

САНКТ-ПЕТЕРБУРГСКИЙ НАЦИОНАЛЬНЫЙ
ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ
ИНФОРМАЦИОННЫХ ТЕХНОЛОГИЙ, МЕХАНИКИ И ОПТИКИ
ФАКУЛЬТЕТ ИНФОКОММУНИКАЦИОННЫХ ТЕХНОЛОГИЙ

Отчет по лабораторной работе №5
по курсу «Алгоритмы и структуры данных»
Тема: “Деревья. Пирамида, пирамидальная сортировка.
Очередь с приоритетом”
Вариант 20

Выполнила:
Толстухина К.А.
К3139

Проверил:
Афанасьев А.В.

Санкт-Петербург
2024 год

Задачи по варианту

Задача №5. Планировщик заданий

Текст задачи

В этой задаче вы создадите программу, которая параллельно обрабатывает список заданий. У вас есть программа, которая распараллеливается и использует n независимых потоков для обработки заданного списка t заданий. Потоки берут задания в том порядке, в котором они указаны во входных данных. Если есть свободный поток, он немедленно берет следующее задание из списка. Если поток начал обработку задания, он не прерывается и не останавливается, пока не завершит обработку задания. Если несколько потоков одновременно пытаются взять задания из списка, поток с меньшим индексом берет задание. Для каждого задания вы точно знаете, сколько времени потребуется любому потоку, чтобы обработать это задание, и это время одинаково для всех потоков. Вам необходимо определить для каждого задания, какой поток будет его обрабатывать и когда он начнет обработку.

Листинг кода

```
import tracemalloc
import time
from lab5.utils import *

t_start = time.perf_counter()
tracemalloc.start()

def parallel_task_processing(count_threads: int,
count_task: int, times: list[int]) -> list[tuple]:
    """
    функция для работы с планировщиком заданий
    :param count_threads: int
    :param count_task: int
    :param times: list[int]
    :return: list[tuple]
    """
```

```

        free_threads = [(0, i) for i in
range(count_threads)]
    result = []
    for t in range(count_task):
        task_time = times[t]
        free_threads.sort()
        free_time, thread_id = free_threads.pop(0)
        start_time = free_time
        end_time = start_time + task_time
        free_threads.append((end_time, thread_id))
        result.append((thread_id, start_time))

    return result

CURRENT_DIR =
os.path.dirname(os.path.abspath(__file__))
TXTF_DIR =
os.path.join(os.path.dirname(CURRENT_DIR), "txtf")
INPUT_PATH = os.path.join(TXTF_DIR, "input.txt")
OUTPUT_PATH = os.path.join(TXTF_DIR, "output.txt")

if __name__ == "__main__":
    lines = open_file(INPUT_PATH)
    num, count_task = map(int, lines[0].split())
    times = list(map(int, lines[1].split()))
    if 1 <= int(num) <= 10 ** 5 and 1 <=
int(count_task) <= 10 ** 5:
        result = parallel_task_processing(num,
count_task, times)
        res = []
        for thread_id, start_time in result:
            res.append(f'{thread_id} {start_time}')
        write_file('\n'.join(res), OUTPUT_PATH)
    else:
        print("Введите корректные данные")

```

```

        print("Время работы: %s секунд" %
(time.perf_counter() - t_start))
        print("Затрачено памяти:",
tracemalloc.get_traced_memory()[1], "байт")
        tracemalloc.stop()

```

Текстовое объяснение

Подключаю две библиотеки для отслеживания памяти и времени. Затем Функция для работы с планировщиком заданий, создаю кучу (приоритетную очередь) для хранения информации о свободных потоках. Каждый поток можно представить как кортеж (время_освобождения, индекс_потока). Далее я извлекаю поток с минимальным временем освобождения, записываю время начала выполнения задания и обновляю время освобождения потока. В конце возвращаю поток обратно в кучу с обновленным временем освобождения. И так до тех пор, пока не обработаются все задачи. Далее вне функции я прописываю пути к необходимым директориям/файлам. И в основной части, я считываю данные, проверяю их валидность, вызываю функцию и вывожу время и память.

Пример работы(скрин файлов)

input.txt		:	output.txt	
1	2 5	✓	1	0 0
2	1 2 3 4 5		2	1 0
	💡		3	0 1
			4	1 2
			5	0 4

	Время	Память
пример	0.001450499999918975 секунд	14768 байт

Вывод: был изучен и реализован алгоритм работы с кучей

Задача №7. Снова сортировка

Текст задачи

Напишите программу пирамидальной сортировки на Python для последовательности в убывающем порядке.

Листинг кода

```
import tracemalloc
import time
from lab5.utils import *

t_start = time.perf_counter()
tracemalloc.start()

def heapify(arr: list, n: int, i: int) -> None:
    """
    восстанавливает свойство кучи
    :param arr: list
    :param n: int
    :param i: int
    :return: None
    """
    largest = i
    left = 2 * i + 1
    right = 2 * i + 2
    if left < n and arr[left] > arr[largest]:
        largest = left
    if right < n and arr[right] > arr[largest]:
        largest = right
    if largest != i:
        arr[i], arr[largest] = arr[largest], arr[i]
```

```

        heapify(arr, n, largest)

def heap_sort(arr: list) -> list:
    """
    пирамидная сортировка массива
    :param arr: list
    :return: list
    """
    n = len(arr)
    for i in range(n // 2 - 1, -1, -1):
        heapify(arr, n, i)
    for i in range(n - 1, 0, -1):
        arr[i], arr[0] = arr[0], arr[i]
        heapify(arr, i, 0)
    return arr

CURRENT_DIR =
os.path.dirname(os.path.abspath(__file__))
TXTF_DIR =
os.path.join(os.path.dirname(CURRENT_DIR), "txtf")
INPUT_PATH = os.path.join(TXTF_DIR, "input.txt")
OUTPUT_PATH = os.path.join(TXTF_DIR, "output.txt")

if __name__ == "__main__":
    lines = open_file(INPUT_PATH)
    num = lines[0]
    arr = list(map(int, lines[1].split()))
    if 1 <= int(num) <= 10 ** 5:
        result = heap_sort(arr)
        write_file(' '.join(map(str, result)),
OUTPUT_PATH)
    else:
        print("Введите корректные данные")

```

```

print("Время работы: %s секунд" %
(time.perf_counter() - t_start))
print("Затрачено памяти:",
tracemalloc.get_traced_memory()[1], "байт")
tracemalloc.stop()

```

Текстовое объяснение

Работа с памятью, временем и файлами аналогична предыдущей задаче. Функция преобразовывает массива в кучу (точнее, в максимальную кучу). Это делается для того, чтобы извлекать наибольшие элементы с каждого шага. Далее на каждом шаге я извлекаю максимальный элемент (корень кучи), помещаю его в конец массива и восстанавливаю кучу для оставшейся части массива. Повторяем процесс, пока весь массив не будет отсортирован.

Пример работы(скрин файлов)

input.txt ×		:	output.txt ×	
1	5	✓	1	2 20 21 23 24
2	20 2 23 21 24			

	Время	Память
ПРИМЕР	0.0011173330058227293 секунд	14770 байт

Вывод: я познакомилась с пирамидальной сортировкой

Дополнительные задачи

Задача №6. Очередь с приоритетом

Текст задачи

Реализуйте очередь с приоритетами. Ваша очередь должна поддерживать следующие операции: добавить элемент, извлечь минимальный элемент, уменьшить элемент, добавленный во время одной из операций.

Листинг кода

```
import tracemalloc
import time
from lab5.utils import *

t_start = time.perf_counter()
tracemalloc.start()

def heapify_up(heap: list, index: int) -> None:
    """
    Процедура для восстановления свойств кучи (вверх)
    :param heap: list
    :param index: int
    :return: None
    """
    while index > 0:
        parent = (index - 1) // 2
        if heap[index] < heap[parent]:
            heap[index], heap[parent] = heap[parent],
            heap[index]
            index = parent
        else:
            break

def heapify_down(heap: list, index: int) -> None:
    """
    Процедура для восстановления свойств кучи (вниз)
```



```

:param heap: list
:param index: int
:return: None
"""
size = len(heap)
while index < size:
    left = 2 * index + 1
    right = 2 * index + 2
    smallest = index

    if left < size and heap[left] <
heap[smallest]:
        smallest = left
    if right < size and heap[right] <
heap[smallest]:
        smallest = right
    if smallest != index:
        heap[index], heap[smallest] =
heap[smallest], heap[index]
        index = smallest
    else:
        break

def add(heap: list, value: int) -> None:
    """
    Добавление элемента в кучу
    :param heap: list
    :param value: int
    :return: None
    """
    heap.append(value)
    heapify_up(heap, len(heap) - 1)

def extract_min(heap: list, removed: set):
    """

```

Извлечение минимального элемента из кучи

```
:param heap: list  
:param removed: set  
"""  
if len(heap) == 0:  
    return "*"   
min_elem = heap[0]  
heap[0] = heap[-1]  
heap.pop()  
heapify_down(heap, 0)  
  
while min_elem in removed:  
    if len(heap) == 0:  
        return "*"   
    min_elem = heap[0]  
    heap[0] = heap[-1]  
    heap.pop()  
    heapify_down(heap, 0)  
return min_elem
```

```
def decrease(heap: list, insertion_order: list,  
index: int, new_value: int, removed: set) -> None:
```

```
    """  
    Уменьшение элемента в куче  
    :param heap: list  
    :param insertion_order: list  
    :param index: int  
    :param new_value: int  
    :param removed: set  
    :return: None  
    """  
    old_value = insertion_order[index]  
    if old_value > new_value:  
        removed.add(old_value)  
        add(heap, new_value)  
        insertion_order[index] = new_value
```

```

def process_operations(n, operations):
    heap = []
    insertion_order = []
    removed = set()
    results = []

    for i in range(n):
        operation = operations[i].split()
        if operation[0] == "A":
            add(heap, int(operation[1]))
            insertion_order.append(int(operation[1]))
        elif operation[0] == "X":
            result = extract_min(heap, removed)
            results.append(str(result))
        elif operation[0] == "D":
            idx = int(operation[1]) - 1
            new_value = int(operation[2])
            decrease(heap, insertion_order, idx,
new_value, removed)

    return results

CURRENT_DIR =
os.path.dirname(os.path.abspath(__file__))
TXTF_DIR =
os.path.join(os.path.dirname(CURRENT_DIR), "txtf")
INPUT_PATH = os.path.join(TXTF_DIR, "input.txt")
OUTPUT_PATH = os.path.join(TXTF_DIR, "output.txt")

if __name__ == "__main__":
    lines = open_file(INPUT_PATH)
    n = int(lines[0])
    operations = [i.strip() for i in lines[1:]]
    if 1 <= n <= 10 ** 6:

```

```
        result = results = process_operations(n,
operations)
        write_file('\n'.join(map(str, result)),
OUTPUT_PATH)
    else:
        print("Введите корректные данные")

    print("Время работы: %s секунд" %
(time.perf_counter() - t_start))
    print("Затрачено памяти:",
tracemalloc.get_traced_memory()[1], "байт")
    tracemalloc.stop()
```

Текстовое объяснение

`heapify_up()` - восстанавливает свойство кучи, начиная с индекса `index` и двигаясь вверх. `heapify_down(heap, index)` — восстанавливает свойство кучи, начиная с индекса `index` и двигаясь вниз. `add(heap, value)` — добавляет новый элемент в кучу и восстанавливает её свойства с помощью функции `heapify_up`. `extract_min(heap, removed)` — извлекает минимальный элемент из кучи, проверяя, не был ли он ранее удалён. Если элемент был удалён, извлекается следующий минимальный элемент. `decrease(heap, insertion_order, index, new_value, removed)` — уменьшает значение элемента, добавленного на `index`-ой операции, и восстанавливает кучу.

Тесты(скрины файлов)

≡ input.txt ×		⋮	≡ output.txt ×	
1	8	✓	1	2
2	A 3		2	1
3	A 4		3	3
4	A 2		4	*
5	X			
6	D 2 1			
7	X			
8	X			
9	X			

	время	память
тест 1	0.0008551250066375 36 секунд	16643 байт

Вывод: была реализована очередь с приоритетом

Задача №1. Куча ли?

Текст задачи

Структуру данных «куча», или, более конкретно, «неубывающая пирамида», можно реализовать на основе массива.

Дан массив целых чисел. Определите, является ли он неубывающей пирамидой.

Листинг кода

```
import tracemalloc
import time
from lab5.utils import *

t_start = time.perf_counter()
tracemalloc.start()

def is_min_heap(arr: list) -> str:
    """
    функция для проверки, является ли массив
    неубывающей пирамидой
    :param arr: list
    :return: str
    """
    n = len(arr)
    for i in range(n):
        left = 2 * i + 1
        right = 2 * i + 2
        if left < n and arr[i] > arr[left]:
            return "NO"
        if right < n and arr[i] > arr[right]:
            return "NO"
    return "YES"

CURRENT_DIR =
os.path.dirname(os.path.abspath(__file__))
TXTF_DIR =
os.path.join(os.path.dirname(CURRENT_DIR), "txtf")
INPUT_PATH = os.path.join(TXTF_DIR, "input.txt")
OUTPUT_PATH = os.path.join(TXTF_DIR, "output.txt")

if __name__ == "__main__":
    lines = open_file(INPUT_PATH)
    num = lines[0]
```

```

arr = list(map(int, lines[1].split()))
if 1 <= int(num) <= 10 ** 6:
    result = is_min_heap(arr)
    write_file(result + '\n', OUTPUT_PATH)
else:
    print("Введите корректные данные")

    print("Время работы: %s секунд" %
(time.perf_counter() - t_start))
    print("Затрачено памяти:",
tracemalloc.get_traced_memory()[1], "байт")
    tracemalloc.stop()

```

Объяснение

Внутри функции проверяют удовлетворяет ли массив условиям, чтобы считаться кучей.

Скрины (файлов)

input.txt		task1.py	output.txt
1	5	1	YES
2	1 3 2 5 4	2	

	время	память
тест 1	0.0011173330058227 293 секунд	14606 байт

Вывод по всей работе: я познакомилась с пирамидой и пирамидальной сортировкой.