Chapter 6 Heapsort

By Y. MENG   email : y(dot)meng201011(at)gmail(dot)com

Section 1 Brief

✓ heapsort is a sorting algorithm
✓ Running time: $O(n\lg n)$ (~ merge sort)
✓ heapsort sorts in place: only a constant number of array elements are stored outside the input array at any time. (~ insertion sort)
✓ heapsort combines the better attributes of merge sort and insertion sort.
★ heapsort introduces another alg design technique: using a data structure, in this case one we call a "heap", to manage information.
★ useful data structures for heapsort:
  ★ heap
  ★ priority queue

△ The (binary) heap data structure is an array object that can be viewed as an almost-complete binary tree.
△ Two attributes:
  △ A.length - number of elements in the array.
  △ A.heap-size : how many elements in the heap are sorted within array A.
▴ Although A[1..A.length] may contain numbers, only the elements in A[1..A.heap-size], where $0 \le A.heap-size \le A.length$, are valid elements of the heap.
▴ The root of the tree is A[1], and given the index i of a node, we can easily compute the indices of its parent, left child, and right child.

PARENT(i)
  $return \lfloor \frac{i}{2} \rfloor$

LEFT(i)
  $return (2 * i)$

RIGHT(i)
  $return (2 * i + 1)$

■ *LEFT(i)*, *RIGHT(i)*, and *PARENT(i)* compute fast with binary representation implementation.
  □ the *LEFT* procedure can compute 2i in one instruction by simply shifting the binary representation of i left by one bit position.
  □ the *RIGHT* procedure can quickly compute 2i+1 by shifting the binary representation of i left by one bit position and then adding in a 1 as the low-order bit.
  □ the *PARENT* procedure can compute $\lfloor \frac{i}{2} \rfloor$ by shifting the binary representation of i right one bit position.
■ two kinds of binary heaps: ***max-heaps*** ($A[PARENT(i)] \ge A[i]$, maximum is root) and ***min-heaps*** ($A[PARENT(i) \le A[i]$, minimum is root).
■ **height (of a node in a heap)** : the number of edges on the longest simple downward path from the node to a leaf.
■ **height (of the heap)** : the height of the heap's root. $height = \Theta(\lg n)$
■ the basic operations on heaps run in time at most proportional on the height of the tree and thus take $O(\lg n)$ time.

⋄ The *MAX-HEAPIFY* procedure, which runs in $O(\lg n)$ time, is the key to maintaining the max-heap property.
⋄ The *BUILD-MAX-HEAP* procedure, which runs in linear time, produces a max-heap from an unsorted input array.
⋄ The *HEAPSORT* procedure, which runs in $O(n\lg n)$ time, sorts an array in place.
⋄ The *MAX-HEAP-INSERT*, *HEAP-EXTRACT-MAX*, *HEAP-INCREASE-KEY*, and *HEAP-MAXIMUM* procedures, which run in $O(\lg n)$ time, allow the heap data structure to implement a priority queue.

Let H be the height of the heap.
Two subtleties to beware of:
⋄ Be careful not to confuse the height of a node (longest distance from a leaf) with its depth (distance from the root). ⋄ If the heap is not a complete binary tree (bottom level is not full), then the nodes at a given level (depth) don't all have the same eigth. e.g., although all nodes at depth H have height 0, nodes at depth $(H-1)$ can have either height 0 or height 1.
⋄ There are at most $\lceil \frac{n}{2^{h+1}} \rceil$ nodes of height h.

Section 2 Maintaining the heap property

// Sink $A[i]$ as deep as possible.

MAX_HEAPIFY(A,i)

```
  $l = LEFT(i)$
  $r = RIGHT(i)$
  $if\, l \le A.heap - size\, and\, A[i] \le A[l]$
    $largest = l$
  $else\, largest = i$
  $if\, r \le largest\, and\, A[largest] \le A[r]$
    $largest = r$
  $if\, largest\, != i$
    $swap\, A[largest]\, and\, A[i]$
    $MAX\_HEAPIFY(A,\ largest)$
```

At each step, the largest of the elements A[i], A[LEFT(i)], and A[RIGHT(i)] is determined, and its index is stored in largest. If A[i] is largest, then the subtree rooted at i is already a max-heap. Otherwise, if either of the two children is the largest, then we swap it with A[i], which makes the *i*th element and its children satisfy the property, then repeat this procedure on subtree rooted at largest recursively.

The running time of *MAX-HEAPIFY* on the subtree of size n rooted at a given node i is $\Theta(1)$ time to fix up the relationships among the elements $A[i]$, $A[LEFT(i)]$, and $A[RIGHT(i)]$, plus the time to run *MAX-HEAPIFY* on a subtree rooted at one of the children of node i. The children's subtrees each have size at most $\frac{2n}{3}$ (worst case, a sorted array and the lowest level is exactly half full.)

- number of subtree is $num_{sub} = \frac{internal\_nodes}{2} + lowest\_nodes = \frac{2^h - 1}{2} + 2^{h-1} = 2^h - 1$
- total number of nodes is $num = internal\_nodes + lowest\_nodes = 2^h + 2^{h-1}$.
- Thus number of subtree is $num_{sub} = \frac{num_{sub}}{num} n \le \frac{2^h}{2^h + 2^{h-1}} n = \frac{2 * 2^{h-1}}{(2+1) * 2^{h-1}} n = \frac{2}{3} n$.

Therefore, the running time of *MAX-HEAPIFY* by the recurrence:

$T(n) \le T(\frac{2n}{3}) + \Theta(1) = O(\lg n)$.

Section 3 Building a heap

■ Use the procedure **MAX-HEAPIFY** in a bottom-up manner to convert an array $A[1..n]$, where $n = A.length$, into a max-heap.

- □ Elements in subarray $A[(\lfloor\frac{n}{2}\rfloor + 1)..n]$ are all leaves of the tree.
- □ Each leaf could be a 1-element heap to begin with.
- □ The procedure **BUILD-MAX-HEAP** goes through the remaining nodes of the tree and runs **MAX-HEAPiFY** on each one.

BUILD-MAX-HEAP(A)

```
$A.heap\_size = A.length$
$for\, i = \lfloor\frac{A.length}{2}\rfloor downto\ 1$
    $MAX\_HEAPIFY(A,\ i)$
```

Correctness

⋄ <u>Loop invariant:</u> At start of every iteration of **for** loop, each node (i + 1), (i + 2), ..., n is root of a max-heap. It is $(n - \lfloor\frac{n}{2}\rfloor)$ many times.

⋄ <u>Initialization:</u> Nodes $\lfloor\frac{n}{2}\rfloor + 1$, $\lfloor\frac{n}{2}\rfloor + 2$, .., n are leaves, which is the root of a trivial max-heap (1-element max-heap). Since $i = \lfloor\frac{n}{2}\rfloor$ before the first iteration of the **for** loop, the invariant is initially TRUE.

⋄ <u>Maintenance:</u> Children of node i are indexed higher than i, so by the loop invariant, they are both roots of max-heaps. Correctly assuming that $(i + 1), (i + 2), \cdots, n$ are all roots of max-heaps, **MAX-HEAPIFY** makes node i a max-heap root. Decrementing i reestablishes the loop invariant at each iteration.

⋄ <u>Termination:</u> When $i = 0$, the loop terminates. By the loop invariant, each node, notably node 1, is the root of a max-heap.

Analysis

Simple instituition:

■ Each call to **MAX-HEAPIFY** costs $O(\lg n)$ times.

■ **BUILD-MAX-HEAP** calls **MAX-HEAPIFY** for $O(n)$ times.

$**$ Total running time: $O(n\lg n)$.

Computing tight upper bound:

□ The time for **MAX-HEAPIFY** to run at a node varies with the height of the node in the tree, and the heights of most nodes are small.

□ An n-element heap has height $\lfloor\lg n\rfloor$ and at most $\lceil\frac{n}{2^{h+1}}\rceil$ nodes of any height h.

□ The time required by **MAX-HEAPIFY** when called on a node of height h is $O(h)$.

■ Total cost of running **BUILD-MAX-HEAP** is:

$$\sum_{h=0}^{\lfloor lgn\rfloor}\lceil\frac{n}{2^{h+1}}\rceil O(h) = O(n\sum_{h=0}^{\lfloor lgn\rfloor}\lceil\frac{h}{2^h}\rceil) = O(n).$$

★ The last summation can be evaluated substituting $x = \frac{1}{2}$ in the formula $(\sum_{k=0}^{\infty}\frac{k}{(x^k} = \frac{x}{(1-x)^2}, for\, |x| < 1)$, which yields $\sum_{h=0}^{\infty}\lceil\frac{h}{2^h}\rceil = \frac{1/2}{(1-1/2)^2} = 2$

Section 4 The heapsort algorithm

Given an input array, the **heapsort algorithm** acts as follows:
◻ Builds a max-heap from the array, using **BUILD-MAX-HEAP**.
◻ Starting with the root (the maximum element), the algorithm places the maximum element into the correct place in the array by swapping it with the element in the last position in the array.
◻ Discard this last node (knowing that it is correct position) by decreasing the heap size, and calling **MAX-HEAPIFY** on the new root.
◻ Reapet this discarding process until only one node (the samllest element) remains, and therefore is in the correct position in the array.

```
HEAPSORT(A, n) // n ~ A.length
  for i ← n downto 2 // i ~ A.heap-size
    swap A[1] and A[i]
    A.heap_size = A.heap_size − 1
    MAX_HEAPIFY(A, 1)
```

Analysis
★ Running time of MAX-HEAPITY is $O(\lg n)$.
★ HEAPSORT calls MAX-HEAPIFY $(n - 1)$ times.
★ Running time of HEAPSORT is $O(n \lg n)$.

```
In [ ]:
```