CS-204: Design and Analysis of Algorithms

220001016, 220001017, 220001018

January 19, 2024

1 Heap Data Structure

A heap is a fundamental data structure used in various algorithms. There are different types of heaps, each serving specific purposes:

- **Min Heap:** Every node has a value less than or equal to the values of its children.
- **Max Heap:** Every node has a value greater than or equal to the values of its children.
- **Binomial Heap**
- **Fibonacci Heap**

Understanding the structural and heap properties of heaps is essential for analyzing algorithms and designing efficient data structures.

1.1 Structural Property

A heap has two main properties:

- **Complete Binary Tree:** A heap is a complete binary tree. All levels are filled, except possibly the last one, which is filled from left to right.
- **Level Completeness:** If there are l levels in a heap, then levels up to $(l-1)$ are fully filled, and the l-th level is left-filled (may not be completely filled).

1.2 Heap Property

The second property is the heap property, which depends on whether the heap is a min heap or a max heap:

- **Min Heap Property:** In a min heap, the value of each node is less than or equal to the values of its children.
- **Max Heap Property:** In a max heap, the value of each node is greater than or equal to the values of its children.

1.3 Heap Level and Number of Nodes

Table 1: Heap Level and Number of Nodes

Heap Level	**Number of Nodes**
0	2^0
1	2^1
2	2^2
3	2^3
...	...
i	2^i

If there are h levels, then the total number of elements for a heap of height h will be:

$$2^0 + 2^1 + 2^2 + \ldots + 2^{h-1} + 2^h = 2^{h+1} - 1$$

1.4 Insertion Operation

Inserting a new node into a heap is an operation that ensures the heap properties are maintained. The process involves the following steps:

- When inserting a new node, it is initially placed at the last position to preserve the structural property of the complete binary tree.
- However, this simple insertion may violate the heap property, which dictates that each node's value must be less than or equal to its children's values (for a min heap).
- To restore the heap property, a trace-back operation is performed, starting from the newly inserted node. If the heap property is violated, the node is swapped with its parent, and this process continues until the heap property is restored.
- The time complexity for inserting a node into a heap is $O(\log n)$, where n is the total number of nodes. This complexity is based on the logarithmic height (h) of the heap.
- To build a min-heap with n elements, the time complexity is given by the sum of logarithmic values from 1 to n, expressed as:

  $$\log 1 + \log 2 + \ldots + \log n = \log n!$$

  In big-O notation, this is $O(\log n!) \approx O(n \log n)$.
- In the last level of the heap, approximately half the total nodes can be present. Hence, the height for the last level is $\log n$.
- The overall time complexity for insertion is $O(\log n)$, making it an efficient operation for maintaining the min-heap property.

1.5 Heapify

Heapify is a crucial operation in maintaining the heap property within a binary tree. This process ensures that the heap property is satisfied at every node, allowing efficient extraction of the minimum (or maximum) element. Here's a detailed breakdown of the heapify operation:

- To apply heapify at a node, its right and left subtrees should follow the heap structure.
- For a min heap, the procedure involves swapping the root with the smallest child and recursively calling heapify at the swapped child node. This process continues until the heap condition is satisfied at the particular node or when a leaf node is reached.
- The time complexity for the heapify operation is $O(\log n)$, where n is the total number of nodes in the heap.
- When applying heapify to a random array, initiate the process from the root level of the tree or the last element of the array. This is done through a tighter bound to gain insights into the underlying operations.
- The number of operations required at a particular level l is l. Therefore, the total number of operations is given by:

$$\frac{n}{2^1} \cdot 1 + \frac{n}{2^2} \cdot 2 + \ldots + \frac{n}{2^i} \cdot i$$

- Taking $n = 2^i$, we get $i = \log n$. The summation becomes:

$$n \sum_{i=1}^{\log n} \frac{i}{2^i}$$

- Now, utilizing the formula:

$$\sum_{x=0}^{\infty} x^i = \frac{1}{1-x}$$

  Differentiating on both sides with x and multiplying x on both sides:

$$\sum_{i=1}^{\log n} i \cdot x^i = x \cdot \frac{1}{(1-x)^2}$$

  Substitute $x = \frac{1}{2}$:

$$\sum_{i=1}^{\log n} i \cdot \left(\frac{1}{2}\right)^i = \frac{1}{2} \cdot \frac{1}{\left(1-\frac{1}{2}\right)^2}$$

$$\sum_{i=1}^{\log n} i \cdot \left(\frac{1}{2}\right)^i = \frac{1}{2} \cdot \frac{1}{\frac{1}{4}}$$

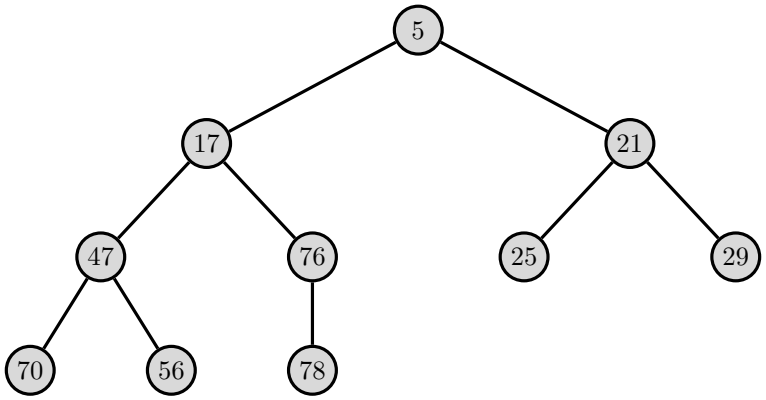
$$\sum_{i=1}^{\log n} i \cdot \left(\frac{1}{2}\right)^i = \frac{1}{2} \cdot 4$$

$$\sum_{i=1}^{\log n} i \cdot \left(\frac{1}{2}\right)^i = 2$$

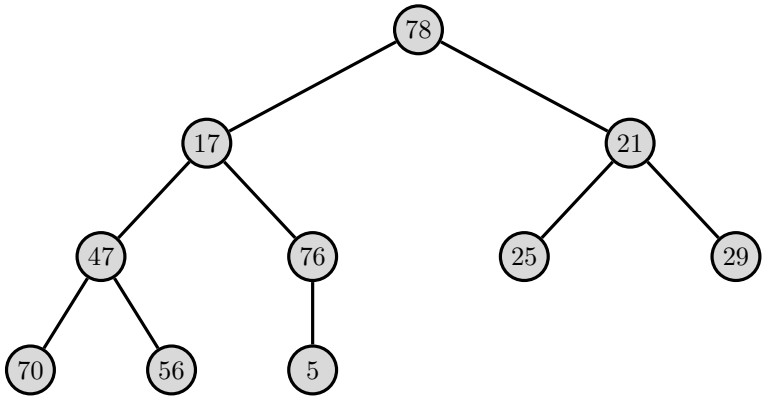
- Therefore, the time complexity to build a min/max heap is $O(n \log(n)))$.

1.6 Delete Minimum Element

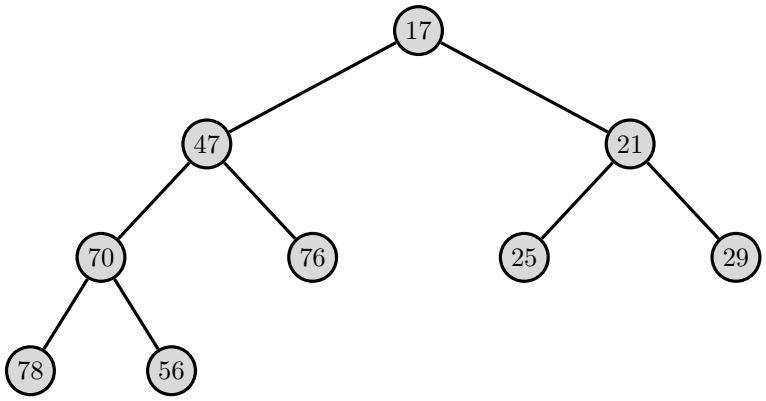
The minimum element in a min heap is always at the root of the heap and the maximum element is always present in one of the leaf node of the heap.



- For deleting the minimum element, swap the root node (i.e, 5) with the last node(i.e, 78) and then apply the heapify operation.



- After applying the heapify operation, the heap will look like this



- Now, we remain with this heap. Again, applying this operation for the other elements, we will get an array of elements (if we represent the heap as an array) that is sorted in descending order, which is

78	76	70	56	47	29	25	21	17	5

1.7 Complexity Analysis

- The time complexity for this action is $O(\log(n))$ where `n` is the number of nodes of the trees
- The time complexity for making the heap is $O(n)$ using bottom-up approach
- Applying all the other operations, the time complexity will be $O(n \log(n))$

This is also the Heapsort algorithm in which if we use the max heap instead of the min heap, then the elements will be sorted in the increasing order.
Hence, the total time complexity for this operation is $O(n \log(n)) + O(n)$, therefore, the resulting total time complexity will be $O(n \log(n))$.

1.7.1 Advantages of Heapsort

- This is an in-place algorithm, hence is better than merge-sort, which has linear space complexity sorting
- The worst case time complexity of this algorithm is $O(n \log(n))$ which is sometimes better than quick sort which has the worst case time complexity as $O(n^2)$ when the elements of the array are reverse sorted.

1.7.2 Still, why it is not used practically?

Due to multiplication with a constant factor, which will determine the ranking in terms of complexity

- For practical uses, Quicksort algorithm is much faster than the Heapsort, because its inner loop can be efficiently implemented on most architectures
- Merge-sort is preferable when merging two arrays and then sorting them in a large database

Heapsort can be more efficient, if we use ternary tree or use m^{th} order tree. Because as the number of children will increase, the height of the tree will decrease.