Министерство науки и высшего образования Российской Федерации

Федеральное государственное бюджетное образовательное   
учреждение высшего образования

«Алтайский государственный технический университет им. И.И.Ползунова»

Заочный институт

[Кафедра ИВТ и ИБ](http://www.altstu.ru/structure/chair/ivtib/)

Отчёт защищен с оценкой \_

Преподаватель \_\_\_\_\_\_\_\_\_\_\_\_ А.В. Бобров

«\_\_\_» \_\_\_\_\_\_\_\_\_\_\_\_ 2021 г.

Комплексный отчет по лабораторным работам

по дисциплине «Структуры данных»

**КО 090301.02.000 О**

обозначение документа

Студент группы 9ИВТ-61 В.В. Ламов

Руководитель работы ст. преподаватель А.В. Бобров

БАРНАУЛ 2021

**Содержание**

[Лабораторная работа №1 Бинарные деревья 3](#_Toc67694268)

[Тесты: 7](#_Toc67694269)

[Лабораторная работа №2 Графы. Формализованные задачи. 9](#_Toc67694270)

[Тесты : 13](#_Toc67694271)

[Лабораторная работа №3 Односвязные списки. 15](#_Toc67694272)

[Тесты: 20](#_Toc67694273)

[Лабораторная работа №4 Двухсвязные списки. 21](#_Toc67694274)

[Тесты: 25](#_Toc67694275)

# Лабораторная работа №1 Бинарные деревья

Цель работы

Приобрести навыки работы с бинарными деревьями

Задание

1. Ознакомиться с постановкой задачи

2. Написать программу для ее реализации

3. Продумать тесты

4. Исходные данные поместить в файл input.dat

5. Результаты вывести на экран. Исходные данные и результаты вывести также в выходной файл output.dat

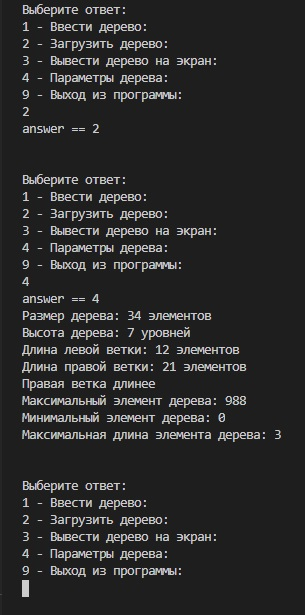
Вариант 2:

Дерево представлено в памяти в виде бинарного дерева. Определить максимальное число потомков одного узла в исходном дереве.

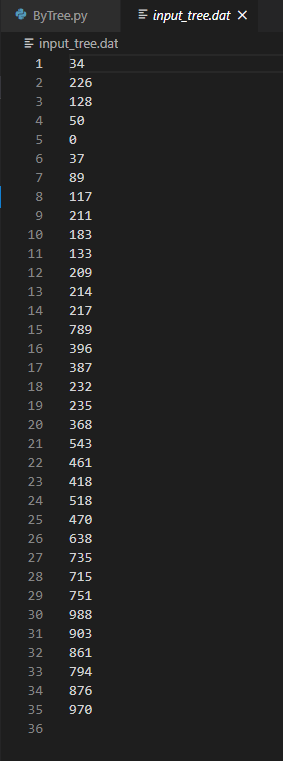
Код программы:

|  |  |
| --- | --- |
| 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  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72  73  74  75  76  77  78  79  80  81  82  83  84  85  86  87  88  89  90  91  92  93  94  95  96  97  98  99  100  101  102  103  104  105  106  107  108  109  110  111  112  113  114  115  116  117  118  119  120  121  122  123  124  125  126  127  128  129  130  131  132  133  134  135  136  137  138  139  140  141  142  143  144  145  146  147  148  149  150  151  152  153  154  155  156  157  158  159  160  161  162  163  164  165  166  167  168  169  170  171  172  173  174  175  176  177  178  179  180  181  182  183  184  185 | **import** **random**  **class** **node**: # узел дерева  **def** **\_\_init\_\_**(self, data = None, left = None, right = None): # инициализатор  self.data = data  self.left = left  self.right = right  **def** **\_\_del\_\_**(self): # деструктор  **print**("удален узел", self.data)  #/\* класс, описывающий само дерево \*/  **class** **Tree**:  **def** **\_\_init\_\_**(self):  self.root = None #корень дерева  self.sizetree = **0**  self.maxnode = **0**  self.minnode = **0**  # /\* функция для добавления узла в дерево \*/  **def** **insert**(self, root, data):  **if**(root **is** None):  self.root = node(data)  self.maxnode = data  self.minnode = data  self.sizetree = **1**  **else**:  **if**(data <= root.data):  **if** root.left **is** None:  root.left = node(data)  self.sizetree += **1**  self.minnode = min(data, self.minnode)  **else**:  self.insert(root.left, data)  **else**:  **if** root.right **is** None:  root.right = node(data)  self.sizetree += **1**  self.maxnode = max(data, self.maxnode)  **else**:  self.insert(root.right, data)  # /\* функция для удаления дерева \*/  **def** **clear**(self, root):  **if** root:  self.clear(root.left)  self.clear(root.right)  self.root = None  self.maxnode = **0**  self.minnode = **0**  self.sizetree = **0**  # /\* функция для вычисления высоты дерева \*/  **def** **height**(self, root, lheight = **0**, rheight = **0**):  **if** root **is** None:  **return** **0**  **else**:  lheight = self.height(root.left)  rheight = self.height(root.right)  **if** lheight > rheight:  **return**(lheight+**1**)  **else**:  **return**(rheight+**1**)  # /\* функция для вычисления разряда ноды \*/  **def** **lenNode**(self):  **if** self.root **is** None:**return** **0**;  rank = len(max(str(self.maxnode),str(self.minnode)))  **return** (rank);  # /\* функция для вычисления высоты дерева \*/  **def** **size**(self, root):  **if** root **is** None:  **return** **0**  **return** (self.size(root.left) + **1** + self.size(root.right));  # /\* функция для записи дерева в файл \*/  **def** **saveTree**(self):  **def** **Preorder**(root, ap\_tt):  **if** root:  ap\_tt.append(root.data)  Node = Preorder(root.left, ap\_tt)  Node = Preorder(root.right, ap\_tt)  **return** ap\_tt;  **if** self.root **is** None:  **return** **0**;  **else**:  ap\_t = list()  ap\_t = Preorder(self.root, ap\_t)  InputFile = open ('input\_tree.dat', 'w')  l\_ap = **0**  InputFile.write(str(l\_ap)+'**\n**')  **for** element **in** ap\_t:  InputFile.write(str(element)+'**\n**')  InputFile.close()  # /\* функция для ввода дерева \*/  **def** **input\_tree**():  wel = "Введите размер дерева: "  wel\_in = input(wel)  l\_ap = int(wel\_in)  **while** l\_ap < **0**:  **print**("размер дерева должен быть целым положительным числом: ")  wel\_in = input(wel)  l\_ap = int(wel\_in)  ap.clear(ap.root)  ap\_list = list()  **for** i **in** range(l\_ap):  ap\_list.append(random.randint(**0**, **1000**))  **print**(ap\_list)  **for** element **in** ap\_list:  ap.insert(ap.root, element)  ap.saveTree()  # /\* функция для загрузки дерева из файла \*/  **def** **load\_tree**():  ap.clear(ap.root)  i = l\_ap = **0**  InputFile = open ('input\_tree.dat')  **for** line **in** InputFile:  **if** i == **0**:  l\_ap = int(line)  **else**:  ap.insert(ap.root, int(line))  i += **1**  InputFile.close()  **def** **AnswerOut**(ap):  **if** ap.root:  app = list()  app.append ("Размер дерева: " + str(ap.sizetree)+" элементов")  app.append ("Высота дерева: " + str(ap.height(ap.root))+" уровней")  lsize = ap.size(ap.root.left)  rsize = ap.size(ap.root.right)  app.append ("Длина левой ветки: " + str(lsize)+" элементов")  app.append ("Длина правой ветки: " + str(rsize)+" элементов")  **if** lsize > rsize:  app.append ("Левая ветка длинее")  **else**:  app.append ("Правая ветка длинее")  app.append ("Максимальный элемент дерева: " + str(ap.maxnode))  app.append ("Минимальный элемент дерева: " + str(ap.minnode))  app.append ("Максимальная длина элемента дерева: " + str(ap.lenNode()))  InputFile = open ('output\_tree.dat', 'w', encoding='utf-8')  **for** str\_app **in** app:  **print** (str\_app)  InputFile.write(str\_app+'**\n**')  InputFile.close()  **else**:  **print** ("2 - Загрузить дерево:")  **def** **menu**():  answer = ""  **while** answer != "9":  **print** ("Выберите ответ:")  **print** ("1 - Ввести дерево:")  **print** ("2 - Загрузить дерево:")  **print** ("3 - Вывести дерево на экран:")  **print** ("4 - Параметры дерева:")  **print** ("9 - Выход из программы:")  answer = input ()  **if** answer == "1":  **print** ("answer == 1")  input\_tree()  **elif** answer == "2":  **print** ("answer == 2")  load\_tree()  **elif** answer == "3":  **print** ("answer == 3")  **if** ap.root:  **print**(str(ap))  # ap.printLevelOrder()  **else**:  **print** ("2 - Загрузить дерево:")  **elif** answer == "4":  **print** ("answer == 4")  AnswerOut(ap)  **elif** answer == "9":  **break**  **print**("**\n**")  ap = Tree()  l\_ap = **0**  menu() |

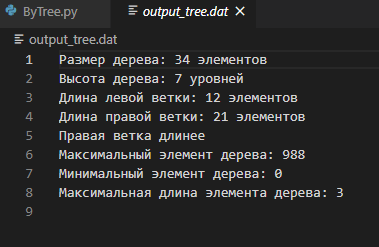
# Тесты:



Содержимое файла input\_tree.dat:



Содержимое файла output\_tree.dat:



# Лабораторная работа №2 Графы. Формализованные задачи.

Цель работы

Приобрести навыки решения задач на графах.

Задание

1. Ознакомиться с постановкой задачи

2.. Выбрать способ представления графа в памяти и написать программу для реализации

задачи.

3. Продумать тесты

4. Исходные данные поместить в файл input.dat

5. Результаты вывести на экран. Исходные данные и результаты вывести также в выходной

файл output.dat

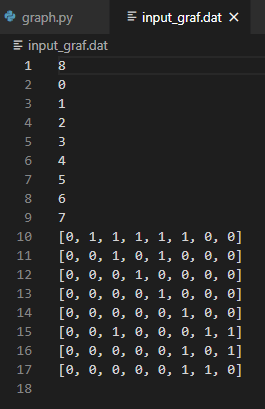
Вариант 2: Написать программу для вычисления числа путей между двумя заданными узлами позаданной матрице смежности.

Код программы:

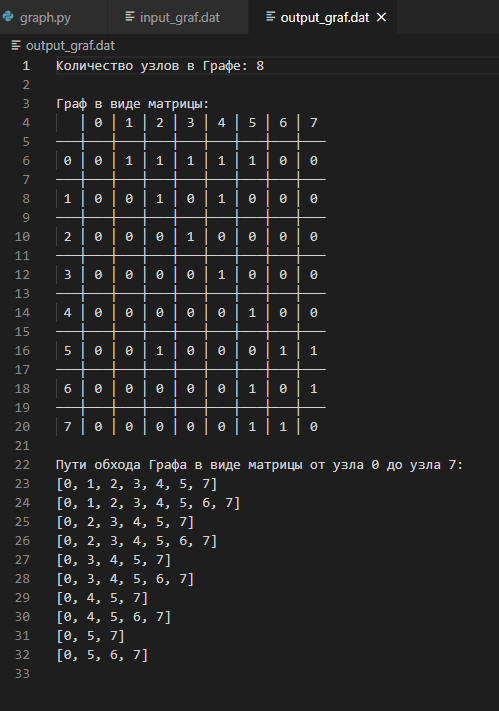
|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72  73  74  75  76  77  78  79  80  81  82  83  84  85  86  87  88  89  90  91  92  93  94  95  96  97  98  99  100  101  102  103  104  105  106  107  108  109  110  111  112  113  114  115  116  117  118  119  120  121  122  123  124  125  126  127  128  129  130  131  132  133  134  135  136  137  138  139  140  141  142  143  144  145  146  147  148  149  150  151  152  153  154  155  156  157  158  159  160  161  162  163  164  165  166  167  168  169  170  171  172  173  174  175  176  177  178  179  180  181  182  183  184  185  186  187  188  189  190  191  192  193  194  195  196  197  198  199  200  201  202  203  204  205  206  207  208  209  210  211  212  213  214  215  216  217  218 | **class** **Vertex**:  **def** **\_\_init\_\_** (self, Name = None, Number = **0**):  self.Name = Name  self.Number = Number  **class** **Edge**:  **def** **\_\_init\_\_** (self, From = None, To = None, Wegth = **0**):  self.From = From  self.To = To  self.Wegth = Wegth  **class** **Graf**:  **def** **\_\_init\_\_** (self):  self.Vertexes = list()  self.Edges = set()    **def** **AddVertex** (self, Name):  self.Vertexes.append(Vertex(Name, len(self.Vertexes)))    **def** **GetVertex** (self, Name):  **return** self.Vertexes[Name]    **def** **AddEdge** (self, From, To, Wegth):  self.Edges.add(Edge(self.GetVertex(From), self.GetVertex(To), Wegth))    **def** **Firstload**(self):  **for** i **in** range(VertexCount):  self.AddVertex(i)  self.AddEdges()  Matrix = self.AddMatrix()  **def** **AddEdges**(self):  grf.AddEdge (**0**, **1**, **1**)  grf.AddEdge (**0**, **2**, **1**)  grf.AddEdge (**0**, **3**, **1**)  grf.AddEdge (**0**, **4**, **1**)  grf.AddEdge (**0**, **5**, **1**)  grf.AddEdge (**1**, **2**, **1**)  grf.AddEdge (**1**, **4**, **1**)  grf.AddEdge (**2**, **3**, **1**)  grf.AddEdge (**3**, **4**, **1**)  grf.AddEdge (**4**, **5**, **1**)  grf.AddEdge (**5**, **2**, **1**)  grf.AddEdge (**5**, **6**, **1**)  grf.AddEdge (**5**, **7**, **1**)  grf.AddEdge (**6**, **5**, **1**)  grf.AddEdge (**6**, **7**, **1**)  grf.AddEdge (**7**, **5**, **1**)  grf.AddEdge (**7**, **6**, **1**)    **def** **Clear**(self):  self.Edges.clear()  self.Vertexes.clear()  **return**;  **def** **AddMatrix**(self):  matrix = [[**0** **for** i **in** range(VertexCount)] **for** j **in** range(VertexCount)]  **for** edge **in** self.Edges:  row = edge.From.Number  column = edge.To.Number  matrix [row][column] = int(edge.Wegth)  InputFile = open ('input\_graf.dat', 'w')  InputFile.write(str(VertexCount)+'**\n**')  **for** vertex **in** self.Vertexes:  InputFile.write(str(vertex.Name)+'**\n**')  **for** row **in** matrix:  InputFile.write(str(row)+'**\n**')  InputFile.close()  **return** matrix;    **def** **SaveOutGraf**(self, matrix, pathes):  InputFile = open ('output\_graf.dat', 'w', encoding='utf-8')  InputFile.write('Количество узлов в Графе: ' + str(VertexCount)+'**\n**')  InputFile.write('**\n**')  InputFile.write('Граф в виде матрицы:'+'**\n**')  **for** line **in** matrix:  InputFile.write(line+'**\n**')  InputFile.write('**\n**')  line = 'Пути обхода Графа в виде матрицы от узла '  line += str(FirstVertex) + ' до узла ' + str(LastVertex) + ':**\n**'  InputFile.write(line)  **for** line **in** pathes:  InputFile.write(str(line)+'**\n**')  InputFile.close()  **def** **LoadMatrixGraf**(self):  InputFile = open ('input\_graf.dat')  i = VertexCount = **0**  matrix = []  self.Clear()  **for** line **in** InputFile:  **if** i == **0**:  VertexCount = int(line)  **elif** i < VertexCount + **1**:  self.AddVertex(int(line))  **else**:  line = line.strip()[**1**:-**1**].split()  row = [int(i.strip(",")) **for** i **in** line]  matrix.append(row)  j = **0**  **for** wigth **in** row:  **if** wigth != **0**:  self.AddEdge(i-VertexCount-**1**, j, wigth)  j += **1**  i += **1**  InputFile.close()  **return** matrix  **def** **OutMatrix**(self, matrix):  Header = " "  Separator = "───"  Names = list()  out = []  **for** vertex **in** self.Vertexes:  Header += f"│ {vertex.Name} "  Separator += "┼───"  Names.append(vertex.Name)  i = **0**  out.append(Header)  **for** row **in** matrix:  out.append(Separator)  String = f" {Names[i]} "  **for** column **in** row:  String += f"│ {column} "  out.append(String)  i += **1**  **return** out  **def** **FindPathes**(self, matrix, vertexFrom, vertexTo):  **def** **Nodes**(first):  ListNodes = []  **for** v **in** range(len(Matrix)):  **if** Matrix[first][v] != **0**:  ListNodes.append(v)  **return** ListNodes;  **def** **FindPath** (v1, v2, path):  endfound = False  path.append(v1)  **if** v1 == **3**:  v = **0**  **while** **not** endfound:  true\_v = Nodes(v1)  **for** v **in** true\_v:  **if** matrix[v1][v2] != **0**:  path.append(v2)  **if** path **not** **in** Pathes:  endfound = True  **break**  **else**:  **del** path[-**1**]  **if** (v **in** path) **or** (v **in** valid):  **continue**;  **else**:  v1 = v  path.append(v1)  **break**  **else**:  path = []  endfound = True  **return** path;    Pathes = []  vertex1 = vertex2 = -**1**  **for** vertex **in** self.Vertexes:  **if** vertex.Name == vertexFrom:  vertex1 = vertex.Number  **elif** vertex.Name == vertexTo:  vertex2 = vertex.Number  **elif** vertex1 > -**1** **and** vertex2 > -**1**:  **break**  path = [vertex1]  valid = []  **for** vertex **in** range(len(matrix)):  **if** (vertex != vertex1) **and** (matrix[vertex1][vertex] != **0**):  found = False  **while** True:  path = [vertex1]  path = FindPath (vertex, vertex2, path)  **if** len(path) == **0**:  **break**  **else**:  Pathes.append(path)  found = True  **if** found:  valid.append(vertex)  **return** Pathes;  # Количество узлов в Графе  VertexCount = **8**  # Количество связей в Графе  EdgesCount = **17**  # Начальный узел обхода Графа  FirstVertex = **0**  # Конечный узел обхода Графа  LastVertex = **7**  # Инициализация Графа  grf = Graf()  # Объявление изходной матрицы  Matrix = []  # Первоначальное заполнение Графа  # и вывод в файл матрицу  grf.Firstload()  # Очистка Графа и заполнение Графа из файла матрицы  Matrix = grf.LoadMatrixGraf()  # Формирование списка матрицы  # для вывода на печать и в файл результата  LineMatrix = grf.OutMatrix(Matrix)  **for** line **in** LineMatrix:  **print**(line)  **print**()  # Формирование путей обхода Графа с начального узла до конечного  # и вывод в список для вывода на печать и в файл результата  Pathes = grf.FindPathes(Matrix, FirstVertex, LastVertex)  **for** line **in** Pathes:  **print**(line)  # Вывод списка матрицы и путей обхода Графа в файл результата  grf.SaveOutGraf(LineMatrix, Pathes) |

# Тесты :

Содержимое файла input\_graf.dat:



Содержимое файла output\_graf.dat:



# Лабораторная работа №3 Односвязные списки.

Цель работы

Приобрести навыки работы с односвязными списками

Задание

1. Ознакомиться с постановкой задачи

2. Написать программу для ее реализации на языке Паскаль

3. Продумать тесты

4. Исходные данные поместить в файл input.dat

5. Результаты вывести на экран. Исходные данные и результаты вывести также в выходной

файл output.dat

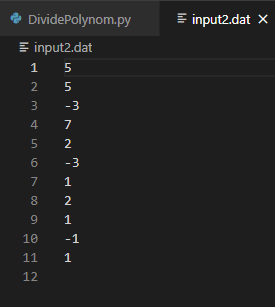
Вариант 2: Написать программу для нахождения остатка от деления двух полиномов.

Код программы:

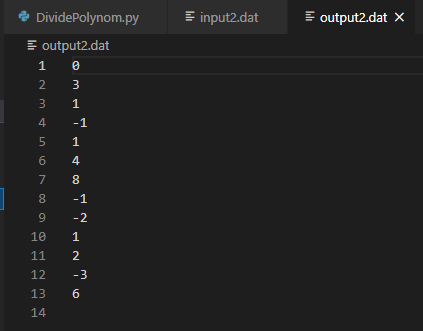
|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72  73  74  75  76  77  78  79  80  81  82  83  84  85  86  87  88  89  90  91  92  93  94  95  96  97  98  99  100  101  102  103  104  105  106  107  108  109  110  111  112  113  114  115  116  117  118  119  120  121  122  123  124  125  126  127  128  129  130  131  132  133  134  135  136  137  138  139  140  141  142  143  144  145  146  147  148  149  150  151  152  153  154  155  156  157  158  159  160  161  162  163  164  165  166  167  168  169  170  171  172  173  174  175  176  177  178  179  180  181  182  183  184  185  186  187  188  189  190  191  192  193  194  195  196  197  198  199  200  201  202  203  204  205  206  207  208  209  210  211  212  213  214  215  216  217  218  219  220  221  222  223  224  225  226  227  228  229  230  231  232  233  234  235  236  237  238  239  240  241  242  243  244  245  246  247  248  249  250  251  252  253  254  255  256  257 | **class** **OneLnkList**:  head = None  **class** **Node**:  element = None  next\_node = None  **def** **\_\_init\_\_**(self, element, next\_node = None):  self.element = element  self.next\_node = next\_node  **def** **append**(self, element):  **if** **not** self.head:  self.head = self.Node(element)  **return** element  node = self.head  **while** node.next\_node:  node = node.next\_node  node.next\_node = self.Node(element)  **def** **edit**(self, element, index):  node = self.head  **for** i **in** range(index):  node = node.next\_node  node.element = element  **def** **insert**(self, index):  node = self.head  prev\_node = self.head  **for** i **in** range(index):  prev\_node = node  node = node.next\_node  prev\_node.next\_node = self.Node(element, next\_node=node)  **def** **length**(self):  **if** **not** self.head:  **return** **0**  i = **1**  node = self.head  **while** node.next\_node:  i += **1**  node = node.next\_node  **return** i  **def** **get**(self, index):  **if** **not** self.head:  **return** ""  node = self.head  **for** i **in** range(index):  node = node.next\_node  **return** node.element  **def** **reverse**(self):  **if** **not** self.head:  **return** ""  index = self.length()  temp\_self = OneLnkList()  **for** i **in** range(index):  temp\_self.append(self.get(index-**1**-i))  self.clear()  **for** i **in** range(index):  self.append(temp\_self.get(i))  **def** **out**(self):  **print**('**\n**')  node = self.head  **if** **not** node:  **return** ""  **while** True:  **print**(node.element)  node = node.next\_node  **if** **not** node:  **break**  **def** **delete**(self, index):  node = self.head  prev\_node = node  **if** index == **0**:  self.head = self.head.next\_node  **for** i **in** range(index):  prev\_node = node  node = node.next\_node  prev\_node.next\_node = node.next\_node  element = node.element  **del** node  **def** **clear**(self):  **if** **not** self.head:  **return** ""  node = self.head  index = self.length()  **for** i **in** range(index)[::-**1**]:  self.delete(i)  **def** **input\_polinom**():  ap.clear()  aq.clear()  wel = "Введите степень первого полинома: "  wel\_in = input(wel)  p = int(wel\_in)  **while** p < **0**:  **print**("степень первого полинома должна быть целым положительным числом: ")  wel\_in = input(wel)  p = int(wel\_in)  wel = "Введите степень второго полинома: "  q = int(input(wel))  **while** q > p:  **print**("степень второго полинома должна быть целым положительным числом больше либо равно степени первого полинома: ")  q = int(input(wel))  i = **0**  **while** i < p+**1**:  wel = 'Введите a(',i,') для первого полинома: '  ap.append(int(input(wel)))  i+=**1**  i = **0**  **while** i < q+**1**:  wel = "Введите a(",i,") для второго полинома: "  aq.append(int(input(wel)))  i+=**1**  InputFile = open ('input2.dat', 'w')  InputFile.write(str(p)+'**\n**')  **for** i **in** range(p):  InputFile.write(str(ap.get(i))+'**\n**')  InputFile.write(str(q)+'**\n**')  **for** i **in** range(q):  InputFile.write(str(aq.get(i))+'**\n**')  InputFile.close()  **def** **load\_polinom**():  ap.clear()  aq.clear()  i = p = q =**0**  InputFile = open ('input2.dat')  **for** line **in** InputFile:  **if** i == **0**:  p = int(line)  **elif** i == p+**2**:  q = int(line)  **elif** q == **0**:  ap.append(int(line))  **else**:  aq.append(int(line))  i += **1**  wel = line  InputFile.close()  **def** **Str\_Polinom**(l\_a):  **def** **str\_x**(a, i):  str\_a = ""  str\_x = ""  **if** i < ap\_c:  **if** a >= **0**:  str\_a = "+"  **if** i > **0**:  **if** a == **0**:  str\_a = ""  str\_x = ""  **else**:  **if** a == **1**:  str\_a = str\_a  **elif** a == -**1**:  str\_a = "-"  **else**:  str\_a = str\_a+str(a)  **if** i == **1**:  str\_x = "x"  **else**:  str\_x = "x^" + str(i)  **else**:  str\_a = str\_a+str(a)  **return** (str\_a+str\_x)  wel = ""  ap\_c = l\_a.length()-**1**  **for** i **in** range(ap\_c+**1**):  wel = str\_x(l\_a.get(i), i) + wel  **return** (wel)  **def** **Div** ():  t = ap.length() # степень остатка  q = aq.length() # степень делителя  r = t-q+**1** # степень частного  i = **0**  **if** apt.length() > **0**:  apt.clear()  **if** at.length() > **0**:  at.clear()  **for** j **in** range (**0**, t):  apt.append(int(ap.get(j)))  apt.reverse()  **while** i < r:  at.append(int(apt.get(i)//aq.get(**0**))) #коэффициент частного  **for** j **in** range (**0**, q):  **if** aq.get(j) != **0**:  a = apt.get(i+j)-(aq.get(j)\*at.get(i))  apt.edit(a, i+j)  i += **1**  apt\_t = OneLnkList()  **for** i **in** range (**0**, t):  **if** apt.get(i) != **0**:  apt\_t.append(int(apt.get(i)))  at.reverse()  apt\_t.reverse()  InputFile = open ('output2.dat', 'w')  InputFile.write(str(p)+'**\n**')  **for** i **in** range(p):  InputFile.write(str(ap.get(i))+'**\n**')  InputFile.write(str(q)+'**\n**')  **for** i **in** range(q):  InputFile.write(str(aq.get(i))+'**\n**')  t = at.length()  InputFile.write(str(t)+'**\n**')  **for** i **in** range(t):  InputFile.write(str(at.get(i))+'**\n**')  t\_t = apt\_t.length()  InputFile.write(str(t\_t)+'**\n**')  **for** i **in** range(t\_t):  InputFile.write(str(apt\_t.get(i))+'**\n**')  InputFile.close()  **return** at, apt\_t  **def** **menu**():  answer = ""  **while** answer != "9":  **print** ("Выберите ответ:")  **print** ("1 - Ввести полиномы:")  **print** ("2 - Загрузить полиномы:")  **print** ("3 - Вывести полиномы на экран:")  **print** ("4 - Деление полиномов:")  **print** ("9 - Выход из программы:")  answer = input ()  **if** answer == "1":  **print** ("answer == 1")  input\_polinom()  **elif** answer == "2":  **print** ("answer == 2")  load\_polinom()  **elif** answer == "3":  **print** ("answer == 3")  **if** ap.length() != **0**:  **print**("Делимое: "+Str\_Polinom(ap))  **print**("Делитель: "+Str\_Polinom(aq))  **else**:  **print** ("2 - ЗАГРУЗИТЬ полиномы:")  **elif** answer == "4":  **print** ("answer == 4")  **if** (ap.length() != **0**) **and** (aq.length() != **0**):  at, apt = Div()  **print**("Делимое: "+Str\_Polinom(ap))  **print**("Делитель: "+Str\_Polinom(aq))  **print**("Частное: "+Str\_Polinom(at))  **print**("Остаток: "+Str\_Polinom(apt))  **else**:  **print** ("2 - ЗАГРУЗИТЬ полиномы:")  **elif** answer == "9":  **break**  **print**("**\n**")  ap = OneLnkList() # Делимое  aq = OneLnkList() # Делитель  at = OneLnkList() # частное  apt = OneLnkList() # Остаток  p = **0**  q = **0**  menu() |

# Тесты:

Содержимое файла input2.dat



Содержимое файла output2.dat



# Лабораторная работа №4 Двухсвязные списки.

Цель работы

Приобрести навыки работы с двухсвязными списками

Задание

1. Ознакомиться с постановкой задачи

2. Написать программу для ее реализации на языке СИ

3. Продумать тесты

4. Исходные данные поместить в файл input.dat

5. Результаты вывести на экран. Исходные данные и результаты вывести также в выходной

файл output.dat

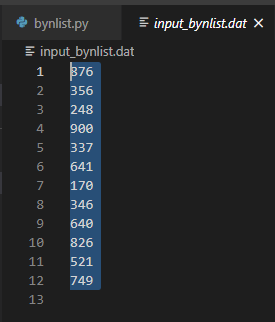
Вариант 2: Связать в новый список все четные числа из списка положительных чисел.

Код программы:

|  |  |
| --- | --- |
| 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  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72  73  74  75  76  77  78  79  80  81  82  83  84  85  86  87  88  89  90  91  92  93  94  95  96  97  98  99  100  101  102  103  104  105  106  107  108  109  110  111  112  113  114  115  116  117  118  119  120  121  122  123  124  125  126  127  128  129  130  131  132  133  134  135  136  137  138  139  140  141  142  143  144  145  146  147  148  149  150  151  152  153  154  155  156  157  158  159  160  161  162  163  164  165  166  167  168  169  170  171  172  173  174  175  176  177  178  179  180  181  182  183  184  185  186  187  188  189 | **import** **random**  **class** **Node**:  **def** **\_\_init\_\_**(self, Name = None, Left = None, Rigth = None):  self.Name = Name  self.Left = Left  self.Rigth = Rigth  **def** **\_\_del\_\_**(self): # деструктор  **print**("Удален узел", self.Name)  **class** **BynlnkList**:  **def** **\_\_init\_\_**(self):  self.root = None  self.end = None    **def** **Exist**(self, Name):  found = False  start = self.root  end = self.end  **while** **not** found:  **if** start.Name == Name:  found = True  **elif** end.Name == Name:  found = True  **else**:  start = start.Rigth  end = end.Left  **return** found;    **def** **AddNode**(self, Name):  currnode = None  **if** **not** self.root:  self.root = Node(Name)  self.end = self.root  currnode = self.end  # elif self.Exist(Name):  # currnode = self.GetNode(Name)  **elif** self.root == self.end:  self.end = Node(Name)  self.end.Left = self.root  self.root.Rigth = self.end  currnode = self.end  **else**:  currnode = self.end  self.end = Node(Name)  currnode.Rigth = self.end  self.end.Left = currnode  currnode = self.end  **return** currnode;    **def** **GetNode**(self, Name):  currnode = None  start = self.root  end = self.end  found = False  **while** **not** found:  **if** start.Name == Name:  currnode = start  found = True  **elif** end.Name == Name:  currnode = end  found = True  **else**:  start = start.Rigth  end = end.Rigth  **return** currnode;    **def** **Clear**(self):  **if** **not** self.end:  **return**;  **if** self.root == self.end:  **del** root  **else**:  **while** self.end:  currnode = self.end  self.end = self.end.Left  **if** self.end:  self.end.Rigth = None  **del** currnode  currnode = self.root  self.root = None  **del** currnode  **return**  **def** **Filling**(self):  al = list()  **for** i **in** range(Size\_bl):  num = random.randint(**0**, Max\_bl)  **if** num **not** **in** al:  al.append(num)  InputFile = open ('input\_bynlist.dat', 'w')  **for** i **in** al:  self.AddNode(i)  InputFile.write(str(i)+'**\n**')  InputFile.close()    **def** **loading**(self):  self.Clear()  InputFile = open ('input\_bynlist.dat')  **for** line **in** InputFile:  self.AddNode(int(line))  InputFile.close()    **def** **listing**(self):  bl\_list = list()  currnode = self.root  **while** currnode:  bl\_list.append(currnode.Name)  currnode = currnode.Rigth  **return** bl\_list  **def** **FillingEven**(bl1):  **def** **Add**(lst, Name):  **if** Name%**2** == **0**:  bl2.AddNode(Name)  bl2 = BynlnkList()  **if** **not** bl1.root:  **print**("Первый список пустой")  **return**  **if** bl2.root:  bl2.Clear()  currnode = bl1.root  **while** True:  **if** currnode:  Add(bl2, currnode.Name)  currnode = currnode.Rigth  **else**:  **break**  **return** bl2;  **def** **NicePrint**(blist):  lst = list()  Header = " № ┼ Лево │ Имя │ Право"  Separator = "──────┼──────┼──────┼──────"  lst.append(Header)  currnode = blist.root  nom = **1**  **while** currnode:  line = ""  **if** **not** currnode.Left:  left = ""  **else**:  left = currnode.Left.Name  name = currnode.Name  **if** **not** currnode.Rigth:  rigth = ""  **else**:  rigth = currnode.Rigth.Name  n = '{:^6}'.format(nom) + "┼"  leftstr = '{:^6}'.format(left) + "┼"  namestr = '{:^6}'.format(name) + "┼"  rigthstr = '{:^6}'.format(rigth)  line = n+leftstr+namestr+rigthstr  lst.append(Separator)  lst.append(line)  currnode = currnode.Rigth  nom += **1**  **return** lst;  **def** **outfile**():  InputFile = open ('output\_bynlist.dat', 'w', encoding='utf-8')  **if** bl.root != None:  InputFile.write('Первый двусвязный список:'+'**\n**')  **for** line **in** bl\_list:  InputFile.write(line+'**\n**')  InputFile.write('**\n**')  **if** bl\_new.root != None:  InputFile.write('Второй двусвязный список:'+'**\n**')  **for** line **in** bl\_new\_list:  InputFile.write(line+'**\n**')  InputFile.close()  Size\_bl = **12**  Max\_bl = **1000**  bl = BynlnkList()  bl.Filling()  bl.loading()  bl\_new = FillingEven(bl)  bl\_list = NicePrint(bl)  bl\_new\_list = NicePrint(bl\_new)  **for** i **in** bl\_list:  **print**(i)  **print**("")  **for** i **in** bl\_new\_list:  **print**(i)  outfile() |

# Тесты:

Содержимое файла input\_bynlist.dat:



Содержимое файла output\_bynlist.dat:

