Міністерство освіти і науки України

Державний університет «Житомирська політехніка»

Факультет інформаційно-комп’ютерних технологій

Кафедра комп’ютерних наук

Звіт

з контрольної роботи №2

з дисципліни «Алгоритми та структури даних»

Виконав студент 1-го курсу, групи ВТ-23-1

спеціальності 121 «Інженерія програмного забезпечення»

Варіант: 4

Нагорний Т. Г.

Керівник Петросян Р. В.

Житомир – 2024

Зміст

Швидке сортування

Хід роботи

Опис: швидке сортування - це рекурсивний алгоритм сортування, який працює на принципі "розділяй і володарюй". Алгоритм обирає один елемент масиву як опорний і розташовує всі елементи менше опорного наліво, а всі елементи більше опорного - справа. Після цього опорний елемент вже знаходиться на своєму правильному місці. Алгоритм рекурсивно застосовується до лівої та правої частини масиву, що знаходяться відносно опорного елемента, доки весь масив не буде відсортований.

Використані структури даних: рекурсія, масив.

Лістинг програми:

```
def quicksort(arr):
    if len(arr) <= 1:
        return arr
    pivot = arr[len(arr) // 2]
    left = [x for x in arr if x < pivot]
    middle = [x for x in arr if x == pivot]
    right = [x for x in arr if x > pivot]
    return quicksort(left) + middle + quicksort(right)


test_data = [5, 3, 8, 6, 2, 7, 1, 4]
print("Не відсортований масив:", test_data)
print("Відсортований масив:", quicksort(test_data))
```

Тестові данні: [5, 3, 8, 6, 2, 7, 1, 4]

```
Не відсортований масив: [5, 3, 8, 6, 2, 7, 1, 4]
Відсортований масив: [1, 2, 3, 4, 5, 6, 7, 8]
```

Результат виконання програми

Сортування за розрядами

Хід роботи

Опис: сортування за розрядами - це алгоритм сортування, який сортує числа за їхніми розрядами, починаючи з найменш значущого розряду і закінчуючи найбільш значущим. Спочатку числа розподіляються по корзинам на основі значення їхніх розрядів. Потім числа збираються разом у відсортованому порядку. Алгоритм повторює цей процес для кожного розряду, поки весь масив не буде відсортований за всіма розрядами.

Використані структури даних: масив.

Лістинг програми:

```python
def counting_sort(arr, exp):
    n = len(arr)
    output = [0] * n
    count = [0] * 10

    for i in range(n):
        index = arr[i] // exp
        count[index % 10] += 1

    for i in range(1, 10):
        count[i] += count[i - 1]

    i = n - 1
    while i >= 0:
        index = arr[i] // exp
        output[count[index % 10] - 1] = arr[i]
        count[index % 10] -= 1
        i -= 1

    for i in range(n):
        arr[i] = output[i]

def radix_sort(arr):
    max_num = max(arr)
    exp = 1
    while max_num // exp > 0:
        counting_sort(arr, exp)
        exp *= 10

test_data = [170, 45, 75, 90, 802, 24, 2, 66]
print("Не відсортований масив:", test_data)
radix_sort(test_data)
print("Відсортований масив:", test_data)
```

Тестові данні: [170, 45, 75, 90, 802, 24, 2, 66]

```
Не відсортований масив: [170, 45, 75, 90, 802, 24, 2, 66]
Відсортований масив: [2, 24, 45, 66, 75, 90, 170, 802]
```

Результат виконання програми

Хеш-таблиця з відкритою адресацією

Хід роботи

Опис: хеш-таблиця з відкритою адресацією - це метод збереження інформації у вигляді пар "ключ-значення", де доступ до значень відбувається через визначений хеш-код ключа. У випадку колізій (коли два ключі відображаються на один і той же індекс хеш-таблиці), використовується відкрита адресація, коли елемент зберігається в наступній вільній комірці. У мові програмування Python тип даних dict реалізований саме за допомогою використання хеш-таблиць.

Використані структури даних: хеш-таблиця, масив.

Лістинг програми:

```python
class HashTable:
    def __init__(self, size):
        self.size = size
        self.hash_table = [None] * self.size

    def hash_function(self, key):
        return key % self.size

    def insert(self, key, value):
        index = self.hash_function(key)
        while self.hash_table[index] is not None:
            index = (index + 1) % self.size
        self.hash_table[index] = (key, value)

    def search(self, key):
        index = self.hash_function(key)
        while self.hash_table[index] is not None and self.hash_table[index][0] !=
key:
            index = (index + 1) % self.size
        if self.hash_table[index] is None:
            return None
        return self.hash_table[index][1]


hash_table = HashTable(3)  # 3 - кількість елементів у хеш-таблиці
hash_table.insert(5, "apple")
hash_table.insert(15, "banana")
hash_table.insert(25, "cherry")

print("Значення для ключа 5:", hash_table.search(5))
print("Значення для ключа 15:", hash_table.search(15))
print("Значення для ключа 25:", hash_table.search(25))
```

Тестові данні:

```python
hash_table = HashTable(3)  # 3 - кількість елементів у хеш-таблиці
hash_table.insert( key: 5,  value: "apple")
hash_table.insert( key: 15,  value: "banana")
hash_table.insert( key: 25,  value: "cherry")
```

```
Значення для ключа 5: apple
Значення для ключа 15: banana
Значення для ключа 25: cherry
```

Результат виконання програми

Алгоритм Дейкстри

Хід роботи

Опис: алгоритм Дейкстри - це алгоритм пошуку найкоротших шляхів в графі з невід'ємними вагами ребер від одної вершини до всіх інших вершин. Алгоритм використовується для знаходження найкоротшого шляху від початкової вершини до всіх інших вершин графа. Алгоритм підтримує два списки вершин: список вершин, для яких вже знайдено найкоротший шлях, та список вершин, для яких шлях поки що невідомий. На кожному кроці алгоритм вибирає вершину з другого списку з найменшою вагою шляху, оновлює ваги сусідніх вершин і переносить вибрану вершину до списку вершин з відомими шляхами.

Використані структури даних: граф, пріоритетна черга, масив.

Лістинг програми:

```
import heapq

def dijkstra(graph, start):
    distances = {vertex: float('infinity') for vertex in graph}
    distances[start] = 0
    priority_queue = [(0, start)]

    while priority_queue:
        current_distance, current_vertex = heapq.heappop(priority_queue)

        if current_distance > distances[current_vertex]:
            continue

        for neighbor, weight in graph[current_vertex].items():
            distance = current_distance + weight
            if distance < distances[neighbor]:
                distances[neighbor] = distance
                heapq.heappush(priority_queue, (distance, neighbor))

    return distances

graph = {
    'A': {'B': 1, 'C': 4},
    'B': {'A': 1, 'C': 2, 'D': 5},
    'C': {'A': 4, 'B': 2, 'D': 1},
    'D': {'B': 5, 'C': 1}
}
start_vertex = 'A'
print(f"Найкоротші відстані від вершини {start_vertex} до інших вершин:")
print(dijkstra(graph, start_vertex))
```

Тестові данні:

```
graph = {
    'A': {'B': 1, 'C': 4},
    'B': {'A': 1, 'C': 2, 'D': 5},
    'C': {'A': 4, 'B': 2, 'D': 1},
    'D': {'B': 5, 'C': 1}
}
```

```
Найкоротші відстані від вершини A до інших вершин:
{'A': 0, 'B': 1, 'C': 3, 'D': 4}
```

Результат виконання програми

Пояснення до виконання програми:

- відстань від вершини A до самої себе 0 (містер очевидність)
- від вершини A до вершини B 1 (пряме ребро між вершинами)
- від вершини A до вершини C 3 (від вершини A до вершини B 1 та від вершини B до вершини C 2, тому і 3, а не 4 як це могло б бути при шляху через одне ребро між A та C)
- від вершини A до вершини D 4 (згадуємо шлях між вершинами A та C + додаємо те, що між вершиною C та D ціна 1, тому ціна шляху 4)

Хеш-таблиця з ланцюжками

Хід роботи

Опис: хеш-таблиця з ланцюжками - це метод збереження даних, який використовує хеш-функцію для визначення індексу, де буде збережено значення. Колізії вирішуються шляхом зберігання кожного значення у вигляді ланцюжка (списку) елементів, які мають однаковий хеш. Кожен елемент цього ланцюжка містить ключ та відповідне значення.

Використані структури даних: Хеш-таблиця, масив, список.

Лістинг програми:

```
class HashTable:
    def __init__(self, size):
        self.size = size
        self.hash_table = [[] for _ in range(size)]

    def hash_function(self, key):
        return key % self.size

    def insert(self, key, value):
        index = self.hash_function(key)
        self.hash_table[index].append((key, value))

    def search(self, key):
        index = self.hash_function(key)
        for k, v in self.hash_table[index]:
            if k == key:
                return v
        return None


hash_table = HashTable(3)
hash_table.insert(5, "apple")
hash_table.insert(15, "banana")
hash_table.insert(25, "cherry")

print("Значення для ключа 5:", hash_table.search(5))
print("Значення для ключа 15:", hash_table.search(15))
print("Значення для ключа 25:", hash_table.search(25))
```

Тестові данні:

```
hash_table = HashTable(3)
hash_table.insert( key: 5,  value: "apple")
hash_table.insert( key: 15,  value: "banana")
hash_table.insert( key: 25,  value: "cherry")
```

```
Значення для ключа 5: apple
Значення для ключа 15: banana
Значення для ключа 25: cherry
```

Результат виконання програми

Алгоритм Флойда

Хід роботи

Опис: алгоритм Флойда використовується для пошуку найкоротших відстаней між усіма парами вершин у напрямленому або ненапрямленому зваженому графі. Він базується на динамічному програмуванні та використовує матрицю відстаней між вершинами. Алгоритм працює у кілька етапів, на кожному з яких відновлюється матриця найкоротших відстаней, використовуючи проміжні вершини. В кінці алгоритму матриця відстаней містить найкоротші відстані між всіма парами вершин.

Використані структури даних: матриця, граф.

Лістинг програми:

```
def floyd_warshall(graph):
    n = len(graph)
    distance = [[float('inf')] * n for _ in range(n)]  # float('inf') - нескінченність, нескінченне число

    for i in range(n):
        for j in range(n):
            if i == j:
                distance[i][j] = 0
            elif graph[i][j] != 0:
                distance[i][j] = graph[i][j]

    for k in range(n):
        for i in range(n):
            for j in range(n):
                distance[i][j] = min(distance[i][j], distance[i][k] + distance[k][j])

    return distance
```

```
graph = [
    [0, 5, 0, 10],
    [0, 0, 3, 0],
    [0, 0, 0, 1],
    [0, 0, 0, 0]
]
print("Матриця найкоротших відстаней:")
for row in floyd_warshall(graph):
    print(row)
```

Тестові данні:

```
graph = [
    [0, 5, 0, 10],
    [0, 0, 3, 0],
    [0, 0, 0, 1],
    [0, 0, 0, 0]
]
```

```
Матриця найкоротших відстаней:
[0, 5, 8, 9]
[inf, 0, 3, 4]
[inf, inf, 0, 1]
[inf, inf, inf, 0]
```

Результат виконання програми

Пояснення до виконання програми:

- від вершини 0
  - до вершини 0 – 0 (очевидно)
  - до вершини 1 – 5 (пряме ребро)
  - до вершини 2 – 8 (від вершини 0 до вершини 1 – 5 + від вершини 1 до вершини 2 3, сумарно 8)
  - до вершини 3 – 9 (ребро між 0 та 1 – 5 + ребро між 1 та 2 – 3 + ребро між 3 та 4 – 1, сумарно 9)
- від вершини 1
  - до вершини 0 – відсутній
  - до вершини 1 – 0 (містер очевидність)
  - до вершини 2 – 3 (пряме ребро)
  - до вершини 3 – 4 (пряме ребро)
- від вершини 2
  - до вершини 0 – відсутній
  - до вершини 1 – відсутній
  - до вершини 2 – 0
  - до вершини 3 – 1 (пряме ребро)
- від вершини 3
  - до вершини 0 – відсутній
  - до вершини 1 - відсутній
  - до вершини 2 - відсутній
  - до вершини 3 – 0 (очевидно)

Алгоритм Шеннон-Фано

Хід роботи

Опис: алгоритм Шеннона-Фано використовується для стиснення даних. Спочатку він розділяє символи або групи символів на дві частини таким чином, щоб сума ймовірностей символів в кожній з частин була приблизно однаковою. Потім він присвоює бітові коди кожному символу так, щоб коди були унікальними і не містили префіксів один одного. Таким чином, виходить більш ефективне кодування, що дозволяє зменшити кількість бітів, потрібних для представлення даних.

Використані структури даних: масив, хеш-таблиця (у вигляді dict python`a).

Лістинг програми:

```
def shannon_fano(data):
    if len(data) == 1:
        return {data[0]: '0'}

    data_freq = {char: data.count(char) for char in set(data)}
    sorted_data_freq = sorted(data_freq.items(), key=lambda x: x[1], reverse=True)

    def divide(freq_list):
        total_freq = sum(freq for _, freq in freq_list)
        running_total = 0
        split_index = None
        for i, (_, freq) in enumerate(freq_list):
            running_total += freq
            if running_total >= total_freq / 2:
                split_index = i
                break
        return freq_list[:split_index + 1], freq_list[split_index + 1:]

    def assign_codes(freq_list, code_prefix):
        if len(freq_list) == 1:
            return {freq_list[0][0]: code_prefix}
        left_freq, right_freq = divide(freq_list)
        codes = {}
        codes.update(assign_codes(left_freq, code_prefix + '0'))
        codes.update(assign_codes(right_freq, code_prefix + '1'))
        return codes

    codes = assign_codes(sorted_data_freq, '')
    return codes



test_data = "abbcccddddeeeee"
expected_codes = {'e': '00', 'd': '01', 'c': '10', 'b': '110', 'a': '111'}

print(shannon_fano(test_data), '\n', expected_codes)
```

Тестові данні:

```
test_data = "abbcccddddeeeee"
expected_codes = {'e': '00', 'd': '01', 'c': '10', 'b': '110', 'a': '111'}
```

```
{'e': '00', 'd': '01', 'c': '10', 'b': '110', 'a': '111'}
 {'e': '00', 'd': '01', 'c': '10', 'b': '110', 'a': '111'}
```

Результат виконання програми

Алгоритм Форда-Беллмана

Хід роботи

Опис: алгоритм Форда-Беллмана використовується для знаходження найкоротших шляхів в графі з вагами на ребрах, навіть якщо він має ребра з від'ємними вагами. Алгоритм працює для напрямлених і ненапрямлених графів.

Використані структури даних: граф, масив.

Лістинг програми:

```
def bellman_ford(graph, start):
    distance = {node: float('inf') for node in graph}
    distance[start] = 0

    for _ in range(len(graph) - 1):
        for node in graph:
            for neighbor, weight in graph[node].items():
                if distance[node] + weight < distance[neighbor]:
                    distance[neighbor] = distance[node] + weight

    for node in graph:
        for neighbor, weight in graph[node].items():
            if distance[node] + weight < distance[neighbor]:
                raise ValueError("Граф містить цикл з від'ємною вагою")

    return distance

graph = {
    'A': {'B': -1, 'C': 4},
    'B': {'C': 3, 'D': 2, 'E': 2},
    'C': {},
    'D': {'B': 1, 'C': 5},
    'E': {'D': -3}
}
start_node = 'A'

print(bellman_ford(graph, start_node))
```

Тестові данні:

```
graph = {
    'A': {'B': -1, 'C': 4},
    'B': {'C': 3, 'D': 2, 'E': 2},
    'C': {},
    'D': {'B': 1, 'C': 5},
    'E': {'D': -3}
}
```

```
{'A': 0, 'B': -1, 'C': 2, 'D': -2, 'E': 1}
```

Результат виконання програми

Алгоритм Хаффмана

Хід роботи

Опис: алгоритм Хаффмана використовується для стиснення даних, особливо ефективний при стисненні текстових даних. Він будує оптимальне бінарне дерево для кожного символу на основі його частоти входження у текст. Часті символи отримують коротші коди, а рідкі - довші, що дозволяє зменшити загальний обсяг даних.

Використані структури даних: масив.

Лістинг програми:

```
import heapq
from collections import Counter


def huffman_encoding(data):
    freq = Counter(data)
    priority_queue = [[weight, [symbol, ""]] for symbol, weight in freq.items()]
    heapq.heapify(priority_queue)

    while len(priority_queue) > 1:
        left = heapq.heappop(priority_queue)
        right = heapq.heappop(priority_queue)
        for pair in left[1:]:
            pair[1] = '0' + pair[1]
        for pair in right[1:]:
            pair[1] = '1' + pair[1]
        heapq.heappush(priority_queue, [left[0] + right[0]] + left[1:] +
right[1:])
```

```
    huffman_code = {}
    for pair in priority_queue[0][1:]:
        huffman_code[pair[0]] = pair[1]

    encoded_data = ''.join(huffman_code[char] for char in data)

    return encoded_data, huffman_code


def huffman_decoding(encoded_data, huffman_code):
    reverse_code = {code: char for char, code in huffman_code.items()}
    decoded_data = ''
    current_code = ''
    for bit in encoded_data:
        current_code += bit
        if current_code in reverse_code:
            decoded_data += reverse_code[current_code]
            current_code = ''
    return decoded_data



data = "hello, world!"

encoded_data, huffman_code = huffman_encoding(data)
decoded_data = huffman_decoding(encoded_data, huffman_code)

print("Закодовані дані:", encoded_data)
print("Декодовані дані:", decoded_data)
```

Тестові данні: hello, world!

```
Закодовані дані: 1101110001011111010100000111100000110111001
Декодовані дані: hello, world!
```

Результат виконання програми

Алгоритм Крута-Морріса-Прата

Хід роботи

Опис: алгоритм Крута-Морріса-Прата використовується для ефективного пошуку всіх входжень підстроки у великому тексті. Він працює на основі зіставлення символів і може знаходити всі входження шуканої підстроки без зайвих порівнянь.

Використані структури даних: масив.

Лістинг програми:

```
def compute_lps_array(pattern):
    length = len(pattern)
    lps = [0] * length
    j = 0
```

```
    for i in range(1, length):
        while j > 0 and pattern[i] != pattern[j]:
            j = lps[j-1]
        if pattern[i] == pattern[j]:
            j += 1
        lps[i] = j
    return lps

def kmp_search(text, pattern):
    count = 0
    lps = compute_lps_array(pattern)
    i, j = 0, 0

    while i < len(text):
        if text[i] == pattern[j]:
            i += 1
            j += 1
        if j == len(pattern):
            count += 1
            j = lps[j-1]
        elif i < len(text) and text[i] != pattern[j]:
            if j != 0:
                j = lps[j-1]
            else:
                i += 1
    return count

text = "123 12 123 111"
pattern = "123"

matches_count = kmp_search(text, pattern)
print("Кількість входжень підстроки:", matches_count)
```

Тестові данні:

```
text = "123 12 123 111"
pattern = "123"
```

```
Кількість входжень підстроки: 2
```

Результат виконання програми