CS-1203 (Monsoon 2023) Quiz 1 (take home) #
Karthik Kunnel Sunil. 09/2023

Problem 1.

Solution.

The given code is implementing two different strategies for creating a max-heap from an array. A max-heap is a complete binary tree in which the value of each node is less than or equal to the values of its children.

Heapify Method:

The array is initially in ascending order: $0, 1, 2, 3, 4, 5, 6, 7, 8, 9$.

After applying the heapify operation, the array is transformed into a max-heap:
$9, 8, 6, 7, 4, 5, 2, 0, 3, 1$.

Arrays Initialization:

Arrays `arr1` and `arr2` are initially the same, both containing the numbers from 0 to 9 in ascending order.

In the heapify method, the `arr1` array is transformed into a max-heap by repeatedly applying the heapify operation to each non-leaf node from the last non-leaf node to the root.

Here's how `arr1` changes with each iteration of the heapify loop:

Before heapify: `arr1` = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9}

We first find the $i = 10/2 - 1 = 4$
then, we have got the function heapify(arr1[], 10, 4) with the following arguments. Once the function is executed the following variables are declared and initialised:
$\longrightarrow$ left $= 2 * 4 + 1 = 9$
$\longrightarrow$ right $= 2 * 4 + 2 = 10$
$\longrightarrow$ largest $= 4$

Now, we evaluate the 'if' conditions:
if(left $<$ n AND arr[left] $>$ arr[largest]), in this case it evaluates to be True. Thus, the following swap occurs.

After heapifying node 4: `arr1` = {0, 1, 2, 3, 9, 5, 6, 7, 8, 4}

The process follows and we end up with the following arr.

After heapifying node 0: `arr1` = {9, 8, 6, 7, 4, 5, 2, 0, 3, 1}

Heap Insert Method:

The array is initially in ascending order: $0, 1, 2, 3, 4, 5, 6, 7, 8, 9$.

The **heapInsert** function inserts an element into a heap. This function takes three parameters: `arr[]` which is the array representing the heap, `n` which is the current size of the heap, and `elm` which is the element to be inserted.

The **heapInsert** function works as follows:

1. Place `elm` at the end of the array (i.e., at index `n`). Set `i` to `n`.

2. If `i` is 0 (i.e., the heap was empty before the insertion), return immediately because the heap now contains just one element and is already a valid heap.

3. Otherwise, enter a loop that continues as long as `i` is not 0. In each iteration of the loop:

   (a) Compute the index of the parent node as $\frac{i-1}{2}$.

   (b) If the value at the parent index is less than the value at `i`, swap these two values.

   (c) Otherwise, break the loop because the heap property has been restored.

   (d) Update `i` to the parent index.

Let's see step by step how the array changes when inserting elements from `arr1[]` = {0,1,2,3,4,5,6,7,8,9}:

- Before any insertions, `arr1` is empty.
- Insert 0: `arr1` = {0}
- Insert 1: `arr1` = {1, 0} (0 and 1 are swapped)
- Insert 2: `arr1` = {2, 0, 1} (1 and 2 are swapped)
- Insert 3: `arr1` = {3, 2, 1, 0} (2 and 3 are swapped)
- Insert 4: `arr1` = {4, 3, 1, 0, 2} (3 and 4 are swapped)
- Insert 5: `arr1` = {5, 4, 1, 0, 2, 3} (4 and 5 are swapped)
- Insert 6: `arr1` = {6, 5, 3, 0, 2, 1, 4} (5 and 6 are swapped, 3 and 4 are swapped)
- Insert 7: `arr1` = {7, 6, 3, 5, 2, 1, 4, 0} (6 and 7 are swapped)
- Insert 8: `arr1` = {8, 7, 3, 6, 2, 1, 4, 0, 5} (7 and 8 are swapped, 6 and 5 are swapped)
- Insert 9: `arr1` = {9, 8, 5, 6, 7, 1, 4, 0, 3, 2} (8 and 9 are swapped, 5 and 2 are swapped, 7 and 3 are swapped)

The resulting array `arr1` represents a max heap, where each parent node is greater than or equal to its children.

After inserting elements into the heap one by one, the array is transformed into a max-heap: $9, 8, 5, 6, 7, 1, 4, 0, 3, 2$.

Comparisons and Swaps:

In the Heapify method, each node is visited once, and each node can be swapped down the tree $\log(n)$ times in the worst case. So, the total number of comparisons and swaps can be up to $n \log(n)$, where n is the number of elements in the array.

In the Heap Insert method, each insertion can cause up to $\log(n)$ comparisons and swaps in the worst case. Since there are n insertions, the total number of comparisons and swaps can be up to $n \log(n)$.

Worst Case Complexity:

The worst-case time complexity for both Heapify and Heap Insert methods is $O(n \log n)$. This is because, in the worst case, each node can cause a cascade of swaps down to the leaf level of the tree, and the height of the tree is $\log(n)$.

However, it is worth noting that while the worst-case complexity is the same for both methods, the Heapify method can be more efficient in practice because it performs the heapify operation from the bottom up, which can reduce the number of swaps compared to the Heap Insert method.