Heap sort

Full- A binary tree is **full** if each node has 0 or 2 child nodes
Complete – A **full** binary tree with all leaf nodes at the same level

What is a Heap?
A **heap** is a binary tree. The tree is completely filled (**complete**) except possibly (**last**) level. The last level is also filled from the left. Last leaf node may be a single child of the parent node. All the leaf nodes are at most at the **last two** levels

Examples of Heaps

What is heap Property?
Heap is Top heavy heap -- MaxHeap A heap is top heavy if the value at any node exceeds the value at its child nodes.
Heap is Bottom heavy heap -- MinHeap A heap is bottom heavy if the value at any node is less than the value at its child nodes.

Example Is the sequence (23,17,14,6,13,10,1,5,7,12) a max heap? No

Heap – A Heap can be easily implemented as an array A[1…n] such that
A[1] is the **root**,
children of A[i] are A[2i] and A[2i+1]
parent of A[i] is $A[\lfloor i/2 \rfloor]$

A complete binary tree with n nodes has height h given by
$h=\lfloor lgn \rfloor$ used in time to adjust nodes in Heapify algorithm

If n is the number of nodes in a heap of height h

What is the Min # of nodes ?
$\mathbf{2^h}$ $2^h \le n$
What is the Max # of nodes ?
$\mathbf{2^{h+1}-1}$ $n \le 2^{h+1}-1 < 2^{h+1}$

therefore $2^h \le n < 2^{h+1}$

What is the exact value of h in terms of n?
$2^h \le n < n+1 \le 2^{h+1}$
$h \le lgn < lg(n+1) \le h+1$
$\mathbf{h=\lfloor lgn \rfloor \le lg(n)}$

What is the Min number of **leaf** nodes?

2^{h-1} ; one leaf at previous level becomes internal node and results in one leaf at next level.

What is the Max number of **leaf** nodes?

$2^h=2^{h-1}+2^{h-1}$; each leaf at previous level becomes internal node and results in two leaf nodes at next level.

In a heap with n nodes has exactly $\lfloor n/2 \rfloor$ internal nodes, $\lceil n/2 \rceil$ leaves (external)

$\lfloor n/2 \rfloor$ internal and $\lceil n/2 \rceil$ external

Show that with the array representation for storing n-element heap, the leaves are the nodes indexed by $\lfloor n/2 \rfloor+1$, $\lfloor n/2 \rfloor+2$, $\lfloor n/2 \rfloor+3$, …, n.

Adjusting the heaps

Invariant: all elements in A(k+1..n) satisfy max heap property. A(2k), A(2k+1) are roots of heaps

AdjustMaxHeap(A, k) -- n= size(A);

```
    Invariant: all elements in A(k+1..n) satisfy heap property. A(2k), A(2k+1) are
    roots of heaps
       index= 2k+1
       if index ≤ n
          if (A(2k) > A(index))
               index= 2k
      else
          index= 2k
          if index > n, return -- no work needed A(k) was a leaf node.
          if A(k) < A(index), exchange(A(k), A(index))

          A(k) > A(2k..n), A(k) > A(2k+1..n), A(k) > A(index)
          Invariant: all elements in A(index+1..n) satisfy heap property; A(2*index)
    and A(2*index+1) are heap roots
          AdjustMaxHeap(A, index)
          Invariant: all elements in A(index..n) satisfy heap property; A(index) is the
    root of a heap
```

Invariant: all elements inA(k..n) satisfy heap property. A(k) is the root of heap

Example

maxHeapify(A,3) on the array A =(27,17,3,16,13,10,1,5,7,12,4,8,9,0)
(1, 2,3,4, 5, 6, 7,8,9, 10,11,12,13,14)
(27,17,3,16,13,10,1,5,7, 12, 4, 8, 9, 0)
(27,17,**10**,16,13,**3**,1,5,7, 12, 4, 8, 9, 0)
(27,17,**10**,16,13,**9**,1,5,7, 12, 4, 8, **3**, 0)

Build a maxHeap on the array A=(5,3,17,10,84,19,6,22,9)

Build max heap

A(1..n) is an array of numbers

```
1. n = heapSize
2. for k = ⌊n/2⌋to 1
Invaraint: all elements in A(k+1..n) satisfy heap property A(2k ), A(2k+1 ) are heap roots
          Adjust(A, k)
          Max_Heapify(A, k)
Invaraint: all elements in A(k..n) satisfy heap property, A(k) is the root of a heap
```

Post condition all elements in A(1..n) satisfy heap property
Invaraint: A(1) is the root of a heap,
therefore A(1..n) is a heap

To build the heap, ⌊n/2⌋ nodes are adjusted. Computation is 2 times the height of the node. Since the heights of the leaves are zero, it means we can look at the heights of all the nodes in the heap. **This leads to**
Sum of heights of all nodes in the heap. Look at the heap from two angles:
$= 2^{h+1}-2-h$

Complexity to Build max heap: O(n)
Sorting:
Insertion sort create binary tree and then traverse left root right. This may not be very efficient in the worst case.
or
Use heap sort

Given: an array of size n
What is heap sort? How to do heap-sort?

SortAlgorithm

```
1. Build heap
2.          ArraySize=n;
   HeapSize=n;
   For k = n downto 2
```

Invariant: A(1..k) is max heap, A(1..k)<A(k+1..n) and A(k+1..n) is sorted

```
        Exchange A(k) and A(1)
        HeapSize=HeapSize-1;
        AdjustMaxheap (A,1):   A(1) to A(k-1)    O(lg n)
```

Invariant: A(1..k-1) is max heap, A(1..k-1)<A(k..n) and A(k..n) is sorted

PostCondition:

A(1..n) is sorted

Total O(n lg n)

Complexity of sorting an array

(1) create a heap O(n)

(2) heap sort (worst case)

A[1..n-1] can be adjusted into a heap in 2 lgn comparison steps, because 2 lgn comparisons (gross estimate) needed to adjust the root element.

Thus heap sorting complexity T_n of an n-element heap amounts to

$T_n = 2\ \lg n + T_{n-1}$

$T_n < 2n\lg n$ (a gross estimate)

□ Total complexity =O(nlgn)

Note. A better estimate will be

$$T_n = 2\ \lg n + T_{n-1}$$
$$T_{n-1} = 2\ \lg (n-1) + T_{n-2}$$
$$T_{n-2} = 2\ \lg (n-2) + T_{n-3}$$
$$\vdots$$
$$T_2 = 2\ \lg (2) + T_1$$

$$T_n = 2\ \lg n! + T_1$$
$$T_n = O(n \lg n)$$