Heapsort

September 27, 2017

1 Heapsort

Heapsort has the good properties of both merge sort and insertion sort.

- It has $O(n \log_2 n)$ worst-case running time.
- It is in-place and requires only a constant amount of extra storage.

It is based on a *data structure* known as a heap.

1.1 The abstract heap data structure

The (binary) heap data structure is an object that we can view as a *nearly complete binary tree*.

- Each node corresponds to an element.
- The tree is completely filled on all levels except possibly the lowest, which is filled from the left up to a point.
- For each node, the operations PARENT(), LEFT(), and RIGHT() give the parent node, left child node, and right child node respectively.

1.2 Implementation of a heap using arrays

Heaps are usually implemented using an array A with two attributes:

- length(A) gives the number of elements in the array, and
- heap-size(A) represents the number of elements in the heap that are stored in A.

So, $0 \leq$ heap-size(A) $\leq$ length(A), and only $A[1, \ldots, \text{heap-size(A)}]$ are considered valid elements of the heap (even though A may contain more elements).

If we index the array by $1, 2, \ldots, n$, and the root node has index 1, then we can implement

PARENT(i)
1 return $\lfloor i/2 \rfloor$

```
LEFT(i)
    return 2i
```

```
RIGHT(i)
    return 2i + 1
```

Viewing the heap as a tree, the *height* of a node in the heap is defined to be the number of edges on the longest simple downward path from the node to a leaf. The height of the heap is defined to be the height of its root.

1.3 Heap property

We are usually interested in heaps that satisfy a particular property. Depending on the property, the heap is called either a *max-heap* or a *min-heap*.

Definition 1 (Max-heap)**.** *A heap A is called a max-heap if the value at every node (except the root node) is less than or equal to the value at its parent. That is,*

$$A[PARENT(i)] \geq A[i] \quad \forall i > 1.$$

This is known as the "max-heap property". In particular, the largest element in a max-heap is stored at the root, and the subtree rooted at any node only contains values less that or equal to the value in that node.

A *min-heap* is similarly defined to have the "min-heap property"

$$A[PARENT(i)] \leq A[i] \quad \forall i > 1.$$

For the heapsort algorithm, we will use max-heaps. The key elements of the algorithm are

- the MAX-HEAPIFY procedure, which is used to maintain the max-heap property, and
- the BUILD-MAX-HEAP procedure, which produces a max-heap from an unordered input array.

Assume that we have a heap that is almost a max-heap, except for the root element. To make it a max-heap, we call the procedure MAX-HEAPIFY, whose inputs are an array A and an index i into the array. When called, MAX-HEAPIFY assumes that the binary trees rooted at $LEFT(i)$ and $RIGHT(i)$ are max-heaps, but that $A[i]$ might be smaller than its children. MAX-HEAPIFY lets the value at $A[i]$ move down the max-heap so that the subtree rooted at i becomes a max-heap.

Outline: At each step, the largest of the elements $A[i], A[LEFT(i)], A[RIGHT(i)]$ is determined, and its index is stored in $largest$.

- If $A[i]$ is largest, then the subtree rooted at node i is already a max-heap and the procedure terminates.
- Otherwise, one of the two children has the largest element, and $A[i]$ is swapped with $A[largest]$.

- Node i and its immediate children now satisfy the max-heap property. However, the node indexed by *largest* now has the original value $A[i]$, and thus that subtree might violate the max-heap property. So we call MAX-HEAPIFY recursively on that subtree.

MAX-HEAPIFY(A, i)

```
l = LEFT(i)
r = RIGHT(i)
largest = i
if (l ≤ heap-size(A) and A[l] > A[i]) {
    largest = l
}
if (r ≤ heap-size(A) and A[r] > A[largest]) {
    largest = r
}
if (largest ≠ i) {
    Swap A[i] and A[largest]
    MAX-HEAPIFY(A, largest)
}
```

1.4 Building a max-heap

We can use MAX-HEAPIFY in a bottom-up manner to convert an array $A[1, \ldots, n]$ into a max-heap. Clearly, all elements $A[i]$ for $i > PARENT(n)$ are leaves of the tree, and so are already 1-element max-heaps.

BUILD-MAX-HEAP(A)

```
heap-size(A) = length(A)
for (i = PARENT(length(A)), ..., 2, 1) {
    MAX-HEAPIFY(A, i)
}
```

1.5 Heapsort

Finally, we come to the heapsort algorithm.

- Use BUILD-MAX-HEAP to build a max-heap on the input array A of length n.
- Since the maximum element of the array is stored at the root $A[1]$, we can put it into its correct final position by swapping with $A[n]$.
- Now, discard this maximum element in $A[n]$ from the heap, by simply decreasing the heap-size attribute.
- The remainder is almost a max-heap, except at the root node. Make it a max-heap by calling MAX-HEAPIFY.
- Repeat.

HEAPSORT(A)

```
BUILD-MAX-HEAP(A)
for (i = length(A), ..., 3, 2) {
    swap A[1] and A[i]
    heap-size(A) = heap-size(A) - 1
    MAX-HEAPIFY(A, 1)
}
```