Тема 15. Структуры данных: деревья

Деревья — одна из фундаментальных структур данных, позволяющая эффективно представлять и обрабатывать иерархическую информацию. От файловых систем и организационных схем до языковых разборов и моделей принятия решений — деревья лежат в основе множества прикладных и теоретических задач. Понимание того, как устроены деревья, какие у них бывают разновидности и как с ними работают алгоритмы, важно не только с точки зрения практического программирования, но и для формирования алгоритмического мышления.

Будет рассмотрен широкий спектр тем: от общего представления деревьев и принципов их построения до бинарных деревьев, деревьев поиска, двоичных куч и приоритетных очередей. Особое внимание уделяется реализациям на языке Python — как с использованием собственных классов, так и с привлечением специализированных библиотек. Помимо теоретических понятий и базовых операций (вставка, удаление, обход), в рамках темы будут даны примеры практического применения деревьев, а также возможности их визуализации, что делает работу с ними более наглядной и интуитивной.

Знакомство с бинарными деревьями

Деревья и их представления

Дерево представляет собой разновидность графа, не содержащего циклов и обладающего особым типом связности. В любом дереве между двумя узлами существует только один путь, что делает эту структуру удобной для представления иерархий и вложенных зависимостей. Основными элементами дерева являются *__узлы__* (или *__вершины__*) *и связи между ними (__ребра__*), при этом важную роль играют отношения между родительскими и дочерними элементами.

Если один узел указывает на другой, то первый называют *__родительским__*, а второй* — **дочерним**. *Несколько узлов, имеющих одного и того же родителя, называются сиблингами. Специальным элементом дерева является* **корень*** — единственный узел, не имеющий родителя. От него начинается вся структура дерева. Узлы, не имеющие потомков, называются *__листьями__*; они завершают ветви дерева. Промежуточные элементы, имеющие потомков, называются внутренними узлами.

Каждому узлу можно сопоставить *__уровень__* — количество шагов от корня до него. Корень дерева имеет уровень 0. Наибольший уровень среди всех узлов определяет *__высоту__* дерева, которая отражает его максимальную вложенность и используется для оценки глубины структуры. Кроме того, важным понятием является *__поддерево__* — это часть дерева, включающая некоторый узел и всех его потомков. Поддеревья позволяют изолировать части дерева для локального анализа или обработки.

Деревья могут быть представлены различными способами в зависимости от поставленных задач и контекста использования. На рис. 1 показаны три наиболее распространённых способа представления: иерархическое, в виде вложенных множеств и линейное.

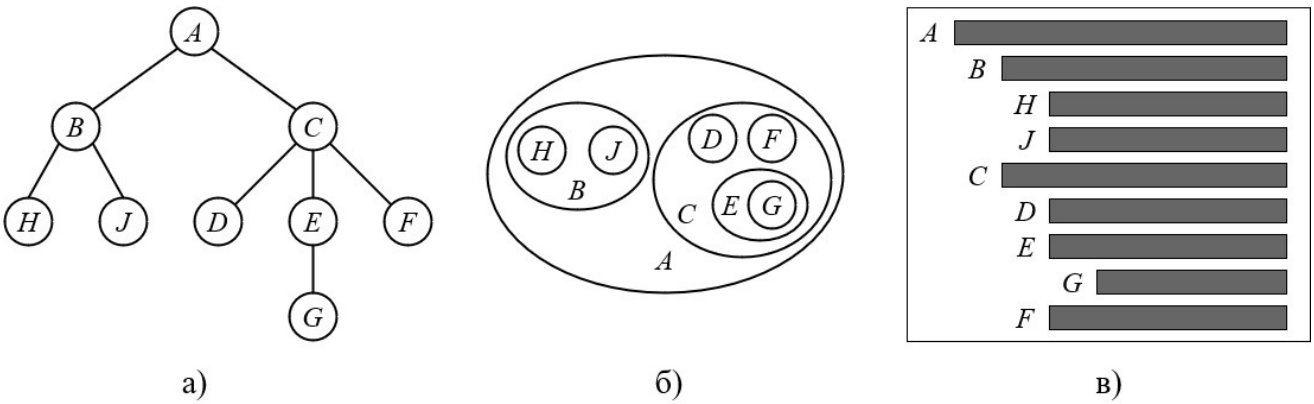


Рисунок 1 — Представление деревьев: а) иерархическая структура, б) множества, в) линейное представление

__Иерархическое__ представление основано на графической форме дерева, где каждый узел связан с подчинёнными ему элементами визуально, сверху вниз. Такое представление даёт наиболее наглядное представление о структуре и взаимоположении узлов.

__Множественное__ представление фокусируется на отношениях вложенности между элементами. Здесь каждый элемент рассматривается как множество, содержащее свои подмножества, что позволяет представить иерархию через включения. Это особенно удобно для логического анализа и формального описания.

__Линейное__ представление строится с помощью обхода дерева — алгоритма, который последовательно посещает все узлы в определённом порядке (например, в глубину или в ширину). Результатом обхода становится линейный список, в котором сохраняется относительное положение элементов. Такой формат удобен для хранения, передачи и обработки дерева средствами стандартных списков.

Рассмотрим пример программной реализации произвольного дерева, позволяющий наглядно понять, как может быть устроена иерархическая структура с помощью классов и методов. Такой подход поможет не только закрепить основные понятия, связанные с деревьями, но и научиться работать с узлами, строить связи между ними и анализировать полученную структуру. Для удобства восприятия создаваемое дерево будет визуализироваться с использованием библиотеки `graphviz`, однако стоит отметить, что существует множество альтернативных способов представления и отображения деревьев, включая текстовые схемы и специализированные графические инструменты.

Для визуального представления деревьев в иерархической форме удобно использовать библиотеку `graphviz`, которая позволяет создавать ориентированные графы. В средах Jupyter и Google Colab её можно установить командой:

In [ ]:
```
!pip install graphviz
```

После установки переходим к определению структуры дерева. Реализация строится на двух классах: `Node`, представляющем отдельный узел, и `Tree`, инкапсулирующем всю структуру. Узел содержит значение и список потомков, а дерево предоставляет методы добавления, поиска и визуализации.

In [ ]:
```
from graphviz import Digraph
```

In [ ]:
```
class Node:
    def __init__(self, value):
        self.value = value
        self.children = []

class Tree:
    def __init__(self):
        self.root = None

    def add_node(self, value, parent_value=None):
        node = Node(value)
        if parent_value is None:
            if self.root is not None:
                raise ValueError("У дерева уже есть корень")
            self.root = node
        else:
            parent_node = self.find_node(parent_value)
            if parent_node is None:
                raise ValueError("Родительский узел не найден")
            parent_node.children.append(node)

    def find_node(self, value):
        return self._find_node(value, self.root)

    def _find_node(self, value, node):
        if node is None:
            return None
        if node.value == value:
            return node
        for child in node.children:
            found = self._find_node(value, child)
            if found is not None:
                return found
        return None

    def visualize(self):
        dot = Digraph()

        def add_nodes_edges(node):
            dot.node(str(id(node)), str(node.value))
            for child in node.children:
                dot.node(str(id(child)), str(child.value))
                dot.edge(str(id(node)), str(id(child)))
                add_nodes_edges(child)

        if self.root:
            add_nodes_edges(self.root)
        return dot
```

Класс `Node` представляет собой структуру, описывающую один узел дерева. Конструктор `__init__` принимает аргумент `value`, который сохраняется в атрибуте `self.value` и может представлять любое значение, например, число, строку или объект. Второй атрибут `self.children` инициализируется пустым списком и предназначен для хранения дочерних узлов, связанных с данным узлом. Таким образом, каждый экземпляр `Node` знает своё значение и содержит список потомков, с которыми может быть связан в структуре дерева. Это позволяет формировать произвольную (неограниченную по числу потомков) иерархию.

Класс `Tree` отвечает за представление всей структуры дерева. Его конструктор `__init__` содержит единственный атрибут `self.root`, инициализируемый значением `None`. Он будет указывать на корневой узел дерева, который создаётся при первом добавлении узла без указания родителя.

Метод `add_node` предназначен для добавления нового узла в дерево. Он принимает два аргумента: `value` — значение нового узла, и `parent_value` — значение родительского узла. Если родительское значение не указано, предполагается, что добавляется корень дерева. В этом случае происходит проверка: если корень уже существует, возбуждается исключение `ValueError`, предотвращающее повторное создание корня. Если же `parent_value` задан, метод ищет родительский узел с указанным значением с помощью метода `find_node`. Если родитель найден, новый узел добавляется в его список потомков. В противном случае возбуждается исключение, указывающее на то, что родительский узел не существует в текущей структуре.

Метод `find_node` реализует поиск узла по значению. Он вызывает вспомогательную рекурсивную функцию `_find_node`, которая начинает обход с корневого узла. В этой функции сначала проверяется, не является ли текущий узел искомым. Если нет, то для каждого дочернего узла вызывается та же функция. Как только значение найдено, оно возвращается наружу. Если поиск доходит до конца дерева без результата, возвращается `None`.

Метод `visualize` позволяет построить наглядное графическое представление дерева с помощью библиотеки `graphviz`. Он создаёт объект `Digraph`, который хранит вершины и рёбра графа. Внутри него определена рекурсивная функция `add_nodes_edges`, которая для каждого узла создаёт вершину (вызов `dot.node`) и соединяет её с дочерними вершинами (вызов `dot.edge`). При добавлении узлов используется идентификатор `id(node)` в качестве имени узла, что гарантирует уникальность и предотвращает коллизии между одинаковыми значениями. Вывод визуализации возвращается в виде объекта `Digraph`. Этот метод полезен для представления структуры дерева в удобном и интуитивно понятном виде.

Для генерации произвольного дерева заданной глубины используется функция `create_tree`. Она строит дерево по уровням, создавая для каждого узла от одного до трёх дочерних элементов. Значения узлов выбираются случайно и перемешиваются перед назначением.

In [ ]:
```python
from random import shuffle, randint
```

In [ ]:
```python
def create_tree(levels):
    tree = Tree()
    total_nodes = sum(3**i for i in range(levels + 1))
    values = list(range(1, total_nodes + 1))
    shuffle(values)

    tree.add_node(values[0])
    index = 1
    queue = [values[0]]

    current_level = 0
    nodes_in_current_level = 1
    nodes_in_next_level = 0

    while index < len(values) and current_level < levels:
        for _ in range(nodes_in_current_level):
            if not queue:
                break
            parent_value = queue.pop(0)
            num_children = randint(1, 3)
            for _ in range(num_children):
                if index >= len(values):
                    break
                child_value = values[index]
                tree.add_node(child_value, parent_value)
                queue.append(child_value)
                index += 1
                nodes_in_next_level += 1
        current_level += 1
        nodes_in_current_level, nodes_in_next_level = nodes_in_next_level, 0

    return tree
```

Функция `create_tree` предназначена для генерации произвольного дерева, глубина которого задаётся параметром `levels`. Это означает, что дерево будет состоять из нескольких уровней, где каждый уровень может содержать несколько узлов, а каждый узел может иметь от одного до трёх потомков. Такая структура позволяет моделировать более общие деревья, приближённые к реальным иерархическим данным.

На первом этапе вычисляется общее количество узлов, которое будет создано. Это делается с помощью формулы `sum(3**i for i in range(levels + 1))`. Здесь предполагается, что на каждом уровне может быть максимум 3^i узлов (аналог расширенного дерева, где каждый узел может иметь до трёх детей). Полученное количество задаёт максимально возможное число узлов, которое понадобится при построении дерева. Затем формируется список `values`, содержащий уникальные значения от `1` до общего числа узлов, и он перемешивается функцией `shuffle`. Это обеспечивает случайность значений и делает структуру дерева менее предсказуемой.

Далее начинается построение дерева. Сначала создаётся корневой узел с первым значением из перемешанного списка. После этого переменная `index` инициализируется значением `1`, так как первый элемент уже использован. В список `queue` добавляется значение корня — он станет первым родителем для последующих узлов.

Построение дерева осуществляется по уровням. Для этого используется цикл `while`, в котором контролируется текущий уровень (`current_level`) и количество узлов на нём (`nodes_in_current_level`). На каждой итерации цикла извлекаются родительские узлы из очереди, и для каждого из них случайным образом определяется количество потомков — от одного до трёх (`num_children = randint(1, 3)`). Это создаёт вариативность в структуре дерева: некоторые узлы будут иметь одного потомка, другие — трёх.

Для каждого создаваемого потомка берётся следующее значение из перемешанного списка, добавляется в дерево с помощью метода `add_node`, в качестве родителя указывается соответствующий узел. Новый узел также добавляется в `queue`, чтобы впоследствии использовать его как родителя для следующего уровня. После добавления всех потомков переменные `index` и `nodes_in_next_level` обновляются соответственно.

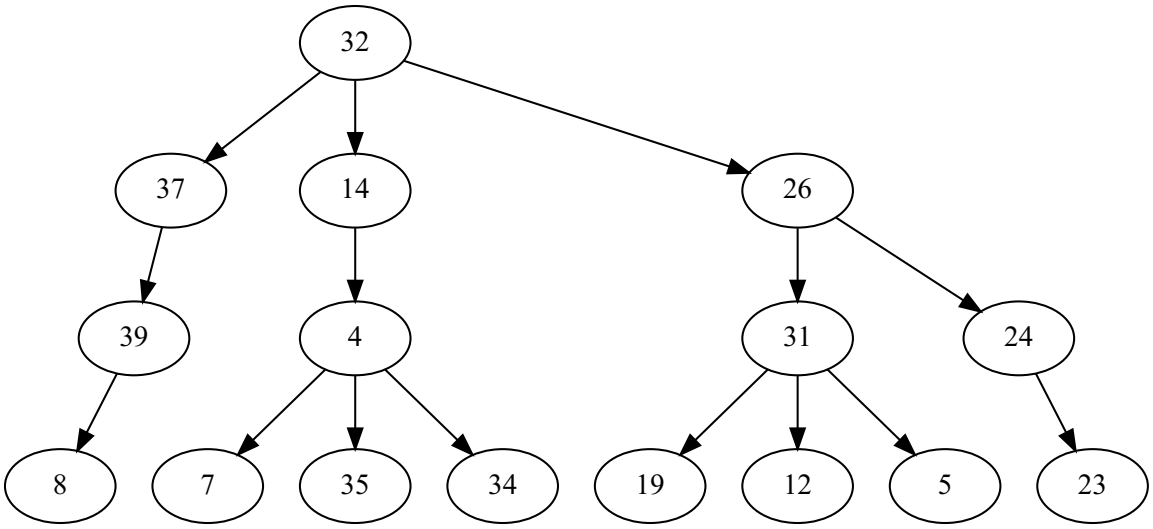
По завершении всех операций уровень увеличивается (`current_level += 1`), и начинается обработка следующего уровня: число узлов в текущем уровне обновляется по значению `nodes_in_next_level`, а счётчик для следующего уровня сбрасывается в ноль.

Когда достигнут заданный уровень (`levels`) или закончатся доступные значения, цикл завершает работу. Результатом выполнения функции становится объект дерева с иерархией, сформированной по заданной глубине, и со случайным распределением значений и числа потомков. Такая функция идеально подходит для тестирования структур и методов работы с деревьями, обеспечивая разнообразие форм и конфигураций.

Для демонстрации работы кода и отображения структуры дерева вызывается функция `create_tree`, а затем применяется метод `visualize()`. Так как используется среда Google Colab (или Jupyter Notebook), результат визуализации будет отрисован прямо в ячейке.

In [ ]:
```
tree = create_tree(3)
tree.visualize()
```

Out[ ]:



В результате построено произвольное дерево глубиной до трёх уровней, а графическое отображение иерархии узлов позволяет наглядно представить его структуру.

Рассмотрим практическое применение реализованной структуры данных на основе дерева. Каждая из представленных задач демонстрирует один из вариантов обхода и обработки узлов, позволяя лучше понять логику рекурсии и работы с иерархическими данными. Для всех примеров используется дерево, созданное функцией `create_tree`, а визуализация выполняется с помощью метода `visualize`.

- Замена каждого значения на сумму значений всех потомков

  Требуется пройти по всем узлам дерева и заменить значение каждого узла на сумму значений всех его потомков. Листовые узлы остаются неизменными, так как у них нет потомков.

In [ ]:
```
def replace_with_sum_of_children(tree, node=None):
    if node is None:
        node = tree.root
    if not node.children:
        return node.value
    sum_of_children = 0
    for child in node.children:
        sum_of_children += replace_with_sum_of_children(tree, child)
    node.value = sum_of_children
    return sum_of_children
```

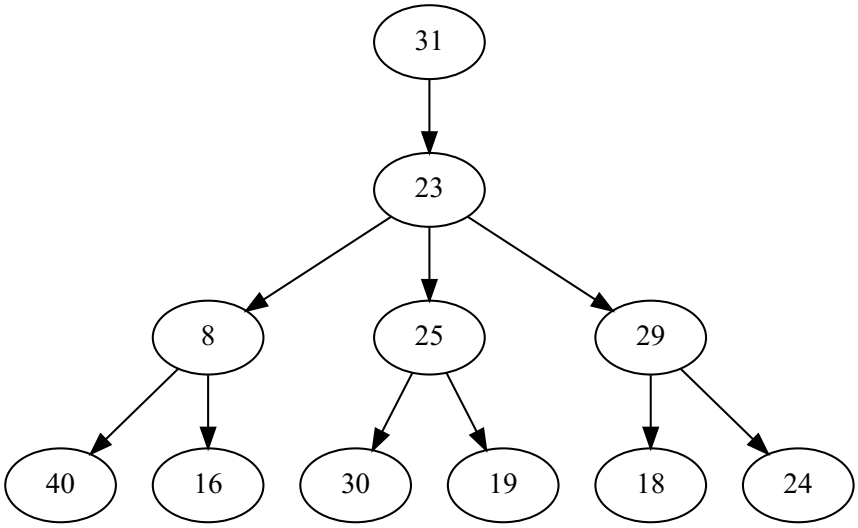
Алгоритм замены значений построен на рекурсивном обходе дерева. Начинается он с корневого узла: если у узла нет дочерних элементов, то есть он является листом, его значение возвращается без изменений. В противном случае функция вызывает сама

себя для каждого дочернего узла, получая от них значения, которые в совокупности составляют сумму всех потомков текущего узла. После завершения обхода дочерних элементов это значение сохраняется в текущем узле, заменяя его изначальное значение, и возвращается как результат, передаваемый на уровень выше. Таким образом, на каждом уровне рекурсии значение узла становится суммой значений всех узлов, расположенных ниже по иерархии.

Пример использования:

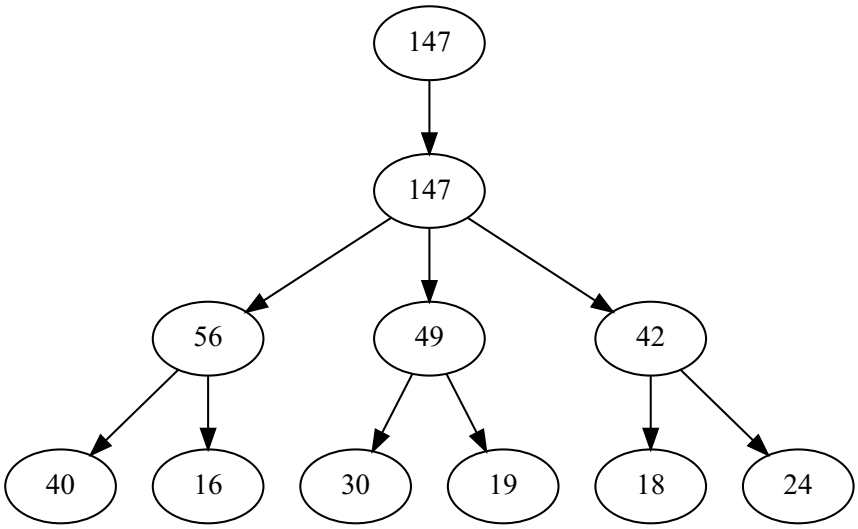
In [ ]:
```
tree = create_tree(3)
print("Исходное дерево:")
display(tree.visualize())
```

```
Исходное дерево:
```



In [ ]:
```
replace_with_sum_of_children(tree)
print("Полученное дерево после замены значений:")
display(tree.visualize())
```

```
Полученное дерево после замены значений:
```



- Удвоение всех нечётных значений

  Необходимо обойти все узлы дерева и удвоить значение каждого узла, если оно является нечётным.

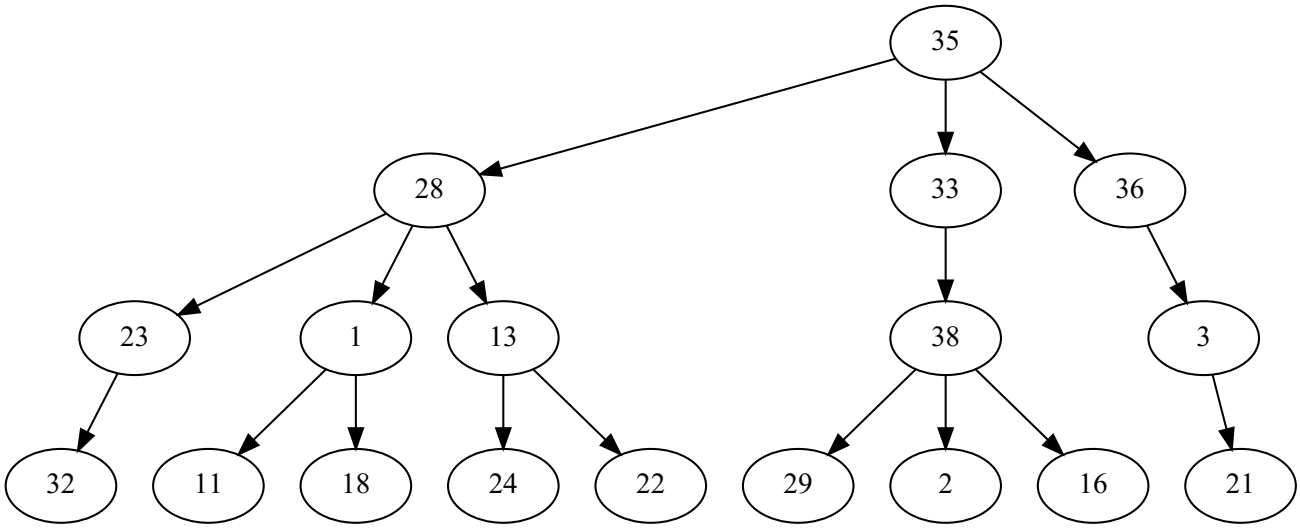
In [ ]:
```
def double_odd_values(tree, node=None):
    if node is None:
        node = tree.root
    if node.value % 2 == 1:
        node.value *= 2
    for child in node.children:
        double_odd_values(tree, child)
```

Рекурсивный алгоритм, реализующий удвоение нечётных значений в дереве, проходит по всем узлам, начиная с корня. На каждом шаге проверяется, является ли значение текущего узла нечётным — для этого используется операция взятия остатка от деления на два. Если условие выполняется, значение узла заменяется на его удвоенное значение. После обработки текущего узла функция вызывается рекурсивно для всех его потомков, тем самым обеспечивая полный обход дерева. Возврат значения не требуется, поскольку модификации производятся непосредственно в структуре дерева.

Пример использования:

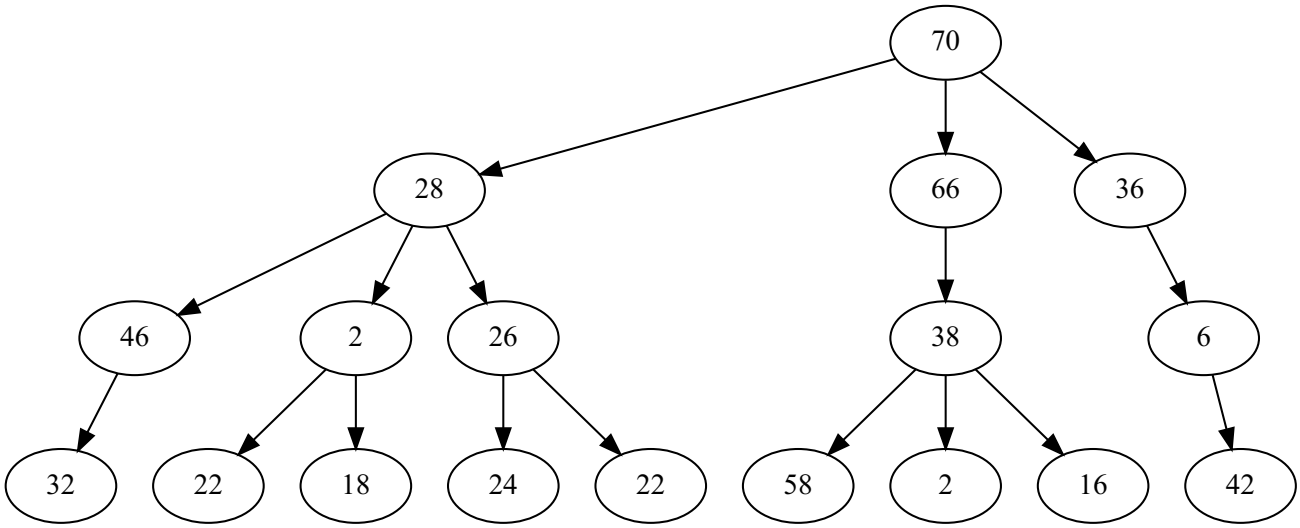
In [ ]:
```
tree = create_tree(3)
print("Исходное дерево:")
display(tree.visualize())
```

```
Исходное дерево:
```



In [ ]:
```
double_odd_values(tree)
print("Дерево после удвоения нечётных значений:")
display(tree.visualize())
```

```
Дерево после удвоения нечётных значений:
```



- Определение всех листьев дерева

  Требуется определить все листья дерева, то есть те узлы, у которых нет потомков. Результатом должна стать коллекция значений листьев.

In [ ]:
```
def find_leaves(tree, node=None, leaves=None):
    if leaves is None:
        leaves = []
    if node is None:
        node = tree.root
    if len(node.children) == 0:
        leaves.append(node.value)
    else:
        for child in node.children:
            find_leaves(tree, child, leaves)
    return leaves
```

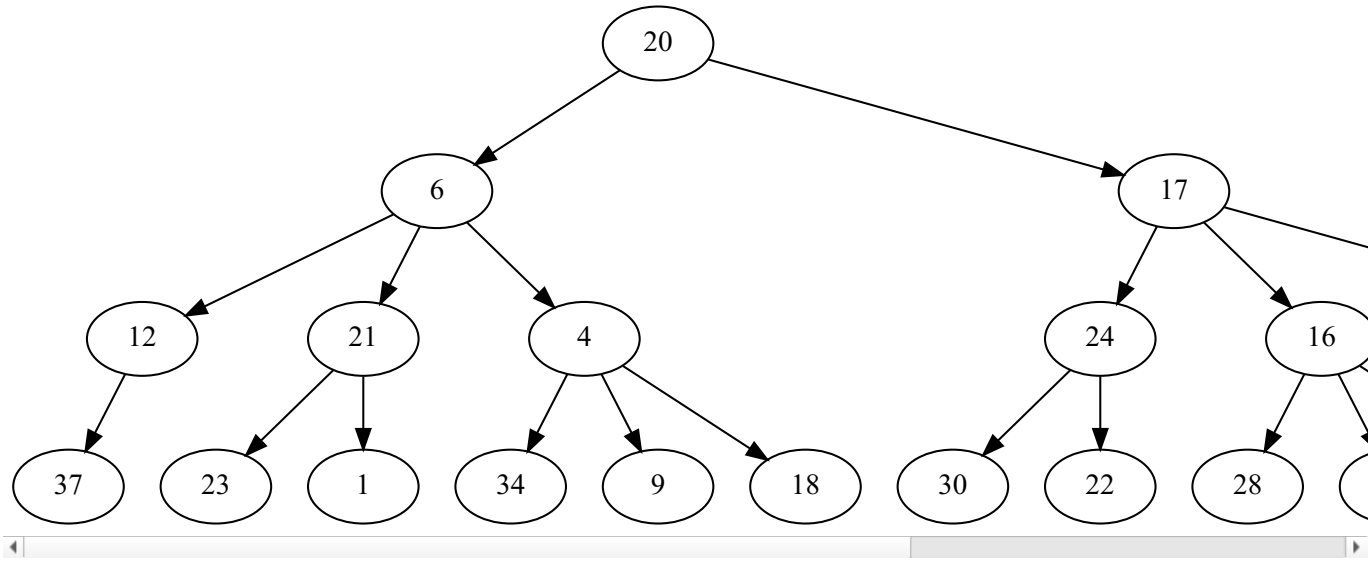
Функция принимает аргументы `tree`, `node` и список `leaves`. На первом шаге она проверяет, был ли передан список `leaves`, и, если нет, инициализирует его пустым списком. Затем, начиная с корня (если `node` не задан), функция проверяет, есть ли у текущего узла дочерние элементы. Если их нет, значение узла добавляется в список `leaves`, так как это лист. В противном случае выполняется рекурсивный вызов для каждого потомка. В результате последовательного обхода всей структуры формируется и возвращается список значений всех терминальных узлов дерева.

Пример использования:

In [ ]:
```
tree = create_tree(3)
```

```
print("Дерево:")
display(tree.visualize())
```

```
Дерево:
```



In [ ]:
```
print("Листья дерева:", find_leaves(tree))
```

```
Листья дерева: [37, 23, 1, 34, 9, 18, 30, 22, 28, 25, 15, 36, 40]
```

Рассмотренные примеры демонстрируют базовые приёмы работы с деревьями: вычисление агрегированных значений на основе структуры, условную модификацию содержимого узлов и извлечение информации о специфических элементах дерева — листьях. Во всех случаях используется рекурсивный подход, позволяющий эффективно обходить и обрабатывать иерархическую структуру. Такие операции формируют основу для более сложных алгоритмов анализа и преобразования деревьев, которые встречаются как в прикладных задачах, так и в построении алгоритмов поиска, оптимизации или компиляции.

Обзор библиотек для работы с деревьями

При работе с деревьями в Python существует множество специализированных библиотек, каждая из которых предоставляет удобные инструменты для создания, анализа и визуализации иерархических структур. Наиболее популярные из них — `anytree`, `graphviz`, `treelib` и `ete3`. Каждая имеет свою специфику и может быть полезна в определённых сценариях.

Библиотека `anytree` предоставляет лаконичный и выразительный синтаксис для создания, навигации и визуализации деревьев. Каждому узлу можно задать родителя, имя, произвольные атрибуты, а затем удобно обойти дерево в любом порядке. Основу составляет класс `Node`, который автоматически строит иерархию при указании параметра `parent`.

Рассмотрим простой пример:

In [ ]:
```
!pip install anytree
```

In [ ]:
```
from anytree import Node, RenderTree
```

In [ ]:
```
root = Node("Родитель")
child1 = Node("Ребёнок 1", parent=root)
child2 = Node("Ребёнок 2", parent=root)
grandchild = Node("Внук", parent=child1)
```

In [ ]:
```
for pre, _, node in RenderTree(root):
    print(f"{pre}{node.name}")
```

```
Родитель
├── Ребёнок 1
│   └── Внук
└── Ребёнок 2
```

В результате выполнения строится дерево из четырёх узлов, иерархически выведенное в текстовом формате.

Библиотека также поддерживает экспорт в формат Graphviz (`DotExporter`) и предоставляет множество функций обхода, например, `PreOrderIter` или `PostOrderIter`, позволяя легко реализовывать анализ дерева. Благодаря встроенной системе атрибутов, каждый узел может содержать данные, использоваться в вычислениях или фильтроваться по условиям.

Библиотека `treelib` реализует дерево через централизованное управление узлами, каждый из которых хранится в специальной таблице внутри объекта дерева. Основой является класс `Tree`, к которому можно добавлять узлы с помощью метода `create_node`, указывая имя, идентификатор и родителя. Это позволяет гибко управлять структурой, а также выполнять различные операции — от обхода до сериализации дерева.

Пример создания и отображения дерева:

In [ ]:
```
!pip install treelib
```

In [ ]:
```
from treelib import Node, Tree
```

In [ ]:
```
tree = Tree()
tree.create_node("Корень", "root")
tree.create_node("Раздел A", "a", parent="root")
tree.create_node("Раздел B", "b", parent="root")
tree.create_node("Подраздел A1", "a1", parent="a")
tree.create_node("Подраздел A2", "a2", parent="a")
tree.show()
```

```
Корень
├── Раздел A
│   ├── Подраздел A1
│   └── Подраздел A2
└── Раздел B
```

Каждому узлу можно присваивать дополнительные данные (`data`), хранить произвольную информацию и использовать её при обработке дерева. Библиотека поддерживает обход в глубину и ширину, позволяет удалять и перемещать поддеревья, копировать ветви и сохранять дерево в виде строки.

Кроме отображения, `treelib` предлагает методы `to_dict()` и `save2file()` , что делает её удобной для интеграции с внешними источниками данных или построения пользовательских интерфейсов. Благодаря строгому контролю идентификаторов и центральной таблице узлов, она хорошо подходит для задач, где важно однозначное управление структурой, например, в визуализаторах, редакторах и образовательных проектах.

Библиотека `graphviz` , хотя и не специализируется исключительно на деревьях, предоставляет мощные инструменты для построения и визуального представления любых графов, включая древовидные структуры. Основной принцип работы основан на декларативном описании узлов и связей между ними, что делает визуализацию гибкой и настраиваемой. Благодаря этому библиотеку часто используют для отладки структур данных, демонстрации и построения отчётов.

Для работы с деревьями используется ориентированный граф (`Digraph`), в который вручную добавляются узлы и рёбра. Ниже приведён пример создания простого дерева вручную:

In [ ]:
```
!pip install graphviz
```

In [ ]:
```
from graphviz import Digraph
```

In [ ]:
```
dot = Digraph()
```

Добавление узлов:

In [ ]:
```
dot.node("A", "Корень")
dot.node("B", "Дочерний 1")
dot.node("C", "Дочерний 2")
dot.node("D", "Внук")
```

Указание связей:

In [ ]:
```
dot.edge("A", "B")
dot.edge("A", "C")
dot.edge("B", "D")
```

Отображение дерева:

In [ ]:
```
dot.render("tree", format="png", cleanup=True)
dot.view()
```

Out[ ]:
```
'tree.pdf'
```

После выполнения этого кода в рабочем каталоге появляются следующие файлы:

- `tree` — исходный `.dot` -файл, содержащий описание графа в текстовом формате. Его содержимое:

```
digraph {
    A [label="Корень"]
    B [label="Дочерний 1"]
    C [label="Дочерний 2"]
    D [label="Внук"]
    A -> B
    A -> C
    B -> D
}
```

Этот файл может быть использован для повторной генерации графа или его редактирования.

- `tree.png` — изображение, автоматически сгенерированное на основе `.dot`-описания. Оно удобно для вставки в отчёты и другие документы.

3. `tree.pdf` — PDF-версия изображения, генерируется по умолчанию (при вызове `view()`).

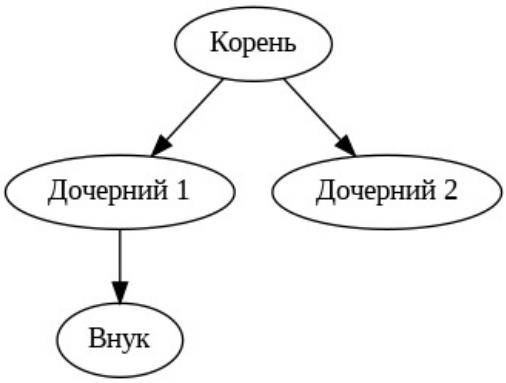


Рисунок 2 — Иерархическое представление дерева, построенного с помощью библиотеки `graphviz`

Кроме ручного добавления, `graphviz` можно использовать совместно с другими библиотеками для автоматизации построения дерева. Например, дерево можно сгенерировать программно, а затем преобразовать в `Digraph`, что было сделано выше в классе `Tree`. Благодаря поддержке разнообразных параметров оформления (`rankdir`, `shape`, `fontname`, `color`) можно тонко настроить внешний вид узлов и связей.

Для более продвинутой работы с деревьями, особенно в области биоинформатики, филогенетики и сложного анализа иерархических структур, используется библиотека `ete3`. Она предоставляет мощные средства для построения, обработки и визуализации деревьев, включая поддержку формата Newick и гибкие стили оформления. Однако важно учитывать, что полноценная графическая визуализация в `ete3` требует наличия GUI-компонентов (например, Qt), которые недоступны в среде Google Colab. В таких случаях можно использовать текстовую визуализацию или перейти к полноценной работе в локальной среде, например, в Jupyter Notebook на компьютере с установленными графическими зависимостями.

In [ ]:
```
!pip install ete3
```

Пример текстовой работы с деревом на основе строки в формате Newick:

In [ ]:
```
from ete3 import Tree
```

In [ ]:
```
newick_str = "((\"Дочерний 1\", \"Внук\"), \"Дочерний 2\")\"Корень\";"
t = Tree(newick_str, format=1)
```

In [ ]:
```
print(t.get_ascii(show_internal=True))
```

```
          /-"Дочерний 1"
       /-|
-"Корень"  \-"Внук"
      |
       \-"Дочерний 2"
```

Этот вывод позволяет увидеть иерархическую структуру дерева без графической отрисовки. Узел `«Корень»` соединяет два поддерева: первое содержит `«Дочерний 1»` и `«Внук»`, а второе — `«Дочерний 2»`. Такой формат может использоваться для отладки, быстрой проверки структуры или логического анализа без визуального интерфейса.

Если требуется построить графическое представление с сохранением изображения, например в файл `ete_tree.png`, необходимо использовать метод `render()`:

```
t.render("ete_tree.png", w=400, units="px")
```

Однако в средах без графического интерфейса эта команда вызовет ошибку. Поэтому для визуализации дерева предпочтительнее использовать библиотеку `graphviz`, которая не требует дополнительных графических компонентов и корректно работает в Google Colab и Jupyter Notebook.

Хотя на практике существует множество удобных библиотек для работы с деревьями — от простых иерархических моделей до специализированных визуализаторов и парсеров, — на начальном этапе изучения важно не полагаться на готовые решения, а самостоятельно построить структуру дерева с помощью классов. Такой подход позволяет глубже понять, как устроена иерархия узлов, каким образом формируются связи между родителями и потомками, как организуется обход и хранение данных. Это не только развивает алгоритмическое мышление, но и даёт базу для понимания более сложных структур, а именно, бинарных деревьев поиска, деревьев решений и др.

Основы бинарных деревьев

Бинарное (двоичное) дерево — это фундаментальная структура данных, представляющая собой разновидность направленного графа без циклов, в котором каждый узел имеет не более двух потомков: левого и правого. Такая структура находит широкое применение в различных областях информатики и алгоритмики благодаря своей компактности, простоте и гибкости. Бинарные деревья используются для организации поиска, упорядочивания данных, построения вычислительных выражений, реализации приоритетных очередей и других прикладных задач. Основным элементом бинарного дерева является узел, который содержит значение и ссылки на два дочерних элемента (при наличии), что позволяет формировать иерархическую структуру с чёткой направленностью.

На рис. 3 представлен пример простого бинарного дерева, в котором корневой узел соединяется с двумя подузлами, а каждый из них, в свою очередь, может иметь собственные дочерние элементы.

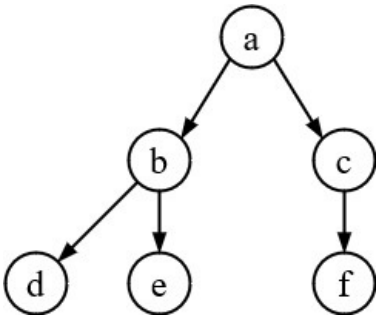


Рисунок 3 — Пример небольшого бинарного дерева

Одним из наиболее изученных и структурно определённых подклассов бинарных деревьев является **полное бинарное дерево**. В полном бинарном дереве каждый узел либо имеет ровно двух потомков, либо является листом, не имеющим потомков. Кроме того, уровни дерева заполняются строго сверху вниз, при этом последний уровень заполняется последовательно слева направо. Такая организация обеспечивает компактное размещение элементов и предсказуемую структуру дерева. Полные бинарные деревья являются удобным форматом для представления в линейных структурах данных, например, в массивах, где отсутствует необходимость в указателях, поскольку положение каждого узла и его потомков можно вычислить по индексам.

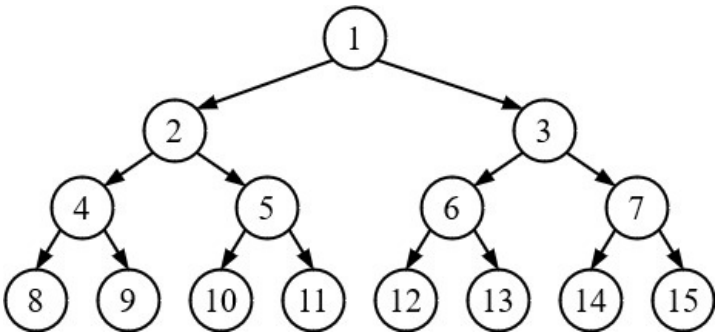


Рисунок 4 — Пример полного бинарного дерева

Формально, если высота полного бинарного дерева обозначается через h (где высота определяется как количество рёбер от корня до самого удалённого листа), то максимальное количество узлов в таком дереве вычисляется по формуле $2^{h+1} - 1$. Эта формула отражает экспоненциальный рост числа узлов с увеличением глубины дерева: на каждом последующем уровне содержится в два раза больше узлов, чем на предыдущем. Такой характер роста лежит в основе логарифмической сложности многих операций в сбалансированных или полноценных деревьях, что делает их особенно эффективными при обработке больших объёмов данных.

В некоторых источниках под полным бинарным деревом понимается структура, в которой все уровни полностью заполнены, за исключением, возможно, последнего, при этом узлы последнего уровня располагаются строго слева направо, без промежутков. Такая трактовка используется, в частности, при описании двоичной кучи — специализированной структуры для организации приоритетных очередей. При хранении такого дерева в виде массива обеспечивается постоянный доступ к потомкам и родителям каждого элемента без использования дополнительных связей, что значительно ускоряет выполнение алгоритмов вставки и удаления элементов.

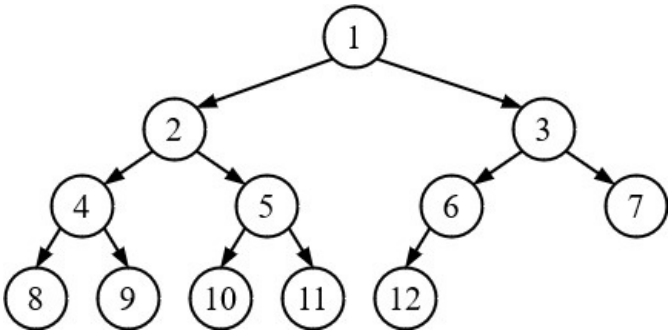


Рисунок 5 — Другой пример полного бинарного дерева

Полные бинарные деревья обладают рядом теоретических и практических достоинств. С одной стороны, они позволяют добиться высокой плотности размещения элементов в памяти и эффективно использовать ресурсы. С другой стороны, строгие ограничения на структуру дерева позволяют применять оптимизированные алгоритмы поиска и сортировки, опирающиеся на фиксированные позиции потомков и родителей. Однако стоит отметить, что поддержание полного состояния дерева при произвольных операциях вставки и удаления может быть ресурсоёмким. В таких случаях требуется применение дополнительных алгоритмов перестроения дерева для сохранения его свойств. Несмотря на это, полные бинарные деревья остаются важным инструментом при решении широкого круга задач, требующих иерархического хранения и быстрой обработки данных.

Для представления бинарного дерева может быть использован простой и наглядный способ — ***вложенные списки***. Такой подход позволяет выразить древовидную структуру в виде списка, каждый элемент которого представляет собой либо отдельное значение, либо поддерево, также оформленное в виде списка. Пустые списки в этом контексте означают отсутствие соответствующего поддерева. Это удобный способ для базового моделирования структуры дерева без использования классов или дополнительных библиотек.

In [ ]:
```
my_tree = [
    'a',  # корень
    ['b',  # левое поддерево
        ['d', [], []],
        ['e', [], []]
    ],
    ['c',  # правое поддерево
        ['f', [], []],
        []
    ]
]
```

В данной структуре `my_tree` представляет собой список, содержащий три элемента. Первый элемент `'a'` — это значение корня дерева. Второй элемент — это левое поддерево, представленное как список, где `'b'` — это значение узла, а следующие два элемента — это подсписки, представляющие левого (`'d'`) и правого (`'e'`) потомков узла `'b'`. Аналогично, третий элемент — правое поддерево с корнем `'c'`, у которого есть только один левый потомок — `'f'`, а правое поддерево отсутствует (обозначено как пустой список).

Для обращения к элементам структуры и проверки её корректности используется следующий код:

In [ ]:
```
print(my_tree)
print('Левое поддерево:', my_tree[1])
print('Корень:', my_tree[0])
print('Правое поддерево:', my_tree[2])
```

```
['a', ['b', ['d', [], []], ['e', [], []]], ['c', ['f', [], []], []]]
Левое поддерево: ['b', ['d', [], []], ['e', [], []]]
Корень: a
Правое поддерево: ['c', ['f', [], []], []]
```

Результатом выполнения этих команд будет полный вывод дерева, его левого и правого поддеревьев, а также корневого узла. Это позволяет убедиться в правильности структуры и получить доступ к нужной части дерева по индексу. Подобный подход обеспечивает простоту анализа структуры, особенно на ранних этапах знакомства с деревьями.

Рассмотрим теперь пример функции, позволяющей вставить новое поддерево слева от указанного узла:

In [ ]:
```
def insert_left(root, new_branch):
    t = root.pop(1)
    if len(t) > 1:
        root.insert(1, [new_branch, t, []])
    else:
        root.insert(1, [new_branch, [], []])
    return root
```

Данная функция принимает два аргумента: `root` — это узел, в который будет производиться вставка, и `new_branch` —

значение нового узла. Сначала из списка `root` извлекается его левое поддерево, чтобы освободить место для новой ветви. Это поддерево сохраняется во временной переменной `t`. Если `t` содержит больше одного элемента, то есть это уже существующее поддерево, оно будет вложено в качестве левого потомка нового узла. Если же `t` — пустой список, новая ветвь создаётся как самостоятельный узел без потомков. В обоих случаях новая ветвь вставляется в нужную позицию, и функция возвращает изменённое дерево.

Пример вызова функции:

In [ ]:
```
insert_left(my_tree[2], 'H')
```

Out[ ]:
```
['c', ['H', ['f', [], []], []], []]
```

В этом примере в правое поддерево корня (узел `'c'`) вставляется новая левая ветвь со значением `'H'`. Если ранее у `'c'` уже был левый потомок `'f'`, он переносится внутрь нового поддерева — становится левым дочерним элементом узла `'H'`. Таким образом, создаётся вложенная иерархия, в которой новый узел интегрируется между существующим родителем и прежним потомком, сохраняя при этом структурную целостность дерева.

Представление бинарного дерева с помощью вложенных списков может быть полезно на этапе знакомства с иерархическими структурами. Оно позволяет интуитивно понять, как формируются связи между узлами, и реализовать базовые операции вручную. Однако при переходе к более сложным задачам — обходы, поиск, модификация узлов или построение алгоритмов — предпочтительнее использовать объектно-ориентированную модель. Она обеспечивает чёткий интерфейс и расширяемость, позволяя сосредоточиться на логике работы с деревом, а не на его внутреннем представлении.

Для этого можно определить класс `BinaryTree`, обеспечивающий удобное API (интерфейс прикладного программирования) для создания и управления бинарным деревом. Он обычно включает следующие методы:

- `BinaryTree(val)` — создание нового экземпляра дерева с заданным корневым значением;
- `get_left_child()` — возвращает левое поддерево текущего узла;
- `get_right_child()` — возвращает правое поддерево текущего узла;
- `get_root_val()` — возвращает значение, хранящееся в текущем узле;
- `set_root_val(val)` — устанавливает новое значение для текущего узла;
- `insert_left(val)` — добавляет новый узел в левое поддерево;
- `insert_right(val)` — добавляет новый узел в правое поддерево.

Подобный набор методов позволяет гибко управлять структурой дерева: создавать иерархии, модифицировать значения, а также строить обходы дерева, реализующие поиск, подсчёт элементов, преобразование структуры и другие операции. Хотя реализация на базе классов требует большего объёма кода, она существенно расширяет возможности по сравнению с вложенными списками, обеспечивая высокую читаемость и удобство в работе с бинарными структурами.

Бинарное дерево может быть представлено не только в виде вложенных списков, но и с использованием структуры, состоящей из узлов и ссылок. В этом подходе каждый элемент дерева реализуется в виде отдельного объекта, содержащего данные и две ссылки — на левого и правого потомков. Такая форма организации соответствует традиционному представлению бинарных деревьев в информатике и закладывает основу для реализации множества алгоритмов на уровне узлов, что делает её удобной и наглядной при изучении рекурсивных структур.

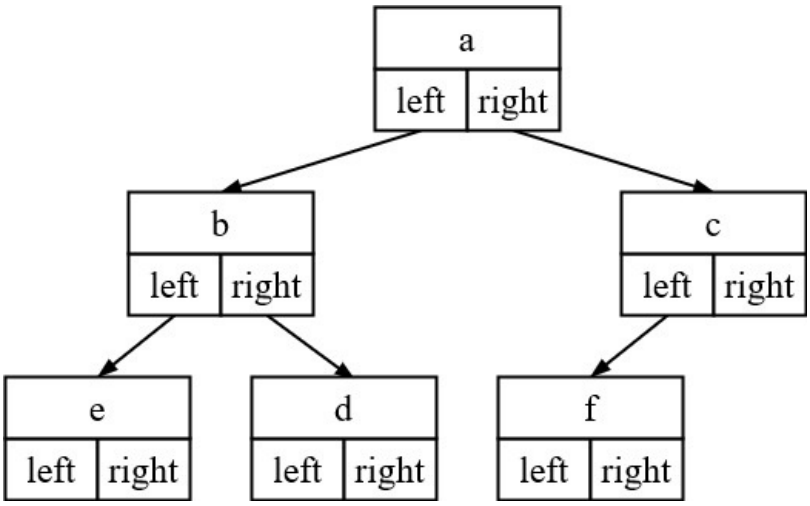


Рисунок 6 — Представление бинарного дерева в виде узлов и ссылок

На рис. 6 приведена визуализация бинарного дерева, в котором каждый прямоугольник представляет собой узел. Верхняя часть узла содержит хранимое значение, а нижняя — две области, обозначенные как `left` и `right`, соответствующие указателям на левого и правого потомков. Если у узла отсутствует один из потомков, соответствующее поле остаётся пустым.

Представление дерева в виде узлов со ссылками обладает рядом существенных преимуществ. Во-первых, оно позволяет гибко модифицировать структуру: добавлять и удалять узлы, перестраивать поддеревья без затрагивания всей иерархии. Во-вторых, чёткое разграничение между ссылками на потомков способствует повышению читаемости кода, особенно в рекурсивных алгоритмах обхода и обработки. Кроме того, использование узлов как объектов соответствует принципам объектно-ориентированного программирования, облегчая реализацию классов, методов и возможное расширение структуры, например, путём добавления ссылок на родительские элементы, меток сбалансированности или других служебных данных.

Именно такая структура лежит в основе большинства реализаций бинарных деревьев в современных языках программирования. Она используется при построении деревьев поиска, приоритетных очередей на основе куч, самобалансирующихся деревьев и других разновидностей. На этом этапе становится целесообразным перейти к описанию класса, моделирующего бинарное дерево как совокупность связанных между собой объектов, что создаёт удобную и гибкую основу для дальнейшего изучения, реализации алгоритмов и решения прикладных задач.

Ниже приведён пример реализации класса `BinaryTree`, в котором каждый узел представлен отдельным объектом, а визуализация осуществляется с помощью библиотеки `graphviz`.

In [ ]:
```
from graphviz import Digraph
```

In [ ]:
```
class BinaryTree:
    def __init__(self, root):
        self.key = root
        self.left_child = None
        self.right_child = None

    def insert_left(self, new_node):
        if self.left_child is None:
            self.left_child = BinaryTree(new_node)
        else:
            t = BinaryTree(new_node)
            t.left_child = self.left_child
            self.left_child = t

    def insert_right(self, new_node):
        if self.right_child is None:
            self.right_child = BinaryTree(new_node)
        else:
            t = BinaryTree(new_node)
            t.right_child = self.right_child
            self.right_child = t

    def get_right_child(self):
        return self.right_child

    def get_left_child(self):
        return self.left_child

    def set_root_val(self, obj):
        self.key = obj

    def get_root_val(self):
        return self.key

    def visualize(self):
        dot = Digraph()

        def add_nodes_edges(node):
            if node is None:
                return
            dot.node(str(id(node)), str(node.key))
            if node.left_child:
                dot.node(str(id(node.left_child)), str(node.left_child.key))
                dot.edge(str(id(node)), str(id(node.left_child)))
                add_nodes_edges(node.left_child)
            if node.right_child:
                dot.node(str(id(node.right_child)), str(node.right_child.key))
                dot.edge(str(id(node)), str(id(node.right_child)))
                add_nodes_edges(node.right_child)

        add_nodes_edges(self)
        return dot
```

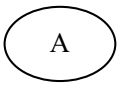
Класс `BinaryTree` включает конструктор, принимающий значение корня, методы для вставки новых узлов слева и справа, а также методы доступа к поддеревьям и значению текущего узла. При вставке нового узла проверяется наличие потомка: если

соответствующий потомок отсутствует, он создаётся на месте; если уже существует, новый узел встраивается поверх старого, который становится дочерним элементом нового узла. Метод `visualize()` строит графическую модель дерева, присваивая каждому узлу уникальный идентификатор с помощью встроенной функции `id()`, и соединяет их ориентированными рёбрами, соответствующими структуре дерева.

Для демонстрации возможностей данной реализации рассмотрим пошаговое построение дерева и визуализацию его состояния на каждом этапе:

In [ ]:
```
r = BinaryTree('A')
r.visualize()
```

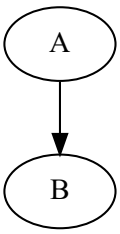
Out[ ]:



На этом этапе дерево содержит только один узел — корень со значением `'A'`.

In [ ]:
```
r.insert_left('B')
r.visualize()
```

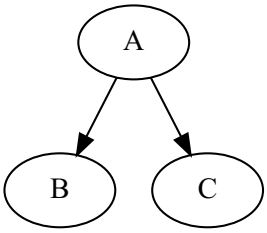
Out[ ]:



Теперь у корня появляется левый потомок `'B'`, связанный с ним ориентированным ребром.

In [ ]:
```
r.insert_right('C')
r.visualize()
```

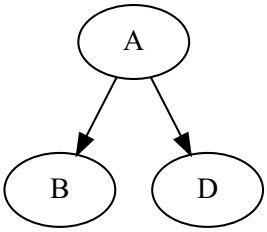
Out[ ]:



Структура дополняется правым поддеревом — к корню `'A'` добавляется правый потомок `'C'`.

In [ ]:
```
r.get_right_child().set_root_val('D')
r.visualize()
```

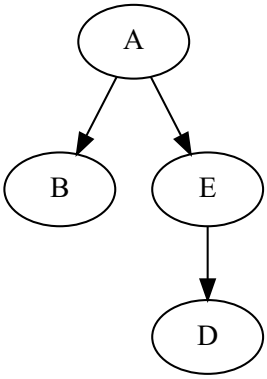
Out[ ]:



Правый узел, ранее имевший значение `'C'`, теперь переименован в `'D'`.

In [ ]:
```
r.insert_right('E')
r.visualize()
```

Out[ ]:



На этом шаге поверх узла `'D'` вставляется новый узел `'E'`, при этом `'D'` становится правым потомком `'E'`.

Поэтапная работа с деревом позволяет не только проверить корректность операций вставки и модификации, но и отследить эволюцию структуры наглядно. Таким образом, реализация через классы вместе с визуализацией обеспечивает удобный и мощный инструмент для обучения, анализа и проектирования алгоритмов, связанных с иерархическими структурами.

Бинарные деревья находят широкое применение при представлении и обработке выражений, включая арифметические, логические и символьные конструкции. Каждое такое выражение может быть организовано в виде дерева, где каждый внутренний узел соответствует операции, а листья содержат операнды. Этот способ организации отражает вложенность операций и приоритет их вычисления, а также обеспечивает возможность последовательного исполнения выражения путём обхода соответствующего дерева.

Процесс разбора выражений с использованием бинарных деревьев включает несколько последовательных этапов. Сначала выполняется ***лексический анализ***: *исходная строка преобразуется в последовательность лексем — токенов, представляющих операторы (* `+`, `−`, `*`, `/`), операнды (например, числа) и скобки. Далее выполняется ***синтаксический анализ**, в ходе которого проверяется соответствие полученной последовательности установленным правилам грамматики. На этом этапе формируется дерево разбора, отражающее вложенность и структуру выражения. Затем осуществляется* **построение бинарного дерева***, в котором каждый узел представляет операцию или операнд, а дочерние узлы соответствуют аргументам или подвыражениям. Финальным этапом является* **вычисление выражения***, производимое путём обхода дерева и последовательного применения операций в соответствии с их иерархией.

На рис. 7 показан пример бинарного дерева, соответствующего выражению `((7 + 3) × (5 − 2))`. Узлы дерева представляют собой математические операции, а листья содержат числовые значения. Такая структура полностью отражает приоритет и порядок вычислений: сначала выполняются сложение и вычитание, а затем — умножение результатов.

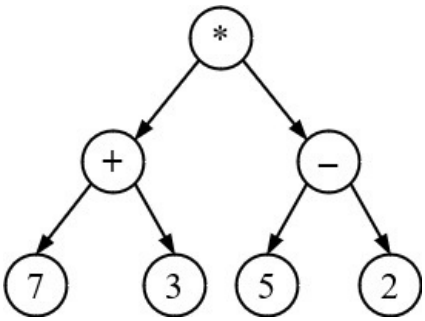


Рисунок 7 — Представление математического выражения ((7 + 3) × (5 − 2)) в виде бинарного дерева

Разбор выражения и построение соответствующего бинарного дерева может быть реализован в несколько этапов. Общий алгоритм построения дерева включает следующие шаги:

1. Если текущий токен — открывающая скобка `'('`, создаётся новый узел в качестве левого потомка текущего узла, после чего происходит переход в этот новый узел.
2. Если текущий токен представляет собой оператор (`+`, `−`, `*`, `/`), его значение присваивается текущему узлу. Затем создаётся правый потомок, и алгоритм спускается в него.
3. Если токен является числом, оно сохраняется в текущем узле в качестве значения, и происходит возврат к родительскому узлу.
4. Если токен — закрывающая скобка `')'`, также осуществляется возврат к родительскому узлу.

Подобный подход позволяет пошагово преобразовать текстовое выражение в структурированное бинарное дерево, пригодное как для визуализации, так и для последующего вычисления.

На рис. 8 показан процесс пошагового построения бинарного дерева, соответствующего арифметическому выражению `(3 + (4 * 5))`. Каждый подрисунок иллюстрирует текущее состояние дерева на определённом этапе разбора выражения. Закрашенным цветом обозначен активный узел — тот, в котором происходит изменение или к которому осуществляется переход.

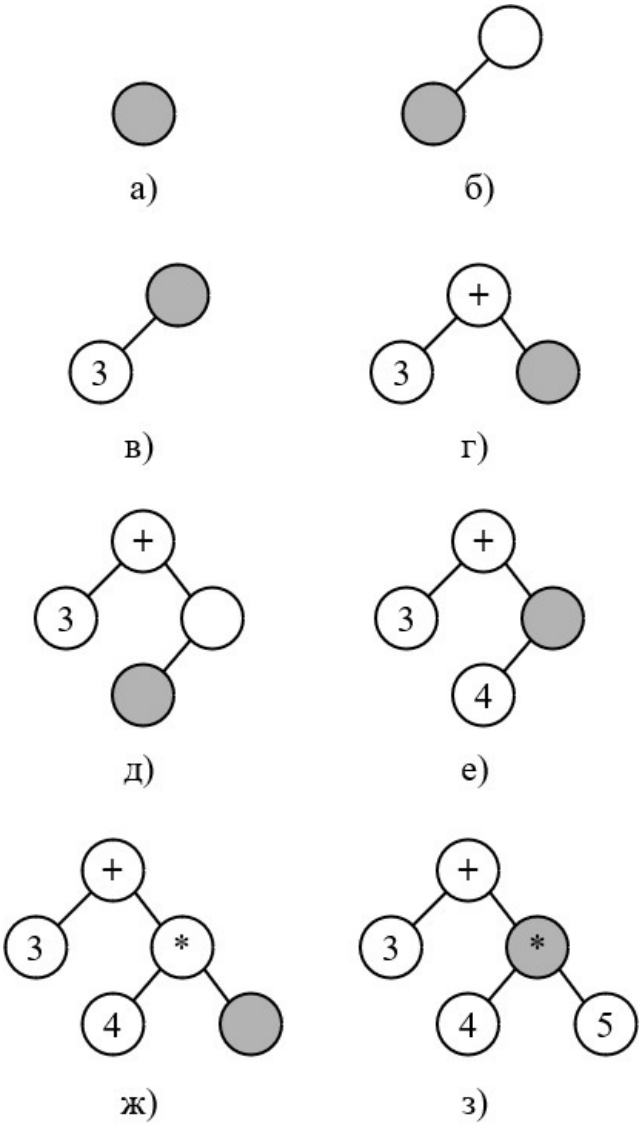


*Рисунок 8 — Разбор математического выражения `(3 + (4 5))`**

На начальном этапе создаётся корневой узел без значения. Открывающая скобка указывает на начало подвыражения, и в корне формируется левая ветвь. В неё записывается число `3`, после чего управление возвращается в родительский узел. Далее в корневой узел записывается оператор `+`, так как он объединяет левый и правый операнды выражения. Правая ветвь корня остаётся пустой, но в неё предстоит добавить следующее подвыражение. Очередная открывающая скобка в выражении инициирует создание нового поддерева, и разбор продолжается в левой ветви правого поддерева. Туда помещается число `4`, которое впоследствии будет участвовать в операции умножения. Затем в текущем узле записывается оператор `*`, объединяющий числа `4` и `5`. После этого добавляется правый потомок, в который помещается значение `5`. Завершение выражения символами `)` приводит к последовательному возвращению к родительским узлам.

В результате построения формируется корректное бинарное дерево, в котором каждая операция соответствует внутреннему узлу, а операнды — листьям. Структура дерева полностью отражает приоритет операций и вложенность подвыражений, обеспечивая тем самым основу для последующего синтаксического анализа и вычисления выражения.

Чтобы реализовать построение бинарного дерева на основе арифметического выражения, представленного в инфиксной форме, потребуется стек. Именно стек позволит организовать возврат к предыдущим узлам дерева при обработке вложенных выражений и корректно выстроить иерархию операций. Воспользуемся собственным классом стека, созданным на базе стандартной коллекции `deque` и расширенным методом `push`.

In [ ]:
```
from collections import deque
```

In [ ]:
```
class Stack(deque):
    def push(self, a):
        self.append(a)
```

Класс `Stack` наследует функциональность двусторонней очереди `deque` из модуля `collections`, предоставляя все стандартные методы, включая `pop()` для извлечения элементов с конца и `[-1]` для получения верхнего элемента стека без удаления. Метод `push()` реализует добавление нового элемента `a` в стек — фактически, это псевдоним к методу `append()`, но более ясно отражает семантику добавления на вершину. Такая обёртка делает интерфейс класса ближе к классическим абстракциям структуры данных, упрощая чтение и поддержку кода.

Этот стек будет использоваться при построении дерева, чтобы сохранить путь к родительским узлам и в нужный момент возвращаться к ним, например, при завершении подвыражения, заключённого в скобки.

Чтобы построить бинарное дерево из произвольного арифметического выражения, необходимо реализовать синтаксический разбор строки, в которой могут присутствовать числа, операторы (`+` , `-` , `*` , `/`) и скобки. Такой разбор позволит корректно интерпретировать вложенные подвыражения и приоритеты операций, формируя иерархическую структуру дерева. В каждом узле дерева будет находиться либо операция, либо числовой операнд, при этом структура дерева будет соответствовать логике вычисления.

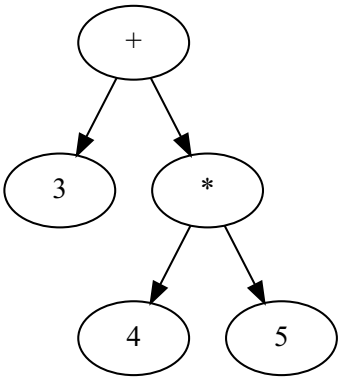
Ниже приведена реализация функции `build_expression_tree` , которая принимает на вход строку с выражением (в инфиксной записи с обязательными скобками) и возвращает бинарное дерево, соответствующее этому выражению:

In [ ]:
```python
def build_expression_tree(tokens):
    stack = Stack()
    tree = BinaryTree('')
    current = tree

    for token in tokens:
        if token == '(':
            current.insert_left('')
            stack.push(current)
            current = current.get_left_child()
        elif token in ['+', '-', '*', '/']:
            current.set_root_val(token)
            current.insert_right('')
            stack.push(current)
            current = current.get_right_child()
        elif token == ')':
            if stack:
                current = stack.pop()
        else:
            current.set_root_val(token)
            if stack:
                current = stack.pop()
    return tree
```

Данная функция строит бинарное дерево на основе списка токенов, полученных из инфиксной записи арифметического выражения с использованием скобок. Сначала создаётся корневой узел дерева с пустым значением и стек для хранения родительских узлов, чтобы можно было возвращаться на уровень вверх при завершении вложенного выражения. При встрече открывающей скобки создаётся новый левый потомок текущего узла, ссылка на текущий узел помещается в стек, и дальнейшая обработка продолжается на уровне этого левого потомка.

Если токен является оператором (`+` , `-` , `*` , `/`), он устанавливается как значение текущего узла, затем создаётся правый потомок и текущий узел снова помещается в стек, чтобы продолжить обработку уже в правом поддереве. При встрече закрывающей скобки происходит выход из текущего контекста: извлекается родительский узел из стека, и обход продолжается уже на этом уровне. Если токен — число (или переменная), оно присваивается текущему узлу, после чего также выполняется переход к родителю.

In [ ]:
```python
expression = "( 3 + ( 4 * 5 ) )"
tokens = expression.strip().split()
tree = build_expression_tree(tokens)
tree.visualize()
```

Out[ ]:


Таким образом, на выходе формируется дерево, в котором каждый оператор занимает внутренний узел, а операнды — листья, что идеально подходит для последующего вычисления выражения или анализа его структуры.

Теперь можно реализовать функцию, которая обходит дерево и, в зависимости от типа узла, либо выполняет арифметическую операцию, либо возвращает числовое значение.

In [ ]:
```python
def evaluate_expression(node):
    if node is None:
```

```
        return 0

    left = node.get_left_child()
    right = node.get_right_child()

    if left is None and right is None:
        return int(node.get_root_val())

    left_value = evaluate_expression(left)
    right_value = evaluate_expression(right)
    operator = node.get_root_val()

    if operator == '+':
        return left_value + right_value
    elif operator == '-':
        return left_value - right_value
    elif operator == '*':
        return left_value * right_value
    elif operator == '/':
        return left_value / right_value
```

Функция `evaluate_expression` на вход принимает корневой узел дерева. Если текущий узел является листом (то есть не имеет потомков), то он содержит числовое значение, которое преобразуется в целое число и возвращается. Если узел внутренний, то сначала происходит рекурсивный вызов функции для левого и правого поддерева, в результате чего вычисляются значения соответствующих подвыражений.

Далее в зависимости от значения в текущем узле (предполагается, что это оператор) выполняется соответствующая арифметическая операция над результатами, полученными от левого и правого поддеревьев. Обход дерева осуществляется в постфиксном порядке, и на каждом уровне возвращается результат подвыражения, пока не будет получено итоговое значение всего выражения в корне дерева.

In [ ]:
```
expression = "( 3 + ( 4 * 5 ) )"
tokens = expression.strip().split()
tree = build_expression_tree(tokens)
print(f"Результат выражения: {evaluate_expression(tree)}")
```

```
Результат выражения: 23
```

Данный пример иллюстрирует ключевую роль бинарного дерева при разборе и вычислении арифметических выражений. Каждое выражение преобразуется в строгую иерархическую структуру, где операторы размещаются во внутренних узлах, а операнды — в листьях дерева. Благодаря бинарной природе, каждый узел имеет не более двух потомков, что соответствует бинарным операциям в математике. Построение дерева на основе инфиксной записи через алгоритм синтаксического разбора и использование рекурсивной функции для вычисления значений демонстрирует, насколько эффективно бинарное дерево может использоваться для хранения, анализа и обработки выражений.

Структура документа также может быть наглядно представлена в виде бинарного дерева, где каждый узел отображает один из логических уровней иерархии: от книги в целом до отдельных подразделов. Подобная модель оказывается полезной для представления сложных, иерархически организованных текстов, например, технических руководств, учебников и нормативных документов. На рис. 9 показан пример, в котором корневой узел обозначает всю книгу, узлы первого уровня соответствуют главам, второго уровня — разделам внутри этих глав, а третьего уровня — подразделам.

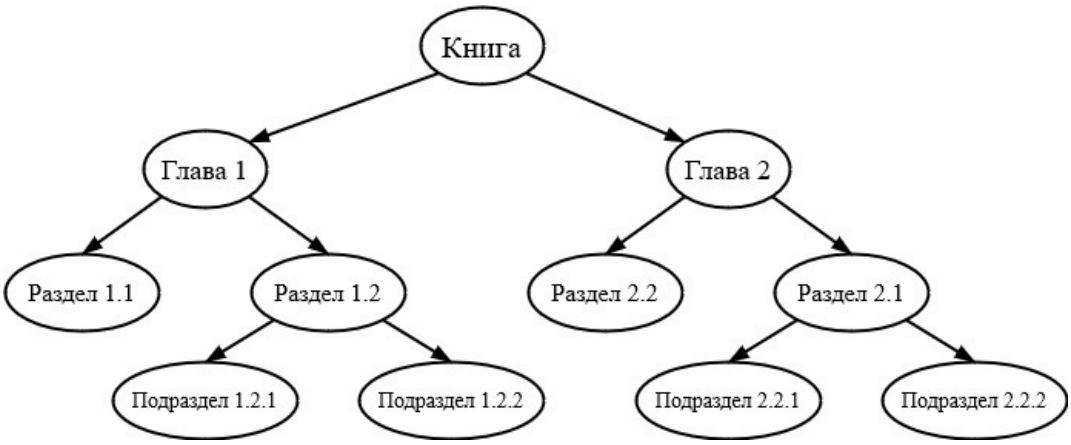


Рисунок 9 — Представление документа (книги) как дерева

Такое структурное представление обеспечивает удобную навигацию и поддержку логических операций: от поиска и перестройки содержимого до автоматической генерации оглавления. Кроме того, древовидная модель способствует стандартизации построения документов и может быть использована в системах управления контентом, электронных библиотеках и генераторах документации. Возможность рекурсивной обработки узлов позволяет реализовать такие функции, как экспорт конкретного раздела, подсчёт вложенных элементов или динамическое обновление структуры при редактировании текста.

Алгоритмы обхода бинарных деревьев

Обход бинарного дерева в **прямом порядке** — это один из базовых способов рекурсивного обхода, при котором порядок действий фиксирован: сначала посещается текущий узел, затем рекурсивно обходится левое поддерево, и только после этого — правое.

Такая стратегия подчёркивает значимость корневых элементов, позволяя уже на первом этапе получить основную информацию, а затем перейти к деталям. Этот подход особенно полезен в ситуациях, когда необходимо сначала обработать или зафиксировать структуру узла (например, при копировании дерева, сериализации или построении выражений), а уже потом переходить к его потомкам.

Рассмотрим на конкретном примере, изображённом на рис. 10.

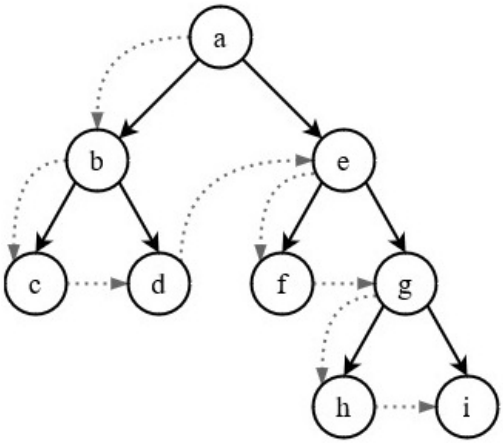


Рисунок 10 — Пример прямого обхода бинарного дерева

Структура дерева начинается с корневого узла `a`. Именно он будет первым в списке посещённых узлов. Далее алгоритм переходит к левому поддереву корня, где первым оказывается узел `b`, а затем его левый потомок `c`. Поскольку у `c` нет детей, дальнейшее продвижение в этом направлении невозможно. Алгоритм возвращается к `b` и переходит в его правое поддерево — к узлу `d`, который также является листом. Таким образом, после обхода левой ветви дерева порядок посещения: `a`, `b`, `c`, `d`.

Следующий этап — переход в правое поддерево корня `a`. Здесь сначала обрабатывается узел `e`, после чего алгоритм направляется в его левую ветвь, где находится узел `f`, не имеющий потомков. После возврата к `e` начинается обработка правого поддерева — узел `g` становится следующим в списке. У `g` есть два потомка: сначала посещается левый `h`, затем правый `i`. Каждый из них — лист, обход которых завершается сразу после посещения. Таким образом, завершающая часть последовательности включает `e`, `f`, `g`, `h`, `i`.

Итоговая последовательность обхода, составленная согласно правилам прямого порядка, будет следующей: `a`, `b`, `c`, `d`, `e`, `f`, `g`, `h`, `i`. Такой способ обходит дерево сверху вниз и слева направо, соблюдая приоритет корня перед поддеревьями. Он отражает порядок, в котором элементы могут быть сохранены или обработаны при предварительном анализе структуры, а также используется при генерации копий дерева или создании его текстовых представлений.

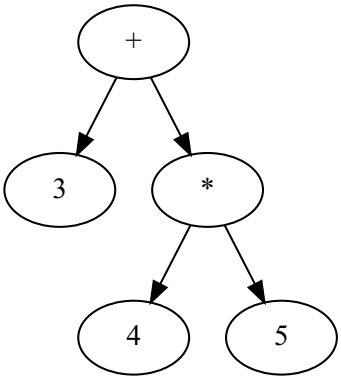
В качестве примера рассмотрим дерево, сформированное на основе арифметического выражения с помощью ранее определённой функции `build_expression_tree`. Оно идеально подходит для демонстрации принципа обхода, поскольку в нём чётко различимы корень, поддеревья и структура иерархии.

Пусть требуется собрать выражение в префиксной форме (то есть в виде, соответствующем порядку прямого обхода), при котором каждый оператор предшествует своим операндам. Такая форма полезна, например, для интерпретаторов, компиляторов или систем символьных вычислений, где важно сохранить порядок операций без использования скобок.

Вначале создадим бинарное дерево:

In [ ]:
```
expression = "( 3 + ( 4 * 5 ) )"
tokens = expression.strip().split()
tree = build_expression_tree(tokens)
tree.visualize()
```

Out[ ]:



Теперь реализуем функцию прямого обхода дерева, возвращающую выражение в префиксной записи:

In [ ]:
```python
def preorder_traversal(node):
    if node is None:
        return []
    result = [str(node.get_root_val())]
    result += preorder_traversal(node.get_left_child())
    result += preorder_traversal(node.get_right_child())
    return result
```

Функция `preorder_traversal` принимает на вход узел дерева и выполняет рекурсивный обход: сначала добавляется значение текущего узла, затем осуществляется обход левого поддерева, а затем правого. Возвращается список строк, отражающих порядок обхода.

Применим функцию к ранее построенному дереву:

In [ ]:
```python
prefix = preorder_traversal(tree)
print(f"Префиксная запись выражения:", " ".join(prefix))
```

```
Префиксная запись выражения: + 3 * 4 5
```

Таким образом, прямой обход бинарного дерева обеспечивает последовательный доступ к узлам в том порядке, в котором они определяют структуру выражения или иерархии: сначала обрабатывается текущий узел, затем левое поддерево, и лишь после этого — правое. Такой порядок полезен в ситуациях, где необходимо сохранить логическую структуру построения, например, при сериализации дерева, построении префиксной формы выражений или генерации шаблонов вывода, поскольку он позволяет точно восстановить исходную структуру без дополнительной информации о скобках или глубине вложенности.

Обратный обход бинарного дерева представляет собой последовательный рекурсивный обход, при котором сначала обрабатывается левое поддерево, затем правое, и только после этого — сам узел. Такая стратегия отражает принцип «снизу вверх»: все дочерние элементы анализируются прежде, чем родительский. Этот порядок полезен в тех случаях, когда требуется сначала обработать составляющие структуры, а затем — агрегирующий их узел.

На рис. 11 показан пример бинарного дерева, для которого используется обратный порядок обхода.

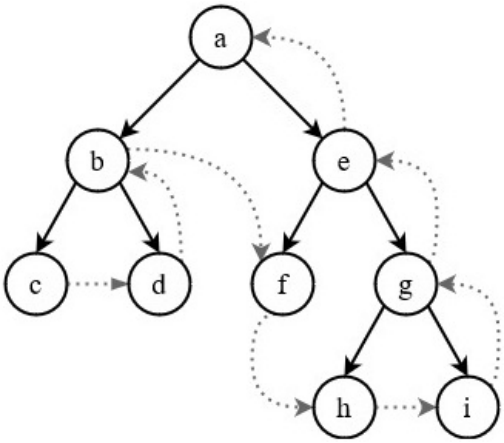


Рисунок 11 — Обратный порядок обхода бинарного дерева

Обход начинается с левого поддерева корня `'a'`, переходя к узлу `'b'`, откуда спускается к его левому потомку `'c'`, который будучи листом, немедленно обрабатывается. Затем происходит возврат к `'b'` и переход к правому потомку `'d'`, который также не имеет поддеревьев и обрабатывается сразу. После этого обрабатывается узел `'b'`, завершив тем самым обход его поддерева.

Следом переходим к правому поддереву корня `'a'` — узлу `'e'`. Сначала рассматривается его левый потомок `'f'`, не имеющий потомков, и он сразу обрабатывается. Затем переходим к узлу `'g'`, вначале спускаясь к его левому потомку `'h'`, затем к правому — `'i'`, после чего только сам `'g'` может быть обработан. Завершив обход обоих поддеревьев узла `'e'`,

переходим к его обработке. И лишь после обхода всей структуры завершается посещением корневого узла `'a'`. Таким образом, в результате обратного обхода узлы посещаются в следующем порядке: `c`, `d`, `b`, `f`, `h`, `i`, `g`, `e`, `a`.

Чтобы продемонстрировать работу алгоритма обратного обхода, реализуем функцию `post_order_traversal`, которая обходит дерево и возвращает список значений узлов в порядке их обработки:

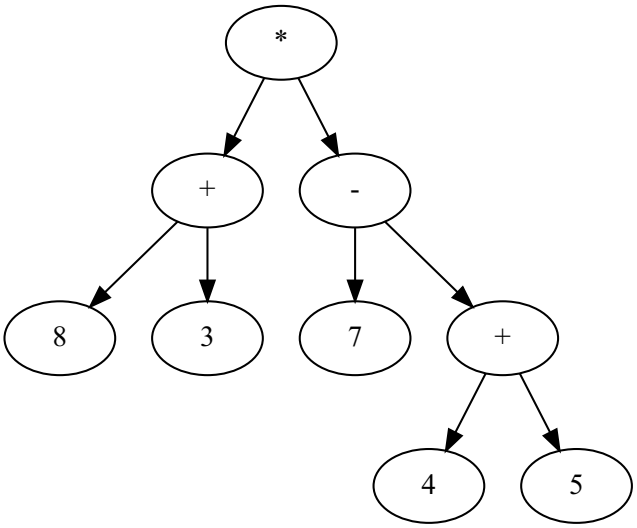
In [ ]:
```
def post_order_traversal(node, result=None):
    if result is None:
        result = []
    if node is not None:
        post_order_traversal(node.get_left_child(), result)
        post_order_traversal(node.get_right_child(), result)
        result.append(node.get_root_val())
    return result
```

Функция принимает два аргумента: сам узел дерева и список `result`, в который накапливаются значения. Если узел отсутствует (равен `None`), дальнейшее выполнение прекращается. В противном случае рекурсивно вызывается сама функция для левого и правого потомков текущего узла, тем самым достигается глубина поддеревьев. Только после завершения этих двух вызовов к списку `result` добавляется значение текущего узла. Такой порядок действий гарантирует, что ни один родительский узел не будет обработан до тех пор, пока не будут полностью пройдены его поддеревья.

Рассмотрим пример использования:

In [ ]:
```
expression = "( ( 8 + 3 ) * ( 7 - ( 4 + 5 ) ) )"
tokens = expression.strip().split()
tree = build_expression_tree(tokens)
tree.visualize()
```

Out[ ]:



In [ ]:
```
order = post_order_traversal(tree)
print("Результат обратного обхода:", order)
```

```
Результат обратного обхода: ['8', '3', '+', '7', '4', '5', '+', '-', '*']
```

Такои образом, обратный обход бинарного дерева представляет собой универсальный механизм рекурсивной обработки иерархических структур, при котором каждый узел обрабатывается лишь после того, как завершена работа с его левым и правым поддеревьями. Это позволяет последовательно спускаться от корня к самым глубоким уровням дерева и лишь затем подниматься обратно, агрегируя или анализируя информацию снизу вверх. Обратный обход может быть полезным в задачах, где результат в узле зависит от результатов в его потомках, например, при вычислении выражений, рекурсивной очистке памяти, удалении узлов или построении сводной информации по поддеревьям.

Симметричный порядок обхода бинарного дерева предполагает строго определённую последовательность действий: сначала выполняется рекурсивный обход левого поддерева, затем обрабатывается текущий узел, и лишь после этого начинается обход правого поддерева. Такой порядок обеспечивает естественное движение по дереву — слева направо, — и позволяет проходить все элементы в логической последовательности, определяемой структурой дерева. Независимо от значений в узлах, симметричный обход всегда проходит через левую ветвь до самого конца, затем обрабатывает вершины на пути возврата и переходит к правым поддеревьям, тем самым охватывая всю структуру дерева без пропусков и повторов.

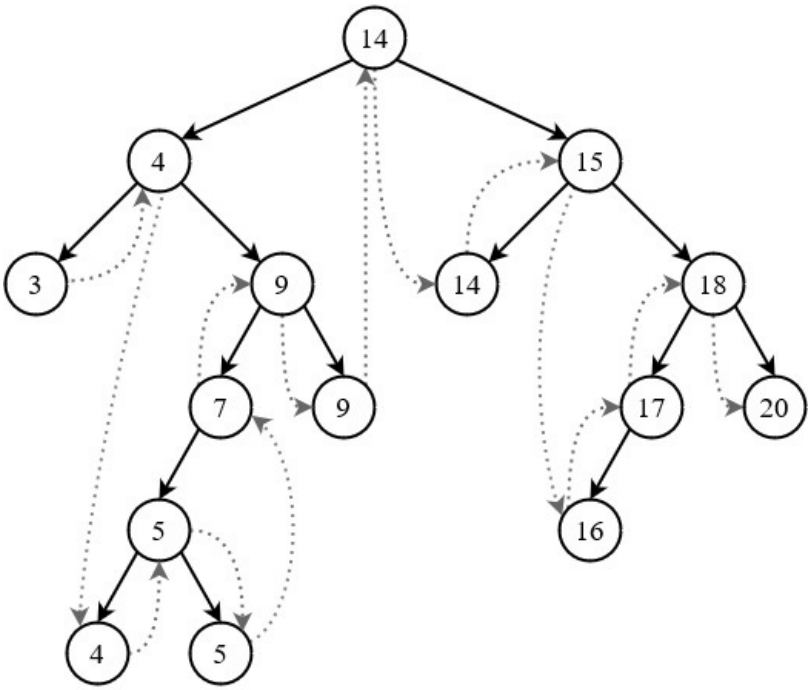


Рисунок 12 — Симметричный порядок обхода бинарного дерева

При симметричном обходе бинарного дерева, представленного на рис. 12, первым шагом является спуск в самое левое поддерево. Это поддерево заканчивается листом со значением `3`, который и становится первым узлом, попавшим в результирующую последовательность. После обработки этого узла происходит возврат к родительскому элементу со значением `4`, который также включается в результат. Далее следует переход к правому поддереву этого узла, представленному более глубокой и разветвлённой структурой.

Внутри правого поддерева происходит спуск к левому поддереву узла `7`, в котором находится узел `5`. Этот узел, в свою очередь, имеет собственное левое поддерево (узел `4`), которое обрабатывается первым, затем сам узел `5`, а затем его правое поддерево — ещё один узел со значением `5`. После завершения обработки всего поддерева узла `5` происходит возврат к узлу `7` и его включение в результат. Затем обрабатывается узел `9`, после чего следует переход к его правому поддереву — также узлу `9`.

Завершив обход левой части дерева, алгоритм возвращается к корню дерева — узлу `14` — и выполняет его обработку. Затем переходит к правому поддереву, начинающемуся с узла `15`. Перед этим обрабатывается левое поддерево узла `15` — узел со значением `14`. После этого сам узел `15` добавляется к результату. Далее осуществляется переход к правому поддереву узла `15`, корнем которого является узел `18`.

В поддереве узла `18` сначала обрабатывается левое поддерево — узел `17`, а затем — его правое поддерево, представленное узлом `16`. После завершения обработки этих двух узлов выполняется переход к самому узлу `18`. Последним в обходе оказывается правое поддерево узла `18`, представленное узлом `20`.

Таким образом, результат симметричного обхода всего дерева: `3`, `4`, `4`, `5`, `5`, `7`, `9`, `9`, `14`, `14`, `15`, `16`, `17`, `18`, `20`.

Симметричный порядок обхода важен, например, при работе с бинарными деревьями поиска, так как он возвращает элементы в отсортированном порядке. При этом сохраняется чёткая логика обработки: всё, что «слева» по смыслу, проходит раньше, чем сам узел, а всё, что «справа», — позже. Такой механизм лежит в основе множества алгоритмов, связанных с анализом и упорядочиванием иерархических данных.

Для демонстрации симметричного обхода решим задачу, в которой необходимо последовательно обойти все узлы бинарного дерева и собрать их значения в список. Сначала построим дерево фиксированной структуры. Для этого задаём словарь `structure`, в котором каждому ключу сопоставляется кортеж из двух элементов — значений левого и правого поддеревьев.

In [ ]:
```
structure = {
    14: (9, 15),
    9: (4, 10),
    4: (3, 7),
    7: (5, None),
    5: (6, 8),
    15: (13, 18),
    18: (17, 20),
    17: (16, None)
    }
```

Далее воспользуемся рекурсивной функцией `build_tree`, принимающая на вход значение корневого узла и карту связей, и

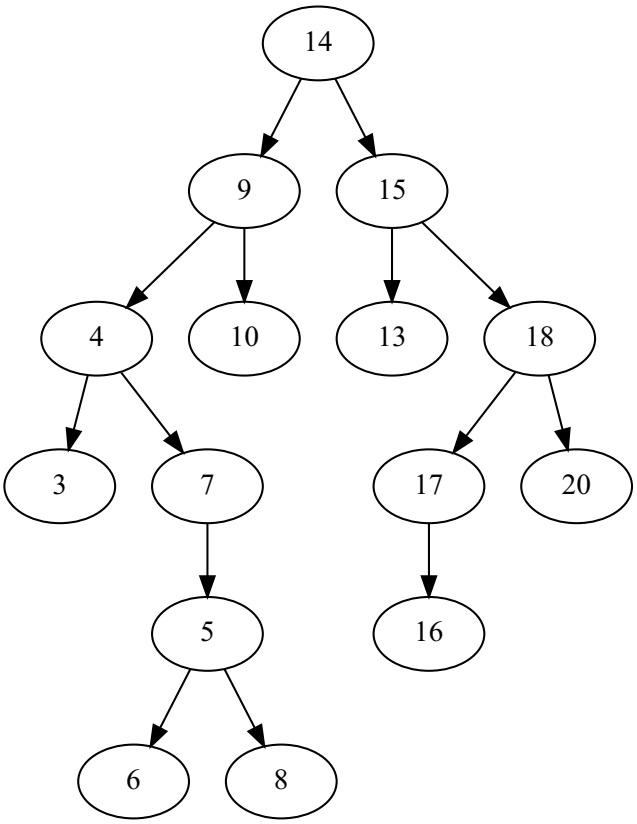
возвращающая дерево, сформированное по данной схеме:

In [ ]:
```
def build_tree(value, mapping):
    if value is None:
        return None
    node = BinaryTree(value)
    left_val, right_val = mapping.get(value, (None, None))
    node.left_child = build_tree(left_val, mapping)
    node.right_child = build_tree(right_val, mapping)
    return node
```

Создаём бинарное дерево:

In [ ]:
```
tree = build_tree(14, structure)
tree.visualize()
```

Out[ ]:



После построения дерева можно выполнить симметричный обход, в ходе которого сначала посещаются все узлы левого поддерева, затем текущий узел, и в завершение — узлы правого поддерева.

In [ ]:
```
def inorder_traversal(node, result=None):
    if result is None:
        result = []
    if node is not None:
        inorder_traversal(node.get_left_child(), result)
        result.append(node.get_root_val())
        inorder_traversal(node.get_right_child(), result)
    return result
```

В теле функции сначала проверяется, передан ли результирующий список `result`, и при необходимости он инициализируется пустым. Далее, если текущий узел `node` не является пустым, выполняется рекурсивный вызов той же функции для левого потомка, находящегося в поддереве `node.get_left_child()`. Этот вызов продолжается до тех пор, пока не будет достигнут самый левый (и потому самый глубокий) узел дерева. После этого, при возвращении из рекурсии, значение текущего узла добавляется в список `result` с помощью `append`, фиксируя его на нужной позиции в порядке обхода. Затем рекурсивный вызов совершается для правого поддерева текущего узла.

In [ ]:
```
print("Результат симметричного обхода:", inorder_traversal(tree))
```

```
Результат симметричного обхода: [3, 4, 6, 5, 8, 7, 9, 10, 14, 13, 15, 16, 17, 18, 20]
```

Такой механизм позволяет получить список всех значений, хранящихся в дереве, в порядке, при котором сначала выводятся все элементы левого поддерева, затем значение текущего узла, и далее — правое поддерево. Благодаря этой логике обхода можно, например, отследить внутреннюю симметрию дерева или использовать полученный список для дальнейших вычислений и визуального анализа структуры.

Типы порядка обхода бинарного дерева отражают разные стратегии прохождения по структуре дерева и позволяют решать широкий круг задач, связанных с обработкой и анализом иерархических данных. Каждый из них задаёт собственную

последовательность посещения узлов, определяя, в какой момент обрабатывается текущий узел относительно его поддеревьев.

Прямой обход используется, когда необходимо сначала зафиксировать текущую вершину, а затем переходить к её потомкам — он подходит, например, для копирования структуры дерева или сериализации. Симметричный обход полезен при работе с бинарными деревьями поиска, так как позволяет извлекать значения в отсортированном порядке. Обратный обход, в свою очередь, сначала обходит поддеревья, а затем обрабатывает текущий узел, что делает его удобным в задачах, где требуется сначала собрать данные из подчинённых элементов, а затем перейти к объединению или итоговой операции.

Понимание этих типов порядка обхода критически важно для построения алгоритмов, связанных с деревьями, в том числе при парсинге выражений, построении синтаксических деревьев, оптимизации и структурировании информации.

Бинарное дерево поиска

Бинарное дерево поиска представляет собой разновидность бинарного дерева, обладающую важным свойством упорядоченности: для каждого узла значения ключей в левом поддереве строго меньше значения ключа самого узла, а значения в правом поддереве — больше либо равны. Эта структура позволяет эффективно реализовать базовые операции, например, добавление нового значения, удаление узла или поиск элемента, при условии соблюдения правил построения. Поскольку все сравнения производятся по ключам, данные в узлах должны поддерживать операцию сравнения.

На рис. 13 показан пример бинарного дерева поиска, в котором каждый узел подчиняется указанному правилу: в левом поддереве содержатся меньшие значения, в правом — большие или равные. Корневой узел имеет значение `70`. Его левый потомок — `31`, меньший `70`; правый потомок — `93`, больший `70`. У узла `31` в левом поддереве находится `14`, а у `14` — узел `23`, также соблюдающий правило. Аналогично, в правом поддереве от `93` расположены `73` и `94`, и оба они удовлетворяют свойству бинарного дерева поиска. Такая структура формирует иерархию, в которой каждый уровень организован по принципу сравнений, что облегчает навигацию и упрощает реализацию алгоритмов.

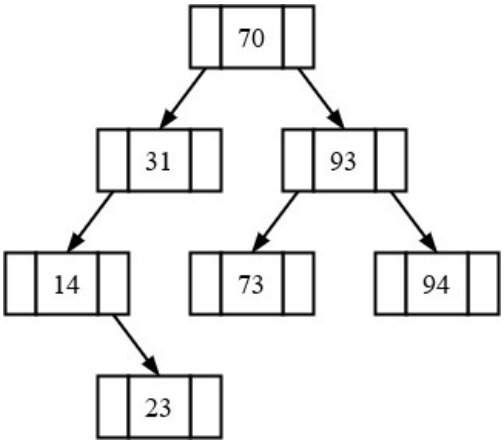


Рисунок 13 — Пример бинарного дерева поиска

Время выполнения операций в бинарном дереве поиска зависит от его высоты. Если структура дерева сбалансирована по высоте, то она близка к полной, и её высота составляет величину порядка logn, где n — число узлов. В этом случае все действия с элементами выполняются за время O(logn), что делает бинарное дерево поиска эффективной структурой для организации упорядоченных коллекций.

При случайной последовательности вставок математическое ожидание высоты также приближается к O(logn), однако на практике это условие выполняется не всегда. Последовательная вставка отсортированных данных может привести к вырождению дерева в линейную структуру, где высота достигает n, и эффективность операций снижается до O(n). Такой сценарий иллюстрирует важность контроля формы дерева: при неудачном порядке вставки даже правильно реализованная структура теряет свои преимущества.

Для предотвращения подобных случаев используются сбалансированные структуры, поддерживающие ограниченную высоту независимо от порядка вставки. В их числе можно назвать AVL-деревья, 2-3-деревья, 2-3-4-деревья и красно-чёрные деревья. Каждая из них реализует собственную стратегию балансировки: например, AVL-деревья используют разность высот поддеревьев и перестроения после каждой модификации; красно-чёрные деревья вводят дополнительную цветовую маркировку и логические ограничения, обеспечивая сбалансированность при добавлении и удалении узлов.

Эти структуры обеспечивают стабильное выполнение операций даже в наихудшем случае за время O(logn), сохраняя компактную и предсказуемую форму дерева. Благодаря этому бинарные деревья поиска и их сбалансированные версии находят применение в базах данных, индексных структурах, компиляторах, планировании задач и других областях, где важны быстрота поиска, вставки и удаления, а также сохранение упорядоченного представления информации.

Для построения бинарного дерева поиска с заранее обеспеченной сбалансированностью используется метод, основанный на предварительной сортировке входных данных и последовательном добавлении элементов из середины диапазона. Такой

подход гарантирует, что дерево будет иметь минимально возможную высоту при заданном множестве ключей. В отличие от последовательной вставки, при которой форма дерева может зависеть от порядка элементов и приводить к вырожденной структуре, построение из отсортированного списка обеспечивает равномерное распределение узлов и сбалансированное ветвление.

Рассмотрим пример реализации бинарного дерева поиска с визуализацией структуры и функцией построения сбалансированного дерева из произвольного множества значений. Для графического отображения дерева используется библиотека `graphviz`, позволяющая отрисовывать ориентированные графы в иерархическом виде.

In [ ]:
```
from graphviz import Digraph
from random import sample
```

In [ ]:
```
class BSTNode:
    def __init__(self, key):
        self.key = key
        self.left = None
        self.right = None

class BSTree:
    def __init__(self):
        self.root = None

    def insert(self, key):
        def _insert(node, key):
            if node is None:
                return BSTNode(key)
            if key < node.key:
                node.left = _insert(node.left, key)
            else:
                node.right = _insert(node.right, key)
            return node

        self.root = _insert(self.root, key)

    def visualize(self):
        dot = Digraph()

        def add_nodes_edges(node):
            if node is None:
                return
            dot.node(str(id(node)), str(node.key))
            if node.left:
                dot.node(str(id(node.left)), str(node.left.key))
                dot.edge(str(id(node)), str(id(node.left)))
                add_nodes_edges(node.left)
            if node.right:
                dot.node(str(id(node.right)), str(node.right.key))
                dot.edge(str(id(node)), str(id(node.right)))
                add_nodes_edges(node.right)

        if self.root:
            add_nodes_edges(self.root)
        return dot
```

Класс `BSTNode` описывает отдельный узел дерева. Каждый узел содержит ключ (`key`) и ссылки на левого и правого потомков, которые инициализируются значением `None`.

Класс `BSTree` инкапсулирует структуру дерева. Метод `insert` реализует добавление нового элемента, обеспечивая сохранение свойства бинарного дерева поиска: все элементы в левом поддереве меньше текущего узла, а в правом — больше или равны ему. Добавление осуществляется рекурсивно, начиная с корня дерева. Метод `visualize` позволяет отобразить дерево в графическом виде: для каждого узла создаётся вершина, а для каждой связи между родителем и потомком — ориентированное ребро.

Для создания сбалансированного дерева реализуется отдельная функция `build_balanced_bst`, которая принимает на вход произвольный набор значений и возвращает дерево, построенное по правилам бинарного поиска. Ключевая идея состоит в сортировке входного множества и выборе среднего элемента в качестве корня, после чего процедура рекурсивно применяется к левой и правой части списка.

In [ ]:
```
def build_balanced_bst(elements):
    tree = BSTree()
    elements = sorted(set(elements))

    def _build_balanced(lst):
        if not lst:
            return None
        mid = len(lst) // 2
        node = BSTNode(lst[mid])
```

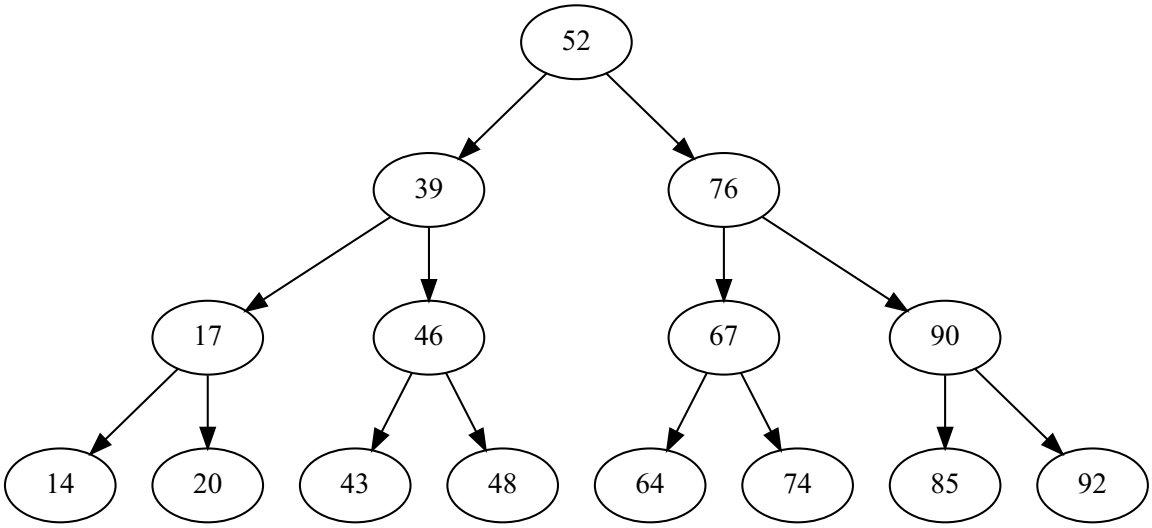
```
        node.left = _build_balanced(lst[:mid])
        node.right = _build_balanced(lst[mid+1:])
        return node

    tree.root = _build_balanced(elements)
    return tree
```

Функция `build_balanced_bst` начинается с удаления повторяющихся значений и сортировки оставшихся элементов. Далее вызывается вложенная функция `_build_balanced`, которая на каждом этапе выбирает средний элемент текущего списка в качестве корневого узла. Элементы слева формируют левое поддерево, элементы справа — правое. Рекурсивный характер построения обеспечивает симметрию и равномерную глубину дерева. В результате получается структура, минимально возможная по высоте, с сохранением всех свойств бинарного дерева поиска.

Пример использования функции:

In [ ]:
```
values = sample(range(10, 100), 15)
tree = build_balanced_bst(values)
tree.visualize()
```

Out[ ]:



Здесь из случайно выбранных пятнадцати чисел строится сбалансированное бинарное дерево поиска, которое можно отобразить с помощью метода `visualize`. Благодаря предварительной сортировке и выбору центральных элементов структура дерева будет сбалансированной, то есть высота дерева окажется близкой к O(logn). Это важно при работе с большими объемами данных, где глубина дерева напрямую влияет на скорость выполнения операций поиска, вставки и удаления.

Операция **вставки** в бинарное дерево поиска основывается на сравнении ключа нового элемента с ключами существующих узлов. При этом сохраняется основное свойство бинарного дерева поиска — для любого узла все элементы левого поддерева меньше его значения, а правого поддерева — больше или равны. Алгоритм вставки включает последовательный спуск от корня к листьям до нахождения пустого места, в которое можно поместить новый узел, не нарушая структуру дерева.

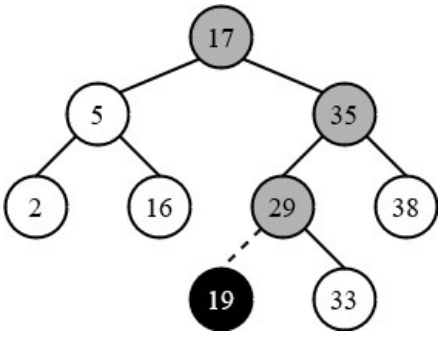


Рисунок 14 — Пример вставки узла в бинарное дерево поиска

На рис. 14 представлен процесс добавления значения `19`. Сравнение начинается с корня дерева — значения `17`. Поскольку `19` больше `17`, поиск продолжается в правом поддереве. Следующий узел — `35`, и `19` оказывается меньше `35`, поэтому переход осуществляется в его левое поддерево. Далее значение 19 сравнивается с `29`, и, снова будучи меньшим, направляется в левое поддерево `29`, которое является пустым. Именно в это место и добавляется новый узел со значением `19`. Таким образом, структура дерева сохраняет свойства бинарного поиска, а добавление завершается после первого обнаруженного пустого указателя.

Описанная логика реализуется в методе `insert`, включённом в класс `BSTree`. При вызове этого метода дерево обрабатывает ключ по указанной схеме и добавляет новый узел в соответствующее место. Приведённая реализация

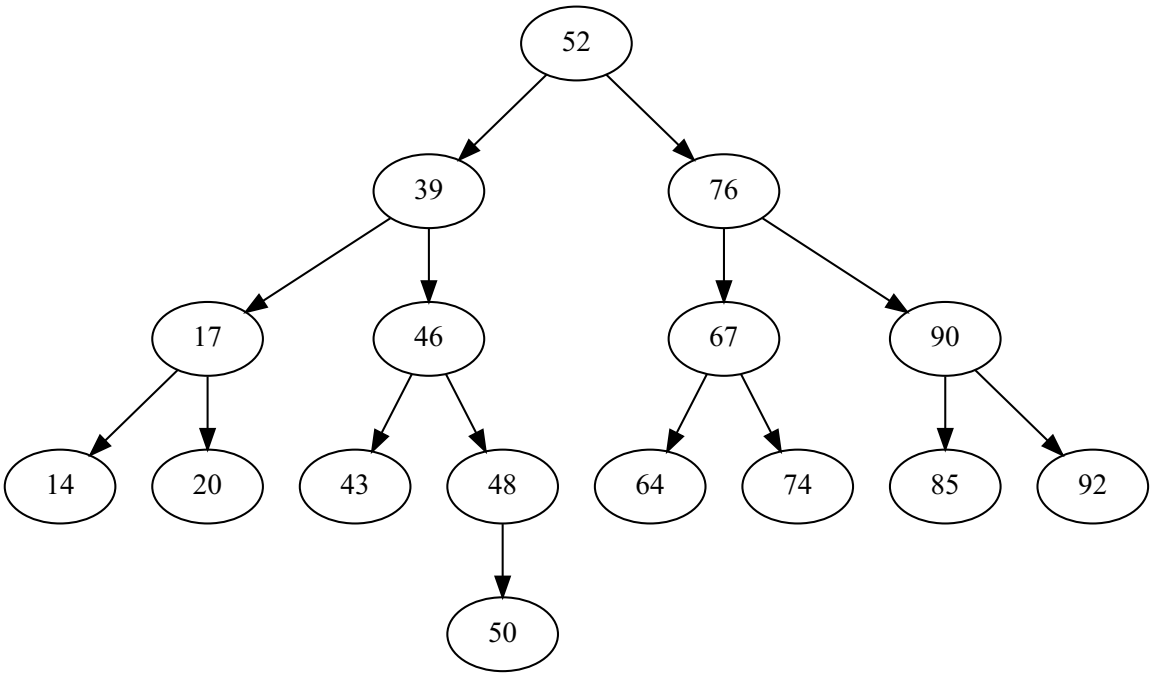
использует рекурсивную вспомогательную функцию `_insert`, которая принимает текущий узел и ключ. Внутри неё проверяется, пуст ли текущий узел: если да, создаётся и возвращается новый экземпляр `BSTNode`; в противном случае выбор направления осуществляется на основе сравнения ключей. Ветка с меньшими значениями обрабатывается через левый потомок, с большими или равными — через правый. После нескольких шагов спуска по дереву добавляется элемент, не нарушающий его упорядоченность.

Подобный механизм не требует полной перестройки дерева и обеспечивает корректное добавление за O(h), где h — высота дерева. В случае сбалансированного дерева этот показатель составляет O(logn), что делает вставку эффективной даже при большом количестве элементов. В практических задачах добавление сочетается с другими операциями (поиском и удалением), образуя основу большинства алгоритмов работы с упорядоченными множествами.

Добавим в уже построенное выше дерево `tree` новый элемент `50` и визуализируем результат:

In [ ]:
```
tree.insert(50)
tree.visualize()
```

Out[ ]:



После вызова метода `insert(50)` элемент будет размещён в правом поддереве узла `48`, поскольку `50` больше `48`, но меньше родительского узла `52`. Таким образом, структура остаётся корректной: все значения в левом поддереве меньше родительского узла, а в правом — больше или равны. Благодаря сохранённой сбалансированности начального дерева глубина поиска и вставки остаётся в пределах O(logn).

Для полноценной работы с бинарным деревом поиска необходимо реализовать механизм **поиска** элементов по ключу. Это расширяет функциональные возможности структуры, позволяя не только добавлять данные, но и извлекать их при необходимости. В данной реализации каждый узел дерева может содержать не только ключ, но и связанную с ним полезную нагрузку (`payload`), которая возвращается при успешном поиске. Метод `get` выполняет рекурсивный обход дерева с использованием вспомогательной функции `_get`. Дополнительно реализованы магические методы `__getitem__` и `__contains__`, позволяющие обращаться к дереву с использованием синтаксиса индексирования и оператора `in`.

Рассмотрим обновлённую версию класса `BSTNode`, в котором теперь предусмотрено поле `payload`, а также модифицированный класс `BSTree` с поддержкой операций поиска:

In [ ]:
```
class BSTNode:
    def __init__(self, key, payload=None):
        self.key = key
        self.payload = payload
        self.left = None
        self.right = None

class BSTree:
    def __init__(self):
        self.root = None

    def insert(self, key, payload=None):
        def _insert(node, key, payload):
            if node is None:
                return BSTNode(key, payload)
            if key < node.key:
                node.left = _insert(node.left, key, payload)
            else:
                node.right = _insert(node.right, key, payload)
```

```
            return node

        self.root = _insert(self.root, key, payload)

    def get(self, key):
        def _get(node, key):
            if node is None:
                return None
            if key == node.key:
                return node
            elif key < node.key:
                return _get(node.left, key)
            else:
                return _get(node.right, key)

        result = _get(self.root, key)
        return result.payload if result else None

    def __getitem__(self, key):
        return self.get(key)

    def __contains__(self, key):
        def _contains(node, key):
            if node is None:
                return False
            if key == node.key:
                return True
            elif key < node.key:
                return _contains(node.left, key)
            else:
                return _contains(node.right, key)

        return _contains(self.root, key)

    def visualize(self):
        from graphviz import Digraph
        dot = Digraph()

        def add_nodes_edges(node):
            if node is None:
                return
            dot.node(str(id(node)), str(node.key))
            if node.left:
                dot.edge(str(id(node)), str(id(node.left)))
                add_nodes_edges(node.left)
            if node.right:
                dot.edge(str(id(node)), str(id(node.right)))
                add_nodes_edges(node.right)

        if self.root:
            add_nodes_edges(self.root)
        return dot
```

Метод `insert` принимает дополнительный аргумент `payload`, который сохраняется в соответствующем узле дерева. Это позволяет ассоциировать с каждым ключом произвольные данные — например, строки, числовые значения или объекты произвольного типа.

Метод `get` инициирует поиск, начиная с корня дерева. Он вызывает вложенную функцию `_get`, которая сравнивает заданный ключ с ключами узлов дерева. Если совпадение найдено, возвращается полезная нагрузка, связанная с ключом, в противном случае возвращается `None`.

Функция `_get` реализует алгоритм поиска в бинарном дереве поиска. Если текущий узел отсутствует, это означает, что элемент с заданным ключом не содержится в дереве. При совпадении ключей возвращается узел, а при расхождении — рекурсивный спуск осуществляется либо в левое, либо в правое поддерево, в зависимости от результата сравнения.

Метод `__getitem__` позволяет использовать объект дерева с синтаксисом индексирования, как в случае с ассоциативными коллекциями, а `__contains__` обеспечивает поддержку оператора `in`, возвращая булев результат.

Таким образом, структура `BSTree` становится универсальным средством хранения и поиска данных, упорядоченных по ключу, с интуитивно понятным интерфейсом, соответствующим стандартному поведению коллекций в языке Python.

Выше была реализована функция `build_balanced_bst`, создающая сбалансированное бинарное дерево поиска из произвольного множества ключей. Однако на этапе построения каждому узлу присваивался только ключ, без дополнительных данных. Для расширения функциональности и демонстрации работы методов поиска удобно модифицировать конструктор узла, добавив возможность хранения полезной нагрузки (`payload`) — произвольной информации, связанной с ключом. Это позволит не только определить наличие элемента, но и получить связанное с ним содержимое.

Перепишем функцию построения дерева с учётом данной модификации:

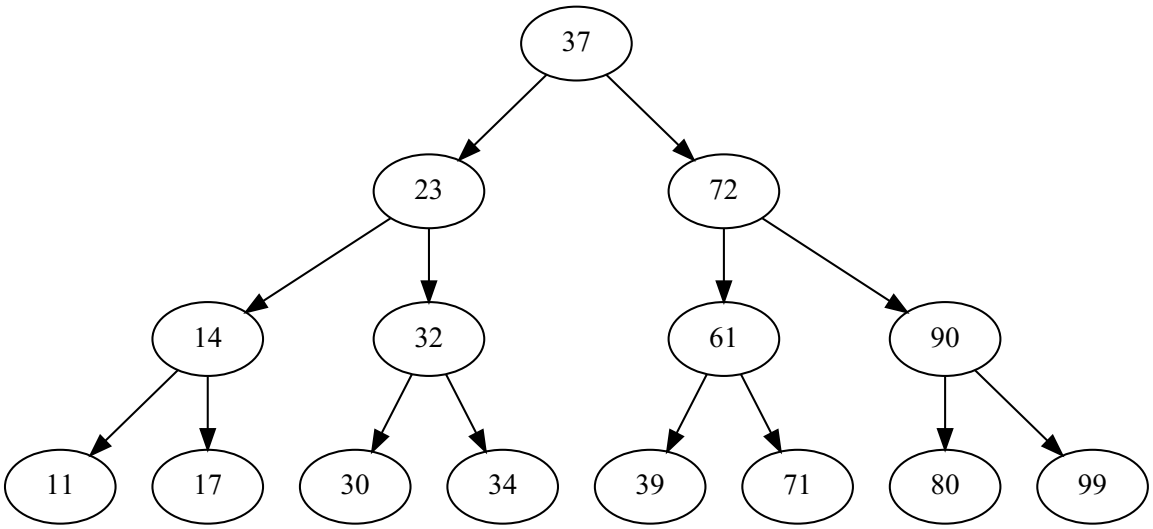
In [ ]:
```python
def build_balanced_bst(elements):
    tree = BSTree()
    elements = sorted(set(elements))

    def _build_balanced(lst):
        if not lst:
            return None
        mid = len(lst) // 2
        key = lst[mid]
        node = BSTNode(key, payload=f"значение: {key}")
        node.left = _build_balanced(lst[:mid])
        node.right = _build_balanced(lst[mid+1:])
        return node

    tree.root = _build_balanced(elements)
    return tree
```

Теперь каждому ключу при создании дерева сопоставляется строка с его числовым значением. Это удобно для отладки и иллюстрации работы поиска: при обращении по ключу возвращается ассоциированная информация.

Продемонстрируем использование новой версии дерева:

In [ ]:
```python
values = sample(range(10, 100), 15)
tree = build_balanced_bst(values)
tree.visualize()
```

Out[ ]:



In [ ]:
```python
test_keys = [values[3], values[7], 105]
```

In [ ]:
```python
for key in test_keys:
    if key in tree:
        print(f"Найдено: ключ {key} → {tree[key]}")
    else:
        print(f"Ключ {key} не найден в дереве.")
```

```
Найдено: ключ 72 → значение: 72
Найдено: ключ 99 → значение: 99
Ключ 105 не найден в дереве.
```

Этот пример показывает, как с помощью оператора `in` можно проверять наличие ключа в дереве, а с помощью индексной нотации — извлекать связанные данные. Благодаря сбалансированной структуре поиск выполняется эффективно, отражая ключевые свойства бинарного дерева поиска.

Удаление узла в бинарном дереве поиска требует внимательного подхода, особенно если необходимо сохранить корректность структуры и соблюдение её упорядоченности. Наиболее простой случай возникает тогда, когда удаляемый узел не имеет ни левого, ни правого потомка, то есть является листом. Такая операция считается базовой и подготавливает к более сложным сценариям, связанным с удалением узлов, имеющих одного или двух потомков.

Удаление листового узла выполняется по следующему алгоритму. Сначала вызывается операция поиска: начиная с корня дерева, производится последовательное сравнение ключей до тех пор, пока не будет найден узел, равный заданному значению. На каждом шаге определяется направление движения — влево, если ключ меньше текущего узла, и вправо — если больше. Когда найден узел, соответствующий удаляемому значению, производится дополнительная проверка: если у него отсутствуют оба потомка, то есть он не имеет дочерних узлов, то можно безопасно удалить этот элемент.

После того как найден узел, подлежащий удалению, необходимо определить, с какой стороны он связан с родительским узлом. Если удаляемый узел является корнем дерева, то это означает, что дерево состоит только из одного элемента. В таком случае для удаления достаточно установить корень дерева в значение `None`. Если же у дерева есть другие элементы, и удаляемый узел не является корнем, нужно найти его родителя — то есть узел, у которого он является потомком. Затем определяется,

находится ли удаляемый узел в левой или правой ветви родителя. После этого соответствующая ссылка (`left` или `right`) у родителя обнуляется, что приводит к логическому исключению узла из дерева. Таким образом, удалённый узел больше не участвует в структуре, и дерево остаётся корректным с точки зрения свойств бинарного поиска.

На рис. 15 показан пример удаления узла со значением `16` из бинарного дерева поиска.

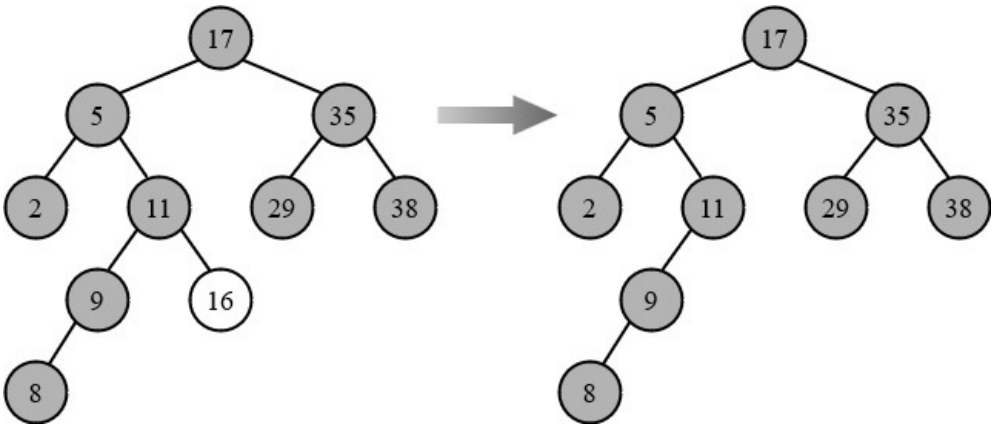


Рисунок 15 — Удаление из бинарного дерева поиска узла, не имеющего потомков

Алгоритм начинает работу с корневого узла — это значение `17` . Сравнение даёт результат: `16` меньше `17` , поэтому переход осуществляется в левое поддерево. Следующее сравнение — с узлом `5` — аналогично указывает на движение вправо, поскольку `16` больше `5` . Далее происходит переход к узлу `11` , у которого правым потомком как раз и является искомое значение — `16` . Проверка подтверждает, что у узла `16` нет потомков. Следовательно, его можно удалить, просто установив правого потомка узла `11` в значение `None` . Структура дерева при этом остаётся валидной: порядок узлов не нарушается, и все свойства бинарного дерева поиска сохраняются.

Удаление узла, имеющего одного потомка, представляет собой промежуточную по сложности операцию в рамках работы с бинарным деревом поиска. Основная задача в этом случае — корректно перенастроить связи так, чтобы структура дерева осталась логически целостной и удовлетворяла основным свойствам упорядоченности.

Алгоритм начинается с нахождения узла, подлежащего удалению. Этот этап полностью совпадает с процедурой поиска: выполняется последовательное сравнение ключа с текущими значениями в узлах дерева и переход в левое или правое поддерево до тех пор, пока не будет найден нужный элемент.

После того как целевой узел найден, анализируется его структура. Если у него только один потомок — левый или правый, — задача сводится к тому, чтобы «переподключить» этого потомка на место удаляемого узла. Существует три возможных ситуации:

- Если удаляемый узел не является корнем, необходимо определить, по какой из сторон (слева или справа) он связан с родительским узлом. Далее соответствующее поле (`left` или `right`) у родителя перенаправляется на единственного потомка удаляемого узла. Таким образом, родительский узел теряет связь с удаляемым и приобретает прямую связь с его потомком. Все поддеревья, находящиеся ниже, при этом сохраняются.
- Если удаляемый узел является корнем дерева, и у него есть только один потомок, то этот потомок становится новым корнем. В данном случае ссылка на текущий корень в объекте дерева просто переустанавливается на этого потомка. Это также сохраняет всю структуру нижестоящих элементов.
- Случай, когда у узла два потомка, требует более сложной перестройки, но он будет рассмотрен отдельно.

На рис. 16 показан пример удаления узла `25` , который является правым потомком корневого узла `17` . У узла `25` есть только один потомок — узел `35` . После удаления `25` его единственный потомок занимает его позицию в дереве. Это означает, что у узла `17` связь `right` теперь указывает непосредственно на `35` . Визуально это выглядит как замена одного узла другим на том же уровне дерева, без изменения остальных элементов.

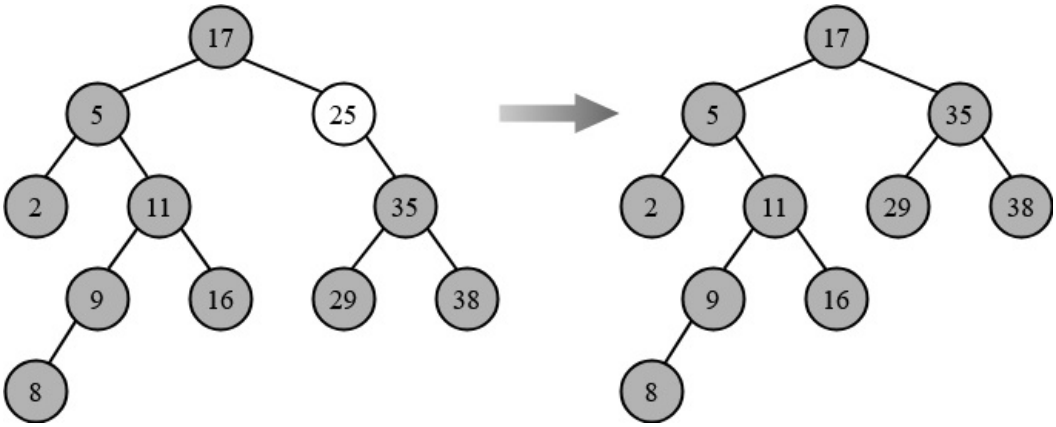


Рисунок 16 — Удаление из бинарного дерева поиска узла, имеющего одного потомка

Подобная операция занимает время O(h), где h — высота дерева, поскольку на её выполнение влияет необходимость первоначального поиска целевого узла. В сбалансированном дереве это эквивалентно O(logn), что делает удаление достаточно эффективной операцией.

Случай, при котором удаляемый узел имеет двух потомков, является наиболее сложным, поскольку требует не просто удаления, а корректной перестройки части структуры дерева. Если у удаляемого узла имеются и левый, и правый потомки, его нельзя заменить напрямую одним из них без нарушения свойства упорядоченности. В этом случае необходимо найти элемент, который может занять его место, сохранив правильный порядок ключей. Для этого используется один из двух подходов: в качестве замены выбирается либо наименьший элемент правого поддерева (преемник), либо наибольший элемент левого поддерева (предшественник). Эти элементы гарантированно располагаются в поддеревьях удаляемого узла и имеют не более одного потомка, что позволяет упростить их извлечение и корректно провести замену.

Процедура замены удаляемого узла в случае наличия двух потомков включает несколько последовательных этапов:

- Определяется стратегия замещения: выбирается либо ***преемник*** (наименьший элемент правого поддерева), либо ***предшественник*** (наибольший элемент левого поддерева). Оба варианта гарантируют сохранение порядка ключей в бинарном дереве поиска.
- Выполняется поиск выбранного узла-заменителя. В случае преемника — это самый левый узел правого поддерева; в случае предшественника — самый правый узел левого поддерева.
- Найденный узел извлекается из своей исходной позиции. Если у него имеется единственный потомок, родитель узла-заменителя перенаправляет соответствующую ссылку на этого потомка, тем самым восстанавливая структуру поддерева.
- На завершающем этапе удаляемый узел заменяется найденным узлом-заменителем. При этом к нему присоединяются левое и правое поддеревья, ранее принадлежавшие удаляемому узлу, что обеспечивает сохранение корректной структуры дерева.

На рис. 17 представлен пример удаления узла со значением `5`.

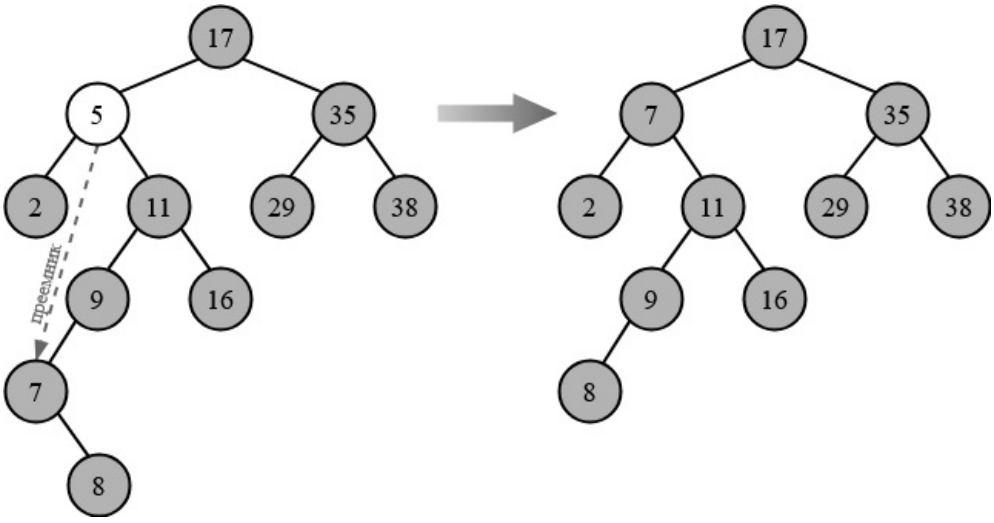


Рисунок 17 — Удаление из бинарного дерева поиска узла, имеющего двух потомков

Узел `5` имеет двух потомков — `2` и `11`. В качестве замены выбран преемник — наименьший элемент правого поддерева, которым является узел `7`. Узел `7` извлекается из своего прежнего положения (где он был левым потомком узла `9`) и устанавливается на место удаляемого узла `5`. Его левым потомком становится узел `2`, а правым — узел `11`, ранее принадлежавшие узлу `5`. Чтобы сохранить корректность поддерева, из которого был извлечён узел `7`, его родитель — узел

`9` — получает нового левого потомка: узел `8`, который ранее был правым потомком узла `7`. В результате порядок элементов сохраняется, и свойства бинарного дерева поиска остаются неизменными.

Таким образом, видно, что удаление в бинарном дереве поиска является важной и технически сложной операцией, требующей учёта структуры поддеревьев удаляемого узла. В зависимости от количества потомков возможны три различных сценария: узел не имеет потомков (лист), имеет одного потомка или имеет двух потомков. Последний случай требует замещения удаляемого узла другим элементом, который сохранит порядок ключей в дереве.

Существует две допустимые стратегии замещения: использование преемника — наименьшего элемента в правом поддереве, или предшественника — наибольшего элемента в левом поддереве. Оба подхода корректны с точки зрения поддержания структуры бинарного дерева поиска. В типичных реализациях используется один из них, однако рассмотрим универсальную реализацию, допускающую явный выбор стратегии.

Ниже представлена доработанная версия классов `BSTNode` и `BSTree`, в которую включён параметризированный метод `delete`. Он позволяет при удалении узла с двумя потомками выбирать подходящую стратегию замещения, указывая её в параметре `strategy`.

In [ ]:
```
from graphviz import Digraph
```

In [ ]:
```
class BSTNode:
    def __init__(self, key, payload=None):
        self.key = key
        self.payload = payload
        self.left = None
        self.right = None

class BSTree:
    def __init__(self):
        self.root = None

    def insert(self, key, payload=None):
        def _insert(node, key, payload):
            if node is None:
                return BSTNode(key, payload)
            if key < node.key:
                node.left = _insert(node.left, key, payload)
            else:
                node.right = _insert(node.right, key, payload)
            return node
        self.root = _insert(self.root, key, payload)

    def delete(self, key, strategy='successor'):
        def _delete(node, key):
            if node is None:
                return None

            if key < node.key:
                node.left = _delete(node.left, key)
            elif key > node.key:
                node.right = _delete(node.right, key)
            else:
                if node.left is None and node.right is None:
                    return None
                if node.left is None:
                    return node.right
                if node.right is None:
                    return node.left
                if strategy == 'successor':
                    replacement = _find_min(node.right)
                    node.right = _delete(node.right, replacement.key)
                elif strategy == 'predecessor':
                    replacement = _find_max(node.left)
                    node.left = _delete(node.left, replacement.key)
                else:
                    raise ValueError("Стратегия должна быть 'successor' или 'predecessor'")

                node.key = replacement.key
                node.payload = replacement.payload

            return node

        def _find_min(node):
            while node.left:
                node = node.left
            return node

        def _find_max(node):
            while node.right:
                node = node.right
```

```
            return node

        self.root = _delete(self.root, key)

    def get(self, key):
        def _get(node, key):
            if node is None:
                return None
            if key == node.key:
                return node
            elif key < node.key:
                return _get(node.left, key)
            else:
                return _get(node.right, key)
        result = _get(self.root, key)
        return result.payload if result else None

    def __getitem__(self, key):
        return self.get(key)

    def __contains__(self, key):
        def _contains(node, key):
            if node is None:
                return False
            if key == node.key:
                return True
            elif key < node.key:
                return _contains(node.left, key)
            else:
                return _contains(node.right, key)
        return _contains(self.root, key)

    def visualize(self):
        dot = Digraph()

        def add_nodes_edges(node):
            if node is None:
                return
            dot.node(str(id(node)), str(node.key))
            if node.left:
                dot.edge(str(id(node)), str(id(node.left)))
                add_nodes_edges(node.left)
            if node.right:
                dot.edge(str(id(node)), str(id(node.right)))
                add_nodes_edges(node.right)

        if self.root:
            add_nodes_edges(self.root)
        return dot
```

Метод `delete` реализован как рекурсивная функция `_delete`, которая осуществляет последовательный спуск по дереву в поисках узла, соответствующего заданному ключу. На каждом шаге выполняется сравнение текущего ключа с ключом узла: если ключ меньше, рекурсивный вызов передаётся в левое поддерево; если больше — в правое. Такая структура позволяет эффективно локализовать целевой узел с использованием логики бинарного поиска.

Как только узел с соответствующим ключом найден, происходит анализ его локальной структуры — определяется количество дочерних узлов. Если ни один из потомков не существует, узел считается листом и удаляется путём возврата `None`, что влечёт за собой обнуление соответствующего указателя в родительском узле. При наличии только одного потомка (левого или правого) возвращается ссылка на этот потомок, что приводит к замене удаляемого узла на его наследника.

Если же у найденного узла имеются оба потомка, возникает необходимость подобрать замещающий элемент, который сможет занять позицию удаляемого узла, сохранив при этом упорядоченность ключей. В этой ситуации вызывается вспомогательная функция, осуществляющая поиск подходящего узла-заменителя в одном из поддеревьев. Выбор стратегии зависит от значения параметра `strategy`, передаваемого в метод `delete`.

Если указана стратегия `'successor'`, используется функция `_find_min`, которая находит наименьший узел в правом поддереве. Для этого выполняется спуск по левым указателям, начиная с правого потомка удаляемого узла, до тех пор, пока не будет достигнут самый левый элемент. Такой узел гарантированно больше всех элементов в левом поддереве, но меньше остальных узлов правого поддерева, а значит, корректно встраивается в структуру на месте удаляемого.

Если выбрана стратегия `'predecessor'`, вызывается функция `_find_max`, реализующая аналогичный алгоритм, но в левом поддереве. Здесь выполняется спуск по правым указателям до самого правого узла, который в данной части дерева является наибольшим. Его замещение обеспечивает корректное положение по отношению ко всем остальным узлам, сохраняя инварианты структуры.

После того как найден узел-заменитель, его значения (ключ и полезная нагрузка) копируются в удаляемый узел, а сам узел-заменитель удаляется из своего исходного положения путём рекурсивного вызова `_delete`. Так как он, как правило, имеет не

более одного потомка, его удаление проходит по упрощённому сценарию. Такой подход гарантирует, что структура дерева остаётся корректной, а все связи восстановлены.

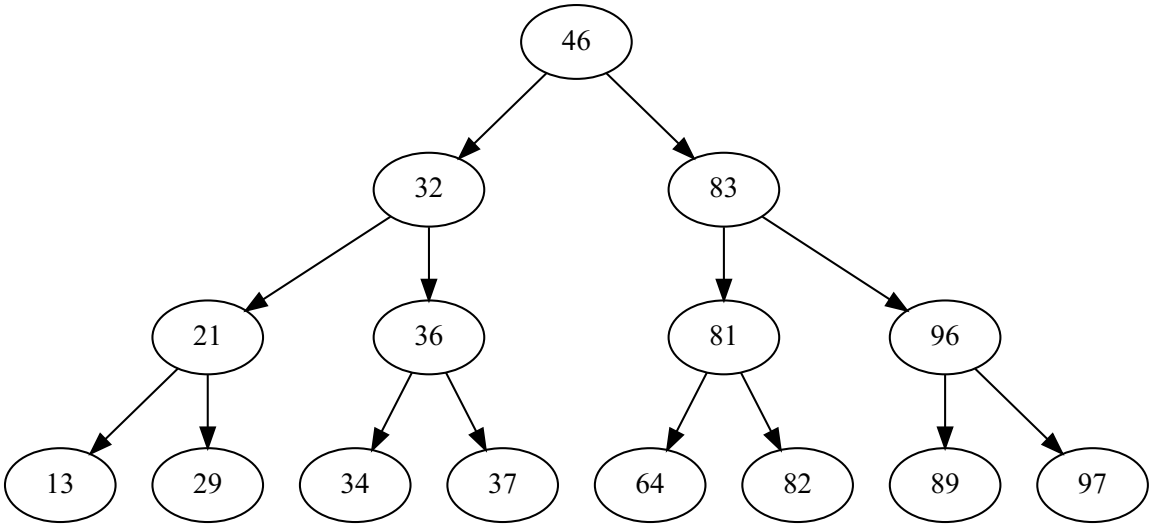
Параметризация стратегии позволяет использовать одну и ту же реализацию метода `delete` для разных вариантов поведения. Это делает код универсальным и расширяемым, а также даёт возможность экспериментировать со структурой дерева, сравнивая поведение при использовании преемника и предшественника.

Для демонстрации различных сценариев удаления узлов в бинарном дереве поиска используется функция `build_balanced_bst`, формирующая сбалансированную структуру из 15 случайных уникальных значений. На этапе построения дерево имеет минимально возможную высоту, что упрощает анализ его изменений.

In [ ]:
```
from random import sample
```

In [ ]:
```
values = sample(range(10, 100), 15)
tree = build_balanced_bst(values)
tree.visualize()
```

Out[ ]:



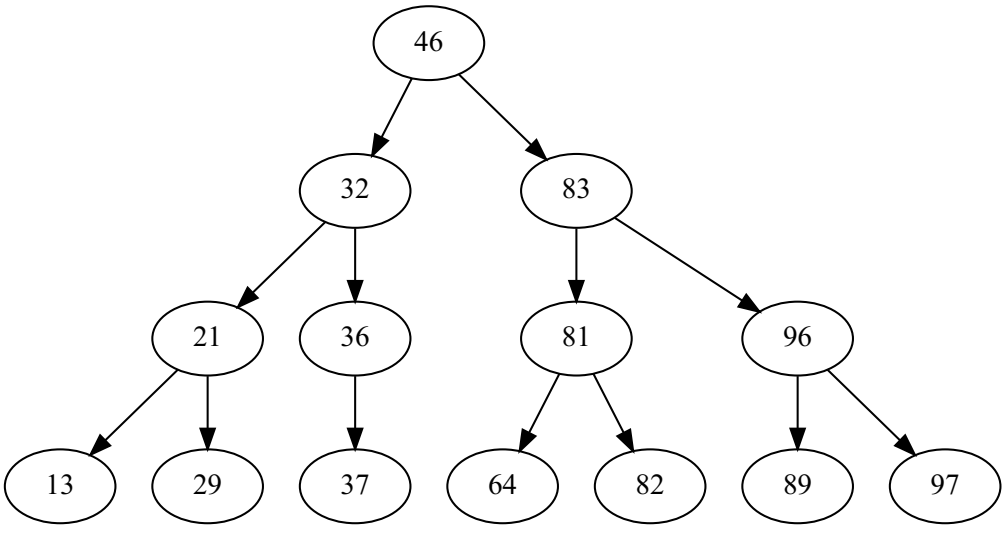
В корне находится узел `46`, левое поддерево которого содержит значения от `13` до `37`, а правое — от `64` до `97`. Структура симметрична и отвечает критериям сбалансированного дерева поиска.

Теперь можно перейти к первому сценарию — удалению ***листа***, то есть узла, не имеющего ни левого, ни правого потомка. Для выполнения операции запрашивается значение удаляемого узла, после чего вызывается метод `delete` с параметром `strategy='successor'`, указывающим на использование стратегии замещения через преемника. После удаления дерево вновь визуализируется, позволяя наглядно убедиться в том, что структура осталась корректной, а порядок элементов не нарушен.

In [ ]:
```
key = int(input("Удаление листа\nВведите ключ для удаления: "))
tree.delete(key, strategy='successor')
tree.visualize()
```

```
Удаление листа
Введите ключ для удаления: 34
```

Out[ ]:



Видно, что удалённый узел `34`, находившийся в левом поддереве узла `36`, исчез из структуры. Остальные связи остались без изменений, дерево сохранило корректность, а порядок элементов не нарушен.

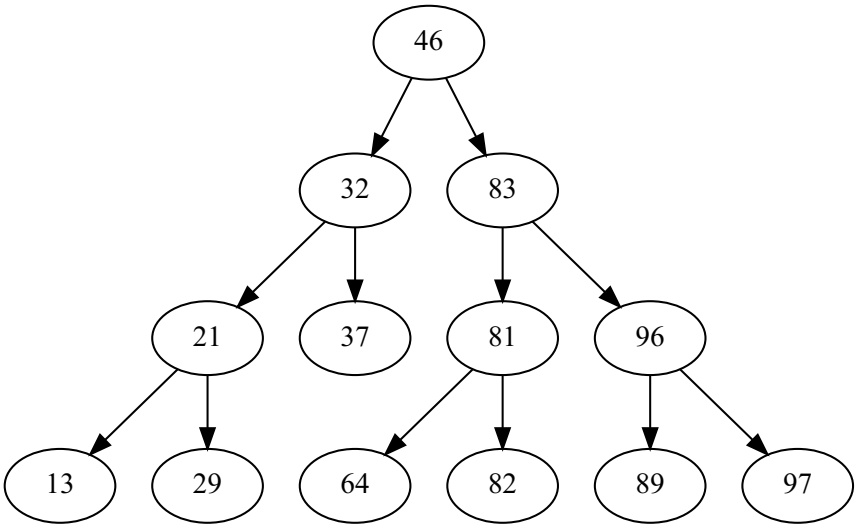
Далее демонстрируется удаление ***узла с одним потомком***. После ввода соответствующего ключа выполняется операция

удаления, и визуально можно проследить, как его единственный потомок занимает место удалённого узла, а связи со всеми нижестоящими элементами сохраняются.

In [ ]:
```
key = int(input("Удаление узла с одним потомком\nВведите ключ для удаления: "))
tree.delete(key, strategy='successor')
tree.visualize()
```

```
Удаление узла с одним потомком
Введите ключ для удаления: 36
```

Out[ ]:



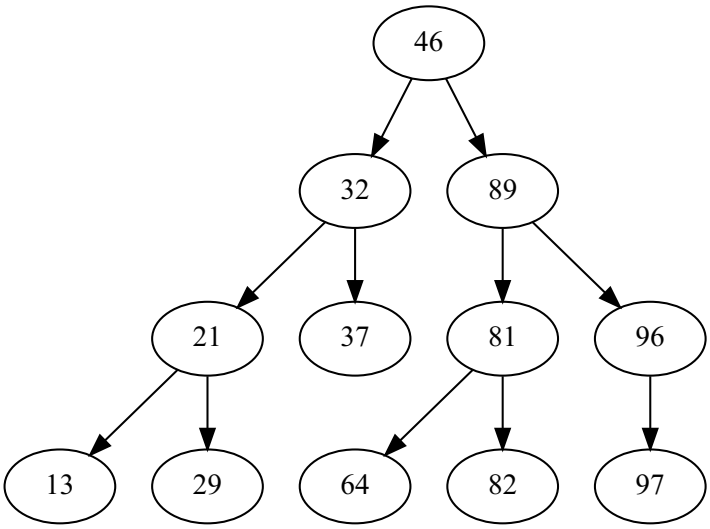
После удаления узла `36` его единственный потомок `37` занимает его место. Связь между узлами `32` и `36` перенаправлена на `37`, в остальном структура дерева не изменилась. Все свойства бинарного дерева поиска соблюдены.

Затем рассматривается удаление ***узла с двумя потомками***, *при котором требуется корректная замена на замещающий узел. В первом варианте используется стратегия по умолчанию, основанная на выборе* **преемника***, то есть минимального узла в правом поддереве.

In [ ]:
```
key = int(input("Удаление узла с двумя потомками (преемник)\nВведите ключ для удаления: "))
tree.delete(key, strategy='successor')
tree.visualize()
```

```
Удаление узла с двумя потомками (преемник)
Введите ключ для удаления: 83
```

Out[ ]:



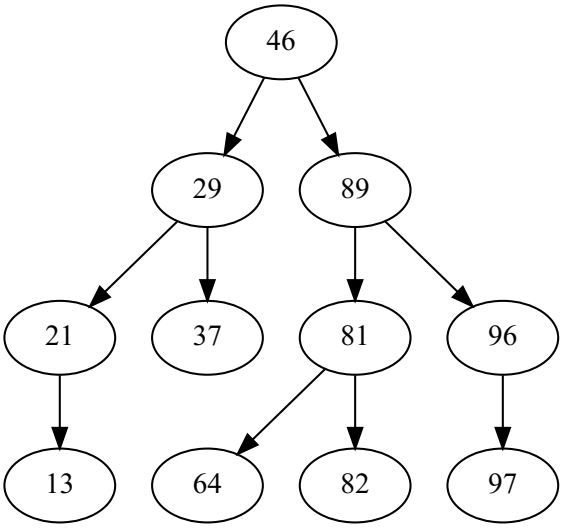
Видно, что узел `83` был удалён, а его место занял преемник — минимальный элемент правого поддерева, то есть `89`. Его прежнее положение в поддереве узла `96` было освобождено. При этом поддеревья были корректно переподключены.

Наконец, выполняется аналогичная операция, но уже с использованием ***предшественника*** — наибольшего элемента левого поддерева. Это позволяет сравнить поведение дерева при использовании двух разных подходов.

In [ ]:
```
key = int(input("Удаление узла с двумя потомками (предшественник)\nВведите ключ для удаления: "))
tree.delete(key, strategy='predecessor')
tree.visualize()
```

```
Удаление узла с двумя потомками (предшественник)
Введите ключ для удаления: 32
```

Out[ ]:



Теперь замещающим узлом становится `29` — наибольший элемент левого поддерева. Он занимает место узла `32`, а в поддереве, где ранее находился, заменяется на `None`, так как у него не было правого потомка. Все связи и иерархия остаются корректными.

Таким образом, каждый из представленных сценариев демонстрирует, как универсальный метод `delete` корректно работает при различных конфигурациях поддеревьев, независимо от выбранной стратегии замещения.

Выбор стратегии замещения при удалении узла с двумя потомками может зависеть как от соглашения, принятого в конкретной реализации, так и от соображений симметрии, предсказуемости поведения или удобства отладки. Замена через преемника (наименьший элемент правого поддерева) считается более распространённой и интуитивной, поскольку позволяет «подняться вверх» от удаляемого узла и продолжить упорядочивание в том же направлении. Предшественник (наибольший элемент левого поддерева) даёт зеркальную альтернативу, и в некоторых случаях может быть предпочтён, например, при анализе глубины поддеревьев или в случае, если правое поддерево особенно «тяжёлое» и хочется минимизировать сдвиг в этой части структуры. Важно подчеркнуть, что оба подхода корректны и сохраняют инвариант упорядоченности — выбор между ними не влияет на логическую правильность дерева, но может отражать предпочтения разработчика или особенности задачи.

Несмотря на корректность всех рассмотренных операций — вставки, поиска и удаления, — в динамически изменяющемся дереве со временем может накапливаться дисбаланс. Даже при начальной сбалансированной структуре последовательные модификации могут постепенно сместить узлы и вытянуть дерево вглубь. Это неизбежно приводит к увеличению его высоты, а вместе с ней — к ухудшению временных характеристик. Таким образом, возникает необходимость не только поддерживать свойства упорядоченности, но и контролировать форму дерева, стремясь к равномерному распределению элементов. Именно в этом контексте вводится понятие сбалансированного бинарного дерева поиска, в том числе — его идеального варианта.

Идеально сбалансированным бинарным деревом поиска называется такое дерево, в котором для любой вершины количество узлов в левом и правом поддеревьях отличается не более чем на один. Такая структура обеспечивает минимально возможную высоту для заданного числа узлов, что, в свою очередь, гарантирует асимптотически оптимальное время выполнения основных операций — поиска, вставки и удаления — порядка O(logn). В идеальном случае дерево имеет форму, близкую к полной, и его элементы равномерно распределены по уровням, что способствует высокой эффективности при работе с большими объёмами данных.

Однако поддержание такой структуры при произвольной последовательности операций вставки и удаления представляет собой крайне сложную задачу. Любое отклонение от тщательно выверенного порядка добавления может нарушить балансировку. На практике это означает, что идеально сбалансированные деревья не строятся динамически без применения специальных алгоритмов поддержания баланса. Поэтому чаще используются ослабленные критерии сбалансированности, при которых допускается некоторое превышение разницы в размерах поддеревьев, но сохраняется приемлемая высота дерева. Такие подходы лежат в основе построения так называемых сбалансированных деревьев, например, AVL-деревьев, красно-чёрных деревьев или B-деревьев.

Противоположностью сбалансированной структуры служит вырожденное дерево — структура, в которой каждый узел имеет лишь одного потомка, превращая дерево фактически в односвязный список. Это может произойти, если элементы вставляются в уже отсортированном порядке, что лишает дерево его ключевого преимущества — логарифмической глубины. В результате операции поиска, вставки и удаления начинают работать за O(n), что сопоставимо с наихудшими случаями линейного поиска в неупорядоченных структурах. На рис. 18 представлен пример такого вырожденного дерева, в котором каждый элемент больше предыдущего и вставляется в качестве правого потомка, формируя цепочку из пяти узлов.

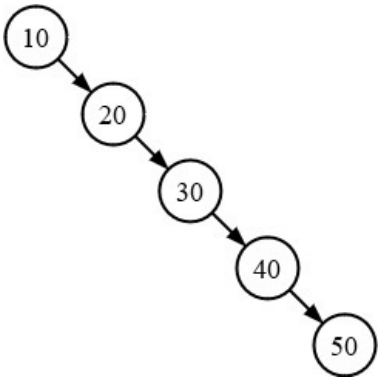


Рисунок 18 — Пример неэффективного бинарного дерева поиска

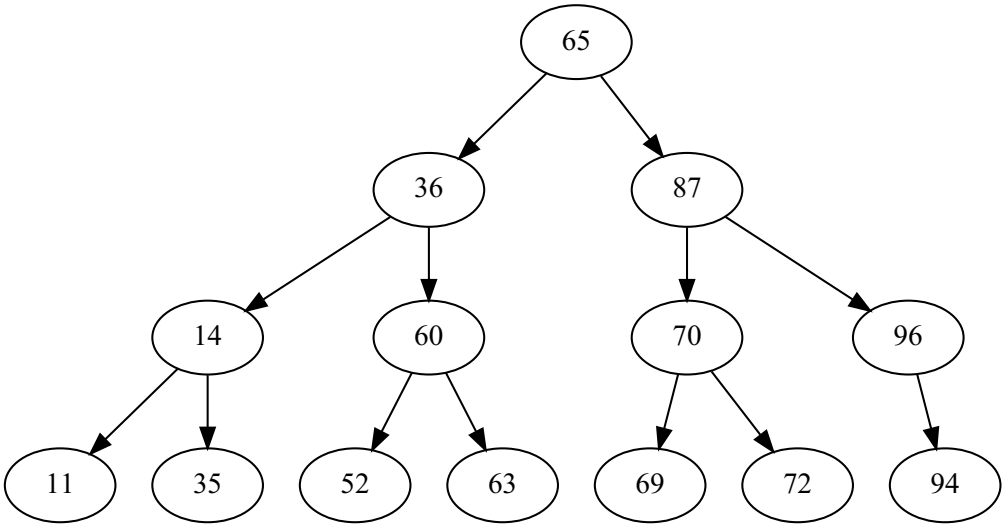
Следующие примеры практического использования бинарного дерева поиска иллюстрируют, как можно применять разработанную структуру для решения задач анализа и поиска информации. Все примеры опираются на ранее реализованные классы `BSTNode` и `BSTree`, а также на вспомогательную функцию `build_balanced_bst`, которая используется для создания сбалансированного дерева из случайного набора значений.

Рассмотрим первый пример, в котором необходимо подсчитать количество узлов в дереве. Эта задача решается с помощью рекурсивного обхода: для каждого узла суммируется единица (текущий узел) и количество узлов в левом и правом поддеревьях. Алгоритм обходит всю структуру, не пропуская ни одного элемента, и позволяет оценить размер дерева, что важно, например, при расчёте глубины или плотности данных.

In [ ]:
```
from random import sample, randint
```

In [ ]:
```
values = sample(range(10, 100), randint(8, 15))
tree = build_balanced_bst(values)
tree.visualize()
```

Out[ ]:



In [ ]:
```
def count_nodes(node):
    if node is None:
        return 0
    return 1 + count_nodes(node.left) + count_nodes(node.right)
```

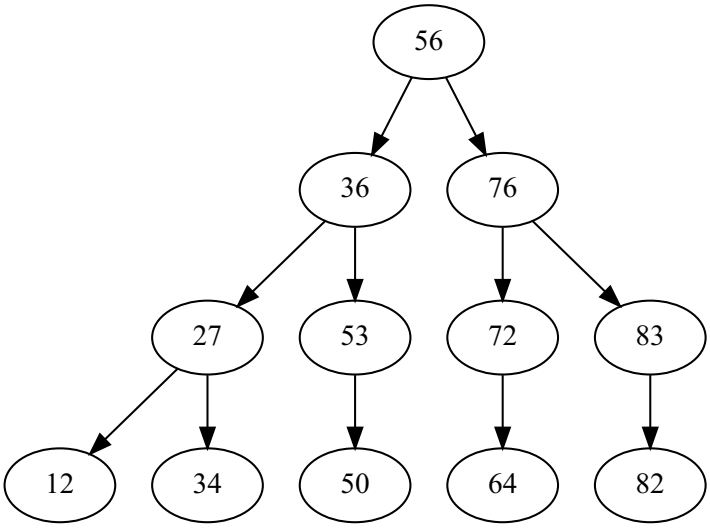
In [ ]:
```
print("Количество узлов в дереве:", count_nodes(tree.root))
```

```
Количество узлов в дереве: 14
```

Во втором примере ставится задача нахождения всех родительских узлов для заданного элемента. Поиск осуществляется путём рекурсивного спуска по дереву: если целевой узел найден среди потомков текущего, то текущий узел добавляется в список родителей. Таким образом формируется цепочка от корня до нужного элемента, содержащая только тех узлов, через которые проходит путь к целевому.

In [ ]:
```
values = sample(range(10, 100), randint(8, 15))
tree = build_balanced_bst(values)
tree.visualize()
```

Out[ ]:



In [ ]:
```python
def find_parents(node, target_key):
    if node is None:
        return []
    if (node.left and node.left.key == target_key) or (node.right and node.right.key == target_key):
        return [node.key]
    left = find_parents(node.left, target_key)
    if left:
        return [node.key] + left
    right = find_parents(node.right, target_key)
    if right:
        return [node.key] + right
    return []
```

In [ ]:
```python
key = int(input("Введите ключ для нахождения всех родительских узлов: "))
```

```
Введите ключ для нахождения всех родительских узлов: 12
```

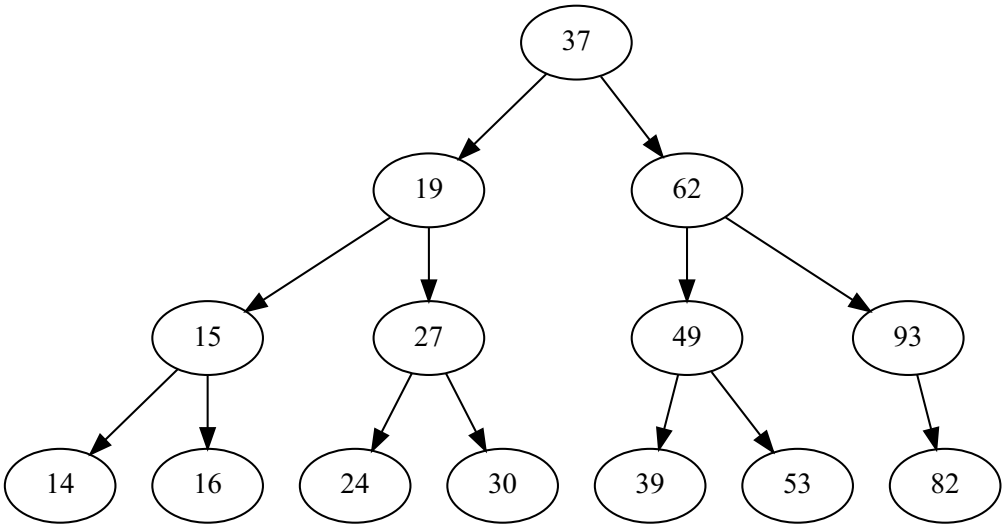
In [ ]:
```python
print(f"Список всех родительских узлов для ключа {key}:", find_parents(tree.root, key))
```

```
Список всех родительских узлов для ключа 12: [56, 36, 27]
```

Третий пример демонстрирует поиск всех узлов, значения которых не меньше заданного. Это реализуется с помощью полного обхода дерева, при котором проверяется каждый ключ, а те, что удовлетворяют условию, сохраняются в результат. Данная задача может применяться при отборе элементов по условию фильтрации.

In [ ]:
```python
values = sample(range(10, 100), randint(8, 15))
tree = build_balanced_bst(values)
tree.visualize()
```

Out[ ]:



In [ ]:
```python
def find_nodes_greater_equal(node, value):
    if node is None:
        return []
    result = []
    if node.key >= value:
        result.append(node.key)
    result += find_nodes_greater_equal(node.left, value)
    result += find_nodes_greater_equal(node.right, value)
    return result
```

In [ ]:
```python
value = int(input("Введите значение: "))
```

```
Введите значение: 27
```

```
In [ ]: print(f"Список ключей, удовлетворяющих условию:", find_nodes_greater_equal(tree.root, value))
```

```
Список ключей, удовлетворяющих условию: [37, 27, 30, 62, 49, 39, 53, 93, 82]
```

Таким образом, бинарное дерево поиска представляет собой универсальную структуру для хранения упорядоченных данных, сочетающую компактность представления с высокой эффективностью выполнения операций. Реализация всех ключевых механизмов (от вставки и поиска до удаления с выбором стратегии замещения) делает эту структуру надёжной основой для организации данных в прикладных задачах. Поддержка ассоциированных значений и визуализация усиливают её прикладную значимость, а проведённые эксперименты демонстрируют, как на её основе можно решать задачи анализа, фильтрации и логической навигации по иерархическим коллекциям.

Двоичные кучи и очереди с приоритетом

Двоичная куча представляет собой частный случай бинарного дерева, обладающий строго определёнными свойствами структуры и порядка. Её устройство делает возможным эффективное выполнение операций, связанных с выбором элемента с наивысшим или наинизшим приоритетом. Основное условие упорядоченности заключается в следующем: значение в любой вершине не превышает значений её потомков. Это означает, что корневой элемент является минимальным среди всех, а по мере продвижения от корня к листьям значения не убывают. Такая структура называется ***минимальной кучей*** и используется в ситуациях, где важен быстрый доступ к наименьшему элементу. Существуют и противоположные реализации, в которых каждое значение в вершине не меньше значений потомков. В этом случае речь идёт о ***максимальной куче***, предназначенной для приоритетного извлечения наибольшего элемента. Обе разновидности кучи организованы по единому принципу, отличаясь лишь направлением сравнения.

На рис. 19 приведён пример минимальной двоичной кучи.

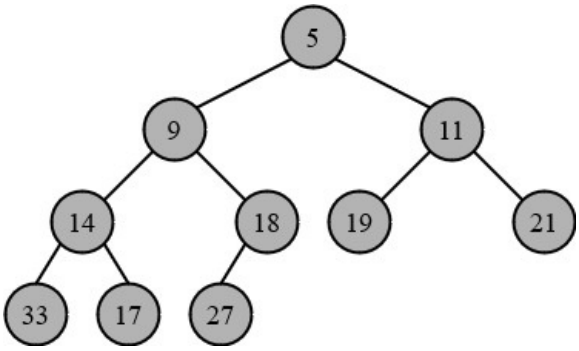


Рисунок 19 — Пример минимальной двоичной кучи

В корне дерева находится наименьшее значение — `5`. Каждый родительский узел сохраняет отношение нестрогого порядка по отношению к своим потомкам: `5` меньше `9` и `11`, `9` меньше `14` и `18`, `11` меньше `19` и `21`, `14` меньше `33` и `17`, а `18` меньше `27`. Эта структура удовлетворяет ключевому свойству минимальной кучи — для любой вершины выполняется условие, что её значение не превосходит значений потомков.

Если бы дерево на рисунке было построено по правилам максимальной кучи, на его вершине располагалось бы наибольшее значение, а далее по пути к листьям происходило бы строгое или нестрогое убывание. Таким образом, при внешнем сходстве форма кучи не определяет направление приоритетов, которое зависит исключительно от логики сравнения между вершинами и их потомками.

Свойство упорядоченности в куче реализуется локально. Это означает, что никакие дополнительные условия не накладываются на значения в левом и правом поддеревьях, за исключением отношения к значению родительского узла. В частности, значения в левом и правом поддеревьях могут быть произвольны по отношению друг к другу, и при этом структура остаётся корректной, если только сохраняется инвариант: родитель не превышает (в минимальной куче) или не уступает (в максимальной куче) каждому из потомков. Это отличие от, например, бинарного дерева поиска, где значения в левом поддереве строго меньше, а в правом — больше либо равны. В куче сравнения и порядок организованы иначе, что делает её подходящей именно для задач с приоритетами, но не для хранения упорядоченных коллекций.

Кроме свойства упорядоченности, двоичная куча обладает определённой структурной формой, которую необходимо сохранять при любой модификации дерева. Она реализуется в виде ***полного бинарного дерева:*** все уровни, за исключением последнего, содержат максимально возможное число узлов, а вершины последнего уровня заполняются слева направо без пропусков. Такая организация обеспечивает компактность хранения, а также ограничивает высоту дерева значением порядка $\log n$, где n — количество элементов. Это, в свою очередь, гарантирует логарифмическую трудоёмкость при выполнении базовых операций.

Структура полного дерева создаёт условия, при которых каждый новый элемент добавляется на последнюю позицию в нижнем уровне. Тем самым поддерживается баланс дерева: ни одна ветвь не оказывается избыточно глубокой. После добавления элемента, возможно нарушение инварианта кучи, если новое значение меньше (в минимальной куче) или больше (в

максимальной) значения своего родителя.

Для восстановления порядка используется механизм, основанный на последовательном подъёме узла по направлению к корню — до тех пор, пока условие упорядоченности не будет восстановлено. Аналогично, при удалении корневого элемента его место занимает последняя вершина в дереве, после чего структура перестраивается через серию сравнений и обменов, искавшихся вниз по направлению к листьям. Благодаря строгой форме дерева и локальности инварианта такие преобразования возможны за ограниченное число шагов и не нарушают общей сбалансированности.

Поддержание этих свойств позволяет эффективно извлекать корневой элемент, добавлять новые вершины и перестраивать дерево без потери упорядоченности. Все эти действия сохраняют структуру полного бинарного дерева, а механизм локальных перестроений делает возможным использование двоичных куч в реальных задачах, связанных с выбором элементов с наивысшим или наинизшим приоритетом. Подобные задачи возникают, в частности, в алгоритмах планирования, в системах обслуживания заявок и в процедурах, связанных с упорядочиванием данных по значимости.

Полнота двоичной кучи позволяет не только эффективно организовать её в памяти, но и упростить способ представления. Особенность полного бинарного дерева заключается в том, что такую структуру можно записать в виде ***линейного списка***, сохранив при этом все связи между вершинами. Это представление опирается на строгую закономерность взаимного расположения индексов элементов в массиве.

Если вершина кучи расположена в списке под индексом p, то её левый потомок занимает позицию 2p + 1, а правый потомок — позицию 2p + 2. Эти формулы обеспечивают прямой переход от родителя к потомкам. В обратную сторону — для любого элемента с индексом $i > 0$ — индекс его родительского узла вычисляется как целая часть от $(i - 1)/2$, то есть $\left\lfloor \frac{i-1}{2} \right\rfloor$. Благодаря этому дерево может быть полностью восстановлено по одному массиву, а навигация по уровням дерева реализуется с помощью простых арифметических операций.

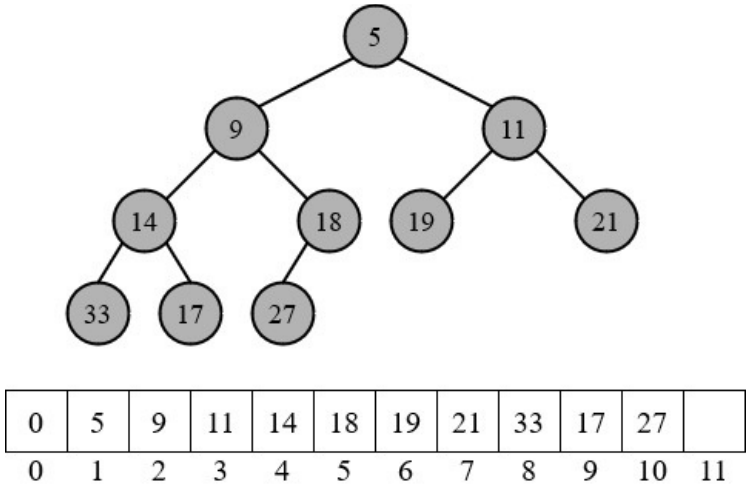


Рисунок 20 — Представление двоичной кучи в виде списка

На рис. 20 показано, как структура минимальной кучи отображается в линейный массив. Верхняя часть рисунка повторяет уже знакомое дерево с вершиной `5` в корне, где каждая вершина удовлетворяет условию, что она не превышает значения своих потомков. Ниже приведено массивное представление той же структуры: элементы дерева записаны в том порядке, в котором они встречаются при поуровневом обходе слева направо. Узел с ключом `5` находится на позиции `0`, его потомки `9` и `11` — на позициях `1` и `2` соответственно. Узел `9` связан с вершинами `14` и `18`, расположенными на позициях `3` и `4`. Аналогичным образом восстанавливаются остальные связи.

Такой способ представления делает возможным реализацию всех операций с кучей — добавления, удаления и поддержания порядка — в рамках одного списка фиксированной длины. Это устраняет необходимость в явных указателях или дополнительных структурах. Вся информация о логике вложенности хранится неявно, через индексы. В результате представление кучи в виде массива объединяет компактность, эффективность и простоту реализации.

Создадим отдельный класс `BinaryHeap`, реализующий базовые методы структуры. Такой класс инкапсулирует логику хранения, поддержания упорядоченности и восстановления инварианта кучи. Его основная задача — моделирование минимальной кучи, в которой каждый родитель содержит значение, не превышающее значений потомков. Это обеспечит быстрый доступ к минимальному элементу, его удаление, вставку новых значений, а также возможность построения кучи на основе произвольного набора данных.

В состав класса войдут следующие методы: инициализация новой кучи, вставка элемента, получение и удаление минимального элемента, проверка на пустоту, вычисление размера и создание кучи на основе неупорядоченного массива. Кроме того, предусмотрим метод визуализации, позволяющий представить структуру текущей кучи в виде дерева, что обеспечит наглядность при анализе работы алгоритмов.

Ниже приведена начальная версия класса `BinaryHeap`, включающая только конструктор. Внутреннее представление кучи реализовано с помощью списка `heap_list`, первым элементом которого служит значение `0`. Этот элемент используется исключительно технически — он занимает позицию с индексом `0`, которая не участвует в логике хранения и обработки, но позволяет применять более простые формулы для навигации по дереву. Переменная `current_size` отслеживает текущее число элементов в куче и инициализируется нулём.

In [ ]:
```
class BinaryHeap:
    def __init__(self):
        self.heap_list = [0]
        self.current_size = 0
```

В дальнейшем в этот класс будут добавлены методы вставки, удаления, перестройки и визуализации.

Добавление нового элемента в кучу представляет собой один из базовых механизмов модификации структуры. Эта операция должна сохранять два ключевых свойства: форму полного бинарного дерева и локальную упорядоченность, характерную для минимальной или максимальной кучи. В рассматриваемом случае используется минимальная куча, где каждый родительский узел не превышает значений своих потомков.

Процесс начинается с размещения нового элемента в первой свободной позиции — в самом конце массива, который представляет кучу. Тем самым сохраняется структура полного дерева: вершины продолжают заполняться слева направо, уровень за уровнем. Однако добавленный элемент может нарушить упорядоченность, поскольку его значение может оказаться меньше, чем у родителя. В таком случае возникает необходимость восстановить инвариант кучи.

Восстановление происходит путём последовательного обмена нового элемента с родительским узлом. Если значение вставленного элемента меньше значения родителя, то они меняются местами. После этого сравнение повторяется уже с новым родителем — и так до тех пор, пока либо элемент не окажется на корне дерева, либо не будет достигнута вершина, у которой значение меньше либо равно вставляемому. Этот процесс получил название ***просеивания вверх*** и обеспечивает восстановление порядка без нарушения структуры дерева. Количество шагов не превышает высоты дерева, то есть составляет не более $\log n$ для n элементов в куче. Это определяет асимптотическую сложность операции как $O(\log n)$.

На рис. 21 представлен пример добавления нового узла со значением `7` в минимальную двоичную кучу. Сначала элемент помещается в конец дерева, занимая позицию правого потомка узла `18`. На этом этапе сохраняется форма полного бинарного дерева, но нарушается упорядоченность: значение `7` меньше значения родителя. В следующем шаге они меняются местами, и `7` поднимается на уровень выше. Его новым родителем становится узел `9`, значение которого также больше, чем `7`. Производится второй обмен, после чего инвариант восстанавливается, и элемент оказывается на корректной позиции внутри кучи. Структура вновь удовлетворяет всем условиям: дерево остаётся полным, а каждый родитель меньше или равен своим потомкам.

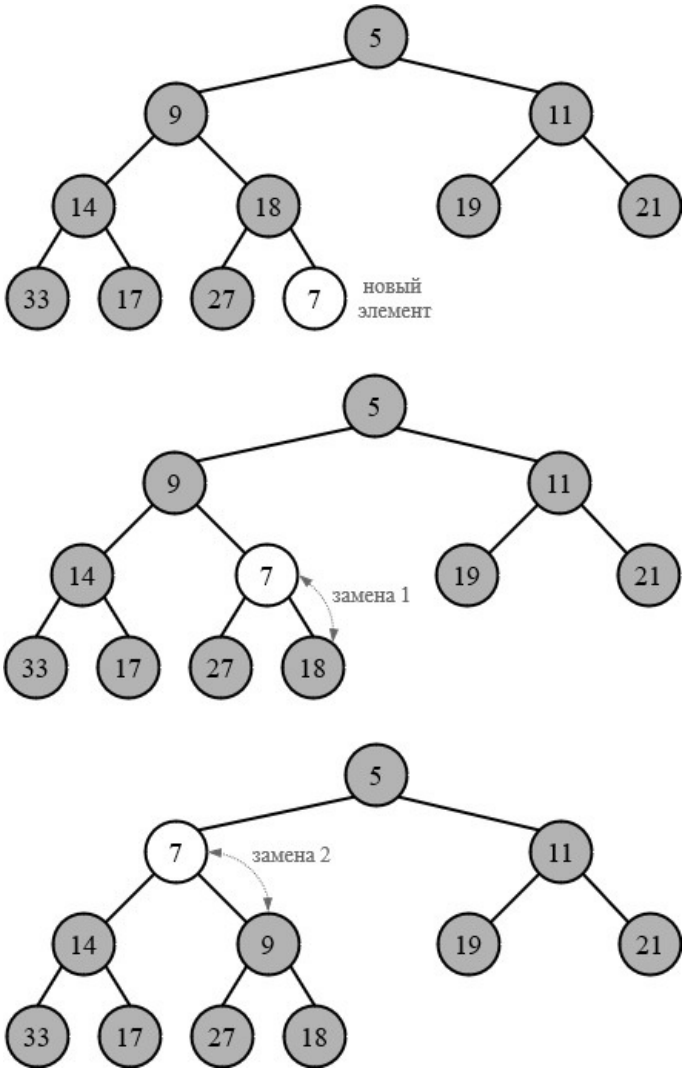


Рисунок 21 — Вставка нового узла в минимальную двоичную кучу

Вставка нового элемента сопровождается локальными перестроениями, которые не затрагивают всё дерево, а охватывают лишь один путь вверх от нового узла. Это делает операцию предсказуемой по времени и удобной для динамического расширения кучи.

Таким образом, добавление элемента в кучу требует не только вставки значения в конец массива, но и восстановления структуры, отражающей свойства минимальной кучи. Это достигается путём последовательного сравнения и обмена элементов при необходимости. В реализации эти действия распределены между двумя методами: `insert` и `perc_up`. Кроме того, для наглядного анализа изменений в структуре кучи после операций вставки реализуем метод визуализации, который отображает текущее состояние дерева с использованием библиотеки `graphviz`.

In [ ]:
```
from graphviz import Digraph
```

In [ ]:
```
class BinaryHeap:
    def __init__(self):
        self.heap_list = [0]
        self.current_size = 0

    def insert(self, k):
        self.heap_list.append(k)
        self.current_size = self.current_size + 1
        self.perc_up(self.current_size)

    def perc_up(self, i):
        while i // 2 > 0:
            if self.heap_list[i] < self.heap_list[i // 2]:
                tmp = self.heap_list[i // 2]
                self.heap_list[i // 2] = self.heap_list[i]
                self.heap_list[i] = tmp
            else:
                break
            i = i // 2

    def visualize(self):
        dot = Digraph()
        n = len(self.heap_list) - 1
        for i in range(1, n + 1):
```

```
            dot.node(str(i), str(self.heap_list[i]))
            parent = i // 2
            if parent > 0:
                dot.edge(str(parent), str(i))
        return dot
```

Метод `insert` добавляет новый элемент в конец списка `heap_list`. Это соответствует добавлению вершины на следующую доступную позицию в полном бинарном дереве. После добавления вызывается метод `perc_up`, задача которого — восстановить порядок, характерный для минимальной кучи.

Метод `perc_up` реализует механизм просеивания вверх. Переданный ему индекс нового элемента используется для сравнения значения с родительским узлом. Если нарушено условие кучи — то есть если элемент меньше своего родителя, — происходит обмен значениями. Индекс обновляется на позицию родителя, и цикл продолжается. Как только достигается вершина или порядок оказывается верным, просеивание завершает работу. Таким образом, каждый новый элемент поднимается до нужного уровня, не нарушая структуры дерева.

Метод `visualize` предназначен для графического представления текущей структуры дерева. Используя библиотеку `graphviz`, он создаёт вершины и рёбра по массиву `heap_list`, начиная с индекса 1. Индексы используются исключительно для внутренних ссылок и не отображаются. Каждому элементу сопоставляется вершина, а связи между родителями и потомками определяются на основе известной формулы: родитель узла с индексом `i` — это узел с индексом `i // 2`.

Описанные методы обеспечивают корректное добавление значений в кучу с сохранением всех её свойств. Вставка происходит за логарифмическое время, пропорциональное высоте дерева, и не требует перестройки всех элементов. Такой подход делает двоичную кучу эффективной структурой для приоритетного хранения данных.

После реализации базовых методов инициализации и вставки возникает необходимость построить кучу на основе произвольного списка чисел. Такая операция позволяет создать рабочую структуру за линейное время, не прибегая к последовательной вставке каждого элемента. Вместо этого используется механизм восстановления инварианта кучи, начиная с последних непустых поддеревьев и продвигаясь вверх к корню. Данный подход известен как ***восстановление снизу*** и обладает временной сложностью O(n).

Ниже реализован метод `build_heap`, который будет добавлен в класс `BinaryHeap`. Он принимает на вход список данных, копирует его в структуру кучи (добавляя служебный элемент `0` в начало), устанавливает актуальный размер и затем вызывает перестройку — начиная с последнего родительского узла.

```
def build_heap(self, alist):
    self.current_size = len(alist)
    self.heap_list = [0] + alist[:]
    i = self.current_size // 2
    while i > 0:
        self.perc_down(i)
        i = i - 1
```

Для корректной работы метода используется вспомогательная процедура `perc_down`, которая реализует просеивание элемента вниз по дереву. Эта операция необходима при нарушении инварианта кучи — когда значение в родительской вершине больше значения у одного из потомков.

```
def perc_down(self, i):
    while (i * 2) <= self.current_size:
        mc = self.min_child(i)
        if self.heap_list[i] > self.heap_list[mc]:
            self.heap_list[i], self.heap_list[mc] = self.heap_list[mc], self.heap_list[i]
        i = mc
def min_child(self, i):
    if i * 2 + 1 > self.current_size:
        return i * 2
    else:
        if self.heap_list[i * 2] < self.heap_list[i * 2 + 1]:
            return i * 2
        else:
            return i * 2 + 1
```

Метод `perc_down` реализует механизм восстановления свойств минимальной кучи, начиная с заданной вершины и двигаясь вниз по дереву. Его задача — переместить элемент на ту позицию, где он не нарушает инвариант: значение родителя должно быть меньше или равно значений потомков. На каждом шаге определяется меньший из двух дочерних элементов. Если значение в родительской вершине превышает значение меньшего потомка, они обмениваются местами. После обмена указатель сдвигается на уровень ниже, и процесс повторяется. Таким образом, элемент «просачивается» вниз до тех пор, пока не окажется в подходящем положении либо не достигнет нижнего уровня дерева.

Метод `min_child` используется как вспомогательная процедура, возвращающая индекс меньшего из двух потомков заданной вершины. Если у текущего узла только один потомок (что возможно на последнем или предпоследнем уровне), возвращается индекс единственного доступного ребёнка. Если оба потомка присутствуют, выбирается тот, чьё значение минимально. Это гарантирует, что при восстановлении порядка в `perc_down` родитель сравнивается именно с тем потомком, чей ключ должен

быть выше в структуре минимальной кучи. Такое поведение обеспечивает корректное продвижение элемента и предотвращает лишние перестроения.

Все эти компоненты объединяются в одном классе `BinaryHeap`:

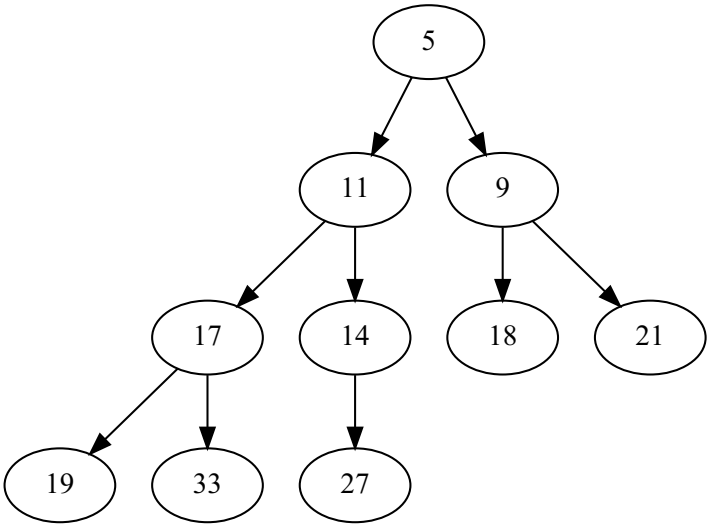
In [ ]:
```
from graphviz import Digraph
```

In [ ]:
```
class BinaryHeap:
    def __init__(self):
        self.heap_list = [0]
        self.current_size = 0

    def insert(self, k):
        self.heap_list.append(k)
        self.current_size = self.current_size + 1
        self.perc_up(self.current_size)

    def perc_up(self, i):
        while i // 2 > 0:
            if self.heap_list[i] < self.heap_list[i // 2]:
                self.heap_list[i], self.heap_list[i // 2] = self.heap_list[i // 2], self.heap_list[i]
            i = i // 2

    def perc_down(self, i):
        while (i * 2) <= self.current_size:
            mc = self.min_child(i)
            if self.heap_list[i] > self.heap_list[mc]:
                self.heap_list[i], self.heap_list[mc] = self.heap_list[mc], self.heap_list[i]
            i = mc

    def min_child(self, i):
        if i * 2 + 1 > self.current_size:
            return i * 2
        else:
            if self.heap_list[i * 2] < self.heap_list[i * 2 + 1]:
                return i * 2
            else:
                return i * 2 + 1

    def build_heap(self, alist):
        self.current_size = len(alist)
        self.heap_list = [0] + alist[:]
        i = self.current_size // 2
        while i > 0:
            self.perc_down(i)
            i = i - 1

    def visualize(self):
        dot = Digraph()
        n = len(self.heap_list) - 1
        for i in range(1, n + 1):
            dot.node(str(i), str(self.heap_list[i]))
            parent = i // 2
            if parent > 0:
                dot.edge(str(parent), str(i))
        return dot
```

Теперь можно перейти к примеру использования. Пусть задан список чисел `[27, 11, 9, 33, 5, 18, 21, 19, 17, 14]`, из которого требуется построить минимальную кучу. Для этого создаётся экземпляр класса, вызывается метод `build_heap`, и отображается результат:

In [ ]:
```
h = BinaryHeap()
h.build_heap([27, 11, 9, 33, 5, 18, 21, 19, 17, 14])
h.visualize()
```

Out[ ]:



В результате выполнения построена минимальная куча, в которой каждый родительский узел не превышает значений потомков. На вершине находится минимальный элемент, а все остальные значения расположены таким образом, чтобы сохранялась структура полного бинарного дерева и соблюдался инвариант кучи.

Удаление корневого элемента из двоичной кучи — одна из ключевых операций, обеспечивающих доступ к элементу с наивысшим приоритетом. В контексте минимальной кучи это означает, что удаляется наименьшее значение, которое всегда находится в корне. После удаления необходимо восстановить как структуру полного бинарного дерева, так и свойства упорядоченности, присущие двоичной куче.

На первом этапе значение, хранящееся в корневом узле, сохраняется в отдельной переменной — оно будет возвращено в качестве результата операции. Далее производится замена: на место удалённого корня помещается последний элемент в куче. Это позволяет сохранить форму полного бинарного дерева, так как удаление происходит с правого края нижнего уровня, а не из середины дерева. Однако после такой замены свойство упорядоченности, как правило, оказывается нарушено. Новый корневой элемент, будучи произвольным, может оказаться больше своих потомков. Для восстановления порядка запускается процесс просеивания вниз.

Механизм просеивания заключается в последовательном продвижении элемента вниз по дереву с помощью локальных обменов. На каждом шаге текущий элемент сравнивается с обоими своими потомками. Если он меньше или равен обоим — процесс завершён. В противном случае выбирается наименьший из двух потомков, и если его значение меньше, происходит обмен. Затем текущий индекс перемещается к этому потомку, и процедура повторяется. Таким образом, элемент опускается до той позиции, на которой порядок снова оказывается соблюдён. Важно, что вся операция происходит вдоль одного пути от корня к листу и затрагивает не более одного узла на каждом уровне. Благодаря этому сложность процесса остаётся в пределах O(logn), где n — общее число элементов в куче.

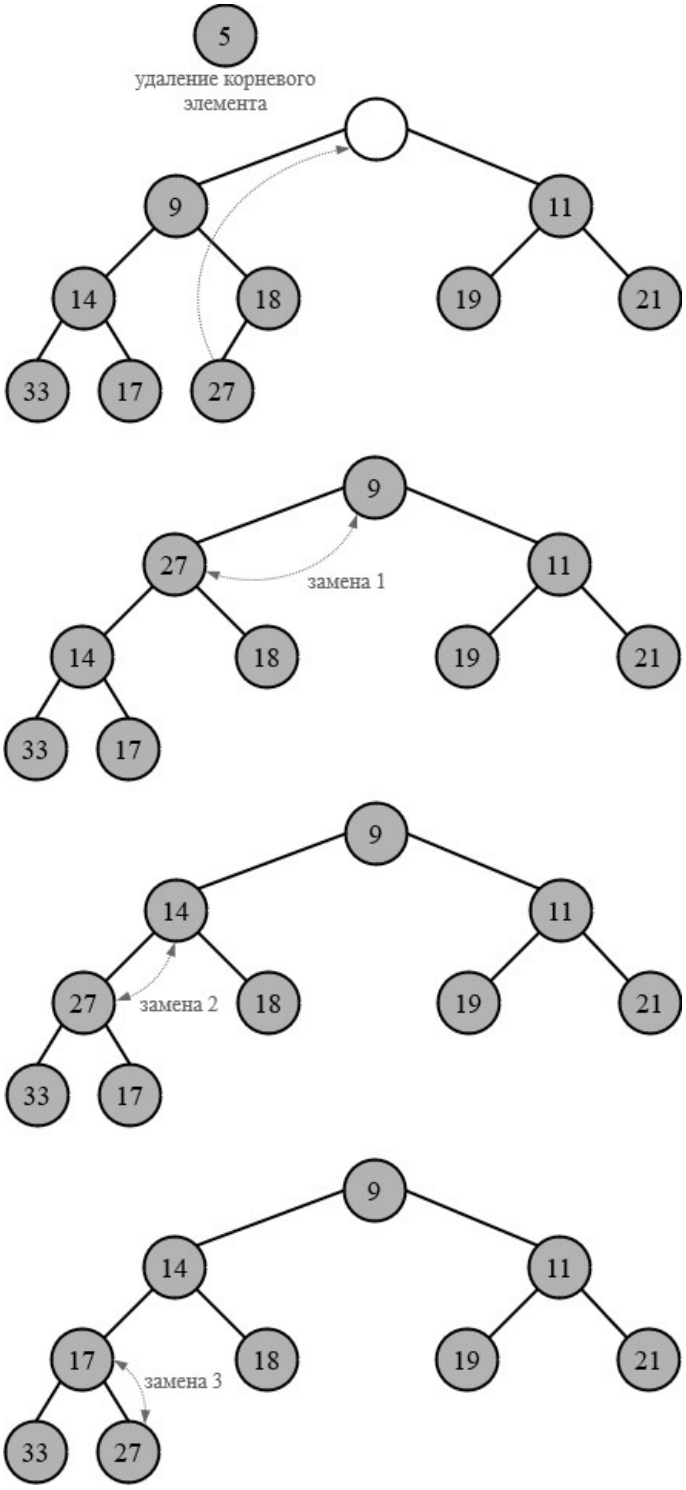


Рисунок 22 — Удаление корневого элемента из двоичной кучи

На рис. 22 показан пошаговый пример удаления минимального элемента из двоичной кучи. В исходном состоянии корень дерева содержит значение `5`, которое подлежит удалению. Последним элементом в дереве на этом этапе является `27`, расположенный в правом поддереве у вершины `18`. Именно он занимает освободившуюся позицию корня. Это немедленно приводит к нарушению инварианта: новое значение в корне больше своих потомков `9` и `11`. Минимальным из них является `9`, поэтому `27` и `9` обмениваются местами.

После первой замены `27` перемещается в левое поддерево, где его новыми потомками становятся `14` и `18`. Порядок снова нарушен — оба потомка меньше, и снова выбирается наименьший из них, то есть `14`. Ещё один обмен, и `27` оказывается в поддереве, где под ним расположены `33` и `17`. В третий раз условие упорядоченности не выполняется, поскольку `17` меньше `27`, и происходит последняя перестановка. После неё значение `27` оказывается в вершине, где оно не нарушает свойства кучи, завершив процесс восстановления.

Таким образом, операция удаления минимального элемента объединяет в себе два механизма: сохранение формы дерева через замену и удаление последней вершины, и восстановление порядка через локальные перестановки вдоль одного нисходящего пути. Это делает её устойчивой и эффективной при любом количестве элементов.

Рассмотрим реализацию удаления корневого элемента, то есть извлечения минимального значения из кучи с последующим восстановлением её структуры. Все действия, необходимые для корректного удаления, уже были подробно рассмотрены выше. Теперь необходимо описать, как эта операция реализуется программно в рамках класса `BinaryHeap`.

Удаление корня сопровождается четырьмя основными действиями: сначала сохраняется значение в корневом узле, затем на его место перемещается последний элемент, последний элемент удаляется из списка, и запускается перестройка дерева, начиная с вершины. Эти шаги реализуются в методе `del_min`:

```
def del_min(self):
    min_val = self.heap_list[1]
    self.heap_list[1] = self.heap_list[self.current_size]
    self.current_size = self.current_size - 1
    self.heap_list.pop()
    self.perc_down(1)
    return min_val
```

Метод получает доступ к первому значимому элементу массива `heap_list[1]`, который соответствует корню дерева, и сохраняет его в переменную `min_val`. Далее этот элемент будет возвращён вызывающей стороне. После этого последний элемент кучи копируется в позицию корня. Размер структуры уменьшается на единицу, а последний элемент физически удаляется из списка. Поскольку на место корня попал элемент, который, скорее всего, нарушает упорядоченность, вызывается метод `perc_down`, который просеивает его вниз до корректного положения. Таким образом, структура кучи восстанавливается, и операция завершается возвратом ранее удалённого значения.

Приведём целиком класс `BinaryHeap`, включающий все ранее реализованные методы: инициализацию, вставку, построение кучи, просеивание вверх и вниз, удаление минимального элемента, а также визуализацию.

In [ ]:
```
from graphviz import Digraph
```

In [ ]:
```
class BinaryHeap:
    def __init__(self):
        self.heap_list = [0]
        self.current_size = 0

    def insert(self, k):
        self.heap_list.append(k)
        self.current_size += 1
        self.perc_up(self.current_size)

    def perc_up(self, i):
        while i // 2 > 0:
            if self.heap_list[i] < self.heap_list[i // 2]:
                self.heap_list[i], self.heap_list[i // 2] = self.heap_list[i // 2], self.heap_list[i]
            i = i // 2

    def perc_down(self, i):
        while (i * 2) <= self.current_size:
            mc = self.min_child(i)
            if self.heap_list[i] > self.heap_list[mc]:
                self.heap_list[i], self.heap_list[mc] = self.heap_list[mc], self.heap_list[i]
            i = mc

    def min_child(self, i):
        if i * 2 + 1 > self.current_size:
            return i * 2
        else:
            if self.heap_list[i * 2] < self.heap_list[i * 2 + 1]:
                return i * 2
            else:
                return i * 2 + 1

    def build_heap(self, alist):
        self.current_size = len(alist)
        self.heap_list = [0] + alist[:]
        i = self.current_size // 2
        while i > 0:
            self.perc_down(i)
            i -= 1

    def del_min(self):
        min_val = self.heap_list[1]
        self.heap_list[1] = self.heap_list[self.current_size]
        self.current_size -= 1
        self.heap_list.pop()
        self.perc_down(1)
        return min_val

    def visualize(self):
        dot = Digraph()
        n = len(self.heap_list) - 1
        for i in range(1, n + 1):
            dot.node(str(i), str(self.heap_list[i]))
            parent = i // 2
            if parent > 0:
                dot.edge(str(parent), str(i))
```

```
        return dot
```

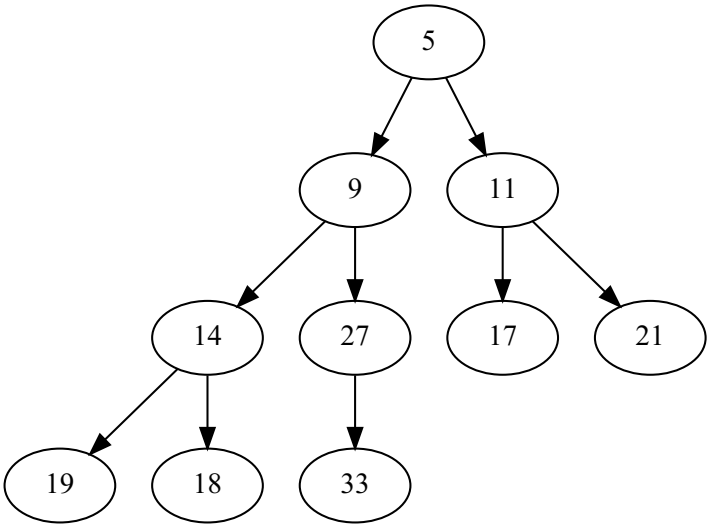
Пример использования:

In [ ]:
```
from random import shuffle
```

In [ ]:
```
elements = [5, 9, 11, 14, 18, 19, 21, 33, 17, 27]
shuffle(elements)
```

In [ ]:
```
heap = BinaryHeap()
heap.build_heap(elements)
heap.visualize()
```

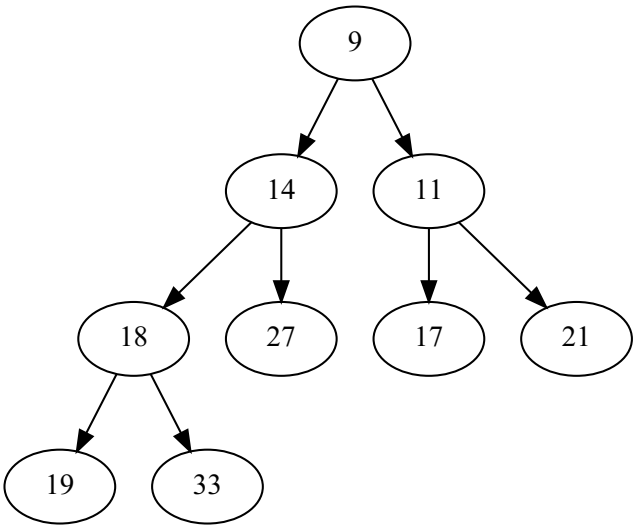
Out[ ]:



In [ ]:
```
min_val = heap.del_min()
print(f"Удалён корневой элемент: {min_val}")
heap.visualize()
```

```
Удалён корневой элемент: 5
```

Out[ ]:



После удаления корневого элемента `5` его место занял последний элемент — `33`, который затем был просеян вниз: сначала обменялся с `9`, затем с `14`, а затем с `18`, пока не оказался на позиции, где свойства кучи восстановлены. В результате новый корень — `9`, а структура по-прежнему соответствует правилам полного бинарного дерева и условиям минимальной кучи.

Переход от минимальной кучи к максимальной не требует изменения структуры хранения или порядка операций, но требует переосмысления логики сравнения при просеивании элементов вверх и вниз. В минимальной куче каждый родитель должен быть меньше или равен своих потомков, в то время как в максимальной — наоборот: значение в каждой вершине должно быть больше или равно значений в её дочерних узлах. Это означает, что при вставке элемент должен подниматься вверх по дереву только в том случае, если он больше своего родителя, а при удалении корня — опускаться вниз, пока он больше не окажется хотя бы одного из своих потомков.

В коде это приводит к замене операторов сравнения в двух основных методах: `perc_up` и `perc_down`. Вместо проверки «меньше» используется проверка «больше». Кроме того, метод `min_child`, определяющий меньшего из двух потомков, заменяется на `max_child`, который возвращает большего. Метод удаления `del_min` логично переименовать в `del_max`, но его структура остаётся прежней: в корень ставится последний элемент, и порядок восстанавливается за счёт просеивания вниз. Таким образом, все изменения сводятся к корректировке логики сравнения, без необходимости переписывать архитектуру класса.

Двоичная куча — это эффективная структура данных для организации приоритетного доступа, позволяющая в логарифмическое время добавлять элементы, удалять корневой узел и поддерживать упорядоченность. Благодаря строгой форме полного бинарного дерева и локальному инварианту, куча объединяет компактность, скорость и простоту реализации. Её можно представить в виде обычного массива, что упрощает программную реализацию и делает структуру удобной для использования в различных алгоритмах: от планирования задач до поиска кратчайших путей.

Закрепим понимание этой структуры на практических задачах. В реальных системах куча используется не как абстрактный набор чисел, а как инструмент для работы с данными, обладающими дополнительными признаками — временем, приоритетом, именем, координатами, метками. Именно в таких ситуациях важным становится умение адаптировать классическую реализацию к конкретному прикладному контексту.

Рассмотрим практические примеры, которые помогут лучше понять, как использовать двоичную кучу в реальных задачах. Каждое решение будет основываться на том классе `BinaryHeap`, который был подробно рассмотрен выше.

Пример 1. Пусть необходимо реализовать простую систему, которая отслеживает заряд подключённых устройств (например, телефонов или ноутбуков в компьютерном классе). Каждое устройство характеризуется идентификатором и текущим уровнем заряда (в процентах). Система должна поддерживать следующие операции:

- добавление нового устройства с текущим уровнем заряда;
- обновление заряда существующего устройства;
- извлечение устройства с наименьшим уровнем заряда (то есть с наивысшим приоритетом на подзарядку).

Реализация будет опираться на минимальную кучу, где ключом служит уровень заряда, а значением — уникальный идентификатор устройства. При обновлении заряда потребуется удалить старую запись и добавить новую. В качестве примера будем использовать строковые идентификаторы и целочисленные значения заряда.

Создадим класс `BatteryMonitor`, который использует двоичную кучу как внутренний механизм:

In [ ]:
```python
class BatteryMonitor:
    def __init__(self):
        self.heap = BinaryHeap()
        self.device_map = {}

    def add_device(self, device_id, charge):
        self.heap.insert((charge, device_id))
        self.device_map[device_id] = charge

    def update_charge(self, device_id, new_charge):
        old_charge = self.device_map.get(device_id)
        if old_charge is not None:
            filtered = [
                (c, d) for (c, d) in self.heap.heap_list[1:]
                if d != device_id
                ]
            self.heap = BinaryHeap()
            self.heap.build_heap(filtered)
        self.add_device(device_id, new_charge)

    def get_lowest_charge(self):
        if self.heap.current_size == 0:
            return None
        charge, device_id = self.heap.del_min()
        del self.device_map[device_id]
        return device_id, charge
```

Класс `BatteryMonitor` представляет собой прикладную обёртку над двоичной кучей, адаптированную под задачу отслеживания устройств и их уровня заряда. Он построен таким образом, чтобы опираться на реализованный выше класс `BinaryHeap`, при этом дополняя его возможностью адресного доступа к устройствам по идентификатору. Вся структура организована так, чтобы при необходимости можно было эффективно извлекать устройство с наименьшим зарядом, а также обновлять информацию об уже существующих элементах.

Метод `__init__` создаёт новую кучу и словарь. В кучу будут помещаться пары, состоящие из уровня заряда и идентификатора устройства, а словарь будет использоваться для хранения актуальных значений заряда для каждого идентификатора. Такой подход позволяет решить две задачи одновременно: поддерживать порядок по приоритету и знать, какие значения привязаны к каким устройствам.

Добавление устройства выполняется методом `add_device`, который принимает идентификатор и уровень заряда. Внутри метода пара `(charge, device_id)` вставляется в кучу, после чего в словарь записывается уровень заряда по ключу `device_id`. Куча сохраняет приоритет по заряду, а словарь позволяет отслеживать наличие устройства и его текущие параметры.

Более сложной является операция обновления заряда. Метод `update_charge` сначала проверяет, существует ли указанное устройство. Если оно есть в системе, значит, в кучу уже была добавлена пара с устаревшим значением. Так как стандартная

реализация двоичной кучи не предусматривает прямого обновления элемента, проще всего удалить старую версию и построить кучу заново. Для этого из текущего списка кучи исключаются все элементы, у которых идентификатор совпадает с переданным, после чего оставшиеся элементы заново подаются в метод `build_heap`. Затем вызывается `add_device`, который вставляет новую версию записи. Этот приём, несмотря на перерасчёт кучи, остаётся приемлемым по трудоёмкости при умеренном числе элементов и позволяет сохранить логическую целостность структуры.

Метод `get_lowest_charge` извлекает из кучи элемент с минимальным зарядом. Если куча пуста, возвращается `None`. В противном случае вызывается `del_min`, и возвращается кортеж из идентификатора и значения заряда. После этого соответствующая запись удаляется и из словаря, чтобы не сохранять лишние данные. Важно отметить, что возвращается именно та пара, которая в текущий момент имеет наименьший заряд, независимо от порядка добавления.

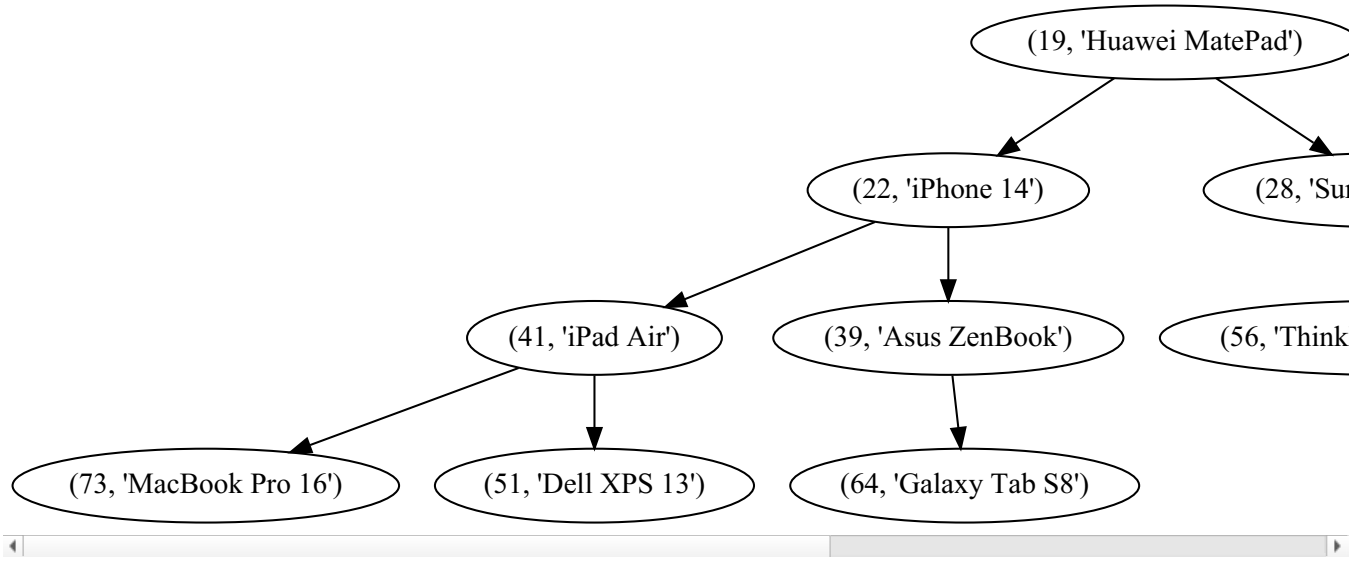
Пример использования:

In [ ]:
```
monitor = BatteryMonitor()
```

In [ ]:
```
monitor.add_device("MacBook Pro 16", 73)
monitor.add_device("iPad Air", 41)
monitor.add_device("ThinkPad X1 Carbon", 56)
monitor.add_device("Surface Laptop", 28)
monitor.add_device("Galaxy Tab S8", 64)
monitor.add_device("iPhone 14", 22)
monitor.add_device("Redmi Note 11", 35)
monitor.add_device("Dell XPS 13", 51)
monitor.add_device("Asus ZenBook", 39)
monitor.add_device("Huawei MatePad", 19)
```

In [ ]:
```
monitor.heap.visualize()
```

Out[ ]:

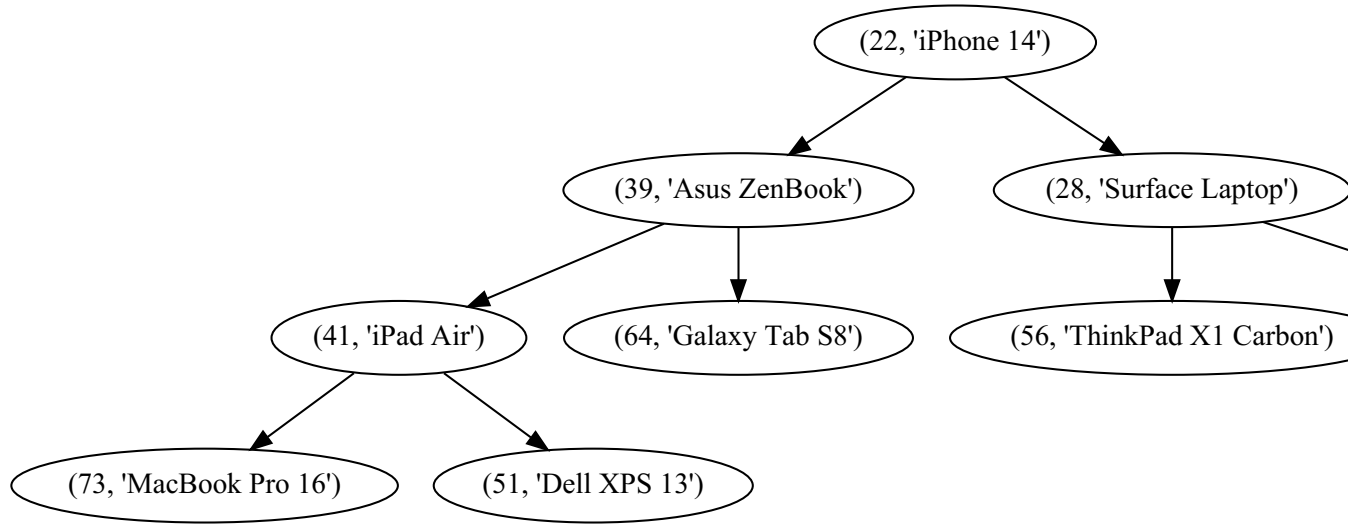


In [ ]:
```
print(monitor.get_lowest_charge())
```

```
('Huawei MatePad', 19)
```

In [ ]:
```
monitor.heap.visualize()
```

Out[ ]:

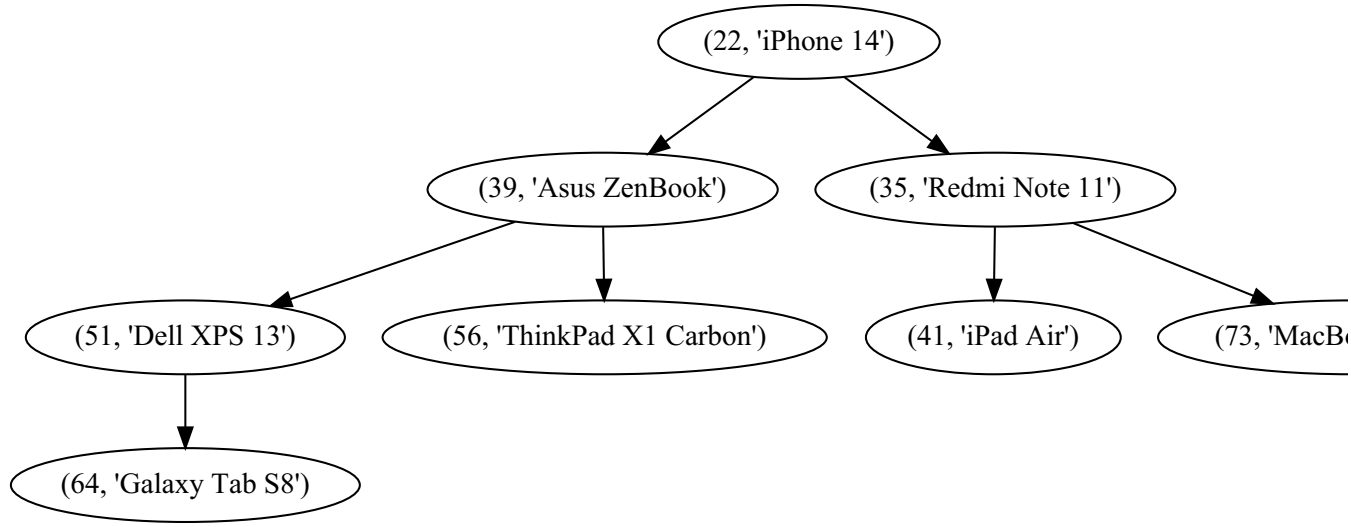


In [ ]:
```
monitor.update_charge("Surface Laptop", 15)
print(monitor.get_lowest_charge())
```

```
('Surface Laptop', 15)
```

In [ ]:
```
monitor.heap.visualize()
```

Out[ ]:



Этот пример иллюстрирует, как с помощью двоичной кучи можно реализовать систему мониторинга с приоритетным обслуживанием, где приоритет напрямую связан с числовыми значениями, и как обойти отсутствие прямой модификации в куче через перестроение.

Пример 2. Пусть необходимо реализовать систему отслеживания срочности задач в рабочей среде. Каждая задача имеет краткое описание, уровень приоритета (меньшее число — выше приоритет) и временную метку постановки (например, в формате `datetime`). Система должна поддерживать следующие операции:

- добавление новой задачи с указанным приоритетом и временем поступления;
- извлечение задачи с наивысшим приоритетом среди тех, что уже поступили к текущему моменту.

Важной особенностью является то, что не каждая задача сразу доступна к исполнению: система должна учитывать только те задачи, у которых момент поступления уже наступил. Таким образом, куча будет упорядочена сначала по приоритету, а затем — по времени. Это делает задачу ближе к реальной модели планировщика, где учитывается как важность задачи, так и допустимое время начала её выполнения.

Создадим класс `TaskScheduler`, который будет использовать двоичную кучу для хранения задач в виде троек: `(priority, timestamp, description)`.

In [ ]:
```
from datetime import datetime
import time
```

In [ ]:
```
class TaskScheduler:
    def __init__(self):
        self.heap = BinaryHeap()

    def add_task(self, description, priority, timestamp):
        self.heap.insert((priority, timestamp, description))

    def get_next_task(self):
        now = datetime.now()
        temp = []

        result = None
        while self.heap.current_size > 0:
            top = self.heap.del_min()
            if top[1] <= now:
                result = top
                break
            else:
                temp.append(top)

        for task in temp:
            self.heap.insert(task)

        return result
```

Метод `add_task` помещает в кучу новую тройку `(priority, timestamp, description)`, где `priority` — это целое число (чем меньше, тем важнее), `timestamp` — объект типа `datetime`, а `description` — произвольная строка. Кортежи сравниваются по элементам слева направо, поэтому порядок извлечения будет соответствовать приоритету, а при равенстве — времени поступления.

Метод `get_next_task` извлекает первую доступную по времени задачу. Он просматривает вершину кучи и проверяет, не наступил ли момент исполнения. Если задача уже доступна, она возвращается. Если время ещё не пришло, задача временно извлекается, а позже возвращается обратно в кучу. Таким образом, сохраняется порядок, и задачи не теряются. Если подходящей задачи нет, возвращается `None`.

Пример использования:

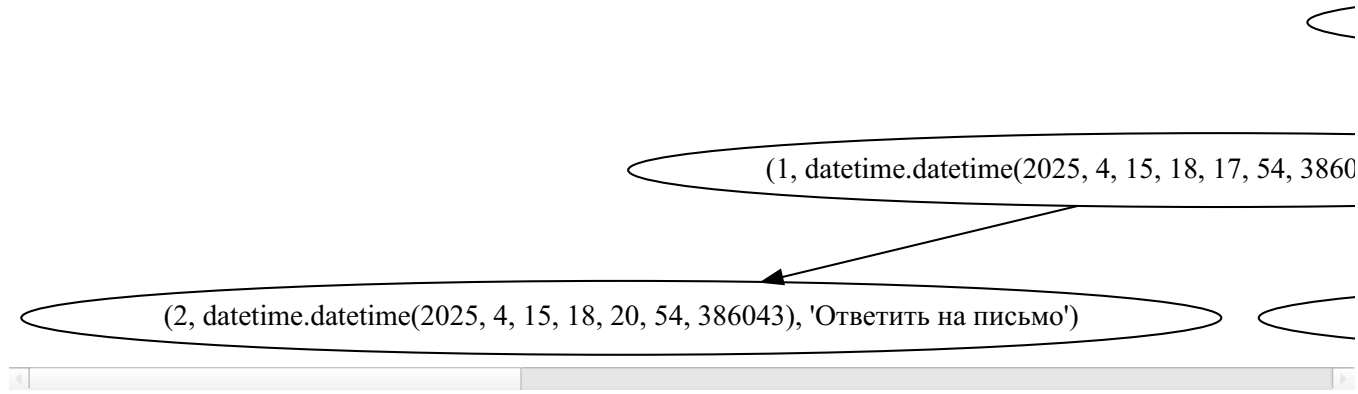
In [ ]:
```
from datetime import datetime, timedelta
```

In [ ]:
```
scheduler = TaskScheduler()
```

In [ ]:
```
now = datetime.now()
scheduler.add_task("Сделать отчёт", 1, now - timedelta(minutes=10))
scheduler.add_task("Ответить на письмо", 2, now + timedelta(minutes=5))
scheduler.add_task("Обновить документацию", 3, now)
scheduler.add_task("Созвон с командой", 1, now + timedelta(minutes=2))
scheduler.add_task("Проверить код", 2, now - timedelta(minutes=1))
```

In [ ]:
```
scheduler.heap.visualize()
```

Out[ ]:

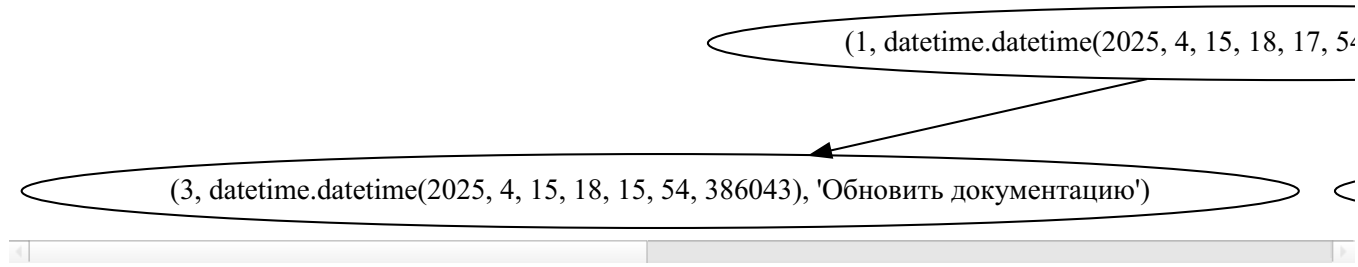


In [ ]:
```
print(scheduler.get_next_task())
time.sleep(1)
print(scheduler.get_next_task())
```

```
(1, datetime.datetime(2025, 4, 15, 18, 5, 54, 386043), 'Сделать отчёт')
(2, datetime.datetime(2025, 4, 15, 18, 14, 54, 386043), 'Проверить код')
```

In [ ]:
```
scheduler.heap.visualize()
```

Out[ ]:



В этом примере в систему добавляются задачи с разными приоритетами и временем поступления. При вызове метода `get_next_task` будет возвращаться наиболее приоритетная задача из тех, чьё время уже наступило. Остальные останутся в очереди до момента, когда их можно будет исполнять. Благодаря использованию двоичной кучи обеспечивается быстрый доступ к нужной задаче, а структура остаётся устойчивой даже при множестве элементов и частых обновлениях.

Такой пример показывает, как можно использовать кучу не только для приоритетов, но и для двойного критерия сортировки, сочетая числовой приоритет и временные ограничения, что делает его близким к задачам обработки событий и управлению задачами в реальных системах.

Оба примера показывают, как двоичная куча может быть адаптирована для решения прикладных задач, в которых важны приоритет и порядок обработки. В первом случае она помогает отслеживать устройства с минимальным зарядом, во втором — управлять задачами с учётом приоритета и времени поступления.

После подробного рассмотрения двоичной кучи как структурф, обеспечивающей приоритетный доступ к элементам, логично перейти к **очереди с приоритетом**. Эта структура представляет собой обобщение обычной очереди, в которой порядок извлечения определяется не временем поступления, а значением приоритета. Каждый элемент в такой очереди сопровождается приоритетом, и в любой момент доступен элемент с наивысшим приоритетом — то есть минимальным или максимальным значением в зависимости от конкретной реализации.

Очередь с приоритетом удобно реализовать с использованием двоичной кучи, поскольку свойства кучи полностью соответствуют требованиям: минимальный (или максимальный) элемент всегда находится в корне, операции вставки и удаления выполняются за логарифмическое время, а структура остаётся компактной и сбалансированной. Внутри кучи элемент очереди может быть представлен в виде пары, где одно значение определяет приоритет, а другое содержит собственно содержимое элемента — задачу, объект, запись и т. д.

При добавлении нового элемента он помещается в конец массива, и запускается операция подъёма, которая переставляет элемент вверх по дереву, пока он не займёт корректное место в соответствии с приоритетом. При извлечении элемента с наивысшим приоритетом — то есть при обращении к корню — он удаляется, на его место перемещается последний элемент, и выполняется операция опускания, которая восстанавливает упорядоченность вдоль пути от вершины к листьям. Такая организация обеспечивает устойчивую работу при постоянных обновлениях очереди.

Кроме вставки и удаления, в очередь с приоритетом может быть добавлена операция получения элемента с наивысшим приоритетом без удаления, что просто сводится к доступу к корню кучи. Также может потребоваться изменение приоритета уже существующего элемента. Поскольку двоичная куча не предоставляет прямого способа изменить значение в произвольной вершине, на практике это реализуется через удаление старого элемента и повторную вставку с новым приоритетом.

Очередь с приоритетом, построенная на двоичной куче, является универсальным инструментом, который лежит в основе многих алгоритмов: планировщиков, симуляторов, маршрутизаторов, поисковых процедур и многого другого.

Рассмотрим класс `PriorityQueue`, который использует ранее реализованную структуру `BinaryHeap` в качестве внутреннего механизма хранения:

In [ ]:
```
class PriorityQueue:
    def __init__(self, items=None):
        self.bh = BinaryHeap()
        if items:
            self.bh.build_heap(items)

    def enqueue(self, item):
        self.bh.insert(item)

    def dequeue(self):
        return self.bh.del_min()

    def peek(self):
        if self.is_empty():
            return None
        return self.bh.heap_list[1]

    def is_empty(self):
        return self.bh.current_size == 0

    def size(self):
        return self.bh.current_size

    def visualize(self):
        return self.bh.visualize()
```

Класс `PriorityQueue` инкапсулирует базовые операции, характерные для очереди с приоритетом, и делает их доступными через соответствующие методы. Метод `__init__` позволяет создать пустую очередь или инициализировать её заранее подготовленным списком. Метод `enqueue` добавляет элемент с приоритетом, представленный в виде кортежа, в котором первым значением указывается числовой приоритет, а вторым — полезная нагрузка. Вставка осуществляется путём добавления элемента в кучу и просеивания вверх, как это реализовано ранее.

Метод `dequeue` извлекает элемент с наивысшим приоритетом, то есть минимальный элемент кучи, — это корень дерева. После удаления корня структура восстанавливается через механизм просеивания вниз. Метод `peek` позволяет получить доступ к корню без удаления. Методы `is_empty` и `size` возвращают информацию о текущем состоянии очереди. Дополнительно добавлен метод `visualize`, возвращающий объект визуализации текущей кучи, что позволяет наглядно отслеживать изменения структуры.

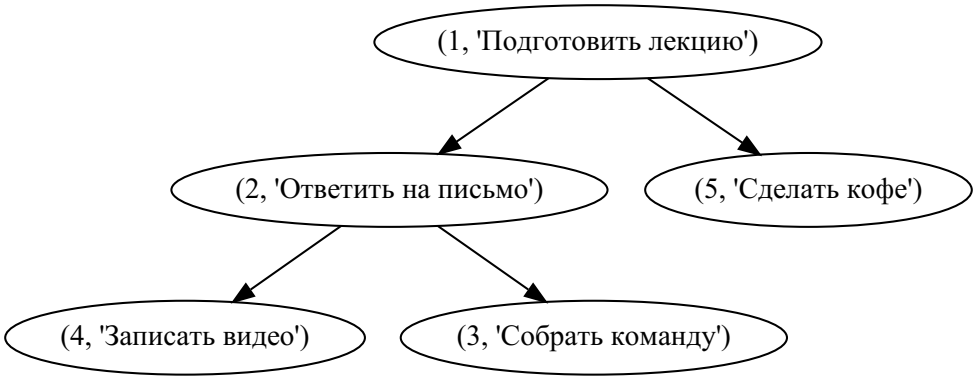
Пример использования очереди с приоритетом:

In [ ]:
```
pq = PriorityQueue()
```

In [ ]:
```
pq.enqueue((4, "Записать видео"))
pq.enqueue((2, "Ответить на письмо"))
pq.enqueue((5, "Сделать кофе"))
pq.enqueue((1, "Подготовить лекцию"))
pq.enqueue((3, "Собрать команду"))
```

In [ ]:
```
pq.visualize()
```

Out[ ]:

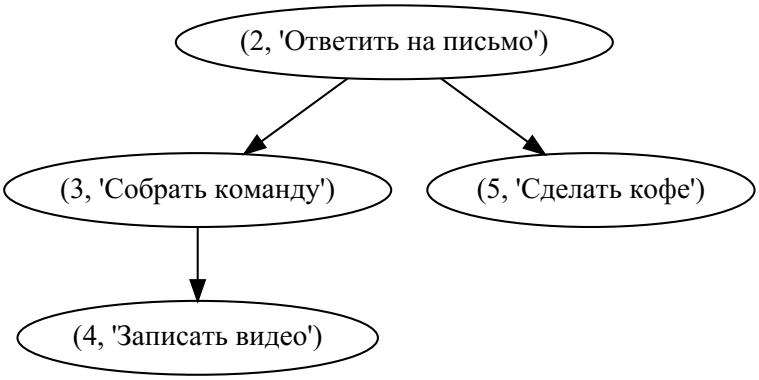


In [ ]:
```
print(pq.dequeue())
print(pq.peek())
```

```
(1, 'Подготовить лекцию')
(2, 'Ответить на письмо')
```

In [ ]:
```
pq.visualize()
```

Out[ ]:



В этом примере в очередь поочерёдно добавляются задачи с приоритетами от 1 до 5. Извлечение происходит в порядке приоритета: сначала выбирается задача «Подготовить лекцию», имеющая наименьший приоритетный индекс, затем — следующая по значению. Структура кучи перестраивается автоматически, сохраняя свойства минимальной кучи и позволяя быстро получать наиболее приоритетную задачу. Метод `visualize` на любом этапе позволяет построить дерево и увидеть его текущее состояние.

В Python существует стандартная библиотека `heapq`, предназначенная для работы с кучами. Эта библиотека реализует эффективные алгоритмы, основанные на двоичной куче, и позволяет использовать её как основу для построения очереди с приоритетом. Несмотря на простоту интерфейса, за методами `heapq` скрыта классическая реализация минимальной кучи, где каждый родительский элемент не превышает значения своих потомков. Это делает библиотеку пригодной для задач, где необходимо быстро извлекать минимальные элементы и поддерживать структуру приоритетного доступа.

Вся работа с кучей в `heapq` строится вокруг обычных списков. В момент инициализации не требуется создавать отдельный класс или структуру: достаточно передать список в функции из модуля `heapq`, и он будет интерпретироваться как куча. Важно, чтобы элементы добавлялись и извлекались исключительно с помощью предоставленных методов, иначе инвариант кучи будет нарушен.

Основные функции библиотеки:

- `heapq.heapify(list)` преобразует неупорядоченный список в минимальную кучу за время O(n). Все элементы остаются в том же списке, который теперь удовлетворяет свойствам кучи.
- `heapq.heappush(heap, item)` добавляет элемент `item` в кучу, сохраняя её свойства. Вставка осуществляется в конец списка с последующим просеиванием вверх.
- `heapq.heappop(heap)` извлекает и возвращает минимальный элемент (то есть корень кучи), перестраивая оставшуюся структуру. Операция выполняется за O(logn).
- `heapq.heappushpop(heap, item)` вставляет элемент и затем сразу извлекает наименьший. Это более эффективно, чем вызов `heappush`, за которым следует `heappop`.
- `heapq.heapreplace(heap, item)` удаляет минимальный элемент и вставляет новый. В отличие от `heappushpop`, всегда извлекает минимум даже при условии, что новый элемент меньше.
- `heapq.nlargest(n, iterable)` и `heapq.nsmallest(n, iterable)` возвращают соответственно `n` наибольших или наименьших элементов из произвольной коллекции. Эти методы полезны, когда не требуется построение кучи вручную.

Поскольку `heapq` реализует минимальную кучу, для имитации максимальной можно использовать приёмы инверсии приоритета. Наиболее распространённый подход — хранить кортежи с приоритетами, умноженными на `-1`. Это сохраняет корректную структуру, но меняет направление сравнения.

Простейший пример использования кучи:

In [ ]:
```
import heapq
```

In [ ]:
```
heap = []
heapq.heappush(heap, 5)
heapq.heappush(heap, 3)
heapq.heappush(heap, 9)
heapq.heappush(heap, 1)
```

In [ ]:
```
print(heapq.heappop(heap))
```

```
1
```

В этом примере элементы добавляются в кучу, а затем извлекается наименьший элемент — `1`. Все операции выполняются за O(logn), так как сохраняется структура двоичной минимальной кучи.

Если требуется реализовать очередь с приоритетом, удобно использовать кортежи, в которых первый элемент определяет приоритет:

In [ ]:
```
tasks = []
heapq.heappush(tasks, (2, "Сделать отчёт"))
heapq.heappush(tasks, (1, "Подготовить лекцию"))
heapq.heappush(tasks, (3, "Сходить на встречу"))
```

In [ ]:
```
print(heapq.heappop(tasks))
```

```
(1, 'Подготовить лекцию')
```

Здесь задача с наименьшим числом приоритета (`1`) будет извлечена первой. Если приоритеты совпадают, элементы сравниваются по следующему полю — в данном случае по строке. Такая структура делает `heapq` удобной и гибкой при использовании в очередях с приоритетом.

Для имитации максимальной кучи, где приоритет задаётся наибольшим значением, можно сохранять отрицательные значения приоритета:

In [ ]:
```
max_heap = []
heapq.heappush(max_heap, (-5, "Пятая задача"))
heapq.heappush(max_heap, (-1, "Первая задача"))
heapq.heappush(max_heap, (-3, "Третья задача"))
```

In [ ]:
```
priority, task = heapq.heappop(max_heap)
print(-priority, task)
```

```
5 Пятая задача
```

Библиотека `heapq` обеспечивает базовый функционал для работы с приоритетами, и при правильной организации данных может быть адаптирована под разнообразные прикладные сценарии: от планирования задач до реализации алгоритмов Дейкстры и A*. Она обеспечивает лаконичный и эффективный способ интеграции кучи в любое приложение.

Таким образом, двоичная куча и основанная на ней очередь с приоритетом представляют собой не просто структуру для хранения элементов, а мощный механизм управления доступом по значимости. Их свойства (логарифмическая сложность операций, компактное представление, локальность инвариантов) делают эти структуры незаменимыми в задачах, где критично быстрое извлечение элементов с заданным приоритетом. Реализация с возможностью визуализации, а также практическое применение в прикладных классах и использование стандартной библиотеки `heapq` показывают, насколько легко эти идеи интегрируются в реальные проекты, обеспечивая не только эффективность, но и ясность логики обработки данных.