

**Technical Report: Patients or Patience**

**Authors: James Price**

**Date: DATE**

**Wordcount: NUMBER Words**

**Pagecount: NUMBER Pages**

**Confidential: NO / YES – INTERNAL ONLY/YES – X DEPARTMENT ONLY**

Contents

[Executive Summary 3](#_Toc40205871)

[Introduction 4](#_Toc40205872)

[Theory 4](#_Toc40205873)

[Implementation 5](#_Toc40205874)

[Cases 5](#_Toc40205875)

[Heap 6](#_Toc40205876)

[Main 7](#_Toc40205877)

[Conclusions 8](#_Toc40205878)

[References 11](#_Toc40205879)

[Appendix A 11](#_Toc40205880)

[main.py 11](#_Toc40205881)

[heap.py 13](#_Toc40205882)

[case.py 15](#_Toc40205883)

# Executive Summary

Hospitals across the UK see a large number of A&E attendances every day and the cases vary in type and severity. The ideal way of managing incoming cases is to utilise a priority queue that will allow high priority cases to be seen sooner than lower priority cases. In order to deal with such a large quantity of data this report demonstrates the use of a binary maxheap in order to sort and manage incoming cases by the thousands with high speed and efficiency. Overall the implementation of the binary heap allows for high speeds of adding and removal of cases with 40,000 cases being added to the queue in under 1.2 seconds meaning it could easily handle a nationwide management of cases across all type one A&E departments.

# Introduction

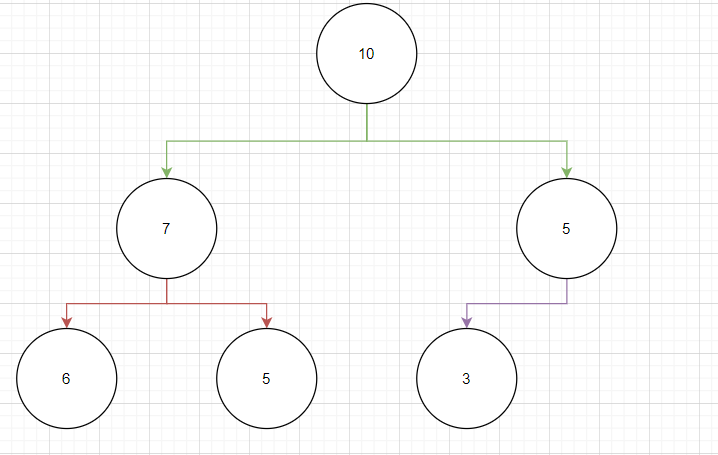
Hospital A&E departments can receive a wide array of different cases being presented on a day to day basis, with some cases being more urgent than others. Hospitals also have a time constraint in regards to getting patients seen on time. In the given brief, the time constraint was 4 hours and if a patient is not seen in under that amount of time the hospital is fined £10,000. As mentioned before, some cases are more urgent than others and as a result, they require a higher priority to ensure they are seen as soon as possible, so the program requires that cases can be queued in order of FIFO; however, that can be overridden by patient priority if it is high enough.

# Theory

A statistics paper for the NHS (Baker, 2020) states that in 2019 the average attendances per day for all Type 1 A&E departments across the country were at 44,366 and assuming an even distribution of attendances across all 132 NHS trusts operating a Type 1 A&E department (What’s going on with A&E waiting times?, 2020), there would be approximately 336 admittances per day. The system implemented could be used nationwide and hold all of the A&E admittances across the country meaning the system would need to be able to handle approximately 44,366 cases per day, with room for extra and during a crisis that number could spike dramatically. Given this information, it is clear that the solution implemented for priority queuing would need to be able to operate fast with large quantities of data; consequently, the data structure implemented would need to be able to operate with the fastest time complexity. This is why a Binary heap will be implemented.

Binary (Max) heaps are based on the binary tree with two extra constraints:

1. The parent node must have an equal or higher value than its children
2. The heap is held as a mostly complete binary tree meaning all nodes have two children except for the second deepest layer and leaf nodes

A visual representation of a binary heap would be like so:

**Figure 1:** a diagram showing how parent nodes point to child nodes

Elements in a list can be treated as nodes of a binary heap using the current index of the selected element and simple formulae to calculate the parent and child nodes (if any):

When I = index of particular node:

def get\_parent\_index(self, i):

return int((i-1)/2)

def get\_left\_child\_index(self, i):

return (2\*i)+1

def get\_right\_child\_index(self, i):

return (2\*i)+2

A binary heap would be ideal for the task because of its worst-case time complexities, which would be especially ideal for large quantities of nodes as binary heap has the following worst-case time complexities:

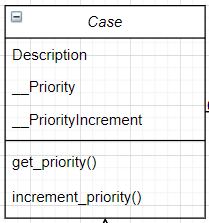
**Table 1:**  The list of worst-case time complexities for main operations to be done on a binary heap

|  |  |
| --- | --- |
| **Operation** | **Time complexity (BigO)** |
| Insert | O(log n) |
| Delete | O(log n) |
| Heapify | O(log n) |
| Peek at root | O(1) |

# Implementation

## Cases

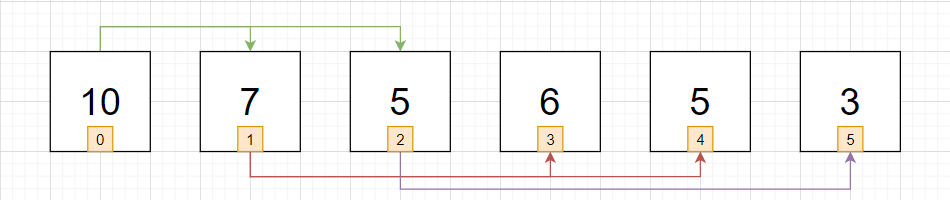
To conform with the brief, the cases that are stored in the queue must have a description and a priority, but the priority must increment by a predetermined amount every 10 minutes; therefore, for the implementation of a case class there would need to be a case description, priority and a priority increase amount defined. To protect the priority and increment values, they are made private to the class and there are procedures implemented to utilise and return the private values.



**Figure 2:** A UML diagram for the case class that holds the case description, priority and priority increment

## Heap

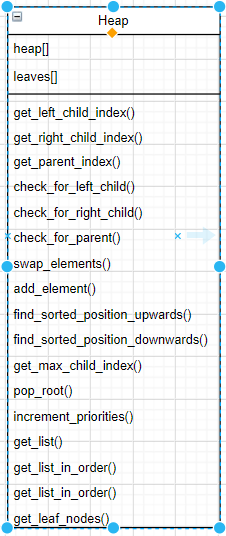
The Heap class holds all of the case elements and has all of the algorithms for adding, removing, sorting and checking the root value. The actual values are stored in the program with an array but the operations that are done to calculate the index positions and manipulate data **treats the array like a heap**. Here is an example array of numbers and how the parent-child relationships would connect those elements:



**Figure 3:** an example of how elements in an array could be treated as a binary heap and how elements link, with arrows showing parent-to-child relationship.

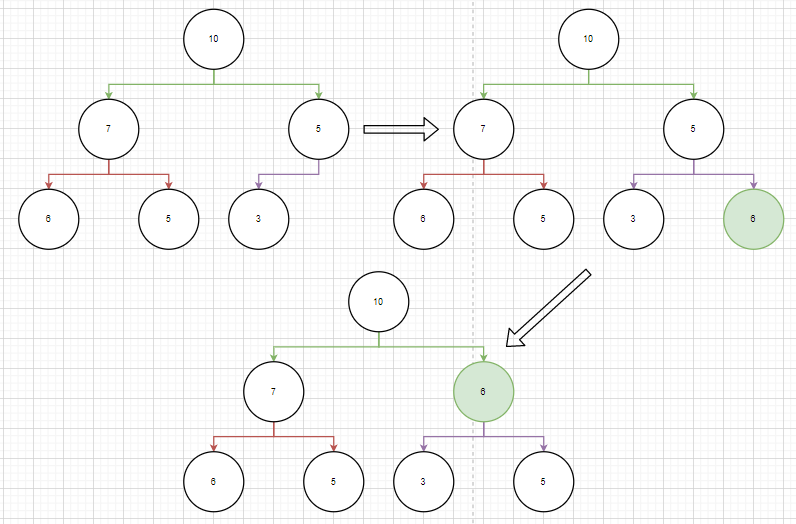
The value 10 (at index 0) will have children 7 and 5 (at indecies 1 and 2 respectively) which was calculated using the formulae **leftChild = 2i+1 and rightChild = 2i+2**.

Below is a list of procedures and attributes present in the MaxHeap class:



**Figure 4:** A UML diagram of the heap class showing its attributes and methods

Adding to the heap works by appending the case to the end of the list then checking it’s priority with its parent node. If its priority is higher than the parent then the nodes are swapped and that process happens indefinitely until its priority is not higher than its current parent, as then it is into the sorted position.

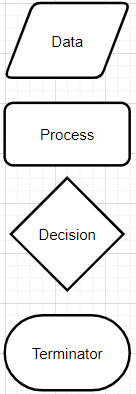
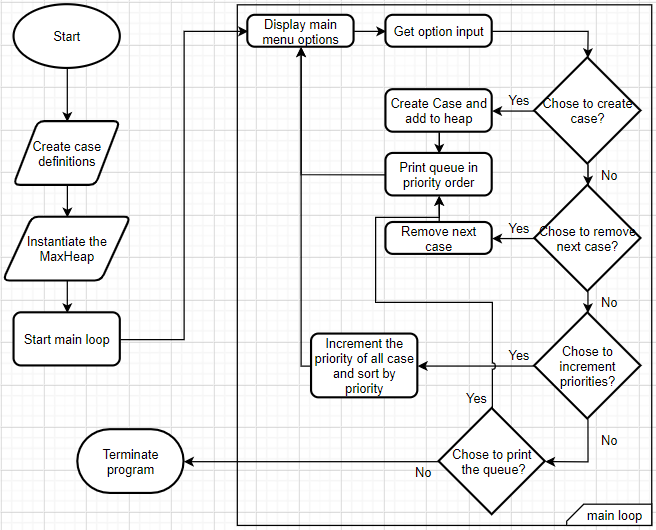


**Figure 5:** An example of adding a node with a value of 6. It gets swapped once with a parent node.

To remove the root node, it is swapped with the last node in the list and then removed. The last node is then sorted downwards in a similar fashion to how newly added nodes are sorted upwards.

## Main

The main class handles the operation of the MaxHeap class and takes the user input to traverse menus and delete or create cases to be handled in the heap. A simple flowchart below shows the operation of the main class:



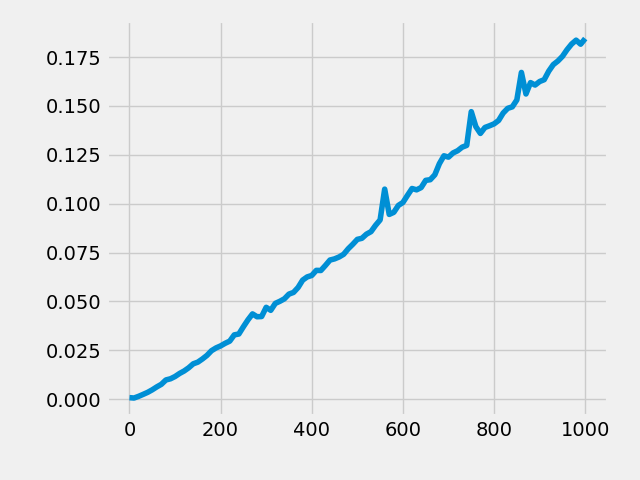
**Figure 6:** A flowchart of the user interface system defined by the main class that operates and triggers the methods in the heap

# Conclusions

Firstly the process of adding elements to the heap and sorting them into the correct position has the worst-case time complexity of O(n). The operation to append a case to the heap is O(1) and the operation of sorting that case into the correct position is O(n) and this is seen in Figure 2.1 as that has a linear increase in time taken to add cases to the heap and sort it into place.

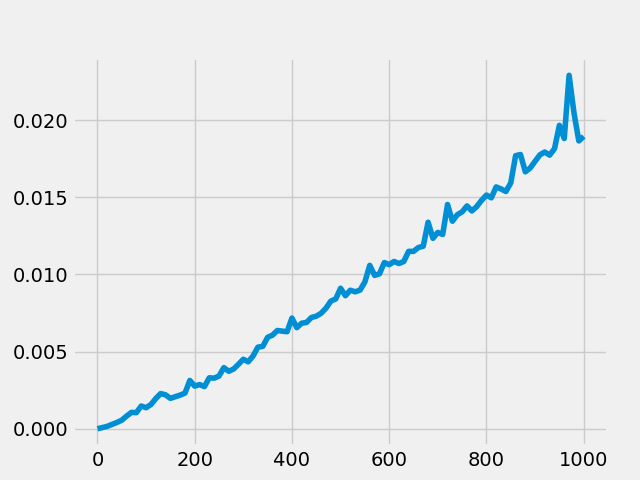
Secondly, the process of removing the next case in the queue would have the same time complexity as the complexity of removing the case would be O(1) and the sorting of the elements would take O(n), just like the adding of elements. Figure 2.2 shows this trend with the removal of 1000 elements taking approximately 0.018 seconds (18ms).

**Figure 7:** The time taken for a given number of cases to be added to the heap and sorted into the correct place



The time taken to add and sort a number of elements into place (seconds)

The number of elements being added to the heap

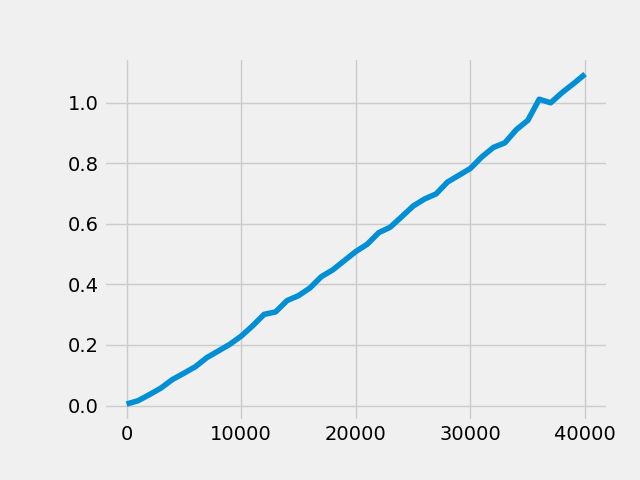


The number of elements

The time taken to remove a given number of elements (seconds)

**Figure 8:** The time taken for a given number of elements to be removed from the heap.

Given that large quantities of data can be added to the queue and sorted into place quickly tens of thousands of entries can be added in fast succession and it would be possible for a binary heap to be implemented into a nation-wide queuing system to link all hospitals under one system to allow for easier statistics collection to analyse the efficiency and effectiveness of all type one A&E departments in the UK as shown by **Figure 8** below where the quantities of data are much larger compared to **Figure 7** and **Figure 6**, which would show use case in individual systems given to each department separately.



**Figure 9:** The time taken to add a given number of elements. Tests were done in increments of 1,000 up to 40,000 elements, with 40,000 elements taking just over 1.1 seconds to be added.

This data shows that a large scale queuing system could be implemented using a binary max heap and would give an opening to possible analytical studies into A&E practices which could, in turn, lead to better practices being implemented into hospitals to help hospitals see patients on time and reduce the risk of the hospital incurring fines.

# References

Baker, C., 2020. *NHS Key Statistics. England, February 2020*. [ebook] pp.4, 5. Available at: <https://researchbriefings.files.parliament.uk/documents/CBP-7281/CBP-7281.pdf> [Accessed 25 April 2020].

The King's Fund. 2020. *What’S Going On With A&E Waiting Times?*. [online] Available at: <https://www.kingsfund.org.uk/projects/urgent-emergency-care/urgent-and-emergency-care-mythbusters> [Accessed 25 April 2020].

# Appendix A

## main.py

from heap import MaxHeap

from case import Case

class patientQueue:

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

self.caseDefinitions = {

"LOC": ["Loss of Consciousness", 24, 6],

"FOS": ["Fits or Seizures", 30, 7],

"CP": ["Chest Pains", 42, 8],

"BD": ["Breathing Difficulties", 60, 10],

"SB": ["Severe Bleeding", 48, 10],

"AR": ["Allergic Reactions", 12, 3],

"BOS": ["Burns or Scalds", 18, 4],

"S": ["Stroke", 36, 2],

"RTA": ["Road Traffic Accident", 54, 5],

"BA": ["Broken Arm", 6, 2]

}

self.menuOneText = "Please select one of the following options:\n(1):Add case\n(2):Retrieve next queued case\n(3):Quit program\n(4 (testing)): Increment time base priority\n(5): Print out entire list"

self.addMenuText ="Please select one of the folloowing cases, or enter '-1' to return to the main menu"

self.queue = MaxHeap()

self.main\_loop()

def menu\_one(self):

print(self.menuOneText)

validChoice = False

while(not validChoice):

choice = input(">>> ")

if (choice in ["1", "2", "3", "4", "5"]):

validChoice = True

else:

print("Invalid option, please re-enter...")

return choice

def add\_menu(self):

print(self.addMenuText)

for key, value in self.caseDefinitions.items():

print("Code: ", key, ", Name: ", value[0])

validChoice = False

while(not validChoice):

choice = input(">>> ")

if(choice in self.caseDefinitions):

self.queue.add\_element(Case(self.caseDefinitions[choice]))

validChoice = True

elif(choice == "-1"):

validChoice = True

else:

print("Invalid option, please re-enter...")

return choice

def print\_queue(self):

print("\_"\*20)

for case in self.queue.get\_list\_in\_order():

print(case.description, ":", case.get\_priority())

def main\_loop(self):

while (True):

menuOneChoice = self.menu\_one()

if(menuOneChoice == "1"):

if(self.add\_menu() == "-1"):

continue

else:

self.print\_queue()

continue

elif(menuOneChoice == "2"):

print(self.queue.pop\_root())

self.print\_queue()

continue

elif(menuOneChoice == "3"):

break

elif(menuOneChoice == "5"):

self.print\_queue()

continue

self.queue.increment\_priorities()

PQ = patientQueue()

## heap.py

class MaxHeap:

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

self.heap = [] #Elements in the heap

self.leaves = [] #Indecies to the leaf nodes in the heap

def get\_left\_child\_index(self, i):

return (2\*i)+1

def get\_right\_child\_index(self, i):

return (2\*i)+2

def get\_parent\_index(self, i):

return int((i-1)/2)

def check\_for\_left\_child(self, i):

return self.get\_left\_child\_index(i) < len(self.heap)

def check\_for\_right\_child(self, i):

return self.get\_right\_child\_index(i) < len(self.heap)

def check\_for\_parent(self, i):

return self.get\_parent\_index(i) >= 0

def swap\_elements(self, i, j):

self.heap[i], self.heap[j] = self.heap[j], self.heap[i]

def add\_element(self, element):

self.heap.append(element)

self.find\_sorted\_position\_in\_heap\_upwards(len(self.heap)-1)

def find\_sorted\_position\_in\_heap\_upwards(self, i):

size = len(self.heap)

if(i > 0):

while (self.check\_for\_parent(i) and self.heap[i].get\_priority() > self.heap[self.get\_parent\_index(i)].get\_priority()):

self.swap\_elements(i, self.get\_parent\_index(i))

i = self.get\_parent\_index(i)

def find\_sorted\_position\_in\_heap\_downwards(self, i):

while(self.check\_for\_left\_child(i)):

maxChildIndex = self.get\_max\_child\_index(i)

if (maxChildIndex == -1):

break

if (self.heap[i].get\_priority() < self.heap[maxChildIndex].get\_priority()):

self.swap\_elements(i, maxChildIndex)

i = maxChildIndex

else:

break

def get\_max\_child\_index(self, i):

if (self.check\_for\_left\_child(i)):

leftChildIndex = self.get\_left\_child\_index(i)

if (self.check\_for\_right\_child(i)):

rightChildIndex = self.get\_right\_child\_index(i)

if (self.heap[leftChildIndex].get\_priority() > self.heap[rightChildIndex].get\_priority()):

return leftChildIndex

else:

return rightChildIndex

else:

return leftChildIndex

else:

return int(-1)

def pop\_root(self):

if len(self.heap) == 0:

return -1

endOfList = len(self.heap) - 1

self.swap\_elements(0, endOfList)

oldRoot = self.heap.pop()

self.find\_sorted\_position\_in\_heap\_downwards(0)

return oldRoot

def increment\_priorities(self):

for element in self.heap:

element.increment\_priority()

self.leaves = []

self.get\_leaf\_nodes(0)

for leaf in self.leaves:

self.find\_sorted\_position\_in\_heap\_upwards(leaf)

def get\_list(self):

return self.heap

def get\_list\_in\_order(self):

tempUnsorted = self.heap[:]

tempSorted = []

while(len(self.heap)>0):

tempSorted.append(self.pop\_root())

self.heap = tempUnsorted

return tempSorted

def get\_leaf\_nodes(self, i):

if (self.check\_for\_left\_child(i)==False and self.check\_for\_right\_child(i)==False):

self.leaves.append(i)

return

if (self.check\_for\_left\_child(i)):

self.get\_leaf\_nodes(self.get\_left\_child\_index(i))

if (self.check\_for\_right\_child(i)):

self.get\_leaf\_nodes(self.get\_right\_child\_index(i))

## case.py

class Case:

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

self.description = details[0]

self.\_\_priority = details[1]

self.\_\_incrementAmount = details[2]

def get\_priority(self):

return self.\_\_priority

def increment\_priority(self):

self.\_\_priority += self.\_\_incrementAmount