Готовые реализации структур данных: очередь, стек, бинарное дерево

О чём вы узнаете

В данном модуле вы:

1. Узнаете о новых библиотеках, которые помогут работать с нестандартными структурами данных.
2. Глубже разберётесь в структурах данных (стек, очередь, связанный список, бинарное дерево).
3. Научитесь делать выбор из множества разных структур данных.

Модуль collections

Большинство структур данных имеет устоявшуюся реализацию, поэтому не нужно создавать их каждый раз вручную и придумывать свои модули.

Если говорить о модулях с готовой реализацией дополнительных структур данных, то первым делом стоит упомянуть модуль collections.

Модуль **collections**

Документация

Установка: модуль collections является встроенным модулем Python, поэтому дополнительная установка не требуется.

Подключение: импортируйте его с помощью import collections.

Наиболее популярные и полезные структуры данных модуля

1. deque

Представляет собой двустороннюю очередь. Позволяет добавлять и удалять элементы как в начале, так и в конце очереди.

Пример использования:

```
from collections import deque
queue = deque()
queue.append(1) # Добавление элемента в конец очереди
queue.append(2)
queue.appendleft(3) # Добавление элемента в начало очереди

print(queue) # Вывод: deque([3, 1, 2])

item = queue.popleft() # Удаление и получение элемента из начала очереди
print(item) # Вывод: 3
```

2. counter

Используется для подсчёта объектов, с его помощью удобно проводить операции подсчёта и анализа данных.

Пример использования:

```
from collections import Counter

data = [1, 2, 3, 1, 2, 1, 3, 4, 5, 4, 2, 1]
counter = Counter(data)

print(counter) # Вывод: Counter({1: 4, 2: 3, 3: 2, 4: 2, 5: 1})

print(counter[1]) # Вывод: 4 (количество вхождений элемента 1)

most_common = counter.most_common(2)

print(most_common) # Вывод: [(1, 4), (2, 3)] (наиболее часто встречающиеся элементы)
```

3. namedtuple

Позволяет создавать именованные кортежи, которые являются неизменяемыми коллекциями элементов с доступом по именам.

Пример использования:

```
from collections import namedtuple

Point = namedtuple('Point', ['x', 'y'])
p = Point(2, 3)

print(p.x) # Вывод: 2
print(p.y) # Вывод: 3
```

4. defaultdict

defaultdict — это подкласс словаря, который предоставляет значение по умолчанию для отсутствующих в словаре ключей. Это удобно при обработке словаря, когда нужно избежать проверок наличия ключей.

Пример использования:

```
from collections import defaultdict

# Создание defaultdict со значением по умолчанию — пустым списком
d = defaultdict(list)

d['apple'].append('red') # Добавление значения 'red' к ключу 'apple'
d['banana'].append('yellow') # Добавление значения 'yellow' к ключу 'banana'
d['apple'].append('green') # Добавление значения 'green' к ключу 'apple'

print(d) # Вывод: defaultdict(<class 'list'>, {'apple': ['red', 'green'], 'banana': ['yellow']})
print(d['apple']) # Вывод: ['red', 'green']

print(d['banana']) # Вывод: ['yellow']

print(d['cherry']) # Вывод: [] (пустой список, значение по умолчанию)
print(d) # Вывод: defaultdict(<class 'list'>, {'apple': ['red', 'green'], 'banana': ['yellow'],
'cherry': []})
```

В этом примере defaultdict(list) создаёт словарь, в котором с каждым отсутствующим ключом будет автоматически сопоставлен пустой список.

Это позволяет добавлять значения к ключам без предварительной проверки их существования. Если вы обратитесь к несуществующему ключу, defaultdict автоматически создаст новую запись со значением по умолчанию.

Таким образом, использование defaultdict позволяет сделать код более читабельным и избежать необходимости проверки наличия ключей в словаре перед добавлением или обращением к ним.

Очереди Queue и LifoQueue

Мы рассмотрим две самые популярные и полезные реализации очереди: Queue и LifoQueue.

В документации вы можете узнать об остальных очередях.

Модуль queue

Установка: модуль queue также является встроенным модулем Python, поэтому нет необходимости в его дополнительной установке.

Подключение: вы можете импортировать классы очередей, например, с помощью

```
from queue import Queue, LifoQueue
```

* Ниже будет использован термин «потокобезопасность».

Вы ещё не рассматривали тему многопоточного программирования, но это является важным преимуществом данных коллекций, поэтому стоит вернуться к ней ещё раз после изучения многопоточности.

Queue

Queue из модуля queue представляет собой осуществление потокобезопасной очереди (queue) в Python. Она реализует принцип First-In/First-Out (FIFO), «первый вошёл — первый вышел», то есть работает как обычная, привычная вам, очередь (пришёл в магазин, встал в начало очереди — первым купил товар, встал в конец очереди — последним купил товар).

Этот класс предоставляет методы для добавления и удаления элементов из очереди и может использоваться для синхронизации данных между потоками в многопоточных приложениях.

Некоторые основные аспекты и примеры использования Queue:

```
from queue import Queue

# Создание экземпляра очереди
q = Queue()

# Добавление элементов в очередь
q.put(1)
q.put(2)
q.put(3)

# Получение и удаление элемента из очереди
item = q.get()
print(item) # Вывод: 1

# Проверка, пуста ли очередь
is_empty = q.empty()
print(is_empty) # Вывод: False

# Получение размера очереди
size = q.qsize()
print(size) # Вывод: 2
# Очистка очереди
q.queue.clear()

# Проверка, пуста ли очередь после очистки
is_empty = q.empty()
print(is_empty) # Вывод: True
```

Важно отметить, что Queue из модуля queue является реализацией очереди, а не стека. Если вам нужен стек, можете использовать LifoQueue, который вы рассмотрите ниже.

LifoQueue

LifoQueue из модуля queue представляет реализацию стека (стека по принципу Last-In/First-Out (LIFO), или «последний вошёл, первый вышел»).

Он предоставляет методы для добавления и удаления элементов из стека.

Посмотрите на ключевые аспекты и примеры использования LifoQueue:

```
from queue import LifoQueue

# Создание экземпляра стека
```

```
stack = LifoQueue()

# Добавление элементов в стек
stack.put(1)
stack.put(2)
stack.put(3)

# Получение и удаление элемента из стека
item = stack.get()
print(item) # Вывод: 3

# Проверка, пуст ли стек
is_empty = stack.empty()
print(is_empty) # Вывод: False

# Получение размера стека
size = stack.qsize()
print(size) # Вывод: 2

# Очистка стека
stack.queue.clear()

# Проверка, пуст ли стек после очистки
is_empty = stack.empty()
print(is_empty) # Вывод: True
```

LifoQueue предоставляет функциональность стека, где последний добавленный элемент будет первым удалённым (LIFO: Last-In/First-Out). Вы можете добавлять элементы в стек при помощи метода put и получать их при помощи метода get.

Важно отметить, что LifoQueue реализует именно структуру данных стека.

Вы можете заметить, что между Queue и LifoQueue нет большой разницы, и вы будете правы, но эта разница имеет ключевое значение, из-за неё каждая структура используется для своих задач.

Примеры задач, которые решаются при помощи стека:

1. Редактор текста с отменой и повтором действий.
   Редакторы текста часто используют стек для реализации функциональности отмены (undo) и повтора (redo) действий. Каждое выполненное действие помещается в стек, и пользователь может отменить или повторить последние действия, извлекая их из стека.
2. Обработка вызовов функций.
   Во время выполнения программы стек может использоваться для управления

вызовами функций. Каждый раз при вызове функции информация о нём помещается в стек и извлекается из стека, когда функция завершается, чтобы вернуться к предыдущему контексту выполнения.

Примеры задач, которые решаются при помощи очереди:

1. Обработка задач веб-сервером.
   Веб-серверы обычно используют очередь для управления запросами от клиентов. Поступившие запросы помещаются в очередь, а сервер обрабатывает их в порядке получения. Это обеспечивает справедливое распределение ресурсов и обработку запросов в порядке очерёдности.
2. Кеширование данных.
   Очередь может использоваться для кеширования данных в системе, наиболее актуальные данные которой хранятся в начале очереди, а более старые — в конце. При достижении максимального размера очереди старые данные будут автоматически удаляться, чтобы освободить место для новых.
3. Обработка задач в фоновом режиме.
   Очередь может использоваться для обработки задач в фоновом режиме, например отправки электронных писем или обработки длинных вычислений. Задачи помещаются в очередь, а отдельный процесс или поток обрабатывает их по мере доступности ресурсов.

Связанные списки

Связанный список (Linked List) — это структура данных, состоящая из узлов, каждый из которых содержит значение и ссылку на следующий узел.

Отличие связанного списка от обычного (который представлен в Python в виде встроенного типа данных list) заключается в способе хранения и организации элементов.

В связанном списке каждый элемент (узел) содержит ссылку на следующий элемент, а в обычном списке элементы хранятся в памяти последовательно.

Преимущества связанных списков:

1. Эффективные операции вставки и удаления.
   Вставка и удаление элементов в середину или в начало списка выполняются быстрее, чем при работе с обычным списком, и не требуют перемещения других элементов или реорганизации памяти.
2. Простота реализации.
   Реализация связанного списка относительно проста по сравнению с другими структурами данных, такими как графы или деревья.
3. Гибкость доступа.
   Связанный список обеспечивает эффективный доступ к элементам по индексу, как в

обычном списке. Однако он не обладает преимуществами произвольного доступа к элементам, как в случае с обычным списком.

Недостатки связанных списков:

1. Ограниченные операции произвольного доступа.
   Доступ к элементам связанного списка осуществляется последовательно, начиная с первого узла. Операции произвольного доступа (например, доступ к элементу по индексу) могут требовать прохода по всем предшествующим узлам, что занимает O(n) времени.
2. Дополнительное использование памяти.
   Каждый узел связанного списка требует дополнительную память для хранения значения и ссылку на следующий узел, что может привести к дополнительному использованию памяти.

Простейшая реализация связанных списков:

```
class Node:
  def __init__(self, data, next=None):
    # Конструктор узла
    self.data = data  # Значение узла
    self.next = next  # Ссылка на следующий узел
```

Так как связанный список — это не какой-либо контейнер с набором объектов, а, скорее, набор обособленных объектов, которые ссылаются друг на друга, то уже этого кода будет достаточно, чтобы сделать связанный список.

```
# Для этого нам надо создать корневой элемент
first = Node(123)
# И следующий элемент, который будет ссылаться на корневой
second = Node(456, first)
# Получаются два отдельных объекта, которые при этом связаны ссылкой (второй объект ссылается на первый)
```

В источниках вы, скорее всего, найдёте реализацию, которая будет включать в себя ещё и второй класс LinkedList. Он является «менеджером», с помощью которого облегчается работа с набором узлов Node.

Применение связанных списков:

1. Реализация стека и очереди.
   Связанные списки часто используются для реализации стека (LIFO — «последний вошёл, первый вышел») и очереди (FIFO — «первый вошёл, первый вышел») структур данных. Ссылки между узлами позволяют эффективно добавлять и удалять элементы как в начале, так и в конце списка.
2. История действий.
   В некоторых приложениях, таких как текстовые редакторы или веб-браузеры,

связанные списки могут использоваться для хранения истории действий пользователя. Каждый элемент списка представляет отдельное действие, и перемещение вперёд или назад в списке позволяет восстановить предыдущие состояния.

3. Реализация графов и деревьев.
   Связанные списки могут использоваться для реализации структур данных, таких как графы и деревья. Каждый узел списка может быть связан с другими узлами, представляя отношения и иерархии между элементами.
4. Обработка больших объёмов данных.
   Связанные списки могут быть полезны при обработке больших объёмов данных, когда требуется эффективное добавление и удаление элементов. Это может быть полезно, например, при чтении и записи больших файлов или при обработке потоков данных в режиме реального времени.
5. Хранение и управление структурированными данными.
   Связанные списки могут использоваться для хранения и управления структурированными данными, такими как списки контактов, задачи в планировщике или элементы меню в приложении. Ссылки между узлами обеспечивают связи и порядок элементов.

Пункт 3 выделен жирным, так как он особенно важен при рассмотрении темы связанных списков.

Идея хранения связи внутри узла нашла широкое практическое применение и начала развиваться. Принцип хранения одной связи, ведущей к следующему объекту, позволяет вам хранить:

1. Связь, ведущую как к следующему, так и к предыдущему объекту (такая структура называется двусвязным списком).
2. Связи с двумя следующими объектами (такие структуры данных относят к древовидным, их много, и они популярны; вы познакомитесь с самым известным из них).
3. Связи с неограниченным количеством, в разных направлениях (графы).

Бинарное дерево

Самый простой вид структуры данных типа «дерево» — это бинарное дерево.

Бинарное дерево состоит из узлов (этим оно похоже на связанный список), каждый из которых имеет не более двух дочерних узлов (в отличие от связанного списка), называемых левым и правым поддеревом.

Каждый узел содержит значение (или ключ) и ссылки на его дочерние узлы.

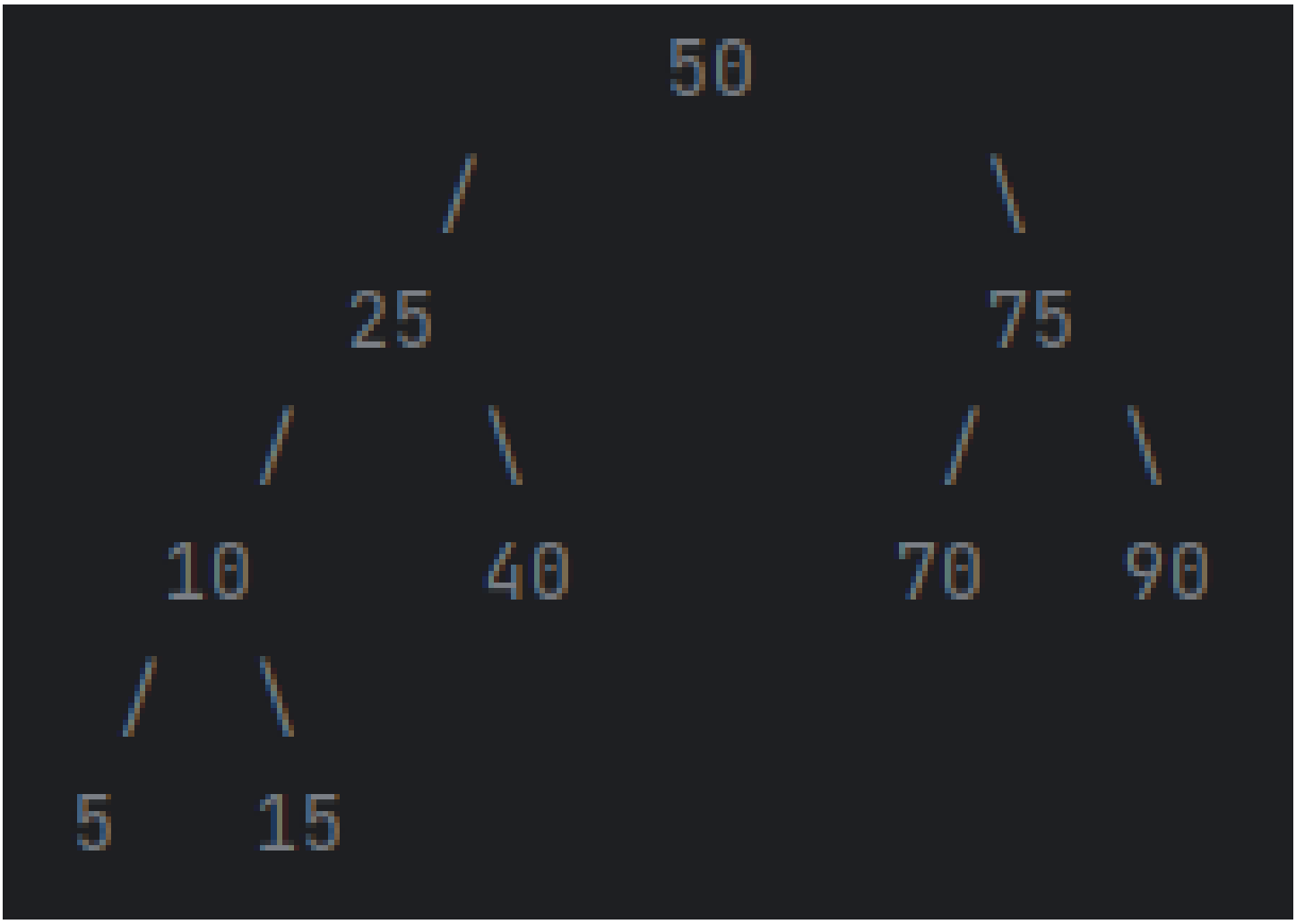
Пример реализации бинарного дерева:

```
class Node:
  def __init__(self, key):
    # Конструктор узла
    self.key = key
    self.left = None
    self.right = None

# Пример создания бинарного дерева
root = Node(10) # Создаём корневой узел
root.left = Node(2) # Добавляем левый дочерний узел
root.right = Node(15) # Добавляем правый дочерний узел

# Визуализация бинарного дерева:
# 10
# / \
# 2 15
```

В данном примере мы создали простое бинарное дерево с корневым узлом, левым и правым дочерними узлами. Это самая простая форма дерева, в которой каждый узел имеет не более двух дочерних узлов.

Обратите внимание, что левый узел содержит значение, которое меньше своего «родителя», а правый узел содержит значение, которое больше своего «родителя».

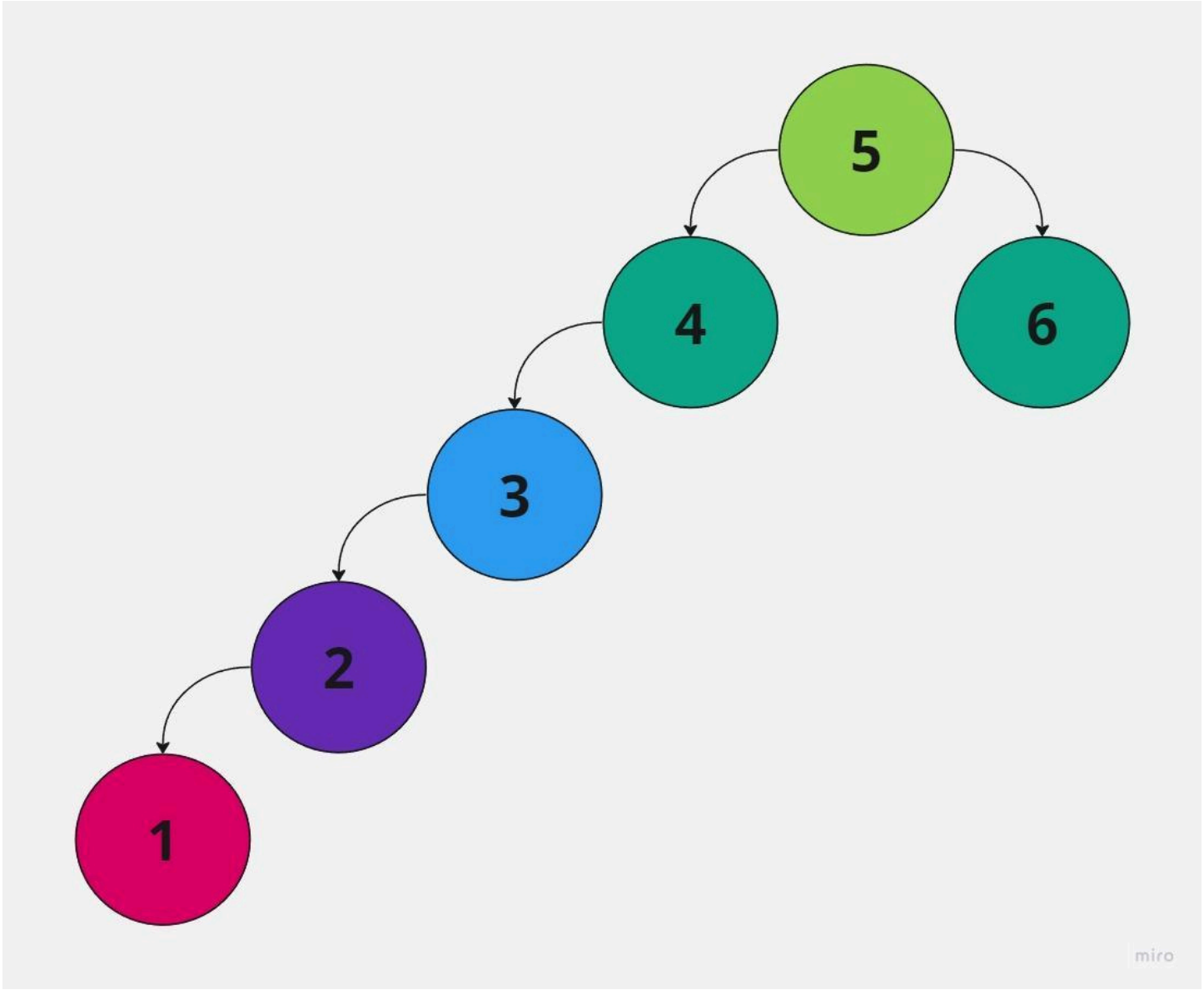
Благодаря этой особенности, деревья часто используют для облегчения поиска элементов.



Каждый узел выполняет оговорённое ранее правило (левый меньше, правый больше). Если вы захотите найти, например, число 15, то вам будет нужно следовать следующей логике:

15 больше или меньше 50?
Если меньше, то двигайтесь влево.
15 больше или меньше 25?
Если меньше, то опять двигайтесь влево.
15 больше или меньше 10?
Если больше — двигайтесь вправо.
И вот вы нашли число 15!

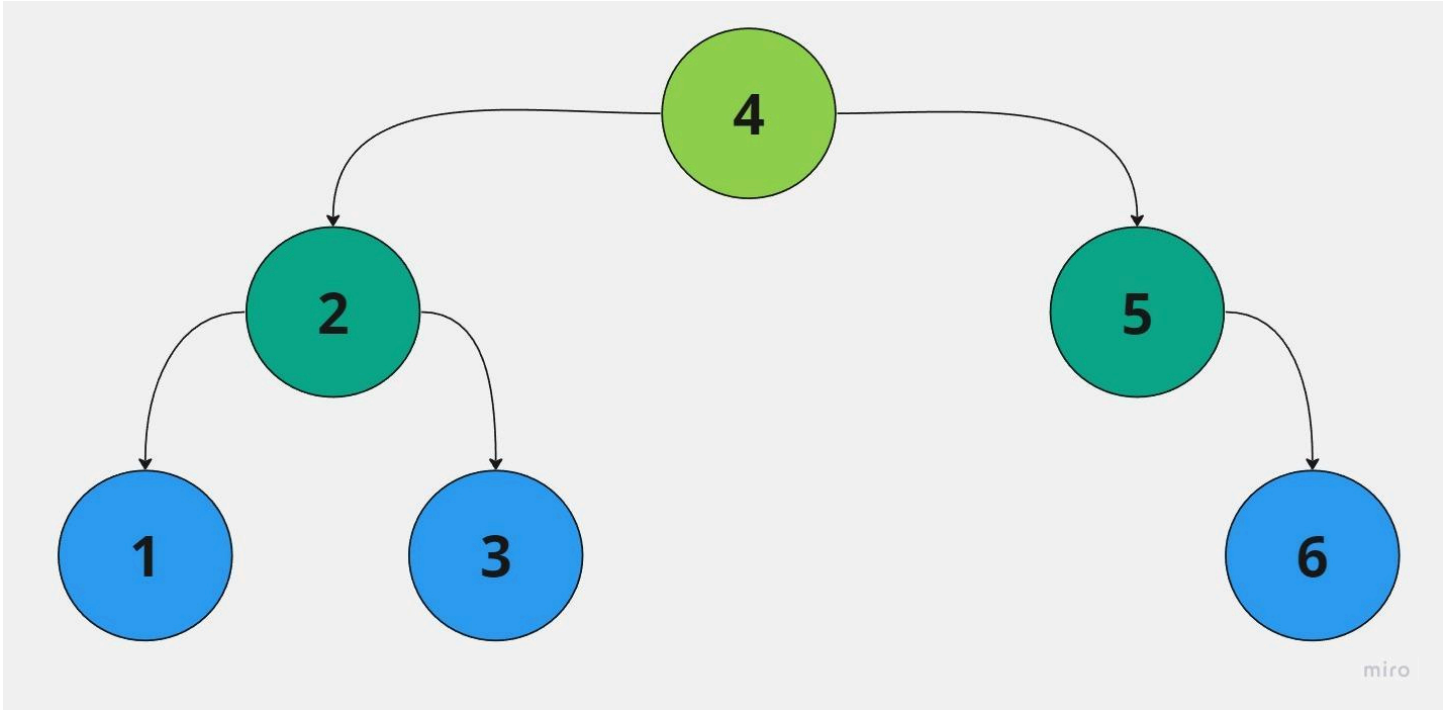
Подобный поиск происходит очень быстро, но поддержание дерева (добавление элементов) имеет свои особенности. Например, дерево может расти неравномерно, новые элементы — добавляться в одну сторону, в итоге глубина конечных элементов (листьев) будет различаться слева и справа, это может выглядеть так:



Вы можете заметить, что в таком случае теряется преимущество поиска, ведь проверяется одно число за другим практически линейно (за одну проверку отбрасывается максимум одно число).

Чтобы получить преимущество в поиске, придётся перебалансировать его. Чем ровнее будет дерево, тем проще будет выполнять по нему поиск (так как за одно сравнение будет откинуто больше элементов).

В итоге дерево из примера может быть приведено к такому виду:



Теперь все листья имеют одинаковую глубину, дерево сбалансировано и вы можете найти любое из чисел за две проверки (в предыдущем примере для поиска числа 1 могло понадобиться пять проверок).

binarytree

binarytree — это модуль, который позволяет создавать и визуализировать бинарные деревья и работать с их различными типами.

Он позволяет легко создавать случайные бинарные деревья и обеспечивает методы для их обхода, поиска и других операций.

Для установки модуля binarytree вы можете использовать менеджер пакетов pip. Откройте терминал и выполните следующую команду:

pip install binarytree

После успешной установки вы можете начать использовать модуль binarytree.

Посмотрите пример кода, демонстрирующий создание и работу с бинарным деревом с использованием binarytree:

```
from binarytree import Node

# Создание бинарного дерева вручную
root = Node(10)
```

```
root.left = Node(5)
root.right = Node(15)
root.left.left = Node(3)
root.left.right = Node(7)

# Вывод структуры дерева
print("Структура бинарного дерева:")
print(root)

# Обход дерева в прямом порядке (preorder)
def preorder(node):
    if node is not None:
        print(node.value)
        preorder(node.left)
        preorder(node.right)

print("Обход дерева в прямом порядке (preorder):")
preorder(root)

# Поиск элемента в дереве
def search(node, value):
    if node is None or node.value == value:
        return node
    if value < node.value:
        return search(node.left, value)
    return search(node.right, value)

print("Поиск элемента в дереве:")
result = search(root, 7)
if result is not None:
    print("Элемент найден!")
else:
    print("Элемент не найден!")
```

В этом примере класс Node импортируется из модуля binarytree и вручную создаётся бинарное дерево.

Затем выводится структура дерева и выполняется его обход в прямом порядке (preorder) с помощью рекурсивной функции preorder.

После этого выполняется поиск элемента в дереве с помощью функции search. В примере иллюстрируется базовое использование модуля binarytree.

Важно отметить, что модуль binarytree предоставляет и другие возможности, такие как генерация случайных деревьев, визуализация дерева и поддержка различных типов бинарных деревьев. Вы можете ознакомиться с подробной документацией модуля binarytree.

Бинарные деревья имеют широкий спектр прикладного применения в различных областях. Вот несколько примеров прикладного использования бинарных деревьев:

1. Базы данных.
   Бинарные деревья могут быть использованы в базах данных для построения индексов. Например, в базе данных, содержащей миллионы записей, можно использовать бинарное дерево для хранения и быстрого поиска ключей записей. Ключи записей будут храниться в узлах дерева, а поиск будет осуществляться по значению ключа.
2. Алгоритмы сжатия данных.
   В алгоритмах сжатия данных, таких как алгоритм Хаффмана, бинарные деревья используются для построения оптимальных префиксных кодов. При этом каждый символ или символьная последовательность представляются в виде пути от корня дерева до соответствующего листового узла.
3. Реализация кеша или кеширования данных.
   Бинарные деревья могут быть использованы для создания кеша данных, в которых часто используемые элементы хранятся в памяти для быстрого доступа. При поиске элемента в кеше бинарное дерево позволяет эффективно проверять наличие элемента и получать его значение.

Это только несколько примеров прикладного использования бинарных деревьев. Бинарные деревья широко применяются в различных областях, в которых требуется организация данных и эффективный доступ к ним.

Итоги

В данном модуле вы:

- познакомились с библиотеками collections, queue, binarytree;
- подробнее изучили стек, очередь, связанные списки, бинарные деревья;
- узнали о прикладном применении всех этих структур данных.