HeapSort

Manoj Gupta

September 11, 2016

As it turns out, we already know of a data-structure that can sort an array. Consider the following algorithm that uses a *heap*.

```
foreach $i \leftarrow 1$ to $n$ do
    INSERT($A[i]$);
end
foreach $i \leftarrow 1$ to $n$ do
    print (DELETE-MIN());
end
```

Note that the first **for** loop adds n integers in the heap. So, the time taken for each insert is at most $O(\log n)$. Similarly, the time taken to delete the minimum element from a heap of size at most n is $O(\log n)$. Since we insert and delete at most n times, the running time of the above algorithm is $O(n \log n)$. The above algorithm sorts the array as the delete-min algorithm always outputs the minimum elements from the current set of elements.

However, for the above algorithm we are using extra space of $O(n)$ for our heap. As we have already seen that heaps can be implemented using an array too. So, can we build the heap with using only constant extra memory and perform heap sort?

Consider the following algorithm for building the heap **in-place**:

```
foreach $i \leftarrow n/2$ to 1 do
    SHIFT-DOWN($i$);
end
```

(a) BUILD-HEAP()

```
if $A[i] < A[2i]$ and $A[i] < A[2i+1]$ then
    return;
end
else
    $i' \leftarrow$ index of child of $v$ with least value;
    swap($A[i], A[i']$);
    SHIFT-DOWN($i'$);
end
```

(b) SHIFT-DOWN(i)

To understand the above algorithm, visualize the array to represent a nearly complete binary tree. We claim that in such a binary tree, all the nodes from $n/2+1$ to n are leaves. A leaf in itself satisfy the heap property which is stated below:

Property 0.1. *For any node v in the heap, the value at the left child of v and the value at the right child is $\leq$ value at v.*

For a leaf – which does not have any children – the heap property is trivially satisfied. Thus, we look at the first internal node in this implicit binary tree, that is $i \leftarrow n/2$. We perform SHIFT-DOWN operation on this node so that heap-property is satisfied for this node. However, when we shift-down, we need to recursively check that heap-property is satisfied for the node that is shift-down. We now show that the BUILD-HEAP() builds a valid heap.

Lemma 0.2. *After the ith iteration of* BUILD-HEAP*(), the subtree under $A[i], A[i+1], \ldots, A[n]$ is a heap.*

Proof. We prove by induction on i. The base case is when $i \in [n/2+1, n]$. Since we have argued that these nodes implicitly represent leaves, the heap-property is satisfied for these nodes. So the subtree under these nodes is trivially a heap.

Assume by induction that after the $(i+1)$th iteration, the subtree under $A[i+1], A[i], \ldots, A[n]$ is a heap. And we have to prove the statement of the lemma for iteration i. Consider the procedure SHIFT-DOWN(i). There are three cases to consider:

1. $A[i] \leq A[2i]$ and $A[i] \leq A[2i+1]$.

   By induction hypothesis, the subtree under $A[2i]$ and $A[2i+1]$ is a heap. This implies that $A[2i]$ and $A[2i+1]$ contains the minimum element of the heap under their respective tree. Thus $A[i]$ is the minimum element in tree rooted at $A[i]$. Thus subtree rooted under $A[i]$ is a heap.

2. $\exists$ a child of i, say $i_1 \in [2i, 2i+1]$, such that $A[i_1]$ is the least element in the set $\{A[2i], A[2i+1]\}$.

   In such a scenario, we swap $A[i]$ by $A[i_1]$. Thus $A[i]$ now contains the minimum element. However, $A[i_1]$ might not satisfy the heap property. To set this node right, we again perform SHIFT-DOWN(i_1). After this procedure is executed the heap property is satisfied for $A[i_1]$ but may not be satisfied for a child of i_1, say i_2. However, this process will not go on forever. That is, there exists an descendent of i , say i_k such that either condition 1 is satisfied for i_k or $A[i_k]$ is a leaf in the heap. In either case, the tree under $A[i]$ is a heap.

□

Let us now find the running time of BUILD-HEAP(). Note that the implicit binary tree represented by the array has $O(\log n)$ height (since the total number of elements in the array is n). Also, all the leaves in this tree can be at height $\log n$ or $\log n - 1$. For such a leaf, BUILD-HEAP() does no processing. Consider all the internal nodes in this tree at level $\log n - 1$. SHIFT-DOWN($\cdot$) on such a node can take c time (where c is come constant). The constant term c occurs due to the fact that SHIFT-DOWN($\cdot$) on such a node can shift down values at most once till a leaf is reached. Also, the total number of internal nodes at level $\log n - 1$ can at most be $2^{\log n - 1}$. Thus the total time taken by BUILD-HEAP() to process internal nodes at level $\log n - 1$ is $c \times 2^{\log n - 1}$. Once can similarly calculate that the total time taken by BUILD-HEAP() to process nodes at level $j (0 \leq j \leq \log n - 1)$ is $(\log n - j) c \times 2^j$. Thus the total time taken by BUILD-HEAP() is

$$S = c \sum_{j=0}^{\log n - 1} (\log n - j) \times 2^j \tag{0.1}$$

Dividing by 2 throughout,

$$S/2 = c \sum_{j=0}^{\log n - 1} (\log n - j) \times 2^{j-1} \tag{0.2}$$

Subtracting Equation 0.2 from Equation 0.1, we get:

$$S/2 = -\frac{\log n}{2} + \sum_{j=0}^{\log n - 1} 2^j = -\frac{\log n}{2} + 2^{\log n} - 1 = -\frac{\log n}{2} + n - 1.$$

Thus, $S = O(n)$ and the running time of BUILD-HEAP() is $O(n)$.

We are now ready to design our **in-place** algorithm fro heapsort.

```
1 BUILD-HEAP();
2 foreach $i \leftarrow 1$ to $n$ do
3     print (DELETE-MIN());
4 end
```

We have already seen that BUILD-HEAP() takes $O(n)$ time. However, steps 2-3 in the above algorithm take $O(n \log n)$ time. This is because we are deleting the minimum element from the heap n times and each such deletion takes $O(\log n)$ time. Thus the running time of heapsort is still $O(n \log n)$.

Let us now take a look at the worst case example for heap sort. Assume that the initial array $A = [1, 2, 3 \ldots, n]$. One can check this array represents a heap implicitly. Even BUILD-HEAP() will not swap any element in array as it is already a heap. Consider the first $n/2$ invocations of DELETE-MIN() from this heap. We claim that each such invocation will take $\log n$ time. Consider the first invocation of DELETE-MIN(). So the minimum element at the top of the heap $A[1] = 1$ is deleted from the heap and the last element of the heap $A[n] = n$ is moved to the top, that is, $A[1]$. We claim that SHIFT-DOWN(1) (See the heap lecture notes) takes $O(\log n)$ time as the value n should again move to the leaf. In fact, the above claim is true for the first $n/2$ deletions. Thus, the running time of heapsort after the first $n/2$ deletions is $\frac{n \log n}{2}$. Thus the running time of heapsort is at least $\approx n \log n$.

What about the best case and the average case running time of heapsort. Unfotunately, this is not such an easy question to answer. In [1], Robert Sedgewick and Russel Schaffer showed the the best case for heapsort also takes $O(n \log n)$ time. From the above two results, we claim that the average case of heapsort is $O(n \log n)$.

References

[1] The Best Case for Heapsort, Robert Sedgewick and Russel Schaffer
ftp://ftp.cs.princeton.edu/techreports/1990/293.pdf