

1 Цель работы

Научиться реализовывать такие базовые структуры данных, как: массив, список, хеш-таблица, множество и стек.

2 Задание

Вариант 4

Задание на реализацию:

Задание 5:

Реализуйте структуру данных «Хеш-таблица» на основе такой структуры данных, как «Односвязный список» (не использовать другие структуры данных). Обращение к элементам хеш-таблицы должно производиться с использованием дандерных (магических) методов Python. Также реализуйте 2 варианта проверки вхождения элемента в структуру данных.

Задание 8:

Реализуйте структуру данных «Стек» на основе такой структуры данных, как «Двусвязный список». Также реализуйте 2 варианта проверки вхождения элемента в структуру данных.

3 Теория по заданиям

В задании 5 присутствуют такие структуры данных как Хеш-таблица и Односвязный список.

Односвязный список – это линейная и однонаправленная структура данных, в которой используются узлы. Узлы хранят в себе данные и ссылку на следующий узел с такими же данными.

Началом односвязного списка называют – head (головой)

Концом односвязного списка – tail (хвостом)

Односвязный список можно визуализировать как на рисунке 3.1.

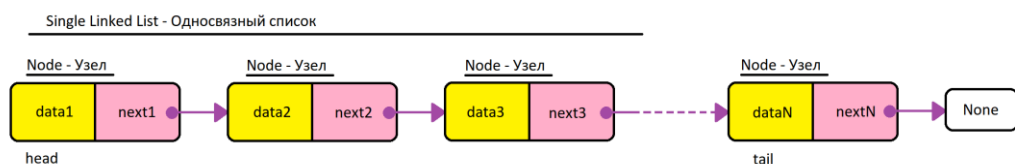


Рисунок 3.1 – Односвязный список

Хэш-Таблица – это структура данных, которая хранит значения с использованием пары ключей и значений. Каждому значению присваивается уникальный ключ, который генерируется с помощью хеш-функции.

Хэш-Таблицу можно визуализировать как на рисунке 3.2.

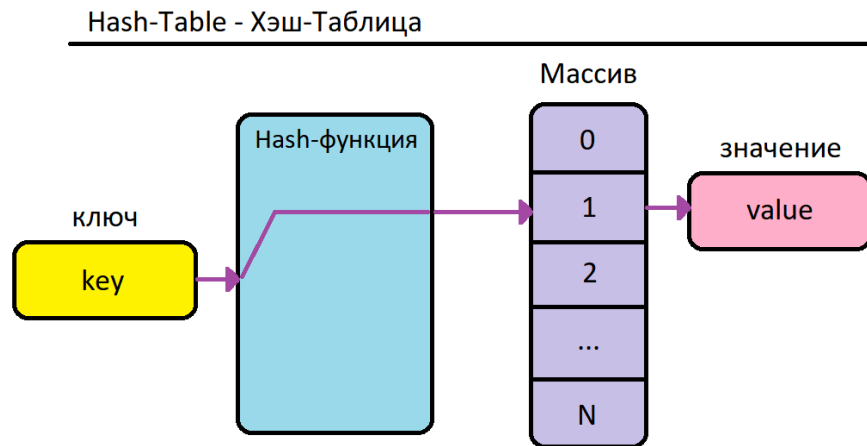


Рисунок 3.2 – Хэш-таблица

Ключ попадает в обработку Хэш-функции и записывается в Массив значение.

Но по заданию нужно реализовать структуру данных Хеш-таблица на основе Односвязного списка и не использовать другие структуры данных.

То есть вместо Массива возьмём за основу Односвязный список, а также хранить ключи и значения в отдельном Односвязном списке. Визуализировать это можно как на рисунке 3.3.

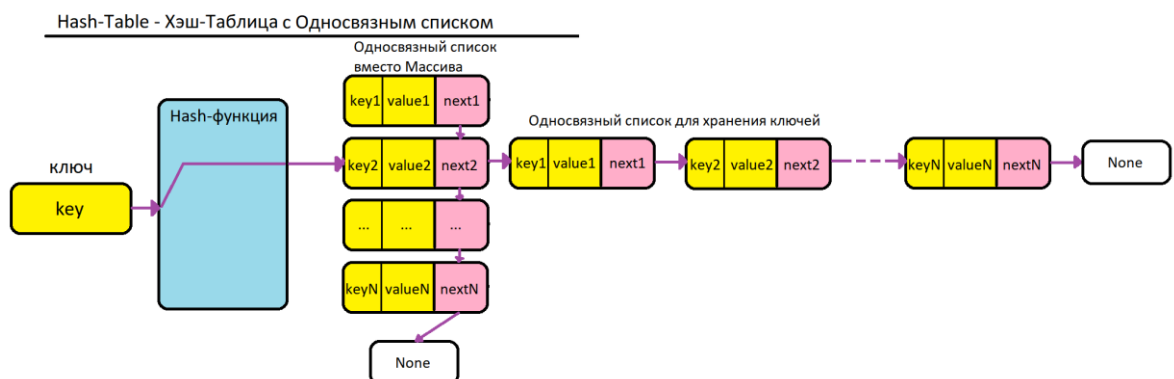


Рисунок 3.3 – Хэш-Таблица с Односвязным списком

В задании 8 присутствуют такие структуры данных как Стек и Двусвязный список.

Двусвязный список – это тоже самое, что и Односвязный список, только вместе с ссылкой на следующий узел прибавляется ссылка на предыдущий узел.

Двусвязный список можно визуализировать как на рисунке 3.4.

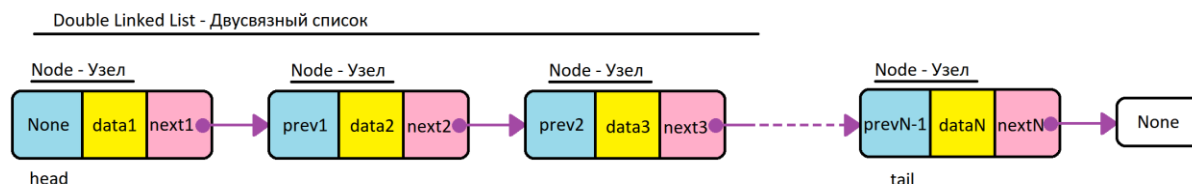


Рисунок 3.4 – Двусвязный список

Стек - это линейная структура данных, в которой элементы могут вставляться и удаляться только с одной стороны списка.

Стек можно визуализировать как на рисунке 3.5.

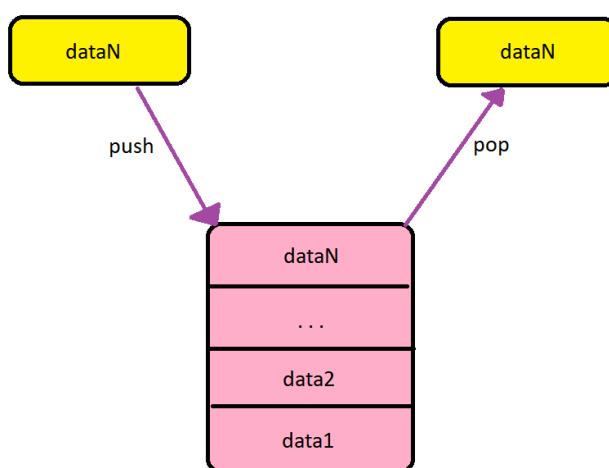


Рисунок 3.5 – Стек

Но по заданию Стек будет реализован на основе Двусвязного списка. Визуализировать можно это как на рисунке 3.6.

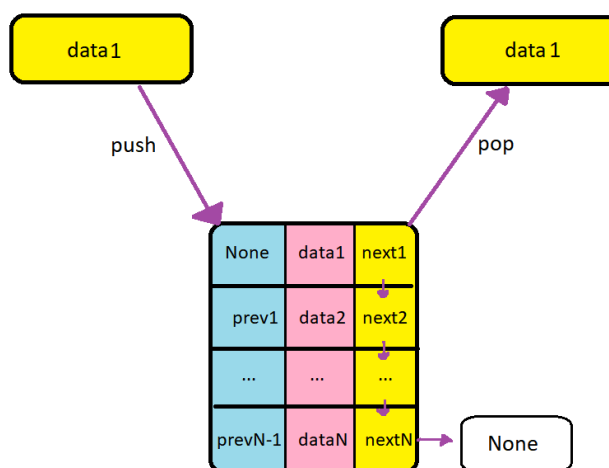


Рисунок 3.6 – Стек с двусвязным списком

4 Код программы для задания 5

Путь файла с Односвязным списком – stucturedata/simplelinkedlists.py

```
from typing import Generic, Optional, TypeVar, Iterator
from dataclasses import dataclass

T = TypeVar("T")

# класс Узла для ОдносвязногоСписка,
# в котором хранится элемент и ссылка на None или на следующий Узел
@dataclass
class Node(Generic[T]):
    data: T
    next: Optional['Node[T]'] = None

class SimpleLinkedList(Generic[T]):
    # две основные переменные:
    # голова (начальный Узел) и хвост (конечный Узел)
    def __init__(self) -> None:
        self.head: Optional[Node[T]] = None
        self.tail: Optional[Node[T]] = None

    # возвращает длину Односвязного списка
    def get_length(self) -> int:
        length: int = 0
        node = self.head
        # если пусто то 0
        if node is None:
            return 0
        # засчитываем каждый Узел
        while node is not None:
            node = node.next
            length += 1
        return length

    # возвращает True если индекс есть в Списке, False - Нету
    def check_range(self, index: int) -> bool:
        length = self.get_length()
        # Если 0, то пусто
        if length == 0:
            return False
        # Если индекс меньше длины и больше или равно нулю, то True,
        # иначе - False
        return 0 <= index < length

    # вставка в начало
    def push_head(self, data: T) -> None:
        # создаем новый Узел с ссылкой на прошлый начальный Узел
        self.head = Node(data, self.head)
```

```

        # Если список пуст, то к хвосту присваивается тот же новый
Узел
        if self.tail is None:
            self.tail = self.head

        # вставка в конец
        def push_tail(self, data: T) -> None:
            # Если список не пуст, то добавляем к хвосту ссылку и
назначаем новых хвост на новый Узел
            if self.head is not None:
                self.tail.next = Node(data, None)
                self.tail = self.tail.next
            else:
                # иначе если пуст, то начало и конец равны
                self.head = Node(data, None)
                self.tail = self.head

        # вставка по индексу, задаётся индекс туда, там и будет стоять
новый Узел
        def insert(self, index: int, data: T) -> None:
            # если в начало, то воспользуемся методом push_head
            if index == 0:
                self.push_head(data)
                return
            elif index == self.get_length():
                # если в конец, то воспользуемся методом push_tail
                self.push_tail(data)
                return
            elif not self.check_range(index):
                # выдаёт ошибку, если индекс вне диапазона списка
                raise IndexError("Index is outside the range of the
SingleLinkedList")

            node = self.head
            for _ in range(index - 1):
                node = node.next
            # добавляем новый Узел
            node.next = Node(data, node.next)

        # получить Узел по индексу
        def get_node(self, index: int) -> Optional[Node[T]]:
            if not self.check_range(index):
                raise IndexError("Index is outside the range of the
SingleLinkedList")
            # поиск значения
            node = self.head
            for _ in range(index):
                node = node.next
            return node

        # получить значение по индексу

```

```

def get(self, index: int) -> Optional[T]:
    return self.get_node(index).data

# удаление Узла по индексу
def remove(self, index: int) -> None:
    # проверка индекса
    if not self.check_range(index):
        raise IndexError("Index is outside the range of the
SingleLinkedList")
    elif index == 0:
        # Если в начало, то сохраняем ссылку головы в саму голову
        self.head = self.head.next
        # Если же список теперь пуст, то в хвосте тоже должно быть
пусто
        if self.head == None:
            self.tail = None
        return

    # удаление Узла по индексу
    node = self.head
    for _ in range(index - 1):
        node = node.next

    # Если удаляется конечный Узел (хвост), то нужно перезаписать
хвост на предпоследний Узел
    if index == self.get_length() - 1:
        self.tail = node
    # Присваиваем к Узлу по index - 1 ссылку на Узел по index + 1
    node.next = node.next.next

# очистка (пустой список)
def clear(self) -> None:
    self.head = None
    self.tail = None

# магический метод итерирования - возвращает Узел
def __iter__(self) -> Iterator[Optional[Node[T]]]:
    for i in range(self.get_length()):
        yield self.get_node(i)

# магический метод вывода класса односвязного списка в формате
[знач1, знач2, ... знач N]
def __str__(self) -> str:
    text = ""
    for i in range(self.get_length()):
        text += str(self.get(i)) + ", "
    if text == "":
        return "[]"
    return f"[{text[:-2]}]"

```

```
# магический метод предназначен для машинно-ориентированного
вывода
def __repr__(self) -> str:
    return self.__str__()
```

Для аннотации типа берём из библиотеки **typing** функции - Generic, Optional, TypeVar, Iterator:

- **TypeVar** – любой тип переменной(если задаётся int то всегда будет только int если str то str и т.д.); (универсальный тип);
- **Generic** – Универсальный класс (Абстрактный базовый класс для универсальных типов.) чтобы использовать нужны TypeVar, только аргументы этого типа. (показывает, что за тип без него(Generic) тип не показывает);
- **Optional** – может хранить либо None либо узел списка (T);
- **Iterator** – указывает на то, что объект имеет реализацию метода iter.

Декоратор **@dataclass** – убирает лишние методы (автоматизирует) `__init__`, `__repr__`, `__str__` и `__eq__` вместо этого можно сразу аннотацию писать

Например было `__init__(self, name) -> self.name = name`, станет `name: str`

Класс **Node** – это Узел для Односвязного списка.

Класс принимает два значения – data (любые данные) и next (ссылка на следующий Узел).

Класс **SimpleLinkedList** – это Односвязный список, хранящий такие методы как:

- `__init__()` – имеет две основные переменные – head (начальный Узел) и tail (конечный Узел);
- `get_length()` – возвращает длину Односвязного списка;
- `check_range(index)` – возвращает True если индекс есть в Списке, False – Нету;
- `push_head(data)` – вставка в начало;
- `push_tail(data)` – вставка в конец;
- `insert(index, data)` – вставка по индексу, задаётся индекс там, где и будет стоять новый Узел;
- `get_node(index)` – получить Узел по индексу;
- `get(index)` – получить значение по индексу;
- `remove(index)` – удаление Узла по индексу;
- `clear()` – очистка (пустой список);
- `__iter__()` – магический метод итерирования - возвращает Узел;

- `__str__()` – магический метод вывода класса односвязного списка в формате [знач1, знач2, ... знач N];
- `__repr__()` – магический метод предназначен для машинно-ориентированного вывода;

Путь файла Хэш-таблицы – structuredata/hashtable.py

```
from typing import Generic, Optional, TypeVar, Iterator
from dataclasses import dataclass
from simplelinkedlists import SimpleLinkedList

# key - ключ
K = TypeVar("K")
# value - значение
V = TypeVar("V")

# класс для хранения ключа и значения,
@dataclass
class KV(Generic[K, V]):
    key: K
    value: V

# Хэш-таблица, принимает вместимость
class HashTable(Generic[K, V]):
    def __init__(self, capacity: int) -> None:
        # вместимость
        self.capacity = capacity
        # Односвязный список с узлами вместо индексов
        # - хранит либо ничего None либо Односвязный список из Узлов
        self.table: SimpleLinkedList[Optional[KV[K, V]]] =
SimpleLinkedList()
        # заполняем хэш-таблицу пустотой
        for _ in range(capacity):
            self.table.push_tail(None)

    # генерируем хэш для ключа
    def hash(self, key: K) -> int:
        h = hash(key)
        # равное распределение ключа по хэшу (зависит от вместимости)
        return h % self.capacity

    # магический метод для назначения к ключу (хэш-ключу) - значение
    def __setitem__(self, key: K, value: V) -> None:
        # создаём хэш-ключ
        index: int = self.hash(key)
        # начальная позиция хэш-таблицы (односвязного списка)
        node = self.table.head
        # доходим до позиции хэш-ключа (индекс)
        for _ in range(index - 1):
```



```

        node = node.next
    # если пусто, то создаём новый Односвязный список
    if node.data is None:
        node.data = SimpleLinkedList()

    # проверяем есть ли в списке ещё ключи
    node_into = node.data.head
    while node_into is not None:
        # если нашёлся ключ, то вписываем в него значение
        if node_into.data.key == key:
            node_into.data.value = value
            return
        node_into = node_into.next
    # если не нашёлся, то создаём новый ключ со значением
    node.data.push_tail(KV(key, value))

# магический метод получения значения по ключу
def __getitem__(self, key: K) -> Optional[V]:
    # создаём хэш-ключ
    index: int = self.hash(key)
    node = self.table.head
    # доходим до позиции хэш-ключа (индекс)
    for _ in range(index - 1):
        node = node.next
    # если нету хэш-ключа, то возвращаем None
    if node.data is None:
        return None
    # перебираем ключи
    node_into = node.data.head
    while node_into is not None:
        # если ключ найден, то возвращаем его значение
        if node_into.data.key == key:
            return node_into.data.value
        node_into = node_into.next
    # если нету ключа в хэш-ключе, то возвращаем None
    return None

# удаление ключа
def remove(self, key: K) -> bool:
    index: int = self.hash(key)
    node = self.table.head
    # доходим до позиции хэш-ключа (индекс)
    for _ in range(index - 1):
        node = node.next
    # если нету хэш-ключа - то не удаляет
    if node.data is None:
        return False
    node_into = node.data.head
    # счётчик индекса (позиция ключа)
    length = 0
    # перебор ключей из хэш-ключа

```

```

while node_into is not None:
    # если есть ключ то удаляется
    if node_into.data.key == key:
        # удаление из Односвязного списка Узел с ключём
        node.data.remove(length)
        # если теперь в хэш-ключе пусто, то назначаем None
        if node.data.head is None:
            node.data = None
        # удаленно успешно
        return True
    node_into = node_into.next
    length += 1
# не удаленно
return False

# возвращает список ключей
def keys(self) -> list[K]:
    keys: list[K] = []
    # it - Узлы хэш-таблицы
    for it in self.table:
        # если пусто, то ищем дальше хэш-ключ
        if it.data is None:
            continue
        node = it.data.head
        # добавляем ключи в список
        while node is not None:
            keys += [node.data.key]
            node = node.next
    return keys

# возвращает список значений
def values(self) -> list[V]:
    value: list[V] = []
    # it - Узлы хэш-таблицы
    for it in self.table:
        # если пусто, то ищем дальше хэш-ключ
        if it.data is None:
            continue
        node = it.data.head
        # добавляем значения ключей в список
        while node is not None:
            value += [node.data.value]
            node = node.next
    return value

# полная очистка хэш-таблицы
def clear(self) -> None:
    self.table: SimpleLinkedList[Optional[KV[K, V]]] =
SimpleLinkedList()
    for _ in range(self.capacity):
        self.table.push_tail(None)

```

```

# магический метод нахождения ключа (есть ли ключ в Хэш-таблице)
def __contains__(self, key: K) -> bool:
    return self.__getitem__(key) is not None

# магический метод итерирования - при итерировании будет
возвращать ключ
def __iter__(self) -> Iterator[K]:
    for key in self.keys():
        yield key

# магический метод вывода класса хэш-таблицы
def __str__(self) -> str:
    return str(self.table)

```

Импортируем класс Односвязного списка в Хэш-Таблицу, для её использования.

Класс **KV** – это класс для хранения ключа (**key**) и значения (**value**).

Класс **HashTable** – это Односвязный список, хранящий такие методы как:

- **__init__(capacity)** – принимает в себя вместимость, создаёт Односвязный список и заполняет его пустотой;
- **hash(key)** – генерирует хэш для ключа;
- **__setitem__(key, value)** – магический метод для назначения к ключу (хэш-ключу) – значение;
- **__getitem__(key)** – магический метод получения значения по ключу;
- **remove(key)** – удаление ключа;
- **keys()** – возвращает список ключей;
- **values()** – возвращает список значений;
- **clear()** – полная очистка хэш-таблицы;
- **__contains__(key)** – магический метод нахождения ключа (есть ли ключ в Хэш-таблице);
- **__iter__()** – магический метод итерирования - при итерировании будет возвращать ключ;
- **__str__()** – магический метод вывода класса хэш-таблицы.

4.1 Тесты

Тесты буду производиться модулем – **pytest**.

Для того, чтобы **pytest** смог проверить значения, нужно к каждой функции добавлять приставку **test_**. А чтобы сравнить ответы нужно использовать **assert**.

Код теста:

```

from structuredata.hashtable import HashTable

hash_table = HashTable(5)
dit = [(1, 1), (5, 2), (9, 3), (99, 4)]
dit2 = [(1, 1), (5, 2), (9, 3), (99, 4), (95, 5), (33, 6)]
dit3 = [(5, 2), (9, 3), (99, 4)]

# ВСТАВКА
def test_push():
    for key, value in dit:
        hash_table[key] = value
    for n in range(len(dit)):
        assert dit[n][1] in hash_table.values()
        assert dit[n][0] in hash_table

# вставка с количеством больше чем вместимость
def test_push_moreCapacity():
    for key, value in dit2:
        hash_table[key] = value
        print(hash_table[key], hash_table.keys())
    for n in range(len(dit2)):
        assert dit2[n][1] in hash_table.values()
        assert dit2[n][0] in hash_table

# очистка хэш-таблицы
def test_clear():
    hash_table.clear()
    assert hash_table.keys() == []
    assert hash_table.values() == []

# проверка методов keys() и values()
def test_keys_and_values():
    for key, value in dit2:
        hash_table[key] = value
        print(hash_table[key], hash_table.keys())
    k = set([i[0] for i in dit2])
    v = set([i[1] for i in dit2])
    assert k == set(hash_table.keys())
    assert v == set(hash_table.values())

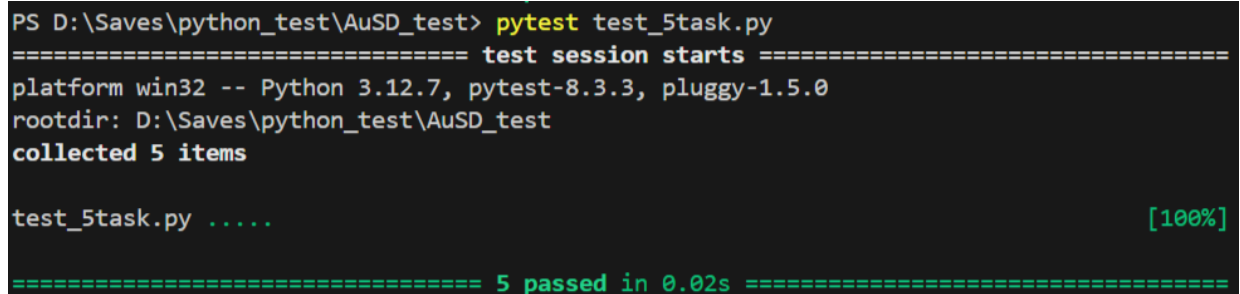
# проверка удаления
def test_remove():
    hash_table.remove(95)
    hash_table.remove(33)
    for n in range(len(dit)):
        assert dit[n][1] in hash_table.values()
        assert dit[n][0] in hash_table
    hash_table.remove(1)
    for n in range(len(dit3)):
        assert dit3[n][1] in hash_table.values()
        assert dit3[n][0] in hash_table

```

Чтобы запустить проверку нужно в консоли написать команду:

pytest <путь файла.py>

Результат пройденных тестов (рисунок 4.1).



```
PS D:\Saves\python_test\AuSD_test> pytest test_5task.py
===== test session starts =====
platform win32 -- Python 3.12.7, pytest-8.3.3, pluggy-1.5.0
rootdir: D:\Saves\python_test\AuSD_test
collected 5 items

test_5task.py ..... [100%]
===== 5 passed in 0.02s =====
```

Рисунок 4.1 – результат проверки теста 5 задания

Все 5 тестов были пройдены успешно.

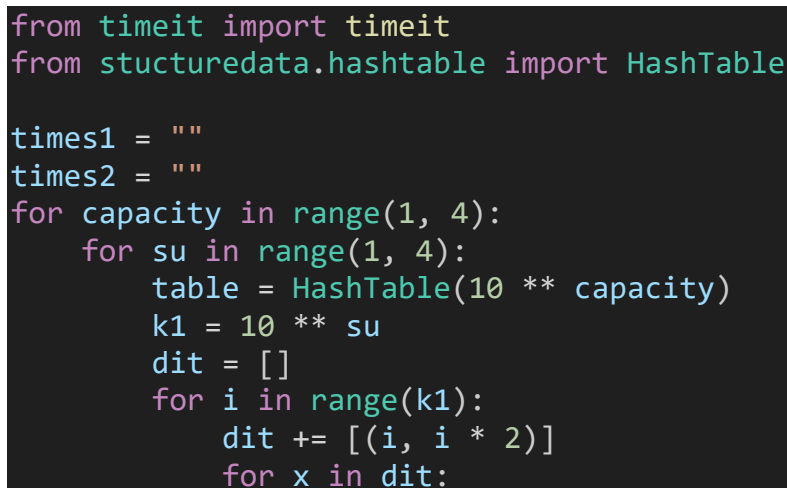
4.2 Бенчмарк

Для подсчёта времени на операции вставки и удаления используем библиотеку **timeit** и её функцию **timeit(stmt, setup, timer, number, globals)** возвращает секунды:

- **stmt** – код, у которого будут считать длительность выполнения;
- **setup** – код, который выполняет один раз;
- **timer** – использует класс **Timer** из модуля **timeit**;
- **number** – количество оборотов, сколько раз будет повторяться код и усреднит время;
- **globals** – словарь из ключей(переменная в коде) и значений(переменную на которую надо заменить).

Был произведён бенчмарк на вместимость и количество значений.

Код программы:



```
from timeit import timeit
from structureddata.hashtable import HashTable

times1 = ""
times2 = ""
for capacity in range(1, 4):
    for su in range(1, 4):
        table = HashTable(10 ** capacity)
        k1 = 10 ** su
        dit = []
        for i in range(k1):
            dit += [(i, i * 2)]
        for x in dit:
```

```

        table[x[0]] = x[1]
code1 = """for x in dit:
    table[x[0]] = x[1]
"""
code2 = """for x in dit:
    table.remove(x[0])
"""
time1 = timeit(code1, number=1, globals={"dit": dit, "table":
table})
time2 = timeit(code2, number=1, globals={"dit": dit, "table":
table})
times1 += f"          {k1}          |          {10 ** capacity}
|          {time1}\n"
times2 += f"          {k1}          |          {10 ** capacity}
|          {time2}\n"

print("Вставка")
print(f"Кол-во элементов      |      Вместимость хэш-таблицы |      Время")
print(times1)
print("Удаление")
print(f"Кол-во элементов      |      Вместимость хэш-таблицы |      Время")
print(times2)

```

Вывод в консоле изображенно на рисунке 4.2.

| Вставка | | |
|------------------|-------------------------|------------------------|
| Кол-во элементов | Вместимость хэш-таблицы | Время |
| 10 | 10 | 2.4299981305375695e-05 |
| 100 | 10 | 0.00029239998548291624 |
| 1000 | 10 | 0.009438299981411546 |
| 10 | 100 | 2.279998987466097e-05 |
| 100 | 100 | 0.00045980000868439674 |
| 1000 | 100 | 0.005409500008681789 |
| 10 | 1000 | 2.3999979021027684e-05 |
| 100 | 1000 | 0.0004594000056385994 |
| 1000 | 1000 | 0.03176079998956993 |
| Удаление | | |
| Кол-во элементов | Вместимость хэш-таблицы | Время |
| 10 | 10 | 0.0001126999850384891 |
| 100 | 10 | 0.000603899999987334 |
| 1000 | 10 | 0.01030460000038147 |
| 10 | 100 | 5.020000389777124e-05 |
| 100 | 100 | 0.0007305000035557896 |
| 1000 | 100 | 0.008107500005280599 |
| 10 | 1000 | 5.1700015319511294e-05 |
| 100 | 1000 | 0.0007083000091370195 |
| 1000 | 1000 | 0.03432279999833554 |

Рисунок 4.2 – вывод в консоли времени в секунды

Можно заметить, что количество элементов сильнее влияет на время, чем вместимость Хэш-таблице.

| Кол-во элементов | Вместимость хэш-таблицы | Время |
|------------------|-------------------------|-------------------------|
| 10 | 10 | 0.000024299981305375695 |
| 10 | 100 | 0.000022799998987466097 |
| 10 | 1000 | 0.000023999979021027684 |
| 100 | 10 | 0.00029239998548291624 |
| 100 | 100 | 0.00045980000868439674 |
| 100 | 1000 | 0.0004594000056385994 |
| 1000 | 10 | 0.009438299981411546 |
| 1000 | 100 | 0.005409500008681789 |
| 1000 | 1000 | 0.03176079998956993 |

Таблица 4.1 – Вставка в Хэш-таблице

| Кол-во элементов | Вместимость хэш-таблицы | Время |
|------------------|-------------------------|-------------------------|
| 10 | 10 | 0.0001126999850384891 |
| 10 | 100 | 0.00005020000389777124 |
| 10 | 1000 | 0.000051700015319511294 |
| 100 | 10 | 0.000603899999987334 |
| 100 | 100 | 0.0007305000035557896 |
| 100 | 1000 | 0.0007083000091370195 |
| 1000 | 10 | 0.01030460000038147 |
| 1000 | 100 | 0.008107500005280599 |
| 1000 | 1000 | 0.03432279999833554 |

Таблица 4.2 – Удаление в Хэш-таблице

5 Код программы для задания 8

Путь файла с Двусвязным списком – structuredata/doublelinkedlists.py

```
from typing import Generic, Optional, TypeVar
from dataclasses import dataclass

T = TypeVar("T")

# класс Узла для двусвязного Списка,
# в котором хранится ссылка на предыдущий Узел,
# элемент и ссылка на None или на следующий Узел
```

```

@dataclass
class Node(Generic[T]):
    data: T
    prev: Optional['Node[T]'] = None
    next: Optional['Node[T]'] = None

class DoubleLinkedList(Generic[T]):
    # две основные переменные:
    # голова (начальный Узел) и хвост (конечный Узел)
    def __init__(self) -> None:
        self.head: Optional[Node[T]] = None
        self.tail: Optional[Node[T]] = None

    # возвращает длину двусвязного списка
    def get_length(self) -> int:
        length: int = 0
        node = self.head
        # если пусто то 0
        if node is None:
            return 0
        # засчитываем каждый Узел
        while node is not None:
            node = node.next
            length += 1
        return length

    # возвращает True если индекс есть в Списке, False - Нету
    def check_range(self, index: int) -> bool:
        length = self.get_length()
        # Если 0, то пусто
        if length == 0:
            return False
        # Если индекс меньше длины и больше или равно нулю, то True,
        # иначе - False
        return 0 <= index < length

    # вставка в начало
    def push_head(self, data: T) -> None:
        # создаем новый Узел с ссылкой на прошлый начальный Узел
        self.head = Node(data, None, self.head)
        # Если список пуст, то к хвосту присваивается тот же новый
        # Узел
        if self.tail is None:
            self.tail = self.head
        else:
            self.head.next.prev = self.head

    # вставка в конец
    def push_tail(self, data: T) -> None:

```



```

        # Если список не пуст, то добавляем к хвосту ссылку и
        # назначаем новых хвост на новый Узел
        if self.head is not None:
            self.tail.next = Node(data, self.tail, None)
            self.tail = self.tail.next
        else:
            # иначе если пуст, то начало и конец равны
            self.head = Node(data, None, None)
            self.tail = self.head

    # вставка по индексу, задаётся индекс туда, там и будет стоять
    # новый Узел
    def insert(self, index: int, data: T) -> None:
        # если в начало, то воспользуемся методом push_head
        if index == 0:
            self.push_head(data)
            return
        elif index == self.get_length():
            # если в конец, то воспользуемся методом push_tail
            self.push_tail(data)
            return
        elif not self.check_range(index):
            # выдаёт ошибку, если индекс вне диапазона списка
            raise IndexError("Index is outside the range of the
SingleLinkedList")

        node = self.head
        for _ in range(index - 1):
            node = node.next
        # добавляем новый Узел
        node.next = Node(data, node, node.next)

    # получить значение по индексу
    def get(self, index: int) -> Optional[T]:
        # проверка индекса
        if not self.check_range(index):
            raise IndexError("Index is outside the range of the
SingleLinkedList")

        # поиск значения
        node = self.head
        for _ in range(index):
            node = node.next
        return node.data

    # удаление Узла по индексу
    def remove(self, index: int) -> Optional[T]:
        # проверка индекса
        if not self.check_range(index):
            raise IndexError("Index is outside the range of the
SingleLinkedList")

```

```

elif index == 0:
    # Если в начало, то сохраняем ссылку головы в саму голову
    past_node = self.head
    self.head = self.head.next
    # Если же список теперь пуст, то в хвосте тоже должно быть
пусто

    if self.head == None:
        self.tail = None
    else:
        self.head.prev = None
    return past_node.data

# удаление Узла по индексу
node = self.head
for _ in range(index - 1):
    node = node.next

# Если удаляется конечный Узел (хвост), то нужно перезаписать
хвост на предпоследний Узел
if index == self.get_length() - 1:
    self.tail = node
else:
    # Присваиваем к Узлу по index + 1 пред. ссылку на Узел по
index - 1
    node.next.next.prev = node
    past_node = node.next
    # Присваиваем к Узлу по index - 1 след. ссылку на Узел по
index + 1
    node.next = node.next.next
    return past_node.data

# очистка (пустой список)
def clear(self) -> None:
    self.head = None
    self.tail = None

# магический метод вывода класса двусвязного списка в формате
[знач1, знач2, ... знач N]
def __str__(self) -> str:
    text = ""
    for i in range(self.get_length()):
        text += str(self.get(i)) + ", "
    if text == "":
        return "[]"
    return f"[{text[:-2]}]"

# магический метод предназначен для машинно-ориентированного
вывода
def __repr__(self) -> str:
    return self.__str__()

```

Класс **Node** – это Узел для Двусвязного списка.

Класс принимает три значения – data (любые данные), prev (ссылка на предыдущий Узел) и next (ссылка на следующий Узел).

Класс **DoubleLinkedList** – это Двусвязный список, хранящий такие методы как:

- **__init__()** – имеет две основные переменные – head (начальный Узел) и tail (конечный Узел);
- **get_length()** – возвращает длину Двусвязного списка;
- **check_range(index)** – возвращает True если индекс есть в Списке, False – Нету;
- **push_head(data)** – вставка в начало;
- **push_tail(data)** – вставка в конец;
- **insert(index, data)** – вставка по индексу, задаётся индекс там, где и будет стоять новый Узел;
- **get(index)** – получить значение по индексу;
- **remove(index)** – удаление Узла по индексу;
- **clear()** – очистка (пустой список);
- **__str__()** – магический метод вывода класса односвязного списка в формате [знач1, знач2, ... знач N];
- **__repr__()** – магический метод предназначен для машинно-ориентированного вывода;

Путь файла со Стекком – structuredata/stack.py

```
from typing import TypeVar, Generic
from doublelinkedlists import DoubleLinkedList

T = TypeVar("T")

# ошибка если вместимость Стэка мала
class SizeIsOver(Exception):
    pass

# ошибка если при просмотре значения, а Стэк пуст
class EmptyStack(Exception):
    pass

# Стэк принимает размер (вместимость)
class Stack(Generic[T]):
    # класс хранит - данную длину, вместимость и Двусвязный список
    def __init__(self, size: int) -> None:
```

```

        self.length: int = 0
        self.size: int = size
        self.arr: DoubleLinkedList[Generic[T]] = DoubleLinkedList()

# возвращает нынешнюю длину
def get_size(self) -> int:
    return self.length

# Стэк пуст?
def is_empty(self) -> bool:
    return self.length == 0

# Вставляет значение на верхушку Двусвязного списка
def push(self, value: T):
    if self.get_size() >= self.size:
        raise SizeIsOver(f"Size of Stack is over: {value}")
    self.arr.push_head(value)
    self.length += 1

# Удаляет значение с верхушки Двусвязного списка
def pop(self) -> T:
    if self.is_empty():
        raise EmptyStack("Stack is empty")
    self.length -= 1
    return self.arr.remove(0)

# Возвращает значение верхушки Двусвязного списка
def peak(self) -> T:
    if self.is_empty():
        raise EmptyStack("Stack is empty")
    return self.arr.get(0)

# Вывод значений
def __str__(self) -> str:
    return str(self.arr)

```

Класс **SizeIsOver(Exception)** – ошибка если вместимость Стэка мала.

Класс **EmptyStack(Exception)** – ошибка если при просмотре значения, а Стэк пуст.

Класс **Stack** – это Стек, хранящий такие методы как:

- **__init__(size)** – класс хранит - данную длину, вместимость и Двусвязный список;
- **get_size()** – возвращает нынешнюю длину;
- **is_empty()** – стэк пуст? (возвращает True если пусто, иначе False);
- **push(value)** – вставляет значение на верхушку Двусвязного списка;
- **pop()** – удаляет значение с верхушки Двусвязного списка;
- **peak()** – возвращает значение верхушки Двусвязного списка;

- `__str__()` – вывод значений.

5.1 Тесты

Код теста:

```
from structuredata.stack import Stack

stack = Stack(5)
lt1 = [5, 4, 3, 2, 1]

# проверка вставки
def test_push():
    for i in range(len(lt1)):
        stack.push(lt1[i])
    m = str(stack).replace("[", "").replace("]", "").replace(",", " ")
    m = m.split()
    n = 0
    lt1.reverse()
    for i in map(int, m):
        assert i == lt1[n]
        n += 1

# проверка удаления
def test_remove():
    assert stack.pop() == lt1[0]
    assert stack.pop() == lt1[1]
    assert stack.pop() == lt1[2]
    assert stack.pop() == lt1[3]
    assert stack.pop() == lt1[4]

# проверка пустой ли стек
def test_empty():
    assert stack.is_empty() is True

# проверка функции peak (верхушки стека)
def test_peak():
    lt1.reverse()
    for i in range(len(lt1)):
        stack.push(lt1[i])
        assert stack.peak() == lt1[i]

# проверка размера стека
def test_size():
    assert stack.get_size() == 5
    stack.pop()
    assert stack.get_size() == 4
    stack.pop()
    assert stack.get_size() == 3
    stack.pop()
    assert stack.get_size() == 2
    stack.pop()
    assert stack.get_size() == 1
```

```

stack.pop()
assert stack.get_size() == 0
stack.push(45)
assert stack.get_size() == 1

```

Результат пройденных тестов (рисунок 4.2).

```

PS D:\Saves\python_test\AuSD_test> pytest test_8task.py
===== test session starts =====
platform win32 -- Python 3.12.7, pytest-8.3.3, pluggy-1.5.0
rootdir: D:\Saves\python_test\AuSD_test
collected 5 items

test_8task.py ..... [100%]

===== 5 passed in 0.02s =====

```

Рисунок 5.1 – результат проверки теста 8 задания

Все 5 тестов были пройдены успешно.

5.2 Бенчмарк

Был произведён бенчмарк на вместимость и количество значений.

Код программы:

```

from timeit import timeit
from structuredata.stack import Stack

times1 = ""
times2 = ""

for capacity in range(1, 4):
    for su in range(1, 4):
        k1 = 10 ** capacity
        stack = Stack(k1)
        dit = []
        for i in range(k1):
            dit += [i * 2]
        code1 = """for x in dit:
                    stack.push(x)
                """
        code2 = """for x in dit:
                    stack.pop()
                """
        time1 = timeit(code1, number=1, globals={"dit": dit, "stack":
stack})
        time2 = timeit(code2, number=1, globals={"dit": dit, "stack":
stack})
        times1 += f"          {k1}          |          {time1}\n"

```

```

times2 += f"           {k1}           |           {time2}\n"
print("Вставка")
print(f"Кол-во элементов (вместимость)           |           Время")
print(times1)
print("Удаление")
print(f"Кол-во элементов (вместимость)           |           Время")
print(times2)

```

Вывод в консоли изображено на рисунке 5.2.

```

Вставка
Кол-во элементов (вместимость)           |           Время
10           |           6.680001388303936e-05
10           |           3.029999788850546e-05
10           |           3.010002546943724e-05
100          |           0.0002767000114545226
100          |           0.0002770999853964895
100          |           0.0002730000123847276
1000         |           0.0029863000090699643
1000         |           0.0027956000121776015
1000         |           0.002766899997368455

Удаление
Кол-во элементов (вместимость)           |           Время
10           |           0.0001145999995060265
10           |           5.339999916031957e-05
10           |           5.2400020649656653e-05
100          |           0.0010095999750774354
100          |           0.0009747999720275402
100          |           0.0008672999974805862
1000         |           0.04505129999597557
1000         |           0.045132100000046194
1000         |           0.04489359998842701

```

Рисунок 5.2 – вывод в консоли времени в секунды

Можно заметить, что количество элементов влияет на время обработки Стека.

| Кол-во элементов (вместимость) | Время |
|--------------------------------------|------------------------|
| 10 | 0.00006680001388303936 |
| 10 | 0.00003029999788850546 |
| 10 | 0.00003010002546943724 |
| 100 | 0.0002767000114545226 |
| 100 | 0.0002770999853964895 |

| | |
|------|-----------------------|
| 100 | 0.0002730000123847276 |
| 1000 | 0.0029863000090699643 |
| 1000 | 0.0027956000121776015 |
| 1000 | 0.002766899997368455 |

Таблица 5.1 – Вставка в Стеке

| Кол-во элементов (вместимость) | Время |
|--------------------------------------|-------------------------|
| 10 | 0.0001145999995060265 |
| 10 | 0.00005339999916031957 |
| 10 | 0.000052400020649656653 |
| 100 | 0.0010095999750774354 |
| 100 | 0.0009747999720275402 |
| 100 | 0.0008672999974805862 |
| 1000 | 0.04505129999597557 |
| 1000 | 0.045132100000046194 |
| 1000 | 0.04489359998842701 |

Таблица 5.2 – Удаление в Стеке

Выводы

В данной лабораторной работе я создала такие структуры данные как:

- Односвязный список
- Двусвязный список
- Хэш-Таблица
- Стек

Научилась применять аннотацию типов с помощью библиотеки `typing`. А также провела тест своих структур данных вместе с бечмарком.