Решая реальные проблемы кодирования, работодатели и рекрутеры стремятся как к эффективности времени выполнения, так и к ресурсов эффективности.

Знание того, какая структура данных лучше всего подходит для текущего решения, повысит производительность программы и сократит время, необходимое для её создания. По этой причине большинству ведущих компаний требуется чёткое понимание структур данных и их тщательное тестирование на собеседовании по кодированию.

**Что такое структуры данных?**

Структуры данных — это структуры кода для хранения и организации данных, которые упрощают изменение, навигацию и доступ к информации. Структуры данных определяют способ сбора данных, функциональные возможности, которые мы можем реализовать, и отношения между данными.

Структуры данных используются практически во всех областях информатики и программирования, от операционных систем до интерфейсной разработки и машинного обучения.

**Структуры данных помогают:**

* Управляйте большими наборами данных и используйте их.
* Быстрый поиск определённых данных в базе данных.
* Создавайте чёткие иерархические или реляционные связи между точками данных.
* Упростите и ускорьте обработку данных.

Структуры данных являются жизненно важными строительными блоками для эффективного решения реальных проблем. Структуры данных — это проверенные и оптимизированные инструменты, которые дают вам удобную основу для организации ваших программ. В конце концов, вам не нужно переделывать колесо (или конструкцию) каждый раз, когда это нужно.

У каждой структуры данных есть задача или ситуация, для решения которой она наиболее подходит. Python имеет 4 встроенных структуры данных, списки, словари, кортежи и Множества. Эти встроенные структуры данных поставляются с методами по умолчанию и негласной оптимизацией, которая упрощает их использование.

Большинство структур данных в Python являются их модифицированными формами или используют встроенные структуры в качестве основы.

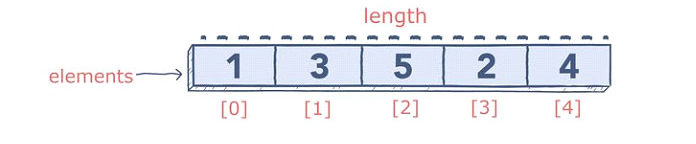
* Список: структуры, похожие на массивы, которые позволяют сохранять Множествоизменяемых объектов одного и того же типа в переменную.
* Кортеж: кортежи — это неизменяемые списки, то есть элементы не могут быть изменены. Он объявлен в круглых скобках вместо квадратных.
* Набор: Множества— это неупорядоченные коллекции, что означает, что элементы неиндексированы и не имеют установленной последовательности. Они объявляются фигурными скобками.
* Словарь (dict): Подобно хэш-карте или хеш-таблицам на других языках, словарь представляет собой Множествопар ключ / значение. Вы инициализируете пустой словарь пустыми фигурными скобками и заполняете его ключами и значениями, разделёнными двоеточиями. Все ключи — уникальные неизменяемые объекты.

Теперь давайте посмотрим, как мы можем использовать эти структуры для создания всех сложных структур, которые ищут интервьюеры.

**Массивы (списки) в Python**

Python не имеет встроенного типа массива, но вы можете использовать списки для всех тех же задач. Массив — это Множествозначений одного типа, сохранённых под тем же именем.

Каждое значение в массиве называется «элементом», и индексирование соответствует его положению. Вы можете получить доступ к определённым элементам, вызвав имя массива с индексом желаемого элемента. Вы также можете получить длину массива с помощью len()метода.



В отличие от языков программирования, таких как Java, которые имеют статические массивы после объявления, массивы Python автоматически увеличиваются или уменьшаются при добавлении / вычитании элементов.

Например, мы могли бы использовать этот append()метод для добавления дополнительного элемента в конец существующего массива вместо объявления нового массива.

Это делает массивы Python особенно простыми в использовании и адаптируемыми на лету.

cars = [«Toyota», «Tesla», «Hyundai»]

print(len(cars))

cars.append(«Honda»)

cars.pop(1)

for x in cars:

  print(x)

**Преимущества:**

* Простота создания и использования последовательностей данных.
* Автоматическое масштабирование в соответствии с меняющимися требованиями к размеру.
* Используется для создания более сложных структур данных.

**Недостатки:**

* Не оптимизирован для научных данных (в отличие от массива NumPy).
* Может управлять только крайним правым концом списка.

**Приложения:**

* Совместное хранилище связанных значений или объектов, т.е. myDogs.
* Коллекции данных, которые вы будете просматривать.
* Коллекции структур данных, например, список кортежей.

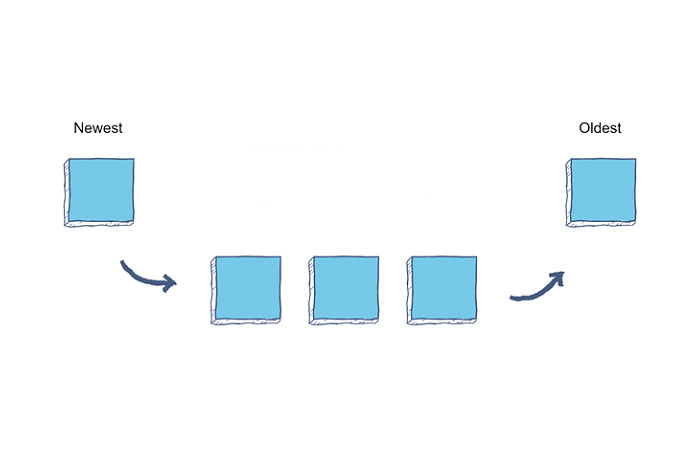
**Общие вопросы собеседования с массивами в Python**

* Удалить чётные числа из списка.
* Объединить два отсортированных списка.
* Найдите минимальное значение в списке.
* Подсписок максимальной суммы.
* Печать продукции всех элементов.

**Очереди в Python**

Очереди — это линейная структура данных, в которой данные хранятся в порядке «первым пришёл — первым ушёл» (FIFO). В отличие от массивов, вы не можете получить доступ к элементам по индексу, а вместо этого можете извлечь только следующий самый старый элемент. Это делает его отличным решением для задач, чувствительных к заказу, таких как обработка онлайн-заказов или хранение голосовой почты.

Вы можете представить очередь как очередь в продуктовом магазине; кассир не выбирает, кого проверять следующим, а скорее обрабатывает человека, который дольше всех стоял в очереди.



Мы могли бы использовать список Python с append()и pop()методами для реализации очереди. Однако это неэффективно, потому что списки должны сдвигать все элементы на один индекс всякий раз, когда вы добавляете новый элемент в начало.   
Вместо этого лучше всего использовать dequeкласс из collectionsмодуля Python. Deques оптимизированы для операций добавления и извлечения. Deque реализация также позволяет создавать двухсторонние очереди, которые могут получить доступ к обеим сторонам очередей через popleft()и popright()методы.

from collections import deque

# Initializing a queue

q = deque()

# Adding elements to a queue

q.append(‘a’)

q.append(‘b’)

q.append(‘c’)

print(«Initial queue»)

print(q)

# Removing elements from a queue

print(«\nElements dequeued from the queue»)

print(q.popleft())

print(q.popleft())

print(q.popleft())

print(«\nQueue after removing elements»)

print(q)

# Uncommenting q.popleft()

# will raise an IndexError

# as queue is now empty

**Преимущества:**

* Автоматически упорядочивает данные в хронологическом порядке.
* Весы в соответствии с требованиями к размеру.
* Эффективное время с dequeклассом.

**Недостатки:**

* Доступ к данным возможен только на концах.

**Приложения:**

* Операции с общим ресурсом, таким как принтер или ядро ЦП.
* Служит временным хранилищем для пакетных систем.
* Обеспечивает простой порядок по умолчанию для задач равной важности.

**Общие вопросы на собеседовании в очереди на Python**

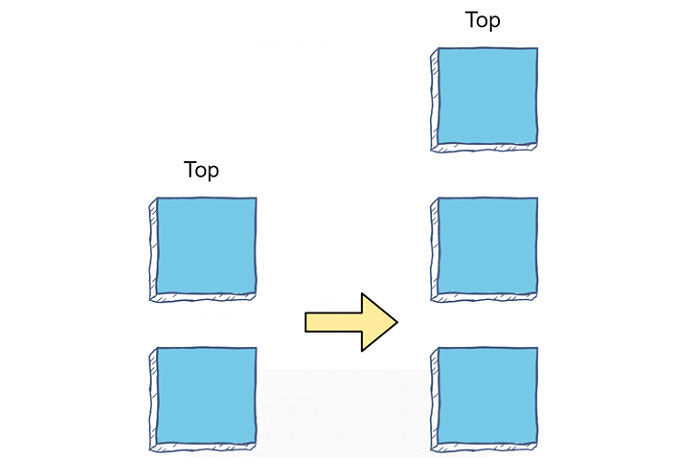
* Обратить первые k элементов очереди.
* Реализуйте очередь, используя связанный список.
* Реализуйте стек с помощью очереди.

**[Читайте также:  Структуры в C](https://bestprogrammer.ru/programmirovanie-i-razrabotka/struktury-v-c" \t "_blank)**

**Стеки в Python**

Стеки представляют собой последовательную структуру данных, которая действует как версия очередей «последний пришёл — первым ушёл» (LIFO). Последний элемент, вставленный в стек, считается верхним в стеке и является единственным доступным элементом. Чтобы получить доступ к среднему элементу, вы должны сначала удалить достаточное количество элементов, чтобы нужный элемент находился на вершине стека.

Многие разработчики представляют стопки как стопку обеденных тарелок; вы можете добавлять или убирать тарелки в верхнюю часть стопки, но вам нужно переместить всю стопку, чтобы разместить одну внизу.



Добавление элементов называется выталкиванием, а удаление элементов — всплывающим сообщением. Вы можете реализовать стеки в Python, используя встроенную структуру списка. При реализации списка в операциях push используется append()метод, а в операциях pop — pop().

stack = []

# append() function to push

# element in the stack

stack.append(‘a’)

stack.append(‘b’)

stack.append(‘c’)

print(‘Initial stack’)

print(stack)

# pop() function to pop

# element from stack in

# LIFO order

print(‘\nElements popped from stack:’)

print(stack.pop())

print(stack.pop())

print(stack.pop())

print(‘\nStack after elements are popped:’)

print(stack)

# uncommenting print(stack.pop())

# will cause an IndexError

# as the stack is now empty

**Преимущества:**

* Предлагает управление данными LIFO, которое невозможно с массивами.
* Автоматическое масштабирование и очистка объекта.
* Простая и надёжная система хранения данных.

**Недостатки:**

* Память стека ограничена.
* Слишком много объектов в стеке приводит к ошибке переполнения стека.

**Приложения:**

* Используется для создания высокореактивных систем.
* Системы управления памятью используют стеки для обработки в первую очередь самых последних запросов.
* Полезно для таких вопросов, как сопоставление скобок.

**Общие вопросы собеседования по стекам в Python**

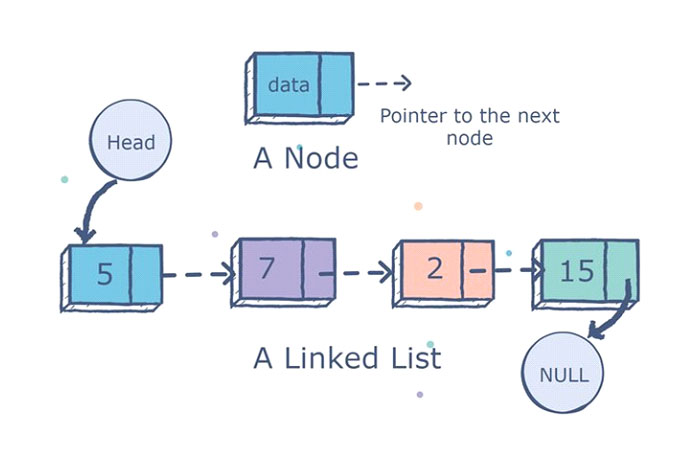
* Реализуйте очередь, используя стеки.
* Вычислить выражение Postfix с помощью стека.
* Следующий по величине элемент, использующий стек.
* Создать min()функцию с использованием стека.

**Связанные списки в Python**

Связанные списки — это последовательный Множестводанных, который использует реляционные указатели на каждом узле данных для связи со следующим узлом в списке.

В отличие от массивов, связанные списки не имеют объективных позиций в списке. Вместо этого у них есть относительные позиции, основанные на их окружающих узлах.

Первый узел в связанном списке называется головным узлом, а последний — хвостовым узлом, который имеет nullуказатель.



Связанные списки могут быть односвязными или дважды связанными в зависимости от того, имеет ли каждый узел только один указатель на следующий узел или он также имеет второй указатель на предыдущий узел.

Вы можете думать о связанных списках как о цепочке; отдельные ссылки имеют связь только со своими ближайшими соседями, но все ссылки вместе образуют более крупную структуру.

Python не имеет встроенной реализации связанных списков и поэтому требует, чтобы вы реализовали Nodeкласс для хранения значения данных и одного или нескольких указателей.

class Node:

    def \_\_init\_\_(self, dataval=None):

        self.dataval = dataval

        self.nextval = None

class SLinkedList:

    def \_\_init\_\_(self):

        self.headval = None

list1 = SLinkedList()

list1.headval = Node(«Mon»)

e2 = Node(«Tue»)

e3 = Node(«Wed»)

# Link first Node to second node

list1.headval.nextval = e2

# Link second Node to third node

e2.nextval = e3

Связанные списки в основном используются для создания расширенных структур данных, таких как графики и деревья, или для задач, требующих частого добавления / удаления элементов по всей структуре.

**Преимущества:**

* Эффективная вставка и удаление новых элементов.
* Проще реорганизовать, чем массивы.
* Полезно в качестве отправной точки для сложных структур данных, таких как графики или деревья.

**Недостатки:**

* Хранение указателей с каждой точкой данных увеличивает использование памяти.
* Всегда должен перемещаться по связанному списку от узла Head, чтобы найти определённый элемент.

**Приложения:**

* Строительный блок для расширенных структур данных.
* Решения, требующие частого добавления и удаления данных.

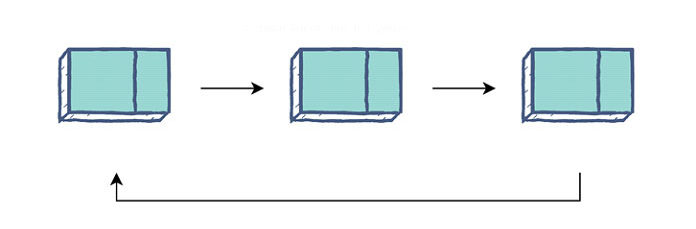
**Общие вопросы собеседования по связному списку в Python**

* Распечатать средний элемент данного связанного списка.
* Удалить повторяющиеся элементы из отсортированного связного списка.
* Проверьте, является ли односвязный список палиндромом.
* Объединить K отсортированных связанных списков.
* Найдите точку пересечения двух связанных списков.

**Циркулярно связанные списки в Python**

Основным недостатком стандартного связного списка является то, что вам всегда нужно начинать с узла Head.

Циклический связанный список устраняет эту проблему, заменяя nullуказатель узла Tail указателем на узел Head. При обходе программа будет следовать указателям, пока не достигнет узла, на котором она была запущена.



Преимущество этой настройки заключается в том, что вы можете начать с любого узла и пройти по всему списку. Это также позволяет вам использовать связанные списки в качестве зацикленной структуры, задав желаемое количество циклов через структуру.

Списки с круговой связью отлично подходят для процессов, которые зацикливаются в течение длительного времени, например, распределение ЦП в операционных системах.

**Преимущества:**

* Может просматривать весь список, начиная с любого узла.
* Делает связанные списки более подходящими для циклических структур.

**Недостатки:**

* Сложнее найти узлы Head и Tail списка без nullмаркера.

**Приложения:**

* Регулярно зацикливающиеся решения, такие как планирование ЦП.
* Решения, в которых вам нужна свобода начать обход с любого узла.

**Общие вопросы собеседования со связным списком в Python**

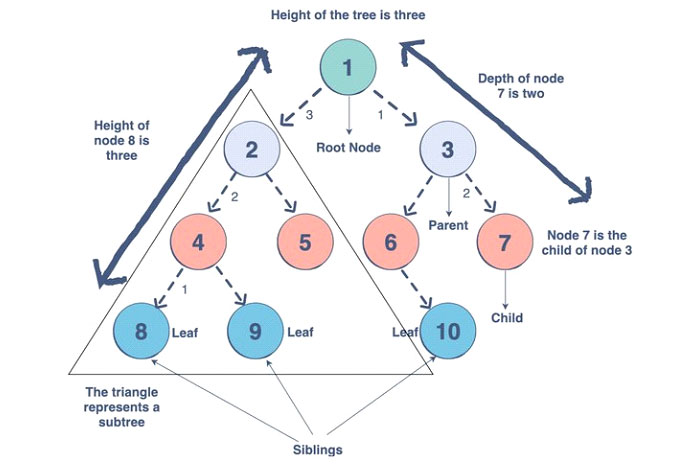
* Обнаружить петлю в связанных списках.
* Перевернуть круговой связанный список.
* Обратный круговой связанный список в группах заданного размера.

**Деревья в Python**

Деревья — это ещё одна основанная на отношениях структура данных, которая специализируется на представлении иерархических структур. Как и связанный список, они заполняются Nodeобъектами, которые содержат значение данных и один или несколько указателей для определения его отношения к непосредственным узлам.

Каждое дерево имеет корневой узел, от которого отходят все остальные узлы. Корень содержит указатели на все элементы непосредственно под ним, которые известны как его дочерние узлы. Эти дочерние узлы могут иметь собственные дочерние узлы. У двоичных деревьев не может быть узлов с более чем двумя дочерними узлами.

Любые узлы на одном уровне называются одноуровневыми узлами. Узлы без подключённых дочерних узлов называются листовыми узлами.



Наиболее распространённое применение двоичного дерева — это двоичное дерево поиска. Деревья двоичного поиска превосходно подходят для поиска больших наборов данных, поскольку временная сложность зависит от глубины дерева, а не от количества узлов.

Деревья двоичного поиска имеют четыре строгих правила:

* Левое поддерево содержит только узлы с элементами меньше корня.
* Правое поддерево содержит только узлы с элементами больше корня.
* Левое и правое поддеревья также должны быть двоичным деревом поиска. Они должны следовать приведённым выше правилам с «корнем» своего дерева.
* Не может быть повторяющихся узлов, т.е. никакие два узла не могут иметь одинаковое значение.

class Node:

    def \_\_init\_\_(self, data):

        self.left = None

        self.right = None

        self.data = data

    def insert(self, data):

# Compare the new value with the parent node

        if self.data:

            if data < self.data:

                if self.left is None:

                    self.left = Node(data)

                else:

                    self.left.insert(data)

            elif data > self.data:

                if self.right is None:

                    self.right = Node(data)

                else:

                    self.right.insert(data)

        else:

            self.data = data

# Print the tree

    def PrintTree(self):

        if self.left:

            self.left.PrintTree()

        print( self.data),

        if self.right:

            self.right.PrintTree()

# Use the insert method to add nodes

root = Node(12)

root.insert(6)

root.insert(14)

root.insert(3)

root.PrintTree()

**Преимущества:**

* Подходит для представления иерархических отношений.
* Динамический размер, отличный масштаб.
* Операции быстрой вставки и удаления.
* В двоичном дереве поиска вставленные узлы сразу же упорядочиваются..
* Деревья двоичного поиска эффективны при поиске; длина толькоO (высота)О ( ч е я г ч т ).

**[Читайте также:  Метод декодирования строки Python](https://bestprogrammer.ru/programmirovanie-i-razrabotka/metod-dekodirovaniya-stroki-python" \t "_blank)**

**Недостатки:**

* Время дорогое, O (войти) 4O ( l o g n ) 4, чтобы изменить или «сбалансировать» деревья или извлечь элементы из известного местоположения.
* Дочерние узлы не содержат информации о своих родительских узлах, и их трудно перемещать назад.
* Работает только для отсортированных списков. Несортированные данные превращаются в линейный поиск.

**Приложения:**

* Отлично подходит для хранения иерархических данных, таких как расположение файла.
* Используется для реализации лучших алгоритмов поиска и сортировки, таких как двоичные деревья поиска и двоичные кучи.

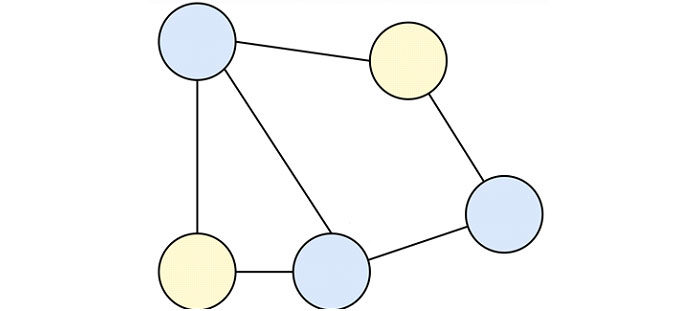
**Общие вопросы на собеседовании с деревом в Python**

* Проверьте, идентичны ли два двоичных дерева.
* Реализовать обход порядка уровней бинарного дерева.
* Распечатайте периметр двоичного дерева поиска.
* Суммируйте все узлы на пути.
* Подключите всех братьев и сестёр двоичного дерева.

**Графы в Python**

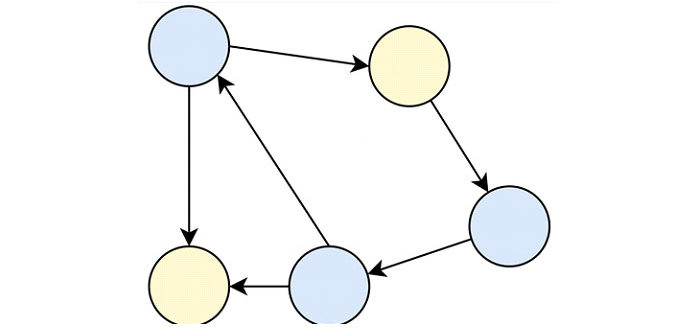
Графы — это структура данных, используемая для визуального представления взаимосвязей между вершинами данных (узлами графа). Связи, соединяющие вершины вместе, называются рёбрами.

Рёбра определяют, какие вершины соединяются, но не указывают направление потока между ними. Каждая вершина имеет соединения с другими вершинами, которые сохраняются в вершине в виде списка, разделённого запятыми.



Undirected

Существуют также специальные графы, называемые ориентированными графами, которые определяют направление взаимосвязи, аналогично связному списку. Направленные графы полезны при моделировании односторонних отношений или структур, подобных блок-схемам.



Directed

Они в основном используются для передачи визуальных сетей веб-структур в кодовой форме. Эти структуры могут моделировать множество различных типов отношений, таких как иерархии, ветвящиеся структуры или просто быть неупорядоченной реляционной сетью. Универсальность и интуитивность графиков делают их фаворитом в науке о данных.

При написании в виде обычного текста графы имеют список вершин и рёбер:

V = {a, b, c, d, e}

E = {ab, ac, bd, cd, de}

В Python графы лучше всего реализовать с использованием словаря с именем каждой вершины в качестве ключа и списком рёбер в качестве значений.

# Create the dictionary with graph elements

graph = { «a» : [«b»,»c»],

                 «b» : [«a», «d»],

                 «c» : [«a», «d»],

                  «d» : [«e»],

                  «e» : [«d»]

         }

# Print the graph

print(graph)

**Преимущества:**

* Быстро передавайте визуальную информацию с помощью кода.
* Подходит для моделирования широкого спектра реальных проблем.
* Простой в освоении синтаксис.

**Недостатки:**

* Связи вершин трудно понять в больших графах.
* Дорогое время для анализа данных с графика.

**Приложения:**

* Отлично подходит для моделирования сетей или веб-структур.
* Используется для моделирования сайтов социальных сетей, таких как Facebook.

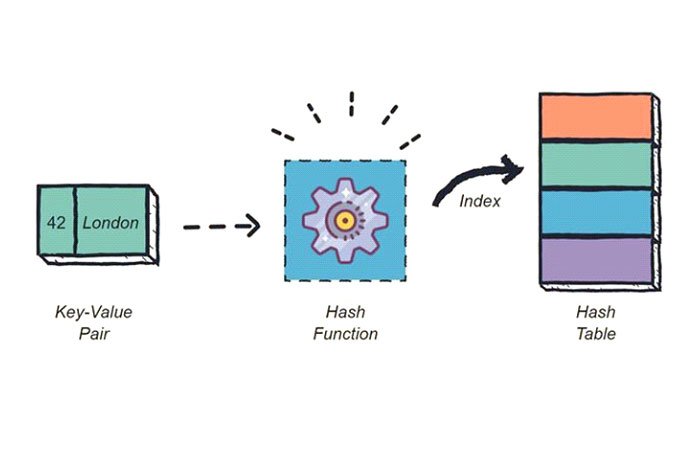
**Общие вопросы собеседования с графиком в Python**

* Обнаружить цикл в ориентированном графе.
* Найдите «материнскую вершину» в ориентированном графе.
* Подсчитать количество рёбер в неориентированном графе.
* Проверить, существует ли путь между двумя вершинами.
* Найдите кратчайший путь между двумя вершинами.

**Хеш-таблицы в Python**

Хеш-таблицы — это сложная структура данных, способная хранить большие объёмы информации и эффективно извлекать определённые элементы.

В этой структуре данных используются пары ключ / значение, где ключ — это имя желаемого элемента, а значение — это данные, хранящиеся под этим именем.



Каждый входной ключ проходит через хеш-функцию, которая преобразует его из начальной формы в целочисленное значение, называемое хешем. Хеш-функции всегда должны генерировать один и тот же хэш из одного и того же ввода, должны быстро вычислять и выдавать значения фиксированной длины. Python включает встроенную hash()функцию, ускоряющую реализацию.

Затем таблица использует хеш-код для нахождения общего местоположения желаемого значения, называемого корзиной хранения. После этого программе нужно будет искать нужное значение только в этой подгруппе, а не во всём пуле данных.

Помимо этой общей структуры, хеш-таблицы могут сильно отличаться в зависимости от приложения. Некоторые могут разрешать ключи из разных типов данных, в то время как некоторые могут иметь разные настройки сегментов или разные хеш-функции.

Вот пример хеш-таблицы в коде Python:

import pprint

class Hashtable:

    def \_\_init\_\_(self, elements):

        self.bucket\_size = len(elements)

        self.buckets = [[] for i in range(self.bucket\_size)]

        self.\_assign\_buckets(elements)

    def \_assign\_buckets(self, elements):

        for key, value in elements: #calculates the hash of each key

            hashed\_value = hash(key)

            index = hashed\_value % self.bucket\_size # positions the element in the bucket using hash

            self.buckets[index].append((key, value)) #adds a tuple in the bucket

    def get\_value(self, input\_key):

        hashed\_value = hash(input\_key)

        index = hashed\_value % self.bucket\_size

        bucket = self.buckets[index]

        for key, value in bucket:

            if key == input\_key:

                return(value)

        return None

    def \_\_str\_\_(self):

        return pprint.pformat(self.buckets) # pformat returns a printable representation of the object

if \_\_name\_\_ == «\_\_main\_\_»:

     capitals = [

        (‘France’, ‘Paris’),

        (‘United States’, ‘Washington D.C.’),

        (‘Italy’, ‘Rome’),

        (‘Canada’, ‘Ottawa’)

    ]

hashtable = Hashtable(capitals)

print(hashtable)

print(f»The capital of Italy is {hashtable.get\_value(‘Italy’)}»)

**Преимущества:**

* Может скрывать ключи в любой форме в целочисленные индексы.
* Чрезвычайно эффективен для больших наборов данных.
* Очень эффективная функция поиска.
* Постоянное количество шагов для каждого поиска и постоянная эффективность добавления или удаления элементов.
* Оптимизирован в Python 3.

**Недостатки:**

* Хэши должны быть уникальными, преобразование двух ключей в один и тот же хэш вызывает ошибку коллизии.
* Ошибки коллизии требуют полного пересмотра хэш-функции.
* Сложно построить для новичков.

**Приложения:**

* Используется для больших баз данных, в которых часто ведётся поиск.
* Системы поиска, использующие клавиши ввода.

**Общие вопросы собеседования с хеш-таблицей в Python**

* Создать хеш-таблицу с нуля (без встроенных функций).
* Формирование слова с помощью хеш-таблицы.
* Найдите два числа, которые в сумме дают «k».
* Реализовать открытую адресацию для обработки коллизий.
* Определите, является ли список циклическим, используя хеш-таблицу.

**Заключение**

Для каждой из этих 8 структур данных существуют десятки вопросов и форматов интервью. Лучший способ подготовиться к собеседованию — продолжать решать практические задачи.

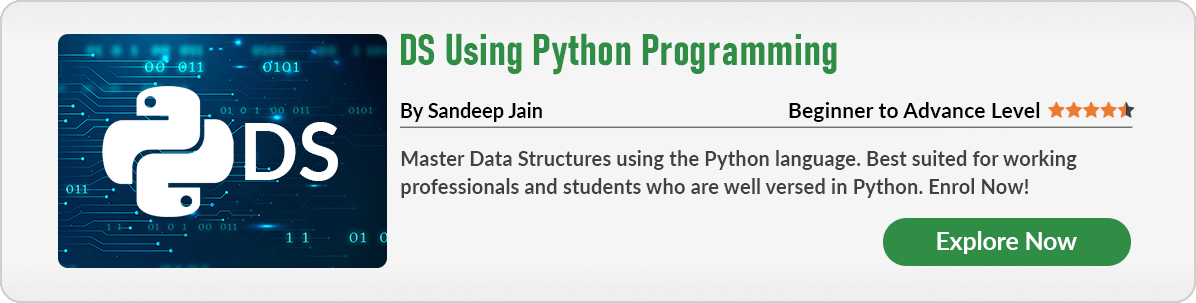


<https://translated.turbopages.org/proxy_u/en-ru.ru.32391eaf-634dd463-9e272939-74722d776562/https/www.geeksforgeeks.org/python-data-structures-and-algorithms/>

Это руководство для начинающих по изучению структур данных и алгоритмов с использованием Python. В этой статье мы обсудим встроенные структуры данных, такие как списки, кортежи, словари и т. Д., И Некоторые пользовательские структуры данных, такие как связанные списки, деревья, графики и т. Д., А Также алгоритмы обхода, а также поиска и сортировки с помощью хороших и хорошо объясненных примерови практические вопросы.

## Списки

[Списки Python](https://translated.turbopages.org/proxy_u/en-ru.ru.d4b8a886-634c7730-2139cb51-74722d776562/https/www.geeksforgeeks.org/python-list/) - это упорядоченные Множестваданных, как и массивы в других языках программирования. Это позволяет использовать различные типы элементов в списке. Реализация списка Python похожа на векторы в C ++ или ArrayList в JAVA. Дорогостоящей операцией является вставка или удаление элемента из начала списка, поскольку все элементы необходимо сдвинуть. Вставка и удаление в конце списка также могут стать дорогостоящими в случае, когда предварительно выделенная память заполняется.

[](https://translated.turbopages.org/proxy_u/en-ru.ru.d4b8a886-634c7730-2139cb51-74722d776562/https/practice.geeksforgeeks.org/courses/Data-Structures-With-Python?utm_source=article&utm_medium=article&utm_campaign=data-structures-with-python-self-paced-gfg-page)

### Пример: Создание списка Python

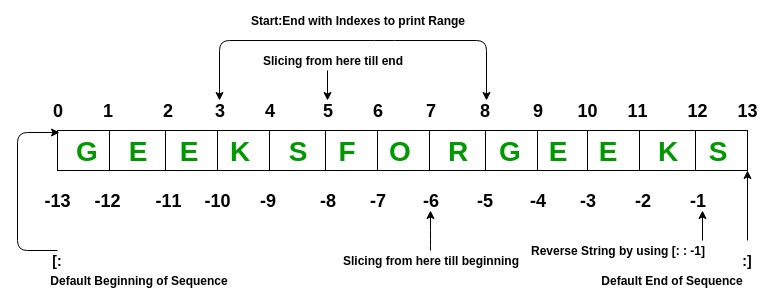
* Python3

|  |
| --- |
| List **=** [1, 2, 3, "GFG", 2.3]  print(List) |

**Вывод**

[1, 2, 3, 'GFG', 2.3]

Доступ к элементам списка можно получить по присвоенному индексу. В python начальный индекс списка, последовательность равна 0, а конечный индекс равен (если есть N элементов) N-1.



### Пример: Операции со списком Python

* Python3

|  |
| --- |
| # Creating a List with  # the use of multiple values  List **=** ["Geeks", "For", "Geeks"]  print("\nList containing multiple values: ")  print(List)    # Creating a Multi-Dimensional List  # (By Nesting a list inside a List)  List2 **=** [['Geeks', 'For'], ['Geeks']]  print("\nMulti-Dimensional List: ")  **print**(List2)    # accessing a element from the  # list using index number  print("Accessing element from the list")  **print**(List[0])  print(List[2])    # accessing a element using  # negative indexing  **print**("Accessing element using negative indexing")    # print the last element of list  **print**(List[**-**1])    # print the third last element of list  **print**(List[**-**3]) |

**Вывод**

Список, содержащий несколько значений:

['Geeks', 'For', 'Geeks']

Многомерный список:

[['Geeks', 'For'], ['Geeks']]

Доступ к элементу из списка

Гики

Гики

Доступ к элементу с использованием отрицательной индексации

Гики

Гики

## Кортеж

[Кортежи Python](https://translated.turbopages.org/proxy_u/en-ru.ru.d4b8a886-634c7730-2139cb51-74722d776562/https/www.geeksforgeeks.org/python-tuples/) похожи на списки, но кортежи [неизменяемы](https://translated.turbopages.org/proxy_u/en-ru.ru.d4b8a886-634c7730-2139cb51-74722d776562/https/www.geeksforgeeks.org/why-do-we-need-immutables-in-python/) по своей природе, т.e. После создания они не могут быть изменены. Как и список, кортеж также может содержать элементы различных типов.

В Python кортежи создаются путем размещения последовательности значений, разделенных запятой, с использованием или без использования круглых скобок для группировки последовательности данных.

**Примечание:** Для создания кортежа из одного элемента в конце должна быть запятая. Например, (8,) создаст кортеж, содержащий 8 в качестве элемента.

### Пример: операции с кортежами Python

* Python3

|  |
| --- |
| # Creating a Tuple with  # the use of Strings  Tuple **=** ('Geeks', 'For')  **print**("\nTuple with the use of String: ")  **print**(Tuple)    # Creating a Tuple with  # the use of list  list1 **=** [1, 2, 4, 5, 6]  print("\nTuple using List: ")  Tuple **=** tuple(list1)    # Accessing element using indexing  **print**("First element of tuple")  **print**(Tuple[0])    # Accessing element from last  # negative indexing  print("\nLast element of tuple")  print(Tuple[**-**1])    **print**("\nThird last element of tuple")  print(Tuple[**-**3]) |

**Вывод**

Кортеж с использованием строки:

('Geeks', 'For')

Кортеж с использованием списка:

Первый элемент кортежа

1

Последний элемент кортежа

6

Третий последний элемент кортежа

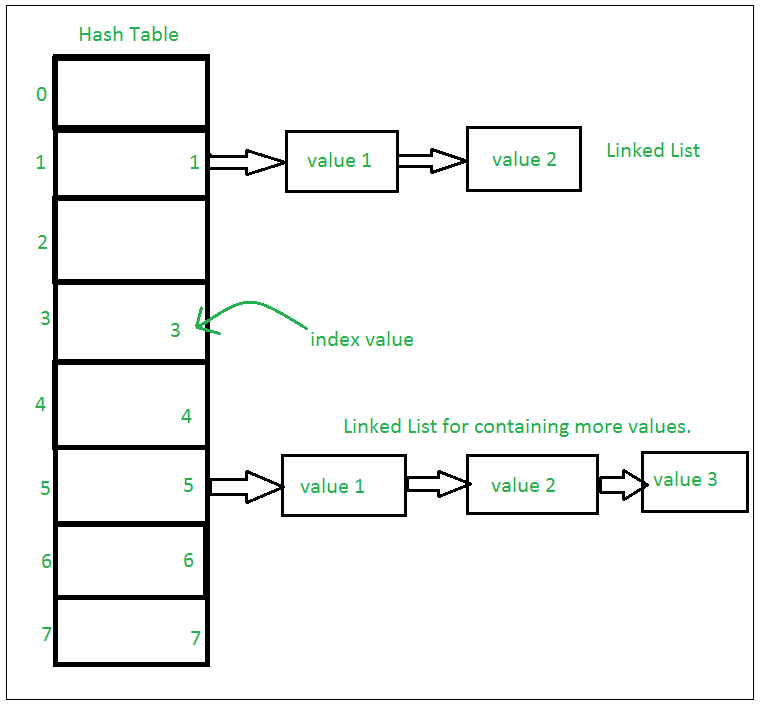
4

## Множество

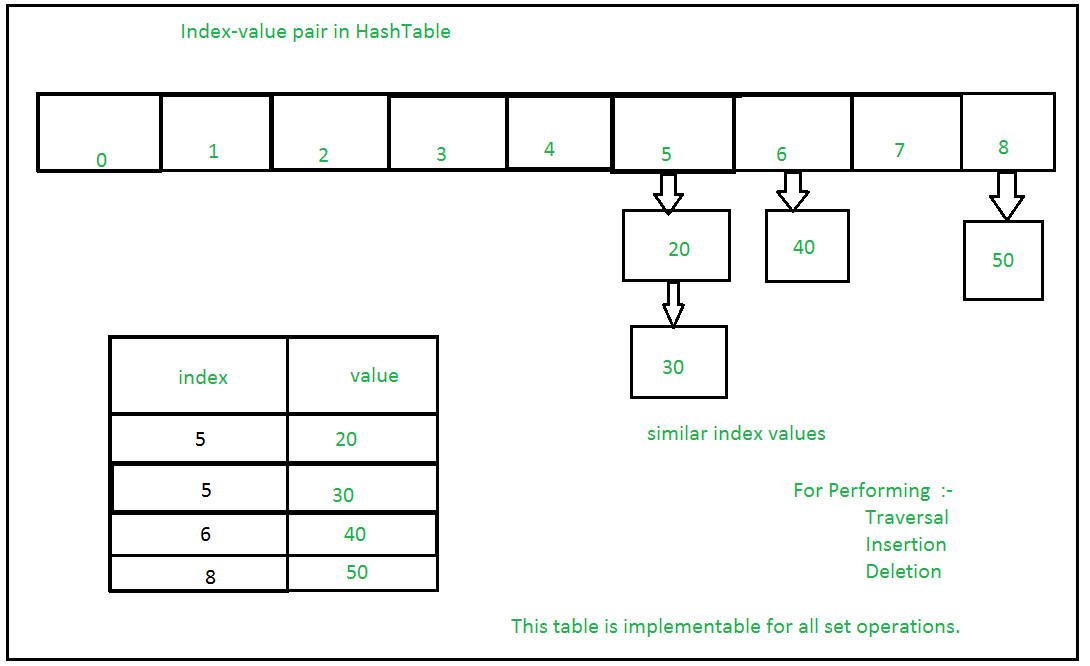
[МножествоPython](https://translated.turbopages.org/proxy_u/en-ru.ru.d4b8a886-634c7730-2139cb51-74722d776562/www.geeksforgeeks.org/sets-in-python/) - это изменяемое множество данных, который не допускает дублирования. Множества основном используются для тестирования членства и устранения повторяющихся записей. Структура данных, используемая в этом, - это хеширование, популярный метод для выполнения вставки, удаления и обхода в O (1) в среднем.

Если в одной и той же позиции индекса присутствует несколько значений, то значение добавляется к этой позиции индекса, чтобы сформировать связанный список. Множества CPython реализованы с использованием словаря с фиктивными переменными, где ключевыми являются элементы, заданные с большей оптимизацией для временной сложности.

**Установить реализацию:**



**Множества с множеством операций над одной хэш-таблицей:**



### Пример: операции набора Python

* Python3

|  |
| --- |
| # Creating a Set with  # a mixed type of values  # (Having numbers and strings)  Set **=** set([1, 2, 'Geeks', 4, 'For', 6, 'Geeks'])  **print**("\nSet with the use of Mixed Values")  **print**(Set)    # Accessing element using  # for loop  print("\nElements of set: ")  **for** i **in** Set:  **print**(i, end **=**" ")  print()    # Checking the element  # using in keyword  print("Geeks" **in** Set) |

**Вывод**

Множествос использованием смешанных значений

{1, 2, 4, 6, ' Для', 'Вундеркиндов'}

Элементы множества:

1 2 4 6 Для гиков

верно

## Замороженные множества

[Замороженные множества](https://translated.turbopages.org/proxy_u/en-ru.ru.d4b8a886-634c7730-2139cb51-74722d776562/https/www.geeksforgeeks.org/frozenset-in-python/) в Python являются неизменяемыми объектами, которые поддерживают только методы и операторы, которые выдают результат, не влияя на замороженный Множествоили Множества, к которым они применяются. В то время как элементы набора могут быть изменены в любое время, элементы замороженного набора остаются неизменными после создания.

### Пример: замороженный МножествоPython

* Python3

|  |
| --- |
| # Same as {"a", "b","c"}  normal\_set **=** set(["a", "b","c"])    **print**("Normal Set")  print(normal\_set)    # A frozen set  frozen\_set **=** frozenset(["e", "f", "g"])    **print**("\nFrozen Set")  **print**(frozen\_set)    # Uncommenting below line would cause error as  # we are trying to add element to a frozen set  # frozen\_set.add("h") |

**Вывод**

Обычный набор

{'a', 'b', 'c'}

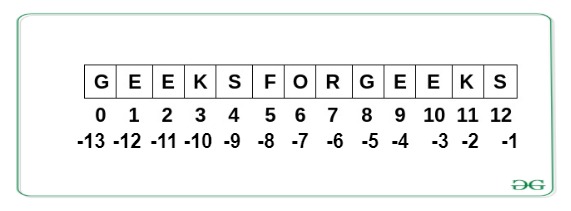
Замороженный набор

frozenset({'f', 'g', 'e'})

## Строка

[Строки Python](https://translated.turbopages.org/proxy_u/en-ru.ru.d4b8a886-634c7730-2139cb51-74722d776562/https/www.geeksforgeeks.org/python-strings/) - это неизменяемый массив байтов, представляющих символы Юникода. В Python нет символьного типа данных, один символ - это просто строка длиной 1.

**Примечание:** поскольку строки неизменяемы, изменение строки приведет к созданию новой копии.



### Пример: Операции со строками Python

* Python3

|  |
| --- |
| String **=** "Welcome to GeeksForGeeks"  **print**("Creating String: ")  **print**(String)    # Printing First character  **print**("\nFirst character of String is: ")  print(String[0])    # Printing Last character  print("\nLast character of String is: ")  **print**(String[**-**1]) |

**Вывод**

Создание строки:

Добро пожаловать в GeeksforGeeks

Первый символ строки:

W

Последний символ строки:

s

## Словарь

[Словарь Python](https://translated.turbopages.org/proxy_u/en-ru.ru.d4b8a886-634c7730-2139cb51-74722d776562/https/www.geeksforgeeks.org/python-dictionary/) - это неупорядоченный Множестводанных, который хранит данные в формате пары ключ: значение. Это похоже на хэш-таблицы на любом другом языке с временной сложностью O (1). Индексация словаря Python выполняется с помощью ключей. Они имеют любой хэшируемый тип, т.e. Объект, который никогда не может изменяться, как строки, числа, кортежи и т.д. Мы можем создать словарь, используя фигурные скобки ({}) или понимание словаря.

### Пример: операции со словарем Python

* Python3

|  |
| --- |
| # Creating a Dictionary  Dict **=** {'Name': 'Geeks', 1: [1, 2, 3, 4]}  **print**("Creating Dictionary: ")  print(Dict)    # accessing a element using key  **print**("Accessing a element using key:")  **print**(Dict['Name'])    # accessing a element using get()  # method  **print**("Accessing a element using get:")  print(Dict.get(1))    # creation using Dictionary comprehension  myDict **=** {x: x**\*\***2 **for** x **in** [1,2,3,4,5]}  print(myDict) |

**Вывод**

Создание словаря:

{'Name': 'Geeks', 1: [1, 2, 3, 4]}

Доступ к элементу с помощью ключа:

Гики

Доступ к элементу с помощью get:

[1, 2, 3, 4]

{1: 1, 2: 4, 3: 9, 4: 16, 5: 25}

## Матрица

Матрица - это 2D-массив, в котором каждый элемент имеет строго одинаковый размер. Для создания матрицы мы будем использовать [пакет NumPy](https://translated.turbopages.org/proxy_u/en-ru.ru.d4b8a886-634c7730-2139cb51-74722d776562/https/www.geeksforgeeks.org/python-numpy/).

### Пример: матричные операции Python NumPy

* Python3

|  |
| --- |
| **import** numpy as np    a **=** np.array([[1,2,3,4],[4,55,1,2],                [8,3,20,19],[11,2,22,21]])  m **=** np.reshape(a,(4, 4))  **print**(m)    # Accessing element  print("\nAccessing Elements")  **print**(a[1])  **print**(a[2][0])    # Adding Element  m **=** np.append(m,[[1, 15,13,11]],0)  print("\nAdding Element")  print(m)    # Deleting Element  m **=** np.delete(m,[1],0)  print("\nDeleting Element")  print(m) |

**Вывод**

[[ 1 2 3 4]

[ 4 55 1 2]

[ 8 3 20 19]

[11 2 22 21]]

Доступ к элементам

[ 4 55 1 2]

8

Добавление элемента

[[ 1 2 3 4]

[ 4 55 1 2]

[ 8 3 20 19]

[11 2 22 21]

[ 1 15 13 11]]

Удаление элемента

[[ 1 2 3 4]

[ 8 3 20 19]

[11 2 22 21]

[ 1 15 13 11]]

## Bytearray

Python Bytearray выдает изменяемую последовательность целых чисел в диапазоне 0 <= x < 256.

### Пример: операции с байтами Python

* Python3

|  |
| --- |
| # Creating bytearray  a **=** bytearray((12, 8, 25, 2))  print("Creating Bytearray:")  **print**(a)    # accessing elements  **print**("\nAccessing Elements:", a[1])    # modifying elements  a[1] **=** 3  **print**("\nAfter Modifying:")  print(a)    # Appending elements  a.append(30)  print("\nAfter Adding Elements:")  print(a) |

**Вывод**

Создание Bytearray:

bytearray(b'\x0c\ x08 \ x19 \x02')

Доступ к элементам: 8

После изменения:

bytearray(b'\x0c\x03\x19 \x02')

После добавления элементов:

bytearray(b'\x0c\x03\x19 \x02 \x1e')

## Связанный список

[Связанный список](https://translated.turbopages.org/proxy_u/en-ru.ru.d4b8a886-634c7730-2139cb51-74722d776562/https/www.geeksforgeeks.org/linked-list-set-1-introduction/) - это линейная структура данных, в которой элементы не хранятся в смежных ячейках памяти. Элементы в связанном списке связаны с помощью указателей, как показано на рисунке ниже:



Связанный список представлен указателем на первый узел связанного списка. Первый узел называется head. Если связанный список пуст, то значение head равно NULL. Каждый узел в списке состоит как минимум из двух частей:

* Данные
* Указатель (или ссылка) на следующий узел

### Пример: определение связанного списка в Python

* Python3

|  |
| --- |
| # Node class  **class** Node:        # Function to initialize the node object  **def** \_\_init\_\_(self, data):          self.data **=** data # Assign data          self.next **=** None # Initialize                          # next as null    # Linked List class  **class** LinkedList:        # Function to initialize the Linked      # List object  **def** \_\_init\_\_(self):          self.head **=** None |

Давайте создадим простой связанный список с 3 узлами.

* Python3

|  |
| --- |
| # A simple Python program to introduce a linked list    # Node class  **class** Node:        # Function to initialise the node object  **def** \_\_init\_\_(self, data):          self.data **=** data # Assign data          self.next **=** None # Initialize next as null      # Linked List class contains a Node object  **class** LinkedList:        # Function to initialize head  **def** \_\_init\_\_(self):          self.head **=** None      # Code execution starts here  **if** \_\_name\_\_**==**'\_\_main\_\_':        # Start with the empty list      llist **=** LinkedList()        llist.head **=** Node(1)      second **=** Node(2)      third **=** Node(3)        '''      Three nodes have been created.      We have references to these three blocks as head,      second and third        llist.head     second             third          |             |                 |          |             |                 |      +----+------+     +----+------+     +----+------+      | 1 | None |     | 2 | None |     | 3 | None |      +----+------+     +----+------+     +----+------+      '''        llist.head.next **=** second; # Link first node with second        '''      Now next of first Node refers to second. So they      both are linked.        llist.head     second             third          |             |                 |          |             |                 |      +----+------+     +----+------+     +----+------+      | 1 | o-------->| 2 | null |     | 3 | null |      +----+------+     +----+------+     +----+------+      '''        second.next **=** third; # Link second node with the third node        '''      Now next of second Node refers to third. So all three      nodes are linked.        llist.head     second             third          |             |                 |          |             |                 |      +----+------+     +----+------+     +----+------+      | 1 | o-------->| 2 | o-------->| 3 | null |      +----+------+     +----+------+     +----+------+      ''' |

### Обход связанного списка

В предыдущей программе мы создали простой связанный список с тремя узлами. Давайте пройдемся по созданному списку и выведем данные каждого узла. Для обхода давайте напишем функцию общего назначения printList(), которая печатает любой заданный список.

* Python3

|  |
| --- |
| # A simple Python program for traversal of a linked list    # Node class  **class** Node:        # Function to initialise the node object  **def** \_\_init\_\_(self, data):          self.data **=** data # Assign data          self.next **=** None # Initialize next as null      # Linked List class contains a Node object  **class** LinkedList:        # Function to initialize head  **def** \_\_init\_\_(self):          self.head **=** None        # This function prints contents of linked list      # starting from head  **def** printList(self):          temp **=** self.head  **while** (temp):  **print** (temp.data)              temp **=** temp.next      # Code execution starts here  **if** \_\_name\_\_**==**'\_\_main\_\_':        # Start with the empty list      llist **=** LinkedList()        llist.head **=** Node(1)      second **=** Node(2)      third **=** Node(3)        llist.head.next **=** second; # Link first node with second      second.next **=** third; # Link second node with the third node        llist.printList() |

**Вывод**

1

2

3

### Другие статьи по связанному списку

* [Вставка связанного списка](https://translated.turbopages.org/proxy_u/en-ru.ru.d4b8a886-634c7730-2139cb51-74722d776562/www.geeksforgeeks.org/linked-list-set-2-inserting-a-node/)
* [Удаление связанного списка (удаление заданного ключа)](https://translated.turbopages.org/proxy_u/en-ru.ru.d4b8a886-634c7730-2139cb51-74722d776562/www.geeksforgeeks.org/linked-list-set-3-deleting-node/)
* [Удаление связанного списка (удаление ключа в заданной позиции)](https://translated.turbopages.org/proxy_u/en-ru.ru.d4b8a886-634c7730-2139cb51-74722d776562/www.geeksforgeeks.org/delete-a-linked-list-node-at-a-given-position/)
* [Найти длину связанного списка (итеративный и рекурсивный)](https://translated.turbopages.org/proxy_u/en-ru.ru.d4b8a886-634c7730-2139cb51-74722d776562/www.geeksforgeeks.org/find-length-of-a-linked-list-iterative-and-recursive/)
* [Поиск элемента в связанном списке (итеративный и рекурсивный)](https://translated.turbopages.org/proxy_u/en-ru.ru.d4b8a886-634c7730-2139cb51-74722d776562/www.geeksforgeeks.org/search-an-element-in-a-linked-list-iterative-and-recursive/)
* [N-й узел из конца связанного списка](https://translated.turbopages.org/proxy_u/en-ru.ru.d4b8a886-634c7730-2139cb51-74722d776562/https/www.geeksforgeeks.org/nth-node-from-the-end-of-a-linked-list/)
* [Обратный связанный список](https://translated.turbopages.org/proxy_u/en-ru.ru.d4b8a886-634c7730-2139cb51-74722d776562/https/www.geeksforgeeks.org/write-a-function-to-reverse-the-nodes-of-a-linked-list/)

[**>>> Подробнее**](https://translated.turbopages.org/proxy_u/en-ru.ru.d4b8a886-634c7730-2139cb51-74722d776562/https/www.geeksforgeeks.org/data-structures/linked-list/singly-linked-list/)

## Стек

[Стек](https://translated.turbopages.org/proxy_u/en-ru.ru.d4b8a886-634c7730-2139cb51-74722d776562/https/www.geeksforgeeks.org/stack-in-python/) - это линейная структура данных, в которой элементы хранятся в порядке ввода/ вывода последним (LIFO) или ввода / вывода первым (FILO). В stack новый элемент добавляется на одном конце, а элемент удаляется только с этого конца. Операции вставки и удаления часто называются push и pop.



**Функции, связанные со стеком, являются:**

* **empty() –** возвращает, является ли стек пустым – Временная сложность: O(1)
* **size() –** возвращает размер стека - Временная сложность: O(1)
* **top() –** возвращает ссылку на самый верхний элемент стека - временная сложность: O(1)
* **push(a) –** вставляет элемент ‘a’ в верхнюю часть стека – Временная сложность: O (1)
* **pop() –** удаляет самый верхний элемент стека - временная сложность: O(1)
* Python3

|  |
| --- |
| stack **=** []    # append() function to push  # element in the stack  stack.append('g')  stack.append('f')  stack.append('g')    print('Initial stack')  print(stack)    # pop() function to pop  # element from stack in  # LIFO order  print('\nElements popped from stack:')  print(stack.pop())  **print**(stack.pop())  print(stack.pop())    print('\nStack after elements are popped:')  print(stack)    # uncommenting print(stack.pop())  # will cause an IndexError  # as the stack is now empty |

**Вывод**

Начальный стек

['g', 'f', 'g']

Элементы, извлеченные из стека:

g

f

g

Стек после извлечения элементов:

[]

### Другие статьи о стеке

* [Преобразование инфикса в постфикс с использованием стека](https://translated.turbopages.org/proxy_u/en-ru.ru.d4b8a886-634c7730-2139cb51-74722d776562/https/www.geeksforgeeks.org/stack-set-2-infix-to-postfix/)
* [Преобразование префикса в инфикс](https://translated.turbopages.org/proxy_u/en-ru.ru.d4b8a886-634c7730-2139cb51-74722d776562/https/www.geeksforgeeks.org/prefix-infix-conversion/)
* [Преобразование префикса в постфикс](https://translated.turbopages.org/proxy_u/en-ru.ru.d4b8a886-634c7730-2139cb51-74722d776562/https/www.geeksforgeeks.org/prefix-postfix-conversion/)
* [Преобразование постфикса в префикс](https://translated.turbopages.org/proxy_u/en-ru.ru.d4b8a886-634c7730-2139cb51-74722d776562/https/www.geeksforgeeks.org/postfix-prefix-conversion/)
* [Постфикс в инфикс](https://translated.turbopages.org/proxy_u/en-ru.ru.d4b8a886-634c7730-2139cb51-74722d776562/https/www.geeksforgeeks.org/postfix-to-infix/)
* [Проверьте наличие сбалансированных круглых скобок в выражении](https://translated.turbopages.org/proxy_u/en-ru.ru.d4b8a886-634c7730-2139cb51-74722d776562/https/www.geeksforgeeks.org/check-for-balanced-parentheses-in-an-expression/)
* [Вычисление постфиксного выражения](https://translated.turbopages.org/proxy_u/en-ru.ru.d4b8a886-634c7730-2139cb51-74722d776562/https/www.geeksforgeeks.org/stack-set-4-evaluation-postfix-expression/)

[**>>> Подробнее**](https://translated.turbopages.org/proxy_u/en-ru.ru.d4b8a886-634c7730-2139cb51-74722d776562/https/www.geeksforgeeks.org/stack-data-structure/)

## Очередь

Как стек, [очередь](https://translated.turbopages.org/proxy_u/en-ru.ru.d4b8a886-634c7730-2139cb51-74722d776562/https/www.geeksforgeeks.org/queue-in-python/) представляет собой линейную структуру данных, в которой элементы хранятся в порядке поступления первыми (FIFO). При использовании очереди сначала удаляется наименее недавно добавленный элемент. Хорошим примером очереди является любая очередь потребителей для ресурса, где первым обслуживается потребитель, пришедший первым.



**Операции, связанные с очередью, являются:**

* **Поставить в очередь:** добавляет элемент в очередь. Если очередь заполнена, то это называется условием переполнения – Временная сложность: O (1)
* **Удаление из очереди:** удаляет элемент из очереди. Элементы отображаются в том же порядке, в котором они были вставлены. Если очередь пуста, то это называется условием недостаточного потока – Временная сложность: O (1)
* **Фронт:** получение начального элемента из очереди – временная сложность: O (1)
* **Сзади:** получить последний элемент из очереди – временная сложность: O (1)
* Python3

|  |
| --- |
| # Initializing a queue  queue **=** []    # Adding elements to the queue  queue.append('g')  queue.append('f')  queue.append('g')    print("Initial queue")  **print**(queue)    # Removing elements from the queue  **print**("\nElements dequeued from queue")  **print**(queue.pop(0))  print(queue.pop(0))  **print**(queue.pop(0))    print("\nQueue after removing elements")  print(queue)    # Uncommenting print(queue.pop(0))  # will raise and IndexError  # as the queue is now empty |

**Вывод**

Начальная очередь

['g', 'f', 'g']

Элементы, удаленные из очереди

g

f

g

Очередь после удаления элементов

[]

### Другие статьи о очереди

* [Реализация очереди с использованием стеков](https://translated.turbopages.org/proxy_u/en-ru.ru.d4b8a886-634c7730-2139cb51-74722d776562/https/www.geeksforgeeks.org/queue-using-stacks/)
* [Реализация стека с использованием очередей](https://translated.turbopages.org/proxy_u/en-ru.ru.d4b8a886-634c7730-2139cb51-74722d776562/https/www.geeksforgeeks.org/implement-stack-using-queue/)
* [Реализация стека с использованием единой очереди](https://translated.turbopages.org/proxy_u/en-ru.ru.d4b8a886-634c7730-2139cb51-74722d776562/https/www.geeksforgeeks.org/implement-a-stack-using-single-queue/)

## Приоритетная очередь

[Приоритетные очереди](https://translated.turbopages.org/proxy_u/en-ru.ru.d4b8a886-634c7730-2139cb51-74722d776562/https/www.geeksforgeeks.org/priority-queue-in-python/) - это абстрактные структуры данных, в которых каждое значение / значение в очереди имеет определенный приоритет. Например, в авиакомпаниях багаж с заголовком “Бизнес” или “Первый класс” прибывает раньше остальных. Приоритетная очередь - это расширение очереди со следующими свойствами.

* Элемент с высоким приоритетом удаляется из очереди перед элементом с низким приоритетом.
* Если два элемента имеют одинаковый приоритет, они обслуживаются в соответствии с их порядком в очереди.
* Python3

|  |
| --- |
| # A simple implementation of Priority Queue  # using Queue.  **class** PriorityQueue(object):  **def** \_\_init\_\_(self):          self.queue **=** []    **def** \_\_str\_\_(self):  **return** ' '.join([str(i) **for** i **in** self.queue])        # for checking if the queue is empty  **def** isEmpty(self):  **return** len(self.queue) **==** 0        # for inserting an element in the queue  **def** insert(self, data):          self.queue.append(data)        # for popping an element based on Priority  **def** delete(self):  **try**:              max **=** 0  **for** i **in** range(len(self.queue)):  **if** self.queue[i] > self.queue[max]:                      max **=** i              item **=** self.queue[max]  **del** self.queue[max]  **return** item  **except** IndexError:  **print**()              exit()    **if** \_\_name\_\_ **==** '\_\_main\_\_':      myQueue **=** PriorityQueue()      myQueue.insert(12)      myQueue.insert(1)      myQueue.insert(14)      myQueue.insert(7)  **print**(myQueue)  **while** **not** myQueue.isEmpty():  **print**(myQueue.delete()) |

**Вывод**

12 1 14 7

14

12

7

1

## Куча

[модуль heapq в Python](https://translated.turbopages.org/proxy_u/en-ru.ru.d4b8a886-634c7730-2139cb51-74722d776562/https/www.geeksforgeeks.org/heap-queue-or-heapq-in-python/) предоставляет структуру данных кучи, которая в основном используется для представления очереди приоритетов. Свойство этой структуры данных заключается в том, что она всегда выдает наименьший элемент (минимальную кучу) всякий раз, когда элемент извлекается. Всякий раз, когда элементы выталкиваются или выталкиваются, структура кучи сохраняется. Элемент heap[0] также каждый раз возвращает наименьший элемент. Он поддерживает извлечение и вставку наименьшего элемента в O (log n) раз.

Как правило, кучи могут быть двух типов:

* **Максимальная куча:** в максимальной куче ключ, присутствующий в корневом узле, должен быть наибольшим среди ключей, присутствующих во всех его дочерних элементах. Одно и то же свойство должно быть рекурсивно верно для всех поддеревьев в этом двоичном дереве.
* **Минимальная куча:** в минимальной куче ключ, присутствующий в корневом узле, должен быть минимальным среди ключей, присутствующих во всех его дочерних элементах. Одно и то же свойство должно быть рекурсивно верно для всех поддеревьев в этом двоичном дереве.



* Python3

|  |
| --- |
| # importing "heapq" to implement heap queue  **import** heapq    # initializing list  li **=** [5, 7, 9, 1, 3]    # using heapify to convert list into heap  heapq.heapify(li)    # printing created heap  print ("The created heap is : ",end**=**"")  print (list(li))    # using heappush() to push elements into heap  # pushes 4  heapq.heappush(li,4)    # printing modified heap  **print** ("The modified heap after push is : ",end**=**"")  print (list(li))    # using heappop() to pop smallest element  print ("The popped and smallest element is : ",end**=**"")  print (heapq.heappop(li)) |

**Вывод**

Созданная куча : [1, 3, 9, 7, 5]

Измененная куча после нажатия : [1, 3, 4, 7, 5, 9]

Всплывающий и наименьший элемент равен: 1

### Другие статьи о куче

* [Двоичная куча](https://translated.turbopages.org/proxy_u/en-ru.ru.d4b8a886-634c7730-2139cb51-74722d776562/geeksquiz.com/binary-heap/)
* [K-й по величине элемент в массиве](https://translated.turbopages.org/proxy_u/en-ru.ru.d4b8a886-634c7730-2139cb51-74722d776562/https/www.geeksforgeeks.org/k-largestor-smallest-elements-in-an-array/)
* [K-й наименьший / наибольший элемент в несортированном массиве](https://translated.turbopages.org/proxy_u/en-ru.ru.d4b8a886-634c7730-2139cb51-74722d776562/https/www.geeksforgeeks.org/kth-smallestlargest-element-unsorted-array/)
* [Сортировка почти отсортированного массива](https://translated.turbopages.org/proxy_u/en-ru.ru.d4b8a886-634c7730-2139cb51-74722d776562/https/www.geeksforgeeks.org/nearly-sorted-algorithm/)
* [K-й непрерывный подмассив с наибольшей суммой](https://translated.turbopages.org/proxy_u/en-ru.ru.d4b8a886-634c7730-2139cb51-74722d776562/https/www.geeksforgeeks.org/k-th-largest-sum-contiguous-subarray/)
* [Минимальная сумма двух чисел, сформированных из цифр массива](https://translated.turbopages.org/proxy_u/en-ru.ru.d4b8a886-634c7730-2139cb51-74722d776562/https/www.geeksforgeeks.org/minimum-sum-two-numbers-formed-digits-array-2/)

[**>>> Подробнее**](https://translated.turbopages.org/proxy_u/en-ru.ru.d4b8a886-634c7730-2139cb51-74722d776562/https/www.geeksforgeeks.org/heap-data-structure/)

## Двоичное дерево

Дерево - это иерархическая структура данных, которая выглядит как показано на рисунке ниже –

дерево

----

j <-- root

/ \

f k

/ \ \

a h z <-- оставляет

Самый верхний узел дерева называется корневым, тогда как самые нижние узлы или узлы, не имеющие дочерних элементов, называются конечными узлами. Узлы, которые находятся непосредственно под узлом, называются его дочерними, а узлы, которые находятся непосредственно над чем-либо, называются его родительскими.

[Двоичное дерево](https://translated.turbopages.org/proxy_u/en-ru.ru.d4b8a886-634c7730-2139cb51-74722d776562/https/www.geeksforgeeks.org/binary-tree-set-1-introduction/) - это дерево, элементы которого могут иметь почти двух дочерних элементов. Поскольку у каждого элемента в двоичном дереве может быть только 2 дочерних элемента, мы обычно называем их левыми и правыми дочерними элементами. Узел двоичного дерева содержит следующие части.

* Данные
* Указатель на левый дочерний элемент
* Указатель на правильный дочерний элемент

### Пример: определение класса узла

* Python3

|  |
| --- |
| # A Python class that represents an individual node  # in a Binary Tree  **class** Node:  **def** \_\_init\_\_(self,key):          self.left **=** None          self.right **=** None          self.val **=** key |

Теперь давайте создадим дерево с 4 узлами на Python. Давайте предположим, что древовидная структура выглядит следующим образом –

дерево

----

1 <-- root

/ \

2 3

/

4

### Пример: добавление данных в дерево

* Python3

|  |
| --- |
| # Python program to introduce Binary Tree    # A class that represents an individual node in a  # Binary Tree  **class** Node:  **def** \_\_init\_\_(self,key):          self.left **=** None          self.right **=** None          self.val **=** key      # create root  root **=** Node(1)  ''' following is the tree after above statement          1      / \      None None'''    root.left     **=** Node(2);  root.right     **=** Node(3);    ''' 2 and 3 become left and right children of 1          1          / \          2     3      / \ / \  None None None None'''      root.left.left **=** Node(4);  '''4 becomes left child of 2          1      /     \      2         3      / \     / \  4 None None None  / \  None None''' |

## Обход дерева

[По деревьям можно перемещаться](https://translated.turbopages.org/proxy_u/en-ru.ru.d4b8a886-634c7730-2139cb51-74722d776562/https/www.geeksforgeeks.org/tree-traversals-inorder-preorder-and-postorder/) разными способами. Ниже приведены обычно используемые способы обхода деревьев. Давайте рассмотрим приведенное ниже дерево –

дерево

----

1 <-- root

/ \

2 3

/ \

4 5

**Обходы в глубину:**

* В порядке (слева, корень, справа) : 4 2 5 1 3
* Предварительный заказ (корневой, левый, правый) : 1 2 4 5 3
* Порядок следования (слева, справа, корень) : 4 5 2 3 1

**Алгоритм в порядке (дерево)**

* Пройдите по левому поддереву, т.е. вызовите Inorder(левое поддерево)
* Зайдите в корень.
* Пройдите по правому поддереву, т.е. Вызовите Inorder(правое поддерево)

**Предварительный порядок алгоритмов (дерево)**

* Зайдите в корень.
* Пройдите по левому поддереву, т.е. вызовите предварительный заказ (левое поддерево)
* Пройдите по правому поддереву, т.е. вызовите предварительный заказ (правое поддерево)

**Порядок следования алгоритмов (дерево)**

* Пройдите по левому поддереву, т.е. вызовите Postorder(левое поддерево)
* Пройдите по правому поддереву, т.е. вызовите Postorder(правое поддерево)
* Зайдите в корень.
* Python3

|  |
| --- |
| # Python program to for tree traversals    # A class that represents an individual node in a  # Binary Tree  **class** Node:  **def** \_\_init\_\_(self, key):          self.left **=** None          self.right **=** None          self.val **=** key      # A function to do inorder tree traversal  **def** printInorder(root):    **if** root:            # First recur on left child          printInorder(root.left)            # then print the data of node  **print**(root.val),            # now recur on right child          printInorder(root.right)      # A function to do postorder tree traversal  **def** printPostorder(root):    **if** root:            # First recur on left child          printPostorder(root.left)            # the recur on right child          printPostorder(root.right)            # now print the data of node          print(root.val),      # A function to do preorder tree traversal  **def** printPreorder(root):    **if** root:            # First print the data of node          print(root.val),            # Then recur on left child          printPreorder(root.left)            # Finally recur on right child          printPreorder(root.right)      # Driver code  root **=** Node(1)  root.left **=** Node(2)  root.right **=** Node(3)  root.left.left **=** Node(4)  root.left.right **=** Node(5)  print("Preorder traversal of binary tree is")  printPreorder(root)    **print**("\nInorder traversal of binary tree is")  printInorder(root)    **print**("\nPostorder traversal of binary tree is")  printPostorder(root) |

**Вывод**

Предварительный порядок обхода двоичного дерева

1

2

4

5

3

Порядок обхода двоичного дерева

4

2

5

1

3

Обход двоичного дерева по порядку является

4

5

2

3

1

Временная сложность – O (n)

**Обход по ширине или порядку уровней**

[Обход дерева по порядку уровней - это обход](https://translated.turbopages.org/proxy_u/en-ru.ru.d4b8a886-634c7730-2139cb51-74722d776562/https/www.geeksforgeeks.org/level-order-tree-traversal/) дерева в ширину. Порядок обхода по уровням вышеупомянутого дерева равен 1 2 3 4 5.

Для каждого узла сначала посещается узел, а затем его дочерние узлы помещаются в очередь FIFO. Ниже приведен алгоритм для того же –

* Создайте пустую очередь q
* temp\_node = root /\*начать с root\*/
* Цикл, пока temp\_node не равен NULL
  + выведите temp\_node->данные.
  + Поставьте в очередь дочерние элементы temp\_node (сначала левые, затем правые дочерние элементы) для q
  + Удаление из очереди узла из q
* Python3

|  |
| --- |
| # Python program to print level  # order traversal using Queue    # A node structure  **class** Node:        # A utility function to create a new node  **def** \_\_init\_\_(self ,key):          self.data **=** key          self.left **=** None          self.right **=** None    # Iterative Method to print the  # height of a binary tree  **def** printLevelOrder(root):        # Base Case  **if** root **is** None:  **return**        # Create an empty queue      # for level order traversal      queue **=** []        # Enqueue Root and initialize height      queue.append(root)    **while**(len(queue) > 0):            # Print front of queue and          # remove it from queue          print (queue[0].data)          node **=** queue.pop(0)            # Enqueue left child  **if** node.left **is** **not** None:              queue.append(node.left)            # Enqueue right child  **if** node.right **is** **not** None:              queue.append(node.right)    # Driver Program to test above function  root **=** Node(1)  root.left **=** Node(2)  root.right **=** Node(3)  root.left.left **=** Node(4)  root.left.right **=** Node(5)    **print** ("Level Order Traversal of binary tree is -")  printLevelOrder(root) |

**Вывод**

Обход порядка уровней двоичного дерева является -

1

2

3

4

5

Временная сложность: O (n)

### Другие статьи о двоичном дереве

* [Вставка в двоичное дерево](https://translated.turbopages.org/proxy_u/en-ru.ru.d4b8a886-634c7730-2139cb51-74722d776562/https/www.geeksforgeeks.org/insertion-binary-tree/)
* [Удаление в двоичном дереве](https://translated.turbopages.org/proxy_u/en-ru.ru.d4b8a886-634c7730-2139cb51-74722d776562/https/www.geeksforgeeks.org/deletion-binary-tree/)
* [Упорядоченный обход дерева без рекурсии](https://translated.turbopages.org/proxy_u/en-ru.ru.d4b8a886-634c7730-2139cb51-74722d776562/https/www.geeksforgeeks.org/inorder-tree-traversal-without-recursion/)
* [Упорядоченный обход дерева без рекурсии и без стека!](https://translated.turbopages.org/proxy_u/en-ru.ru.d4b8a886-634c7730-2139cb51-74722d776562/https/www.geeksforgeeks.org/inorder-tree-traversal-without-recursion-and-without-stack/)
* [Печать обхода после заказа из заданных обходов Inorder и Preorder](https://translated.turbopages.org/proxy_u/en-ru.ru.d4b8a886-634c7730-2139cb51-74722d776562/https/www.geeksforgeeks.org/print-postorder-from-given-inorder-and-preorder-traversals/)
* [Найти в обратном порядке обхода по британскому летнему времени от предзаказ обхода](https://translated.turbopages.org/proxy_u/en-ru.ru.d4b8a886-634c7730-2139cb51-74722d776562/https/www.geeksforgeeks.org/find-postorder-traversal-of-bst-from-preorder-traversal/)

[**>>> Подробнее**](https://translated.turbopages.org/proxy_u/en-ru.ru.d4b8a886-634c7730-2139cb51-74722d776562/https/www.geeksforgeeks.org/binary-tree-data-structure/)

## ****Бинарное дерево поиска****

[Двоичное дерево поиска - это структура данных двоичного дерева](https://translated.turbopages.org/proxy_u/en-ru.ru.d4b8a886-634c7730-2139cb51-74722d776562/https/www.geeksforgeeks.org/binary-search-tree-set-1-search-and-insertion/) на основе узлов, которая обладает следующими свойствами:

* Левое поддерево узла содержит только узлы с ключами, меньшими, чем ключ узла.
* Правое поддерево узла содержит только узлы с ключами, превышающими ключ узла.
* Левое и правое поддерево также должны быть двоичным деревом поиска.

Бинарное дерево поиска

Вышеуказанные свойства двоичного дерева поиска обеспечивают упорядочение ключей, чтобы такие операции, как поиск, минимум и максимум, можно было выполнять быстро. Если порядка нет, то нам, возможно, придется сравнивать каждый ключ для поиска заданного ключа.

### Элемент поиска

* Начните с корня.
* Сравните элемент поиска с root, если меньше root, затем выполните рекурсию для left, иначе выполните рекурсию для right.
* Если элемент для поиска найден где угодно, верните true , иначе верните false .
* Python3

|  |
| --- |
| # A utility function to search a given key in BST  **def** search(root,key):        # Base Cases: root is null or key is present at root  **if** root **is** None **or** root.val **==** key:  **return** root        # Key is greater than root's key  **if** root.val < key:  **return** search(root.right,key)        # Key is smaller than root's key  **return** search(root.left,key) |

### Вставка ключа

* Начните с корня.
* Сравните вставляемый элемент с root, если меньше root, затем выполните рекурсию для left, иначе выполните рекурсию для right .
* Дойдя до конца, просто вставьте этот узел слева (если меньше текущего), а затем справа.
* Python3

|  |
| --- |
| # Python program to demonstrate  # insert operation in binary search tree    # A utility class that represents  # an individual node in a BST  **class** Node:  **def** \_\_init\_\_(self, key):          self.left **=** None          self.right **=** None          self.val **=** key    # A utility function to insert  # a new node with the given key  **def** insert(root, key):  **if** root **is** None:  **return** Node(key)  **else**:  **if** root.val **==** key:  **return** root  **elif** root.val < key:              root.right **=** insert(root.right, key)  **else**:              root.left **=** insert(root.left, key)  **return** root    # A utility function to do inorder tree traversal  **def** inorder(root):  **if** root:          inorder(root.left)  **print**(root.val)          inorder(root.right)      # Driver program to test the above functions  # Let us create the following BST  # 50  # /     \  # 30     70  # / \ / \  # 20 40 60 80    r **=** Node(50)  r **=** insert(r, 30)  r **=** insert(r, 20)  r **=** insert(r, 40)  r **=** insert(r, 70)  r **=** insert(r, 60)  r **=** insert(r, 80)    # Print inorder traversal of the BST  inorder(r) |

**Вывод**

20

30

40

50

60

70

80

### Другие статьи о бинарном дереве поиска

* [Бинарное дерево поиска – ключ удаления](https://translated.turbopages.org/proxy_u/en-ru.ru.d4b8a886-634c7730-2139cb51-74722d776562/https/www.geeksforgeeks.org/binary-search-tree-set-2-delete/)
* [Построить по британскому летнему времени с учетом предзаказ обхода | Множество1](https://translated.turbopages.org/proxy_u/en-ru.ru.d4b8a886-634c7730-2139cb51-74722d776562/https/www.geeksforgeeks.org/construct-bst-from-given-preorder-traversa/)
* [Преобразование двоичного дерева в двоичное дерево поиска](https://translated.turbopages.org/proxy_u/en-ru.ru.d4b8a886-634c7730-2139cb51-74722d776562/https/www.geeksforgeeks.org/binary-tree-to-binary-search-tree-conversion/)
* [Найдите узел с минимальным значением в двоичном дереве поиска](https://translated.turbopages.org/proxy_u/en-ru.ru.d4b8a886-634c7730-2139cb51-74722d776562/https/www.geeksforgeeks.org/find-the-minimum-element-in-a-binary-search-tree/)
* [Программа для проверки, является ли двоичное дерево BST или нет по британскому летнему времени](https://translated.turbopages.org/proxy_u/en-ru.ru.d4b8a886-634c7730-2139cb51-74722d776562/https/www.geeksforgeeks.org/a-program-to-check-if-a-binary-tree-is-bst-or-not/)

[**>>> Подробнее**](https://translated.turbopages.org/proxy_u/en-ru.ru.d4b8a886-634c7730-2139cb51-74722d776562/https/www.geeksforgeeks.org/binary-search-tree-data-structure/)

## Графики

[Граф](https://translated.turbopages.org/proxy_u/en-ru.ru.d4b8a886-634c7730-2139cb51-74722d776562/https/www.geeksforgeeks.org/graph-and-its-representations/) - это нелинейная структура данных, состоящая из узлов и ребер. Узлы иногда также называют вершинами, а ребра - это линии или дуги, которые соединяют любые два узла в графе. Более формально граф можно определить как граф, состоящий из конечного набора вершин (или узлов) и набора ребер, которые соединяют пару узлов.



В приведенном выше графике множество вершин V = {0,1,2,3,4} и множество ребер E = {01, 12, 23, 34, 04, 14, 13}. Следующие два являются наиболее часто используемыми представлениями графа.

* Матрица смежности
* Список смежности

### Матрица смежности

Матрица смежности представляет собой 2D-массив размером V x V, где V - количество вершин в графе. Пусть 2D-массив равен adj[][], слот adj[i] [j] = 1 указывает, что существует ребро от вершины i до вершины j. Матрица смежности для неориентированного графа всегда симметрична. Матрица смежности также используется для представления взвешенных графиков. Если adj[i][j] = w, то существует ребро из вершины i в вершину j с весом w.

* Python3

|  |
| --- |
| # A simple representation of graph using Adjacency Matrix  **class** Graph:  **def** \_\_init\_\_(self,numvertex):          self.adjMatrix **=** [[**-**1]**\***numvertex **for** x **in** range(numvertex)]          self.numvertex **=** numvertex          self.vertices **=** {}          self.verticeslist **=**[0]**\***numvertex    **def** set\_vertex(self,vtx,id):  **if** 0<**=**vtx<**=**self.numvertex:              self.vertices[id] **=** vtx              self.verticeslist[vtx] **=** id    **def** set\_edge(self,frm,to,cost**=**0):          frm **=** self.vertices[frm]          to **=** self.vertices[to]          self.adjMatrix[frm][to] **=** cost            # for directed graph do not add this          self.adjMatrix[to][frm] **=** cost    **def** get\_vertex(self):  **return** self.verticeslist    **def** get\_edges(self):          edges**=**[]  **for** i **in** range (self.numvertex):  **for** j **in** range (self.numvertex):  **if** (self.adjMatrix[i][j]!**=-**1):                      edges.append((self.verticeslist[i],self.verticeslist[j],self.adjMatrix[i][j]))  **return** edges    **def** get\_matrix(self):  **return** self.adjMatrix    G **=**Graph(6)  G.set\_vertex(0,'a')  G.set\_vertex(1,'b')  G.set\_vertex(2,'c')  G.set\_vertex(3,'d')  G.set\_vertex(4,'e')  G.set\_vertex(5,'f')  G.set\_edge('a','e',10)  G.set\_edge('a','c',20)  G.set\_edge('c','b',30)  G.set\_edge('b','e',40)  G.set\_edge('e','d',50)  G.set\_edge('f','e',60)    **print**("Vertices of Graph")  print(G.get\_vertex())    print("Edges of Graph")  print(G.get\_edges())    **print**("Adjacency Matrix of Graph")  print(G.get\_matrix()) |

**Вывод**

*Вершины графа*

*[‘a’, ‘b’, ‘c’, ‘d’, ‘e’, ‘f’]*

*Ребра графа*

*[(‘a’, ‘c’, 20), (‘a’, ‘e’, 10), (‘b’, ‘c’, 30), (‘b’, ‘e’, 40), (‘c’, ‘a’, 20), (‘c’, ‘b’, 30), (‘d’, ‘e’, 50), (‘e’, ‘a’, 10), (‘e’, ‘b’, 40), (‘e’, ‘d’, 50), (‘e’, ‘f’, 60), (‘f’, ‘e’, 60)]*

*Матрица смежности графа*

*[[-1, -1, 20, -1, 10, -1], [-1, -1, 30, -1, 40, -1], [20, 30, -1, -1, -1, -1], [-1, -1, -1, -1, 50, -1], [10, 40, -1, 50, -1, 60], [-1, -1, -1, -1, 60, -1]]*

### Список смежности

Используется массив списков. Размер массива равен количеству вершин. Пусть массив будет массивом[]. Массив элементов [i] представляет список вершин, смежных с i-й вершиной. Это представление также можно использовать для представления взвешенного графика. Веса ребер могут быть представлены в виде списков пар. Ниже приведено представление списка смежности приведенного выше графика.

Представление списка смежности графа

* Python3

|  |
| --- |
| # A class to represent the adjacency list of the node  **class** AdjNode:  **def** \_\_init\_\_(self, data):          self.vertex **=** data          self.next **=** None      # A class to represent a graph. A graph  # is the list of the adjacency lists.  # Size of the array will be the no. of the  # vertices "V"  **class** Graph:  **def** \_\_init\_\_(self, vertices):          self.V **=** vertices          self.graph **=** [None] **\*** self.V        # Function to add an edge in an undirected graph  **def** add\_edge(self, src, dest):            # Adding the node to the source node          node **=** AdjNode(dest)          node.next **=** self.graph[src]          self.graph[src] **=** node            # Adding the source node to the destination as          # it is the undirected graph          node **=** AdjNode(src)          node.next **=** self.graph[dest]          self.graph[dest] **=** node        # Function to print the graph  **def** print\_graph(self):  **for** i **in** range(self.V):              print("Adjacency list of vertex {}\n head".format(i), end**=**"")              temp **=** self.graph[i]  **while** temp:                  print(" -> {}".format(temp.vertex), end**=**"")                  temp **=** temp.next              print(" \n")      # Driver program to the above graph class  **if** \_\_name\_\_ **==** "\_\_main\_\_":      V **=** 5      graph **=** Graph(V)      graph.add\_edge(0, 1)      graph.add\_edge(0, 4)      graph.add\_edge(1, 2)      graph.add\_edge(1, 3)      graph.add\_edge(1, 4)      graph.add\_edge(2, 3)      graph.add\_edge(3, 4)        graph.print\_graph() |

**Вывод**

Список смежности вершин 0

head -> 4 -> 1

Список смежности вершины 1

глава -> 4 -> 3 -> 2 -> 0

Список смежности вершин 2

head -> 3 -> 1

Список смежности вершин 3

head -> 4 -> 2 -> 1

Список смежности вершин 4

head -> 3 -> 1 -> 0

## Обход графа

### Поиск по ширине или BFS

[Обход графа в ширину](https://translated.turbopages.org/proxy_u/en-ru.ru.d4b8a886-634c7730-2139cb51-74722d776562/https/www.geeksforgeeks.org/breadth-first-search-or-bfs-for-a-graph/) аналогичен обходу дерева в ширину. Единственная загвоздка здесь в том, что, в отличие от деревьев, графики могут содержать циклы, поэтому мы можем снова прийти к тому же узлу. Чтобы избежать обработки узла более одного раза, мы используем логический массив посещений. Для простоты предполагается, что все вершины достижимы из начальной вершины.

Например, на следующем графике мы начинаем обход с вершины 2. Когда мы приходим к вершине 0, мы ищем все смежные с ней вершины. 2 также является смежной вершиной 0. Если мы не будем отмечать посещенные вершины, то 2 будут обработаны снова, и это станет непрерывным процессом. Обход следующего графика в ширину равен 2, 0, 3, 1.



* Python3

|  |
| --- |
| # Python3 Program to print BFS traversal  # from a given source vertex. BFS(int s)  # traverses vertices reachable from s.  **from** collections **import** defaultdict    # This class represents a directed graph  # using adjacency list representation  **class** Graph:        # Constructor  **def** \_\_init\_\_(self):            # default dictionary to store graph          self.graph **=** defaultdict(list)        # function to add an edge to graph  **def** addEdge(self,u,v):          self.graph[u].append(v)        # Function to print a BFS of graph  **def** BFS(self, s):            # Mark all the vertices as not visited          visited **=** [False] **\*** (max(self.graph) **+** 1)            # Create a queue for BFS          queue **=** []            # Mark the source node as          # visited and enqueue it          queue.append(s)          visited[s] **=** True    **while** queue:                # Dequeue a vertex from              # queue and print it              s **=** queue.pop(0)  **print** (s, end **=** " ")                # Get all adjacent vertices of the              # dequeued vertex s. If a adjacent              # has not been visited, then mark it              # visited and enqueue it  **for** i **in** self.graph[s]:  **if** visited[i] **==** False:                      queue.append(i)                      visited[i] **=** True    # Driver code    # Create a graph given in  # the above diagram  g **=** Graph()  g.addEdge(0, 1)  g.addEdge(0, 2)  g.addEdge(1, 2)  g.addEdge(2, 0)  g.addEdge(2, 3)  g.addEdge(3, 3)    **print** ("Following is Breadth First Traversal"                  " (starting from vertex 2)")  g.BFS(2) |

**Вывод**

Ниже приведен первый обход по ширине (начиная с вершины 2)

2 0 3 1

Временная сложность: O (V + E), где V - количество вершин в графе, а E - количество ребер в графе.

### Поиск в глубину или DFS

[Первый обход графа по глубине](https://translated.turbopages.org/proxy_u/en-ru.ru.d4b8a886-634c7730-2139cb51-74722d776562/https/www.geeksforgeeks.org/depth-first-search-or-dfs-for-a-graph/) аналогичен первому обходу дерева по глубине. Единственная загвоздка здесь в том, что, в отличие от деревьев, графики могут содержать циклы, узел может быть посещен дважды. Чтобы избежать обработки узла более одного раза, используйте логический массив посещений.

**Алгоритм:**

* Создайте рекурсивную функцию, которая принимает индекс узла и посещенный массив.
* Отметьте текущий узел как посещенный и распечатайте узел.
* Пройдите по всем смежным и немаркированным узлам и вызовите рекурсивную функцию с индексом соседнего узла.
* Python3

|  |
| --- |
| # Python3 program to print DFS traversal  # from a given  graph  **from** collections **import** defaultdict    # This class represents a directed graph using  # adjacency list representation  **class** Graph:        # Constructor  **def** \_\_init\_\_(self):            # default dictionary to store graph          self.graph **=** defaultdict(list)        # function to add an edge to graph  **def** addEdge(self, u, v):          self.graph[u].append(v)        # A function used by DFS  **def** DFSUtil(self, v, visited):            # Mark the current node as visited          # and print it          visited.add(v)          print(v, end**=**' ')            # Recur for all the vertices          # adjacent to this vertex  **for** neighbour **in** self.graph[v]:  **if** neighbour **not** **in** visited:                  self.DFSUtil(neighbour, visited)        # The function to do DFS traversal. It uses      # recursive DFSUtil()  **def** DFS(self, v):            # Create a set to store visited vertices          visited **=** set()            # Call the recursive helper function          # to print DFS traversal          self.DFSUtil(v, visited)    # Driver code    # Create a graph given  # in the above diagram  g **=** Graph()  g.addEdge(0, 1)  g.addEdge(0, 2)  g.addEdge(1, 2)  g.addEdge(2, 0)  g.addEdge(2, 3)  g.addEdge(3, 3)    **print**("Following is DFS from (starting from vertex 2)")  g.DFS(2) |

**Вывод**

Ниже приведен DFS из (начиная с вершины 2)

2 0 1 3

### Другие статьи о графе

* [Представления графов с использованием set и hash](https://translated.turbopages.org/proxy_u/en-ru.ru.d4b8a886-634c7730-2139cb51-74722d776562/https/www.geeksforgeeks.org/graph-representations-using-set-hash/)
* [Найти материнскую вершину в графе](https://translated.turbopages.org/proxy_u/en-ru.ru.d4b8a886-634c7730-2139cb51-74722d776562/https/www.geeksforgeeks.org/find-a-mother-vertex-in-a-graph/)
* [Итеративный поиск в глубину](https://translated.turbopages.org/proxy_u/en-ru.ru.d4b8a886-634c7730-2139cb51-74722d776562/https/www.geeksforgeeks.org/iterative-depth-first-traversal/)
* [Подсчитайте количество узлов на заданном уровне в дереве с помощью BFS](https://translated.turbopages.org/proxy_u/en-ru.ru.d4b8a886-634c7730-2139cb51-74722d776562/https/www.geeksforgeeks.org/count-number-nodes-given-level-using-bfs/)
* [Подсчитайте все возможные пути между двумя вершинами](https://translated.turbopages.org/proxy_u/en-ru.ru.d4b8a886-634c7730-2139cb51-74722d776562/https/www.geeksforgeeks.org/count-possible-paths-two-vertices/)

[**>>> Подробнее**](https://translated.turbopages.org/proxy_u/en-ru.ru.d4b8a886-634c7730-2139cb51-74722d776562/https/www.geeksforgeeks.org/graph-data-structure-and-algorithms/)

## Рекурсия

Процесс, в котором функция вызывает саму себя прямо или косвенно, называется рекурсией, а соответствующая функция называется [рекурсивной функцией](https://translated.turbopages.org/proxy_u/en-ru.ru.d4b8a886-634c7730-2139cb51-74722d776562/https/www.geeksforgeeks.org/recursion-in-python/). Используя рекурсивные алгоритмы, некоторые проблемы могут быть решены довольно легко. Примерами таких проблем являются Ханойские башни (TOH), обходы дерева Inorder / Preorder / Postorder, DFS графа и т. Д.

### Каково базовое условие в рекурсии?

В рекурсивной программе предоставляется решение базового варианта, а решение большей проблемы выражается в терминах меньших проблем.

факт определения (n):

# базовый вариант

, если (n < = 1)

возвращает 1

else

возвращает n\*факт(n-1)

В приведенном выше примере определен базовый вариант для n < = 1, и большее значение числа может быть решено путем преобразования в меньшее, пока не будет достигнут базовый вариант.

### Как память выделяется для различных вызовов функций при рекурсии?

Когда какая-либо функция вызывается из main(), ей выделяется память в стеке. Рекурсивная функция вызывает саму себя, память для вызываемой функции выделяется поверх памяти, выделенной для вызывающей функции, и для каждого вызова функции создается отдельная копия локальных переменных. Когда достигается базовый вариант, функция возвращает свое значение функции, с помощью которой она вызывается, память освобождается, и процесс продолжается.

Давайте рассмотрим пример того, как работает рекурсия, используя простую функцию.

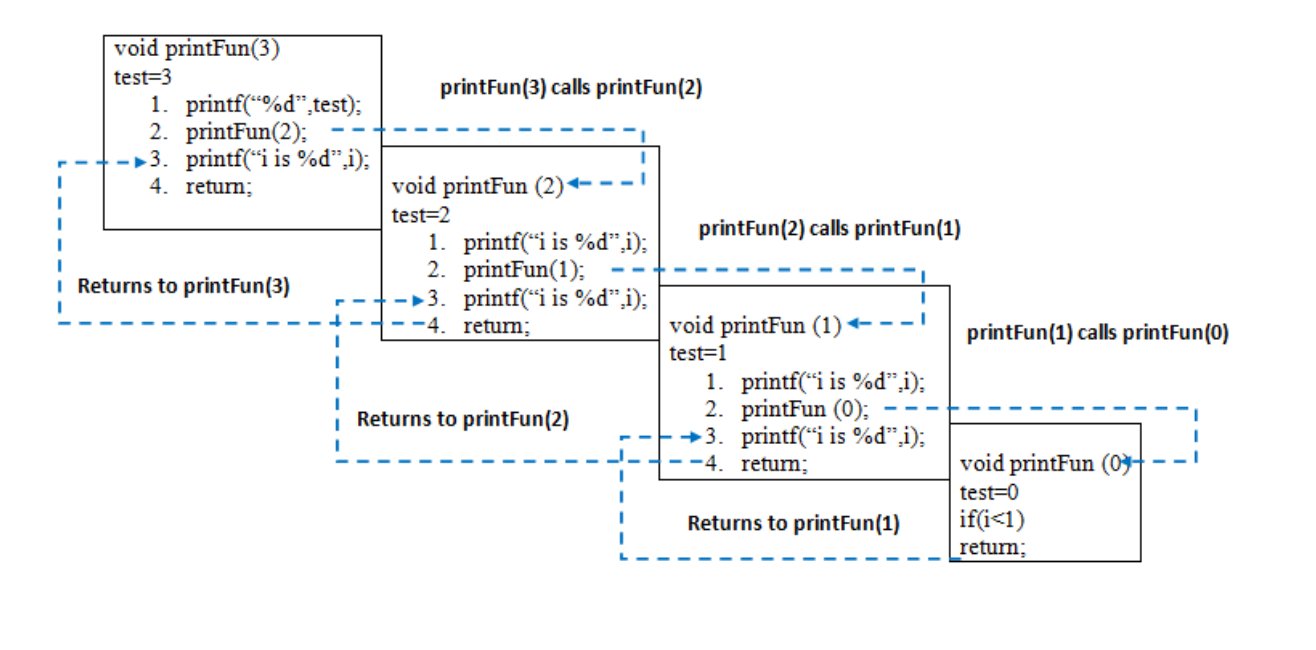
* Python3

|  |
| --- |
| # A Python 3 program to  # demonstrate working of  # recursion    **def** printFun(test):    **if** (test < 1):  **return**  **else**:            print(test, end**=**" ")          printFun(test**-**1)  # statement 2  **print**(test, end**=**" ")  **return**    # Driver Code  test **=** 3  printFun(test) |

**Вывод**

3 2 1 1 2 3

Стек памяти показан на диаграмме ниже.



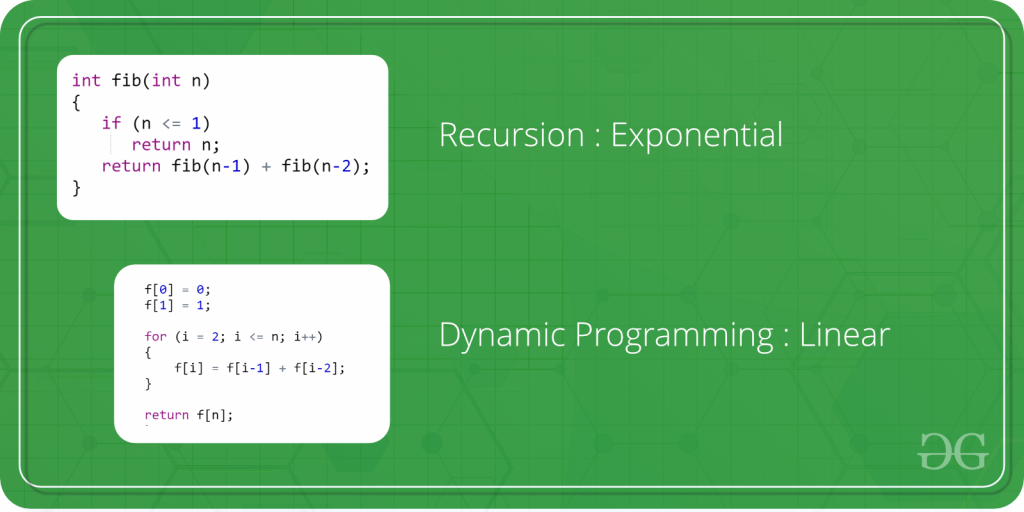
### Другие статьи о рекурсии

* [Рекурсия](https://translated.turbopages.org/proxy_u/en-ru.ru.d4b8a886-634c7730-2139cb51-74722d776562/https/www.geeksforgeeks.org/recursion/)
* [Рекурсия в Python](https://translated.turbopages.org/proxy_u/en-ru.ru.d4b8a886-634c7730-2139cb51-74722d776562/https/www.geeksforgeeks.org/recursion-in-python/)
* [Практические вопросы для рекурсии | Множество1](https://translated.turbopages.org/proxy_u/en-ru.ru.d4b8a886-634c7730-2139cb51-74722d776562/https/www.geeksforgeeks.org/practice-questions-for-recursion/)
* [Практические вопросы для рекурсии | Множество2](https://translated.turbopages.org/proxy_u/en-ru.ru.d4b8a886-634c7730-2139cb51-74722d776562/https/www.geeksforgeeks.org/practice-questions-for-recursion-set-2/)
* [Практические вопросы для рекурсии | Множество3](https://translated.turbopages.org/proxy_u/en-ru.ru.d4b8a886-634c7730-2139cb51-74722d776562/https/www.geeksforgeeks.org/practice-questions-for-recursion-set-3/)
* [Практические вопросы для рекурсии | Множество4](https://translated.turbopages.org/proxy_u/en-ru.ru.d4b8a886-634c7730-2139cb51-74722d776562/https/www.geeksforgeeks.org/practice-questions-for-recursion-set-4/)
* [Практические вопросы для рекурсии | Множество5](https://translated.turbopages.org/proxy_u/en-ru.ru.d4b8a886-634c7730-2139cb51-74722d776562/https/www.geeksforgeeks.org/practice-questions-for-recursion-set-5/)
* [Практические вопросы для рекурсии | Множество6](https://translated.turbopages.org/proxy_u/en-ru.ru.d4b8a886-634c7730-2139cb51-74722d776562/https/www.geeksforgeeks.org/practice-questions-for-recursion-set-6/)
* [Практические вопросы для рекурсии | Множество7](https://translated.turbopages.org/proxy_u/en-ru.ru.d4b8a886-634c7730-2139cb51-74722d776562/https/www.geeksforgeeks.org/practice-questions-for-recursion-set-7/)

[**>>> Подробнее**](https://translated.turbopages.org/proxy_u/en-ru.ru.d4b8a886-634c7730-2139cb51-74722d776562/https/www.geeksforgeeks.org/tag/recursion/)

## ****Динамическое программирование****

[Динамическое программирование](https://translated.turbopages.org/proxy_u/en-ru.ru.d4b8a886-634c7730-2139cb51-74722d776562/https/www.geeksforgeeks.org/dynamic-programming/) - это в основном оптимизация по сравнению с простой рекурсией. Везде, где мы видим рекурсивное решение с повторяющимися вызовами для одних и тех же входных данных, мы можем оптимизировать его с помощью динамического программирования. Идея состоит в том, чтобы просто сохранить результаты подзадач, чтобы нам не приходилось повторно вычислять их при необходимости позже. Эта простая оптимизация сокращает временные сложности с экспоненциальных до полиномиальных. Например, если мы напишем простое рекурсивное решение для чисел Фибоначчи, мы получим экспоненциальную временную сложность, а если мы оптимизируем его, сохраняя решения подзадач, временная сложность уменьшается до линейной.



### Табулирование против запоминания

Существует два разных способа хранения значений, чтобы значения подзадачи можно было использовать повторно. Здесь мы обсудим две модели решения проблемы динамического программирования (DP):

* **Табулирование:** снизу вверх
* **Запоминание:** сверху вниз

### Табулирование

Как следует из самого названия, начиная с самого низа и накапливая ответы сверху. Давайте обсудим с точки зрения перехода состояния.

Давайте опишем состояние для нашей задачи DP как dp [x] с dp [0] в качестве базового состояния и dp [n] в качестве конечного состояния. Итак, нам нужно найти значение состояния назначения, то есть dp [n] .

Если мы начнем наш переход из нашего базового состояния, то есть dp [0], и будем следовать нашему соотношению перехода состояния, чтобы достичь нашего целевого состояния dp [n], мы называем это подходом снизу вверх, поскольку совершенно ясно, что мы начали наш переход из нижнего базового состояния и достигли самого верхнего желаемогосостояние.

Теперь, почему мы называем это методом табуляции?

Чтобы знать это, давайте сначала напишем некоторый код для вычисления факториала числа, используя подход снизу вверх. Опять же, в качестве нашей общей процедуры для решения DP мы сначала определяем состояние. В этом случае мы определяем состояние как dp[x], где dp[x] означает нахождение факториала x.

Теперь совершенно очевидно, что dp[x + 1] = dp [x] \* (x + 1)

# Табличная версия для нахождения факториала x.

dp = [0]\*MAXN

# базовый вариант

dp[0] = 1;

для i в диапазоне (n + 1):

dp[i] = dp[i-1] \* i

### Запоминание

Еще раз, давайте опишем это в терминах перехода состояния. Если нам нужно найти значение для некоторого состояния, скажем, dp [n], и вместо того, чтобы начинать с базового состояния, то есть dp [0], мы запрашиваем наш ответ у состояний, которые могут достичь состояния назначения dp [n], следуя соотношению перехода состояния, тогда это верхниймода на DP.

Здесь мы начинаем наше путешествие с самого верхнего состояния назначения и вычисляем его ответ, принимая во внимание значения состояний, которые могут достигать состояния назначения, пока мы не достигнем самого нижнего базового состояния.

Еще раз, давайте напишем код для факториальной задачи сверху вниз

# Сохраненная версия для поиска факториала x.

# Для ускорения мы сохраняем значения

Количество вычисляемых состояний

# инициализируется значением -1

dp[0]\*MAXN

# верните факт x!

решаемая задача (x):

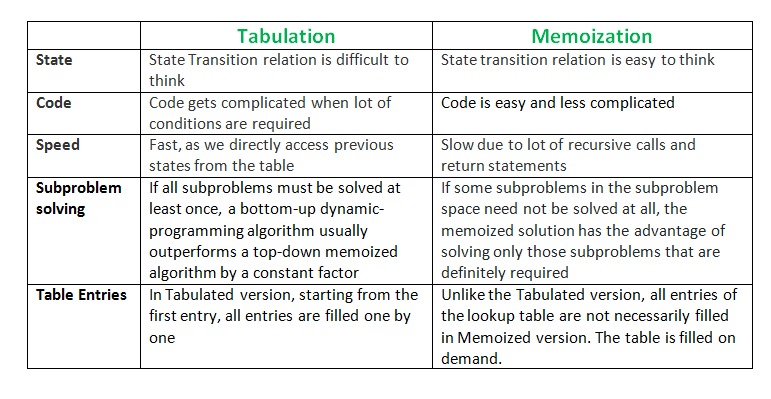
если (x==0)

верните 1

, если (dp[x]!=-1)

возвращает dp[x]

возврат (dp[x] = x \* решение (x-1))



### Другие статьи по динамическому программированию

* [Оптимальное свойство подструктуры](https://translated.turbopages.org/proxy_u/en-ru.ru.d4b8a886-634c7730-2139cb51-74722d776562/https/www.geeksforgeeks.org/dynamic-programming-set-2-optimal-substructure-property/)
* [Свойство перекрывающихся подзадач](https://translated.turbopages.org/proxy_u/en-ru.ru.d4b8a886-634c7730-2139cb51-74722d776562/https/www.geeksforgeeks.org/dynamic-programming-set-1/)
* [Числа Фибоначчи](https://translated.turbopages.org/proxy_u/en-ru.ru.d4b8a886-634c7730-2139cb51-74722d776562/https/www.geeksforgeeks.org/program-for-nth-fibonacci-number/)
* [Подмножество с суммой, кратной m](https://translated.turbopages.org/proxy_u/en-ru.ru.d4b8a886-634c7730-2139cb51-74722d776562/https/www.geeksforgeeks.org/subset-sum-divisible-m/)
* [Подпоследовательность увеличения максимальной суммы](https://translated.turbopages.org/proxy_u/en-ru.ru.d4b8a886-634c7730-2139cb51-74722d776562/https/www.geeksforgeeks.org/dynamic-programming-set-14-maximum-sum-increasing-subsequence/)
* [Самая длинная общая подстрока](https://translated.turbopages.org/proxy_u/en-ru.ru.d4b8a886-634c7730-2139cb51-74722d776562/https/www.geeksforgeeks.org/longest-common-substring/)

[**>>> Подробнее**](https://translated.turbopages.org/proxy_u/en-ru.ru.d4b8a886-634c7730-2139cb51-74722d776562/https/www.geeksforgeeks.org/dynamic-programming/)

## Алгоритмы поиска

### Линейный поиск

* Начните с крайнего левого элемента arr[] и один за другим сравните x с каждым элементом arr[]
* Если x совпадает с элементом, верните индекс.
* Если x не совпадает ни с одним из элементов, верните значение -1.



* Python3

|  |
| --- |
| # Python3 code to linearly search x in arr[].  # If x is present then return its location,  # otherwise return -1      **def** search(arr, n, x):    **for** i **in** range(0, n):  **if** (arr[i] **==** x):  **return** i  **return** **-**1      # Driver Code  arr **=** [2, 3, 4, 10, 40]  x **=** 10  n **=** len(arr)    # Function call  result **=** search(arr, n, x)  **if**(result **==** **-**1):      print("Element is not present in array")  **else**:      print("Element is present at index", result) |

**Вывод**

Элемент присутствует в индексе 3

Временная сложность вышеупомянутого алгоритма равна O (n).

Для получения дополнительной информации обратитесь к [линейному поиску](https://translated.turbopages.org/proxy_u/en-ru.ru.d4b8a886-634c7730-2139cb51-74722d776562/https/www.geeksforgeeks.org/linear-search/).

### Бинарный поиск

Выполните поиск в отсортированном массиве, многократно разделив интервал поиска пополам. Начните с интервала, охватывающего весь массив. Если значение ключа поиска меньше, чем элемент в середине интервала, сузьте интервал до нижней половины. В противном случае сузьте его до верхней половины. Многократно проверяйте, пока не будет найдено значение или интервал не станет пустым.



* Python3

|  |
| --- |
| # Python3 Program for recursive binary search.    # Returns index of x in arr if present, else -1  **def** binarySearch (arr, l, r, x):        # Check base case  **if** r >**=** l:            mid **=** l **+** (r **-** l) **//** 2            # If element is present at the middle itself  **if** arr[mid] **==** x:  **return** mid            # If element is smaller than mid, then it          # can only be present in left subarray  **elif** arr[mid] > x:  **return** binarySearch(arr, l, mid**-**1, x)            # Else the element can only be present          # in right subarray  **else**:  **return** binarySearch(arr, mid **+** 1, r, x)    **else**:          # Element is not present in the array  **return** **-**1    # Driver Code  arr **=** [ 2, 3, 4, 10, 40 ]  x **=** 10    # Function call  result **=** binarySearch(arr, 0, len(arr)**-**1, x)    **if** result !**=** **-**1:      print ("Element is present at index % d" **%** result)  **else**:      print ("Element is not present in array") |

**Вывод**

Элемент присутствует в индексе 3

Временная сложность вышеупомянутого алгоритма равна O (log (n)).

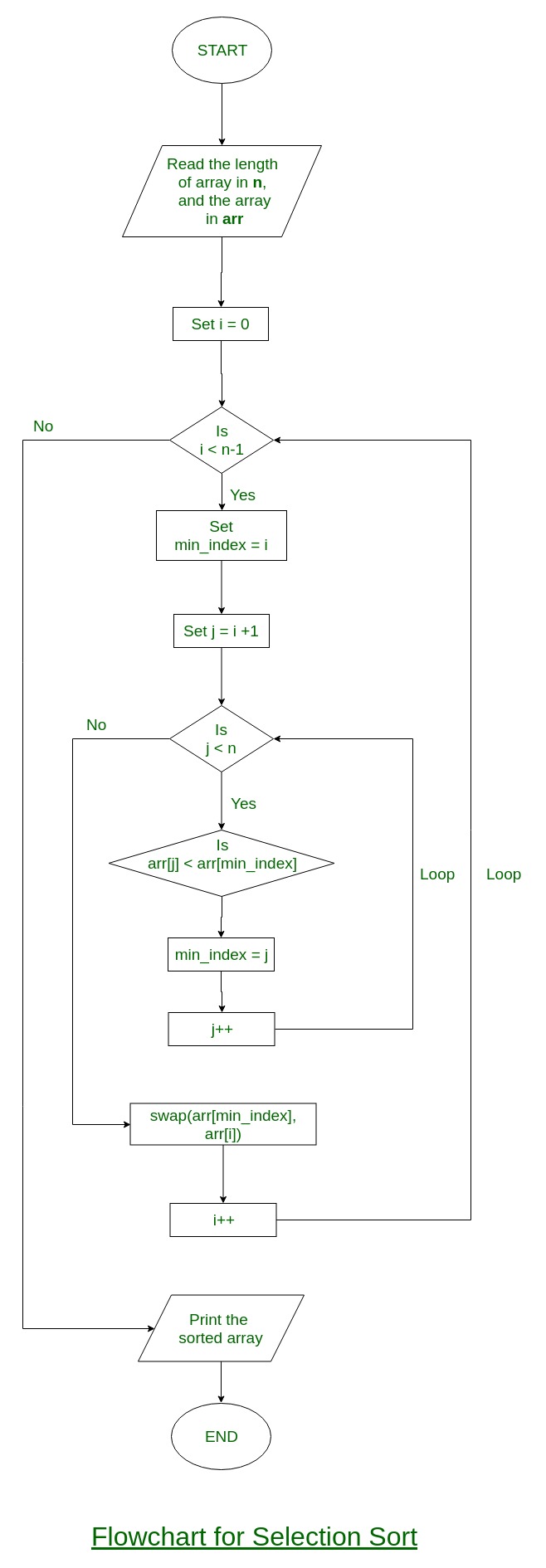
Для получения дополнительной информации обратитесь к [бинарному поиску](https://translated.turbopages.org/proxy_u/en-ru.ru.d4b8a886-634c7730-2139cb51-74722d776562/https/www.geeksforgeeks.org/binary-search/).

## Алгоритмы сортировки

### Сортировка выборки

Алгоритм сортировки [выборки сортирует](https://translated.turbopages.org/proxy_u/en-ru.ru.d4b8a886-634c7730-2139cb51-74722d776562/https/www.geeksforgeeks.org/selection-sort/) массив, повторно находя минимальный элемент (с учетом порядка возрастания) из несортированной части и помещая его в начало. На каждой итерации сортировки выбора выбирается минимальный элемент (с учетом порядка возрастания) из несортированного подмассива и перемещается в сортированный подмассив.

### Блок-схема сортировки выбора:



* Python3

|  |
| --- |
| # Python program for implementation of Selection  # Sort  **import** sys      A **=** [64, 25, 12, 22, 11]    # Traverse through all array elements  **for** i **in** range(len(A)):        # Find the minimum element in remaining      # unsorted array      min\_idx **=** i  **for** j **in** range(i**+**1, len(A)):  **if** A[min\_idx] > A[j]:              min\_idx **=** j        # Swap the found minimum element with      # the first element      A[i], A[min\_idx] **=** A[min\_idx], A[i]    # Driver code to test above  **print** ("Sorted array")  **for** i **in** range(len(A)):      print("%d" **%**A[i]), |

**Вывод**

Сортированный массив

11

12

22

25

64

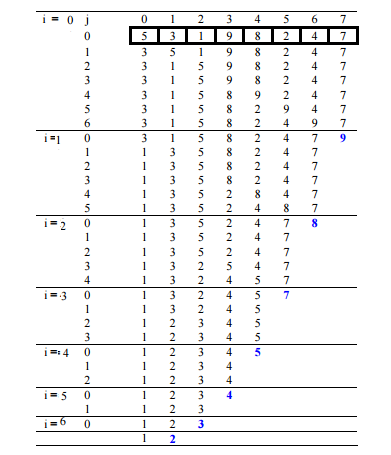
**Временная сложность:** O (n 2), поскольку существует два вложенных цикла.

**Вспомогательное пространство:** O(1)

### Пузырьковая сортировка

[Пузырьковая сортировка](https://translated.turbopages.org/proxy_u/en-ru.ru.d4b8a886-634c7730-2139cb51-74722d776562/https/www.geeksforgeeks.org/bubble-sort/) - это простейший алгоритм сортировки, который работает путем многократной замены соседних элементов, если они расположены в неправильном порядке.

**Иллюстрация :**



* Python3

|  |
| --- |
| # Python program for implementation of Bubble Sort    **def** bubbleSort(arr):      n **=** len(arr)        # Traverse through all array elements  **for** i **in** range(n):            # Last i elements are already in place  **for** j **in** range(0, n**-**i**-**1):                # traverse the array from 0 to n-i-1              # Swap if the element found is greater              # than the next element  **if** arr[j] > arr[j**+**1] :                  arr[j], arr[j**+**1] **=** arr[j**+**1], arr[j]    # Driver code to test above  arr **=** [64, 34, 25, 12, 22, 11, 90]    bubbleSort(arr)    print ("Sorted array is:")  **for** i **in** range(len(arr)):      print ("%d" **%**arr[i]), |

**Вывод**

Отсортированный массив - это:

11

12

22

25

34

64

90

**Временная сложность:** O (n 2)

### Сортировка по вставке

Для сортировки массива размером n в порядке возрастания с помощью [сортировки по вставке](https://translated.turbopages.org/proxy_u/en-ru.ru.d4b8a886-634c7730-2139cb51-74722d776562/https/www.geeksforgeeks.org/insertion-sort/):

* Выполните итерацию от arr[1] до arr[n] по массиву.
* Сравните текущий элемент (ключ) с его предшественником.
* Если ключевой элемент меньше, чем его предшественник, сравните его с предыдущими элементами. Переместите большие элементы на одну позицию вверх, чтобы освободить место для заменяемого элемента.

**Иллюстрация:**



* Python3

|  |
| --- |
| # Python program for implementation of Insertion Sort    # Function to do insertion sort  **def** insertionSort(arr):        # Traverse through 1 to len(arr)  **for** i **in** range(1, len(arr)):            key **=** arr[i]            # Move elements of arr[0..i-1], that are          # greater than key, to one position ahead          # of their current position          j **=** i**-**1  **while** j >**=** 0 **and** key < arr[j] :                  arr[j **+** 1] **=** arr[j]                  j **-=** 1          arr[j **+** 1] **=** key      # Driver code to test above  arr **=** [12, 11, 13, 5, 6]  insertionSort(arr)  **for** i **in** range(len(arr)):      print ("% d" **%** arr[i]) |

**Вывод**

5

6

11

12

13

**Временная сложность:** O (n 2))

### Сортировка слиянием

Как и быстрая сортировка, [сортировка слиянием](https://translated.turbopages.org/proxy_u/en-ru.ru.d4b8a886-634c7730-2139cb51-74722d776562/https/www.geeksforgeeks.org/merge-sort/) - это алгоритм "Разделяй и властвуй". Он делит входной массив на две половины, вызывает себя для двух половин, а затем объединяет две отсортированные половины. Функция merge() используется для объединения двух половин. Слияние (arr, l, m, r) является ключевым процессом, который предполагает, что arr[l..m] и arr [m + 1..r] отсортированы и объединяет два отсортированных подмассива в один.

Сортировка слиянием(arr[], l, r)

Если r > l

1. Найдите среднюю точку, чтобы разделить массив на две половины:

средний m = l + (r-l)/2

2. Вызовите сортировку слиянием для первой половины:

Вызовите сортировку слиянием(arr, l, m)

3. Вызовите сортировку слиянием для второй половины:

Вызовите сортировку слиянием(arr, m+1, r)

4. Объедините две половины, отсортированные на шаге 2 и 3:

Вызовите слияние (arr, l, m, r)



* Python3

|  |
| --- |
| # Python program for implementation of MergeSort  **def** mergeSort(arr):  **if** len(arr) > 1:            # Finding the mid of the array          mid **=** len(arr)**//**2            # Dividing the array elements          L **=** arr[:mid]            # into 2 halves          R **=** arr[mid:]            # Sorting the first half          mergeSort(L)            # Sorting the second half          mergeSort(R)            i **=** j **=** k **=** 0            # Copy data to temp arrays L[] and R[]  **while** i < len(L) **and** j < len(R):  **if** L[i] < R[j]:                  arr[k] **=** L[i]                  i **+=** 1  **else**:                  arr[k] **=** R[j]                  j **+=** 1              k **+=** 1            # Checking if any element was left  **while** i < len(L):              arr[k] **=** L[i]              i **+=** 1              k **+=** 1    **while** j < len(R):              arr[k] **=** R[j]              j **+=** 1              k **+=** 1    # Code to print the list      **def** printList(arr):  **for** i **in** range(len(arr)):          print(arr[i], end**=**" ")      print()      # Driver Code  **if** \_\_name\_\_ **==** '\_\_main\_\_':      arr **=** [12, 11, 13, 5, 6, 7]      print("Given array is", end**=**"\n")      printList(arr)      mergeSort(arr)      print("Sorted array is: ", end**=**"\n")      printList(arr) |

**Вывод**

Данный массив равен

12 11 13 5 6 7

Отсортированный массив - это:

5 6 7 11 12 13

**Временная сложность:** O (n (logn))

### Быстрая сортировка

Как и сортировка слиянием, быстрая [сортировка](https://translated.turbopages.org/proxy_u/en-ru.ru.d4b8a886-634c7730-2139cb51-74722d776562/www.geeksforgeeks.org/quick-sort/) - это алгоритм "Разделяй и властвуй". Он выбирает элемент в качестве pivot и разбивает данный массив вокруг выбранного pivot. Существует много разных версий quickSort, которые по-разному выбирают pivot.

Всегда выбирайте первый элемент в качестве pivot.

* Всегда выбирайте последний элемент в качестве pivot (реализовано ниже)
* Выберите случайный элемент в качестве pivot.
* Выберите медиану в качестве точки опоры.

Ключевым процессом в quickSort является partition() . Цель разделов - получить массив и элемент x массива в качестве pivot, поместить x в правильное положение в отсортированном массиве и поместить все меньшие элементы (меньше x) перед x и поместить все большие элементы (больше x) после x. Все это должно быть сделано за линейное время.

/\* низкий -> Начальный индекс, высокий -> Конечный индекс \*/

Быстрая сортировка(arr[], low, high)

{

если (низкий < высокий)

{

/\* pi - это индекс разделения, arr[pi] теперь

находится в нужном месте \*/

pi = раздел (arr, low, high);

Быстрая сортировка (arr, low, pi - 1); // Перед

быстрой сортировкой pi(arr, pi + 1, high); // После pi

}

}



**Алгоритм разделения**

Может быть много способов сделать разделение, следующий псевдокод использует метод, приведенный в книге CLRS. Логика проста, мы начинаем с самого левого элемента и отслеживаем индекс меньших (или равных) элементов как i . Во время обхода, если мы находим меньший элемент, мы меняем текущий элемент на arr [i] . В противном случае мы игнорируем текущий элемент.

/\* низкий -> Начальный индекс, высокий -> Конечный индекс \*/

Быстрая сортировка(arr[], low, high)

{

если (низкий < высокий)

{

/\* pi - это индекс разделения, arr[pi] теперь

находится в нужном месте \*/

pi = раздел (arr, low, high);

Быстрая сортировка (arr, low, pi - 1); // Перед

быстрой сортировкой pi(arr, pi + 1, high); // После pi

}

}

* Python3

|  |
| --- |
| # Python3 implementation of QuickSort    # This Function handles sorting part of quick sort  # start and end points to first and last element of  # an array respectively  **def** partition(start, end, array):        # Initializing pivot's index to start      pivot\_index **=** start      pivot **=** array[pivot\_index]        # This loop runs till start pointer crosses      # end pointer, and when it does we swap the      # pivot with element on end pointer  **while** start < end:            # Increment the start pointer till it finds an          # element greater than pivot  **while** start < len(array) **and** array[start] <**=** pivot:              start **+=** 1            # Decrement the end pointer till it finds an          # element less than pivot  **while** array[end] > pivot:              end **-=** 1            # If start and end have not crossed each other,          # swap the numbers on start and end  **if**(start < end):              array[start], array[end] **=** array[end], array[start]        # Swap pivot element with element on end pointer.      # This puts pivot on its correct sorted place.      array[end], array[pivot\_index] **=** array[pivot\_index], array[end]        # Returning end pointer to divide the array into 2  **return** end    # The main function that implements QuickSort  **def** quick\_sort(start, end, array):    **if** (start < end):            # p is partitioning index, array[p]          # is at right place          p **=** partition(start, end, array)            # Sort elements before partition          # and after partition          quick\_sort(start, p **-** 1, array)          quick\_sort(p **+** 1, end, array)    # Driver code  array **=** [ 10, 7, 8, 9, 1, 5 ]  quick\_sort(0, len(array) **-** 1, array)    print(f'Sorted array: {array}') |

**Вывод**

Сортированный массив: [1, 5, 7, 8, 9, 10]

**Временная сложность:** O (n (logn))

### ShellSort

[ShellSort](https://translated.turbopages.org/proxy_u/en-ru.ru.d4b8a886-634c7730-2139cb51-74722d776562/https/www.geeksforgeeks.org/shellsort/) - это, в основном, разновидность сортировки вставкой. При сортировке вставкой мы перемещаем элементы только на одну позицию вперед. Когда элемент должен быть перемещен далеко вперед, задействовано много движений. Идея ShellSort заключается в том, чтобы разрешить обмен удаленными элементами. В ShellSort мы делаем массив отсортированным по h для большого значения h. Мы продолжаем уменьшать значение h, пока оно не станет равным 1. Говорят, что массив отсортирован по h, если отсортированы все подсписки каждого h-го элемента.

* Python3

|  |
| --- |
| # Python3 program for implementation of Shell Sort    **def** shellSort(arr):      gap **=** len(arr) **//** 2 # initialize the gap    **while** gap > 0:          i **=** 0          j **=** gap            # check the array in from left to right          # till the last possible index of j  **while** j < len(arr):    **if** arr[i] >arr[j]:                  arr[i],arr[j] **=** arr[j],arr[i]                i **+=** 1              j **+=** 1                # now, we look back from ith index to the left              # we swap the values which are not in the right order.              k **=** i  **while** k **-** gap > **-**1:    **if** arr[k **-** gap] > arr[k]:                      arr[k**-**gap],arr[k] **=** arr[k],arr[k**-**gap]                  k **-=** 1            gap **//=** 2      # driver to check the code  arr2 **=** [12, 34, 54, 2, 3]  print("input array:",arr2)    shellSort(arr2)  print("sorted array",arr2) |

**Вывод**

входной массив: [12, 34, 54, 2, 3]

сортированный массив [2, 3, 12, 34, 54]

**Временная сложность:** O (n 2).