**МИНОБРНАУКИ РОССИИ**

**Санкт-Петербургский государственный**

**электротехнический университет**

**«ЛЭТИ» им. В.И. Ульянова (Ленина)**

**Кафедра МО ЭВМ**

отчет

**по лабораторной работе №1**

**по дисциплине «Построение и анализ алгоритмов»**

Тема: Поиск с возвратом

|  |  |  |
| --- | --- | --- |
| Студент гр. 3342 |  | Иванов Д. М. |
| Преподаватель |  | Виноградова Е. В. |

Санкт-Петербург

2025

## Задание

У Вовы много квадратных обрезков доски. Их стороны (размер) изменяются от 1 до N−1, и у него есть неограниченное число обрезков любого размера. Но ему очень хочется получить большую столешницу - квадрат размера N. Он может получить ее, собрав из уже имеющихся обрезков(квадратов).

Например, столешница размера 7×7 может быть построена из 9 обрезков.



Рисунок 1 – пример расположения квадратов для тестового сучая

Внутри столешницы не должно быть пустот, обрезки не должны выходить за пределы столешницы и не должны перекрываться. Кроме того, Вова хочет использовать минимально возможное число обрезков.

Входные данные:

Размер столешницы - одно целое число N (2≤N≤20).

Выходные данные:

Одно число K, задающее минимальное количество обрезков(квадратов), из которых можно построить столешницу(квадрат) заданного размера N. Далее должны идти K строк, каждая из которых должна содержать три целых числа x,y и w, задающие координаты левого верхнего угла (1≤x,y≤N) и длину стороны соответствующего обрезка(квадрата).

Пример входных данных

7

Соответствующие выходные данные

9

1 1 2

1 3 2

3 1 1

4 1 1

3 2 2

5 1 3

4 4 4

1 5 3

3 4 1

Индивидуальный вариант: 2и. Итеративный бэктрекинг. Исследование времени выполнения от размера квадрата.

**Описание алгоритма**

Основная идея алгоритма заключается в использовании итеративного бэктрекинга (поиска с возвратом), который позволяет перебирать все возможные варианты заполнения столешницы квадратами, начиная с самых больших и постепенно уменьшая их размер.

В данном переборе используется стек для хранения состояний и их обхода. Каждое состояние представляет из себя вариант заполнения столешницы в виде матрицы клеток. Алгоритм извлекает состояние из стека, обрабатывает его и добавляет в стек новые состояния, которые могут быть получены из текущего. Он находит некоторую пустую клетку и добавляет в нее все возможные квадраты разных размеров. Как только столешница становится заполненной, программа удаляет из стека данное состояние, тем самым делая возврат, и идет рассматривать следующие. Такой обход идет до того момента, пока стек не опустеет. По итогу, программа возвращает решение, которое содержало меньше всего квадратов (во время обхода это хранится в отдельной переменной).

Однако данный полный перебор будет неэффективным. Есть возможность оптимизировать программу, чтобы она не рассматривала некоторые заведомо невыгодные состояния. Так что были добавлены следующие оптимизации алгоритма:

1) Если текущее состояние во время своего заполнения и обработки имеет больше квадратов чем лучшее решение, оно отбрасывается, и алгоритм переходит к следующему состоянию.

2) Очевидно, что для использования минимального количества квадратов нужно брать квадраты большого размера. Так что изначально можно поставить в левом верхнем углу квадрат размера от n/2 до n-1, который будет являться самым большим для данной столешницы. Также справа и снизу от него будут некоторые свободные клетки. На них также выгоднее всего расположить квадраты максимально возможного размера. В итоге получим уже заполненную верхнюю левую часть столешницы. Для остальных клеток будет использован описанный выше перебор.

3) Для нахождения следующей пустой клекти используется не построчный обход, одновременно и по строкам, и по столбцам: симметрично относительно диагонали. Таким образом, мы быстрее находим пространство для вставки квадрата большего размера.

Сложность по времени: O(n^(n^2 + 2))

Создание сетки столешницы – O(n^2).

Основоной цикл – O(n^(n^2)). В худшем случае каждое состояние может порождать O(n) новых состояний (для каждого возможного размера квадрата). Максимальное количество квадратов (количество шагов) может достигать O(n^2) (если каждый квадрат имеет размер 1x1 и нужно заполнить n^2 клеток).

Вспомогательные функции (нахождение новой клетки, размещение квадрата, проверка возможности размещения квадрата) – O(n^2) каждая.

Общая сложность: для каждого шага основого цикла применяются вспомогательные функции, поэтому получаем: O(n^(n^2) \* n^2) = O(n^(n^2 + 2)).

Сложность получилась очень большая. Однако мы добавили некоторые оптимизации, которые позволяют избежать эти худшие случаи и иметь заполненную столешницу уже на первых шагах алгоритма.

Сложность по памяти:

Матрица n\*n занимает O(n^2) места.

## Выполнение работы

1) Создание класса Node

Прежде всего необходимо создать класс, из объектов которого будет состоять наш список. Класс Node будет иметь следующие поля:

self.arr – массив чисел

self.next – ссылка на следующий объекта класса Node. Изначально наш элемент никуда не ссылается, поэтому присваиваем None.

Методы:

\_\_init\_\_(self, array=[]) – конструктор класса. Необязательным аргументов является передача массива чисел, который присваивается полю self.arr.

2) Создание класса UnrolledLinkedList

Поля:

self.head – первый элемент списка, который относится к классу Node

self.length – количество связанных между собой массивов

self.len\_of\_node\_array – фиксированный размер одного узла. Он расчитывается в функции calculate\_optimal\_node\_size(num\_elements) по специальной формуле(файл Calculate\_size.py, в котором находится формула, приводится в приложении). Внутри списка массивы не могут иметь длину больше этого значения. В противном случае придется в дальнейшем данные разбивать на несколько узлов.

Методы:

\_\_init\_\_(self, arr=[]) – конструктор класса. Устанавливает для полей нужные значения: self.head = None, self.lenght = 0. Запускает метод make\_linked\_list(описан ниже).

make\_linked\_list(self, arr) – метод, который по переданному при инициализации массиву формирует развернутый связаный список с фиксированной длиной. Через цикл идет заполнение структуры и балансировка узлов при их заполнении. До тех пор пока не будут взяты все числа массива.

push(self, element) – вставка переданного числа в конец списка. Через поле next переходим к последнему узлу списка, который ссылается на None. И в сам массив добавляется число. Если происходит перебор чисел, то массив разбивается на 2 части, и вторая часть переходит в новый созданный узел списка.

insert(self, element, index) – вставка переданного числа по индексу. В первую очередь через этот индекс находится индекс самого узла, внутрь которого нужно вставить число. После вставик в массив, по аналогии с методом push, присходит проверка на размер получившегося массива. И в случае необходимости разбиение узла на две части и возможное соединение со следующим элементов списка.

find\_node\_by\_index\_of\_element(self, index) – метод для нахождения узла списка через переданный индекс относительно всех чисел списка и его индекса относительно остальных узлов. Создается счеткчик, в который добавляется длина массива каждого нового узла, параллельно через поле next происходит переход к следующему узлу. Процесс идет до момента, пока счетчик меньше переданного в функцию аргумента.

delete\_number(self, index) – удаление числа из списка по переденному индексу. В начале при помощи функции find\_node\_by\_index\_of\_element, описанной выше, находится сам узел, его индекс и индекс первого элемента относительно всего списка. Через эти данные через метод del удаляется число из массива. Идет проверка: если массив получился пустым, весь узел удалается через метод delete\_arr(будет описана дальше), и если возможно данный массив связать с массивом следующего узла, то происходит объединение.

delete\_arr(self, index) – метод для удаление узла через переданный индекс узла. С помощью цикла программа доходит до нужного узла. Затем у предыдущего узла меняется поле next, после чего он ссылается на другой элемент списка.

find\_index\_of\_number(self, number) – нахождение индкса переданного числа в списке. Идет обход, начиная с head, всех узлов списка, пока в данном массиве не будет найдено нужное нам число. После нахождения суммируем все найденные ранее длины массивов и возвращаем полученный индекс.

print\_list(self) – вывод списка. В каждой строке программа выводит индекс узла и его массив чисел.

3) Проверка работоспособности программы

В первую очередь напишем файл с тестами для программы test.py(код приводится в приложении). В этом файле будут проверяться такие методы, как поиск индекса числа, вставка и удаление в разных частях списка(начало, середина, конец). Для вставки и удаления будет проверяться, что при верно введенных индексах метод выполняет работу и возвращает True. Однако при неверных индексах не происходит аварийное завершение программы, а метод просто возвращает False.

Дальше пойдет проверка правильности выполнения тех же методов в файле main.py(код приводится в приложении). Функция check принимает два массива чисел: arr1 – для заполнения, arr2 – для поиска и удаления. 50 чисел добавляются постепенно через push. Еще 3 элемента подаются в разные части списка через insert. После этого находится индекс каждого элемента из arr2 и удаляется

## Анализ полученных значений

В конце необходимо оценить эффективность данной структуры, сравнив её с односвязанным списком и массивом (list() в Python). Для замерки времени работы использовалась библиотека time. Оценивались такие методы, как нахождение индексов элементов, вставка и удаление чисел в разных частях структур. Проверились операции на маленьких (1000), средних (10000) и больших (100000) наборах данных. Результаты представлены в виде графиков (см. ри. 1).

По результатм ее исследования и сравнения с другими структурами данных (см. рис. 1) можно сделать следующие выводы.

На малых объемах данных разница в большинстве случаев незаметна. Однако с увеличением размеров структур различие по времени выполнения операций становится более существенным.

Вставка и удаление элементов в середину и конец выполняется намного быстрее, чем для связанного списка, и практически одинаково с массивом. Такая же ситуация с нахождением элементов в тех же местах данных. При вставке и удалении в начало структуры развернутый список работает медленнее, чем односвязанный, но быстрее, чем массив. Единственное место, где проигрывает наша структура – это нахождение элемента в начале.

В плане потребления памяти развернутый связный список и массив потребляют меньше ресурсов по сравнению с односвязным списком, так как односвязный список требует дополнительной памяти для хранения указателей на следующие элементы. В нашей структуре данных это используется более эффективно.

Таким образом, развернутый связанных список показывает хорошую эффективность по времени выполнения операция и использования памяти по сравнению с другими структурами данных. Это делает его более привлекательным при работе большим объемом данных.

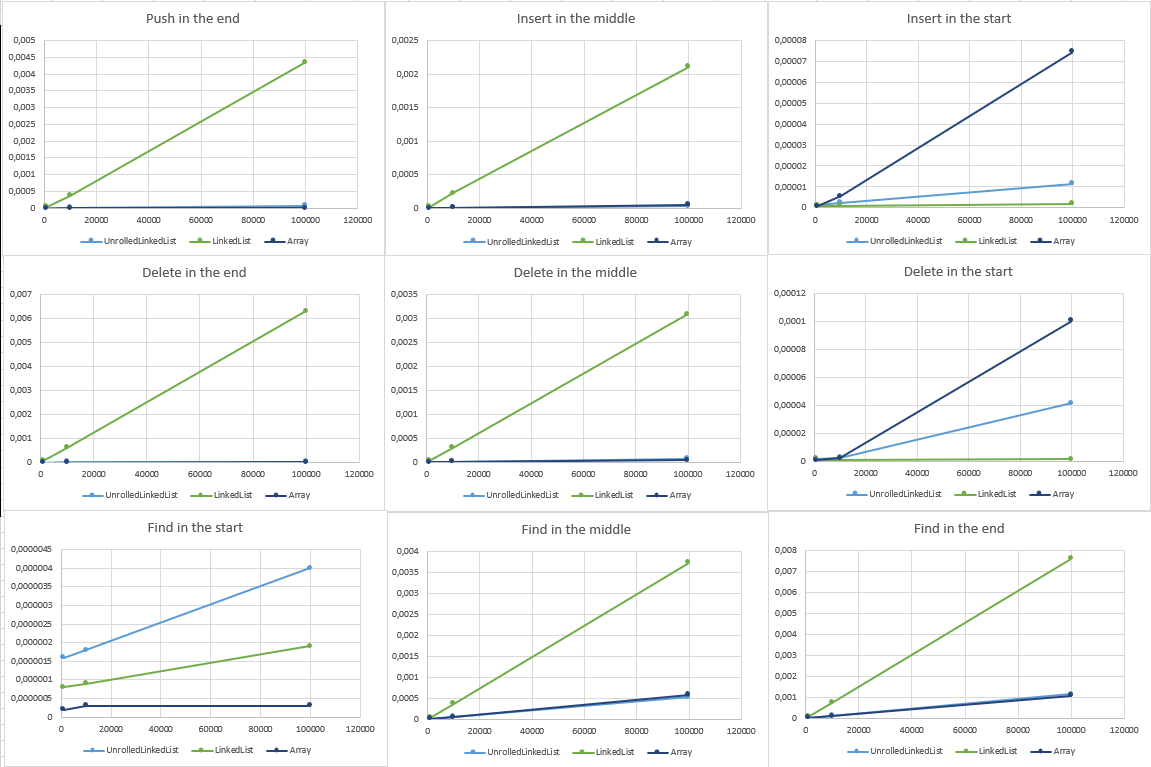


Рисунок 1 – графики, показывающие время выполнения операций для различных структур

## Выводы

Была реализована такая структура данных, как развернутый связанный список. На языке Python написаны основные методы для нее. Реализация включала такие основные операции, как добавление, удаление и поиск. Также структура была проверена на работоспособность и эффективность по времени относительно массива и односвязанного списка.

# Приложение А Исходный код программы

Название файла: UnrolledLinkedList.py

"""Module include structure unrolled linked list"""

import dataclasses

from Calculate\_size import calculate\_optimal\_node\_size

@dataclasses.dataclass

class Node:

"""Class of element of list"""

def \_\_init\_\_(self, array=None):

if array is None:

array = []

self.arr = array.copy()

self.next = None

class UnrolledLinkedList:

"""Class of structure unrolled linked list"""

def \_\_init\_\_(self, arr=None):

if arr is None:

arr = []

self.head = None

self.length = 0

self.len\_of\_node\_array = calculate\_optimal\_node\_size(len(arr))

self.make\_linked\_list(arr)

def make\_linked\_list(self, arr):

"""Function of make linked list with initialization"""

if len(arr) == 0:

return

self.length = 0

k = 0

lis = []

index = 0

flag\_head = False

el = self.head

while index < len(arr):

if k < self.len\_of\_node\_array:

lis.append(arr[index])

k += 1

index += 1

if k >= self.len\_of\_node\_array:

if not flag\_head:

self.head = Node(lis)

self.length += 1

flag\_head = True

el = self.head

else:

el.next = Node(lis)

self.length += 1

el = el.next

lis = []

k = len(lis)

if len(lis) != 0:

if not flag\_head:

if len(lis) > self.len\_of\_node\_array:

self.head = Node(lis[:self.len\_of\_node\_array // 2])

self.head.next = Node(lis[self.len\_of\_node\_array // 2:])

return

self.head = Node(lis)

return

if el is None and len(el.arr) + len(lis) <= self.len\_of\_node\_array:

el.arr += lis

else:

el.next = Node(lis)

def push(self, element):

"""Function of push new element to the end"""

if self.head is None:

self.head = Node([element])

return True

el = self.head

while el.next is not None:

el = el.next

if len(el.arr) + 1 > self.len\_of\_node\_array:

massive = el.arr

el.arr = massive[:self.len\_of\_node\_array // 2]

el.next = Node(massive[self.len\_of\_node\_array // 2:] + [element])

self.length += 1

return True

el.arr.append(element)

return True

def insert(self, element, index):

"""Function of insert element with some index"""

if index < 0:

return False

el = self.head

start\_index = len(el.arr)

while start\_index <= index:

el = el.next

if el is None:

return False

start\_index += len(el.arr)

if el is None:

return False

el.arr.insert(index - start\_index, element)

if len(el.arr) <= self.len\_of\_node\_array:

return True

half\_length = self.len\_of\_node\_array // 2

new\_array = el.arr[half\_length:]

el.arr = el.arr[:half\_length]

tmp = el.next

el.next = Node(new\_array)

el.next.next = tmp

self.length += 1

return True

def delete\_number(self, index):

"""Function of delete element with some index"""

el, start\_index, index\_node = self.find\_node\_by\_index\_of\_element(index)

if el is None:

return False

index\_node\_next = index\_node + 1

del el.arr[index - start\_index]

if len(el.arr) == 0:

self.delete\_arr(index\_node)

return True

if el.next is not None:

if len(el.next.arr) + len(el.arr) <= self.len\_of\_node\_array:

el.arr += el.next.arr

self.delete\_arr(index\_node\_next)

return True

def find\_node\_by\_index\_of\_element(self, index):

"""Function of find node with some index"""

index\_node = 0

if index < 0:

return None, -1, -1

el = self.head

k = len(el.arr)

while k <= index:

el = el.next

index\_node += 1

if el is None:

return None, -1, -1

k += len(el.arr)

return (el, k - len(el.arr), index\_node)

def delete\_arr(self, index):

"""Function of delete some array from the list"""

if index == 0:

self.head = self.head.next

self.length -= 1

return

el = self.head

k = 0

while k < index - 1 and el.next is not None:

el = el.next

k += 1

el.next = el.next.next

self.length -= 1

def find\_index\_of\_number(self, number):

"""Function of find number index in all list"""

k = 0

index = 0

el = self.head

while el is not None:

if number in el.arr:

return index + el.arr.index(number)

index += len(el.arr)

el = el.next

k += 1

return None

def print\_list(self):

"""Function printing list"""

k = 0

el = self.head

while el is not None:

string = ' '.join([str(x) for x in el.arr])

print(f"Node {k}: {string}")

el = el.next

k += 1

Название файла: test.py

"""Module include test of unrolled linked list"""

from UnrolledLinkedList import UnrolledLinkedList

def make\_list():

"""function of make list object"""

lis = list(range(1, 40))

inrolled\_lis = UnrolledLinkedList(lis)

return inrolled\_lis

def test\_find\_start():

"""test\_1"""

lis = make\_list()

assert lis.find\_index\_of\_number(1) == 0

def test\_find\_middle():

"""test\_2"""

lis = make\_list()

assert lis.find\_index\_of\_number(14) == 13

def test\_find\_end():

"""test\_3"""

lis = make\_list()

assert lis.find\_index\_of\_number(39) == 38

def test\_delete\_number\_with\_right\_index\_in\_the\_end():

"""test\_4"""

lis = make\_list()

assert lis.delete\_number(38) is True

def test\_delete\_number\_with\_right\_index\_in\_the\_middle():

"""test\_5"""

lis = make\_list()

assert lis.delete\_number(14) is True

def test\_delete\_number\_with\_right\_index\_in\_the\_start():

"""test\_6"""

lis = make\_list()

assert lis.delete\_number(0) is True

def test\_delete\_number\_with\_wrong\_high\_index():

"""test\_7"""

lis = make\_list()

assert lis.delete\_number(55) is False

def test\_delete\_number\_with\_wrong\_low\_index():

"""test\_8"""

lis = make\_list()

assert lis.delete\_number(-2) is False

def test\_add\_numbers\_with\_right\_index\_int\_the\_middle():

"""test\_9"""

lis = make\_list()

assert lis.insert(99, 3) is True

def test\_add\_numbers\_with\_right\_index\_int\_the\_start():

"""test\_10"""

lis = make\_list()

assert lis.insert(66, 0) is True

def test\_add\_numbers\_with\_right\_index\_int\_the\_end():

"""test\_11"""

lis = make\_list()

assert lis.push(77) is True

def test\_add\_numbers\_with\_wrong\_low\_index():

"""test\_12"""

lis = make\_list()

assert lis.insert(123, -3) is False

def test\_add\_numbers\_with\_wrong\_high\_index():

"""test\_13"""

lis = make\_list()

assert lis.insert(70, 61) is False

Название файла: main.py

"""Module include check of workable unrolled linked list"""

from UnrolledLinkedList import UnrolledLinkedList

def check(arr1, arr2, n\_array=None):

"""function of check work of unrolled linked list"""

lis = UnrolledLinkedList(arr1[:47])

if n\_array is not None:

lis.len\_of\_node\_array = n\_array

print("List node size:", lis.len\_of\_node\_array)

lis.print\_list()

print('------------')

for i in arr1[47:50]:

lis.push(i)

print("Push elements")

lis.print\_list()

print('------------')

list\_for\_insert = [23, 49, 0]

for i in range(3):

lis.insert(arr1[50+i], list\_for\_insert[i])

print(f"Insert {arr1[50+i]} to index {list\_for\_insert[i]}")

lis.print\_list()

print('------------')

for i in arr2:

index = lis.find\_index\_of\_number(i)

lis.delete\_number(index)

print(f"Found {i} with index {index} and deleted it")

lis.print\_list()

print('------------')

arr\_with\_start\_elements = [54, 22, 60, 68, 31, 5, 17, 37, 32, 46, 49, 6, 62, 77, 16, 19, 20, 14, 3, 56, 56, 11, 58, 77, 95, 44, 11, 21,

32, 88, 58, 23, 30, 41, 89, 48, 89, 29, 76, 45, 37, 99, 97, 17, 46, 59, 91, 21, 61, 36, 101, 102, 103]

arr\_with\_elements\_for\_finding = [68, 21, 54, 36]

check(arr\_with\_start\_elements, arr\_with\_elements\_for\_finding)

Название файла: Calculate\_size.py

"""Module include function of find node size"""

from math import ceil

def calculate\_optimal\_node\_size(num\_elements):

"""function of calculate node size"""

memory = num\_elements \* 4

return ceil(memory / 64) + 1