# Assignment - 4

# Vijaya Krishna Sameeraj Jonnavithula

# Student ID : 005029574

# Course : 2024 Fall - Algorithms and Data Structures (MSCS-532-B01) - Second Bi-term

# Date : November 10, 2024

Github : https://github.com/VijayaKrishnaSameerajJonnavithula/Assignment-4

**Heapsort Implementation and Analysis :**Code :

def heapify(arr, n, i):

largest = i # Initialize largest as root

left = 2 \* i + 1 # Left child

right = 2 \* i + 2 # Right child

# Check if left child exists and is greater than root

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

largest = left

# Check if right child exists and is greater than current largest

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

largest = right

# Change root if needed

if largest != i:

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

heapify(arr, n, largest) # Recursively heapify the affected sub-tree

def heapsort(arr):

n = len(arr)

# Build a max-heap

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

heapify(arr, n, i)

# Extract elements one by one from the heap

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

arr[i], arr[0] = arr[0], arr[i] # Swap the root with the end

heapify(arr, i, 0) # Heapify the reduced heap

# Example usage

arr = [12, 11, 13, 5, 6, 7]

heapsort(arr)

print("Sorted array is:", arr)

A screenshot of a computer program

Description automatically generated

A black screen with colorful text

Description automatically generated

**Analysis of Implementation :**

Time Complexity Analysis:

* Constructing the Max-Heap: This process requires O(n) time. Heapify can be called n times, with each call taking up to O(logn) of time.
* Removing Components and Preserving the Heap Property: The loop executes n times, and heapify requires O(logn) time for each extraction. As a result, this step takes O(nlogn) time.

Total Time Complexity:

* The worst scenario is 𝑂 (𝑛 log ⁡ 𝑛).
* O(nlogn) Mean Situation: 𝑂 (𝑛 log ⁡ 𝑛) O(nlogn)
* In the best scenario, 𝑂(𝑛 log ⁡ 𝑛) O(nlogn)

An explanation of O(nlogn) Complexity for 𝑂(𝑛 log):

Each removal operation using Heapsort always takes O(logn) time, with n removals required to completely sort the array.

Complexity of Space:

* Since Heapsort is an in-place sorting algorithm (no extra arrays are required), its space complexity is O(1).
* Minimal Auxiliary Space; constant space is used for indexing and switching

**Comparison with Other Sorting Algorithms**

We are comparing Heap, Quick and Merge sort:   
  
Code :   
  
import time

import random

from Heap import heapsort

# Sample implementations of Quicksort and Merge Sort

def quicksort(arr):

if len(arr) <= 1:

return arr

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

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

middle = [x for x in arr if x == pivot]

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

return quicksort(left) + middle + quicksort(right)

def merge\_sort(arr):

if len(arr) <= 1:

return arr

mid = len(arr) // 2

left = merge\_sort(arr[:mid])

right = merge\_sort(arr[mid:])

return merge(left, right)

def merge(left, right):

result = []

i = j = 0

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

if left[i] < right[j]:

result.append(left[i])

i += 1

else:

result.append(right[j])

j += 1

result.extend(left[i:])

result.extend(right[j:])

return result

# Timing utility

def time\_sorting\_algorithm(algorithm, arr):

start\_time = time.time()

algorithm(arr)

end\_time = time.time()

return end\_time - start\_time

# Test on different input types and sizes

sizes = [1000, 5000, 10000]

input\_types = ["random", "sorted", "reverse"]

for size in sizes:

for input\_type in input\_types:

if input\_type == "random":

arr = [random.randint(0, 100000) for \_ in range(size)]

elif input\_type == "sorted":

arr = list(range(size))

elif input\_type == "reverse":

arr = list(range(size, 0, -1))

arr\_copy = arr.copy()

print(f"Size: {size}, Input: {input\_type}")

print("Heapsort:", time\_sorting\_algorithm(heapsort, arr.copy()))

print("Quicksort:", time\_sorting\_algorithm(quicksort, arr.copy()))

print("Merge Sort:", time\_sorting\_algorithm(merge\_sort, arr\_copy))

print()

A screenshot of a computer screen

Description automatically generated

A screen shot of a computer

Description automatically generated

Examination of the Found Outcomes:

* Although Heapsort consistently performs well across input distributions, its higher constant factors cause it to operate more slowly than Quicksort.
* Quicksort often operates more quickly for random data, but for sorted or reverse-sorted data (as prevented by use randomized pivots), it may deteriorate to O(n 2).
* Merge Sort consumes 𝑂 (𝑛) O(n) more space but has 𝑂 (𝑛 log ⁡𝑛) O(nlogn) complexity for all scenarios.

**Priority Queue Implementation and Applications**

Data Structure Selection and Rationale

* Utilized Data Structure: Array-based implementation of a binary heap (or Python list).
* Reasoning: Binary heaps are ideally suited for arrays and lists because they allow for the representation of parent-child connections through straightforward index computations. The array operations are localized and simple to handle, hence this method guarantees efficient 𝑂 (log ⁡ 𝑛) O(logn) time complexity for insertions and extractions.

Code:   
  
class Task:

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

self.task\_id = task\_id

self.priority = priority

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):

self.heap = []

def insert(self, task):

# Add the new task to the end of the heap

self.heap.append(task)

# Maintain the max-heap property

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

def \_bubble\_up(self, index):

while index > 0:

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

if self.heap[index].priority > self.heap[parent\_index].priority:

# Swap if the current node's priority is greater than the parent's

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

index = parent\_index

else:

break

def extract\_max(self):

if len(self.heap) == 0:

return None

# Swap the root with the last element

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

max\_task = self.heap.pop()

# Restore the heap property

self.\_bubble\_down(0)

return max\_task

def \_bubble\_down(self, index):

n = len(self.heap)

while 2 \* index + 1 < n:

left = 2 \* index + 1

right = 2 \* index + 2

largest = index

if left < n and self.heap[left].priority > self.heap[largest].priority:

largest = left

if right < n and self.heap[right].priority > self.heap[largest].priority:

largest = right

if largest != index:

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

index = largest

else:

break

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

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

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

if new\_priority > task.priority:

task.priority = new\_priority

self.\_bubble\_up(i)

else:

task.priority = new\_priority

self.\_bubble\_down(i)

return True

return False # Task not found

def is\_empty(self):

return len(self.heap) == 0

# Example usage

pq = PriorityQueue()

pq.insert(Task(1, 10, '10:00', '12:00'))

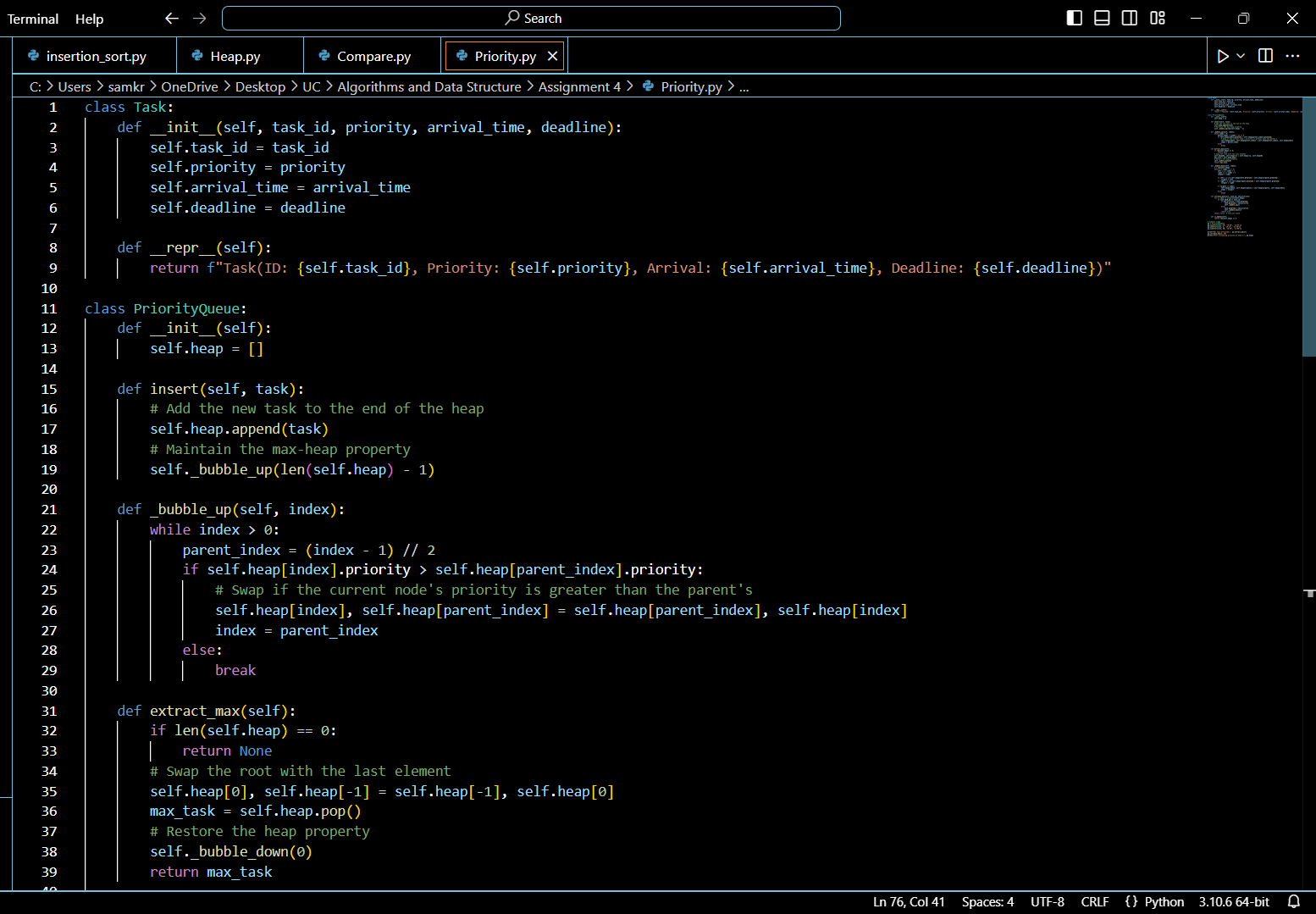
pq.insert(Task(2, 5, '10:15', '12:30'))

pq.insert(Task(3, 20, '10:30', '11:00'))

print("Max task extracted:", pq.extract\_max())

pq.increase\_key(2, 15)

print("After increasing priority of task 2:", pq.heap)



A computer screen with text on it

Description automatically generated

Design of Task Classes

Task ID, priority, arrival time, and deadline are among the relevant details for tasks that are stored in the Task class.

Max-Heap vs. Min-Heap Option: The highest-priority task is extracted first using a max-heap.

Rationale: This is in line with scheduling techniques that prioritise higher-priority jobs (such as CPU scheduling).

A black screen with text on it

Description automatically generated

Fundamental Functions

Operation Insert (insert(task))

A new job is inserted by adding it to the heap array's end and then "bubbling up" to preserve the heap property.

Analysis of Time Complexity: 𝑂 (log ⁡ 𝑛)

O(logn)

A black background with a black square

Description automatically generated with medium confidence

Operation to Extract Max (extract\_max())

The process of deleting the job with the highest priority entails "bubbling down" to restore the heap property after replacing the first element (root) with the last element.

Analysis of Time Complexity: 𝑂 (log ⁡ 𝑛)

O(logn)

A computer screen with text

Description automatically generated

The operation "increase/decrease\_key(task, new\_priority)" increases or decreases the key.

Locate the task, change its priority, and bubble up or bubble down to preserve the heap property in order to change the priority of an existing job.

Analysis of Time Complexity: 𝑂 (log ⁡ 𝑛)

O(logn)

A computer screen with text on it

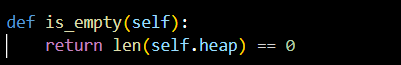
Description automatically generated

Is Empty Operation (is\_empty())

A simple check to determine if the priority queue is empty.

Time Complexity Analysis:

𝑂(1)



An overview of the time complexity

* Operation Insertion: O(logn) 𝑂 (log ⁡ 𝑛)
* Extract Maximum/Minimum The operation is 𝑂 (log ⁡ 𝑛) O(logn).
* Rise/Ffall Crucial Function: O(logn) 𝑂 (log ⁡ 𝑛)
* Is the operation empty? 𝑂 (1 ) O(1)

**REFERENCE**

Python Official Documentation:

Python Lists: Useful for understanding how lists work, as lists are used to implement heaps.

Python's heapq Module: Provides built-in functions for min-heaps, which can be adapted for max-heaps.

Algorithm and Data Structure Textbooks:

"Introduction to Algorithms" by Cormen, Leiserson, Rivest, and Stein: A comprehensive textbook that covers heaps and priority queues in detail, including theoretical aspects and pseudocode.

"Algorithms, 4th Edition" by Robert Sedgewick and Kevin Wayne: A practical book that explains sorting and priority queues, with illustrations and code examples.

Online Tutorials and Articles:

GeeksforGeeks Priority Queue: Link: Provides explanations and examples for implementing priority queues using binary heaps.

Educative.io Courses: Offers detailed courses on data structures and algorithms, including heaps and priority queues.