

CHAPTER 12

Heaps

LEARNING OBJECTIVE

A heap is a specialized tree-based data structure. There are several variants of heaps which are the prototypical implementations of priority queues. We have already discussed priority queues in Chapter 8. Heaps are also crucial in several efficient graph algorithms. In this chapter, we will discuss three types of heaps—binary heaps, binomial heaps, and Fibonacci heaps.

12.1 BINARY HEAPS

A binary heap is a complete binary tree in which every node satisfies the heap property which states that:

If B is a child of A, then $\text{key}(A) \geq \text{key}(B)$

This implies that elements at every node will be either greater than or equal to the element at its left and right child. Thus, the root node has the highest key value in the heap. Such a heap is commonly known as a *max-heap*.

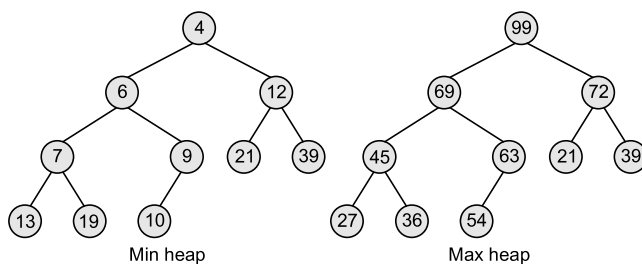


Figure 12.1 Binary heaps

Alternatively, elements at every node will be either less than or equal to the element at its left and right child. Thus, the root has the lowest key value. Such a heap is called a *min-heap*.

Figure 12.1 shows a binary min heap and a binary max heap. The properties of binary heaps are given as follows:

- Since a heap is defined as a complete binary tree, all its elements can be stored

sequentially in an array. It follows the same rules as that of a complete binary tree. That is, if an element is at position i in the array, then its left child is stored at position $2i$ and its right child at position $2i+1$. Conversely, an element at position i has its parent stored at position $i/2$.

- Being a complete binary tree, all the levels of the tree except the last level are completely filled.
- The height of a binary tree is given as $\log_2 n$, where n is the number of elements.
- Heaps (also known as partially ordered trees) are a very popular data structure for implementing priority queues.

A binary heap is a useful data structure in which elements can be added randomly but only the element with the highest value is removed in case of max heap and lowest value in case of min heap. A binary tree is an efficient data structure, but a binary heap is more space efficient and simpler.

12.1.1 Inserting a New Element in a Binary Heap

Consider a max heap H with n elements. Inserting a new value into the heap is done in the following two steps:

1. Add the new value at the bottom of H in such a way that H is still a complete binary tree but not necessarily a heap.
2. Let the new value rise to its appropriate place in H so that H now becomes a heap as well.

To do this, compare the new value with its parent to check if they are in the correct order. If they are, then the procedure halts, else the new value and its parent's value are swapped and Step 2 is repeated.

Example 12.1 Consider the max heap given in Fig. 12.2 and insert 99 in it.

Solution

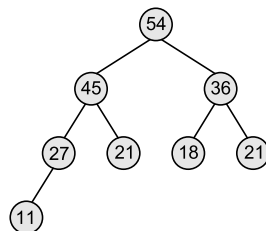


Figure 12.2 Binary heap

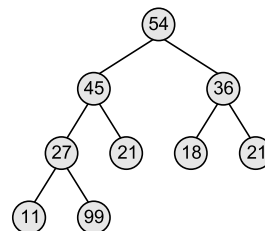


Figure 12.3 Binary heap after insertion of 99

The first step says that insert the element in the heap so that the heap is a complete binary tree. So, insert the new value as the right child of node 27 in the heap. This is illustrated in Fig. 12.3.

Now, as per the second step, let the new value rise to its appropriate place in H so that H becomes a heap as well. Compare 99 with its parent node value. If it is less than its parent's value, then the new node is in its appropriate place and H is a heap. If the new value is greater than that of its parent's node, then swap the two values. Repeat the whole process until H becomes a heap. This is illustrated in Fig. 12.4.

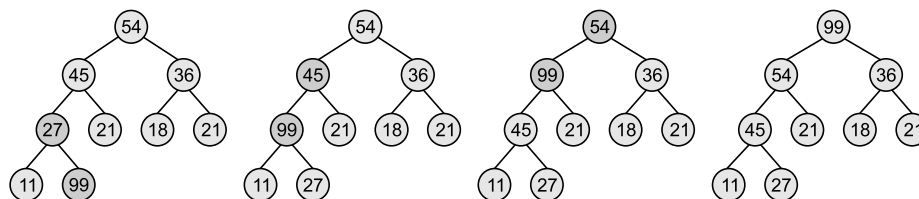


Figure 12.4 Heapify the binary heap

Example 12.2 Build a max heap H from the given set of numbers: 45, 36, 54, 27, 63, 72, 61, and 18. Also draw the memory representation of the heap.

Solution

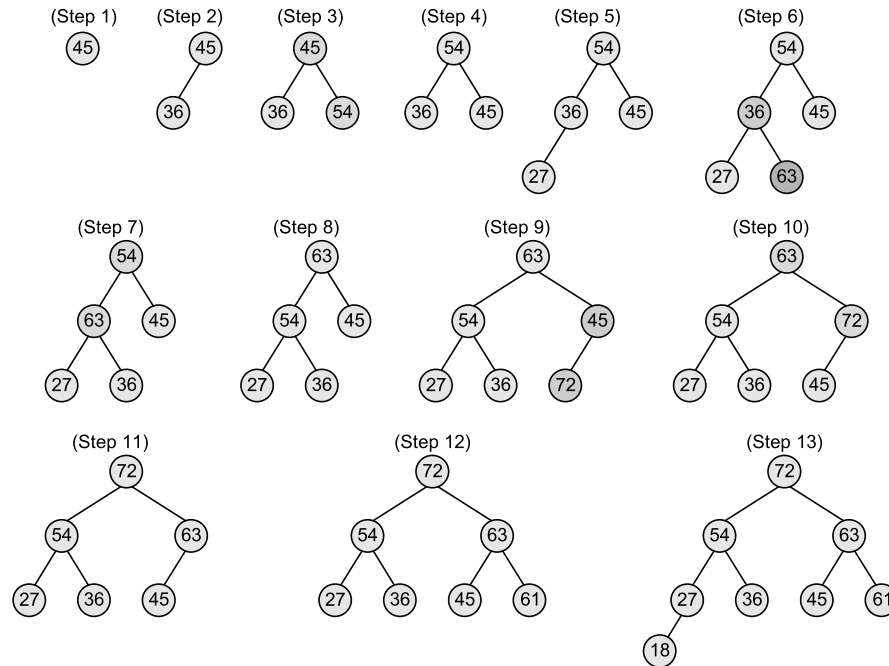


Figure 12.5

The memory representation of H can be given as shown in Fig. 12.6.

HEAP[1]	HEAP[2]	HEAP[3]	HEAP[4]	HEAP[5]	HEAP[6]	HEAP[7]	HEAP[8]	HEAP[9]	HEAP[10]
72	54	63	27	36	45	61	18		

Figure 12.6 Memory representation of binary heap H

After discussing the concept behind inserting a new value in the heap, let us now look at the algorithm to do so as shown in Fig. 12.7. We assume that H with n elements is stored in array $HEAP$.

```

Step 1: [Add the new value and set its POS]
        SET  $N = N + 1$ ,  $POS = N$ 
Step 2: SET  $HEAP[N] = VAL$ 
Step 3: [Find appropriate location of VAL]
        Repeat Steps 4 and 5 while  $POS > 1$ 
Step 4:   SET  $PAR = POS/2$ 
Step 5:   IF  $HEAP[POS] \leq HEAP[PAR]$ ,
           then Goto Step 6.
           ELSE
               SWAP  $HEAP[POS]$ ,  $HEAP[PAR]$ 
                $POS = PAR$ 
           [END OF IF]
        [END OF LOOP]
Step 6: RETURN
  
```

Figure 12.7 Algorithm to insert an element in a max heap

VAL has to be inserted in $HEAP$. The location of VAL as it rises in the heap is given by POS , and PAR denotes the location of the parent of VAL .

Note that this algorithm inserts a single value in the heap. In order to build a heap, use this algorithm in a loop. For example, to build a heap with 9 elements, use a `for` loop that executes 9 times and in each pass, a single value is inserted.

The complexity of this algorithm in the average case is $O(1)$. This is because a binary heap has $O(\log n)$ height. Since approximately 50% of the elements are leaves and 75% are in the bottom two levels, the new element to be inserted will only move a few levels upwards to maintain the heap.

In the worst case, insertion of a single value may take $O(\log n)$ time and, similarly, to build a heap of n elements, the algorithm will execute in $O(n \log n)$ time.

12.1.2 Deleting an Element from a Binary Heap

Example 12.3 Consider the max heap H shown in Fig. 12.8 and delete the root node's value.

Solution

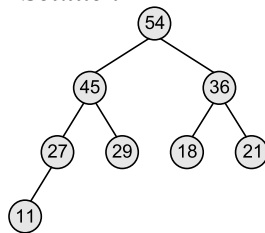


Figure 12.8 Binary heap

Consider a max heap H having n elements. An element is always deleted from the root of the heap. So, deleting an element from the heap is done in the following three steps:

1. Replace the root node's value with the last node's value so that H is still a complete binary tree but not necessarily a heap.
2. Delete the last node.
3. Sink down the new root node's value so that H satisfies the heap property. In this step, interchange the root node's value with its child node's value (whichever is largest among its children).

Here, the value of root node = 54 and the value of the last node = 11. So, replace 54 with 11 and delete the last node.

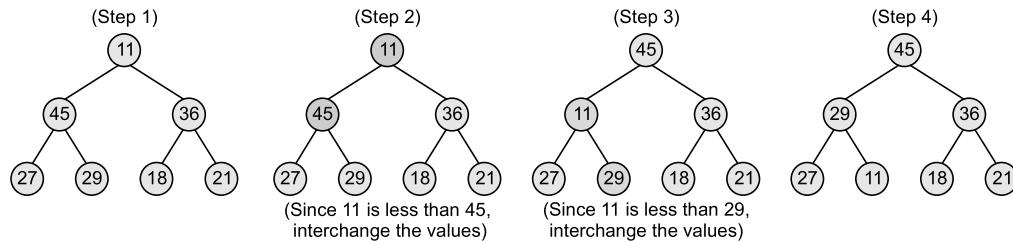


Figure 12.9 Binary heap

After discussing the concept behind deleting the root element from the heap, let us look at the algorithm given in Fig. 12.10. We assume that heap H with n elements is stored using a sequential array called `HEAP`. `LAST` is the last element in the heap and `PTR`, `LEFT`, and `RIGHT` denote the position of `LAST` and its left and right children respectively as it moves down the heap.

```

Step 1: [Remove the last node from the heap]
        SET LAST = HEAP[N], SET N = N - 1
Step 2: [Initialization]
        SET PTR = 1, LEFT = 2, RIGHT = 3
Step 3: SET HEAP[PTR] = LAST
Step 4: Repeat Steps 5 to 7 while LEFT <= N
Step 5: IF HEAP[PTR] >= HEAP[LEFT] AND
        HEAP[PTR] >= HEAP[RIGHT]
        Go to Step 8
        [END OF IF]
Step 6: IF HEAP[RIGHT] <= HEAP[LEFT]
        SWAP HEAP[PTR], HEAP[LEFT]
        SET PTR = LEFT
        ELSE
        SWAP HEAP[PTR], HEAP[RIGHT]
        SET PTR = RIGHT
        [END OF IF]
Step 7: SET LEFT = 2 * PTR and RIGHT = LEFT + 1
        [END OF LOOP]
Step 8: RETURN
  
```

Figure 12.10 Algorithm to delete the root element from a max heap

12.1.3 Applications of Binary Heaps

Binary heaps are mainly applied for

1. Sorting an array using *heapsort* algorithm. We will discuss *heapsort* algorithm in Chapter 14.
2. Implementing priority queues.

12.1.4 Binary Heap Implementation of Priority Queues

In Chapter 8, we learned about priority queues. We have also seen how priority queues can be implemented using linked lists. A priority queue is similar to a queue in which an item is



Figure 12.11 Priority queue visualization

dequeued (or removed) from the front. However, unlike a regular queue, in a priority queue the logical order of elements is determined by their priority. While the higher priority elements are added at the front of the queue, elements with lower priority are added at the rear.

Conceptually, we can think of a priority queue as a bag of priorities shown in Fig. 12.11. In this bag you can insert any priority but you can take out one with the highest value.

Though we can easily implement priority queues using a linear array, but we should first consider the time required to insert an element in the array and then sort it. We need $O(n)$ time to insert an element and at least $O(n \log n)$ time to sort the array. Therefore, a better way to implement a priority queue is by using a binary heap which allows both enqueue and dequeue of elements in $O(\log n)$ time.

12.2 BINOMIAL HEAPS

A binomial heap H is a set of binomial trees that satisfy the binomial heap properties. First, let us discuss what a binomial tree is.

A *binomial tree* is an ordered tree that can be recursively defined as follows:

- A binomial tree of order 0 has a single node.
- A binomial tree of order i has a root node whose children are the root nodes of binomial trees of order $i-1, i-2, \dots, 2, 1$, and 0.
- A binomial tree B_i has 2^i nodes.
- The height of a binomial tree B_i is i .

Look at Fig. 12.12 which shows a few binomial trees of different orders. We can construct a binomial tree B_i from two binomial trees of order B_{i-1} by linking them together in such a way that the root of one is the leftmost child of the root of another.

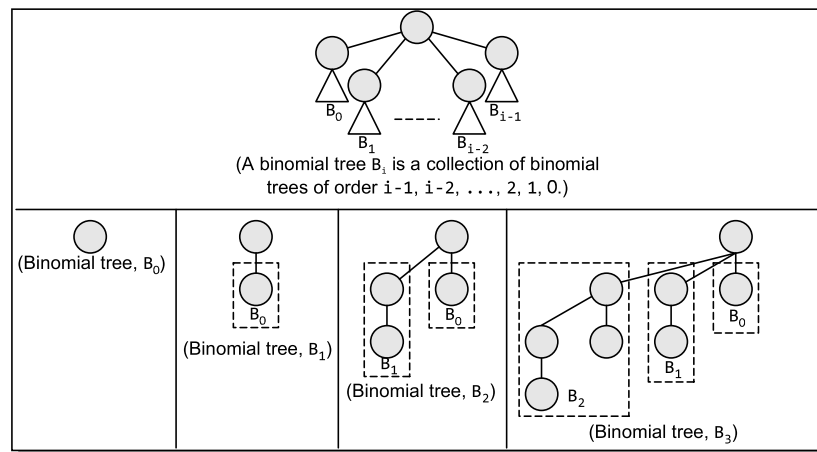


Figure 12.12 Binomial trees

A *binomial heap* H is a collection of binomial trees that satisfy the following properties:

- Every binomial tree in H satisfies the minimum heap property (i.e., the key of a node is either greater than or equal to the key of its parent).
- There can be one or zero binomial trees for each order including zero order.