NAME : - Jinggui Tan & Siddhant Barua
CWid :- 10427381 & 10439929

ASSIGNMENT 3 Report

The following snapshots lists our implementation of our minHeap, maxHeap and medianHeap function.

The Swap Function

```
29  void maxHeap::swap(int i, int j){
30      int foobar = heapArray[i];
31      heapArray[i] = heapArray[j];
32      heapArray[j] = foobar;
33  }
```

Swap the value of index i and j in the array of the heap.

The grow function.

```
17  void maxHeap::grow(){
18      if(heapSize>=capacity){
19          capacity*=2;
20          int* newArray = new int[capacity];
21          for(int i=0;i<heapSize;i++){
22              newArray[i]=heapArray[i];
23          }
24          delete[] heapArray;
25          heapArray = newArray;
26      }
27  }
28
```

Each time we insert new key, we call grow() to help us check if we have enough space for new key. If space is limited, we double the space.

The Delete Function

```
61  void maxHeap::deleteKey(int i)
62  {
63      swap(heapSize-1,i);
64      heapSize--;
65      maxHeapify(i);
66  }
```

This calls the swap function at the second last position of the HeapArray and swaps it with i and then decrements the heapSize by 1. After this maxHeapify is called on the subheap(for which root is index i) to make sure the remaining nodes in the heap follow the maxHeap or minHeap properties.

NAME : - Jinggui Tan & Siddhant Barua
CWid :- 10427381 & 10439929

For Deleting in minHeap we just change the maxHeapify call to minHeapify

```
63  void minHeap::deleteKey(int i)
64  {
65      swap(heapSize-1,i);
66      heapSize--;
67      minHeapify(i);
68  }
```

The Insert Key function maxHeap.

```
35  void maxHeap::insertKey(int k)
36  {
37      grow();
38      heapArray[heapSize]=k;
39      heapSize++;
40      for(int i=parent(heapSize-1);i>=0;i--)
41          maxHeapify(i);
42  }
43
```

Call grow() to check capacity to make sure there is no overflow.
Then add the new key at the end of the heap as new leaf.
Because we can't guarantee the order as new key inserted, we need to rebuild the heap. Call max/minHeapify for each node who has child(that means from (n-2))/2 to root, n is the size of heap.
Finally don't forget to increase heapSize

The InsertKey function minHeap.

```
36  void minHeap::insertKey(int k)
37  {
38      grow();
39      heapArray[heapSize]=k;
40      heapSize++;
41      for(int i=parent(heapSize-1);i>=0;i--)
42          minHeapify(i);
43  }
```

It's the same as the one from maxHeap. Add then rebuild.

NAME : - Jinggui Tan & Siddhant Barua
CWid :- 10427381 & 10439929

The extractMax Function

```
54  int maxHeap::extractMax()
55  {
56      int offer = heapArray[0];
57      deleteKey(0);
58      return offer;
59  }
```

As everyone knows, the root holds the max value of maxHeap. So we get the value of root and return it. Done? No. We need to delete root and rebuild the maxHeap to make sure the second max value comes to root. Delete function helps heap to follow maxheap or minheap properties.

The extractMin Function

```
55  int minHeap::extractMin()
56  {
57      // just want return offer XD
58      int offer=heapArray[0];
59      deleteKey(0);
60      return offer;
61  }
62
```

Stay the same as maxHeap.

NAME : - Jinggui Tan & Siddhant Barua
CWid :- 10427381 & 10439929

MaxHeapify Function

```
68  void maxHeap::maxHeapify(int i)
69  {
70      int l=left(i);
71      int r=right(i);
72      if(r<heapSize && heapArray[l]>heapArray[r]){
73          if(heapArray[l]>heapArray[i]){
74              swap(l,i);
75              maxHeapify(l);
76          }
77      }else if(r<heapSize && heapArray[r]>heapArray[l]){
78          if(heapArray[r]>heapArray[i]){
79              swap(r,i);
80              maxHeapify(r);
81          }
82      }else if(l<heapSize){
83          if(heapArray[l]>heapArray[i]){
84              swap(l,i);
85              maxHeapify(l);
86          }
87      }
88  }
```

If there are left child and right child, we compare value of parent and childs. If any child has greater value(smaller value for minHeap), then exchange this child with parent. Like insertion, exchange may break the heap law, we need to rebuild the subheap of that child.
If there is no right child, just compre left child with parent then decide if we need to exchange and rebuild.
If parent has greater value than any child, then we do nothing. Just return.

NAME : - Jinggui Tan & Siddhant Barua
CWid :- 10427381 & 10439929

MinHeapify Function

```
70  void minHeap::minHeapify(int i)
71  {
72      int l=left(i);
73      int r=right(i);
74      if(r<heapSize && heapArray[l]<heapArray[r]){
75          if(heapArray[l]<heapArray[i]){
76              swap(l,i);
77              minHeapify(l);
78          }
79      }else if(r<heapSize && heapArray[r]<heapArray[l]){
80          if(heapArray[r]<heapArray[i]){
81              swap(r,i);
82              minHeapify(r);
83          }
84      }else if(l<heapSize){
85          if(heapArray[l]<heapArray[i]){
86              swap(l,i);
87              minHeapify(l);
88          }
89      }
90  }
```

It's the same as maxHeapify

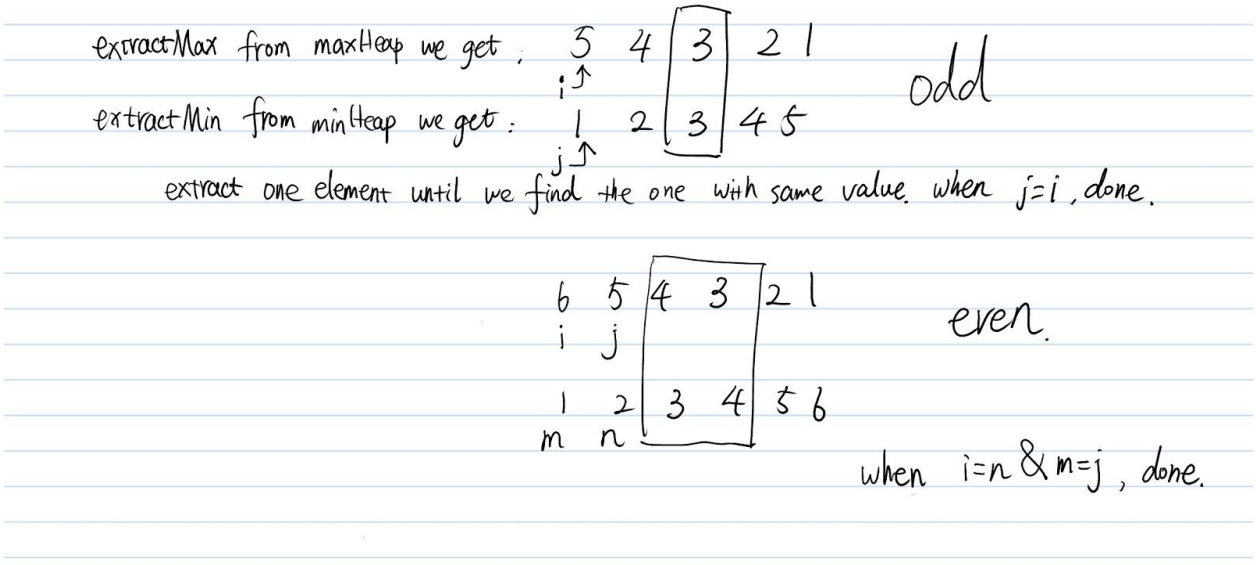
NAME : - Jinggui Tan & Siddhant Barua
CWid :- 10427381 & 10439929

Median Heaps Function

```
30  int medianHeaps::getMedian()
31  {
32      if(maxH->size()==0){
33          cout<<"not enough elements!";
34          return -1;
35      }
36      if(maxH->size()%2!=0){
37          int min, max;
38          for(int i=0; i<maxH->size();i++){
39              min=minH->extractMin();
40              max=maxH->extractMax();
41              if(min==max) {
42                  return min;
43              }
44          }
45      }else{
46          int min1, min2, max1, max2;
47          min1=minH->extractMin();
48          max1=maxH->extractMax();
49          for(int i=0; i<maxH->size();i++){
50              min2=minH->extractMin();
51              max2=maxH->extractMax();
52              if(min1==max2 && min2==max1){
53                  return (min1+min2)/2;
54              }
55              min1=min2;
56              max1=max2;
57          }
58      }
59
60  }
61
```

There are 2 cases, odd and even numbers of elements. Check the picture below.

NAME : - Jinggui Tan & Siddhant Barua
CWid :- 10427381 & 10439929



Now we test for the following cases.

1. *No Elements - test pass (Main won't let empty element , it automatically sets m to at least 1)*
2. *One Element - test pass*
3. *Odd Elements - test pass*
4. *Even Elements - test pass*
5. *Elements with all negative numbers - test pass*
6. *Elements with partial negative numbers - test pass*
7. *All the same elements - test pass*
8. *Large Input - test pass*

NAME : - Jinggui Tan & Siddhant Barua
CWid :- 10427381 & 10439929

Snapshots of the test cases

```
tanjinggui@penguin:~/cs590hw3$ ./hw3  1
insert: 15
The median was found, and maintained successfully!
tanjinggui@penguin:~/cs590hw3$ ./hw3  1
insert: 51
The median was found, and maintained successfully!
```

```
tanjinggui@penguin:~/cs590hw3$ ./hw3  100
100 numbers inserted.
The median was found, and maintained successfully!
```

```
tanjinggui@penguin:~/cs590hw3$ ./hw3  99
99 numbers inserted.
The median was found, and maintained successfully!
```

```
tanjinggui@penguin:~/cs590hw3$ ./hw3 5
insert: -42
insert: -13
insert: -67
insert: -93
insert: -83
The median was found, and maintained successfully!
tanjinggui@penguin:~/cs590hw3$ ./hw3 5
insert: -67
insert: -3
insert: -81
insert: -48
insert: -6
The median was found, and maintained successfully!
```

```
tanjinggui@penguin:~/cs590hw3$ ./hw3  50000
50000 numbers inserted.
The median was found, and maintained successfully!
```

NAME : - Jinggui Tan & Siddhant Barua
CWid :- 10427381 & 10439929

```
tanjinggui@penguin:~/cs590hw3$ ./hw3  5
insert: 75
insert: -40
insert: 37
insert: -80
insert: 72
5 numbers inserted.
The median was found, and maintained successfully!
```