Heaps

Status	Complete
Domain	
Read	☑
Watch	☑
FlashCards	☑
Lab	☑
Are You Done?	🔥🔥🔥🔥
Author	Gaurav Gupta (gauravgupta369)
Complete	☑
Related to Exam Day (Property)	
Study Date	@October 15, 2022

Binary Heaps

A binary heap is a tree with some special properties.

1. **Its a complete binary tree.**
2. **The value of any node is either ≥ or ≤ to the values of its childrens.**

Types of Binary Heap

Min Heap

The value of a node must be less than or equal to the values of its children.

```
          11
     15        12
 16    17  14    13
```

Max heap

The value of a node must be greater than or equal to the values of its children.

```
          17
     15        10
 12    11  8    7
```

Array Representation of Binary Heaps

```
          11
     15        12
 16    17  14    13
```

```
arr = [11, 15, 12, 16, 17, 14, 13]
```

```
If a node is at index (i) then
Its left child would be at  - 2 * i
Its right child would be at - 2 * i + 1
Its parent would be at      - floor(i / 2)
```

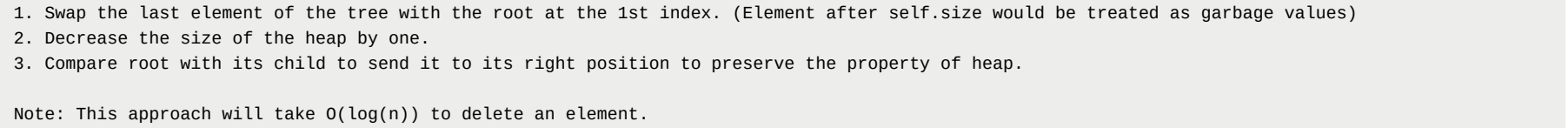
```
Note: If an array is ascending sorted then it will always be a min-heap but reverse is not true.
Note: If an array is descending sorted then it will always be a max-heap but reverse is not true.
Note: In a max-heap largest element would always be the root of the heap, and the second-largest would be either of the children of the root.
Note: In a min-heap smallest element would always be the root of the heap, and the second-smaller would be either of the children of the root.
```

When to use Heaps

If we have a running stream of numbers and we need to find max, second max, min or second min than heap is the best choice.

▼ Insertion in Max Heap / Min Heap

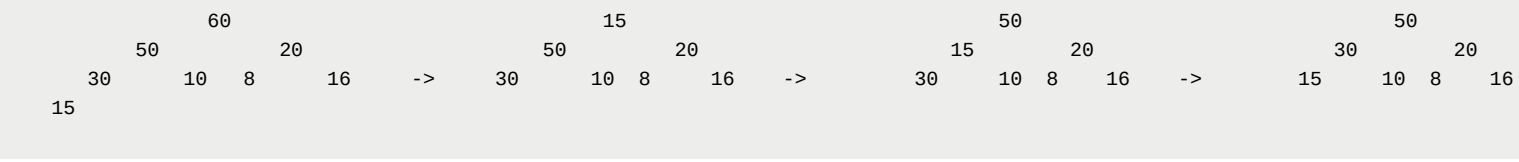
```
To insert an element in a binary heap we first add that element at the end of the heap and
  we compare it with its parent and swap them until it reaches its right position.
Algo:
1. Insert the element at the end of the tree.
2. Increase the size of the tree.
3. Compare the last element with its parent and swap if required and repeat the same process until it reaches its right position.

Note: Best Case Scenario O(1), Worst Case Scenario O(log(n)).
Note: This approach will take O(n * log(n)) to insert n elements in the heap.
```




```
# adding element to a max-heap
class MaxHeap:
    def __init__(self):
        self.heap = [None]
        self.size = 0

    def push(self, ele):
        self.heap.append(ele)
        self.size += 1
        self.perculate_up()

    def perculate_up(self):
        if self.size <= 1: return
        cur = self.size
        parent = floor(cur / 2)
        while parent > 0:
            if self.heap[parent] < self.heap[cur]:
                self.heap[parent], self.heap[cur] = self.heap[cur], self.heap[parent]
            cur = parent
            parent = floor(cur / 2)
```

▼ Deletion in Max Heap / Min Heap

```
To delete an element we can either swap the root element with the last element of the tree or we can put the last element of the tree at the root,
and we can reduce the size of the heap by one, secondly, we can compare the root with its child and swap it until it reaches its right position.
Algo:
1. Swap the last element of the tree with the root at the 1st index. (Element after self.size would be treated as garbage values)
2. Decrease the size of the heap by one.
3. Compare root with its child to send it to its right position to preserve the property of heap.

Note: This approach will take O(log(n)) to delete an element.
```




```
class MaxHeap:
    def __init__(self):
        self.heap = [None]
        self.size = 0

    def pop(self):
        if self.size < 1: return
        self.heap[1], self.heap[self.size] = self.heap[self.size], self.heap[1]
        self.size -= 1
        self.perculate_down()
        return self.heap[self.size + 1]

    def perculate_down(self):
        if self.size < 2: return
        cur = 1
        c1 = cur * 2
        c2 = cur * 2 + 1
        while c1 <= self.size or c2<= self.size:
            if self.heap[c1] > self.heap[cur]:
                self.heap[c1], self.heap[cur] = self.heap[cur], self.heap[c1]
                cur = c1
                c1 = cur * 2
                c2 = cur * 2 + 1
            elif c2 <= self.size and self.heap[c2] > self.heap[cur]:
                self.heap[c2], self.heap[cur] = self.heap[cur], self.heap[c2]
                cur = c2
```

```
                c1 = cur * 2
                c2 = cur * 2 + 1
            else:
                break
```

▼ Heap Sort

If we have a fixed amount of number in a heap then we can take advantage of working of heap, means if we delete any element it would go to the self.seize of heap, and hence we will get the sorted array if we delete all the elements of heap.

```
Algo:
1. Build a max-heap
2. Delete all the nodes of the heap, in this case, we will get an ascending sorted array.

It will take O(n * log(n)) to create the heap and O(n * log(n)) to delete the heap as well with O(n) space.
TC: O(n * log(n)), SC: O(n)
```

```
from math import floor

class MaxHeap:
    def __init__(self):
        self.heap = [None]
        self.size = 0

    def push(self, ele):
        self.size += 1
        if len(self.heap) > self.size:
            self.heap[self.size] = ele
        else:
            self.heap.append(ele)
        self.perculate_up()

    def perculate_up(self):
        if self.size <= 1: return
        cur = self.size
        parent = floor(cur / 2)
        while parent > 0:
            if self.heap[parent] < self.heap[cur]:
                self.heap[parent], self.heap[cur] = self.heap[cur], self.heap[parent]
            cur = parent
            parent = floor(cur / 2)

    def pop(self):
        if self.size < 1: return
        self.heap[1], self.heap[self.size] = self.heap[self.size], self.heap[1]
        self.size -= 1
        self.perculate_down()
        return self.heap[self.size + 1]

    def perculate_down(self):
        if self.size < 2: return
        cur = 1
        c1 = cur * 2
        c2 = cur * 2 + 1
        while c1 <= self.size or c2<= self.size:
            if self.heap[c1] > self.heap[cur]:
                self.heap[c1], self.heap[cur] = self.heap[cur], self.heap[c1]
                cur = c1
                c1 = cur * 2
                c2 = cur * 2 + 1
            elif c2 <= self.size and self.heap[c2] > self.heap[cur]:
                self.heap[c2], self.heap[cur] = self.heap[cur], self.heap[c2]
                cur = c2
                c1 = cur * 2
                c2 = cur * 2 + 1
            else:
                break
```

```
class HeapSort:
    def __init__(self):
        self.max_heap_obj = MaxHeap()

    def sort(self, elements):
        reverse_sorted_elements = []
        for ele in elements:
            self.max_heap_obj.push(ele)

        for _ in range(len(elements)):
            reverse_sorted_elements.append(self.max_heap_obj.pop())

        sorted_elements =  self.max_heap_obj.heap[1:]

        return sorted_elements, reverse_sorted_elements

HeapSort().sort([10, 8, 1, 2, 6, 7, 8, 4, 5, 1, 2, 3, 19])
```

▼ Heapify

The Process of rearranging the heap by comparing each parent with their children recursively is called Heapify.

```
In heapify we fill the array from the last node of the tree to the root and we compare the parent node with
its child nodes using the perculate_down approach.

arr = [30, 10, 50, 20, 60, 8, 16]

step1:
        -                       -                       -
   -        -       ->      -        60    ->        8       60      ->
20   50  10   30         20   50  10   30         20   50  10   30        20   50  10   30




2 ^ h * 0 + 2 ^ (h - 1) * 1 .... 2 ^ 0 * h
2 ^ h( 0 + 1 / 2 + 1 / 2 ^ 2 + ... 1 / 2 ^ h)
2 ^ h(1)
2 ^ log(n) (1)
n
=> O(n)
```

```
               60                          15                          50                          50
         50          20              50          20              15          20              30          20
      30      10   8      16     ->     30      10  8     16    ->        30     10  8    16    ->        15     10  8    16
   15
```

▼ Priority Queue

```
In Priority Queue Every Element has some priority and we insert and delete elements based on their Priority.
Most of the time we choose an element as its Priority.
```

▼ Time Complexity of Other Data Structures

Common Data Structure Operations

Data Structure	Time Complexity								Space Complexity
	Average				Worst				Worst
	Access	Search	Insertion	Deletion	Access	Search	Insertion	Deletion	
Array	Θ(1)	Θ(n)	Θ(n)	Θ(n)	O(1)	O(n)	O(n)	O(n)	O(n)
Stack	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Queue	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Singly-Linked List	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Doubly-Linked List	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Skip List	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(n)	O(n)	O(n)	O(n)	O(n log(n))
Hash Table	N/A	Θ(1)	Θ(1)	Θ(1)	N/A	O(n)	O(n)	O(n)	O(n)
Binary Search Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(n)	O(n)	O(n)	O(n)	O(n)
Cartesian Tree	N/A	Θ(log(n))	Θ(log(n))	Θ(log(n))	N/A	O(n)	O(n)	O(n)	O(n)
B-Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
Red-Black Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
Splay Tree	N/A	Θ(log(n))	Θ(log(n))	Θ(log(n))	N/A	O(log(n))	O(log(n))	O(log(n))	O(n)
AVL Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
KD Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(n)	O(n)	O(n)	O(n)	O(n)