Heaps

Manoj Gupta

August 15, 2016

In the following problem, you have to design a data-structure that can support the following operations:

1. INSERT(a) : Insert an integer a in the data-structure
2. DELETE-MIN() : Deletes the minimum integer from the current data-structure.

We first describe an algorithm that can insert an element in $O(1)$ time and delete-min in $O(n)$ time. This can be done using a linked list where an element can be added in the front to the head of the list. For DELETE-MIN(), we can scan the list while maintaining the minimum element. Thus, DELETE-MIN() can be executed in $O(n)$ time.

Now, we describe an algorithm that can insert an element in $O(n)$ time and delete-min in $O(1)$ time. To this end, we maintain an ordered or sorted list. Whenever we have to insert an element, we can traverse the list (using two pointers) and add the element in its correct position (after doing some pointer manipulation). So, insertion takes $O(n)$ time. For DELETE-MIN(), we can just remove the first element from the list. Thus DELETE-MIN() takes $O(1)$ time.

Thus, we have seen two data-structure whose running time are contrasting:

	INSERT(a)	DELETE-MIN()
List	$O(1)$	$O(n)$
Ordered List	$O(n)$	O(1)

A natural question arises whether we can insert and delete on $O(1)$ time. This is indeed a challenging problem and we will now describe a data-structure that can do both the operations in $O(\log n)$ time. Note that we sacrifice the $O(1)$ time for one of the operation but this leads to exponential improvement in the other operation.

To get $O(\log n)$ performance, we need to design a data-structure such that the "length of the data-structure" is $O(\log n)$. This is contrast with the above two data-structure whose length is $O(n)$, thats why their running time was $O(n)$.

However, this presents a unique challenge: how can the length of the data-structure is $O(\log n)$ when the number of elements in it is n. The answer to the above problem is a tree which is defined as follows:

1. Each tree has a unique root.
2. Each node in the tree has following fields.
   (a) value
   (b) at most one left child
   (c) at most one right child
   (d) at most one parent

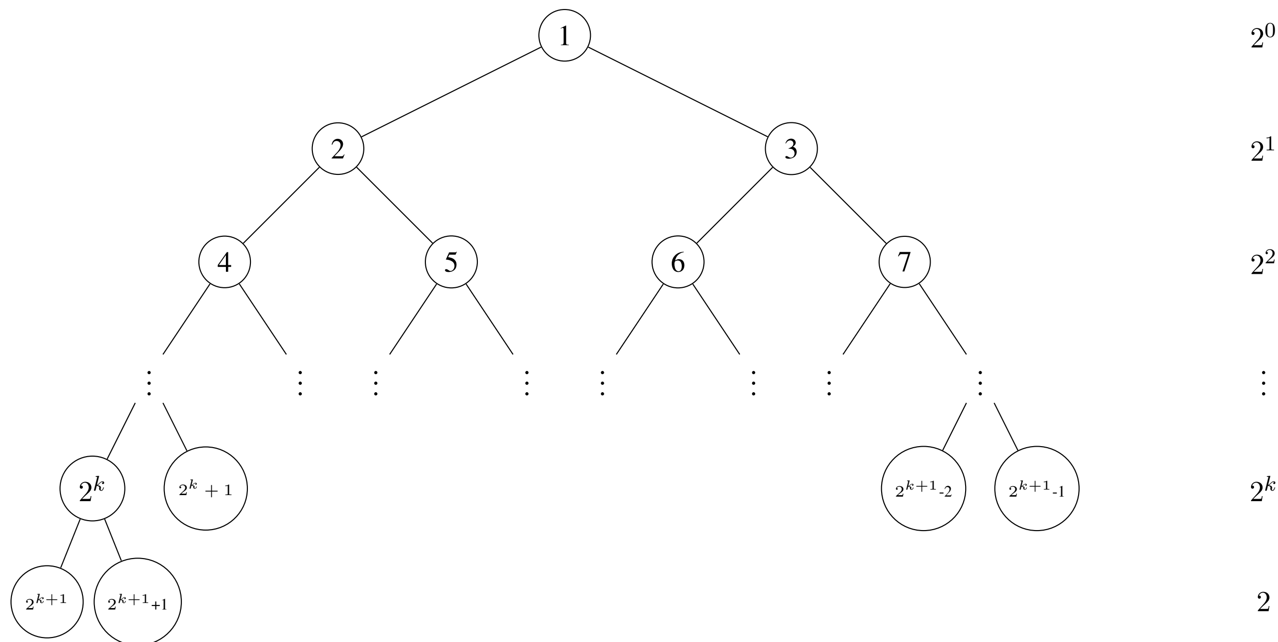

Figure 1: A heap which contains numbers $[1 \ldots 2^{k+1} + 1]$

If a node has no left or right child, then it is called a leaf. A node which is not a leaf is called an internal node. All nodes in the tree except the root has a unique parent.

3. For each internal node v, $v.value \leq (v.left).value$, if the left child of v exists. Similarly, $v.value \leq (v.right).value$, if the right child of v exists.

   The above property is called the heap property. Heap property implies that the root contains the minimum element of the heap.

4. All the levels i (except the last one) contains 2^i nodes. On the last level, the nodes are added from the left to right.

   One of the main requirement of our data-structure is that its "length" is $O(\log n)$. The above condition will be used crucially to prove this fact (See Lemma 0.1).

An example of heap can be seen in Figure 1. We now prove the following lemma:

Lemma 0.1. *The distance from a root to leaf in a heap is* $\leq O(\log n)$, *or the height of the heap is* $O(\log n)$.

Proof. Let us assume that we have heap that contains n elements at $k+1$ levels. Since condition (4) holds for the heap, all the levels i $(1 \leq i \leq k)$ are saturated. However, level $k+1$ might contain few elements. Let l be the total number of elements at level $k+1$. This implies that that the total number of elements in the heap is $\sum_{i=0}^{k} 2^k + l$. however the total number of elements in the heap is n. So $\sum_{i=0}^{k} 2^k + l = n \implies \sum_{i=0}^{k} 2^k \leq n \implies 2^{k+1} - 1 \leq n$. Thus $k+1 \leq \log(n+1)$. Thus the height of the heap is $O(\log n)$. $\square$

We now use heap to perform the two operations. Assume that we have a heap in Figure 2 and we want to now perform INSERT(4). To satisfy condition (4), we add a new node (with value 4) at the last level. However, the heap property (condition (3)) might not hold after this addition. We now shift up node 4 to the root trying to satisfy the heap property at each iteration. We first check if value at the parent of node 4 is greater than 4. If yes, then we swap 4 with the value at its parent node. This process goes on till we hit

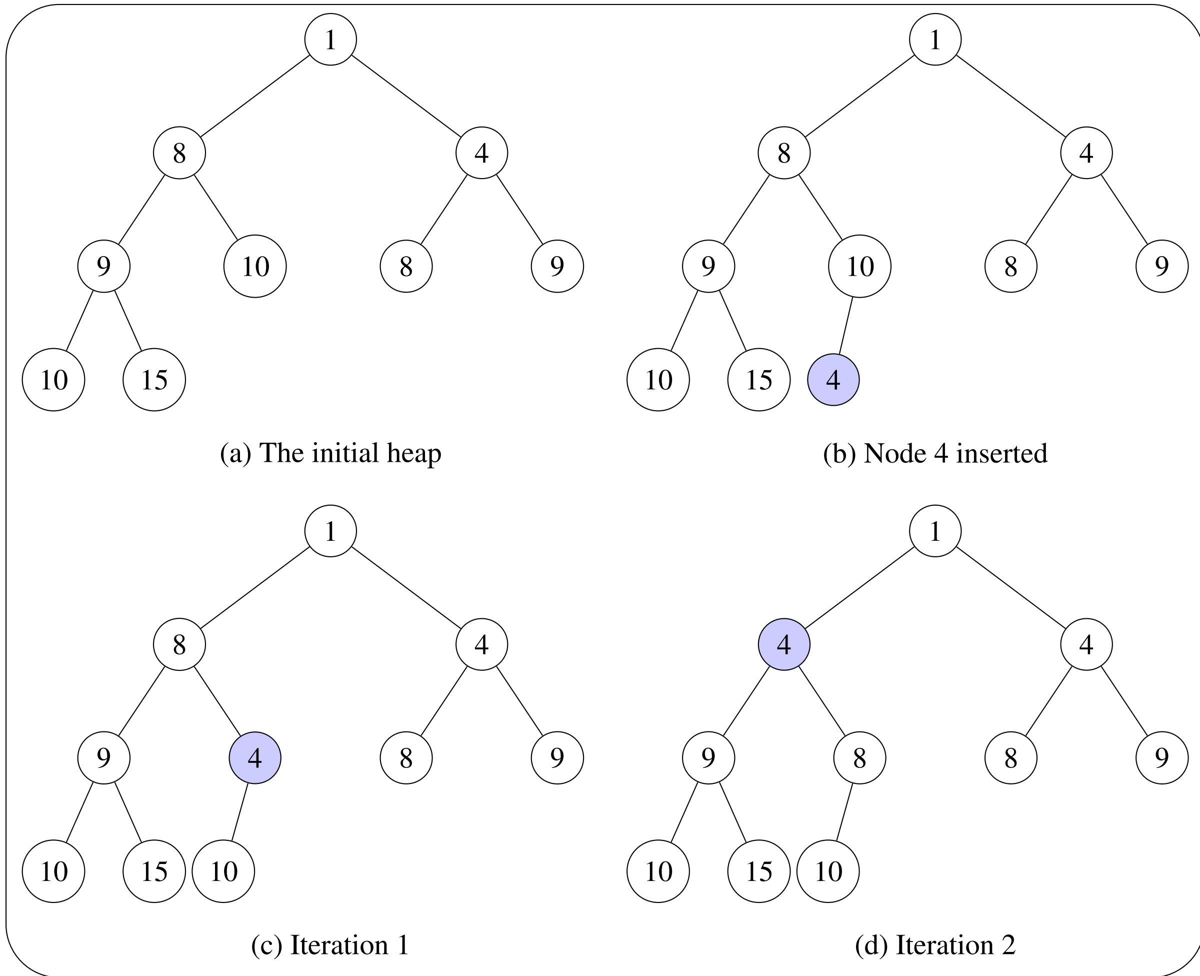

Figure 2: A run of our algorithm after INSERT(4)

the root or we arrive at a node such that the heap property holds at its root. Please refer to Figure 2 for a pictorial run of this algorithm.

The pseudo code for the algorithm is given below:

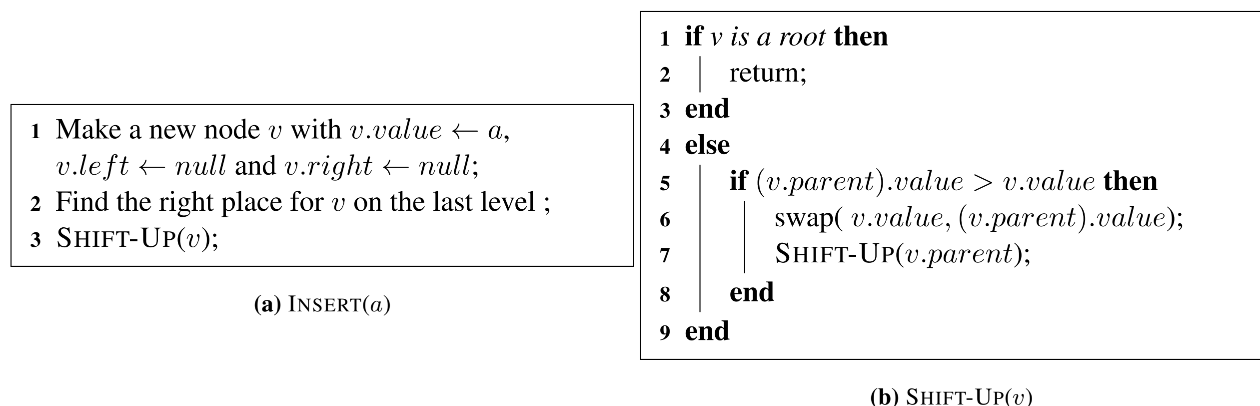

Figure 3: Inserting a node in the heap

Except step (2) in Figure (a) (of INSERT(a)), the running time of INSERT(a) depends on the running time of SHIFT-UP(v). SHIFT-UP(v) moves from the leaf to the root swapping the values parent and child if the heap property is not satisfied for the parent. Thus the running time of SHIFT-UP($\cdot$) depends on the height of the tree. Using lemma 0.1, we have seen that the height of the heap is $O(\log n)$. Thus we have shown the following lemma:

Lemma 0.2. *The running time of the procedure* INSERT*(a) in Figure 3(a) (except Step 2 in* INSERT*(a)) is* $O(\log n)$.

Let us now focus on the procedure DELETE-MIN(). From our construction, we see that root of the heap contains the minimum element. So, we can return this element. However, we have to delete this element and process the heap. To this end, we use the following procedure:

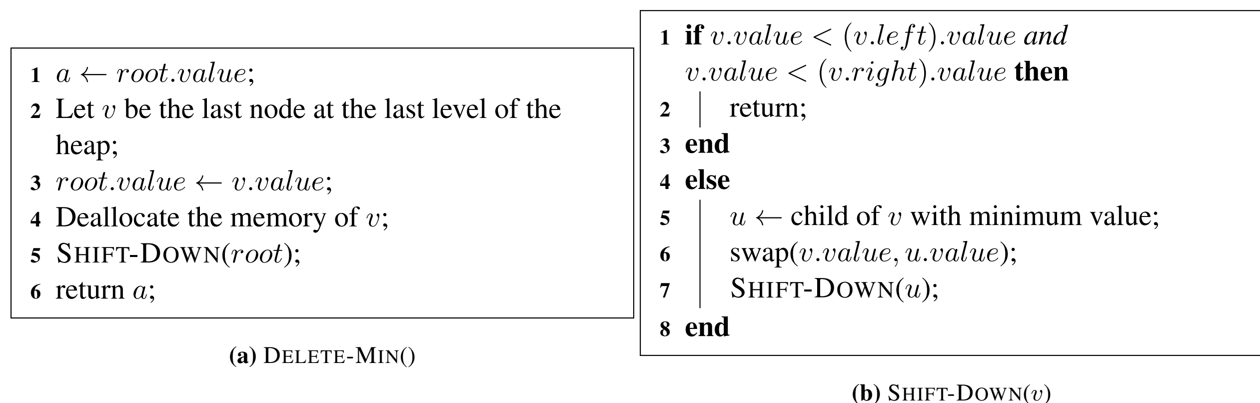

Figure 4: Deletion from the heap

In principle, the DELETE-MIN() procedure is same as the INSERT(a) procedure. However, the novel step in this procedure is to choose the last node at the last level of the heap and copy its value to the root. This is done to maintain the condition (4) of the heap. Like lemma 0.2, we can show the following lemma:

Lemma 0.3. *The running time of the procedure* DELETE-MIN*() in Figure 4(a) (except Step 2 in* DELETE-MIN*()) is* $O(\log n)$.

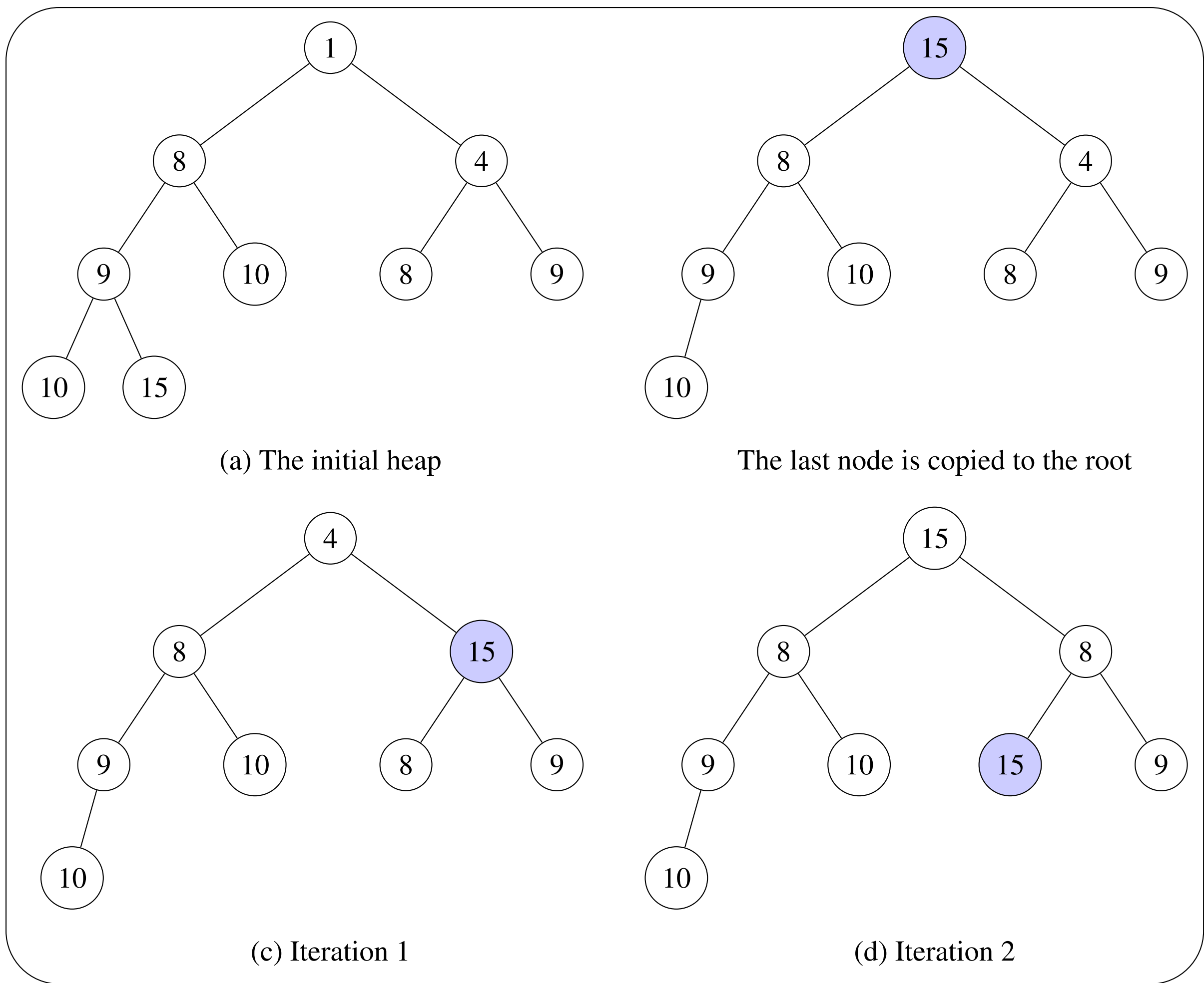

Figure 5: A run of our algorithm after DELETE-MIN()

A pictorial run of the procedure DELETE-MIN() is shown in Figure 5.