Linked List

In [ ]:
```
class Node:
    def __init__(self, data):
        self.data = data
        self.next = None

class LinkedList:
    def __init__(self):
        self.head = None
        self.size = 0

    def __len__(self):
        return self.size

    def is_empty(self):
        return self.size == 0

    def append(self, data):
        new_node = Node(data)
        if self.head == None:
            self.head = new_node
        else:
            current = self.head
            while current.next is not None:
                current = current.next
            current.next = new_node
        self.size += 1

    def prepend(self, data):
        new_node = Node(data)
        new_node.next = self.head
        self.head = new_node
        self.size += 1

    def __str__(self):
        elements = []
        current = self.head
        while current is not None:
            elements.append(str(current.data))
            current = current.next
        return " -> ".join(elements) + " -> None"
```

In [ ]:
```
ll = LinkedList()

ll.append(10)
ll.append(20)
ll.prepend(5)


print("Список:", ll)
print("Длина списка:", len(ll))
```

```
Список: 5 -> 10 -> 20 -> None
Длина списка: 3
```

DoubleLinked List

In [1]:
```
class Node:
    def __init__(self, data):
        self.data = data  # Данные узла
        self.next = None  # Ссылка на следующий узел
        self.prev = None  # Ссылка на предыдущий узел


class DoublyLinkedList:
    def __init__(self):
        self.head = None  # Головной узел
        self.tail = None  # Хвостовой узел

    def append(self, data):
        new_node = Node(data)
        if self.head is None:  # Если список пуст
            self.head = new_node
            self.tail = new_node
        else:
            new_node.prev = self.tail
            self.tail.next = new_node
            self.tail = new_node
```

```
    def prepend(self, data):
        new_node = Node(data)
        if self.head is None:  # Если список пуст
            self.head = new_node
            self.tail = new_node
        else:
            new_node.next = self.head
            self.head.prev = new_node
            self.head = new_node

    def insert_after(self, prev_node, data):
        if prev_node is None:
            print("Указанный узел не существует")
            return

        new_node = Node(data)
        new_node.next = prev_node.next
        prev_node.next = new_node
        new_node.prev = prev_node

        if new_node.next is not None:
            new_node.next.prev = new_node
        else:
            self.tail = new_node

    def delete(self, node):
        if self.head is None or node is None:
            return

        # Если удаляемый узел - головной
        if self.head == node:
            self.head = node.next

        # Если удаляемый узел - хвостовой
        if self.tail == node:
            self.tail = node.prev

        # Изменяем ссылки соседних узлов
        if node.next is not None:
            node.next.prev = node.prev
        if node.prev is not None:
            node.prev.next = node.next

    def display(self):
        current = self.head
        while current:
            print(current.data, end=" <-> ")
            current = current.next
        print("None")
```

In [2]:
```
dll = DoublyLinkedList()

dll.append(1)
dll.append(2)
dll.append(3)
dll.prepend(0)


dll.display()
```

```
0 <-> 1 <-> 2 <-> 3 <-> None
```

Cycled DoubleLinked List

In [ ]:
```
class Node:
    def __init__(self, data):
        self.data = data  # Данные узла
        self.next = None  # Ссылка на следующий узел
        self.prev = None  # Ссылка на предыдущий узел


class CircularDoublyLinkedList:
    def __init__(self):
        self.head = None  # Головной узел

    def append(self, data):
        """Добавление элемента в конец списка"""
        new_node = Node(data)

        if self.head is None:  # Если список пуст
            self.head = new_node
            new_node.next = new_node
            new_node.prev = new_node
```

```
        else:
            # Получаем последний узел (перед head)
            last_node = self.head.prev

            # Настраиваем связи нового узла
            new_node.next = self.head
            new_node.prev = last_node

            # Обновляем связи соседних узлов
            last_node.next = new_node
            self.head.prev = new_node

    def prepend(self, data):
        """Добавление элемента в начало списка"""
        self.append(data)  # Просто добавляем в конец
        self.head = self.head.prev  # И делаем новый узел головным

    def insert_after(self, prev_node, data):
        """Вставка элемента после указанного узла"""
        if prev_node is None:
            print("Указанный узел не существует")
            return

        new_node = Node(data)

        # Настраиваем связи нового узла
        new_node.next = prev_node.next
        new_node.prev = prev_node

        # Обновляем связи соседних узлов
        prev_node.next.prev = new_node
        prev_node.next = new_node

    def delete(self, node):
        """Удаление указанного узла"""
        if self.head is None or node is None:
            return

        # Если удаляется единственный узел
        if self.head.next == self.head and self.head == node:
            self.head = None
            return

        # Если удаляется головной узел
        if self.head == node:
            self.head = node.next

        # Перенаправляем ссылки соседних узлов
        node.prev.next = node.next
        node.next.prev = node.prev

    def display(self):
        """Вывод списка от начала до конца"""
        if self.head is None:
            print("Список пуст")
            return

        current = self.head
        while True:
            print(current.data, end=" <-> ")
            current = current.next
            if current == self.head:
                break
        print("(head)")
```

In [ ]:
```
cdll = CircularDoublyLinkedList()

cdll.append(1)
cdll.append(2)
cdll.append(3)
cdll.prepend(0)

cdll.display()
```

```
0 <-> 1 <-> 2 <-> 3 <-> (head)
```

Trees

Binary Tree

In [ ]:
```
class TreeNode:
    """Узел бинарного дерева"""
```

```
    def __init__(self, value):
        self.value = value  # Значение узла
        self.left = None    # Левый потомок
        self.right = None   # Правый потомок


class BinaryTree:
    """Бинарное дерево"""
    def __init__(self):
        self.root = None  # Корень дерева

    def insert(self, value):
        """Вставка значения в дерево"""
        if self.root is None:
            self.root = TreeNode(value)
        else:
            self._insert_recursive(self.root, value)

    def _insert_recursive(self, node, value):
        """Рекурсивная вспомогательная функция для вставки"""
        if value < node.value:
            if node.left is None:
                node.left = TreeNode(value)
            else:
                self._insert_recursive(node.left, value)
        else:
            if node.right is None:
                node.right = TreeNode(value)
            else:
                self._insert_recursive(node.right, value)

    def search(self, value):
        """Поиск значения в дереве"""
        return self._search_recursive(self.root, value)

    def _search_recursive(self, node, value):
        """Рекурсивная вспомогательная функция для поиска"""
        if node is None:
            return False
        if node.value == value:
            return True
        elif value < node.value:
            return self._search_recursive(node.left, value)
        else:
            return self._search_recursive(node.right, value)

    def inorder_traversal(self):
        """Обход дерева в порядке in-order (левый, корень, правый)"""
        result = []
        self._inorder_recursive(self.root, result)
        return result

    def _inorder_recursive(self, node, result):
        """Рекурсивный вспомогательный метод для in-order обхода"""
        if node:
            self._inorder_recursive(node.left, result)
            result.append(node.value)
            self._inorder_recursive(node.right, result)

    def preorder_traversal(self):
        """Обход дерева в порядке pre-order (корень, левый, правый)"""
        result = []
        self._preorder_recursive(self.root, result)
        return result

    def _preorder_recursive(self, node, result):
        """Рекурсивный вспомогательный метод для pre-order обхода"""
        if node:
            result.append(node.value)
            self._preorder_recursive(node.left, result)
            self._preorder_recursive(node.right, result)

    def postorder_traversal(self):
        """Обход дерева в порядке post-order (левый, правый, корень)"""
        result = []
        self._postorder_recursive(self.root, result)
        return result

    def _postorder_recursive(self, node, result):
        """Рекурсивный вспомогательный метод для post-order обхода"""
        if node:
            self._postorder_recursive(node.left, result)
            self._postorder_recursive(node.right, result)
```

```
            result.append(node.value)

    def height(self):
        """Вычисление высоты дерева"""
        return self._height_recursive(self.root)

    def _height_recursive(self, node):
        """Рекурсивный вспомогательный метод для вычисления высоты"""
        if node is None:
            return -1
        left_height = self._height_recursive(node.left)
        right_height = self._height_recursive(node.right)
        return max(left_height, right_height) + 1
```

In [ ]:
```
tree = BinaryTree()

# Вставляем значения в дерево
values = [50, 30, 70, 20, 40, 60, 80]
for value in values:
    tree.insert(value)

print("In-order обход:", tree.inorder_traversal())
print("Pre-order обход:", tree.preorder_traversal())
print("Post-order обход:", tree.postorder_traversal())

print("\nПоиск значений:")
print("40 в дереве?", tree.search(40))
print("100 в дереве?", tree.search(100))

print("\nВысота дерева:", tree.height())
```

```
In-order обход: [20, 30, 40, 50, 60, 70, 80]
Pre-order обход: [50, 30, 20, 40, 70, 60, 80]
Post-order обход: [20, 40, 30, 60, 80, 70, 50]

Поиск значений:
40 в дереве? True
100 в дереве? False

Высота дерева: 2
```

Binary Tree Search

In [ ]:
```
class TreeNode:
    """Узел бинарного дерева поиска"""
    def __init__(self, key):
        self.key = key        # Ключ узла
        self.left = None      # Левый потомок
        self.right = None     # Правый потомок
        self.parent = None    # Родительский узел (опционально)


class BinarySearchTree:
    """Бинарное дерево поиска (BST)"""
    def __init__(self):
        self.root = None

    def insert(self, key):
        """Вставка ключа в дерево"""
        if self.root is None:
            self.root = TreeNode(key)
        else:
            self._insert_recursive(self.root, key)

    def _insert_recursive(self, node, key):
        """Рекурсивная вставка ключа"""
        if key < node.key:
            if node.left is None:
                node.left = TreeNode(key)
                node.left.parent = node
            else:
                self._insert_recursive(node.left, key)
        else:
            if node.right is None:
                node.right = TreeNode(key)
                node.right.parent = node
            else:
                self._insert_recursive(node.right, key)

    def search(self, key):
        """Поиск ключа в дереве"""
        return self._search_recursive(self.root, key)
```

```
    def _search_recursive(self, node, key):
        """Рекурсивный поиск ключа"""
        if node is None:
            return None
        if node.key == key:
            return node
        elif key < node.key:
            return self._search_recursive(node.left, key)
        else:
            return self._search_recursive(node.right, key)

    def delete(self, key):
        """Удаление ключа из дерева"""
        node = self.search(key)
        if node is None:
            return False

        # Случай 1: У узла нет потомков
        if node.left is None and node.right is None:
            self._replace_node(node, None)

        # Случай 2: У узла один потомок
        elif node.left is None:
            self._replace_node(node, node.right)
        elif node.right is None:
            self._replace_node(node, node.left)

        # Случай 3: У узла два потомка
        else:
            successor = self._find_min(node.right)
            node.key = successor.key
            self._replace_node(successor, successor.right)

        return True

    def _replace_node(self, node, new_node):
        """Замена узла в дереве"""
        if node.parent is None:
            self.root = new_node
        elif node == node.parent.left:
            node.parent.left = new_node
        else:
            node.parent.right = new_node

        if new_node is not None:
            new_node.parent = node.parent

    def _find_min(self, node):
        """Поиск узла с минимальным ключом в поддереве"""
        while node.left is not None:
            node = node.left
        return node

    def inorder_traversal(self):
        """Центрированный обход (возвращает отсортированный список)"""
        result = []
        self._inorder_recursive(self.root, result)
        return result

    def _inorder_recursive(self, node, result):
        """Рекурсивный центрированный обход"""
        if node:
            self._inorder_recursive(node.left, result)
            result.append(node.key)
            self._inorder_recursive(node.right, result)

    def print_tree(self):
        """Визуализация дерева"""
        self._print_recursive(self.root, 0)

    def _print_recursive(self, node, level):
        """Рекурсивная визуализация"""
        if node is not None:
            self._print_recursive(node.right, level + 1)
            print('    ' * 4 * level + '->', node.key)
            self._print_recursive(node.left, level + 1)
```

In [ ]:

```
bst = BinarySearchTree()

# Вставка элементов
keys = [50, 30, 70, 20, 40, 60, 80]
for key in keys:
    bst.insert(key)
```

```
print("In-order обход (отсортированный список):", bst.inorder_traversal())

print("\nПоиск элементов:")
print("40 в дереве:", "Да" if bst.search(40) else "Нет")
print("100 в дереве:", "Да" if bst.search(100) else "Нет")

print("\nДерево до удаления:")
bst.print_tree()

# Удаление элемента
bst.delete(30)
print("\nДерево после удаления 30:")
bst.print_tree()

print("\nIn-order обход после удаления:", bst.inorder_traversal())
```

```
In-order обход (отсортированный список): [20, 30, 40, 50, 60, 70, 80]

Поиск элементов:
40 в дереве: Да
100 в дереве: Нет

Дерево до удаления:
        -> 80
    -> 70
        -> 60
-> 50
        -> 40
    -> 30
        -> 20

Дерево после удаления 30:
        -> 80
    -> 70
        -> 60
-> 50
    -> 40
        -> 20

In-order обход после удаления: [20, 40, 50, 60, 70, 80]
```

Двоичная куча

In [ ]:

```
class BinaryHeap:
    """Двоичная куча (минимальная по умолчанию)"""
    def __init__(self, max_heap=False):
        self.heap = []
        self.max_heap = max_heap  # Если True, то это максимальная куча

    def parent(self, i):
        """Индекс родителя для узла i"""
        return (i - 1) // 2

    def left_child(self, i):
        """Индекс левого потомка для узла i"""
        return 2 * i + 1

    def right_child(self, i):
        """Индекс правого потомка для узла i"""
        return 2 * i + 2

    def compare(self, a, b):
        """Сравнение элементов в зависимости от типа кучи"""
        if self.max_heap:
            return a > b
        else:
            return a < b

    def insert(self, key):
        """Вставка элемента в кучу"""
        self.heap.append(key)
        self.sift_up(len(self.heap) - 1)

    def sift_up(self, i):
        """Всплытие элемента"""
        while i > 0 and self.compare(self.heap[i], self.heap[self.parent(i)]):
            self.heap[i], self.heap[self.parent(i)] = self.heap[self.parent(i)], self.heap[i]
            i = self.parent(i)

    def extract(self):
        """Извлечение корневого элемента (min или max)"""
        if not self.heap:
```

```
            return None

        root = self.heap[0]
        last = self.heap.pop()

        if self.heap:
            self.heap[0] = last
            self.sift_down(0)

        return root

    def sift_down(self, i):
        """Погружение элемента"""
        min_max_index = i
        left = self.left_child(i)
        right = self.right_child(i)

        if left < len(self.heap) and self.compare(self.heap[left], self.heap[min_max_index]):
            min_max_index = left

        if right < len(self.heap) and self.compare(self.heap[right], self.heap[min_max_index]):
            min_max_index = right

        if i != min_max_index:
            self.heap[i], self.heap[min_max_index] = self.heap[min_max_index], self.heap[i]
            self.sift_down(min_max_index)

    def peek(self):
        """Получение корневого элемента без извлечения"""
        return self.heap[0] if self.heap else None

    def size(self):
        """Размер кучи"""
        return len(self.heap)

    def is_empty(self):
        """Проверка на пустоту"""
        return len(self.heap) == 0

    def build_heap(self, array):
        """Построение кучи из массива"""
        self.heap = array.copy()
        for i in range(len(self.heap) // 2, -1, -1):
            self.sift_down(i)

    def __str__(self):
        """Визуализация кучи"""
        levels = []
        level = 0
        i = 0
        while i < len(self.heap):
            level_nodes = []
            level_size = 2 ** level
            for j in range(level_size):
                if i + j < len(self.heap):
                    level_nodes.append(str(self.heap[i + j]))
            levels.append(' '.join(level_nodes))
            i += level_size
            level += 1
        return '\n'.join(levels)
```

In [ ]:

```
print("Минимальная куча:")
min_heap = BinaryHeap()
for num in [4, 2, 8, 1, 5, 7]:
    min_heap.insert(num)

print(min_heap)
print("Минимальный элемент:", min_heap.peek())
print("Извлеченный элемент:", min_heap.extract())
print("Куча после извлечения:")
print(min_heap)

print("\nМаксимальная куча:")
max_heap = BinaryHeap(max_heap=True)
for num in [4, 2, 8, 1, 5, 7]:
    max_heap.insert(num)

print(max_heap)
print("Максимальный элемент:", max_heap.peek())
print("Извлеченный элемент:", max_heap.extract())
print("Куча после извлечения:")
print(max_heap)
```

```
print("\nПостроение кучи из массива:")
arr = [9, 3, 6, 2, 7, 1, 5]
heap = BinaryHeap()
heap.build_heap(arr)
print(heap)
```

```
Минимальная куча:
1
2 7
4 5 8
Минимальный элемент: 1
Извлеченный элемент: 1
Куча после извлечения:
2
4 7
8 5

Максимальная куча:
8
5 7
1 2 4
Максимальный элемент: 8
Извлеченный элемент: 8
Куча после извлечения:
7
5 4
1 2

Построение кучи из массива:
1
2 5
3 7 6 9
```

Очередь с приоритетом

In [ ]:

```
class PriorityQueue:
    """Очередь с приоритетом (на основе двоичной кучи)"""
    def __init__(self, max_priority=False):
        """
        Инициализация очереди с приоритетом
        :param max_priority: Если True, то первыми извлекаются элементы с наибольшим приоритетом
        """
        self.heap = []
        self.max_priority = max_priority

    def _compare(self, a, b):
        """Сравнение приоритетов"""
        if self.max_priority:
            return a[0] > b[0]  # Для максимального приоритета
        else:
            return a[0] < b[0]  # Для минимального приоритета

    def push(self, priority, item):
        """Добавление элемента в очередь с приоритетом"""
        entry = (priority, item)
        self.heap.append(entry)
        self._sift_up(len(self.heap) - 1)

    def pop(self):
        """Извлечение элемента с наивысшим приоритетом"""
        if not self.heap:
            raise IndexError("Очередь пуста")

        # Сохраняем элемент с наивысшим приоритетом
        root = self.heap[0]

        # Перемещаем последний элемент в корень и выполняем sift-down
        last = self.heap.pop()
        if self.heap:
            self.heap[0] = last
            self._sift_down(0)

        return root[1]  # Возвращаем только элемент, без приоритета

    def peek(self):
        """Просмотр элемента с наивысшим приоритетом без извлечения"""
        if not self.heap:
            raise IndexError("Очередь пуста")
        return self.heap[0][1]

    def _sift_up(self, index):
        """Всплытие элемента"""
        parent = (index - 1) // 2
```

```
        while index > 0 and self._compare(self.heap[index], self.heap[parent]):
            self.heap[index], self.heap[parent] = self.heap[parent], self.heap[index]
            index = parent
            parent = (index - 1) // 2

    def _sift_down(self, index):
        """Погружение элемента"""
        size = len(self.heap)
        while True:
            left = 2 * index + 1
            right = 2 * index + 2
            smallest_or_largest = index

            if left < size and self._compare(self.heap[left], self.heap[smallest_or_largest]):
                smallest_or_largest = left

            if right < size and self._compare(self.heap[right], self.heap[smallest_or_largest]):
                smallest_or_largest = right

            if smallest_or_largest == index:
                break

            self.heap[index], self.heap[smallest_or_largest] = self.heap[smallest_or_largest], self.heap[index]
            index = smallest_or_largest

    def __len__(self):
        """Количество элементов в очереди"""
        return len(self.heap)

    def is_empty(self):
        """Проверка на пустоту"""
        return len(self.heap) == 0

    def __str__(self):
        """Строковое представление очереди"""
        return str([(priority, item) for priority, item in self.heap])
```

In [ ]:

```
print("Очередь с приоритетом (min):")
pq_min = PriorityQueue()

# Добавляем элементы с приоритетами
pq_min.push(3, "Задача 1")
pq_min.push(1, "Задача 2")
pq_min.push(2, "Задача 3")
pq_min.push(4, "Задача 4")

print("Состояние очереди:", pq_min)
print("Размер очереди:", len(pq_min))

print("\nИзвлечение элементов:")
while not pq_min.is_empty():
    print("Извлечено:", pq_min.pop())

print("\nОчередь с приоритетом (max):")
pq_max = PriorityQueue(max_priority=True)

# Добавляем элементы с приоритетами
pq_max.push(3, "Задача A")
pq_max.push(1, "Задача B")
pq_max.push(2, "Задача C")
pq_max.push(4, "Задача D")

print("Состояние очереди:", pq_max)

print("\nИзвлечение элементов:")
while not pq_max.is_empty():
    print("Извлечено:", pq_max.pop())
```

```
Очередь с приоритетом (min):
Состояние очереди: [(1, 'Задача 2'), (3, 'Задача 1'), (2, 'Задача 3'), (4, 'Задача 4')]
Размер очереди: 4

Извлечение элементов:
Извлечено: Задача 2
Извлечено: Задача 3
Извлечено: Задача 1
Извлечено: Задача 4

Очередь с приоритетом (max):
Состояние очереди: [(4, 'Задача D'), (3, 'Задача A'), (2, 'Задача C'), (1, 'Задача B')]

Извлечение элементов:
Извлечено: Задача D
Извлечено: Задача A
Извлечено: Задача C
Извлечено: Задача B
```

Очередь с приоритетом на основе кучи

In [ ]:

```python
class PriorityQueue:
    """Очередь с приоритетом, реализованная на основе двоичной кучи"""

    def __init__(self, max_heap=False):
        """
        Инициализация очереди с приоритетом
        :param max_heap: если True, то максимальный элемент имеет высший приоритет
        """
        self.heap = []
        self.max_heap = max_heap

    def _parent(self, i):
        """Возвращает индекс родителя для узла i"""
        return (i - 1) // 2

    def _left_child(self, i):
        """Возвращает индекс левого потомка для узла i"""
        return 2 * i + 1

    def _right_child(self, i):
        """Возвращает индекс правого потомка для узла i"""
        return 2 * i + 2

    def _should_swap(self, a, b):
        """Определяет, нужно ли менять местами элементы a и b"""
        if self.max_heap:
            return a > b  # Для максимальной кучи
        return a < b  # Для минимальной кучи

    def push(self, priority, value):
        """Добавляет элемент в очередь с приоритетом"""
        entry = (priority, value)
        self.heap.append(entry)
        self._sift_up(len(self.heap) - 1)

    def pop(self):
        """Извлекает и возвращает элемент с наивысшим приоритетом"""
        if not self.heap:
            raise IndexError("Queue is empty")

        # Сохраняем корневой элемент
        root = self.heap[0]

        # Перемещаем последний элемент в корень
        last = self.heap.pop()
        if self.heap:
            self.heap[0] = last
            self._sift_down(0)

        return root[1]  # Возвращаем только значение

    def peek(self):
        """Возвращает элемент с наивысшим приоритетом без извлечения"""
        if not self.heap:
            raise IndexError("Queue is empty")
        return self.heap[0][1]

    def _sift_up(self, i):
        """Поднимает элемент на правильную позицию в куче"""
        while i > 0 and self._should_swap(
            self.heap[i][0],
            self.heap[self._parent(i)][0]
```

```
        ):
            parent_idx = self._parent(i)
            self.heap[i], self.heap[parent_idx] = self.heap[parent_idx], self.heap[i]
            i = parent_idx

    def _sift_down(self, i):
        """Опускает элемент на правильную позицию в куче"""
        size = len(self.heap)
        while True:
            left = self._left_child(i)
            right = self._right_child(i)
            candidate = i

            if left < size and self._should_swap(
                self.heap[left][0],
                self.heap[candidate][0]
            ):
                candidate = left

            if right < size and self._should_swap(
                self.heap[right][0],
                self.heap[candidate][0]
            ):
                candidate = right

            if candidate == i:
                break

            self.heap[i], self.heap[candidate] = self.heap[candidate], self.heap[i]
            i = candidate

    def __len__(self):
        """Возвращает количество элементов в очереди"""
        return len(self.heap)

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

    def __str__(self):
        """Строковое представление очереди"""
        return str([f"{priority}:{value}" for priority, value in self.heap])
```

In [ ]:

```
print("Очередь с приоритетом (min-heap):")
pq = PriorityQueue()

# Добавляем элементы
pq.push(3, "Task 1")
pq.push(1, "Task 2")
pq.push(4, "Task 3")
pq.push(2, "Task 4")
pq.push(5, "Task 5")

print(f"Состояние очереди: {pq}")
print(f"Размер очереди: {len(pq)}")
print(f"Следующий элемент: {pq.peek()}")

print("\nИзвлечение элементов:")
while not pq.is_empty():
    print(f"Извлечено: {pq.pop()}")

print("\nОчередь с приоритетом (max-heap):")
pq_max = PriorityQueue(max_heap=True)

# Добавляем элементы
pq_max.push(3, "Task A")
pq_max.push(1, "Task B")
pq_max.push(4, "Task C")
pq_max.push(2, "Task D")

print(f"Состояние очереди: {pq_max}")

print("\nИзвлечение элементов:")
while not pq_max.is_empty():
    print(f"Извлечено: {pq_max.pop()}")
```

```
Очередь с приоритетом (min-heap):
Состояние очереди: ['1:Task 2', '2:Task 4', '4:Task 3', '3:Task 1', '5:Task 5']
Размер очереди: 5
Следующий элемент: Task 2

Извлечение элементов:
Извлечено: Task 2
Извлечено: Task 4
Извлечено: Task 1
Извлечено: Task 3
Извлечено: Task 5

Очередь с приоритетом (max-heap):
Состояние очереди: ['4:Task C', '2:Task D', '3:Task A', '1:Task B']

Извлечение элементов:
Извлечено: Task C
Извлечено: Task A
Извлечено: Task D
Извлечено: Task B
```

Хеш-Таблицы

In [ ]:
```python
class HashTable:
    def __init__(self, size=11):
        self.size = size
        self.table = [[] for _ in range(size)]

    def _hash(self, key):
        return hash(key) % self.size

    def insert(self, key, value): # Метод цепочек
        index = self._hash(key)
        for i, (k, _) in enumerate(self.table[index]):
            if k == key:
                self.table[index][i] = (key, value)
                return
        self.table[index].append((key, value))

    def delete(self, key):
        index = self._hash(key)
        for i, (k, _) in enumerate(self.table[index]):
            if k == key:
                del self.table[index][i]
                return
        raise KeyError(f'Ключ {key} не найден')

    def get(self, key):
        index = self._hash(key)
        for i, (k, v) in enumerate(self.table[index]):
            if k == key:
                return v
        raise KeyError(f'Ключ {key} не найден')

    def __str__(self):
        lines = []
        for i, chain in enumerate(self.table):
            if chain:
                line = ", ".join(f"{k}: {v}" for k, v in chain)
                lines.append(f"{i}: {line}")
            else:
                lines.append(f"{i}: empty")
        return "\n".join(lines)
```

In [ ]:
```python
ht = HashTable()

for key, value in zip([54, 26, 93, 17, 77, 31, 44, 20, 55], 'abcdefghi'):
    ht.insert(key, value)

print(ht)
```

```
0: 77: e, 44: g, 55: i
1: empty
2: empty
3: empty
4: 26: b
5: 93: c
6: 17: d
7: empty
8: empty
9: 31: f, 20: h
10: 54: a
```

In [ ]:
```python
class OpenAddressingHashTable:
    def __init__(self, size=11):
        self.size = size
        self.table = [None] * size
        self.DELETED = object()

    def _hash(self, key, i):
        return (key + i) % self.size

    def insert(self, key, value):
        for i in range(self.size):
            j = self._hash(key, i)
            if self.table[j] is None or self.table[j] is self.DELETED:
                self.table[j] = (key, value)
                return
            if self.table[j][0] == key:
                self.table[j] = (key, value)
                return
        raise Exception('Хеш-таблица переполнена')

    def get(self, key):
        for i in range(self.size):
            j = self._hash(key, i)
            if self.table[j] is None:
                break
            if self.table[j] is not self.DELETED and self.table[j][0] == key:
                return self.table[j][1]
        raise KeyError(f'Ключ {key} не найден')

    def delete(self, key):
        for i in range(self.size):
            j = self._hash(key, i)
            if self.table[j] is None:
                break
            if self.table[j] is not self.DELETED and self.table[j][0] == key:
                self.table[j] = self.DELETED
                return
        raise KeyError(f'Ключ {key} не найден')

    def display(self):
        for i, slot in enumerate(self.table):
            if slot is None:
                print(f'{i}: ∅')
            elif slot is self.DELETED:
                print(f'{i}: DEL')
            else:
                print(f'{i}: {slot[0]}:{slot[1]}')
```

Bubble sort

In [ ]:
```python
def bubble_sort(arr, ascending=True):
    n = len(arr)
    for i in range(n-1):
        for j in range(n - 1 - i):
            if (arr[j] > arr[j+1]) == ascending:
                arr[j], arr[j+1] = arr[j+1], arr[j]
    return arr
```

Binary search

In [3]:
```python
def binary_search(arr, x):
    low = 0
    high = len(arr) - 1
    while low <= high:
        mid = (low + high) // 2
        if arr[mid] == x:
            return mid
        elif arr[mid] < x:
            low = mid + 1
        else:
            high = mid - 1
    return -1
```

Selection sort

In [4]:
```python
def selection_sort(arr, reverse=False):
    n = len(arr)
    for i in range(n):
        min_idx = i
```

```
        for j in range(i + 1, n):
            if reverse:
                if arr[j] > arr[min_idx]:
                    min_idx = j
            else:
                if arr[j] < arr[min_idx]:
                    min_idx = j
        arr[i], arr[min_idx] = arr[min_idx], arr[i]
    return arr
```

Insertion Sort

In [5]:
```
def insertion_sort(arr, reverse=False):
    for i in range(1, len(arr)):
        key = arr[i]
        j = i - 1
        while j >= 0 and ((not reverse and arr[j] > key) or (reverse and arr[j] < key)):
            arr[j + 1] = arr[j]
            j -= 1
        arr[j + 1] = key
    return arr
```

Quick Sort

In [7]:
```
def _quick_sort(data, ascending=True):
    if len(data) <= 1:
        return data
    pivot = data[0]
    less = [x for x in data[1:] if (x < pivot) == ascending]
    greater = [x for x in data[1:] if (x >= pivot) == ascending]
    return _quick_sort(less, ascending) + [pivot] + _quick_sort(greater, ascending)


def quick_sort(data, ascending=True):
    return _quick_sort(data, ascending)
```

Shell Sort

In [8]:
```
def shell_sort(arr, reverse=False):
    gap = len(arr) // 2
    while gap > 0:
        for i in range(gap, len(arr)):
            temp = arr[i]
            j = i
            while j >= gap and ((not reverse and arr[j - gap] > temp)
                                or (reverse and arr[j - gap] < temp)):
                arr[j] = arr[j - gap]
                j -= gap
            arr[j] = temp
        gap //= 2
    return arr
```

Merge Sort

In [9]:
```
def merge_sort(arr, reverse=False):
    if len(arr) <= 1:
        return arr

    mid = len(arr) // 2
    left_half = arr[:mid]
    right_half = arr[mid:]

    left_half = merge_sort(left_half, reverse=reverse)
    right_half = merge_sort(right_half, reverse=reverse)

    return merge(left_half, right_half, reverse=reverse)

def merge(left_half, right_half, reverse=False):
    result = []
    i = 0
    j = 0

    while i < len(left_half) and j < len(right_half):
        if (not reverse and left_half[i] <= right_half[j]) or (reverse and left_half[i] >= right_half[j]):
            result.append(left_half[i])
            i += 1
        else:
            result.append(right_half[j])
```

```
            j += 1

    result += left_half[i:]
    result += right_half[j:]

    return result
```

In [ ]: