**Практическая работа №5.**

**Тема: «**Структуры данных «стек» и «очередь».

**Цель работы:** изучить СД «стек» и «очередь», научиться их программо реализовывать и использовать.

Реализуем генератор задач. Класс инициализируется двумя очередями для каждого типа задач. Публичный метод gen\_task позволяет генерировать задачи, инициализируя класс Task случайными значениями из заданного диапазона и помещая его в соответствующую 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.

Класс генератора задач.

class TaskGenerator:

def \_\_init\_\_(self):

self.queue1 = MyQueue()

self.queue2 = MyQueue()

def \_\_str\_\_(self):

out = str(self.queue1) + '\n' + str(self.queue2)

return out + '\n'

def gen\_task(self):

task = Task(rd.randint(1, 2), rd.randint(4, 8))

if task.get\_type() == 1:

self.queue1.push(task)

else:

self.queue2.push(task)

def get\_task(self):

queue = rd.randint(1, 2)

if queue == 1 and not self.queue1.check\_empty():

task = self.queue1.pop()

elif queue == 2 and not self.queue2.check\_empty():

task = self.queue2.pop()

elif queue == 1 and self.queue1.check\_empty():

task = self.queue2.pop()

elif queue == 2 and self.queue2.check\_empty():

task = self.queue1.pop()

else:

task = None

return task

def none\_task(self):

return self.queue1.check\_empty() and self.queue2.check\_empty()

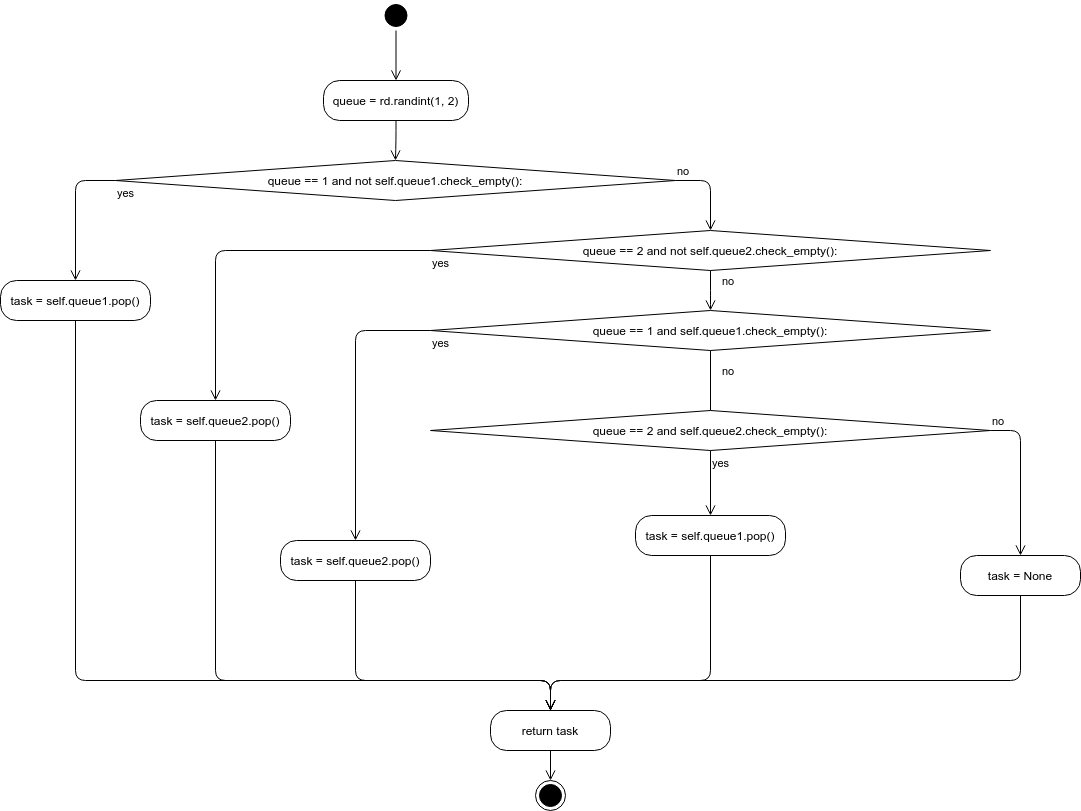


Рисунок 2. Диаграмма деятельностей для метода get\_task.

Реализуем класс процессора. Данный класс инициализируется двумя потоками класса данных Thread (хранит значения типа задачи, времени её выполнения и состояние простоя), соответствующих первому и второму процессору и стеком для отброшенных задач. Публичный метод add\_task позволяет добавлять задания на потоки. 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.

Класс процессора.

@dataclass()

class Thread:

work\_time: int = None

task\_type: int = None

idle: bool = True

class Processor:

def \_\_init\_\_(self):

self.thread1 = Thread()

self.thread2 = Thread()

self.wait = MyStack()

def \_\_str\_\_(self):

out = '|thread|type|time|idle |\n'

out += '{:<9}{:<5}{:<5}{:<6}'.format(' 1', str(self.thread1.task\_type), str(self.thread1.work\_time),

str(self.thread1.idle)) + '\n'

out += '{:<9}{:<5}{:<5}{:<6}'.format(' 2', str(self.thread2.task\_type), str(self.thread2.work\_time),

str(self.thread2.idle))

return out

def add\_task(self, task: Task):

if task.get\_type() == 1:

if self.thread1.idle:

self.thread1.task\_type = task.get\_type()

self.thread1.work\_time = task.get\_time()

self.thread1.idle = False

elif self.thread1.task\_type == 2:

denied\_task = Task(self.thread1.task\_type, self.thread1.work\_time)

self.thread1.task\_type = task.get\_type()

self.thread1.work\_time = task.get\_time()

self.wait.push(denied\_task)

else:

self.wait.push(task)

elif task.get\_type() == 2:

if self.thread2.idle:

self.thread2.task\_type = task.get\_type()

self.thread2.work\_time = task.get\_time()

self.thread2.idle = False

elif self.thread1.idle:

self.thread1.task\_type = task.get\_type()

self.thread1.work\_time = task.get\_time()

self.thread1.idle = False

else:

self.wait.push(task)

def \_\_run\_task\_t1(self):

self.thread1.work\_time -= 1

if self.thread1.work\_time <= 0:

self.thread1.idle = True

self.thread1.task\_type = None

self.thread1.work\_time = None

def \_\_run\_task\_t2(self):

self.thread2.work\_time -= 1

if self.thread2.work\_time <= 0:

self.thread2.idle = True

self.thread2.task\_type = None

self.thread2.work\_time = None

def running(self):

if not self.thread1.idle:

self.\_\_run\_task\_t1()

else:

self.thread1.idle = True

if not self.thread2.idle:

self.\_\_run\_task\_t2()

else:

self.thread2.idle = True

def idle\_thread(self):

return self.thread1.idle or self.thread2.idle

def idle\_proc(self):

return self.thread1.idle and self.thread2.idle

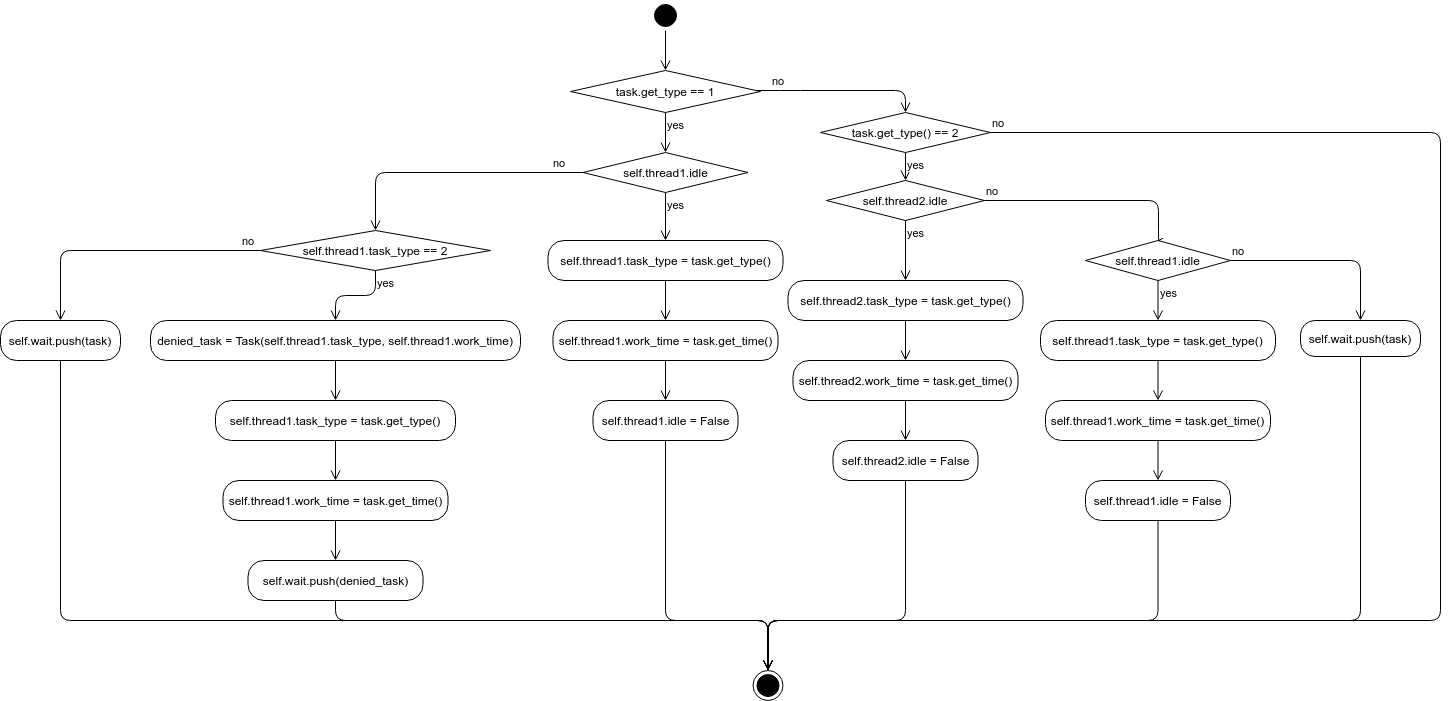


Рисунок 3. Диаграмма деятельностей для метода add\_task.

Исходный код программы представлен ниже. Логика работы приведена на диаграмме деятельностей (Рисунок 4).

Исходный код программы.

from Stack\_and\_Queue.processor import Processor

from Stack\_and\_Queue.task import TaskGenerator

generator = TaskGenerator()

processor = Processor()

for i in range(50):

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() and processor.idle\_proc():

break

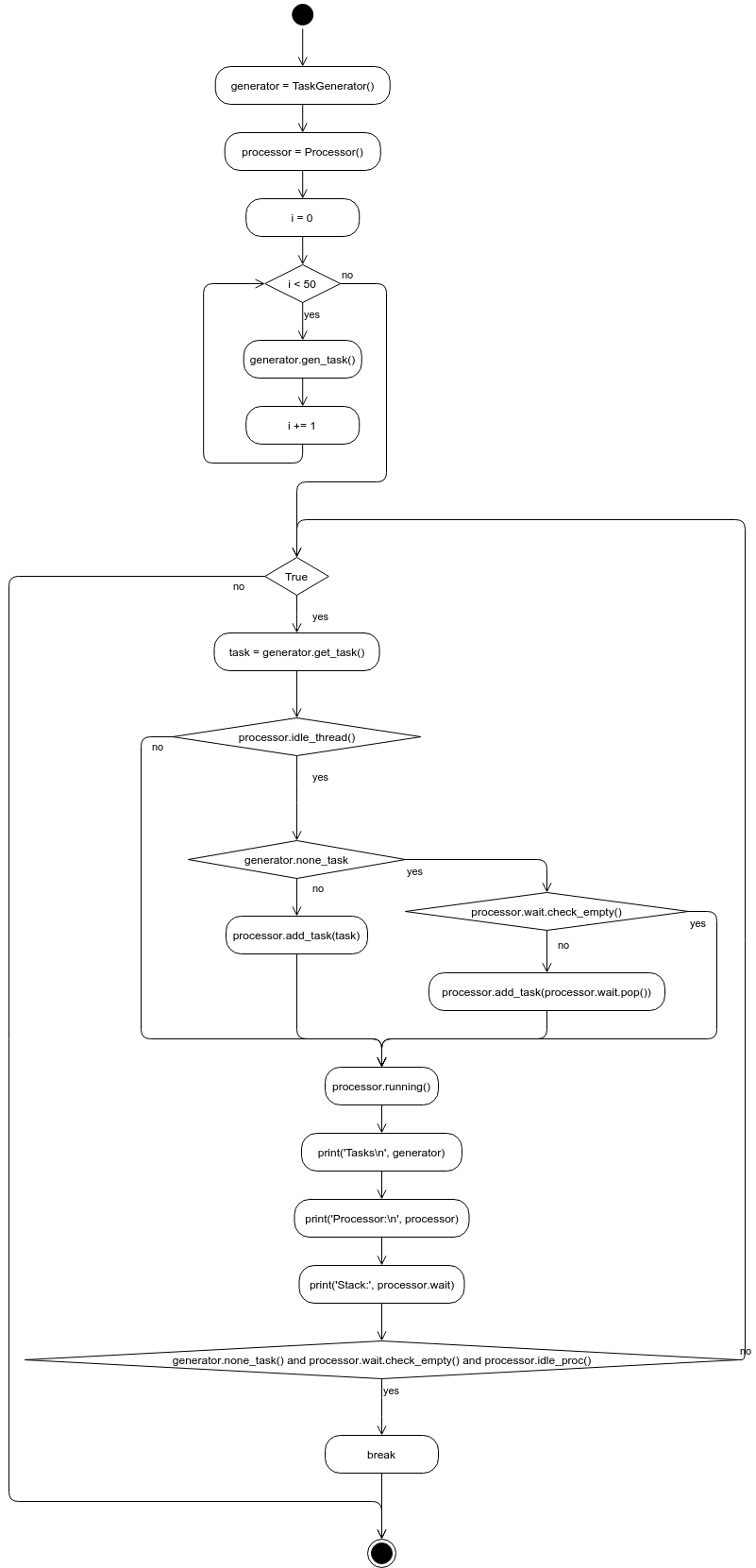


Рисунок 4. Диаграмма деятельностей для программы.

**Вывод:** в ходе выполнения данной практической работы мы изучили структуры данных «стек» и «очередь» их реализацию и использование.