**Assignment 4: Heap Data Structure: Implementation, Analysis, and Applications**

Heapsort Implementation and Analysis

1. Implementation

- Implement the Heapsort algorithm using Python. Ensure your implementation is clear, efficient, and follows the correct steps for building a max-heap, extracting the maximum element, and maintaining the heap property.

**Code:**

def heapify(arr, n, i):

largest\_value = i

left\_element = 2 \* i + 1

right\_element = 2 \* i + 2

if left\_element < n and arr[left\_element] > arr[largest\_value]:

largest\_value = left\_element

if right\_element < n and arr[right\_element] > arr[largest\_value]:

largest\_value = right\_element

if largest\_value != i:

arr[i], arr[largest\_value] = arr[largest\_value], arr[i]

heapify(arr, n, largest\_value)

def heapsort\_array(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)

# added random values

arr = [56, 10, 40, 80, 5, 58, 25, 89]

print("Befor Sorting", arr)

heapsort\_array(arr)

# Display Sorted Value

print("After Sorting array is", arr)

2. Analysis of Implementation

* Time Complexity:

- Building the Heap: O(n)  
- Extracting the Maximum Element: O(log n)  
- Total Time Complexity: O(n log n) in all cases

* Space Complexity:   
    
  - In-place Sorting: Heapsort uses O(1) auxiliary space since the sorting is done within the input array.

3. Comparison with Other Sorting Algorithms:

Compare the running times of Heapsort, Quicksort, and Merge Sort using different input sizes and distributions.

**Code:**

import time

import random

def generate\_arrays(size):

arr\_random = random.sample(range(size \* 10), size)#Random

arr\_sorted = sorted(arr\_random)# Sorted

arr\_reverse\_sorted = sorted(arr\_random, reverse=True)# Reverse sorted

return arr\_random, arr\_sorted, arr\_reverse\_sorted

# Merge Sort Function

def merge\_sort(arr):

if len(arr) > 1:

mid\_element = len(arr) // 2

left\_element = arr[:mid\_element]

right\_element = arr[mid\_element:]

merge\_sort(left\_element)

merge\_sort(right\_element)

i = j = k = 0

while i < len(left\_element) and j < len(right\_element):

if left\_element[i] < right\_element[j]:

arr[k] = left\_element[i]

i += 1

else:

arr[k] = right\_element[j]

j+=1

k += 1

while i < len(left\_element):

arr[k] = left\_element[i]

i += 1

k += 1

while j < len(right\_element):

arr[k] = right\_element[j]

j += 1

k += 1

# Heapsort Function

def heapify(arr, n, i):

largest = i

left\_element = 2 \* i + 1

right\_element = 2 \* i + 2

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

largest = left\_element

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

largest = right\_element

if largest != i:

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

heapify(arr, n, largest)

def heapsort(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)

# Quicksort Function

def quicksort(arr):

if len(arr) <= 1:

return arr

pivot = arr[len(arr) // 2]

left\_element = [x for x in arr if x < pivot]

mid\_elementdle = [x for x in arr if x == pivot]

right\_element = [x for x in arr if x > pivot]

return quicksort(left\_element) + mid\_elementdle + quicksort(right\_element)

#comparing all sorting

def compare\_sorts():

sizes = [500, 3000, 75000, 13000, 16500]

for size in sizes:

arr\_random, arr\_sorted, arr\_reverse\_sorted = generate\_arrays(size)

for distribution, arr in zip(["Random", "Sorted", "Reverse-sorted"], [arr\_random, arr\_sorted, arr\_reverse\_sorted]):

print(f"\nSize of Array: {size}, Distribution: {distribution}")

# Merge Sort

start\_time = time.time()

merge\_sort(arr.copy())

end\_time = time.time()

print(f"Time of Merge Sort: {end\_time - start\_time:.6f} seconds")

# Quicksort

start\_time = time.time()

quicksort(arr.copy())

end\_time = time.time()

print(f"Time of Quicksort: {end\_time - start\_time:.6f} seconds")

# Heapsort

start\_time = time.time()

heapsort(arr.copy())

end\_time = time.time()

print(f"Time of Heapsort: {end\_time - start\_time:.6f} seconds")

# Timsort

start\_time = time.time()

sorted(arr.copy())

end\_time = time.time()

print(f"Time of Timsort: {end\_time - start\_time:.6f} seconds")

# compare sorting algorithm

compare\_sorts()

**Priority Queue Implementation and Applications**

**Part A: Priority Queue Implementation**

1. Data Structure:

Which Data structure?

- Here I am Choosing Array Data Structure for Heap

Why? (Implementation)

- Memory Efficiency: Arrays allow for a compact memory representation because the stack is a complete binary tree.

- Efficient use of space: Because pointers do not require additional memory. Arrays therefore provide good space efficiency for binary heaps.

**Code:**

class Task:

def \_\_init\_\_(self, task\_id, priority, arrival\_time, deadline):

self.task\_id = task\_id

self.priority = priority# Priority level

self.arrival\_time = arrival\_time

self.deadline = deadline

def \_\_repr\_\_(self):

return f"Task(ID: {self.task\_id}, Priority: {self.priority}, Arrival: {self.arrival\_time}, Deadline: {self.deadline})"

class PriorityQueue:

def \_\_init\_\_(self, max\_heap=True):

self.heap = []

self.max\_heap = max\_heap

def insert\_task(self, task):

self.heap.append(task)

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

def max\_or\_min\_task(self):

if self.is\_empty():

return None

self.heap[0], self.heap[-1] = self.heap[-1], self.heap[0]

max\_or\_min\_task = self.heap.pop()

self.\_heapify\_down(0)

return max\_or\_min\_task

def increase\_or\_decrease\_key(self, task\_id, new\_priority):

# Find the task

for idx, task in enumerate(self.heap):

if task.task\_id == task\_id:

old\_priority = task.priority

task.priority = new\_priority

if (self.max\_heap and new\_priority > old\_priority) or (not self.max\_heap and new\_priority < old\_priority):

self.\_heapify\_up(idx)

else:

self.\_heapify\_down(idx)

return task

return None

def \_heapify\_up(self, idx):

parent\_idx = (idx - 1) // 2

while idx > 0 and self.compare\_task(self.heap[idx], self.heap[parent\_idx]):

self.heap[idx], self.heap[parent\_idx] = self.heap[parent\_idx], self.heap[idx]

idx = parent\_idx

parent\_idx = (idx - 1) // 2

def \_heapify\_down(self, idx):

size = len(self.heap)

while idx < size:

left\_child = 2 \* idx + 1

right\_child = 2 \* idx + 2

if left\_child >= size:

break

if right\_child < size and self.compare\_task(self.heap[right\_child], self.heap[left\_child]):

child\_idx = right\_child

else:

child\_idx = left\_child

if self.compare\_task(self.heap[child\_idx], self.heap[idx]):

self.heap[idx], self.heap[child\_idx] = self.heap[child\_idx], self.heap[idx]

idx = child\_idx

else:

break

def compare\_task(self, task1, task2):

if self.max\_heap:

return task1.priority > task2.priority#Max heap comparison

else:

return task1.priority < task2.priority# min heap comparison

def is\_empty(self):

return len(self.heap) == 0

pq = PriorityQueue(max\_heap=True)#max-heap priority

pq.insert\_task(Task(101, 10, 0, 20))

pq.insert\_task(Task(102, 15, 4, 25))

pq.insert\_task(Task(103, 5, 7, 30))

print("Heap\_after\_insertions:", pq.heap)

print("Maximum priority task:", pq.max\_or\_min\_task())

print("Heap after extraction:", pq.heap)

pq.increase\_or\_decrease\_key(103, 20)#Increase priority

print("Heap after increasing task 103 priority:", pq.heap) #print in

print("Is heap empty?", pq.is\_empty())

GitHub Link : <https://github.com/Tushar-4747/MSCS532_Assignment4.git>