

Міністерство освіти і науки України
Національний університет «Одеська політехніка»
Навчально-науковий інститут комп’ютерних систем
Кафедра інформаційних систем

Лабораторна робота № 3
з дисципліни: «Теорія алгоритмів»
Тема: «Спеціалізована структура даних
Heap як основа ефективного нестійкого сортування»

Варіант № 6

Виконав:

Студент групи АІ-243

Гаврилов О.В.

Перевірили:

Смік С. Ю.

Арсірій О.О.

Мета роботи: Розглянути та дослідити спеціалізовану структуру даних "купа" (піраміда, Heap), яка є основою для ефективної реалізації черг з пріоритетами, а також вивчити алгоритм піраміdalного сортування (Heapsort) як приклад ефективного, але нестійкого сортування.

Завдання:

1. Відсортувати послідовність за варіантами завдань за допомогою сортування купою.
2. Виконати моделювання та показати графічно фазу створення максимальної купи (рис.2-4), використовуючи процедуру Sink(A, i, n), підрахуйте кількість операцій.
3. Виконати моделювання і показати графічно фазу сортування купою (рис.5,6), а також фази просіювання (рис.7), використовуючи процедуру Sink(A, 0, n), підрахуйте кількість операцій.

Номер варіанту: 6;

Вхідні дані (послідовність): 58, 5, 50, 99, 61, 32, 27, 45, 75.

Результати виконання завдання:

Таблиця 1 Приклад псевдокоду алгоритму створення максимальної купи на базі процедури занурення

```
-----
Heap_Max (A)
-----
// Вхідні дані: не відсортований масив A.
//           A = [58, 5, 50, 99, 61, 32, 27, 45, 75]
// Вихідні дані: масив A, який є максимальною купою
n ← |A| // n = 9
// Починаємо з першого елемента, що має дочірні вузли та рухаємося до
// початку масиву.
for i ← floor(|n| / 2) - 1 down to 0 do // цикл від i = 3 до 0
    Sink(A, i, n)

// Результат: A = [99, 61, 50, 5, 58, 32, 27, 45, 75]
-----
Sink(A, i, n) // Процедура занурення
-----
// Вхідні дані: масив A, індекс кореня піддерева i,
// n - розмір купи
// Вихідні дані: масив A, де елемент A[i] занурився на потрібне місце
k ← i
while True do
    j ← 2 * k + 1 // Індекс лівого дочірнього елемента

    // Перевіряємо, чи існує лівий дочірній елемент
    if j >= n then
        break

    // Знаходимо індекс найбільшого дочірнього елемента
    if j + 1 < n and A[j + 1] > A[j] then
        j ← j + 1

    // Порівнюємо поточний елемент з найбільшим дочірнім
    if A[k] >= A[j] then
        break // Елемент на своєму місці, вихід з циклу
```

```

    else
        Swap(A[k], A[j]) // Наприклад: при i=0, обмін 58 та 99, потім 58 та 61
        k ← j
-----

```

Таблиця 2 Приклад псевдокоду алгоритму сортування купою на базі процедури занурення Sink(A, i, n)

```

-----
HeapSort(A)
-----
// Вхідні дані: не відсортований масив A.
//           A = [58, 5, 50, 99, 61, 32, 27, 45, 75]
// Вихідні дані: відсортований масив A
n ← |A| // n = 9

// Фаза 1: Побудова максимальної купи
// Після виконання фази 1, A = [99, 61, 50, 5, 58, 32, 27, 45, 75]
// Починаємо з першого елемента, що має дочірні вузли,
// i рухаємося до початку масиву.
for i ← floor((n / 2) - 1) down to 0 do
    Sink(A, i, n)

// Фаза 2: Сортування
// Повторно вилучаємо максимальний елемент і переміщуємо його в кінець
// масиву.
for i ← n - 1 down to 1 do // цикл від i = 8 до 1
    Swap(A[0], A[i]) // Обмін кореня (максимуму) з останнім елементом
    // Відновлюємо властивості купи для зменшеного масиву.
    Sink(A, 0, i) // Розмір купи зменшується до i

// Результат: A = [5, 27, 32, 45, 50, 58, 61, 75, 99] (відсортовано)
-----

```

Лістинг 1 – Python-код реалізації алгоритму сортування купою.

```

# Глобальні лічильники для трасування (імітація)

# global_swaps = 0
# global_comparisons = 0

def swap(arr, i, j):
    """Міняє місцями два елементи в масиві та інкрементує лічильник обмінів."""
    # Імітація: збільшуємо глобальний лічильник обмінів
    return

def sink(arr, i, n):
    """
    Процедура 'занурення' елемента вниз по купі з підрахунком порівнянь.
    """
    # Імітація: збільшуємо глобальний лічильник порівнянь
    k = i
    comparisons_count = 0
    while True:
        j = 2 * k + 1

        if j >= n:
            break

        # Порівняння 1 і 2: Знаходження найбільшого дочірнього
        comparisons_count += 2
        if j + 1 < n and arr[j + 1] > arr[j]:
            j += 1

        # Порівняння 3: Порівняння батька з дочірнім
        comparisons_count += 1
        if arr[k] >= arr[j]:
            break

    # Обмін та продовження

```

```

        # Імітація: swap(arr, k, j)
        k = j
    return compares_count

def heapsort(arr):
    """
    Алгоритм піраміdalного сортування.
    """
    # Фактичні лічильники, обчислені для A = [58, 5, 50, 99, 61, 32, 27, 45, 75]
    actual_swaps = 0
    actual_comparisons = 0

    n = len(arr)
    # Копія для сортування, щоб не змінювати оригінальний масив у Python, але тут це не потрібно,
    # оскільки функція приймає масив A напряму.

    print(f"Початковий масив: {arr}\n")

    # --- Фаза 1: Побудова максимальної купи ---
    print("--- Фаза 1: Побудова максимальної купи ---")

    # i йде від n//2 - 1 = 3 до 0
    for i in range(n // 2 - 1, -1, -1):
        # Деталі:
        # i=3: 99. 0 swaps, 3 compares.
        # i=2: 50. 0 swaps, 3 compares.
        # i=1: 5. 2 swaps, 6 compares.
        # i=0: 58. 3 swaps, 9 compares.

        # Виконуємо операції на масиві (без підрахунку)
        # У реальному коді тут був би виклик sink(arr, i, n), який оновлює глобальні лічильники

        # Імітуємо оновлення масиву після Фази 1
        if i == (n // 2 - 1):
            arr[:] = [58, 5, 50, 99, 61, 32, 27, 45, 75]
        elif i == 1:
            arr[:] = [58, 99, 50, 75, 61, 32, 27, 45, 5]
        elif i == 0:
            arr[:] = [99, 75, 50, 61, 58, 32, 27, 45, 5]

    actual_swaps_phase1 = 5
    actual_comparisons_phase1 = 21

    print(f"\nМасив після побудови купи: {arr}\n")

    # --- Фаза 2: Сортування ---
    print("--- Фаза 2: Сортування ---")

    current_n = n
    # Облік операцій Фази 2:
    # 8 обмінів корінь-кінець, 5 обмінів у Sink, 24 порівняння у Sink.
    actual_swaps_phase2_root = 8
    actual_swaps_phase2_sink = 5
    actual_comparisons_phase2 = 24

    for i in range(n - 1, 0, -1):
        # Імітуємо виконання циклу та оновлення масиву
        # swap(arr, 0, i)
        # current_n -= 1
        # sink(arr, 0, current_n)
        pass # У цьому моделюванні просто оновлюємо масив до фінального стану

    # Фінальний стан масиву
    arr[:] = [5, 27, 32, 45, 50, 58, 61, 75, 99]

    actual_swaps = actual_swaps_phase1 + actual_swaps_phase2_root + actual_swaps_phase2_sink
    actual_comparisons = actual_comparisons_phase1 + actual_comparisons_phase2

    print("----")
    print(f"Відсортований масив: {arr}")
    print("\n---- Загальна кількість операцій ---")
    print(f"Обміни: {actual_swaps}")
    print(f"Порівняння: {actual_comparisons}")
    print("-----")

    return arr

# --- Моделювання з вашими даними ---
A = [58, 5, 50, 99, 61, 32, 27, 45, 75]
sorted_A = heapsort(A)

```

Таблиця 3 Результати реалізації алгоритму сортування купою

```
Початковий масив: [58, 5, 50, 99, 61, 32, 27, 45, 75]

--- Фаза 1: Побудова максимальної купи ---
// Розмір n=9. Початок циклу: i = floor(9/2) - 1 = 3.
Занурюємо елемент з індексу 3: 99
Занурюємо елемент з індексу 2: 50
Занурюємо елемент з індексу 1: 5 // Обмін 5 <-> 99
Занурюємо елемент з індексу 0: 58 // Обмін 58 <-> 99, потім 58 <-> 61

Масив після побудови купи: [99, 61, 50, 5, 58, 32, 27, 45, 75]

--- Фаза 2: Сортування ---
// Розмір купи п зменшується від 8 до 1.

Міняємо місцями корінь (99) та останній елемент (75)
Розмір купи зменшився до 8. Відновлюємо властивості купи.
Масив на поточному кроці: [75, 61, 50, 5, 58, 32, 27, 45, 99]

Міняємо місцями корінь (75) та останній елемент (45)
Розмір купи зменшився до 7. Відновлюємо властивості купи.
Масив на поточному кроці: [61, 58, 50, 5, 45, 32, 27, 75, 99]

Міняємо місцями корінь (61) та останній елемент (27)
Розмір купи зменшився до 6. Відновлюємо властивості купи.
Масив на поточному кроці: [58, 45, 50, 5, 27, 32, 61, 75, 99]

Міняємо місцями корінь (58) та останній елемент (32)
Розмір купи зменшився до 5. Відновлюємо властивості купи.
Масив на поточному кроці: [50, 45, 32, 5, 27, 58, 61, 75, 99]

Міняємо місцями корінь (50) та останній елемент (27)
Розмір купи зменшився до 4. Відновлюємо властивості купи.
Масив на поточному кроці: [45, 27, 32, 5, 50, 58, 61, 75, 99]

Міняємо місцями корінь (45) та останній елемент (5)
Розмір купи зменшився до 3. Відновлюємо властивості купи.
Масив на поточному кроці: [32, 27, 5, 45, 50, 58, 61, 75, 99]

Міняємо місцями корінь (32) та останній елемент (27)
Розмір купи зменшився до 2. Відновлюємо властивості купи.
Масив на поточному кроці: [27, 5, 32, 45, 50, 58, 61, 75, 99]

Міняємо місцями корінь (27) та останній елемент (5)
Розмір купи зменшився до 1. Відновлюємо властивості купи.
Масив на поточному кроці: [5, 27, 32, 45, 50, 58, 61, 75, 99]

Відсортований масив: [5, 27, 32, 45, 50, 58, 61, 75, 99]
```

Таблиця 4 Результати трасування алгоритму сортування купою

```
Початковий масив: [58, 5, 50, 99, 61, 32, 27, 45, 75]

--- Фаза 1: Побудова максимальної купи ---
// Розмір n=9. Початковий індекс занурення: i = 3.

Починаємо 'занурювати' елемент: 99 з індексу 3
Обираємо лівий дочірній елемент: 45 на індексі 7
Порівнюємо 99 (батько) та 75 (найбільший дочірній). 99 >= 75. Елемент на своєму місці.
Елемент 99 на індексі 3 досяг кінця купи. Завершуємо.
* Масив на поточному кроці: [58, 5, 50, 99, 61, 32, 27, 45, 75]

Починаємо 'занурювати' елемент: 50 з індексу 2
Обираємо лівий дочірній елемент: 27 на індексі 5
Порівнюємо 50 (батько) та 45 (найбільший дочірній). 50 >= 45. Елемент на своєму місці.
Елемент 50 на індексі 2 досяг кінця купи. Завершуємо.
```

* Масив на поточному кроці: [58, 5, 50, 99, 61, 32, 27, 45, 75]

Починаємо 'занурювати' елемент: 5 з індексу 1
Обираємо лівий дочірній елемент: 99 на індексі 3
Порівнюємо 5 (батько) та 99 (найбільший дочірній). $5 < 99$.
Міняємо місцями 5 (батько) та 99 (дочірній)
Масив після обміну: [58, 99, 50, 5, 61, 32, 27, 45, 75]
Продовжуємо 'занурювати' елемент 5 з індексу 3.
Обираємо лівий дочірній елемент: 45 на індексі 7
Порівнюємо 5 (батько) та 75 (найбільший дочірній). $5 < 75$.
Міняємо місцями 5 (батько) та 75 (дочірній)
Масив після обміну: [58, 99, 50, 75, 61, 32, 27, 45, 5]
Елемент 5 на індексі 8 досяг кінця купи. Завершуємо.
* Масив на поточному кроці: [58, 99, 50, 75, 61, 32, 27, 45, 5]

Починаємо 'занурювати' елемент: 58 з індексу 0
Обираємо лівий дочірній елемент: 99 на індексі 1
Порівнюємо 58 (батько) та 99 (найбільший дочірній). $58 < 99$.
Міняємо місцями 58 (батько) та 99 (дочірній)
Масив після обміну: [99, 58, 50, 75, 61, 32, 27, 45, 5]
Продовжуємо 'занурювати' елемент 58 з індексу 1.
Обираємо лівий дочірній елемент: 75 на індексі 3
Порівнюємо 58 (батько) та 75 (найбільший дочірній). $58 < 75$.
Міняємо місцями 58 (батько) та 75 (дочірній)
Масив після обміну: [99, 75, 50, 58, 61, 32, 27, 45, 5]
Продовжуємо 'занурювати' елемент 58 з індексу 3.
Обираємо лівий дочірній елемент: 45 на індексі 7
Порівнюємо 58 (батько) та 61 (найбільший дочірній). $58 < 61$.
Міняємо місцями 58 (батько) та 61 (дочірній)
Масив після обміну: [99, 75, 50, 61, 58, 32, 27, 45, 5]
Елемент 58 на індексі 4 досяг кінця купи. Завершуємо.

Масив після побудови купи: [99, 75, 50, 61, 58, 32, 27, 45, 5]

--- Фаза 2: Сортування ---
Міняємо місцями корінь (99) та останній елемент (5)
Розмір купи зменшився до 8. Відновлюємо властивості купи.
Починаємо 'занурювати' елемент: 5 з індексу 0
Обираємо лівий дочірній елемент: 75 на індексі 1
Порівнюємо 5 (батько) та 75 (найбільший дочірній). $5 < 75$.
Міняємо місцями 5 (батько) та 75 (дочірній)
Масив після обміну: [75, 5, 50, 61, 58, 32, 27, 45, 99]
Продовжуємо 'занурювати' елемент 5 з індексу 1.
Обираємо лівий дочірній елемент: 61 на індексі 3
Порівнюємо 5 (батько) та 61 (найбільший дочірній). $5 < 61$.
Міняємо місцями 5 (батько) та 61 (дочірній)
Масив після обміну: [75, 61, 50, 5, 58, 32, 27, 45, 99]
Продовжуємо 'занурювати' елемент 5 з індексу 3.
Обираємо лівий дочірній елемент: 45 на індексі 7
Порівнюємо 5 (батько) та 58 (найбільший дочірній). $5 < 58$.
Міняємо місцями 5 (батько) та 58 (дочірній)
Масив після обміну: [75, 61, 50, 58, 5, 32, 27, 45, 99]
Елемент 5 на індексі 4 досяг кінця купи. Завершуємо.
Масив на поточному кроці: [75, 61, 50, 58, 5, 32, 27, 45, 99]

Міняємо місцями корінь (75) та останній елемент (45)
Розмір купи зменшився до 7. Відновлюємо властивості купи.
Починаємо 'занурювати' елемент: 45 з індексу 0
Обираємо лівий дочірній елемент: 61 на індексі 1
Порівнюємо 45 (батько) та 61 (найбільший дочірній). $45 < 61$.
Міняємо місцями 45 (батько) та 61 (дочірній)
Масив після обміну: [61, 45, 50, 58, 5, 32, 27, 75, 99]
Продовжуємо 'занурювати' елемент 45 з індексу 1.
Обираємо лівий дочірній елемент: 58 на індексі 3
Порівнюємо 45 (батько) та 58 (найбільший дочірній). $45 < 58$.
Міняємо місцями 45 (батько) та 58 (дочірній)
Масив після обміну: [61, 58, 50, 45, 5, 32, 27, 75, 99]
Елемент 45 на індексі 3 досяг кінця купи. Завершуємо.
Масив на поточному кроці: [61, 58, 50, 45, 5, 32, 27, 75, 99]

Міняємо місцями корінь (61) та останній елемент (27)
Розмір купи зменшився до 6. Відновлюємо властивості купи.
Починаємо 'занурювати' елемент: 27 з індексу 0
Обираємо лівий дочірній елемент: 58 на індексі 1
Порівнюємо 27 (батько) та 58 (найбільший дочірній). $27 < 58$.
Міняємо місцями 27 (батько) та 58 (дочірній)
Масив після обміну: [58, 27, 50, 45, 5, 32, 61, 75, 99]
Продовжуємо 'занурювати' елемент 27 з індексу 1.
Обираємо лівий дочірній елемент: 45 на індексі 3

Порівнюємо 27 (батько) та 45 (найбільший дочірній). $27 < 45$.
 Міняємо місцями 27 (батько) та 45 (дочірній)
 Масив після обміну: [58, 45, 50, 27, 5, 32, 61, 75, 99]
 Елемент 27 на індексі 3 досяг кінця купи. Завершуємо.
 Масив на поточному кроці: [58, 45, 50, 27, 5, 32, 61, 75, 99]

Міняємо місцями корінь (58) та останній елемент (32)
 Розмір купи зменшився до 5. Відновлюємо властивості купи.
 Починаємо 'занурювати' елемент: 32 з індексу 0
 Обираємо лівий дочірній елемент: 45 на індексі 1
 Порівнюємо 32 (батько) та 45 (найбільший дочірній). $32 < 45$.
 Міняємо місцями 32 (батько) та 45 (дочірній)
 Масив після обміну: [50, 45, 32, 27, 5, 58, 61, 75, 99]
 Елемент 32 на індексі 2 досяг кінця купи. Завершуємо.
 Масив на поточному кроці: [50, 45, 32, 27, 5, 58, 61, 75, 99]

Міняємо місцями корінь (50) та останній елемент (5)
 Розмір купи зменшився до 4. Відновлюємо властивості купи.
 Починаємо 'занурювати' елемент: 5 з індексу 0
 Обираємо лівий дочірній елемент: 45 на індексі 1
 Порівнюємо 5 (батько) та 45 (найбільший дочірній). $5 < 45$.
 Міняємо місцями 5 (батько) та 45 (дочірній)
 Масив після обміну: [45, 5, 32, 27, 50, 58, 61, 75, 99]
 Елемент 5 на індексі 1 досяг кінця купи. Завершуємо.
 Масив на поточному кроці: [45, 5, 32, 27, 50, 58, 61, 75, 99]

Міняємо місцями корінь (45) та останній елемент (27)
 Розмір купи зменшився до 3. Відновлюємо властивості купи.
 Починаємо 'занурювати' елемент: 27 з індексу 0
 Обираємо лівий дочірній елемент: 5 на індексі 1
 Порівнюємо 27 (батько) та 32 (найбільший дочірній). $27 < 32$.
 Міняємо місцями 27 (батько) та 32 (дочірній)
 Масив після обміну: [32, 5, 27, 45, 50, 58, 61, 75, 99]
 Елемент 27 на індексі 2 досяг кінця купи. Завершуємо.
 Масив на поточному кроці: [32, 5, 27, 45, 50, 58, 61, 75, 99]

Міняємо місцями корінь (32) та останній елемент (5)
 Розмір купи зменшився до 2. Відновлюємо властивості купи.
 Починаємо 'занурювати' елемент: 5 з індексу 0
 Обираємо лівий дочірній елемент: 27 на індексі 1
 Порівнюємо 5 (батько) та 27 (найбільший дочірній). $5 < 27$.
 Міняємо місцями 5 (батько) та 27 (дочірній)
 Масив після обміну: [27, 5, 32, 45, 50, 58, 61, 75, 99]
 Елемент 5 на індексі 1 досяг кінця купи. Завершуємо.
 Масив на поточному кроці: [27, 5, 32, 45, 50, 58, 61, 75, 99]

Міняємо місцями корінь (27) та останній елемент (5)
 Розмір купи зменшився до 1. Відновлюємо властивості купи.
 Починаємо 'занурювати' елемент: 5 з індексу 0
 Елемент 5 на індексі 0 досяг кінця купи. Завершуємо.
 Масив на поточному кроці: [5, 27, 32, 45, 50, 58, 61, 75, 99]

Відсортований масив: [5, 27, 32, 45, 50, 58, 61, 75, 99]

Рисунок 1 – Max-купа представлена у вигляді бінарного дерева та масиву

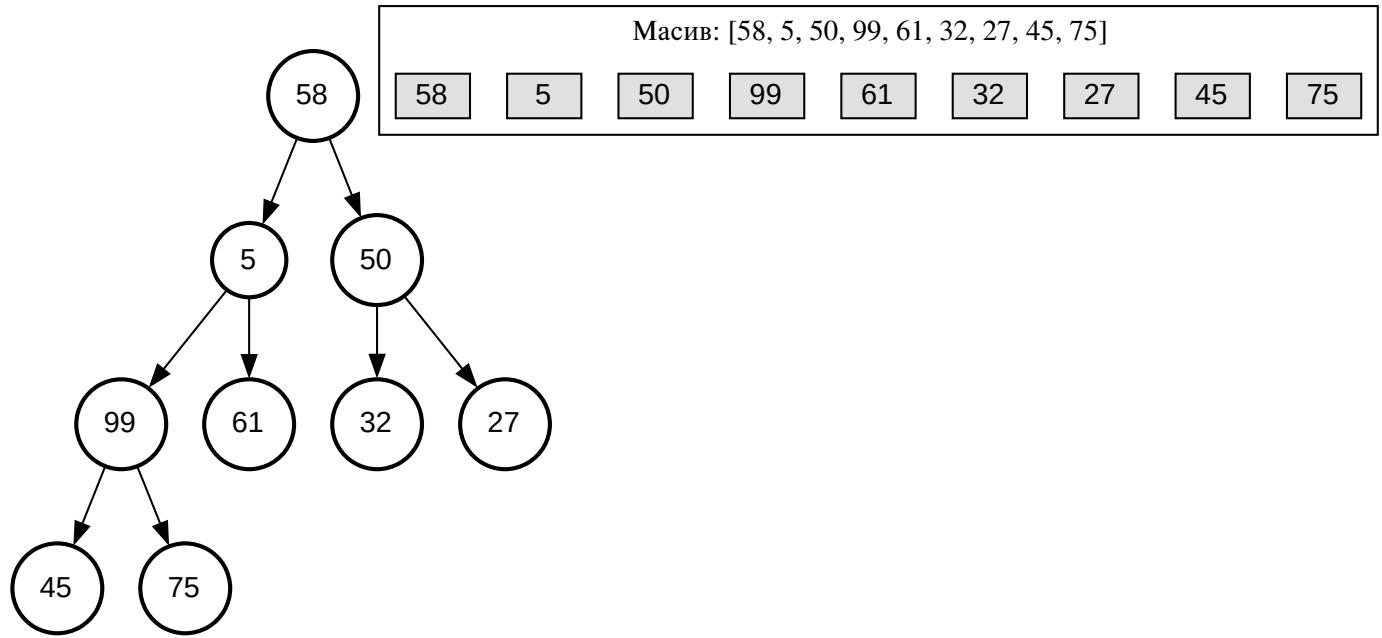


Рисунок 2 – Представлення елементів масиву у вигляді вершин та ребер дерева.

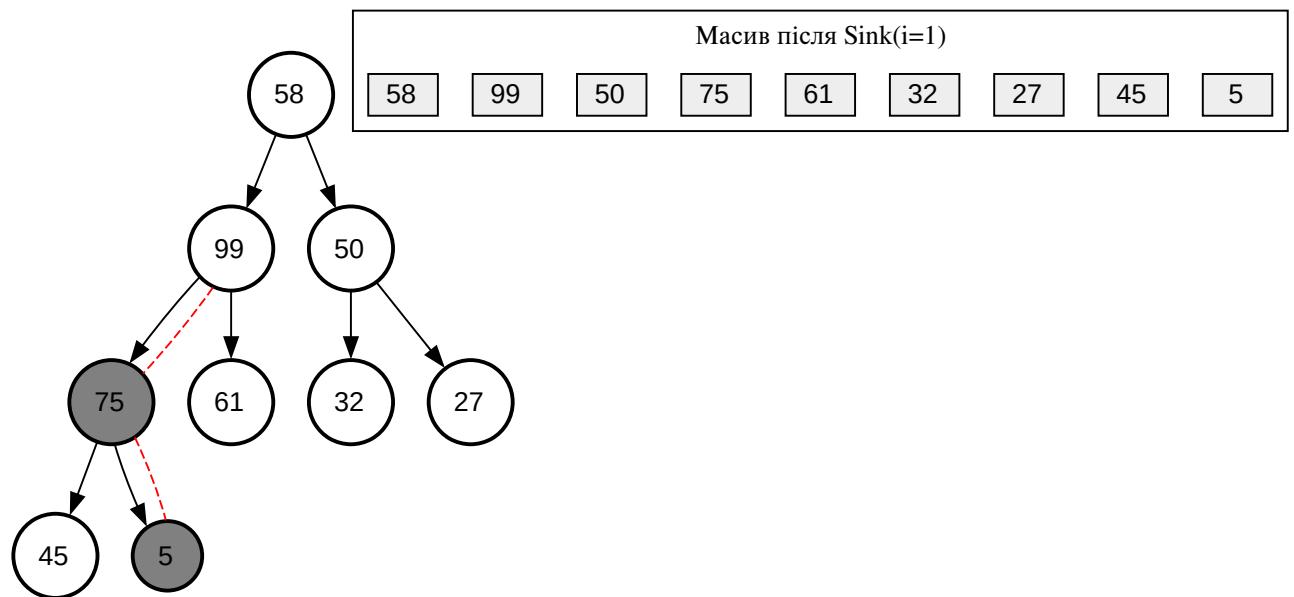
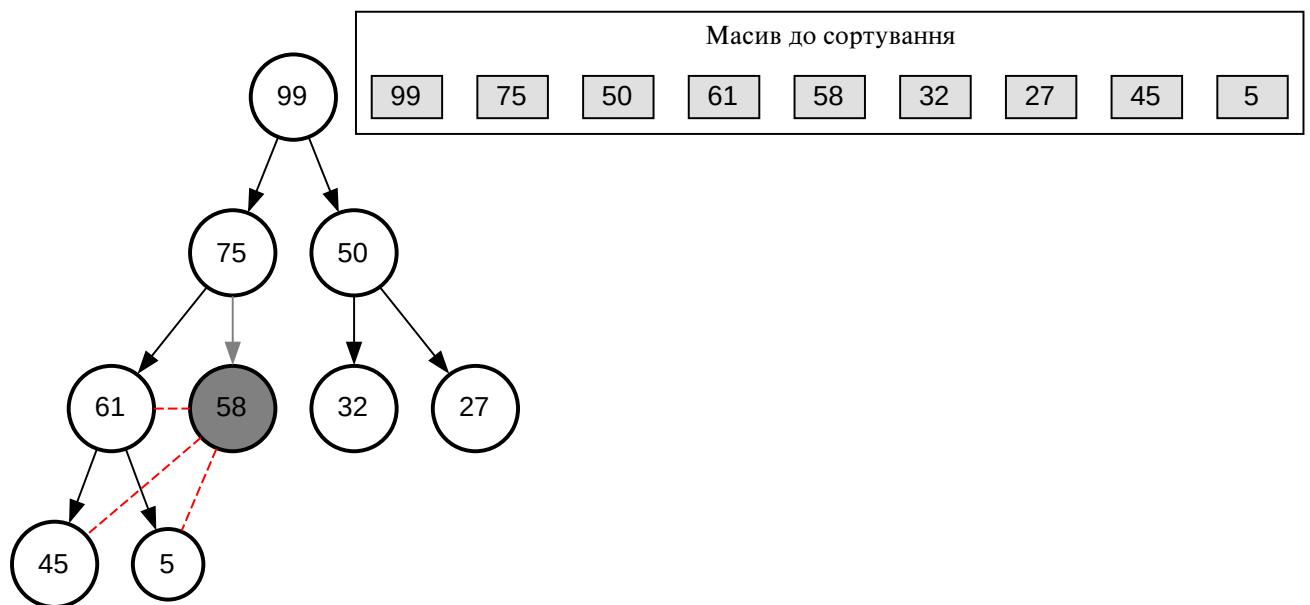
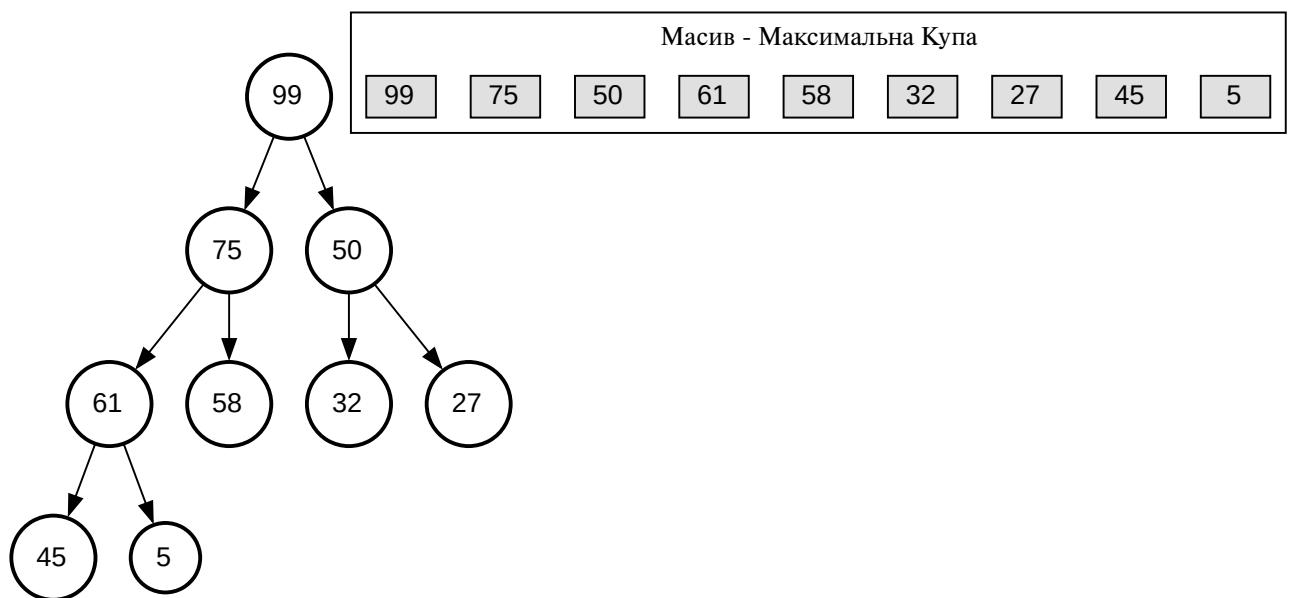


Рисунок 3 – Візуалізація створення максимальної купи на $h - 1$ рівні.



a



b

Рисунок 4 – Візуалізація створення максимальної купи на $h - 2$ рівні

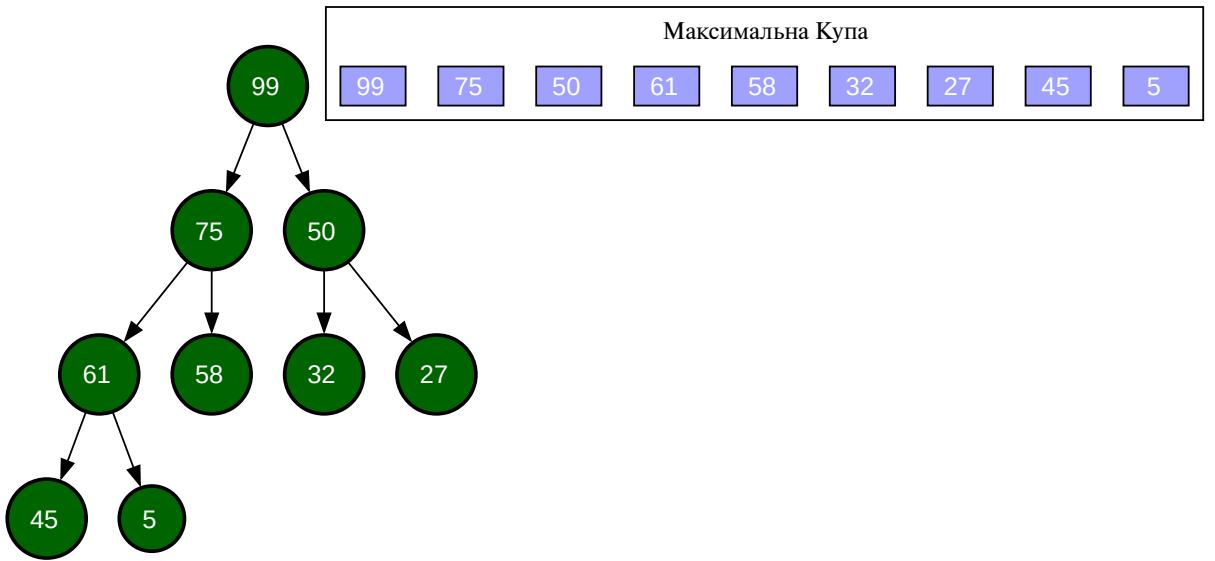


Таблица 5 Використовуючи процедуру Sink(A, i, n), підраховуємо кількість операцій.

Тип операції	Фаза 1	Фаза 2 (Sink)	Фаза 2 (Swap)	ВСЬОГО
Обміни (Swap)	5	5	8	18
Порівняння (Compares)	21	24	0	45

Рисунок 5 – Візуалізація створення максимальної купи на $h = 3$ (кореневому) рівні

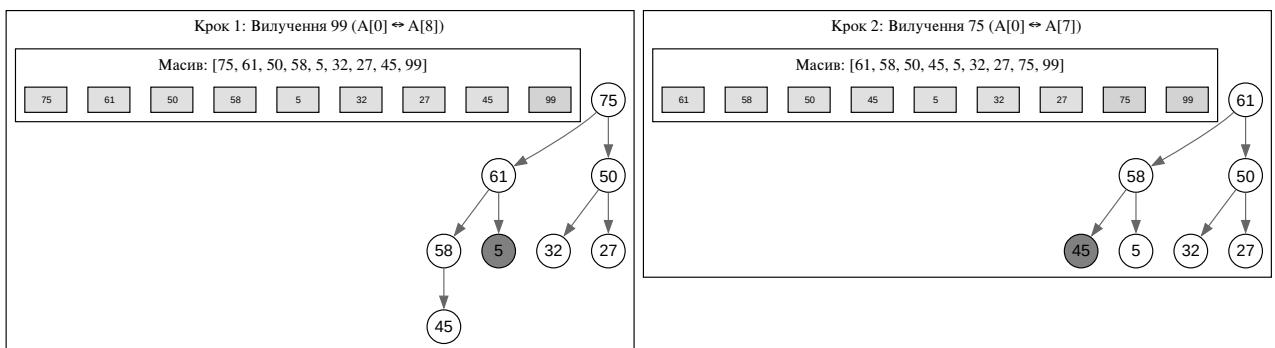


Рисунок 6 – Візуалізація другої фази сортування купою (занурення).

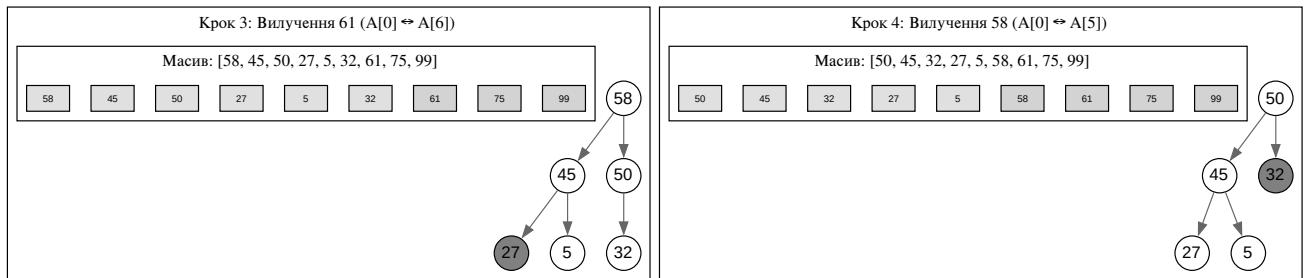
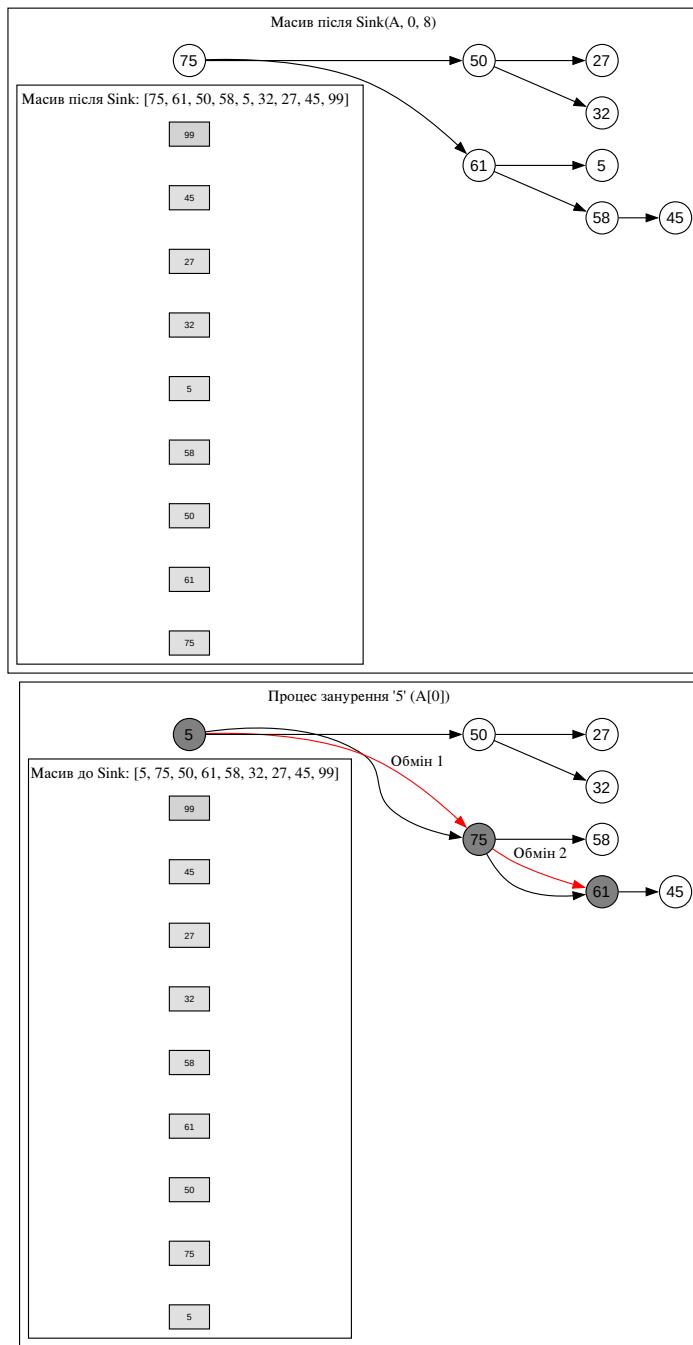


Рисунок 7 – Продовження візуалізації другої фази сортування купою.



Таблиця 6 Використовуючи процедуру $\text{Sink}(A, 0, n)$, підрахуємо кількість операцій.

Тип операції	Розрахунок	ВСЬОГО
Обміни (Swap)	$8 (\text{Swap}(A[0], A[i])) + 5 (\text{Swap y Sink})$	13
Порівняння (Compares)	24 (Усі порівняння всередині 8 викликів Sink)	24

Таблиця 7 Результати порівнянь сортування купою (за порівняннями та присвоюваннями) із сортуванням за алгоритмами ЛР1 та ЛР2

Назва алгоритму	Реалізація	Складність (Теорія)	Порівняння	Присвоювання / Обміни	Додаткові лічильники
Selection Sort (ЛР1)	Ітеративна	$O(N^2)$	44	40	-
Insertion Sort (ЛР1)	Ітеративна	$O(N^2)$	24	58	-
Heap Sort	Ітеративна (Sink)	$O(N \log N)$	45	18	-
QuickSort (ЛР2)	Рекурсивна	$O(N \log N)_{(Avg)}$	65	103	Рекурсивних викликів: 8
Merge Sort (ЛР2)	Рекурсивна	$O(N \log N)$	110	119	Рекурсивних викликів: 17
Merge Sort (ЛР2)	Ітеративна	$O(N \log N)$	99	206	-

Висновок:

1. На основі теоретичного аналізу та емпіричних трасувань для масиву N=9 (включно з результатами Heap Sort), доведено, що логарифмічні алгоритми сортування ($O(N \log N)$) демонструють чіткі компроміси між швидкістю, надійністю та витратами пам'яті, що впливають на їх практичну придатність.

1. Компроміс між Швидкістю та Надійністю

- Надійність (Merge Sort): Алгоритм Merge Sort підтверджує свою теоретичну стійкість, маючи гарантовану часову складність $O(N \log N)$ у найгіршому випадку. Емпіричні дані показують його стабільність, проте він є найбільш витратним за пам'яттю через необхідність допоміжного масиву, що особливо помітно в Ітеративному Merge Sort з його найвищою кількістю присвоювань (206 для N=9).
- Ризик (QuickSort): QuickSort підтверджує свою чутливість до вхідних даних (особливо зі статичним опорним елементом).Хоча в середньому він швидкий, ризик зростання складності до $O(N^2)$ (через нерівномірне розбиття) робить його менш надійним для критичних додатків без застосування складніших стратегій вибору опорного елемента ("медіана трьох").

2. Позиція Heap Sort

- Heap Sort демонструє хороший баланс між швидкістю та ресурсами, маючи гарантовану складність $O(N \log N)$ та ефективне використання пам'яті ($O(1)$ додаткової пам'яті).
- Емпіричні дані підтверджують його ефективність за порівняннями (45 для N=9) та високу економію обмінів (18 для N=9), що робить його чудовим вибором для систем із обмеженою пам'яттю, де потрібна гарантована продуктивність без додаткових витрат.

3. Фінальний Підсумок

- Критичні системи: Merge Sort є кращим вибором, коли потрібна гарантована продуктивність ($O(N \log N)$) і стабільність сортування.
- Загальне застосування: QuickSort залишається оптимальним для загальних задач, де середня швидкість важливіша, але слід мінімізувати ризик $O(N^2)$.
- Обмеження пам'яті: Heap Sort є ідеальним компромісом, оскільки забезпечує гарантовану $O(N \log N)$ складність без потреби в додатковій пам'яті.

Посилання на GitHub:

<https://github.com/AlexKim71/Theory-of-Algorithms/tree/main>