Design and Analysis of Algorithm

(Heap Sort)

Lecture -14 - 17

Overview

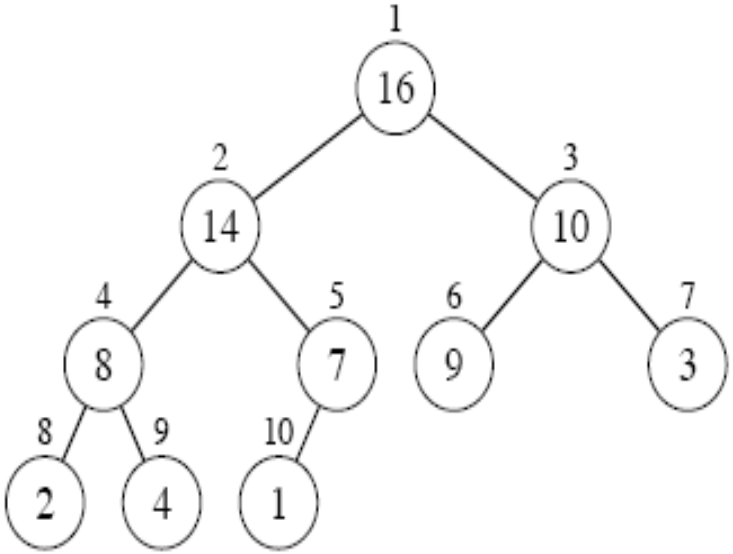
- *O(n* lg *n)* worst case time complexity.
- Sorts in place-like insertion sort.
- *O(n) is* the tight bound analysis of Build Heap.

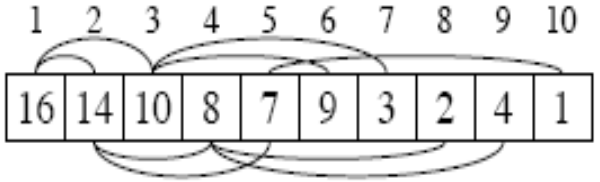
Heap data structure

- Heap A (*not* garbage-collected storage) is a nearly complete binary tree.
  - ***Height*** of node = # of edges on a longest simple path from the node down to a leaf.
  - ***Height*** of heap = height of root = $\Theta(\lg n)$.
- A heap can be stored as an array A.
  - Root of tree is $A[1]$.
  - Parent of $A[i] = A[\lfloor i/2 \rfloor]$.
  - Left child of $A[i] = A[2i]$.
  - Right child of $A[i] = A[2i + 1]$.
  - Computing is fast with binary representation implementation.

Example

Example: of a max-heap. *[Arcs above and below the array on the right go between parents and children. There is no significance to whether an arc is drawn above or below the array.]*





Example

- Given an array of size N. The task is to sort the array elements by using Heap Sort.
  - Input:
  - N=10
  - Arr[]:{16, 4, 10, 14, 7, 9, 3, 2, 8, 1}
  - Output: 1 2 3 4 7 8 9 10 14 16

Example

- Given an array of size N. The task is to sort the array elements by using Heap Sort.
  - Input:
  - N = 10
  - arr[] = {10,9,8,7,6,5,4,3,2,1}
  - Output:1 2 3 4 5 6 7 8 9 10

Heap property

- For max-heaps (largest element at root), ***max-heap property:*** for all nodes i, excluding the root, $A[\text{PARENT}(i)] \geq A[i]$.
- For min-heaps (smallest element at root), ***min-heap property:*** for all nodes i, excluding the root, $A[\text{PARENT}(i)] \leq A[i]$.

Maintaining the heap property

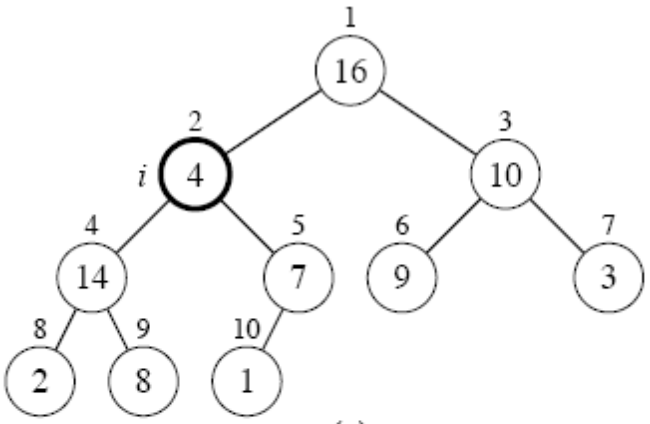
MAX-HEAPIFY is important for manipulating max-heaps. It is used to maintain the max-heap property.

- Before MAX-HEAPIFY, $A[i]$ may be smaller than its children.
- Assume left and right subtrees of i are max-heaps.
- After MAX-HEAPIFY, subtree rooted at i is a max-heap.

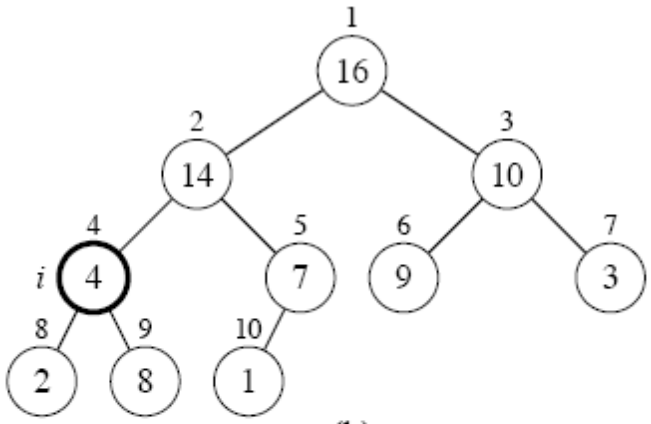
The way MAX-HEAPIFY works:

- Compare $A[i]$, $A[\text{LEFT}(i)]$, and $A[\text{RIGHT}(i)]$.
- If necessary, swap $A[i]$ with the larger of the two children to preserve heap property.
- Continue this process of comparing and swapping down the heap, until subtree rooted at i is max-heap. If we hit a leaf, then the subtree rooted at the leaf is trivially a max-heap.

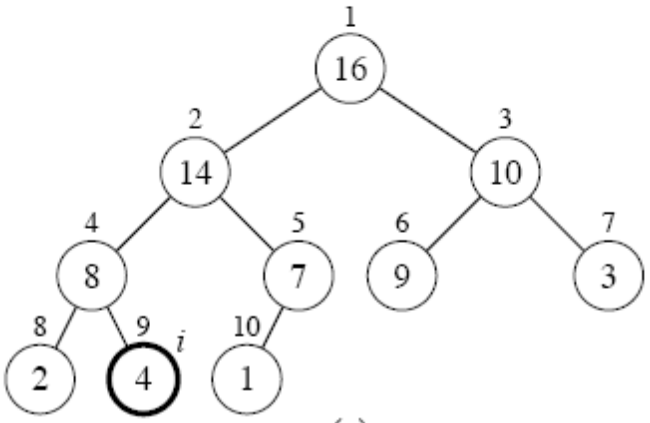
Run MAX-HEAPIFY on the following heap example.



(a)



(b)



(c)

```
MAX-HEAPIFY(A, i, n)
l ← LEFT(i)
r ← RIGHT(i)
if l ≤ n and A[l] > A[i]
   then largest ← l
   else  largest ← i
if r ≤ n and A[r] > A[largest]
   then largest ← r
if largest ≠ i
   then exchange A[i] ↔ A[largest]
        MAX-HEAPIFY(A, largest, n)
```

```
MAX-HEAPIFY(A, i, n)
l ← LEFT(i)
r ← RIGHT(i)
if l ≤ n and A[l] > A[i]
   then largest ← l
   else largest ← i
if r ≤ n and A[r] > A[largest]
   then largest ← r
if largest ≠ i
   then exchange A[i] ↔ A[largest]
        MAX-HEAPIFY(A, largest, n)
```

The Time Complexity of MAX-HEAPIFY (A,i,n) is $O(\log n)$.
[Because any element insert in a tree required maximum log n time]

Building a heap

BUILD-MAX-HEAP(A, n)

for $i \leftarrow \lfloor n/2 \rfloor$ **downto** 1

  do MAX-HEAPIFY(A, i, n)

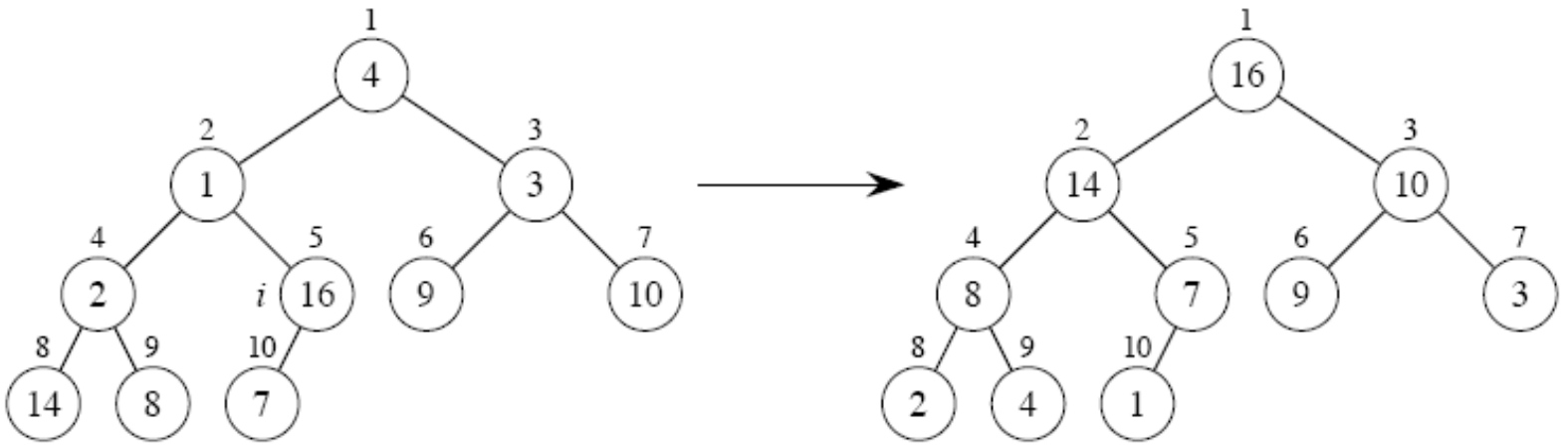
[Parameter n replaces both attributes length[A] and heap-size[A].]

Example

Building a max-heap from the following unsorted array results in the first heap example.

- i starts off as 5.
- MAX-HEAPIFY is applied to subtrees rooted at nodes (in order): 16, 2, 3, 1, 4.

	1	2	3	4	5	6	7	8	9	10
A	4	1	3	2	16	9	10	14	8	7



- Node 2 violates the max-heap property.
- Compare node 2 with its children, and then swap it with the larger of the two children.
- Continue down the tree, swapping until the value is properly placed at the root of a subtree that is a max-heap. In this case, the max-heap is a leaf.

Time: $O(\lg n)$.

Correctness: *[Instead of book's formal analysis with recurrence, just come up with* $O(\lg n)$ *intuitively.]* Heap is almost-complete binary tree, hence must process $O(\lg n)$ levels, with constant work at each level (comparing 3 items and maybe swapping 2).

Correctness

Initialization:

Initialization: we know that each node $\lfloor n/2 \rfloor + 1, \lfloor n/2 \rfloor + 2, \ldots, n$ is a leaf, which is the root of a trivial max-heap. Since $i = \lfloor n/2 \rfloor$ before the first iteration of the **for** loop, the invariant is initially true.

Maintenance: Children of node i are indexed higher than i, so by the loop invariant, they are both roots of max-heaps. Correctly assuming that $i+1, i+2, \ldots, n$ are all roots of max-heaps, MAX-HEAPIFY makes node i a max-heap root. Decrementing i re-establishes the loop invariant at each iteration.

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

Analysis

- **Simple bound:** O(n) calls to MAX-HEAPIFY, each of which takes O(lg n) time ⇒ O(n lg n).

- **Tighter analysis observation**:

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

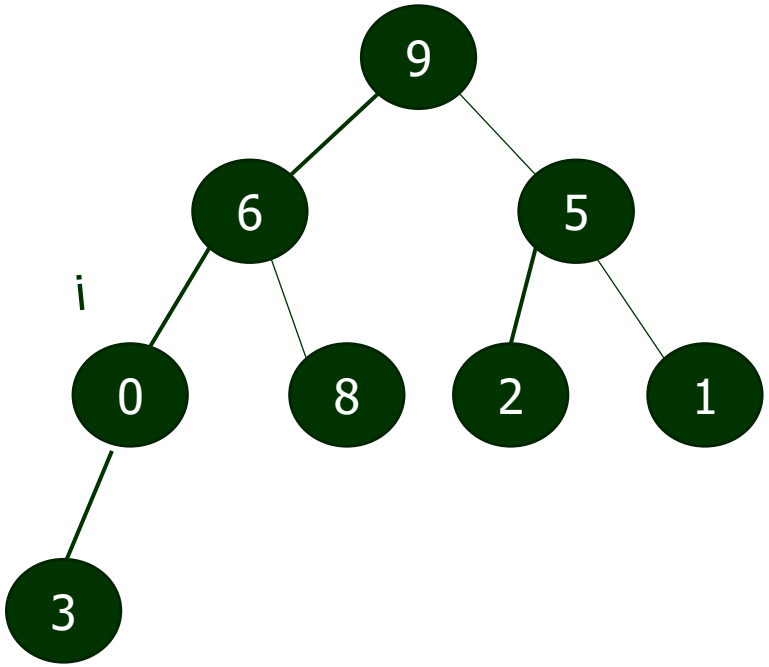
Tighter analysis Proof

BUILD-MAX-HEAP(A,n)

$for\ i \leftarrow \lfloor n/2 \rfloor\ downto\ 1$

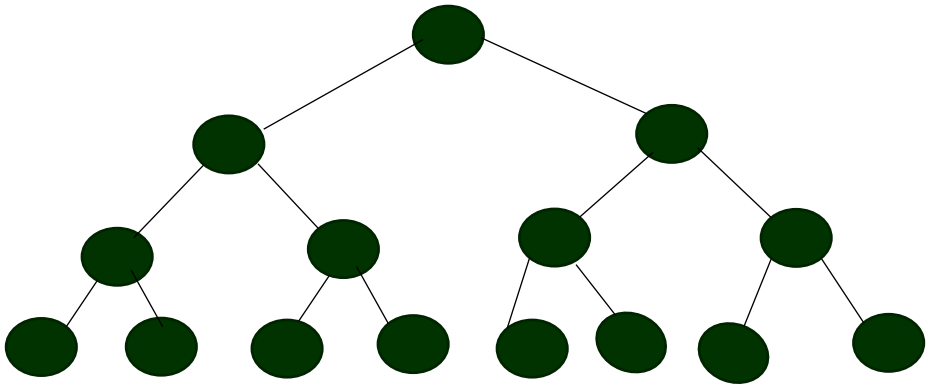
do MAX-HEAPIFY (A,i,n)

1	2	3	4	5	6	7	8
9	6	5	0	8	2	1	3



Tighter analysis Proof

- For easy understanding, Let us take a complete binary Tree,



- The height of a node is the number of edges from the node to the deepest leaf.
- The depth of a node is the no of edges from the root of the node.

Tighter analysis Proof

- All the leaves are of height 0, therefore there are 8 nodes at height 0.
- 4 numbers of nodes at height 1.
- 2 numbers of nodes at height 2.
- and one node at height 3.

Tighter analysis Proof

- Hence the question is how many nodes are there at height ‘h’ in a complete binary tree?
- The answer is :
- If there are n nodes in tree, then at most $\left\lceil \frac{n}{2^{h+1}} \right\rceil$ nodes are available at height h.

Tighter analysis Proof

- Now if we apply MAX-HEAPIFY() on any node of any level, then the time taken by MAX-HEAPIFY() is the height of the node.(i.e. $\left\lceil \frac{n}{2^{h+1}} \right\rceil O(h)$)
- **Hence in case of root the time taken is** $\log n$.
- **Hence**

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

Work done by Max-Heapify() on the number of nodes of height h.

Tighter analysis Proof

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

$$\leq O\left(n \sum_{h=0}^{\infty} \frac{h}{2^h 2}\right)$$

$$\leq O\left(\frac{n}{2} \sum_{h=0}^{\infty} \frac{h}{2^h}\right)$$

$$\leq O\left(\frac{n}{2} \sum_{h=0}^{\infty} h \left(\frac{1}{2}\right)^h\right)$$

$$\leq O\left(\frac{n}{2} \left[\frac{1/2}{(1-1/2)^2}\right]\right) \Longrightarrow O(\frac{n}{2} 2\ )$$

$$T(n) \Longrightarrow O(n\ )$$

This is a Harmonic Series. By Integrating and Differentiating the series we get the value is 2.
(Refer to next slide or cormen Appendix A.8)

Hence the running time of **BUILD-MAX-HEAP(A,n)** is $O(n\ )$ in tight bound .

Tighter analysis Proof

$$\sum_{k=0}^{\infty} x^k = \frac{1}{1-x} \quad \text{[value of Infanite G P Series]}$$

$$\sum_{k=0}^{\infty} x^k = (1-x)^{-1}$$

Differentiate both side:

$$\sum_{k=0}^{\infty} k.x^{k-1} = (-1)(1-x)^{-2}(-1) = \frac{1}{(1-x)^2}$$

Multiply x both side

$$\sum_{k=0}^{\infty} k.x^k == \frac{x}{(1-x)^2}$$

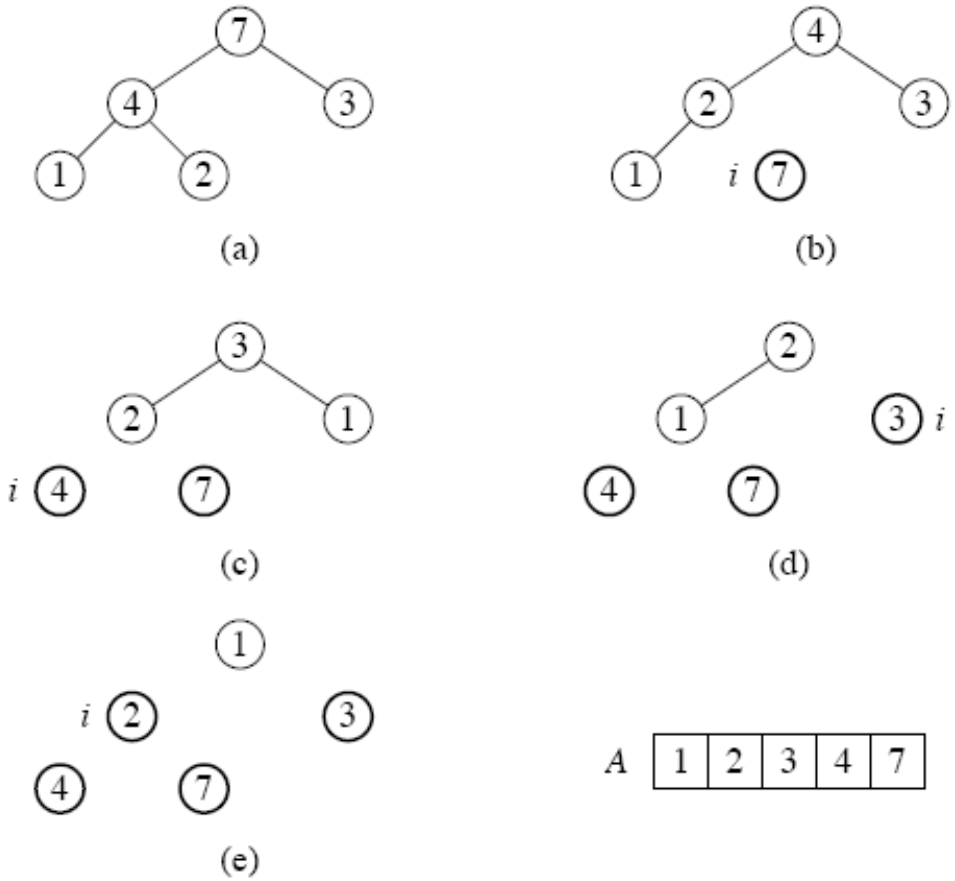
The heapsort algorithm

Given an input array, the heapsort algorithm acts as follows:

- Builds a max-heap from the array.
- 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 in its correct place) by decreasing the heap size, and calling MAX-HEAPIFY on the new (possibly incorrectly-placed) root.
- Repeat this "discarding" process until only one node (the smallest element) remains, and therefore is in the correct place in the array.

```
HEAPSORT(A, n)
BUILD-MAX-HEAP(A, n)
for i ← n downto 2
    do exchange A[1] ↔ A[i]
        MAX-HEAPIFY(A, 1, i − 1)
```

Example



Analysis

- BUILD-MAX-HEAP: O(n)
- **for** loop: n – 1 times
- exchange elements: O(1)
- MAX-HEAPIFY: O(lg n)

Total time: O(n lg n).

Heap implementation of priority queue

- Heaps efficiently implement priority queues. These notes will deal with max priority queues implemented with max-heaps. Min-priority queues are implemented with min-heaps similarly.
- A heap gives a good compromise between fast insertion but slow extraction and vice versa. Both operations take O(lg n) time.

Priority queue

- Maintains a dynamic set S of elements.
- Each set element has a **key**-an associated value.
- Max-priority queue supports dynamic-set operations:
  - INSERT(S, x): inserts element x into set S.
  - MAXIMUM(S): returns element of S with largest key.
  - EXTRACT-MAX(S): removes and returns element of S with largest key.
  - INCREASE-KEY(S, x, k): increases value of element x's key to k. Assume $k \geq$ x's current key value.
- Example max-priority queue application: schedule jobs on shared computer.

- Min-priority queue supports similar operations:
  - INSERT(S, x): inserts element x into set S.
  - MINIMUM(S): returns element of S with smallest key.
  - EXTRACT-MIN(S): removes and returns element of S with smallest key.
  - DECREASE-KEY(S, x, k): decreases value of element x's key to k. Assume $k \leq$ x's current key value.
- Example min-priority queue application:

  event - driven simulator.

Finding the maximum element

Getting the maximum element is easy: it's the root.

HEAP-MAXIMUM*(A)*

return *A*[1]

Finding the maximum element

Getting the maximum element is easy: it's the root.

```
HEAP-MAXIMUM(A)
    return A[1]
```

Time: $\mathbf{O}(1)$.

Extracting max element

Given the array A:

- Make sure heap is not empty.
- Make a copy of the maximum element (the root).
- Make the last node in the tree the new root.
- Re-heapify the heap, with one fewer node.
- Return the copy of the maximum element.

Extracting max element

Given the array *A*:

- Make sure heap is not empty.
- Make a copy of the maximum element (the root).
- Make the last node in the tree the new root.
- Re-heapify the heap, with one fewer node.
- Return the copy of the maximum element.

HEAP-EXTRACT-MAX*(A, n)*

if $n < 1$

then error .heap underflow.

$max \leftarrow A[1]$

$A[1] \leftarrow A[n]$

MAX-HEAPIFY*(A,* 1*,* $n - 1$*)* remakes heap

return *max*

Extracting max element

Given the array *A*:

- Make sure heap is not empty.
- Make a copy of the maximum element (the root).
- Make the last node in the tree the new root.
- Re-heapify the heap, with one fewer node.
- Return the copy of the maximum element.

HEAP-EXTRACT-MAX*(A, n)*

if $n < 1$

then error .heap underflow.

$max \leftarrow A[1]$

$A[1] \leftarrow A[n]$

MAX-HEAPIFY*(A,* 1*,* $n - 1$*)* remakes heap

return *max*

Time Complexity is $O(\log n)$

- **HEAP-INCREASE-KEY**(A,i,key)

1. If key<A[i]
2. error” new key is smaller than the current key”.
3. A[i]= key
4. While i>1 and A[parent(i)]<A[i]
5. swap(A[parent(i)], A[i])
6. i=parent(i)

- **HEAP-INCREASE-KEY**(A,i,key)

1. If key<A[i]
2. error" new key is smaller than the current key".
3. A[i]= key
4. While i>1 and A[parent(i)]<A[i]
5. swap(A[parent(i)], A[i])
6. i=parent(i)

The running time of **HEAP-INCREASE-KEY**(A,i,key) is $O(\log n)$

- Example(from own)

- MAX-HEAP-INSERT(A,key)

1. A.heap-size= A.heap-size+1
2. A[heap-size]=-∞
3. HEAP-INCREASE-KEY(A,heap-size,key)

- MAX-HEAP-INSERT(A,key)

1. A.heap-size= A.heap-size+1
2. A[heap-size]=-∞
3. HEAP-INCREASE-KEY(A,heap-size,key)

The running time of MAX-HEAP-INSERT(A,key) is $O(\log n)$.


Thank U