Segment-2

Heap

1. Introduction to Heaps

- **Definition**: A heap is a special tree-based data structure that satisfies the heap property.
    - **Max Heap**: For any given node N, the value of N is greater than or equal to the values of its children.
    - **Min Heap**: For any given node N, the value of N is less than or equal to the values of its children.

Characteristics of Heaps

- Complete binary tree: All levels are fully filled except possibly for the last level, which is filled from left to right.
- Efficiently implemented using arrays.

2. Heap Construction Algorithm

The goal of the Heap Construction Algorithm is to convert an unsorted array into a valid heap. The most common approach to do this is to build a Max Heap (or Min Heap) using the `heapify` procedure.

Steps to Build a Max Heap

1. **Start from the Last Non-Leaf Node**: The last non-leaf node is found at index $\lfloor n/2 \rfloor - 1$, where n is the number of elements in the array.
2. **Heapify the Nodes**: Call the `heapify` function on each node from the last non-leaf node up to the root node (index 0).

The `heapify` Function

The `heapify` function ensures that a subtree rooted at index i maintains the heap property:

1. Compare the node at index I with its children.
2. If the node is smaller than its largest child, swap them.
3. Recursively call `heapify` on the affected child.

Example

Let’s consider an example array:

`Array: [3, 5, 1, 10, 2, 7]`

Step 1: Identify the Last Non-Leaf Node

For our array with 6 elements:

- Last non-leaf node index ⌊6/2⌋−1=2

Step 2: Heapify from the Last Non-Leaf Node to the Root

1. **Heapify at index 2 (value 1)**:
   - Left child index: 5 (value 7)
   - Right child index: N/A (only one child)
   - Largest is 7 (child). Swap 1 and 7.

   Array becomes: `[3, 5, 7, 10, 2, 1]`

   Now, call `heapify` on index 5:

   - No children (end of heap), so no action is needed.
2. **Heapify at index 1 (value 5)**:
   - Left child index: 3 (value 10)
   - Right child index: 4 (value 2)
   - Largest is 10 (left child). Swap 5 and 10.

   Array becomes: `[3, 10, 7, 5, 2, 1]`

   Now, call `heapify` on index 3:

   - No children, so no action needed.
3. **Heapify at index 0 (value 3)**:
   - Left child index: 1 (value 10)
   - Right child index: 2 (value 7)
   - Largest is 10 (left child). Swap 3 and 10.

   Array becomes: `[10, 3, 7, 5, 2, 1]`

   Now, call `heapify` on index 1 (current value 3):

   - Left child index: 3 (value 5)
   - Right child index: 4 (value 2)
   - Largest is 5 (left child). Swap 3 and 5.

   Array becomes: `[10, 5, 7, 3, 2, 1]`

   Now, call `heapify` on index 3:

   - No children, so no action needed.

Final Result

After applying the heapify process from the last non-leaf node up to the root, the array is transformed into a Max Heap:

```
Max Heap: [10, 5, 7, 3, 2, 1]
```

Summary

The Heap Construction Algorithm efficiently builds a heap in O(n) time by applying the `heapify` operation from the bottom up. This ensures that the entire array satisfies the heap property.

Insertion Operation in a Max Heap

A **max heap** is a complete binary tree where the value of each node is greater than or equal to the values of its children. This property allows for efficient access to the maximum element, which is always at the root of the heap.

Steps for Inserting an Element into a Max Heap

1. **Add the Element**: Insert the new element at the end of the heap (the last position in the array representation).
2. **Heapify Up**: Compare the inserted element with its parent. If the inserted element is greater, swap it with the parent. Repeat this process until the max heap property is restored or the element reaches the root.

Example

Let's go through an example step-by-step.

Initial Max Heap

Consider the following max heap represented as an array:

```
Heap: [10, 7, 9, 5, 6, 8]
```

This corresponds to the binary tree:

```
       10
      /  \
    7     9
   / \   / \
  5   6 8
```

Inserting a New Element

Insert the element `12`.

1. **Add the Element**: First, append `12` to the end of the heap:

```
Heap: [10, 7, 9, 5, 6, 8, 12]
```

The binary tree now looks like:

```
       10
      /  \
     7     9
    / \   / \
   5   6 8   12
```

2. **Heapify Up**: Compare `12` with its parent. The parent of `12` (index 7) is `9` (index 3). Since `12 > 9`, we swap them:

```
Heap: [10, 7, 12, 5, 6, 8, 9]
```

The binary tree is now:

```
       10
      /  \
     7     12
    / \   / \
   5   6 8   9
```

3. **Continue Heapifying Up**: Now, compare `12` with its new parent, `10` (index 0). Since `12 > 10`, we swap again:

```
Heap: [12, 7, 10, 5, 6, 8, 9]
```

The final binary tree looks like:

```
       12
      /  \
     7     10
    / \   / \
   5   6 8   9
```

Final Heap

The resulting max heap after inserting `12` is:

```
Heap: [12, 7, 10, 5, 6, 8, 9]
```

Summary

The insertion operation in a max heap involves placing the new element at the end of the heap and then restoring the max heap property through the heapify-up process. The time complexity of this operation is O(logn) due to the potential height of the heap.

Deletion Operation in a Max Heap

In a max heap, the deletion operation typically refers to removing the maximum element, which is always at the root. The process involves a few steps to maintain the heap properties.

Steps for Deleting the Maximum Element from a Max Heap

1. **Remove the Root**: Replace the root of the heap with the last element in the heap (the last element in the array representation).
2. **Heapify Down**: Restore the max heap property by "sifting down" the new root until it is in the correct position.

Example

Let’s go through the deletion process step-by-step.

Initial Max Heap

Consider the following max heap represented as an array:

```
Heap: [15, 10, 14, 8, 7, 9, 13]
```

This corresponds to the binary tree:

```
       15
      /  \
     10   14
    / \   / \
   8   7 9  13
```

Deleting the Maximum Element

Step 1: Remove the Root

The root of the heap is `15`. We replace it with the last element, which is `13`.

```
Heap after replacement: [13, 10, 14, 8, 7, 9]
```

Now the binary tree looks like:

```
       13
      /  \
     10   14
```

```
  / \   / \
 8   7 9
```

Step 2: Heapify Down

Now we need to restore the max heap property. The new root is `13`.

1. Compare `13` with its children `10` and `14`. The larger child is `14`. Since `14 > 13`, we swap them.

```
Heap after swap: [14, 10, 13, 8, 7, 9]
```

The binary tree now looks like:

```
      14
     /  \
    10   13
   / \   / \
  8   7 9
```

2. Now, compare `13` with its new children, which are `9` (left) and no right child. Since `13 > 9`, it is in the correct position, and no further swaps are needed.

Final Heap

The resulting max heap after deleting the maximum element `15` is:

```
Heap: [14, 10, 13, 8, 7, 9]
```

This corresponds to the binary tree:

```
      14
     /  \
    10   13
   / \   / \
  8   7 9
```

Summary

The deletion operation in a max heap involves:

- Removing the root element.
- Replacing it with the last element.
- Restoring the max heap property through the heapify-down process.

The time complexity of the deletion operation is O(logn) due to the potential height of the heap.

3. Heap-Sort

Heap-Sort is an efficient sorting algorithm that uses a binary heap data structure. It consists of two main phases: building a max heap and then sorting the array by repeatedly extracting the maximum element.

Steps of Heap-Sort

1. **Build a Max Heap** from the input array.
2. **Sort the Array**: Repeatedly swap the maximum element (the root of the heap) with the last element and reduce the size of the heap, then heapify the root.

Pseudocode for Heap Sort

1. Heap Sort Function

```
HEAP-SORT(A)
 BUILD-HEAP(A)
 for i ← length(A) down to 2 do
     swap A[1] with A[i]         // Move current root to end
     HEAP-DECREASE-KEY (A, 1, i-1) // Restore heap property
```

2. Build Heap Function

```
BUILD-HEAP(A)
 n ← length(A)
 for i ← ⌊n/2⌋ downto 1 do
     MAX-HEAPIFY(A, i, n)
```

3. Max-Heapify Function

```
MAX-HEAPIFY(A, i, n)
 left ← 2*i
 right ← 2*i + 1
 largest ← i
 if left ≤ n and A[left] > A[largest] then
     largest ← left
 if right ≤ n and A[right] > A[largest] then
     largest ← right
 if largest ≠ i then
    swap A[i] with A[largest]
    MAX-HEAPIFY(A, largest, n)
```

Explanation of the Pseudocode

1. **HEAP-SORT**:
   - Calls `BUILD-HEAP` to transform the array into a max-heap.
   - Iterates from the last element to the second element, swapping the root of the heap with the current element, and then restoring the heap property.
2. **BUILD-HEAP**:
   - Converts the array into a max-heap by calling `MAX-HEAPIFY` on each non-leaf node from the bottom up.
3. **MAX-HEAPIFY**:
   - Ensures that the subtree rooted at index `i` satisfies the max-heap property.

   - Compares the current node with its children and swaps with the largest if necessary, then recurses down the tree.

Summary

This pseudocode follows the method described in Cormen's "Introduction to Algorithms," detailing how to implement Heap Sort using the max-heap structure. The algorithm efficiently sorts an array in O(nlogn) time and performs in-place sorting with O(1) additional space.

Example

Let's sort the array:

```
Array: [3, 5, 1, 10, 2, 7]
```

Step 1: Build a Max Heap

1. **Initial Array**: `[3, 5, 1, 10, 2, 7]`
2. **Heapify Process**:
   - Start from the last non-leaf node: index $\lfloor 6/2 \rfloor = 3$ (considering 1-based indexing).
   - **Heapify at index 3** (value 1):
     - Left child (index 6, value 7).
     - Swap 1 and 7.
     - Array: `[3, 5, 7, 10, 2, 1]`.
   - **Heapify at index 2** (value 5):
     - Left child (index 4, value 10).
     - Swap 5 and 10.
     - Array: `[3, 10, 7, 5, 2, 1]`.
   - **Heapify at index 1** (value 3):
     - Left child (index 2, value 10).
     - Swap 3 and 10.
     - Array: `[10, 3, 7, 5, 2, 1]`.
     - Now heapify at index 2 (current value 3):
       - Left child (index 4, value 5).
       - Swap 3 and 5.
       - Array: `[10, 5, 7, 3, 2, 1]`.

The resulting Max Heap is:

```
Max Heap: [10, 5, 7, 3, 2, 1]
```

Step 2: Sort the Array

1. **First Extraction**:
   - Swap 10 (root) with 1 (last element).
   - Array: `[1, 5, 7, 3, 2, 10]`.
   - Reduce heap size (n = 5) and heapify the root:
     - Swap 1 and 7.
     - Array: `[7, 5, 1, 3, 2, 10]`.

2. **Second Extraction**:
    - Swap 7 with 2.
    - Array: `[2, 5, 1, 3, 7, 10]`.
    - Heapify the root:
        - Swap 2 and 5.
        - Array: `[5, 2, 1, 3, 7, 10]`.
        - Heapify at index 2:
            - Swap 2 and 3.
            - Array: `[5, 3, 1, 2, 7, 10]`.
3. **Third Extraction**:
    - Swap 5 with 1.
    - Array: `[1, 3, 5, 2, 7, 10]`.
    - Heapify the root:
        - Swap 1 and 3.
        - Array: `[3, 1, 5, 2, 7, 10]`.
4. **Fourth Extraction**:
    - Swap 3 with 1.
    - Array: `[1, 3, 5, 2, 7, 10]`.
    - Heapify the root:
        - No swaps needed, array remains `[3, 1, 5, 2, 7, 10]`.
5. **Fifth Extraction**:
    - Swap 3 with 1.
    - Array: `[1, 2, 3, 5, 7, 10]`.

After all extractions, the sorted array is:

```
Sorted Array: [1, 2, 3, 5, 7, 10]
```

Summary

Heap-Sort sorts an array efficiently using the properties of a max heap. Its time complexity is O(nlogn), and it performs in-place with a space complexity of O(1). This makes Heap-Sort a great choice for sorting large datasets.

4. Priority Queue

A **priority queue** is an abstract data type that operates similarly to a regular queue, but with one key difference: each element in a priority queue has a priority associated with it. Elements are dequeued based on their priority rather than their order in the queue.

Key Operations

- **Insert**: Add an element with a specified priority.
- **Extract-Max/Min**: Remove and return the element with the highest (or lowest) priority.
- **Peek**: Return the element with the highest (or lowest) priority without removing it.

Implementation

Priority queues are often implemented using heaps:

- **Max Heap**: For a max-priority queue, the highest priority element is at the root.
- **Min Heap**: For a min-priority queue, the lowest priority element is at the root.

Example

Let's consider an example of a max-priority queue:

1. **Elements to be Inserted**:
    - (Task A, priority 2)
    - (Task B, priority 5)
    - (Task C, priority 1)
    - (Task D, priority 3)
2. **Inserting Elements**:
    - **Insert Task A**: Queue: [(Task A, 2)]
    - **Insert Task B**: Queue: [(Task B, 5), (Task A, 2)]
    - **Insert Task C**: Queue: [(Task B, 5), (Task A, 2), (Task C, 1)]
    - **Insert Task D**: Queue: [(Task B, 5), (Task D, 3), (Task A, 2), (Task C, 1)]

    The queue is structured based on the priority, with the highest priority at the front.

3. **Extracting Elements**:
    - **Extract-Max**: Remove and return Task B (priority 5).
        - Queue after extraction: [(Task D, 3), (Task A, 2), (Task C, 1)]
    - **Extract-Max**: Remove and return Task D (priority 3).
        - Queue after extraction: [(Task A, 2), (Task C, 1)]
    - **Extract-Max**: Remove and return Task A (priority 2).
        - Queue after extraction: [(Task C, 1)]
    - **Extract-Max**: Remove and return Task C (priority 1).
        - Queue is now empty.

Summary

In a priority queue:

- The highest priority elements are served before lower priority ones.
- Insertion and extraction operations are efficient, typically O(logn) when implemented with a heap.
- It's useful in scenarios like task scheduling, Dijkstra's algorithm for shortest paths, and event simulation systems.

5. Complexity Analysis of Related Algorithms

- **Heap Construction**: O(n)
- **Heap-Sort**: O(n log n)
- **Priority Queue Operations**:
    - Insert: O(log n)
    - Extract: O(log n)
    - Peek: O(1)

Summary of Heap Sort: Time and Space Complexity

- **Time Complexity:**
    - Best Case: O(n log n)
    - Average Case: O(n log n)
    - Worst Case: O(n log n)
- **Space Complexity:** O(1) (in-place)

Heap Sort is efficient for large datasets and is particularly useful when memory usage is a concern due to its constant space complexity.

6. Applications of Heaps

- **Heap-Sort**: Efficient sorting algorithm.
- **Priority Queues**: Used in scheduling algorithms, Dijkstra’s algorithm for shortest paths, and more.
- **Median Maintenance**: Efficiently finding the median in a data stream.