Date: / /

S M T W T F S

H.W. 2

Answer to the Question No 1

To analyze the worst-case time complexity of a 3-way search, let's analyze the process.

At first, to define the recurrence relation for the worst-case time complexity of a 3-way search, in a 3-way search, to divide the array into three parts by choosing $n/3$ and $2n/3$

i. To compare the key elements at $n/3$ and $2n/3$

ii. Based on the comparison, search recursively of three subarrays:

From the start 0 to $n/3 - 1$

From $n/3 + 1$ to $2n/3 - 1$

From $2n/3 + 1$ to $n - 1$

The recurrence relation for the worst-case time complexity $T(n)$ is:

$$T(n) = T\left(\frac{n}{3}\right) + O(1)$$

Here, $n/3$ is equal time to search the reduced interval;

$O(1)$ is equal constant time for comparisons and splits.

Date: / /

Now, to expand the recurrence relation step by step:

1st: $T(n) = T(n/3) + O(1)$

2nd: $T(n/3 = T(n/3 \cdot \frac{1}{3}) + O(1)$

$= T(n/9) + O(1)$

Substituting into the first equation:

$T(n) = (T(n/9) + O(1)) + O(1)$

$\therefore T(n) = T(n/9) + O(2)$

3rd:

$T(n) = T(n/9 \cdot \frac{1}{3}) + O(2) + O(1)$

$= T(n/27) + O(3)$

~~After~~ K :

$T(n) = T(n/3^k) + O(k)$

$n/3^k = 1$

Solving for K : $3^k = n$

$\Rightarrow k = \log_3 n$

Date: / /

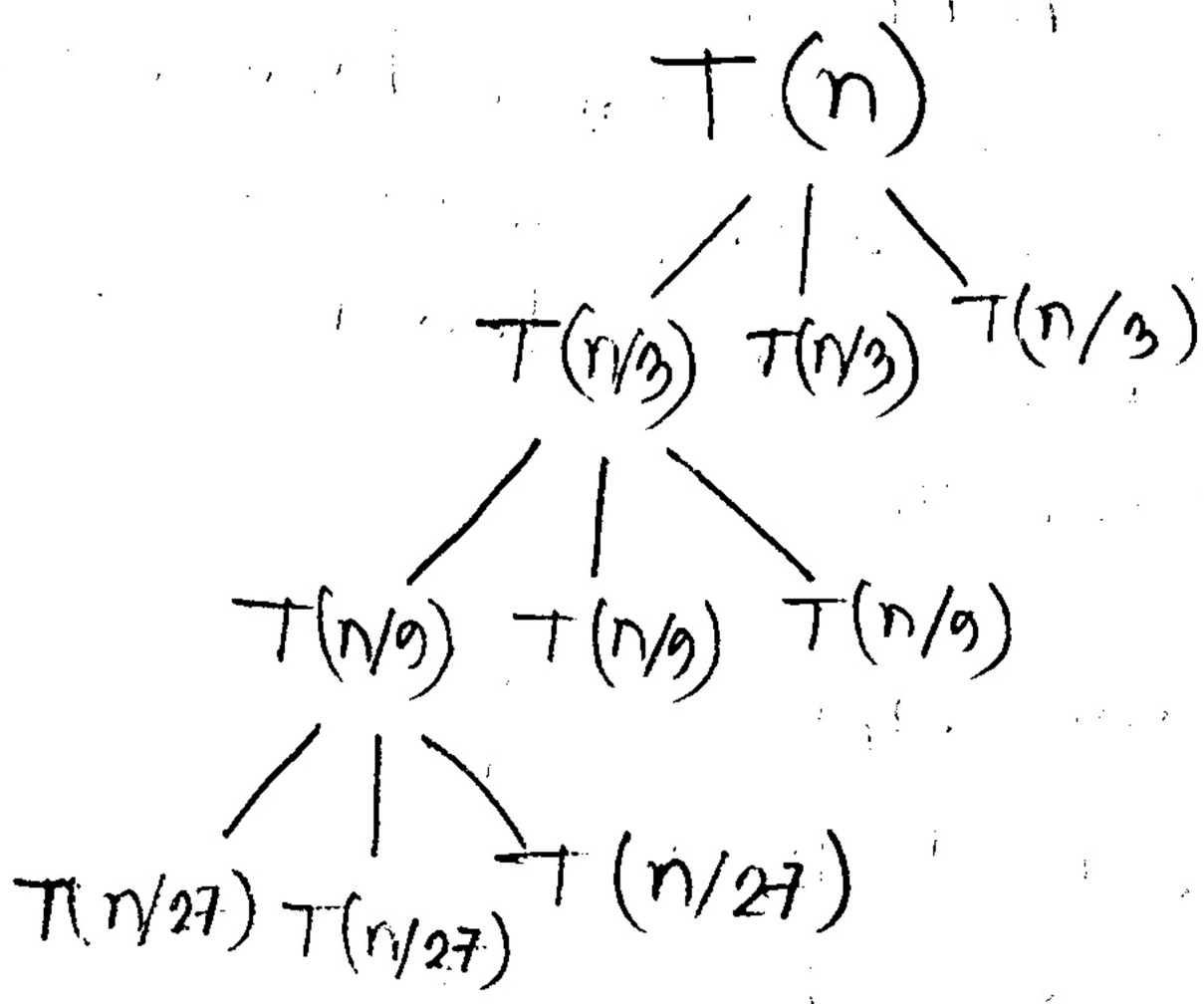
S M T W T F S

So. $T(n) = T(1) + O(\log_3 n)$

$T(n) = O(\log_3 n)$ [$T(1)$ is a constant]

The worst-case time complexity of 3-way search is : $O(\log_3 n)$

Now, to draw a simple tree structure is the best way to represent the recursion.



T(1) [Base case]

Date: / /

S M T W T F S

Lastly, to write the pseudocode for 3-way search is given below -

```
function three-way-search(arr, left, right, key):
    if left > right:
        return -1; // key not found

mid 1 = left + (right - left)/3
mid 2 = right - (right - left)/3

if arr[mid 1] == key:
    return mid 1
if arr[mid 2] == key:
    return mid 2

if key < arr[mid 1];
    return three_way- search (arr, left, mid 1-1, key)

else if key > arr[mid 2]:
    return three-way- search (arr, mid 2+1, right, key)

else:
    return three-way-search (arr, mid 1+1, mid 2-1, key)
```

The function splits the input array into three parts and recursively searches in the appropriate segment.

Date: / /

S M T W T F S

Answer to the Question No-2

a] No. BUILD-MAX-HEAP and BUILD-MAX-HEAP' do not always create the same heap when run on the same input array.

Now, analyzing the reason;

BUILD-MAX-HEAP(A,n) is a bottom-up approach that starts from the last non-leaf node and applies MAX-HEAPIFY to build the heap in O(n) time. On the other hand, BUILD-MAX-HEAP'(A,n) is a top down approach where each element is inserted into the heap using MAX-HEAP-INSERT.

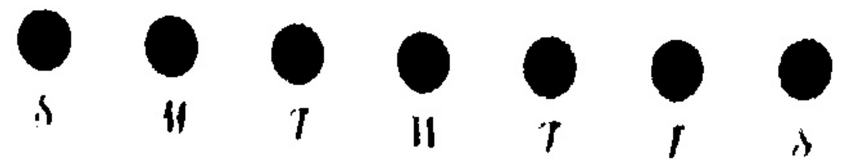
For example, the input array:

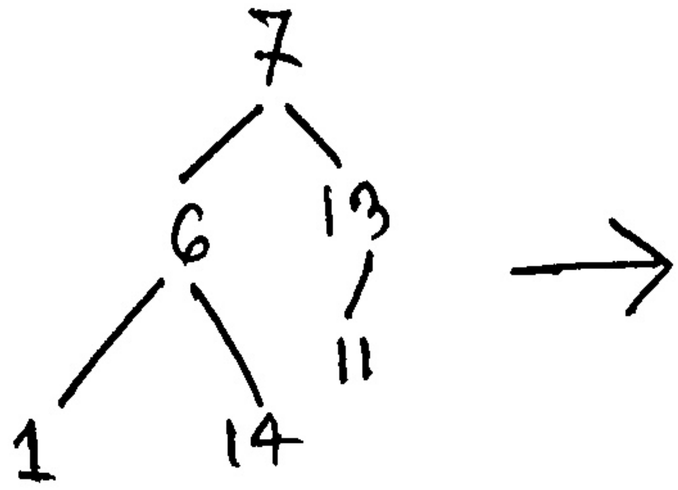
A = [7, 6, 13, 1, 14, 11]

Applying BUILD-MAX-HEAP(A, 6) (Bottom-up method)

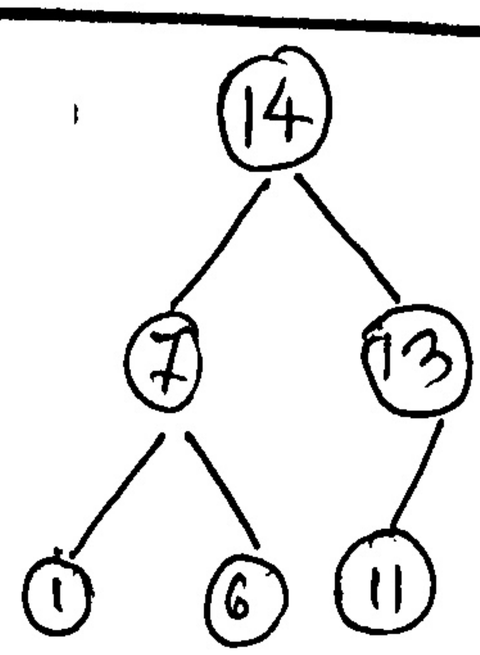
The tree before heapify:

Date: / /



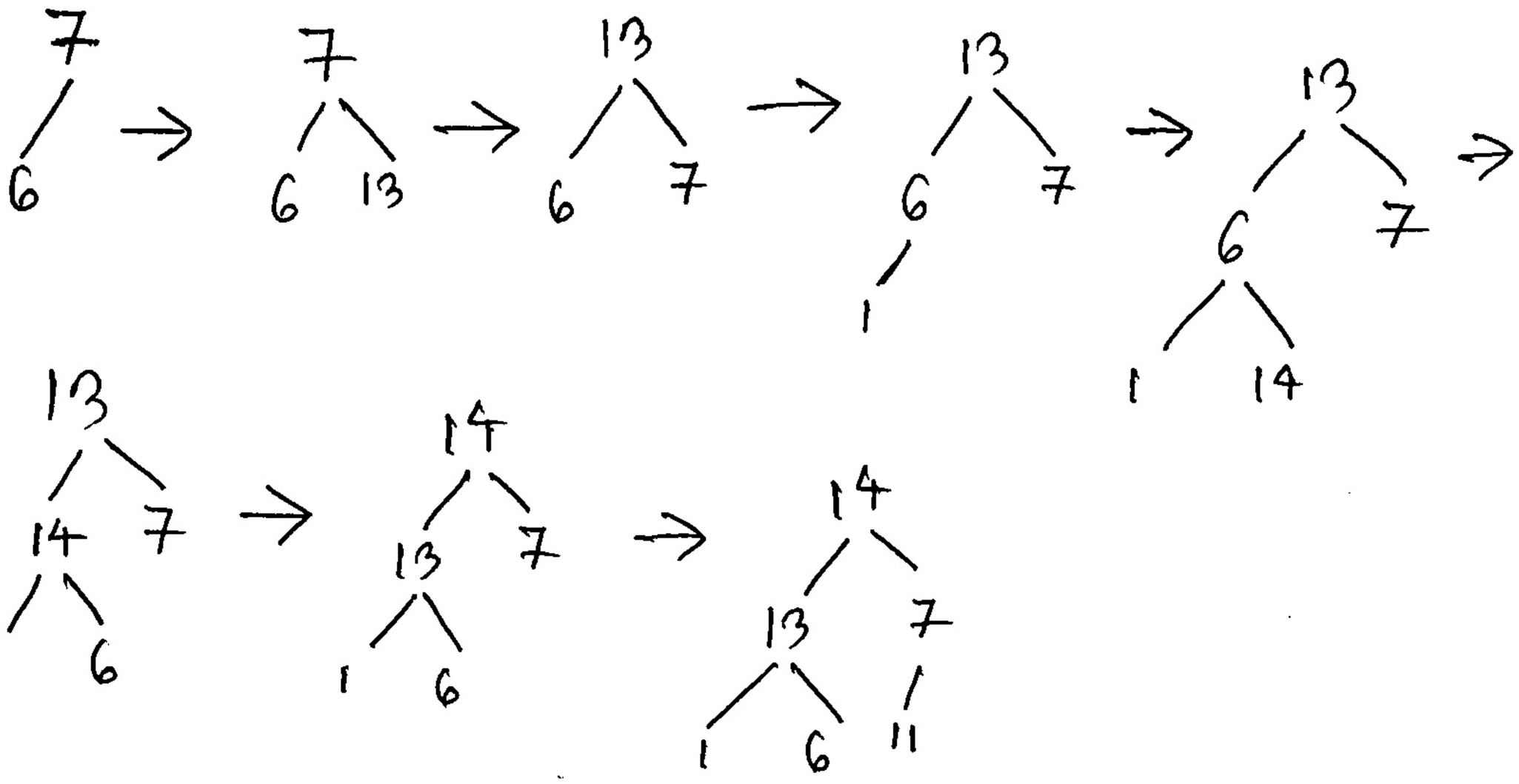


Before Heapify



After Heapify

Now, applying BUILD-MAX-HEAP'(A, 6) (top-down method):



Both methods will ensure a valid maxheap, but their final structures might differ due to different insertion and heapification processes.

Date: / /

b] The BUILD-MAX-HEAP' algorithm starts with an empty heap and inserts n elements one by one using MAX-HEAP-INSERT. This process, inserting an element into a heap requires at most $O(\log n)$ operations in the worst case.

Finding the total cost of BUILD-MAX-HEAP;
The worst case scenario where why every inserted element bubbles up to the root.

When no bubble-up needed → $O(1)$
At most 1 swap → $O(1)$
At most 2 swaps → $O(\log 3)$
At most 2 swaps → $O(\log 4)$
At most 3 swaps → $O(\log 5)$
. . . . . .
At most $\log n$ swaps → $O(\log^n)$

Based on (a) example

Thus, $T(n) = O\left(\sum_{k=1}^{n} \log k\right)$, here the total cost of inserting n elements..

To use the integral:

$$\sum_{k=1}^{n} \log_k = \int_1^n \log x \, dx$$

Date: / /

So, $\int_1^n \log x \, dx = [x \log x]_1^n - [x]_1^n$

$= (n \log n - n) - (1 \log 1 - 1)$

$= n \log n$

$\therefore \sum_{K=1}^{n} \log K = n \log n$

Now, to write that,

$T(n) = \Theta(n \log n)$

$\therefore T(n) = \Theta(n \log n)$

Shown that, the worst case, BUILD-MAX-HEAP' requires $\Theta(n \log n)$ time to build an n-element heap.

Date:

Answer to the Question 3

a To argue that TRE-QUICKSORT (A, 1, n) correctly sorts the array A[1:n]. The-Quicksort algorithm calls PARTITION (A, P, r) which rearranges elements in the array like that elements smaller than the pivot are placed in the left subarray and also greater than or equal to the pivot are placed in the right subarray. In addition to the algorithm calls TRE-QUICKSORT (A, P, q-1) to recursively sort the left subarray, ensuring that all elements in the left partition are sorted. The right subarray is processed iteratively by updating P = q+1, ensuring that it is also completely sorted.

Now, The partition (A, P, r) rearranges the elements like as the elements in A[P: q-1] are less than or equal to the pivot and A[q+1, r) are greater than or equal to the pivot.

When q ≥ r, the recursion stops.

Date: / /

S M T W T F S

A recursive call for the right subarray. it updates p to q+1 and continues the loop processing the right subarray iteratively.

Thus, TRE-QUICKSORT preserves the logic of the standard QuickSort algorithm while eliminating the another recursive call. So, TRE-QUICKSOR (A, 1, n) correctly sorts the array A[1:n] while optimizing stack.

b|| To determine and prove a seenario where TRE-QUICKSORT's stack depth is $\theta(n)$

The stack depth is determined by how many recursive calls exist. The wost case occurs when PARTITION always selects the smallest or largest element as the pivot, leading to highly unbalanced partitions.

Consider TRE-QUICKSORT (A, 1, n) where A = [1, 2, 3, 4, 5]

TRE-QUICKSORT (A, 1, 5)
↳ TRE-QUICKSORT (A, 1, 4)
↳ TRE-QUICKSORT (A, 1, 3)
↳ TRE-QUICKSORT (A, 1, 2)
↳ TRE-QUICKSORT (A, 1, 1)

Date: / /

The stack grows proportional to n, Maximum stack depth = n, which means $\Theta(n)$ stack depth. Now to prove it mathematically:

$$T(n) = T(n-1) + O(n)$$

$$T(n-1) = T(n-2) + O(n-1)$$ [Expanding recursively]

$$T(n-2) = T(n-3) + O(n-2)$$

$$\vdots$$

$$T(1) = O(1)$$

To sum up,

$$T(n) = O(n+(n-1)+(n-2)+\cdots+1)$$

$$\Rightarrow T(n) = O(n-(n+1)/2)$$
$$= O(n^2)$$

So, $O(1)$ extra stack space, and we have n recursive calls, ~~the~~ stack depth is $\Theta(n)$.

For an array A[1, 2, 3, 4, 5], to write graphical

$$T(5) \rightarrow T(4) \rightarrow T(3) \rightarrow T(2) \rightarrow T(1)$$

$n = 5$, therefore 5 steps to complete the recursive.

This is proven that in the worst case, the stack depth of Prove TRE-QUICKSORT will be $\Theta(n)$.

Date: / /

S M T W T F S

4] To ensure the worst-case stack depth is $\Theta(\log n)$ while maintaining the $O(n \log n)$ desired running time, to modify the TRE-QUICKSORT algorithm. If the partition size is $n-1$ and 0, recursion depth can reach $\Theta(n)$ in the worst case. This happens because the recursive call is always made on the left subarray first.

Now, to write the modified algorithm :-

```
UPD-TRE_QUICKSORT (A, P, r)
  while P<r
    q = PARTITION (A, P, r)
if (q-P) < (r-q)
  TRE-QUICKSORT (A, P, q-1)
  P = q+1
else :
TRE-QUICKSORT (A, q+1, r)
  r = q-1
```

The modified algorithm is to add a comparison operation before each recursive call and to perform the partition operations always on the smaller subarrays. This way, the stack

Date: / /

depth is promised to be $\Theta(\log n)$. the overall runtime, however, is increased by $a\log n$

$$O(n\log n) + O(\log n) = O(n\log n)$$

Thus, the algorithm has a stack depth of $\Theta(\log n)$, and the expected running time is $O(n\log n)$. The worst case stack depth for the modified algorithm will be $\Theta(n\log n)$.