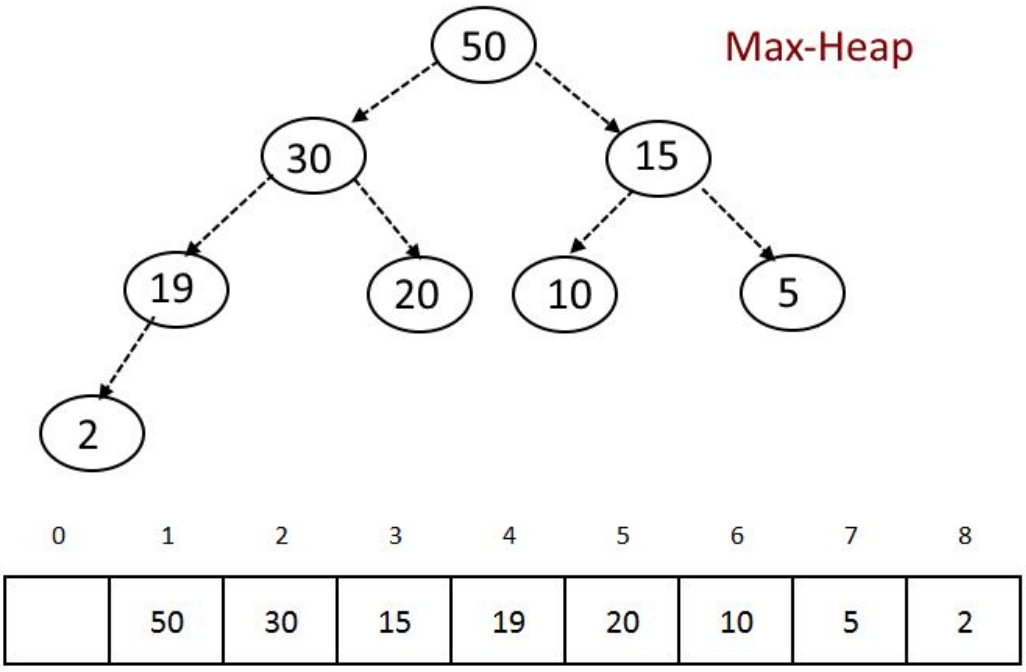
Data Structures

Heap Implementation

Implementing Binary Heap & Sorting Techniques



Names :

Khadija Assem Saad (No. 27)

Nourhan Magdi Mohamed (No. 69)

Source Code :

https://github.com/khadijaAssem/Heaps

Content

- Introduction: page(3).
- INode interface: page(3).
- IHeap interface: page(4).
- ISort interface: page(5).
- Execution time:page(6).
- Tests :page(11).

Introduction

Implementing a binary MAX-heap using array of nodes each node has an index and a value stored in it the root node is of index 1 and the left node is calculated by the equation :
leftNode index = ParentIndex*2
and right Node index is calculated by the equation :
RhightNode index = ParentIndex*2 +1.

For the INode interface implementation :

Functions:

- **Get left child:**

  Returns the left child of the current and returns null if it was a leaf node.

- **Get Right child:**

  Returns the right child of the current and returns null if it was a leaf node.

- **Get Parent:**

  Returns the parent node of the current node.

- **Values Getter and Setter:**

  Sets and gets the value stored in the current node.

For the IHeap interface implementation:

Functions:

- **Get Root:**

  Returns the root of the binary heap.

- **Size:**

  Returns the size of the binary heap.

- **Heapify:**

  Keeps the maximum property of the MAX Heap runs in O(lg n).

- **Extract:**

  Removes the root of the MAX Heap (The Greatest Element) and keeps the property of the MAX Heap by the heapify runs in O(lg n).

- **Insert:**

  Creates a new node in the heap and keeps the property of the MAX Heap.

- **Build:**

  Builds a MAX Heap from a given arrayList.

For the ISort interface implementation:

Functions:

➔ **HeapSort:**

Sorts a given arraylist and returns a heap using heap sort algorithm as following :

- Build a max heap from the input data.
- Loop for all elements and extract them one by one .
- The execution time is O(nlogn)

➔ **SortSlow:**

Sorts a given arraylist by a slow technique and we use bubble sort as following:

- Loop for all elements and repeatedly swapping the adjacent elements if they are in wrong order.
- The execution time is O(n^2)

➔ **SortFast:**

Sorts a given arraylist by a fast technique and we use merge sort as following:

- divide input array in two halves
- call itself for the two halves
- then merge the two sorted halves.
- The execution time is O(nlogn)

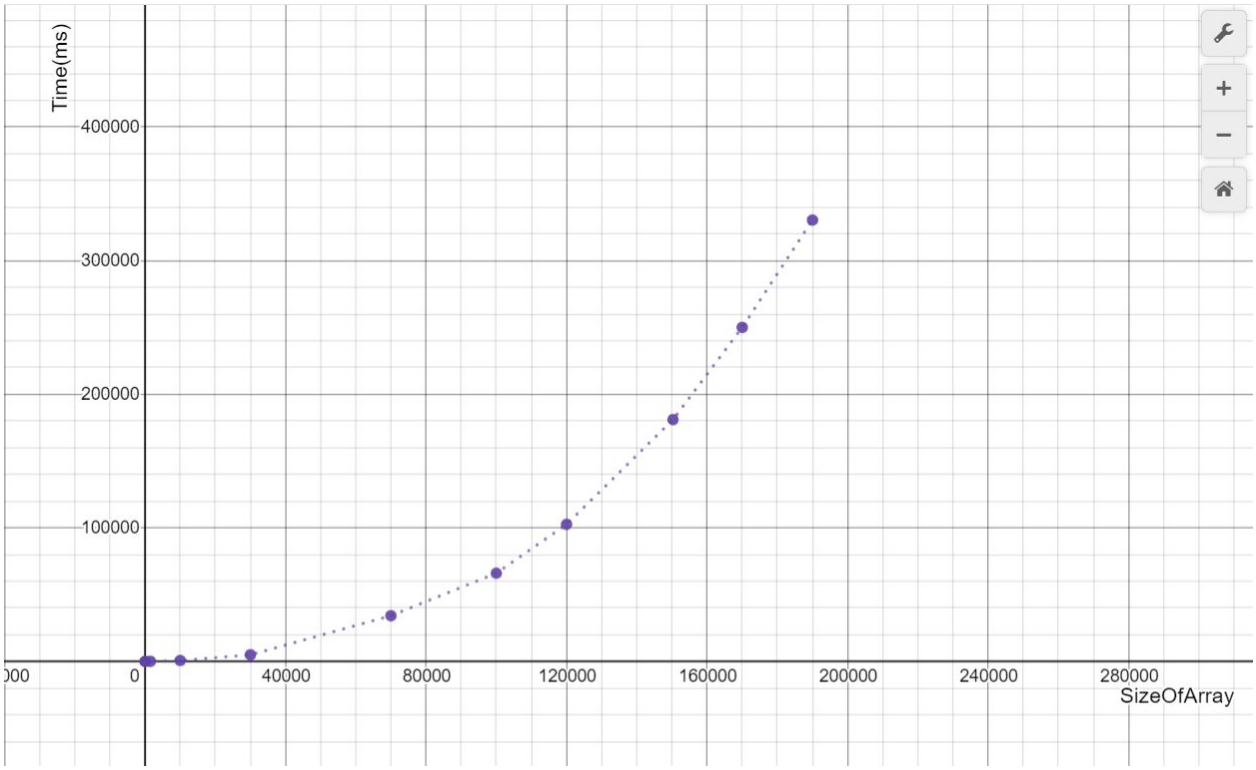
Execution time:

- To test our implementation and analyze the running time performance, we generate a dataset of random numbers and plot the relationship between the execution time of the sorting algorithm versus the input size as following :

For slow sort O(n^2):

Table & graph are like following :

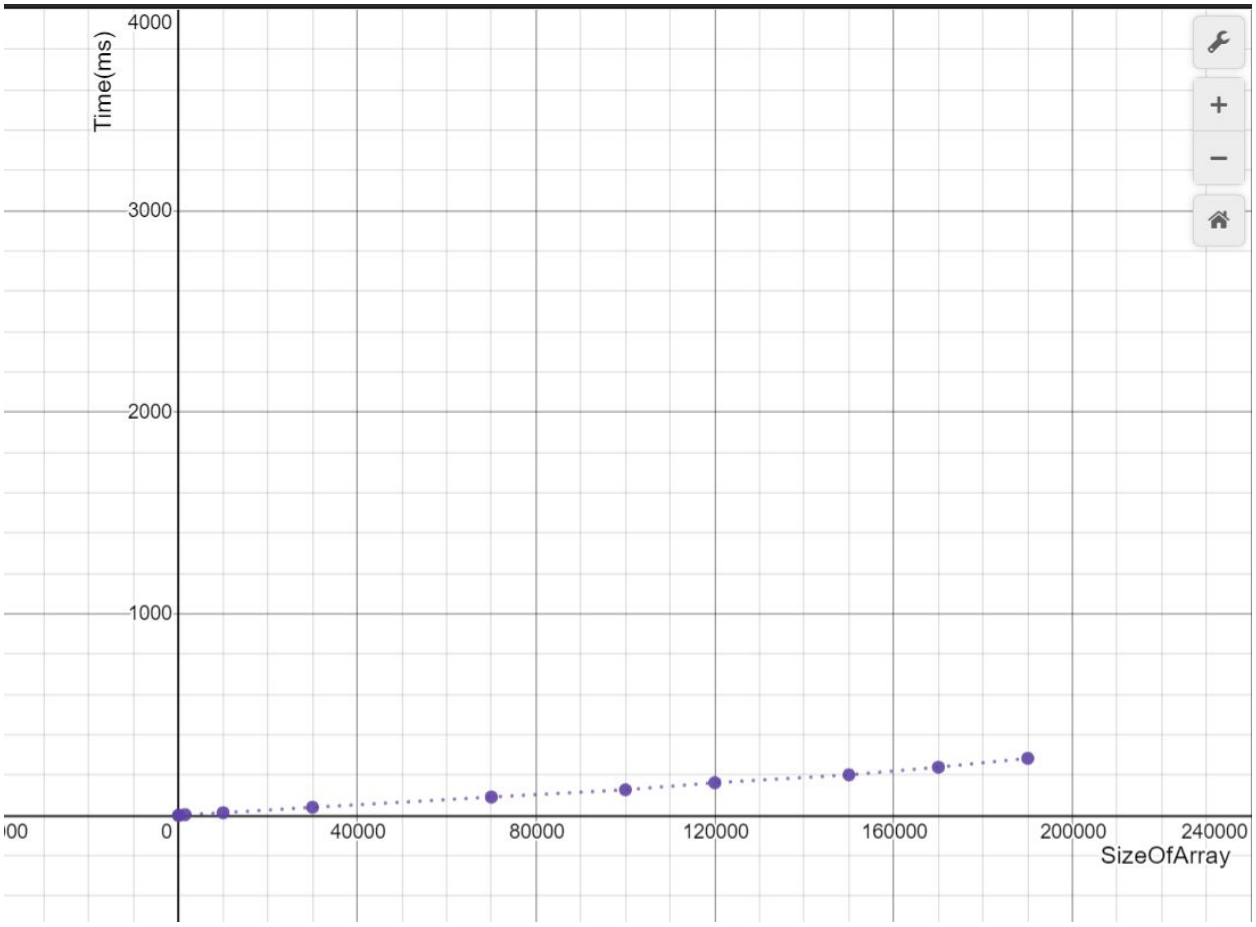
n	t
10	0
100	3
1500	78
10000	665
30000	4899
70000	34221
100000	66000
120000	102593
150300	181000
170000	250000
190000	330354


Time(ms)
400000
300000
200000
100000
000
0
40000
80000
120000
160000
200000
240000
280000
SizeOfArray

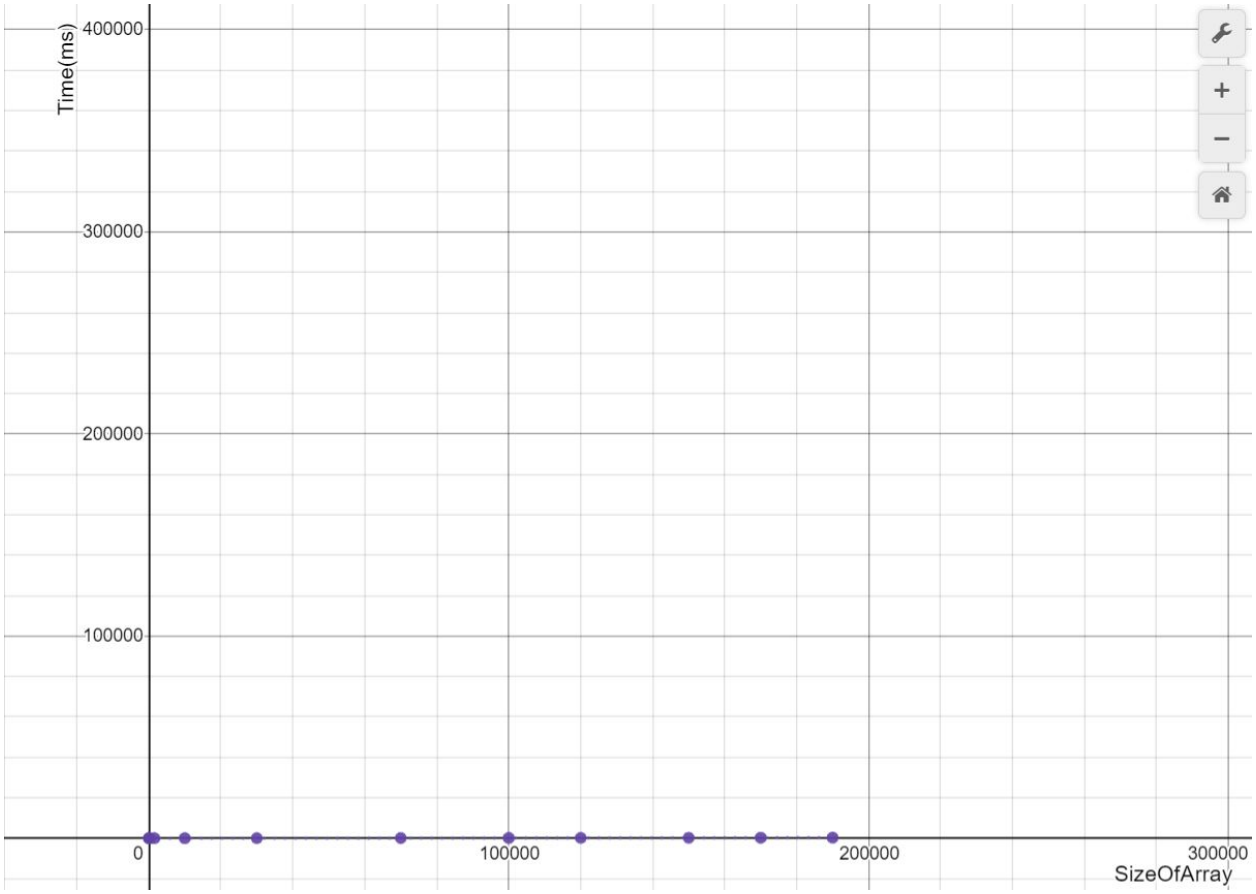
For fast sort O(nlogn):

Table & graph are like following :

n	t
10	0
100	0
1500	3.5
10000	13
30000	40
70000	91
100000	127
120000	162
150000	201
170000	239
190000	283


Time(ms)
4000
3000
2000
1000
0
40000
80000
120000
160000
200000
240000
SizeOfArray

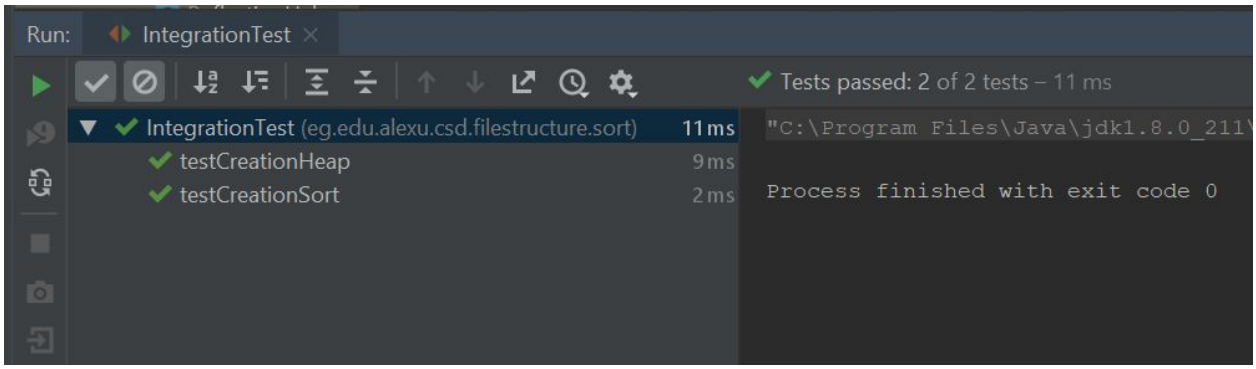
** if we make the scale like the slow one it will be like :



- ❏ We notice that O(nlogn)is better and takes too short time compared to O(n^2)

Tests :

The two Integration tests are successfully passed



The 38 Unit Test are successfully passed