Федеральное государственное бюджетное образовательное учреждение высшего образования
«Сибирский государственный университет телекоммуникаций и информатики»
(СибГУТИ)

Институт информатики и вычислительной техники

09.03.01 "Информатика и вычислительная техника"
профиль "Программное обеспечение средств
вычислительной техники и автоматизированных систем"

Кафедра прикладной математики и кибернетики

Современные технологии программирования

Лабораторная работа №10

Практическая работа. Абстрактный тип данных (ADT)

Множество (на шаблоне)

Выполнил:

студент гр.ИП-213 Дмитриев Антон Александрович
ФИО студента

«__» _____ 2025 г.

Проверил:

Преподаватель

ФИО преподавателя

«__» _____ 2025 г. Оценка_____

Новосибирск 2025 г.

1. **Задание**

1. В соответствии с приведенной ниже спецификацией реализуйте шаблон классов «множество». Для тестирования в качестве параметра шаблона T выберите типы:

- int;
- TFrac (простая дробь), разработанный вами ранее.

2. Протестировать каждую операцию, определенную на типе данных, используя средства модульного тестирования.

3. Если необходимо, предусмотрите возбуждение исключительных ситуаций.

2. Исходные тексты программ.

Class.py:

```
import math
from typing import TypeVar, Generic, Iterator

class TFrac:
    """Класс для представления простой дроби"""

    def __init__(self, numerator=0, denominator=1):
        """Конструктор дроби"""
        if denominator == 0:
            raise ValueError("Знаменатель не может быть равен 0")

        self.numerator = numerator
        self.denominator = denominator
        self._normalize()

    def _normalize(self):
        """Приведение дроби к нормальной форме"""
        if self.denominator < 0:
            self.numerator = -self.numerator
            self.denominator = -self.denominator

        gcd_val = math.gcd(abs(self.numerator), self.denominator)
        if gcd_val > 1:
            self.numerator //= gcd_val
            self.denominator //= gcd_val

    def __eq__(self, other):
        """Проверка равенства дробей"""
        if isinstance(other, TFrac):
            return (self.numerator == other.numerator and
                    self.denominator == other.denominator)
        return False

    def __hash__(self):
        """Хэш для использования в множестве"""
        return hash((self.numerator, self.denominator))

    def __str__(self):
        """Строковое представление дроби"""
        return f"{self.numerator}/{self.denominator}"

    def __repr__(self):
```

```
        """Представление для отладки"""
        return f"TFrac({self.numerator}, {self.denominator})"

    def copy(self):
        """Создание копии дроби"""
        return TFrac(self.numerator, self.denominator)

T = TypeVar('T')


class tset(Generic[T]):
    """Шаблон класса множество"""

    def __init__(self):
        """Конструктор - создает пустое множество"""
        self._data = set()

    def clear(self) -> None:
        """Опустошить множество"""
        self._data.clear()

    def add(self, d: T) -> None:
        """Добавить элемент в множество"""
        self._data.add(d)

    def remove(self, d: T) -> None:
        """Удалить элемент из множества"""
        if d in self._data:
            self._data.remove(d)
        else:
            raise KeyError(f"Элемент {d} не найден в множестве")

    def is_empty(self) -> bool:
        """Проверить, пусто ли множество"""
        return not self._data

    def contains(self, d: T) -> bool:
        """Проверить, принадлежит ли элемент множеству"""
        return d in self._data

    def union(self, q: 'tset[T]') -> 'tset[T]':
        """Объединить множества (сложение)"""
        result = tset[T]()
        result._data = self._data.union(q._data)
        return result
```

```
def difference(self, q: 'tset[T]') -> 'tset[T]':
    """Вычесть множество"""
    result = tset[T]()
    result._data = self._data.difference(q._data)
    return result

def intersection(self, q: 'tset[T]') -> 'tset[T]':
    """Пересечение множеств (умножение)"""
    result = tset[T]()
    result._data = self._data.intersection(q._data)
    return result

def size(self) -> int:
    """Количество элементов во множестве"""
    return len(self._data)

def get_element(self, j: int) -> T:
    """Получить элемент по индексу (для перебора)"""
    if j < 1 or j > self.size():
        raise IndexError(f"Индекс {j} вне диапазона [1, {self.size()}]")

    # Преобразуем множество в список для доступа по индексу
    return list(self._data)[j - 1]

def __str__(self) -> str:
    """Строковое представление множества"""
    elements = list(self._data)
    return "{" + ", ".join(str(e) for e in elements) + "}"

def __eq__(self, other: object) -> bool:
    """Проверка равенства множеств"""
    if not isinstance(other, tset):
        return False
    return self._data == other._data

def __iter__(self) -> Iterator[T]:
    """Итератор по элементам множества"""
    return iter(self._data)

def copy(self) -> 'tset[T]':
    """Создать копию множества"""
    result = tset[T]()
    result._data = self._data.copy()
    return result
```

Tests.py:

```
import pytest
from lab10 import tset
from lab10 import TFrac



class TestTSetInt:
    """Тесты для множества целых чисел"""

    def test_constructor(self):
        """Тест конструктора"""
        s = tset[int]()
        assert s.is_empty()
        assert s.size() == 0

    def test_add(self):
        """Тест добавления элементов"""
        s = tset[int]()
        s.add(1)
        s.add(2)
        s.add(1)  # Дубликат не должен добавиться

        assert not s.is_empty()
        assert s.size() == 2
        assert s.contains(1)
        assert s.contains(2)
        assert not s.contains(3)

    def test_remove(self):
        """Тест удаления элементов"""
```

```
    s = tset[int]()
    s.add(1)
    s.add(2)
    s.add(3)

    s.remove(2)
    assert s.size() == 2
    assert not s.contains(2)
    assert s.contains(1)
    assert s.contains(3)

    # Проверка исключения при удалении несуществующего элемента
    with pytest.raises(KeyError):
        s.remove(5)

def test_clear(self):
    """Тест очистки множества"""
    s = tset[int]()
    s.add(1)
    s.add(2)
    s.add(3)

    assert s.size() == 3
    s.clear()
    assert s.is_empty()
    assert s.size() == 0

def test_union(self):
    """Тест объединения множеств"""
    s1 = tset[int]()
```

```
    s1.add(1)
    s1.add(2)
    s1.add(3)

    s2 = tset[int]()
    s2.add(3)
    s2.add(4)
    s2.add(5)

    result = s1.union(s2)
    assert result.size() == 5
    assert result.contains(1)
    assert result.contains(2)
    assert result.contains(3)
    assert result.contains(4)
    assert result.contains(5)

    # Проверка, что исходные множества не изменились
    assert s1.size() == 3
    assert s2.size() == 3

def test_union_test_cases(self):
    """Тест объединения по тестовым наборам из задания"""
    # Тест 1: оба множества пусты
    s1 = tset[int]()
    s2 = tset[int]()
    result = s1.union(s2)
    assert result.is_empty()

    # Тест 2: первое множество пустое, второе с элементом 0
```

```
s1 = tset[int]()
s2 = tset[int]()
s2.add(0)
result = s1.union(s2)
assert result.size() == 1
assert result.contains(0)

# Тест 3: первое множество {1}, второе {0}
s1 = tset[int]()
s2 = tset[int]()
s1.add(1)
s2.add(0)
result = s1.union(s2)
assert result.size() == 2
assert result.contains(1)
assert result.contains(0)

# Тест 4: оба множества {1, 0}
s1 = tset[int]()
s2 = tset[int]()
s1.add(1)
s1.add(0)
s2.add(1)
s2.add(0)
result = s1.union(s2)
assert result.size() == 2
assert result.contains(1)
assert result.contains(0)

# Тест 5: {1, 2, 3} ∪ {3, 4, 5} = {1, 2, 3, 4, 5}
```

```
    s1 = tset[int]()
    s2 = tset[int]()
    s1.add(1)
    s1.add(2)
    s1.add(3)
    s2.add(3)
    s2.add(4)
    s2.add(5)
    result = s1.union(s2)
    assert result.size() == 5
    for i in range(1, 6):
        assert result.contains(i)

def test_difference(self):
    """Тест вычитания множеств"""
    s1 = tset[int]()
    s1.add(1)
    s1.add(2)
    s1.add(3)
    s1.add(4)

    s2 = tset[int]()
    s2.add(3)
    s2.add(4)
    s2.add(5)

    result = s1.difference(s2)
    assert result.size() == 2
    assert result.contains(1)
    assert result.contains(2)
```

```
    assert not result.contains(3)
    assert not result.contains(4)
    assert not result.contains(5)

def test_intersection(self):
    """Тест пересечения множеств"""
    s1 = tset[int]()
    s1.add(1)
    s1.add(2)
    s1.add(3)
    s1.add(4)

    s2 = tset[int]()
    s2.add(3)
    s2.add(4)
    s2.add(5)
    s2.add(6)

    result = s1.intersection(s2)
    assert result.size() == 2
    assert result.contains(3)
    assert result.contains(4)
    assert not result.contains(1)
    assert not result.contains(2)
    assert not result.contains(5)

def test_get_element(self):
    """Тест получения элемента по индексу"""
    s = tset[int]()
    s.add(10)
```

```
    s.add(20)
    s.add(30)

    elements = set()
    for i in range(1, s.size() + 1):
        elements.add(s.get_element(i))

    assert elements == {10, 20, 30}

    # Проверка исключения при неверном индексе
    with pytest.raises(IndexError):
        s.get_element(0)
    with pytest.raises(IndexError):
        s.get_element(4)

def test_iteration(self):
    """Тест итерации по множеству"""
    s = tset[int]()
    s.add(1)
    s.add(2)
    s.add(3)

    elements = list(s)
    assert set(elements) == {1, 2, 3}
    assert len(elements) == 3

def test_copy(self):
    """Тест копирования множества"""
    s1 = tset[int]()
    s1.add(1)
```

```
        s1.add(2)
        s1.add(3)

        s2 = s1.copy()
        assert s1 == s2

        # Проверка, что копия независима
        s2.add(4)
        assert s1.size() == 3
        assert s2.size() == 4


class TestTSetTFrac:
    """Тесты для множества дробей"""

    def test_fraction_operations(self):
        """Тест основных операций с дробями"""
        s = tset[TFrac]()

        f1 = TFrac(1, 2)
        f2 = TFrac(2, 3)
        f3 = TFrac(3, 4)

        s.add(f1)
        s.add(f2)
        s.add(f1)  # Дубликат
        s.add(f3)

        assert s.size() == 3
        assert s.contains(TFrac(1, 2))
```

```
    assert s.contains(TFrac(2, 3))
    assert s.contains(TFrac(3, 4))
    assert not s.contains(TFrac(1, 4))

def test_fraction_normalization(self):
    """Тест нормализации дробей в множестве"""
    s = tset[TFrac]()

    f1 = TFrac(2, 4)  # 1/2 после нормализации
    f2 = TFrac(1, 2)  # 1/2

    s.add(f1)
    s.add(f2)

    # Дроби должны считаться равными после нормализации
    assert s.size() == 1

def test_fraction_union(self):
    """Тест объединения множеств дробей"""
    s1 = tset[TFrac]()
    s2 = tset[TFrac]()

    s1.add(TFrac(1, 2))
    s1.add(TFrac(1, 3))

    s2.add(TFrac(1, 3))
    s2.add(TFrac(2, 3))

    result = s1.union(s2)
    assert result.size() == 3
```

```
    assert result.contains(TFrac(1, 2))
    assert result.contains(TFrac(1, 3))
    assert result.contains(TFrac(2, 3))

def test_fraction_intersection(self):
    """Тест пересечения множеств дробей"""
    s1 = tset[TFrac]()
    s2 = tset[TFrac]()

    s1.add(TFrac(1, 2))
    s1.add(TFrac(1, 3))
    s1.add(TFrac(1, 4))

    s2.add(TFrac(1, 3))
    s2.add(TFrac(1, 4))
    s2.add(TFrac(1, 5))

    result = s1.intersection(s2)
    assert result.size() == 2
    assert result.contains(TFrac(1, 3))
    assert result.contains(TFrac(1, 4))
    assert not result.contains(TFrac(1, 2))
    assert not result.contains(TFrac(1, 5))

def test_fraction_difference(self):
    """Тест вычитания множеств дробей"""
    s1 = tset[TFrac]()
    s2 = tset[TFrac]()

    s1.add(TFrac(1, 2))
```

```
        s1.add(TFrac(1, 3))
        s1.add(TFrac(1, 4))

        s2.add(TFrac(1, 3))
        s2.add(TFrac(1, 5))

        result = s1.difference(s2)
        assert result.size() == 2
        assert result.contains(TFrac(1, 2))
        assert result.contains(TFrac(1, 4))
        assert not result.contains(TFrac(1, 3))
        assert not result.contains(TFrac(1, 5))

    def test_fraction_negative_values(self):
        """Тест с отрицательными дробями"""
        s = tset[TFrac]()

        s.add(TFrac(-1, 2))
        s.add(TFrac(1, -2))  # Должно нормализоваться к -1/2
        s.add(TFrac(2, 4))   # 1/2

        # 1/(-2) должно нормализоваться до -1/2
        assert s.size() == 2
        assert s.contains(TFrac(-1, 2))
        assert s.contains(TFrac(1, 2))


def test_edge_cases():
    """Тест граничных случаев"""
    # Пустое множество
```

```
s = tset[int]()
assert s.is_empty()
assert s.size() == 0
assert str(s) == "{}"

# Добавление и удаление одного элемента
s.add(42)
assert not s.is_empty()
assert s.size() == 1
assert s.contains(42)

s.remove(42)
assert s.is_empty()

# Объединение с пустым множеством
s1 = tset[int]()
s2 = tset[int]()
s2.add(1)
s2.add(2)

result = s1.union(s2)
assert result.size() == 2
assert result == s2

# Вычитание пустого множества
result = s2.difference(s1)
assert result.size() == 2
assert result == s2

# Пересечение непересекающихся множеств
```

```
    s1 = tset[int]()
    s2 = tset[int]()
    s1.add(1)
    s1.add(2)
    s2.add(3)
    s2.add(4)

    result = s1.intersection(s2)
    assert result.is_empty()


if __name__ == "__main__":
    pytest.main([__file__, "-v"])
```