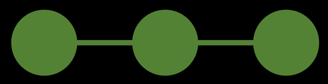
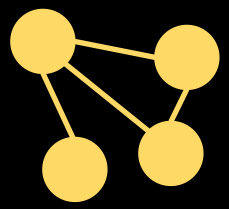
Heaps

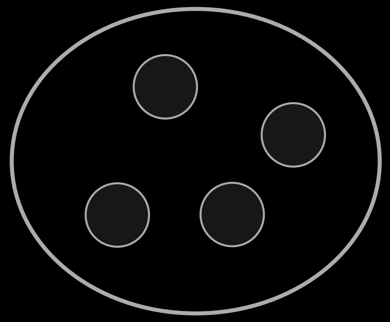
Data Structures & Algorithm

Categories of Data Structures

Linear Ordered	Non-linear Ordered	Not Ordered
Lists	Trees	Sets
Stacks	Graphs	Tables/Maps
Queues		





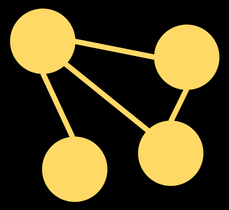


Recap

- Splay Trees
    - Performance
- Red Black Trees
    - Properties
    - Use Cases

Non-linear Ordered

Trees



Agenda

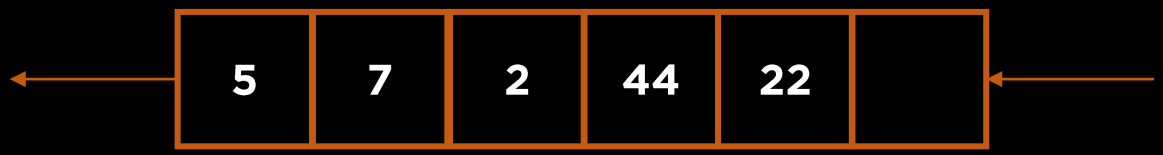
- Priority Queues
  - Motivation
  - Ways of Implementation
- Heaps
  - Properties
  - Implementation
  - Insertion
  - Deletion
  - Heap Sort

Priority Queues

Problem

Queues

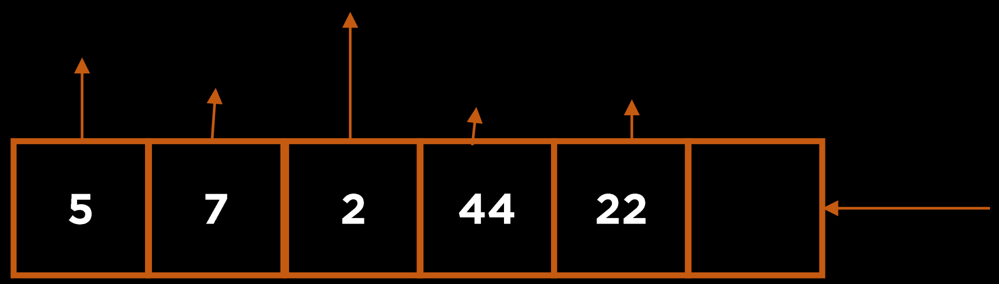
- Queue supported FIFO principle
- Here, “first-in” basis was the priority
- What if we want to generalize this feature of priority?



Problem

Enter Priority Queue!

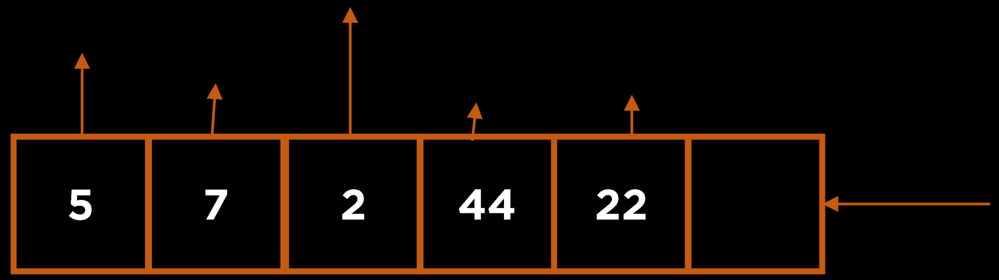
- All elements inserted have some priority
- Elements with highest or lowest priority is removed first



Problem

Priority Queue

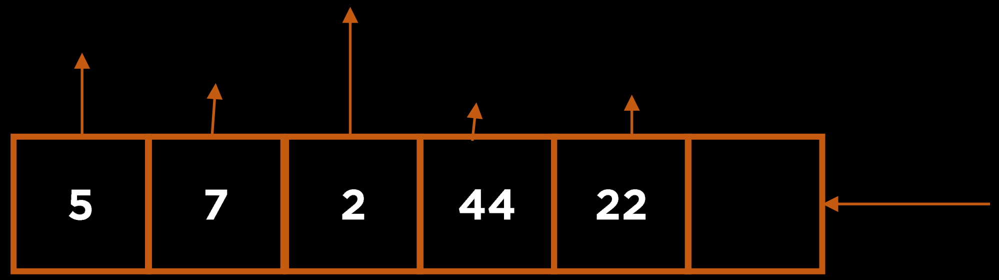
- A priority queue is a generalization of a queue where each element is assigned a priority and elements come out in order by priority



Problem

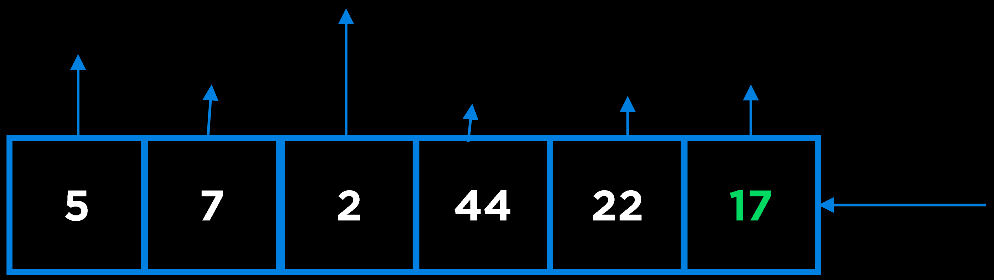
Priority Queue (Central Idea)

- Keep track of highest or lowest priority in a fast way
- Abstract Data Type
  - Insertion (p) – Adds a new element with priority p
  - ExtractMin() or ExtractMax() – Extracts the element with min or max priority

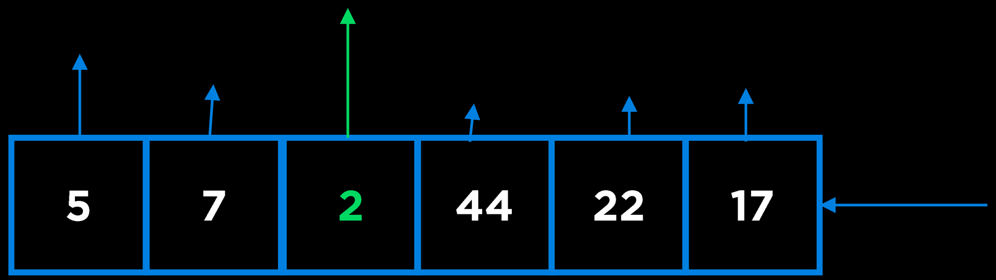


Problem

Insert (17)



ExtractMin()



Priority Queues

How can we design this data structure so that Insert and Extract() operations are fast?

Priority Queues

How can we design this data structure so that Insert and Extract() operations are fast?

Approach 1: Unsorted Array

5	7	2	44		

Insert (p)

ExtractMin()

Priority Queues

How can we design this data structure so that Insert and Extract() operations are fast?

Approach 1: Unsorted Array

5	7	2	44		

Insert (p)

Add p at the end of the array: O(1)

ExtractMin()

Find the min in the array and then shift: O(n)

Priority Queues

How can we design this data structure so that Insert and Extract() operations are fast?

Approach 2: Sorted Array

2	5	7	44		

Insert (p)

ExtractMin()

Priority Queues

How can we design this data structure so that Insert and Extract() operations are fast?

Approach 2: Sorted Array

2	5	7	44		

Insert (p)

Find a position for p in O(log n) using Binary Search, then shift elements: O(n)

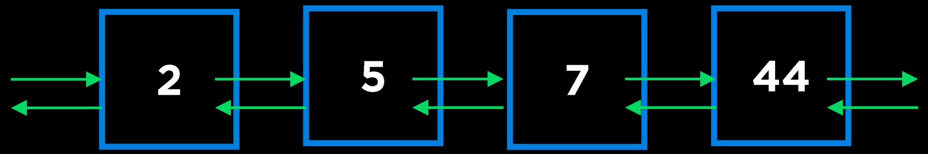
ExtractMin()

Find the min in the array at first place: O(1)

Priority Queues

How can we design this data structure so that Insert and Extract() operations are fast?

Approach 3: Sorted List



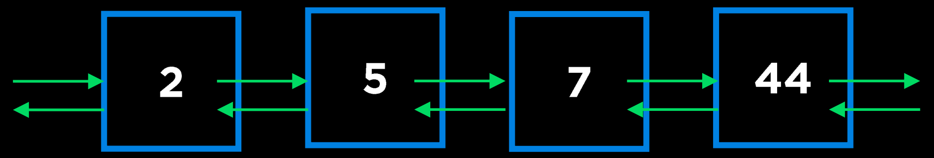
Insert (p)

ExtractMin()

Priority Queues

How can we design this data structure so that Insert and Extract() operations are fast?

Approach 3: Sorted List



Insert (p)

Find a position for p in O(n) using Linear Search, then add in O(1): O(n)

ExtractMin()

Find the min in the list at first place: O(1)

Priority Queues

How can we design this data structure so that Insert and Extract() operations are fast?

	Insert	ExtractMin
Unsorted Array/List	O(1)	O(n)
Sorted Array/List	O(n)	O(1)

Priority Queues

How can we design this data structure so that Insert and Extract() operations are fast?

	Insert	ExtractMin
Unsorted Array/List	O(1)	O(n)
Sorted Array List	O(n)	O(1)
Binary Heap	O(log n)	O(log n)

Problem

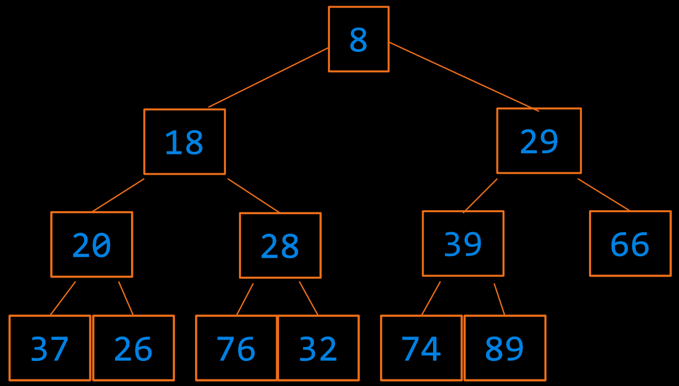
Use Cases

- Huffman Trees
- Dijkstra’s Shortest Path Algorithm
- Prim’s Algorithm for calculating Minimum Spanning Tree
- Scheduling Job
- K largest elements
- Heap Sort
- Many more …

Heaps

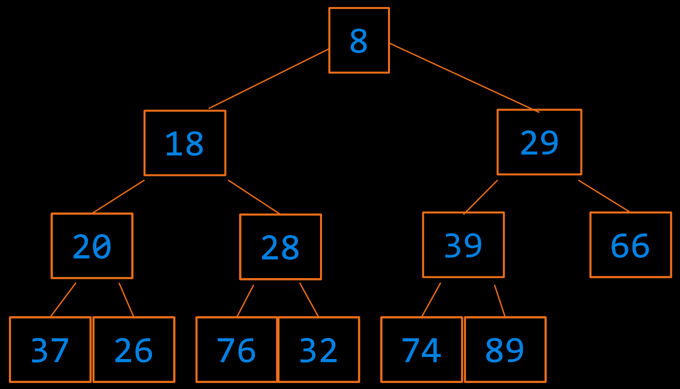
Binary Heap

- Complete Binary Tree
- Each Node is less than its children for a min-heap and Each Node is greater than its children for a max-heap
- Root is the smallest for a min-heap and largest element for a max-heap
- Only the root can be removed (ExtractMin or ExtractMax)



Binary Heap

Heap Representation

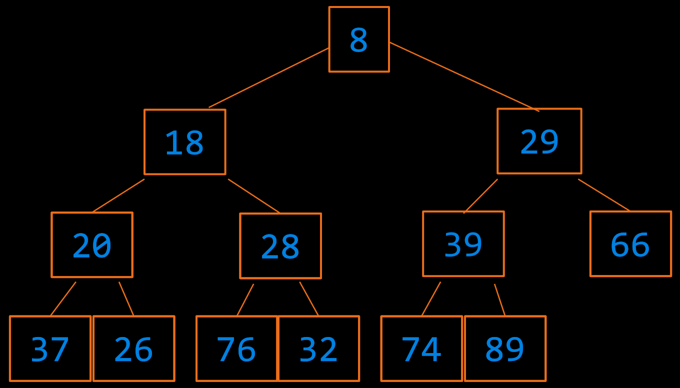



```
class HeapNode
{
    int value;
    HeapNode* left;
    HeapNode* right;
}
```

left and right are min-heaps

Binary Heap

Heap Representation




```
int Heap[];
```

For a node at position p,

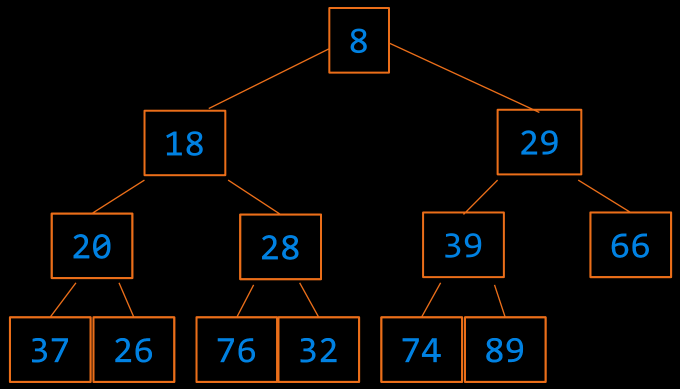
L. child position: $2p + 1$
R. child position: $2p + 2$

A node at position c can find its parent at floor((c – 1)/2)

0	1	2	3	4	5	6	7	8	9	10	11	12	13
8	18	29	20	28	39	66	37	26	76	32	74	89	

Binary Heap Insertion

Heap Insertion

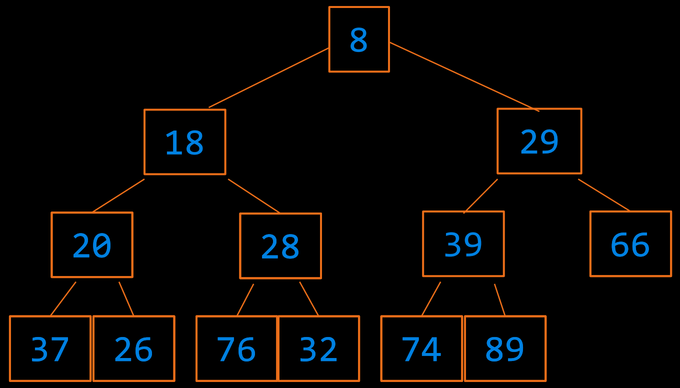


Algorithm for Inserting in a Heap

1. Insert the new item in the next position at the bottom of the heap.
2. `while` new item is not at the root and new item is smaller than its parent
3. Swap the new item with its parent, moving the new item up the heap.

Binary Heap Insertion

Heap Insertion



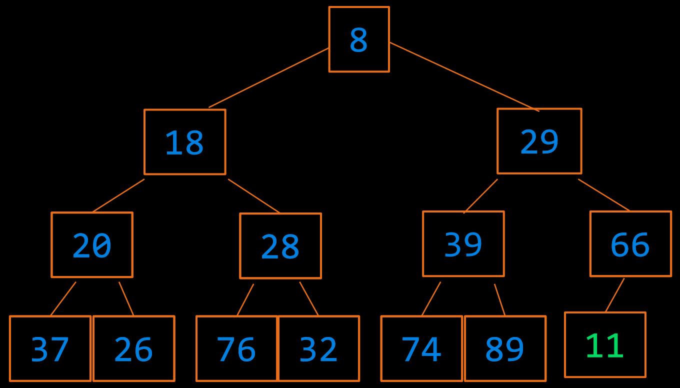

```
1. Insert the new element at the end of the array and set
   child to arr.size() - 1
2. Set parent to (child - 1)/ 2
3. while (parent >= 0 and arr[parent] > arr[child])
      Swap arr[parent] and arr[child]
      Set child equal to parent
      Set parent equal to (child-1)/2
```

insert 11

0	1	2	3	4	5	6	7	8	9	10	11	12	13
8	18	29	20	28	39	66	37	26	76	32	74	89	

Binary Heap Insertion

Heap Insertion



insert 11

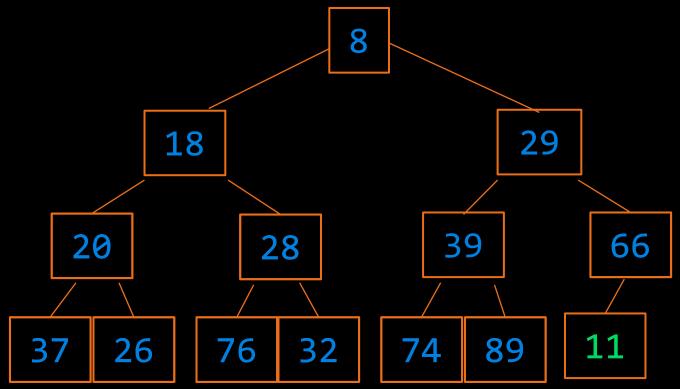
1. Insert the new element at the end of the array and set child to arr.size() - 1
2. Set parent to (child - 1)/ 2
3. while (parent >= 0 and arr[parent] > arr[child])
    Swap arr[parent] and arr[child]
    Set child equal to parent
    Set parent equal to (child-1)/2

child = 13

0	1	2	3	4	5	6	7	8	9	10	11	12	13
8	18	29	20	28	39	66	37	26	76	32	74	89	11

Binary Heap Insertion

Heap Insertion



insert 11

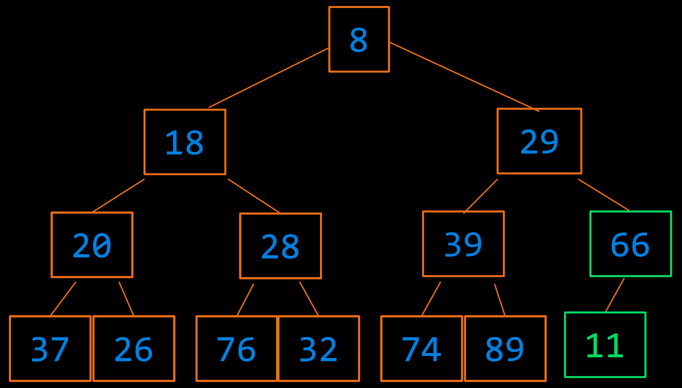
0	1	2	3	4	5	6	7	8	9	10	11	12	13
8	18	29	20	28	39	66	37	26	76	32	74	89	11

```
1. Insert the new element at the end of the array and set
   child to arr.size() - 1
2. Set parent to (child - 1)/ 2
3. while (parent >= 0 and arr[parent] > arr[child])
      Swap arr[parent] and arr[child]
      Set child equal to parent
      Set parent equal to (child-1)/2
```

```
child = 13
parent = 6
```

Binary Heap Insertion

Heap Insertion




```
1. Insert the new element at the end of the array and set
   child to arr.size() - 1
2. Set parent to (child - 1)/ 2
3. while (parent >= 0 and arr[parent] > arr[child])
      Swap arr[parent] and arr[child]
      Set child equal to parent
      Set parent equal to (child-1)/2
```

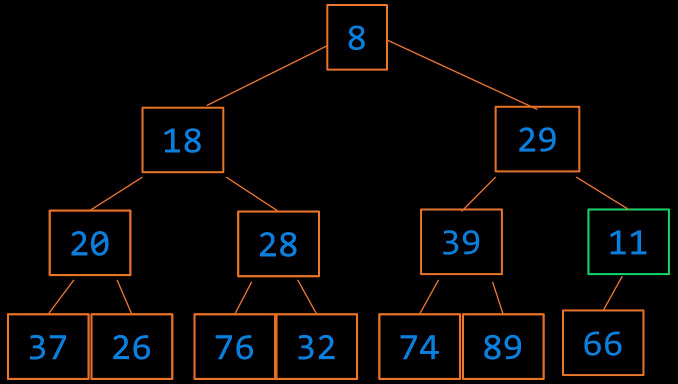
child = 13
parent = 6

insert 11

0	1	2	3	4	5	6	7	8	9	10	11	12	13
8	18	29	20	28	39	66	37	26	76	32	74	89	11

Binary Heap Insertion

Heap Insertion



1. Insert the new element at the end of the array and set child to arr.size() – 1
2. Set parent to (child – 1)/ 2
3. while (parent >= 0 and arr[parent] > arr[child])
    Swap arr[parent] and arr[child]
    Set child equal to parent
    Set parent equal to (child-1)/2

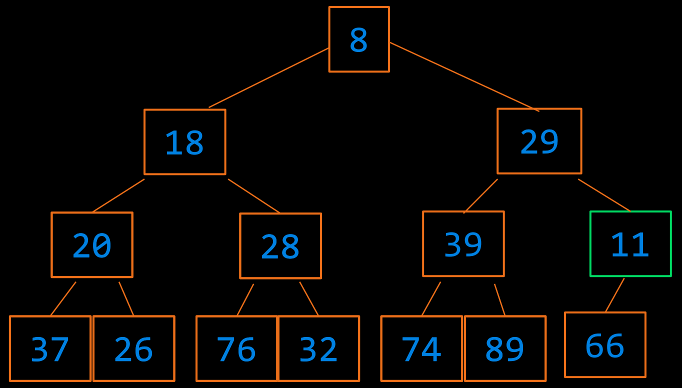
child = 13
parent = 6

insert 11

0	1	2	3	4	5	6	7	8	9	10	11	12	13
8	18	29	20	28	39	11	37	26	76	32	74	89	66

Binary Heap Insertion

Heap Insertion



insert 11

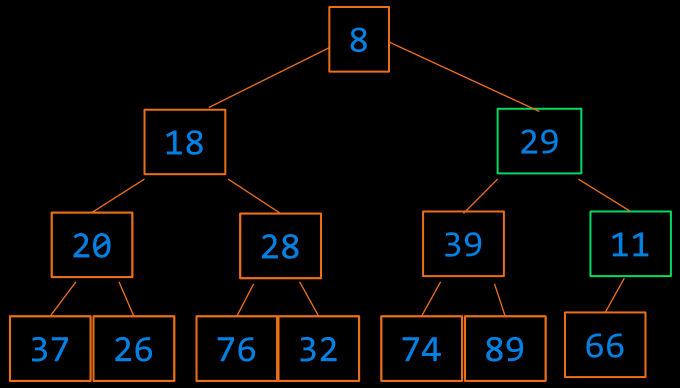
```
1. Insert the new element at the end of the array and set
   child to arr.size() - 1
2. Set parent to (child - 1)/ 2
3. while (parent >= 0 and arr[parent] > arr[child])
      Swap arr[parent] and arr[child]
      Set child equal to parent
      Set parent equal to (child-1)/2
```

```
child = 13 | 6
parent = 6 | 2
```

0	1	2	3	4	5	6	7	8	9	10	11	12	13
8	18	29	20	28	39	11	37	26	76	32	74	89	66

Binary Heap Insertion

Heap Insertion



insert 11

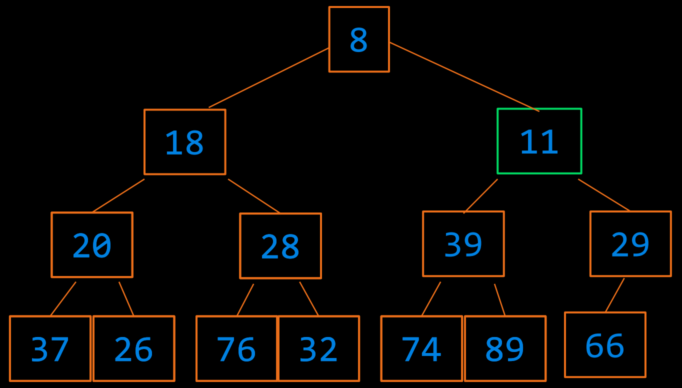
0	1	2	3	4	5	6	7	8	9	10	11	12	13
8	18	29	20	28	39	11	37	26	76	32	74	89	66

```
1. Insert the new element at the end of the array and set
   child to arr.size() - 1
2. Set parent to (child - 1)/ 2
3. while (parent >= 0 and arr[parent] > arr[child])
      Swap arr[parent] and arr[child]
      Set child equal to parent
      Set parent equal to (child-1)/2
```

```
child = 13 | 6
parent = 6 | 2
```

Binary Heap Insertion

Heap Insertion



1. Insert the new element at the end of the array and set child to arr.size() - 1
2. Set parent to (child - 1)/ 2
3. while (parent >= 0 and arr[parent] > arr[child])
   Swap arr[parent] and arr[child]
   Set child equal to parent
   Set parent equal to (child-1)/2

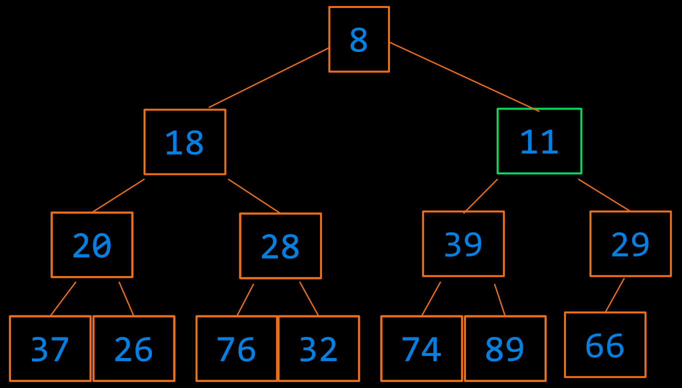
child = 13 | 6
parent = 6 | 2

insert 11

0	1	2	3	4	5	6	7	8	9	10	11	12	13
8	18	11	20	28	39	29	37	26	76	32	74	89	66

Binary Heap Insertion

Heap Insertion



1. Insert the new element at the end of the array and set child to arr.size() - 1
2. Set parent to (child - 1)/ 2
3. while (parent >= 0 and arr[parent] > arr[child])
    Swap arr[parent] and arr[child]
    Set child equal to parent
    Set parent equal to (child-1)/2

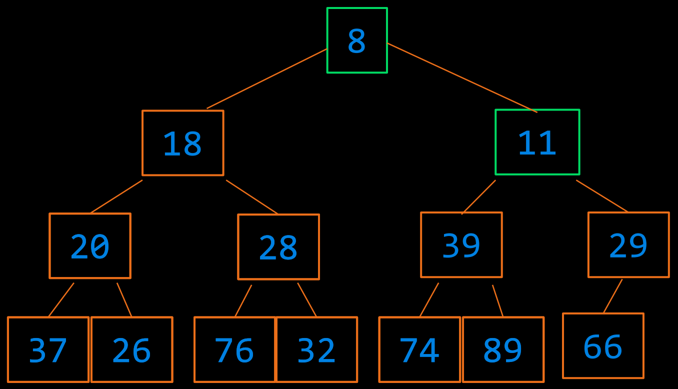
child = 13 | 6 | 2
parent = 6 | 2 | 0

insert 11

0	1	2	3	4	5	6	7	8	9	10	11	12	13
8	18	11	20	28	39	29	37	26	76	32	74	89	66

Binary Heap Insertion

Heap Insertion




```
1. Insert the new element at the end of the array and set
   child to arr.size() - 1
2. Set parent to (child - 1)/ 2
3. while (parent >= 0 and arr[parent] > arr[child])
      Swap arr[parent] and arr[child]
      Set child equal to parent
      Set parent equal to (child-1)/2
```

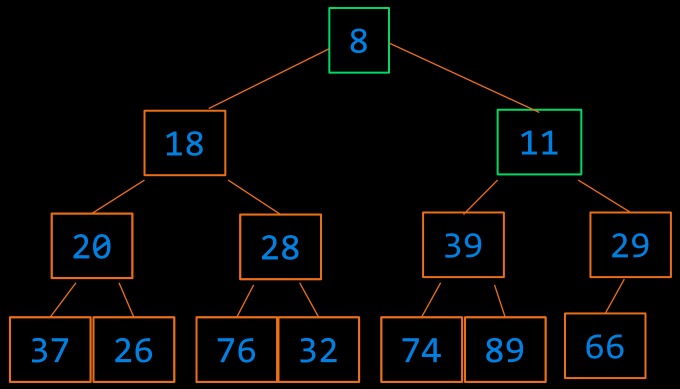
```
child = 13 | 6 | 2
parent = 6 | 2 | 0
```

insert 11

0	1	2	3	4	5	6	7	8	9	10	11	12	13
8	18	11	20	28	39	29	37	26	76	32	74	89	66

Binary Heap Insertion

Heap Insertion



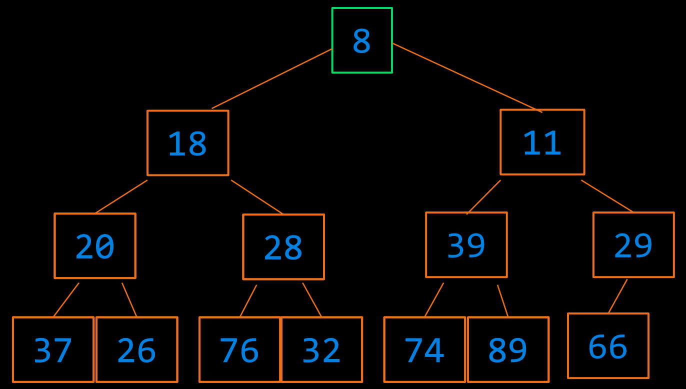

```
1. Insert the new element at the end of the array and set
   child to arr.size() - 1
2. Set parent to (child - 1)/ 2
3. while (parent >= 0 and arr[parent] > arr[child])
      Swap arr[parent] and arr[child]
      Set child equal to parent
      Set parent equal to (child-1)/2
```

```
child = 13 | 6 | 2
parent = 6 | 2 | 0
```

O(log n) time to insert!

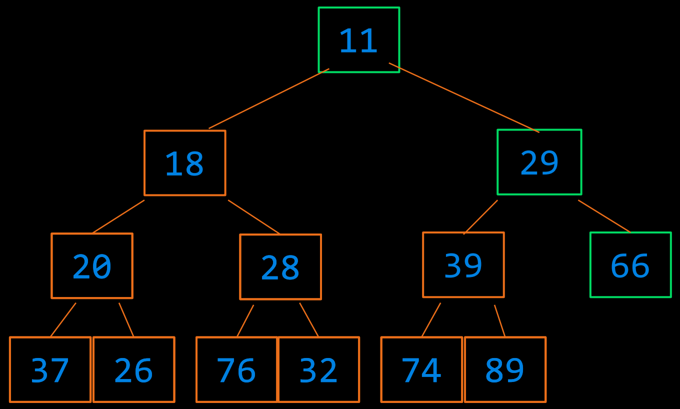
Binary MinHeap Deletion

Heap Deletion (ExtractMin)



Binary MinHeap Deletion

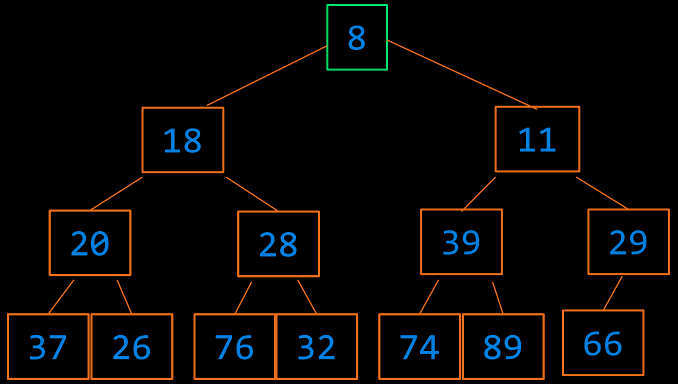
Heap Deletion (ExtractMin)



O(log n) time to ExtractMin!

Binary MinHeap Deletion

Heap Deletion (ExtractMin)



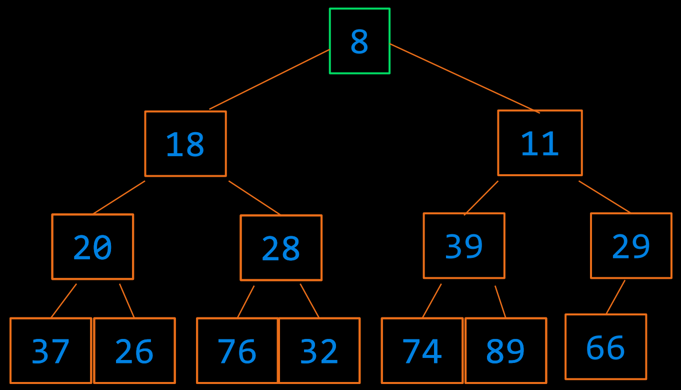
Algorithm for Removal from a Heap

1. Remove the item in the root node by replacing it with the last item in the heap (LIH).
2. `while` item LIH has children and item LIH is larger than either of its children
3.   Swap item LIH with its smaller child, moving LIH down the heap.

Binary MinHeap Deletion

Heap Deletion (ExtractMin)

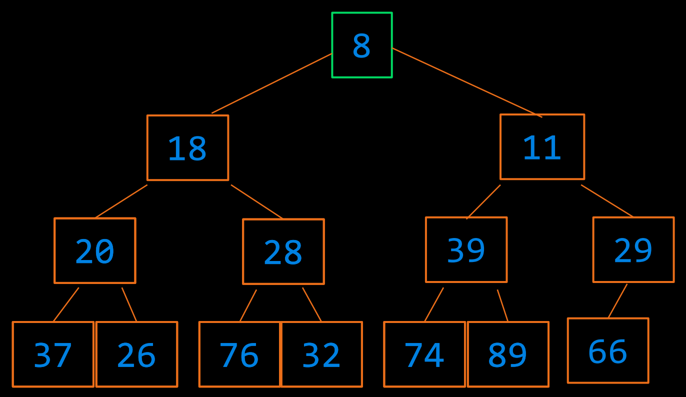
```
//arr[] contains heap
//currentSize contains number of items in heap

//Remove the minimum item.
void extractMin( )
{
      arr[0] = arr[--currentSize];
      heapifyDown(0);
}

void heapifyDown(int index)
{
    1. if index is a leaf -> stop
    2. Find the smallest child of node at index
    3. Swap node at index with smallest_child_index
    4. heapifyDown(smallest_child_index)
}
```



Heap Sort

- Algorithm:
  - Insert n items into heap
  - Remove n items from heap and place in array

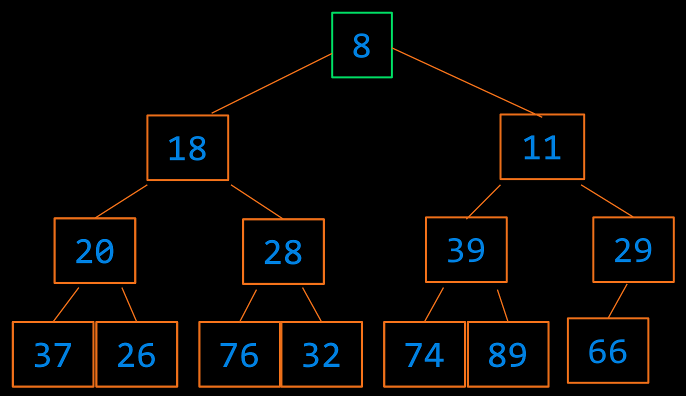
- Performance: O (n log n)



Heap Sort

- Algorithm:
  - Insert n items into heap – O(nlogn) + extra space
  - Remove n items from heap and place in array – O(nlogn)

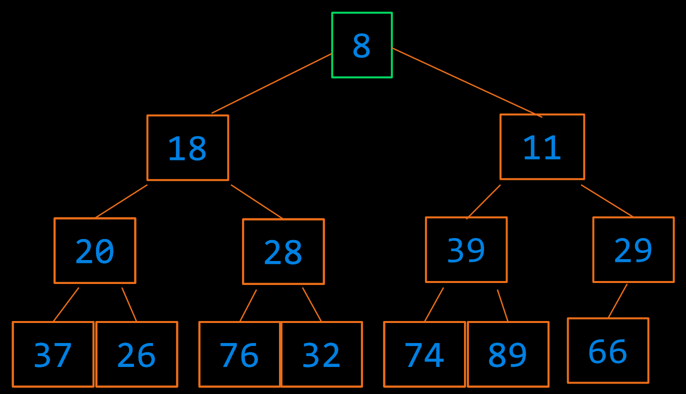
- Performance: O (n log n)



Heap Building

- Building heap inplace:

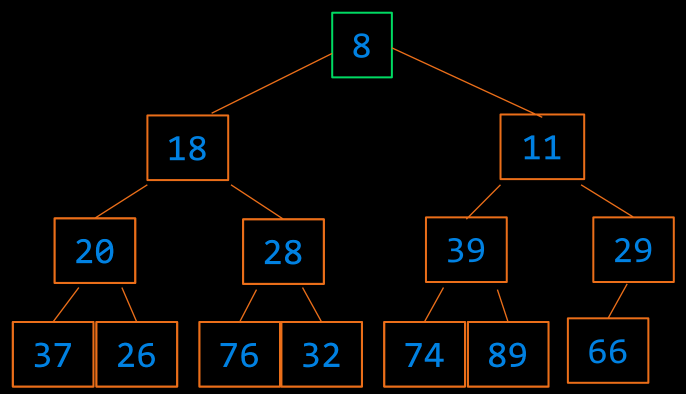
```
for(i = size/2; i >= 0; i--)
        heapifyDown(i)
```



Heap Sort

- Algorithm:
  - Insert n items into heap – O(nlogn) + extra space
  - Remove n items from heap and place in array – O(nlogn)

- Performance: O (n log n)



- Building heap inplace in O(n):

```
for(i = size/2; i >= 0; i--)
    heapifyDown(i)
```

- Since node is close to leaf, heapifyDown is faster

- 1 unit of time for second last level (n/2 nodes), log n for level 0 (1 node)

- T(BuildHeap) = n/2.0 + n/4.1 + n/8.2 …
  = n. SumofSeries(i/2^(i+1)) = 2n

Resources

- Heap Visualization: https://www.cs.usfca.edu/~galles/visualization/Heap.html
- Proof: https://stackoverflow.com/questions/9755721/how-can-building-a-heap-be-on-time-complexity

Mentimeter

Menti.com

6266 4037

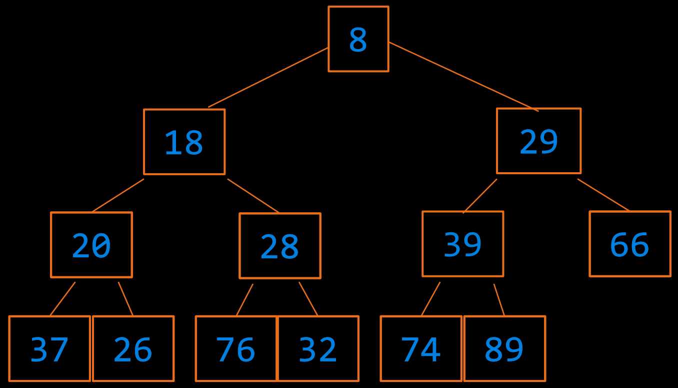
K Largest Elements

Find the largest K items in a stream of N items

- Billions of Transactions in Stock Market
- Weather points of data
- Fraud Detection in Credit Cards

Our interest: Some Largest values/Smallest Values

Constraint:



K Largest Elements – Idea 0

Find the K largest items in an Unsorted List : Sort the array and print arr[n-k … n]

https://stepik.org/lesson/390629/step/2?unit=379729

K Largest Elements – Idea 0

Find the K largest items in an Unsorted List : Sort the array and print arr[n-k … n]

Complexity: O(N log N)

https://stepik.org/lesson/390629/step/2?unit=379729

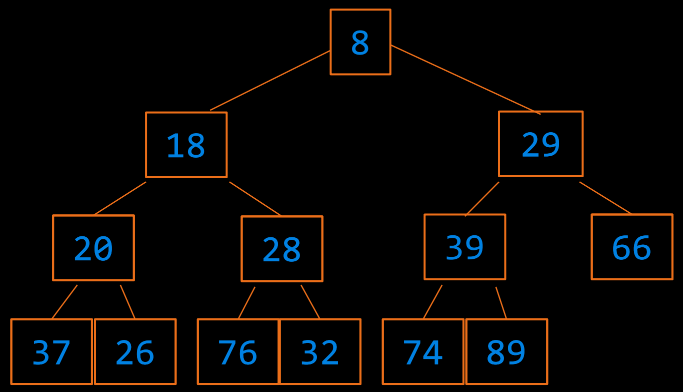
K Largest Elements

Find the largest K items in a stream of N items

- Billions of Transactions in Stock Market
- Weather points of data
- Fraud Detection in Credit Cards

Our interest: Some Largest values/Smallest Values

Constraint: Can we do better than the Sort technique?



K^{th} Largest Elements – Idea 1

Find the K^{th} largest item in an Unsorted List (Max Heap)

K^th Largest Elements – Idea 1

Find the K^th largest item in an Unsorted List (Max Heap)

```
int kthlargest(vector<int>& nums, int k)
{
    //build a max heap
    priority_queue<int> pq(nums.begin(), nums.end());
    //Remove top k-1 elements
    for (int i = k - 1; i > 0; i--)
        pq.pop();
    return pq.top();
}
```

Complexity: , Space:

K^{th} Largest Elements – Idea 1

Find the K^{th} largest item in an Unsorted List (Max Heap)

```
int kthlargest(vector<int>& nums, int k)
{
    //build a max heap
    priority_queue<int> pq(nums.begin(), nums.end());
    //Remove top k-1 elements
    for (int i = k - 1; i > 0; i--)
        pq.pop();
    return pq.top();
}
```

Complexity: O(N + K log N) using Max Heaps, Space: O(N)

Kth Largest Elements – Idea 1

Find the Kth largest item in an Unsorted List (Max Heap)

```cpp
int kthlargest(vector<int>& nums, int k)
{
    //build a max heap
    priority_queue<int> pq(nums.begin(), nums.end());
    //Remove top k-1 elements
    for (int i = k - 1; i > 0; i--)
        pq.pop();
    return pq.top();
}
```

Complexity: O(N + K log N) using Max Heaps, Space: O(N)

Too much Time and Space!

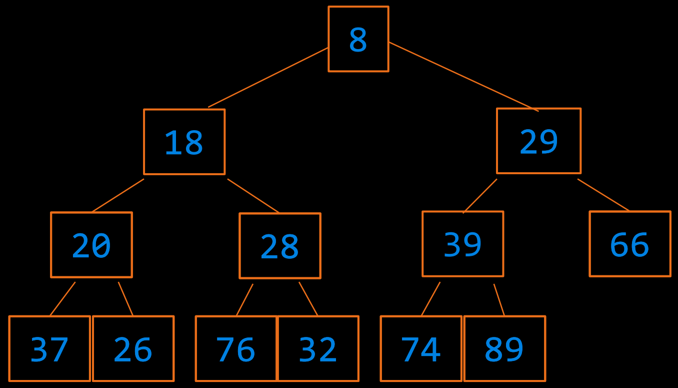
K Largest Elements

Find the largest K items in a stream of N items

- Billions of Transactions in Stock Market
- Weather points of data
- Fraud Detection in Credit Cards

Our interest: Some Largest values/Smallest Values

Constraint: Can’t store N items



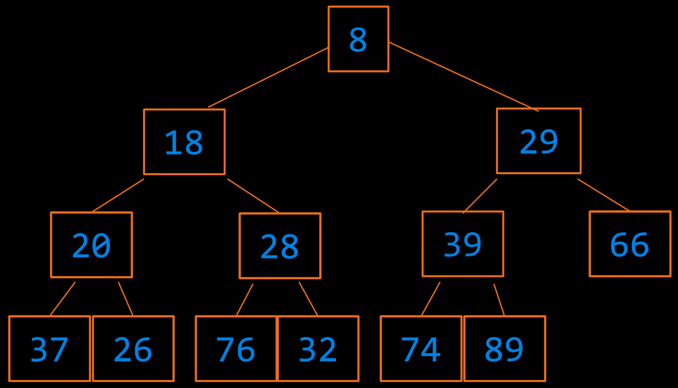
K Largest Elements – Idea 2

Find the largest K items in a stream of N items

- Billions of Transactions in Stock Market
- Weather points of data
- Fraud Detection in Credit Cards

Our interest: Some Largest values/Smallest Values

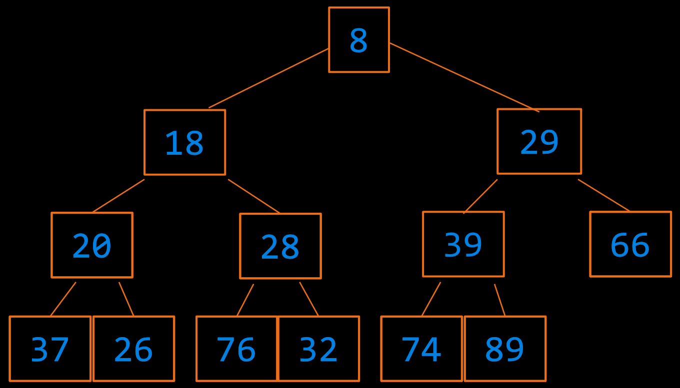
Constraint: Can’t store N items



Idea: Use a Min Priority Queue

1. Push items into a Minimum Priority Queue
2. Delete an element when the queue’s size is greater than K

K Largest Elements – Idea 2



Idea: Use a Min Priority Queue

1. Push items into a Minimum Priority Queue
2. Delete an element when the queue’s size is greater than K

K^{th} Largest Elements – Idea 2

Find the K^{th} largest item in an Unsorted List (Min Heap)

```
int kthlargest(vector<int>& nums, int k)
{
    //min heap
    priority_queue<int, vector<int>, greater<int>> pq;
    for (int i : nums)
    {
        if(pq.size() > k && i < pq.top())
           continue;
        pq.push(i);
        if (pq.size() > k)
            pq.pop();
    }
    return pq.top();
}
```

Complexity: , Space:

K^th Largest Elements – Idea 2

Find the K^th largest item in an Unsorted List (Min Heap)

```
int kthlargest(vector<int>& nums, int k)
{
    //min heap
    priority_queue<int, vector<int>, greater<int>> pq;
    for (int i : nums)
    {
        if(pq.size() > k && i < pq.top())
           continue;
        pq.push(i);
        if (pq.size() > k)
            pq.pop();
    }
    return pq.top();
}
```

Complexity: O(N log K) using Min Heaps, Space: O(K)

Running Median Problem

Find the Median of Running Integers

```
Stream: 5, 11, 22, 0, 2, 54, 8, 9
```

https://stepik.org/lesson/390629/step/4?unit=379729

Running Median Problem

Find the Median of Running Integers

Idea: Use a Max Heap to keep a track of lower numbers and a Min Heap to keep track of higher
Constraint: Min and Max Heap Properties + Size of Min Heap and Max Heap differ by at most 1

Adding an Element

Rebalancing

Returning Median

Stream: 5, 11, 22, 0, 2, 54, 8, 9

https://stepik.org/lesson/390629/step/4?unit=379729

Running Median Problem

Find the Median of Running Integers

Idea: Use a Max Heap to keep a track of lower numbers and a Min Heap to keep track of higher
Constraint: Min and Max Heap Properties + Size of Min Heap and Max Heap differ by at most 1

Adding an Element

Rebalancing

Returning Median

Max Heap: Lowers

Min Heap: Highers

Stream: 5, 11, 22, 0, 2, 54, 8, 9

Running Median Problem

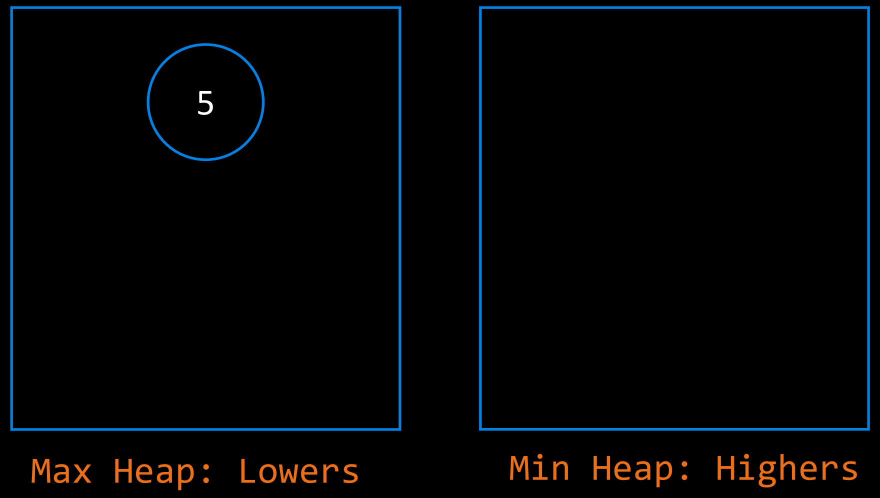
Find the Median of Running Integers

Idea: Use a Max Heap to keep a track of lower numbers and a Min Heap to keep track of higher
Constraint: Min and Max Heap Properties + Size of Min Heap and Max Heap differ by at most 1

Adding an Element

Rebalancing

Returning Median



If both the heaps are empty add, 5 to lowers

Stream: 5, 11, 22, 0, 2, 54, 8, 9

Running Median Problem

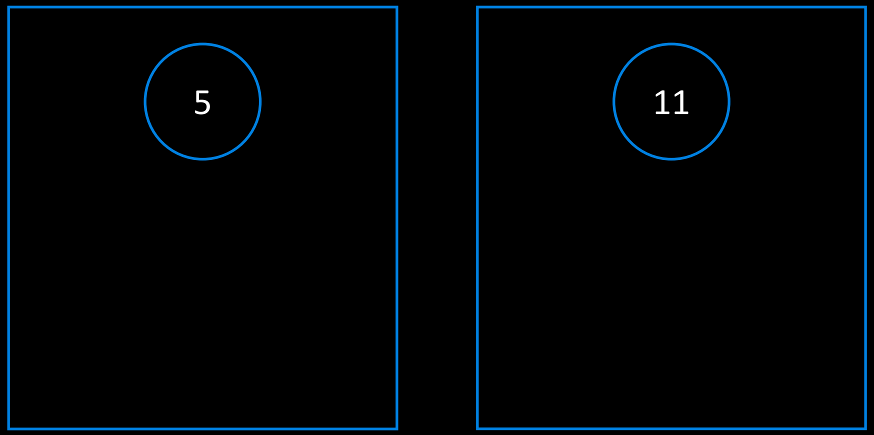
Find the Median of Running Integers

Idea: Use a Max Heap to keep a track of lower numbers and a Min Heap to keep track of higher
Constraint: Min and Max Heap Properties + Size of Min Heap and Max Heap differ by at most 1

Adding an Element

Rebalancing

Returning Median



Max Heap: Lowers

Min Heap: Highers

11 is compared with root of lowers. If it is less than the root, add to lowers. Otherwise add to highers

Stream: 5, 11, 22, 0, 2, 54, 8, 9

Running Median Problem

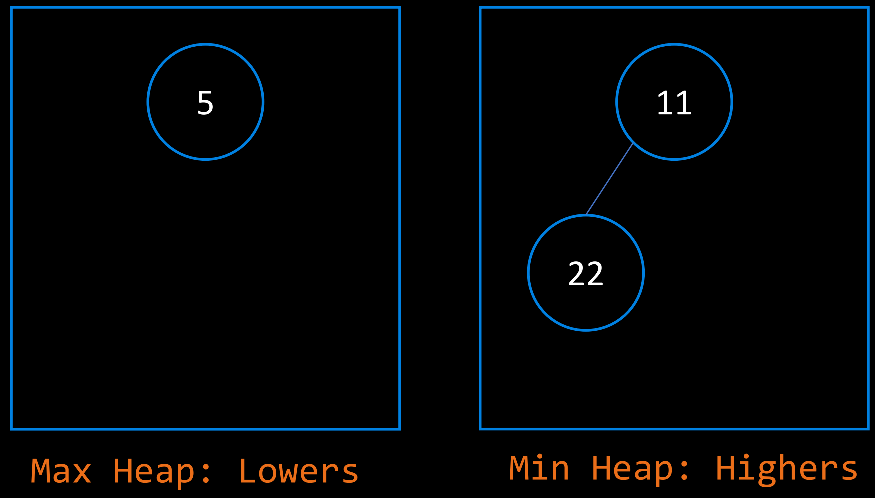
Find the Median of Running Integers

Idea: Use a Max Heap to keep a track of lower numbers and a Min Heap to keep track of higher
Constraint: Min and Max Heap Properties + Size of Min Heap and Max Heap differ by at most 1

Adding an Element

Rebalancing

Returning Median



22 is compared with root of lowers. If it is less than the root, add to lowers. Otherwise add to highers

Stream: 5, 11, 22, 0, 2, 54, 8, 9

Running Median Problem

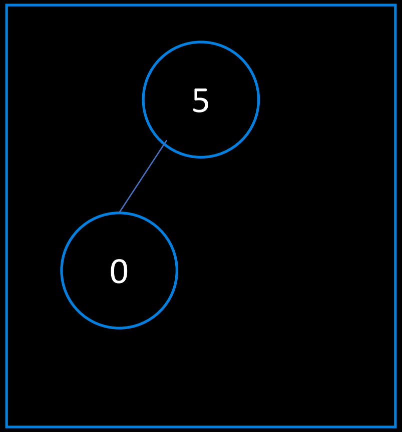
Find the Median of Running Integers

Idea: Use a Max Heap to keep a track of lower numbers and a Min Heap to keep track of higher
Constraint: Min and Max Heap Properties + Size of Min Heap and Max Heap differ by at most 1

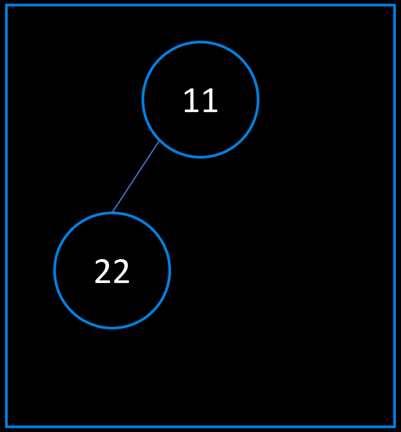
Adding an Element

Rebalancing

Returning Median



Max Heap: Lowers



Min Heap: Highers

0 is compared with root of lowers. If it is less than the root, add to lowers. Otherwise add to highers

Stream: 5, 11, 22, 0, 2, 54, 8, 9

Running Median Problem

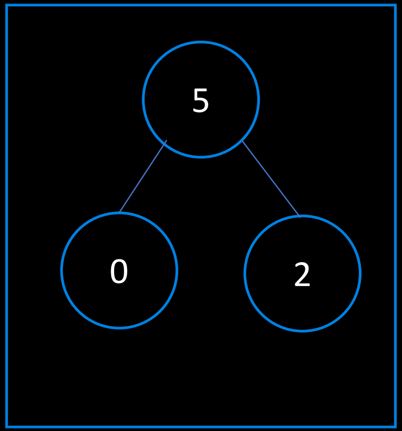
Find the Median of Running Integers

Idea: Use a Max Heap to keep a track of lower numbers and a Min Heap to keep track of higher
Constraint: Min and Max Heap Properties + Size of Min Heap and Max Heap differ by at most 1

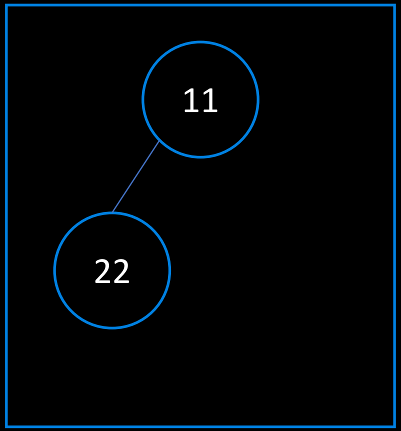
Adding an Element

Rebalancing

Returning Median



Max Heap: Lowers



Min Heap: Highers

2 is compared with root of lowers. If it is less than the root, add to lowers. Otherwise add to highers

Stream: 5, 11, 22, 0, 2, 54, 8, 9

Running Median Problem

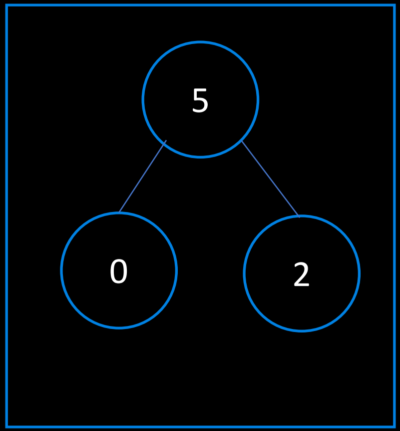
Find the Median of Running Integers

Idea: Use a Max Heap to keep a track of lower numbers and a Min Heap to keep track of higher
Constraint: Min and Max Heap Properties + Size of Min Heap and Max Heap differ by at most 1

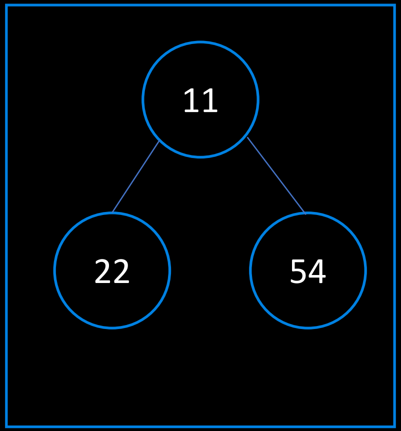
Adding an Element

Rebalancing

Returning Median



Max Heap: Lowers



Min Heap: Highers

54 is compared with root of lowers. If it is less than the root, add to lowers. Otherwise add to highers

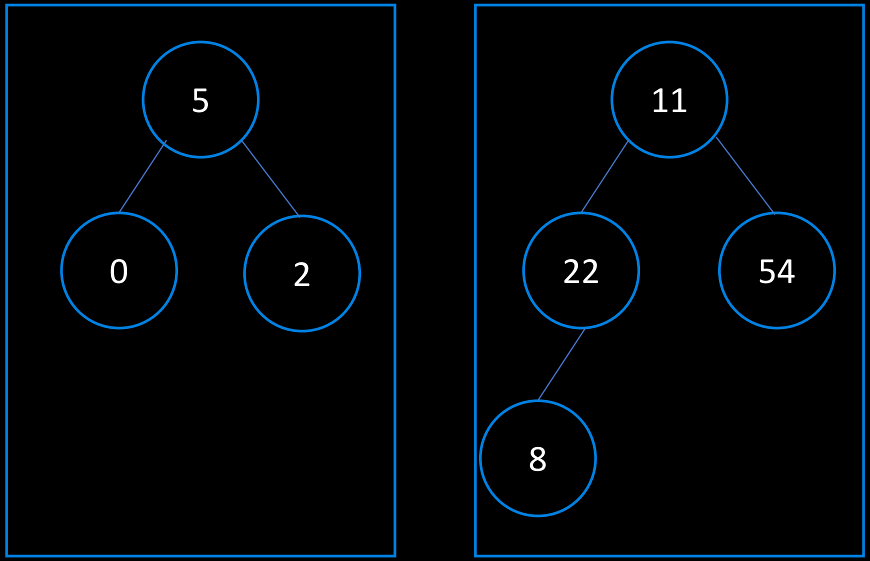
Stream: 5, 11, 22, 0, 2, 54, 8, 9

Running Median Problem

Find the Median of Running Integers

Idea: Use a Max Heap to keep a track of lower numbers and a Min Heap to keep track of higher
Constraint: Min and Max Heap Properties + Size of Min Heap and Max Heap differ by at most 1

Adding an Element | Rebalancing | Returning Median



Max Heap: Lowers

Min Heap: Highers

8 is compared with root of lowers. If it is less than the root, add to lowers. Otherwise add to highers

Stream: 5, 11, 22, 0, 2, 54, 8, 9

Running Median Problem

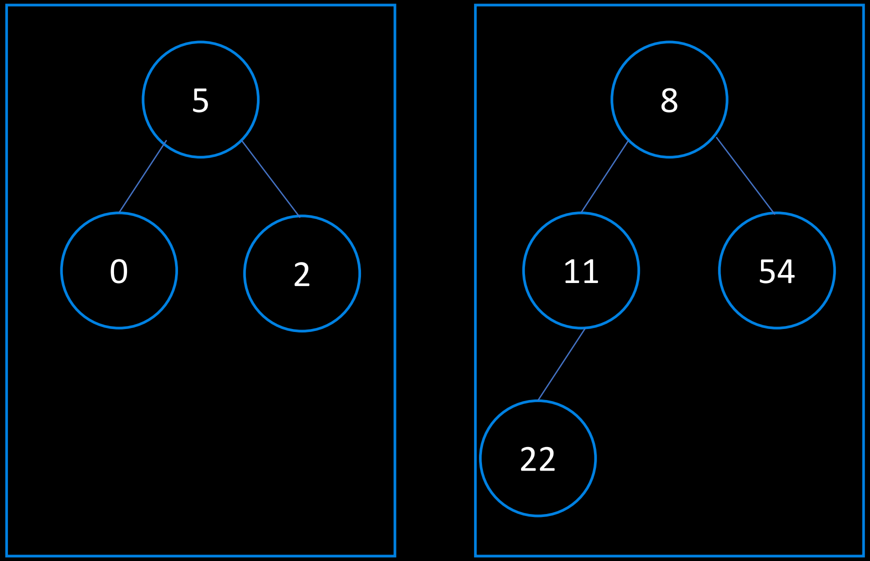
Find the Median of Running Integers

Idea: Use a Max Heap to keep a track of lower numbers and a Min Heap to keep track of higher
Constraint: Min and Max Heap Properties + Size of Min Heap and Max Heap differ by at most 1

Adding an Element

Rebalancing

Returning Median



Max Heap: Lowers

Min Heap: Highers

8 is compared with root of lowers. If it is less than the root, add to lowers. Otherwise add to highers. Min Heap: Highers – HeapifyUp.

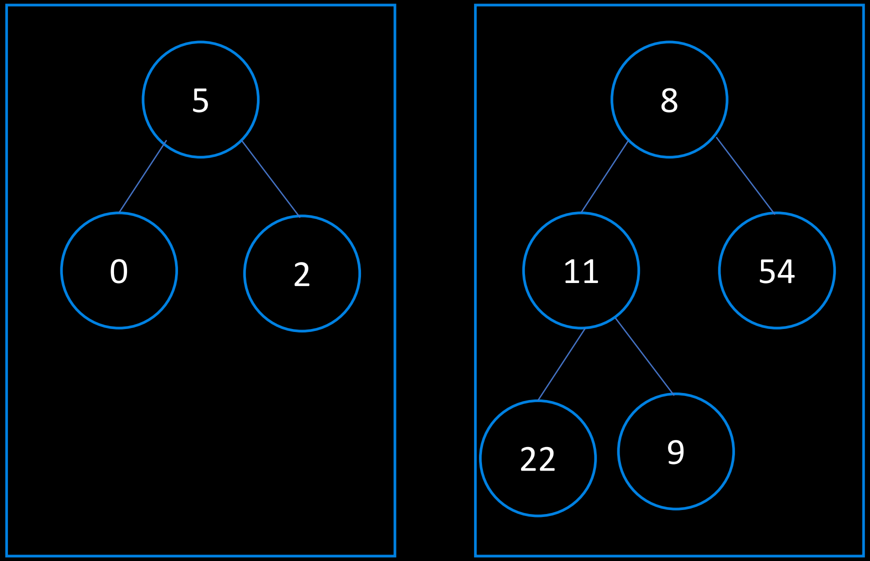
Stream: 5, 11, 22, 0, 2, 54, 8, 9

Running Median Problem

Find the Median of Running Integers

Idea: Use a Max Heap to keep a track of lower numbers and a Min Heap to keep track of higher
Constraint: Min and Max Heap Properties + Size of Min Heap and Max Heap differ by at most 1

Adding an Element | Rebalancing | Returning Median



Max Heap: Lowers   Min Heap: Highers

9 is compared with root of lowers. If it is less than the root, add to lowers. Otherwise add to highers.

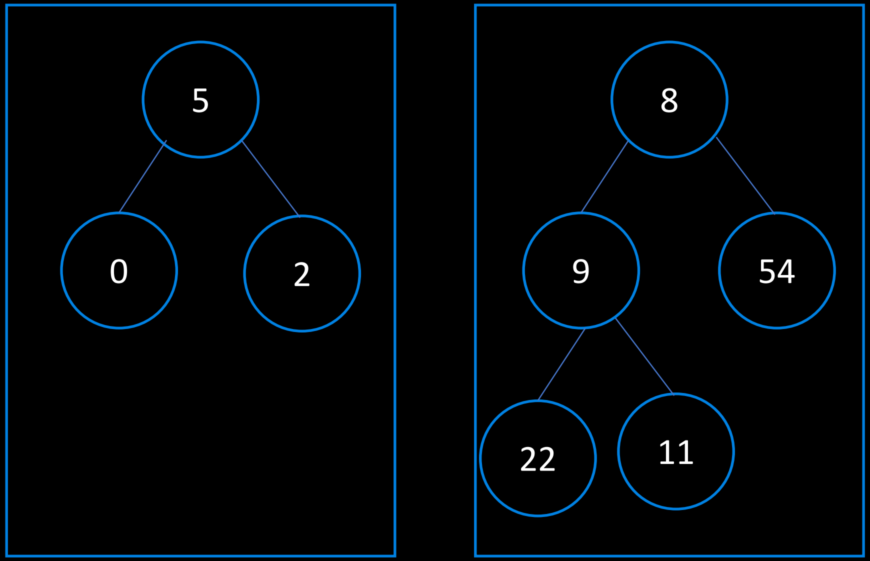
Stream: 5, 11, 22, 0, 2, 54, 8, 9

Running Median Problem

Find the Median of Running Integers

Idea: Use a Max Heap to keep a track of lower numbers and a Min Heap to keep track of higher
Constraint: Min and Max Heap Properties + Size of Min Heap and Max Heap differ by at most 1

Adding an Element   Rebalancing   Returning Median



Max Heap: Lowers   Min Heap: Highers

9 is compared with root of lowers. If it is less than the root, add to lowers. Otherwise add to highers. Min Heap: Highers – HeapifyUp.

Stream: 5, 11, 22, 0, 2, 54, 8, 9

Running Median Problem

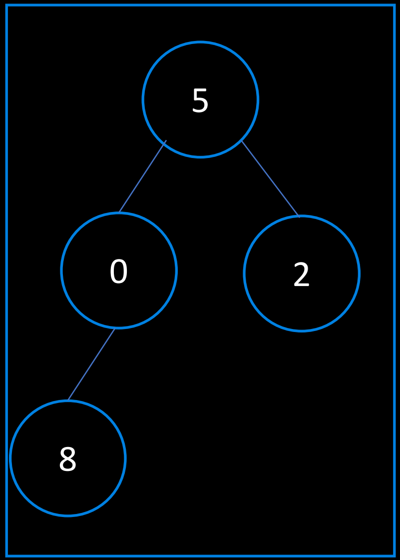
Find the Median of Running Integers

Idea: Use a Max Heap to keep a track of lower numbers and a Min Heap to keep track of higher
Constraint: Min and Max Heap Properties + Size of Min Heap and Max Heap differ by at most 1

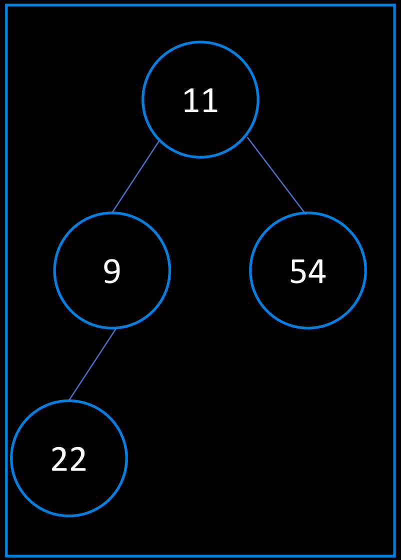
Adding an Element

Rebalancing

Returning Median



Max Heap: Lowers



Min Heap: Highers

9 is compared with root of lowers. If it is less than the root, add to lowers. Otherwise add to highers. Rebalancing. Move root of larger heap to smaller heap.

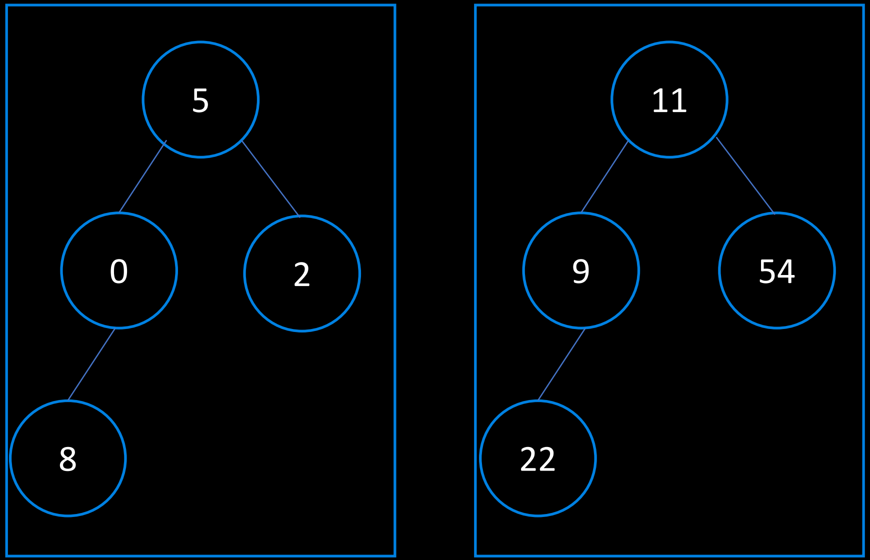
Stream: 5, 11, 22, 0, 2, 54, 8, 9

Running Median Problem

Find the Median of Running Integers

Idea: Use a Max Heap to keep a track of lower numbers and a Min Heap to keep track of higher
Constraint: Min and Max Heap Properties + Size of Min Heap and Max Heap differ by at most 1

Adding an Element   Rebalancing   Returning Median



Max Heap: Lowers   Min Heap: Highers

9 is compared with root of lowers. If it is less than the root, add to lowers. Otherwise add to highers. Heapify up in lowers and Heapify down in higher.

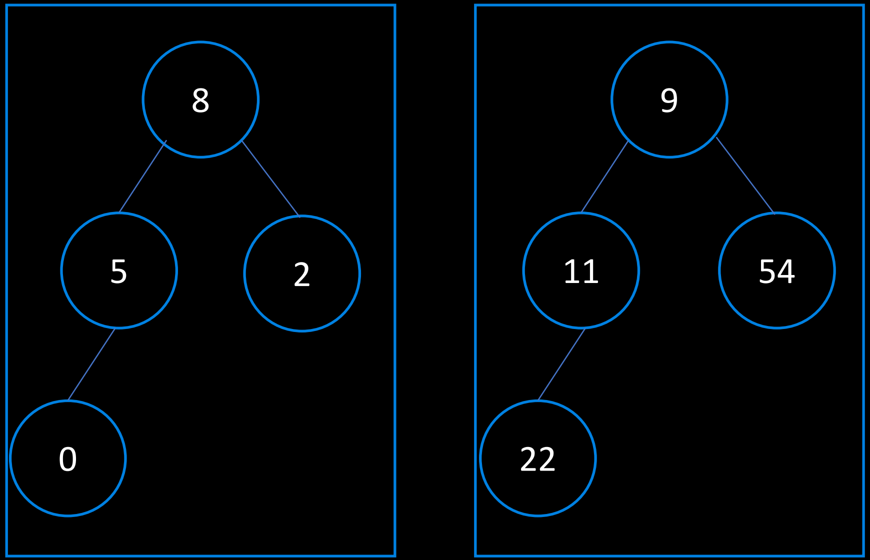
Stream: 5, 11, 22, 0, 2, 54, 8, 9

Running Median Problem

Find the Median of Running Integers

Idea: Use a Max Heap to keep a track of lower numbers and a Min Heap to keep track of higher
Constraint: Min and Max Heap Properties + Size of Min Heap and Max Heap differ by at most 1

Adding an Element   Rebalancing   Returning Median



Max Heap: Lowers   Min Heap: Highers

9 is compared with root of lowers. If it is less than the root, add to lowers. Otherwise add to highers. Heapify up in lowers and Heapify down in higher.

Stream: 5, 11, 22, 0, 2, 54, 8, 9

Running Median Problem

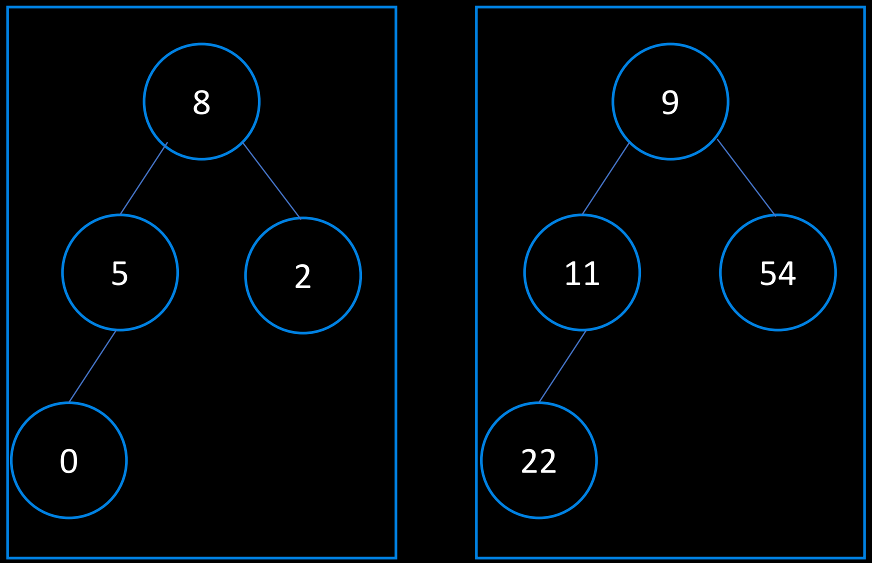
Find the Median of Running Integers

Idea: Use a Max Heap to keep a track of lower numbers and a Min Heap to keep track of higher
Constraint: Min and Max Heap Properties + Size of Min Heap and Max Heap differ by at most 1

Adding an Element

Rebalancing

Returning Median



Max Heap: Lowers

Min Heap: Highers

Median: Average of two roots if heaps are of equal size; Otherwise, the root of larger heap

Median = 8.5

Stream: 5, 11, 22, 0, 2, 54, 8, 9

Running Median Problem

Find the Median of Running Integers

Idea: Use a Max Heap to keep a track of lower numbers and a Min Heap to keep track of highers

```
Max heap, lowers stores elements to the left of median
Min heap, highers stores elements to the right of median

1. Adding an Element, e:
      if lowers.size = 0 or e < lowers.root:
            lowers.add(e)
      else
            highers.add(e)

2. Rebalancing:
      Find biggerHeap and smallerHeap from highers and lowers
      if (biggerHeap.size - smallerHeap.size) = 2:
            smallerHeap.add(biggerHeap.extractMin())

3. Returning Median:
      if size of both heaps are equal:
            return (lowers.max + highers.min)/2
      else
            return the root of bigger heap (lowers.max or higher.min)
```

Resources

- Running Medians Video: https://www.youtube.com/watch?v=VmogG01IjYc

Questions