

Insertion and Deletion in Heap

Insertion in Heaps

The insertion operation is also similar to that of the deletion process.

Given a Binary Heap and a new element to be added to this Heap. The task is to insert the new element to the Heap maintaining the properties of Heap.

Process of Insertion: Elements can be inserted to the heap following a similar approach as discussed above for deletion. The idea is to:

- First increase the heap size by 1, so that it can store the new element.
- Insert the new element at the end of the Heap.
- This newly inserted element may distort the properties of Heap for its parents. So, in order to keep the properties of Heap, **heapify** this newly inserted element following a bottom-up approach.



Dash



All



Articles



Videos



Problems



Quiz

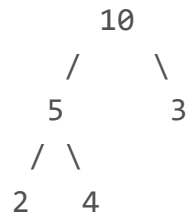
<< Prev

Next >>



Illustration:

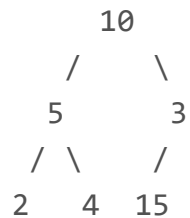
Suppose the Heap is a Max-Heap as:



The new element to be inserted is 15.

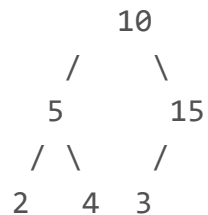
Process:

Step 1: Insert the new element at the end.

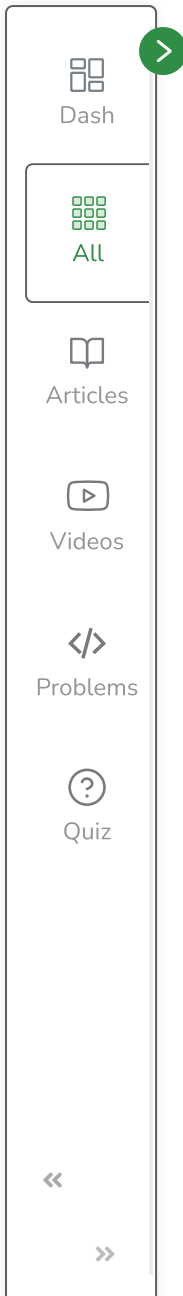
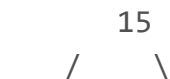


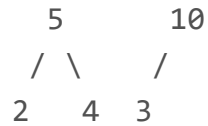
Step 2: Heapify the new element following bottom-up approach.

-> 15 is more than its parent 3, swap them.

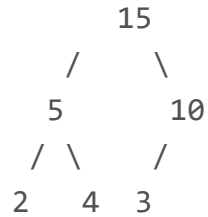


-> 15 is again more than its parent 10, swap them.





Therefore, the final heap after insertion is:



Implementation:

Python

```

# program to insert new element to Heap

# Function to heapify ith node in a Heap
# of size n following a Bottom-up approach

def heapify(arr, n, i):
    parent = int(((i-1)/2))
    # For Max-Heap
    # If current node is greater than its parent
    # Swap both of them and call heapify again
    # for the parent
    if arr[parent] > 0:
        if arr[i] > arr[parent]:
            arr[i], arr[parent] = arr[parent], arr[i]
            # Recursively heapify the parent node
            heapify(arr, n, parent)
    # Function to insert a new node to the Heap
  
```





Dash



All



Articles



Videos



Problems



Quiz



```
def insertNode(arr, key):
    global n
    # Increase the size of Heap by 1
    n += 1
    # Insert the element at end of Heap
    arr.append(key)
    # Heapify the new node following a
    # Bottom-up approach
    heapify(arr, n, n-1)
# A utility function to print array of size n
```

```
def printArr(arr, n):
    for i in range(n):
        print(arr[i], end=" ")
```

```
# Driver Code
# Array representation of Max-Heap
'''
```

```
    10
   / \
  5  3
 / \
2  4
'''
```

```
arr = [10, 5, 3, 2, 4, 1, 7]
```

```
n = 7
```

```
key = 15
```

```
insertNode(arr, key)
```

```
printArr(arr, n)
```

```
# Final Heap will be:
```

```
'''
```

```
    15
   / \
  5  10
 / \ /
2 4 3
```





Dash



All



Articles



Videos



Problems



Quiz



Output

15 5 10 2 4 3

Deletion in Heap

Given a Binary Heap and an element present in the given Heap. The task is to delete an element from this Heap.

The standard deletion operation on Heap is to delete the element present at the root node of the Heap. That is if it is a Max Heap, the standard deletion operation will delete the maximum element and if it is a Min heap, it will delete the minimum element.

Process of Root Deletion (Or Extract Min in Min Heap):

Since deleting an element at any intermediary position in the heap can be costly, so we can simply replace the

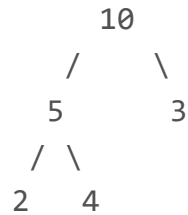


element to be deleted by the last element and delete the last element of the Heap.

- Replace the root or element to be deleted by the last element.
- Delete the last element from the Heap.
- Since, the last element is now placed at the position of the root node. So, it may not follow the heap property. Therefore, **heapify** the last node placed at the position of root.

Illustration:

Suppose the Heap is a Max-Heap as:

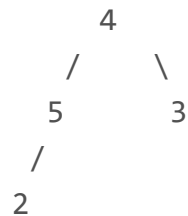


The element to be deleted is root, i.e. 10.

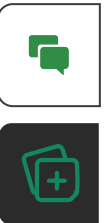
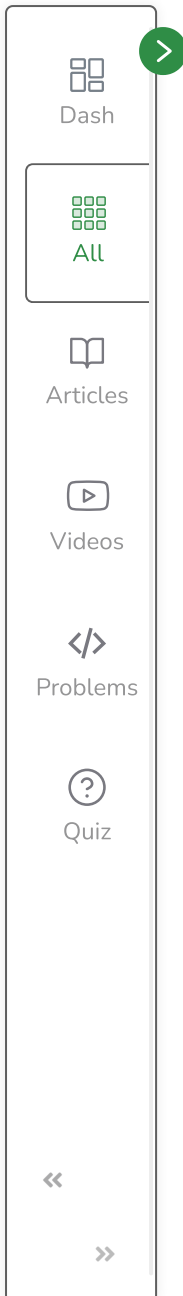
Process:

The last element is 4.

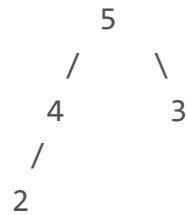
Step 1: Replace the last element with root, and delete it.



Step 2: Heapify root.



Final Heap:



Implementation:

Python

```

# Python 3 program for implement deletion in Heaps

# To heapify a subtree rooted with node i which is
# an index of arr[] and n is the size of heap
def heapify(arr, n, i):


    largest = i #Initialize largest as root
    l = 2 * i + 1 # Left = 2*i + 1
    r = 2 * i + 2 # right = 2*i + 2


    #If left child is larger than root
    if (l < n and arr[l] > arr[largest]):
        largest = l


    #If right child is larger than largest so far
    if (r < n and arr[r] > arr[largest]):
        largest = r

    # If largest is not root
    if (largest != i):
        arr[i], arr[largest] = arr[largest], arr[i]


```


 Dash


 All





Get 90% Refund!

 Courses

 Videos

 Problems

 Quiz



Tutorials

Jobs

Practice

Contests



```
#Recursively heapify the affected sub-tree
heapify(arr, n, largest)
```

```
#Function to delete the root from Heap
```

```
def deleteRoot(arr):
```

```
    global n
```

```
    # Get the Last element
```

```
    lastElement = arr[n - 1]
```

```
    # Replace root with Last element
```

```
    arr[0] = lastElement
```



```
# heapify the root node
```

```
heapify(arr, n, 0)
```



```
# A utility function to print array of size n
```

```
def printArray(arr, n):
```



```
    for i in range(n):
```

```
        print(arr[i],end=" ")
```



```
    print()
```

```
# Driver Code
```

```
if __name__ == '__main__':
```

```
    # Array representation of Max-Heap
```

```
    #      10
```

```
    #    /  \
```

```
    #   5   3
```

```
    #  /  \
```

```
    # 2   4
```

```
    arr = [ 10, 5, 3, 2, 4 ]
```

```
    n = len(arr)
```

```
    deleteRoot(arr)
```




```
printArray(arr, n)
```



Dash



All



Articles



Videos



Problems



Quiz



Output

5 4 3 2

Time complexity: $O(\log n)$ where n is no of elements in the heap

Auxiliary Space: $O(n)$

[Mark as Read](#)[🔔 Report An Issue](#)

If you are facing any issue on this page. Please let us know.