Task 33 - ✓

Задачу решить двумя способами:

1. Реализовать класс стек через использование списка:
   - Создать класс `Stack` с методами:
     - `push(item)` : добавляет элемент `item` в верхушку стека;
     - `pop()` : удаляет и возвращает элемент из верхушки стека;
     - `peek()` : возвращает элемент из верхушки стека без его удаления;
     - `is_empty()` : проверяет, пуст ли стек;
     - `size()` : возвращает текущий размер стека.
2. Реализовать класс стек через связный список:
   - Создать класс `Node` для представления узла связного списка с двумя атрибутами: `data` (значение узла) и `next` (ссылка на следующий узел).
   - Создать класс `Stack` с методами:
     - `push(item)` : добавляет элемент `item` в верхушку стека;
     - `pop()` : удаляет и возвращает элемент из верхушки стека;
     - `peek()` : возвращает элемент из верхушки стека без его удаления;
     - `is_empty()` : проверяет, пуст ли стек;
     - `size()` : возвращает текущий размер стека/

Оба способа реализации должны обеспечивать функциональность стека: добавление элементов в верхушку, удаление элементов из верхушки, получение элемента из верхушки без его удаления, проверку на пустоту и получение текущего размера стека.

---

10. Дан стек. Необходимо проверить, является ли его содержимое последовательностью арифметической прогрессии.

In [53]:
```python
def is_arithmetic_progression(stack):
    if len(stack) < 2:
        return True

    temp = []
    values = []

    while len(stack) > 0:
        values.append(stack.items.pop())

    # Копируем в обратном порядке, чтобы восстановить стек и сохранить порядок
    for item in reversed(values):
        stack.push(item)


    diff = values[1] - values[0]
    for i in range(1, len(values)):
        if values[i] - values[i - 1] != diff:
            return False
    return True
```

In [55]:
```python
# 1 case Last in -> first out

class Stack:
    def __init__(self, items=None):
        if items == None:
            self.items = []
        else:
            self.items = list(items)

    def push(self, item):
        self.items.append(item)

    def pop(self, item):
        if not self.is_empty():
            return self.items.pop()
        else:
            raise IndexError("trying to pop from empty stack")

    def peek(self):
        if not self.is_empty():
            return self.items[-1]
        else:
```

```
            raise IndexError("trying to peek from empty stack")


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

    def size(self):
        return len(self.items)

    def __str__(self):
        return ", ".join(map(str, reversed(self.items)))

    def __len__(self):
        return len(self.items)
```

In [57]:
```
s = Stack([1, 3, 5, 7, 9])  # 9 — на вершине
print(is_arithmetic_progression(s))  # True

s = Stack([2, 4, 7])
print(is_arithmetic_progression(s))  # False
```

```
True
False
```

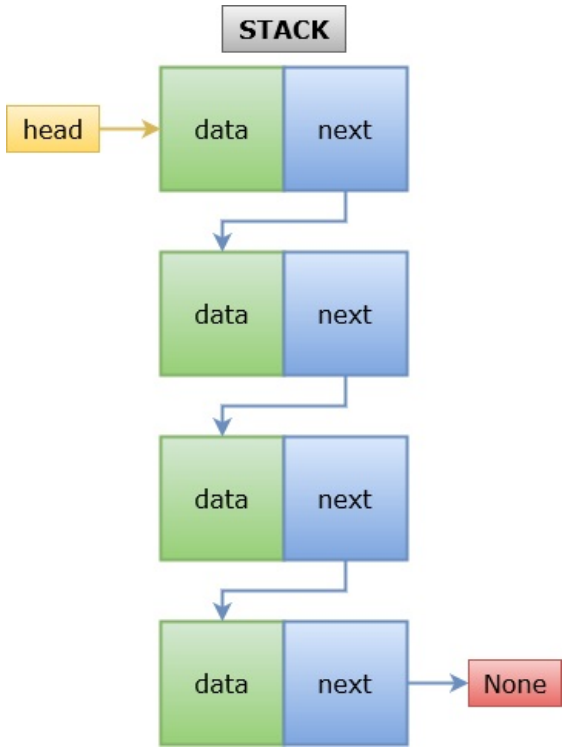

Рисунок 27 — Схема реализации стека через связный список

In [64]:
```
def is_arithmetic_progression(stack):
    if stack.is_empty() or stack.head.next is None:
        return True  # 0 или 1 элемент — прогрессия

    values = []

    # Сохраняем элементы без изменения стека
    current = stack.head
    while current:
        values.append(current.data)
        current = current.next

    # Проверяем на арифметическую прогрессию
    diff = values[1] - values[0]
    for i in range(1, len(values)):
        if values[i] - values[i - 1] != diff:
            return False
    return True
```

In [66]:
```
# 2 case
class Node:
    def __init__(self, data):
        self.data = data
        self.next = None

class Stack:
    def __init__(self):
```

```
        self.head = None

    def is_empty(self):
        return self.head is None

    def push(self, item):
        new_node = Node(item)
        new_node.next = self.head
        self.head = new_node

    def pop(self):
        if self.is_empty():
            return None
        popped_item = self.head.data
        self.head = self.head.next
        return popped_item

    def peek(self):
        if self.is_empty():
            return None
        return self.head.data

    def __str__(self):
        current = self.head
        stack_str = ""
        while current:
            stack_str += str(current.data) + " → "
            current = current.next
        return stack_str.rstrip(" → ")
```

In [70]:
```
s = Stack()
for item in [9, 7, 5, 3, 1]:
    s.push(item)

print(s)
print(is_arithmetic_progression(s))

s2 = Stack()
for item in [10, 7, 4, 0]:
    s2.push(item)

print(s2)
print(is_arithmetic_progression(s2))

s3 = Stack()
for item in [5, 3, 0, -2]:
    s3.push(item)

print(s3)
print(is_arithmetic_progression(s3))
```

```
1 → 3 → 5 → 7 → 9
True
0 → 4 → 7 → 10
False
-2 → 0 → 3 → 5
False
```

Task 34 - ✓

Задачу решить двумя способами:

1. Реализовать класс очередь через использование списка:
   - Создать класс `Queue` с методами:
     - `enqueue(item)` : добавляет элемент `item` в конец очереди;
     - `dequeue()` : удаляет и возвращает элемент из начала очереди;
     - `peek()` : возвращает элемент из начала очереди без его удаления;
     - `is_empty()` : проверяет, пуста ли очередь;
     - `size()` : возвращает текущий размер очереди.
2. Реализовать класс очередь через связный список:
   - Создать класс `Node` для представления узла связного списка с двумя атрибутами: `data` (значение узла) и `next` (ссылка на следующий узел).
   - Создать класс `Queue` с методами:
     - `enqueue(item)` : добавляет элемент `item` в конец очереди;
     - `dequeue()` : удаляет и возвращает элемент из начала очереди;

- `peek()` : возвращает элемент из начала очереди без его удаления;
- `is_empty()` : проверяет, пуста ли очередь;
- `size()` : возвращает текущий размер очереди.

Оба способа реализации должны обеспечивать функциональность очереди: добавление элементов в конец, удаление элементов из начала, получение элемента из начала без его удаления, проверку на пустоту и получение текущего размера очереди.

---

10. Создать класс очереди, который будет хранить только элементы определенного типа данных. Тип элементов задается при инициализации объекта класса очереди. При добавлении элемента, если его тип не соответствует заданному, то он не должен добавляться.

Класс очередь

In [73]:
```python
class Queue:
    def __init__(self, items=None):
        if items == None:
            self.items = []
        else:
            self.items = list(items)

    def enqueue(self, item):
        self.items.append(item)

    def dequeue(self):
        if not self.is_empty():
            return self.items.pop(0)
        else:
            raise IndexError("trying to pop from empty Queue")

    def peek(self):
        if not self.is_empty():
            return self.items[0]
        else:
            raise IndexError("trying to peek from empty Queue")

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

    def size(self):
        return len(self.items)

    def __str__(self):
        return ", ".join(map(str, self.items))
```

Класс очередь через связный список

In [39]:
```python
class Node:
    def __init__(self, data):
        self.data = data
        self.next = None



class LinkedQueue:
    def __init__(self):
        self.head = None
        self.tail = None
        self.count = 0

    def is_empty(self):
        return self.head is None

    def enqueue(self, item):
        new_node = Node(item)
        if self.is_empty():
            self.head = new_node
            self.tail = new_node
        else:
            self.tail.next = new_node
            self.tail = new_node
        self.count += 1

    def dequeue(self):
        if self.is_empty():
            raise IndexError("dequeue from empty queue")
        removed_data = self.head.data
        self.head = self.head.next
        self.count -= 1
```

```
        if self.head is None:
            self.tail = None
        return removed_data

    def peek(self):
        if self.is_empty():
            raise IndexError("peek from empty queue")
        return self.head.data

    def size(self):
        return self.count

    def __str__(self):
        elements = []
        current = self.head
        while current:
            elements.append(str(current.data))
            current = current.next
        return "Head -> " + " -> ".join(elements) + " -> None"
```

In [41]:
```
q = LinkedQueue()
q.enqueue(5)
q.enqueue(10)
q.enqueue(15)
print(q)
q.dequeue()
print(q)
```

```
Head -> 5 -> 10 -> 15 -> None
Head -> 10 -> 15 -> None
```

10. **Создать класс очереди, который будет хранить только элементы определенного типа данных. Тип элементов задается при инициализации объекта класса очереди. При добавлении элемента, если его тип не соответствует заданному, то он не должен добавляться.**

In [53]:
```
class Node:
    def __init__(self, data):
        self.data = data
        self.next = None


class TypedQueue:
    def __init__(self, data_type):
        self.head = None
        self.tail = None
        self.count = 0
        self.data_type = data_type

    def is_empty(self):
        return self.head is None

    def enqueue(self, item):
        if not isinstance(item, self.data_type):
            print(f"Ошибка: элемент {item} не добавлен (ожидался тип {self.data_type.__name__})")
            return
        new_node = Node(item)
        if self.is_empty():
            self.head = self.tail = new_node
        else:
            self.tail.next = new_node
            self.tail = new_node
        self.count += 1

    def dequeue(self):
        if self.is_empty():
            raise IndexError("dequeue from empty queue")
        data = self.head.data
        self.head = self.head.next
        self.count -= 1
        if self.head is None:
            self.tail = None
        return data

    def peek(self):
        if self.is_empty():
            raise IndexError("peek from empty queue")
        return self.head.data

    def size(self):
        return self.count

    def __str__(self):
```

```
        elements = []
        current = self.head
        while current:
            elements.append(str(current.data))
            current = current.next
        return f"TypedQueue<{self.data_type.__name__}>: head -> " + " -> ".join(elements) + " -> None"
```

In [57]:
```
q = TypedQueue(int)

q.enqueue(10)
q.enqueue("hello")
q.enqueue(25)

print(q)
print(q.dequeue())
print(q.peek())
print(q)
```

```
Ошибка: элемент hello не добавлен (ожидался тип int)
TypedQueue<int>: head -> 10 -> 25 -> None
10
25
TypedQueue<int>: head -> 25 -> None
```

Task 35 - ✓

10. Создайте двусвязный список для хранения информации о заказах в интернетмагазине. Каждый элемент списка должен содержать номер заказа, дату создания, список товаров, их количество и стоимость, а также информацию о доставке и оплате.

In [1]:
```
class OrderNode:
    def __init__(self, order_id, date, items, delivery, payment):
        self.order_id = order_id
        self.date = date
        self.items = items
        self.delivery = delivery
        self.payment = payment

        self.prev = None
        self.next = None

    def __str__(self):
        item_list = "\n    ".join(
            [f"{item['name']} x{item['quantity']} - {item['price']}₽" for item in self.items]
        )
        return (
            f"Заказ {self.order_id} от {self.date}\n"
            f"Товары:\n    {item_list}\n"
            f"Доставка: {self.delivery}\n"
            f"Оплата: {self.payment}"
        )


class OrderList:
    def __init__(self):
        self.head = None
        self.tail = None

    def add_order(self, order_id, date, items, delivery, payment):
        new_order = OrderNode(order_id, date, items, delivery, payment)
        if self.tail:
            self.tail.next = new_order
            new_order.prev = self.tail
            self.tail = new_order
        else:
            self.head = self.tail = new_order

    def show_orders(self):
        current = self.head
        while current:
            print(current)
            print("-" * 40)
            current = current.next

    def show_latest_order(self):
        if self.tail:
            print("Последний заказ:\n", self.tail)
        else:
            print("Список заказов пуст.")
```

In [5]:
```
orders = OrderList()

orders.add_order(
    "ORD-001",
    "2025-05-04",
    [
        {"name": "Macbook", "quantity": 1, "price": 95000},
        {"name": "Mouse", "quantity": 2, "price": 1500},
    ],
    delivery="Курьер, Москва",
    payment="Картой онлайн"
)

orders.add_order(
    "ORD-002",
    "2025-05-05",
    [
        {"name": "Смартфон", "quantity": 1, "price": 58000},
    ],
    delivery="Самовывоз, Санкт-Петербург",
    payment="Оплата при получении"
)

orders.show_orders()
```

```
Заказ ORD-001 от 2025-05-04
Товары:
    Macbook x1 - 95000₽
    Mouse x2 - 1500₽
Доставка: Курьер, Москва
Оплата: Картой онлайн
----------------------------------------
Заказ ORD-002 от 2025-05-05
Товары:
    Смартфон x1 - 58000₽
Доставка: Самовывоз, Санкт-Петербург
Оплата: Оплата при получении
----------------------------------------
```

Task 36 - ✓

10. Реализовать функцию, которая разделяет двусвязный список на два списка, один из которых содержит все элементы, меньшие заданного значения, а другой — все элементы, большие или равные заданному значению.

In [9]:
```
class Node:
    def __init__(self, value):
        self.value = value
        self.prev = None
        self.next = None

class DoublyLinkedList:
    def __init__(self):
        self.head = None
        self.tail = None

    def append(self, value):
        new_node = Node(value)
        if self.tail:  # не пустой список
            self.tail.next = new_node
            new_node.prev = self.tail
            self.tail = new_node
        else:  # пустой список
            self.head = self.tail = new_node

    def __iter__(self):
        current = self.head
        while current:
            yield current.value
            current = current.next

    def print_list(self):
        print(" <-> ".join(str(val) for val in self))

def split_list_by_value(original_list, threshold):
    less_list = DoublyLinkedList()
    greater_equal_list = DoublyLinkedList()

    current = original_list.head
    while current:
        if current.value < threshold:
            less_list.append(current.value)
```

```
        else:
            greater_equal_list.append(current.value)
        current = current.next

    return less_list, greater_equal_list
```

In [11]:
```
dll = DoublyLinkedList()
for val in [10, 5, 8, 3, 12, 7, 15]:
    dll.append(val)

print("Исходный список:")
dll.print_list()

less, greater_equal = split_list_by_value(dll, 8)

print("\nЭлементы < 8:")
less.print_list()

print("\nЭлементы ≥ 8:")
greater_equal.print_list()
```

```
Исходный список:
10 <-> 5 <-> 8 <-> 3 <-> 12 <-> 7 <-> 15

Элементы < 8:
5 <-> 3 <-> 7

Элементы ≥ 8:
10 <-> 8 <-> 12 <-> 15
```

Task 37 - ✓

10. Реализовать функцию, которая проверяет, содержится ли заданный элемент в циклическом двусвязном списке.

In [44]:
```
class Node:
    def __init__(self, data):
        self.data = data
        self.next = None
        self.prev = None

class CircledLinked:
    def __init__(self):
        self.head = None

    def append(self, item):
        new_node = Node(item)
        if self.head == None:
            self.head = new_node
            new_node.next = new_node
            new_node.prev = new_node
        else:
            # Найдём "хвост" — последний элемент перед головой
            tail = self.head.prev

            # Устанавливаем связи между текущим хвостом и новым узлом
            tail.next = new_node     # хвост указывает на новый узел
            new_node.prev = tail     # новый узел указывает назад на хвост

            # Новый узел указывает на голову
            new_node.next = self.head

            # Голова указывает назад на новый узел (он стал новым хвостом)
            self.head.prev = new_node
```

In [46]:
```
def contains(cdll, target):
    if not cdll.head:
        return False

    current = cdll.head
    while True:
        if current.data == target:
            return True
        current = current.next
        if current == cdll.head:  # вернулись в начало
            break
    return False
```

In [48]:
```
cdll = CircledLinked()
for val in [10, 20, 30, 40]:
    cdll.append(val)
```

```
print(contains(cdll, 30))
print(contains(cdll, 50))
```

```
True
False
```

Task 38 - ✓

10. Необходимо отсортировать массив дат и вывести результат на экран. В зависимости от переданного параметра отсортировать массив дат по возрастанию или по убыванию даты, используя алгоритмы сортировки: сортировку выбором, сортировку пузырьком и быструю сортировку. Сравнить время выполнения алгоритмов сортировки с помощью декоратора. Даты хранятся в файле.

Декоратор

In [66]:
```
import time

def timer(func):
    def wrapper(data, ascending=True):
        start = time.time()
        result = func(data.copy(), ascending)
        end = time.time()
        print(f"{func.__name__}: {end - start:.6f} сек")
        return result
    return wrapper
```

Сортировка выбором

In [113…
```
@timer
def selection_sort(arr, ascending=True):
    n = len(arr)
    for i in range(n):
        min_idx = i
        for j in range(i + 1, n):
            if (arr[j] < arr[min_idx]) == ascending:
                min_idx = j
        arr[i], arr[min_idx] = arr[min_idx], arr[i]
    return arr
```

Сортирвока пузырьком

In [116…
```
@timer
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
```

Быстрая сортировка

In [119…
```
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)

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

Открытие файла

In [208…
```
from datetime import datetime

with open("dates.txt", 'r') as f:
    dates = [datetime.strptime(line.strip(), '%Y-%m-%d') for line in f if line.strip()]
dates
```

Out[208…
```
[datetime.datetime(2023, 4, 15, 0, 0),
 datetime.datetime(2021, 12, 1, 0, 0),
 datetime.datetime(2021, 6, 6, 0, 0),
 datetime.datetime(2022, 6, 30, 0, 0),
 datetime.datetime(2020, 11, 25, 0, 0),
 datetime.datetime(2023, 1, 1, 0, 0),
 datetime.datetime(2023, 8, 23, 0, 0),
 datetime.datetime(2025, 8, 23, 0, 0),
 datetime.datetime(2024, 12, 23, 0, 0),
 datetime.datetime(2021, 11, 12, 0, 0),
 datetime.datetime(2022, 5, 10, 0, 0),
 datetime.datetime(2024, 8, 9, 0, 0),
 datetime.datetime(2006, 8, 27, 0, 0),
 datetime.datetime(2007, 12, 23, 0, 0),
 datetime.datetime(2008, 11, 12, 0, 0),
 datetime.datetime(2009, 5, 10, 0, 0),
 datetime.datetime(2001, 8, 9, 0, 0),
 datetime.datetime(2003, 8, 27, 0, 0),
 datetime.datetime(2007, 1, 1, 0, 0),
 datetime.datetime(2008, 2, 20, 0, 0),
 datetime.datetime(2009, 12, 10, 0, 0),
 datetime.datetime(2001, 8, 14, 0, 0),
 datetime.datetime(2023, 12, 21, 0, 0)]
```

In [243…
```
sorted_selection = selection_sort(dates, ascending=True)
sorted_bubble = bubble_sort(dates, ascending=True)
sorted_quick = quick_sort(dates, ascending=True)
```

```
selection_sort: 0.000064 сек
bubble_sort: 0.000096 сек
quick_sort: 0.000083 сек
```

Task 39

10. Реализовать класс бинарного дерева. Написать функцию для нахождения всех узлов, которые имеют двух потомков в бинарном дереве.

In [90]:
```
class TreeNode:
    def __init__(self, value):
        self.value = value
        self.left = None
        self.right = None

class BinaryTree:
    def __init__(self, root_value):
        self.root = TreeNode(root_value)

    def insert(self, value):
        self._insert_recursive(self.root, value)

    def _insert_recursive(self, current_node, value):
        if value < current_node.value:
            if current_node.left is None:
                current_node.left = TreeNode(value)
            else:
                self._insert_recursive(current_node.left, value)
        else:
            if current_node.right is None:
                current_node.right = TreeNode(value)
            else:
                self._insert_recursive(current_node.right, value)
```

In [92]:
```
def find_nodes_with_two_children(node, result=None):
    if result is None:
        result = []

    if node is None:
        return result

    # Если у узла есть оба потомка, добавляем его в результат
    if node.left is not None and node.right is not None:
        result.append(node.value)

    # Рекурсивно проверяем левое и правое поддеревья
    find_nodes_with_two_children(node.left, result)
    find_nodes_with_two_children(node.right, result)

    return result
```

In [118…
```
tree = BinaryTree(10)
tree.insert(5)
```

```
tree.insert(15)
tree.insert(3)
tree.insert(7)
tree.insert(12)
tree.insert(20)


nodes_with_two_children = find_nodes_with_two_children(tree.root)
print("Узлы с двумя потомками:", nodes_with_two_children)
```

```
Узлы с двумя потомками: [10, 5, 15]
```

Task 40

10. Имеется система управления задачами. Каждая задача имеет приоритет и дедлайн. Реализовать структуру данных на основе двоичной кучи, которая будет поддерживать операции добавления задачи и извлечения задачи с наивысшим приоритетом и дедлайном до определенного времени.

In [120…

```
from datetime import datetime

class Task:
    def __init__(self, description, priority, deadline):
        self.description = description
        self.priority = priority
        self.deadline = deadline

    def __repr__(self):
        return f"Task('{self.description}', priority={self.priority}, deadline={self.deadline})"

class TaskPriorityQueue:
    def __init__(self):
        self.tasks = []

    def add_task(self, task):
        """Добавляет задачу в кучу"""
        self.tasks.append(task)
        self._sift_up(len(self.tasks) - 1)

    def get_highest_priority_task(self, max_deadline=None):
        """
        Извлекает задачу с наивысшим приоритетом
        Если указан `max_deadline`, возвращает только задачи с дедлайном <= max_deadline
        """
        if not self.tasks:
            return None

        if max_deadline is None:
            return self._extract_max()
        else:
            # Сначала ищем подходящие задачи
            valid_tasks = [task for task in self.tasks if task.deadline <= max_deadline]
            if not valid_tasks:
                return None

            # Временное удаление и восстановление кучи
            max_task = max(valid_tasks, key=lambda x: x.priority)
            self.tasks.remove(max_task)
            self._heapify()
            return max_task

    def _sift_up(self, index):
        """Поднимает задачу вверх по куче, если её приоритет выше родителя"""
        parent = (index - 1) // 2
        if index > 0 and self.tasks[index].priority > self.tasks[parent].priority:
            self.tasks[index], self.tasks[parent] = self.tasks[parent], self.tasks[index]
            self._sift_up(parent)

    def _sift_down(self, index):
        """Опускает задачу вниз, если её приоритет меньше потомков"""
        left = 2 * index + 1
        right = 2 * index + 2
        largest = index

        if left < len(self.tasks) and self.tasks[left].priority > self.tasks[largest].priority:
            largest = left
        if right < len(self.tasks) and self.tasks[right].priority > self.tasks[largest].priority:
            largest = right

        if largest != index:
            self.tasks[index], self.tasks[largest] = self.tasks[largest], self.tasks[index]
            self._sift_down(largest)
```

```
    def _extract_max(self):
        """Извлекает задачу с максимальным приоритетом."""
        if not self.tasks:
            return None
        max_task = self.tasks[0]
        self.tasks[0] = self.tasks[-1]
        self.tasks.pop()
        self._sift_down(0)
        return max_task

    def _heapify(self):
        """Восстанавливает кучу после удаления."""
        for i in range(len(self.tasks) // 2, -1, -1):
            self._sift_down(i)
```

In [126…
```
task_queue = TaskPriorityQueue()


task_queue.add_task(Task("Написать код", 3, datetime(2023, 12, 31)))
task_queue.add_task(Task("Проверить баги", 5, datetime(2023, 11, 15)))
task_queue.add_task(Task("Обновить документацию", 2, datetime(2023, 10, 20)))


print(task_queue.get_highest_priority_task())

# Извлекаем задачу с дедлайном до 2023-12-01
print(task_queue.get_highest_priority_task(max_deadline=datetime(2023, 12, 1)))
```

```
Task('Проверить баги', priority=5, deadline=2023-11-15 00:00:00)
Task('Обновить документацию', priority=2, deadline=2023-10-20 00:00:00)
```

Task 41

10.

а) Создать класс «Фильм» с полями «Название», «Режиссер», «Год выпуска» и «Жанр». Создать хеш-таблицу для хранения объектов класса «Фильм» по ключу — названию фильма.
б) Написать функцию для нахождения элемента в хеш-таблице, который наиболее близок по значению к заданному числу.
в) Реализуйте хеш-таблицу для хранения информации о пациентах в больнице. Ключом является номер медицинской карты, значение — объект, содержащий информацию о пациенте (ФИО, диагноз, лечение и т.д.). Используйте метод разрешения коллизий методом открытой адресации с квадратичным пробированием.

Пункт а

Метод цепочек - вместо хранения одного элемента в ячейке хеш-таблицы, в каждой ячейке хранится список (или связанный список) всех элементов, чей хеш совпал

In [128…
```
class Film:
    def __init__(self, title, director, year, genre):
        self.title = title
        self.director = director
        self.year = year
        self.genre = genre

    def __str__(self):
        return f"{self.title} ({self.year}), реж. {self.director}, жанр: {self.genre}"

class FilmHashTable:
    def __init__(self, size=10):
        self.size = size
        self.table = [[] for _ in range(size)]  # Используем метод цепочек

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

    def add_film(self, film):
        key = film.title
        hash_value = self._hash(key)
        bucket = self.table[hash_value]

        # Проверяем, нет ли уже фильма с таким названием
        for i, (k, v) in enumerate(bucket):
            if k == key:
                bucket[i] = (key, film)
                return
        bucket.append((key, film))

    def get_film(self, title):
        hash_value = self._hash(title)
```

```
        bucket = self.table[hash_value]

        for k, v in bucket:
            if k == title:
                return v
        return None

    def __str__(self):
        result = []
        for bucket in self.table:
            for key, film in bucket:
                result.append(f"{key}: {film}")
        return "\n".join(result)
```

In [130…
```
film_table = FilmHashTable()
film_table.add_film(Film("Крестный отец", "Фрэнсис Форд Коппола", 1972, "криминальная драма"))
film_table.add_film(Film("Побег из Шоушенка", "Фрэнк Дарабонт", 1994, "драма"))

print(film_table.get_film("Крестный отец"))
```

```
Крестный отец (1972), реж. Фрэнсис Форд Коппола, жанр: криминальная драма
```

Пункт б

In [62]:
```
class NumericHashTable:
    def __init__(self, size=10):
        self.size = size
        self.table = [None] * size

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

    def insert(self, key, value):
        hash_value = self._hash(key)
        if self.table[hash_value] is None:
            self.table[hash_value] = (key, value)
        else:
            # открытая адресация с линейным пробированием для разрешения коллизий
            next_slot = (hash_value + 1) % self.size
            while self.table[next_slot] is not None and next_slot != hash_value:
                next_slot = (next_slot + 1) % self.size
            if self.table[next_slot] is None:
                self.table[next_slot] = (key, value)
            else:
                raise Exception("Хеш-таблица заполнена")

    def find_closest(self, target):
        if not any(self.table):
            return None

        closest = None
        min_diff = float('inf')

        for item in self.table:
            if item is not None:
                key, value = item
                current_diff = abs(key - target)
                if current_diff < min_diff:
                    min_diff = current_diff
                    closest = value

        return closest
```

In [132…
```
num_table = NumericHashTable()
num_table.insert(10, "Десять")
num_table.insert(20, "Двадцать")
num_table.insert(30, "Тридцать")

print(num_table.find_closest(18))
```

```
Двадцать
```

Пункт в

Открытая адресация (open addressing) — ищем другую свободную ячейку
Квадратичное пробирование - это разновидность открытой адресации, где при коллизии ищем следующую свободную ячейку по формуле:

$$h(k, i) = h(h(k) + i^2) mod m$$

- h(k) — хеш-функция

- i — номер попытки (0, 1, 2, ...)
- m — размер таблицы

In [72]:
```python
class Patient:
    def __init__(self, name, diagnosis, treatment):
        self.name = name
        self.diagnosis = diagnosis
        self.treatment = treatment

    def __repr__(self):
        return f"{self.name}, Диагноз: {self.diagnosis}, Лечение: {self.treatment}"

class PatientHashTable:
    def __init__(self, size=10):
        self.size = size
        self.table = [None] * size

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

    def insert(self, key, patient):
        idx = self._hash(key)        # начальный индекс
        i = 1                        # шаг
        while self.table[idx] is not None and self.table[idx][0] != key:
            idx = (idx + i**2) % self.size
            i += 1
        self.table[idx] = (key, patient)

    def get(self, key):
        idx = self._hash(key)
        i = 1
        while self.table[idx] is not None:
            if self.table[idx][0] == key:
                return self.table[idx][1]
            idx = (idx + i**2) % self.size
            i += 1
        return None
```

In [80]:
```python
hash_table = PatientHashTable()

hash_table.insert(123, Patient("Иванов И.И.", "Грипп", "Покой и витамины"))
hash_table.insert(133, Patient("Петров П.П.", "ОРВИ", "Тёплое питье"))
print(hash_table.get(123))
```

```
Иванов И.И., Диагноз: Грипп, Лечение: Покой и витамины
```