Практическая работа №5

Тема: Структуры данных «Стек» и «Очередь».

Цель работы: Изучить СД типа «дерево», научиться их программно реализовывать и использовать.

Стек — это коллекция, элементы которой получают по принципу «последний вошел, первый вышел» (Last-In-First-Out или LIFO). Это значит, что мы будем иметь доступ только к последнему добавленному элементу. Очереди очень похожи на стеки. Они также не дают доступа к произвольному элементу, но, в отличие от стека, элементы кладутся (enqueue) и забираются (dequeue) с разных концов. Такой метод называется «первый вошел, первый вышел» (First-In-First-Out или FIFO). То есть забирать элементы из очереди мы будем в том же порядке, что и клали. Как реальная очередь или конвейер.

Необходимо реализовать систему, изображенную на рисунке 1.

Рисунок 1 – Система для реализации

					АиСД.09.03.02.070000.ПР		
Изм.	*Лист*	*№ докум.*	*Подпись*	*Дат*			
Разраб.		*Клейменкин Д.*			Практическая работа №4 «Структуры данных «Стек» и «Очередь»»	*Лит.*	*Лист* 2 / *Листов* 9
Провер.		*Береза А.Н.*					
Реценз						*ИСОиП (филиал) ДГТУ в г.Шахты ИСТ-Tb21*	
Н. Контр.							
Утверд.							

В систему поступают запросы на выполнение задач трех типов – T_1, T_2, T_3, каждая для своего процессора. Поступающие запросы ставятся в очередь. Если в «голове» очереди находится задача Ti и процессор Pi свободен, то распределитель ставит задачу на выполнение в процессор Pi, а если процессор Pi занят, то распределитель отправляет задачу в стек и из очереди извлекается следующая задача. Задача из стека поступает в соответствующий ей свободный процессор только тогда, когда очередь пуста.

Реализуем класс задачи, который предоставляет доступ к полям данных задачи . Содержит поля двух типов: тип задания и время на выполнение задания, которые заполняются при инициализации класса. Для реализации данного класса, воспользуемся кодом, представленым ниже:

```
@dataclass()
class TaskData:
    time: int = None
    type: int = None

class Task:
    def __init__(self, task_type, task_time):
        self.current_task = TaskData()
        self.current_task.time = task_time
        self.current_task.type = task_type

    def __str__(self):
        return '[' + str(self.get_type()) + ',' + str(self.get_t
ime()) + ']'

    def get_time(self):
        return self.current_task.time

    def get_type(self):
        return self.current_task.type
```

Реализуем генератор задач. Класс инициализируется одной очередью очередями для трех типов задач. Публичный метод gen_task позволяет генерировать задачи, инициализируя класс Task случайными значениями из заданного диапазона и помещая его в очередь. Публичный метод get_task позволяет получить задачу для выполнения. Диаграмма деятельностей для этого метода представлена на рисунке 2. Публичный метод none_task возвращает истинное значение, если обе очереди пусты.

```
class TaskGenerator:
    def __init__(self):
        self.queue = MyQueue()

    def __str__(self):
        out = str(self.queue)
        return out + '\n'

    def gen_task(self):
        task = Task(rd.randint(1, 3), rd.randint(4, 8))
        self.queue.push(task)

    def get_task(self):
        if not self.queue.check_empty():
            task = self.queue.pop()
        else:
            task = None
        return task

    def none_task(self):
        return self.queue.check_empty()
```

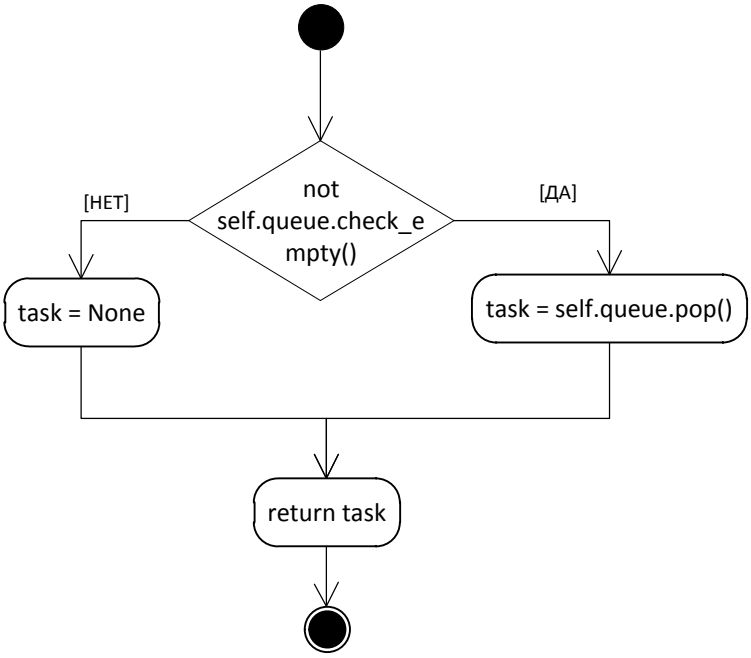

Рисунок 2 – Диаграмма деятельностей для метода get_task

Реализуем класс процессора. Данный класс инициализируется одним потоком класса данных Thread (хранит значения типа задачи, времени её выполнения и состояние простоя), соответствующих первому, второму и третьему процессорам и стеком для отброшенных задач. Публичный метод

add_task позволяет добавлять задания на поток. Его диаграмма деятельностей представлена на рисунке 3. Приватные методы run_task_t1 и run_task_t2 как бы выполняют задачу, уменьшая значение времени выполнения на единицу за шаг цикла. Публичный метод running эти приватные методы для имитации работы процессора. Публичные методы idle_thread и idle_proc для проверки состояния простоя хотя бы одного ядра в первом случае, и всего процессора во втором.

```
@dataclass()
class Thread:
    work_time: int = None
    task_type: int = None
    idle: bool = True

class Processor:
    def __init__(self):
        self.thread = Thread()
        self.wait = MyStack()

    def __str__(self):
        out = '|thread|type|time|idle |\n'
        out += '{:<9}{:<5}{:<5}{:<6}'.format('  1', str(self.thr
ead.task_type), str(self.thread.work_time),
                                                str(self.thread.idl
e))
        return out

    def add_task(self, task: Task):
        if task.get_type() == 1:
            if self.thread.idle:
                self.thread.task_type = task.get_type()
                self.thread.work_time = task.get_time()
                self.thread.idle = False
            elif self.thread.task_type == 2:
                denied_task = Task(self.thread.task_type, self.t
hread.work_time)
                self.thread.task_type = task.get_type()
                self.thread.work_time = task.get_time()
                self.wait.push(denied_task)
            elif self.thread.task_type == 3:
                denied_task = Task(self.thread.task_type, self.t
hread.work_time)
                self.thread.task_type = task.get_type()
                self.thread.work_time = task.get_time()
                self.wait.push(denied_task)
            else:
                self.wait.push(task)

    def __run_task(self):
```

```
            self.thread.work_time -= 1
            if self.thread.work_time <= 0:
                self.thread.idle = True
                self.thread.task_type = None
                self.thread.work_time = None

    def running(self):
        if not self.thread.idle:
            self.__run_task()
        else:
            self.thread.idle = True

    def idle_thread(self):
        return self.thread.idle

    def idle_proc(self):
        return self.thread.idle
```

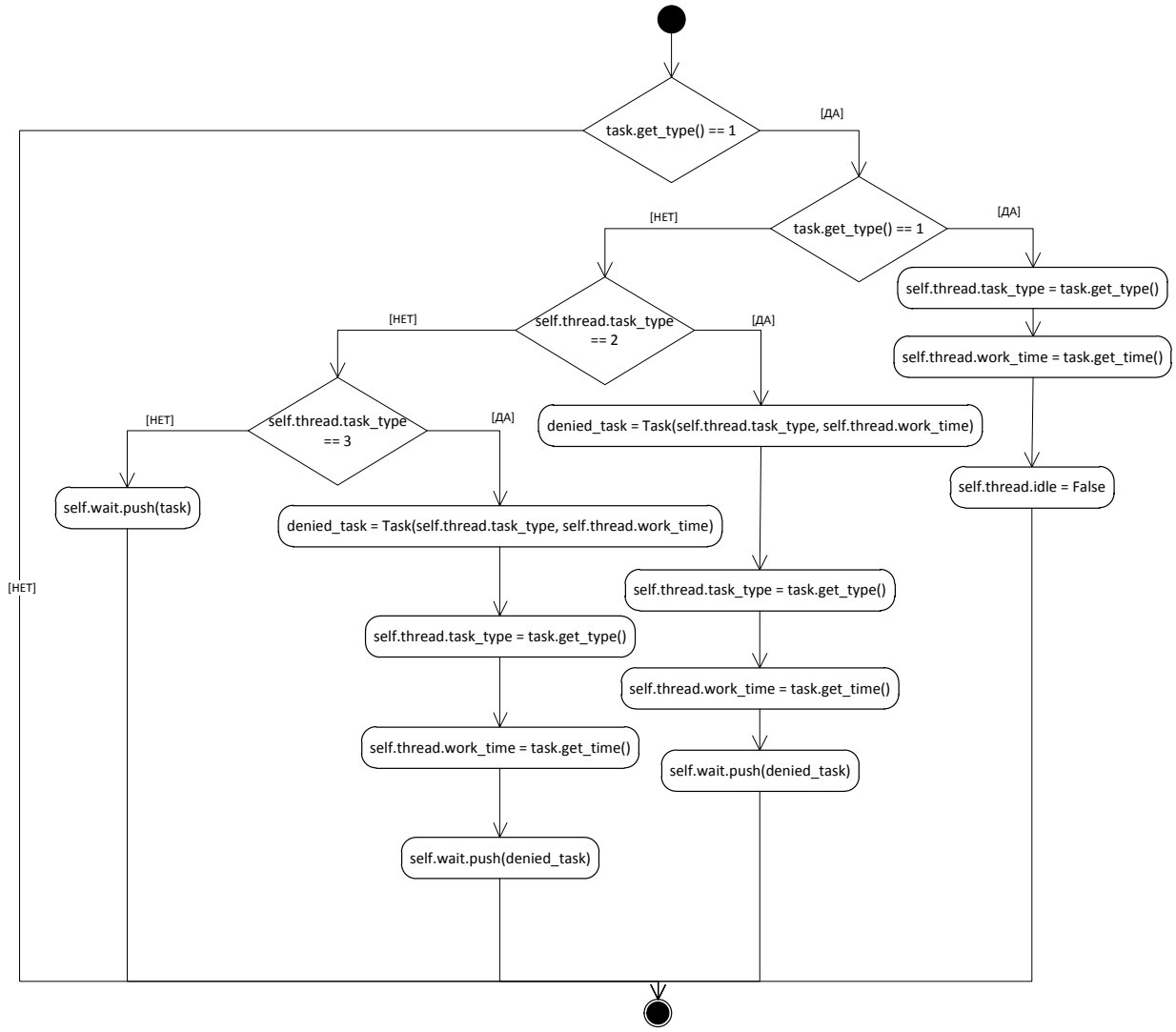

Рисунок 1 – Диаграмма деятельностей для метода add_task

Исходный код программы представлен ниже. Диаграмма деятельности для данного кода изображена на рисунке 4.

```
generator = TaskGenerator()
processor = Processor()

for i in range(10):
    generator.gen_task()

while True:
    task = generator.get_task()
    if processor.idle_thread():
        if not generator.none_task():
            processor.add_task(task)
        elif not processor.wait.check_empty():
            processor.add_task(processor.wait.pop())
    processor.running()
    print('Tasks\n', generator)
    print('Processor:\n', processor)
    print('Stack:', processor.wait)
    if generator.none_task() and processor.wait.check_empty() an
d processor.idle_proc():
        break
```

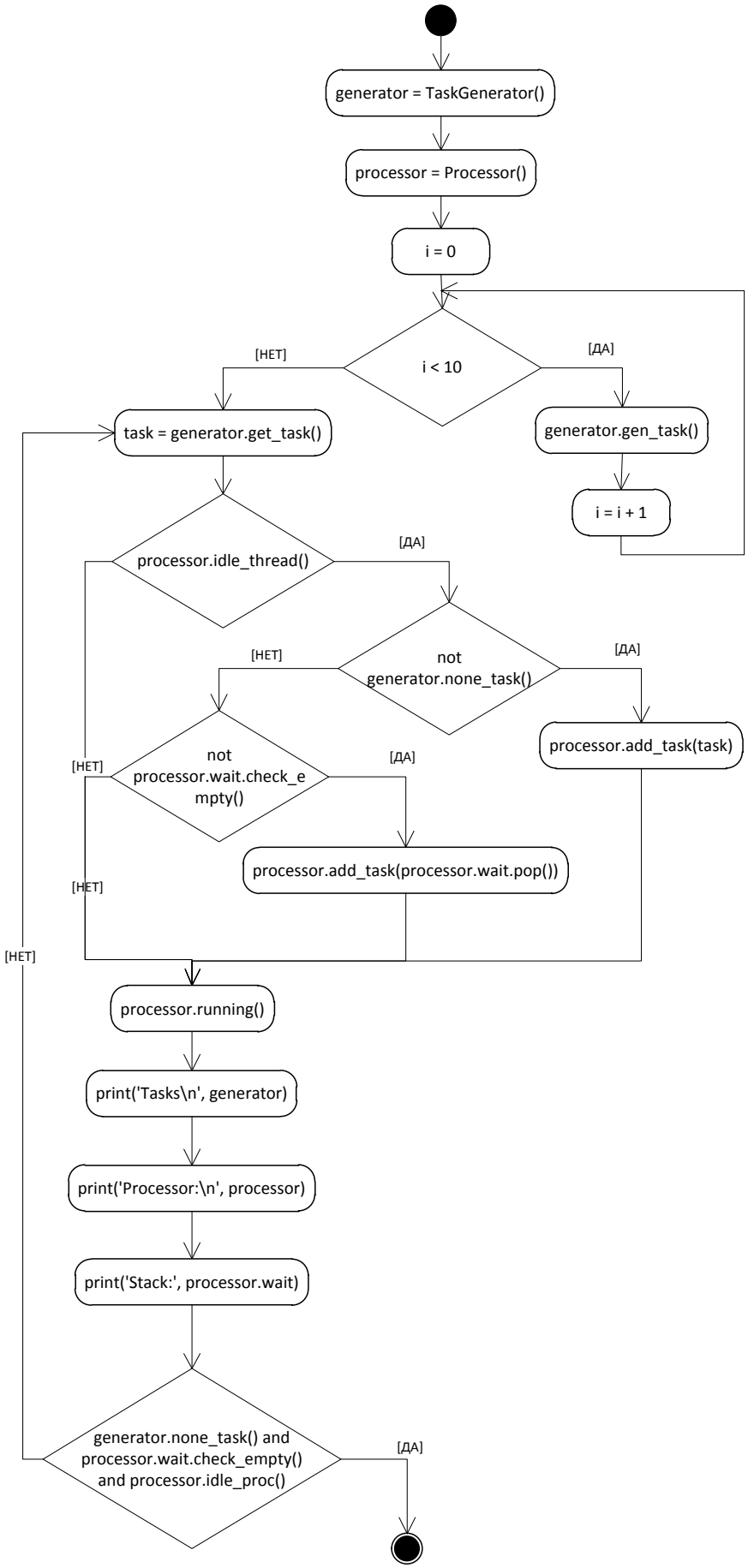

Рисунок 2 – Диаграмма деятельности для программы

Вывод. Мы изучили СД типа «Стек» и «Очередь», научились их программно реализовывать и использовать.