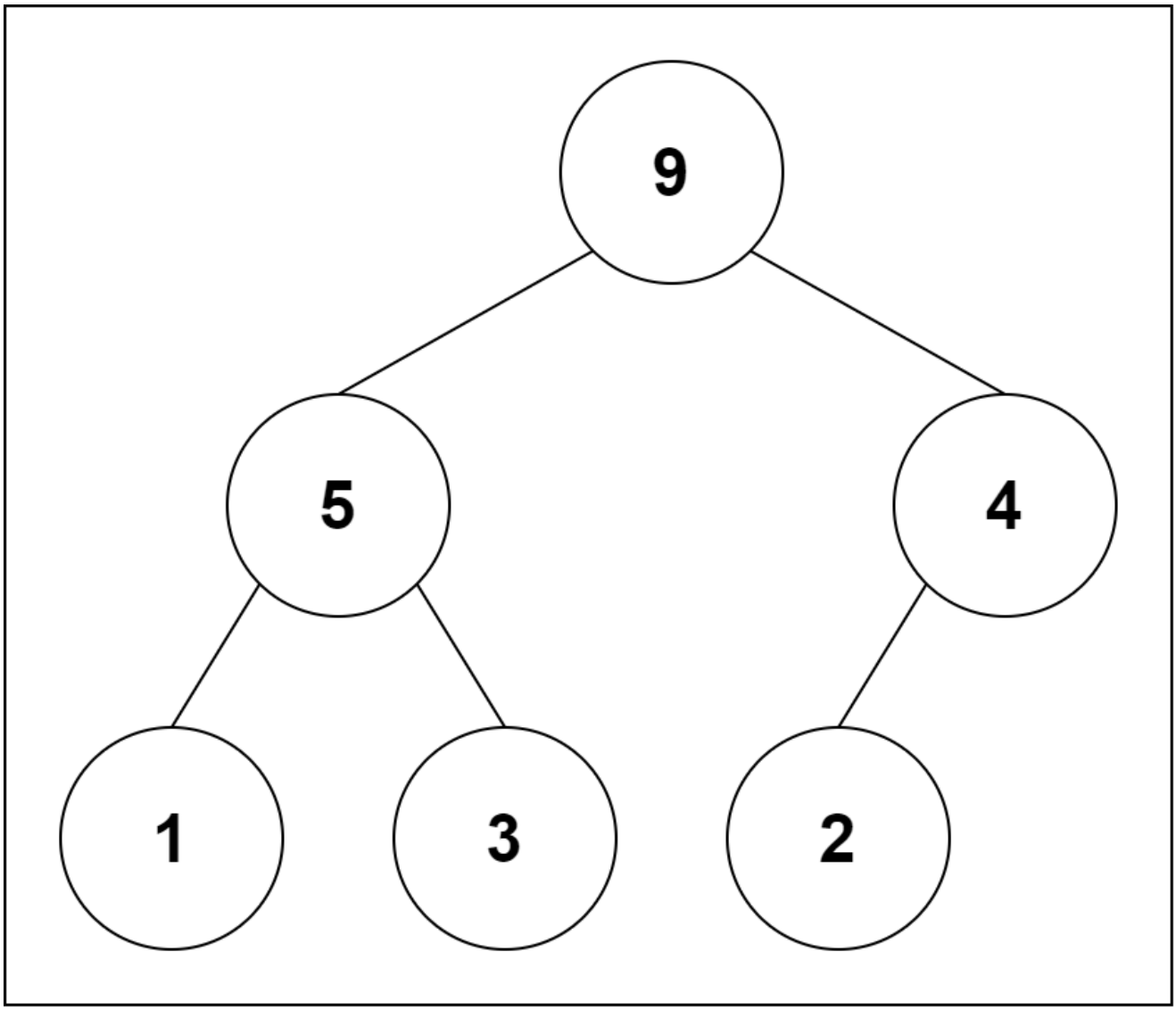
Heap Data Structure

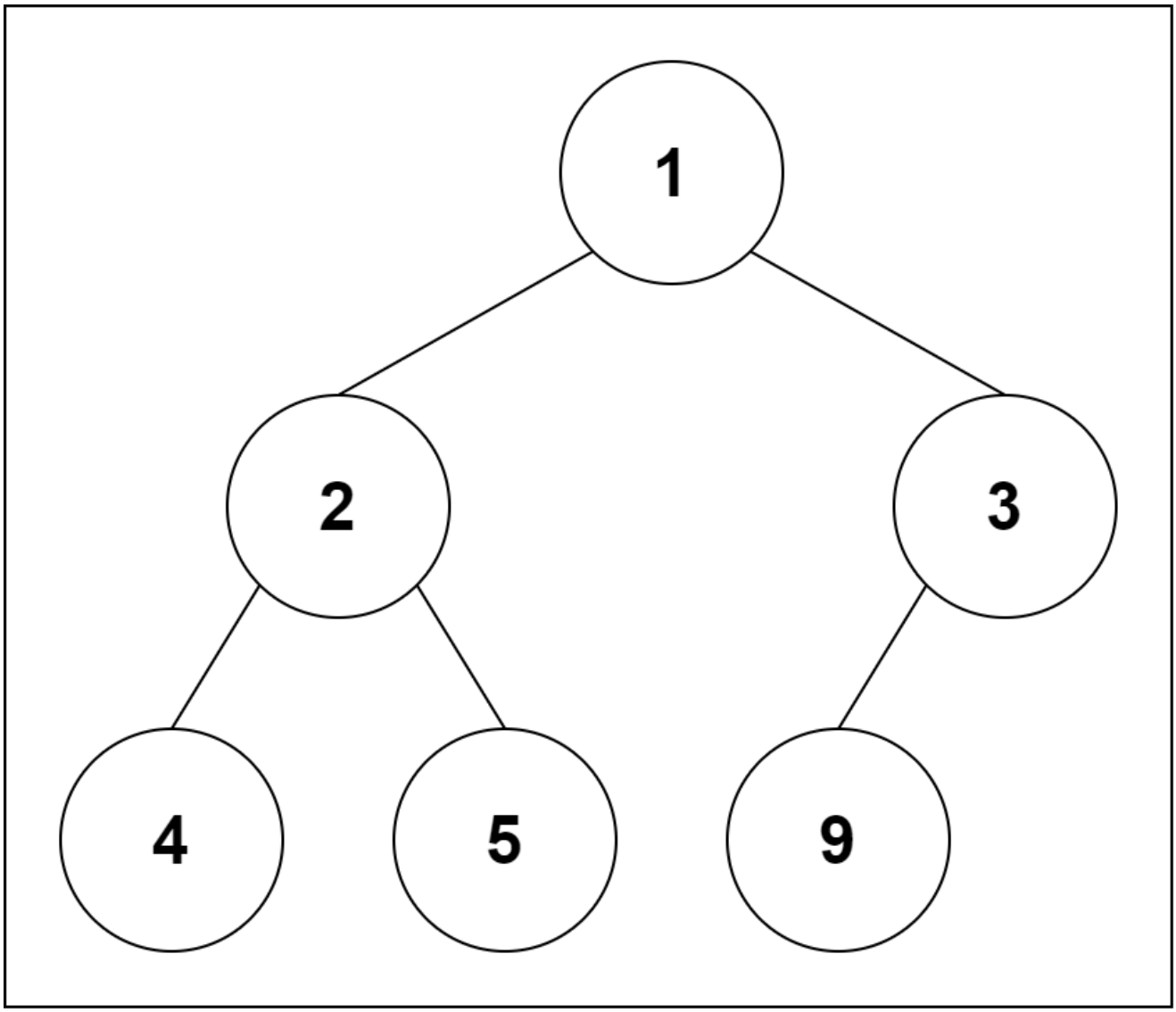
Alright, let's dive into the world of the heap data structure. Think of a heap as a special type of binary tree that follows some interesting rules.

Imagine you have a bunch of numbers and you want to arrange them in a certain way. Well, a heap is like a tree that helps you do just that! But it's not just any tree, it's a complete binary tree. That means every level of the tree is filled up, except maybe the last level, and if there are any gaps, they are to the left.

Now, here's the cool part. In a heap, each node has some special qualities. If you're dealing with a max heap, every parent node is always bigger than its child nodes. And guess what? The biggest number is right at the top, the root node!



On the other hand, if you're talking about a min heap, it's the opposite. Every parent node is smaller than its children, and the smallest number is right up there at the root.



And guess what we call this whole heap thing when we're being a bit more specific? A binary heap! It's like a magical tree that helps us keep track of data in a way that's super efficient.

So, whether you're aiming to have the biggest number on top or the smallest, a heap is your go-to structure to make things organized and quick to access.

Heap Operations

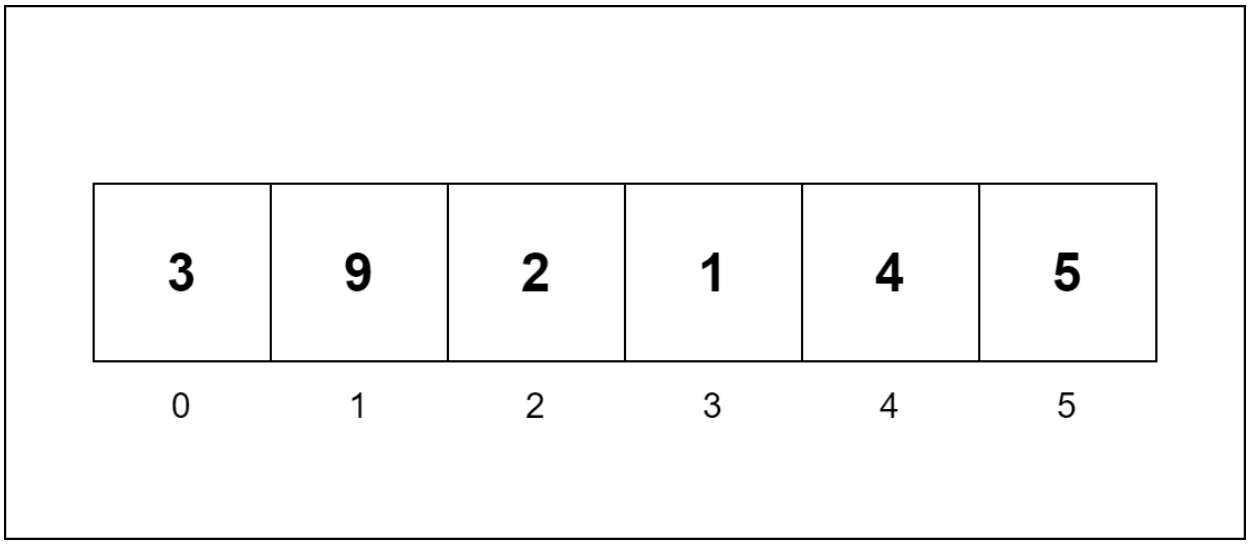
Let's explore the important things we can do with a heap. Think of these as the cool tricks that a heap can perform. I'll explain each of them to you.

Heapify

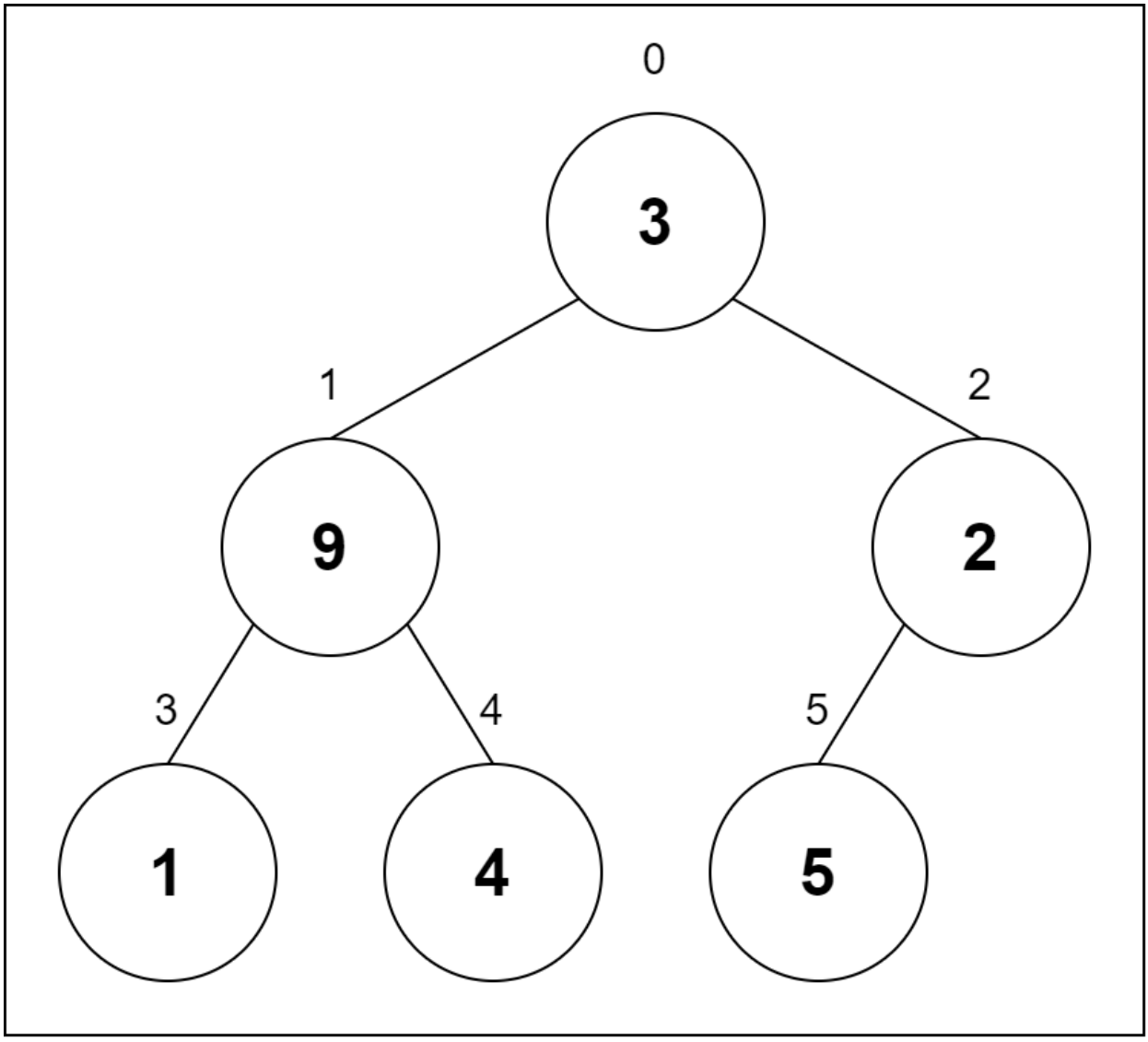
Alright, let's dive into a concept called "Heapify." It's like giving our binary tree a special structure so that it follows the heap property. Remember, a heap can be a Min-Heap or a Max-Heap.

Here's how we do it:

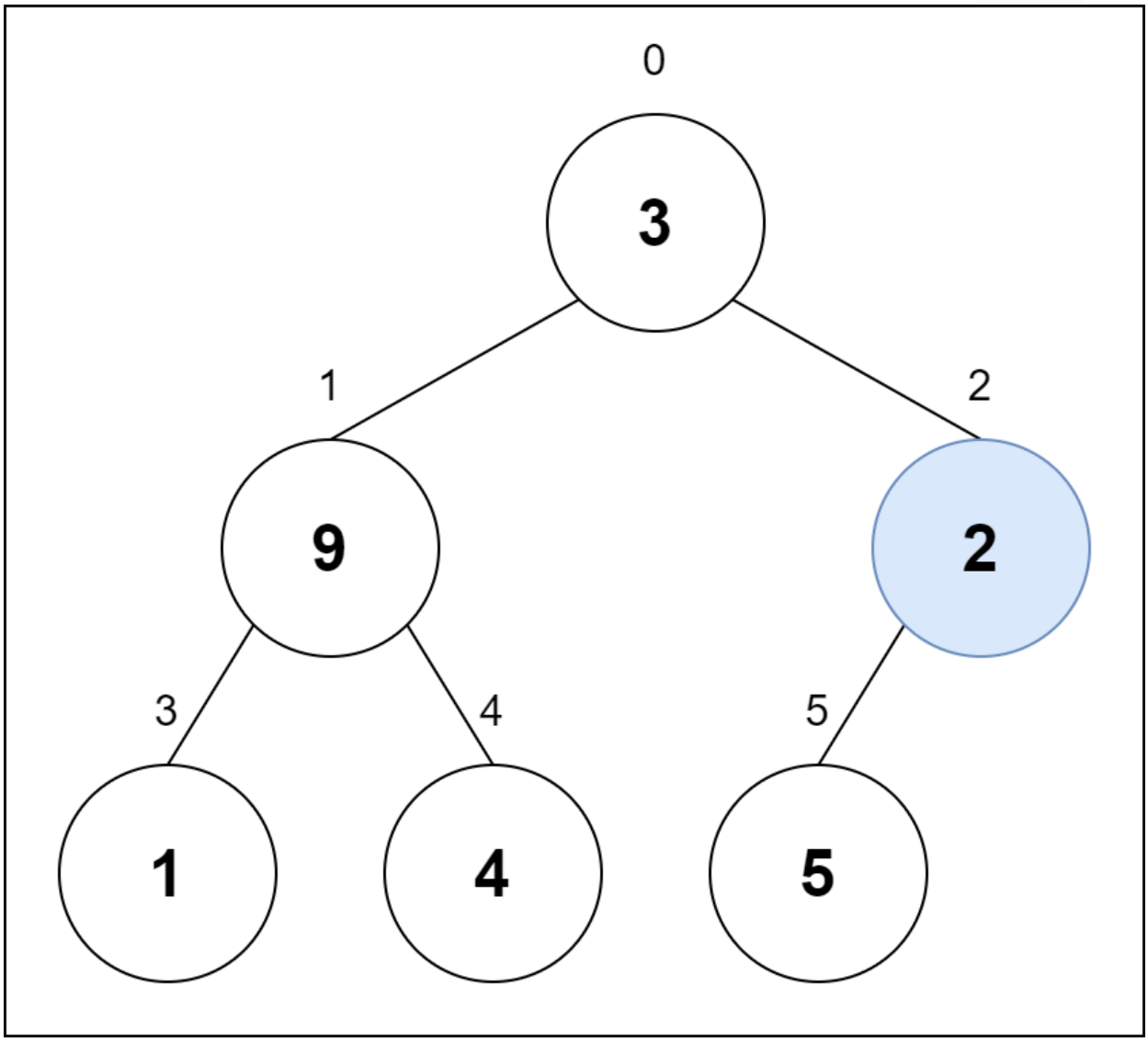
1. Imagine we have an array of numbers: {3, 9, 2, 1, 4, 5}.



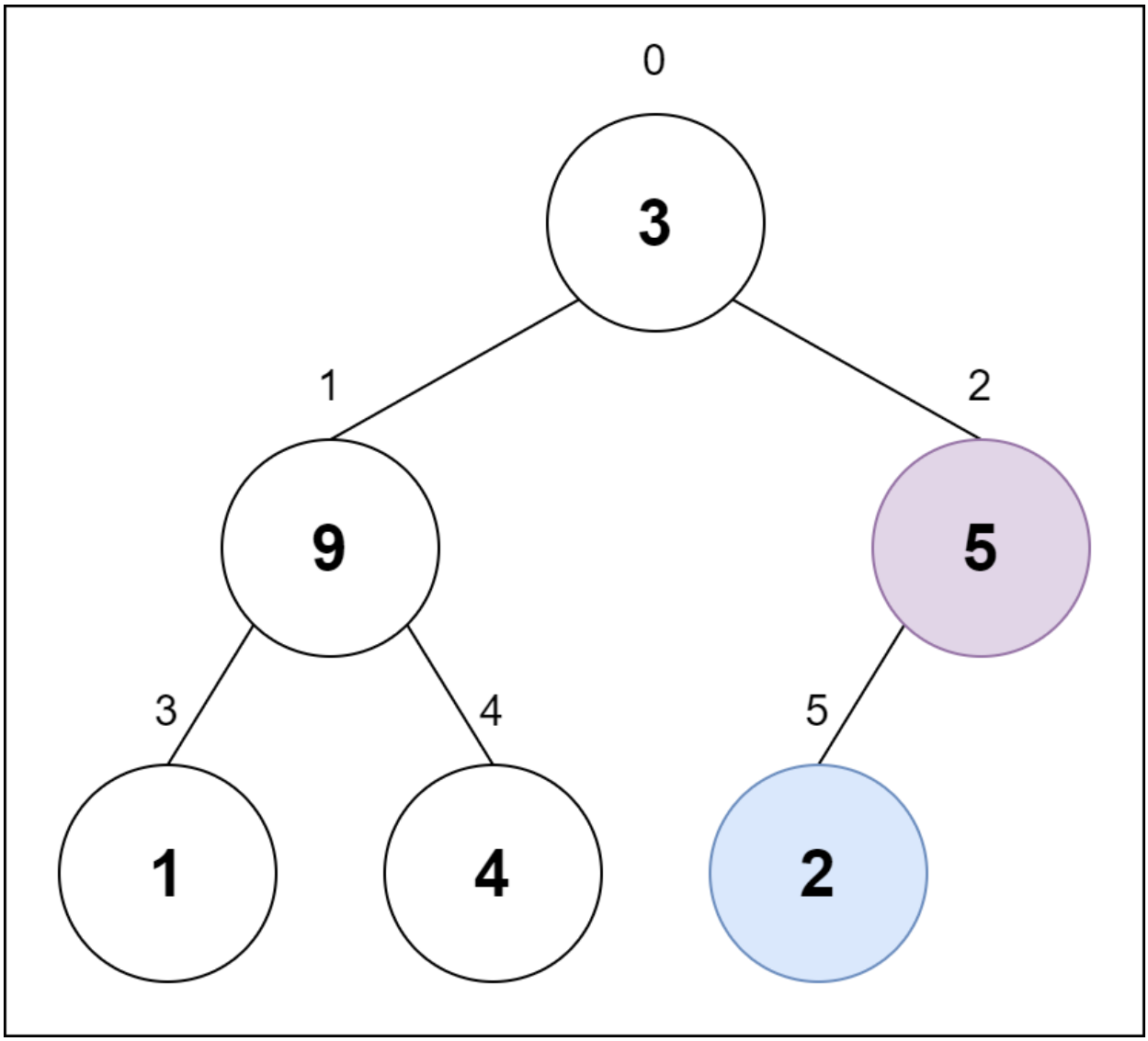
2. First, we create a complete binary tree using this array.



3. Now, we start from the first non-leaf node in the array. Its index is given by `n/2 - 1`.



4. We'll call this current element the "largest."
5. The index of its left child is given by `2i + 1`, and the right child's index is `2i + 2`.
    - If the left child is larger than the current element, we set its index as the largest.
    - Similarly, if the right child is larger than the current element, we update the largest index.
6. If the largest index is not the same as the current index, we swap the elements at these indices.



7. We repeat these steps for each non-leaf node in a bottom-up manner. This process is called "Heapify."

Algorithm

```
Heapify(array, size, i)
  set i as largest
  leftChild = 2i + 1
  rightChild = 2i + 2

  if leftChild > array[largest]
    set leftChildIndex as largest
  if rightChild > array[largest]
    set rightChildIndex as largest

  swap array[i] and array[largest]
```

- Start at a given index `i`.
- Compare the element at index `i` with its left and right child elements.

- If the left child is greater, set it as the largest index.
- If the right child is greater than the current largest, update the largest index.
- If the largest index is not `i`, swap the elements at `i` and `largest` indices.
- Repeat this process for each non-leaf node.

To create a Max-Heap:

```
MaxHeap(array, size)
  loop from the first index of non-leaf node down to zero
    call heapify
```

- We loop from the first non-leaf node index down to zero.
- For each index, we call the heapify function to ensure that the subtree rooted at that index follows the Max-Heap property.

For a Min-Heap, the comparison is reversed: both left and right children must be smaller than their parent. This way, we can organize our data in a way that's efficient for various applications.

Insert Element into Heap

Alright, let's learn how to insert an element into a heap, like adding a new student to a seating arrangement. We'll focus on Max Heap for now.

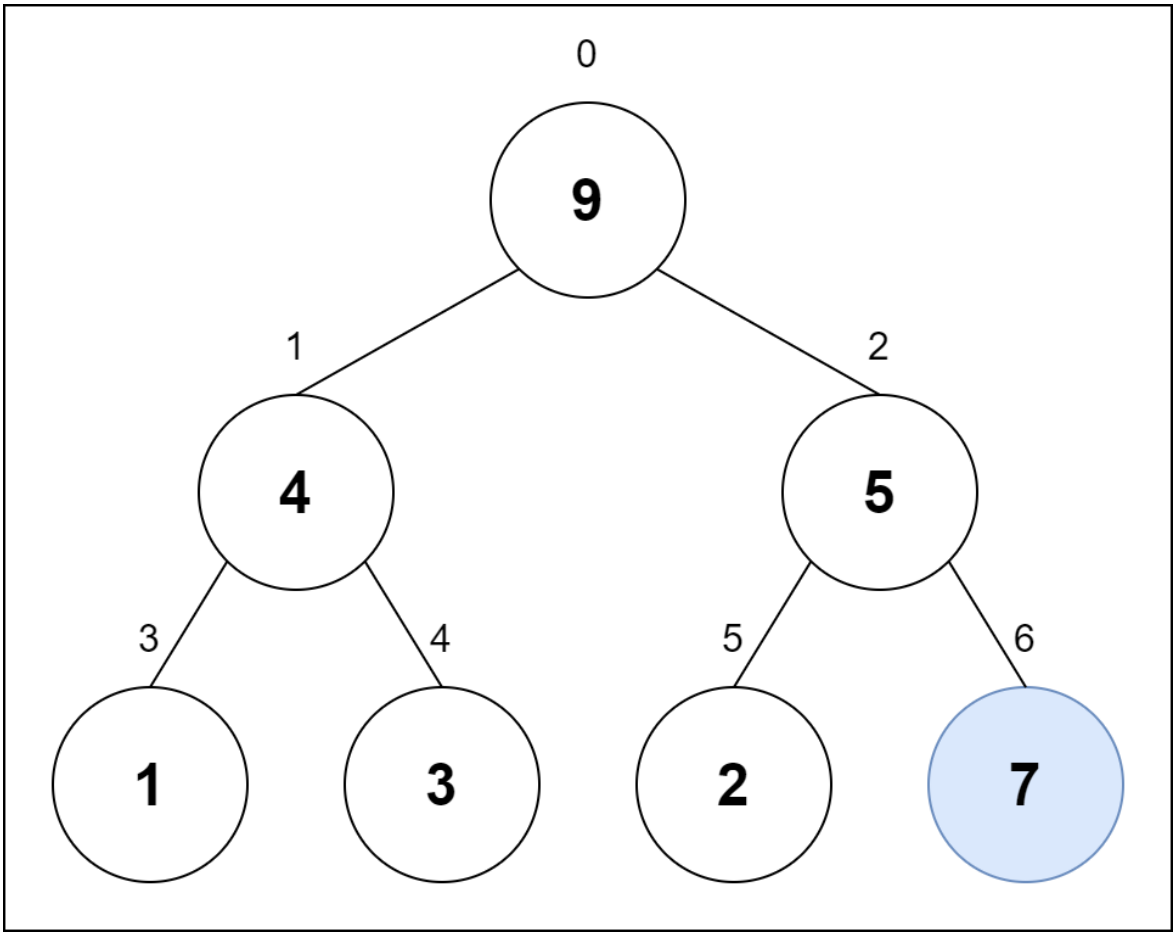
Here's how we do it:

1. First, check if there's no node in the heap. If it's empty, just create a new node with your data.
2. But if there are already some students (nodes) in the arrangement (heap), put the new student (node) at the end, on the last row, from left to right.
3. After adding the student, we need to make sure the arrangement follows the rules. This is where "heapify" comes into play. It's like adjusting the seating arrangement to maintain the order.

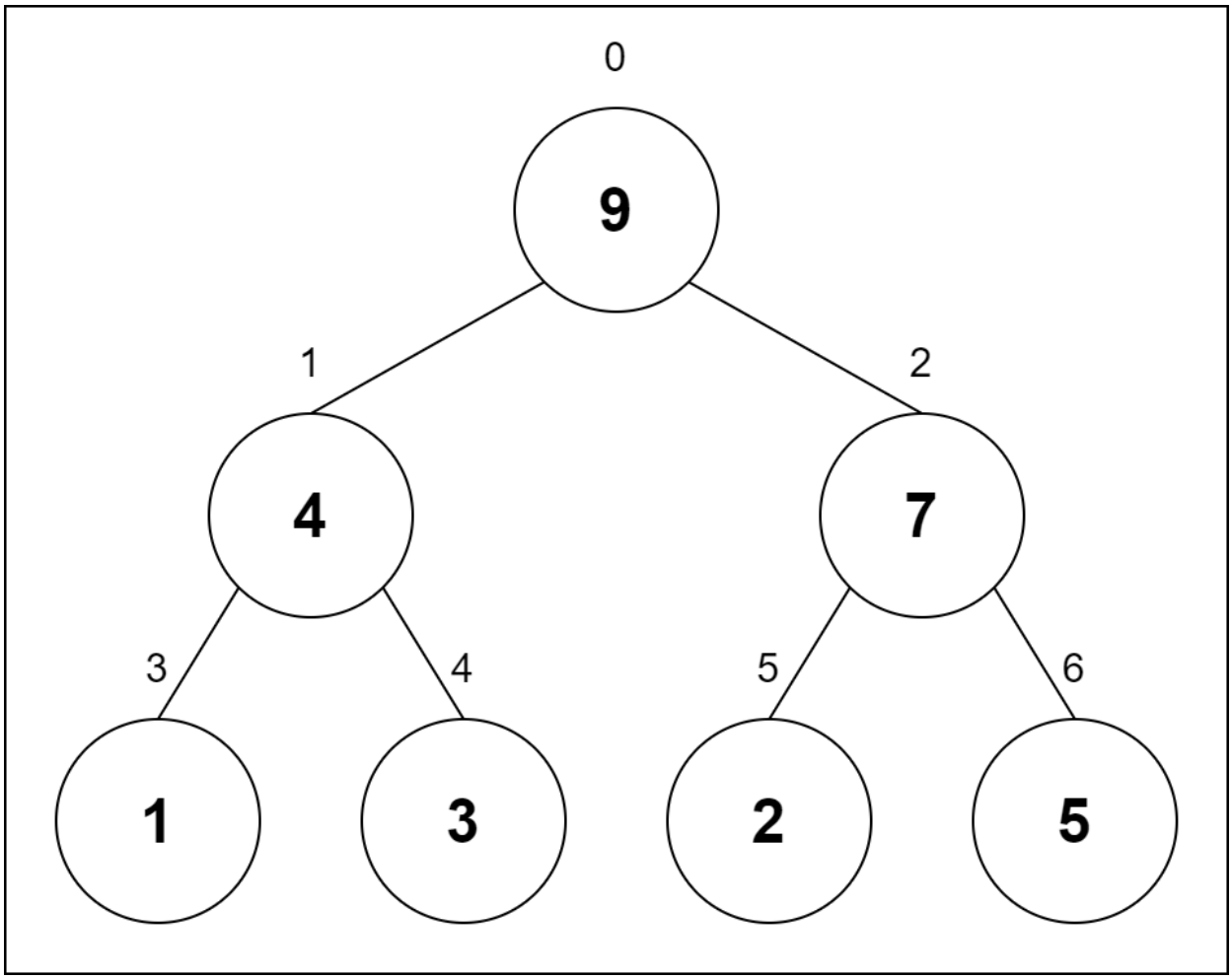
```
If there is no node,
  create a newNode.
else (a node is already present)
  insert the newNode at the end (last node from left to right.)

heapify the array
```

1. If there are no nodes, create a new node with your data.
2. If there are already nodes:
    - Put the new student (node) at the end of the arrangement.
    - This is kind of like adding a new leaf node at the end of the tree.



3. After inserting, we do heapify. This means we adjust the arrangement so that the larger students (nodes) are on top, following the Max Heap rules.



4. For Min Heap, the process is similar, but this time we make sure that the parent node is smaller than the new node. This way, the smaller students (nodes) are at the top.

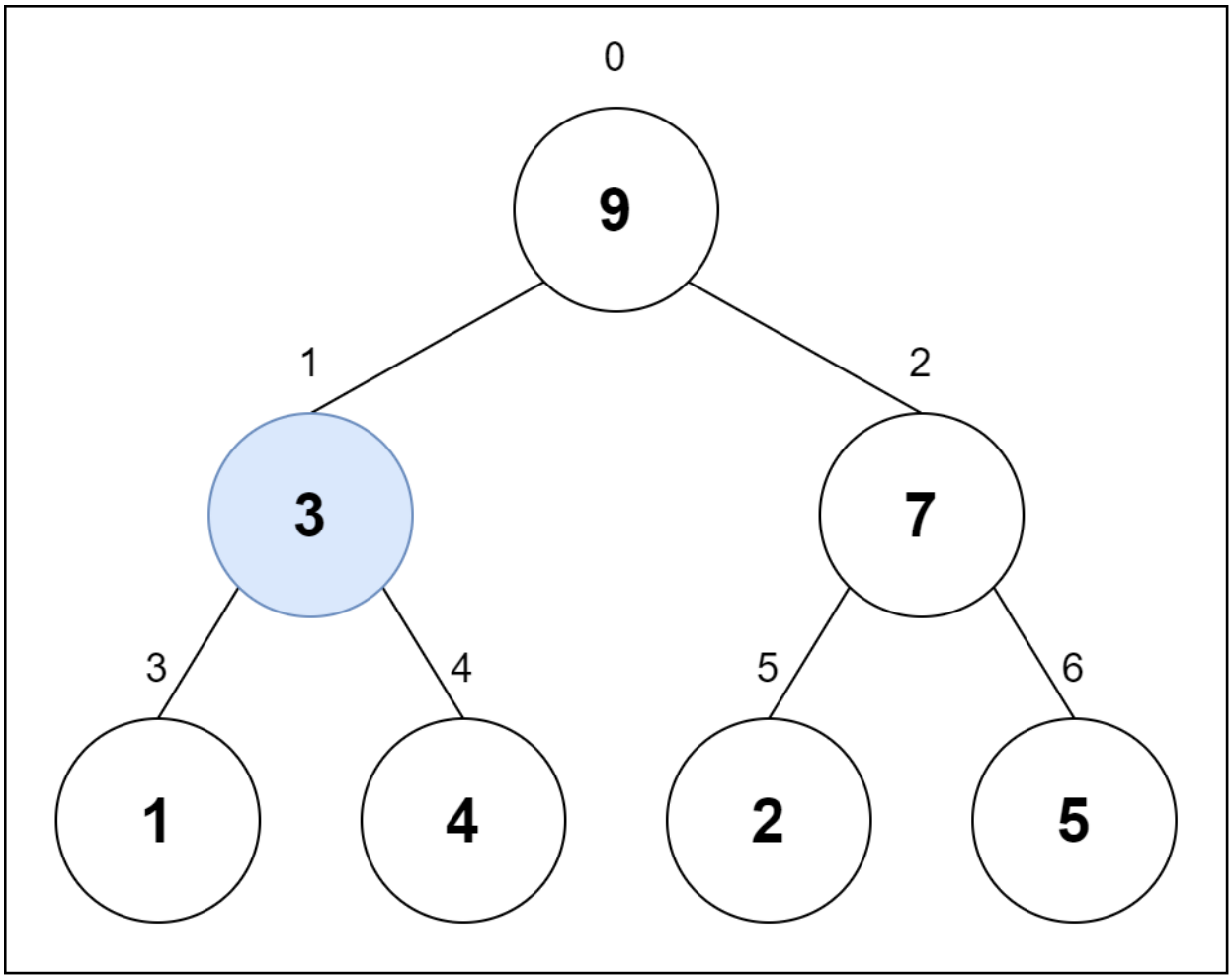
In simpler terms, imagine you're adding a new member to a club photo. You put them at the end of the row, and then you adjust the height of everyone so that the tallest people are in the back for a Max Height or the shortest are in the front for a Min Heap. This helps in keeping things organized and efficient when working with data.

Delete Element from Heap

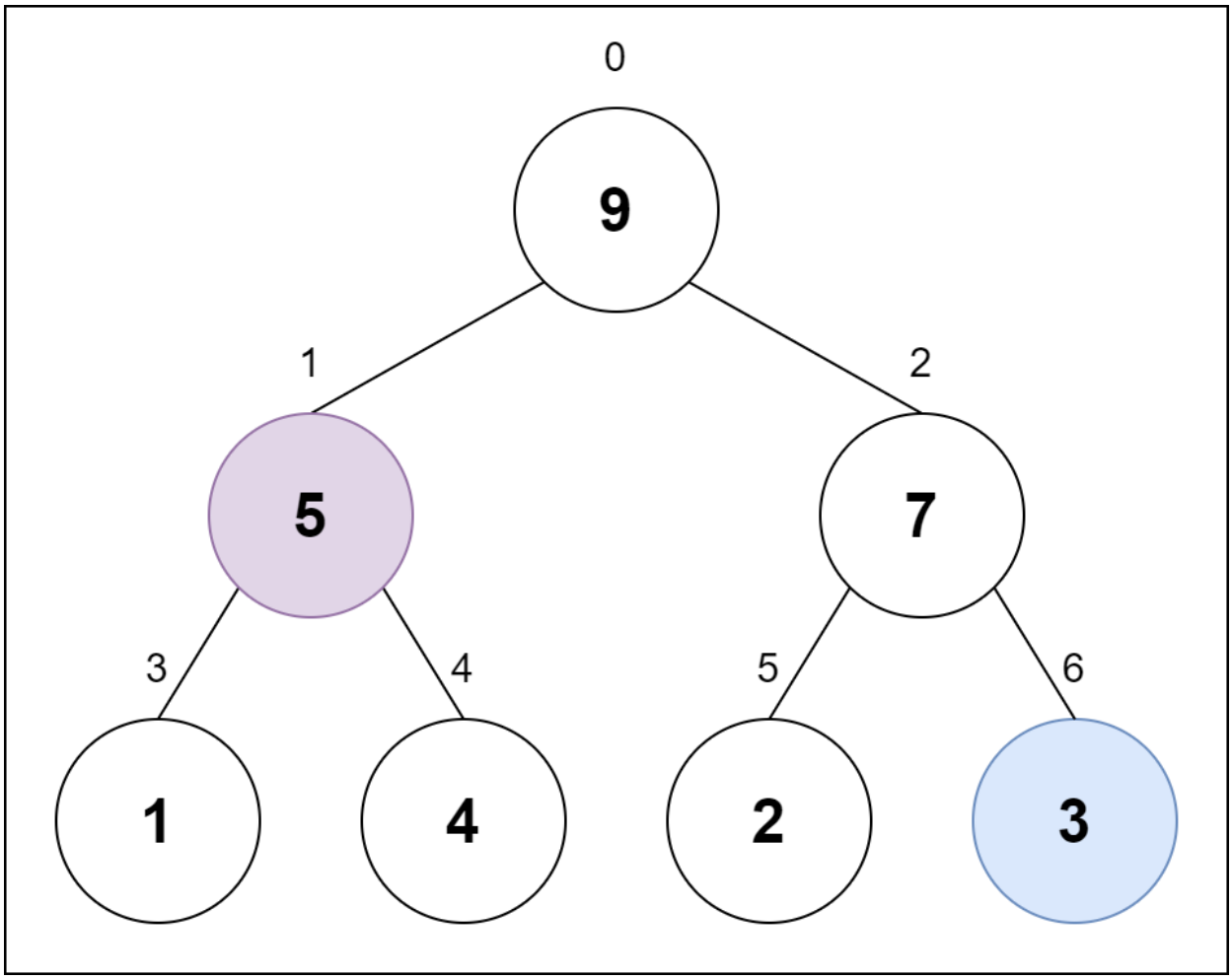
Alright, let's dive into how we can remove an element from a heap, like taking a student out of a seating arrangement. We'll focus on Max Heap for now.

Here's the process broken down:

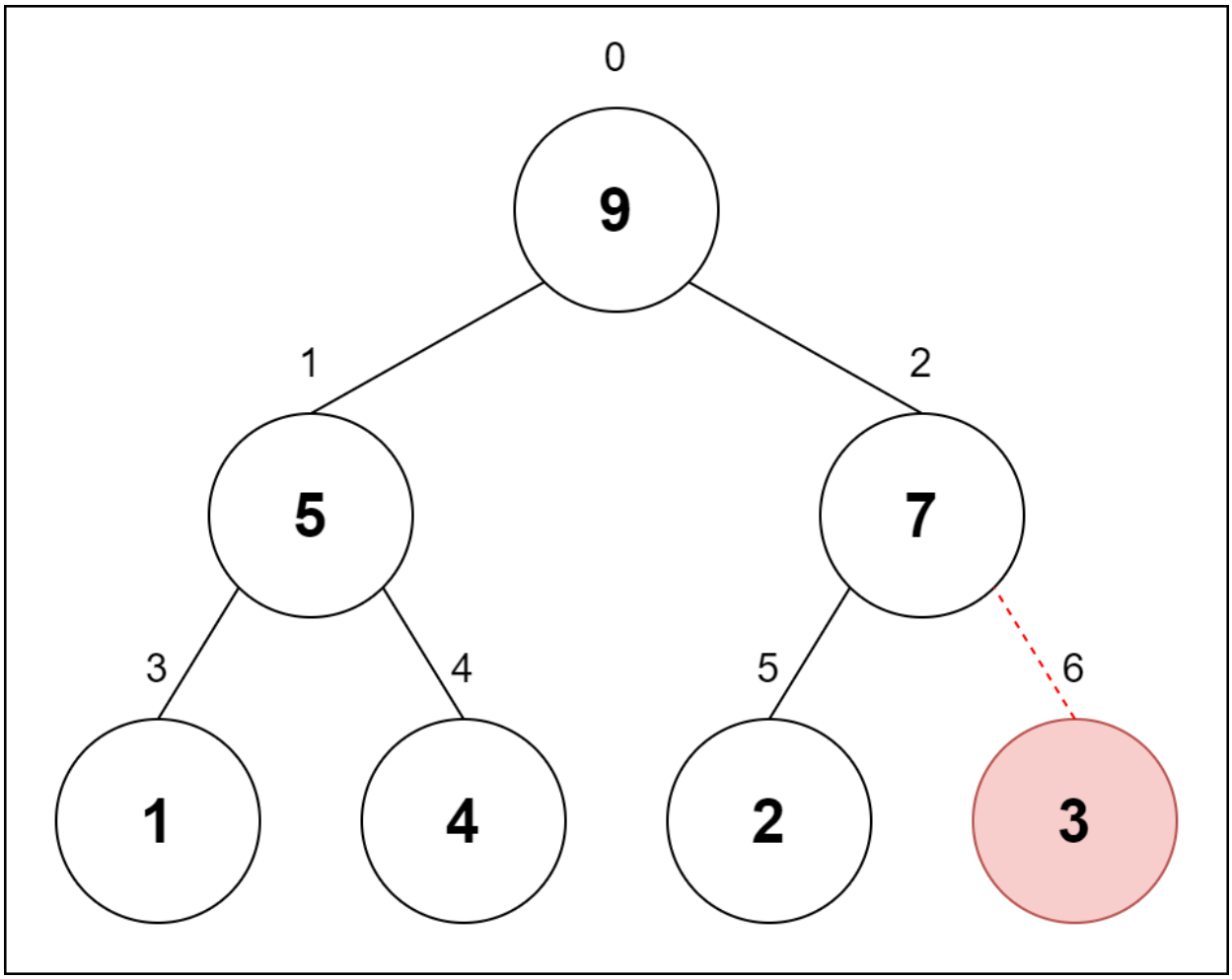
1. First, you choose the student (node) you want to remove. This is like picking the person you want to take out of the arrangement.



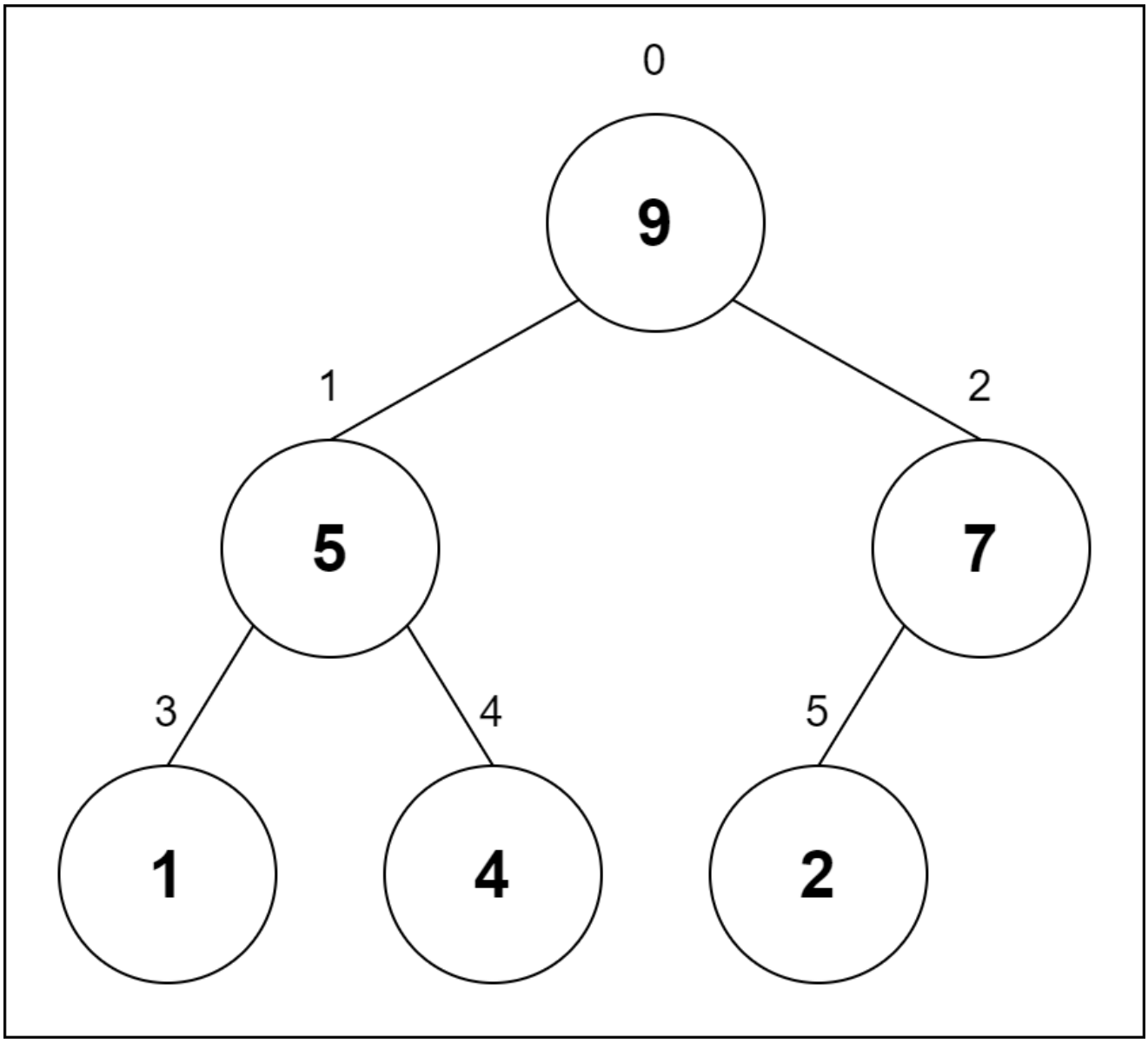
2. Then, you swap that student with the very last student in the arrangement. Think of it as if you're swapping the student you want to remove with the last person in the row.



3. Once the swap is done, you can take out the student you wanted to remove. It's like removing that student from the group photo.



4. But wait, we need to make sure the remaining students (nodes) still follow the rules. This is where "heapify" comes in. We adjust the arrangement again to maintain the order.




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

heapify the array
```

Algorithm Steps:

1. Choose the student (node) you want to remove.
2. Swap that student with the last student in the row.
3. Remove the student from the end (last node).
4. Heapify the arrangement to make sure the tallest students are still at the top for Max Heap.
5. For Min Heap, the process is similar, but this time we make sure both child nodes are greater than the parent, so the smaller students are still on top.

To put it in real-world terms, think of it like removing a friend from a group picture. You swap them with someone at the end of the line, then you remove them from the picture. And of course, you adjust everyone's height so that the tallest ones are still in the back for a Max Heap, or the shortest are in the front for a Min Heap. This helps to keep our data organized and efficient.

Peek (Find max/min)

Alright, let's talk about the "Peek" operation in heaps. Imagine you have a line of students, and you want to find out who's the tallest (for Max Heap) or the shortest (for Min Heap) without actually removing anyone from the line.

Here's how it works:

1. When you do the "Peek" operation, you're simply looking at the very first student in the line. This student is like the "root node" of the heap.
2. For a Max Heap, that student will be the tallest among all, and for a Min Heap, they'll be the shortest.

Now, here's the nitty-gritty in simple steps:

1. You do the "Peek" operation.
2. You see the student at the very front of the line (the root node).

3. That student will be the tallest for Max Heap or the shortest for Min Heap.

Think of it like checking who's leading the parade. The student at the front is the leader, either the tallest or the shortest, depending on whether it's a Max Heap or a Min Heap. And you're just checking without making any changes to the arrangement.

Extract-Max/Min

Now, let's dive into the "Extract-Max" and "Extract-Min" operations in heaps. Imagine you have a collection of items, like a bunch of toys, and you want to take out the most valuable one (for Max Heap) or the least valuable one (for Min Heap).

Here's how these operations work:

- **Extract-Max**:
  1. You look at the heap.
  2. You pick the item with the highest value (the maximum) from the collection.
  3. You remove that item from the heap.
  4. Now, the heap rearranges itself so that the next highest item takes the place of the one you removed.
- **Extract-Min**:
  1. You look at the heap.
  2. You pick the item with the lowest value (the minimum) from the collection.
  3. You remove that item from the heap.
  4. The heap adjusts itself, making sure the next lowest item is now at the front.

So, if you have a heap of toys, for instance, and you do an "Extract-Max," you're getting the most valuable toy. If you do an "Extract-Min," you're getting the least valuable one.

Think of it like picking the best or the least preferred item from a collection and then letting the collection reorganize itself. It's like taking out the shiniest gem from a treasure chest and letting the other gems shuffle into place.

C++ Example

```cpp
// Max-Heap data structure in C++

#include <iostream>
#include <vector>
using namespace std;
```

```
void swap(int *a, int *b)
{
  int temp = *b;
  *b = *a;
  *a = temp;
}
void heapify(vector<int> &hT, int i)
{
  int size = hT.size();
  int largest = i;
  int l = 2 * i + 1;
  int r = 2 * i + 2;
  if (l < size && hT[l] > hT[largest])
    largest = l;
  if (r < size && hT[r] > hT[largest])
    largest = r;

  if (largest != i)
  {
    swap(&hT[i], &hT[largest]);
    heapify(hT, largest);
  }
}
void insert(vector<int> &hT, int newNum)
{
  int size = hT.size();
  if (size == 0)
  {
    hT.push_back(newNum);
  }
  else
  {
    hT.push_back(newNum);
    for (int i = size / 2 - 1; i >= 0; i--)
    {
      heapify(hT, i);
    }
  }
}
void deleteNode(vector<int> &hT, int num)
{
  int size = hT.size();
  int i;
  for (i = 0; i < size; i++)
  {
    if (num == hT[i])
      break;
  }
  swap(&hT[i], &hT[size - 1]);
```

```
  hT.pop_back();
  for (int i = size / 2 - 1; i >= 0; i--)
  {
    heapify(hT, i);
  }
}
void printArray(vector<int> &hT)
{
  for (int i = 0; i < hT.size(); ++i)
    cout << hT[i] << " ";
  cout << "\n";
}

int main()
{
  vector<int> heapTree;

  insert(heapTree, 3);
  insert(heapTree, 4);
  insert(heapTree, 9);
  insert(heapTree, 5);
  insert(heapTree, 2);

  cout << "Max-Heap array: ";
  printArray(heapTree);

  deleteNode(heapTree, 4);

  cout << "After deleting an element: ";

  printArray(heapTree);
}
```