The following materials have been collected from the numerous sources including my own and my students over the years of teaching and experiences of programming. Please help me to keep this tutorial up-to-date by reporting any issues or questions. Please send any comments or criticisms to idebtor@gmail.com. Your assistances and comments will be appreciated.

Find Median from data stream (Leetcode 295)

Table of Contents

Problem Definition

This problem is based on Leetcode 295. Interactively, solve for the median of stream of integers using two heaps, min-heap and max-heap. This implementation uses the **heap-size invariant algorithm**:

Heap-size invariant:

- Keep the size(max-heap) is always N/2.
- Keep the size of min-heap is greater than the size of max-heap by 1 if N is odd. N is the total number of integers in the stream.

```
PS C:\GitHub\nowicx\psets\pset-final> ./medianx
        MaxHeap:
        MinHeap:
         Median: 0

        g - grow                t - trim
        x - grow N              y - trim N
        h - heap-ordered?       c - clear
        m - show mode:[tree]
        Command(q to quit):
```

Algorithm – Add or Grow

We begin with two heaps: a min-heap and a max-heap.

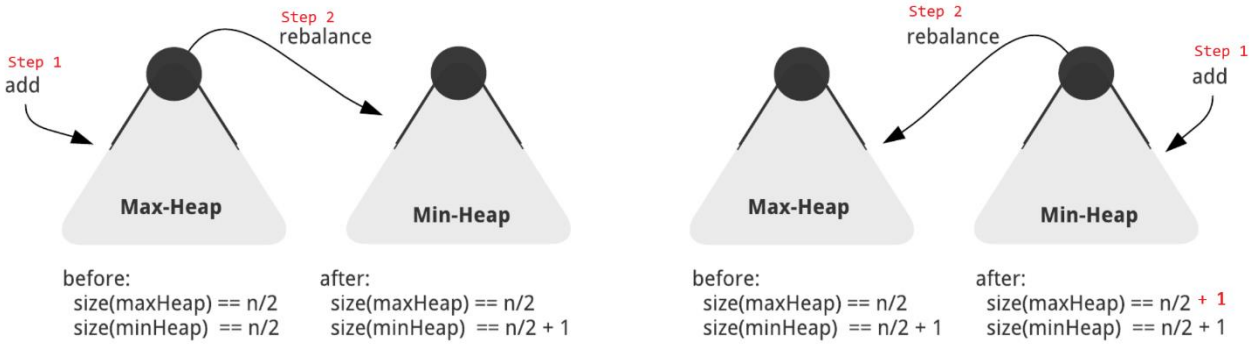
Next, let's introduce a condition: Keep the size of the max-heap must be (n / 2) at all times, while the size of the min-heap can be either (n / 2) or (n / 2) + 1, depending on the total number of elements in the two heaps. In other words, we can allow only the min-heap to have an extra element, when the total number of elements is odd.

With our heap size invariant, we can compute the median as the average of the root elements of both heaps, if the sizes of both heaps are (n / 2). Otherwise, the root element of the min-heap is the median.

When we add a new integer, we have two scenarios:

1. Total no. of existing elements is even
   size(min-heap) == size(max-heap) == (n / 2)
2. Total no. of existing elements is odd
   size(max-heap) == (n / 2)
   size(min-heap) == (n / 2) + 1

We can maintain the invariant by adding the new element to one of the heaps and rebalancing every time.



The **rebalancing** works by moving the largest element from the max-heap to the min-heap, or by moving the smallest element from the min-heap to the max-heap. This way, though we are not comparing the new integer before adding it to a heap, the subsequent rebalancing ensures that **we honor the underlying invariant** of smaller and larger halves.

Sample Run for "grow:1, 3, 4, 5, 7, 9, 8, 6, 4, 2, 0"

```
        Enter a key to grow: 1
        [Tree built using iteration]
1
        MaxHeap:
        MinHeap: root:1 size:1 capa:2 height:0
         Median: 1
```

```
        Enter a key to grow: 3
        [Tree built using iteration]
1
        [Tree built using iteration]
3
        MaxHeap: root:1 size:1 capa:2 height:0
        MinHeap: root:3 size:1 capa:4 height:0
         Median: 2
```

```
        Enter a key to grow: 5
        [Tree built using iteration]
1
        [Tree built using recursion]
  3
 /
5

        MaxHeap: root:1 size:1 capa:4 height:0
        MinHeap: root:3 size:2 capa:4 height:1
         Median: 3
```

```
        Enter a key to grow: 7
        [Tree built using recursion]
  3
 /
1

        [Tree built using recursion]
  5
 /
7

        MaxHeap: root:3 size:2 capa:4 height:1
        MinHeap: root:5 size:2 capa:4 height:1
         Median: 4
```

. . . at the end of "`grow:1, 3, 4, 5, 7, 9, 8, 6, 4, 2, 0`"

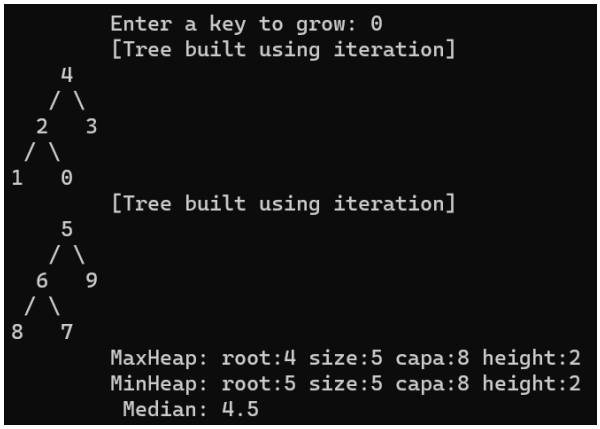
```
        Enter a key to grow: 0
        [Tree built using iteration]
    4
   / \
  2   3
 / \
1   0

        [Tree built using iteration]
    5
   / \
  6   9
 / \
8   7

        MaxHeap: root:4 size:5 capa:8 height:2
        MinHeap: root:5 size:5 capa:8 height:2
         Median: 4.5
```

Algorithm – Delete or Trim

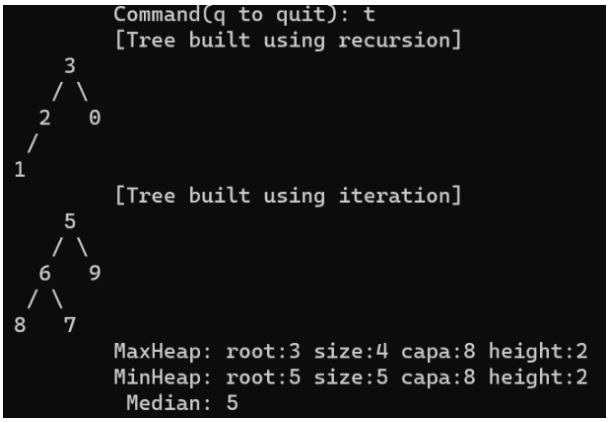
In a heap data structure, the 'delete' or 'trim' operation always removes the root node which is the minimum or maximum value of the heap. In our two-heap data structures, we always keep the maximum and minimum values to compute the median.

To accomplish this purpose and keep the heap-size invariant, we simply trim one from min-heap if the size of the min-heap is greater than that of the max-heap. Trim one from max-heap if two heaps have the same size.

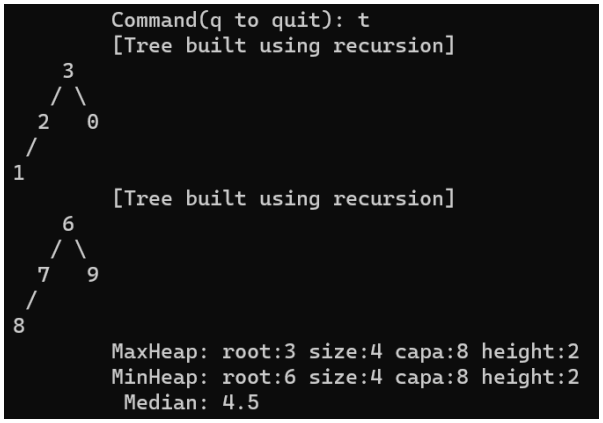
Sample Run for "trim" twice



(Before)



(After the first trim)



(After the second trim)

Time Complexity

The time complexity of getMedian() costs O(1) time, while add runs in time O(log n) with exactly the same number of operations.

Build

Files Provided:

- heap.h – **update from GitHub**
- heap.cpp – **use your own code** from the previous pset13-14
- median.cpp – **skeleton code; For instructions, read the code comments.**
- heapprint.cpp
- treeprint.cpp

Build script example:

- `g++ median.cpp heap.cpp treeprint.cpp heapprint.cpp -I../../include -L../../lib -lnowic -o median`

Test Cases – Filled with your results

- grow operations

Num	Max-heap(level order)	Min-heap(level order)	Median
1		1	1
3	3	1	2
5	3 1	5 7	4
7	3 1	5 7 9	7
9			
8			
6			
4			
2			
0	4 2 3 1 0	5 6 9 8 7	4.5

- trim operations

trim	Max-heap(level order)	Min-heap(level order)	Median
	4 2 3 1 0	5 6 9 8 7	4.5
1st	3 2 0 1	5 6 9 9 8 7	5
2nd	3 2 0 1	6 7 9 8	4.5
3rd			
4th			
5th			
6th			
7th			
8th			
9th		9	9

- Do your own test cases for Grow N and Trim N operations.

(Starting at size = 0; grow N – 5) (Trim N – 2)

```
        Enter a number of nodes to grow: 5
        [Tree built using recursion]
  2
 /
1

        [Tree built using iteration]
  3
 / \
5   4

        MaxHeap: root:2 size:2 capa:4 height:1
        MinHeap: root:3 size:3 capa:4 height:1
         Median: 3
```

```
        Command(q to quit): y

        Enter a number of nodes to trim: 2
        [Tree built using iteration]
1
        [Tree built using recursion]
  4
 /
5

        MaxHeap: root:1 size:1 capa:4 height:0
        MinHeap: root:4 size:2 capa:4 height:1
         Median: 4
```

References

- Leetcode 295: Find median from data stream
- Median of Stream of Integers using Heap in Java

Submitting your solution

- Include the following line at the top of your every file with your name signed.
  On my honour, I pledge that I have neither received nor provided improper assistance in the completion of this assignment. Signed: _____
- Make sure your code **compiles** and **runs** right before you submit it. Don't make "a tiny last-minute change" and assume your code still compiles. You will not receive sympathy for code that "almost" works.
- If you only manage to work out the homework partially before the deadline, you still need to turn it in. However, don't turn it in if it does not compile and run.
- Place your source files in the folder you and I are sharing.
- After submitting, if you realize one of your programs is flawed, you may fix it and submit again as long as it is **before the deadline**. You may submit as often as you like. **Only the last version** you submit before the deadline will be graded.

Files to submit

- Submit the following files at the **final folder. M**ake sure that your code should compile and run with g++ on console.
  - **median.cpp**
  - An image capture (or file) of the Test Cases filled with your results

Due

- **No late work** will be accepted since this is considered as the final exam.