**ЛАБОРАТОРНА РОБОТА № 1**

**1.1 Тема роботи**

Лінійні структури даних

**1.2 Мета роботи**

1.2.1 Вивчити основні концепції побудови лінійних структур  даних: зв’язних списків, стеків, куп та черг з пріоритетами. 1.1.2 Навчитися обирати та реалізовувати структури даних для  сортування, вставки, видалення та пошуку елементів.

1.2.3 Навчитися реалізовувати та застосовувати алгоритм  пірамідального сортування на практиці.

**1.3 Завдання**

Розв’язати індивідуальне завдання за допомогою  розробленої реалізації пірамідального сортування.

Варіант № 15

На підприємстві дані про кожного співробітника в одному з відділів включають наступні параметри:

– прізвище, ім’я, по батькові;

– посада;

– стаж;

– домашня адреса;

– дата народження.

Вивести перелік співробітників у порядку зменшення їх віку, для кожного вказуючи кількість повних років. У випадку однакової дати народження виводити спочатку співробітника з меншим стажем.

**1.4 Хід роботи**

**1.4.1 Код програми**

from datetime import datetime

employees = [

    {

        "прізвище": "Іванов",

        "ім`я": "Петро",

        "по батькові": "Олександрович",

        "посада": "інженер",

        "стаж": 5,

        "домашня адреса": "вул. Головна, 123",

        "дата народження": "1990-05-15"

    },

        {

        "прізвище": "Іванов",

        "ім`я": "Данило",

        "по батькові": "Олександрович",

        "посада": "інженер",

        "стаж": 4,

        "домашня адреса": "вул. Головна, 123",

        "дата народження": "1990-05-15"

    },

    {

        "прізвище": "Петрова",

        "ім`я": "Марія",

        "по батькові": "Іванівна",

        "посада": "бухгалтер",

        "стаж": 2,

        "домашня адреса": "вул. Центральна, 45",

        "дата народження": "1992-10-20"

    },

    {

        "прізвище": "Сидоренко",

        "ім`я": "Олег",

        "посада": "менеджер",

        "стаж": 3,

        "домашня адреса": "вул. Паркова, 67",

        "дата народження": "1992-03-30"

    }

]

def heapify(arr, n, i):

    largest = i

    left = 2 \* i + 1

    right = 2 \* i + 2

    if left < n and (calculate\_age(arr[i]["дата народження"]) < calculate\_age(arr[left]["дата народження"]) or

                     (calculate\_age(arr[i]["дата народження"]) == calculate\_age(arr[left]["дата народження"]) and arr[i]["стаж"] < arr[left]["стаж"])):

        largest = left

    if right < n and (calculate\_age(arr[largest]["дата народження"]) < calculate\_age(arr[right]["дата народження"]) or

                      (calculate\_age(arr[largest]["дата народження"]) == calculate\_age(arr[right]["дата народження"]) and arr[largest]["стаж"] < arr[right]["стаж"])):

        largest = right

    if largest != i:

        arr[i], arr[largest] = arr[largest], arr[i]

        heapify(arr, n, largest)

def heap\_sort(arr):

    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)

def calculate\_age(date\_of\_birth):

    today = datetime.today()

    birth\_date = datetime.strptime(date\_of\_birth, "%Y-%m-%d")

    age = today.year - birth\_date.year

    return age

heap\_sort(employees)

for employee in employees:

    age = calculate\_age(employee["дата народження"])

    print(f"{employee['прізвище']} {employee['ім`я']} - {age} років, стаж: {employee['стаж']} років")

class Node:

    def \_\_init\_\_(self, data):

        self.data = data

        self.next = None

        self.prev = None

class DoublyLinkedList:

    def \_\_init\_\_(self):

        self.head = None

        self.tail = None

    def add\_to\_beginning(self, data):

        new\_node = Node(data)

        if self.head is None:

            self.head = new\_node

            self.tail = new\_node

        else:

            new\_node.next = self.head

            self.head.prev = new\_node

            self.head = new\_node

    def add\_after(self, prev\_node, data):

        if prev\_node is None:

            return

        new\_node = Node(data)

        new\_node.next = prev\_node.next

        prev\_node.next = new\_node

        new\_node.prev = prev\_node

        if new\_node.next:

            new\_node.next.prev = new\_node

        else:

            self.tail = new\_node

    def find\_node(self, data):

        current = self.head

        while current:

            if current.data == data:

                return current

            current = current.next

        return None

    def delete\_node(self, node):

        if node is None:

            return

        if node.prev:

            node.prev.next = node.next

        else:

            self.head = node.next

        if node.next:

            node.next.prev = node.prev

        else:

            self.tail = node.prev

    def display\_from\_start(self):

        current = self.head

        while current:

            print(current.data, end=' ')

            current = current.next

        print()

    def display\_from\_end(self):

        current = self.tail

        while current:

            print(current.data, end=' ')

            current = current.prev

        print()

class PriorityQueue:

    def \_\_init\_\_(self):

        self.heap = []

    def insert(self, item):

        self.heap.append(item)

        self.\_heapify\_up(len(self.heap) - 1)

    def remove(self):

        if not self.heap:

            return None

        if len(self.heap) == 1:

            return self.heap.pop()

        root = self.heap[0]

        self.heap[0] = self.heap.pop()

        self.\_heapify\_down(0)

        return root

    def \_heapify\_up(self, index):

        while index > 0:

            parent\_index = (index - 1) // 2

            if self.heap[index] < self.heap[parent\_index]:

                self.heap[index], self.heap[parent\_index] = self.heap[parent\_index], self.heap[index]

                index = parent\_index

            else:

                break

    def \_heapify\_down(self, index):

        left\_child\_index = 2 \* index + 1

        right\_child\_index = 2 \* index + 2

        smallest = index

        if left\_child\_index < len(self.heap) and self.heap[left\_child\_index] < self.heap[smallest]:

            smallest = left\_child\_index

        if right\_child\_index < len(self.heap) and self.heap[right\_child\_index] < self.heap[smallest]:

            smallest = right\_child\_index

        if smallest != index:

            self.heap[index], self.heap[smallest] = self.heap[smallest], self.heap[index]

            self.\_heapify\_down(smallest)

    def build\_heap(self, arr):

        self.heap = arr

        for i in range(len(arr) // 2, -1, -1):

            self.\_heapify\_down(i)

    def sort(self):

        sorted\_arr = []

        while True:

            item = self.remove()

            if item is None:

                break

            sorted\_arr.append(item)

        return sorted\_arr

    def display(self):

        for item in self.heap:

            print(item, end=' ')

        print()

def heap\_sort(arr):

    priority\_queue = PriorityQueue()

    for item in arr:

        priority\_queue.insert(item)

    sorted\_arr = []

    while True:

        item = priority\_queue.remove()

        if item is None:

            break

        sorted\_arr.append(item)

    return sorted\_arr

if \_\_name\_\_ == "\_\_main\_\_":

    print("\nDoubly Linked List:")

    linked\_list = DoublyLinkedList()

    linked\_list.add\_to\_beginning(9)

    linked\_list.add\_to\_beginning(5)

    linked\_list.add\_to\_beginning(1)

    linked\_list.add\_to\_beginning(2)

    linked\_list.add\_after(linked\_list.find\_node(5), 4)

    linked\_list.display\_from\_start()

    linked\_list.delete\_node(linked\_list.find\_node(1))

    linked\_list.display\_from\_start()

    linked\_list.display\_from\_end()

    print("\nPriority Queue:")

    priority\_queue = PriorityQueue()

    arr = [9, 7, 5, 2, 10, 1]

    for item in arr:

        priority\_queue.insert(item)

    priority\_queue.display()

    sorted\_arr = priority\_queue.sort()

    print("Sorted Array:", sorted\_arr)

    unsorted\_arr = [4, 8, 2, 6, 7, 1]

    priority\_queue.build\_heap(unsorted\_arr)

    print("Sorted Array:")

    priority\_queue.display()

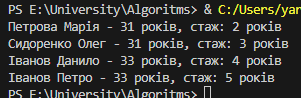
    print("\nSort:")

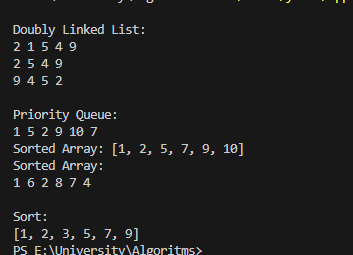
    arr2 = [7, 3, 1, 5, 9, 2]

    sorted\_arr = heap\_sort(arr2)

    print(sorted\_arr)

**1.4.2 Приклад роботи програми**





**1.4.3 Відповіді на контрольні запитання**

**1.** **Що представляє собою піраміда і яким властивостям така структура даних повинна задовольняти?**

Піраміда це структура даних, що представляється як бінарне дерево, де кожен батьківський елемент має або більше або менше значення від дочірніх елементів.

**2.** **У яких випадках краще використовувати структуру даних типу купа, а у яких – чергу з пріоритетами?**

Черга з пріоритетами буде більш відповідною до задач, де кожен елемент має пріоритет і потрібно обробляти їх у визначеному порядку. Коли «купа» буде кращим варіантом коли необхідно відсортувати данні або шукати максимальний\мінімальний елемент.

**10. У чому полягає алгоритм пірамідального сортування?**

Під час пірамідального сортування:

1. Створюється купа з вихідного масиву даних, забезпечуючи правильний порядок.
2. Корінь купи обмінюється з останнім елементом масиву для впорядкування.
3. Відновлюється порядок у купі, і цей процес повторюється до завершення сортування всього масиву.

**1.5 Висновок**

За цю лабораторну роботу, я вивчив основні концепції побудови лінійних структур  даних: зв’язних списків, стеків, куп та черг з пріоритетами, навчився обирати та реалізовувати структури даних для  сортування, вставки, видалення та пошуку елементів.