Vrushabh Shashikant Bhagwatkar
2019BCS074
B-40
AA
TY

# Practical 2

## Implementing Heap sort Algorithm and performing time complexity Analysis.

* Overview of Heapsort →

Heapsort algorithm mainly consists of two parts - converting the list into a heap and adding the max element from the heap to the end of the list, while maintaining the heap structure. For easy implementation we can use max heap structure, where the max value always exists at the root. After converting the list into a heap, we take the max element from it and add it to the end of the list. We repeat this untill elements in heap become zero. This indicates that we have arranged all items in the list as per the correct order.

★ Time complexity of Heap data structure →

Because we make use of a binary tree, the bottom of the heap contains the maximum number of nodes. As we go up a level, the number of nodes decreases by half. considering there are 'n' number of nodes, then the number of nodes starting from the bottom-most level would be, n/2, n/4, n/8 n/16 and so on.

★ complexity of creating a new node

Therefore when we insert a new value in the heap when making the heap, the max number of steps we would need to take comes out to be O(nlog(n)). As we use binary trees, we know that the max height of such a structure is always O(log(n)). ∴ insertion of new value would be O(log(n)).

★ Complexity of removing max node from heap-

Likewise, when we remove the max valued node from the heap, to add to the end of the list, the max number of steps req would be O(log(n)). Since we swap max val. node till it comes down to bottom level, the max no. of steps would be O(nlog(n)).

* Complexity of creating heap –
As per the discussion above no. of steps we would take is,
$(n/2 \times 0) + (n/4 \times 1) + (n/8 \times 2) + (n/16 \times 3) + \ldots n$
summation of this series will give – $n/2$
∴ it would be $O(n)$.

* Average case time complexity of Heap sort
In terms of total complexity, we already know that we can create a heap in $O(n)$ time and do insertion and removal of nodes in $O(\log(n))$ time. In terms of avg time, we need to take into acc all possible inputs. If total no. of node is n, then,
- $\log(n)/2$ comparision in first iteration
- $\log(n-1)/2$ in second.
- and so on, ...

So, mathematically, it would sum up to
$= (\log(n))/2 + (\log(n-1))/2 + \ldots$
$= 1/2\,(\log(n!))$ (after approximation)
$= 1/2\,(n \times \log(n) - n + O(\log(n)))$
$= O(n\log(n))$.

* Conclusion – The time complexity of heapsort is $O(n\log(n))$ which can also be seen in graph below.

In [8]:
```
import time
from numpy.random import seed
from numpy.random import randint
import matplotlib.pyplot as plt

def left(i):
    return 2 * i + 1

def right(i):
    return 2 * i + 2

def heapSize(arr):
    return len(arr)-1

def MaxHeapify(arr, i):
    l = left(i)
    r = right(i)

    if l<= heapSize(arr) and arr[l] > arr[i] :
        largest = l
    else:
        largest = i
    if r<= heapSize(arr) and arr[r] > arr[largest]:
        largest = r
    if largest != i:
        arr[i], arr[largest]= arr[largest], arr[i]
        MaxHeapify(arr, largest)

def BuildMaxHeap(arr):
    for i in range(int(heapSize(arr)/2)-1, -1, -1):
        MaxHeapify(arr, i)

def HeapSort(arr):
    BuildMaxHeap(arr)
    arrr = list()
    heapSize1 = heapSize(arr)
    for i in range(heapSize(arr), 0, -1):
        arr[0], arr[i]=arr[i], arr[0]
        arrr.append(arr[heapSize1])
        arr = arr[:-1]
        heapSize1 = heapSize1-1
        MaxHeapify(arr, 0)
```

```
elements = list()
times = list()
for i in range(1, 10):

    a = randint(0, 10000 * i, 10000 * i)
    start = time.clock()
    HeapSort(a)
    end = time.clock()
    elements.append(len(a))
    times.append(end-start)

plt.xlabel('List Length')
plt.ylabel('Time Complexity')
plt.plot(elements, times, label ='Heap Sort')
plt.legend()
plt.show()
```