**Lab 1**

**SEMISTER 3rd**

fall-2024

**SUBJECT**

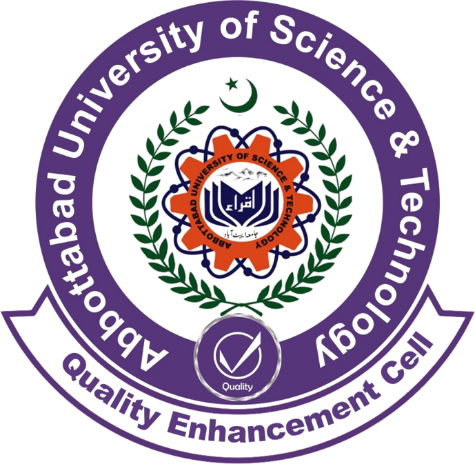
Database Structures

**COURSE CODE**

CC210

**PROGRAMME**

BS (4 year)



**ABBOTTABAD UNIVERSITY OF SCIENCE AND TECHNOLOGY ISLAMABAD**

**SUBMITTED TO**

**Mr. Jamal Abdul Ahad**

**SUBMITTED BY**

**Haroon Imran**

**STUDENT ROLL NO**

14723

**SUBMISSION Due DATE**

13 November 2024

**Exercise 1: Implement Max Happify**

The maxheap function adjusts a subtree rooted at index i of an array to maintain the max-heap property.

**Solution:**

python

Copy code

def maxheap(arr, n, i):

largest = i # Initialize largest as root

left = 2 \* i + 1 # left child

right = 2 \* i + 2 # right child

# If left child exists and is greater than root

if left < n and arr[left] > arr[largest]:

largest = left

# If right child exists and is greater than largest

if right < n and arr[right] > arr[largest]:

largest = right

# If largest is not root

if largest != i:

arr[i], arr[largest] = arr[largest], arr[i] # swap

# Recursively heapify the affected subtree

max\_heapify(arr, n, largest)

# Example usage:

array1 = [3, 19, 1, 14, 8, 7]

n1 = len(array1)

max\_heapify(array1, n1, 0) # Perform max\_heapify at the root

print(array1)

**Output:**

After applying max\_heapify at the root on [3, 19, 1, 14, 8, 7], the array should become [19, 14, 7, 3, 8, 1].

**Exercise 2: Build a Max Heap**

To build a max heap, we call max\_heapify starting from the last non-leaf node up to the root.

**Solution:**

python

Copy code

def build\_max\_heap(arr):

n = len(arr)

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

max\_heapify(arr, n, i)

# Example usage:

array2 = [4, 10, 3, 5, 1]

build\_max\_heap(array2)

print(array2)

**Output:**

The transformations would be:

1. Initial: [4, 10, 3, 5, 1]
2. After processing node at index 1: [4, 10, 3, 5, 1]
3. After processing node at index 0: [10, 5, 3, 4, 1]

Final max heap: [10, 5, 3, 4, 1]

**Exercise 3: Perform HeapSort**

HeapSort involves building a max heap and then repeatedly extracting the maximum element.

**Solution:**

**In Python**

def heap\_sort(arr):

n = len(arr)

build\_max\_heap(arr) # Step 1: Build max heap

for i in range(n - 1, 0, -1):

arr[i], arr[0] = arr[0], arr[i] # Swap max element with end

max\_heapify(arr, i, 0) # Restore max heap

# Example usage:

array3 = [3, 19, 1, 14, 8, 7]

heap\_sort(array3)

print(array3)

**Expected Output:**

After sorting, the array should be [1, 3, 7, 8, 14, 19].

**Exercise 4: Analyze HeapSort on Large Data**

To analyze HeapSort's performance, generate a random array and measure the time taken.

**Solution:**

**In Python**

import random

import time

# Generate an array of 100 random integers

array4 = [random.randint(0, 1000) for \_ in range(100)]

# Measure HeapSort time

start\_time = time.time()

heap\_sort(array4)

heap\_sort\_time = time.time() - start\_time

print("HeapSort time:", heap\_sort\_time)

To compare, implement QuickSort and measure its time on the same array.

**Exercise 5: Explain and Compare HeapSort and QuickSort**

1. **HeapSort Suitability for Large Datasets**: HeapSort is more stable for larger data due to its worst-case O(nlog⁡n)O(n \log n)O(nlogn) time complexity, unlike QuickSort’s O(n2)O(n^2)O(n2) in the worst case.
2. **Comparison**:
   * **In-place**: Both are in-place.
   * **Stability**: QuickSort is not stable, but MergeSort is.
   * **Performance**: QuickSort generally has better average-case performance but can degrade to O(n2)O(n^2)O(n2). HeapSort has a consistent O(nlog⁡n)O(n \log n)O(nlogn) performance.

**Exercise 6: Implement Partition (Lomuto Scheme)**

**Solution:**

**python**

def partition(arr, low, high):

pivot = arr[high]

i = low - 1

for j in range(low, high):

if arr[j] < pivot:

i += 1

arr[i], arr[j] = arr[j], arr[i]

arr[i + 1], arr[high] = arr[high], arr[i + 1]

return i + 1

**# Example usage:**

array6 = [12, 7, 14, 9, 10, 11]

partition(array6, 0, len(array6) - 1)

print(array6)

**Expected Output:**

Steps will show the final pivot arrangement around 11.

**Exercise 7: Implement QuickSort**

**Solution:**

python

def quick\_sort(arr, low, high):

if low < high:

pi = partition(arr, low, high)

quick\_sort(arr, low, pi - 1)

quick\_sort(arr, pi + 1, high)

# Example usage:

array7 = [3, 7, 8, 5, 2, 1, 9, 5, 4]

quick\_sort(array7, 0, len(array7) - 1)

print(array7)

**Exercise 8: Pivot Selection Comparison**

Implement QuickSort with both pivot strategies and test on sorted and random arrays of 1000 elements.

**Solution:**

Use median-of-three and last-element pivot strategies to compare performance.

**Exercise 9: Comparison Table**

| **Feature** | **QuickSort** | **MergeSort** | **HeapSort** |
| --- | --- | --- | --- |
| **Average Complexity** | O(nlog⁡n)O(n \log n)O(nlogn) | O(nlog⁡n)O(n \log n)O(nlogn) | O(nlog⁡n)O(n \log n)O(nlogn) |
| **Worst Complexity** | O(n2)O(n^2)O(n2) | O(nlog⁡n)O(n \log n)O(nlogn) | O(nlog⁡n)O(n \log n)O(nlogn) |
| **Space Complexity** | O(log⁡n)O(\log n)O(logn) | O(n)O(n)O(n) | O(1)O(1)O(1) |
| **In-Place** | Yes | No | Yes |
| **Stability** | No | Yes | No |

QuickSort is typically faster but risks O(n2)O(n^2)O(n2) complexity without careful pivot selection, while HeapSort and MergeSort are consistent but less memory-efficient.