**DS-II Project Report**

**Group members:**

* **Behjet Ansari (ba04079)**
* **Laiba Fatima Khan (lk04067)**

**Instructor: Dr. Shahid Hussain**

**Course: Data Structures II**

**Date: 29th April, 2019.**

**Requirement:**

Use two different data structures to solve a real-world problem and compare the efficiency of data structures.

**Proposal:**

We chose a practical business-related example of patients being tackled in a hospital according to their injury levels. We are going to implement the working and solution using ***Priority Queue*** and ***Array-based List*** data structures. The scenario below further explains our idea in detail.

**Scenario:**

Patients are coming in at a steady rate in the hospital and there is only one doctor available. There are patients of various levels of injuries so to ensure optimum care for every patient, they are prioritized according to their injury levels. For example, the first patient who walks in is served immediately. A person with a pneumonia comes in next and requires a check-up. You add him to the queue and he waits in line for the doctor to become available. Next, a man with a severe injury comes through the door. He is assigned a higher priority because he has a higher medical liability. So, the man with the cold is bumped down in line. Similarly, as more patients stand in the queue, their priority is decided and they shift places in the queue accordingly.

**Approach:**

We have designed our program with priorities 0 to 4, 0 being the highest injury level, 4 being the lowest.

* **Priority Queue Implementation:**

We have used Python-3 built-in Priority Queue.

**Add ():** For each new patient a priority is assigned, and data is entered in priority queue in form of tuples: (priority, patient\_id).

**Remove ():** Each time the slot is free and patient with highest priority goes for treatment, for those with same priority, treatment is first come first served based (lowest id goes first).

* **Array Based List Implementation:**

**Add ():** The data we enter is sorted, as patients are given increasing ID numbers as they come. For the Array-based List, we have 5 lists nested lists inside a list; the index of each representing priorities/injury levels. As patients come, they will be added into their respective nested list according to their priority.

**Remove ():** Each time the slot is free, patient with the highest priority (id in the lowest-index non-empty nested list) goes for treatment. For patients having same priority, treatment is first come first served based (nested lists are used as FIFO queues).

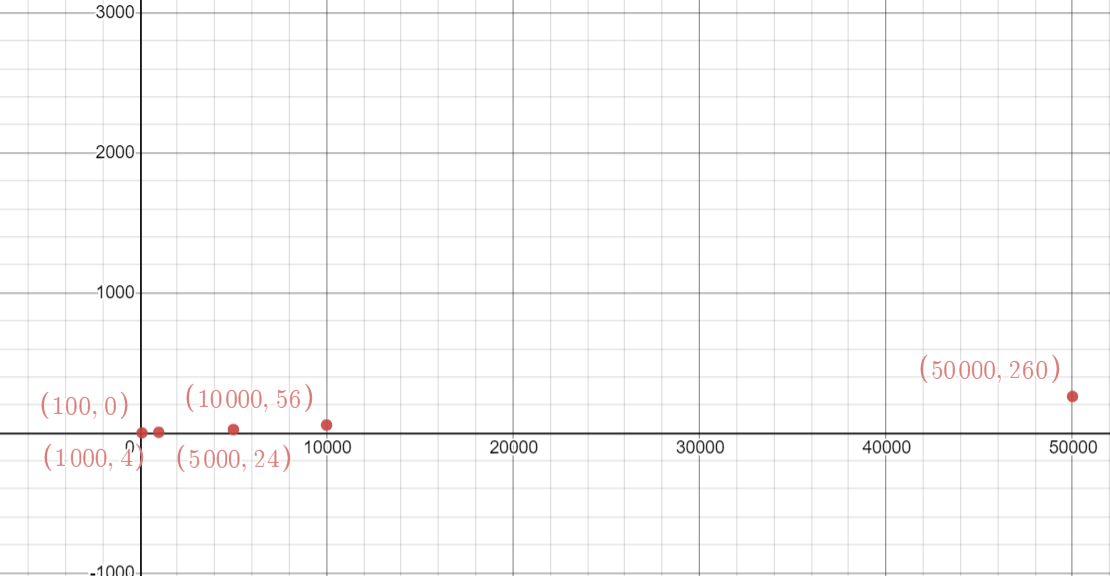
**Theoretical Time Complexities:**

|  |  |  |
| --- | --- | --- |
| **Operation** | **Priority\_Queue** | **Array\_Based List** |
| **Add ()** | **O(n)** | **O (1)** |
| **Remove ()** | **O (1)** | **O (1)** |

**Empirical Analysis on Machine 1:**

**Built-in Priority Queue Implementation:**

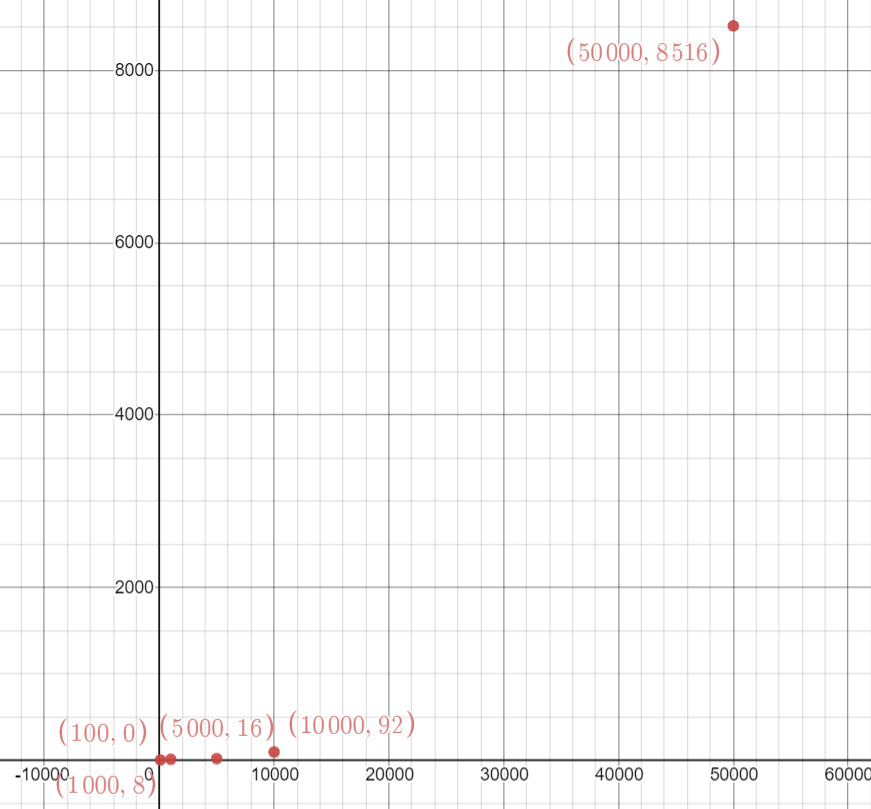
|  |  |  |  |
| --- | --- | --- | --- |
| **Add Inputs** | **Remove Inputs** | **Add () Time**  **m-s** | **Remove () Time**  **m-s** |
| **100** | **5** | **0** | **0** |
| **1000** | **10** | **4** | **0** |
| **5000** | **50** | **24** | **0** |
| **10000** | **100** | **56** | **0** |
| **50000** | **500** | **260** | **0** |
| **100000** | **1000** | **528** | **0** |

****

**Figure.1: Add Function**

**Array Based List Implementation:**

|  |  |  |  |
| --- | --- | --- | --- |
| **Add Inputs** | **Remove Inputs** | **Add () Time**  **m-s** | **Remove () Time**  **m-s** |
| **100** | **5** | **0** | **0** |
| **1000** | **10** | **8** | **0** |
| **5000** | **50** | **92** | **0** |
| **10000** | **100** | **340** | **0** |
| **50000** | **500** | **8516** | **4** |
| **100000** | **1000** | **35356** | **16** |

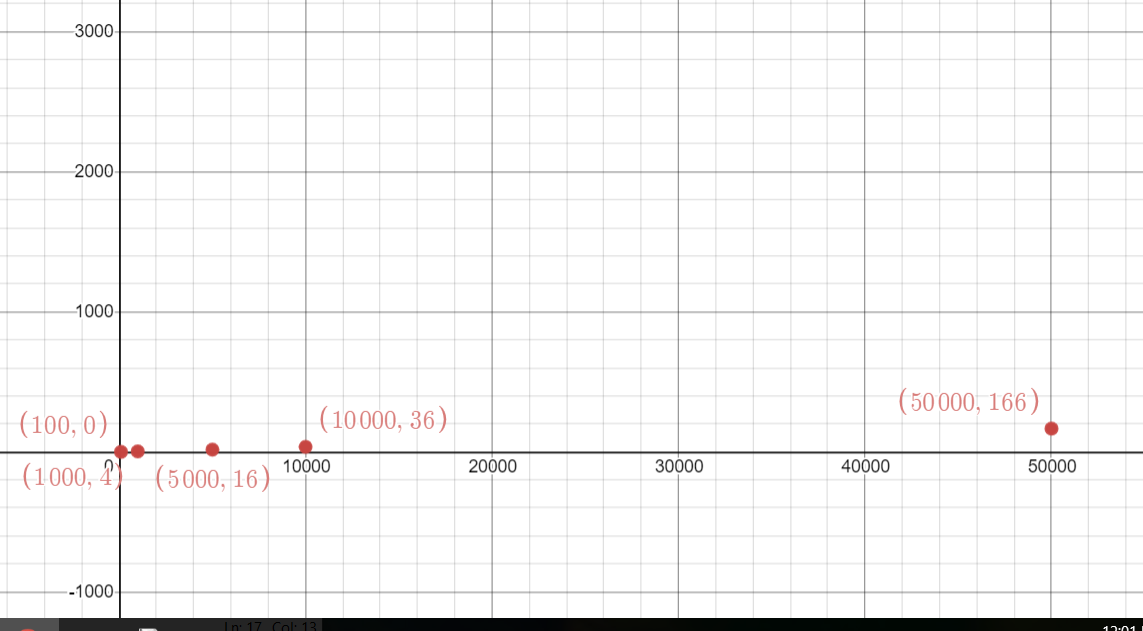
****

**Figure.2: Add Function**

**Empirical Analysis on Machine 2:**

**Built-in Priority Queue Implementation:**

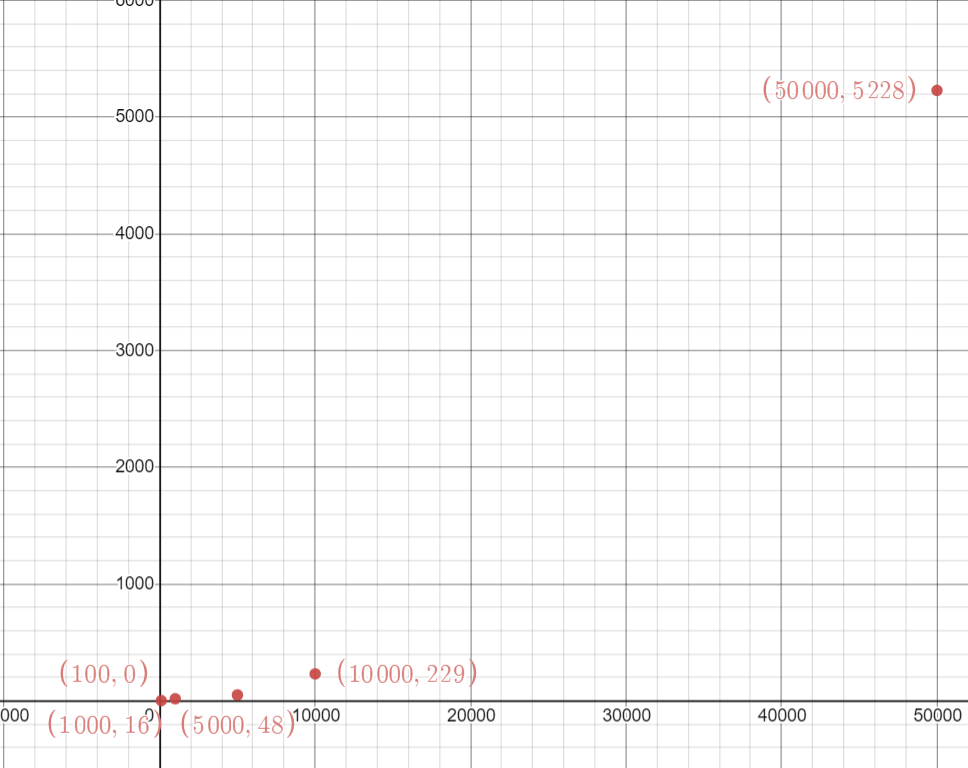
|  |  |  |  |
| --- | --- | --- | --- |
| **Add Inputs** | **Remove Inputs** | **Add () Time**  **m-s** | **Remove () Time**  **m-s** |
| **100** | **5** | **0** | **0** |
| **1000** | **10** | **4** | **0** |
| **5000** | **50** | **16** | **0** |
| **10,000** | **100** | **36** | **0** |
| **50,000** | **500** | **166** | **0** |
| **100,000** | **1000** | **558** | **0** |

****

**Figure.3: Add Function**

**Array Based List Implementation:**

|  |  |  |  |
| --- | --- | --- | --- |
| **Add Inputs** | **Remove Inputs** | **Add () Time**  **m-s** | **Remove () Time**  **m-s** |
| **100** | **5** | **0** | **0** |
| **1000** | **10** | **16** | **0** |
| **5000** | **50** | **48** | **0** |
| **10,000** | **100** | **229** | **0** |
| **50,000** | **500** | **5228** | **0** |
| **100,000** | **1000** | **9545** | **0** |

****

**Figure.4: Add Function.**

**Comparison of Efficiency:**

1. **Theoretical Comparison:**

**Add ()**: Theoretically, Add () runs in constant time for Array\_Based List and in O(n) time for Priority Queue, which means that Array\_Based List is more efficient for insertion.

**Remove():**

Theoretically, Remove() runs in constant time for both Array\_Based List and Priority Queue, which means that both data structures are equally efficient for deletion.

**2) Empirical Comparison:**

**Add ()**: Comparing Add () time for both data structures using same input values, we observed that for higher values, time taken by Array\_Based List is way higher than that for Priority Queue, making Priority Queue more efficient for insertion of large data.

**Remove ():** Comparing Remove () time for both data structures using same input values, we observed that both data structures run Remove () in constant time, hence Array\_Based List and Priority Queue both are equally efficient for deletion.

**Possible Reasoning for Difference in Theoretical and Empirical Analysis of Insertion:**

Built-in Priority Queue was found to be more efficient for large data insertion since it is highly optimized data structure, whereas our implementation was very basic and not at all optimized comparing to it.

**Our Conclusion after Efficiency Comparison:**

Array\_Based List and Priority Queue are equally efficient for Deletion but Priority Queue was observed to be more efficient for Insertion of large data. So overall comparison of Insertion/Deletion makes built-in Priority Queue a more efficient choice for implementation of our system.

**But,**

We are making this conclusion assuming that the data is sorted, which is actually playing a big role in optimization and efficiency of Priority Queue implementation. Since Array\_Based List is using FIFO queue for inner lists (patients with same priority) it will run in same time for both sorted and unsorted data. On the other hand, for implementation using Priority Queue, patients with same priorities are tackled using smallest IDs (patients were given IDs in ascending order) and there is no direct method of tackling same priorities given that data was unsorted (IDs were randomly assigned)

**Array\_Based\_List\_Implementation.py**

#!/bin/python3

import time

import math

import os

import random

import re

import sys

from enum import Enum

class Stopwatch(object):

def \_\_init\_\_(self):

"""Initialize a new Stopwatch, but do not start timing."""

self.start\_time = None

self.elapsed\_time = 0

def start(self):

"""Start timing."""

self.start\_time = time.time()\*1000

def stop(self):

"""Stop timing."""

if not (self.start\_time):

return

stop\_time = time.time()\*1000

self.elapsed\_time += int(round(stop\_time - self.start\_time))

self.start\_time = None

def time\_elapsed(self):

if (self.start\_time):

self.stop()

return self.elapsed\_time

def reset(self):

self.start\_time = None

self.elapsed\_time = 0

class PriorityQueue(object):

def \_\_init\_\_(self):

self.priority\_queue=[]

self.size=0

self.total\_injury\_levels=5

#since 5 injury levels:

for i in range(0,self.total\_injury\_levels):

self.priority\_queue.append([])

def priority(self,id):

return id%4

def add\_to\_pqueue(self,id):

priority=self.priority(id)

if not self.find\_id(id):

self.priority\_queue[priority].append(id)

self.size+=1

def find\_id(self,id):

priority=self.priority(id)

if id in self.priority\_queue[priority]:

return True

return False

def remove\_highest\_priority(self):

if self.size>0:

for i in range((self.total\_injury\_levels-1)):

if self.priority\_queue[i]:

self.size-=1

x = self.priority\_queue[i].pop(0)

return x

return "Priority Queue is Empty, no patient to handle."

def get\_size(self):

return self.size

if \_\_name\_\_ == '\_\_main\_\_':

watch\_add=Stopwatch()

watch\_find=Stopwatch()

watch\_remove= Stopwatch()

patients\_data = PriorityQueue()

####### ADD #######

add\_patient= int(input("How many patients to add?"))

for add\_itr in range(add\_patient):

patient\_id = add\_itr

watch\_add.start()

patients\_data.add\_to\_pqueue(patient\_id)

watch\_add.stop()

#print(patients\_data.priority\_queue)

print("PRIORITY\_QUEUE\_ADD\_TIME for",add\_patient,"inputs is", watch\_add.time\_elapsed(),"milli seconds")

watch\_add.reset()

####### GET SIZE #######

print("Patients Data Size:",patients\_data.get\_size())

''' ####### FIND #######

find\_patient= int(input("How many patients to find?"))

for find\_itr in range(find\_patient):

patient\_id = random.randint(0,10\*\*8)

watch\_find.start()

patients\_data.find\_id(patient\_id)

watch\_find.stop()

print(patients\_data.priority\_queue)

print("PRIORITY\_QUEUE\_FIND\_TIME for",find\_patient,"inputs is", watch\_find.time\_elapsed(),"milli seconds")

watch\_find.reset()

'''

####### REMOVE #######

remove\_patient= int(input("How many patients are to be treated?"))

for remove\_itr in range(remove\_patient):

watch\_remove.start()

#print(patients\_data.remove\_highest\_priority())

patients\_data.remove\_highest\_priority()

watch\_remove.stop()

print("PRIORITY\_QUEUE\_REMOVE\_TIME for",remove\_patient,"inputs is", watch\_remove.time\_elapsed(),"milli seconds")

watch\_remove.reset()

**Priority\_Queue\_Implementation.py**

#!/bin/python3

import time

import math

import os

import random

import re

import sys

from enum import Enum

from queue import PriorityQueue

class Stopwatch(object):

def \_\_init\_\_(self):

"""Initialize a new Stopwatch, but do not start timing."""

self.start\_time = None

self.elapsed\_time = 0

def start(self):

"""Start timing."""

self.start\_time = time.time()\*1000

def stop(self):

"""Stop timing."""

if not (self.start\_time):

return

stop\_time = time.time()\*1000

self.elapsed\_time += int(round(stop\_time - self.start\_time))

self.start\_time = None

def time\_elapsed(self):

if (self.start\_time):

self.stop()

return self.elapsed\_time

def reset(self):

self.start\_time = None

self.elapsed\_time = 0

if \_\_name\_\_ == '\_\_main\_\_':

watch\_add=Stopwatch()

watch\_remove= Stopwatch()

patients\_data = PriorityQueue()

####### ADD #######

add\_patient= int(input("How many patients to add?"))

for add\_itr in range(add\_patient):

patient\_id = add\_itr

priority=patient\_id%4

watch\_add.start()

patients\_data.put((priority,patient\_id))

watch\_add.stop()

print("PRIORITY\_QUEUE\_ADD\_TIME for",add\_patient,"inputs is", watch\_add.time\_elapsed(),"milli seconds")

watch\_add.reset()

####### REMOVE #######

remove\_patient= int(input("How many patients are to be treated?"))

a=remove\_patient

while not patients\_data.empty() and remove\_patient>0:

watch\_remove.start()

remove\_patient-=1

watch\_remove.stop()

print("PRIORITY\_QUEUE\_REMOVE\_TIME for",a,"inputs is", watch\_remove.time\_elapsed(),"milli seconds")

watch\_remove.reset()