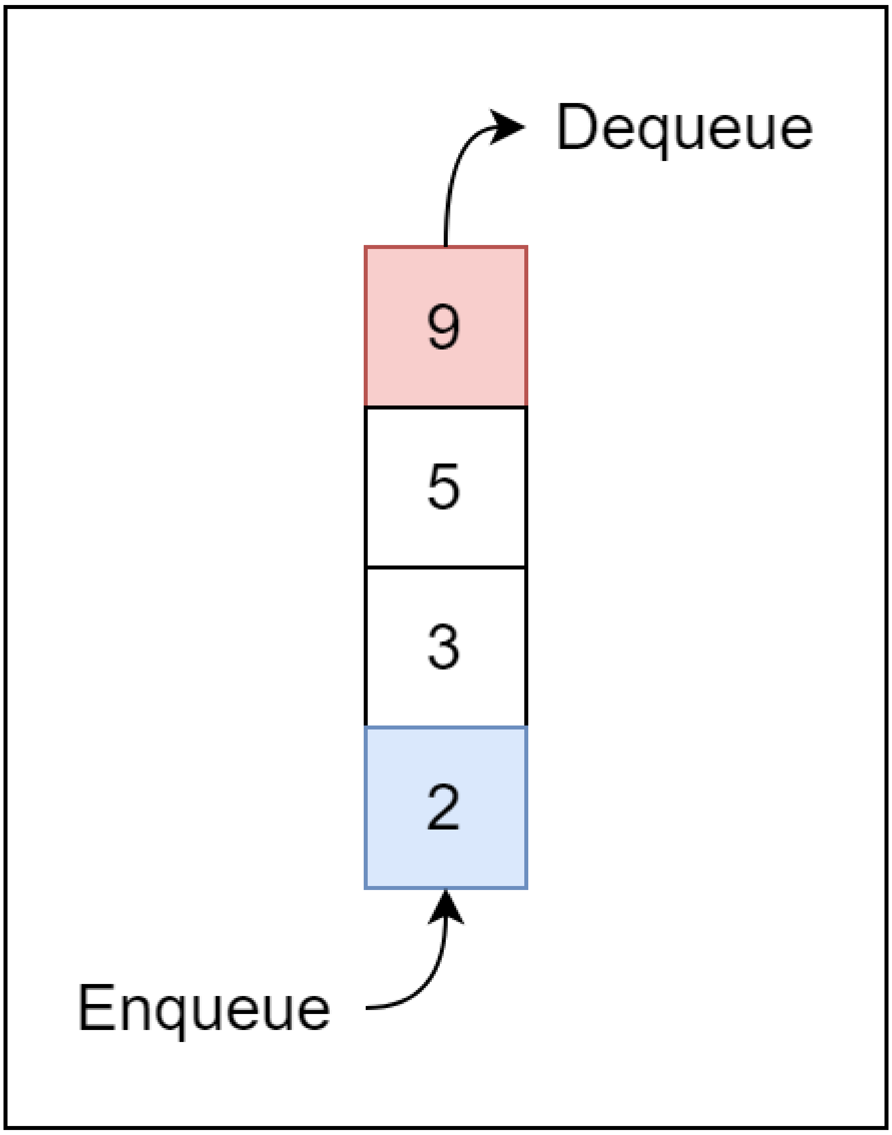
Priority Queue

Alright, let's talk about something called a priority queue. Imagine it as a special type of line where each person has a priority number. The cool thing is that the person with the highest priority gets served first.

But, if two people have the same priority, then they are served based on who arrived first in the line.

Now, how do we decide these priorities? Well, usually, we use the value of the thing itself. For instance, if we're dealing with numbers, the bigger the number, the higher the priority. But sometimes, we might flip that around and say the smaller number has a higher priority.

And the great thing is, you can set these priorities based on whatever you need for your task. It's like giving importance to different tasks in your to-do list.



Difference between Priority Queue and Normal Queue

Now, let's talk about the difference between a normal queue and a priority queue.

So, in a regular queue, the rule is "first come, first served." Imagine standing in line at a food truck - the first person in line gets their order first, then the next, and so on.

But in a priority queue, it's a bit different. Here, the important thing is the priority attached to each item. The one with the highest priority gets served before the others. It's like a special line for VIPs. So, even if someone arrived later, if they have higher priority, they get served first.

Implementation of Priority Queue

Alright, let's talk about how we can actually put this priority queue concept into action!

You can implement a priority queue using different tools, like an array, a linked list, a heap data structure, or even a binary search tree. But among these options, the heap data structure really shines when it comes to efficiency.

So, in this tutorial, we'll focus on using the heap data structure to build our priority queue. Specifically, we'll use something called a max-heap. If you're curious about how max-heaps work, you can learn more about them by checking out materials on max-heap and min-heap.

Now, if we compare different implementations of the priority queue, you'll see some differences in how they handle operations like peeking (checking the top element), inserting, and deleting.

Operations	peek	insert	delete
Linked List	`O(1)`	`O(n)`	`O(1)`
Binary Heap	`O(1)`	`O(log n)`	`O(log n)`
Binary Search Tree	`O(1)`	`O(log n)`	`O(log n)`

So, when we consider all these factors, using a binary heap for our priority queue seems like a great choice due to its balance between efficiency and ease of implementation.

Priority Queue Operations

Alright, let's dive into the essential actions we can perform with a priority queue!

So, a priority queue involves three main operations: inserting new elements, removing elements, and peeking at elements.

Now, before we delve deeper into how a priority queue works, I'd recommend taking a look at the heap data structure. It'll give you a solid foundation on binary heaps, which we use to build and understand priority queues in this discussion. This knowledge will really help you grasp the concepts better.

---

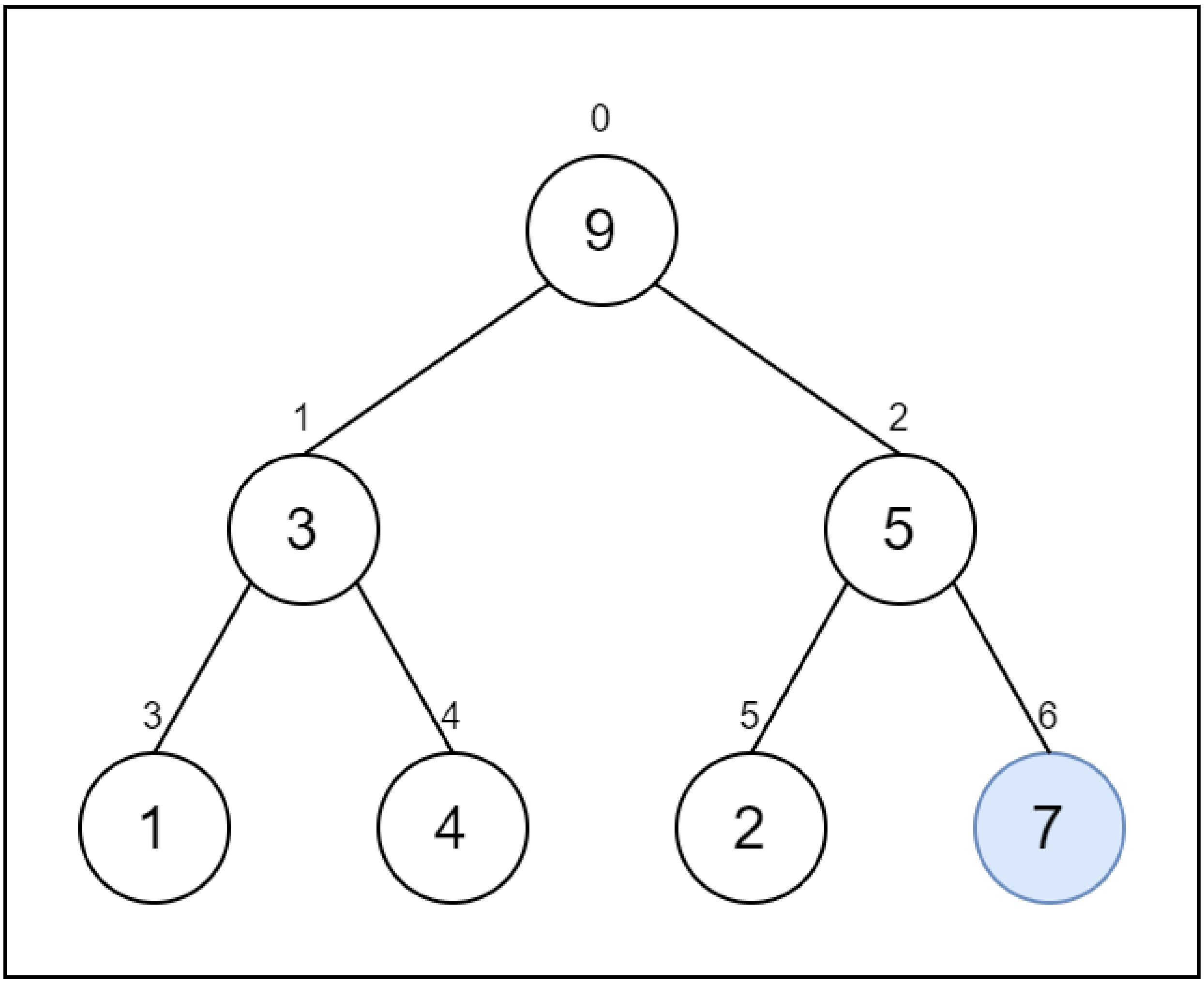
1. Inserting an Element into the Priority Queue

Let's now learn how to add an element to a priority queue, particularly a max-heap. This process involves a couple of simple steps.

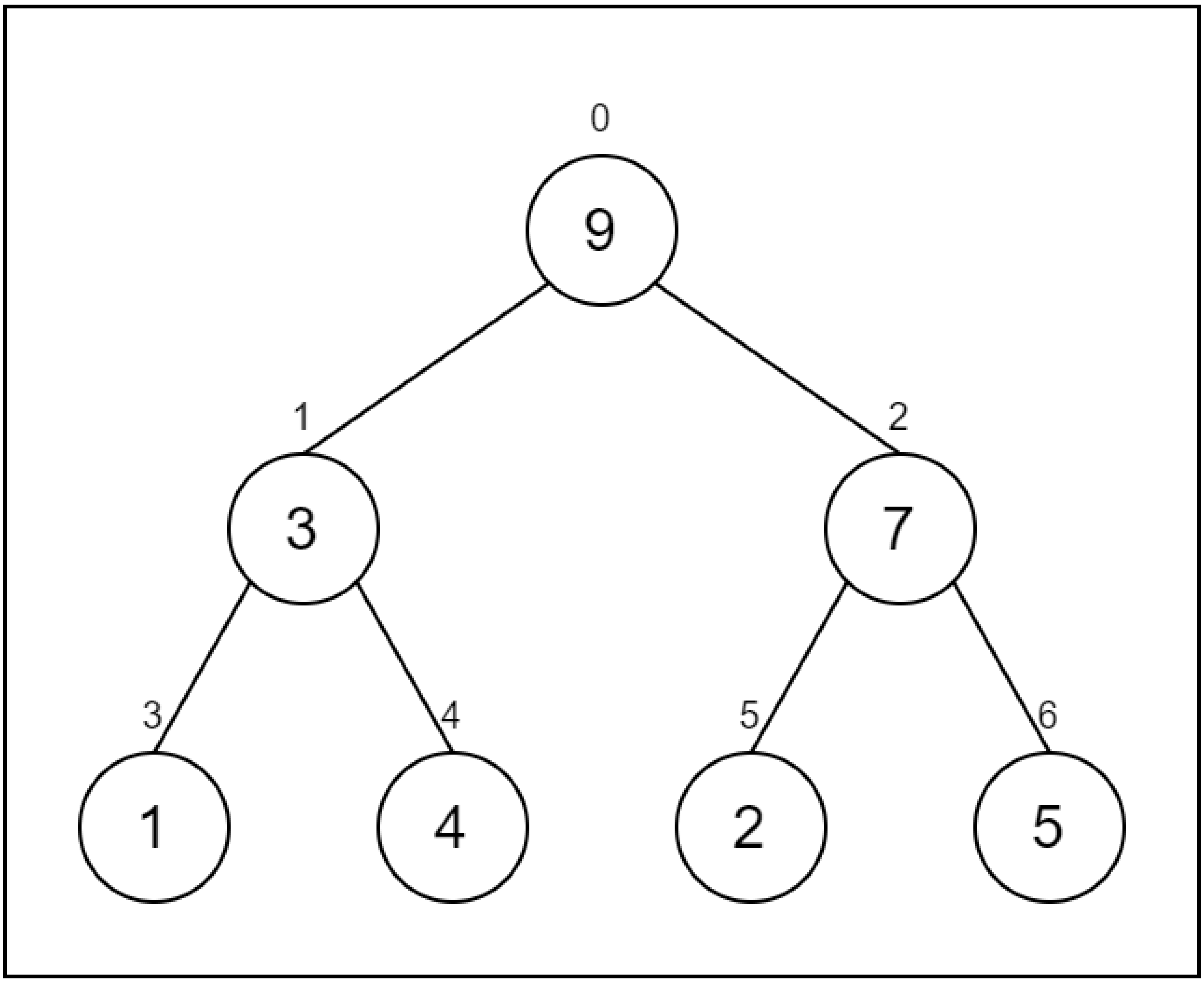
First, we insert the new element at the end of the tree. Then, we perform a step known as "heapify." This essentially ensures that the heap property is maintained, meaning the parent node should be larger (for max-heap) or smaller (for min-heap) than its child nodes.

Here's the algorithm for inserting an element into a priority queue (max-heap):

- If there's no node, create a new node.
- If there are already nodes, insert the new node at the end, which is the last position from left to right.



- Perform heapify on the array to rearrange the elements and maintain the heap structure.



```
If there is no node,
  create a newNode.
else (a node is already present)
  insert the newNode at the end (last node from left to right.)

heapify the array
```

Remember, for a min-heap, we adjust the algorithm slightly so that the parent node is always smaller than the new node. This ensures the proper order in the heap.

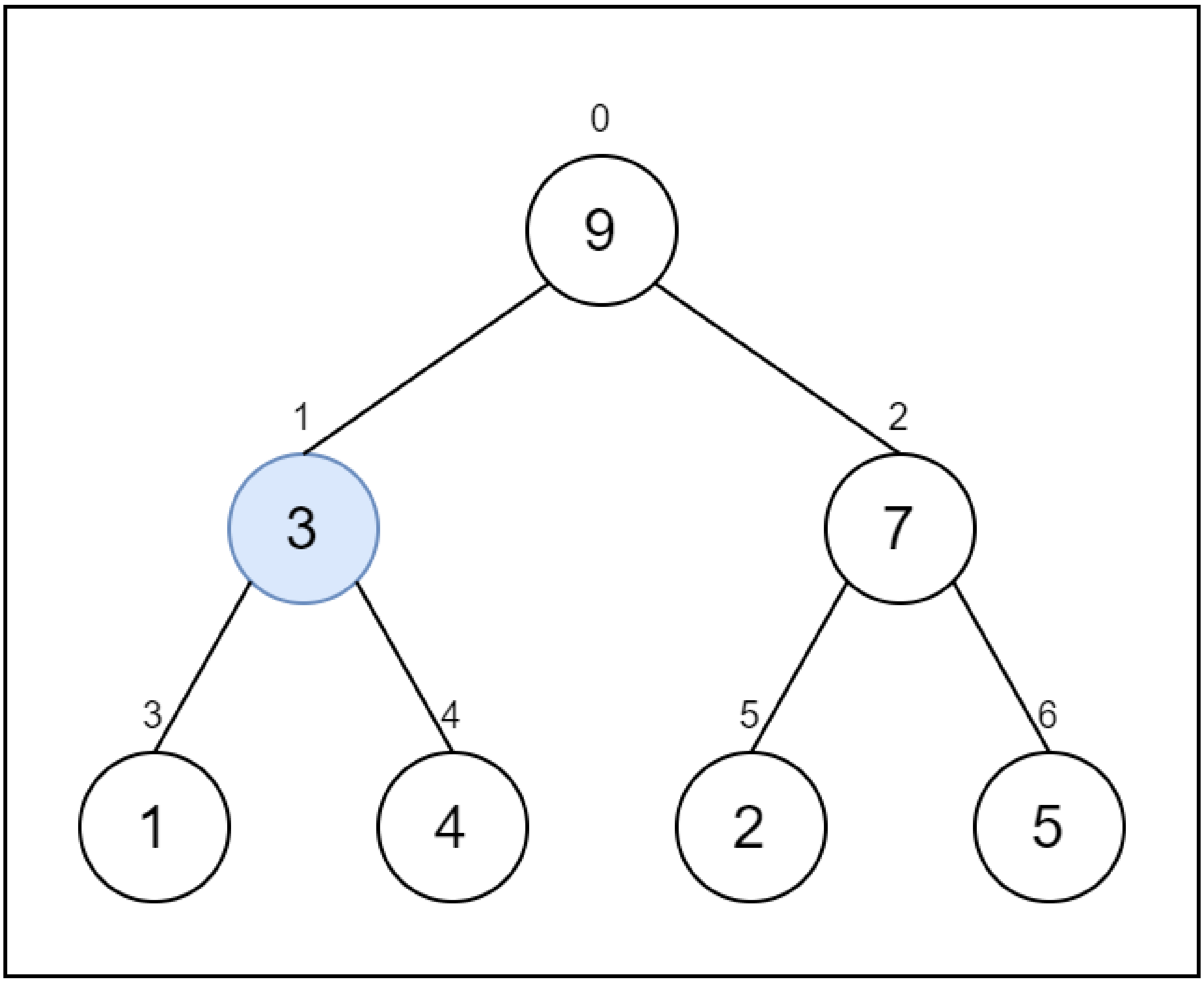
---

2. Deleting an Element from the Priority Queue

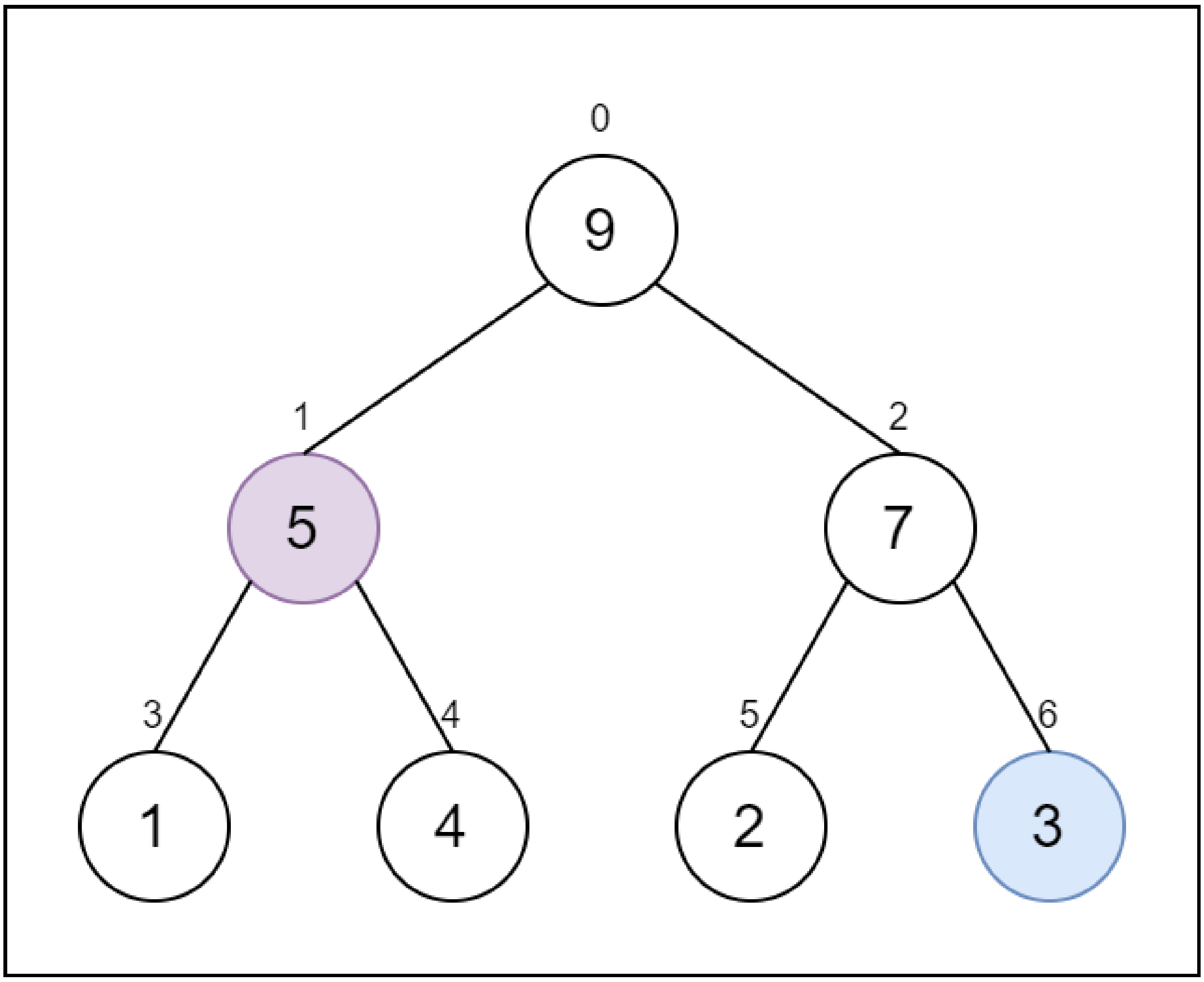
Now, let's delve into the process of removing an element from a priority queue, particularly a max-heap. This involves a series of steps to ensure that the heap properties are maintained.

Here's how you delete an element from a priority queue (max-heap):

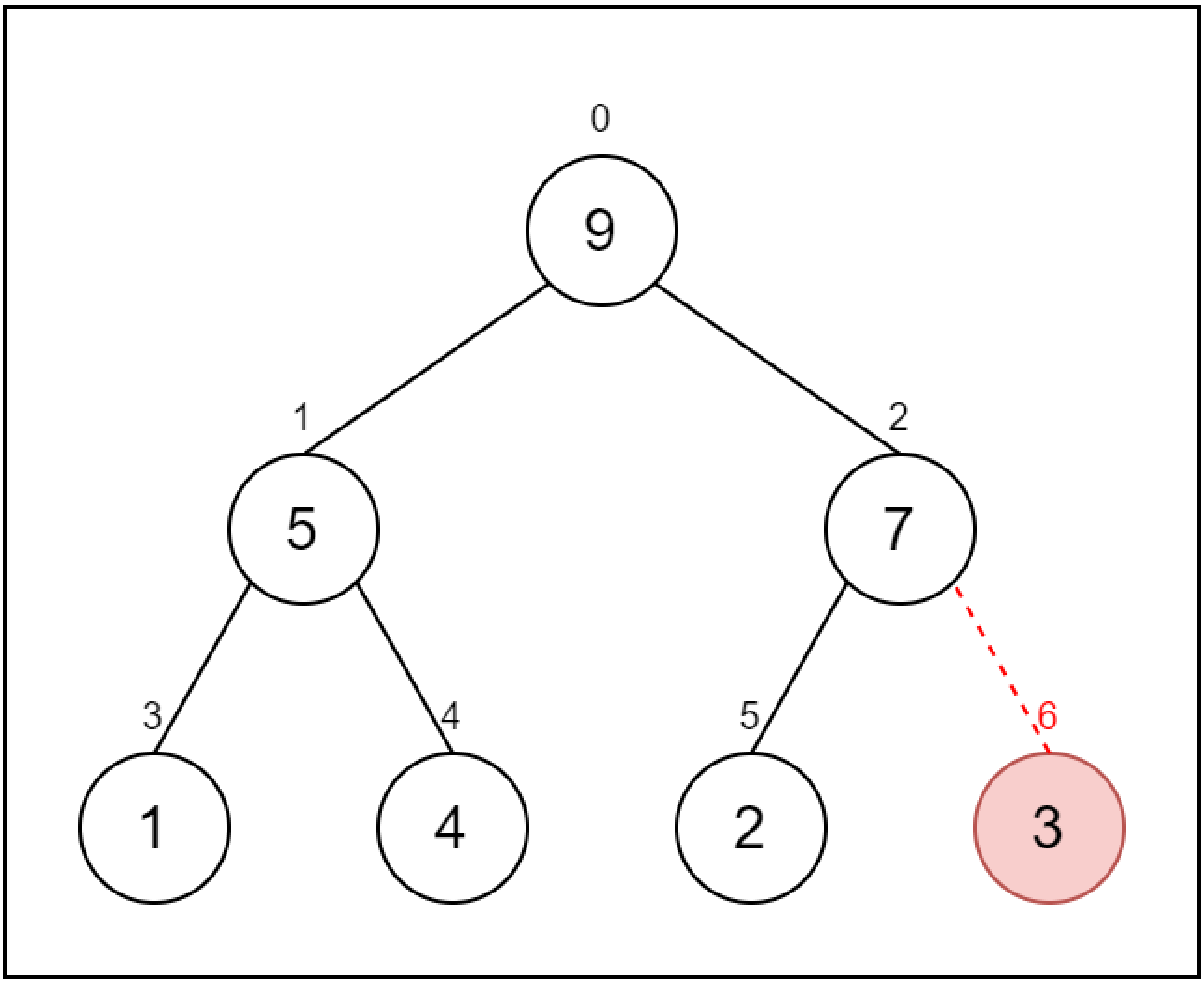
1. First, you identify the element you want to delete.



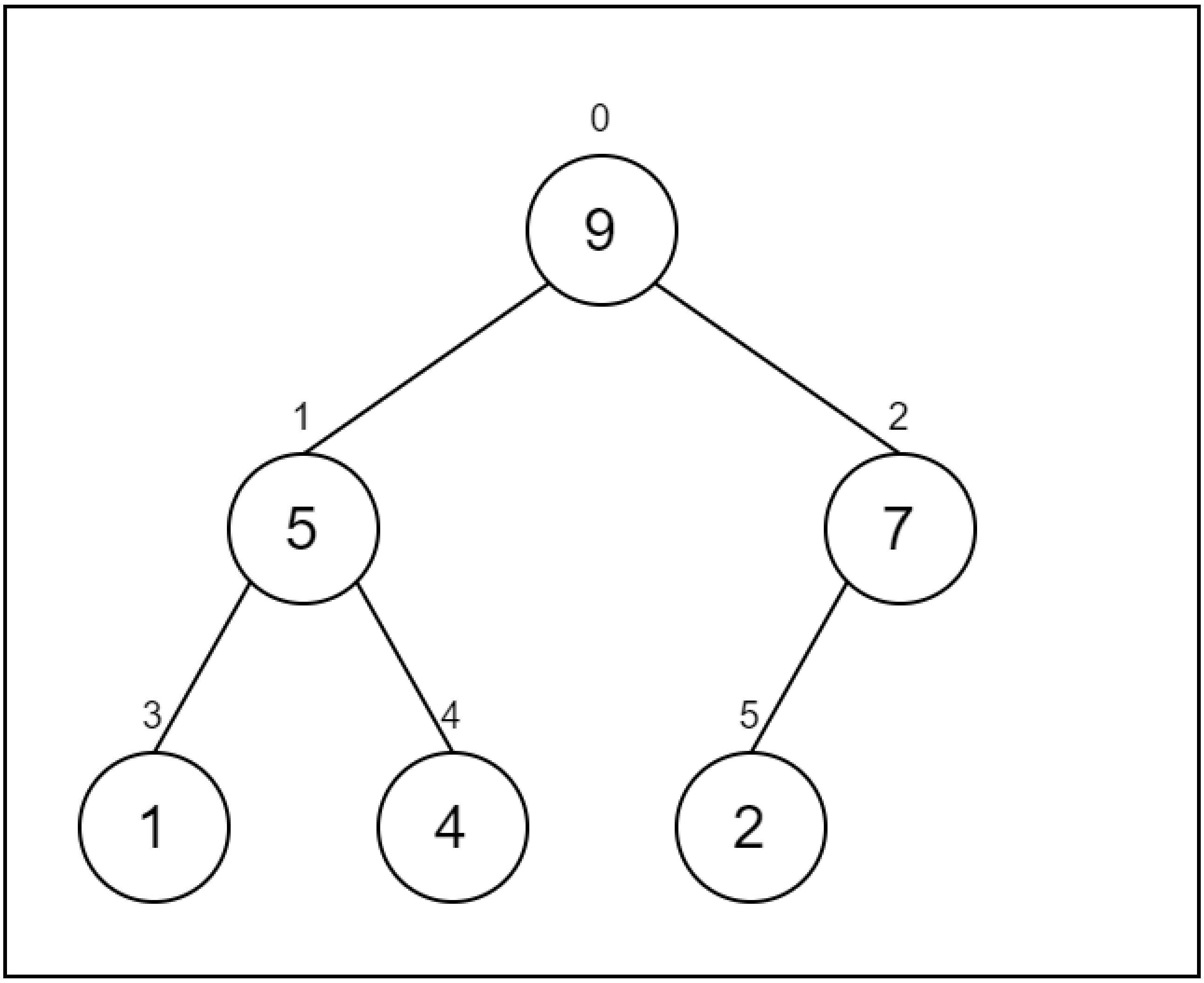
2. Then, you swap this element with the last element in the heap.



3. After the swap, you remove the last element, effectively "extracting" the element you intended to delete.



4. Finally, you perform heapify to reorganize the heap and maintain the proper order.



Here's the algorithm to delete an element from a priority queue (max-heap):

- If the node to be deleted is a leaf node, simply remove it.
- If not, swap the node to be deleted with the last leaf node in the heap.
- After swapping, remove the node you wanted to delete.
- Lastly, perform heapify on the array to rearrange the elements correctly.

```
If nodeToBeDeleted is the leafNode
  remove the node
Else swap nodeToBeDeleted with the lastLeafNode
  remove noteToBeDeleted

heapify the array
```

Remember, for a min-heap, we modify the algorithm so that both child nodes are smaller than the current node, ensuring the appropriate structure of the heap.

3. Peeking from the Priority Queue (Find max/min)

Now, let's talk about the "peek" operation, which allows us to take a look at the topmost element of a priority queue without actually removing it. This is helpful when you want to know the maximum or minimum value without altering the order of the queue.

Here's how the "peek" operation works:

- When you perform a peek operation on a max-heap, it will return the maximum element present in the heap.
- Similarly, when you perform a peek operation on a min-heap, it will give you the minimum element.

The process is simple:

```
return rootNode
```

By returning the root node, which is the highest or lowest priority element, depending on the type of heap, you get a quick way to access this significant value without any changes to the heap's structure.

---

4. Extract-Max/Min from the Priority Queue

Now, let's explore the "Extract-Max" and "Extract-Min" operations in a priority queue, specifically in the context of max-heaps and min-heaps.

When you perform the "Extract-Max" operation on a max-heap, it retrieves the node with the maximum value from the heap and removes it. Similarly, the "Extract-Min" operation on a min-heap does the same but for the minimum value.

Here's a simple way to understand it:

- With "Extract-Max," you take out the highest priority element (the one with the largest value) from a max-heap.
- With "Extract-Min," you do the same but from a min-heap, extracting the lowest priority element (the one with the smallest value).

In both cases, the extracted element is removed from the heap, and the structure of the heap is adjusted accordingly to maintain its properties.

Priority Queue Implementations in C++

```python
# Priority Queue implementation in Python


# Function to heapify the tree
def heapify(arr, n, i):
    # Find the largest among root, left child and right child
    largest = i
    l = 2 * i + 1
    r = 2 * i + 2

    if l < n and arr[i] < arr[l]:
        largest = l

    if r < n and arr[largest] < arr[r]:
        largest = r

    # Swap and continue heapifying if root is not largest
    if largest != i:
        arr[i], arr[largest] = arr[largest], arr[i]
        heapify(arr, n, largest)


# Function to insert an element into the tree
def insert(array, newNum):
    size = len(array)
    if size == 0:
        array.append(newNum)
    else:
        array.append(newNum)
        for i in range((size // 2) - 1, -1, -1):
            heapify(array, size, i)


# Function to delete an element from the tree
def deleteNode(array, num):
    size = len(array)
    i = 0
    for i in range(0, size):
        if num == array[i]:
            break

    array[i], array[size - 1] = array[size - 1], array[i]

    array.remove(size - 1)

    for i in range((len(array) // 2) - 1, -1, -1):
        heapify(array, len(array), i)
```

```python
arr = []

insert(arr, 3)
insert(arr, 4)
insert(arr, 9)
insert(arr, 5)
insert(arr, 2)

print ("Max-Heap array: " + str(arr))

deleteNode(arr, 4)
print("After deleting an element: " + str(arr))
```

Priority Queue Applications

Let's take a look at some practical ways priority queues are used in various applications:

1. **Dijkstra's Algorithm**: Priority queues are a key component in Dijkstra's algorithm, a method used in finding the shortest path between nodes in a graph. The algorithm relies on the concept of choosing the next node with the lowest distance value, which aligns with the priority queue's functionality.
2. **Implementing Stack**: Believe it or not, priority queues can be used to build stacks. By assigning priority values to elements and extracting them according to their priorities, you can effectively create a stack-like behavior where the most "important" element is accessed first.
3. **Load Balancing and Interrupt Handling**: In operating systems, priority queues help manage tasks efficiently. For instance, in load balancing, processes are assigned priorities to distribute the computing load evenly. Similarly, for interrupt handling, where hardware signals need prompt attention, priority queues ensure that the most critical tasks are addressed first.
4. **Data Compression using Huffman Code**: Priority queues are essential in creating Huffman codes, a technique for data compression. In Huffman coding, characters are assigned varying-length codes based on their frequency of occurrence. Priority queues come into play during the process of constructing the code tree.

By understanding these applications, we can see how priority queues play a crucial role in optimizing various computational tasks and improving the overall efficiency of systems.