Heaps (Using Arrays)

Manoj Gupta

August 21, 2016

In the last lecture, we designed heaps that can perform 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.

If we know that the total number of inserts in the heap is n, then arrays becomes the choice of implementation. Arrays have an added advantage over the pointer based implementation of lists. Let $A[1 \ldots n]$ be the array which will store the array. Since the array is of size n, we will never be out of space for insertions. In addition, we will have a counter $countA$ which stores the number of elements in the array. We use the following arithmetic to represent the parent-child relationship:

1. If an elements is stored in $A[i]$, then its left and right child are stored in $A[2i]$ and $A[2i+1]$ respectively.
2. Similarly, the parent of an element stored in $A[i]$ $(i \neq 1)$ is $A[i/2]$.

An array and it implicit heap representation is shown in the figure below.

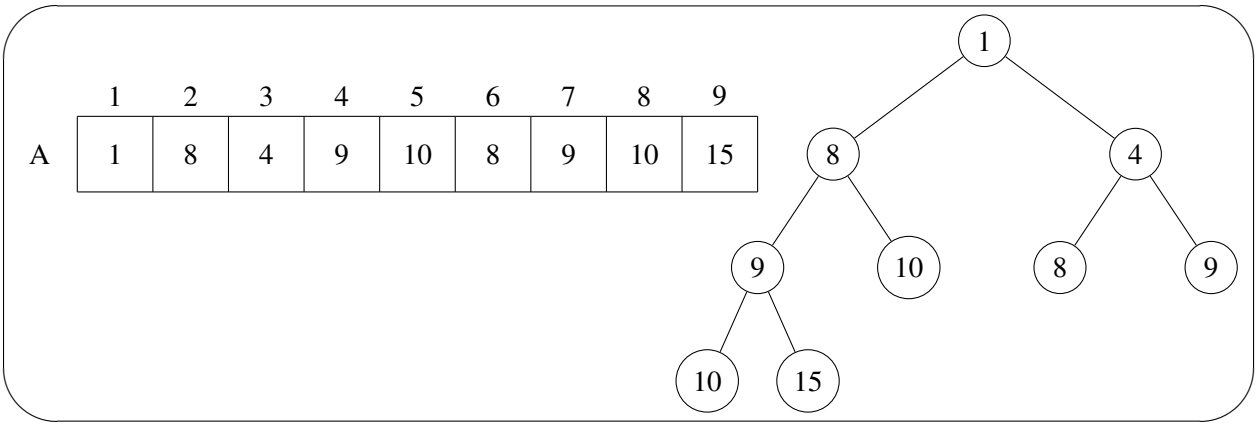

Figure 1: An array and its heap representation

The algorithm for inserting an element is same as in the pointer representation. However, there are several simplifications. To find the temporary position of this newly added element, we just increment the counter $countA$ and add this element at the last position in the array. After that the procedure is same as it counter-part in the pointer implementation, with the difference that we can move to parent of an element just using arithmetic manipulation in the array based implementation.

The pseudo code for the algorithm is given in Figure 2.

```
1 countA ← countA + 1;
2 A[countA] ← a;
3 SHIFT-UP(countA);
```

(a) INSERT(a)

```
1 if i = 1 then
2 |   return;
3 end
4 else
5 |   if A[i/2] > A[i] then
6 |   |   swap( A[i/2], A[i]);
7 |   |   SHIFT-UP(i/2);
8 |   end
9 end
```

(b) SHIFT-UP(i)

Figure 2: Inserting a node in the heap

Similarly, the algorithm for deleting the minimum is also same, with the difference that we can find the last node in the array in $O(1)$ time by using the counter $countA$.

```
1 a ← A[1];
2 A[1] ← A[countA];
3 countA ← countA − 1;
4 SHIFT-DOWN(1);
5 return a;
```

(a) DELETE-MIN()

```
1 if A[i] < A[2i] and A[i] < A[2i + 1] then
2 |   return;
3 end
4 else
5 |   j ← index of child of v with minimum value;
6 |   swap(A[i], A[j]);
7 |   SHIFT-DOWN(j);
8 end
```

(b) SHIFT-DOWN(i)

Figure 3: Deletion from the heap