Name : Eslam Mohamed Abbas Abbas Hammad
Section : 2
Department : CS
ID:1000263795

(a) Algorithms for Heapsort

Heapsort requires two main algorithms:

Build-Max-Heap: Constructs a max-heap from an unordered array.
Heapify: Maintains the max-heap property.
Heapsort: Sorts the array by repeatedly extracting the maximum element.

Algorithm 1: Build-Max-Heap

—----------

Build-Max-Heap(A)
Input: An array A of size n
Output: A max-heap

1. n = length(A)
2. for i = ⌊n / 2⌋ down to 1:
3. Heapify(A, i, n)

Algorithm 2: Heapify

—----------

Heapify(A, i, n)
Input: An array A, index i, and heap size n
Output: Maintains max-heap property

1. largest = i
2. left = 2 * i
3. right = 2 * i + 1
4. if left ≤ n and A[left] > A[largest]:
5. largest = left
6. if right ≤ n and A[right] > A[largest]:
7. largest = right
8. if largest ≠ i:
9. Swap A[i] and A[largest]
10. Heapify(A, largest, n)

Algorithm 3: Heapsort

—----------

Heapsort(A)
Input: An array A of size n
Output: Sorted array A

1. Build-Max-Heap(A)
2. n = length(A)
3. for i = n down to 2:
4. Swap A[1] and A[i]
5. n = n - 1
6. Heapify(A, 1, n)

(b) Analysis of Heapsort Algorithms

1. Time Complexity

Heapify: For a node at depth dd, Heapify runs in O(d)O(d). Summing over all nodes gives O(logn)O(logn).

Build-Max-Heap: The total cost is O(n)O(n), derived from summing O(logn)O(logn) for all levels of the heap.

Heapsort: Runs O(logn)O(logn) operations nn times, resulting in O(nlogn)O(nlogn). Thus, the overall time complexity of Heapsort is O(nlogn)O(nlogn).

2. Space Complexity

Heapsort is in-place and requires no additional space apart from the input array and a few variables. Thus, the space complexity is O(1)O(1).

3. Stability

Heapsort is not stable, as the relative order of equal elements is not preserved during swapping.

(c) Code oF Heapsort

```
def heapify(arr, n, i):
      """Maintains the max-heap property."""
      largest = i  # Initialize largest as root
      left = 2 * i + 1  # Left child
      right = 2 * i + 2  # Right child

      # Check if left child exists and is greater than root
      if left < n and arr[left] > arr[largest]:
      largest = left

      # Check if right child exists and is greater than largest so far
      if right < n and arr[right] > arr[largest]:
      largest = right

      # If largest is not root, swap and continue heapifying
      if largest != i:
```

```
        arr[i], arr[largest] = arr[largest], arr[i]
        heapify(arr, n, largest)

def build_max_heap(arr):
        """Builds a max-heap from the array."""
        n = len(arr)
        for i in range(n // 2 - 1, -1, -1):
        heapify(arr, n, i)

def heapsort(arr):
        """Sorts the array using Heapsort."""
        n = len(arr)

        # Build a max-heap
        build_max_heap(arr)

        # Extract elements from the heap one by one
        for i in range(n - 1, 0, -1):
        # Move current root to the end
        arr[0], arr[i] = arr[i], arr[0]
        # Call heapify on the reduced heap
        heapify(arr, i, 0)

# Example usage
numbers = [4, 10, 3, 5, 1]
heapsort(numbers)
print("Sorted array:", numbers)
```
