Очередь (Queue)

Очередь — это линейная структура данных, работающая по принципу "первый вошел, первый вышел" (FIFO, First In First Out). Это значит, что элементы добавляются в конец очереди и удаляются из начала.

Основные операции

- enqueue: Добавление элемента в конец очереди.
- dequeue: Удаление элемента из начала очереди.
- is_empty: Проверка, пуста ли очередь.
- peek: Просмотр элемента в начале очереди без его удаления.

Применение очередей

Обработка данных в реальном времени: Например, очереди сообщений в системах реального времени.

Управление задачами: Организация выполнения задач в порядке их поступления (например, принтерная очередь).

Асинхронное выполнение задач: Обработка событий и задач в асинхронных системах.

Пример реализации очереди

Давайте реализуем очередь на основе встроенного списка Python, а затем рассмотрим её использование.

Реализация очереди с использованием списка

```python
class Queue:
    def __init__(self):
        self.queue = []

    def enqueue(self, data):
        """Добавляет элемент в конец очереди."""
        self.queue.append(data)
        print(f"Enqueued: {data}")

    def dequeue(self):
        """Удаляет элемент из начала очереди."""
        if self.is_empty():
            return "Queue is empty"
        data = self.queue.pop(0)
        print(f"Dequeued: {data}")
        return data

    def is_empty(self):
        """Проверяет, пуста ли очередь."""
        return len(self.queue) == 0

    def peek(self):
        """Просматривает элемент в начале очереди без его удаления."""
        if self.is_empty():
            return "Queue is empty"
        return self.queue[0]

    def display(self):
        """Отображает текущую очередь."""
        print("Queue:", self.queue)

# Пример использования
q = Queue()
q.enqueue(1)
q.enqueue(2)
q.enqueue(3)
q.display()  # Вывод: Queue: [1, 2, 3]
print(q.peek())  # Вывод: 1
q.dequeue()
q.display()  # Вывод: Queue: [2, 3]
print(q.is_empty())  # Вывод: False
q.dequeue()
q.dequeue()
print(q.is_empty())  # Вывод: True
```

Пример 1: Очередь задач

Представьте себе очередь задач для принтера. Задачи добавляются в очередь в порядке поступления и выполняются также в этом порядке.

```
class PrinterQueue:
    def __init__(self):
        self.queue = []

    def add_job(self, job):
        """Добавляет задание в очередь печати."""
        self.queue.append(job)
        print(f"Job '{job}' added to the queue")

    def print_job(self):
        """Выполняет задание из начала очереди."""
        if self.is_empty():
            print("No jobs in the queue")
            return
        job = self.queue.pop(0)
        print(f"Printing job: {job}")

    def is_empty(self):
        """Проверяет, пуста ли очередь."""
        return len(self.queue) == 0

    def display_queue(self):
        """Отображает текущую очередь заданий."""
        print("Current print queue:", self.queue)

# Пример использования
printer_queue = PrinterQueue()
printer_queue.add_job("Document1.pdf")
printer_queue.add_job("Photo.jpg")
printer_queue.add_job("Report.docx")
printer_queue.display_queue()  # Вывод: Current print queue: ['Document1.pdf', 'Photo.jpg
printer_queue.print_job()  # Вывод: Printing job: Document1.pdf
printer_queue.display_queue()  # Вывод: Current print queue: ['Photo.jpg', 'Report.docx']
```

Пример 2: Обработка данных в реальном времени

Представим систему обработки данных, где данные поступают в очередь и обрабатываются по мере поступления.

```python
class DataQueue:
    def __init__(self):
        self.queue = []

    def add_data(self, data):
        """Добавляет данные в очередь."""
        self.queue.append(data)
        print(f"Data '{data}' added to the queue")

    def process_data(self):
        """Обрабатывает данные из начала очереди."""
        if self.is_empty():
            print("No data to process")
            return
        data = self.queue.pop(0)
        print(f"Processing data: {data}")

    def is_empty(self):
        """Проверяет, пуста ли очередь."""
        return len(self.queue) == 0

    def display_queue(self):
        """Отображает текущую очередь данных."""
        print("Current data queue:", self.queue)

# Пример использования
data_queue = DataQueue()
data_queue.add_data("Sensor1: 23.5°C")
data_queue.add_data("Sensor2: 24.0°C")
data_queue.add_data("Sensor3: 22.8°C")
data_queue.display_queue()  # Вывод: Current data queue: ['Sensor1: 23.5°C', 'Sensor2: 24
data_queue.process_data()  # Вывод: Processing data: Sensor1: 23.5°C
data_queue.display_queue()  # Вывод: Current data queue: ['Sensor2: 24.0°C', 'Sensor3: 22
```

Задания

1. **Система обработки заказов в ресторане**

Представьте, что вы пишете программу для управления заказами в ресторане. Заказы поступают в очередь и обрабатываются по мере их поступления. Реализуйте класс OrderQueue с методами add_order, process_order и display_orders.

2. **Очередь в банке**

Создайте класс BankQueue, который моделирует очередь клиентов в банке. Класс должен иметь методы для добавления клиента в очередь (enqueue), обслуживания клиента (dequeue) и просмотра следующего клиента в очереди (peek). Также добавьте метод для отображения текущего состояния очереди (display_queue).

3. **Обработка запросов к серверу**

Создайте систему обработки запросов к серверу. Запросы добавляются в очередь по мере их поступления, и сервер обрабатывает их по очереди. Напишите класс ServerQueue с методами для добавления запроса, обработки запроса и отображения текущей очереди запросов.

Деревья (Trees)

Дерево — это иерархическая структура данных, состоящая из узлов, где каждый узел содержит значение и указатели на дочерние узлы. Самый верхний узел называется корнем, а узлы без дочерних узлов — листьями.

Основные термины

- Корень (Root): Самый верхний узел дерева.
- Узел (Node): Элемент дерева, содержащий данные и указатели на дочерние узлы.
- Лист (Leaf): Узел без дочерних узлов.
- Родитель (Parent): Узел, который указывает на текущий узел.
- Дочерний узел (Child): Узел, на который указывает текущий узел.
- Высота (Height): Длина самого длинного пути от корня до листа.
- Глубина (Depth): Расстояние от корня до узла.

Типы деревьев

- **Бинарное дерево (Binary Tree)**: Каждый узел имеет не более двух дочерних узлов.
- **Бинарное дерево поиска (Binary Search Tree, BST)**: Левое поддерево содержит узлы с значениями меньше корневого узла, а правое поддерево — с значениями больше корневого узла.
- **Сбалансированное дерево (Balanced Tree)**: Дерево, в котором высота левого и правого поддеревьев каждого узла различается не более чем на единицу.
- **Куча (Heap)**: Дерево, в котором каждый узел имеет значение, большее или меньшее, чем значения его дочерних узлов (max-heap или min-heap).

Основные операции

- **Вставка (Insertion)**: Добавление узла в дерево.
- **Удаление (Deletion)**: Удаление узла из дерева.
- **Поиск (Search)**: Поиск узла по значению.
- **Обход дерева (Traversal)**: Посещение всех узлов дерева (прямой, симметричный, обратный обход).

Пример реализации бинарного дерева поиска (BST)

Узлы дерева:

```
1  class TreeNode:
2      def __init__(self, key):
3          self.left = None
4          self.right = None
5          self.val = key
```

Бинарное дерево поиска:

```python
class BinarySearchTree:
    def __init__(self):
        self.root = None

    def insert(self, key):
        if self.root is None:
            self.root = TreeNode(key)
        else:
            self._insert(self.root, key)

    def _insert(self, node, key):
        if key < node.val:
            if node.left is None:
                node.left = TreeNode(key)
            else:
                self._insert(node.left, key)
        else:
            if node.right is None:
                node.right = TreeNode(key)
            else:
                self._insert(node.right, key)

    def search(self, key):
        return self._search(self.root, key)

    def _search(self, node, key):
        if node is None or node.val == key:
            return node
        if key < node.val:
            return self._search(node.left, key)
        return self._search(node.right, key)

    def inorder_traversal(self, node):
        if node:
            self.inorder_traversal(node.left)
            print(node.val, end=' ')
            self.inorder_traversal(node.right)
```

Пример использования:

```
bst = BinarySearchTree()
bst.insert(50)
bst.insert(30)
bst.insert(20)
bst.insert(40)
bst.insert(70)
bst.insert(60)
bst.insert(80)

bst.inorder_traversal(bst.root)  # Вывод: 20 30 40 50 60 70 80
print()

# Поиск узла
print(bst.search(60))  # Вывод: <__main__.TreeNode object at 0x...> (если узел найден)
print(bst.search(100))  # Вывод: None (если узел не найден)
```

Обход дерева (Traversal)

Обход дерева — это процесс посещения всех узлов дерева в определенном порядке. Существуют три основных типа обхода бинарного дерева:

- **Прямой обход (Preorder Traversal)**: Посещение корня, затем левого поддерева, затем правого поддерева.
- **Симметричный обход (Inorder Traversal)**: Посещение левого поддерева, затем корня, затем правого поддерева.
- **Обратный обход (Postorder Traversal)**: Посещение левого поддерева, затем правого поддерева, затем корня.

Реализация обходов:

```
    def preorder_traversal(self, node):
        if node:
            print(node.val, end=' ')
            self.preorder_traversal(node.left)
            self.preorder_traversal(node.right)

    def postorder_traversal(self, node):
        if node:
            self.postorder_traversal(node.left)
            self.postorder_traversal(node.right)
            print(node.val, end=' ')
```

```
print("Inorder Traversal: ", end='')
bst.inorder_traversal(bst.root)  # Вывод: 20 30 40 50 60 70 80
print()

print("Preorder Traversal: ", end='')
bst.preorder_traversal(bst.root)  # Вывод: 50 30 20 40 70 60 80
print()

print("Postorder Traversal: ", end='')
bst.postorder_traversal(bst.root)  # Вывод: 20 40 30 60 80 70 50
print()
```

Удаление узла из дерева

Удаление узла из дерева — это более сложная операция, которая зависит от числа дочерних узлов у удаляемого узла:

- **Узел без дочерних узлов**: Просто удаляем узел.
- **Узел с одним дочерним узлом**: Удаляем узел и заменяем его его дочерним узлом.
- **Узел с двумя дочерними узлами**: Находим наименьший узел в правом поддереве (или наибольший в левом поддереве), заменяем значение удаляемого узла этим значением и удаляем этот наименьший узел в правом поддереве.

```
    def _delete(self, node, key):
        if node is None:
            return node

        if key < node.val:
            node.left = self._delete(node.left, key)
        elif key > node.val:
            node.right = self._delete(node.right, key)
        else:
            # Узел с одним дочерним узлом или без дочерних узлов
            if node.left is None:
                return node.right
            elif node.right is None:
                return node.left

            # Узел с двумя дочерними узлами:
            # Получить inorder преемника (наименьший в правом поддереве)
            node.val = self._min_value_node(node.right).val

            # Удалить inorder преемника
            node.right = self._delete(node.right, node.val)

        return node

    def _min_value_node(self, node):
        current = node
        while current.left is not None:
            current = current.left
        return current
```

```
# Удаление узлов
bst.delete(20)
print("Inorder Traversal after deleting 20: ", end='')
bst.inorder_traversal(bst.root)  # Вывод: 30 40 50 60 70 80
print()

bst.delete(30)
print("Inorder Traversal after deleting 30: ", end='')
bst.inorder_traversal(bst.root)  # Вывод: 40 50 60 70 80
print()

bst.delete(50)
print("Inorder Traversal after deleting 50: ", end='')
bst.inorder_traversal(bst.root)  # Вывод: 40 60 70 80
print()
```

Задания

1. **Реализовать метод для нахождения максимального значения в дереве**

Реализуйте метод для поиска максимального значения в бинарном дереве поиска. Этот метод должен быть частью класса BinarySearchTree.

2. **Подсчет числа листьев в дереве**

Реализуйте метод для подсчета числа листьев (узлов без дочерних узлов) в бинарном дереве поиска. Этот метод должен быть частью класса BinarySearchTree.

3. **Проверка сбалансированности дерева**

Реализуйте метод для проверки, является ли бинарное дерево сбалансированным. Дерево считается сбалансированным, если высота левого и правого поддеревьев любого узла отличается не более чем на единицу. Этот метод должен быть частью класса BinarySearchTree.