

МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ

**ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ**

**ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ**

**«ДОНСКОЙ ГОСУДАРСТВЕННЫЙ ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ»**

**(ДГТУ)**

Кафедра «Вычислительные системы и информационная безопасность»

**Методические указания к контрольной работе по дисциплине   
«**Алгоритмы и структуры данных**»**

**для студентов 2-го курса**

**направления подготовки 09.03.02 Информационные системы и технологии**

**заочной формы обучения**

Ростов-на-Дону

2021 г.

**Алгоритм выбора варианта контрольной работы**

Контрольная работа заключается в выполнении 4 заданий, приведенных в данном документе.

Номер варианта выбирается по следующему алгоритму:

Для выбора варианта необходимо взять предпоследнюю и последнюю цифры номера зачетной книжки. Номер варианта находится на пересечении соответствующей строки и столбца.

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | Последняя цифра номера зачетной книжки | | | | | | | | | | |
|  |  | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
| Предпоследняя цифра номера зачетной книжки | 0 | 1 | 2 | 5 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
| 1 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 |
| 2 | 21 | 22 | 23 | 24 | 25 | 26 | 27 | 28 | 29 | 30 |
| 3 | 31 | 32 | 33 | 34 | 35 | 36 | 37 | 38 | 39 | 40 |
| 4 | 1 | 2 | 5 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
| 5 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 |
| 6 | 21 | 22 | 23 | 24 | 25 | 26 | 27 | 28 | 29 | 30 |
| 7 | 31 | 32 | 33 | 34 | 35 | 36 | 37 | 38 | 39 | 40 |
| 8 | 1 | 2 | 5 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
| 9 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 |

Например, для зачетки с номером 123456 необходимо взять номер варианта из 5-ой строки и 6-го столбца (вариант 17).

Задание 1

**Сортировка с помощью двоичного дерева**

Краткие теоретические сведения

**Сортировка с помощью двоичного дерева** (сортировка двоичным деревом, сортировка деревом, древесная сортировка, сортировка с помощью бинарного дерева, [англ.](https://ru.wikipedia.org/wiki/%D0%90%D0%BD%D0%B3%D0%BB%D0%B8%D0%B9%D1%81%D0%BA%D0%B8%D0%B9_%D1%8F%D0%B7%D1%8B%D0%BA) *tree sort*) — универсальный алгоритм сортировки, заключающийся в построении двоичного дерева поиска по ключам массива (списка), с последующей сборкой результирующего массива путём обхода узлов построенного дерева в необходимом порядке следования ключей. Данная сортировка является оптимальной при получении данных путём непосредственного чтения из потока (например, файла, сокета или консоли).

Процедура добавления объекта в бинарное дерево имеет среднюю алгоритмическую сложность порядка O(log(n)). Соответственно, для n объектов сложность будет составлять O(n\*log(n)), что относит сортировку с помощью двоичного дерева к группе «быстрых сортировок». Однако, сложность добавления объекта в разбалансированное дерево может достигать O(n), что может привести к общей сложности порядка O(n2).

При физическом развёртывании древовидной структуры в памяти требуется не менее чем 4n ячеек дополнительной памяти (каждый узел должен содержать ссылки на элемент исходного массива, на родительский элемент, на левый и правый лист), однако, существуют способы уменьшить требуемую дополнительную память.

Суть алгоритма в том, что на основании массива строится так называемое **декартово дерево**. А из построенного декартового дерева очень легко получить все элементы в порядке возрастания или убывания.

С другой стороны, хранение данных в дереве поиска весьма удобно с организационной точки зрения, позволяет с этими данными делать очень многие операции, сортировать в том числе. Родитель в двоичном дереве поиска необязательно больше, чем потомок. Но при этом левый потомок всегда меньше родителя, правый потомок всегда больше родителя (или равен ему).

Максимальный элемент массива при построении дерева поиска попадает где-то в правое поддерево. Чтобы этот максимум извлечь, дерево поиска надо полностью обойти.

Чем более несбалансированным будет дерево поиска, тем более затратным будет его обход.

Рекурсивный обход дерева может запросто стать долгим путешествием по крайне извилистому лабиринту и временна́я сложность часто деградирует до O(**n**2).

**Задание**

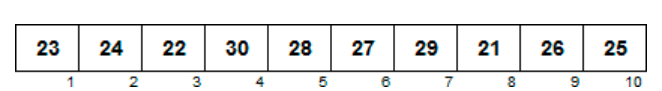
Дана исходная последовательность в соответствии с вариантом:

23 24 22 30 28 27 29 21 26 25.

1. Построить дерево сортировки на основе исходной последовательности.
2. Определить параметры полученного дерева: глубину дерева и является ли оно сбалансированным.
3. Сборка отсортированного массива путем обхода узлов в необходимой последовательности.

**Пример выполнения работы.**

Дана последовательность чисел



Построить бинарное дерево сортировки.

Этапы сортировки:

* **Этап 1. На основании массива строим бинарное дерево поиска.** Первый элемент массива кладём в корень, остальные элементы сравниваем сначала с корнем, затем, в зависимости от сравнения, движемся вниз по левым или правым веткам (и по пути сравниваем элемент массива с имеющимися узлами дерева). В конце концов очередной элемент доходит до конца какой-либо ветки и сам становится узлом. Если в дереве имеется узел равный добавляемому, то можно двигаться как влево, так и вправо.
* **Этап 2. Обходим построенное бинарное дерево поиска от минимума к максимуму.** Дело в том, что строгая организация этого дерева (левый потомок меньше, чем родитель, а правый больше или равен чем родитель) позволяет простым рекурсивным способом обойти данное дерево от меньших элементов к большим. Что в свою очередь выдаёт узлы дерева в возрастающем порядке.

Построение дерева имеет следующую алгоритмическую сложность – в среднем вставка нового узла обходится в O(log **n**), таким образом временна́я сложность первого этапа — O(**n** log **n**). Построенное дерево представлено на рис. 1.

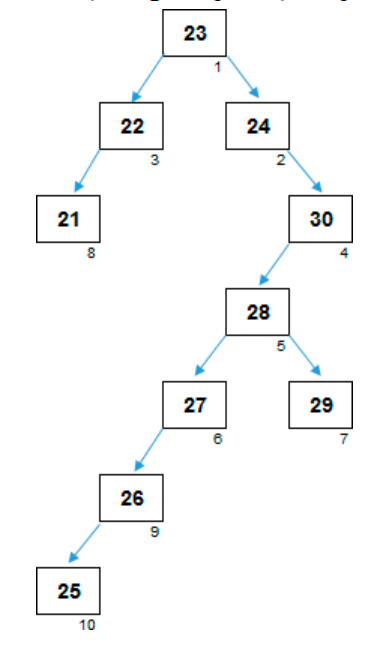


Рис. 1. Бинарное дерево сортировки.

Дерево не является сбалансированным. Глубина равна 7.

Внешний вид дерева сильно зависит от того, как изначально расположены элементы массива. На случайных данных, дерево с высокой долей вероятности получится несбалансирвоанным, с гораздо большим количеством уровней, по сравнению с оптимальным. В нашем примере образовалось 7 уровней. Если бы элементы массива располагались, например, в таком порядке:

25, 28, 29, 23, 30, 27, 24, 22, 21, 26

дерево поиска бы получилось сбалансированным, с минимально возможным количеством уровней (= 4). Сортировка с помощью сбалансированного дерева имела бы сложность по времени: O(**n** log **n**). Но в случае несбалансированного дерева может деградировать и до O(**n2**).

## Этап 2. Построение отсортированного массива (по возрастанию):

1. Делим дерево на две части: левая, корень и правая часть = (левая), 23, (правая).
2. Затем левую и правую части рассматриваем как отдельные деревья, выделяя в них левую, вершину и правую части ((левая = 21), вершина = 22, правая - нет), 23, (левая – нет, вершина = 24, (правая)). Таким образом получили часть отсортированного массива (21, 22, 23, 24, (правая)), в котором надо раскрыть правую часть от 24 вершины. Там находится вершина 30, у которой есть левая ветвь и нет правой. Получаем (21, 22, 23, 24, (левая), 30 (правая- нет)), то есть (21, 22, 23, 24, (левая), 30).
3. Раскрыв левую ветвь от вершины 30 получим: (21, 22, 23, 24, (левая),28, (правая), 30), а затем, раскрыв левую и правую ветви, получим (21, 22, 23, 24, (левая), 27, 28, 29, 30).
4. В результате получим 21, 22, 23, 24, 25, 26, 27, 28, 29, 30.

При необходимости сортировки по убыванию делим дерево на три части: правая, корень и левая часть = (правая), 23, (левая). И т.д.

Если к выше полученному дереву необходимо добавить еще узел, равный 24, то его можно считать меньше узла, равного 24, имеющегося в дереве, так и считать его больше 24. В первом случае, узел 24 будет добавлен слева от уже имеющегося в дереве узла 24, во втором случае он будет добавлен слева от узла 25. Первый случай не увеличивает глубину дерева и является более предпочтительным.

**Варианты заданий.**

Вариант 1: 53, 52, 23, 57, 30, 59, 27, 46, 19, 5, 52, 35, 51, 48, 46, 55, 3, 49, 14, 17, 4, 54, 59, 16, 32.

Вариант 2: 21, 13, 12, 8, 36, 35, 51, 6, 12, 38, 28, 17, 50, 19, 58, 24, 47, 48, 36, 22, 35, 39, 35, 4, 50.

Вариант 3: 4, 9, 19, 26, 25, 51, 4, 53, 15, 4, 47, 3, 9, 40, 16, 50, 13, 14, 30, 37, 36, 6, 56, 23, 21.

Вариант 4: 3, 53, 30, 25, 55, 15, 33, 3, 49, 28, 33, 23, 9, 44, 13, 13, 20, 26, 44, 7, 26, 49, 34, 46, 18.

Вариант 5: 5, 37, 30, 41, 21, 38, 10, 54, 22, 49, 10, 30, 2, 45, 50, 18, 25, 31, 1, 40, 58, 6, 47, 3, 57.

Вариант 6: 59, 41, 41, 11, 11, 58, 42, 53, 8, 9, 16, 2, 20, 54, 15, 42, 50, 6, 49, 48, 12, 7, 37, 5, 37.

Вариант 7: 28, 56, 12, 18, 36, 15, 19, 6, 1, 57, 44, 26, 28, 28, 51, 25, 42, 23, 23, 14, 14, 5, 47, 49, 20.

Вариант 8: 25, 59, 42, 29, 50, 21, 9, 54, 31, 47, 37, 21, 57, 22, 18, 13, 46, 53, 7, 54, 37, 53, 45, 20, 32.

Вариант 9: 49, 10, 47, 26, 33, 5, 4, 50, 59, 2, 39, 57, 8, 23, 30, 54, 10, 49, 39, 14, 3, 54, 29, 40, 4.

Вариант 10: 1, 57, 13, 53, 27, 11, 17, 45, 51, 16, 21, 52, 51, 25, 36, 12, 56, 16, 2, 20, 36, 23, 38, 44, 42.

Вариант 11: 11, 4, 15, 12, 58, 11, 39, 54, 17, 16, 46, 45, 23, 10, 34, 8, 17, 51, 40, 28, 34, 14, 1, 16, 57.

Вариант 12: 4, 51, 5, 43, 16, 54, 4, 36, 30, 52, 27, 50, 49, 5, 35, 30, 47, 26, 58, 9, 40, 41, 44, 30, 59.

Вариант 13: 51, 11, 40, 17, 52, 44, 24, 6, 37, 7, 40, 19, 6, 38, 33, 35, 50, 56, 13, 9, 43, 48, 19, 51, 46.

Вариант 14: 23, 35, 22, 31, 29, 35, 46, 20, 16, 52, 16, 10, 10, 15, 11, 31, 57, 24, 18, 40, 14, 41, 20, 3, 56.

Вариант 15: 17, 40, 59, 23, 22, 28, 43, 16, 52, 47, 28, 8, 12, 15, 59, 1, 20, 42, 17, 19, 9, 20, 29, 55, 3.

Вариант 16: 46, 5, 38, 41, 46, 24, 45, 11, 56, 20, 5, 18, 39, 10, 52, 10, 26, 55, 28, 59, 26, 46, 43, 13, 10.

Вариант 17: 40, 59, 46, 34, 23, 17, 46, 15, 42, 36, 12, 57, 49, 39, 5, 28, 6, 39, 46, 27, 1, 32, 37, 51, 19.

Вариант 18: 17, 37, 22, 33, 38, 39, 9, 41, 56, 46, 56, 9, 18, 43, 15, 57, 57, 51, 54, 12, 55, 7, 47, 44, 56.

Вариант 19: 28, 54, 11, 30, 23, 36, 3, 21, 43, 5, 15, 4, 50, 12, 46, 58, 48, 29, 41, 17, 5, 21, 15, 26, 20.

Вариант 20: 38, 26, 50, 28, 27, 37, 40, 59, 3, 33, 4, 37, 11, 30, 27, 59, 36, 45, 17, 54, 50, 20, 52, 31, 13.

Вариант 21: 53, 29, 59, 2, 58, 51, 17, 13, 27, 41, 17, 57, 11, 39, 3, 52, 4, 29, 28, 7, 6, 22, 4, 57, 59.

Вариант 22: 34, 1, 37, 45, 23, 34, 10, 40, 13, 36, 51, 33, 58, 23, 37, 9, 44, 43, 52, 49, 40, 17, 27, 35, 5.

Вариант 23: 7, 56, 15, 52, 35, 12, 52, 13, 19, 3, 27, 20, 37, 22, 56, 44, 4, 21, 23, 34, 15, 56, 25, 17, 55.

Вариант 24: 42, 56, 29, 19, 48, 4, 6, 42, 13, 22, 46, 13, 20, 17, 29, 57, 40, 50, 28, 20, 52, 36, 20, 33, 2.

Вариант 25: 52, 27, 34, 10, 12, 36, 5, 59, 56, 53, 51, 57, 23, 50, 23, 27, 50, 49, 38, 25, 9, 40, 8, 1, 19.

Вариант 26: 35, 21, 41, 33, 17, 13, 26, 26, 22, 59, 21, 22, 58, 18, 16, 14, 40, 31, 56, 27, 35, 19, 11, 6, 48.

Вариант 27: 28, 27, 21, 3, 57, 35, 17, 28, 20, 50, 42, 53, 13, 7, 28, 8, 10, 30, 50, 41, 46, 4, 14, 36, 59.

Вариант 28: 47, 11, 36, 39, 55, 31, 32, 23, 29, 49, 54, 37, 57, 9, 53, 56, 23, 58, 24, 41, 34, 23, 30, 44, 40.

Вариант 29: 31, 1, 39, 53, 20, 27, 54, 31, 28, 12, 34, 14, 31, 44, 4, 33, 27, 31, 16, 39, 20, 38, 50, 20, 16.

Вариант 30: 8, 48, 22, 35, 15, 24, 53, 20, 14, 5, 18, 56, 2, 55, 12, 14, 37, 12, 38, 30, 15, 7, 41, 9, 10.

Вариант 31: 2, 11, 14, 29, 39, 44, 28, 20, 44, 59, 28, 12, 46, 11, 10, 37, 14, 50, 29, 7, 43, 49, 21, 10, 42.

Вариант 32: 38, 53, 48, 14, 53, 50, 9, 5, 5, 15, 14, 5, 36, 30, 18, 36, 1, 27, 5, 32, 35, 54, 29, 15, 59.

Вариант 33: 20, 27, 55, 26, 46, 8, 14, 48, 5, 14, 3, 14, 24, 42, 35, 53, 47, 38, 44, 58, 16, 19, 36, 57, 56.

Вариант 34: 43, 20, 7, 30, 6, 17, 21, 55, 51, 13, 5, 22, 46, 57, 39, 29, 16, 15, 43, 45, 10, 42, 9, 16, 29.

Вариант 35: 19, 53, 41, 39, 51, 20, 21, 51, 49, 49, 50, 42, 30, 23, 38, 7, 14, 46, 4, 23, 3, 42, 29, 43, 27.

Вариант 36: 24, 47, 46, 35, 22, 6, 34, 26, 22, 38, 36, 32, 41, 4, 19, 40, 44, 50, 38, 4, 40, 54, 58, 38, 33.

Вариант 37: 45, 15, 37, 30, 31, 22, 20, 18, 27, 50, 22, 6, 39, 39, 47, 27, 3, 27, 47, 34, 30, 50, 31, 52, 22.

Вариант 38: 10, 8, 43, 8, 14, 6, 26, 22, 38, 50, 34, 58, 58, 33, 26, 36, 37, 14, 51, 23, 33, 1, 9, 49, 27.

Вариант 39: 47, 28, 5, 14, 2, 16, 22, 32, 49, 4, 42, 41, 15, 28, 46, 50, 53, 27, 16, 21, 46, 51, 11, 25, 19.

Вариант 40: 11, 39, 1, 45, 31, 51, 55, 41, 57, 11, 17, 51, 57, 44, 4, 20, 15, 2, 6, 45, 41, 54, 8, 54, 40.

Задание 2

Построение Хеш-таблицы

**Краткие теоретические сведения**

Для наглядности рассмотрим стандартные контейнеры и затраты времени на выполнение различных операций.

| **Контейнер \ операция** | **Insert (Вставка элемента)** | **Remove (удаление элемента)** | **Find (поиск элемента)** |
| --- | --- | --- | --- |
| Array | O(N) | O(N) | O(N) |
| List | O(1) | O(1) | O(N) |
| Sorted array | O(N) | O(N) | O(logN) |
| Бинарное дерево поиска | O(logN) | O(logN) | O(logN) |
| **Хеш-таблица** | O(1) | O(1) | O(1) |

Из этой таблицы очень хорошо понятно, почему же стоит использовать хеш-таблицы. Все операции с хеш-таблицей могут быть выполнены за один шаг. Однако, такой результат может быть получен при оптимальном выборе хеш-функции. Но тогда возникает противоположный вопрос: *почему же тогда ими не пользуются постоянно?* Ответ очень прост: как и всегда, невозможно получить все сразу, а именно: и скорость, и память. Хеш-таблицы тяжеловесные, и, хоть они и быстро отвечают на вопросы основных операций, пользоваться ими все время очень затратно.

**Понятие хеш-таблицы.** Хеш-таблица — это контейнер. Реализация у него, возможно, и не очевидная, но довольно простая.

Мы определяем функцию хеширования, которая по каждому входящему элементу будет определять натуральное число. А уже дальше по этому натуральному числу мы будем класть элемент в (допустим) массив. Тогда имея такую функцию мы можем за O(1) обработать элемент.

**Проблема коллизии.** Естественно, возникает вопрос, почему невозможно такое, что мы попадем дважды в одну ячейку массива, ведь представить функцию, которая ставит в сравнение каждому элементу совершенно различные натуральные числа просто невозможно. Именно так возникает проблема коллизии, или проблемы, когда хеш-функция выдает одинаковое натуральное число для разных элементов.

Существует несколько решений данной проблемы: метод цепочек и метод двойного хеширования.

**Решение проблемы коллизии методом цепочек.** Хеш-функция (при входе g) будет возвращать натуральное число s, которое будет для нас начальным. То есть первое, что мы сделаем, попробуем поставить элемент g на позицию s в нашем массиве. Но что, если это место уже занято? Будем искать ближайшее свободное увеличивая позицию на 1. При выходе за пределы массива переходим на первую строку массива и продолжает поиск свободной ячейки.

**Решение проблемы коллизии методом двойного хеширования.**

Мы будем (как несложно догадаться из названия) использовать две хеш-функции, возвращающие взаимопростые натуральные числа. Одна хеш-функция (при входе g) будет возвращать натуральное число s, которое будет для нас начальным. То есть первое, что мы сделаем, попробуем поставить элемент g на позицию s в нашем массиве. Но что, если это место уже занято? Именно здесь нам пригодится вторая хеш-функция, которая будет возвращать t — шаг, с которым мы будем в дальнейшем искать место, куда бы поставить элемент g.

Мы будем рассматривать сначала элемент s, потом s + t, затем s + 2\*t и т.д. Естественно, чтобы не выйти за границы массива, мы обязаны смотреть на номер элемента по модулю (остатку от деления на размер массива).

Наконец мы объяснили все самые важные моменты, можно перейти к непосредственному написанию кода, где уже можно будет рассмотреть все оставшиеся нюансы.

**Реализация хеш-таблицы**

Для наглядности будем реализовывать хеш-таблицу, хранящую строки.

Начнем с определения самих хеш-функций, реализуем их методом Горнера. Важным параметром корректности хеш-функции является то, что возвращаемое значение должно быть взаимопросто с размером таблицы. Для уменьшения дублирования кода, будем использовать две структуры, ссылающиеся на реализацию самой хеш-функции.

**Задание.** Построить 6 хеш-таблиц для слов в соответствии с вариантом.

Размер первых трех хеш-таблицы принять равным (количество заданных слов + 4).

Размер последних трех хеш-таблицы принять равным (количество заданных слов).

Даны следующие три хеш-функции.

S1 = Код первой буквы слова % размер хеш-таблицы +1.

S2 = Сумма кодов всех символов слова % размер хеш-таблицы +1.

S3 = (Сумма кодов всех символов слова, умноженных на позицию символа в слове) % размер хеш-таблицы + 1.

где % – операция взятия остатка от деления (**mod**). Например, 25 % 3 = 25 **mod** 3 = 1;

25 % 10 = 5; 22 % 8 = 6.

Коды букв приведены в таблице:

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| a | b | c | d | e | f | g | h | i | J | k | l | m | n | o | p | q | r | s | t | u | v | w | x | y | z |
| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 | 21 | 22 | 23 | 24 | 25 | 26 |

Например, для слова **axe** и размере хеш-таблицы равном 10, получим:

S1 = 1, так как код первой буквы слова **axe** =1 + 1 = 2.

S2 = (Код буквы «a» + Код буквы «x»+ Код буквы «e») % 10 = (1+24+5) %10 + 1=1.

S2 = (Код буквы «a»\*1 + Код буквы «x»\*2+ Код буквы «e»\*3) % 10 = (1\*1+24\*2+5\*3) %10 + 1=5.

Определить среднее количество операций для построения хеш-таблицы и поиска каждого слова по одному разу в хеш-таблице. На основе полученных данных выбрать наиболее предпочтительную из двух хеш-функций для каждого размера хеш-таблицы.

**Пример**: Даны слова: allo, coral, boat, brig, echo, lost, sample. Построим 3 хеш-таблицы размером 11 с использованием каждой из трех хеш-функций и определим наиболее предпочтительную.

Для помощи в заполнении таблиц приветствуется использование программного обеспечения, написанного на языке программирования или использование электронной таблицы Excel.

Вычислим значения хеш-функций для всех слов.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| № пп | Слово | S1 | S2 | S3 |
| 1 | allo | 1 % 11+1 = 2 | (1+12+12+15) %11+1=40%11 +1 = 8 | (1+12\*2+12\*3+15\*4) %11+1=121%11 +1 = 1 |
| 2 | coral | 3 % 11+1 = 4 | (3+15+18+1+12) %11+1=49%11+1 = 6 | (3+15\*2+18\*3+1\*4+12\*5) %11+1=141%11+1 = 10 |
| 3 | coat | 3 % 11+1 = 4 | (3+15+1+20) %11+1=39%11+1 =7 | (3+15\*2+1\*3+20\*4) %11+1=116%11+1 =7 |
| 4 | brig | 2 % 11+1 = 3 | (2+18+9+7) %11+1=36%11+1 = 6 | (2+18+2+9+3+7\*4) %11+1=93%11+1 = 6 |
| 5 | echo | 5 % 11+1 = 6 | (5+3+8+15) %11+1=31%11+1 = 10 | (5+3\*2+8+3+15+4) %11+1=65%11+1 = 11 |
| 6 | lost | 12 %11+1 = 2 | (12+15+19+20) %11 +1 = 66%11+1 =1 | (12+15\*2+19\*3+20\*4) %11 +1 = 179%11+1 =4 |
| 7 | sample | 19 %11+1 = 9 | (19+1+13+16+12+5) %11 +1= 66 %11+1 = 1 | (19+1\*2+13\*3+16\*4+12\*5+5\*6) %11 +1= 114 %11+1 = 5 |

Построим хеш-таблицы размером 11 для различных хеш-функций: S1, S2, S3.

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |  |  |  |
| 1 |  |  | 1 | lost |  | 1 | allo |  |
| 2 | allo |  | 2 | sample | +1 | 2 |  |  |
| 3 | brig |  | 3 |  |  | 3 |  |  |
| 4 | coral |  | 4 |  |  | 4 | lost |  |
| 5 | coat | +1 | 5 |  |  | 5 | sample |  |
| 6 | echo |  | 6 | coral |  | 6 | brig |  |
| 7 | lost | +5 | 7 | coat |  | 7 | coat |  |
| 8 |  |  | 8 | allo |  | 8 |  |  |
| 9 | sample |  | 9 | brig | +3 | 9 |  |  |
| 10 |  |  | 10 | echo |  | 10 | coral |  |
| 11 |  |  | 11 |  |  | 11 | echo |  |

Таким образом, хеш-таблица, построенная по хеш-функции S1 требует при поиске по одному разу каждого слова 6 лишние операции, хеш-таблица, построенная по хеш-функции S2, требует при поиске по одному разу каждого слова 4 лишние операции, а хеш-таблица, построенная по хеш-функции S3 не требует лишних операций, то есть количество операций равно количеству искомых слов. Таким образом, наиболее предпочтительная хеш-функция S3.

**Пример**: Даны слова: allo, coral, boat, brig, echo, lost, sample. Построим 3 хеш-таблицы размером 7 с использованием каждой из трех хеш-функций и определим наиболее предпочтительную.

Вычислим значения хеш-функций для всех слов.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| № пп | Слово | S1 | S2 | S3 |
| 1 | allo | 1 % 7+1 = 2 | (1+12+12+15) %7+1=40%7 +1 = 6 | (1+12\*2+12\*3+15\*4) %7+1=121%7 +1 = 3 |
| 2 | coral | 3 % 7+1 = 4 | (3+15+18+1+12) %7+1=49%7+1 = 1 | (3+15\*2+18\*3+1\*4+12\*5) %7+1=141%7+1 = 2 |
| 3 | coat | 3 % 7+1 = 4 | (3+15+1+20) %7+1=39%7+1 =5 | (3+15\*2+1\*3+20\*4) %7+1=116%7+1 =5 |
| 4 | brig | 2 % 7+1 = 3 | (2+18+9+7) %7+1=36%7+1 = 2 | (2+18+2+9+3+7\*4) %7+1=93%7+1 = 3 |
| 5 | echo | 5 % 7+1 = 6 | (5+3+8+15) %7+1=31%7+1 = 4 | (5+3\*2+8+3+15+4) %7+1=65%7+1 = 3 |
| 6 | lost | 12 %7+1 = 6 | (12+15+19+20) %7 +1 = 66%7+1 =4 | (12+15\*2+19\*3+20\*4) %7 +1 = 179%7+1 =5 |
| 7 | sample | 19 %7+1 = 6 | (19+1+13+16+12+5) %7 +1= 66 %7+1 = 4 | (19+1\*2+13\*3+16\*4+12\*5+5\*6) %7 +1= 114 %7+1 = 3 |

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |  |  |  |
| 1 | sample | +2 | 1 | Coral |  | 1 | sample | +5 |
| 2 | allo |  | 2 | Brig |  | 2 | Coral |  |
| 3 | brig |  | 3 | sample | +6 | 3 | Allo |  |
| 4 | coral |  | 4 | Echo |  | 4 | brig | +1 |
| 5 | coat | +1 | 5 | Coat |  | 5 | Coat |  |
| 6 | echo |  | 6 | Allo |  | 6 | echo | +1 |
| 7 | lost | +1 | 7 | lost | +3 | 7 | lost | +2 |

Таким образом, хеш-таблица, построенная по хеш-функции S1, требует при поиске по одному разу каждого слова 4 лишние операции, хеш-таблица, построенная по хеш-функции S2, требует при поиске каждого слова по одному разу 8 лишних операций, а хеш-таблица, построенная по хеш-функции S3 требует при поиске по одному разу каждого слова 9 лишних операций. Таким образом, наиболее предпочтительная хеш-функция S1.

Чем больше таблица, тем больше возможностей поставить слово на свое место и таблица, занимая больше места в памяти позволяет сэкономит время. При большем размере таблицы наиболее эффективной оказалась третья хеш-функция, а при меньшем размере таблицы – первая хеш-функция.

Для помощи в заполнении таблиц приветствуется использование программного обеспечения, написанного на языке программирования или использование электронной таблицы Excel.

Вариант 1

fruit, apricot, pineapple, banana, grape, pear, melon, lemon, mandarin, peach, plum, apple, lime, kiwi, fig, mango, pomelo

Вариант 2

silver, beige, snowy, white, turquoise, pale, bronze, yellow, pearl, green, golden, indigo, brown, cream, azure, lemon, aqua, olive, orange

Вариант 3

berry, cherry, blackberry, strawberry, viburnum, cranberry, dogwood, raspberry, blueberry, elder, gooseberry, sea-buckthorn, rowan, redcurrant

Вариант 4

house, apartment, room, balcony, bedroom, closet, doorbell, door, upstairs, roof, window, garage, hall, kitchen, hallway, mailbox, shower, stairs, toilet

Вариант 5

bathroom, toilet, restroom, freshener, towel, hamper, scale, shelf, fan, mirror, sink, bath, toothpaste, soap, shower, sponge, comb, razor

Вариант 6

tableware, plate, bowl, cup, mug, spoon, fork, knife, glass, saucepan, lid, ladle, jug, kettle, coffeepot, teapot, pan, pepper shaker, bowl, bottle, napkin

Вариант 7

kitchen, dishwasher, faucet, sink, sponge, microwave, canister, stove, burner, oven, potholder, toaster, cookbook, freezer, table, chair, pail

Вариант 8

furniture, box, bookcase, cupboard, shelf, shelving, table, desk, chair, sofa, bench, bed, stool, bureau, mirror, lamp, safe, interior

Вариант 9

health, plaster, bruise, bump, cough, cut, diarrhea, fever, headache, medicine, scratch, shot, sick, sneeze, sunburn, thermometer, tooth

Вариант 10

town, city, building, skyscraper, palace, bank, bookstore, bus, stop, bakery, coffee shop, café, restaurant, drugstore, pharmacy, office

Вариант 10

dentist, doctor, hospital, flower, shop, gas, station, library, theatre, cinema, museum, church, park, pool, post, office, hotel

Вариант 11

university, school, supermarket, store, train, station, underground, ground, street, avenue, square, alley, area, map, shop, library

Вариант 12

flower, rose, tulip, carnation, gladiolus, lily, orchid, peony, daisy, violet, lotus, sunflower, snowdrop, narcissus, daffodil, dahlia, iris, poppy, crocus

Вариант 13

flower, dandelion, lavender, camellia, bouquet, pumpkin, dill, haricot, garlic, sorrel, goose, sea, buckthorn, rowan, choke, currant

Вариант 14

vegetable, basil, eggplant, bean, pea, cabbage, potato, onion, carrot, cucumber, pepper, parsley, tomato, radish, beet, celery,

Вариант 15

transport, airplane, ambulance, bus, car, truck, fire truck, helicopter, motorcycle, overpass, police, car, road, boat, sailboat, ship

Вариант 16

street, submarine, subway, tractor, traffic, train, crossing, van, scooter, bicycle, motor, metro, road, light, line, wheel, boat, flagman, liner

Вариант 17

airport, airplane, wing, tail, cockpit, cabin, fuselage, carriage, lavatory, window, aisle, seat, armrest, ticket, suitcase, backpack, ship

Вариант 18

baggage, life vest, carry-on, flight, takeoff, landing, runway, gate, boarding, passenger, passport, visa, customs, terminal, schedule, security

Вариант 19

guard, trolley, liquid, hijack, emergency, delay, destination, middle, index, little, finger, fingernail, suit, woolen, jean, jupiter, saturn, uranus

Вариант 20

clothes, blouse, skirt, dress, shirt, trousers, socks, stockings, tights, swimsuit, underpants, sweater, cardigan, jersey, jeans, shorts

Вариант 21

shirt, cap, hat, shoes, boots, sneakers, slippers, sandals, jacket, coat, raincoat, fur coat, belt, tie, glove, scarf, nightie, pajama, tracksuit

Вариант 22

anorak, dungarees, umbrella, watch, earring, ring, necklace, bracelet, glasses, chiffon, chinchilla, cashmere, cambric, flax, jeans

Вариант 23

textile, cotton, fur, silk, synthetic, velour, velvet, viscose, suede, chamois, stockinet, sateen, nylon, mink, leather, lace, gauze, crochet

Вариант 24

space, planet, star, sun, mercury, venus, earth, mars, neptune, satellite, moon, comet, asteroid, meteorite, spaceship, cosmonaut, astronaut

Вариант 25

sport, football, soccer, basketball, volleyball, baseball, hockey, badminton, tennis, rugby, golf, billiards, swimming, biathlon, wrestling

Вариант 26

Profession, teacher, interpreter, waiter, taxi, driver, driver, engineer, lawyer, accountant, manager, consultant, administrator, economist, financier

Вариант 27

builder, nurse, secretary, vet, scientist, dentist, writer, musician, actor, artist, singer, stylist, barber, journalist, composer, courier, shop, assistant

Вариант 28

guide, designer, programmer, architect, coach, athlete, barman, bodyguard, cashier, cleaner, cook, electrician, plumber, librarian, doctor

Вариант 29

police, fireman, military, politician, postman, priest, gymnastics, fencing, race, polo, bowling, marinated tomatoes, tea, coffee, cocoa

Вариант 30

pet, cat, kitten, dog, puppy, hamster, guinea pig, farm, chicken, rooster, turkey, goat, kid, sheep, lamb, bull, cow, calf, horse, pig, piglet, rabbit

Вариант 31

food, ham, sausage, meat, egg, pizza, soup, bread, milk, porridge, flour, garlic, salad, fish, chicken, pork, beef, steak, spice, butter, potato,

Вариант 32

macaroni, cheese, rice, pasta, ketchup, sauce, bean, pepper, salt, sandwich, cutlet, caviar, vegetables, fruit, baked, fried, pickled, cucumbers,

Вариант 33

time, past, future, present, age, millennium, century, decade, year, month, week, day, hour, half, minute, second, watch, clock, calendar, schedule

Вариант 34

family, mother, father, parents, son, daughter, children, sister, brother, grandson, aunt, uncle, niece, nephew, man, woman, baby

Вариант 35

body, head, shoulder, arm, hand, elbow, chest, stomach, back, bottom, thigh, waist, leg, knee, calf, calves, foot, feet, ankle, heel

Вариант 36

thumb, finger, shin, toe, toenail, neck, face, nose, mouth, lip, tooth, teeth, tongue, eye, eyebrow, eyelash, pupil, eyelid, chin, cheek, forehead

Вариант 37

Tool, hammer, mallet, handsaw, sandpaper, plane, drill, hacksaw, vise, screwdriver, awl, file, chisel, wrench, pliers, scissors, pocketknife

Вариант 38

toolbox, axe, spade, bucket, ladder, stepladder, nail, screw, bolt, nut, tree, smoke, list, trest, **chair, armchair, sconce, ceiling, painting, frame, bookcase**

Вариант 39

piece, coin, one, penny, pence, two, fivepenny, five, tenpenny, ten, twenty, twenty, fifty, fifty, pound, quid, quarter, half, dollar, two, hundred

Вариант 40

sense, taste, sight, smell, touch, hearing, glimpse, scan, see, notice, glance, observe, listen, hear, watch, look, stare, overhear, delicious

Задание 3

**Работа со структурами данных (коллекциями)**

**Цель работы**: изучить методы работы с динамическими списками библиотеки .NET.

**Краткие теоретические сведения**

**1.1. Коллекции**

Коллекция в C# – совокупность объектов. В среде .NET Framework имеется немало интерфейсов и классов, в которых определяются и реализуются различные типы коллекций. Коллекции упрощают решение различных задач программирования, требующих сложных структур данных. К ним относятся списки, стеки, динамические массивы, очереди, хеш-таблицы.

Главное преимущество коллекций заключается в том, что они стандартизируют обработку групп объектов в программе. В среде .NET Framework поддерживаются четыре типа коллекций: необобщенные, специальные, с поразрядной организацией и обобщенные.

Первоначально существовали только классы необобщенных коллекций. Затем появились обобщенные версии классов и интерфейсов.

Необобщенные коллекции оперируют данными типа Object (базовый класс для всех классов). Таким образом, они служат для хранения объектов любого типа, причем в одной коллекции допускается наличие объектов разных типов. Это преимущество, если нужно оперировать объектами разных типов в одном списке или если типы объектов заранее неизвестны. Но эти коллекции не обеспечивают типовую безопасность. Они размещаются в пространстве имен System.Collections.

Специальные коллекции оперируют данными конкретного типа (например, для символьных строк StringCollection и StringDictionary) или же делают это каким-то особым образом (для битовых элементов длиной 32 бита – BitVector32). Они размещаются в пространстве имен System.Collections.Specialized.

Коллекции с поразрядной организацией поддерживают поразрядные операции над двоичными разрядами (И, ИЛИ). К ним относится только класс BitArray. Они размещаются в пространстве имен System.Collections.

Обобщенные коллекции обеспечивают обобщенную реализацию нескольких стандартных структур данных, включая связанные списки, стеки, очереди и словари. Такие коллекции являются типизированными. Это означает, что в обобщенной коллекции могут храниться только такие элементы данных, которые совместимы по типу с данной коллекцией. Это исключает случайное несовпадение типов и обеспечивают типовую безопасность. Они размещаются в пространстве имен System.Collections.Generic.

Инициализация коллекций осуществляется следующим образом:

List<int> myList = new List<int>{0,1,2,3,4,5,6,7,8};

Если контейнер управляет коллекцией классов и структур, то можно смешивать синтаксис инициализации объектов с инициализации объектов с синтаксисом инициализации коллекций, создавая некоторый функциональный код:

List<Point> m = new List<int>

{

new Point {X=2, Y=3},

new Point {X=6, Y=4},

new Point (Color.Red) {X=2, Y=3}

};

**1.2. Класс List<T>**

Класс List<T> представляет собой односвязный список с возможностью обращения к элементам по номерам и имеет следующие методы:

1. Метод AddRange() – добавление множества элементов в коллекцию за один прием. Метод AddRange принимает один аргумент типа IEnumerable<T>

points.AddRange(new Point[]

{

new Point {X=5, Y=6},

new Point(5,8)

}

);

2. Метод Insert() – вставка элемента в определенную позицию.

Например, points.Insert(3, new Point(3,5));

Если указывается индекс, превышающий количество элементов в коллекции, то генерируется исключение типа ArgumentOutOfRangeException.

3. Метод InsertRange() – вставка нескольких элементов в определенную позицию за один прием аналогично AddRange.

4. Доступ к элементам осуществляется с помощью индексатора []. Первый элемент доступен по индексу 0.

Например, Point p = points[0];

4. Проход по элементам коллекции возможен с помощью оператора foreach благодаря реализации интерфейса IEnumerable. Например,

foreach(Point p in points)

Console.WriteLine(p);

5. Метод ForEach(), который может использоваться вместо оператора foreach также, объявленный с параметром Action<T>. Action должен быть объявлен как метод с параметром того же типа, что и элемент коллекции и возвращающий void.

Например,

points.ForEach(Console.WriteLine);

points.ForEach(p=>Console.WriteLine(p));

6. Метод RemoveAt() – удаление элемента по индексу.

7. Метод Remove() – удаление элемента по ссылке.

8. Метод RemoveRange() удаляет множество элементов из коллекции. Первый параметр определяет индекс, начиная с которого располагаются удаляемые элементы, а второй параметр задает количество удаляемых элементов.

10. Метод Clear() – удаление всех элементов коллекции.

11. Метод IndexOf(), LastIndexOf(), FindIndex(),   
FindLastIndex(), Find(), FindLast() – поиск элемента коллекции.

16. Метод Exists() – проверка существования элемента.

17. Метод Sort() осуществляет сортировку элементов.

**1.3. Класс LinkedList<T>**

Класс LinkedList<T> является представителем двунаправленного списка, где каждый элемент списка содержит ссылки на предыдущий и следующий элементы (рис. 1). Итератор поддерживает обход в обе стороны. Реализует методы получения, удаления и вставки в начало, середину и конец списка.



Рис. 1. Двусвязный список

Преимущество связного списка проявляется в том, что операция вставки элемента в середину выполняется очень быстро. При этом только ссылки Next предыдущего элемента и Previous следующего элемента должны быть изменены так, чтобы указывать на вставляемый элемент (рис. 2).

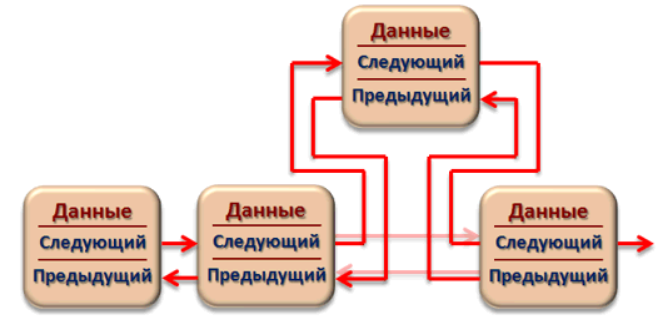


Рис. 2. Добавление элемента

Естественно, у связных списков есть и свои недостатки. Так, например, все элементы связных списков доступны лишь друг за другом. Поэтому для нахождения элемента, находящегося в середине или конце списка, требуется довольно много времени. Двусвязный список не может просто хранить элементы внутри себя. Вместе с каждым из них ему необходимо иметь информацию о следующем и предыдущем элементах. Для этого LinkedList<T> содержит элементы класса LinkedListNode<T>. Класс LinkedListNode<T> определяет свойства List, Next, Previous и Value. Свойство List возвращает объект LinkedList<T>, ассоциированный с узлом. Свойства Next и Previous предназначены для итераций по списку и для доступа к следующему и предыдущему элементам. Свойство Value типа T возвращает элемент, ассоциированный с узлом.

Сам класс LinkedList<T> определяет члены для доступа к первому (First) и последнему (Last) элементам в списке, для вставки элементов в определенные позиции (AddAfter(), AddBefore(), AddFirst(), AddLast()), для удаления элементов из заданных позиций (Remove(), RemoveFirst(), RemoveLast()) и для нахождения элементов, начиная поиск либо с начала (Find()), либо с конца (FindLast()) списка.

Пример использования связных списков:

// Создание двусвязного списка

LinkedList<int> link = new LinkedList<int>();

// Добавление элементов

link.AddLast(1);

link.AddLast(2);

link.AddFirst(0); //Будет получен список «0 1 2»

//Перебор элементов в прямом направлении

LinkedListNode<string> node;

for (node = link.First; node != null; node = node.Next)

Console.Write(node.Value + "\t");

//Перебор элементов в обратном направлении

for (node = link.Last; node != null; node = node.Previous)

Console.Write(node.Value + "\t");

//Вывод пятого элемента

for (node = link.First; i<5; node = node.Next)

Console.Write(node.Value + "\t");

//Добавление после элемента node числа 2

link.AddAfter(node, 2);

**Класс Stack<T>**

Класс Stack<T> представляет коллекцию элементов, работающую по алгоритму «последний вошел – первый вышел» (LIFO) и имеет следующие методы:

Метод Push() – вставка элемента в вершину стека.

Метод Pop() – извлечение элемента из вершины стека.

Метод Peek() – вернуть элемент из вершины стека без его удаления.

При попытке извлечения элемента из пустого стека генерируется исключение InvalidOperationException.

**1.4. Класс Queue<T>**

Класс Queue<T> представляет коллекцию элементов, работающую по принципу «первый вошел – первый вышел» (FIFO) и имеет следующие методы:

Метод Enqueue () – вставка элемента в конец очереди.

Метод Dequeue() – извлечение элемента из начала очереди.

Метод Peek() – вернуть элемент из начала очереди без его удаления.

При попытке извлечения элемента из пустой очереди генерируется исключение InvalidOperationException.

**1.5. Класс BitArray**

Класс BitArray служит для хранения отдельных битов в коллекции. Изменение количества элементов можно осуществить через свойство Length.

BitArray bits2 = new BitArray(7);

bits1.SetAll(false);

bits1[3] = true;

bits1.Length = bits1.Length + 1;

bits1[7] = true;

**Порядок выполнения работы**

1. Создайте приложение Windows Forms в среде C#.

2. Создайте исходные и результирующие контейнеры с элементами, являющимися целыми числами, в соответствии с индивидуальным заданием. Обычно необходимо создать 1-2 исходных контейнера и 1-2 – результирующих.

3. Напишите метод для ввода исходных данных и заполнения исходных контейнеров начальными значениями, заданными пользователем.

В случае ввода некорректных значений необходимо, чтобы программа не завершалась аварийно.

4. Напишите метод для выполнения действий в соответствии с индивидуальным заданием.

5. В случае невозможности выполнения необходимых действий выдать соответствующее сообщение.

6. Выведите содержимое результирующих контейнеров и данные в соответствии с индивидуальным заданием на форму.

**Индивидуальные задания**

Вариант 1. Имеется 1 исходный стек и 1 результирующий. Перемещать элементы из первого стека во второй, пока значение вершины первого стека не станет четным. Если в первом стеке нет элементов с четными значениями, то переместить из первого стека во второй все элементы. Вывести на экран значения вершин обеих стеков. Если первый стек пустой, то вывести **null**.

Вариант 2. Имеется 1 исходный стек. Создать 2 результирующих стека, переместив в первый из них все элементы исходного стека с четными значениями, а во второй – с нечетными. Один из результирующих стеков может остаться пустым. Вывести значения вершин обеих стеков. Если стек пустой, то вывести **null**.

Вариант 3. Имеется очередь. Создать 2 результирующих стека, переместив в первый из них все элементы исходного стека с четными значениями, а во второй – с нечетными. Один результирующих стеков может остаться пустым. Вывести значения вершин обеих стеков. Если стек пустой, то вывести **null**.

Вариант 4. Имеется очередь. Создать двусвязный список. Извлекать из очереди элементы, пока значение первого элемента очереди не станет четным и заносить значения извлеченных элементов в связанный список. Если очередь не содержит элементов с четными значениями, то извлечь все ее элементы. Вывести на экран **null**, если первая очередь стала пустой.

Вариант 5. Имеется очередь. Переместить из этой очереди в результирующую очередь вторую N элементов. Если исходная очередь содержит менее N элементов, то перенести все элементы из исходной очереди в результирующую. Вывести на экран **null**, если первая очередь стала пустой.

Вариант 6. Имеется очередь. Перемещать элементы из нее в конец результирующей очереди, пока значение начального элемента первой очереди не станет четным. Если входная очередь не содержит четных элементов, то перенести все элементы из первой очереди во вторую. Вывести на экран **null**, если первая очередь стала пустой.

Вариант 7. Имеются 2 входные очереди с одинаковым количеством элементов. Создать результирующую очередь, в которой элементы из первой и второй очередей чередуются. Вывести на экран первый и последний элементы полученной очереди.

Вариант 8. Имеется двусвязный список и число D. Найти в двусвязном списке первое встретившееся число D и вывести на экран 2 числа перед найденным и 2 числа после найденного. Если перед и после найденного числа не имеется 2-х элементов, то вместо отсутствующих элементов вывести «**null»**.

Вариант 9. Имеется двусвязный список и число D. Найти в двусвязном списке первое встретившееся число D и перенести элементы перед найденным в обратном порядке в результирующую очередь. Если перед найденным элементом нет чисел, то очередь останется пустой. На форму вывести перенесенные элементы, и затем первый элемент в полученной очереди. Если очередь осталась пустой, то вывести «**null»**.

Вариант 10. Имеется очередь и массив с необязательно одинаковым количеством элементов. Перенести элементы из них поочередно в двусвязный список. В конец перенести все оставшиеся элементы из более длинного исходного контейнера.

Вариант 11. Имеется двусвязный список и числа D, D1 и D2. Найти в контейнере элемент равный числу D, и занести число D1 перед ним, а число D2 после него. На форму вывести количество чисел равных D1 в контейнере.

Вариант 12. Имеется двусвязный список и числа D1 и D2. Найти в контейнере все элементы равные числу D1, и занести число D2 перед и после каждого из них. На экран вывести количество элементов равных D1 в контейнере. Если их нет, то вывести **null**.

Вариант 13. Имеется двусвязный список, очередь и число D. Найти в двусвязном списке первое попавшееся число D, и занести перед ним элементы из очереди до тех пор, пока первым элементом очереди не станет число D. Если в очереди нет числа равного D, то перенести из очереди все элементы. На экран вывести количество первый элемент очереди. Если очередь пуста, то вывести «**null»**.

Вариант 14. Имеется двусвязный список. Продублировать в нем первый и последний элементы. Продублированные элементы добавлять перед соответствующими существующими элементами.

Вариант 15. Имеется двусвязный список. Продублировать в нем все элементы с нечетными номерами. Продублированные элементы добавлять перед соответствующими существующими элементами с такими же значениями.

Вариант 16. Имеется двусвязный список. Удалить из него все элементы с нечетными номерами. Вывести на экран значения всех удаленных элементов.

Вариант 17. Имеется двусвязный список и число *D*. Переместить в конец списка все элементы из него, равные числу *D*. Вывести на экран количество перенесенных элементов. Если перенесенных элементов нет, то вывести «**null»**.

Вариант 18. Имеется список. Создать битовый массив такой же размерности, в котором элементы в позиции *i* будут принимать значение **true**, если число *i*-ой позиции исходного списка отрицательное и **false** – в противном случае.

Вариант 19. Имеется список. Создать битовый массив такой же размерности, в котором элементы в позиции *i* будут принимать значение **true**, если число в *i*-ой позиции меньше числа в *i*+1 позиции и **false** – в противном случае. В последний элемент битового массива занести **true**, если последний элемент списка положительный и **false** – в противном случае.

Вариант 20. Имеется 1 исходный стек и 1 результирующий. Перемещать элементы из первого стека во второй, пока значение вершины первого стека не станет четным. Если в первом стеке нет элементов с четными значениями, то переместить из первого стека во второй все элементы. Вывести на экран значения вершин обеих стеков. Если первый стек пустой, то вывести **null**.

Вариант 21. Имеется 1 исходный стек. Создать 2 результирующих стека, переместив в первый из них все элементы исходного стека с четными значениями, а во второй – с нечетными. Один из результирующих стеков может остаться пустым. Вывести значения вершин обеих стеков. Если стек пустой, то вывести **null**.

Вариант 22. Имеется очередь. Создать 2 результирующих стека, переместив в первый из них все элементы исходного стека с четными значениями, а во второй – с нечетными. Один результирующих стеков может остаться пустым. Вывести значения вершин обеих стеков. Если стек пустой, то вывести **null**.

Вариант 23. Имеется очередь. Создать двусвязный список. Извлекать из очереди элементы, пока значение первого элемента очереди не станет четным и заносить значения извлеченных элементов в связанный список. Если очередь не содержит элементов с четными значениями, то извлечь все ее элементы. Вывести на экран **null**, если первая очередь стала пустой.

Вариант 24. Имеется очередь. Переместить из этой очереди в результирующую очередь вторую N элементов. Если исходная очередь содержит менее N элементов, то перенести все элементы из исходной очереди в результирующую. Вывести на экран **null**, если первая очередь стала пустой.

Вариант 25. Имеется очередь. Перемещать элементы из нее в конец результирующей очереди, пока значение начального элемента первой очереди не станет четным. Если входная очередь не содержит четных элементов, то перенести все элементы из первой очереди во вторую. Вывести на экран **null**, если первая очередь стала пустой.

Вариант 26. Имеются 2 входные очереди с одинаковым количеством элементов. Создать результирующую очередь, в которой элементы из первой и второй очередей чередуются. Вывести на экран первый и последний элементы полученной очереди.

Вариант 27. Имеется двусвязный список и число D. Найти в двусвязном списке первое встретившееся число D и вывести на экран 2 числа перед найденным и 2 числа после найденного. Если перед и после найденного числа не имеется 2-х элементов, то вместо отсутствующих элементов вывести «**null»**.

Вариант 28. Имеется двусвязный список и число D. Найти в двусвязном списке первое встретившееся число D и перенести элементы перед найденным в обратном порядке в результирующую очередь. Если перед найденным элементом нет чисел, то очередь останется пустой. На форму вывести перенесенные элементы, и затем первый элемент в полученной очереди. Если очередь осталась пустой, то вывести «**null»**.

Вариант 29. Имеется очередь и массив с необязательно одинаковым количеством элементов. Перенести элементы из них поочередно в двусвязный список. В конец перенести все оставшиеся элементы из более длинного исходного контейнера.

Вариант 30. Имеется двусвязный список и числа D, D1 и D2. Найти в контейнере элемент равный числу D, и занести число D1 перед ним, а число D2 после него. На форму вывести количество чисел равных D1 в контейнере.

Вариант 31. Имеется двусвязный список и числа D1 и D2. Найти в контейнере все элементы равные числу D1, и занести число D2 перед и после каждого из них. На экран вывести количество элементов равных D1 в контейнере. Если их нет, то вывести **null**.

Вариант 32. Имеется двусвязный список, очередь и число D. Найти в двусвязном списке первое попавшееся число D, и занести перед ним элементы из очереди до тех пор, пока первым элементом очереди не станет число D. Если в очереди нет числа равного D, то перенести из очереди все элементы. На экран вывести количество первый элемент очереди. Если очередь пуста, то вывести «**null»**.

Вариант 33. Имеется двусвязный список. Продублировать в нем первый и последний элементы. Продублированные элементы добавлять перед соответствующими существующими элементами.

Вариант 34. Имеется двусвязный список. Продублировать в нем все элементы с нечетными номерами. Продублированные элементы добавлять перед соответствующими существующими элементами с такими же значениями.

Вариант 35. Имеется двусвязный список. Удалить из него все элементы с нечетными номерами. Вывести на экран значения всех удаленных элементов.

Вариант 36. Имеется двусвязный список и число *D*. Переместить в конец списка все элементы из него, равные числу *D*. Вывести на экран количество перенесенных элементов. Если перенесенных элементов нет, то вывести «**null»**.

Вариант 37. Имеется список. Создать битовый массив такой же размерности, в котором элементы в позиции *i* будут принимать значение **true**, если число *i*-ой позиции исходного списка отрицательное и **false** – в противном случае.

Вариант 38. Имеется список. Создать битовый массив такой же размерности, в котором элементы в позиции *i* будут принимать значение **true**, если число в *i*-ой позиции меньше числа в *i*+1 позиции и **false** – в противном случае. В последний элемент битового массива занести **true**, если последний элемент списка положительный и **false** – в противном случае.

Вариант 39. Имеется очередь. Создать двусвязный список. Извлекать из очереди элементы, пока значение первого элемента очереди не станет четным и заносить значения извлеченных элементов в связанный список. Если очередь не содержит элементов с четными значениями, то извлечь все ее элементы. Вывести на экран **null**, если первая очередь стала пустой.

Вариант 40. Имеется очередь. Переместить из этой очереди в результирующую очередь вторую N элементов. Если исходная очередь содержит менее N элементов, то перенести все элементы из исходной очереди в результирующую. Вывести на экран **null**, если первая очередь стала пустой.

Задание 4. **Рекурсивные алгоритмы**

**Рекурсивный алгоритм**— алгоритм, в описании которого прямо или косвенно содержится обращение к самому себе. В технике процедурного программирования данное понятие распространяется на функцию, которая реализует решение отдельного блока задачи посредством вызова из своего тела других функций, в том числе и себя самой. Если при этом на очередном этапе работы функция организует обращение к самой себе, то такая функция является рекурсивной.

Рекурсивный вызов процедуры мало чем отличается от вызова другой функции. Выполняется тот же код, но с другими значениями параметров и локальных переменных.

Важным для понимания идеи рекурсии является то, что в подобных подпрограммах можно выделить две серии шагов. Первая серия – это шаги рекурсивного погружения подпрограммы в себя до тех пор, пока выбранный параметр не достигнет граничного значения. Это важное требование должно выполняться, чтобы функция не создала бесконечную последовательность вызовов самой себя. Вторая серия – это шаги рекурсивного выхода до тех пор, пока выбранный параметр не достигнет конечного значения.

Рассмотрим примеры реализации рекурсий.

Пример реализации прямой рекурсии для вычисления факториала может выглядеть так:

Int64 Factorial(Int64 num)

{

return num \* Factorial (num – 1);

}

Здесь в теле функции Factorial осуществляется вывоз самой функции Factorial. Такая рекурсия будет осуществляться бесконечно, поскольку отсутствует условие выхода из рекурсии.

Из этого примера видно, что очевидная опасность рекурсии заключается в бесконечной рекурсии. Если неправильно построить алгоритм, то функция может пропустить условие остановки рекурсии и выполняться бесконечно.

Для нахождения условия остановки необходимо выделить базу рекурсии. Выделение базы рекурсии предполагает нахождение в решаемой задаче тривиальных случаев, результат для которых очевиден и не требует проведения расчетов. Верно найденная база рекурсии обеспечивает завершенность рекурсивных обращений, которые в конечном итоге сводятся к базовому случаю.

Функция также может вызывать себя бесконечно, если условие остановки не прекращает все возможные пути рекурсии. В следующей ошибочной версии функции Factorial2, функция будет бесконечно вызывать себя, если входное значение — не целое число, или если оно меньше 0.

Int64 Factorial2 (Int64 num)

{

Int64 fact2;

if (num == 0)

fact2 = 1;

else

fact2 = num \* Factorial2 (num-1);

return fact2;

}

Для устранения бесконечной рекурсии необходимо сделать следующее. Вначале функция Factorial2 должна проверять, что число меньше или равно 0. Факториал для чисел меньше нуля не определен, но это условие проверяется для подстраховки. Если бы функция проверяла только условие равенства числа нулю, то для отрицательных чисел рекурсия была бы бесконечной. Если входное значение меньше или равно 0, функция возвращает значение 1. В остальных случаях, значение функции равно произведению входного значения на факториал от входного значения, уменьшенного на единицу. То, что эта рекурсивная функция, в конце концов, остановится, гарантируется двумя фактами. Во-первых, при каждом последующем вызове, значение параметра num уменьшается на единицу. Во-вторых, когда num становится меньше либо равным 0, функция останавливает рекурсию.

Хотя рекурсия и может упростить понимание некоторых проблем, люди обычно не мыслят рекурсивно. Они обычно стремятся разбить сложные задачи на задачи меньшего объема, которые могут быть выполнены последовательно одна за другой до полного завершения. Для того чтобы думать рекурсивно, нужно разбить задачу на подзадачи, которые затем можно разбить на подзадачи меньшего размера. В какой-то момент подзадачи становятся настолько простыми, что могут быть выполнены непосредственно. Когда завершится выполнение подзадач, большие подзадачи, которые из них составлены, также будут выполнены. Исходная задача окажется выполненной, когда будут все выполнены образующие ее подзадачи.

Рассмотрим примеры решения задач с использованием рекурсивных функций.

**Пример 1.** Определить является ли строка палиндромом с использованием рекурсии.

Строка является палиндромом, если первый символ строки равен последнему символу строки и палиндромом является строка без первого и последнего символа. Такое определение палиндрома является рекурсивным.

Окончание рекурсии гарантируется уменьшением длины строки на 2 символа на каждой рекурсии и выходом из рекурсии при уменьшении строки до 0 символов.

Причем при очередном обращении к рекурсивной функции строка будет короче на 2 символа.

Когда получен результат по подстроке, можно определить является ли вся строка палиндромом: для этого необходимо, чтобы подстрока была палиндромом и первый и последний символы строки были равны друг другу.

Равенство первого и последнего символов строки определяется выражением s[0] == s[s.Length - 1].

Подстрока без первой и последней букв определяется выражением

s.Substring(1, s.Length – 2);

Рекурсивная функция возвращает результат в виде логической величины f равной true – если подстрока является палиндромом и false – в противном случае.

private void button1\_Click(object sender, EventArgs e)

{

string s = "aceeca";

bool b = palindrom(s);

Text = b.ToString();

}

private bool palindrom(string s)

{

if (s.Length == 0) return true;

bool f = palindrom(s.Substring(1, s.Length - 2));

return f & (s[0] == s[s.Length - 1]);

}

В выше приведенном примере имеется ошибка. Пример не учитывает тот факт, что строка может содержать нечетное количество символов и ее длина не станет равной 0.

**Пример 2.** Возвести число *a* в степень *n* с использованием рекурсии.

На каждом шаге рекурсии будем уменьшать степень в 2 раза, то есть *a* в степени *n* запишем в виде . Однако для нечетного *n* . Таким образом, степень будет уменьшаться на каждом шаге в 2 раза и при уменьшении степени до 1 задача становится тривиальной.

Программа будет иметь следующий вид

private void button1\_Click(object sender, EventArgs e)

{

int a = 3;

int n = 33;

Int64 b = V\_Stepen(a, n);

textBox1.Text = b.ToString();

}

private Int64 V\_Stepen(int a, int n)

{

if (n==1) return a;

Int64 f;

if (n%2==0)

f = V\_Stepen (a, n / 2) \* V\_Stepen (a, n / 2);

else

f = a \* V\_Stepen (a, n / 2) \* V\_Stepen (a, n / 2);

return f;

}

Данный алгоритм является неоптимальным по быстродействию.

**Пример 3.** Вычислить  с использованием рекуррентной формулы . При этом известно, что , x - целое, положительное.

Поскольку аргумент *x* функции *F* уменьшается на каждом шаге на 1, то в определенный момент будет вызвана функция *F*(1) для которой известен результат равный 1. С этого значения начинаются шаги рекурсивного выхода.



Таким образом функция *F*(*x*) будет иметь вид:

**int** *F*(**int** *x*)

{

**if**(*x*>1) **return** *F*(*x*-1)+2\**x*+1;

**else** **return** *x*;

}

Вызов функции, размещенный, например, в обработчике события Click для кнопки будет иметь вид:

**int** *x*=**int**.Parse(textBox1.Text);

**int** *y* = *F*(*x*);

label1.Text = *y*.ToString();

**Пример 4.** Вычислить , . Ясно, что по рекуррентной формуле можно осуществлять вычисления, начиная с . Значит для n=0,1,2 должны быть заданы начальные значения: F(0)=0, F(1)=2, F(2)=3, которые позволят начать шаги рекурсивного выхода.

Функция будет иметь вид:

**int** *F*(**int** *n*)

{

**if**(*n*<0) **throw** **new** ArgumentException();

**if**(*n*==0) return 0;

**else** **if**(*n*==1) **return** 2;

**else** **if**(*n*==2) **return** 3;

**else** **return** F(*n*-2)\*F(*n*-3);

}

Вызов рекурсивной функции *F*, размещенный, например, в функции buttonClick будет иметь вид:

**int** *x*=**int**.Parse(textBox1.Text);

**int** *y* = *F*(*x*);

label1.Text = *y*.ToString();

**Пример 5.** Ханойские Башни — это головоломка, которую в 1883 г. придумал французский математик Эдуард Люка. Суть головоломки в следующем. Есть три стержня и восемь дисков разных диаметров. Вначале все диски собраны на одном стержне так, что меньшие диски лежат на больших. Люка предлагал переложить все диски с первого стержня на третий, используя второй. При этом следует соблюдать следующее правило: при перекладывании дисков нельзя класть диск поверх диска меньшего радиуса; за один ход можно переносить лишь один диск.

Если бы в пирамиде был только 1 диск, то решение очевидно. Если дисков 2, то переложим сначала меньший диск на второй стержень, затем перенесем второй диск на третий стержень, а за ним первый диск.

При большем количестве дисков для того чтобы перенести самый большой диск, нужно сначала перенести все диски кроме последнего на второй стержень, потом перенести самый большой на третий, после чего останется перенести все остальные диски со второго на третий. Задачу о переносе N-1 диска мы решаем аналогично, только поменяем стержни местами (N-2 диска перенесем со второго на первый, оставшийся диск перенесем со второго на третий и затем все диски с первого на третий). То есть задачу о N-1 дисках сведем к задаче о N-2 дисках, ту в свою очередь к N-3 дискам, и так вплоть до 1 диска. Этот метод легко программируется с помощью рекурсии.

Задача «Перенести *n* дисков со стержня *from* на стержень *to»*

сводится к трем следующим задачам:

«Перенести *n*-1 диск со стерженя *from* на стержень не-*to*» (рекурсия).

«Перенести *n*-й диск со стерженя *from* на стержень *to*» (тривиальная).

«Перенести *n*-1 диск со стерженя не-*to* на стержень *to*» (рекурсия).

То есть задача переноса пирамиды сводится к двум задачам переноса пирамиды, но в каждой пирамиде на один диск меньше, чем в исходной, и одной тривиальной задаче переноса одного диска.

Каждый диск пронумеруем целым числом, соответствующим размеру диска.

Поскольку диски помещаются и извлекаются со стержней по правилу «первый вошел — последний вышел», для хранения номеров дисков на каждом стержне предлагается использовать стек (как соответствующий логике задачи и наиболее простым по реализации). Стеков должно быть фиксированное количество (3 — по количеству стержней). В стеке будут храниться номера дисков.

Stack<int>[] h;

Для запуска процесса перестановки дисков на форму установим кнопку Button button1.

При нажатии на кнопку button1 создадим стеки, разместим 8 дисков на первом стержне (в 0-м стеке) и запустим процесс их перестановки с первого стержня на третий в количество 8 штук.

private void button1\_Click(object sender, EventArgs e)

{

h = new Stack<int>[3];

h[0] = new Stack<int>();

h[1] = new Stack<int>();

h[2] = new Stack<int>();

h[0].Push(8);

h[0].Push(7);

h[0].Push(6);

h[0].Push(5);

h[0].Push(4);

h[0].Push(3);

h[0].Push(2);

h[0].Push(1);

Hanoi(1, 3, 8);

}

Функция Hanoi имеет следующий заголовок

private void Hanoi(int from, int to, int cou)

и служит для перестановки дисков со стержня номер *from* на стержень номер *to* в количестве *cou* штук. Данная задача является сложной, и поэтому будем ее упрощать путем уменьшения количества переставляемых дисков на один.

Перестановка же одного диска со стержня *from* на стержень *to* является элементарной операцией, которая выполняется командой

*h*[*to* - 1].Push(*h*[*from* - 1].Pop());

Таким образом функция Hanoi будет иметь вид:

private void Hanoi(int from, int to, int c)

{

int to2 = 6 - from - to; //to2 – номер промежуточной башни

if (c > 1)

{

Hanoi(from, to2, c – 1);

Hanoi(from, to, 1);

Hanoi(to2, to, c – 1);

}

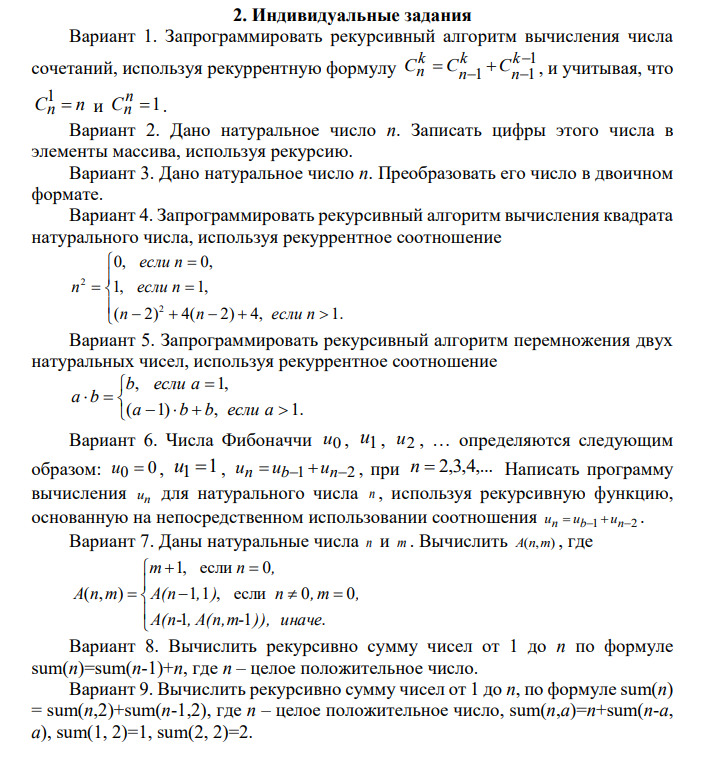
else

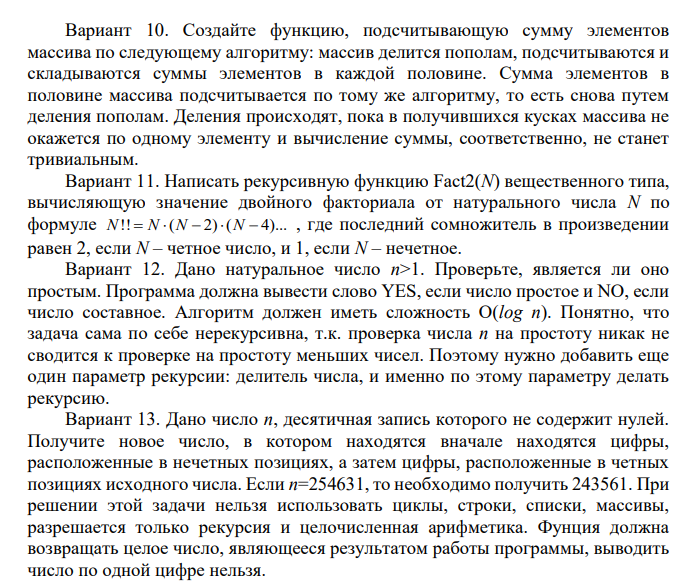
{

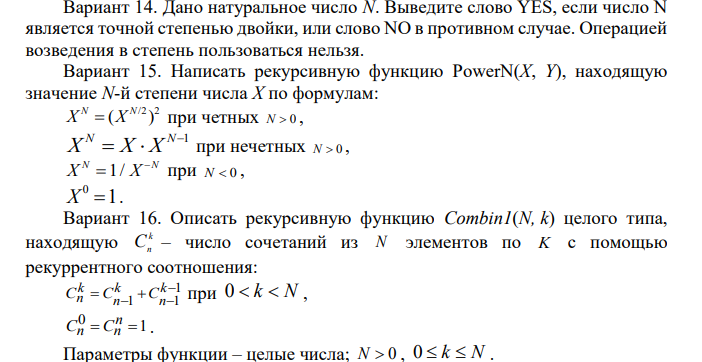
h[to - 1].Push(h[from - 1].Pop());

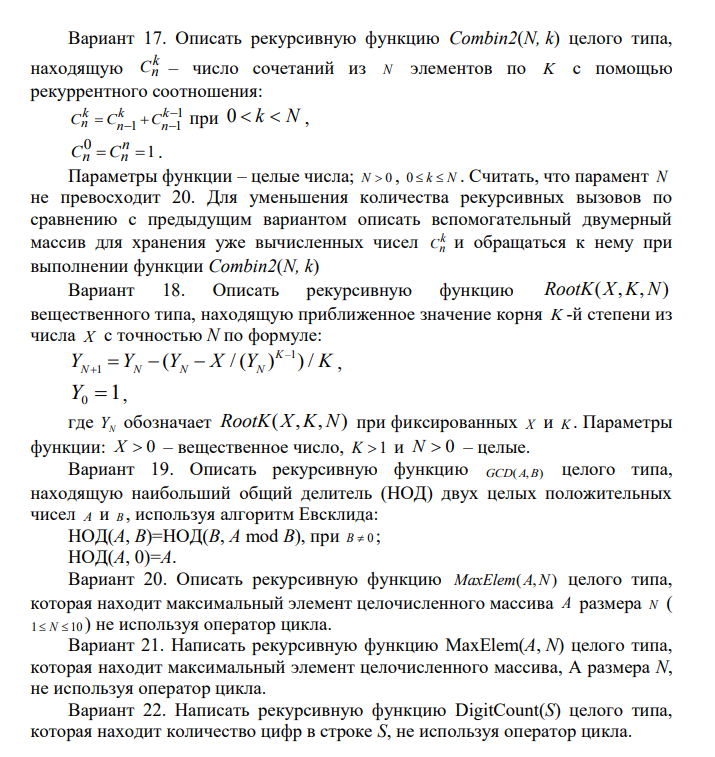
}

}









**Варианты 23- 40.** Выполнить вариант, номер которого определяется по формуле «вариант – 20».

**Пример выполнения работы**

Вычислить  с использованием рекуррентной формулы . При этом известно, что , x - целое, положительное.

Поскольку аргумент *x* функции *F* уменьшается на каждом шаге на 1, то в определенный момент будет вызвана функция *F*(1) для которой известен результат равный 1. С этого значения начинаются шаги рекурсивного выхода.



Таким образом функция *F*(*x*) будет иметь вид:

**int** *F*(**int** *x*)

{

**if**(*x*>1) **return** *F*(*x*-1)+2\**x*+1;

**else** **return** *x*;

}

Вызов функции, размещенный, например, в обработчике события Click для кнопки будет иметь вид:

**int** *x*=**int**.Parse(textBox1.Text);

**int** *y* = *F*(*x*);

label1.Text = *y*.ToString();

Таким образом весь текст консольной программы будет иметь вид

using System;

using System.Collections.Generic;

using System.IO;

using System.Linq;

namespace ConsoleApp3

{

class Program

{

static int F(int x)

{

if (x > 1) return F(x - 1) + 2 \* x + 1;

else return x;

}

static void Main(string[] args)

{

int x = int.Parse(Console.ReadLine());

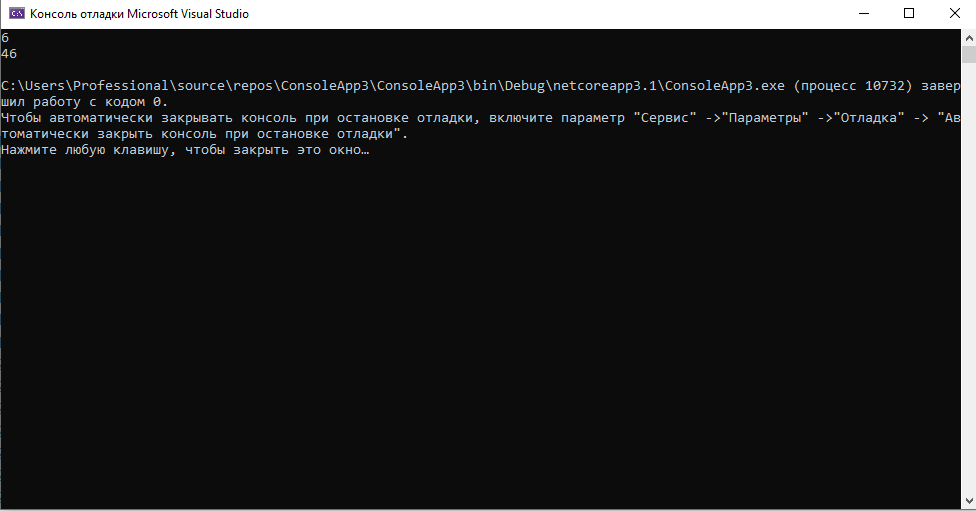
int y = F(x);

Console.WriteLine(y.ToString());

}

}

}



Задание 5.

Изучить предлагаемую тему и написать отчет по изученному материалу.

1. Алгоритмы сортировки
2. Алгоритмы поиска
3. Суффиксный массив
4. Хеш-функции
5. Строковые алгоритмы
6. Оптимизационные алгоритмы
7. Задача о рюкзаке
8. Способы представления графов и алгоритмы на графах
9. Деревья поиска
10. Рекурсивные алгоритмы
11. Бинарные деревья и алгоритмы обхода бинарных деревьев
12. Бинарные деревья и алгоритмы построения бинарных деревьев поиска
13. Хеш-функции
14. Задача нахождения длины наибольшей возрастающей подпоследовательности
15. Циклические линейные списки
16. Односвязные и двусвязные линейные списки
17. Задача коммивояжера
18. Задача о ранце
19. Древовидные структуры данных
20. Динамическое программирование
21. Алгоритмы сортировки
22. Алгоритмы поиска
23. Суффиксный массив
24. Хеш-функции
25. Строковые алгоритмы
26. Оптимизационные алгоритмы
27. Задача о рюкзаке
28. Способы представления графов и алгоритмы на графах
29. Деревья поиска
30. Рекурсивные алгоритмы
31. Бинарные деревья и алгоритмы обхода бинарных деревьев
32. Бинарные деревья и алгоритмы построения бинарных деревьев поиска
33. Хеш-функции
34. Линейные списковые структуры
35. Циклические линейные списки
36. Задачи комбинаторики
37. Задача коммивояжера
38. Задача о ранце
39. Древовидные структуры данных
40. Динамическое программирование

**Список литературы**

1. Ландовский, В. В. Алгоритмы обработки данных : учебное пособие / В. В. Ландовский. — Новосибирск : Новосибирский государственный технический университет, 2018. — 67 c. — ISBN 978-5-7782-3645-5. — Текст : электронный // Электронно-библиотечная система IPR BOOKS : [сайт]. — URL: http://www.iprbookshop.ru/91316.html (дата обращения: 24.05.2020). — Режим доступа: для авторизир. пользователей

2. Курапова, Е. В. Структуры и алгоритмы обработки данных : лабораторный практикум / Е. В. Курапова, Е. П. Мачикина. — Новосибирск : Сибирский государственный университет телекоммуникаций и информатики, 2015. — 23 c. — ISBN 2227-8397. — Текст : электронный // Электронно-библиотечная система IPR BOOKS : [сайт]. — URL: http://www.iprbookshop.ru/55501.html (дата обращения: 24.05.2020). — Режим доступа: для авторизир. пользователей практика

3. Назаренко, П. А. Алгоритмы и структуры данных : учебное пособие / П. А. Назаренко. — Самара : Поволжский государственный университет телекоммуникаций и информатики, 2015. — 130 c. — ISBN 2227-8397. — Текст : электронный // Электронно-библиотечная система IPR BOOKS : [сайт]. — URL: http://www.iprbookshop.ru/71819.html (дата обращения: 24.05.2020). — Режим доступа: для авторизир. пользователей

4. Мейер, Б. Инструменты, алгоритмы и структуры данных / Б. Мейер. — 2-е изд. — Москва : Интернет-Университет Информационных Технологий (ИНТУИТ), 2016. — 542 c. — ISBN 2227-8397. — Текст : электронный // Электронно-библиотечная система IPR BOOKS : [сайт]. — URL: http://www.iprbookshop.ru/73680.html (дата обращения: 24.05.2020). — Режим доступа: для авторизир. пользователей

5. Алексеев, В. Е. Графы и алгоритмы : учебное пособие / В. Е. Алексеев, В. А. Таланов. — 3-е изд. — Москва, Саратов : Интернет-Университет Информационных Технологий (ИНТУИТ), Ай Пи Ар Медиа, 2020. — 153 c. — ISBN 978-5-4497-0366-8. — Текст : электронный // Электронно-библиотечная система IPR BOOKS : [сайт]. — URL: http://www.iprbookshop.ru/89434.html (дата обращения: 24.05.2020). — Режим доступа: для авторизир. пользователей