Heaps, Priority Queues, and Heapsort

CLRS: Ch 6

A *binary heap* is a binary tree satisfying the following shape and order properties.

Shape Property For each positive integer n, all n-node binary heaps have the same shape. To get from an $(n-1)$-node heap to an n-node heap, find a shallowest, leftmost, non-full node and add a new node as its child, a left child if possible.

Order Property Given any pair of parent-child nodes, the parent's key is less than or equal to the child's key.

We say that a node x has a *faulty parent* if the key of x's parent is greater than the key of x. A node x has a *faulty child* if the key of some child of x is less than the key of x. If x has 2 faulty children, the child with smaller key is the *most faulty*.
Efficiency of heap algorithms are based on two basic manoeuvers: (i) *up swapping*, i.e., swapping a node with a faulty parent with its parent, (ii) *down swapping*, i.e., swapping a node with faulty child(ren) with its most faulty child.

Observation
Let H be a heap-shape binary tree containing exactly one node x with faulty parent. Up-swapping x either restores heap order in H or make the new x the only node with faulty parent.
Let H be a heap-shape binary tree containing exactly one node x with faulty child(ren). Down-swapping x either restores heap order in H or make the new x the only node with faulty child(ren).

[PICTURES MISSING]

A *priority queue* is an Abstract Data Type (ADT) that supports 2 operations:

INSERT(k, H) - add a new item k into the queue H

EXTRACT-MIN(H) - delete the minimum item from queue H, returning its value.

We can implement a priority queue with balanced binary search trees, e.g., red-black trees, achieving time $O(\log n)$ for the two queue operations.

Another implementation uses binary heaps, achieving the same time $O(\log n)$ per operation.

Priority Queue Implementation using Heaps

INSERT(k, H): First, create a new node x containing key k. Second, insert node x as described into a heap, making sure to maintain the heap shape property. Finally, starting from x, repeatedly perform up-swapping until H obeys the heap order. E.g.,

[PICTURES MISSING]

EXTRACT-MIN(H): (Except when the heap H is trivial, EXTRACT-MIN will actually delete a node that is different from the root. The node to delete is one whose deletion results in a binary tree with heap shape property.)

First, remember the key of the root. Second, copy the key from the node to be deleted into the root. Third, delete that node. Fourth, starting at the root, repeatedly perform down-swapping until heap order is achieved. Finally, return with the remembered key. E.g.,

[PICTURES MISSING]

Correctness This follows from the Observation.

Running Time Analysis

The running time for both INSERT and EXTRACT-MIN is dominated by the time spent for up-swappings or down-swappings, which is proportional to the height of the heap. Since heaps are "almost complete" binary trees (by the heap shape property), their heights are $O(\log n)$. Therefore, both priortity queue operations take time $O(\log n)$ when implemented using binary heaps.

Heapsort

A priority queue Q can be used to sort n given numbers $A[1..n]$ as follows:

```
$Q \leftarrow$ empty priority queue
**for** $i \leftarrow 1$ **to** $n$ **do** INSERT($A[i]$, $Q$);
**for** $i \leftarrow 1$ **to** $n$ **do** $A[i] \leftarrow$ EXTRACT-MIN($Q$);
```

This algorithm takes time $O(n \log n)$ when Q is implemented using heap because:

(i) the first for loop inserts each number in time $O(\log n)$, using total time $n \times O(\log n) = O(n \log n)$;

(ii) the second for loop extracts each number in time $O(\log n)$, using total time $n \times O(\log n) = O(n \log n)$.

Theorem. n numbers can be sorted in time $O(n \log n)$. □

Notes

1. Binary-tree algorithms often work by climbing up or down along a root-to-leaf path. The efficiency of these algorithms thus relies on the tree having small height, either in the worst-case sense or amortized sense.

2. Heap algorithms work by having only one faulty point and then repeatedly applying a simple maneouver to move the faulty point along a root-to-leaf tree path until the fault is gone. This design idea is used in many efficient binary tree algorithms.

3. Any heap-shaped binary tree can be turned into a heap in $O(n)$ time.

4. Heaps admit efficient implementation using arrays: simply index the tree nodes in level order! Such an implementation has a smaller overhead when compared to

binary trees implementation using node cells and pointers. In the array implementation of heaps, the parent, left child, and right child of any node x can be found quickly by doing arithmetic on the array index of x.

5. Our heaps are *min-heaps*. We get *max-heaps* if the order property is defined using $\geq$ instead of $\leq$.

6. Binary heaps can be generalized to k*-ary heaps* for integers $k \geq 2$ in the natural way. In fact, binary heaps are simply 2-ary heaps.

7. Heaps support other operations efficiently, e.g., MINIMUM (for min-heaps), MAXIMUM (for max-heaps), DECREASE-KEY, INCREASE-KEY, DELETE.

8. Fibonacci heaps (CLRS Ch 19) are more efficient than binary heaps on some operations. They are also mergeable, i.e., one can efficiently combine two Fibonacci heaps into one.

9. The bound of $O(n \log n)$ for sorting n numbers is best possible. Every comparison-based sorting algorithm requires time $\Omega(n \log n)$. (See CLRS Ch 8.1.)