Heap/Priority Queue

Presented by Yasin Ceran

March 7, 2024

Introduction to Heaps

- A heap is a complete binary tree used to implement a Priority Queue.
- Unlike standard queues (FIFO), priority queues remove elements based on priority.
- Heaps are categorized into two types:

Heap Types

- **Min Heap:** Smallest value at the root; smallest value has the highest priority.
- **Max Heap:** Largest value at the root; largest value has the highest priority.

- Focus: Min heaps (Max heaps follow similar principles).

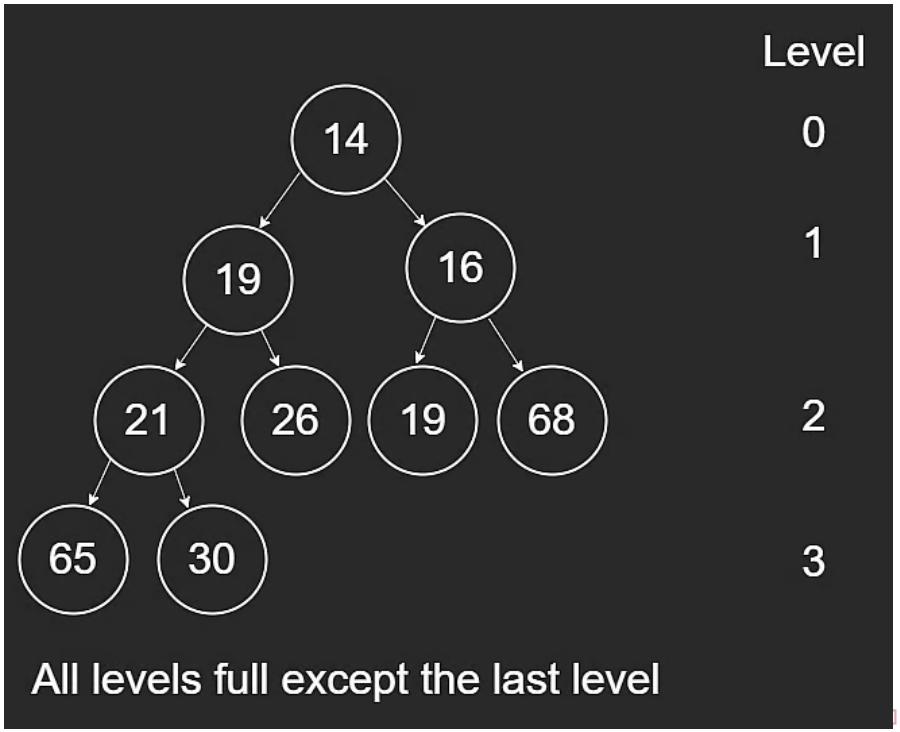
Heap Properties

Structure Property:

- Must be a complete binary tree.
- Levels are filled from left to right without gaps.

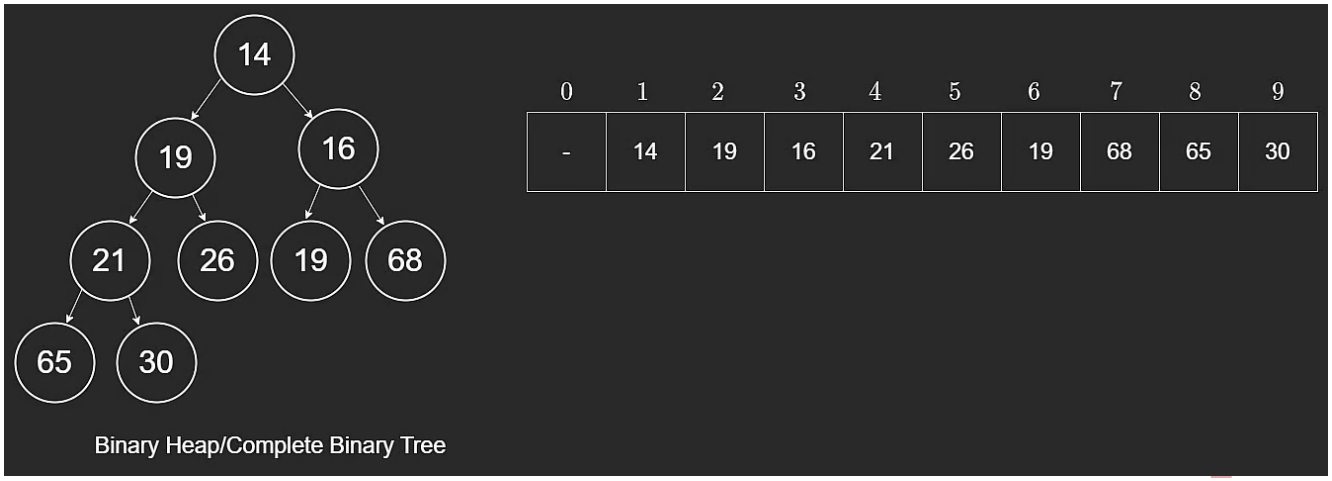
Order Property:

- Min Heap: All descendants greater than the root.
- Max Heap: All descendants smaller than the root.

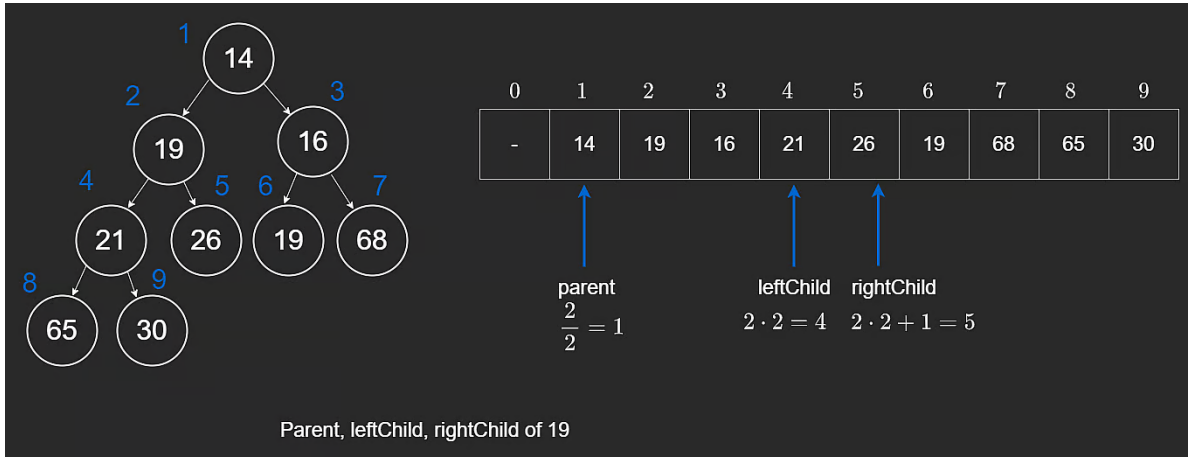


Binary Heap Implementation Using Arrays-1

- Binary heaps, though conceptually represented as trees, are implemented using arrays for efficiency.
- **Example Heap:**
  $[14, 19, 16, 21, 26, 19, 68, 65, 30, \text{null}, \text{null}, \text{null}, \text{null}, \text{null}, \text{null}]$
- We use an array of size $n + 1$, where n is the number of nodes in the binary heap.
- Nodes are inserted into the array based on their level-order (Breadth-First Search order):
  - The array is filled level by level, from left to right.
  - This ensures that the complete binary tree property is maintained.



Node Relationships in a Binary Heap



- **Why Indexing Starts at 1:**
  - Starting at index 1 simplifies the calculation for finding a node's left child, right child, and parent.
  - For a node at index i:
    - Left Child Index: $2 \times i$
    - Right Child Index: $2 \times i + 1$
    - Parent Index: $i/2$

- **Example:** If we want to find the children of node 14 (index 1):
  - Left Child (index $2 \times 1 = 2$) is 19.
  - Right Child (index $2 \times 1 + 1 = 3$) is 16.
  - This demonstrates the utility of starting indexing from 1 for a complete binary tree.

Maintaining Heap Properties

Maintaining Heap Properties

- **Maintaining Min-Heap Properties:** Whenever operations such as add or remove are performed on the heap, it's crucial to ensure that the min-heap properties are preserved.
- This involves adjusting the heap to maintain the structural and order properties, ensuring the smallest value remains at the root.
- The integrity of the parent-child relationship, defined by the index formulas, must also be preserved.

```
class Heap:
    heap = a list of integers, objects etc.
    constructor():
        heap = intialize the list
        heap.add(0)
```

Concept

Push Operation in Heaps

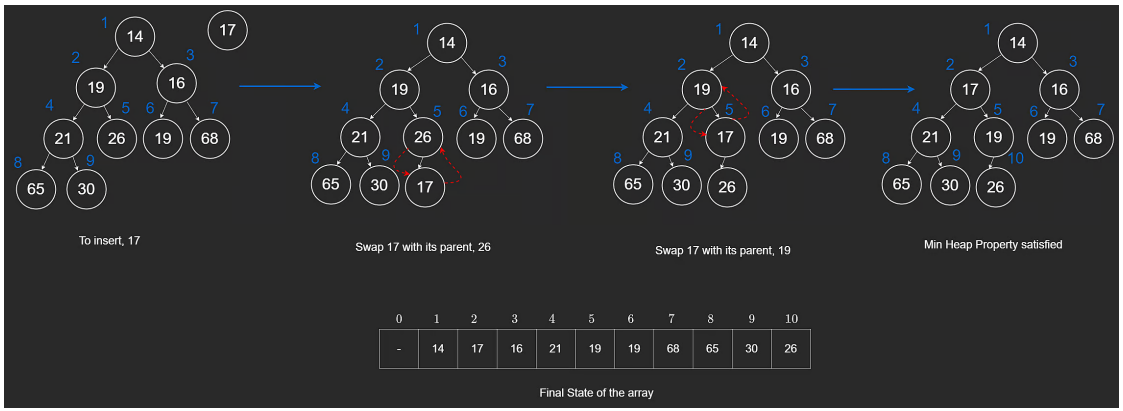
- To insert a new element into a heap (push operation), we must ensure that the heap's structural and order properties are maintained.
- The element is initially added at the next available position to keep the tree complete.
- We then adjust the heap to restore the heap property (percolate up).

Example

Example

Pushing 17 into the Heap Consider the binary heap:
[14, 19, 16, 21, 26, 19, 68, 65, 30, null, null, null, null, null, null]

- We wish to insert 17.
- 17 is added at the 10th index, creating a need to adjust the heap to maintain the min-heap property.
- Since $17 < 26$, 17 swaps with 26.
- 17 now compares with 19, its new parent, and since $17 < 19$, another swap occurs.
- The process continues until 17 is in a position where it is greater than its parent and smaller than or equal to its children, maintaining the min-heap property.
- The heap now properly reflects the insertion of 17, maintaining the complete tree structure and the min-heap order property.



Example Pseudo code

```
def push(self, val):
    self.heap.append(val)
    i = len(self.heap) - 1

    # Percolate up
    while i > 1 and self.heap[i] < self.heap[i // 2]:
        tmp = self.heap[i]
        self.heap[i] = self.heap[i // 2]
        self.heap[i // 2] = tmp
        i = i // 2
```

- The "//" indicates taking the floor of the resulting answer so we can round down, e.g. 5.5 becomes 5 since indices are whole numbers.
- Since we know the tree will always be balanced, the time complexity of the push operation is O(log n)

Pop Operation in Heaps

- Popping the root element from a heap involves more steps than pushing a new element due to the need to maintain both the heap's structure and order properties.
- Simply replacing the root with one of its children can violate the heap's structure property.

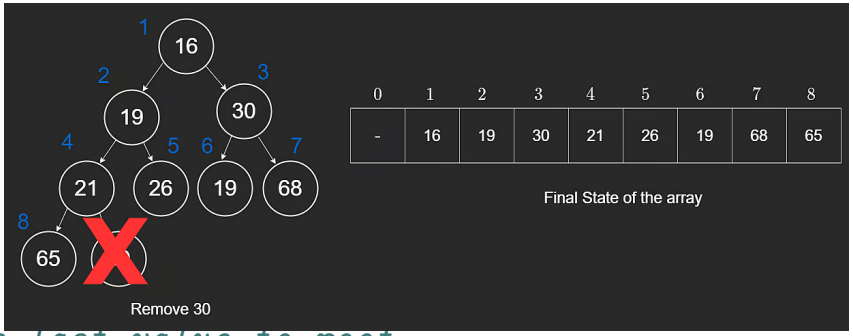
The Incorrect Way to Pop

- An intuitive but incorrect method might involve replacing the root node with the minimum of its two children.
- This approach disrupts the heap's complete binary tree structure, leading to an incomplete level.

The Correct Method for Pop Operation

- The correct approach involves replacing the root node with the right-most node of the last level, maintaining the structure property.
- To restore the order property, the new root is then swapped down the tree with the minimum of its left and right children until the heap order is restored.

Pop Example and Pseudo Code




```
def pop(self):
    if len(self.heap) == 1:
        return None
    if len(self.heap) == 2:
        return self.heap.pop()

    res = self.heap[1]
    self.heap[1] = self.heap.pop() # Move last value to root
    i = 1
    while 2 * i < len(self.heap): # Percolate down
        if (2 * i + 1 < len(self.heap) and
        self.heap[2 * i + 1] < self.heap[2 * i] and
        self.heap[i] > self.heap[2 * i + 1]):
            tmp = self.heap[i] # Swap right child
            self.heap[i] = self.heap[2 * i + 1]
            self.heap[2 * i + 1] = tmp
            i = 2 * i + 1
        elif self.heap[i] > self.heap[2 * i]:
            tmp = self.heap[i] # Swap left child
            self.heap[i] = self.heap[2 * i]
            self.heap[2 * i] = tmp
            i = 2 * i
        else:
            break
    return res
```

Time Complexity of Heap Operations

Operation	Big-O Time
Get Min/Max	$O(1)$
Push	$O(\log n)$
Pop	$O(\log n)$

Table: Summary of the time complexity for key heap operations.

- The **Get Min/Max** operation benefits from the heap's property of having the minimum or maximum element at the root, allowing for constant-time retrieval.
- Both **Push** and **Pop** operations involve maintaining the heap structure and order properties, leading to logarithmic time complexity due to the heap's balanced nature.

Heapify: Building a Heap Efficiently

- Building a binary heap from n elements can be optimized beyond the $O(n \log n)$ time complexity of inserting elements one by one.
- **Heapify** offers a more efficient approach, allowing for heap construction in $O(n)$ time.

Concept:

- The goal is to ensure the heap is a complete binary tree and maintain the heap order property.
- Since leaf nodes inherently satisfy heap properties, heapify focuses on non-leaf nodes.

Implementation Insight:

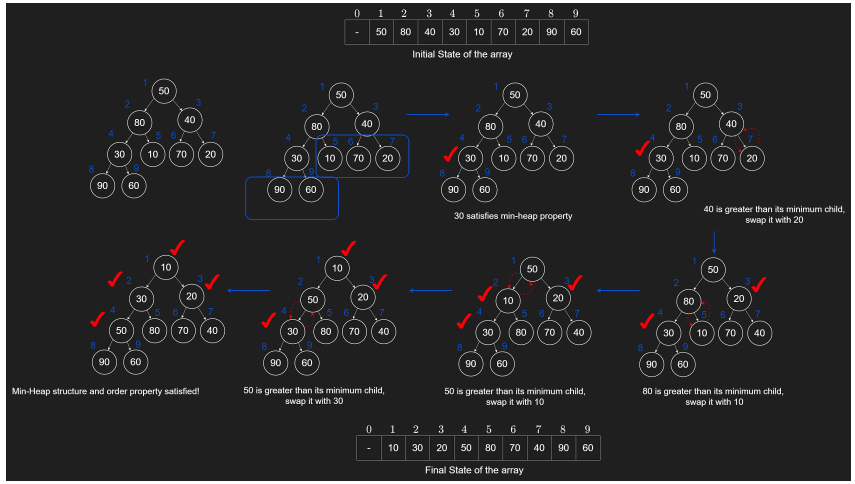
- Start at heap.length//2 to skip leaf nodes.
- Apply a "percolate down" process similar to the pop operation to adjust each node.
- This bottom-up approach efficiently transforms an unordered array into a heap.

- The efficiency of heapify stems from minimizing the number of comparisons and swaps needed to build the heap structure from an arbitrary list of elements.

Pseudo Code

```
def heapify(self, arr):
    # 0-th position is moved to the end
    arr.append(arr[0])

    self.heap = arr
    cur = (len(self.heap) - 1) // 2
    while cur > 0:
        # Percolate down
        i = cur
        while 2 * i < len(self.heap):
            if (2 * i + 1 < len(self.heap) and
            self.heap[2 * i + 1] < self.heap[2 * i] and
            self.heap[i] > self.heap[2 * i + 1]):
                # Swap right child
                tmp = self.heap[i]
                self.heap[i] = self.heap[2 * i + 1]
                self.heap[2 * i + 1] = tmp
                i = 2 * i + 1
            elif self.heap[i] > self.heap[2 * i]:
                # Swap left child
                tmp = self.heap[i]
                self.heap[i] = self.heap[2 * i]
                self.heap[2 * i] = tmp
                i = 2 * i
            else:
                break
        cur -= 1
```



Time Complexity of Heapify

- **Understanding the Complexity:**
  - A binary tree with n nodes has $\approx n/2$ leaf nodes.
  - Heapify skips leaf nodes, focusing on $n/2$ non-leaf nodes.
  - Non-leaf nodes percolate down varying levels, with deeper nodes percolating fewer levels.
- **Analysis:**
  - The work required decreases as we move up the tree, balancing the increased work of nodes percolating down from higher levels.
  - This distribution of work across the tree levels results in an overall time complexity of $O(n)$.
- Knowing the specific mathematical derivation of the $O(n)$ complexity is less critical than understanding that Heapify efficiently transforms an unordered array into a heap in linear time.

Summary: Heap Properties

- **Heap Types:** Two primary heap structures are min heaps and max heaps.
- **Min Heap:** Smallest value at the root; prioritizes lower values.
- **Max Heap:** Largest value at the root; prioritizes higher values.
- **Properties:**
  - *Structure Property:* Complete binary tree; filled level by level.
  - *Order Property:* For min heaps, parent nodes are less than children; for max heaps, parent nodes are greater than children.
- **Implementation:** Typically implemented as arrays for efficiency.

Summary: Push and Pop Operations

- **Push (Insertion):**
  - Added at the next available position to maintain the complete tree structure.
  - Percolate up to maintain heap order; $O(\log n)$ complexity.
- **Pop (Removal):**
  - Root is removed; replaced by the right-most leaf to maintain structure.
  - Percolate down to restore order; $O(\log n)$ complexity.
- Ensuring both operations maintain the heap's structure and order properties is crucial.

Summary: Heapify Process

- **Heapify:** Efficient algorithm to build a heap from an unordered array.
- **Objective:** Transform into a heap maintaining structure and order properties in $O(n)$.
- **Process:**
  - Start from the last non-leaf node $(n/2)$ and percolate down.
  - Adjusts both structure and order without individual insertions.
- **Efficiency:** Achieves linear time complexity, $O(n)$, more efficient than $O(n \log n)$ for successive insertions.
- Heapify underpins many heap operations and algorithms, providing a foundation for efficient priority queue management.