CS 514

Quiz 2 - Divide and Conquer, Heap and Priority Queue

Q1. Perform Quick Sort on the following array, taking the pivot as the middle element every time. Show the intermediate output after each timestep

arr = [2, 1, 5, 3, 4, 6]

Solution -

Code –

```
def partition(arr,left,right):
    i = left
    j = right
    print("Array indexes considered = ",i,j)
    pivot = arr[(left + right)//2]
    print("Pivot Element = " ,pivot)
    while (i<=j):
        while arr[i] < pivot:
            i += 1
        while arr[j] > pivot:
            j -= 1
        print("Intermediate Array(before sorting) =", arr)
        arr[i], arr[j] = arr[j], arr[i]
        print("Intermediate Array(after sorting) =",arr)
        if (i>=j):
            return j

def qsort(input_array, low, high):
    if low < high:
        p_index = partition(input_array, low, high)
        if (low < p_index - 1):
            qsort(input_array,low,p_index -1)
        if p_index < high:
            qsort(input_array,p_index+1,high)
    return input_array

if __name__ == "__main__":
    arr = [2,1,5,3,4,6]
    n = len(arr)
    print(qsort(arr,0,n-1))
```

Output of the code -

Array indexes considered = 0 5
Pivot Element = 5
Intermediate Array(before sorting) = [2, 1, 5, 3, 4, 6]
Intermediate Array(after sorting) = [2, 1, 4, 3, 5, 6]
Intermediate Array(before sorting) = [2, 1, 4, 3, 5, 6]

Intermediate Array(after sorting) = [2, 1, 4, 3, 5, 6]
Array indexes considered = 0 3
Pivot Element = 1
Intermediate Array(before sorting) = [2, 1, 4, 3, 5, 6]
Intermediate Array(after sorting) = [1, 2, 4, 3, 5, 6]
Intermediate Array(before sorting) = [1, 2, 4, 3, 5, 6]
Intermediate Array(after sorting) = [1, 2, 4, 3, 5, 6]
Array indexes considered = 1 3
Pivot Element = 4
Intermediate Array(before sorting) = [1, 2, 4, 3, 5, 6]
Intermediate Array(after sorting) = [1, 2, 3, 4, 5, 6]
Intermediate Array(before sorting) = [1, 2, 3, 4, 5, 6]
Intermediate Array(after sorting) = [1, 2, 3, 4, 5, 6]
Array indexes considered = 1 2
Pivot Element = 2
Intermediate Array(before sorting) = [1, 2, 3, 4, 5, 6]
Intermediate Array(after sorting) = [1, 2, 3, 4, 5, 6]
[1, 2, 3, 4, 5, 6]

Process finished with exit code 0

Explanation –

The quicksort code given considers the pivot as the middle element of the subset of the array always. In the first iteration, the whole array , i.e. [2,1,5,3,4,6] is considered, which is from index 0 to 5. The pivot here will be thee middle element - 5(index – 2) and the list is sorted wrt the pivot, that is all the elements lesser than 5 are moved to left side of the 5 and all the elements larger than 5 will go on the right of 5.
The we recursively sort the left and the right sub lists taking the middle element as pivot. The intermediate arrays will be –

Input array - [2, 1, 5, 3, 4, 6]
Pivot = 5, After sorting - [2, 1, 4, 3, 5, 6]

Array considered – left sublist of previous array – [2,1,4,3]
Pivot – 1, After sorting - [1, 2, 4, 3, 5, 6] where the sorted part has all elements greater than 1 on the right side of 1.

Array considered – right sublist of the pivot 1 subarray – [2,4,3]
Pivot – 4, After sorting - [1, 2, 3, 4, 5, 6] where the sorted part has all elements greater than 1 on the right side of 1.

Array considered – right sublist of pivot 4 subarray – [2,3]
Pivot – 2, After sorting - [1, 2, 3, 4, 5, 6] where these 2 elements are sorted.

Final Arrray – [1, 2, 3, 4, 5, 6]

Q2. Consider a Divide and Conquer algorithm that divides each problem into 2 sub-problems of size 3n/4 each. Write the recurrence relation for this algorithm, and compute the time complexity using Master Theorem.

Answer –

As the algorithm divides each problem into 2 sub problems of size 3n/4 each always, the recurrence relation T(n) of the algorithm will be of the form

$T(n) = 2T(3n/4) + f(n)$

Here as we are not sure what is f(n), we derive time complexity of the recurrence relation assuming 3 cases of f(n) – O(1), O(n) and $O(n^2)$

a. **f(n) = O(1)**

Now, $T(n) = 2T(3n/4) + O(1)$

We know the Master theorem general form is

$$T(n) = aT(n/b) + f(n)$$

where,

a is the number of subproblems,
b is the factor by which the problem size is reduced, and
f(n) is the time complexity outside the recursive calls.

Here, a = 2, b = 4/3 and f(n) = O(1)
For deriving the time complexity, we compare f(n) to $n^{(\log_b (a))}$.
As O(1) is present,
We get – $n^0 = n^{(\log_b (a) + \varepsilon)}$

So, $\log_b (a) + \varepsilon = 0 \Rightarrow \log_{4/3} 2 + \varepsilon = 0$

On solving this equation we get $\varepsilon = \log_3 4$ which is > 0 (Case 1 of Master theorem)
So T(n) =. $\Theta(n^{(\log_b (a))})$.

So, $T(n) = \Theta(n^{(\log_{4/3} (2))})$.

b. **f(n) = O(n)**

Now, $T(n) = 2T(3n/4) + O(n)$

We know the Master theorem general form is

$$T(n) = aT(n/b) + f(n)$$

where,

a is the number of subproblems,
b is the factor by which the problem size is reduced, and
f(n) is the time complexity outside the recursive calls.

Here, a = 2, b = 4/3 and f(n) = O(n)

Now, we compute n^ ($\log_b$ (a)).

$$n^\wedge(\log_{4/3}(2)) = n^\wedge(\log(2)/\log(4/3))$$

Compare f(n) with n^ ($\log_b$ (a)):

As n^($\log_b$ (a)) grows with n, this case falls into case 2 of the Master theorem.

Therefore, the time complexity is

$$\mathbf{T(n) = \Theta((n^\wedge(\log_b(a))^* \log n) = \Theta((n^\wedge(\log_{4/3}(2))^* \log n)}$$

c. **f(n) = O(n^2)**

Now, T(n) = 2T(3n/4) + O(n^2)

We know the Master theorem general form is

$$T(n) = aT(n/b) + f(n)$$

where,

a is the number of subproblems,
b is the factor by which the problem size is reduced, and
f(n) is the time complexity outside the recursive calls.

Here, a = 2, b = 4/3 and f(n) = O(n^2)

Now, we compute n^ ($\log_b$ (a)).

$$n^\wedge(\log_{4/3}(2)) = n^\wedge(\log(2)/\log(4/3))$$

Compare f(n) with n^ ($\log_b$ (a)):

As f(n) = O(n^2) and n^($\log_b$ (a)) grows faster than n^2, this case falls into case 3 of the Master theorem.
Therefore, the time complexity is

$$\mathbf{T(n) = \Theta(f(n)) = \Theta(n^2)}$$

Q3. Given a sequence of numbers: 19, 6, 8, 11, 4, 5

a) **Draw a binary min-heap (in a tree form) by inserting the above numbers and reading them from left to right.**

b) Show a tree that can be the result after the call to deleteMin() on the above heap.

c) Show a tree after another call to deleteMin().

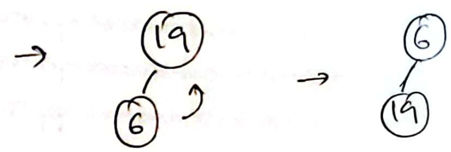
Q2.] Solution →

Input Array → [19, 6, 8, 11, 4, 5]

We start by adding 19 as we are reading the array from left to right.

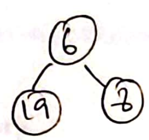
Step 1.) → 19 [Till now, 19 is the only node]

Step 2.) We add 6 to the tree. It will be added as a child to node 19 but will be bubbled in place of 19 as it is less than 19

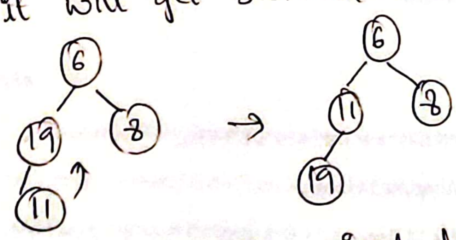


19 will be child of 6.

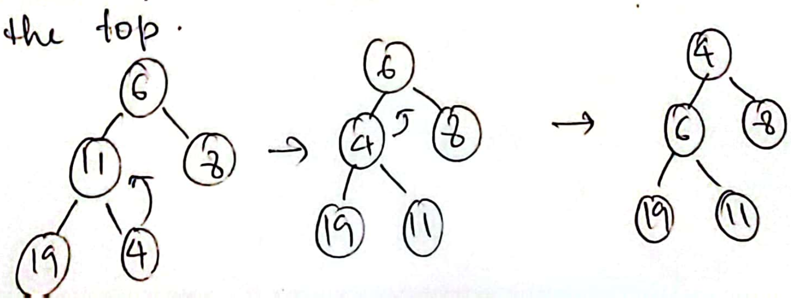
Step 3.) We add 8 to this tree. 8 will be added as a child of 6 and as 8 is larger than 6, it stays as 6's child



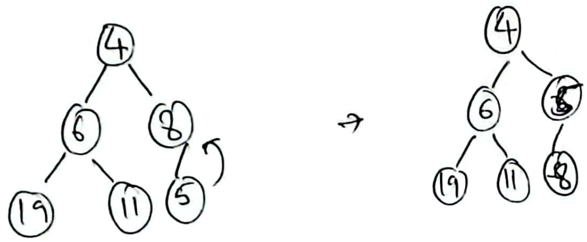
Step 4.) We add 11 as left child of 19 and as 11 is greater than 19, it will get bubbled to the top at 19's initial place



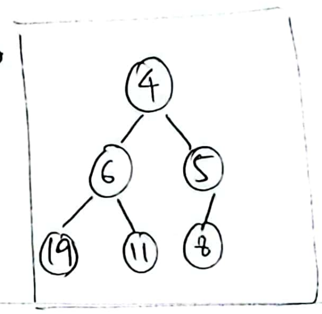
Step 5.) We add 4 as the right child of 11, but it bubbles to the top.



Step 6.] We add 5 as the left child of 8 node. It will be swapped with node 8 as $5 < 8$

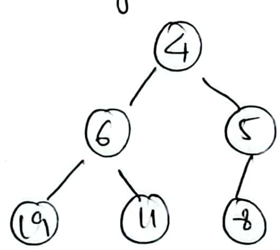


The final min heap will be →

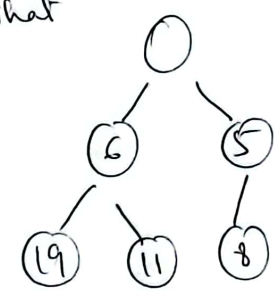


Q6b.) Show a tree that can be the result of after the call to deleteMin() on the above heap.

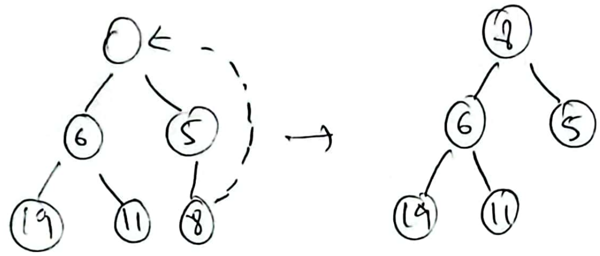
Answer - The binary tree we have here is as shown



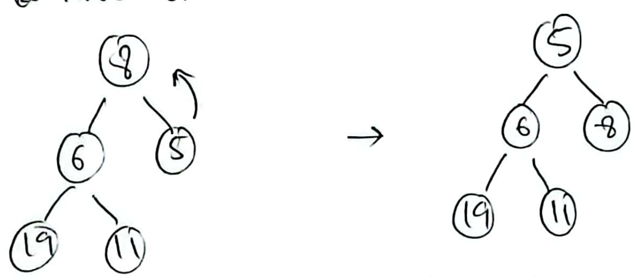
On calling deletemin(), 4 is removed and there is a vacant space at that spot.



As there is a free space at that spot, the last node which is 8 is moved there



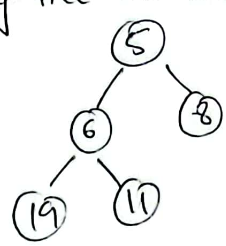
As we have to maintain min-heap property and as 8 is greater than its children, we need to swap 8 with one of its children. We compare the children & swap it with the smallest child.



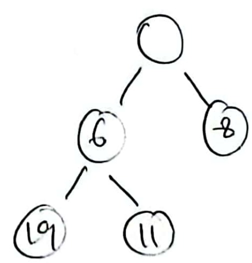
This is a tree which is a result after delete min() is called.

Q c) Show a tree after another call to deletemin()

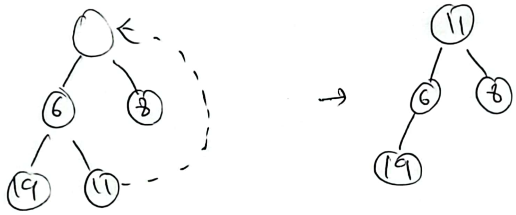
Answer - The binary tree we have now is



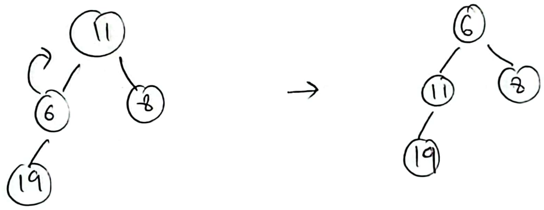
On calling deletemin(), 5 is removed and there is a vacant space at that spot →



As there is a vacant space at that spot, the last node which is 11 is moved there



As we have to maintain min-heap property and here, 11 is greater than both of its children, we swap it with the smallest child which is 6



This is the resultant tree

Q4. What is the big-Oh time complexity of getting a sorted array out of a max heap? Justify your answer.

The removal of the maximum element from a max heap is a key operation in transforming it into a sorted array. This operation involves two main steps: locating and removing the maximum element, followed by restoring the heap property.
a. **Removal of Maximum Element (O(log(n)))** - The process of finding and removing the maximum element is a logarithmic operation, denoted as O(log(n)). This is because the maximum element is located at the root of the heap, and to remove it while maintaining the heap structure, the algorithm needs to traverse the height of the heap, which is log(n) in the worst case.
b. **Re-balancing the Heap (O(log(n))** - After removing the maximum element, the heap needs to be re-balanced to maintain its properties. The re-balancing process, which involves moving elements to their correct positions, also has a time complexity of O(log(n)).
c. **Total Time Complexity (O(nlog(n)))** - Since you repeat this removal process for all n elements in the heap, the overall time complexity becomes O(n * log(n)), where n is the number of elements in the heap.