# Assignment 2 report

Jesper Wingren
Jw223rn@student.lnu.se
1dt907

## Innehållsförteckning

# 1. Queue

## 1.1 Setup

The Queue is setup using different methods to make the different things a queue should be able to do.

### 1.1.1 Queue

The queue holds 3 properties when initialized, the array of size 10, the size 10, front value and back value. To begin with both front and back values are 0 because it is empty. The enqueue method sets the backvalue to the given enqueue element and then moves it back. This is if the list is not full because if so, the list must double in size. The dequeue almost works by removing a random element by swapping the random element with the back item then decreasing the size and moving back the back value. If after this the list is less than half the size of the list it halves the size making memory usage lower. The method count gives us the number of elements in list while empty checks if back and front are the same in that case meaning it is empty.

### 1.1.2 Iterator

The iterator works by creating a list and copying the elements from the array over to it and then shuffling them. The getNext first checks if there are more elements by checking hasNext method. If it has more, it returns the element at indexcount which is counted for every element removed.

# 2. AVL vs BST

## 2.1 Setup

I have setup two different classes for the different trees and then testing them and comparing the results.

### 2.1.1 BST

The binary search tree is setup like a bst tree is supposed to where the values are always put left If less and right if higher. We have the basic add methods and the delete methods, the delete methods are more complicated as we could remove values inside three which means you have to relocate nodes to connect the tree together. There is also a depth method which calculates the depth of the tree but going down each side and comparing them, there is also the minvalue which goes to the farthest left to get the min value.

### 2.1.2 AVL

The avl tree is setup almost like the bst but instead of just adding values it always balances it out after each change in the tree. This is done by the balance method; this uses several other helping methods that rotates the nodes in the setup way. There are also the double rotation methods which uses the other rotation methods. These balance methods make the tree always be balanced, by balanced means that it branches out the nodes each add/delete making the depth much lower but also takes more time to make.

## 2.2 Results

```
Average height of BST trees (random input):
12
Time to create 5000 BST trees (random input): 0.175738542
Average height of AVL trees (random input):
6
Time to create 5000 AVL trees (random input): 0.190524084
Average height of BST trees (sorted input):
1024
Time to create 5000 BST trees (sorted input): 8.903184917
Average height of AVL trees (sorted input):
10
Time to create 5000 AVL trees (sorted input): 0.218231
```

In the picture you can see the results. Some things to notice is for example that the bst takes a long time to add and delete values if the input is sorted because this will just be a long line of nodes and not a "tree". Something else to notice is that the avl trees take longer but the average height is half of the bst. The height makes a big difference when for example finding a value in the tree because a smaller depth allows for faster searching. You can also see that when working with sorted lists it's not good to use either of these trees because it increases in both time and depth.

# 3. Ticket problem

## 3.1 Setup

This ticket problem I solved by making a heap as a priority queue. The heap has nodes which includes their priority level and the name of the person. If you then want to get a person from the heap the person with the highest priority(lowest number) and first added returns to the user.

### 3.1.1 Insert person method

The insert works as excepted for a heap by adding the person to the back of the heap and heapify up.

### 3.1.2 Get person method

The get person works by swapping the root(the person first in queue) with the back then heapify down to balance the heap.

### 3.1.3 Delmaxprio method

The delmaxprio first finds the maximum prio(highest number but lowest prio) in the tree and stores that so it can delete all nodes with that priority level. It then moves through the tree deleting all nodes with that priority level.

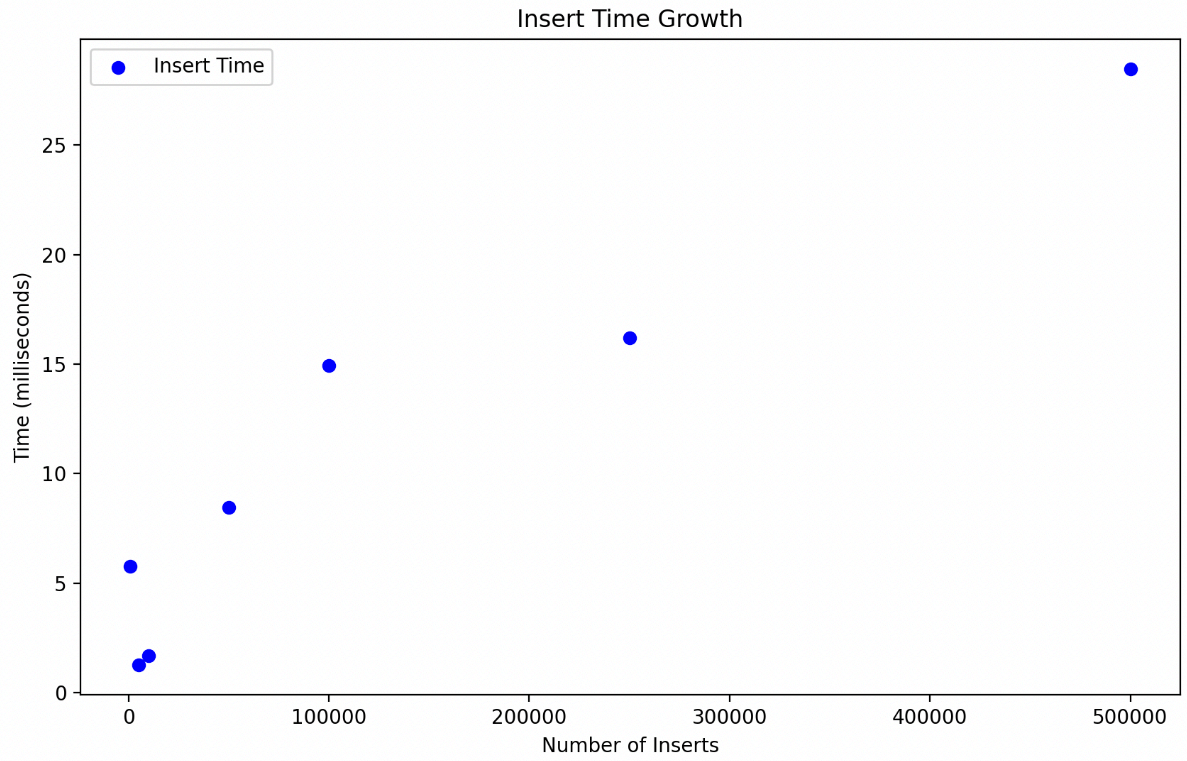
### 3.1.4 Swapprio method

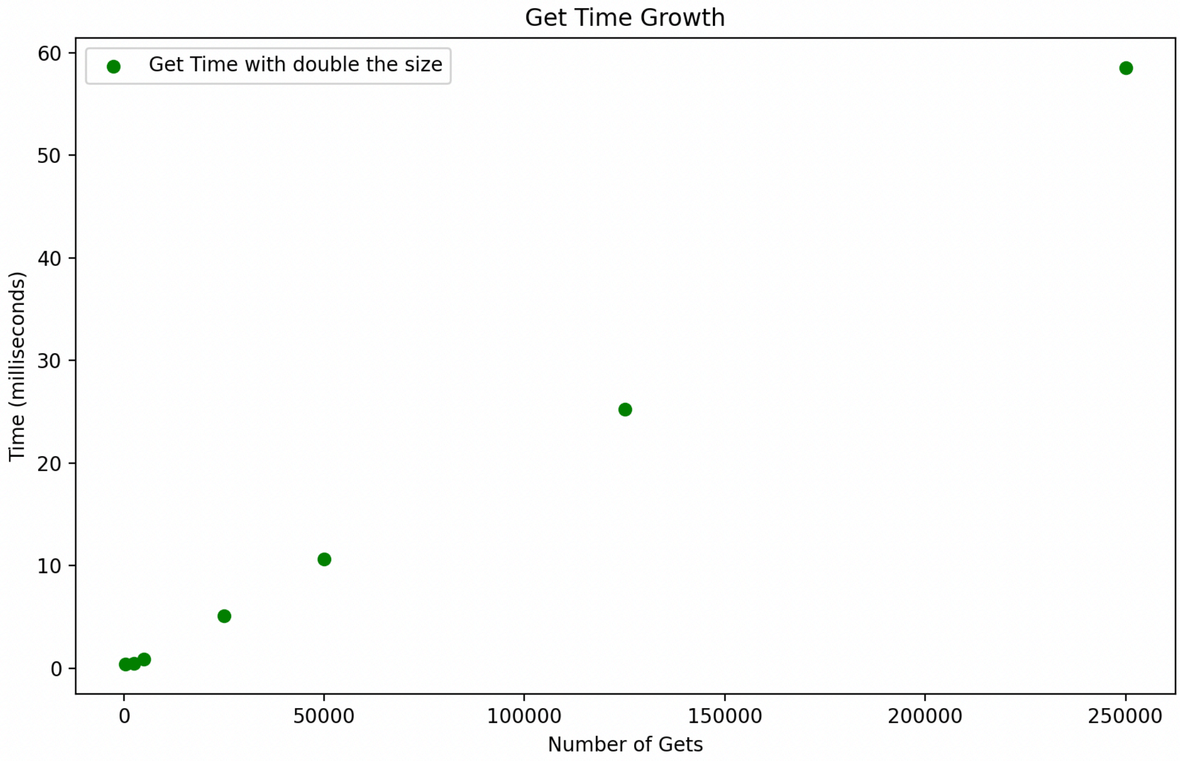
The swap priority method works by getting the positions of the nodes to be swapped and then creates new nodes and then balances the tree up again.

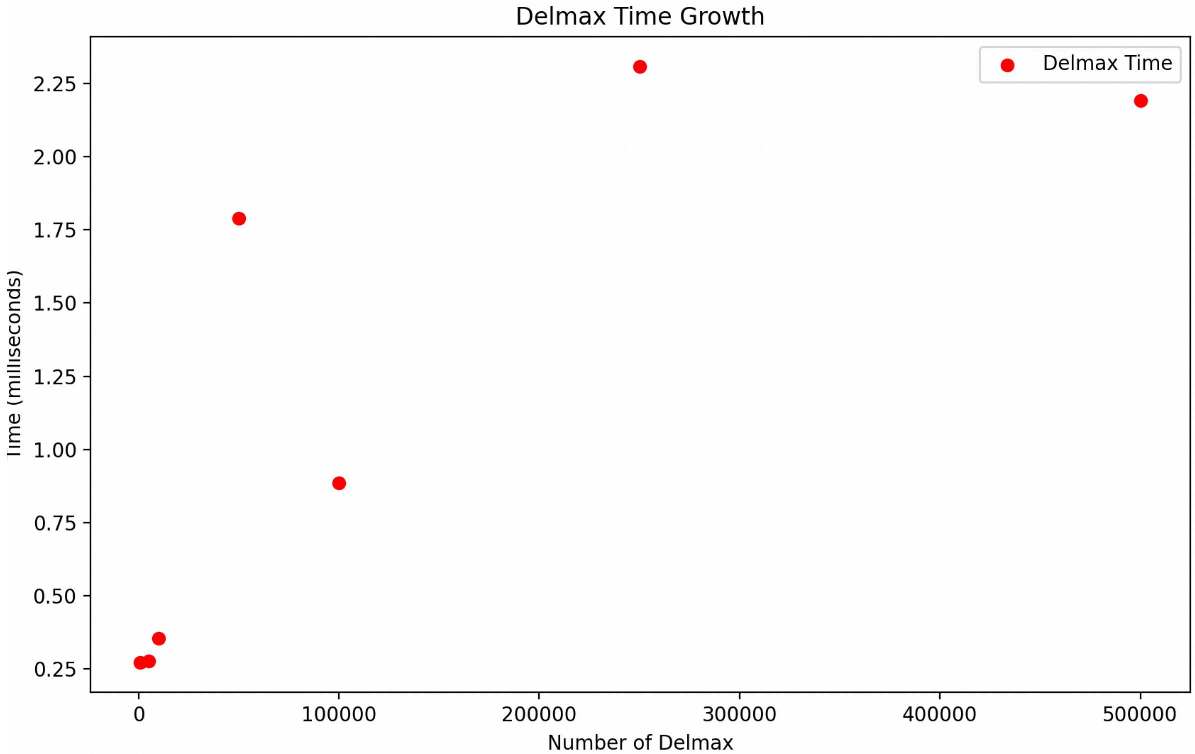
## 3.2 Results

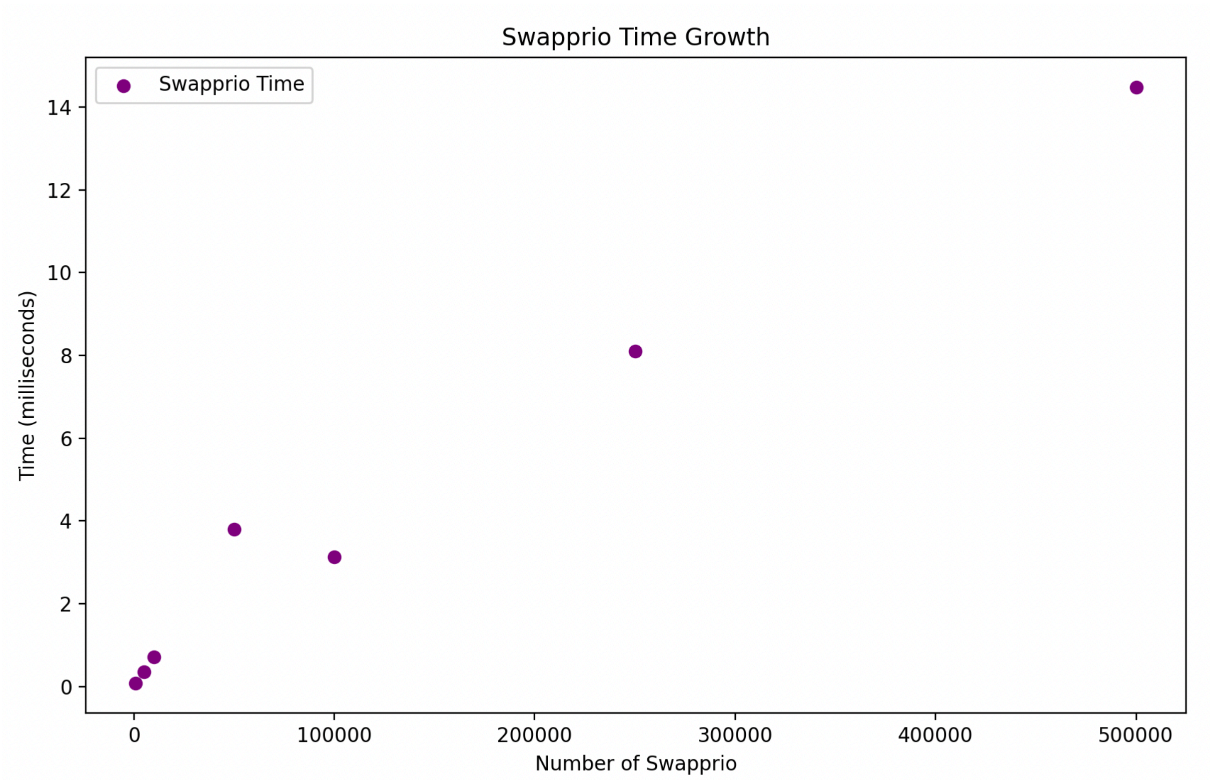
The graphs look a bit weird, but you can still see some growth but because the time is so small it is affected by a lot of things.

```
Time for 1000 inserts 5.146375 Milliseconds
Time for 5000 inserts 2.311084 Milliseconds
Time for 10000 inserts 1.705125 Milliseconds
Time for 50000 inserts 7.696292 Milliseconds
Time for 100000 inserts 7.637167 Milliseconds
Time for 250000 inserts 16.798917 Milliseconds
Time for 500000 inserts 22.87925 Milliseconds
----------------------------
Time for 500 gets in a queue with 1000 persons 0.460167 Milliseconds
Time for 2500 gets in a queue with 5000 persons 0.44575 Milliseconds
Time for 5000 gets in a queue with 10000 persons 0.867083 Milliseconds
Time for 25000 gets in a queue with 50000 persons 5.566292 Milliseconds
Time for 50000 gets in a queue with 100000 persons 10.98075 Milliseconds
Time for 125000 gets in a queue with 250000 persons 26.289083 Milliseconds
Time for 250000 gets in a queue with 500000 persons 58.605541 Milliseconds
----------------------------
Time for delmax with 1000 elements 0.364167 Milliseconds
Time for delmax with 5000 elements 0.473208 Milliseconds
Time for delmax with 10000 elements 0.289625 Milliseconds
Time for delmax with 50000 elements 1.570291 Milliseconds
Time for delmax with 100000 elements 0.9375 Milliseconds
Time for delmax with 250000 elements 1.67575 Milliseconds
Time for delmax with 500000 elements 3.4155 Milliseconds
----------------------------
Time for swapprio with 1000 elements 0.091542 Milliseconds
Time for swapprio with 5000 elements 0.380125 Milliseconds
Time for swapprio with 10000 elements 0.723416 Milliseconds
Time for swapprio with 50000 elements 3.853833 Milliseconds
Time for swapprio with 100000 elements 2.856209 Milliseconds
Time for swapprio with 250000 elements 2.343917 Milliseconds
Time for swapprio with 500000 elements 6.221542 Milliseconds
```




Get Time Growth
Get Time with double the size
Time (milliseconds)
Number of Gets
0
10
20
30
40
50
60
0
50000
100000
150000
200000
250000


Delmax Time Growth
Delmax Time
Time (milliseconds)
Number of Delmax
0.25
0.50
0.75
1.00
1.25
1.50
1.75
2.00
2.25
0
100000
200000
300000
400000
500000




All the methods has a time complexity of O(logN) in average but the delmaxprio depends alot on how the prio is distributed between the different persons in the queue.

# 4. Quicksort

## 4.1 Setup

Quicksort is setup to occur for as many recursions as the user wants to and after this it can either hop to insert- or heap sort based on the user's choice.

### 4.1.1 Quicksort

Quicksort is setup with a set depth of recursions and what method to use after that. The partition method fixes the pivot while the medianCalc method calculates the pivot index to be used. The median is gathered by taking three values and getting that median which the is supposed to increase the sorting speed of the algorithm. The sort methods basically sorts the array but only does the partitions and sorts until the depth has been reached and then calls for either heap- or insert sort.

### 4.1.2 Insertsort

Insertsort goes through the list and sort it piece by piece using for loops that iterates over all values. This then swaps if the elements are in the incorrect order.

### 4.1.3 Heapsort

Heapsort has a property of the array and the size of the heap. The sort method uses the sink method on each non-leaf node then extracting the maximum element from the heap and placing it in the array. The sink method moves an element until it's in the correct place and compares it to its children making sure it's in the correct place.

## 4.2 Results

```
For the depth 10 insertSort is faster
For the depth 15 insertSort is faster
For the depth 20 insertSort is faster
For the depth 25 heapSort is faster
For the depth 30 heapSort is faster
For the depth 35 insertSort is faster
For the depth 40 insertSort is faster
For the depth 45 insertSort is faster
For the depth 50 insertSort is faster
```

Here we can see the results from the experiment. This experiment was conducted on an array with 100 000 random elements. The depths above 35 is somewhat not relevant as there is just a small difference but here, we can see insert sort is better when the list is less sorted before while heap sort works better on a list which already is somewhat sorted.