HW-2

Abu Noman Md Sakib
UOM581

Answer to the Question No-1:

To setup the recurrence relation for the worst-case time complexity of a 3-way search, we need to first analyze the process. The Pseudocode for 3-way search is given below:

```
function threeWaySearch(arr, left, right, key):
    if left > right:
        return -1; // key not found.

    mid1 = left + (right-left)/3
    mid2 = right - (right - left)/3

    if arr[mid1] == key:
        return mid1
    if arr[mid2] == key:
        return mid2

    if key < arr[mid1]:
        return threeWaySearch(arr, left, mid1-1, key)
    else if key > arr[mid2]:
        return threeWaySearch(arr, mid2+1, right, key)
    else:
        return threeWaySearch(arr, mid1+1, mid2-1, key)
```

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

Now, each iteration makes two comparisons at positions $n/3$ and $2n/3$. The function then recursively searches one of the three parts, each of size $n/3$. So, the recurrence relation is:

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

where, $T(n)$ is the time complexity and $O(1)$ refers to the comparisons.

We solve the recurrence using the iterative expansion method as shown below:

$$T(n) = T(n/3) + O(1)$$
$$T(n/3) = T(n/9) + O(1)$$

Therefore, $T(n) = T(n/9) + O(1) + O(1)$
$= T(n/9) + 2\,O(1)$

$$T(n/9) = T(n/27) + O(1)$$

Similarly, $T(n) = T(n/27) + 3\,O(1) = T(n/3^3) + 3\,O(1)$

So, after k iterations, $T(n) = T(n/3^k) + k\,O(1)$

The recursion stops when $n/3^k = 1$, solving that,

$$n/3^k = 1$$

$$\Rightarrow 3^k = n$$

$$\Rightarrow k = \log_3 n$$

Therefore, $T(n) = O(\log_3 n)$

So, the worst case time complexity of 3-way search is $O(\log_3 n)$

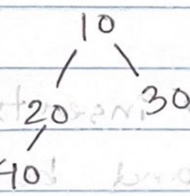
Answer to the Question No-2:

a) To compare if BUILD-MAX-HEAP and BUILD-MAX-HEAP' creates the same heap, we need iterate through both processes. BUILD-MAX-HEAP constructs a max-heap by starting from the last non-leaf node and calling MAX-HEAPIFY to push elements down in the heap. BUILD-MAX-HEAP' constructs the heap incrementally by inserting each element one at a time using MAX-HEAP-INSERT.

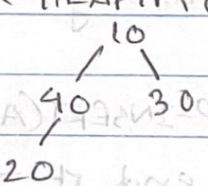
BUILD-MAX-HEAP and BUILD-MAX-HEAP' do not always create the same heap for the same input array.

Let, an input array → [10, 20, 30, 40].
Using BUILD-MAX-HEAP,

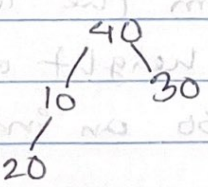
Initial:



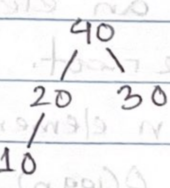
MAX-HEAPIFY (A,2) on 20:



MAX-HEAPIFY (A, 1) on 10:

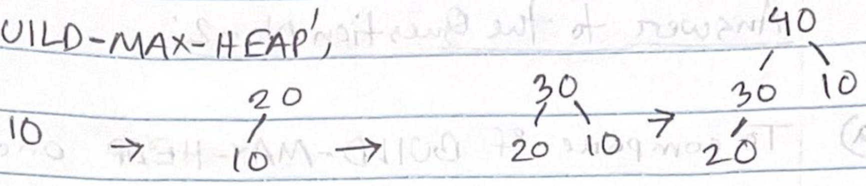


MAX-HEAPIFY (A,2) on 10:



Final heap → [40, 20, 30, 10]

Using BUILD-MAX-HEAP',



final heap → [40, 30, 10, 20]

So, BUILD-MAX-HEAP produced [40, 20, 30, 10] while BUILD-MAX-HEAP' produced [40, 30, 10, 20]. Thus, it is proven that, BUILD-MAX-HEAP and BUILD-MAX-HEAP' do not always generate the same heap.

b) To analyze the time complexity of BUILD-MAX-HEAP', we need to understand that the algorithm calls MAX-HEAP-INSERT n times, where each call inserts an element and maintains the heap properly.

MAX-HEAP-INSERT(A, A[i], n) → inserts an element at the end of the heap and bubbles it up to maintain the heap property. In worst case, an element moves from the last level to the root. The maximum height of a heap with n elements is $O(\log n)$, so an insertion takes $O(\log n)$ time.

Therefore,

$$T(n) = \sum_{i=1}^{n} O(\log i)$$

$$\sum_{i=1}^{n} \log i = \int_{1}^{n} \log x \, dx$$

$$= [x \log x]_1^n - [x]_1^n$$

$$= n \log n - n + 1$$

$$\simeq n \log n \rightarrow O(n \log n)$$

Thus, BUILD-MAX-HEAP' runs in $\Theta(n \log n)$ to build an n-element heap in the worst case scenario.

[showed]

Answer to the Question No - 3:

a) From the section 7.1, which contains two recursive calls to itself, it is proven that the QUICKSORT algorithm correctly sorts the array A. TRE-QUICKSORT differs from QUICKSORT in only the last line of the loop. In the TRE-QUICKSORT, we eliminated the second recursive call and replaced it with an iterative approach.

if $p \geq r$, the function terminates
the PARTITION(A,p,r) → rearranges the element such that 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.
The function recursively sorts the left subarray and then updates p value.

Each iteration correctly partitions and sorts the left subarray, then updates p to process the right subarray iteratively. The TRE-QUICKSORT effectively performs the sort in the same manner as QUICKSORT. Therefore, TRE-QUICKSORT correctly sorts the array A.

b) The worst case scenario for stack dept occurs when the partitioning is highly unbalanced. For example, in an already sorted or reverse sorted input, the partition picks the smallest (or largest) element as the pivot. So, we get,

↳ TRE-QUICKSORT (A, 1, n)
↳ TRE-QUICKSORT(A, 1, n-1)
↳ TRE-QUICKSORT(A, 1, n-2)
⋮
↳ TRE-QUICKSORT(A, 1, 1)

This results in n recursive calls, leading in a stack depth of $\Theta(n)$.

For, an array [1, 2, 3, 4, 5], the graphical representation will be

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

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

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

① To ensure a maximum stack depth of $\Theta(\log n)$, while maintaining the $O(n\log n)$ expected running time of the algorithm, we need to make sure to perform the recursive operation on the smaller subarray and iterate on the larger subarray.

The modified TRE-QUICKSORT will be:

```
TRE-QUICKSORT-MOD (A, p, r)
  while p<r:
      q = PARTITION (A, p, r)
      if (q-p < r-q):
          TRE-QUICKSORT-MOD (A, p, q-1)
          p = q+1
      else:
          TRE-QUICKSORT-MOD (A, q+1, r)
          r = q-1
```

This modified algorithm applies recursion on the smaller portion and iterates over the larger one in each iteration. Since, each recursive call operates on half the elements (at most), the depth follows a logarithmic pattern. The runtime will be,

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

Therefore, keeping the running time at $O(n\log n)$, the worst case stack depth for the modified algorithm will be $\Theta(\log n)$.