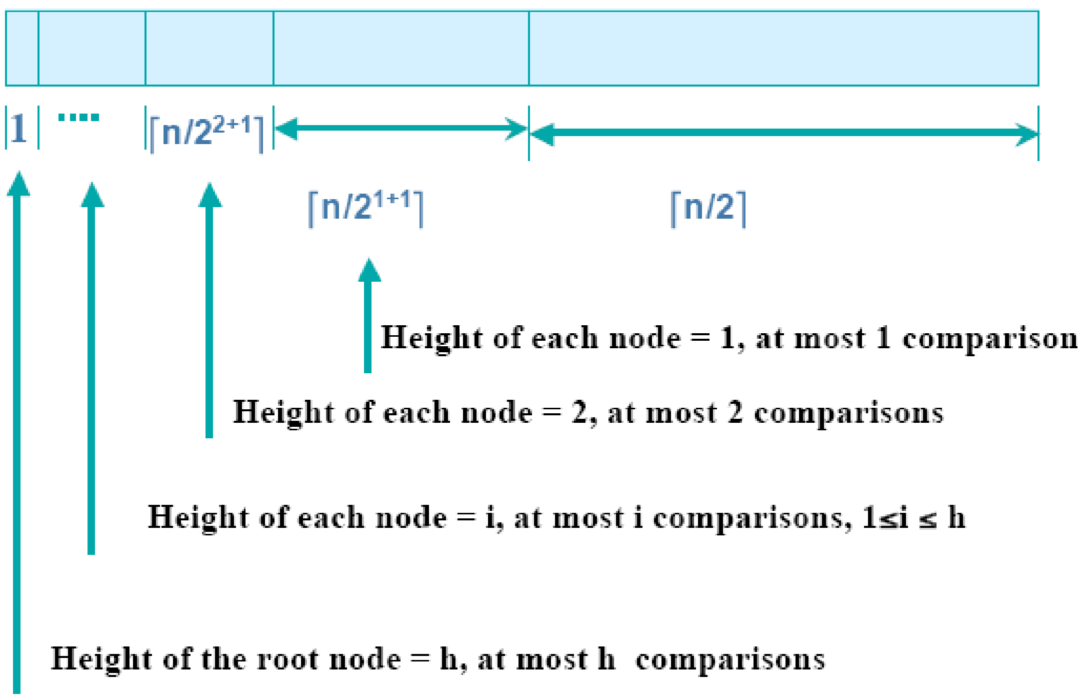
Analysis of Build Max Heap Algorithm

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.

- n-element heap has height $\lfloor \lg n \rfloor$

    and

- at most $\lceil n/2^{h+1} \rceil$ nodes of any height h.



Complexity analysis of Build-Heap (1)

- For each height $0 < h \leq \lg n$, the number of nodes in the tree is at most $n/2^{h+1}$
- For each node, the amount of work is proportional to its height h, $O(h) \rightarrow n/2^{h+1} .O(h)$
- Summing over all heights, we obtain:

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

Complexity analysis of Build-Heap (2)

- We use the fact that $\sum_{k=0}^{\infty} kx^k = \frac{x}{(1-x)^2} \quad for\ |x|<1$

$$\sum_{h=0}^{\infty} \left\lceil \frac{h}{2^h} \right\rceil = \frac{1/2}{(1-1/2)^2} = 2$$

- Therefore:

$$T(n) = O\left(n \sum_{h=0}^{\lfloor \lg n \rfloor} \left\lceil \frac{h}{2^{h+1}} \right\rceil \right) = O\left(n \sum_{h=0}^{\infty} \left\lceil \frac{h}{2^h} \right\rceil \right) = O(n)$$

- Building a heap takes only linear time and space!

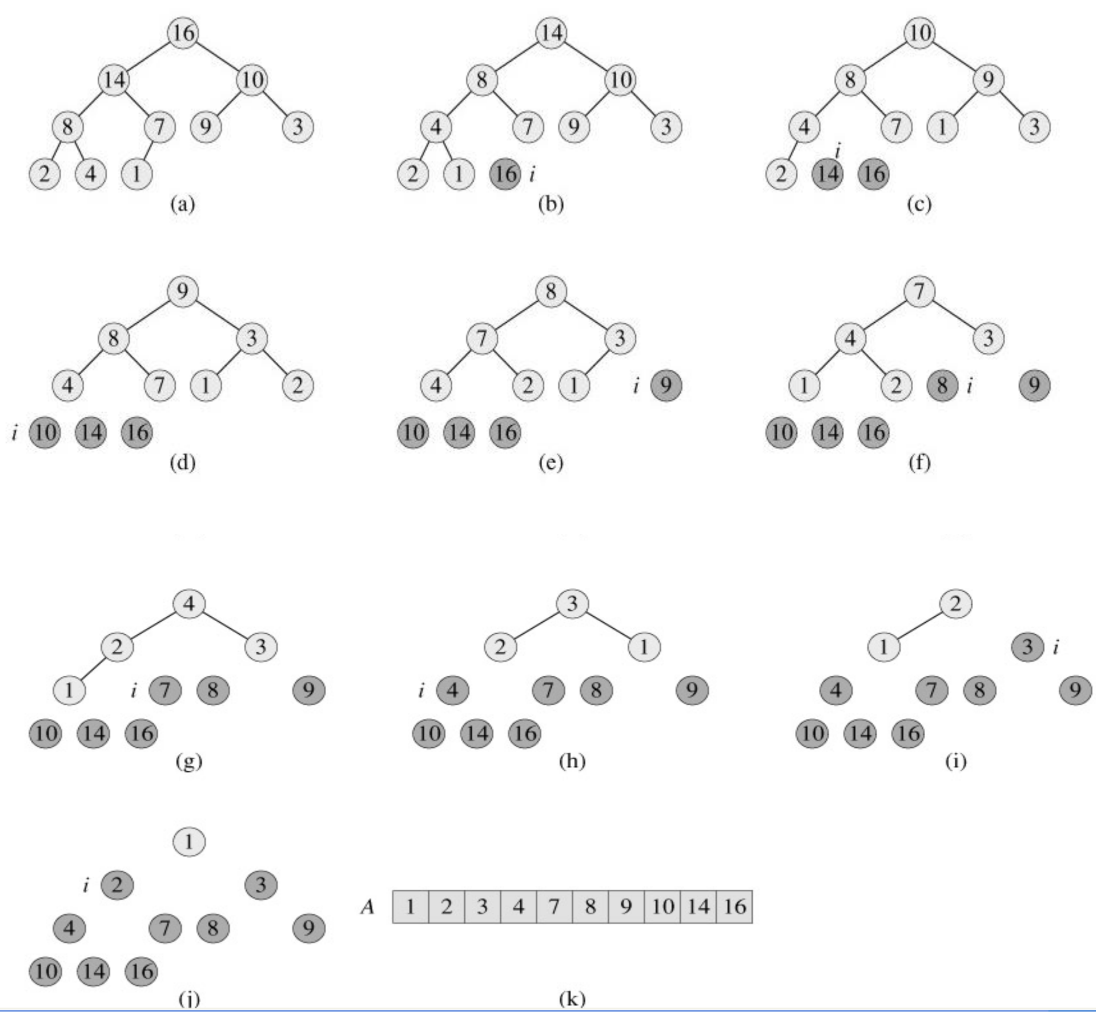
The HEAPSORT Algorithm

Input : Array A[1…n], n = A.length

Output : Sorted array A[1…n]

```
HEAPSORT(A)
1.      BUILD_MAX_HEAP[A]
2.      for  i = A.length down to 2
3.              exchange A[1] with A[i]
4.              A.heap_size = A.heap_size - 1;
5.              MAX_HEAPIFY(A, 1)
```

The operation of HEAPSORT



Heapsort Complexity

Running Time:

Step1 : BUILD_MAX_HEAP takes O(n)

Step 2 to 5 : MAX_HEAPIFY takes O(log n) and there are (n -1) calls

Running Time is O(n log n)

Priority Queues

- Heap data structure itself has many uses.
- One of the most popular applications of a heap: its use as an efficient priority queue.
- As with heaps, there are two kinds of priority queues:

  - max-priority queues

  - min-priority queues
- We will focus here on how to implement max-priority queues, which are in turn based on max-heaps
- ***priority queue*** is a data structure for maintaining a set S of elements, each with an associated value called a ***key***. A ***max-priority queue*** supports the following operations.
- INSERT(S, x) inserts the element x into the set S. This operation could be written as $S = S \cup \{x\}$.
- EXTRACT-MAX(S) removes and returns the element of S with the largest key.

- One application of max-priority queues is to schedule jobs on a shared computer.

The max-priority queue keeps track of the jobs to be performed and their relative priorities. When a job is finished or interrupted, the highest-priority job is selected from those pending using EXTRACT-MAX. A new job can be added to the queue at any time using INSERT.

HEAP_EXTRACT_MAX

HEAP_EXTRACT_MAX(A[1 .. n])

This will remove the maximum element from heap and return it

Input : heap(A)

Output : Maximum element or root, heap(A[1..n-1])

1. if A.heap_size >= 1
    2. max = A[1]
    3. A[1] = A[A.heap_size]
    4. A.heap_size = A.heap_size -1
    5. MAX_HEAPIFY(A,1)
    6. return max

Running time : O(log n)

HEAP_INSERT

HEAP_INSERT(A, key)

This will add a new element to the heap

Input : heap(A[1..n]), key - the new element

Output : heap(A[1..n+1]), with k in the heap

1. A.heap_size = A.heap_size + 1
2. i = A.heap_size // assume A[i] = - ∞
3. while i > 1 and A[PARENT(i)] < key

4. A[i] = A[PARENT(i)]
5. i = PARENT(i)
6. A[i] = key

Running time : O(lg n)

Summary

- Complete binary Tree
- Heap property
- Heap
- Maintaining heap Property(HEAPIFY)
- Building Heaps
- HeapSort Algorithm
- Priority queues.
- Heap Extract Max.
- Heap Insert.