7

Heaps and Priority Queues

A heap data structure is a tree-based data structure in which each node of the tree has a specific relationship with other nodes, and they are stored in a specific order. Depending upon the specific order of the nodes in the tree, heaps can be of different types, such as a `min` heap and a `max` heap.

A priority queue is an important data structure that is similar to the queue and stack data structures that stores data along with the priority associated with them. In this, the data is served according to the priority. Priority queues can be implemented using an array, linked list, and trees; however, they are often implemented using a heap as it is very efficient.

In this chapter, we will learn the following:

- The concept of the heap data structure and different operations on it
- Understanding the concept of the priority queue and its implementation using Python

Heaps

A heap data structure is a specialization of a tree in which the nodes are ordered in a specific way. A heap is a data structure where each data elements satisfies a `heap` property, and the `heap` property states that there must be a certain relationship between a parent node and its child nodes. According to this certain relationship in the tree, the heaps can be of two types, in other words, `max` heaps and `min` heaps. In a `max` heap, each parent node value must always be greater than or equal to all its children. In this kind of tree, the `root` node must be the greatest value in the tree. For example, see *Figure 7.1* showing the `max` heap in which all the nodes have greater values compared to their children:

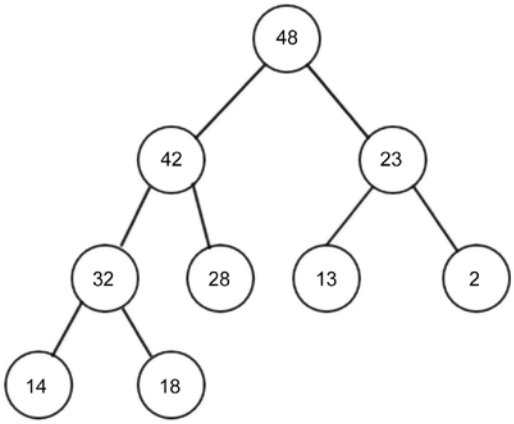

Figure 7.1: An example of a max heap

In a `min` heap, the relationship between parent and children is that the value of the parent node must always be less than or equal to its children. This rule should be followed by all the nodes in the tree. In the `min` heap, the `root` node holds the lowest value. For example, see *Figure 7.2* showing the `min` heap in which all the nodes have smaller values compared to their children:

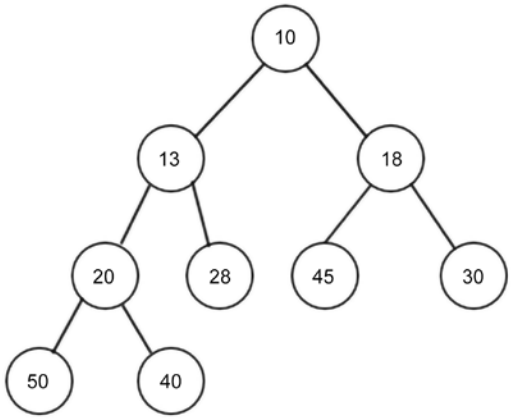

Figure 7.2: An example of a min heap

The heap is an important data structure due to its several applications and uses in implementing heap sort algorithms and priority queues. We will be discussing these in detail later in the chapter. The heap can be any kind of tree; however the most common type of heap is a binary heap in which each node has at most two children.

If the binary heap is a **complete binary tree** with `n` nodes, then it will have a minimum height of $\log_2 n$.

A complete binary tree is one in which each row must be fully filled before starting to fill the next row, as shown in the following *Figure 7.3*:

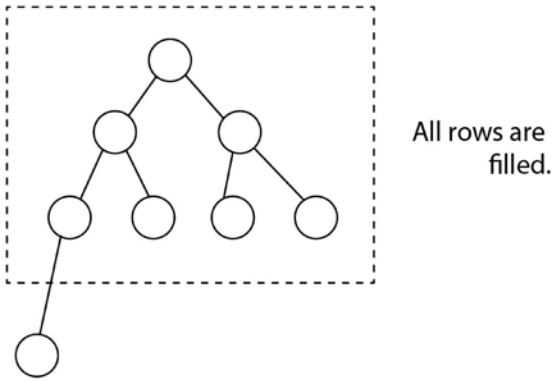

Figure 7.3: An example of a complete binary tree

In order to implement the heap, we can derive a relationship between parent and child nodes in `index` values. The relationship is that the children of any node at the `n` index can be retrieved easily, in other words, the left child will be located at `2n`, and the right child will be located at `2n + 1`. For example, the node `C` would be at the index of `3`, since node `C` is a right child of the node `A`, which is at index `1`, so it becomes `2n+1 = 2*1 + 1 = 3`. This relationship always holds true. Let's say we have a list of elements `{A, B, C, D, E}` as shown in *Figure 7.4*. If we store any element at an index of `i`, then its parent can be stored at index `i/2`, for example, if the index of the node `D` is 4, then its parent would be at 4/2 = 2, index 2. The index of root has to be starting from 1 in the array. See *Figure 7.4* to understand the concept:

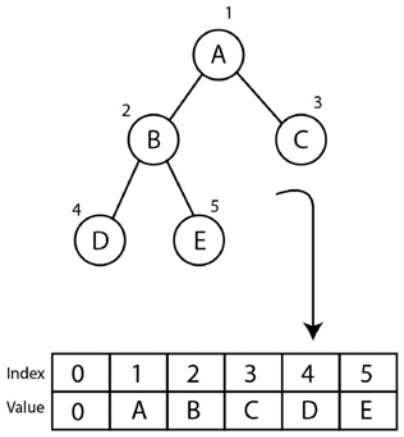

Figure 7.4: Binary tree and index positions of all the nodes

This relation between parent and child is a complete binary tree. In respect of indexing values, it is very important in order to efficiently retrieve, search, and store the data elements in the heap. Due to this property, it is very easy to implement the heap. The only constraint is that we should have indexing starting from `1`, and if we implement the heap using an array, then we have to add one dummy element at index `0` in the array. Next, let's understand the implementation of the heap. It is important to note that we will be discussing all the concepts with respect to the `min` heap, and the implementation for the `max` heap will be very similar to it, with the only difference being the `heap` property.

Let's discuss the implementation of the `min` heap using Python. We start with the `heap` class, as follows:

```
    class MinHeap:
        def __init__(self):
            self.heap = [0]
            self.size = 0
```

We initialize the heap list with a zero to represent the dummy first element, and we are adding a dummy element just to start the indexing of data items from `1` since if we start indexing from `1`, accessing of the elements becomes very easy due to the parent-child relationship. We also create a variable to hold the size of the heap. We will further discuss different operations, such as `insert`, `delete`, and `delete` at a specific location in the heap. Let's start with the insertion operation in the heap.

Insert operation

The insertion of an item into a `min` heap works in two steps. First, we add the new element to the end of the list (which we understand to be the bottom of the tree), and we increment the size of the heap by one. Secondly, after each insertion operation, we need to arrange the new element up in the heap tree, to organize all the nodes in such a way that satisfies the `heap` property, which in this case is that each node must be larger than its parent. In other words, the value of the parent node must always be less than or equal to its children, and the lowest element in the `min-heap` needs to be the root element. Therefore, we first insert an element into the last heap of the tree; however, after inserting an element into the heap, it is possible that the `heap` property is violated. In that case, the nodes have to be rearranged so that all the nodes satisfy the `heap` property. This process is called heapifying. To heapify the `min` heap, we need to find the minimum of its children and swap it with the current element, and this process has to be repeated until the `heap` property is satisfied for all the nodes.

Let's consider an example of adding an element in the `min` heap, such as inserting a new node with a value of 2 in *Figure 7.5*:

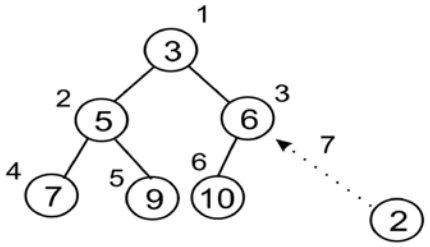

Figure 7.5: Insertion of a new node 2 in the existing heap

The new element will be added to the last position in the third row or level. Its index value is 7. We compare that value with its parent. The parent is at index `7/2 = 3` (integer division). The parent node holds value 6, which is higher than the new node value (in other words, 2), so according to the property of the `min` heap, we swap these values, as shown in *Figure 7.6*:

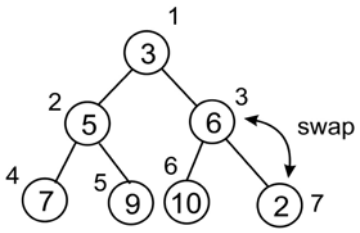

Figure 7.6: Swapping nodes 2 and 6 to maintain the heap property

The new data element has been swapped and moved up to index 3. Since, we have to check all the nodes up to the root, we check the index of its parent node which is *3/2 = 1 (integer division)*, so we continue the process to heapify.

So, we compare both of these elements, and swap again, as shown in *Figure 7.7*:

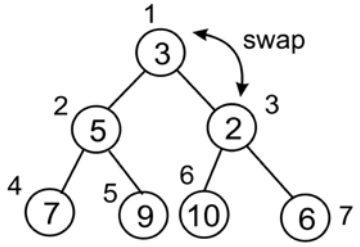

Figure 7.7: Swapping nodes 2 and 3 to maintain the heap property

After the final swap, we reach the root. Here, we can notice that this heap adheres to the definition of the `min` heap, as shown in *Figure 7.8*:

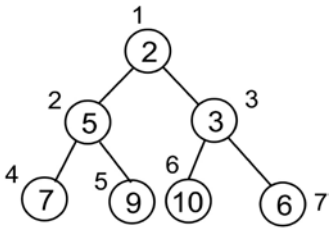

Figure 7.8: Final heap after insertion of a new node 2

Now, let's take another example to see how to create and insert elements in a heap. We start with the construction of a heap by inserting 10 elements, one by one. The elements are `{4, 8, 7, 2, 9, 10, 5, 1, 3, 6}`. We can see a step-by-step process to insert elements into the heap in *Figure 7.9*:

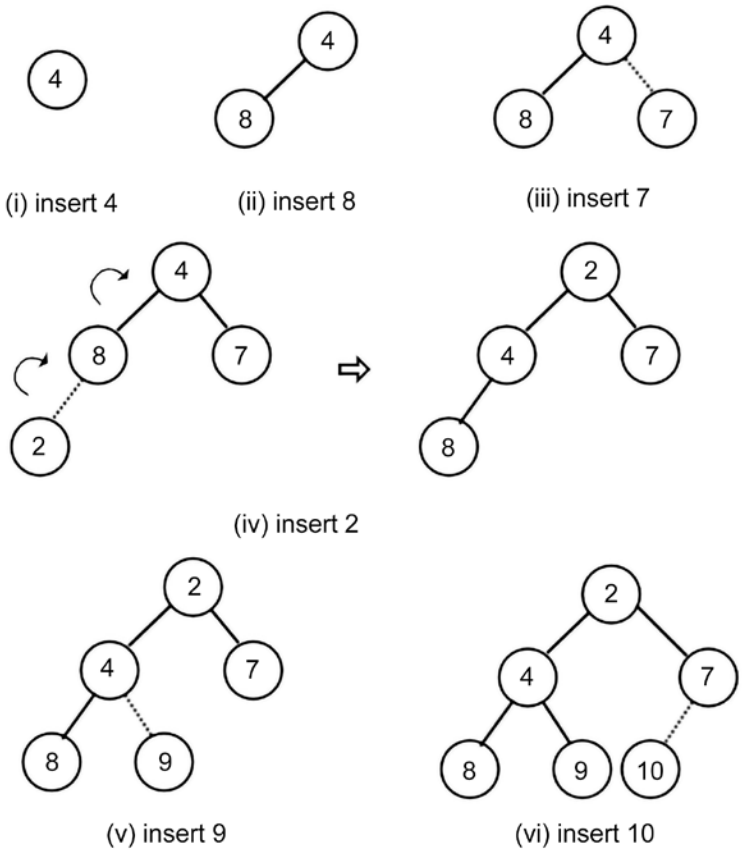

Figure 7.9: The step-by-step procedure to create a heap

We can see, in the preceding diagram, a step-by-step process to insert elements into the heap. Here, we continue adding elements, as shown in *Figure 7.10*:

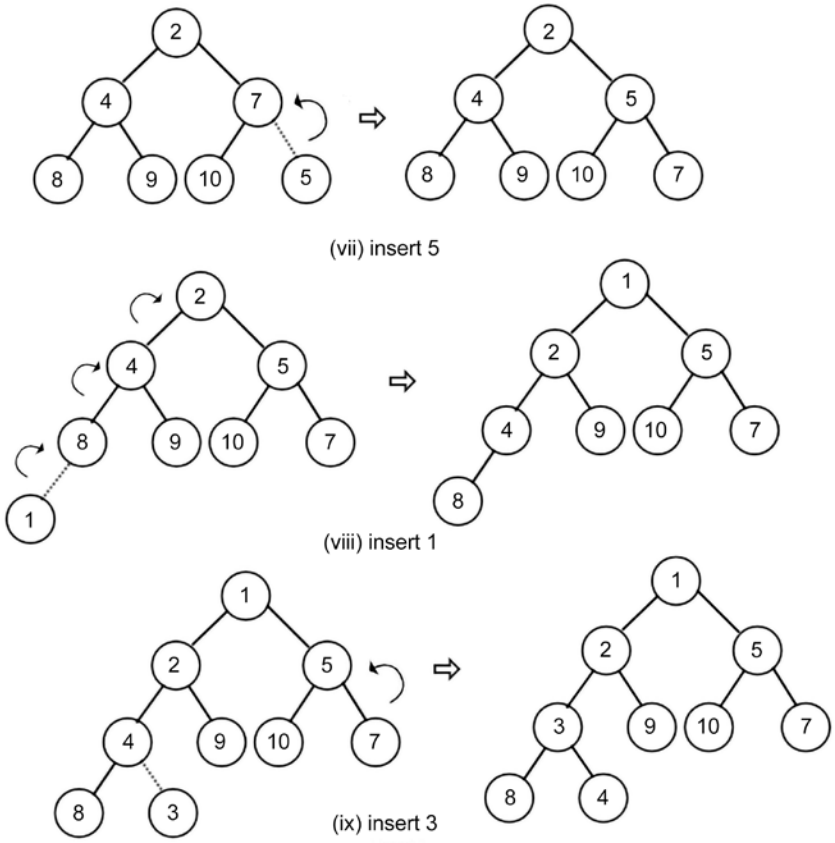

Figure 7.10: Steps 7 to 9 in creating the heap

Finally, we insert an element, 6, into the heap, as shown in *Figure 7.11*:

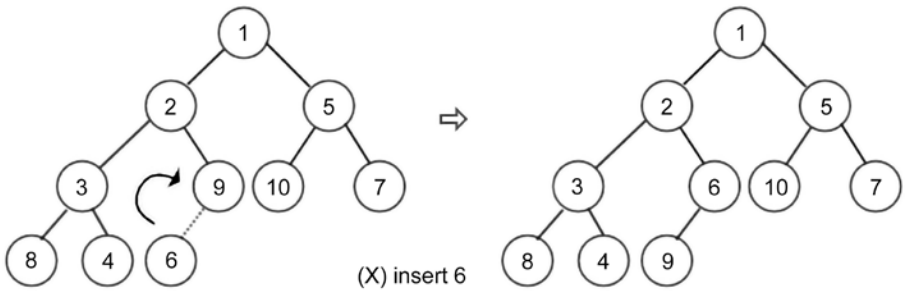

Figure 7.11: Last step and construction of the final heap

The implementation of the insertion operation in the heap is discussed as follows. Firstly, we create a helper method, called the `arrange`, that takes care of arrangements of all the nodes after insertion of a new node. Here is the implementation of the `arrange()` method, which should be defined in the `MinHeap` class:

```
    def arrange(self, k):
        while k // 2 > 0:
            if self.heap[k] < self.heap[k//2]:
                self.heap[k], self.heap[k//2] = self.heap[k//2], self.
heap[k]
            k //= 2
```

We execute the loop until we reach up to the `root` node; until then, we can keep arranging the element. Here, we are using integer division. The loop will break out after the following condition:

```
    while k // 2 > 0:
```

After that, we compare the values between the parent and child node. If the parent is greater than the child, swap the two values:

```
        if self.heap[k] < self.heap[k//2]:
            self.heap[k], self.heap[k//2] = self.heap[k//2], self.heap[k]
```

Finally, after each iteration, we move up in the tree:

```
             k //= 2
```

This method ensures that the elements are ordered properly.

Now, for adding new elements in the heap, we need to use the following `insert` method, which should be defined in the `MinHeap` class:

```
    def insert(self, item):
        self.heap.append(item)
        self.size += 1
        self.arrange(self.size)
```

In the above code, we can insert an element using the `append` method; then we increase the size of the heap. Then, in the last line of the `insert` method, we call the `arrange()` method to reorganize the heap (heapify it) to ensure that all the nodes in the heap satisfy the `heap` property.

Now, let's create the heap and insert that data {4, 8, 7, 2, 9, 10, 5, 1, 3, 6} using the `insert()` method, which is defined in the `MinHeap` class, as shown in the following code:

```
    h = MinHeap()
    for i in (4, 8, 7, 2, 9, 10, 5, 1, 3, 6):
        h.insert(i)
```

We can print the heap list, just to inspect how the elements are ordered. If you redraw this as a tree structure, you will notice that it meets the required properties of a heap, similar to what we created manually:

```
    print(h.heap)
```

The output of the above code is as follows:

```
[0, 1, 2, 5, 3, 6, 10, 7, 8, 4, 9]
```

We can see in the output that all the data items of the heap in the array are as in the index position as per *Figure 7.11*. Next, we will discuss the delete operation in the heap.

Delete operation

The `delete` operation removes an element from the heap. To delete any element from the heap, let's first discuss how we can delete the root element since it is mostly used for several use cases, such as finding the minimum or maximum element in a heap. Remember, in a `min-heap`, the root element denotes the minimum value of the list, and the root of the `max-heap` gives the maximum value of the list of elements.

Once we delete the root element from the heap, we make the last element of the heap the new root of the heap. In that case, the `heap` property will not be satisfied by the tree. So, we have to reorganize the nodes of the tree such that all the nodes of the tree satisfy the `heap` property. The delete operation in `min-heap` works as follows.

1. Once we delete the `root` node, we need a new `root` node. For this, we take the last item from the list and make it the new root.
2. Since the selected last node might not be the lowest element of the heap, we have to reorganize the nodes of the heap.
3. We reorganize the nodes from the `root` node to the last node (which is made into a new root); this process is called heapify. Since we move from top to bottom (which means from the `root` node down to the last element) of the heap, this process is called percolate down.

Let's consider an example to help us understand this concept in the following heap. First, we delete the `root` node that has value 2, as shown in *Figure 7.12*:

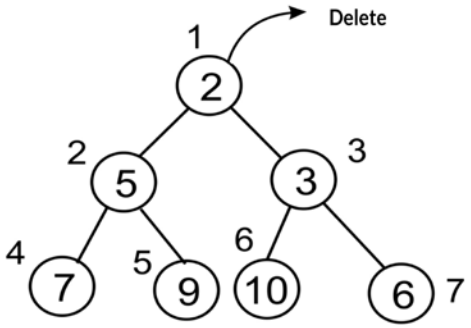

Figure 7.12: Deletion of a node with value 2 at the root in the existing heap

Once we delete the root, next we need to choose a node that can be the new root; commonly, we choose to take the last node, in other words, node 6 at index 7. So, the last element, 6, is placed at the root position, as shown in *Figure 7.13*:

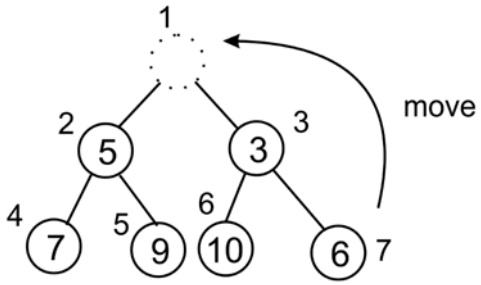

Figure 7.13: Moving the last element, in other words, node 6 to the root position

After moving the last element to the new root, clearly this tree is now not satisfying the `min-heap` property. So, we have to reorganize the nodes of the heap, hence we move down from the root to the nodes in the heap, that is, heapify the tree. So, we compare the value of the newly replaced node with all its children nodes in the tree. In this example, we compare the two children of the root, that is, 5 and 3. Since the right child is smaller, its index is 3, which is represented as *(root index * 2 + 1)*. We will go ahead with this node and compare the new `root` node with the value at this index, as shown in *Figure 7.14*:

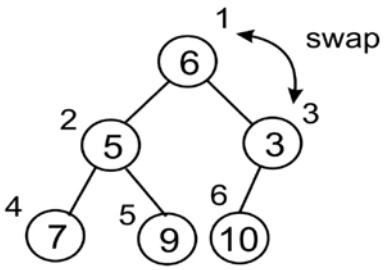

Figure 7.14: Swapping of the root node with the node 3

Now, the node with value 6 should be moved down to index 3 as per the `min heap` property. Next, we need to compare it to its children down to the heap. Here, we only have one child, so we don't need to worry about which child to compare it against (for a `min` heap, it is always the lesser child), as shown in *Figure 7.15*:

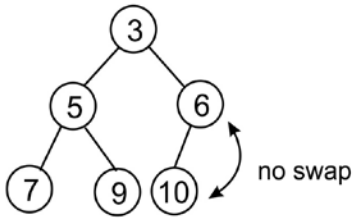

Figure 7.15: Swapping of node 6 and node 10

There is no need to swap here since it is following the `min-heap` property. After reaching the last one, the final heap adheres to the `min-heap` property.

In order to implement the deletion of the `root` node from the heap using Python, firstly, we implement the percolate-down process, in other words, the `sink()` method. Before we implement the `sink()` method, we implement a `helper` method for finding out which of the children to compare against the parent node. This `helper` method is `minchild()`, which should be defined in the `MinHeap` class:

```
    def minchild(self, k):
        if k * 2 + 1 > self.size:
            return k * 2
        elif self.heap[k*2] < self.heap[k*2+1]:
            return k * 2
        else:
            return k * 2 + 1
```

In this method, firstly, we check if we get beyond the end of the list—if we do, then we return the index of the left child:

```
        if k * 2 + 1 > self.size:
            return k * 2
```

Otherwise, we simply return the index of the lesser of the two children:

```
        elif self.heap[k*2] < self.heap[k*2+1]:
            return k * 2
        else:
            return k * 2 + 1
```

Now we can create the `sink()` method. The `sink()` method should be defined in the `MinHeap` class:

```
def sink(self, k):
        while k * 2 <= self.size:
            mc = self.minchild(k)
            if self.heap[k] > self.heap[mc]:
                self.heap[k], self.heap[mc] = self.heap[mc], self.heap[k]
            k = mc
```

In the above code, we first run the loop until the end of the tree so that we can sink (move down) our element down as far as is needed; this is shown in the following code snippet:

```
    def sink(self, k):
          while k*2 <= self.size:
```

Next, we need to know which of the left or right children to compare against. This is where we make use of the `minindex()` function, as shown in the following code snippet:

```
            mi = self.minchild(k)
```

Next, we compare parent and child to see whether we need to make the swap, as we did in the `arrange()` method during the insertion operation:

```
            if self.heap[k] > self.heap[mc]:
                self.heap[k], self.heap[mc] = self.heap[mc], self.heap[k]
```

Finally, we need to make sure that we move down the tree in each iteration so that we don't get stuck in a loop, as follows:

```
            k = mc
```

Now, we can implement the main `delete_at_root()` method itself, which should be defined in the `MinHeap` class:

```
    def delete_at_root(self):
        item = self.heap[1]
        self.heap[1] = self.heap[self.size]
        self.size -= 1
        self.heap.pop()
        self.sink(1)
        return item
```

In the above code for deletion of the `root` node, we first copy the root element in a variable item, and then the last element is moved to the `root` node in the following statement:

```
self.heap[1] = self.heap[self.size]
```

Further, we reduce the size of the heap, and remove the element from the heap, and then we use the `sink()` method to reorganize the heap element so that all the elements of the heap follow the `heap` property.

We can now use the following code to delete the `root` node from the heap. Let's first insert some data items `{2, 3, 5, 7, 9, 10, 6}` in the heap and then remove the `root` node:

```
h = MinHeap()
for i in (2, 3, 5, 7, 9, 10, 6):
    h.insert(i)
print(h.heap)
n = h.delete_at_root()
print(n)
print(h.heap)
```

The output of the above code is as follows:

```
[0, 2, 3, 5, 7, 9, 10, 6]
2
[0, 3, 6, 5, 7, 9, 10]
```

We can see in the output that the root element 2 is returned in the new heap, and that the data elements are rearranged so that all the nodes of the heap are following the `heap` property (indexes of the nodes can checked as shown in *Figure 7.16*). Next, we will discuss if we want to delete any node with the given index position.

Deleting an element at a specific location from a heap

Generally, we delete an element at the root, however, an element can be deleted at a specific location from the heap. Let us understand it with an example. Given the following heap, let's assume that we want to delete a node with value 3 at index 2. After deleting the node with value 3, we move the last node to the deleted node, in other words, the node with value 15, as shown in *Figure 7.16*:

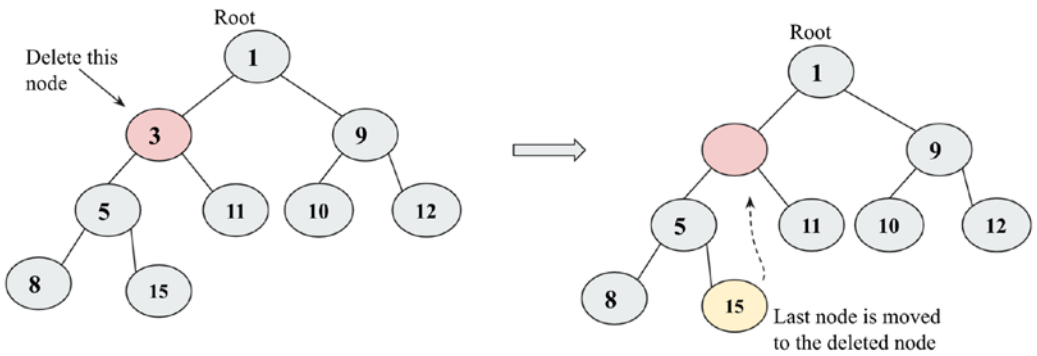

Figure 7.16: The deletion of node 3 from the heap

After shifting the last element to the deleted node, we compare this with its root element since it is already greater than the root element, so we do not swap. Next, we compare this element with all of its children, and since the left child is smaller, it is swapped with the left child, as shown in *Figure 7.17*:

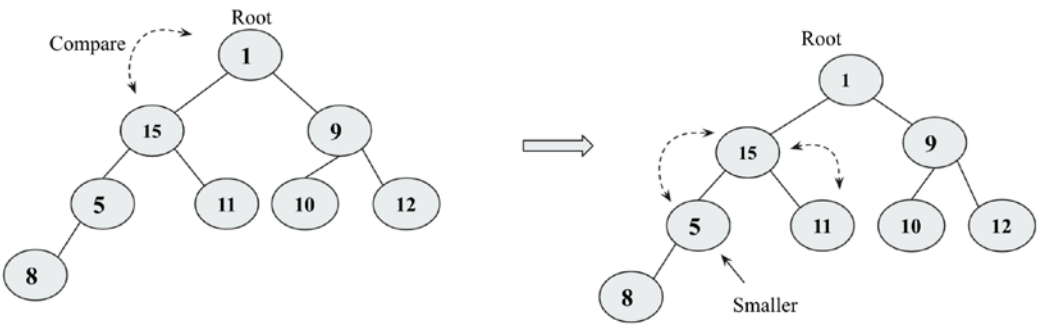

Figure 7.17: A comparison of node 15 with 5 and 11, and swapping node 15 and node 5

After swapping node 15 with node 5, we move down in the heap. Next, we compare node 15 with its child, node 8. Finally, node 8 and node 15 are swapped. Now, the final tree follows the heap property, as shown in *Figure 7.18*:

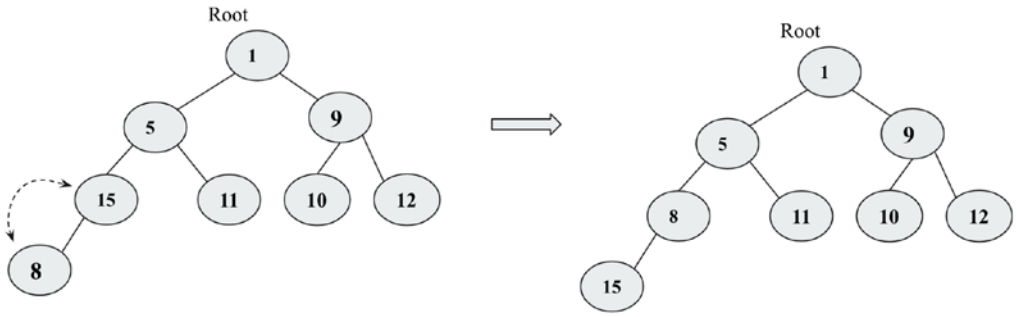

Figure 7.18: The final heap after swapping node 8 and node 15

The implementation of the delete operation for removing a data item at any given index location is given below, which should be defined in the `MinHeap` class:

```
    def delete_at_location(self, location):
        item = self.heap[location]
        self.heap[location] = self.heap[self.size]
        self.size -= 1
        self.heap.pop()
        self.sink(location)
        return item
```

This implementation is very similar to what we have seen in the previous section for deleting the root element. The only difference is that in this code, we have specified the index location that has to be deleted. The following code snippet demonstrates the deletion of a node at a specific location 2 from the heap created from data elements `{4, 8, 7, 2, 9, 10, 5, 1, 3, 6}`:

```
h = MinHeap()
for i in (4, 8, 7, 2, 9, 10, 5, 1, 3, 6):
    h.insert(i)
print(h.heap)

n = h.delete_at_location(2)
print(n)
print(h.heap)
```

The output of the preceding code is as follows:

```
[0, 1, 2, 5, 3, 6, 10, 7, 8, 4, 9]
2
[0, 1, 3, 5, 4, 6, 10, 7, 8, 9]
```

In the above output, we see that, before and after, the heap nodes are placed according to their index positions. We have discussed the concepts and implementation using examples of `min-heap`; all these operations and concepts can be easily implemented for a `max-heap` by simply reversing the logic in conditions where we ensured that the parent node should have smaller values compared to the children in `min-heap`. Now in the case of `max-heap`, we have to make the larger value in the parent. Heaps are used in various applications such as to implement heap sort and priority queues, which we will discuss in subsequent sections.

Heap sort

Heap is an important data structure for sorting a list of elements since it is very suitable for a large number of elements. If we want to sort a list of elements, say in ascending order, we can use `min-heap` for this purpose; we first create a `min-heap` of all the given data elements, and as per the `heap` property, the smallest data value will be stored at the root of the heap. With the help of the `heap` property, it is straightforward to sort the elements. The process is as follows:

1. Create a `min-heap` using all the given data elements.
2. Read and delete the root element, which is the minimum value. After that, copy the last element of the tree to the new root, and further reorganize the tree to maintain the `heap` property.
3. Now, we repeat *step 2* until we get all the elements.
4. Finally, we get the sorted list of elements.

The data elements are stored in the heap adhering to the `heap` property; whenever a new element is added or deleted, the `heap` property is maintained using the `arrange()` and `sink()` `helper` methods, respectively, as discussed in previous sections.

In order to implement heap sort using the heap data structure, first we create a heap with the data items `{4, 8, 7, 2, 9, 10, 5, 1, 3, 6}` using the below code (details of the creation of the heap are given in previous sections):

```
    h = MinHeap()
    unsorted_list = [4, 8, 7, 2, 9, 10, 5, 1, 3, 6]
    for i in unsorted_list:
        h.insert(i)
    print("Unsorted list: {}".format(unsorted_list))
```

In the above code, the `min-heap`, `h`, is created and the elements in `unsorted_list` are inserted. After each call to the `insert()` method, the heap order property is restored by the subsequent call to the `sink` method.

After creation of the heap, next, we read and delete the root element. In each iteration, we get the minimum value, and thus the data items in ascending order. The implementation of the `heap_sort()` method should be defined in the `minHeap` class (it uses the `delete_at_root()` method discussed in previous sections):

```
        def heap_sort(self):
            sorted_list = []
            for node in range(self.size):
                n = self.delete_at_root()
                sorted_list.append(n)

            return sorted_list
```

In the above code, we create an empty array, `sorted_list`, which stores all the data elements in sorted order. Then we run the loop for the number of items in the list. In each iteration, we call the `delete_at_root()` method to get the minimum value, which is appended to `sorted_list`.

Now we can use the heap sort algorithm using the following code:

```
print("Unsorted list: {}".format(unsorted_list))
print("Sorted list: {}".format(h.heap_sort()))
```

The output of the above code is as follows:

```
Unsorted list: [4, 8, 7, 2, 9, 10, 5, 1, 3, 6]
Sorted list: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

The time complexity to build the heap using the insert method takes `O(n)` times. Further, to reorganize the tree after deleting the root element takes `O(log n)` since we go from top to bottom in the heap tree, and the height of the heap is `log2(n)`, hence the complexity of rearranging the tree is `O(log n)`. So, overall, the worst-case time complexity of the heap sort is `O(n logn)`. Heapsort is very efficient in general, giving a worst-case, average-case and best-case complexity of `O(nlogn)`.

Priority queues

A priority queue is a data structure that is similar to a queue in which data is retrieved based on the **First In, First Out** (**FIFO**) policy, but in the priority queue, priority is attached with the data. In the priority queue, the data is retrieved based on the priority associated with the data elements, the data elements with the highest priority are retrieved before the lower priority data elements, and if two data elements have the same priority, they are retrieved according to the **FIFO** policy.

We can assign the priority of the data depending upon the application. It is used in many applications, such as CPU scheduling, and many algorithms also rely on priority queues, such as Dijkstra's shortest-path, A* search, and Huffman codes for data compression.

So, in the priority queue, the item with the highest priority is served first. The priority queue stores the data according to the priority associated with the data, so insertion of an element will be at a specific position in the priority queue. Priority queues can be considered as modified queues that return the items in the order of highest priority instead of returning the items in the **FIFO** order. A priority queue can be implemented by modifying an enqueue position by inserting the item according to the priority. It is demonstrated in *Figure 7.19*, in which given the queue, a new item 5 is added to the queue at a specific index (here assuming that the data items having higher values have higher priority):

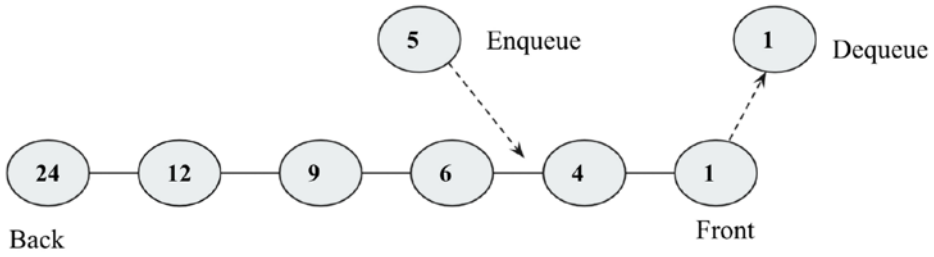

Figure 7.19: A demonstration of a priority queue

Let's understand the priority queue with an example. When we receive data elements in an order, the elements are enqueued in the priority queue in the order of priority (assuming that the higher data value is of higher importance). Firstly, the priority queue is empty, so 3 is added initially in the queue; the next data element is 8, which will be enqueued at the start since it is greater than 3. Next, the data item is 2, then 6, and finally, 10, which are enqueued in the priority queue as per their priority, and when the dequeue operation is applied, the high priority item will be dequeued first. All the steps are represented in *Figure 7.20*:

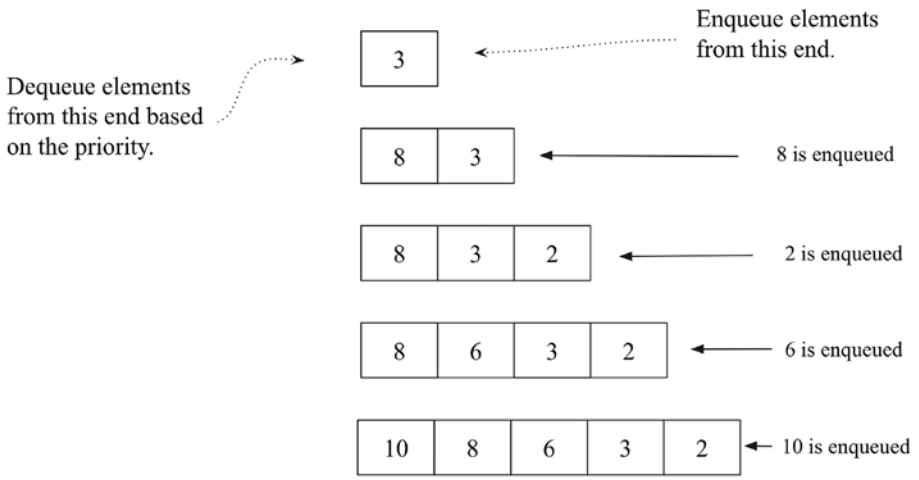

Figure 7.20: A step-by-step procedure to create a priority queue

Let us discuss the implementation of a priority queue in Python. We first define the node class. A node class will have the data elements along with the priority associated with the data in the priority queue:

```
# class for Node with data and priority
class Node:
  def __init__(self, info, priority):
    self.info = info
    self.priority = priority
```

Next, we define the `PriorityQueue` class and initialize the queue:

```
# class for Priority queue
class PriorityQueue:
  def __init__(self):
    self.queue = []
```

Next, let us discuss the implementation of the insertion operation for adding a new data element to the priority queue. In the implementation, we assume that the data element has high priority if it has a smaller priority value (for example, a data element with the priority value 1 has higher priority compared to the data element that has a priority value 4). The following are cases of insertion of elements in a priority queue:

1. Insertion of a data element to the priority queue when the queue is initially empty.
2. If the queue is not empty, we perform the traversal of the queue and reach the appropriate index position in the queue according to the associated priorities by comparing the priorities of the existing node with the new node. We add the new node before the node that has a priority greater than the new node.
3. If the new node has a lower priority than the high priority value, then the node will be added to the start of the queue.

The implementation of the `insert()` method is as follows, which should be defined in the `PriorityQueue` class:

```
    def insert(self, node):
        if len(self.queue) == 0:
            # add the new node
            self.queue.append(node)
        else:
            # traverse the queue to find the right place for new node
            for x in range(0, len(self.queue)):
                # if the priority of new node is greater
                if node.priority >= self.queue[x].priority:
                    # if we have traversed the complete queue
                    if x == (len(self.queue)-1):
                        # add new node at the end
                        self.queue.insert(x+1, node)
                    else:
                        continue
                else:
                    self.queue.insert(x, node)
                    return True
```

In the above code, we first append a new data element when the queue is empty, and then we iteratively reach the appropriate position by comparing the priorities associated with the data elements.

Next, when we apply the delete operation in the priority queue, the highest priority data element is returned and removed from the queue. It should be defined in the `PriorityQueue` class as follows:

```
    def delete(self):
        # remove the first node from the queue
        x = self.queue.pop(0)
        print("Deleted data with the given priority-", x.info, x.priority)
        return x
```

In the preceding code, we get the top element with the highest priority value. Further, the implementation of the `show()` method that prints all the data elements of the priority queue in the order of the priorities should be defined in the `PriorityQueue` class:

```
    def show(self):
        for x in self.queue:
            print(str(x.info)+ " - "+ str(x.priority))
```

Now, let's consider an example to see how to use the priority queue in which we firstly add data elements (`"Cat"`, "`Bat`", `"Rat"`, `"Ant"`, and `"Lion"`) with associated priorities `13`, `2`, `1`, `26`, and `25`, respectively:

```
p = PriorityQueue()
p.insert(Node("Cat", 13))
p.insert(Node("Bat", 2))
p.insert(Node("Rat", 1))
p.insert(Node("Ant", 26))
p.insert(Node("Lion", 25))
p.show()
p.delete()
```

The output of the above code is as follows:

```
Rat - 1
Bat - 2
Cat - 13
Lion - 25
Ant - 26
Deleted data with the given priority- Rat 1
```

Priority queues can be implemented using several data structures; in the above example, we saw its implementation using a list of tuples where the tuple contains the priority as the first element and the value data item as the next element. However, the priority queues are mostly implemented using a heap, since it is efficient with the worst-case time complexity of `O(log n)` in insertion and deletion operations.

The implementation of the priority queue using heap is very similar to what we have discussed in the `min-heap` implementation. The only difference is that now we store the priorities associated with the data elements, and we create a `min-heap` tree considering the priority values using a list of tuples in Python. For completeness, the code for the priority queue using heaps is as follows:

```
class PriorityQueueHeap:
    def __init__(self):
        self.heap = [()]
        self.size = 0

    def arrange(self, k):
        while k // 2 > 0:
            if self.heap[k][0] < self.heap[k//2][0]:
                self.heap[k], self.heap[k//2] = self.heap[k//2], self.heap[k]
            k //= 2

    def insert(self,priority, item):
        self.heap.append((priority, item))
        self.size += 1
        self.arrange(self.size)

    def sink(self, k):
        while k * 2 <= self.size:
            mc = self.minchild(k)
            if self.heap[k][0] > self.heap[mc][0]:
                self.heap[k], self.heap[mc] = self.heap[mc], self.heap[k]
            k = mc

    def minchild(self, k):
        if k * 2 + 1 > self.size:
            return k * 2
        elif self.heap[k*2][0] < self.heap[k*2+1][0]:
```

```
            return k * 2
        else:
            return k * 2 + 1

    def delete_at_root(self):
        item = self.heap[1][1]
        self.heap[1] = self.heap[self.size]
        self.size -= 1
        self.heap.pop()
        self.sink(1)
        return item
```

We use the code below to create a priority queue with data elements `"Bat"`, `"Cat"`, `"Rat"`, `"Ant"`, `"Lion"`, and `"Bear"` with the associated priority values 2, 13, 18, 26, 3, and 4, respectively:

```
h = PriorityQueueHeap()
h.insert(2, "Bat")
h.insert(13,"Cat")
h.insert(18, "Rat")
h.insert(26, "Ant")
h.insert(3, "Lion")
h.insert(4, "Bear")
h.heap
```

The output of the above code is as follows:

```
[(), (2, 'Bat'), (3, 'Lion'),  (4, 'Bear'),  (26, 'Ant'),  (13, 'Cat'),
(18, 'Rat')]
```

In the above output, we can see that it shows a `min-heap` tree that adheres to the `min-heap` property. Now we can use the code below to remove the data elements:

```
for i in range(h.size):
    n = h.delete_at_root()
    print(n)
    print(h.heap)
```

The output of the preceding code is as follows:

```
'Bat
[(), (3, 'Lion'), (13, 'Cat'), (4, 'Bear'), (26, 'Ant'), (18, 'Rat')]
```

```
Lion
[(), (4, 'Bear'), (13, 'Cat'), (18, 'Rat'), (26, 'Ant')]
Bear
[(), (13, 'Cat'), (26, 'Ant'), (18, 'Rat')]
Cat
[(), (18, 'Rat'), (26, 'Ant')]
Rat
[(), (26, 'Ant')]
Ant
[()]
```

In the above output, we can see that the data items are produced according to the priorities associated with the data elements.

Summary

In this chapter, we have discussed an important data structure, in other words, the heap data structure. We also discussed heap properties for `min-heap` and `max-heap`. We have seen the implementation of several operations that can be applied to the heap data structure, such as heapifying, and the insertion and deletion of a data element from the heap. We have also discussed two of the important applications of the heap—heap sort and a priority queue. The heap is an important data structure since it has many applications, such as sorting, selecting minimum and maximum values in a list, graph algorithms, and priority queues. Moreover, the heap can also be useful when we have to repeatedly remove a data object with the highest or lowest priority values.

In the next chapter, we will discuss the concepts of **Hashing** and **Symbol Tables**.

Exercises

1. What will be the time complexity for deleting an arbitrary element from the `min-heap`?
2. What will be the time complexity for finding the `kth` smallest element from the `min-heap`?
3. What will be the worst-case time complexity for ascertaining the smallest element from a binary `max-heap` and binary `min-heap`?
4. What will be the time complexity to make a `max-heap` that combines two `max-heap` each of size `n`?

5. The level order traversal of `max-heap` is 12, 9, 7, 4, and 2. After inserting new elements 1 and 8, what will be the final `max-heap` and the level order traversal of the final `max-heap`?
6. Which of the following is a binary `max-heap`?

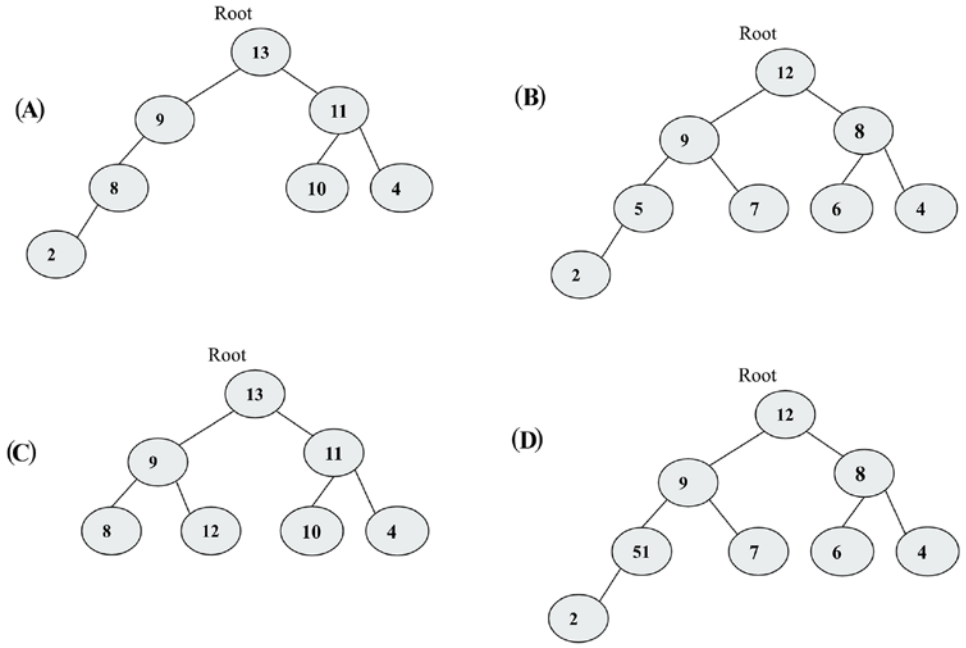

Figure 7.21: Example trees

Join our community on Discord

Join our community's Discord space for discussions with the author and other readers:

`https://packt.link/MEvK4`