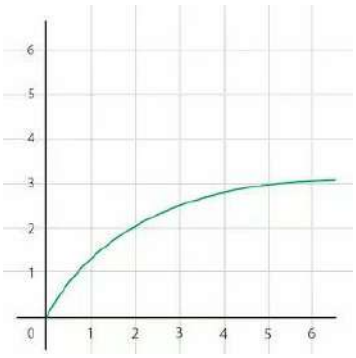


Рисунок 4. Логарифмическая сложность.

Примером алгоритма с логарифмическим временем выполнения может служить бинарный поиск. В бинарном поиске мы делим массив пополам на каждом шаге, пока не найдем искомый элемент. Таким

образом, количество шагов, необходимых для поиска, растет логарифмически по отношению к размеру массива.

Вот простой пример функции на JavaScript, которая имеет логарифмическое время выполнения:

```
        // n = 8
function logNFunc(n){
  while(n > 1){
    n = Math.floor(n / 2);
  }
}
```

Функция logNFunc работает так:

```
        n = 8

Итерация 1 = 8 / 2 = 4
Итерация 2 = 4 / 2 = 2
Итерация 3 = 2 / 2 = 1
```

То есть мы достигли 1 всего за 3 шага. Алгоритмы с логарифмическим временем выполнения очень эффективны: они могут обрабатывать большие объемы данных с относительно небольшим увеличением времени выполнения. На графике, приведенном выше, видно, что на больших числах кривая становится все более и более плоской, больше напоминая прямую константного времени.

O(n) — линейное время. Этот вариант мы уже рассматривали выше, в примере с пациентами стоматолога. Линейная временная сложность означает, что время выполнения алгоритма растет прямо пропорционально размеру входных данных — если размер входных данных увеличивается в 6 раз, время выполнения алгоритма также увеличится в 6 раз:

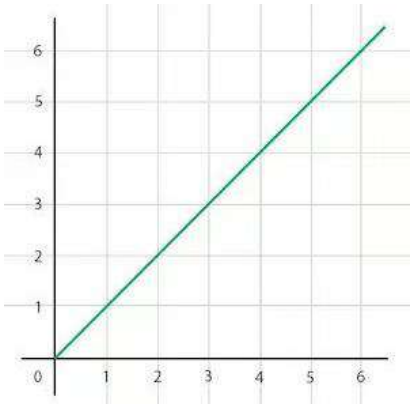


Рисунок 5. Линейная сложность.

Примером алгоритма с линейной временной сложностью может служить простой цикл, который проходит через каждый элемент массива. Например, как в этой функции:

```
        function nFunc(n){
```

```
  for(let i = 0; i < n; i++){
    console.log("Step 👉", i + 1);
  }
}
```

Алгоритмы с линейной временной сложностью тоже считаются очень эффективными, хотя, как очевидно, при обработке очень больших объемов данных они уступают в эффективности алгоритмам с логарифмическим временем выполнения.

O(n log n) — линейно-логарифмическое время. Если объединить последние две временные сложности (логарифмическую и линейную), мы получаем линейно-логарифмическое время, которое иногда также называют **псевдолинейным**. Такое время выполнения часто встречается в алгоритмах, основанных на стратегии «разделяй и властвуй» — сортировка слиянием, быстрая сортировка и сортировка кучей.

Рассмотрим пример такого алгоритма:

```
        // n = 4
function nLogNFunc(n){
  let temp = n;
  while(n > 1){
    n = Math.floor(n/2);
    for(let i = 0; i < temp; i++){
      console.log("Step 👉", i + 1);
    }
  }
}
```

Сначала мы создаем переменную temp и присваиваем ей значение n, чтобы скопировать значение n в temp. Затем мы начинаем цикл while, в котором мы делим значение n пополам. На следующей строке у нас есть цикл for, который итерируетсяtemp раз. Таким образом, временная сложность внешнего цикла равна **O(log n)**, а внутреннего **O(n)**:

Итоговая сложность будет равна **O(n log n)**:

```
Итоговая сложность = O(n * log n);
                   = O(n log n);
```

Как видно по графику, при больших объемах данных логарифмическая часть функции растет довольно медленно, и в результате увеличение времени выполнения происходит почти линейно:

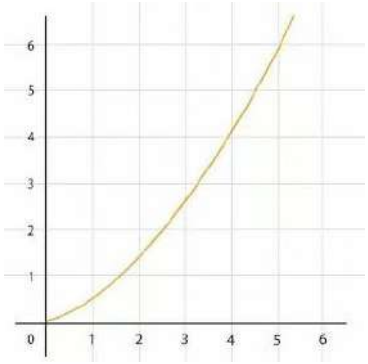


Рисунок 6. Линейно-логарифмическая сложность.

Линейно-логарифмическое время — последний класс сложности, который считается очень эффективным: все последующие типы алгоритмов работают гораздо медленнее.

O(n^2) — квадратичное время. Квадратичная временная сложность означает, что время выполнения алгоритма увеличивается пропорционально квадрату размера входных данных: если размер входных данных удваивается, время выполнения алгоритма увеличится в четыре раза. Примерами алгоритмов с квадратичной временной сложностью являются неэффективные алгоритмы сортировки — пузырьковая сортировка, сортировка вставками и сортировка выбором.

Рассмотрим на примере:

```
        // n = 4
function nSquaredNFunc(n){
 for(let j = 0; j < n; j++){
   for(let k = 0; k < n; k++){
     console.log(j, k);
   }
 }
}
```

Эта функция принимает n в качестве аргумента. Внешний цикл будет выполнен nраз, где каждая итерация будет включать в себя выполнение внутреннего цикла. Внутренний цикл также будет выполнен n раз:

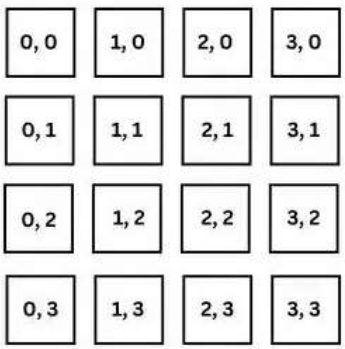


Рисунок 7. Матрица.

Эта матрица (рис.7) состоит из 4 строк и 4 столбцов, потому что n в нашем коде было равно 4. Подсчитаем количество шагов, необходимых для вывода матрицы 8x8:

```
Временная сложность = n * n
                    = 8 * 8
                    = 64
```

Как видно по графику, время выполнения функции растет гораздо быстрее, чем у алгоритма с линейно-логарифмической сложностью:

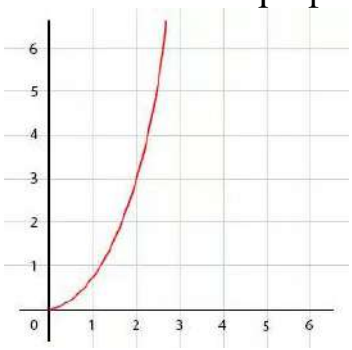


Рисунок 8. Квадратичная сложность.

O(2^n) — экспоненциальное время. Экспоненциальная временная сложность означает, что время выполнения алгоритма увеличивается экспоненциально: даже небольшое увеличение размера входных данных приводит к значительному увеличению времени выполнения.

Рассмотрим функцию fib(n), которая вычисляет числа Фибоначчи с использованием рекурсии. В этой функции для каждого числа nвызываются две функции fib(n — 1) и fib(n — 2) :

```
function fib(n){
  if(n === 0){
    return 0;
  }
  if(n === 1){
    return 1;
  }

  return fib(n - 1) + fib(n - 2)
}
```

Это приводит к тому, что количество вызовов функции удваивается на каждом уровне рекурсии. Например, на первом уровне функция вызывается два раза, на втором уровне — четыре раза, на третьем уровне — восемь раз и так далее.

Рекурсивное вычисление ряда Фибоначчи

Временная сложность этой функции составляет **O(2^n)**, поскольку количество вызовов функции удваивается на каждом уровне рекурсии,

что делает функцию неэффективной для больших значений n. В целом, алгоритмы с экспоненциальной сложностью непригодны для работы с большими объемами данных:

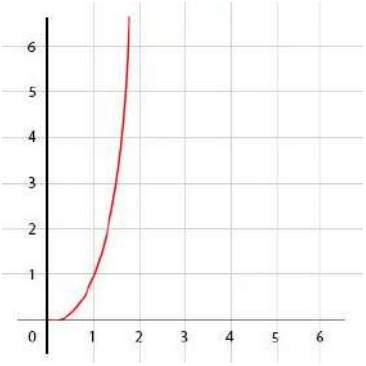


Рисунок 9. Экспоненциальная сложность.

O(n!) — факториальное время. Факториальное время подходит для решения реальных задач даже меньше, чем экспоненциальное, поскольку оно увеличивается гораздо быстрее.

Пример алгоритма с факториальной сложностью — функция, которая пытается найти все возможные перестановки набора элементов:

```
        function f(n){
  if(n === 0){
    return 0;
  }

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

Количество таких перестановок растет факториально, что делает алгоритм совершенно непригодным для работы с большими объемами данных:

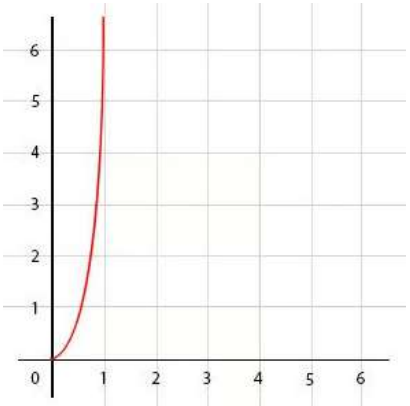


Рисунок 10. График роста факториальных перестановок.

Все рассмотренные выше примеры учитывали лишь один входной параметр. Но что, если таких параметров будет больше? В общем случае,

если имеется алгоритм с несколькими входными параметрами, его сложность определяется как функция от этих параметров:

- Если операции выполняются последовательно для каждого входного параметра, сложность будет суммой их размеров, **O(m + n)**.
- Если же операции вложены друг в друга (например, вложенные циклы), сложность будет произведением их размеров, **O(mn)**.

O(m+n) означает, что время выполнения алгоритма зависит от суммы размеров входных данных **m** и **n**:

```
        function mn(m, n){
  for(let i = 0; i < m; i++){
    console.log("m => ", i);
  }
  for(let j = 0; j < n; j++){
    console.log("n => ", j);
  }
}
```

В этом примере первый цикл выполняется **m** раз, а второй — **n** раз. Следовательно, общее время выполнения функции пропорционально сумме **m + n**. Это означает, что сложность алгоритма является линейной относительно каждого из входных параметров и общая сложность равна **O(m + n)**.

O(mn) означает, что время выполнения алгоритма зависит от произведения размеров входных данных **m** и **n**:

```
        function mn(m, n){
  for(let i = 0; i < m; i++){
    for(let j = 0; j < n; j++){
      console.log(i, j);
    }
  }
}
```

Здесь внутренний цикл выполняется **n** раз для каждой итерации внешнего цикла, который выполняется **m** раз. Таким образом, общее количество операций, выполняемых функцией, равно **m * n**. Следовательно, сложность алгоритма является квадратичной относительно входных параметров и общая сложность равна **O(mn)**.

Пространственная сложность

Эффективность алгоритмов определяется не только по времени выполнения, но и по объему использованной памяти (пространства в терминологии большого О).

O(1) — константное пространство. Константная пространственная сложность означает, что количество памяти, необходимое алгоритму, не зависит от размера входных данных. В этом

примере используется только две переменных: nи result. Независимо от размера n, количество памяти, используемое этой функцией, остается постоянным:

```
        function constantSpace(n) {
    let result = n * n;
    console.log(result);
}
```

O(log n) — логарифмическое пространство. Пространственная сложность O(log n) означает, что количество памяти, необходимое алгоритму, растет логарифмически с увеличением размера входных данных. В этом примере используется бинарный поиск, и с каждой итерацией цикла размер данных уменьшается вдвое:

```
    function binarySearch(array, x) {
    let start = 0, end = array.length - 1;

    while (start <= end) {
        let mid = Math.floor((start + end) / 2);
        if (array[mid] === x) return true;
        else if (array[mid] < x)
             start = mid + 1;
        else
             end = mid - 1;
    }

    return false;
}
```

O(n) — линейное пространство. Линейная пространственная сложность означает, что количество памяти, необходимое алгоритму, увеличивается пропорционально размеру входных данных:

```
        function linearSpace(n) {
    let array = [];
    for(let i = 0; i < n; i++) {
        array[i] = i;
    }
    console.log(array);
}
```

Алгоритмы с квадратичной пространственной сложностью используют объем памяти, который растет квадратично по мере увеличения размера входных данных. Пространственная сложность этого примера составляет **O(n^2) — квадратичное пространство**, поскольку в функции создается матрица, размер которой равен **n^2**:

```
    function quadraticSpace(n) {
    let matrix = [];
    for(let i = 0; i < n; i++) {
        matrix[i] = [];
        for(let j = 0; j < n; j++) {
            matrix[i][j] = i + j;
        }
    }
    console.log(matrix);
}
```

Выводы

Концепция большого «О» может показаться сложной на первый взгляд, но разобраться в ней необходимо: наивный подход к решению задач зачастую приводит к чрезмерно высокой временной и/или пространственной сложности, что делает такие решения непрактичными для использования в реальных масштабных проектах.

Инвестируя время в понимание вычислительной сложности и изучение существующих оптимальных алгоритмов, разработчики приобретают способность создавать действительно производительные, масштабируемые и ресурсоэффективные системы, которые могут справляться с самыми сложными задачами в реальном мире.

Контрольные вопросы

1. Что такое сложность алгоритма и почему она важна?
2. Что означает нотация Big O и как она используется для оценки алгоритмов?
3. Какие существуют основные классы сложности (O(1), O(n), O(n^2), O(log n), и т.д.)?
4. Как оценить временную сложность алгоритма?
5. В чем разница между временной и пространственной сложностью?
6. Как можно провести анализ сложности алгоритма?
7. Что такое лучший, худший и средний случаи сложности?
8. Как влияет размер входных данных на производительность алгоритма?
9. Что представляет собой квадратичное пространство?
10. Приведите пример алгоритма с известной сложностью и объясните, как вы пришли к этой оценке.

ТЕМА 12. АЛГОРИТМЫ СОРТИРОВКИ. КВАДРАТИЧНАЯ, ЛОГАРИФМИЧЕСКАЯ И ЛИНЕЙНАЯ СЛОЖНОСТЬ СОРТИРОВОК

В этой теме мы рассмотрим пять основных алгоритмов сортировки данных в массиве. Начнем с самого простого — пузырьковой сортировки — и закончим быстрой сортировкой.

Метод Swap

Для упрощения кода и улучшения читаемости мы введем метод Swap, который будет менять местами значения в массиве по индексу.

```
void Swap(T[] items, int left, int right)
{
    if (left != right)
    {
        T temp = items[left];
        items[left] = items[right];
        items[right] = temp;
    }
}
```

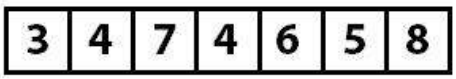
Пузырьковая сортировка

Сложность	Наилучший случай	В среднем	Наихудший случай
Время	O(n)	$O(n^2)$	$O(n^2)$
Память	O(1)	O(1)	O(1)

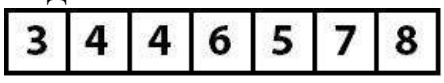
Сортировка пузырьком — это самый простой алгоритм сортировки. Он проходит по массиву несколько раз, на каждом этапе перемещая самое большое значение из неотсортированных в конец массива.

Например, у нас есть массив целых чисел:

При первом проходе по массиву мы сравниваем значения 3 и 7. Поскольку 7 больше 3, мы оставляем их как есть. После чего сравниваем 7 и 4. 4 меньше 7, поэтому мы меняем их местами, перемещая семерку на одну позицию ближе к концу массива. Теперь он выглядит так:



Этот процесс повторяется до тех пор, пока семерка не дойдет почти до конца массива. В конце она сравнивается с элементом 8, которое больше, а значит, обмена не происходит. После того, как мы обошли массив один раз, он выглядит так:



Поскольку был совершен по крайней мере один обмен значений, нам нужно пройти по массиву еще раз. В результате этого прохода мы перемещаем на место число 6.

3	4	4	5	6	7	8

И снова был произведен как минимум один обмен, а значит, проходим по массиву еще раз.

При следующем проходе обмена не производится, что означает, что наш массив отсортирован, и алгоритм закончил свою работу.

```
public void Sort(T[] items)
{
    bool swapped;

    do
    {
        swapped = false;
        for (int i = 1; i < items.Length; i++) {
            if (items[i - 1].CompareTo(items[i]) >
0)
            {
                Swap(items, i - 1, i);
                swapped = true;
            }
        }
    } while (swapped != false);
}
```

Сортировка вставками

Сложность	Наилучший случай	В среднем	Наихудший случай
Время	O(n)	$O(n^2)$	$O(n^2)$
Память	O(1)	O(1)	O(1)

Сортировка вставками работает, проходя по массиву и перемещая нужное значение в начало массива. После того, как обработана очередная позиция, мы знаем, что все позиции до нее отсортированы, а после нее — нет.

Важный момент: сортировка вставками обрабатывает элементы массива по порядку. Поскольку алгоритм проходит по элементам слева направо, мы знаем, что все, что слева от текущего индекса — уже отсортировано. На рисунке 1 показано, как увеличивается отсортированная часть массива с каждым проходом:

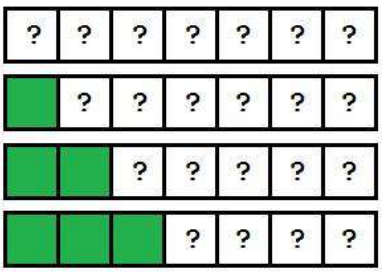


Рисунок 1. Сортировка массива.

Постепенно отсортированная часть массива растет, и, в конце концов, массив окажется упорядоченным.

Давайте взглянем на конкретный пример. Вот наш неотсортированный массив, который мы будем использовать:

Алгоритм начинает работу с индекса 0 и значения 3. Поскольку это первый индекс, массив до него включительно считается отсортированным.

Далее мы переходим к числу 7. Поскольку 7 больше, чем любое значение в отсортированной части, мы переходим к следующему элементу.

На этом этапе элементы с индексами 0..1 отсортированы, а про элементы с индексами 2..n ничего не известно.

Следующим проверяется значение 4. Так как оно меньше семи, мы должны перенести его на правильную позицию в отсортированную часть массива. Остается вопрос: как ее определить? Это осуществляется методом FindInsertionIndex. Он сравнивает переданное ему значение (4) с каждым значением в отсортированной части, пока не найдет место для вставки.

Итак, мы нашли индекс 1 (между значениями 3 и 7). Метод Insert осуществляет вставку, удаляя вставляемое значение из массива и сдвигая все значения, начиная с индекса для вставки, вправо. Теперь массив выглядит так:

Теперь часть массива, начиная от нулевого элемента и заканчивая элементом с индексом 2, отсортирована. Следующий проход начинается с индекса 3 и значения 4. По мере работы алгоритма мы продолжаем делать такие вставки.

Когда больше нет возможностей для вставок, массив считается полностью отсортированным, и работа алгоритма закончена.

```
public void Sort(T[] items)
```

```
{
    int sortedRangeEndIndex = 1;

    while (sortedRangeEndIndex < items.Length)
    {
        if
(items[sortedRangeEndIndex].CompareTo(items[sortedR
angeEndIndex - 1]) < 0)
        {
            int insertIndex =
FindInsertionIndex(items,
items[sortedRangeEndIndex]);
            Insert(items, insertIndex,
sortedRangeEndIndex);
        }

        sortedRangeEndIndex++;
    }
}
private int FindInsertionIndex(T[] items, T
valueToInsert)
{
    for (int index = 0; index < items.Length;
index++) {
        if (items[index].CompareTo(valueToInsert) >
0)
        {
            return index;
        }
    }

    throw new InvalidOperationException("The
insertion index was not found");
}
private void Insert(T[] itemArray, int
indexInsertingAt, int indexInsertingFrom)
{
    // itemArray =       0 1 2 4 5 6 3 7
    // insertingAt =     3
    // insertingFrom =   6
    //
    // Действия:
```

```
    //  1: Сохранить текущий индекс в temp
    //  2: Заменить indexInsertingAt на
indexInsertingFrom
    //  3: Заменить indexInsertingAt на
indexInsertingFrom в позиции +1
    //     Сдвинуть элементы влево на один.
    //  4: Записать temp на позицию в массиве + 1.


    // Шаг 1.
    T temp = itemArray[indexInsertingAt];

    // Шаг 2.

    itemArray[indexInsertingAt] =
itemArray[indexInsertingFrom];

    // Шаг 3.
    for (int current = indexInsertingFrom; current
> indexInsertingAt; current--)
    {
        itemArray[current] = itemArray[current -
1];
    }

    // Шаг 4.
    itemArray[indexInsertingAt + 1] = temp;
}
```

Сортировка выбором

Сложность	Наилучший случай	В среднем	Наихудший случай
Время	O(n)	$O(n^2)$	$O(n^2)$
Память	O(1)	O(1)	O(1)

Сортировка выбором — это некий гибрид между пузырьковой и сортировкой вставками. Как и сортировка пузырьком, этот алгоритм проходит по массиву раз за разом, перемещая одно значение на правильную позицию. Однако, в отличие от пузырьковой сортировки, он выбирает наименьшее неотсортированное значение вместо наибольшего. Как и при сортировке вставками, упорядоченная часть массива расположена в начале, в то время как в пузырьковой сортировке она находится в конце.

Давайте посмотрим на работу сортировки выбором на нашем неотсортированном массиве.

При первом проходе алгоритм с помощью метода FindIndexOfSmallestFromIndex пытается найти наименьшее значение в массиве и переместить его в начало.

Имея такой маленький массив, мы сразу можем сказать, что наименьшее значение — 3, и оно уже находится на правильной позиции. На этом этапе мы знаем, что на первой позиции в массиве (индекс 0) находится самое маленькое значение, следовательно, начало массива уже отсортировано. Поэтому мы начинаем второй проход — на этот раз по индексам от 1 до n – 1.

На втором проходе мы определяем, что наименьшее значение — 4. Мы меняем его местами со вторым элементом, семеркой, после чего 4 встает на свою правильную позицию.

Теперь неотсортированная часть массива начинается с индекса 2. Она растет на один элемент при каждом проходе алгоритма. Если на каком-либо проходе мы не сделали ни одного обмена, это означает, что массив отсортирован.

После еще двух проходов алгоритм завершает свою работу:


```
public void Sort(T[] items)
{
    int sortedRangeEnd = 0;
     while (sortedRangeEnd < items.Length)
    {
        int nextIndex =
FindIndexOfSmallestFromIndex(items,
sortedRangeEnd);
        Swap(items, sortedRangeEnd, nextIndex);

        sortedRangeEnd++;
    }
}
 private int FindIndexOfSmallestFromIndex(T[]
items, int sortedRangeEnd)
{
    T currentSmallest = items[sortedRangeEnd];
```

```
    int currentSmallestIndex = sortedRangeEnd;

    for (int i = sortedRangeEnd + 1; i <
items.Length; i++)
    {
        if (currentSmallest.CompareTo(items[i]) >
0)
        {
            currentSmallest = items[i];
            currentSmallestIndex = i;
        }
    }

    return currentSmallestIndex;
}
```

Сортировка слиянием

Сложность	Наилучший случай	В среднем	Наихудший случай
Время	O(n·log n)	O(n·log n)	O(n·log n)
Память	O(n)	O(n)	O(n)

Алгоритм «Разделяй и властвуй»

До сих пор мы рассматривали линейные алгоритмы. Они используют мало дополнительной памяти, но имеют квадратичную сложность. На примере сортировки слиянием мы посмотрим на алгоритм типа «разделяй и властвуй» *(divide and conquer)*.

Алгоритмы этого типа работают, разделяя крупную задачу на более мелкие, решаемые проще. Мы пользуемся ими каждый день. К примеру, поиск в телефонной книге — один из примеров такого алгоритма.

Если вы хотите найти человека по фамилии Петров, вы не станете искать, начиная с буквы А и переворачивая по одной странице. Вы, скорее всего, откроете книгу где-то посередине. Если попадете на букву Т, перелистнете несколько страниц назад, возможно, слишком много — до буквы О. Тогда вы пойдете вперед. Таким образом, перелистывая туда и обратно все меньшее количество страниц, вы, в конце концов, найдете нужную.

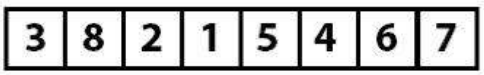
Эффективность алгоритмов. Предположим, что в телефонной книге 1000 страниц. Если вы открываете ее на середине, вы отбрасываете 500 страниц, в которых нет искомого человека. Если вы не попали на нужную страницу, вы выбираете правую или левую сторону и снова оставляете половину доступных вариантов. Теперь вам надо просмотреть 250 страниц. Таким образом мы делим нашу задачу пополам снова и

снова и можем найти человека в телефонной книге всего за 10 просмотров. Это составляет 1% от всего количества страниц, которые нам пришлось бы просмотреть при линейном поиске.

Сортировка слиянием

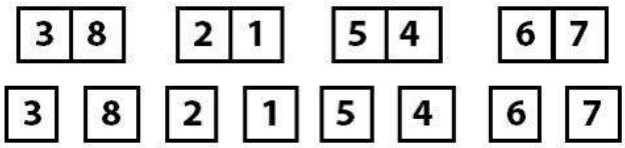
При сортировке слиянием мы разделяем массив пополам до тех пор, пока каждый участок не станет длиной в один элемент. Затем эти участки возвращаются на место (сливаются) в правильном порядке.

Давайте посмотрим на такой массив:

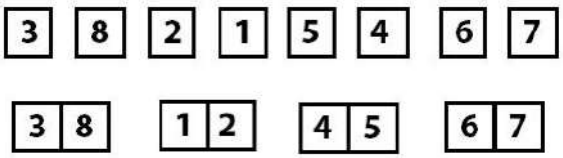


Разделим его пополам:

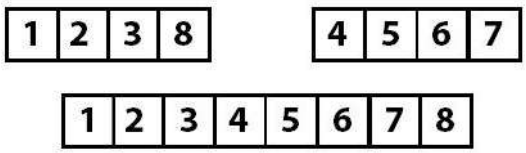
И будем делить каждую часть пополам, пока не останутся части с одним элементом:



Теперь, когда мы разделили массив на максимально короткие участки, мы сливаем их в правильном порядке.



Сначала мы получаем группы по два отсортированных элемента, потом «собираем» их в группы по четыре элемента и в конце собираем все вместе в отсортированный массив.



Для работы алгоритма мы должны реализовать следующие операции:

1. Операцию для рекурсивного разделения массива на группы (метод Sort).
2. Слияние в правильном порядке (метод Merge).

Стоит отметить, что в отличие от линейных алгоритмов сортировки, сортировка слиянием будет делить и склеивать массив вне зависимости от того, был он отсортирован изначально или нет. Поэтому, несмотря на то, что в худшем случае он отработает быстрее, чем линейный, в лучшем случае его производительность будет ниже, чем у линейного. Поэтому сортировка слиянием — не самое лучшее решение, когда надо отсортировать частично упорядоченный массив.

```
public void Sort(T[] items)
```

```
{
    if (items.Length <= 1)
    {
         return;
    }

    int leftSize = items.Length / 2;
    int rightSize = items.Length - leftSize;
    T[] left = new T[leftSize];
    T[] right = new T[rightSize];
    Array.Copy(items, 0, left, 0, leftSize);
    Array.Copy(items, leftSize, right, 0,
rightSize);
    Sort(left);
    Sort(right);
    Merge(items, left, right);
}

private void Merge(T[] items, T[] left, T[] right)
{
    int leftIndex = 0;
    int rightIndex = 0;
    int targetIndex = 0;
    int remaining = left.Length + right.Length;
    while(remaining > 0)
    {
        if (leftIndex >= left.Length)
        {
            items[targetIndex] =
right[rightIndex++];
        }
        else if (rightIndex >= right.Length)
        {
            items[targetIndex] = left[leftIndex++];
        }
        else if
(left[leftIndex].CompareTo(right[rightIndex]) < 0)
        {
            items[targetIndex] = left[leftIndex++];
        }
        else
        {
```

```
                items[targetIndex] =
right[rightIndex++];
            }

            targetIndex++;
            remaining--;
        }
}
```

Быстрая сортировка

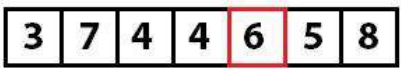
Сложность	Наилучший случай	В среднем	Наихудший случай
Время	O(n·log n)	O(n·log n)	$O(n^2)$
Память	O(1)	O(1)	O(1)

Быстрая сортировка — это еще один алгоритм типа «разделяй и властвуй». Он работает, рекурсивно повторяя следующие шаги:

1. Выбрать ключевой индекс и разделить по нему массив на две части. Это можно делать разными способами, но в данной статье мы используем случайное число.
2. Переместить все элементы больше ключевого в правую часть массива, а все элементы меньше ключевого — в левую. Теперь ключевой элемент находится в правильной позиции — он больше любого элемента слева и меньше любого элемента справа.
3. Повторяем первые два шага, пока массив не будет полностью отсортирован.

Давайте посмотрим на работу алгоритма на следующем массиве:

Сначала мы случайным образом выбираем ключевой элемент:



Теперь, когда мы знаем ключевой индекс (4), мы берем значение, находящееся по этому индексу (6), и переносим значения в массиве так, чтобы все числа больше или равные ключевому были в правой части, а все числа меньше ключевого — в левой. Обратите внимание, что в процессе переноса значений индекс ключевого элемента может измениться (мы увидим это вскоре).

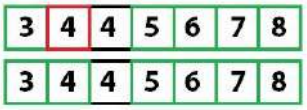
Перемещение значений осуществляется методом partition.

На этом этапе мы знаем, что значение 6 находится на правильной позиции. Теперь мы повторяем этот процесс для правой и левой частей массива.

Мы рекурсивно вызываем метод quicksort на каждой из частей. Ключевым элементом в левой части становится пятерка. При перемещении значений она изменит свой индекс. Главное — помнить, что нам важно именно ключевое значение, а не его индекс.

Снова применяем быструю сортировку:

И еще раз:



У нас осталось одно неотсортированное значение, а, поскольку мы знаем, что все остальное уже отсортировано, алгоритм завершает работу.

```
Random _pivotRng = new Random();

public void Sort(T[] items)
{
    quicksort(items, 0, items.Length - 1);
}
private void quicksort(T[] items, int left, int
right)
{
    if (left < right)
    {
        int pivotIndex = _pivotRng.Next(left,
right);
        int newPivot = partition(items, left,
right, pivotIndex);

        quicksort(items, left, newPivot - 1);
        quicksort(items, newPivot + 1, right);
    }
}
private int partition(T[] items, int left, int
right, int pivotIndex)
{
    T pivotValue = items[pivotIndex];

    Swap(items, pivotIndex, right);
```

```
    int storeIndex = left;

    for (int i = left; i < right; i++)
    {
        if (items[i].CompareTo(pivotValue) < 0)
        {
            Swap(items, i, storeIndex);
            storeIndex += 1;
        }
    }
     Swap(items, storeIndex, right);
    return storeIndex;
}
```

Контрольные вопросы

1. Что такое сортировка и для чего она используется?
2. В чем разница между квадратной, логарифмической и линейной сложностью сортировки?
3. Какие алгоритмы относятся к квадратичной сложности (например, сортировка пузырьком, сортировка вставками)?
4. Как работает алгоритм быстрой сортировки и какова его сложность?
5. Что такое сортировка слиянием и как она работает?
6. Каковы преимущества и недостатки различных алгоритмов сортировки?
7. Что такое стабильная и нестабильная сортировка?
8. Как работает алгоритм сортировки выбором и какова его сложность?
9. Приведите пример алгоритма сортировки с линейной сложностью (например, счетная сортировка) и объясните, как он работает.
10. Как можно оценить производительность алгоритмов сортировки на различных наборах данных?

ТЕМА 13. ВВЕДЕНИЕ В ТЕХНИКИ ПОИСКА

Бинарное дерево поиска — это усовершенствованный алгоритм, используемый для анализа узла, его левой и правой ветвей, которые моделируются в древовидной структуре, и возврата значения. Алгоритм бинарного дерева поиска (BST) разработан на основе архитектуры базового алгоритма двоичного поиска; следовательно, он обеспечивает более быстрый поиск, вставку и удаление узлов. Это делает программу действительно быстрой и точной.

Атрибуты двоичного дерева поиска

BST состоит из нескольких узлов и состоит из следующих атрибутов:

- Узлы дерева представлены в отношениях родитель-потомок.
- Каждый родительский узел может иметь ноль дочерних узлов или максимум два подузла или поддерева с левой и правой стороны.
- Каждое поддерево, также известное как двоичное дерево поиска, имеет подветви справа и слева от себя.
- Все узлы связаны парами ключ-значение.
- Ключи узлов, присутствующих в левом поддереве, меньше ключей их родительского узла.
- Аналогично, ключи узлов левого поддерева имеют меньшие значения, чем ключи их родительского узла.

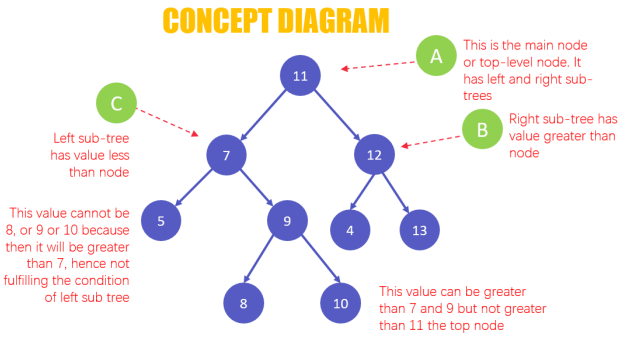


Рисунок 1. Бинарное дерево поиска.

A. Существует основной узел или родительский уровень 11. Под ним находятся левые и правые узлы/ветви со своими значениями ключей.
B. Правое поддерево имеет ключевые значения, превышающие значения родительского узла.
C. Левое поддерево имеет значения, чем родительский узел

Свойства двоичного дерева поиска

Двумя основными факторами, которые делают двоичное дерево поиска оптимальным решением любых реальных проблем, являются скорость и точность.

Благодаря тому, что двоичный поиск осуществляется в формате ветвей с отношениями родитель-потомок, алгоритм знает, в каком месте дерева нужно искать элементы. Это уменьшает количество сравнений «ключ-значение», которые программа должна выполнить, чтобы найти нужный элемент.

Кроме того, если искомый элемент больше или меньше родительского узла, узел знает, на какой стороне дерева искать. Причина в том, что левое поддерево всегда меньше родительского узла, а правое поддерево всегда имеет значения, равные или большие, чем родительский узел.

BST обычно используется для реализации сложного поиска, надежной игровой логики, действий автозаполнения и графики. Алгоритм эффективно поддерживает такие операции, как поиск, вставка и удаление.

Типы двоичных деревьев

Три вида бинарных деревьев:

- Полное двоичное дерево: все уровни дерева заполнены возможными исключениями последнего уровня. Аналогично, все узлы заполнены и направлены в крайнее левое положение. Все узлы имеют по два дочерних узла, кроме листа.
- Сбалансированное или идеальное двоичное дерево. В дереве все узлы имеют по два дочерних узла. Кроме того, уровень каждого подузла одинаковый.

Работа двоичного дерева поиска

Дерево всегда имеет корневой узел и дополнительные дочерние узлы, как слева, так и справа. Алгоритм выполняет все операции путем сравнения значений с корнем и его дальнейшими дочерними узлами в левом или правом поддереве соответственно.

В зависимости от элемента, который необходимо вставить, найти или удалить, после сравнения алгоритм может легко удалить левое или правое поддерево корневого узла.

BST в первую очередь предлагает для вашего использования следующие три типа операций:

- Поиск: ищет элемент в двоичном дереве.
- Вставка: добавляет элемент в двоичное дерево.
- Удаление: удалить элемент из двоичного дерева.

Каждая операция имеет свою структуру и метод выполнения/анализа, но самой сложной из всех является операция удаления.

Поиск

Всегда начинайте анализ дерева в корневом узле, а затем двигайтесь дальше к правому или левому поддереву корневого узла в зависимости от того, меньше или больше искомого элемента, чем корень.

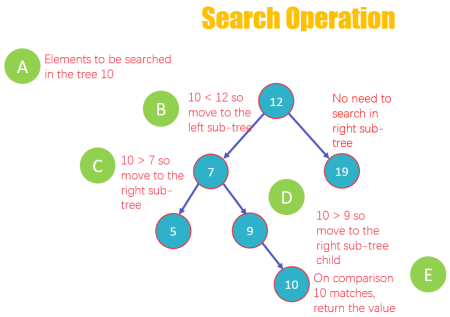


Рисунок 2. Операции поиска.

A. Элемент для поиска равен 10.
B. Сравните элемент с корневым узлом 12, 10 < 12, следовательно, вы перейдете в левое поддерево. Нет необходимости анализировать правое поддерево
C. Теперь сравните 10 с узлом 7, 10 > 7, поэтому перейдите в правое поддерево.
D. Затем сравните 10 со следующим узлом, равным 9, 10 > 9, посмотрите на дочерний элемент правого поддерева.
E. 10 совпадений со значением в узле, 10 = 10, возвращают значение пользователю.

Псевдокод для поиска в BST

```
search(element, root)
     if !root
     return -1
     if root.value == element
     return 1
     if root.value < element
     search(element, root.right)
      else
     search(element, root.left)
```

Вставка

Это очень простая операция. Сначала вставляется корневой узел, затем следующее значение сравнивается с корневым узлом. Если значение больше корня, оно добавляется в правое поддерево, а если меньше корня, оно добавляется в левое поддерево.

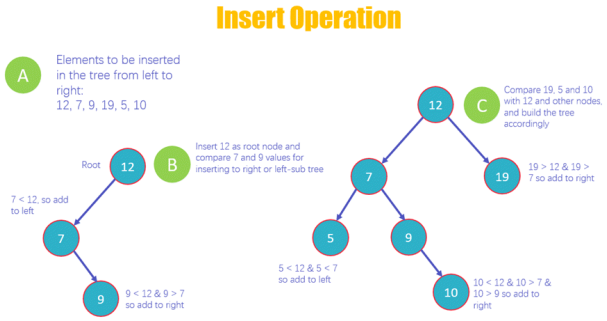


Рисунок 3. Операция вставки узла.

A. Существует список из 6 элементов, которые необходимо вставить в BST в порядке слева направо.
B. Вставьте 12 в качестве корневого узла и сравните следующие значения 7 и 9 для вставки соответственно в правое и левое поддерево.
C. Сравните оставшиеся значения 19, 5 и 10 с корневым узлом 12 и разместите их соответственно. 19 > 12 поместите его как правого дочернего элемента 12, 5 < 12 и 5 < 7, следовательно, поместите его как левого дочернего элемента 7. Теперь сравните 10, 10 < 12 и 10 > 7 и 10 > 9, поместите 10. как правое поддерево 9.

Псевдокод для вставки узла в BST

```
insert (element, root)
    Node x = root
    Node y = NULL
    while x:
     y = x
    if x.value < element.value
        x = x.right
    else
        x = x.left
    if y.value < element
     y.right = element
    else
     y.left = element
```

Удаление

Для удаления узла из BST существуют некоторые случаи, например, удаление корня или удаление листового узла. Также после удаления корня нам нужно подумать о корневом узле.

Допустим, мы хотим удалить листовой узел, мы можем просто удалить его, но если мы хотим удалить корень, нам нужно заменить значение корня другим узлом. Давайте возьмем следующий пример:

- Случай 1. Узел с нулевыми дочерними элементами: это самая простая ситуация, вам просто нужно удалить узел, у которого нет дочерних элементов ни справа, ни слева.
- Случай 2 — Узел с одним дочерним узлом: после удаления узла просто соедините его дочерний узел с родительским узлом удаленного значения.
- Случай 3 Узел с двумя детьми: это самая сложная ситуация, и она работает по следующим двум правилам.
  - 3a – По порядку предшественника: вам необходимо удалить узел с двумя дочерними элементами и заменить его наибольшим значением в левом поддереве удаленного узла.
  - 3b – В порядке преемника: вам необходимо удалить узел с двумя дочерними элементами и заменить его наибольшим значением в правом поддереве удаленного узла.

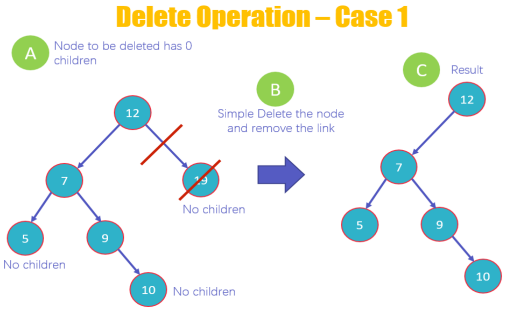


Рисунок 4. Операция удаления узла без потомка.

A. Это первый случай удаления, при котором вы удаляете узел, не имеющий дочерних элементов (рис.4). Как видно на схеме, у 19, 10 и 5 детей нет. Но мы удалим 19.
B. Удалите значение 19 и удалите ссылку из узла.
C. Посмотреть новую структуру БСТ без 19

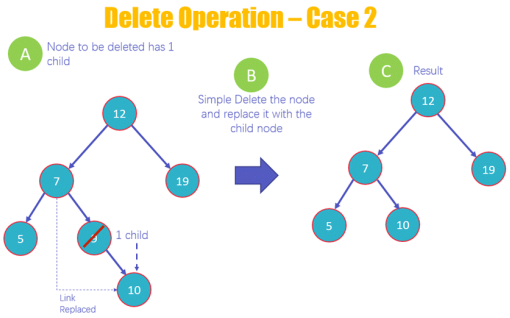


Рисунок 5. Операция удаления узла с потомком.

A. Это второй случай удаления, в котором вы удаляете узел, имеющий 1 дочерний узел (рис.5), как вы можете видеть на диаграмме, что у 9 есть один дочерний узел.
B. Удалите узел 9 и замените его дочерним 10 и добавьте ссылку с 7 на 10.

C. Посмотреть новую структуру БСТ без 9

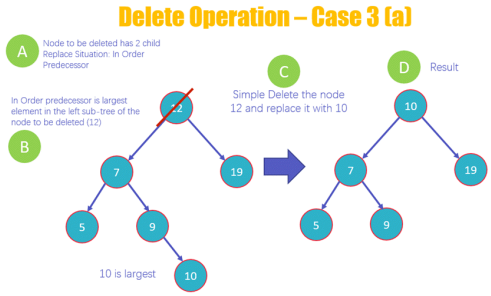


Рисунок 6. Операция удаления узла с двумя потомками.

A. Здесь вы удалите узел 12, у которого есть два дочерних элемента.
B. Удаление узла будет происходить на основе правила предшественника по порядку, что означает, что его заменит самый большой элемент в левом поддереве из 12.
C. Удалите узел 12 и замените его 10, так как это наибольшее значение в левом поддереве.
D. Посмотреть новую структуру BST после удаления 12

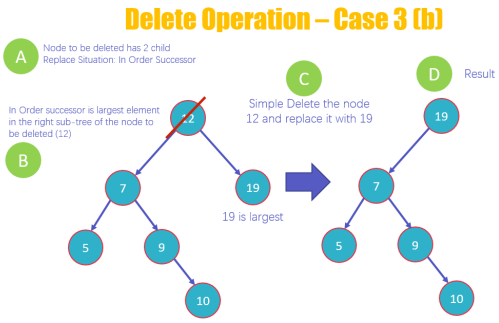


Рисунок 7. Новая структура дерева после удаления узла с двумя потомками.

A. 1. Удалить узел 12, у которого есть два дочерних элемента.
B. 2 Удаление узла будет происходить на основе правила «Последователь по порядку», что означает, что самый большой элемент в правом поддереве из 12 заменит его.
C. 3. Удалите узел 12 и замените его 19, так как это наибольшее значение в правом поддереве.
D. 4 Просмотрите новую структуру BST после удаления 12

Псевдокод для удаления узла

```
delete (value, root):
    Node x = root
    Node y = NULL
# searching the node
    while x:
      y = x
    if x.value < value
```

```
        x = x.right
    else if x.value > value
        x = x.left
    else if value == x
        break
# if the node is not null, then replace it with
successor
    if y.left or y.right:
     newNode = GetInOrderSuccessor(y)
     root.value = newNode.value
# After copying the value of successor to the root
#we're deleting the successor
     free(newNode)
    else
     free(y)
```

Выводы

BST — это алгоритм расширенного уровня, выполняющий различные операции на основе сравнения значений узла с корневым узлом.

Любая точка в иерархии родитель-потомок представляет собой узел. По крайней мере, один родительский или корневой узел присутствует постоянно.

Есть левое поддерево и правое поддерево. Левое поддерево содержит значения, меньшие, чем корневой узел. Однако правое поддерево содержит значение, большее, чем корневой узел.

Каждый узел может иметь ноль, один или два потомка.

Двоичное дерево поиска облегчает основные операции, такие как поиск, вставка и удаление.

Удаление, будучи наиболее сложным, имеет несколько случаев, например, узел без дочернего элемента, узел с одним дочерним элементом и узел с двумя дочерними элементами.

Алгоритм используется в реальных решениях, таких как игры, данные автозаполнения и графика.

Контрольные вопросы

1. Что такое поиск в контексте структур данных и алгоритмов?
2. Какие основные типы поиска существуют (линейный поиск, бинарный поиск)?
3. Как работает линейный поиск и какова его сложность?
4. Как работает бинарный поиск и при каких условиях его можно применять?
5. Какие ограничения существуют у бинарного поиска?

6. Что такое интерполяционный поиск и как он отличается от бинарного?
7. Какова сложность алгоритмов поиска и какие факторы на нее влияют?
8. Какие структуры данных чаще всего используются для оптимизации поиска?
9. Как можно улучшить эффективность поиска в больших объемах данных?
10. Приведите пример реальной задачи, где используются различные техники поиска.

ТЕМА 14. СТРУКТУРА ДАННЫХ ГРАФ

Графы превратились в невероятно сильное средство моделирования и получения данных из соцсетей, веб-страниц и ссылок, а также определения местоположения и маршрутов в GPS. Любой набор объектов, которые связаны друг с другом, можно сейчас представить с помощью графа.

Рассмотрим 10 основных графовых алгоритмов, которые становятся очень полезными для анализа, а также области их применения.

Граф состоит из конечного множества вершин (узлов) и набора рёбер, соединяющих эти вершины. Две вершины считаются смежными, если они соединены друг с другом одним и тем же ребром.

Ниже приведён ряд базовых понятий, относящихся к графам. Они проиллюстрированы примерами на рисунке 1.

- **Порядок:** число вершин в графе.
- **Размер:** число рёбер в графе.
- **Степень вершины:** число рёбер, инцидентных вершине.
- **Изолированная вершина:** вершина, которая не связана ни с одной другой вершиной графа.
- **Петля**: ребро, вершины которого совпадают.
- **Ориентированный граф:** граф, в котором все рёбра имеют направление, определяющее начальную и конечную вершину.
- **Неориентированный граф:** граф с рёбрами, которые не имеют направления.
- **Взвешенный граф:** рёбра такого графа имеют определённый вес.
- **Невзвешенный граф:** рёбра такого графа не имеют никаких весов.

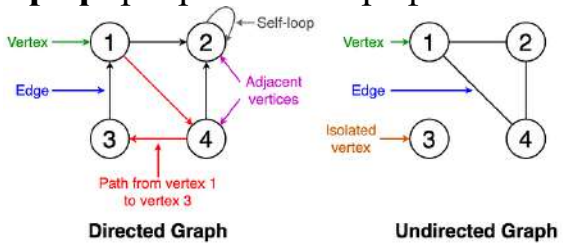


Рисунок 1. Терминология графов (схематичное изображение)

1. Поиск в ширину

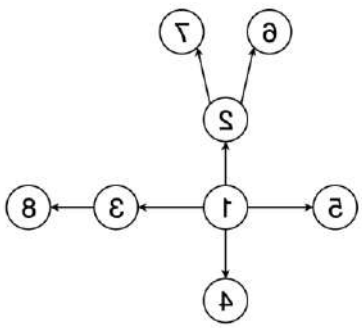


Рисунок 2. Визуальное отображение обхода на графах (поиск в ширину)

Обход или *поиск* — это одна из фундаментальных операций, выполняемых на графах. ***Поиск в ширину*** начинается с определённой вершины, затем исследуются все её соседи на данной глубине и происходит переход к вершинам следующего уровня. В графах, в отличие от деревьев, могут быть циклы — пути, в которых первая и последняя вершины совпадают. Поэтому необходимо отслеживать посещённые алгоритмом вершины. При реализации алгоритма поиска в ширину используется структура данных «очередь».

На рисунке 2 показан пример того, как выглядит поиск в ширину на графе. Жёлтым цветом помечаются обнаруженные вершины, красным — посещённые.

Применяется для:

- определения кратчайших путей и минимальных остовных деревьев;
- индексации веб-страниц поисковыми ботами;
- поиска в соцсетях;
- нахождения доступных соседних узлов в одноуровневых сетях, таких как BitTorrent.

2. Поиск в глубину

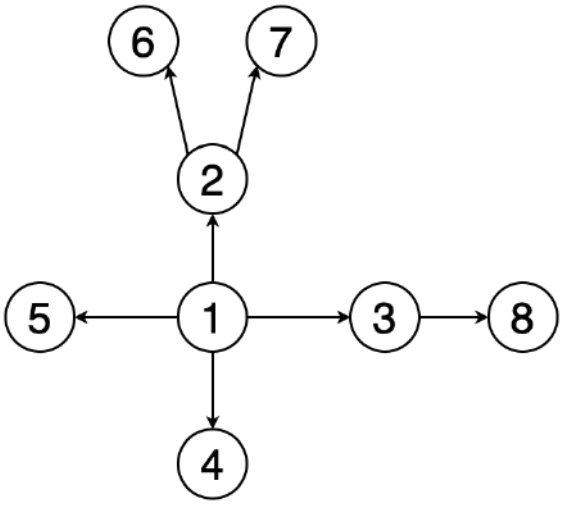


Рисунок 3. Визуальное отображение обхода на графах (поиск в глубину).

Поиск в глубину начинается с определённой вершины, затем уходит как можно дальше вдоль каждой ветви и возвращается обратно. Здесь тоже необходимо отслеживать посещённые алгоритмом вершины. Для того, чтобы стало возможным возвращение обратно, при реализации алгоритма поиска в глубину используется структура данных «стек».

На рисунке 3 показан пример того, как выглядит поиск в глубину на том же графе, который использован на рисунке 2. Граф обходится на всю глубину каждой ветви с возвращением обратно.

Применяется:

- для нахождения пути между двумя вершинами;
- для обнаружения циклов на графе;
- в топологической сортировке;
- в головоломках с единственным решением (например, лабиринтах).

3. Кратчайший путь

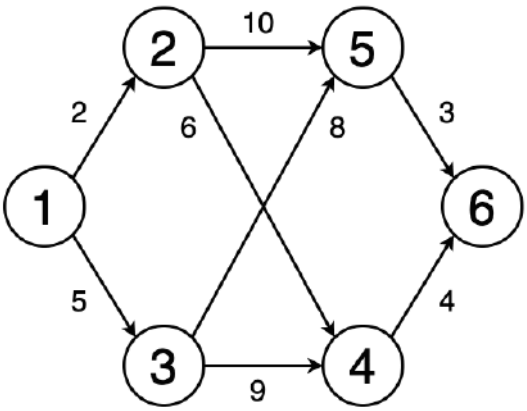


Рисунок 4. Визуальное отображение кратчайшего пути от вершины 1 к вершине 6.

Кратчайший путь от одной вершины графа к другой — это путь, при котором сумма весов рёбер, его составляющих, должна быть минимальна.

На рисунке 4 показан кратчайший путь на графе от вершины 1 до вершины 6.

Алгоритмы нахождения кратчайшего пути:

1. Алгоритм Дейкстры.
2. Алгоритм Беллмана-Форда.

Применение алгоритмов кратчайшего пути

- в картографических сервисах типа Google maps или Apple maps для прокладки маршрутов и определения местоположения;
- в сетях для решения проблемы минимальной задержки пути;
- в абстрактных автоматах для определения через переход между различными состояниями возможных вариантов достижения некоторого целевого состояния, например минимально возможного количества ходов, необходимого для победы в игре.

4. Обнаружение циклов

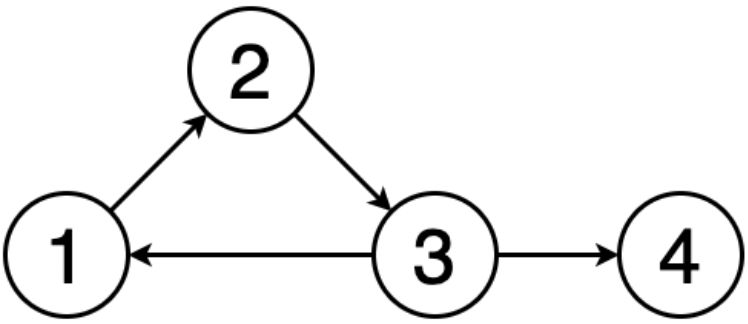


Рисунок 5. Цикл.

Цикл — это путь, в котором первая и последняя вершины графа совпадают. То есть путь, начинающийся и завершающийся в одной и той же вершине, называется циклом. ***Обнаружение циклов*** — это процесс выявления таких циклов. На рисунке 5 показано, как происходит обнаружение цикла.

Алгоритмы обнаружения цикла:

1. Алгоритм Флойда.
2. Алгоритм Брента.

Применяются:

- в распределённых алгоритмах, использующих сообщения;
- для обработки крупных графов с использованием распределённой системы обработки в кластере;
- для обнаружения взаимоблокировок в системах с параллельным выполнением;
- в криптографических приложениях для выявления ключей сообщения, которые могут соответствовать одному и тому же зашифрованному значению.

5. Минимальное остовное дерево

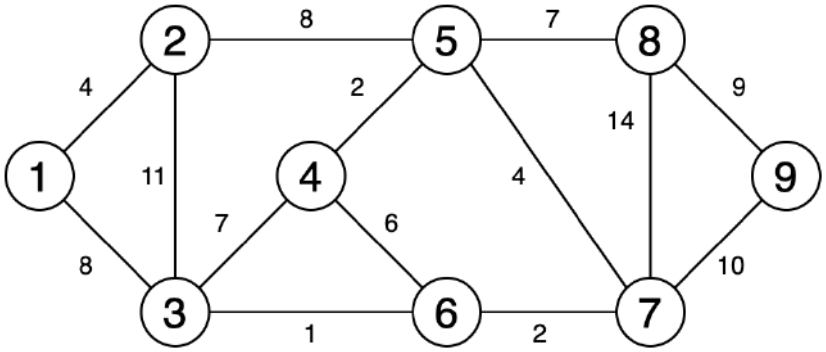


Рисунок 6. Визуальное отображение минимального остовного дерева.

Минимальное остовное дерево — это подмножество рёбер графа, которое соединяет все вершины, имеющие минимальную сумму весов рёбер, и без циклов.

На рисунке 6 показан процесс получения минимального остовного дерева.

Алгоритмы поиска минимального остовного дерева:

1. Алгоритм Прима.
2. Алгоритм Крускала.

Применяются:

- для создания деревьев для распределения данных в компьютерных сетях;
- в кластерном анализе с использованием графов;
- при сегментации изображений;
- при социально-географическом районировании, когда смежные регионы объединяются.

Контрольные вопросы

1. Что такое граф и из каких компонентов он состоит?
2. В чем разница между направленными и ненаправленными графами?
3. Что такое взвешенные графы и в каких случаях они используются?
4. Как можно представить граф в памяти компьютера (например, с помощью матрицы смежности и списка смежности)?
5. Какие основные операции можно выполнять с графами?
6. Каковы алгоритмы обхода графа (например, обход в глубину и обход в ширину)?
7. Что такое циклы в графах, и как их можно обнаружить?
8. Как работают алгоритмы поиска кратчайшего пути (например, алгоритм Дейкстры)?
9. Какие алгоритмы могут быть использованы для нахождения минимального остовного дерева (например, алгоритм Краскала)?

ТЕМА 15. ПРОБЛЕМЫ ХЕШИРОВАНИЯ

Рассмотрим задачу про европейские столицы, в условиях которой указаны пары — город и дата основания. Зная дату, мы можем определить город. Для решения задачи используем следующую таблицу:

Таблица 1.

881	Вена	1237	Берлин	1323	Вильнюс
1614	Тирана	1167	Копенгаген	963	Люксембург
1067	Минск	1191	Дублин	1275	Амстердам
752	Ватикан	1786	Рейкьявик	1200	Варшава
1872	Будапешт	856	Мадрид	1147	Москва

Пары с годом и городом в таблице расположены в произвольном порядке, поэтому поиск нужной столицы требует полного перебора.

Временная сложность такого алгоритма составляет O(N), что довольно медленно. Бинарный поиск позволяет сократить это время до O(logN) . На большой таблице из миллиона записей среднее количество сравнений для линейного поиска равно 500000, а для бинарного — всего 10.

Но и это не предел. Если поиск нужно выполнять с максимальной скоростью, мы можем воспользоваться еще одной структурой данных. Это **хэш-таблица**, которая обеспечивает временную сложность O(1).

Хэш-таблица похожа на массив, потому что в ней тоже есть операция индексации. В JavaScript доступ к элементу массива осуществляется по индексу, записанному в квадратных скобках. В качестве индекса можно использовать последовательные целые числа. В случае с хэш-таблицей в качестве индекса можно брать любые структуры данных — строки, дату и время, массивы.

Рассмотрим как устроены хэш-таблицы и как их реализовывать.

Разреженные массивы

Простейший способ определить столицу по году основания — использовать массив:

```
let capitals = [];

capitals[881] = 'Вена';
capitals[1237] = 'Берлин';
capitals[1323] = 'Вильнюс';
capitals[1614] = 'Тирана';
capitals[1167] = 'Копенгаген';
capitals[963] = 'Люксембург';
capitals[1067] = 'Минск';
capitals[1191] = 'Дублин';
```

```
capitals[1275] = 'Амстердам';
capitals[752] = 'Ватикан';
capitals[1786] = 'Рейкьявик';
capitals[1200] = 'Варшава';
capitals[1872] = 'Будапешт';
capitals[856] = 'Мадрид';
capitals[1147] = 'Москва';
```

Такой массив занимает много памяти. В нем хранится всего 15 городов, но при этом размер массива — это целых 1873 элемента:

```
console.log(capitals.length); // => 1873
```

Так получилось потому, что индексами в массиве могут быть только последовательные целые числа, начиная с 0. Если мы добавляем элемент с индексом 752, в массив добавляются и неопределенные элементы с индексами от 0 до 751. Длина массива получилась равной 1873, потому что:

- Наибольший год основания — 1872
- Первым в массиве всегда стоит элемент с индексом 0

Если мы применим массив, доступ к элементам выполнится очень быстро — за константное время . Но при этом мы впустую расходуем память — в нашем примере массив заполнен всего на 0,8%.

Массивы, в которых большая часть элементов не определена, называются **разреженными**. Чтобы сэкономить память, программисты стараются уплотнить массив. Скажем, в нашем примере годы находятся достаточно далеко друг от друга — на каждые двадцать пустых элементов приходится только один элемент с данными. Мы можем уменьшить массив в двадцать раз с помощью простого трюка.

Возьмем числа 0, 1, 2 и 3, поделим их на 20 и получим такой результат:

- $0 \div 20 = 0$
- $1 \div 20 = 0,05$
- $2 \div 20 = 0,1$
- $3 \div 20 = 0,15$

Целая часть этих чисел будет равна 0, пока мы не доберемся до такой операции:

$$19 \div 20 = 0,95$$

Начиная с 20, целая часть после деления будет равна 1:

- $20 \div 20 = 1$
- $21 \div 20 = 1,05$

Начиная с 40, целая часть будет равна 2.

Получается, что после преобразования числа схлопываются:

- От 0 до 19 → 0
- От 20 до 39 → 1
- От 40 до 59 → 2 и так далее

Чтобы получать целую часть, воспользуемся стандартной функцией Math.floor:

```
let capitals = [];

capitals[Math.floor(881 / 20)] = 'Вена';
capitals[Math.floor(1237 / 20)] = 'Берлин';
capitals[Math.floor(1323 / 20)] = 'Вильнюс';
capitals[Math.floor(1614 / 20)] = 'Тирана';
capitals[Math.floor(1167 / 20)] = 'Копенгаген';
capitals[Math.floor(963 / 20)] = 'Люксембург';
capitals[Math.floor(1067 / 20)] = 'Минск';
capitals[Math.floor(1191 / 20)] = 'Дублин';
capitals[Math.floor(1275 / 20)] = 'Амстердам';
capitals[Math.floor(752 / 20)] = 'Ватикан';
capitals[Math.floor(1786 / 20)] = 'Рейкьявик';
capitals[Math.floor(1200 / 20)] = 'Варшава';
capitals[Math.floor(1872 / 20)] = 'Будапешт';
capitals[Math.floor(856 / 20)] = 'Мадрид';
capitals[Math.floor(1147 / 20)] = 'Москва';
```

Чтобы узнать, какой город основан в 1191 году, мы проверяем элемент с индексом 1191/20:

```
const getCity = (year) => capitals[Math.floor(year
/ 20)];

let city = getCity(1191); // Дублин
```

У такого подхода есть недостаток — довольно трудно предсказать размер массива. Когда речь идет о годах основания, мы можем проанализировать данные и прикинуть, что каждые двадцать лет можно схлопнуть в один элемент.

Для чисел 0 до 1999 нам достаточно массива из ста элементов. Последний элемент такого уплотненного массива имеет индекс 99, а Math.floor(1999/20) как раз равно 99.

Но в другой задаче это может не сработать — например, если требуется хранить в массиве индексы, равные миллиону или миллиарду. Коэффициент 20 слишком мал для таких чисел, потому что даже уплотненный массив все равно будет слишком большим и слишком пустым.

Но мы не можем анализировать каждую возникающую задачу. Нам бы хотелось иметь универсальный способ, подходящий для разных случаев и позволяющий контролировать размер массива. Поэтому мы рассмотрим еще один способ.

Модульная арифметика

Предположим, мы фиксируем размер массива. Пусть у нас будет массив из ста элементов:

```
let capitals = new Array(100);
```

Мы хотим сохранить в него несколько элементов, но их индексы могут быть произвольными целыми числами. Чтобы преобразовать индекс в число от 0 до 99, мы должны вычислить остаток от деления индекса на 100.

Иногда вместо «остаток от деления на » говорят «модуль по основанию ». В JavaScript и многих других языках программирования для вычисления модуля используют оператор %:

```
console.log(1999 % 20); // => 19
```

Решим нашу задачу со столицами с помощью модуля:

```
let capitals = new Array(100);

capitals[881 % 100] = 'Вена';
capitals[1237 % 100] = 'Берлин';
capitals[1323 % 100] = 'Вильнюс';
capitals[1614 % 100] = 'Тирана';
capitals[1167 % 100] = 'Копенгаген';
capitals[963 % 100] = 'Люксембург';
capitals[1067 % 100] = 'Минск';
capitals[1191 % 100] = 'Дублин';
capitals[1275 % 100] = 'Амстердам';
capitals[752 % 100] = 'Ватикан';
capitals[1786 % 100] = 'Рейкьявик';
capitals[1200 % 100] = 'Варшава';
capitals[1872 % 100] = 'Будапешт';
capitals[856 % 100] = 'Мадрид';
capitals[1147 % 100] = 'Москва';

const getCity = (year) => capitals[year % 100];

let city = getCity(1167); // Копенгаген
```

Теперь мы точно знаем, что наш массив не вырастет до больших размеров, как это может случиться с операцией деления. Но мы можем обнаружить другую проблему.

Попробуем узнать, какой город основан в 1167 году. Если верить нашей таблице, это Копенгаген. Но программа говорит совсем другое:

```
console.log(getCity(1167)); // => Минск
```

Минск основан в 1067 году, а Копенгаген — в 1167. Годы отличаются, но у них один и тот же остаток от деления на 100, а именно 67.

Ситуация, когда разные элементы после преобразования индекса попадают в одну и ту же ячейку массива, называется **коллизией**. На самом деле это не ошибка, а вполне вероятная ситуация, хотя и не очень частая.

Коллизии

Справиться с коллизиями нетрудно. Вместо того чтобы хранить в каждой ячейке массива простое значение, мы размещаем там односвязный список. При добавлении элемента мы добавляем в список пару из года и города:

- Год — это индекс, по которому мы сохраняем и извлекаем данные. Обычно его называют **ключом**
- Название — это значение

Попробуем реализовать в коде:

```
import LinkedList from './LinkedList.js';

let capitals = new Array(100);

const setCapital = (year, city) => {
  let index = year % capitals.length;
  if (typeof(capitals[index]) === 'undefined') {
    capitals[index] = new LinkedList();
  }

  capitals[index].add({ key: year, value: city });
};
```

Мы уже разработали структуру данных LinkedList, поэтому мы можем просто импортировать ее. Размер массива capitals всегда будет равен ста элементам.

По умолчанию все ячейки массива пусты, и связный список создается только при первой записи в ячейку. В каждом списке может храниться несколько элементов. Чтобы различать их, мы сохраняем не просто значение, а пару из ключа (key) и значения (value).

При поиске точно также вычисляем индекс:

```
const getCapital = (year) => {
  let index = year % capitals.length;
  if (typeof(capitals[index]) === 'undefined') {
```

```
    return undefined;
  }
  for (let pair of capitals[index]) {
    if (pair.key === year) {
      return pair.value;
    }
  }

  return undefined;
};
```

Если в ячейке ничего нет, значит, мы ничего и не записывали.

Но если в ячейке что-то есть, то это список, по которому мы пробегаем и пытаемся найти пару с заданным ключом (*key*). Обнаружив пару, возвращаем ее значение (*value*).

Алгоритмическая сложность этих операций зависит от того, насколько часто элементы попадают в одну и ту же ячейку — в один и тот же связный список.

Если у нас будет случайный набор из ста чисел, он достаточно равномерно распределится по массиву: в большинстве ячеек будет храниться одно число, в некоторых — два, и в некоторых — ни одного. Алгоритмическая сложность записи и чтения в таком случае будет близка O(1) к — это один из самых быстрых возможных алгоритмов.

Если числа не будут случайными, может получиться так, что в одном из списков их окажется слишком много — тогда скорость вставки и проверки значительно снизится. Подробнее мы обсудим эту ситуацию позже, а пока запомним, что предложенный нами алгоритм любит равномерные случайные ключи.

Хэш-функция

Теперь попробуем решить обратную задачу — определять год основания столицы, зная ее название. Заглянем в таблицу 1, приведенную в начале, и вспомним, что год основания Мадрида — 856, а Москвы — 1147.

Наша структура данных может хранить значения с любым целочисленным ключом, не только с годом. Но «Мадрид» и «Москва» — это не числа, а строки.

При этом компьютеры умеют работать только с числами, и любые объекты в них хранятся как последовательности чисел. Каждая буква хранится как одно число, которое называют **кодом символа**:

```
const capital = 'Мадрид';
for (let i = 0; i < capital.length; i++) {
  console.log(capital.charCodeAt(i));
}
```

```
// => 1052
// => 1072
// => 1076
// => 1088
// => 1080
// => 1076
```

Слово «Мадрид» хранится как последовательность чисел 1052, 1072, 1076, 1088, 1080 и 1076. В простейшем случае мы могли бы использовать для вычисления индекса код первой буквы — в нашем примере 1052. Но это значит все города на М (Мадрид, Москва и Минск) попадут в один и тот же список. В таком случае скорость поиска будет не очень высокой.

Чтобы этого избежать, нам нужны ключи, которые очень похожи на случайные. Мы можем взять все наши числа и подвергнуть их какой-нибудь обработке — например, вычислить их сумму или произведение.

Однако сложение и умножение — слишком простые операции. Все слова, состоящие из одних и тех же букв (ведро, вроде и древо), будут иметь один и тот же ключ — сумму или произведение кодов символов.

Наихудший случай

Выше мы говорили, что обычно вставка и поиск в хеш-таблице выполняются за время O(1). Но если хеш-функция выбрана неудачно, все значения могут оказаться в одном связном списке во внутреннем массиве.

Тогда время поиска составит O(N), что обычно считается медленным. К счастью, эта ситуация практически невозможна. Тем не менее, если вы соберетесь писать собственную хеш-таблицу, посвятите время выбору хорошей хеш-функции.

Есть еще одна возможная сложность. В массиве, который мы использовали в примерах выше, мы резервировали сто элементов, потому что рассчитывали, что данные будут распределены равномерно. Однако если мы поместим в такой массив 10000 элементов, то даже при идеальном распределении в каждом связном списке окажется сто элементов. Это значит, что поиск и вставка все-таки будут медленными.

Обычно подсчитывают **коэффициент заполнения хеш-таблицы** (load factor) — это частное от деления количества вставленных элементов на размер внутреннего массива. Если он достигает заранее выбранного порога в 60–80%, внутренний массив увеличивают вдвое, а индексы всех элементов пересчитывают.

Точно также, если коэффициент заполнения становится слишком маленьким, 20–25%, массив уменьшают вдвое. У массива есть нижняя граница в 128 или 256 элементов, чтобы не перестраивать его слишком часто, пока он маленький.

Хэш-таблицы и функция вставки

Перепишем функцию вставки, опираясь на все новые знания. Вместе с этим реализуем хеш-таблицу в виде новой структуры данных, спрятав внутри класса массив с данными:

```
class Hash {
  let table = [];
  let count = 0;

  hash(s) {
    const k = 65537;
    const m = 2**20;

    let result = 0, power = 1;
    for (let i = 0; i < s.length; i++) {
      result = (result + power * s.charCodeAt(i)) %
m;
      power = (power * k) % m;
    }

    return result;
  }

  calculateIndex(table, key) {
    return this.hash(String(key)) % table.length;
  }

  rebuildTableIfNeed() {
    if (this.table.length == 0) {
      this.table.length = 128;
    } else {
      const loadFactor =
this.count/this.table.length;
      if (loadFactor >= 0.8) {
        let newTable = new Array(this.table.length
* 2);
        for (let list in this.table) {
          for (let pair of list) {
            const newIndex =
this.calculateIndex(newTable, pair.key)
            if (typeof(newTable[newIndex]) ===
'undefined') {
```

```
              newTable[newIndex] = new
LinkedList();
            }

            newTable[newIndex].add(pair);
          }
        }

        this.table = newTable;
      }
    }
  }

  set(key, value) {
    this.rebuildTableIfNeed();

    const index = this.calculateIndex(this.table,
key);
    if (typeof(this.table[index]) === 'undefined')
{
      this.table[index] = new LinkedList();
    }

    this.table[index].add({ key: key, value: value
});
    this.count += 1;
  }

  get(key) {
    const index = this.calculateIndex(this.table,
key);
    if (typeof(this.table[index].fore()) ===
'undefined') {
      return undefined;
    }

    for (let pair of this.table[index]) {
      if (pair.key === key) {
        return pair.value;
      }
    }
    return undefined;
```

```
  }
}
```

Весь код из этого примера мы уже обсуждали, кроме удвоения массива. Создавая новый массив, в два раза больше текущего, мы должны скопировать все элементы, пересчитав их индексы.

Именно за это отвечает вложенный цикл:

```
let newTable = new Array(this.table.length * 2);
for (let list in this.table) {
  for (let pair of list) {
    const newIndex = this.calculateIndex(newTable,
pair.key)
    if (typeof(newTable[newIndex]) === 'undefined')
{
      newTable[newIndex] = new LinkedList();
    }
    newTable[newIndex].add(pair);
  }
}
```

В самом начале внутренний массив имеет размер 0, чтобы не тратить память, если в хеш-таблице нет ни одного элемента. При вставке первого элемента мы увеличиваем внутренний массив сразу до 128 элементов.

Контрольные вопросы

1. Что такое хеширование и для чего оно используется?
2. Как работает хеш-функция и какие основные свойства она должна иметь?
3. Что такое хеш-таблица и каковы ее основные операции (вставка, удаление, поиск)?
4. Какие существуют методы разрешения коллизий в хеш-таблицах?
5. В чем разница между открытой и закрытой адресацией?
6. Каковы преимущества и недостатки хеширования по сравнению с другими структурами данных?
7. Что такое фактор загрузки в контексте хеш-таблиц и как он влияет на производительность?
8. Как можно оценить производительность хеш-таблицы при различных условиях?
9. Приведите пример хеш-функции и объясните, как она работает.
10. В каких ситуациях хеширование не рекомендуется использовать?

ПРАКТИЧЕСКОЕ ЗАНЯТИЕ 1. ТИПЫ ДАННЫХ. РАСПРЕДЕЛЕНИЕ ПАМЯТИ В ПРОГРАММИРОВАНИИ

Цели:

Студенты получат теоретические знания о типах данных и аллокации памяти в программировании, а также научатся применять их на практике.

Материалы:

1. Средства программирования (Python, C или Java)
2. Компьютеры
3. Диаграммы для демонстрации оперативной и дисковой памяти
4. Руководства и примеры кода

План занятия:

1. Введение

- Что такое типы данных и аллокация памяти?
- Разнообразие типов данных в программировании и их особенности.

2. Типы данных

- **Простые типы данных**:
  - Целые числа (int)
  - Числа с плавающей точкой (float)
  - Символы (char)
  - Логические (boolean)
- **Сложные типы данных**:
  - Массивы (array)
  - Списки (list)
  - Структуры (struct)
  - Переменные (variables)

Практика: Приведите краткие примеры для каждого типа данных. Попросите студентов написать код, используя указанный язык.

3. Аллокация памяти

- Что такое аллокация памяти в программировании?
- **Стек** и **куча**: их различия и применение.
- Как происходит аллокация памяти при объявлении переменных.

Практика: Студенты пишут следующие примеры кода:

Python:

```
# Простые типы данных
a = 10          # Целое число
b = 3.14        # Число с плавающей точкой
c = 'A'         # Символ
d = True        # Логическое

print(type(a), type(b), type(c), type(d))
```

C:
```
#include <stdio.h>
int main() {
    int a = 10;          // Целое число
    float b = 3.14;     // Число с плавающей точкой
    char c = 'A';       // Символ
    int d = 1;          // Логическое
    printf("a: %d, b: %.2f, c: %c, d: %d\n", a, b,
c, d);
    return 0;
}
```
4. Практика аллокации памяти (30 минут)

- Показать студентам, как происходит аллокация памяти при создании переменных и массивов.

Пример на C:
```
#include <stdio.h>
#include <stdlib.h>

int main() {
    int *array;
    int n;
    printf("Введите размер массива: ");
    scanf("%d", &n);
    // Аллокация памяти в куче
    array = (int *)malloc(n * sizeof(int));

    // Ввод значений в массив
    for (int i = 0; i < n; i++) {
        array[i] = i * 2; // Присвоение значений
каждому элементу
    }
    // Вывод элементов массива
    for (int i = 0; i < n; i++) {
        printf("%d ", array[i]);
    }
    // Освобождение памяти
    free(array);
    return 0;
}
```
5. Вопросы и ответы

- Задайте студентам вопросы о типах данных и аллокации памяти. Обсуждение.

6. Заключение

- Повторите полученные знания.
- Обсудите, как применять типы данных и аллокацию памяти в будущем программировании.

ПРАКТИЧЕСКОЕ ЗАНЯТИЕ 2.
ВИДЫ АЛГОРИТМОВ. РАБОТА С ЛИНЕЙНЫМИ, РАЗВЕТВЛЯЮЩИМИСЯ И ПОВТОРЯЮЩИМИСЯ АЛГОРИТМАМИ

Цели:

Студенты получат теоретические знания о различных типах алгоритмов и научатся применять их на практике.

Материалы:

1. Средства программирования (Python, C, Java или другой язык)
2. Компьютеры
3. Схемы и диаграммы для визуализации алгоритмов
4. Примеры кода и задачи для практики

План занятия:

1. Введение

- Определение алгоритма и его роль в программировании.
- Разнообразие типов алгоритмов: линейные, ветвящиеся и итеративные.

2. Линейные алгоритмы

- Определение линейного алгоритма.
- Примеры линейных алгоритмов (например, суммирование чисел, поиск максимального значения в массиве).

Практика: Напишите код для линейного алгоритма, который находит сумму чисел в массиве.

Пример на Python:

```
def sum_array(arr):
    total = 0
    for number in arr:
        total += number
    return total

numbers = [1, 2, 3, 4, 5]
print("Сумма чисел:", sum_array(numbers))
```

3. Ветвящиеся алгоритмы

- Определение ветвящегося алгоритма.

- Примеры ветвящихся алгоритмов (например, алгоритм проверки четности числа).

Практика: Напишите код, который определяет, является ли введенное число четным или нечетным.

Пример на C:

```
#include <stdio.h>

int main() {
    int number;
    printf("Введите число: ");
    scanf("%d", &number);
    if (number % 2 == 0) {
        printf("Число четное\n");
    } else {
        printf("Число нечетное\n");
    }

    return 0;
}
```

4. Итеративные алгоритмы

- Определение итеративного алгоритма.
- Примеры итеративных алгоритмов (например, вычисление факториала).

Практика: Напишите код для вычисления факториала числа с использованием итерации.

Пример на Java:

```
import java.util.Scanner;
public class Factorial {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        System.out.print("Введите число: ");
        int n = scanner.nextInt();
        int factorial = 1;
        for (int i = 1; i <= n; i++) {
            factorial *= i;
        }
```

```
        System.out.println("Факториал числа " + n +
" равен: " + factorial);
    }
}
```

5. Сравнение алгоритмов

- Обсудите преимущества и недостатки каждого типа алгоритма.
- Когда лучше использовать линейные, ветвящиеся и итеративные алгоритмы.

6. Вопросы и ответы

- Ответьте на вопросы студентов о типах алгоритмов и их применении.

7. Заключение

- Подведите итоги занятия.
- Обсудите, как знание типов алгоритмов может помочь в будущем программиров

ПРАКТИЧЕСКОЕ ЗАНЯТИЕ 3.
СТЕК КАК ADT, МЕТОДЫ СТРУКТУРЫ ДАННЫХ СТЕКА

Цели:

Студенты получат теоретические знания о стеке как ADT и научатся применять его методы на практике.

Материалы:

1. Средства программирования (Python, C, Java или другой язык)
2. Компьютеры
3. Схемы и диаграммы для визуализации работы стека
4. Примеры кода для практики

План занятия:

1. Введение

- Определение стека и его особенности как ADT.
- Принцип работы стека (LIFO — Last In, First Out).
- Операции стека: push, pop, peek, isEmpty, size.

2. Реализация стека (30 минут)

- Пример реализации стека на разных языках программирования.

Пример на Python:

```
class Stack:
    def __init__(self):
        self.items = []
    def push(self, item):
        self.items.append(item)
    def pop(self):
        if not self.is_empty():
            return self.items.pop()
        raise IndexError("Stack is empty")
    def peek(self):
        if not self.is_empty():
            return self.items[-1]
        raise IndexError("Stack is empty")

    def is_empty(self):
        return len(self.items) == 0
    def size(self):
        return len(self.items)

# Пример использования стека
stack = Stack()
stack.push(1)
stack.push(2)
```

```
stack.push(3)
print("Текущий элемент:", stack.peek())
print("Удаление элемента:", stack.pop())
print("Размер стека:", stack.size())
```

Пример на C:

```
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#define MAX 100
typedef struct {
    int items[MAX];
    int top;
} Stack;
void init(Stack *s) {
    s->top = -1;
}
bool is_empty(Stack *s) {
    return s->top == -1;
}
bool is_full(Stack *s) {
    return s->top == MAX - 1;
}
void push(Stack *s, int item) {
    if (!is_full(s)) {
        s->items[++s->top] = item;
    } else {
        printf("Стек переполнен\n");
    }
}
int pop(Stack *s) {
    if (!is_empty(s)) {
        return s->items[s->top--];
    }
    printf("Стек пуст\n");
    return -1; // Возврат значения ошибки
}
int peek(Stack *s) {
    if (!is_empty(s)) {
        return s->items[s->top];
    }
    printf("Стек пуст\n");
```

```
    return -1; // Возврат значения ошибки
}
// Пример использования стека
int main() {
    Stack stack;
    init(&stack);
    push(&stack, 1);
    push(&stack, 2);
    push(&stack, 3);
    printf("Текущий элемент: %d\n", peek(&stack));
    printf("Удаление элемента: %d\n", pop(&stack));
    return 0;  }
```

3. Применение Стека

- Обсуждение случаев использования стека:
  - Обратный порядок данных
  - Реализация рекурсии
  - Проверка правильности скобочной последовательности

4. Практика

- Студенты пишут программу, которая использует стек для проверки правильности скобочной последовательности (например, (([])).

Пример на Python:

```
def is_balanced(expression):
    stack = Stack()
    brackets = {')': '(', ']': '[', '}': '{'}
    for char in expression:
        if char in brackets.values():
            stack.push(char)
        elif char in brackets.keys():
            if stack.is_empty() or stack.pop() !=
brackets[char]:
                return False
    return stack.is_empty()

# Пример использования
expr = "(([]))"
print("Скобочная последовательность правильная:",
is_balanced(expr))
```

5. Вопросы и ответы

- Ответьте на вопросы студентов о стеке и его методах.

6. Заключение

- Подведите итоги занятия.
- Обсудите, как знание стека и его методов может помочь в будущем программировании.

ПРАКТИЧЕСКОЕ ЗАНЯТИЕ 4.
ОЧЕРЕДЬ КАК ADT, ОСНОВНЫЕ МЕТОДЫ ОЧЕРЕДИ, ЛИНЕЙНАЯ И КРУГОВАЯ ОЧЕРЕДЬ И ИХ ПРИМЕНЕНИЕ

Цели:

Студенты получат теоретические знания о очереди как ADT и научатся применять ее методы на практике.

Материалы:

1. Средства программирования (Python, C, Java или другой язык)
2. Компьютеры
3. Схемы и диаграммы для визуализации работы очереди
4. Примеры кода для практики

План занятия:

1. Введение

- Определение очереди и ее особенности как ADT.
- Принцип работы очереди (FIFO — First In, First Out).
- Операции очереди: enqueue, dequeue, peek, isEmpty, size.

2. Реализация очереди (30 минут)

- Пример реализации линейной и циклической очереди на разных языках программирования.

Пример линейной очереди на Python:

```
class Queue:
    def __init__(self):
        self.items = []
    def enqueue(self, item):
        self.items.append(item)
    def dequeue(self):
        if not self.is_empty():
            return self.items.pop(0)
        raise IndexError("Очередь пуста")
    def peek(self):
        if not self.is_empty():
            return self.items[0]
        raise IndexError("Очередь пуста")
    def is_empty(self):
        return len(self.items) == 0
    def size(self):
        return len(self.items)

# Пример использования очереди
queue = Queue()
queue.enqueue(1)
```

```
queue.enqueue(2)
queue.enqueue(3)
print("Первый элемент:", queue.peek())
print("Удаление элемента:", queue.dequeue())
print("Размер очереди:", queue.size())
```

Пример циклической очереди на C:

```
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#define MAX 100

typedef struct {
    int items[MAX];
    int front, rear;
} CircularQueue;

void init(CircularQueue *q) {
    q->front = -1;
    q->rear = -1;
}
bool is_empty(CircularQueue *q) {
    return q->front == -1;
}
bool is_full(CircularQueue *q) {
    return (q->rear + 1) % MAX == q->front;
}
void enqueue(CircularQueue *q, int item) {
    if (!is_full(q)) {
        if (is_empty(q)) {
            q->front = 0;
        }
        q->rear = (q->rear + 1) % MAX;
        q->items[q->rear] = item;
    } else {
        printf("Очередь переполнена\n");
    }
}

int dequeue(CircularQueue *q) {
    if (!is_empty(q)) {
        int item = q->items[q->front];
```

```
        if (q->front == q->rear) {
            q->front = -1;
            q->rear = -1;
        } else {
            q->front = (q->front + 1) % MAX;
        }
        return item;
    }
    printf("Очередь пуста\n");
    return -1; // Возврат значения ошибки
}
// Пример использования очереди
int main() {
    CircularQueue queue;
    init(&queue);
    enqueue(&queue, 1);
    enqueue(&queue, 2);
    enqueue(&queue, 3);
    printf("Удаление элемента: %d\n",
dequeue(&queue));
    return 0;
}
```

3. Применение очереди

- Обсуждение случаев использования очереди:
  - Управление задачами (очереди задач)
  - Сетевые принтеры и другие устройства, работающие по принципу очереди
  - Алгоритмы обработки данных (например, BFS в графах)

4. Практика

- Студенты пишут программу, которая моделирует очередь клиентов в магазине, позволяя им добавлять и удалять клиентов из очереди.

Пример на Python:

```
class CustomerQueue:
    def __init__(self):
        self.queue = Queue()
    def add_customer(self, customer_name):
        self.queue.enqueue(customer_name)
        print(f"Клиент {customer_name} добавлен в
очередь.")
    def serve_customer(self):
        if not self.queue.is_empty():
```

```
            customer = self.queue.dequeue()
            print(f"Обслуживаем клиента
{customer}.")
        else:
            print("Очередь пуста.")

# Пример использования очереди клиентов
shop_queue = CustomerQueue()
shop_queue.add_customer("Алексей")
shop_queue.add_customer("Мария")
shop_queue.serve_customer()
```

5. Вопросы и ответы

- Ответьте на вопросы студентов о очереди и ее методах.

6. Заключение

- Подведите итоги занятия.
- Обсудите, как знание очереди и ее методов может помочь в будущем программировании.

ПРАКТИЧЕСКОЕ ЗАНЯТИЕ 5.
СТРУКТУРА ДАННЫХ СТРОК. ОСНОВНЫЕ МЕТОДЫ И АЛГОРИТМЫ СТРУКТУРЫ ДАННЫХ СТРОК

Цели:

Студенты получат теоретические знания о строковых структурах данных и научатся применять их методы на практике.

Материалы:

1. Средства программирования (Python, C, Java или другой язык)
2. Компьютеры
3. Схемы и диаграммы для визуализации работы со строками
4. Примеры кода для практики

План занятия:

1. Введение

- Определение строковой структуры данных.
- Основные характеристики строк (неизменяемость в некоторых языках, порядок символов).
- Применение строк в программировании.

2. Основные методы строк

- length() - получение длины строки.
- substring(start, end) - извлечение подстроки.
- indexOf(substring) - поиск подстроки.
- toUpperCase() и toLowerCase() - преобразование регистра.
- split(delimiter) - разбиение строки на массив.

Практика: Напишите код, который демонстрирует использование этих методов.

Пример на Python:

```
text = "Hello, World!"
# Получение длины строки
print("Длина строки:", len(text))
# Извлечение подстроки
substring = text[0:5]
print("Подстрока:", substring)
# Поиск подстроки
index = text.index("World")
print("Индекс подстроки 'World':", index)
# Преобразование регистра
print("Верхний регистр:", text.upper())
print("Нижний регистр:", text.lower())
# Разбиение строки
words = text.split(", ")
print("Слова в строке:", words)
```

3. Алгоритмы работы со строками (30 минут)

- Алгоритм поиска подстроки: алгоритм Кнута-Морриса-Пратта (KMP).
- Алгоритм подсчета вхождений символа в строку.

Пример алгоритма поиска подстроки на Python:

```
def kmp_search(pattern, text):
    # Подготовка префиксной таблицы
    lps = [0] * len(pattern)
    j = 0  # индекс для pattern
    i = 1  # индекс для lps
    while i < len(pattern):
        if pattern[i] == pattern[j]:
            j += 1
            lps[i] = j
            i += 1
        else:
            if j != 0:
                j = lps[j - 1]
            else:
                lps[i] = 0
                i += 1
    # Поиск подстроки
    i = 0  # индекс для text
    j = 0  # индекс для pattern
    while i < len(text):
        if pattern[j] == text[i]:
            i += 1
            j += 1
        if j == len(pattern):
            print("Подстрока найдена в индексе:", i
- j)
            j = lps[j - 1]
        elif i < len(text) and pattern[j] !=
text[i]:
            if j != 0:
                j = lps[j - 1]
            else:
                i += 1

# Пример использования
text = "ABC ABCDAB ABCDABCDABDE"
```

```
pattern = "ABCDABD"
kmp_search(pattern, text)
```

4. Практика

- Студенты пишут программу, которая выполняет следующие задачи:
  - Находит все вхождения подстроки в строке.
  - Подсчитывает количество вхождений символа в строку.

Пример на Python:

```
def count_occurrences(text, char):
    return text.count(char)
text = "Hello, World!"
char = 'o'
print(f"Количество вхождений символа '{char}':",
count_occurrences(text, char))
```

5. Вопросы и ответы

- Ответьте на вопросы студентов о строках, их методах и алгоритмах.

6. Заключение

- Подведите итоги занятия.
- Обсудите, как знание строковых структур данных может помочь в будущем программировании.

ПРАКТИЧЕСКОЕ ЗАНЯТИЕ 6.
СТАТИЧЕСКАЯ И ДИНАМИЧЕСКАЯ СТРУКТУРА СПИСКОВ, ОСНОВНЫЕ МЕТОДЫ РАБОТЫ СО СПИСКАМИ

Цели:

Студенты получат теоретические знания о статических и динамических списках, а также научатся применять основные методы работы с ними на практике.

Материалы:

1. Средства программирования (Python, C, Java или другой язык)
2. Компьютеры
3. Схемы и диаграммы для визуализации структур списков
4. Примеры кода для практики

План занятия:

1. Введение

- Определение статических и динамических списков.
- Отличия между статическими и динамическими структурами данных.
- Примеры использования списков в программировании.

2. Статические списки

- Определение статического списка.
- Преимущества и недостатки статических списков.

Пример реализации статического списка на C:

```
#include <stdio.h>
#define MAX_SIZE 5
typedef struct {
    int items[MAX_SIZE];
    int size;
} StaticList;
void init_list(StaticList *list) {
    list->size = 0;
}

void add(StaticList *list, int item) {
    if (list->size < MAX_SIZE) {
        list->items[list->size++] = item;
    } else {
        printf("Список переполнен\n");
    }
}
void display(StaticList *list) {
    for (int i = 0; i < list->size; i++) {
```

```
        printf("%d ", list->items[i]);
    }
    printf("\n");
}
// Пример использования
int main() {
    StaticList list;
    init_list(&list);
    add(&list, 1);
    add(&list, 2);
    add(&list, 3);
    display(&list);
    return 0;
}
```

3. Динамические списки

- Определение динамического списка.
- Преимущества и недостатки динамических списков.

Пример реализации динамического списка на Python:

```
class DynamicList:
    def __init__(self):
        self.items = []
    def add(self, item):
        self.items.append(item)
    def display(self):
        for item in self.items:
            print(item, end=' ')
        print()
# Пример использования
dynamic_list = DynamicList()
dynamic_list.add(1)
dynamic_list.add(2)
dynamic_list.add(3)
dynamic_list.display()
```

4. Основные методы списков

- Обсуждение основных методов работы со списками:
  - Добавление элементов
  - Удаление элементов
  - Поиск элементов
  - Сортировка элементов

Пример метода для удаления элемента из динамического списка на Python:

```
def remove_item(self, item):
    if item in self.items:
        self.items.remove(item)
    else:
        print("Элемент не найден")
# Добавьте метод в класс DynamicList и
протестируйте его
```

5. Практика

Студенты пишут программу, которая реализует динамический список и использует все основные методы: добавление, удаление и отображение элементов.

Пример задания:

1. Создайте динамический список.
2. Реализуйте методы для добавления, удаления и отображения элементов.
3. Напишите код для поиска элемента в списке.

6. Вопросы и ответы

Ответьте на вопросы студентов о статических и динамических списках и их методах.

7. Заключение

- Подведите итоги занятия.
- Обсудите, как знание списков и их методов может помочь в будущем программировании.

ПРАКТИЧЕСКОЕ ЗАНЯТИЕ 7.
МАССИВЫ. РАБОТА С МАССИВАМИ. СТРУКТУРА ДАННЫХ МАТРИЦЫ И ОПЕРАЦИИ НАД НИМИ

Цели:

Студенты получат теоретические знания о массивах и матрицах, а также научатся работать с ними на практике.

Материалы:

1. Средства программирования (Python, C, Java или другой язык)
2. Компьютеры
3. Диаграммы и схемы для визуализации массивов и матриц
4. Примеры кода для практики

План занятия:

1. Введение

- Определение массива и его характеристики.
- Применение массивов в программировании.
- Введение в матрицы: определение и применение.

2. Работа с массивами (30 минут)

- Пример создания и инициализации массива.

Пример на Python:

```
# Создание и инициализация массива
array = [1, 2, 3, 4, 5]
# Доступ к элементам массива
print("Первый элемент:", array[0])
print("Последний элемент:", array[-1])
# Итерация по массиву
for number in array:
    print("Элемент массива:", number)
```

Пример на C:

```
#include <stdio.h>
int main() {
    int array[] = {1, 2, 3, 4, 5};
    int length = sizeof(array) / sizeof(array[0]);
    // Доступ к элементам массива
    printf("Первый элемент: %d\n", array[0]);
    printf("Последний элемент: %d\n", array[length
- 1]);
    // Итерация по массиву
    for (int i = 0; i < length; i++) {
        printf("Элемент массива: %d\n", array[i]);
    }
    return 0;
```

```
}
```

3. Операции над Массивами

- Примеры операций: поиск, сортировка, добавление и удаление элементов.

Пример на Python: Поиск максимального элемента:

```
def find_max(arr):
    max_value = arr[0]
    for number in arr:
        if number > max_value:
            max_value = number
    return max_value
array = [3, 1, 4, 1, 5, 9, 2]
print("Максимальный элемент:", find_max(array))
```

4. Введение в матрицы

- Определение матрицы и её характеристики.
- Создание и инициализация матрицы.

Пример на Python:

```
# Создание и инициализация матрицы
matrix = [
    [1, 2, 3],
    [4, 5, 6],
    [7, 8, 9]
]
# Доступ к элементам матрицы
print("Элемент в первой строке и втором столбце:",
matrix[0][1])
# Итерация по матрице
for row in matrix:
    for value in row:
        print("Элемент матрицы:", value)
```

5. Операции над матрицами (30 минут)

- Примеры операций над матрицами: сложение, умножение, транспонирование.

Пример на Python: Сложение двух матриц:

```
def add_matrices(matrix1, matrix2):
    result = []
    for i in range(len(matrix1)):
        row = []
        for j in range(len(matrix1[i])):
```

```
            row.append(matrix1[i][j] +
matrix2[i][j])
        result.append(row)
    return result
matrix1 = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
matrix2 = [[9, 8, 7], [6, 5, 4], [3, 2, 1]]
result_matrix = add_matrices(matrix1, matrix2)
print("Результат сложения матриц:")
for row in result_matrix:
    print(row)
```

6. Вопросы и ответы

- Ответьте на вопросы студентов о массивах и матрицах.

7. Заключение

- Подведите итоги занятия.
- Обсудите, как использование массивов и матриц может помочь в будущем программировании.

ПРАКТИЧЕСКОЕ ЗАНЯТИЕ 8.
СВЯЗАННЫЕ СПИСКИ. РАБОТА С ОСНОВНЫМИ МЕТОДАМИ В СВЯЗАННЫХ СПИСКАХ

Цели:

Студенты получат теоретические знания о связных списках и научатся применять их методы на практике.

Материалы:

1. Средства программирования (Python, C, Java или другой язык)
2. Компьютеры
3. Схемы и диаграммы для визуализации структуры связного списка
4. Примеры кода для практики

План занятия:

1. Введение

- Определение связного списка и его особенности.
- Структура связного списка: узлы (nodes) и ссылки (pointers).
- Преимущества и недостатки связных списков по сравнению с массивами.

2. Реализация связного списка (30 минут)

- Пример реализации связного списка на разных языках программирования.

Пример на Python:

```
class Node:
    def __init__(self, data):
        self.data = data
        self.next = None
class LinkedList:
    def __init__(self):
        self.head = None
    def append(self, data):
        new_node = Node(data)
        if not self.head:
            self.head = new_node
            return
        last = self.head
        while last.next:
            last = last.next
        last.next = new_node
    def display(self):
        current = self.head
        while current:
            print(current.data, end=" -> ")
```

```
            current = current.next
        print("None")

# Пример использования связного списка
linked_list = LinkedList()
linked_list.append(1)
linked_list.append(2)
linked_list.append(3)
linked_list.display()
```

Пример на C:

```
#include <stdio.h>
#include <stdlib.h>
typedef struct Node {
    int data;
    struct Node* next;
} Node;
typedef struct LinkedList {
    Node* head;
} LinkedList;
void init(LinkedList* list) {
    list->head = NULL;
}
void append(LinkedList* list, int data) {
    Node* new_node = (Node*)malloc(sizeof(Node));
    new_node->data = data;
    new_node->next = NULL;
    if (list->head == NULL) {
        list->head = new_node;
        return;
    }
    Node* last = list->head;
    while (last->next) {
        last = last->next;
    }
    last->next = new_node;
}
void display(LinkedList* list) {
    Node* current = list->head;
    while (current) {
        printf("%d -> ", current->data);
        current = current->next;
```

```
    }
    printf("NULL\n");
}

// Пример использования связного списка
int main() {
    LinkedList list;
    init(&list);
    append(&list, 1);
    append(&list, 2);
    append(&list, 3);
    display(&list);
    return 0;
}
```

3. Основные методы связного списка

- **Добавление** элемента в начало, конец и в произвольное место.
- **Удаление** элемента по значению или индексу.
- **Поиск** элемента в списке.

Пример на Python для удаления элемента:

```
def delete_node(self, key):
    current = self.head
    if current and current.data == key:
        self.head = current.next
        current = None
        return
    while current and current.next:
        if current.next.data == key:
            current.next = current.next.next
            return
        current = current.next
```

4. Применение связного списка

- Обсуждение случаев использования связных списков:
  - Динамическое выделение памяти.
  - Реализация других структур данных (например, стеков и очередей).

5. Практика

- Студенты реализуют методы добавления, удаления и поиска элементов в связном списке.

Задача: Реализуйте метод поиска элемента и метод удаления элемента по значению.

Пример на Python для поиска элемента:

```
def search(self, key):
    current = self.head
    while current:
        if current.data == key:
            return True
        current = current.next
    return False
```

6. Вопросы и ответы

- Ответьте на вопросы студентов о связных списках и их методах.

7. Заключение

- Подведите итоги занятия.
- Обсудите, как знание связных списков и их методов может помочь в будущем программировании.

ПРАКТИЧЕСКОЕ ЗАНЯТИЕ 9. ПРИНЦИП РЕКУРСИИ, РЕКУРСИЯ И ИТЕРАЦИЯ, ПРИМЕРЫ РЕКУРСИИ: ХАНОЙСКАЯ БАШНЯ И ПОСЛЕДОВАТЕЛЬНОСТЬ ФИБОНАЧЧИ, ПРИМЕНЕНИЯ РЕКУРСИИ

Цели:

Студенты получат теоретические знания о рекурсии, сравнят рекурсию с итерацией и изучат примеры применения рекурсии, включая задачу Ханойских башен и последовательность Фибоначчи.

Материалы:

1. Средства программирования (Python, C, Java или другой язык)
2. Компьютеры
3. Схемы для визуализации работы рекурсивных алгоритмов
4. Примеры кода для практики

План занятия:

1. Введение в рекурсию

- Определение рекурсии.
- Принцип работы рекурсии: базовый случай и рекурсивный случай.
- Сравнение рекурсии и итерации: когда использовать что?

2. Пример 1: Последовательность Фибоначчи

- Определение последовательности Фибоначчи.
- Рекурсивная реализация вычисления F(n).

Пример на Python:

```
def fibonacci(n):
    if n <= 0:
        return 0
    elif n == 1:
        return 1
    else:
        return fibonacci(n - 1) + fibonacci(n - 2)
# Пример использования
n = 10
print("Фибоначчи до", n, ":", [fibonacci(i) for i
in range(n)])
```

Пример 2: Задача Ханойских башен

- Объяснение задачи Ханойских башен.
- Алгоритм решения с использованием рекурсии.

Пример на Python:

```
def hanoi(n, source, target, auxiliary):
    if n == 1:
```

```
        print(f"Переместить диск 1 из {source} в
{target}")
        return
    hanoi(n - 1, source, auxiliary, target)
    print(f"Переместить диск {n} из {source} в
{target}")
    hanoi(n - 1, auxiliary, target, source)
# Пример использования
n = 3
hanoi(n, 'A', 'C', 'B')  # A - начальная, C -
целевая, B – вспомогательная
```

4. Приложения рекурсии

- Обсуждение реальных приложений рекурсии в программировании:
  - Обработка деревьев и графов
  - Решение задач о коммивояжере
  - Разделение и завоевание (divide and conquer) алгоритмы

5. Практика

- Студенты пишут программу для решения задачи Ханойских башен и вычисления последовательности Фибоначчи.
- Попросите студентов сравнить рекурсивный и итеративный подход к решению одной из задач.

6. Вопросы и ответы

- Ответьте на вопросы студентов о рекурсии и ее примерах.

7. Заключение

- Подведите итоги занятия.
- Обсудите, как знание рекурсии может помочь в будущем программировании.

ПРАКТИЧЕСКОЕ ЗАНЯТИЕ 10.
ВЫПОЛНЕНИЕ ОПЕРАЦИЙ НАД СТРУКТУРОЙ ДЕРЕВА

Цели:

Студенты получат теоретические знания о деревьях как структуре данных и научатся выполнять различные операции над деревьями.

Материалы:

1. Средства программирования (Python, C, Java или другой язык)
2. Компьютеры
3. Схемы и диаграммы для визуализации работы деревьев
4. Примеры кода для практики

План занятия:

1. Введение

- Определение дерева и его компоненты (узлы, корень, листья).
- Разнообразие деревьев (бинарные деревья, бинарные деревья поиска, AVL-деревья и т.д.).
- Основные операции над деревьями: вставка, удаление, обход.

2. Бинарное дерево

- Определение бинарного дерева и его особенности.
- Пример структуры бинарного дерева.

Пример на Python:

```
class Node:
    def __init__(self, key):
        self.left = None
        self.right = None
        self.value = key
class BinaryTree:
    def __init__(self):
        self.root = None
    def insert(self, key):
        if self.root is None:
            self.root = Node(key)
        else:
            self._insert_rec(self.root, key)
    def _insert_rec(self, node, key):
        if key < node.value:
            if node.left is None:
                node.left = Node(key)
            else:
                self._insert_rec(node.left, key)
        else:
            if node.right is None:
```

```
                node.right = Node(key)
            else:
                self._insert_rec(node.right, key)
    def inorder_traversal(self, node):
        if node:
            self.inorder_traversal(node.left)
            print(node.value, end=' ')
            self.inorder_traversal(node.right)
# Пример использования
tree = BinaryTree()
tree.insert(5)
tree.insert(3)
tree.insert(7)
print("Обход дерева (симметричный):", end=' ')
tree.inorder_traversal(tree.root)
```

3. Операции над Деревьями (30 минут)

- **Вставка узла**: демонстрация, как добавлять элементы в бинарное дерево.
- **Удаление узла**: обсуждение и пример удаления узла из бинарного дерева.
- **Обход дерева**: объяснение методов обхода (симметричный, прямой, обратный).

Пример на C:

```
#include <stdio.h>
#include <stdlib.h>
typedef struct Node {
    int key;
    struct Node* left;
    struct Node* right;
} Node;
Node* newNode(int item) {
    Node* temp = (Node*)malloc(sizeof(Node));
    temp->key = item;
    temp->left = temp->right = NULL;
    return temp;
}
Node* insert(Node* node, int key) {
    if (node == NULL) return newNode(key);
    if (key < node->key)
        node->left = insert(node->left, key);
    else
        node->right = insert(node->right, key);
```

```
    return node;
}
void inorder(Node* root) {
    if (root != NULL) {
        inorder(root->left);
        printf("%d ", root->key);
        inorder(root->right);
    }
}
// Пример использования
int main() {
    Node* root = NULL;
    root = insert(root, 5);
    insert(root, 3);
    insert(root, 7);
    printf("Обход дерева (симметричный): ");
    inorder(root);
    return 0;
}
```

4. Практика

- Студенты выполняют вставку, удаление и обход узлов в дереве.
- Задача: реализовать алгоритм для симметричного и прямого обхода бинарного дерева.

5. Вопросы и ответы

- Ответьте на вопросы студентов о деревьях и операциях над ними.

6. Заключение

- Подведите итоги занятия.
- Обсудите, как знание работы с деревьями может помочь в будущем программировании.

ПРАКТИЧЕСКОЕ ЗАНЯТИЕ 11. АЛГОРИТМЫ СОРТИРОВКИ: СОРТИРОВКА ВЫБОРОМ, ПУЗЫРЬКОВАЯ СОРТИРОВКА, СОРТИРОВКА ВСТАВКАМИ

Цели:

Студенты получат теоретические знания о различных алгоритмах сортировки и научатся реализовывать их на практике.

Материалы:

1. Средства программирования (Python, C, Java или другой язык)
2. Компьютеры
3. Схемы и диаграммы для визуализации работы алгоритмов
4. Примеры кода и задачи для практики

План занятия:

1. Введение

- Определение сортировки и её значимость в программировании.
- Обзор алгоритмов сортировки: сортировка выбором, пузырьком и вставками.

2. Сортировка выбором

- Объяснение алгоритма сортировки выбором.
- Пример работы алгоритма на массиве.

Пример на Python:

```
def selection_sort(arr):
    for i in range(len(arr)):
        min_index = i
        for j in range(i + 1, len(arr)):
            if arr[j] < arr[min_index]:
                min_index = j
        arr[i], arr[min_index] = arr[min_index],
arr[i]
# Пример использования
numbers = [64, 25, 12, 22, 11]
selection_sort(numbers)
print("Отсортированный массив:", numbers)
```

3. Сортировка пузырьком

- Объяснение алгоритма сортировки пузырьком.
- Пример работы алгоритма на массиве.

Пример на C:

```
#include <stdio.h>
void bubble_sort(int arr[], int n) {
    for (int i = 0; i < n-1; i++) {
        for (int j = 0; j < n-i-1; j++) {
```

```
            if (arr[j] > arr[j+1]) {
                // Обмен значениями
                int temp = arr[j];
                arr[j] = arr[j+1];
                arr[j+1] = temp;
            }
        }
    }
}

int main() {
    int arr[] = {64, 34, 25, 12, 22, 11, 90};
    int n = sizeof(arr)/sizeof(arr[0]);
    bubble_sort(arr, n);
    printf("Отсортированный массив: ");
    for (int i = 0; i < n; i++)
        printf("%d ", arr[i]);
    return 0;
}
```

4. Сортировка вставками

- Объяснение алгоритма сортировки вставками.
- Пример работы алгоритма на массиве.

Пример на Java:

```
public class InsertionSort {
    public static void insertionSort(int[] arr) {
        for (int i = 1; i < arr.length; i++) {
            int key = arr[i];
            int j = i - 1;
            while (j >= 0 && arr[j] > key) {
                arr[j + 1] = arr[j];
                j--;
            }
            arr[j + 1] = key;
        }
    }

    public static void main(String[] args) {
        int[] arr = {12, 11, 13, 5, 6};
        insertionSort(arr);
        System.out.print("Отсортированный массив:
");
        for (int i : arr) {
```

```
                System.out.print(i + " ");
            }
        }
}
```

5. Сравнение алгоритмов

- Обсудите преимущества и недостатки каждого алгоритма.
- Сравнение временной сложности:
  - Сортировка выбором: O(n^2)
  - Сортировка пузырьком: O(n^2)
  - Сортировка вставками: O(n^2) в худшем случае, O(n) в лучшем.

6. Практика

- Студенты реализуют один из алгоритмов сортировки на выбранном языке программирования и тестируют его на различных входных данных.

7. Вопросы и ответы

- Ответьте на вопросы студентов о алгоритмах сортировки.

8. Заключение

- Подведите итоги занятия.
- Обсудите, как знание алгоритмов сортировки может помочь в будущем программировании.

ПРАКТИЧЕСКОЕ ЗАНЯТИЕ 12. АЛГОРИТМЫ СОРТИРОВКИ: БЫСТРАЯ СОРТИРОВКА, СОРТИРОВКА СЛИЯНИЕМ, ПИРАМИДАЛЬНАЯ СОРТИРОВКА

Цели:

Студенты получат теоретические знания о алгоритмах сортировки и научатся применять их на практике.

Материалы:

1. Средства программирования (Python, C, Java или другой язык)
2. Компьютеры
3. Примеры кода для практики
4. Схемы, иллюстрирующие работу алгоритмов сортировки

План занятия:

1. Введение

- Обсуждение важности алгоритмов сортировки в программировании.
- Краткий обзор трех алгоритмов: быстрой сортировки, сортировки слиянием и сортировки кучей.

2. Быстрая сортировка

- Объяснение алгоритма быстрой сортировки (разделяй и властвуй).
- Пример реализации быстрой сортировки.

Пример на Python:

```
def quick_sort(arr):
    if len(arr) <= 1:
        return arr
    pivot = arr[len(arr) // 2]
    left = [x for x in arr if x < pivot]
    middle = [x for x in arr if x == pivot]
    right = [x for x in arr if x > pivot]
    return quick_sort(left) + middle +
quick_sort(right)
# Пример использования
numbers = [10, 7, 8, 9, 1, 5]
print("Отсортированный массив:",
quick_sort(numbers))
```

3. Сортировка слиянием

- Объяснение алгоритма сортировки слиянием (разделяй и властвуй).
- Пример реализации сортировки слиянием.

Пример на Python:

```
def merge_sort(arr):
    if len(arr) <= 1:
```

```
        return arr
    mid = len(arr) // 2
    left = merge_sort(arr[:mid])
    right = merge_sort(arr[mid:])
    return merge(left, right)
def merge(left, right):
    result = []
    i = j = 0
    while i < len(left) and j < len(right):
        if left[i] < right[j]:
            result.append(left[i])
            i += 1
        else:
            result.append(right[j])
            j += 1
    result.extend(left[i:])
    result.extend(right[j:])
    return result

# Пример использования
numbers = [10, 7, 8, 9, 1, 5]
print("Отсортированный массив:",
merge_sort(numbers))
```

4. Сортировка кучей

- Объяснение алгоритма сортировки кучей (на основе структуры данных куча).
- Пример реализации сортировки кучей.

Пример на Python:

```
def heapify(arr, n, i):
    largest = i
    left = 2 * i + 1
    right = 2 * i + 2
    if left < n and arr[i] < arr[left]:
        largest = left
    if right < n and arr[largest] < arr[right]:
        largest = right
    if largest != i:
        arr[i], arr[largest] = arr[largest], arr[i]
        heapify(arr, n, largest)
def heap_sort(arr):
    n = len(arr)
```

```
    for i in range(n // 2 - 1, -1, -1):
        heapify(arr, n, i)
    for i in range(n - 1, 0, -1):
        arr[i], arr[0] = arr[0], arr[i]
        heapify(arr, i, 0)
# Пример использования
numbers = [10, 7, 8, 9, 1, 5]
heap_sort(numbers)
print("Отсортированный массив:", numbers)
```

5. Сравнение алгоритмов

- Обсудите преимущества и недостатки каждого алгоритма:
  - Быстрая сортировка: эффективность, худший случай.
  - Сортировка слиянием: стабильность, потребление памяти.
  - Сортировка кучей: нет необходимости в дополнительной памяти, но медленнее.

6. Практика

- Студенты реализуют один из алгоритмов сортировки по своему выбору на заданном языке программирования и тестируют его на различных данных.

7. Вопросы и ответы

- Ответьте на вопросы студентов о различных алгоритмах сортировки.

8. Заключение

- Подведите итоги занятия.
- Обсудите, как знание алгоритмов сортировки может помочь в будущем программировании.

ПРАКТИЧЕСКОЕ ЗАНЯТИЕ 13.
АЛГОРИТМЫ ПОИСКА: БИНАРНЫЙ ПОИСК, ЛИНЕЙНЫЙ ПОИСК

Цели:

Студенты получат теоретические знания о двоичном и линейном поиске и научатся применять эти алгоритмы на практике.

Материалы:

1. Средства программирования (Python, C, Java или другой язык)
2. Компьютеры
3. Диаграммы для визуализации алгоритмов поиска
4. Примеры кода и задачи для практики

План занятия:

1. Введение

- Объяснение важности поиска в программировании.
- Определение линейного и двоичного поиска.
- Когда использовать каждый из методов.

2. Линейный п оиск

- Определение линейного поиска и его принцип работы.
- Пример работы линейного поиска.

Пример на Python:

```
def linear_search(arr, target):
    for index, value in enumerate(arr):
        if value == target:
            return index
    return -1
# Пример использования
numbers = [10, 20, 30, 40, 50]
target = 30
result = linear_search(numbers, target)
if result != -1:
    print(f"Элемент найден на индексе: {result}")
else:
    print("Элемент не найден.")
```

Пример на C:

```
#include <stdio.h>
int linear_search(int arr[], int size, int target)
{
    for (int i = 0; i < size; i++) {
        if (arr[i] == target) {
            return i; // Возвращает индекс
найденного элемента
```

```
        }
    }
    return -1; // Возвращает -1, если элемент не
найден
}
// Пример использования
int main() {
    int numbers[] = {10, 20, 30, 40, 50};
    int target = 30;
    int size = sizeof(numbers) /
sizeof(numbers[0]);
    int result = linear_search(numbers, size,
target);
    if (result != -1) {
        printf("Элемент найден на индексе: %d\n",
result);
    } else {
        printf("Элемент не найден.\n");
    }
    return 0;
}
```

3. Двоичный поиск

- Определение двоичного поиска и условия его применения (массив должен быть отсортирован).
- Пример работы двоичного поиска.

Пример на Python:

```
def binary_search(arr, target):
    left, right = 0, len(arr) - 1
    while left <= right:
        mid = (left + right) // 2
        if arr[mid] == target:
            return mid
        elif arr[mid] < target:
            left = mid + 1
        else:
            right = mid - 1
    return -1
# Пример использования
sorted_numbers = [10, 20, 30, 40, 50]
target = 30
result = binary_search(sorted_numbers, target)
if result != -1:
```

```
    print(f"Элемент найден на индексе: {result}")
else:
    print("Элемент не найден.")
```

Пример на C:

```
#include <stdio.h>
int binary_search(int arr[], int size, int target)
{
    int left = 0;
    int right = size - 1;

    while (left <= right) {
        int mid = left + (right - left) / 2;
        if (arr[mid] == target) {
            return mid; // Возвращает индекс
найденного элемента
        }
        if (arr[mid] < target) {
            left = mid + 1;
        } else {
            right = mid - 1;
        }
    }
    return -1; // Возвращает -1, если элемент не
найден
}

// Пример использования
int main() {
    int sorted_numbers[] = {10, 20, 30, 40, 50};
    int target = 30;
    int size = sizeof(sorted_numbers) /
sizeof(sorted_numbers[0]);
    int result = binary_search(sorted_numbers,
size, target);
    if (result != -1) {
        printf("Элемент найден на индексе: %d\n",
result);
    } else {
        printf("Элемент не найден.\n");
    }
    return 0;
```

```
}
```

4. Сравнение алгоритмов

- Обсудите преимущества и недостатки линейного и двоичного поиска.
- Время выполнения: линейный поиск O(n), двоичный поиск O(log n).

5. Практика

- Студенты пишут программы, которые сравнивают время выполнения линейного и двоичного поиска для одного и того же набора данных.

6. Вопросы и ответы

- Ответьте на вопросы студентов о линейном и двоичном поиске.

7. Заключение

- Подведите итоги занятия.
- Обсудите, как знание алгоритмов поиска может помочь в будущем программировании.

ПРАКТИЧЕСКОЕ ЗАНЯТИЕ 14.
СТРУКТУРА ДАННЫХ ГРАФ. АЛГОРИТМЫ ГРАФА. АЛГОРИТМ BFS, АЛГОРИТМ ДЕЙКСТРЫ, АЛГОРИТМ УОРШАЛЛА, АЛГОРИТМЫ КРАСКАЛА И РАНДОМНЫЙ ВЫБОР

Цели:

Студенты получат теоретические знания о графовых структурах данных и различных алгоритмах, а также научатся применять их на практике.

Материалы:

1. Средства программирования (Python, C++, Java или другой язык)
2. Компьютеры
3. Схемы и диаграммы для визуализации графов и алгоритмов
4. Примеры кода и задачи для практики

План занятия:

1. Введение в графы

- Определение графа и его компоненты: вершины и ребра.
- Различие между направленными и ненаправленными графами.
- Примеры применения графов в реальных задачах (поиск, маршрутизация и др.).

2. Алгоритм поиска в ширину (BFS) (1)

- Объяснение алгоритма BFS.
- Пример его применения.

Пример на Python:

```
from collections import deque
def bfs(graph, start):
    visited = set()
    queue = deque([start])
        while queue:
        vertex = queue.popleft()
        if vertex not in visited:
            print(vertex, end=" ")
            visited.add(vertex)
            queue.extend(neighbor for neighbor in
graph[vertex] if neighbor not in visited)

# Пример графа
graph = {
    'A': ['B', 'C'],
    'B': ['A', 'D', 'E'],
    'C': ['A', 'F'],
    'D': ['B'],
    'E': ['B', 'F'],
```

```
    'F': ['C', 'E']
}

print("BFS Traversal: ")
bfs(graph, 'A')
```

3. Алгоритм Дейкстры

- Объяснение алгоритма Дейкстры для поиска кратчайшего пути.
- Пример его реализации.

Пример на Python:

```
import heapq
def dijkstra(graph, start):
    queue = []
    distances = {vertex: float('infinity') for
vertex in graph}
    distances[start] = 0
    heapq.heappush(queue, (0, start))
    while queue:
        current_distance, current_vertex =
heapq.heappop(queue)

        if current_distance >
distances[current_vertex]:
            continue
        for neighbor, weight in
graph[current_vertex].items():
            distance = current_distance + weight
            if distance < distances[neighbor]:
                distances[neighbor] = distance
                heapq.heappush(queue, (distance,
neighbor))

    return distances

# Пример графа с весами
graph = {
    'A': {'B': 1, 'C': 4},
    'B': {'A': 1, 'D': 2, 'E': 2},
    'C': {'A': 4, 'F': 3},
    'D': {'B': 2, 'F': 1},
    'E': {'B': 2, 'F': 5},
    'F': {'C': 3, 'D': 1, 'E': 5}
}
```

```
print("Distances from A: ", dijkstra(graph, 'A'))
import heapq
def dijkstra(graph, start):
    queue = []
    distances = {vertex: float('infinity') for
vertex in graph}
    distances[start] = 0
    heapq.heappush(queue, (0, start))
    while queue:
        current_distance, current_vertex =
heapq.heappop(queue)
        if current_distance >
distances[current_vertex]:
            continue
        for neighbor, weight in
graph[current_vertex].items():
            distance = current_distance + weight
            if distance < distances[neighbor]:
                distances[neighbor] = distance
                heapq.heappush(queue, (distance,
neighbor))

    return distances
# Пример графа с весами
graph = {
    'A': {'B': 1, 'C': 4},
    'B': {'A': 1, 'D': 2, 'E': 2},
    'C': {'A': 4, 'F': 3},
    'D': {'B': 2, 'F': 1},
    'E': {'B': 2, 'F': 5},
    'F': {'C': 3, 'D': 1, 'E': 5}
}

print("Distances from A: ", dijkstra(graph, 'A'))
```

4. Алгоритм Уоршала

- Объяснение алгоритма Уоршала для нахождения всех кратчайших путей.

Пример на Python:

```
def warshall(graph):
    n = len(graph)
    dist = [[float('inf')] * n for _ in range(n)]
    for u in range(n):
```

```
        for v in range(n):
            if u == v or graph[u][v] != 0:
                dist[u][v] = graph[u][v]
    for k in range(n):
        for i in range(n):
            for j in range(n):
                if dist[i][k] + dist[k][j] <
dist[i][j]:
                    dist[i][j] = dist[i][k] +
dist[k][j]
    return dist

# Пример графа
graph = [
    [0, 3, 0, 0, 0],
    [0, 0, 1, 5, 0],
    [0, 0, 0, 2, 0],
    [0, 0, 0, 0, 4],
    [0, 0, 0, 0, 0]
]
print("All pairs shortest paths: ",
warshall(graph))
```

5. Алгоритм Краскала

- Объяснение алгоритма Краскала для нахождения минимального остовного дерева.

Пример на Python:

```
class UnionFind:
    def __init__(self, size):
        self.root = list(range(size))
        self.rank = [1] * size
    def find(self, x):
        if self.root[x] != x:
            self.root[x] = self.find(self.root[x])
        return self.root[x]
    def union(self, x, y):
        rootX = self.find(x)
        rootY = self.find(y)
        if rootX != rootY:
            if self.rank[rootX] > self.rank[rootY]:
                self.root[rootY] = rootX
            elif self.rank[rootX] <
self.rank[rootY]:
```

```
                self.root[rootX] = rootY
            else:
                self.root[rootY] = rootX
                self.rank[rootX] += 1
def kruskal(vertices, edges):
    edges.sort(key=lambda x: x[2])  # Сортируем по
весу
    uf = UnionFind(len(vertices))
    mst = []
    for u, v, weight in edges:
        if uf.find(u) != uf.find(v):
            uf.union(u, v)
            mst.append((u, v, weight))

    return mst
# Пример графа
vertices = [0, 1, 2, 3]
edges = [(0, 1, 1), (0, 2, 3), (1, 2, 2), (1, 3,
6), (2, 3, 5)]
print("Minimum Spanning Tree: ", kruskal(vertices,
edges))
```

6. Алгоритм Round-Robin

- Объяснение алгоритма Round-Robin для управления процессами в операционных системах.

Пример на Python:

```
def round_robin(processes, burst_time, quantum):
    n = len(processes)
    remaining_time = list(burst_time)
    time = 0
    while True:
        done = True
        for i in range(n):
            if remaining_time[i] > 0:
                done = False
                if remaining_time[i] > quantum:
                    time += quantum
                    remaining_time[i] -= quantum
                else:
                    time += remaining_time[i]
                    remaining_time[i] = 0
```

```
                    print(f"Процесс {processes[i]}
завершен в {time}")
        if done:
            break
processes = [1, 2, 3]
burst_time = [10, 5, 8]
quantum = 4
round_robin(processes, burst_time, quantum)
```

7. Практика (30 минут)

- Студенты пишут программы для реализации каждого алгоритма и решают небольшие задачи на графах.

8. Вопросы и ответы

- Ответьте на вопросы студентов о графах и алгоритмах.

9. Заключение

- Подведите итоги занятия.
- Обсудите, как знание графов и алгоритмов может помочь в будущем программировании.

ПРАКТИЧЕСКОЕ ЗАНЯТИЕ 15. ХЕШИРОВАНИЕ: ХЕШ-ФУНКЦИЯ И ХЕШ-ТАБЛИЦЫ, ТЕХНИКИ РАЗРЕШЕНИЯ КОЛЛИЗИЙ, СРАВНЕНИЕ ЭФФЕКТИВНОСТИ РАЗЛИЧНЫХ ТЕХНИК ПОИСКА

Цели:

Студенты получат теоретические знания о хешировании и научатся применять хеш-функции и хеш-таблицы, а также смогут сравнить эффективность различных техник поиска.

Материалы:

1. Средства программирования (Python, C, Java или другой язык)
2. Компьютеры
3. Схемы для визуализации хеширования и разрешения коллизий
4. Примеры кода и задачи для практики

План занятия:

1. Введение

- Определение хеширования и его применение.
- Что такое хеш-функция и хеш-таблица.
- Зачем нужно разрешение коллизий.

2. Хеш-функции и хеш-таблицы

- Принцип работы хеш-функции (преобразование ключа в индекс).
- Пример хеш-функции и как она используется в хеш-таблице.

Пример на Python:

```
class HashTable:
    def __init__(self):
        self.size = 10
        self.table = [None] * self.size

    def hash_function(self, key):
        return hash(key) % self.size
    def insert(self, key, value):
        index = self.hash_function(key)
        self.table[index] = value
    def get(self, key):
        index = self.hash_function(key)
        return self.table[index]

# Пример использования
ht = HashTable()
ht.insert("key1", "value1")
print("Значение для key1:", ht.get("key1"))
```

3. Техники разрешения коллизий

- Обсуждение различных техник разрешения коллизий:
  - Метод цепочек (chaining)
  - Открытая адресация (open addressing)

Пример метода цепочек на Python:

```
class HashTableWithChaining:
    def __init__(self):
        self.size = 10
        self.table = [[] for _ in range(self.size)]

    def hash_function(self, key):
        return hash(key) % self.size

    def insert(self, key, value):
        index = self.hash_function(key)
        self.table[index].append((key, value))
    def get(self, key):
        index = self.hash_function(key)
        for k, v in self.table[index]:
            if k == key:
                return v
        return None
# Пример использования
ht_chain = HashTableWithChaining()
ht_chain.insert("key1", "value1")
ht_chain.insert("key1", "value2")  # пример
коллизии
print("Значение для key1:", ht_chain.get("key1"))
```

4. Сравнение эффективности различных техник поиска

- Обсуждение различных техник поиска: линейный поиск, бинарный поиск, хеширование.
- Сравнение временной сложности этих техник.

Пример линейного поиска на Python:

```
def linear_search(arr, target):
    for index, value in enumerate(arr):
        if value == target:
            return index
    return -1
# Пример использования
arr = [1, 2, 3, 4, 5]
print("Индекс 3:", linear_search(arr, 3))
```

Пример бинарного поиска на Python:

```
def binary_search(arr, target):
    left, right = 0, len(arr) - 1
    while left <= right:
        mid = (left + right) // 2
        if arr[mid] == target:
            return mid
        elif arr[mid] < target:
            left = mid + 1
        else:
            right = mid - 1
    return -1
# Пример использования
sorted_arr = [1, 2, 3, 4, 5]
print("Индекс 3:", binary_search(sorted_arr, 3))
```

5. Практика

- Студенты создают простую хеш-таблицу и реализуют технику разрешения коллизий методом цепочек.
- Они также сравнивают эффективность хеширования с другими техниками поиска.

6. Вопросы и ответы

- Ответьте на вопросы студентов о хешировании и его применении.

7. Заключение

- Подведите итоги занятия.
- Обсудите, как знание хеширования и техники разрешения коллизий могут помочь в будущем программировании.

ГЛОССАРИЙ

Алгоритм — последовательность шагов или инструкций для решения задачи за конечное количество шагов.

Асимптотическая сложность — функция, описывающая зависимость времени выполнения алгоритма от размера входных данных (например, O(n)O(n)O(n), O(n2)O(n^2)O(n2)).

Анализ времени выполнения (Time Complexity) — оценка количества операций, которые выполняет алгоритм в зависимости от размера входных данных.

Анализ использования памяти (Space Complexity) — оценка количества памяти, которую алгоритм использует в зависимости от размера входных данных.

Big-O нотация — математическая запись, описывающая верхнюю границу (наихудший случай) сложности алгоритма. Например, O(n)O(n)O(n) означает, что время выполнения алгоритма растет линейно с увеличением размера данных.

Амортизированная сложность — средняя сложность операции в ходе последовательности операций, включающая оценку худших случаев, но при этом учитывающая повторяющиеся эффективные операции.

Жадный алгоритм (Greedy Algorithm) — алгоритм, который принимает локально оптимальные решения на каждом шаге, стремясь найти глобально оптимальное решение.

Рекурсия — метод решения задачи, при котором функция вызывает сама себя. Основой рекурсии является наличие условий завершения (базового случая), чтобы предотвратить бесконечную рекурсию.

Итерация — повторение набора инструкций до достижения определенного условия.

Динамическое программирование — метод решения задач, при котором решение задачи разбивается на подзадачи, результаты которых запоминаются и используются повторно, что сокращает количество операций.

Структуры данных — способы организации и хранения данных для эффективной работы с ними.

Массив — структура данных, представляющая собой последовательность элементов, доступ к которым осуществляется по индексу.

Связный список — структура данных, в которой элементы (узлы) хранят ссылки на следующие элементы. Разделяют **односвязные** и **двусвязные** списки.

Стек (Stack) — структура данных типа LIFO (последний пришел — первый вышел), где доступ возможен только к последнему добавленному элементу. Основные операции: push, pop, peek.

Очередь (Queue) — структура данных типа FIFO (первый пришел — первый вышел), где элементы добавляются в конец и удаляются из начала. Операции: enqueue, dequeue.

Дек (Deque) — двусторонняя очередь, поддерживающая добавление и удаление элементов с обоих концов.

Хэш-таблица (Hash Table) — структура данных, которая использует хэш-функцию для отображения ключей в индексы массива, что позволяет эффективно искать, добавлять и удалять элементы.

Дерево — иерархическая структура данных, в которой каждый узел имеет дочерние узлы, а самый верхний узел называется корнем. Примеры: бинарное дерево, дерево поиска, AVL-дерево.

Бинарное дерево поиска (Binary Search Tree) — дерево, в котором каждый узел имеет не более двух дочерних узлов, и значения левого поддерева меньше значения узла, а правого — больше.

Дерево с балансировкой (Balanced Tree) — дерево, которое поддерживает примерно равную высоту всех поддеревьев, что обеспечивает логарифмическое время на основные операции.

Куча (Heap) — структура данных на основе бинарного дерева, в которой родительский узел всегда больше (максимальная куча) или меньше (минимальная куча) своих дочерних узлов.

Граф — структура данных, состоящая из множества вершин (узлов) и ребер, соединяющих эти вершины. Графы могут быть направленными или ненаправленными.

Обход графа — процесс посещения всех вершин графа. Существует два основных метода обхода: **поиск в глубину (DFS)** и **поиск в ширину (BFS)**.

Поиск в глубину (DFS) — алгоритм обхода или поиска в графе, который углубляется в граф, переходя по ребрам до тех пор, пока это возможно, а затем возвращается назад.

Поиск в ширину (BFS) — алгоритм обхода или поиска в графе, который исследует все вершины на одном уровне перед переходом к вершинам следующего уровня.

Алгоритм Дейкстры (Dijkstra's Algorithm) — алгоритм для поиска кратчайшего пути в графе с ненегативными весами ребер.

Алгоритм Флойда-Уоршелла (Floyd-Warshall Algorithm) — Алгоритм поиска кратчайших путей между всеми парами вершин в графе.

Жадный алгоритм (Greedy Algorithm) — алгоритм, который на каждом шаге принимает локально оптимальное решение, не заботясь о глобальной оптимальности.

Разделяй и властвуй (Divide and Conquer) — метод решения задач, при котором задача делится на несколько меньших задач, каждая из которых решается рекурсивно, а затем их решения комбинируются.

Сортировка — процесс упорядочивания элементов массива или списка. Наиболее известные алгоритмы сортировки:

Пузырьковая сортировка (Bubble Sort) — простой, но неэффективный алгоритм сортировки.

Быстрая сортировка (Quick Sort) — алгоритм "разделяй и властвуй", который рекурсивно разбивает массив на части.

Сортировка слиянием (Merge Sort) — еще один алгоритм "разделяй и властвуй", который разбивает массив пополам и затем сливает отсортированные части.

Пирамидальная сортировка (Heap Sort) — сортировка с использованием структуры данных куча.

Сортировка вставками (Insertion Sort) — элементы последовательно вставляются на свои места в отсортированной части массива.

Жадный алгоритм (Greedy Algorithm) — стратегия решения задач, где на каждом шаге выбирается наилучший локальный вариант, надеясь, что итоговое решение будет глобально оптимальным.

Алгоритм Кнута-Морриса-Пратта (KMP) — Алгоритм поиска подстроки, использующий предварительную обработку строки для ускорения поиска.

Алгоритм Хаффмана (Huffman Coding) — алгоритм сжатия данных, использующий частоты символов для построения префиксного кодирования с минимальной средней длиной кода.

NP-полные задачи (NP-complete problems) — класс задач, которые не имеют известных полиномиальных решений, но если бы одно из таких решений было найдено, все остальные задачи класса можно было бы решить за полиномиальное время.

Префиксное дерево (Trie) — дерево, используемое для хранения набора строк, где каждая строка представляется путем от корня до листа.

Битовое поле (Bitset) — структура данных, представляющая собой массив битов (0 или 1), часто используемая для экономии памяти

ЛИТЕРАТУРА

1. Michael T. Goodrich, Roberto Tamassia, Michael H. Goldwasser. Data Structures and Algorithms in Python. John Wiley & Sons. 2013. –770p.
2. Никлаус Вирт. Алгоритмы и структуры данных. Новая версия для Оберона + CD / Пер. с англ. Ткачев Ф. В. – М.: ДМК Пресс, 2010. – 272с.
3. A.A. Avezov, Ma’lumotlar tuzilmasi va algoritmlar / [Matn]: o‘quv qo‘llanma / A.A. Avezov - Buxoro: “BUXORO DETERMINANTI” MCHJning Kamolot nashriyoti, 2023. - 120b.
1. Альфред, В., Хопкрофт, Джон, Ульман, Джеффри, Д. Структуры данных и алгоритмы/ Пер. с англ.: Уч. Пос. – М. Издателъский дом «Вилъямс», 2000. – 384 с.: ил.
2. Пушкин Е.В. Структуры данных и алгоритмы: реализация на C/C++. - СПб.: ФТК СПБГПУ, 2009.- 200 с.
3. Овсянников, А. В. Алгоритмы и структуры данных: учебно-методический комплекс для спетсиалъности 1-31 03 07 «Прикладная информатика (по направлениям)». Ч. 1 Минск: БГУ, 2015. – 124 с.
4. Домнин Л. Н. Элементы теории графов: учеб. Пособие / Л. Н. Домнин. – Пенза: Изд-во Пенз. Гос. Ун-та, 2007. – 144 с. 75 ил., 13 табл., библиогр 18 назв.
5. Никлаус Вирт, Алгоритмы и структуры данных. Новая версия для Оберона + СД / Пер. с англ. Ткачев Ф. В. – М.: ДМК Пресс, 2010. – 272 с.: ил.

Информационные ресурсы:

1. www.gov.uz – O‘zbekiston Respublikasi hukumat portali.
2. www.lex.uz - O‘zbekiston Respublikasi Qonun hujjatlari milliy bazasi.
3. www.ziyonet.uz – Axborot ta’lim portali.
4. www.edu.uz – Oliy va o‘rta maxsus ta’lim vazirligi portali.
5. www.buxdu.uz – buxdu rasmiy sayti.
6. http://www.uzbekcoders.uz - bir million o‘zbek dasturchisi loyihasi
7. https://www.geeksforgeeks.org/

Для заметок

Г.И. Атаева, У.Н. Хамроев, Ш.Ж. Сейтов, Д.Н. Хамроева

АЛГОРИТМЫ И СТРУКТУРЫ ДАННЫХ

УЧЕБНИК

Редактор: Э. Эшов
Технический редактор: Д. Абдурахмонова
Корректор: М. Шодиева
Художественный руководитель: М. Сатторов

Издательская лицензия № 022853. 04.03.2022.
Разрешение на печать с оригинального макета: 21.12.2024.
Формат 60x84. Гарнитура Times New Roman 1/16.
Электрографический печать.
Печатная форма 11,5. Тираж 100. экз. Заказ №_.

KAMOLOT

ООО “БУХОРО ДЕТЕРМИНАНТИ” отпечатано в типографии, город Бухара ул. Намозгох 24
+998 91 310 27 22