Please see attached Code for implementations as required

[x] **7.1 (2pt) Sort the sequence (3, 1, 4, 1, 5, 9, 2, 6, 5) using insertion sort**

Unsorted Array Problem 7.1

[0]: 3

[1]: 1

[2]: 4

[3]: 1

[4]: 5

[5]: 9

[6]: 2

[7]: 6

[8]: 5

Sorted Array Problem 7.1:

[0]: 1

[1]: 1

[2]: 2

[3]: 3

[4]: 4

[5]: 5

[6]: 5

[7]: 6

[8]: 9

[x] **7.2 (1pt) What is the running time of insertion Sort if all elements are equal?**

O(N)

[x] **7.11 (4pt: 1pt for building heap, 2pt for sort, 1pt for memory use) Show how heapsort processes the input (142, 543, 123, 65, 453, 879, 572, 434, 111, 242, 811, 102)**

Please see attached Code for implementations as required

Unsorted Array Problem 7.11

[0]: 142

[1]: 543

[2]: 123

[3]: 65

[4]: 453

[5]: 879

[6]: 572

[7]: 434

[8]: 111

[9]: 242

[10]: 811

[11]: 102

Enqueued Element: 142

Binary Heap Data

[0]: 0

[1]: 142

[2]: 0

[3]: 0

[4]: 0

[5]: 0

[6]: 65

[7]: 0

[8]: 0

[9]: 0

[10]: 0

[11]: 0

[12]: 0

Please see attached Code for implementations as required

Enqueued Element: 543

Binary Heap Data

[0]: 0

[1]: 142

[2]: 543

[3]: 0

[4]: 0

[5]: 0

[6]: 65

[7]: 0

[8]: 0

[9]: 0

[10]: 0

[11]: 0

[12]: 0

Enqueued Element: 123

Binary Heap Data

[0]: 0

[1]: 123

[2]: 543

[3]: 142

[4]: 0

[5]: 0

[6]: 65

[7]: 0

[8]: 0

[9]: 0

[10]: 0

[11]: 0

[12]: 0

Please see attached Code for implementations as required

Enqueued Element: 65

Binary Heap Data

[0]: 0

[1]: 65

[2]: 123

[3]: 142

[4]: 543

[5]: 0

[6]: 65

[7]: 0

[8]: 0

[9]: 0

[10]: 0

[11]: 0

[12]: 0

Enqueued Element: 453

Binary Heap Data

[0]: 0

[1]: 65

[2]: 123

[3]: 142

[4]: 543

[5]: 453

[6]: 65

[7]: 0

[8]: 0

[9]: 0

[10]: 0

[11]: 0

[12]: 0

Please see attached Code for implementations as required

Enqueued Element: 879

Binary Heap Data

[0]: 0

[1]: 65

[2]: 123

[3]: 142

[4]: 543

[5]: 453

[6]: 879

[7]: 0

[8]: 0

[9]: 0

[10]: 0

[11]: 0

[12]: 0

Enqueued Element: 572

Binary Heap Data

[0]: 0

[1]: 65

[2]: 123

[3]: 142

[4]: 543

[5]: 453

[6]: 879

[7]: 572

[8]: 0

[9]: 0

[10]: 0

[11]: 0

[12]: 0

Please see attached Code for implementations as required

Enqueued Element: 434

Binary Heap Data

[0]: 0

[1]: 65

[2]: 123

[3]: 142

[4]: 434

[5]: 453

[6]: 879

[7]: 572

[8]: 543

[9]: 0

[10]: 0

[11]: 0

[12]: 0

Enqueued Element: 111

Binary Heap Data

[0]: 0

[1]: 65

[2]: 111

[3]: 142

[4]: 123

[5]: 453

[6]: 879

[7]: 572

[8]: 543

[9]: 434

[10]: 0

[11]: 0

[12]: 0

Please see attached Code for implementations as required

Enqueued Element: 242

Binary Heap Data

[0]: 0

[1]: 65

[2]: 111

[3]: 142

[4]: 123

[5]: 242

[6]: 879

[7]: 572

[8]: 543

[9]: 434

[10]: 453

[11]: 0

[12]: 0

Enqueued Element: 811

Binary Heap Data

[0]: 0

[1]: 65

[2]: 111

[3]: 142

[4]: 123

[5]: 242

[6]: 879

[7]: 572

[8]: 543

[9]: 434

[10]: 453

[11]: 811

[12]: 0

Please see attached Code for implementations as required

Enqueued Element: 102

Binary Heap Data

[0]: 0

[1]: 65

[2]: 111

[3]: 102

[4]: 123

[5]: 242

[6]: 142

[7]: 572

[8]: 543

[9]: 434

[10]: 453

[11]: 811

[12]: 879

The heap is empty.

Sorted Array Problem 7.11

[0]: 65

[1]: 102

[2]: 111

[3]: 123

[4]: 142

[5]: 242

[6]: 434

[7]: 453

[8]: 543

[9]: 572

[10]: 811

[11]: 879

Please see attached Code for implementations as required

[X] **7.12 (1pt) What is the running time of heapsort for presorted input?**

O(log(N)) to build heap * O(N) to evaluate each item.

So, the time complexity is O(Nlog(N))

[x] **7.15 (2pt) Sort (3, 1, 4, 1, 5, 9, 2, 6) using mergesort**

Unsorted Array problem 7.15

[0]: 3

[1]: 1

[2]: 4

[3]: 1

[4]: 5

[5]: 9

[6]: 2

[7]: 6

Sorted Array Problem 7.15:

[0]: 1

[1]: 1

[2]: 2

[3]: 3

[4]: 4

[5]: 5

[6]: 6

[7]: 9

[X] **7.17 (3pt) Determine the running time of merge sort for**

[X] **a. Sorted input:**

O(Nlog(N)) this is because it still divides the array log(N) times and makes N comparisons whether or not its sorted so O(log(N)) * O(N)

[X] **b. Reverse-Ordered Input:**

Please see attached Code for implementations as required

O(Nlog(N)) this is because it splits the Array log(N) times, makes N comparisons and N copies so O(log(N)) * O(N) + O(N)

[X] **c. Random input:**

O(Nlog(N)) is the average case, so this is for random input, but the same method for reverse order applies here as the comparisons and copying are still done

[x] **Chapter 7 Bonus Homework**

[x] **7.3 (1pt) Suppose we exchange elements a[i] and a[i+k] which were originally out of order. Prove that at least 1 and at most 2k-1 inversions are removed.**

We know that an inversion occurs only if an object is out of order.

The best-case scenario for this problem is if a[i] and a[i+k] are the only objects out of order.

Average case is N(N-1)/4 inversions.

If a[i] and a[i+k] are the only objects out of order this removes every inversion but the initial

Implying N(N-1)/4 - N-1((N-1)-1)/4 = = -(N-1) = N-1 inversions removed.

Therefore, there will be N-1 inversions removed.

Now if Every object is out of order in the worst case

There will be N(N-1)/4 - N(N-1)/4 + 1 = -1 = 1 inversion removed

Therefore, there will be at least 1 and at most 2k-1 inversions removed

[x] **7.19 (1pt) Sort (3, 1, 4, 1, 5, 9, 2, 6, 5, 3, 5) using quicksort with median-of-three partitioning and a cutoff of 3.**

Unsorted Array Problem 7.19:

[0]: 3

[1]: 1

[2]: 4

[3]: 1

[4]: 5

Please see attached Code for implementations as required

[5]: 9

[6]: 2

[7]: 6

[8]: 5

[9]: 3

[10]: 5

Sorted Array Problem 7.19:

[0]: 1

[1]: 1

[2]: 2

[3]: 3

[4]: 3

[5]: 4

[6]: 5

[7]: 5

[8]: 5

[9]: 6

[10]: 9