Rushil Gupta December 7, 2024

CS-1204 Problem Set 4

Collaborators: -

Problem 2

1) Build a Max-Heap

To convert the array $[20, 10, 30, 25, 17, 29, 93]$ into a max-heap using the heapify algorithm, we perform the following steps:

1. **Initial Array:**

$$[20,\ 10,\ 30,\ 25,\ 17,\ 29,\ 93]$$

2. **Heapify at Index 2:**
   - Current element: 30
   - Left child (Index 5): 29
   - Right child (Index 6): 93
   - Since $93 > 30$, swap 30 and 93.

$$[20,\ 10,\ 93,\ 25,\ 17,\ 29,\ 30]$$

3. **Heapify at Index 1:**
   - Current element: 10
   - Left child (Index 3): 25
   - Right child (Index 4): 17
   - Since $25 > 10$, swap 10 and 25.

$$[20,\ 25,\ 93,\ 10,\ 17,\ 29,\ 30]$$

4. **Heapify at Index 0:**
   - Current element: 20
   - Left child (Index 1): 25
   - Right child (Index 2): 93

- Since $93 > 20$, swap 20 and 93.

$$[93,\ 25,\ 20,\ 10,\ 17,\ 29,\ 30]$$

5. **Heapify at Index 2 (After Swap):**
   - Current element: 20
   - Left child (Index 5): 29
   - Right child (Index 6): 30
   - Since $30 > 20$, swap 20 and 30.

$$[93,\ 25,\ 30,\ 10,\ 17,\ 29,\ 20]$$

2) Delete Elements

We perform successive delete operations to remove the maximum element from the heap until it is empty. After each deletion, we record the removed element and show the state of the heap.

1. **Initial Heap:**

$$[93,\ 25,\ 30,\ 10,\ 17,\ 29,\ 20]$$

2. **Delete Maximum Element (93):**
   - Swap 93 with the last element 20.
   - Remove 93 from the heap.
   - Heap before heapify:

$$[20,\ 25,\ 30,\ 10,\ 17,\ 29]$$

   - Heapify at Index 0:
     - Current element: 20
     - Left child: 25
     - Right child: 30
     - Swap 20 with 30.

$$[30,\ 25,\ 20,\ 10,\ 17,\ 29]$$

   - Heapify at Index 2:
     - Current element: 20
     - Left child: 29

  - No right child.
  - Swap 20 with 29.

$$[30,\ 25,\ 29,\ 10,\ 17,\ 20]$$

3. Delete Maximum Element (30):

- Swap 30 with the last element 20.
- Remove 30 from the heap.
- Heap before heapify:

$$[20,\ 25,\ 29,\ 10,\ 17]$$

- Heapify at Index 0:
  - Current element: 20
  - Left child: 25
  - Right child: 29
  - Swap 20 with 29.

$$[29,\ 25,\ 20,\ 10,\ 17]$$

4. Delete Maximum Element (29):

- Swap 29 with the last element 17.
- Remove 29 from the heap.
- Heap before heapify:

$$[17,\ 25,\ 20,\ 10]$$

- Heapify at Index 0:
  - Current element: 17
  - Left child: 25
  - Right child: 20
  - Swap 17 with 25.

$$[25,\ 17,\ 20,\ 10]$$

5. Delete Maximum Element (25):

- Swap 25 with the last element 10.
- Remove 25 from the heap.
- Heap before heapify:

$$[10,\ 17,\ 20]$$

- Heapify at Index 0:

- Current element: 10
- Left child: 17
- Right child: 20
- Swap 10 with 20.

$$[20,\ 17,\ 10]$$

6. **Delete Maximum Element (20):**

   - Swap 20 with the last element 10.
   - Remove 20 from the heap.
   - Heap before heapify:

     $$[10,\ 17]$$

   - Heapify at Index 0:
     - Current element: 10
     - Left child: 17
     - Swap 10 with 17.

     $$[17,\ 10]$$

7. **Delete Maximum Element (17):**

   - Swap 17 with the last element 10.
   - Remove 17 from the heap.
   - Heap before heapify:

     $$[10]$$

8. **Delete Maximum Element (10):**

   - Remove 10 from the heap.
   - Heap is now empty.

3) Resulting Array

- **Removed Elements:**

  $$[93,\ 30,\ 29,\ 25,\ 20,\ 17,\ 10]$$

- **Special Property:**

  The resulting array of removed elements is sorted in **descending order**. This is a characteristic outcome of repeatedly deleting the maximum element from a max-heap, effectively performing a heap sort.

- **Time Complexity Analysis:**
  - **Building the Max-Heap:** We perform heapify at $n//2$ nodes. Heapify is a recursive process that takes $O(\log n)$ time for each node. However, since the invariant when we call the heapify function is that the left and right subtrees are already max-heaps, we can say that the time complexity is $O(n)$.
  - **Deleting Elements:** Each deletion operation takes $O(\log n)$ time due to the need to maintain the heap property after removal. Since there are n deletions, this step takes $O(n \log n)$ time.
  - **Overall Time Complexity:** The total time complexity for forming the resulting sorted array from the original heap is $O(n \log n)$.

  Space Complexity: The space complexity is $O(n)$. This is because we store the removed elements in a separate array, which requires additional space proportional to the number of elements n. All heap operations are performed in-place on the original array, requiring $O(1)$ extra space. However, the separate array for removed elements dominates the space usage, resulting in an overall space complexity of $O(n)$.

  However, as an extension, a modification can be made so that the space complexity is $O(1)$. This can be done by storing the deleted elements in last indices of the array itself. But, this would give an array sorted in ascending order. But, to sort the array in descending order, we can use a min-heap instead of a max-heap.