ISTANBUL TECHNICAL UNIVERSITY

COMPUTER ENGINEERING DEPARTMENT

BLG 335-E
ANALYSIS OF ALGORITHMS

HOMEWORK-2 REPORT

EMRE UYSAL

PART A

1.INTRODUCTION

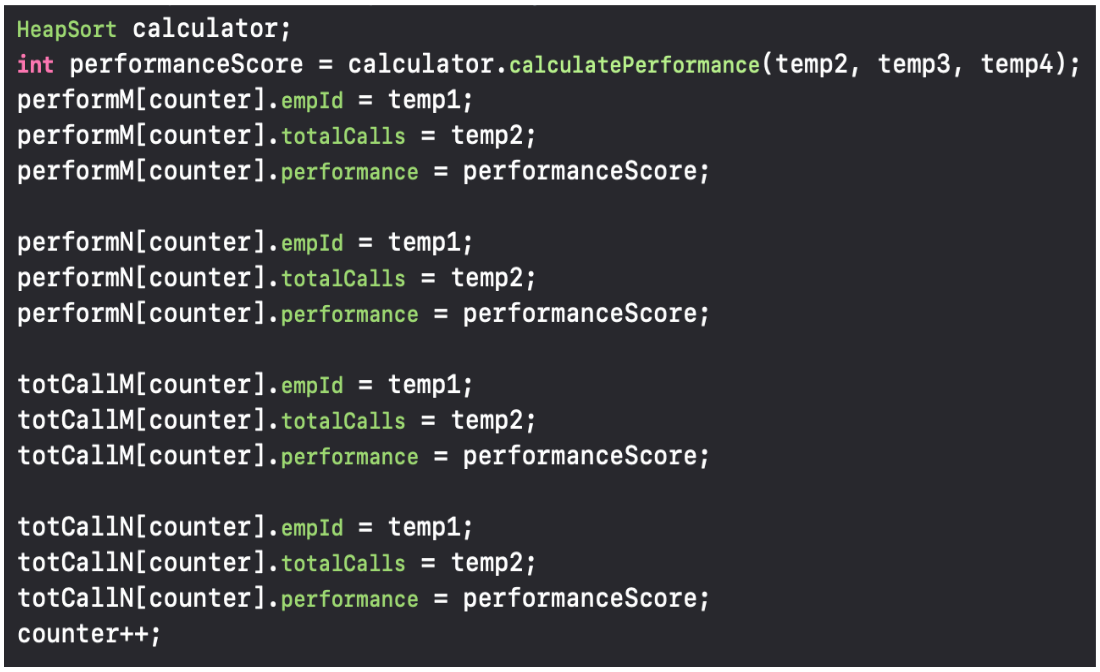
In this homework, we needed to implement Heap Sort and Priority Queue functions. After building our heap using day1.csv file , making updates and insert operations required. The detailed explanation made in the other parts of the report.

2.STEP BY STEP EXPLANATION

2.1 First Step

In this step, I read the day1.csv and calculated the performance values for each id. I created performM heap for maximum performance heap, then performN heap for minimum performance heap. After that I created totCallM heap for maximum call heap and totCallN heap for minimum call heap.

Then I put them into my array of structures. All structures has the value of performance, totalCalls and empId. So with this type of implementation, when making swap operations we have capability to change all values in one step.

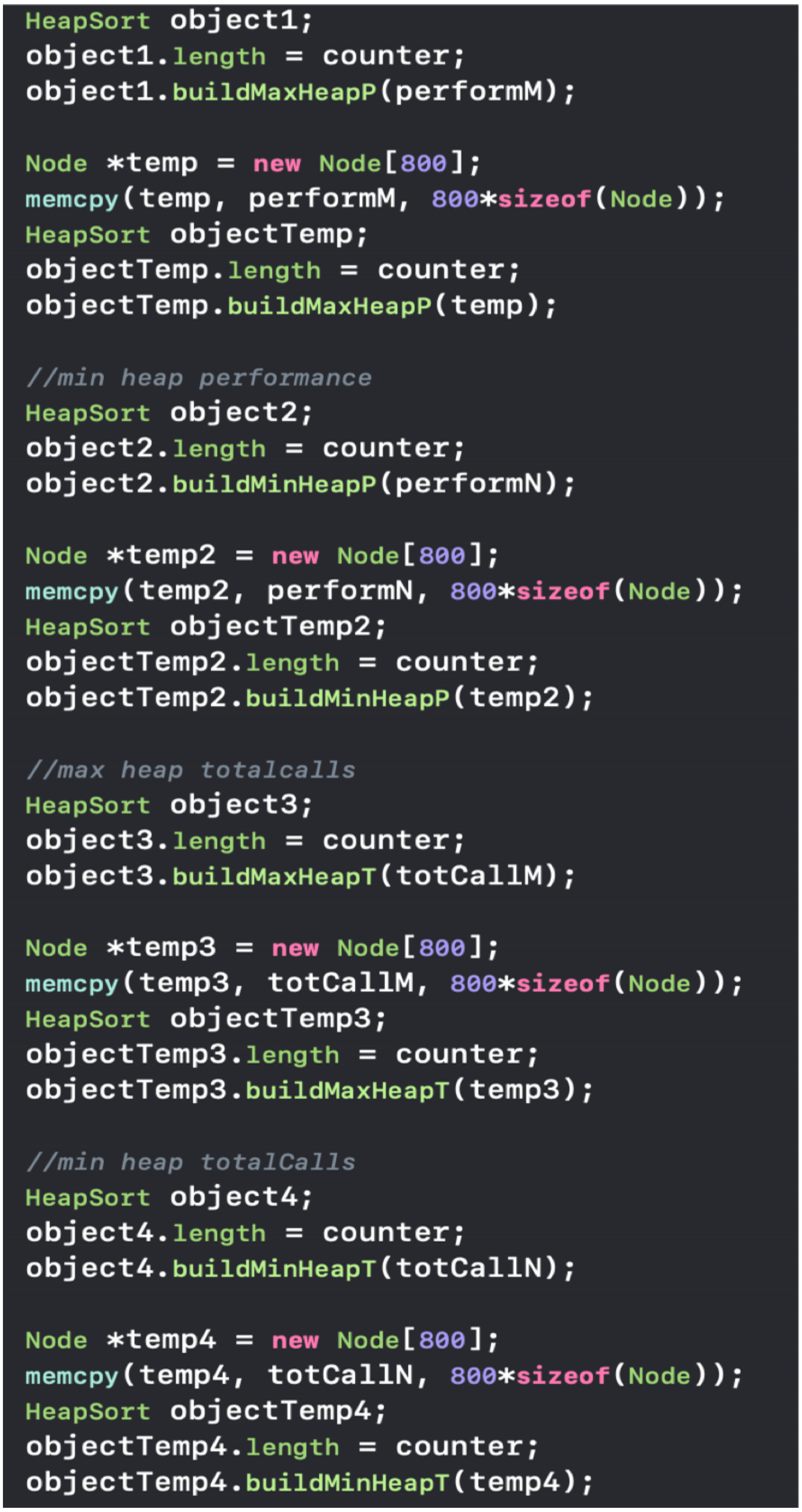


```
HeapSort calculator;
int performanceScore = calculator.calculatePerformance(temp2, temp3, temp4);
performM[counter].empId = temp1;
performM[counter].totalCalls = temp2;
performM[counter].performance = performanceScore;

performN[counter].empId = temp1;
performN[counter].totalCalls = temp2;
performN[counter].performance = performanceScore;

totCallM[counter].empId = temp1;
totCallM[counter].totalCalls = temp2;
totCallM[counter].performance = performanceScore;

totCallN[counter].empId = temp1;
totCallN[counter].totalCalls = temp2;
totCallN[counter].performance = performanceScore;
counter++;
```

putting the day1.csv into array

2.2 Second Step

In this step I created my objects from the HeapSort Class and I set my length. Then I created necessary heaps that we need to use in the other operations. For printing the result to screen I copied heaps into temp heaps. After that I extracted values from these heaps and deleted them after printing the results.

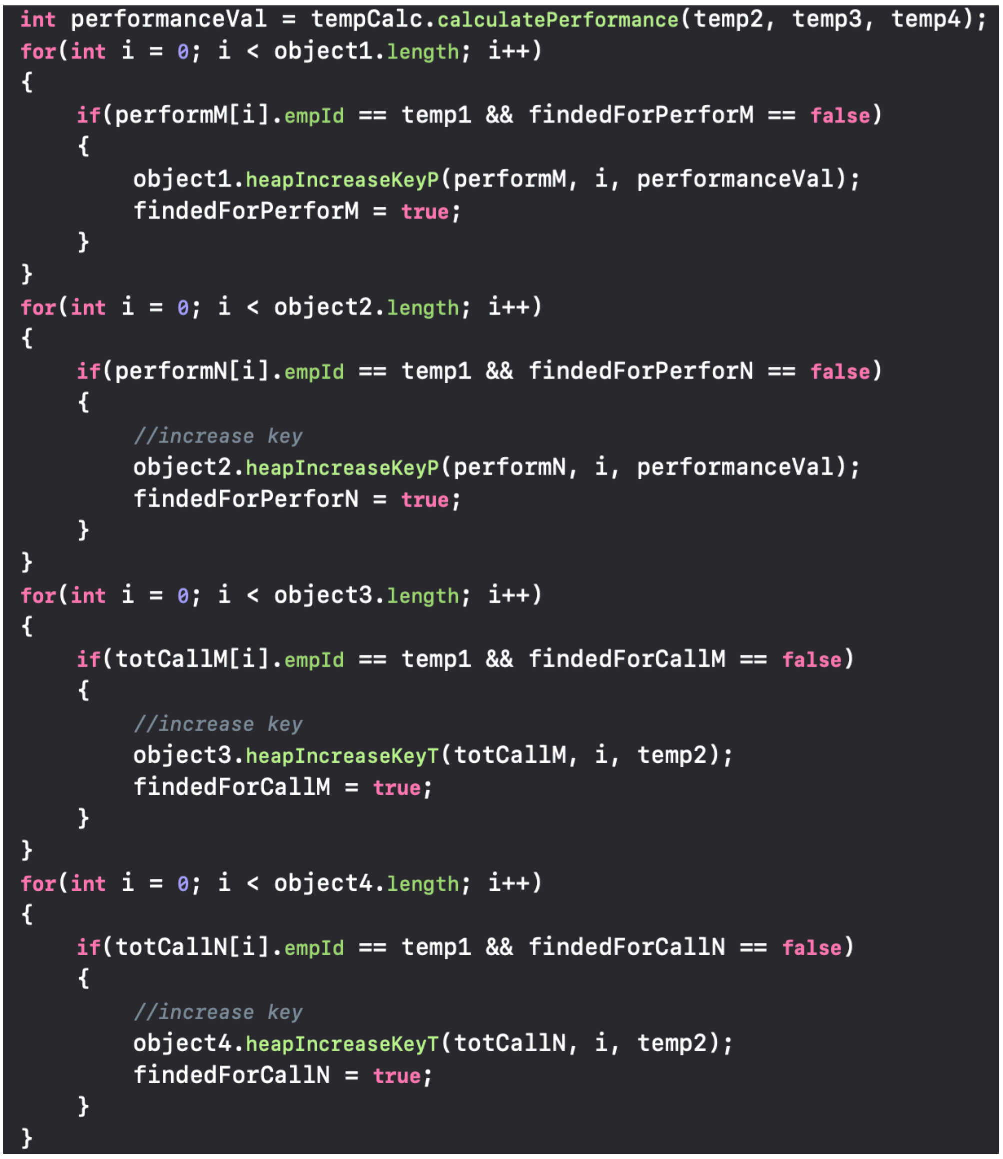


```
HeapSort object1;
object1.length = counter;
object1.buildMaxHeapP(performM);

Node *temp = new Node[800];
memcpy(temp, performM, 800*sizeof(Node));
HeapSort objectTemp;
objectTemp.length = counter;
objectTemp.buildMaxHeapP(temp);

//min heap performance
HeapSort object2;
object2.length = counter;
object2.buildMinHeapP(performN);

Node *temp2 = new Node[800];
memcpy(temp2, performN, 800*sizeof(Node));
HeapSort objectTemp2;
objectTemp2.length = counter;
objectTemp2.buildMinHeapP(temp2);

//max heap totalcalls
HeapSort object3;
object3.length = counter;
object3.buildMaxHeapT(totCallM);

Node *temp3 = new Node[800];
memcpy(temp3, totCallM, 800*sizeof(Node));
HeapSort objectTemp3;
objectTemp3.length = counter;
objectTemp3.buildMaxHeapT(temp3);

//min heap totalCalls
HeapSort object4;
object4.length = counter;
object4.buildMinHeapT(totCallN);

Node *temp4 = new Node[800];
memcpy(temp4, totCallN, 800*sizeof(Node));
HeapSort objectTemp4;
objectTemp4.length = counter;
objectTemp4.buildMinHeapT(temp4);
```

creating objects and building the heaps

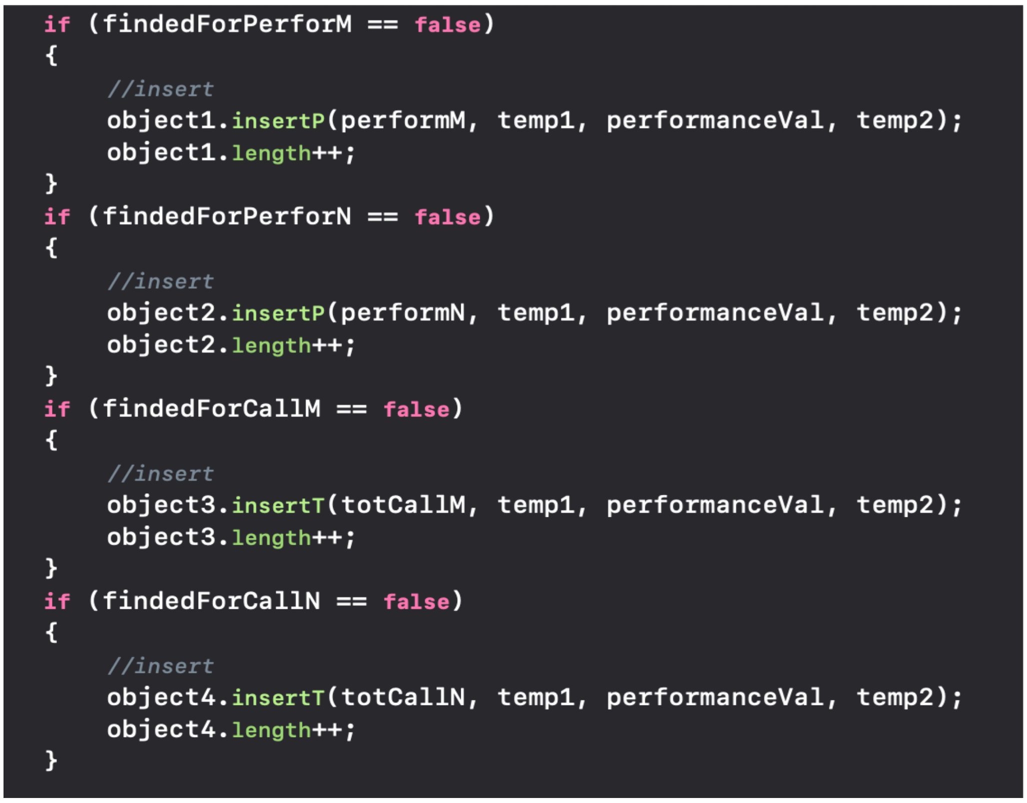
2.3 Third Step

In this part I automatically read the day files step by step when getting the values from a line, I calculated the performance as it was in the first part. Then I searched for the current id in the all heaps and gave them a flag. When the id founded by the program flag is set to true and increasekey function called. If a flag did not set, I called insert function for it and inserted the required values into arrays. I also incremented the length of the array after insert operation. Then I gave the daily report by copying the original heaps into temps and extracted values from the temp heaps.



```
int performanceVal = tempCalc.calculatePerformance(temp2, temp3, temp4);
for(int i = 0; i < object1.length; i++)
{
    if(performM[i].empId == temp1 && findedForPerforM == false)
    {
        object1.heapIncreaseKeyP(performM, i, performanceVal);
        findedForPerforM = true;
    }
}
for(int i = 0; i < object2.length; i++)
{
    if(performN[i].empId == temp1 && findedForPerforN == false)
    {
        //increase key
        object2.heapIncreaseKeyP(performN, i, performanceVal);
        findedForPerforN = true;
    }
}
for(int i = 0; i < object3.length; i++)
{
    if(totCallM[i].empId == temp1 && findedForCallM == false)
    {
        //increase key
        object3.heapIncreaseKeyT(totCallM, i, temp2);
        findedForCallM = true;
    }
}
for(int i = 0; i < object4.length; i++)
{
    if(totCallN[i].empId == temp1 && findedForCallN == false)
    {
        //increase key
        object4.heapIncreaseKeyT(totCallN, i, temp2);
        findedForCallN = true;
    }
}
```

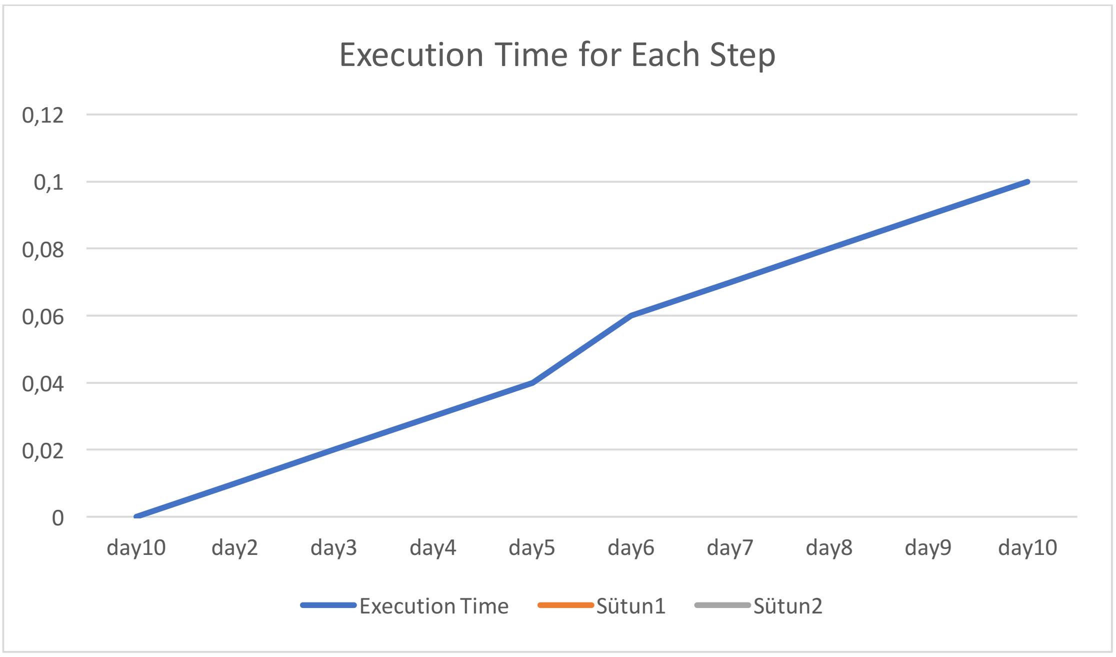
search ids, set the flags, increasekey



```
if (findedForPerforM == false)
{
    //insert
    object1.insertP(performM, temp1, performanceVal, temp2);
    object1.length++;
}
if (findedForPerforN == false)
{
    //insert
    object2.insertP(performN, temp1, performanceVal, temp2);
    object2.length++;
}
if (findedForCallM == false)
{
    //insert
    object3.insertT(totCallM, temp1, performanceVal, temp2);
    object3.length++;
}
if (findedForCallN == false)
{
    //insert
    object4.insertT(totCallN, temp1, performanceVal, temp2);
    object4.length++;
}
```

insert if id does not exist

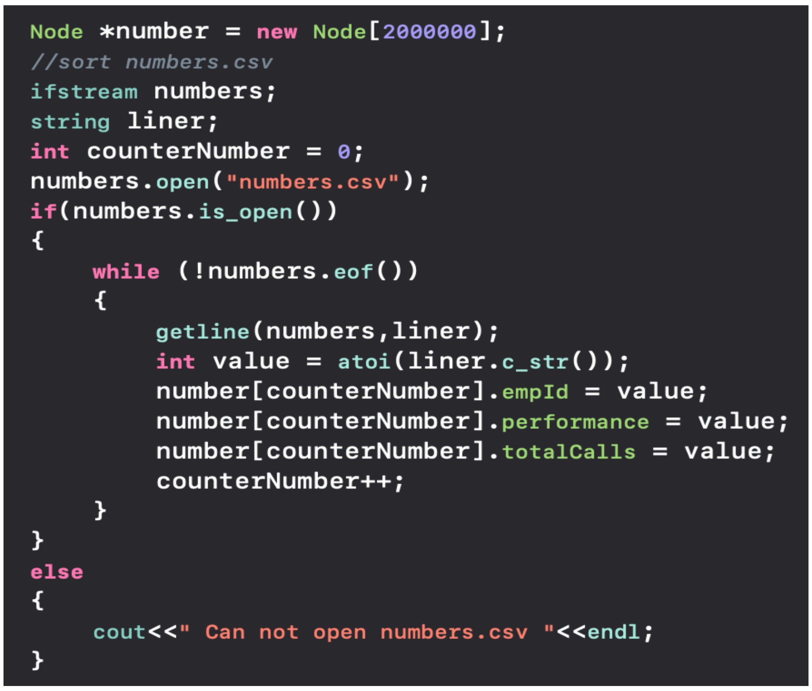
PART B



This graphic shows that execution time increased because heapsize increased for each step.

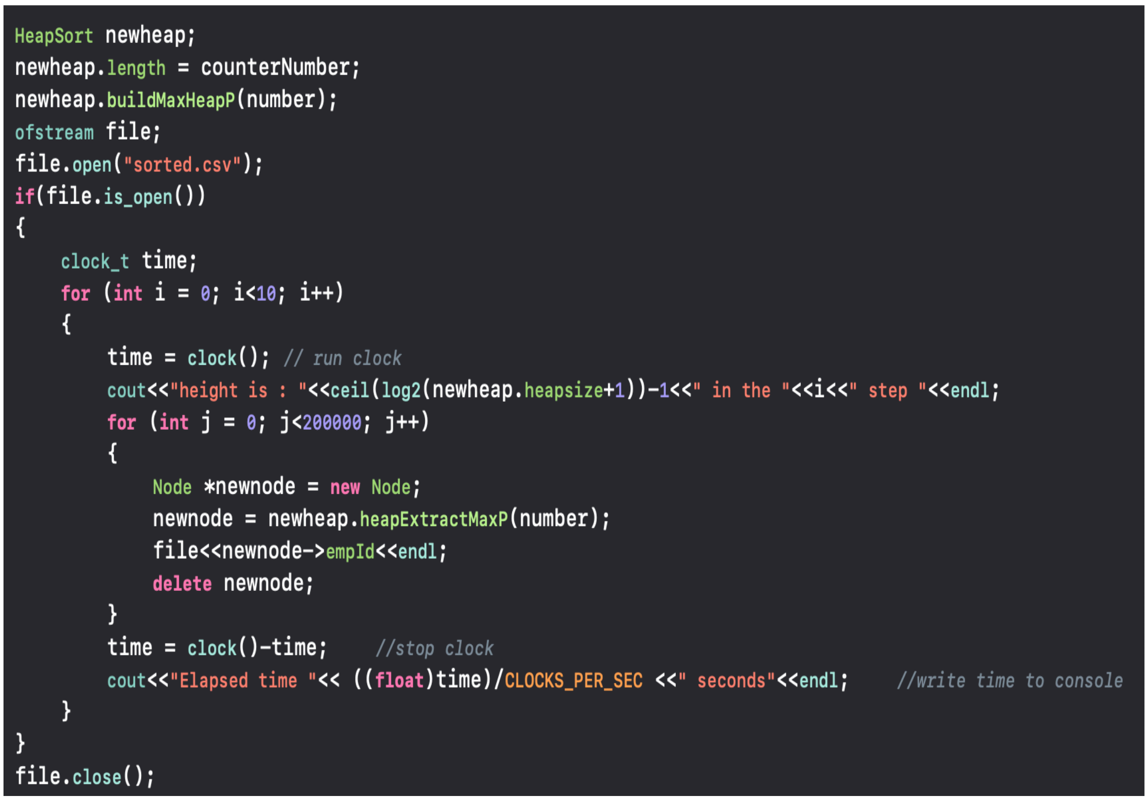
PART C

In this step I made the operations for numbers.csv. I read the file into an array and built the heap. Then I sorted the file in 10 steps by using the largest 200000. For each step I printed the elapsed time and height. For checking the correctness of my result, I wrote the sorted arrays into a file named “sorted.csv” and checked the values.



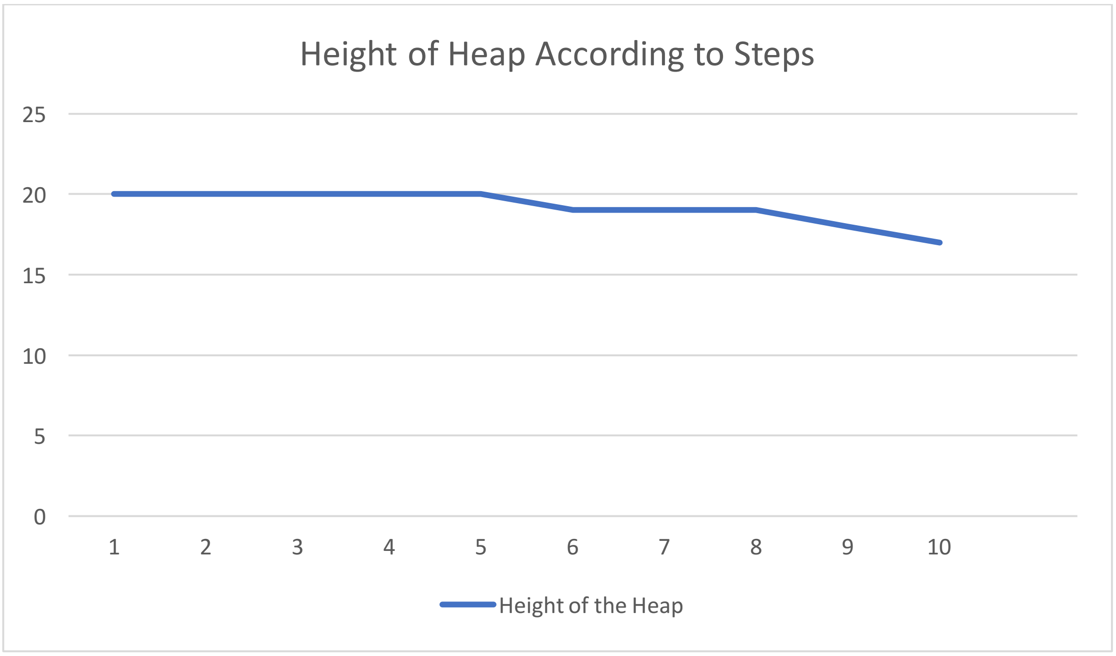
```
Node *number = new Node[2000000];
//sort numbers.csv
ifstream numbers;
string liner;
int counterNumber = 0;
numbers.open("numbers.csv");
if(numbers.is_open())
{
     while (!numbers.eof())
     {
          getline(numbers,liner);
          int value = atoi(liner.c_str());
          number[counterNumber].empId = value;
          number[counterNumber].performance = value;
          number[counterNumber].totalCalls = value;
          counterNumber++;
     }
}
else
{
     cout<<" Can not open numbers.csv "<<endl;
}
```

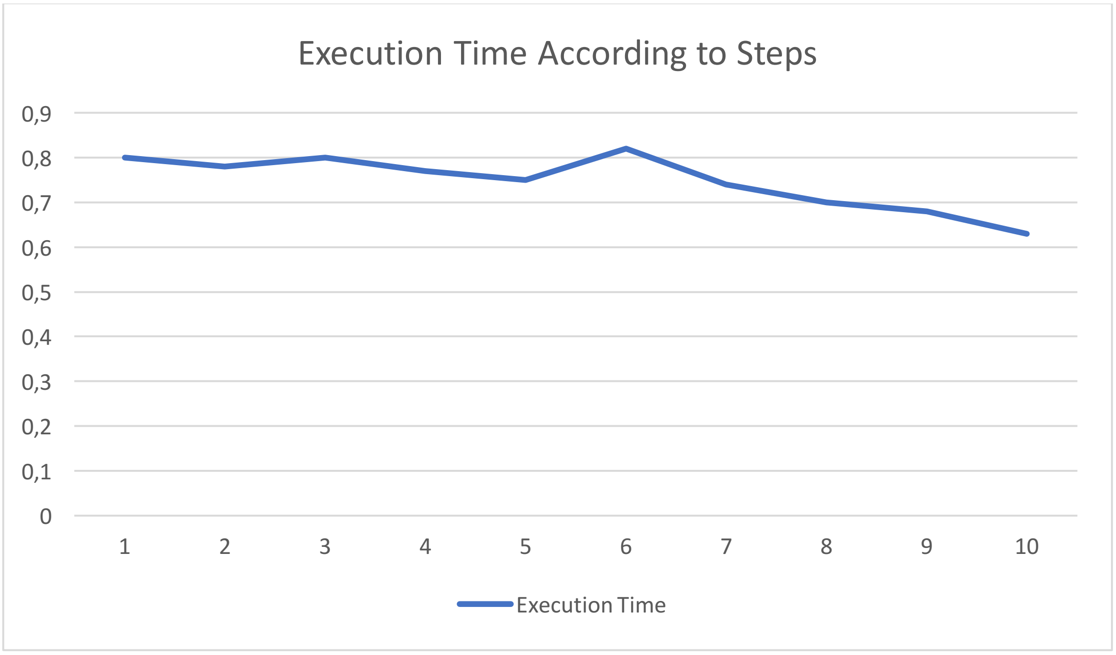
read the numbers.csv



```
HeapSort newheap;
newheap.length = counterNumber;
newheap.buildMaxHeapP(number);
ofstream file;
file.open("sorted.csv");
if(file.is_open())
{
    clock_t time;
    for (int i = 0; i<10; i++)
    {
        time = clock(); // run clock
        cout<<"height is : "<<ceil(log2(newheap.heapsize+1))-1<<" in the "<<i<<" step "<<endl;
        for (int j = 0; j<200000; j++)
        {
            Node *newnode = new Node;
            newnode = newheap.heapExtractMaxP(number);
            file<<newnode->empId<<endl;
            delete newnode;
        }
        time = clock()-time;     //stop clock
        cout<<"Elapsed time "<< ((float)time)/CLOCKS_PER_SEC <<" seconds"<<endl;     //write time to console
    }
}
file.close();
```

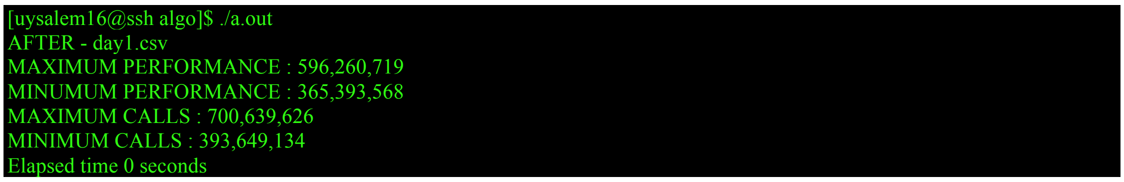
write to file in 10 steps by extracting



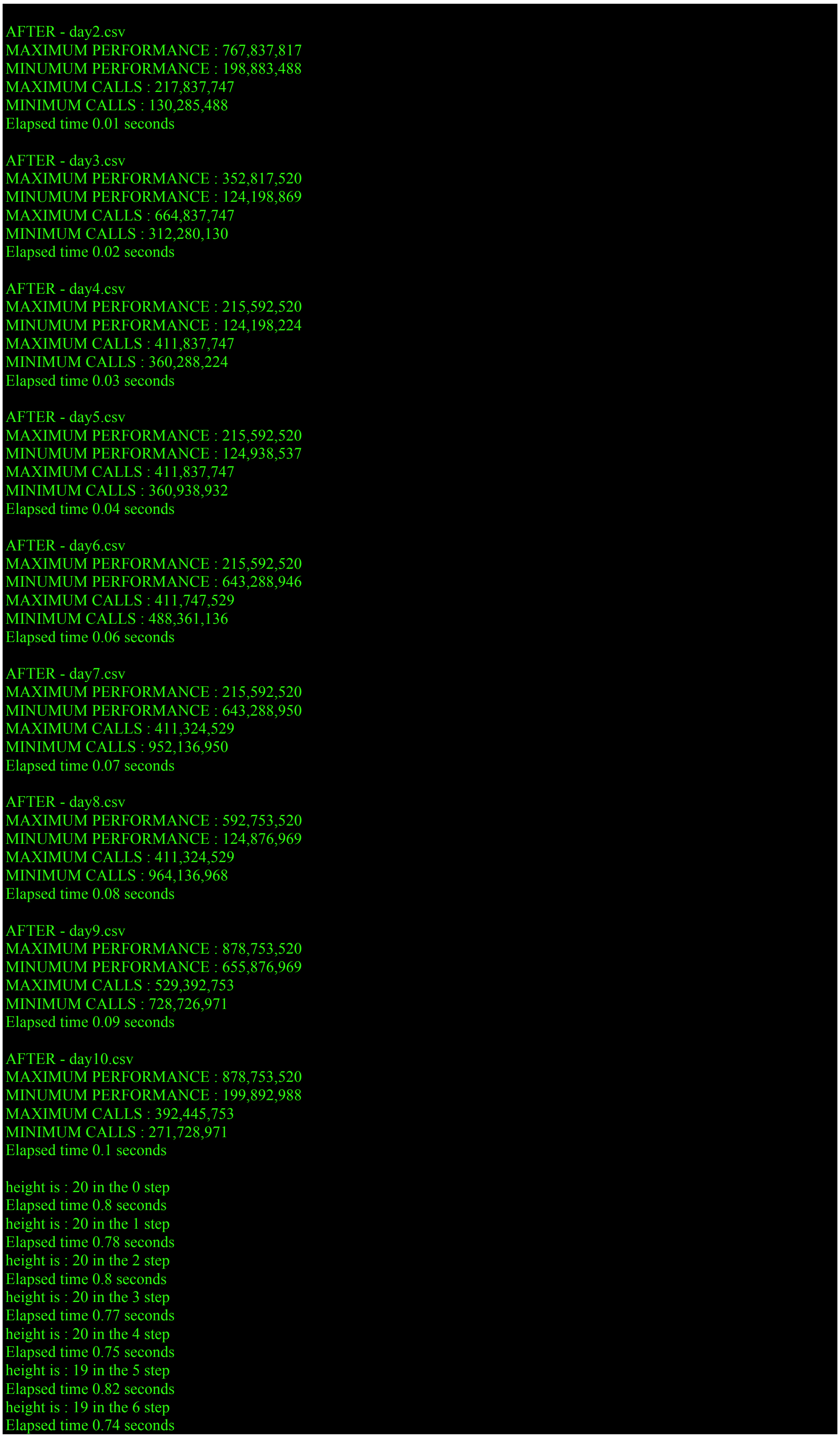


3.FINAL EXPLANATION

In the final step, I can say that I have learned many things about heapsort algorithm. I also developed myself about managing a heap. I gave the all outputs of my code below. From the plots I can say that; if the heap height decrease, elapsed time decreases according to height. If the heapsize increase, elapsed time increase when building the heap. You can see elapsed time from my ssh output below.



```
[uysalem16@ssh algo]$ ./a.out
AFTER - day1.csv
MAXIMUM PERFORMANCE : 596,260,719
MINUMUM PERFORMANCE : 365,393,568
MAXIMUM CALLS : 700,639,626
MINIMUM CALLS : 393,649,134
Elapsed time 0 seconds
```



```
AFTER - day2.csv
MAXIMUM PERFORMANCE : 767,837,817
MINUMUM PERFORMANCE : 198,883,488
MAXIMUM CALLS : 217,837,747
MINIMUM CALLS : 130,285,488
Elapsed time 0.01 seconds

AFTER - day3.csv
MAXIMUM PERFORMANCE : 352,817,520
MINUMUM PERFORMANCE : 124,198,869
MAXIMUM CALLS : 664,837,747
MINIMUM CALLS : 312,280,130
Elapsed time 0.02 seconds

AFTER - day4.csv
MAXIMUM PERFORMANCE : 215,592,520
MINUMUM PERFORMANCE : 124,198,224
MAXIMUM CALLS : 411,837,747
MINIMUM CALLS : 360,288,224
Elapsed time 0.03 seconds

AFTER - day5.csv
MAXIMUM PERFORMANCE : 215,592,520
MINUMUM PERFORMANCE : 124,938,537
MAXIMUM CALLS : 411,837,747
MINIMUM CALLS : 360,938,932
Elapsed time 0.04 seconds

AFTER - day6.csv
MAXIMUM PERFORMANCE : 215,592,520
MINUMUM PERFORMANCE : 643,288,946
MAXIMUM CALLS : 411,747,529
MINIMUM CALLS : 488,361,136
Elapsed time 0.06 seconds

AFTER - day7.csv
MAXIMUM PERFORMANCE : 215,592,520
MINUMUM PERFORMANCE : 643,288,950
MAXIMUM CALLS : 411,324,529
MINIMUM CALLS : 952,136,950
Elapsed time 0.07 seconds

AFTER - day8.csv
MAXIMUM PERFORMANCE : 592,753,520
MINUMUM PERFORMANCE : 124,876,969
MAXIMUM CALLS : 411,324,529
MINIMUM CALLS : 964,136,968
Elapsed time 0.08 seconds

AFTER - day9.csv
MAXIMUM PERFORMANCE : 878,753,520
MINUMUM PERFORMANCE : 655,876,969
MAXIMUM CALLS : 529,392,753
MINIMUM CALLS : 728,726,971
Elapsed time 0.09 seconds

AFTER - day10.csv
MAXIMUM PERFORMANCE : 878,753,520
MINUMUM PERFORMANCE : 199,892,988
MAXIMUM CALLS : 392,445,753
MINIMUM CALLS : 271,728,971
Elapsed time 0.1 seconds

height is : 20 in the 0 step
Elapsed time 0.8 seconds
height is : 20 in the 1 step
Elapsed time 0.78 seconds
height is : 20 in the 2 step
Elapsed time 0.8 seconds
height is : 20 in the 3 step
Elapsed time 0.77 seconds
height is : 20 in the 4 step
Elapsed time 0.75 seconds
height is : 19 in the 5 step
Elapsed time 0.82 seconds
height is : 19 in the 6 step
Elapsed time 0.74 seconds
```

```
height is : 19 in the 7 step
Elapsed time 0.7 seconds
height is : 18 in the 8 step
Elapsed time 0.68 seconds
height is : 17 in the 9 step
Elapsed time 0.63 seconds
```