|  |  |  |
| --- | --- | --- |
| МИНОБРНАУКИ РОССИИ | | |
| Федеральное государственное автономное  образовательное учреждение высшего образования  «Пермский государственный национальный  исследовательский университет» | | |
|  | Институт компьютерных наук и технологий | |
| **ОТЧЁТ**  по индивидуальной работе №2  по дисциплине «Языки программирования»  Вариант 12 | | |
|  | | Работу выполнил  студент группы ИТ-5-2024 1 курса  Чирков Егор Юрьевич  «18» июня 2025 г. |
| Работу проверил(а)  \_\_\_\_\_\_\_\_\_\_\_\_\_\_ Фамилия И.О.  «\_\_\_» \_\_\_\_\_\_\_\_\_ 2025 г. |
| Пермь 2025 | | |

СОДЕРЖАНИЕ

[Постановка задачи 3](#_Toc200544779)

[Алгоритм решения 4](#_Toc200544780)

[Обоснование выбора структур и типов данных 7](#_Toc200544781)

[Тестирование 9](#_Toc200544782)

[Код программы 12](#_Toc200544783)

# Постановка задачи

Разработать программу, моделирующую упрощенную версию игры "Шарики". В игре имеется линия шариков различных цветов, представленных целыми числами от 0 до 9. Когда в линии образуется непрерывная цепочка из трех и более шариков одного цвета, эта цепочка уничтожается. После уничтожения цепочки шарики сдвигаются, и процесс уничтожения может повториться, пока в линии не останется ни одной цепочки, удовлетворяющей условию удаления.

Цель программы: По заданному начальному состоянию линии шариков определить и вывести общее количество шариков, которые будут уничтожены в процессе игры.

# Алгоритм решения

Для решения поставленной задачи был выбран следующий алгоритм:

**1. Представление данных:**

• Линия шариков представляется в виде односвязного списка (LinkedList).

• Каждый шарик (узел списка) хранит информацию о своем цвете (целое число).

**2. Структуры данных:**

• Node (узел списка):

\* color: Целое число, представляющее цвет шарика.

\* next: Указатель на следующий узел в списке (шарик в линии).

• LinkedList (односвязный список):

\* head: Указатель на первый узел списка (начало линии шариков).

\* tail: Указатель на последний узел списка (конец линии шариков).

\* size: Количество узлов (шариков) в списке.

3. Основные методы класса LinkedList:

• append(color): Добавляет новый шарик заданного цвета в конец линии.

• destroy\_chain(start, end): Удаляет непрерывную цепочку шариков одного цвета, начинающуюся с узла start и заканчивающуюся узлом end.

• check\_and\_destroy(): Основной метод, реализующий логику игры. Он ищет в списке непрерывные цепочки из трех и более шариков одного цвета и, если находит, удаляет их. Метод повторяет этот процесс до тех пор, пока в списке не останется ни одной цепочки, подлежащей удалению.

4. Алгоритм check\_and\_destroy():

• Инициализируется переменная total\_destroyed для хранения общего количества уничтоженных шариков.

• Запускается цикл while destroyed, который выполняется до тех пор, пока на очередной итерации не будет уничтожено ни одной цепочки.

• Внутри цикла:

\* Устанавливаются указатели current и start на начало списка (первый шарик).

\* Инициализируется переменная chain\_length для отслеживания длины текущей цепочки.

\* Происходит итерация по списку. Если текущий шарик и следующий имеют одинаковый цвет, увеличивается chain\_length.

\* Если цвет следующего шарика отличается, проверяется длина текущей цепочки (chain\_length).

\* Если chain\_length >= 3, вызывается метод destroy\_chain(start, current) для удаления цепочки. Обновляется total\_destroyed. Флаг destroyed устанавливается в True, и цикл прерывается, чтобы начать поиск новых цепочек с начала списка (важно, поскольку после удаления цепочки могут образоваться новые).

\* Если цепочка заканчивается в конце списка, выполняется аналогичная проверка и, при необходимости, удаление.

5. Выбор структур данных:

**•** Односвязный список был выбран, поскольку в данной задаче требуется эффективное удаление элементов из середины списка и перестройка связей между оставшимися элементами. Операции вставки и удаления в односвязном списке (при наличии указателей на нужные узлы) выполняются относительно быстро (O(1)).

• Использование массива потребовало бы сдвига элементов после каждого удаления цепочки, что при большом количестве шариков могло бы существенно замедлить работу программы.

• Двусвязный список мог бы упростить операцию destroy\_chain, сделав ее более эффективной, но сложность алгоритма поиска цепочек и общая логика программы остались бы прежними.

# Обоснование выбора структур и типов данных

**1. Целые числа для цветов шариков:** По условию задачи, цвета шариков представлены целыми числами от 0 до 9. Это позволяет использовать эффективные операции сравнения и хранения данных.

**2. Односвязный список (LinkedList) для представления линии шариков:**

• Обоснование: Односвязный список был выбран в качестве основной структуры данных для представления линии шариков из-за следующих факторов:

• Динамическое изменение размера: Количество шариков в линии может изменяться в процессе игры (при удалении цепочек). Односвязный список позволяет легко добавлять и удалять элементы, не требуя предварительного определения максимального размера линии.

• Эффективное удаление элементов: Основная операция в игре - удаление цепочек шариков. В односвязном списке удаление элемента (при наличии указателя на предшествующий элемент) выполняется за константное время O(1), так как требуется только изменить указатель next у предыдущего элемента.

• Не требуется произвольный доступ: В данной игре не требуется часто обращаться к шарикам по их индексу (как это было бы необходимо, например, при реализации сортировки). Доступ к элементам осуществляется последовательно, что хорошо соответствует структуре односвязного списка.

• Классы Node и LinkedList: Реализация через классы Node и LinkedList соответствует принципам объектно-ориентированного программирования и позволяет инкапсулировать данные (цвет шарика) и операции (добавление, удаление, поиск цепочек) в отдельные сущности.

• Альтернативы:

• Массив (список Python list): Использование массива потребовало бы сдвига элементов после каждого удаления цепочки, что при большом количестве шариков (до 100000) привело бы к значительному снижению производительности (сложность O(n) для каждой операции удаления).

• Двусвязный список: Двусвязный список упростил бы удаление элементов, поскольку позволял бы быстро находить как следующий, так и предыдущий элементы. Однако, это потребовало бы больше памяти для хранения дополнительного указателя prev в каждом узле и усложнило бы код, при этом принципиально не улучшив асимптотическую сложность алгоритма. В данном случае, выигрыш от использования двусвязного списка не оправдывает усложнение кода.

3. **Атрибуты head, tail и size в классе LinkedList:**

• head: Указатель на первый элемент списка. Необходим для начала обхода списка и выполнения различных операций (добавление, удаление, поиск).

• tail: Указатель на последний элемент списка. Позволяет добавлять новые элементы в конец списка за константное время O(1). Без tail добавление в конец списка требовало бы обхода всего списка, что заняло бы O(n) времени.

• size: Хранит текущее количество элементов в списке. Позволяет быстро определять размер списка (за O(1) время) без необходимости обхода всех элементов. Это полезно для различных проверок и операций.

4. **Тип данных bool для флага destroyed:**

• Обоснование: Флаг destroyed используется для отслеживания, были ли удалены какие-либо цепочки на текущей итерации основного цикла игры. Использование типа bool (истина или ложь) является наиболее подходящим для этой цели, поскольку:

• Явное представление состояния: bool явно указывает, произошло ли событие (удаление цепочки) или нет.

• Читаемость: Код, использующий bool, становится более читаемым и понятным, чем код, использующий, например, целые числа для представления логических значений.

• Эффективность: bool занимает минимальный объем памяти.

# Тестирование

**Тест 1: Базовый случай с одной цепочкой в начале линии**

=====Добро пожаловать в игру 'Шарики'!=====

Ваша цель: уничтожить цепочки из трех и более шариков одного цвета.

Введите количество шариков в линии (максимум 100.000), или 'exit' для выхода: 5

Введите цвет шарика 1 (от 0 до 9): 1

Введите цвет шарика 2 (от 0 до 9): 1

Введите цвет шарика 3 (от 0 до 9): 1

Введите цвет шарика 4 (от 0 до 9): 2

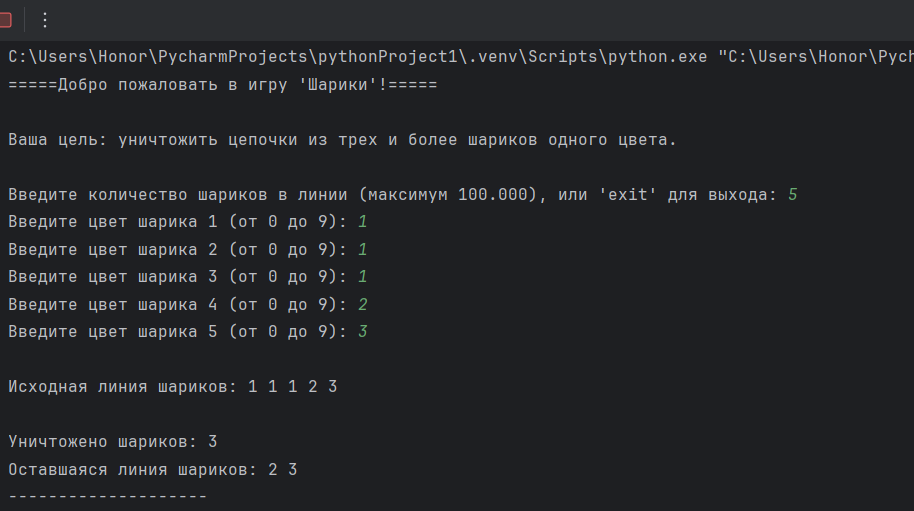
Введите цвет шарика 5 (от 0 до 9): 3

Исходная линия шариков: 1 1 1 2 3

Уничтожено шариков: 3

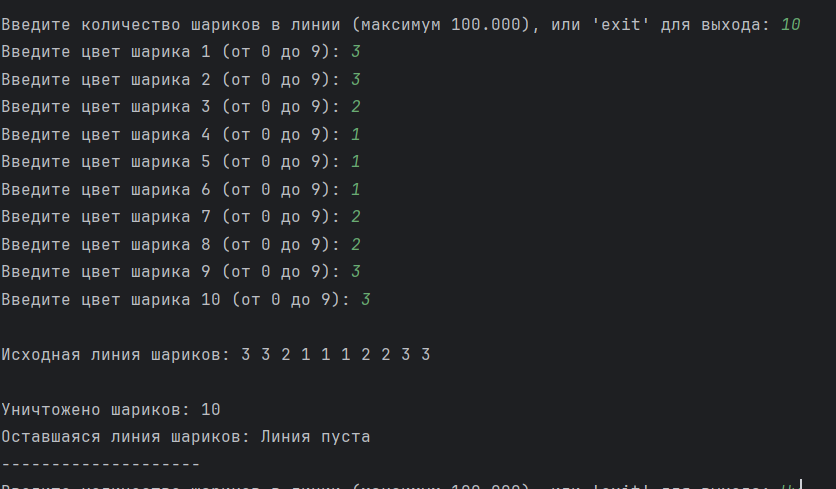
Оставшаяся линия шариков: 2 3

--------------------



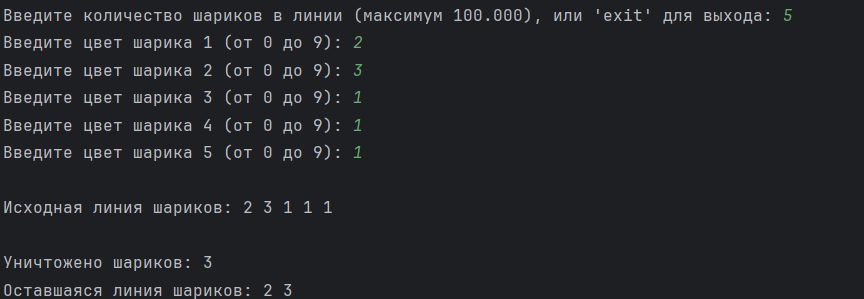
Тест проверяет, правильно ли удаляется цепочка из трех шариков одного цвета в начале линии.

**Тест 2: Несколько цепочек, удаляемых последовательно**



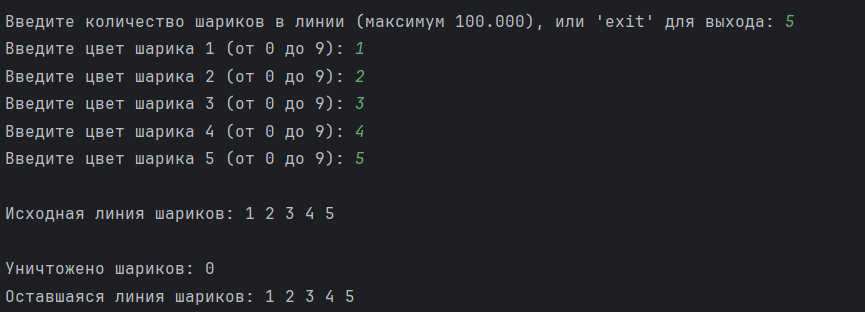
Тест проверяет, правильно ли программа обрабатывает ситуацию, когда после удаления одной цепочки образуются новые цепочки, которые тоже необходимо удалить.

**Тест 3: Цепочка в конце линии**

****

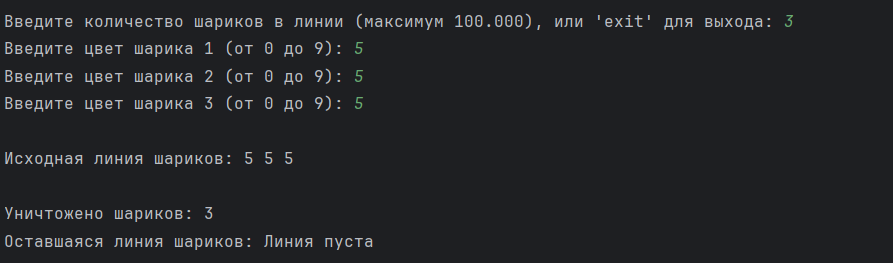
Тест проверяет, правильно ли удаляется цепочка в конце линии.

**Тест 4: Линия, не содержащая цепочек для удаления**

****

Тест проверяет, правильно ли программа обрабатывает ситуацию, когда в линии нет цепочек, удовлетворяющих условию удаления.

**Тест 5: Пограничный случай (все шарики образуют одну цепочку)**

****

Тест проверяет, правильно ли программа обрабатывает случай, когда все шарики в линии имеют один и тот же цвет и образуют цепочку, подлежащую удалению.

# Код программы

class Node:  
 *"""Узел односвязного списка для хранения цвета шарика."""* def \_\_init\_\_(self, color):  
 *"""Инициализация узла."""* self.color = color # Цвет шарика (целое число)  
 self.next = None # Указатель на следующий узел (изначально None)  
  
 def \_\_str\_\_(self):  
 *"""Возвращает строковое представление узла (цвет шарика)."""* return str(self.color)  
  
  
class LinkedList:  
 *"""Односвязный список для представления линии шариков."""* def \_\_init\_\_(self):  
 *"""Инициализация списка."""* self.head = None # Голова списка (первый узел, изначально None)  
 self.tail = None # Хвост списка (последний узел, изначально None)  
 self.size = 0 # Количество шариков в линии (изначально 0)  
  
 def append(self, color):  
 *"""Добавляет шарик заданного цвета в конец линии."""* new\_node = Node(color) # Создаем новый узел с заданным цветом  
 if self.head is None:  
 # Если список пуст:  
 self.head = new\_node # Новый узел становится головой  
 self.tail = new\_node # И хвостом  
 else:  
 # Если в списке уже есть элементы:  
 self.tail.next = new\_node # Добавляем новый узел в конец списка  
 self.tail = new\_node # Обновляем хвост списка  
 self.size += 1 # Увеличиваем размер списка  
  
 def \_\_len\_\_(self):  
 *"""Возвращает длину списка (количество шариков)."""* return self.size  
  
 def \_\_str\_\_(self):  
 *"""Возвращает строковое представление линии шариков."""* if self.head is None:  
 # Если список пуст:  
 return "Линия пуста"  
 current = self.head # Начинаем с головы списка  
 result = "" # Инициализируем пустую строку для результата  
 while current:  
 # Пока не дошли до конца списка:  
 result += str(current.color) + " " # Добавляем цвет текущего шарика в строку  
 current = current.next # Переходим к следующему шарику  
 return result.strip() # Возвращаем строку без лишних пробелов в начале и конце  
  
 def destroy\_chain(self, start, end):  
 *"""Удаляет цепочку шариков, начиная с узла start и заканчивая узлом end."""* destroyed\_count = 0  
 if start is self.head:  
 # Если удаляется цепочка в начале списка:  
 if end.next is None: # До конца списка  
 self.head = None # Список становится пустым  
 self.tail = None # Обнуляем указатель на хвост  
 destroyed\_count = self.size # Все элементы удалены  
 self.size = 0 # Размер списка равен 0  
 else:  
 # Если цепочка удаляется только в начале списка:  
 self.head = end.next # Новой головой становится элемент после цепочки  
 destroyed\_count = 0 # Обнуляем счетчик, т.к. нужно пересчитать  
 current = start  
 # Устанавливаем `current` на начало удаляемой цепочки (`start`).  
  
 while current != end.next:  
 # Перебираем узлы от начала цепочки (`start`) до узла, следующего за концом цепочки (`end.next`).  
  
 destroyed\_count += 1  
 # Увеличиваем счетчик удаленных узлов (`destroyed\_count`) на каждом шаге.  
  
 current = current.next  
 # Переходим к следующему узлу в цепочке.  
  
 self.size -= destroyed\_count  
 # Обновляем размер списка (`self.size`), вычитая количество удаленных узлов.  
  
 else:  
 # Удаление в середине списка или в конце  
 current = self.head  
 # Начинаем с головы списка.  
 while current.next != start:  
 current = current.next  
 # Ищем узел, \*предшествующий\* началу удаляемой цепочки.  
  
 current.next = end.next # Перестраиваем связь: предыдущий\_узел.next = следующий\_после\_цепочки  
  
 destroyed\_count = 0 # Обнуляем счетчик удаленных узлов.  
 current = start # Начинаем с начала удаляемой цепочки.  
 while current != end.next: # Итерируемся по узлам удаляемой цепочки.  
 destroyed\_count += 1 # Итерируемся по узлам удаляемой цепочки.  
 if current == end:  
 break  
 # Прерываем цикл, если `current` достиг конца удаляемой цепочки.  
  
 current = current.next # Переходим к следующему узлу в цепочке.  
 if end.next == None:  
 self.tail = current  
 # Если удаляем до конца списка, обновляем указатель на хвост.  
 self.size -= destroyed\_count # Обновляем размер списка.  
  
 return destroyed\_count # Возвращаем количество удаленных узлов.  
  
  
 def check\_and\_destroy(self):  
 *"""Проверяет наличие цепочек и удаляет их, повторяя до тех пор, пока есть что удалять."""* total\_destroyed = 0 # Суммарное кол-во разрушенных шаров  
 destroyed = True # Флаг, показывающий, были ли уничтожены шары на текущей итерации  
 while destroyed: # Цикл повторяется, пока есть что уничтожать  
 destroyed = False # Сбрасываем флаг перед каждой итерацией  
 destroyed\_count = 0 # Инициализируем счетчик уничтоженных шариков на текущей итерации  
  
 if self.head is None or self.head.next is None or self.head.next.next is None:  
 # Если шаров меньше трех, то цепочки не существует, выходим  
 return total\_destroyed  
  
 current = self.head # Начинаем с головы списка  
 start = self.head # Начало текущей цепочки  
 chain\_length = 1 # Длина текущей цепочки  
  
 while current.next: # Пока не дошли до предпоследнего элемента  
 if current.color == current.next.color:  
 # Если текущий и следующий шарики одного цвета, увеличиваем длину цепочки  
 chain\_length += 1  
 else:  
 # Иначе (цвет отличается):  
 if chain\_length >= 3:  
 # Если цепочка достаточно длинная (>= 3 шаров):  
 destroyed\_count += self.destroy\_chain(start, current) # Уничтожаем цепочку  
 total\_destroyed += destroyed\_count # Добавляем уничтоженные шары к общему количеству  
 destroyed = True # Устанавливаем флаг, что были разрушения  
 break # Прерываем цикл, чтобы начать проверку сначала  
 start = current.next # Начинаем новую цепочку со следующего элемента  
 chain\_length = 1 # Сбрасываем длину цепочки  
 current = current.next # Переходим к следующему шарику  
  
 if not destroyed: # Если после цикла не было разрушений  
 if chain\_length >= 3:  
 # Проверяем, не заканчивается ли цепочка в конце списка  
 destroyed\_count += self.destroy\_chain(start, self.tail) # Уничтожаем цепочку в конце  
 total\_destroyed += destroyed\_count# Добавляем уничтоженные шары к общему количеству  
 destroyed = True# Устанавливаем флаг, что были разрушения  
 break # Выходим из внешнего цикла, т.к. разрушений больше нет  
  
 return total\_destroyed # Возвращаем общее количество разрушенных шаров  
  
  
def get\_valid\_input(prompt, data\_type, validation\_func=None):  
 *"""Получает корректный ввод от пользователя с проверкой типа и дополнительной валидацией."""* while True:  
 try:  
 value = data\_type(input(prompt)) # Получаем ввод от пользователя и преобразуем к нужному типу  
 if validation\_func is None or validation\_func(value):  
 # Если нет функции валидации или ввод прошел валидацию:  
 return value # Возвращаем ввод  
 else:  
 # Иначе:  
 print("Ошибка: Некорректный ввод. Пожалуйста, проверьте условия.")  
 except ValueError:  
 # Если ввод не может быть преобразован к нужному типу:  
 print("Ошибка: Введите значение корректного типа.")  
 except Exception as e:  
 # Если произошла другая ошибка:  
 print(f"Ошибка: {e}")  
  
  
def main():  
 *"""Основная функция программы."""* print("=====Добро пожаловать в игру 'Шарики'!=====\n")  
 print("Ваша цель: уничтожить цепочки из трех и более шариков одного цвета.\n")  
  
 while True: # Добавляем цикл для повторных игр  
 num\_balls = get\_valid\_input("Введите количество шариков в линии (максимум 100.000), или 'exit' для выхода: ",  
 lambda x: int(x) if x.isdigit() else x,  
 lambda x: (isinstance(x, int) and 1 <= x <= 100000) or x == 'exit')  
  
 if num\_balls == 'exit':  
 print("Выход из игры.")  
 break # Завершаем игру  
  
 ball\_colors = []  
 # Инициализируем пустой список для хранения цветов шариков.  
  
 for i in range(num\_balls):  
 color = get\_valid\_input(f"Введите цвет шарика {i + 1} (от 0 до 9): ", int, lambda x: 0 <= x <= 9)  
 ball\_colors.append(color) # Добавляем цвет в список  
 # Запрашиваем цвет каждого шарика у пользователя и добавляем его в список `ball\_colors`.  
  
 ball\_line = LinkedList()  
 # Создаем новый экземпляр связного списка `LinkedList`.  
  
 for color in ball\_colors:  
 ball\_line.append(color)  
 # Добавляем каждый цвет шарика из списка `ball\_colors` в связный список `ball\_line`.  
  
 # Создаем связный список на основе введенных данных  
 # Комментарий, поясняющий назначение этого блока кода.  
  
  
 print("\nИсходная линия шариков:", ball\_line)  
  
 destroyed = ball\_line.check\_and\_destroy()  
 # Запускаем процесс уничтожения цепочек  
  
 print("\nУничтожено шариков:", destroyed)  
 print("Оставшаяся линия шариков:", ball\_line)  
 print("-" \* 20) # Разделитель между играми  
  
  
if \_\_name\_\_ == "\_\_main\_\_":  
 # Проверяем, что скрипт запущен напрямую (а не импортирован).  
  
 try:  
 main()  
 # Пытаемся выполнить основную функцию программы.  
  
 except Exception as e:  
 print(f"Произошла непредвиденная ошибка: {e}")  
 # Перехватываем любые исключения, возникающие в `main()`, и выводим сообщение об ошибке.