MA 252: Data Structures and Algorithms

Lecture 11

http://www.iitg.ernet.in/psm/indexing_ma252/y12/index.html

Partha Sarathi Mandal

Dept. of Mathematics, IIT Guwahati

Heap Sort

HeapSort(A)

```
1.    Build-Max-Heap(A)
2.    for i ← length(A) downto 2
3.        do swap A[1] ↔ A[i]
4.            heap-size[A] ← heap-size[A] – 1
5.            Max-Heapify(A,1)
```

Build-Max-Heap(A)

```
1. heapsize(A) ← length(A)
2. for i ← ⌊length(A)/2⌋ downto 1
3.       do Max-Heapify(A, i)
```

Max-Heapify(A,i)

```
0.  left ← 2i
1.  right ← 2i + 1
     ▶indices of left & right children of A[i]
2.  largest ← i
3.  if left ≤ heapsize(A) and A[left] > A[i]
4.       then largest ← left
5.  if right ≤ heapsize(A) and A[right] > A[largest]
6.       then largest ← right
7.  if largest ≠ i
8.       then exchange A[i] ↔ A[largest])
9.       Max-Heapify(A, largest)
```

Max-Heapify: Running Time

Running Time of Max-Heapify:

- Every line is $\Theta(1)$ time -- except the recursive call
- In the worst-case of the recursion, Max-Heapify takes $O(h)$ time when node A[i] has height h in the heap, therefore
- $T(n) = O(\lg n)$

Build-Max-Heap

- **Intuition:** uses Max-Heapify in a bottom-up manner to convert unordered array A into a heap.
- Key point is that the leaves are already heaps. Elements A[($\lfloor n/2 \rfloor$ + 1) ... n] are all leaves.
- So the work starts at parents of leaves...then, grandparents of leaves...etc.

Build-Max-Heap(A)

1. **heapsize(A)** $\leftarrow$ **length(A)**
2. for $i \leftarrow \lfloor length(A)/2 \rfloor$ downto 1
3. do **Max-Heapify(A, i)**

Build-Max-Heap

Running Time of Build-Max-Heap

- Approximately $n/2$ calls to Max-Heapify ($O(n)$ calls)
- Simple upper bound: Each call takes $O(\lg n)$ time & $O(n\lg n)$ time total.
- Is it possible to make some tighter bound for Build-Max-Heap?
- What is the tighter bound for Build-Max-Heap ?
- Answer is $O(n)$.
- Now question is how ?

```
Build-Max-Heap(A)
heapsize(A) ← length(A)
for i ← ⌊length(A)/2⌋ downto 1
    do Max-Heapify(A, i)
```

Build-Max-Heap

- Proof of tighter bound ($O(n)$) relies on following theorem:
- **Theorem 1:** The number of nodes at height h in a maxheap $\lceil n/2^{h+1} \rceil$.

  Height of a node = longest distance from a leaf.

  Depth of a node = distance from the root.

  – Let H be the height of the tree. If the heap is not a complete binary tree (because the bottom level is not full), then the nodes at a given depth don't all have the same height. Eg., although all the nodes with depth H have height 0, the nodes with depth H-1 may have either height 0 or 1.

Build-Max-Heap(A)

1. **heapsize(A) ← length(A)**
2. for $i \leftarrow \lfloor length(A)/2 \rfloor$ downto 1
3. do **Max-Heapify(A, i)**

Theorem : The number of nodes at height h in a maxheap $\lceil n/2^{h+1} \rceil$.

Proof: Let H be the height of the heap.

The proof is by induction on h, the height of each node. The number of nodes in the heap is n.

Basis: Show the thm holds for nodes with $h = 0$. The tree leaves (nodes at height 0) are at depths H and H-1.

Let x be the number of nodes on the (possibly incomplete) lowest level of the heap.

Note that n-x is odd, since the n-x nodes above the last row of the tree form a complete binary tree, which has an odd number of nodes.

Therefore, if n is even, x is odd, and if n is odd, x is even.

Proof of the *Thm.*

- If *x* is even, then there are x/2 nodes at depth H - 1 that are parents of depth H nodes, so there are 2^{H-1} - x/2 nodes at depth H-1 that are not parents of depth H nodes. Thus the total number of height-0 nodes is
  $x + 2^{H-1} - x/2 = 2^{H-1} + x/2 = (2^H+x)/2 = \lceil (2^H+x-1)/2 \rceil = \lceil n/2 \rceil$
- If *x* is odd, then by a similar argument to the even case we obtain that the total number of height 0 nodes is
  $x + 2^{H-1} - (x+1)/2 = 2^{H-1} + (x-1)/2 = (2^H+x-1)/2 = \lceil n/2 \rceil$
- Thus, the # of leaves = $\lceil n/2^{0+1} \rceil$ and the thm holds for the base case.

Proof of the *Thm*. Contd…

Inductive step: Show that if the thm holds for height $h\text{-}1$, it holds for h.

Let n_h be the number of nodes at height h in the n-node tree T.

Consider the tree T' formed by removing the leaves of T. It has $n' = n - n_0$ nodes. We know from the base case that $n_0 = \lceil n/2 \rceil$, so $n' = n - \lceil n/2 \rceil = \lfloor n/2 \rfloor$

Note that the nodes at height h in T would be at height h-1 if the leaves of the tree were removed--i.e., they are at height h-1 in T'. Letting n'_{h-1} denote the number of nodes at height h-1 in T', we have $n_h = n'_{h-1}$

$n_h = n'_{h-1} \leq \lceil n'/2^h \rceil$ (by the IHOP) $= \lceil \lfloor n/2 \rfloor / 2^h \rceil \leq \lceil (n/2)/2^h \rceil = \lceil n/2^{h+1} \rceil$

Proof of the *Thm*. Contd…

- Since the time of Max-Heapify when called on a node of height h is $O(h)$, the time of B-M-H is

$$\sum_{h=0}^{\lg n} \frac{n}{2^{h+1}} O(h) = O(n \sum_{h=0}^{\lg n} \frac{h}{2^h})$$

- and since the last summation turns out to be a constant, the running time is $O(n)$.
- Therefore, we can build a max-heap from an unordered array in linear time.

Running time of HeapSort

We'll see that

- *Build-Max-Heap*(A) takes
  $O(|A|) = O(n)$ time
- *Max-Heapify*(A,1) takes
  $O(\lg|A|) = O(\lg n)$ time

Running time of *HeapSort*:

- One call to *Build-Max-Heap*()
  $\Rightarrow O(n)$ time
- *n-1* calls to *Max-Heapify*() each takes $O(\lg n)$ time
  $\Rightarrow O(n\lg n)$ time

Heap Sort

HeapSort(A)

1. **Build-Max-Heap(A)** // O(n)
2. **for** i ← length(A) **downto** 2 //O(n)
3. **do** swap A[1] ↔ A[i] //O(n)
4. heap-size[A] ← heap-size[A] – 1 //O(n)
5. **Max-Heapify(A,1)** // O($n$lg n)

Build-Max-Heap(A)
1. heapsize(A) ← length(A)
2. for i ← ⌊$length$(A)/2⌋ downto 1
3. do Max-Heapify(A, i)

Max-Heapify(A,i)
0. left ← 2i
1. right ← 2i + 1
▶indices of left & right children of A[i]
2. largest ← i
3. if left ≤ heapsize(A) and A[left] > A[i]
4. then largest ← left
5. if right ≤ heapsize(A) and A[right] > A[largest]
6. then largest ← right
7. if largest ≠ i
8. then exchange A[i] ↔ A[largest])
9. Max-Heapify(A, largest)

Heapsort Time and Space Usage

- An array implementation of a heap uses $O(n)$ space
  - one array element for each node in heap.
- Heapsort uses $O(n)$ space and is **in place.**
- Running time is as good as merge sort, $O(n\lg n)$ in worst case.