ISTANBUL TECHNICAL UNIVERSITY

COMPUTER ENGINEERING DEPARTMENT

BLG 335E

Analysis of Algorithms I
HOMEWORK REPORT

HOMEWORK NO : 2

HOMEWORK DATE : 02.12.2022

150200915 : ADIL MAHMUDLU

SPRING 2022

Contents

1 DATA STRUCTURES

The only Data Structure used in the implementation is Max Heap. The Max Heap contains data such as the double vector that constitutes the heap, the capacity, and the heapSize. The data structure also has multiple functions, including but not limited to heapify, buildHeap, and heapSort.

1.1 Heapify function

```
1  heapify(i)
2  l <- 2*i+1
3  r <- 2*i+2
4  largest <- i
5  if l < heapSize and heap[l] > heap[i]
6      then largest <- l
7  if r < heapSize and heap[r] > heap[largest]
8      then largest <- r
9  if largest != i
10     then exchange A[i] <-> A[largest]
11     heapify(largest)
```

In the code, lines 2-10 are done in O(1) complexity, but line 11 depends on the input parameter i. The running time complexity can be described as $T(n) \leq T(2n/3) + O(1)$ which results in O(logn) complexity according to the second case of Master Method.

1.2 BuildHeap function

```
1  buildHeap()
2  for i <- (heapsize-2)/2 downto 0 do
3      heapify(i)
```

The time complexity of this function depends on the number of $heapify$ functions called. Although there might be n/2 calls in the worst case, most of them take less time than worst case because they are close to leaves. Therefore, from calculations, the tight worst running time is $O(n)$.

1.3 HeapSort function

```
1  buildHeap()
2  if not sorted
3      then size <- heapSize-1                  O(1)
4      buildHeap()                              O(n)
5      for i < heapSize-1 downto 1 do           O(n-1)
6          exchange heap[0] <-> heap[i]         O(1)
7          heapSize <- heapSize-1               O(1)
```

```
8           heapify(0)                                  O(logn)
9       heapSize <- size                                O(1)
10      sorted <- true                                  O(1)
```

The time complexity is $O(1)+O(n)+O(n-1)*(O(1)+O(1)+O(logn))+O(1)+O(1) = O(nlogn)$.

1.4 Insert function

```
1  insert(k)
2  if heapSize == capacity
3      then throw error
4  else
5      then heapSize <- heapSize+1                     O(1)
6      i <- heapSize-1                                 O(1)
7      heap[i] <- k                                    O(1)
8      while i > 0 and heap[(i-1)/2] < heap[i] do      O(logn)
9          exchange heap[i] <-> heap[(i-1)/2]          O(1)
10         i <- (i-1)/2                                O(1)
11     sorted <- false                                 O(1)
12         sum <- sum + k                              O(1)
```

The time complexity can be calculated via $O(1)+O(1)+O(1)+O(logn)*(O(1)+O(1))+O(1)+O(1) = O(logn)$.

1.5 Mean calculation

```
1  getMean()
2  return sum/heapSize    O(1)
```

The time complexity is $O(1)$.

1.6 Standard Deviation calculation

```
1  getSTD()
2  mean <- getMean()                        O(1)
3  total <- 0                               O(1)
4  for i <- o to heapSize do                O(n)
5      total <- power(heap[i]-mean, 2)      O(1)
6  total <- sqrt(total/(heapSize-1))        O(1)
```

The time complexity is $O(1)+O(1)+O(n)*O(1)+O(1) = O(n)$.

1.7 Minimum calculation

```
1  getMin()
2  return heap[0]        O(1)
```

The time complexity is $O(1)$.

1.8 First quartile calculation

```
1 getFirstq()
2 low <- (heapSize-1)/4                          O(1)
3 high <- (heapSize-1)/4+1                       O(1)
4 part <- (heapSize-1)/4.0 - low                 O(1)
5 return heap[low]*(1-part)+heap[high]*part      O(1)
```

The time complexity is $O(1)$.

1.9 Median calculation

```
1 getMedian()
2 if heapSize%2 == 1
3     then return heap[heapSize/2]                                  O(1)
4 else
5     then return (heap[heapSize/2]+heap[heapSize/2-1])/2           O(1)
```

The time complexity is $O(1)$.

1.10 Third quartile calculation

```
1 getThirdq()
2 low <- 3*(heapSize-1)/4                        O(1)
3 high <- 3*(heapSize-1)/4+1                     O(1)
4 part <- 3.0*(heapSize-1)/4.0 - low             O(1)
5 return heap[low]*(1-part)+heap[high]*part      O(1)
```

The time complexity is $O(1)$.

1.11 Maximum calculation

```
1 getMax()
2 if sorted == 1
3     then return heap[heapSize-1]     O(1)
4 else
5     then return heap[0]              O(1)
```

The time complexity is $O(1)$.

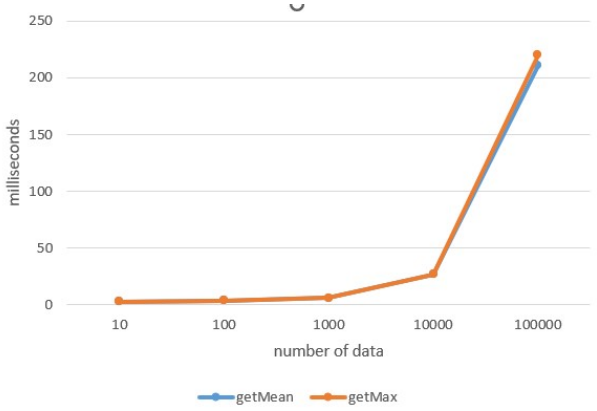
1.12 Function usages in estimating statistics

The functions were used depending on the estimators and operations given in the file. If the estimators include min, firstq, median, or thirdq, sorting the heap is necessary, otherwise heap is not sorted. When reading the add operation from the input file, insert function is used with respective parameter. On the other hand, if the print operation is read fromm the input file, the code sorts the heap if necessary, as stated before, and calls the according estimator functions. The heapSort and insert functions are the only

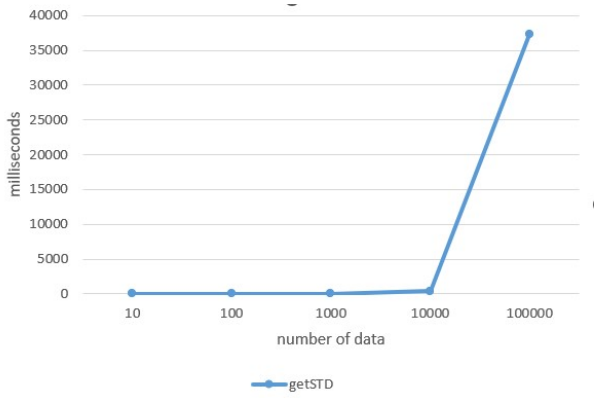
functions that are used in the main.cpp file. The rest of the heap functions, namely, heapify and buildHeap, are only called inside the heapSort function and within each other.

2 Statistics

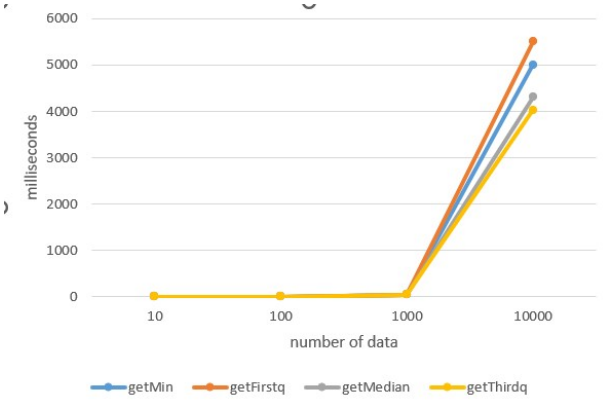
The exsecution time of the program for multiple different number of data and for different types of estimations are given in the below line plots. The getMean, getSTD and getMax functions don't call heapify, heapSort, or buildHeap functions. The number of insert functions used in the estimation is the same as the number of data. The number of times the heapSort and buildHeap functions were called is the same, and are given for minimum and quartile estimator functions. THe heapify function's usage is not suitable for line plot, as it does not convey much information.



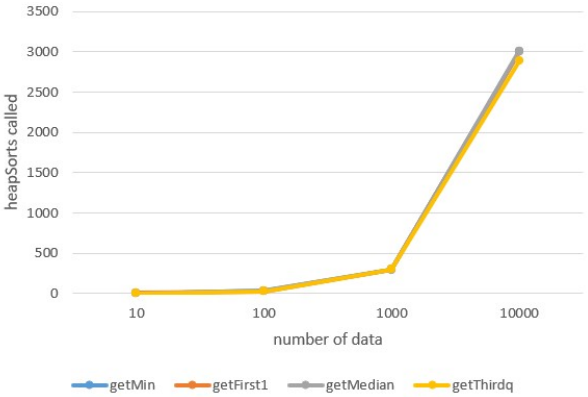
(a) Execution times of getMax and getMean functions



(b) Execution time of getSTD function



(c) Execution time of quartiles and minimum estimator functions



(d) Number of calls to the heapSort function when estimating quartiles and minimum

3 SLIDING WINDOW APPROACH

For finding minimum, sliding window approach gives the better time complexity than the one we got using heapSort. Sliding window approach gives O(n) complexity, while using heapSort before finding minimum gives O(nlogn) complexity. Finding mean was done by saving the sum data in the heap itself, and therefore sliding window approach is obsolete. For maximum, it takes O(1) time so sliding window approach is not needed here either. On the other hand, it is not possible to use this technique for finding quartiles. STD was found using this technique.

REFERENCES