Лабораторна робота 3

Тема: Ефективні алгоритми реалізації дерев.
Ціль: Засвоїти метод реалізації дерев на різноманітних структурах даних. Реалізація обходів дерева.
Опорні знання: Мови програмування Python, C. Поняття АТД та реалізації АТД. АТД Дерево.
Завданння: Ознайомитися з теоретичним матеріалом та виконати завдання, визначені в розділі, підготувати відповіді на контрольні запитання, оформити протокол виконання роботи.

Завдання 1
1. Реалізувати АТД Дерево на основі масиву
2. Реалізувати методи пошуку даного елементу обходами зліва направо, зверху вниз, знизу вверх
3. Оцінити складність методів пошуку даного елементу

Завдання 2.
1. Реалізувати АТД Бінарне дерево.
2. Реалізувати алгоритм Хафмена методом побудови бінарного дерева.
3. Оцінити складність методу Хафмена.

1. Перелік операцій відповідного АТД зі специфікаціями.

__init__(розмір): Конструктор класу для створення дерева заданого розміру.
insert(індекс, значення): Вставка значення в дерево на заданий індекс.
get_left_child(індекс): Отримання значення лівої дитини вузла за індексом.
get_right_child(індекс): Отримання значення правої дитини вузла за індексом.
left_to_right_traversal(індекс): Обхід дерева зліва направо та виведення значень.
top_to_bottom_traversal(індекс): Обхід дерева зверху вниз та виведення значень.
bottom_to_top_traversal(індекс): Обхід дерева знизу вверх та виведення значень

2. Опис структур даних відповідного АТД.

self.tree: Масив для представлення дерева. Кожен елемент масиву відповідає вузлу дерева.

3. Програмний код з реалізацією АТД.

```
class ArrayTree:
    def __init__(self, size):
        # Конструктор класу, створює масив заданого розміру для представлення дерева
        self.tree = [None] * size

    def insert(self, index, value):
        # Метод для вставки значення в масив на заданий індекс
        self.tree[index] = value

    def get_left_child(self, index):
        # Метод для отримання значення лівого вузла за його індексом
        left_child_index = 2 * index + 1
        if left_child_index < len(self.tree):
            return self.tree[left_child_index]
        return None

    def get_right_child(self, index):
        # Метод для отримання значення право вузла за його індексом
        right_child_index = 2 * index + 2
```

```
        if right_child_index < len(self.tree):
            return self.tree[right_child_index]
        return None

    def left_to_right_traversal(self, index):
        # Обхід дерева зліва направо
        if index < len(self.tree) and self.tree[index] is not None:
            self.left_to_right_traversal(2 * index + 1)  # left child
            print(self.tree[index], end=" ")
            self.left_to_right_traversal(2 * index + 2)  # right child

    def top_to_bottom_traversal(self, index):
        # Обхід дерева зверху вниз
        if index < len(self.tree) and self.tree[index] is not None:
            print(self.tree[index], end=" ")
            self.top_to_bottom_traversal(2 * index + 1)  # left child
            self.top_to_bottom_traversal(2 * index + 2)  # right child

    def bottom_to_top_traversal(self, index):
        # Обхід дерева знизу вверх
        if index < len(self.tree) and self.tree[index] is not None:
            self.bottom_to_top_traversal(2 * index + 1)  # left child
            self.bottom_to_top_traversal(2 * index + 2)  # right child
            print(self.tree[index], end=" ")

# Створення об'єкту класу ArrayTree з розміром 10
tree = ArrayTree(10)

# Вставка значень у дерево на певні індекси
tree.insert(0, 1)
tree.insert(1, 2)
tree.insert(2, 3)
tree.insert(3, 4)
tree.insert(4, 5)

# Обхід дерева зліва направо
print("Left to Right Traversal:")
tree.left_to_right_traversal(0)
print("\n")

# Обхід дерева зверху вниз
print("Top to Bottom Traversal:")
tree.top_to_bottom_traversal(0)
print("\n")

# Обхід дерева знизу вверх
print("Bottom to Top Traversal:")
tree.bottom_to_top_traversal(0)
print(«\n")
```

Оцінити складність методів пошуку даного елементу:

Метод вставки (insert):

Часова складність: O(1)

Пояснення: Метод insert просто присвоює значення певному індексу у масиві. Незалежно від розміру масиву операція виконується за постійний час.

Метод отримання лівого дочірнього елемента (get_left_child):

Часова складність: O(1)

Пояснення: Обчислення індексу лівого дочірнього елемента та доступ до відповідного значення у масиві є операціями постійного часу.

Метод отримання правого дочірнього елемента (get_right_child):

Часова складність: O(1)

Пояснення: Аналогічно методу get_left_child, обчислення індексу правого дочірнього елемента та доступ до відповідного значення у масиві є операціями постійного часу.

Метод обходу зліва направо (left_to_right_traversal):

Часова складність: O(n)

Пояснення: Метод обходить весь масив у глибину. У найгіршому випадку він відвідає кожен елемент один раз, що призводить до часової складності O(n), де n - кількість елементів у масиві.

Метод обходу зверху вниз (top_to_bottom_traversal):

Часова складність: O(n)

Пояснення: Аналогічно методу left_to_right_traversal, обхід зверху вниз відвідає кожен елемент у масиві один раз, що призводить до часової складності O(n).

Метод обходу знизу вгору (bottom_to_top_traversal):

Часова складність: O(n)

Пояснення: Подібно до попередніх методів обходу, обхід знизу вгору відвідає кожен елемент у масиві один раз, що призводить до часової складності O(n).

4. Опис структур даних методу Хаффмана.

HuffmanNode - Вузол бінарного дерева:
Описує вузол бінарного дерева Хаффмана.
Має поля:

value: сума частоти (значення) вузла.
char: символ (для листків дерева).
left: посилання на лівого нащадка.
right: посилання на правого нащадка.

Має метод __lt__ для порівняння вузлів при використанні у min-heap.
build_huffman_tree - Функція побудови бінарного дерева:
Побудова бінарного дерева Хаффмана на основі частотного словника.
Використовує min-heap для оптимізації вибору вузлів з найменшою частотою.
Повертає кореневий вузол побудованого бінарного дерева.
print_huffman_tree - Функція виведення на екран бінарного дерева:
Рекурсивно виводить символ, частоту та код Хаффмана для кожного вузла дерева.

5. Програмний код з реалізацією методу Хаффмана.

```
import heapq

class HuffmanNode:
    def __init__(self, value, char=None):
        self.value = value
        self.char = char
        self.left = None
        self.right = None

    # Додано метод порівняння для визначення порядку в min-heap
    def __lt__(self, other):
        return self.value < other.value

def build_huffman_tree(freq_dict):
    heap = [HuffmanNode(freq, char) for char, freq in freq_dict.items()]
    heapq.heapify(heap)

    while len(heap) > 1:
        left = heapq.heappop(heap)
        right = heapq.heappop(heap)

        merged_node = HuffmanNode(left.value + right.value)
```

```
        merged_node.left, merged_node.right = left, right

        heapq.heappush(heap, merged_node)

    return heap[0]

def print_huffman_tree(node, encoding='', separator='-'):
    if node is not None:
        if node.char is not None:
            print(f"Symbol: {node.char}, Frequency: {node.value}, Encoding: {encoding}")

        print_huffman_tree(node.left, encoding + '0')
        print_huffman_tree(node.right, encoding + '1')

# Приклад використання
frequency_dict = {'a': 8, 'b': 3, 'c': 1, 'd': 6}
huffman_tree_root = build_huffman_tree(frequency_dict)

# Виведення на екран створеного бінарного дерева Хафмена
print("Huffman Tree:")
print_huffman_tree(huffman_tree_root)
```

Оцінити складність методу Хафмена:

Метод побудови дерева Хаффмана (build_huffman_tree):

Часова складність: O(n log n), де n - кількість символів (листків) у вхідному словнику freq_dict.

Пояснення:

Створення початкового min-heap займає O(n) часу, де n - кількість символів.

Кожна ітерація циклу while використовує heapop та heappush, кожне з яких виконується за O(log n).

Кількість ітерацій циклу while дорівнює n-1, оскільки кожда ітерація об'єднує два найменші вузли.

Таким чином, загальна часова складність методу для побудови дерева Хаффмана є O(n log n).

Контрольні запитання:

1. Основні операції АТД Дерево: Ініціалізація, вставка, видалення, обходи.
2. Методи реалізації АТД Дерево на основі різних структур даних: На основі масиву, списку, вузлового представлення.
3. Основні методи обходів дерева: Префіксний (пошук зліва направо), інфіксний (зверху вниз), постфіксний (знизу вверх).
4. Методи реалізації АТД Дерево: Порівняння за ефективністю, простотою реалізації та універсальністю.
5. Постановка задачі кодування та метод Хаффмена: Задача: Мінімізація довжини коду для представлення даних. Метод Хаффмена: побудова бінарного дерева з мінімальними вагами.
6. Структури даних для методу Хаффмена: Структури даних для зберігання та обробки частот символів, пріоритетна черга для вузлів дерева.