Heapsort

HeapSort introduces another algorithm design technique using a data structure "heap" to manage information.

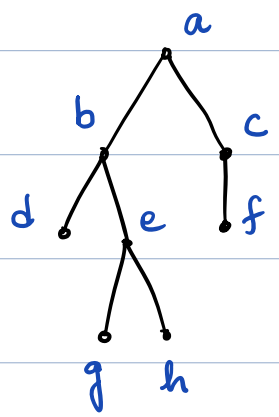
Running time : $O(n \log n)$

Plan

- Review of some basic definitions
- Heaps
- Heap sort.

Binary tree :

It is a tree in which every node|vertex has at most two children.



Refer to any Standard textbook for the terminology.

'a' : root node

b is the Parent of d

d and e are children of b

d, g, h, f are leaves | leaf nodes [Nodes with NO Child]

a, b, c, d, e are internal nodes [Nodes with atleast one child]

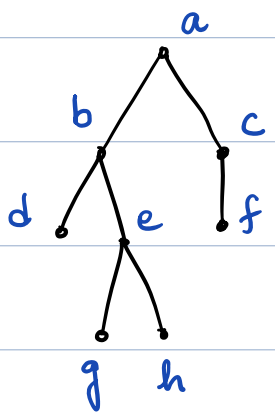
b is an ancestor of g

f is a descendant of a

Def:

The height of a node in a binary tree is the number of edges on the longest simple downward path from the node to a leaf. Height of the tree is height of the root.

Example:



① Height of the node g is zero and b is two

② Height of the tree is Three.

The depth of a node is the number edges from the node to the tree's root.

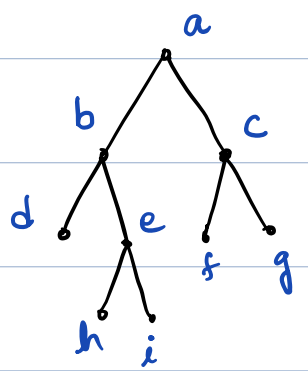
① Depth of the node e is two.

② Depth of root node is zero.

Level of node = Its depth + 1

Complete Binary tree :

A complete binary tree is a binary tree in which every level, except possibly the last level. is completly filled.



Heap (or Binary Heap): It is a complete binary tree with two additional constraints.

① A heap is complete binary tree that is all levels of the tree, except possibly last are filled and if the last level of the tree is not complete, the nodes of that level are filled from left to right. [Structural Property]

② Heap Property: The key (or value) stored at each node is either greater than or equal to or less than or equal to the keys stored at the children.

Array representation of heap

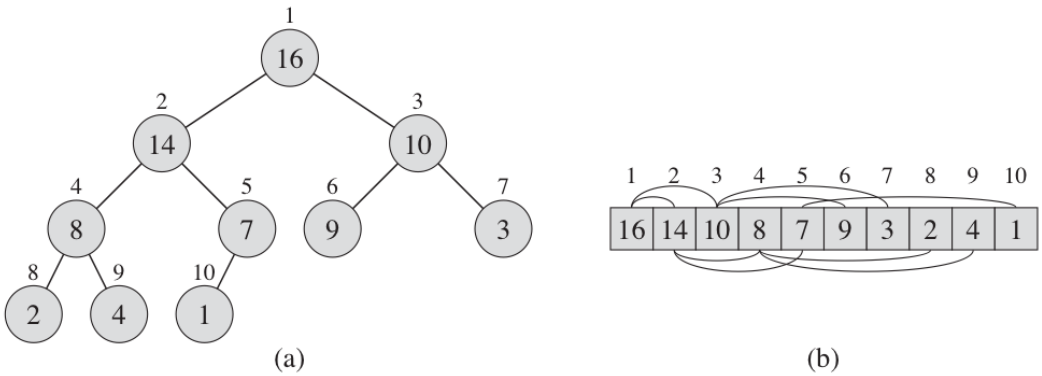


Figure 6.1 A max-heap viewed as **(a)** a binary tree and **(b)** an array. The number within the circle at each node in the tree is the value stored at that node. The number above a node is the corresponding index in the array. Above and below the array are lines showing parent-child relationships; parents are always to the left of their children. The tree has height three; the node at index 4 (with value 8) has height one.

Given an index i of a node, the index of its parent is $\lfloor \frac{i}{2} \rfloor$, index of left child is $2i$ and index of right child is $2i+1$.

There are two kinds of binary heaps

① **Max-heaps**: For every node i, other than the root

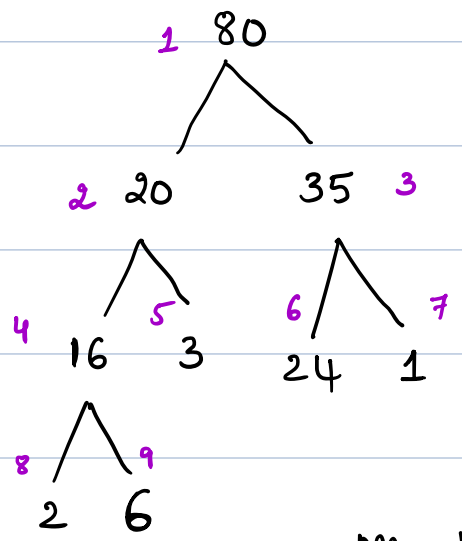
$$A[\text{Parent}(i)] \geq A[i]$$

i.e, the value of a node is at most the value of its Parent.

The Maximum element in a max-heap is at the root.

$[80, 20, 35, 16, 3, 24, 1, 2, 6]$

Example:



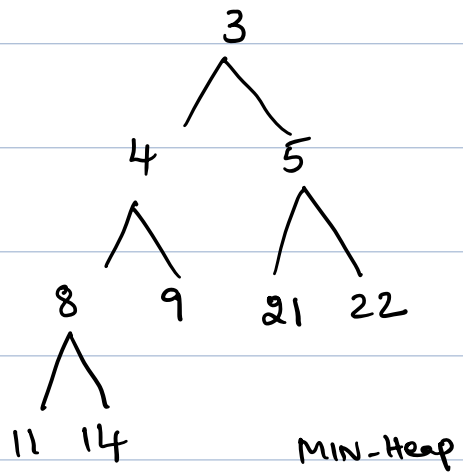
MAx-heap

② **Min-heaps**: For every node i, other than the root

$$A[\text{Parent}(i)] \leq A[i]$$

The Minimum element in a Min-heap is at the root.

Example:



MIN-Heap

We mainly focus on Max-heaps in this notes.

Short Quiz: [For ① & ② write the Proofs]

① What are the minimum and maximum number of elements in a complete binary tree of height h?

$$2^h \leq \#\text{ of elements} \leq 2^{h+1} - 1$$

② What is the height of a Complete binary tree which has n-nodes?

$$\lfloor \log n \rfloor$$

③ In an n-element heap, What are the indices of the leaf nodes?

Ans $I = \{ \lfloor \frac{n}{2} \rfloor + 1 \ldots n \}$

Suppose i is an index in I

then its Children will be at $2i$ and $2i+1$

$2(i) = 2\lfloor \frac{n}{2} \rfloor + 2 > n$, not Possible.

Suppose i is an index with no children

ie. $2i$ and $2i+1$ are $> n$ ie, $i > \frac{n}{2}$

ie, $i \in \{ \lfloor \frac{n}{2} \rfloor + 1 \ldots n \}$

Overview of the Algorithm – HEAP SORT

Q How to use Heap (max-heap) to sort the elements of an array.

If Can make a max-heap from the given array elements. Then we can easily get the maximum element. (Present at root)

↓ Then

Maybe again if we can make max-heap from the remaining elements we can get 2nd largest element

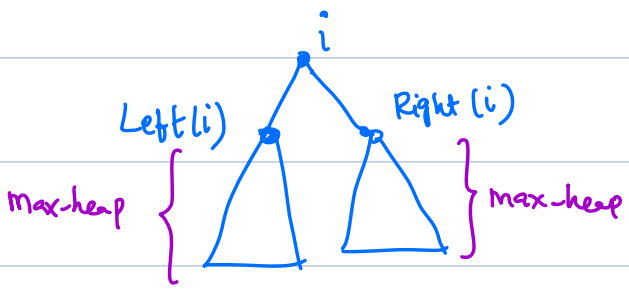
↓ Then

Repeat the above steps

Maintaining the heap Property

We describe Procedure The Max-Heapify to maintain the max-heap Property.

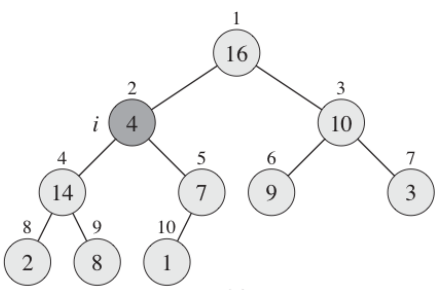
Given input an array A and an index i of the array. The max-heapify assumes that binary trees rooted at Left(i) and Right(i) are max-heaps, but A[i] might be smaller than its children (hence voilates max-heap Property).



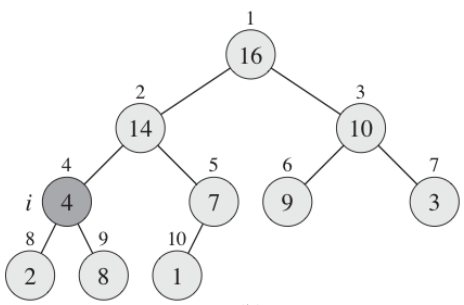
Our goal is to maintain the heap Property.

```
MAX-HEAPIFY(A, i)
1  l = LEFT(i)
2  r = RIGHT(i)
3  if l ≤ A.heap-size and A[l] > A[i]
4       largest = l
5  else largest = i
6  if r ≤ A.heap-size and A[r] > A[largest]
7       largest = r
8  if largest ≠ i
9       exchange A[i] with A[largest]
10      MAX-HEAPIFY(A, largest)
```

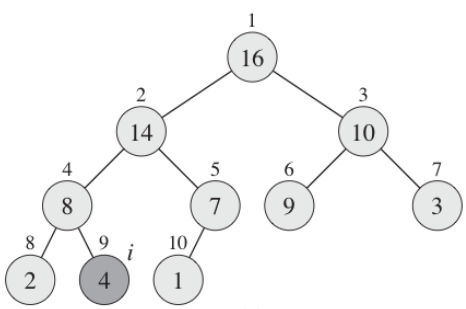
Example: The action of Max-Heapify (A, 2)



(a)



(b)



(c)

Running time:

① To compare A[i], A[Left(i)] and A[Right(i)] take $\Theta(1)$ time at given node i.

② We have to repeat the above task at most $O(h)$ time, h is the height of the node i.

∴ the running time of MAX-HEAPIFY is $O(h) = O(\log n)$

Building a heap:

Q: Given an array $A[1,..n]$, how to build a max-heap representing A?

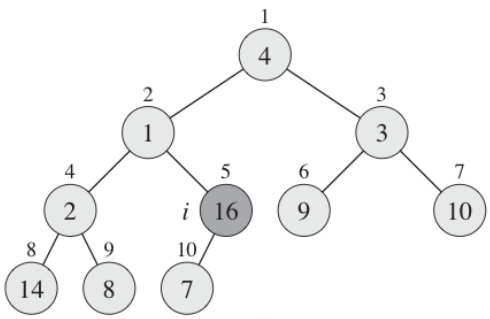
idea is to use Max-heapify in a bottom-up manner to convert an array into a max-heap.

Note: The elements in the subarray $A[\lfloor \frac{n}{2} \rfloor + 1, .. n]$ are all leaves of the tree, so each is a 1-element heap.

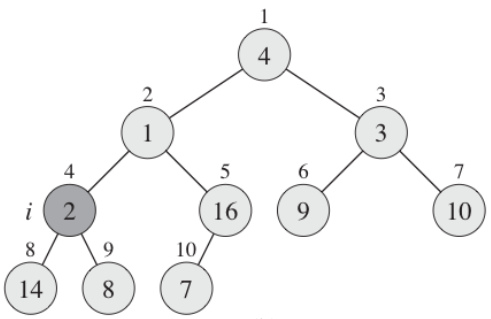
```
BUILD-MAX-HEAP(A)
1  A.heap-size = A.length
2  for i = ⌊A.length/2⌋ downto 1
3      MAX-HEAPIFY(A, i)
```

Example

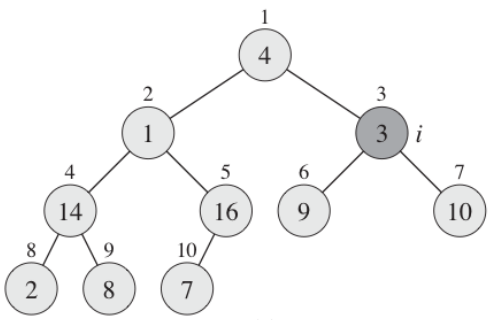
A	4	1	3	2	16	9	10	14	8	7



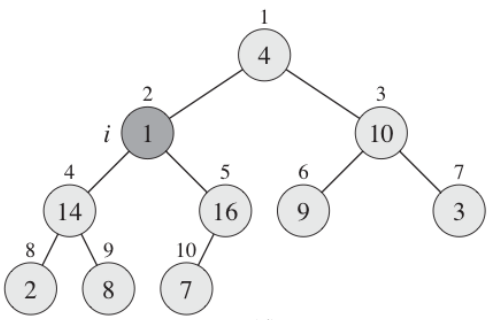
(a)



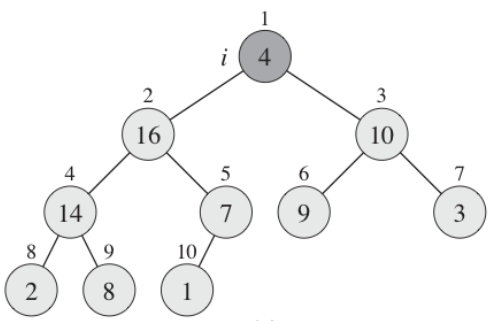
(b)



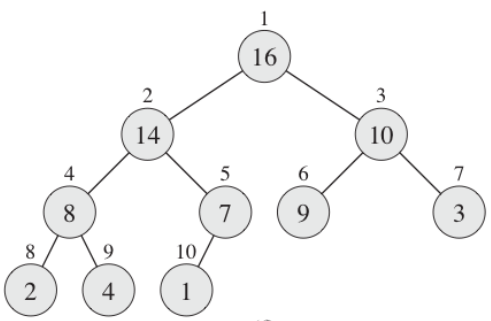
(c)



(d)



(e)



(f)

Running time:

```
BUILD-MAX-HEAP(A)
1  A.heap-size = A.length
2  for i = ⌊A.length/2⌋ downto 1        → O(n)
3      MAX-HEAPIFY(A, i)                → O(log n)
```

From the above, the running time of the algorithm is $O(n \log n)$. As we will see next this is not asymptotically tight.

We will derive a better upperbound on running time.

The time for Max-Heapify at a node depends on the height of the node in the tree.

Recall:

(a) An n-element heap has height $\lfloor \log n \rfloor$

(b) An n-element heap has at most $\left\lceil \frac{n}{2^{h+1}} \right\rceil$ nodes of any height h. [Exercise]

The time required for max-heapify when called on a node of height h is $O(h)$, so the running time of Build-max-heap is

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

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

$$\left\{ \sum_{k=0}^{\infty} k x^k = \frac{x}{(1-x)^2} \quad |x| < 1 \right\}$$

From (1) we get

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

$$= O(n).$$

ie, we can build a max-heap from an unordered array in linear time.

Heapsort algorithm

IDEA: Given an array A, we build a first max-heap and then maximum element of the array is stored at the root A[1].

We exchange A[1] with A[n] and discard node n from the heap.

Now the new root element may violate the max-heap property. We use max-heapify to restore the max-heap property.

The heapsort algorithm repeats the above process for the max-heap of size n-1 to heap of size 2.

```
HEAPSORT(A)
1  BUILD-MAX-HEAP(A)                    → O(n)
2  for i = A.length downto 2            → n-1 times
3      exchange A[1] with A[i]
4      A.heap-size = A.heap-size - 1
5      MAX-HEAPIFY(A, 1)                → O(log n)
```

Running time:

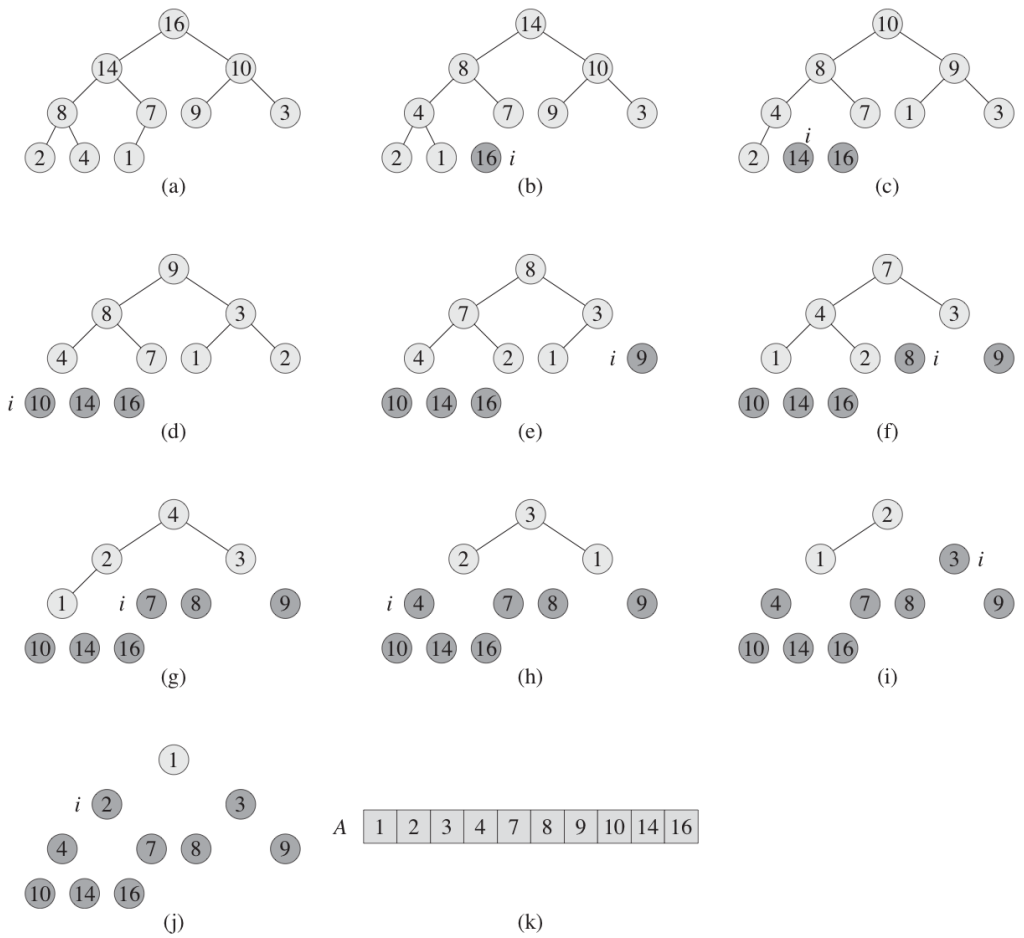
Build-max-Heap Procedure takes $O(n)$ time.

max-heapify called $(n-1)$ times.

So total running time $= O(n) + (n-1)\,O(\log n)$

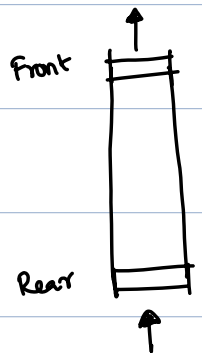
$= O(n \log n)$

Example



Priority Queue: (max-priority queue)

Queue: First in first out



A **Priority queue** is different from queue, instead of being a first-in first-out, values come out in order by Priority.

Priority queues are used in Scheduling Process in a Computer and they are used in some algorithms like Prim's and Huffman Codes

Some implementation:

First Idea

n - # of elements

Array of unordered elements:

Insert: Constant time, we only have insert it at n and increment n.

Maximum: returns the highest priority element.

This will take $O(n)$ time { we have to scan the whole array }

Extract-max: Removes and returns the highest priority element.

Takes $O(n)$ time: First find the highest priority elent then swap it with last elemet and decrement n.

Second Idea:-

Sorted array: Array is sorted (lowest to highest priority elements.) based on the Priorities.

Insert: O(n) time: ① O(log n) steps to find the place i where it belongs,

② We need to shift elements to make space for the insertion, This takes O(n) time.

Maximum: O(1) time

Extract max: O(1) time (Since highest priority element is at the end of the array)

We can repeat the above ideas using linked-list also

Next idea to use HEAPS. (

We want the MAX-Priority Queue to support the following operations.

INSERT(S, x) : Inserts the element x into the set S.

MAXIMUM(S) : returns the maximum element of S

EXTRACT-MAX(S) : Removes & returns the maximum element of S.

INCREASE-KEY(S, x, k) : Increases the value of element x's key to the new value k,

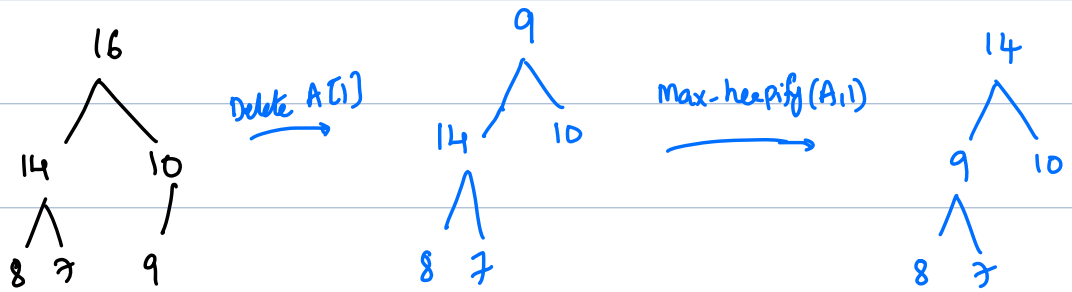
[we assume $k \geq x.key$]

Heap as Priority queue : [Max-Heap]

Maximum : O(1) time

Extract max : Swap the root A[1] with A[A.heap size]

and decrease the size then call Max-Heapify (A, 1)




```
HEAP-EXTRACT-MAX(A)
1  if A.heap-size < 1
2        error "heap underflow"
3  max = A[1]
4  A[1] = A[A.heap-size]
5  A.heap-size = A.heap-size − 1
6  MAX-HEAPIFY(A, 1)
7  return max
```

constant

line 6 → O(log n)

Running time: O(log n)

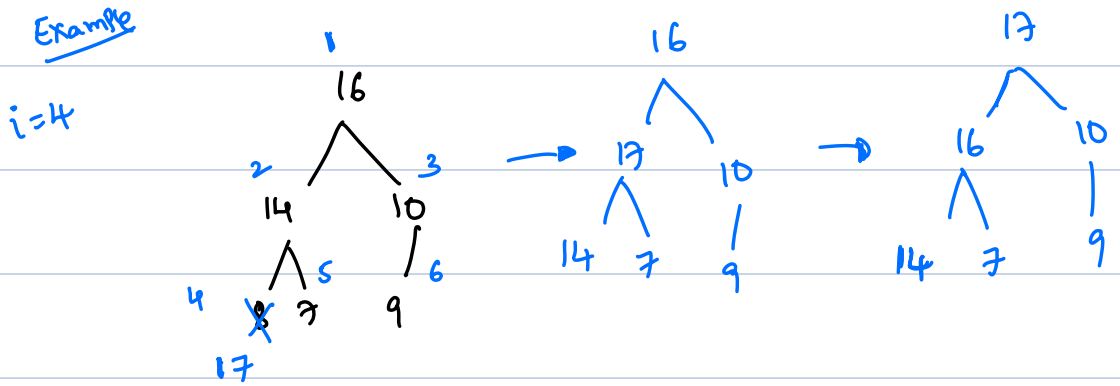
Increase key: increases the key of elemt A[i] to its new value.

Increasing the key of A[i] might voilate the max-heap Property.

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

```
1  if key < A[i]
2      error "new key is smaller than current key"
3  A[i] = key
4  while i > 1 and A[PARENT(i)] < A[i]
5      exchange A[i] with A[PARENT(i)]
6      i = PARENT(i)
```

Example



Runningtime: $O(\log n)$

Similary we can implement Decrease key.

Insert:

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

The INSERT Procedure first expands the max_heap by adding to the tree a leaf node whose key is $-\infty$. then it calls HEAP-INCREASE-KEY to set the key of this new node to its correct value and maintain the max_heap property.

Running time: $O(\log n)$

Summary

A Heap can support any priority-queue operation on a set of size n in $O(\log n)$ time.