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 приоритетами.

Вариант 14

Выполнил: Мурашов Н.А.

К3141

Проверил: Афанасьев А.В.

Санкт-Петербург 2024 г.

# Задачи по варианту

# Задача 3:

# Текст задачи.

# 

# Листинг кода.

# Lab5/Task3/src/NetworkPacketsProcessor.py  
  
*"""Модуль для обработки сетевых пакетов с ограничением буфера."""*class NetworkPacketsProcessor:  
 *"""  
 Класс для вычисления времени обработки пакетов в сети.  
 """* @staticmethod  
 def network\_packets(buffer\_size, packets):  
 *"""  
 Рассчитывает стартовое время обработки для каждого пакета.  
  
 :param buffer\_size: Размер буфера.  
 :param packets: Список пакетов в формате (arrival\_time, processing\_time).  
 :return: Список времен начала обработки, либо -1 если пакет отброшен.  
 """* buffer = []  
 start\_times = []  
  
 for arrival\_time, processing\_time in packets:  
 while buffer and buffer[0] <= arrival\_time:  
 buffer.pop(0)  
  
 if len(buffer) >= buffer\_size:  
 start\_times.append('-1')  
 else:  
 if not buffer:  
 start\_time = arrival\_time  
 else:  
 start\_time = buffer[-1]  
 start\_times.append(str(start\_time))  
 buffer.append(start\_time + processing\_time)  
 return start\_times

Текстовое объяснение решения.

Для обработки пакетов с фиксированным буфером используется очередь. Мы проверяем, вмещается ли новый пакет в буфер, и, если нет, он отклоняется. Время начала обработки вычисляется на основе текущего состояния буфера и времени прибытия.

|  |  |  |
| --- | --- | --- |
|  | Время выполнения | Затраты памяти |
| Нижняя граница диапазона значений входных данных из текста задачи | 0.004003 сек | 0.30МБ |
| Пример из задачи | 0.05832 сек | 0.52 МБ |
| Пример из задачи | 0.09345 сек | 1.45 МБ |
| Верхняя граница диапазона значений входных данных из текста задачи | 0.42455 сек | 1.99 МБ |

Вывод по задаче.

Реализованный алгоритм продемонстрировал работу очереди в условиях ограниченных ресурсов. Это применимо для оптимизации сетевых приложений.

## Задача 6:

# Текст задачи.

# 

# Листинг кода.

*"""Модуль для обработки операций с приоритетной очередью."""*import heapq  
  
class PriorityQueueProcessor:  
 *"""  
 Класс для обработки операций с приоритетной очередью:  
 - A x: Добавить элемент x.  
 - X: Извлечь минимальный элемент.  
 - D x y: Уменьшить значение элемента, добавленного в строке x+1, до y.  
 """* def \_\_init\_\_(self):  
 self.heap = [] # Мини-куча  
 self.element\_map = {} # id → значение  
 self.id\_map = {} # строка → id элемента  
 self.removed = set() # множество удаленных элементов  
 self.current\_id = 0 # уникальный id для каждого добавленного элемента  
  
 def process\_operations(self, operations):  
 *"""  
 Обрабатывает список операций над приоритетной очередью.  
  
 :param operations: Список строк, каждая строка - операция.  
 :return: Список результатов для операций X.  
 """* results = []  
 for i, operation in enumerate(operations):  
 parts = operation.split()  
 cmd = parts[0]  
 if cmd == "A":  
 # Добавляем элемент  
 x = int(parts[1])  
 heapq.heappush(self.heap, (x, self.current\_id))  
 self.element\_map[self.current\_id] = x  
 self.id\_map[i+1] = self.current\_id  
 self.current\_id += 1  
  
 elif cmd == "X":  
 # Извлекаем минимальный  
 self.\_clean\_heap()  
 if self.heap:  
 \_, elem\_id = heapq.heappop(self.heap)  
 results.append(str(self.element\_map[elem\_id]))  
 self.removed.add(elem\_id)  
 else:  
 results.append("\*")  
  
 elif cmd == "D":  
 # Уменьшаем значение элемента  
 x = int(parts[1]) + 1  
 y = int(parts[2])  
 elem\_id = self.id\_map[x]  
 self.element\_map[elem\_id] = y  
 heapq.heappush(self.heap, (y, elem\_id))  
  
 return results  
  
 def \_clean\_heap(self):  
 *"""Удаляет из кучи элементы, которые уже были удалены."""* while self.heap and self.heap[0][1] in self.removed:  
 heapq.heappop(self.heap)

Текстовое объяснение решения.

Реализована минимальная куча для обработки операций добавления, извлечения минимального элемента и уменьшения значения. Операции оптимизированы с помощью библиотеки heapq, а корректность значений поддерживается через дополнительные структуры данных.

|  |  |  |
| --- | --- | --- |
|  | Время выполнения | Затраты памяти |
| Нижняя граница диапазона значений входных данных из текста задачи | 0.003465 сек | 0.35 МБ |
| Пример из задачи | 0.03958 сек | 0.56 МБ |
| Пример из задачи | 0.10345 сек | 1.59 МБ |
| Верхняя граница диапазона значений входных данных из текста задачи | 0.4124 сек | 1.89 МБ |

Вывод по задаче.  
Алгоритм приоритетной очереди позволяет эффективно обрабатывать задачи, требующие работы с отсортированными элементами, например, в планировщиках задач или маршрутизации.

# Дополнительные задачи

## Задача 2:

# Текст задачи.

# 

# Листинг кода.

*"""Модуль для вычисления высоты дерева по списку родителей."""*class TreeHeightCalculator:  
 *"""  
 Класс для вычисления высоты дерева.  
 Считается, что -1 в списке родителей обозначает корень дерева.  
 """* def tree\_height(self, n, parents):  
 *"""  
 Вычисляет высоту дерева.  
  
 :param n: Количество узлов.  
 :param parents: Список родителей для каждого узла (индекс = узел, значение = родитель).  
 :return: Высота дерева.  
 """* tree = [[] for \_ in range(n)]  
 root = -1  
  
 for i in range(n):  
 if parents[i] == -1:  
 root = i  
 else:  
 tree[parents[i]].append(i)  
  
 def height(node):  
 if not tree[node]:  
 return 1  
 max\_h = 0  
 for child in tree[node]:  
 max\_h = max(max\_h, height(child))  
 return max\_h + 1  
  
 return height(root)

Текстовое объяснение решения.

Алгоритм использует список родителей, где -1 обозначает корень дерева. Мы строим дерево как граф с помощью списков смежности. Высота определяется рекурсивным обходом дерева от корня, вычисляя максимальную глубину дочерних узлов.

|  |  |  |
| --- | --- | --- |
|  | Время выполнения | Затраты памяти |
| Нижняя граница диапазона значений входных данных из текста задачи | 0.01134 сек | 1.67 МБ |
| Пример из задачи | 0.02689 сек | 1.77 МБ |
| Пример из задачи | 0.07581 сек | 1.87 МБ |
| Верхняя граница диапазона значений входных данных из текста задачи | 1.02345 сек | 1.90 МБ |

Вывод по задаче.  
Задача продемонстрировала способ эффективного представления дерева в памяти и вычисления его высоты, что полезно в алгоритмах обработки иерархических структур данных.

## Задача 7:

# Текст задачи.

# 

# Листинг кода.

# Lab5/Task7/src/HeapSorter.py  
  
*"""Модуль для пирамидальной сортировки массива."""*class HeapSorter:  
 *"""  
 Класс для выполнения пирамидальной (heap) сортировки массива.  
 """* def heapify(self, arr, n, i):  
 *"""  
 Преобразование поддерева с корнем в индексе i в max-heap.  
  
 :param arr: Массив.  
 :param n: Размер участка массива.  
 :param i: Индекс корня поддерева.  
 """* 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]  
 self.heapify(arr, n, largest)  
  
 def heapsort(self, arr):  
 *"""  
 Пирамидальная сортировка массива arr.  
  
 :param arr: Массив для сортировки.  
 """* n = len(arr)  
  
 # Построение max-heap  
 for i in range(n // 2 - 1, -1, -1):  
 self.heapify(arr, n, i)  
  
 # Извлечение элементов из кучи  
 for i in range(n - 1, 0, -1):  
 arr[0], arr[i] = arr[i], arr[0]  
 self.heapify(arr, i, 0)

Текстовое объяснение решения.

Алгоритм использует двухэтапный подход: построение max-heap из массива и извлечение элементов для сортировки. Построение кучи требует O(n) времени, а сортировка выполняется за O(nlogn).

|  |  |  |
| --- | --- | --- |
|  | Время выполнения | Затраты памяти |
| Нижняя граница диапазона значений | 0.00782 сек | 0.48 МБ |
| входных данных из текста задачи |  |  |
| Пример из задачи | 0.06548 сек | 1.48 МБ |
| Пример из задачи | 0.25642 сек | 1.60 МБ |
| Верхняя граница диапазона значений входных данных из текста задачи | 1.10294 сек | 1.91 МБ |

Вывод по задаче.  
Heapsort — это эффективный алгоритм сортировки без использования дополнительной памяти. Он подходит для задач, где стабильность сортировки не важна.

# Вывод

В данной лабораторной работе изучены и реализованы ключевые структуры данных, такие как деревья, очереди и кучи, а также эффективный алгоритм сортировки (heapsort). Каждый из алгоритмов демонстрирует принципы работы с данными в различных сценариях, включая управление памятью, оптимизацию времени выполнения и применение рекурсии. Эти знания закладывают базу для дальнейшего изучения алгоритмов и их практического применения в реальных задачах.