Heap Sorting

Heap Sorting

Topics

- Introduction to Heaps
  - *Heap properties*
  - *Heap types*

- Heaps Operations
  - *Heap Fixing*
  - *Heap Building*
  - *Heap Sorting*

- Analysis of Heap Operations
  - *Heap fixing*
  - *Heap building*
  - *Heap sorting*

Binary Heaps

Definition

A *binary heap*, simply referred to as *heap*, is a *binary tree* with *completeness* and *order* properties.

Completeness Property: The completeness property implies that the tree has nodes at all levels, except possibly at the lowest level. Further, at bottom level the nodes (leafs) are organized in *left to right* positions. In other words, the nodes in the right positions at the last level may be missing. A tree that contains nodes at all levels is called *complete tree*

Order Property: The order property implies that key in each node is larger than, or equal to the keys in child nodes. This structure is referred to as ***max-heap*** . By contrast, in a ***min-heap,*** each parental node contains a key which is smaller than or equal to the keys in the child nodes. Thus, order property implies

$\mathbf{key}(parent) \geq \mathbf{key}(left),\quad \mathbf{key}(parent) \geq \mathbf{key}\,(right)$ *(max-heap)*

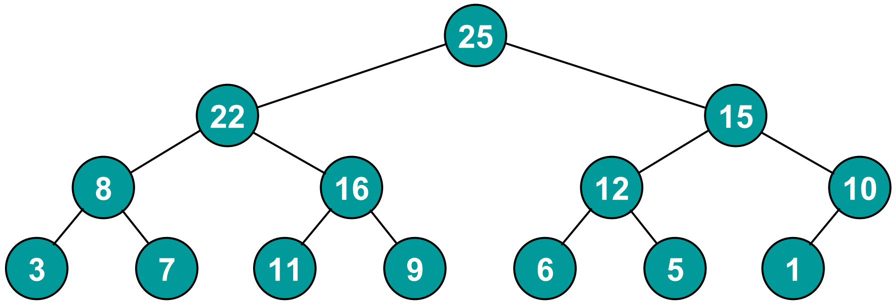
$\mathbf{key}(parent) \leq \mathbf{key}(left),\quad \mathbf{key}(parent) \leq \mathbf{key}\,(right)$ *(min-heap)*

➢ Depending upon an application of heap, each node may contain other data items.
Such a heap is said to be a ***priority queue*** . The keys are usually, the priorities attached to the data items. In scheduling application , for example, priorities are usually the order of scheduling of different tasks.

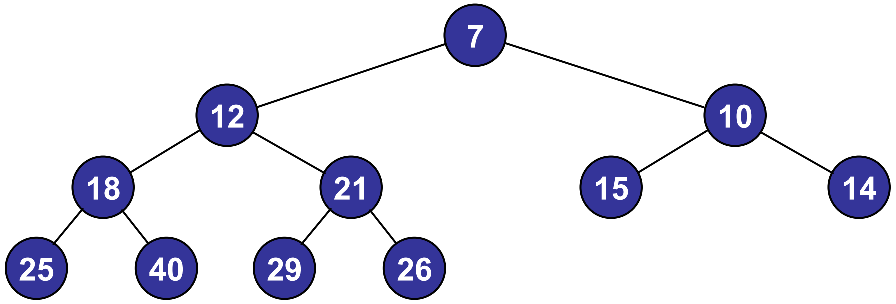
Binary Heaps

Examples

The examples of max-heap and min-heap are shown in figures *(a)* and *(b)* below. Note that each subtree in the heap is also a smaller heap of the same type.



(a) Sample max-heap

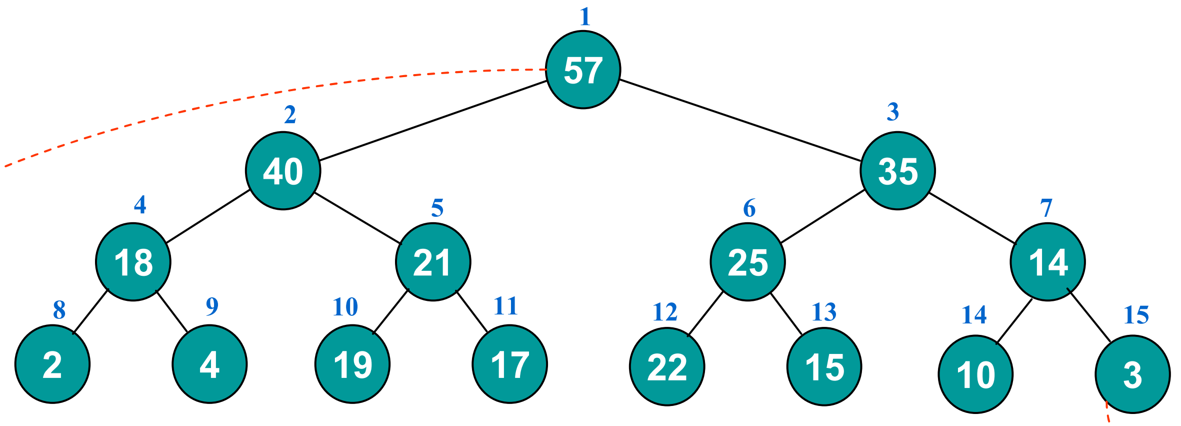


(b) Sample min-heap

Binary Heaps

Array Implementation

A heap can be implemented as an *array*. The array is filled with *heap keys,* starting with the key at the *root*, and moving from *top to bottom* and from *left to right* across the binary tree. The array representation of heap in figure *(a)* is shown in figure *(b)*. The numbers next to nodes indicate the associated *array indexes*



(a) Binary heap

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
57	**40**	**35**	**18**	**21**	**25**	**14**	**2**	**4**	**19**	**17**	**22**	**15**	**10**	**3**

(a) Array implementation

Binary Heap

Basic Operations

The basic operations on a heap data structure, implemented as an array, are:

PARENT(*i*) - *returns index of parent of the node identified by the index i*

LEFT(*i*) - *returns index of left child of the node identified by index i*

RIGHT(*i*) - *returns index of right child of the node identified by index i*

HEAPIFY(*A*, *i*) - *restores heap order property by fixing up an offending key, stored in a node with index i*

BUILD-HEAP(*A*) – *converts, an unordered array A, into a binary heap*

SORT(A) – s***orts array A***

Basic Operations

Implementation

The pseudo code (Source: *T. Cormen et al*) for computing indexes of parent and child nodes is given below

```
PARENT( i )        ► k returns index of parent node  of a key with index i
1   $k \leftarrow \lfloor i / 2 \rfloor$
2   return  k

LEFT( i )          ► k returns index of left child of a key with index i
1     $k \leftarrow 2i$
2     return k

RIGHT( i )         ► k returns index of right child of a key with index i
1    $k \leftarrow 2i + 1$
2     return  k
```

➢ The procedures **PARENT**, **LEFT**, and **RIGHT** require ***fixed*** amount of time to do the computations on array indexes. Thus, *running time for each of the above operations is $\theta(1)$*

Heap Fixing-Up

Algorithm

The *heapify* procedure is used to fix up a given node that violates the order property. It consists of following steps:

Step #1: Compare the key of the offending node with the key of the left child
Select the index of the bigger key and store in the index variable *largest*

Step #2: Compare the key identified by index *largest* with the key in the right child

Step #3: Save index of the bigger key into the variable *largest*

Step #4: If the index of offending key is not equal to *largest*, then exchange key of the parent node with the key identified by *largest*; else, exit

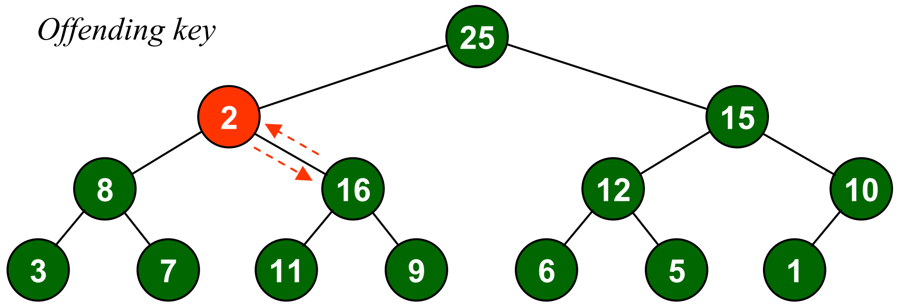
Step #5:Repeat *Steps #1* through *Step #4* , with the offending key identified by the index *largest*

- In essence, offending key is ***trickled down*** until the heap order property is restored.

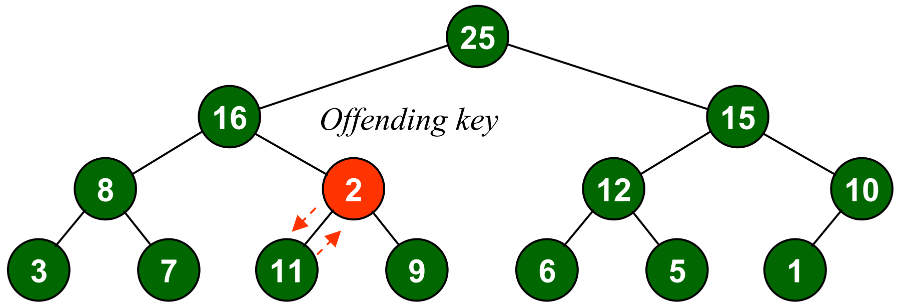
Heap Fixing-Up
Example

The heap fixing-up procedure is illustrated by the following example

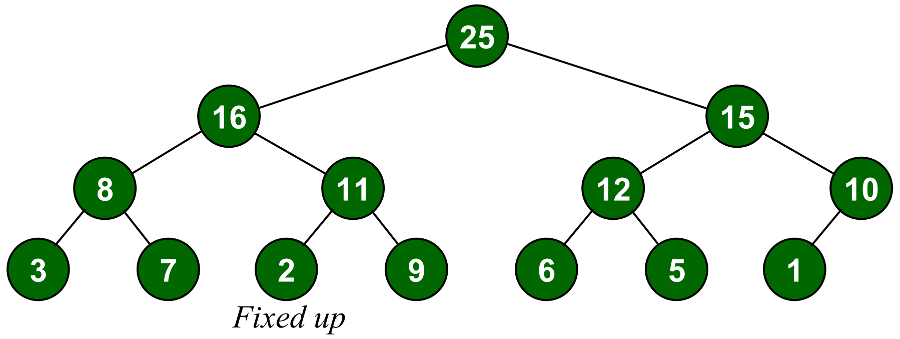
(1) All keys in the tree structure satisfy heap order property, except key 2, in the leftmost position at level 1 (shaded red). To fix up the heap, the offending key is exchanged with the larger key 16 in the right child



(2) The key 2 in the new position again violates the heap order property. It is exchanged with the larger key 11 in the left child.



(3) All keys satisfy the max-heap order property. It is also a complete tree The heap is fixed up.



Heapify Algorithm

Implementation

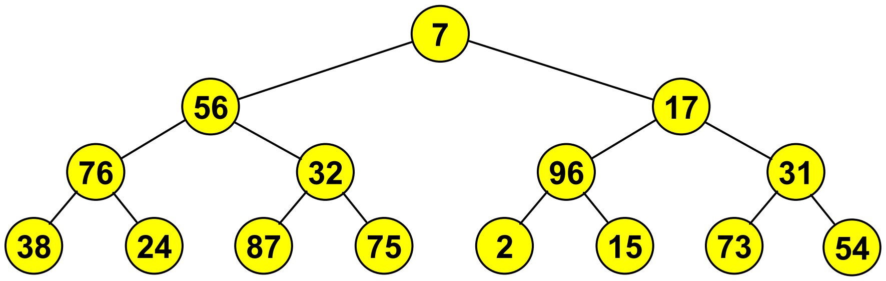
The heap is assumed to be implemented as an array. The *index i* of the *offending key, heap size n* and array *A* are passed on to the procedure *HEAFIFY(Ref: T.Cormen et al; with slight modification . Comments have been added for elaboration of steps).* The key of offending node is repeatedly compared with keys in child nodes, until heap *order property* is restored

```
HEAPIFY(i, n, A) ▶ i is the index of key to be fixed, n is heap size, A is input array
1   l ← 2i               ▶ l stores the index of left child  of parent identified by index i
2   r ← 2i+1             ▶ r stores the index of right child of parent identified by index i
3   if  l ≤ n and A[l] > A[i]    ▶ checks if key in left child is larger than key in  parent node
4      then largest ← l          ▶ if  so, it is saved in variable largest
5   else largest ← i             ▶ else, key at right child is stored in largest
6   if  r ≤ n  and  A[r] > A[largest] ▶ larger of  the keys at left and right nodes  is chosen
7      then largest ← r
8   if  largest ≠ i
9      then exchange A[i] ↔ A[largest]  ▶ elements pointed by index largest and index i are exchanged
10          HEAPIFY(largest, n, A,)       ▶ HEAPIFY procedure is applied recursively to the key
                                          in new location
11   return  A                          ▶ Heap is fixed up
```

Heap Building

Example

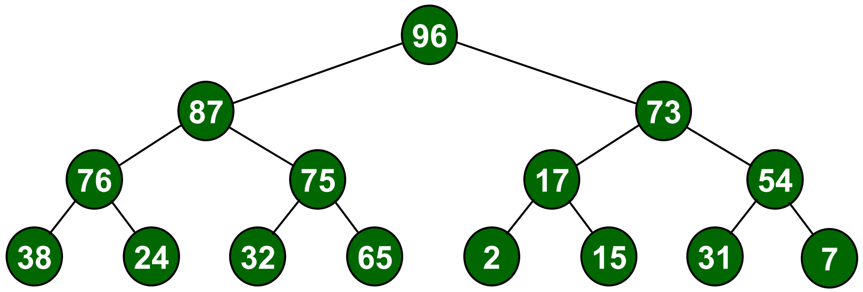
The heap building operation is usually needed to convert a set of *randomly distribute keys* in a tree to a heap. Figure *(i)* shows a sample tree containing random keys. Figure *(ii)* shows a array implementation of binary tree . Figure *(iii)* displays the *tree converted to a max-heap* Figure *(iv)* shows array implementation of the max-heap



(i) A binary tree with random keys

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
7	56	17	76	32	96	31	38	24	87	75	2	15	73	54

(ii) Array implementation of binary tree



(iii) Binary tree converted to a max-heap

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
96	87	73	76	75	17	54	38	24	32	56	2	15	31	7

(iv) Array implementation of heap

Heap Building

Algorithm

The heap building procedure is performed in bottom up fashion, starting with the *parental node of the last right-most leaf*. It consists of following steps:

Step # 1: Select the key at *parent of the last node* in the tree

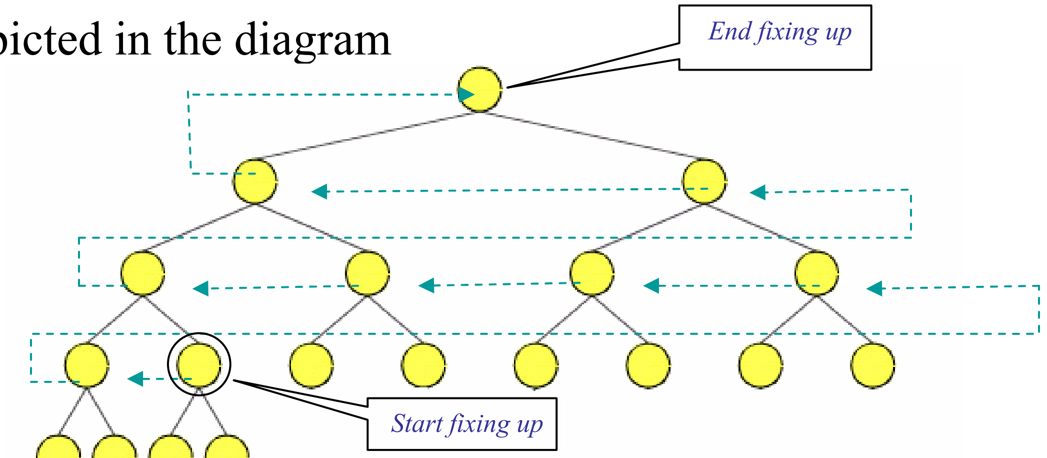
Step #2 : *Heapify* the tree using the key in the chosen node

Step #3: Choose the key in the next *adjacent node* on the left at the *same level* or the key in the *extreme right node at next higher level*. Fix-up the heap using the selected key

.

Step #4: Repeat *Step # 3*, until the root is reached.

- The steps are depicted in the diagram



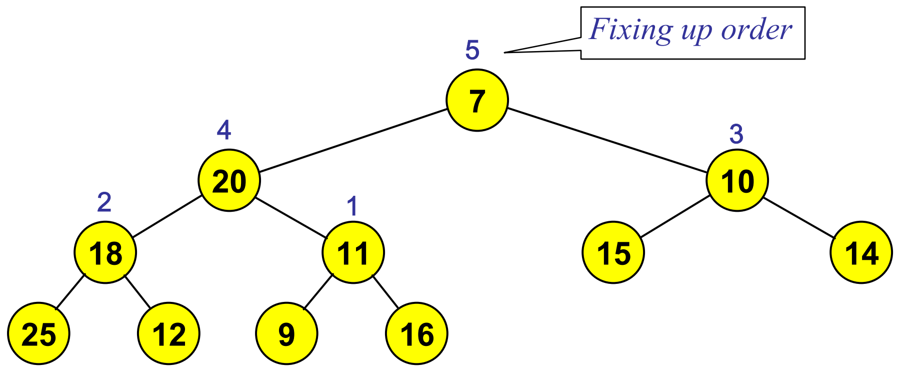
Heap Building Procedure

Heap Building

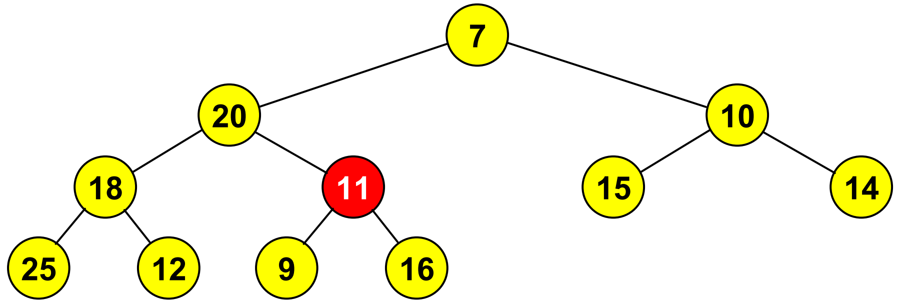
Example

The heap building procedure is demonstrated by the following example

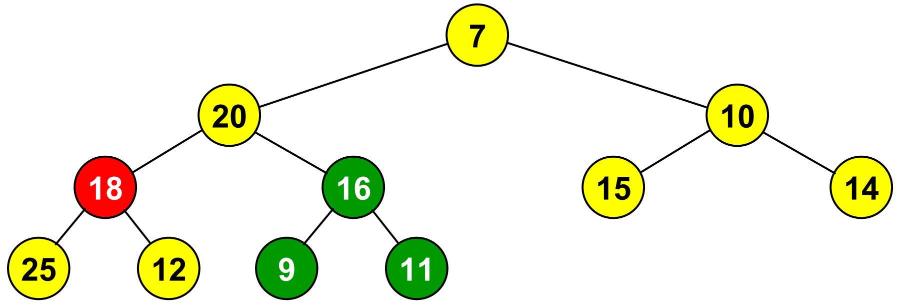
*(1) The sample binary tree is complete. However, it contains a set of keys that violate the heap order property. The tree is converted into a heap by fixing-up the keys, **starting with the parent of the last key 16**, and proceeding **from right to left and bottom up manner**. The numbers on top of nodes show order of fixing up*



(2) The last key in the tree is 16, and its parent contains the key 11. This key is fixed up by exchanging it with the larger key in the right child. The fixed up keys are shaded green



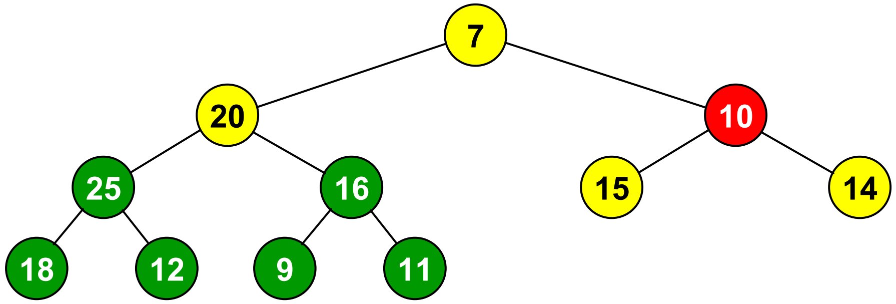
(3) Next the key 18 in the left child is fixed up by exchanging it with the larger key 25 in the left child



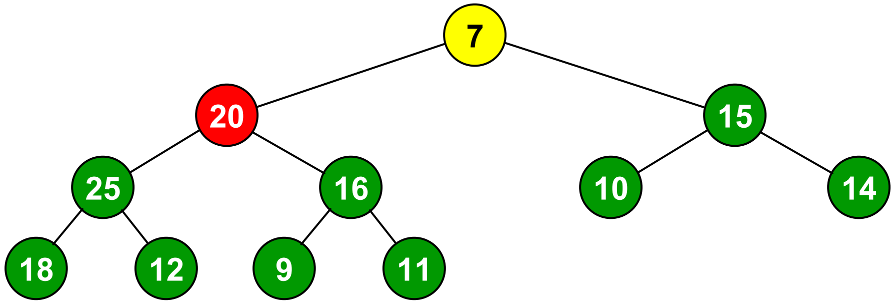
Heap Building

Example

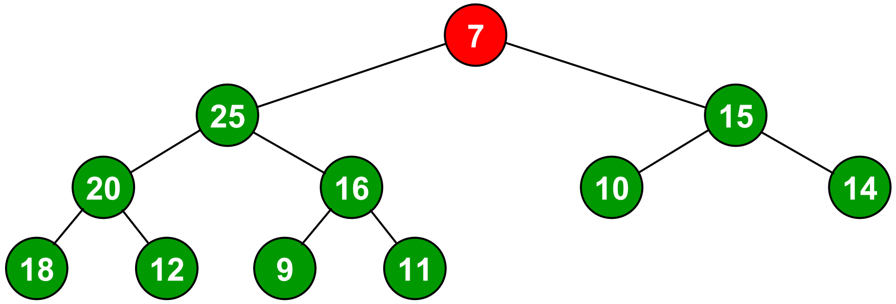
(4) The rightmost key 10, at next higher level is considered. It is fixed up by exchanging with the larger key 15 in the left child



(5) Next the key on the left at the same level is fixed up. It is exchanged with the larger key in the left node

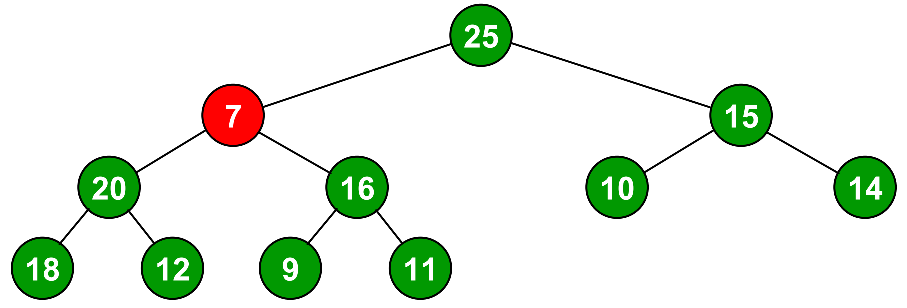


(6) Finally, the key 7 in the root is fixed up. It is exchanged with the larger key in the left child.

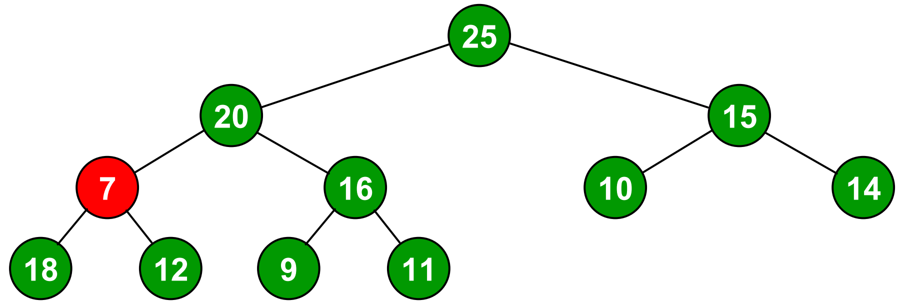


Heap Building
Example

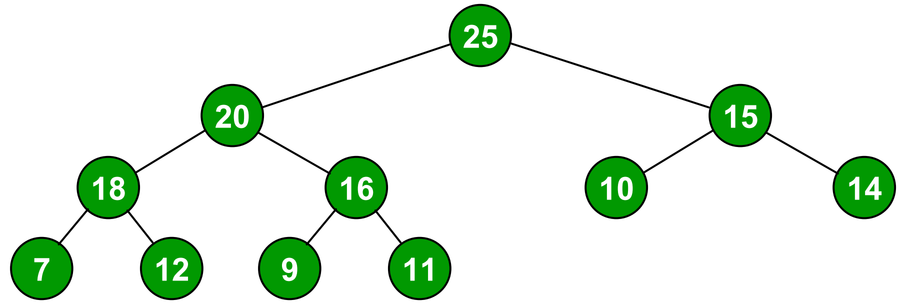
(7) The key 7 in new position violates the order property. It is fixed up by exchanging with the larger key 20 in the left child



(8) The key 7 in new position violates the order property. It is fixed up by exchanging with the larger key 18 in the left child



(9) The binary tree is now converted into a ***max heap****. The tree is complete. Key in each node is larger than the keys in the left child and the right child*



Heap Building Algorithm

Implementation

The heap building procedure works in top-up manner, starting with the parental node of last right most child at the bottom level (*Ref: T.Cormen et al , with slight modification*). The pseudo code for heap building procedure is as follows.

```
BUILD-HEAP( A)                               ► A is input array to be converted to heap
    n ←length [A]                            ► n is the number of keys in the array

    for i ← ⌊ length [A] / 2 ⌋ down to 1     ► ⌊ length [A] / 2 ⌋ is the index of
                                               parent of the last node in the tree

        do  HEAPIFY ( i, n, A)               ► Call heapify procedure to fix-up
                                               the next key with index i
```

Heap Sorting
Algorithm

Heap sorting works in two phases:

Phase-I (Heap Building)

Step #1**:** Map array to a binary tree

Step #2: Convert the binary tree to a heap

Phase-II (Keys exchanges and heap fixing)

Step #1: Exchange last key in the heap with the first key in the root

Step # 2 *:* Detach last key from the tree

Step #3: Covert the subtree, without the detached nodes, into heap by fixing-up key in the root

Step #4: Repeat *Step#2* through *Step #3* until the subtree reduces to a pair of nodes

➢ All detached keys and the last pair of keys in the subtree are now sorted in ascending order

Heap Sorting

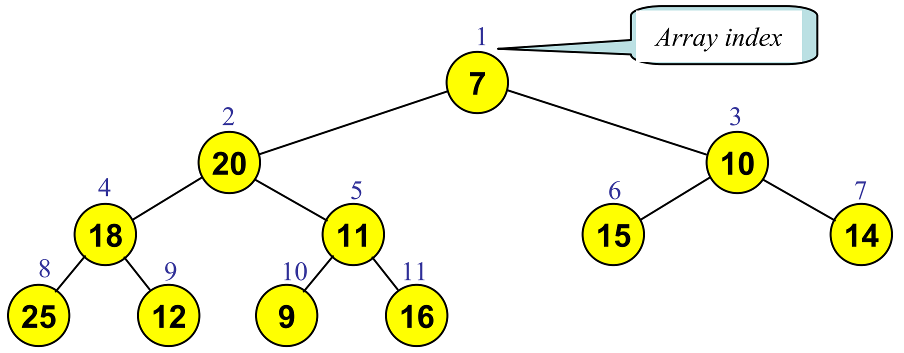
Example

The heap sorting algorithm is is demonstrated by the following example.

(1) An ***unsorted sample array*** *is show in figure (a). It is mapped into a* ***complete binary tree*** *depicted in figure (b). The numbers on top of the tree nodes are the indexes of the array cells. Thus, the root contains the first key 7 in the array, and the last node at the bottom stores key 16 in cell 11 of the array*

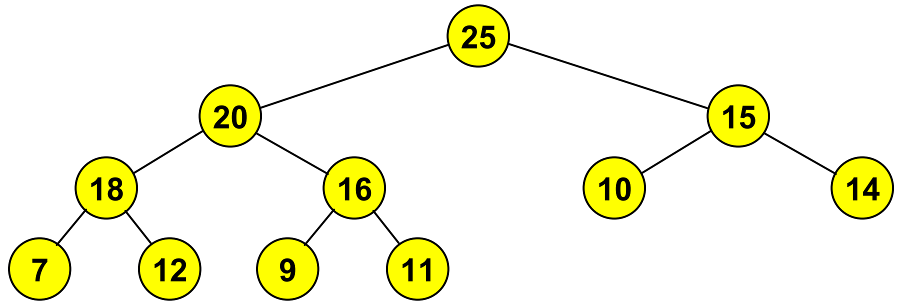
index	1	2	3	4	5	6	7	8	9	10	11
key	**7**	**20**	**10**	**18**	**11**	**15**	**14**	**25**	**12**	**9**	**16**

(a) Sample unsorted array



(b) Binary tree representation

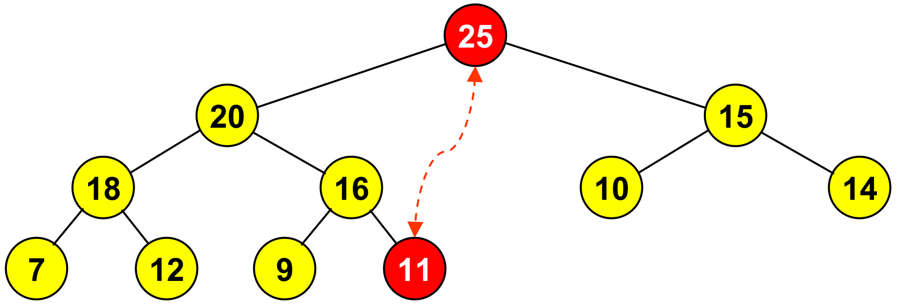
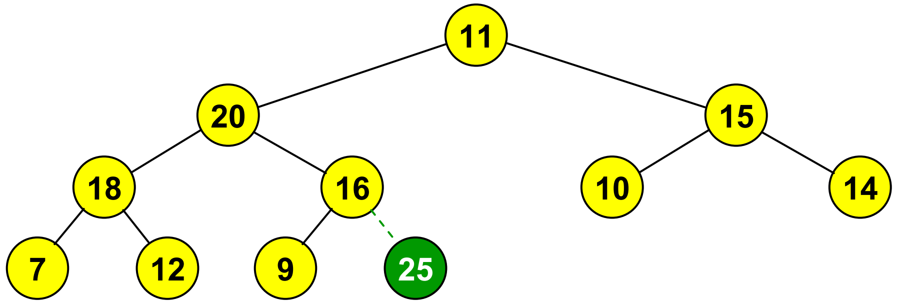
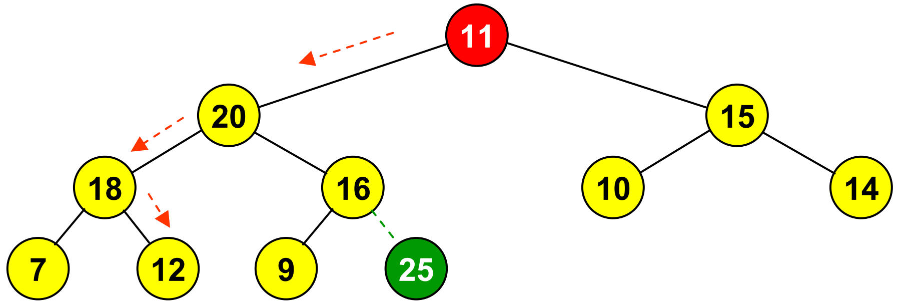
(2) The binary tree is converted into a ***max heap*** *by using the* ***heap building procedure,*** *as illustrated in the last example. The heap is shown in figure (c). It will be processed further to arrange the keys into an ascending order.*



(c) Tree converted into max heap

Building Heap

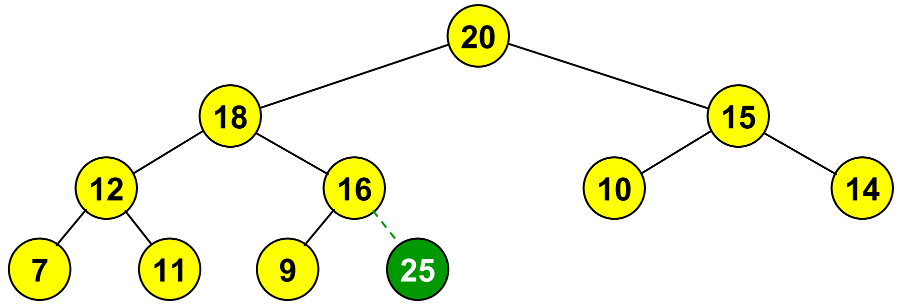
Example

*(3) In order to perform heap sorting, the **key 25 in the root is exchanged with the last key 11** in the heap, as shown the diagram*	 25 20 15 18 16 10 14 7 12 9 11
*(4) After the exchange, the **last key is discarded (de-linked) from the tree.** The tree now consists of **ten nodes.** The detached node is shaded green. The missing link is depicted by broken line.*	 11 20 15 18 16 10 14 7 12 9 25
(5) The key 11 in the root violates heap property. It is fixed up by exchanging with the larger keys in the child nodes. The movement of key 11 is depicted by arrows.	 11 20 15 18 16 10 14 7 12 9 25

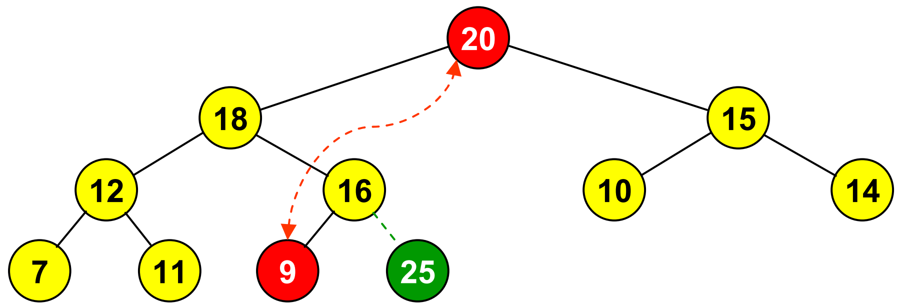
Building Heap

Example

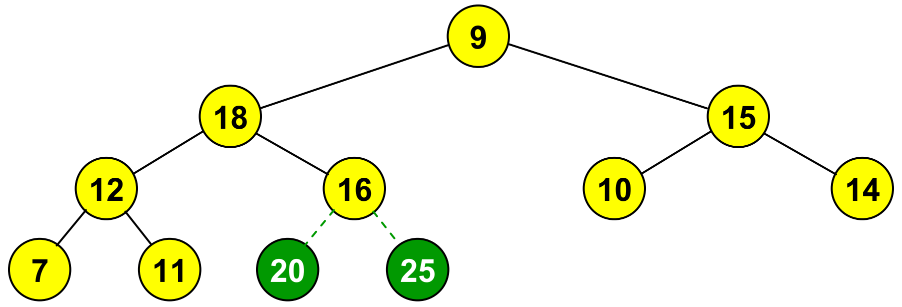
(6) As a result of fixing up, the key 20 is moved to the root, and key 11 to the bottom level



(7) Next, the key in the root 20 is exchanged with the last key 9 in the newly built heap



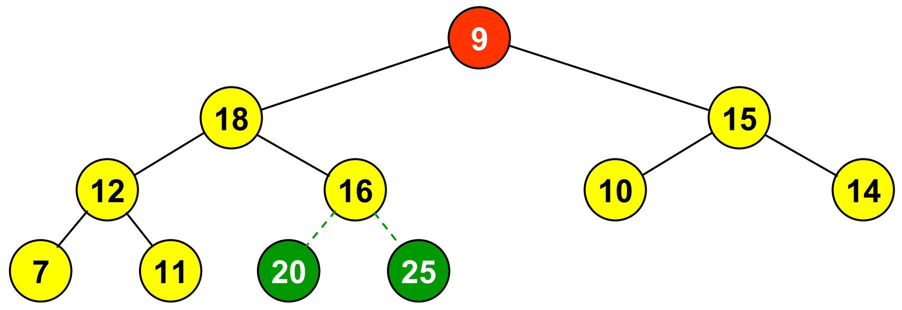
(8) After the exchange, the last key is de-linked from the heap. The subtree consists of ***nine nodes***



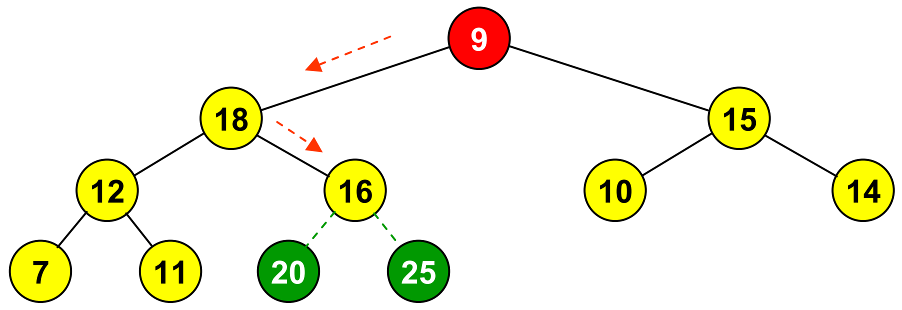
Building Heap

Example

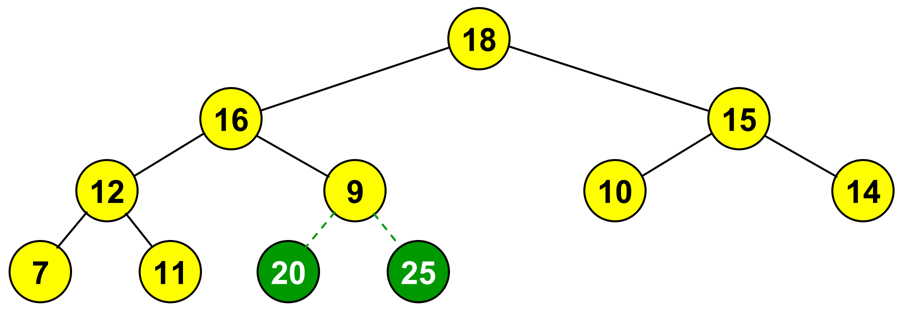
(9) The key 9 in the root, violates the max heap property.



(10) The heap is fixed up by exchanging the key 9 with larger keys in the child nodes



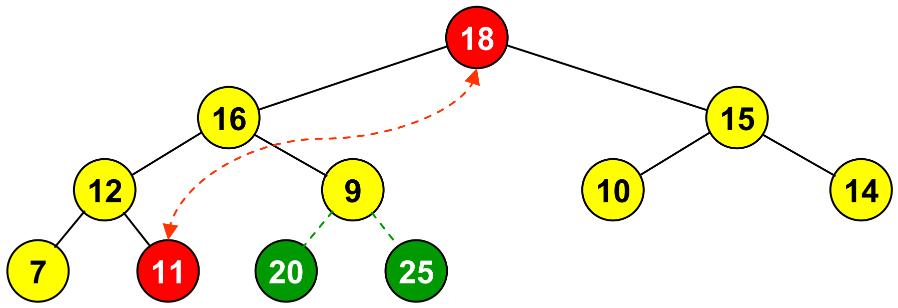
(11) The subtree is now converted into max heap



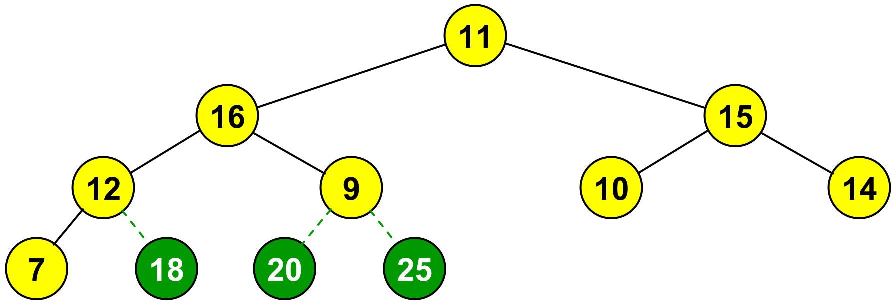
Building Heap

Example

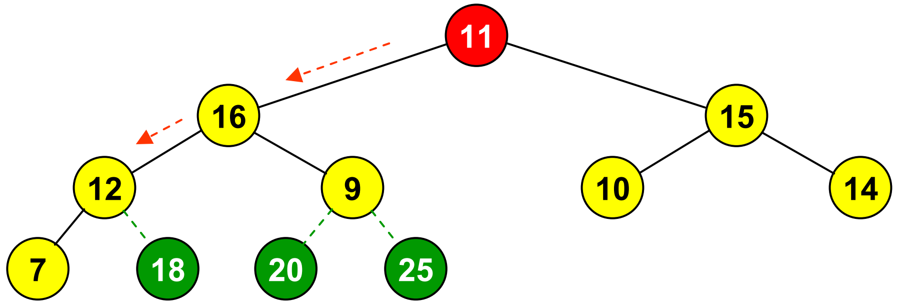
(12) The key 18 in the root is exchanged with the last key 11 in the newly built tree



(13) After exchange of keys, the last node of the tree with key 18 is detached from the tree. The tree now consists of ***eight nodes***



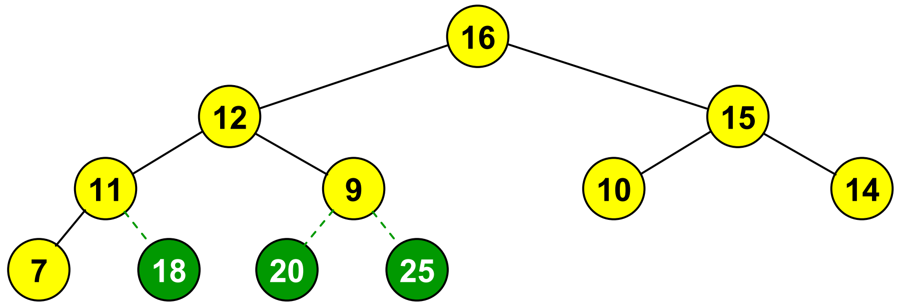
(14) The key 11 in the root violates the max heap property. It is fixed up by exchanging with the larger keys 16 and 12 in the child nodes



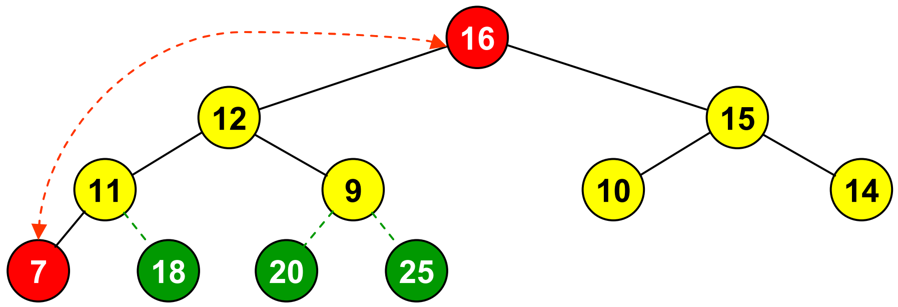
Building Heap

Example

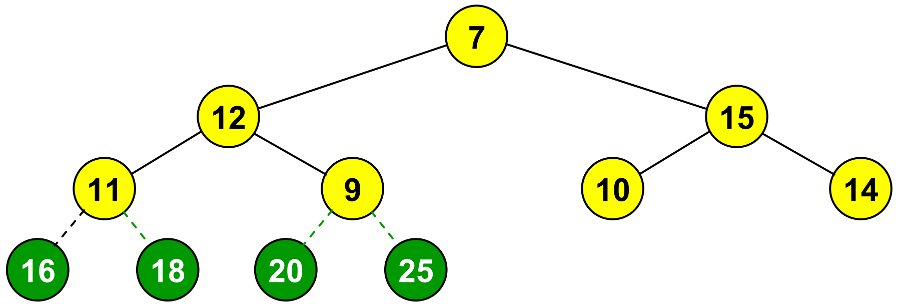
(15) The subtree is now transformed into a max heap , with key 16 in the root and key 7 in the last node



(16) The key 16 in the root is exchanged with the key 7 in the last node of the heap

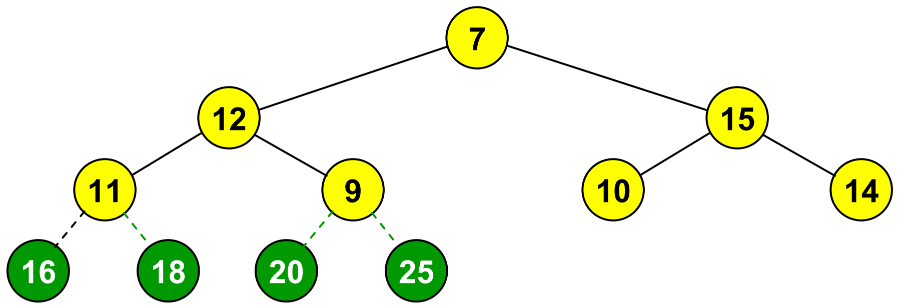
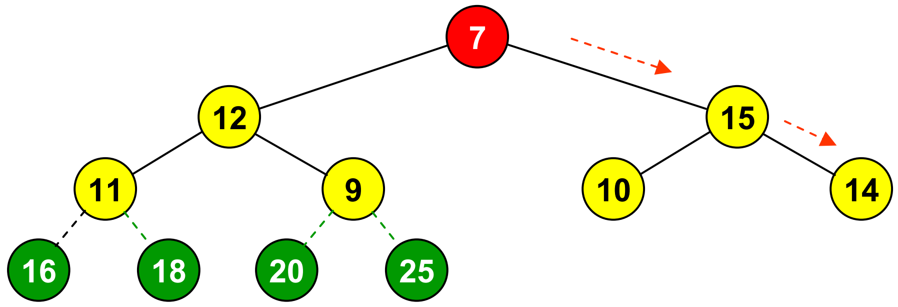
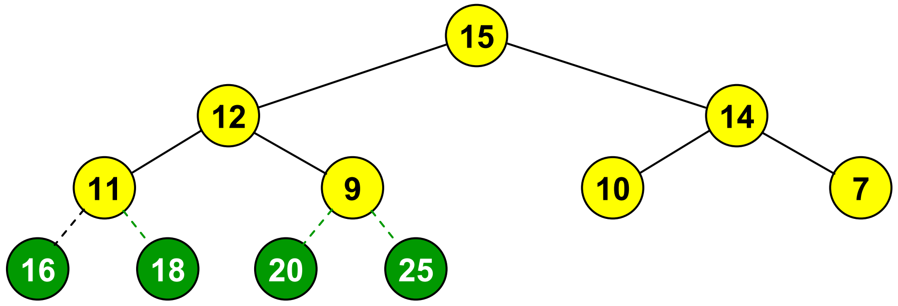


(17) After the exchange, the node containing key 16 is detached from the tree. The tree now consists of ***seven nodes.***



Building Heap

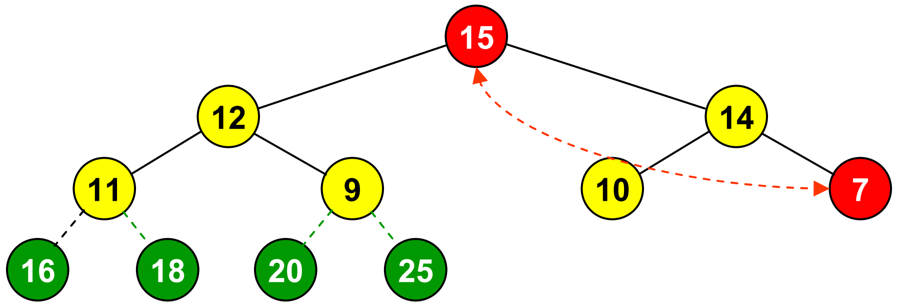
Example

(18) After the exchange the key 7 is in the root . The last key in the subtree is 14	
(19) The key 7 in the root violates max heap property. It is fixed up exchanging with the larger keys 15 and 14 in the child nodes	
(20) Heap is now fixed up with key 15 in the root and key 7 in the last node	

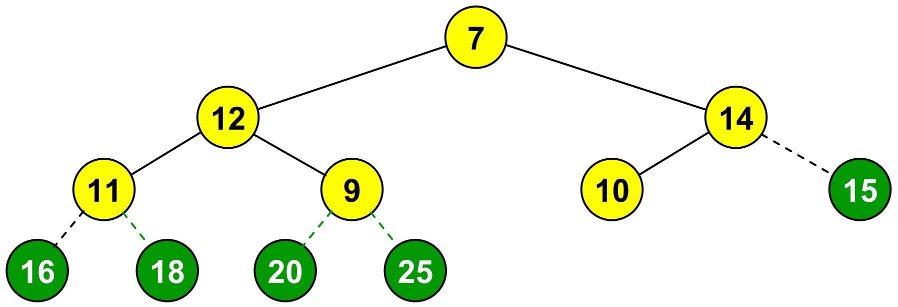
Building Heap

Example

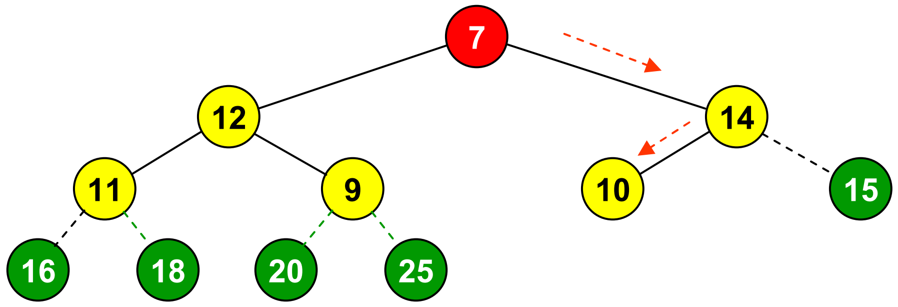
(21) The key 15 in the root is exchanged with last key 7 in the heap



(22) The node with exchanged key 15 is removed from the tree. The tree now consists of ***six nodes***



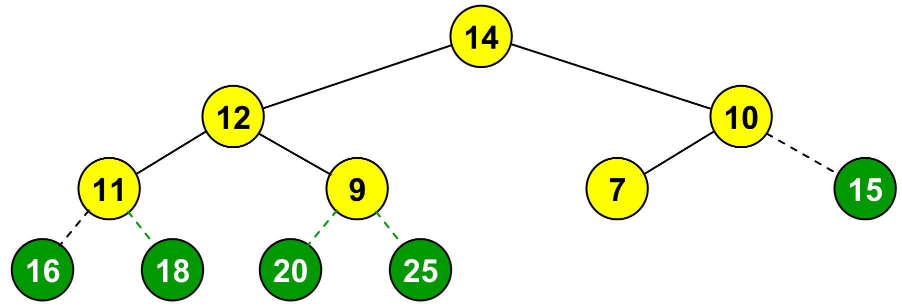
(23) The key 7 in the root violates the max heap property. It is fixed up by exchanging with the larger keys 14 and 10 in the child nodes



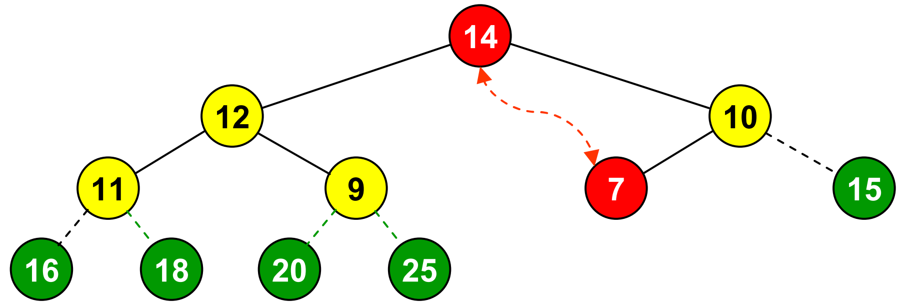
Building Heap

Example

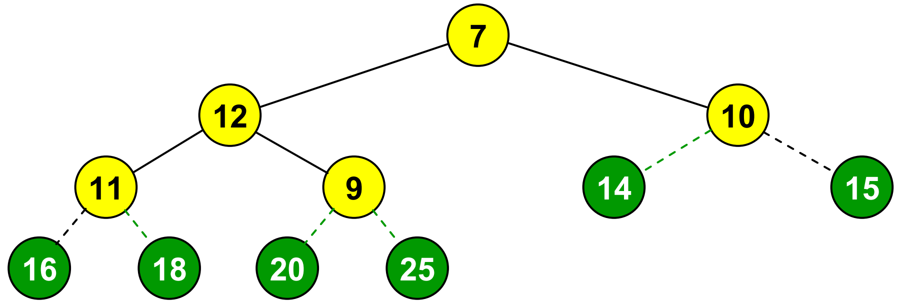
(24) The heap is now fixed up, with key 14 in the root and key 7 in the last node.



(25) The key in the root is exchanged with the last key in the heap



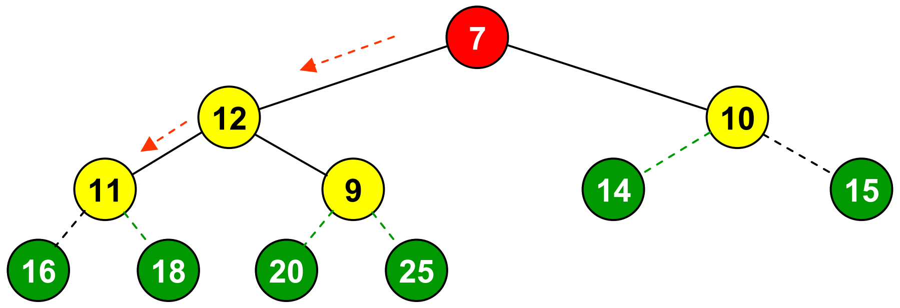
*(26) The last key is de-linked from the tree. The tree now contains **five nodes***



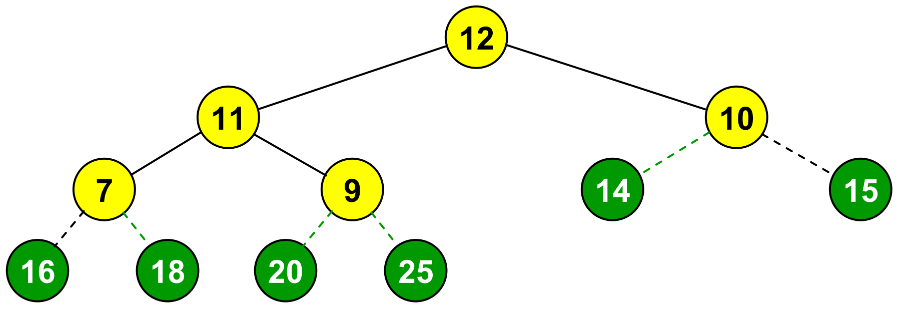
Building Heap

Example

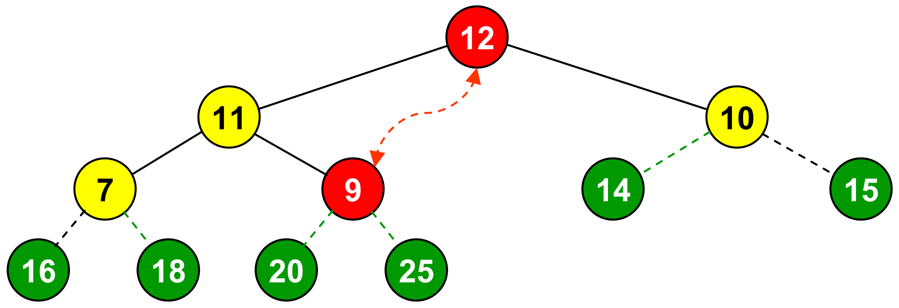
(27) The key in the root violates heap order property. It is fixed by exchanging with the larger keys 12, 11 in the child nodes



(28) The subtree is converted into a max heap, with largest key in the root.



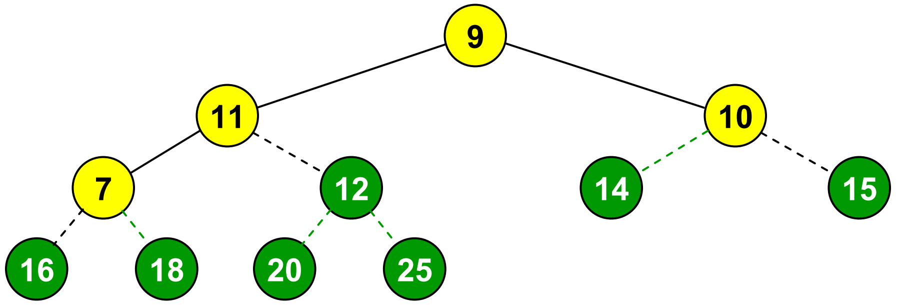
(29) The key 12 in the root is exchanged with the last key 9 in the heap



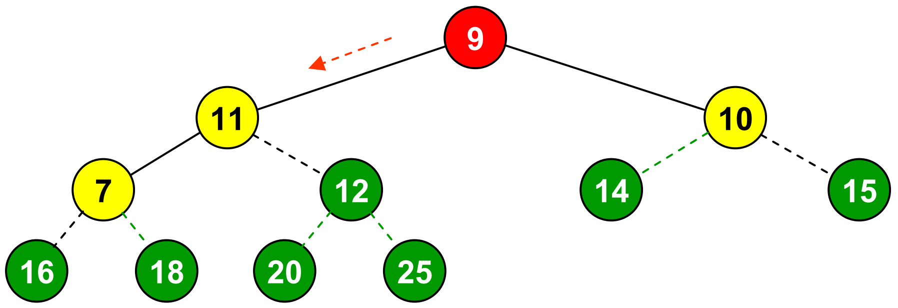
Building Heap

Example

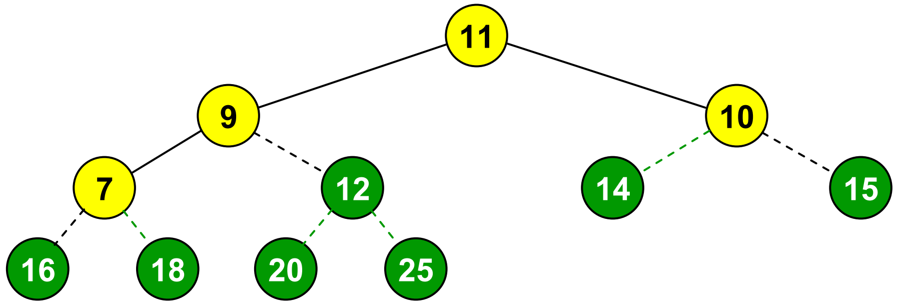
*(30) After the exchange, the node with key 12 is detached from the subtree. The tree now consists of **four nodes***



(31) The key 9 in the root violates the heap order property. It is fixed up by exchanging with the larger key 11 in the child node



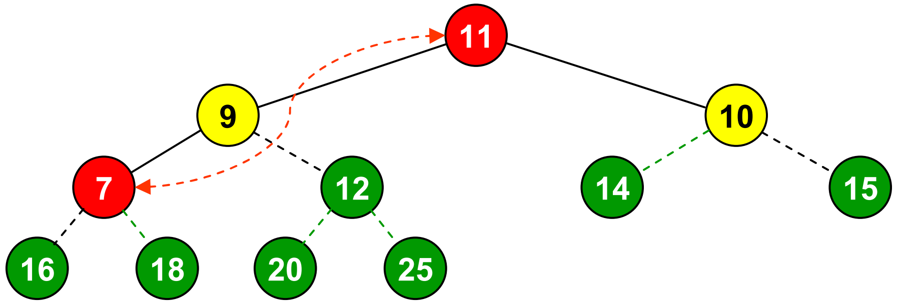
(32) The subtree is converted into a max heap



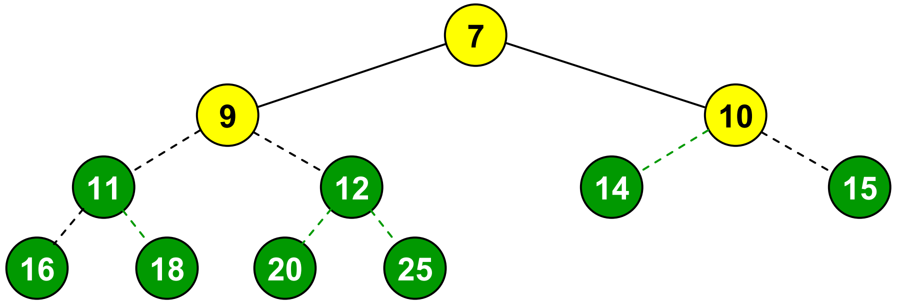
Building Heap

Example

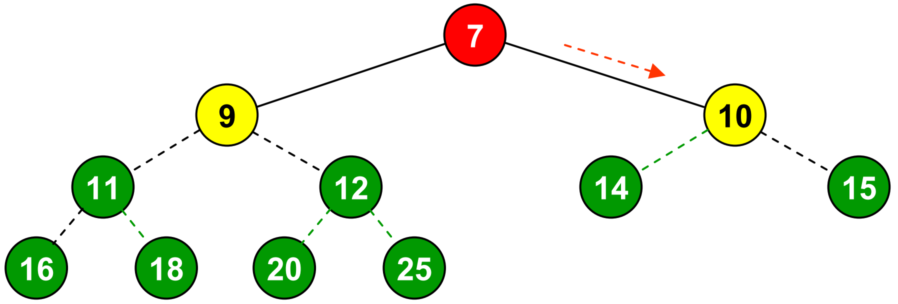
(33) The key 11 in the root is exchanged with the last key 7 in the heap



(34) After the exchange the key 11 is detached from the tree. The subtree consists of ***three nodes***



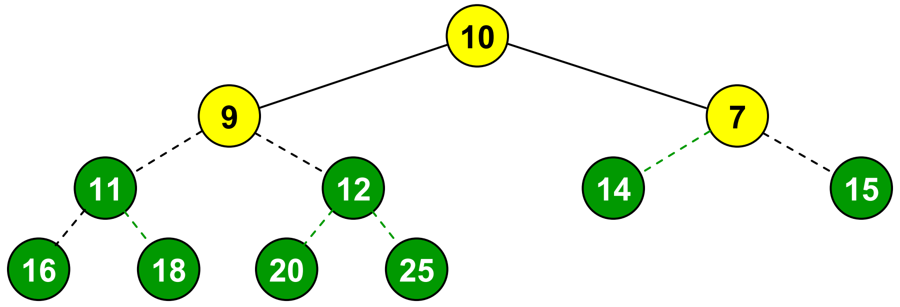
(35) The key 7 in the root violates heap property. It is exchanged with the larger key 10 in the right child



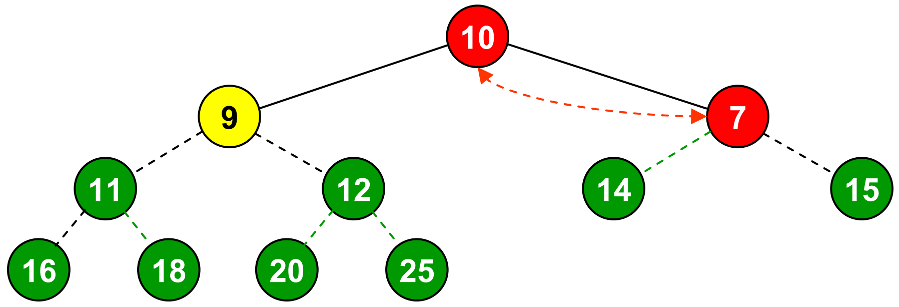
Building Heap

Example

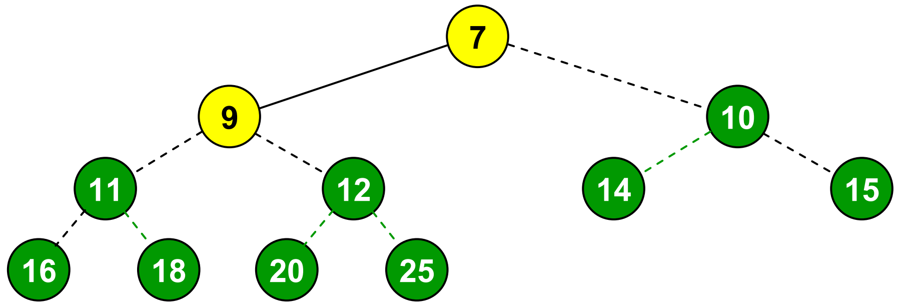
(36) The max heap consists of three nodes, with the largest key 10 in the root.



(37) The key in the root is exchanged with the key in the last node of the heap



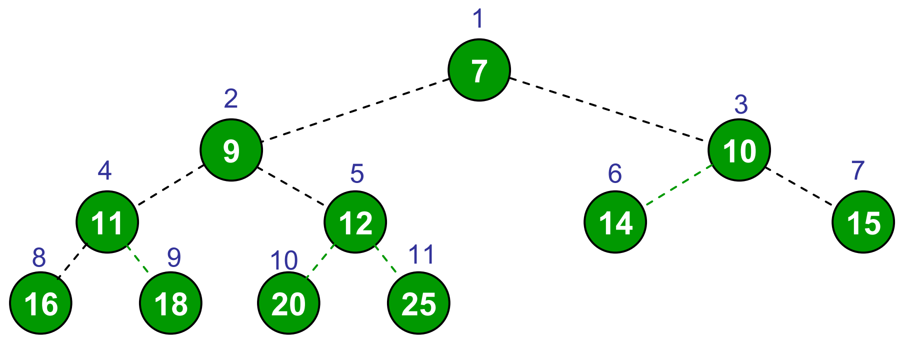
*(38) After the exchange, the node with key 10 is detached from the subtree The subtree now consists of **two nodes.** The sorting procedure terminates.*



Building Heap

Example

(40) The last node with key 9 is detached from the root. Beginning with the root, the detached nodes are ***now arranged in ascending order.*** *The order is given by the a number on top of each node, as shown in figure (a)*



(a) The keys in the tree nodes are arranged in ascending order

(41) Figure (b) shows the array corresponding to the binary tree. It stores keys which have been ***sorted in ascending order*** *. For comparison, figure (c) shows the original unsorted array*

1	2	3	4	5	6	7	8	9	10	11
7	**9**	**10**	**11**	**12**	**14**	**15**	**16**	**18**	**20**	**25**

(b) Sorted Array

1	2	3	4	5	6	7	8	9	10	11
7	**20**	**10**	**18**	**11**	**15**	**14**	**25**	**12**	**9**	**16**

(c) Original unsorted array

Heap Sorting

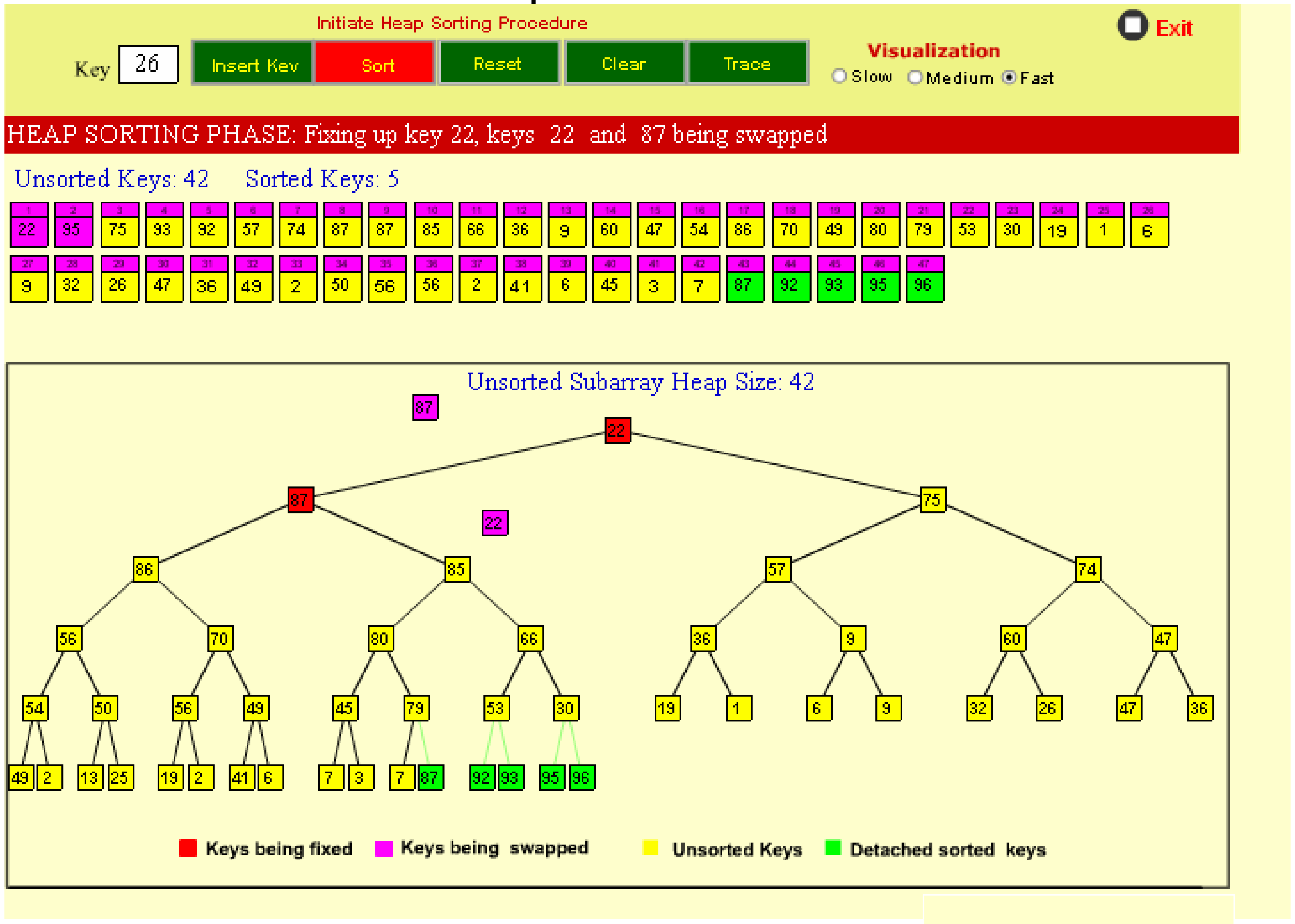
Implementation

The HEAPSORT method sorts an input array *A[1..n]* into ascending order by using order property of max heap. It uses two other heap procedures : **BUILD-HEAP** and **HEAPIFY** (*Source: T.Cormen et al*). The first procedure converts the array into a ***max heap***. The second procedure ***fixes the heap.*** In each cycle the key at the root is exchanged with the last node of the tree. The larger of the exchanged key is discarded

```
HEAPSORT( A)
1  BUILD-HEAP(A)                     ► Call BUILD-HEAP procedure to convert array into a heap
2  for j ← length[A] downto 2        ► Use bottom-up procedure
3  do exchange A[1] ↔ A[j]           ► Exchange last key in the heap with the key in the root
4       heap-size ← heap-size – 1    ► Reduce the size of heap by one ( one key is discarded)
5       HEAPIFY(j, heap-size, A )    ► Fix up the key at root by calling HEAPIFY procedure
```

Visualization

Heap Sort Visualization



Visualization: Slow Medium [✓ Fast]

HEAP SORTING PHASE: Fixing up key 22, keys 22 and 87 being swapped

Unsorted Keys: 42   Sorted Keys: 5

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26
22	95	75	93	92	57	74	87	87	85	66	36	9	60	47	54	86	70	49	80	79	53	30	19	1	6

27	28	29	30	31	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47
9	32	26	47	36	49	2	50	56	56	2	41	6	45	3	7	87	92	93	95	96

Unsorted Subarray Heap Size: 42

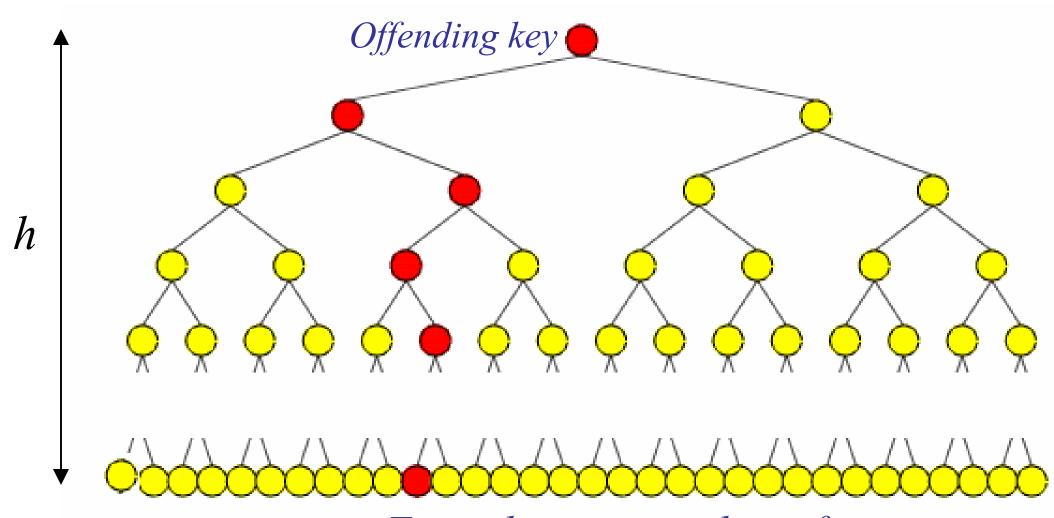
🟥 Keys being fixed 🟪 Keys being swapped 🟨 Unsorted Keys 🟩 Detached sorted keys

Analysis of Heap Operations

Analysis of Heap Fixing

Worst Case Scenario

An *offending key* is the key in some tree node that violates the heap order property. In worst case the offending key appears at the root. In this situation the heap completeness property is not restored until the key is moved to the bottom level. .



Typical worst case heap fixing

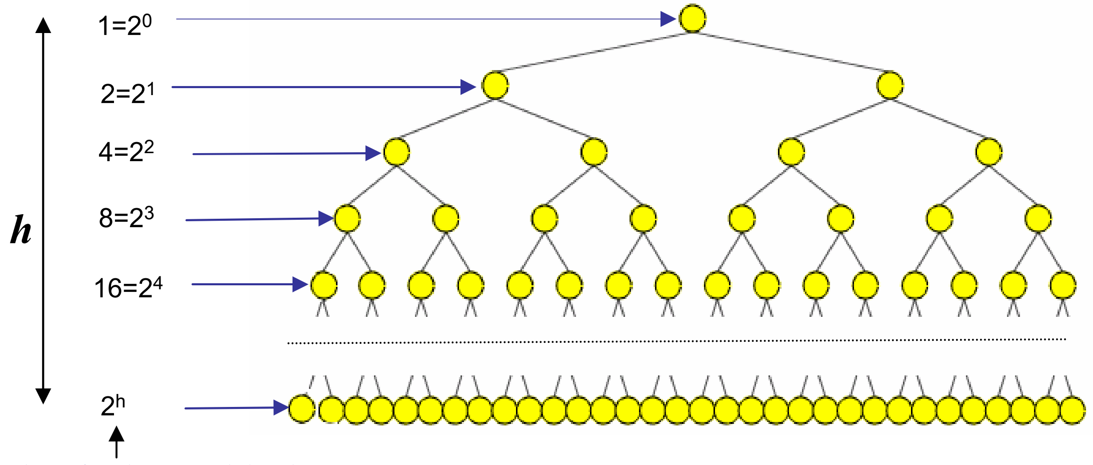
At each level, the offending key is compared twice: First, the key in the parent node is compared with the key in a left child node . Next, the larger of the keys is compared with the key in the right child. If *c* is cost of one comparison, then total number of comparisons in worst case would be *2ch*, where *h* is the height of the heap. Thus, the *worst running time T(h)*, in terms of *heap height*, is given by

$$T_{worst}\,(h) = 2ch$$

Analysis of Heap Fixing

Worst Running Time

In order to express the running time in terms of number of nodes, we assume that the heap is full. In other words, the last level contains *maximum number of nodes*, as shown in the figure



Number of nodes at each level

The binary tree contains 2^0 nodes at level 0, 2^1 nodes at level $1 \ldots 2^h$ nodes at level h

If n is total number of nodes

$$n = 2^0 + 2^1 + 2^2 + \ldots\ldots\ldots\ldots + 2^h = 2^{h+1} - 1 \quad \textit{(Summing geometric series)}$$

Taking logarithm, and rearranging

$$n = lg(n+1) - 1$$

As shown previously, the worst running time for fixing up the is

$T_{fix}(h) = 2ch$, where c is unit cost of comparing a pair of keys and h is heap height

Substituting for h, the *worst running time for fixing a heap of size n* is

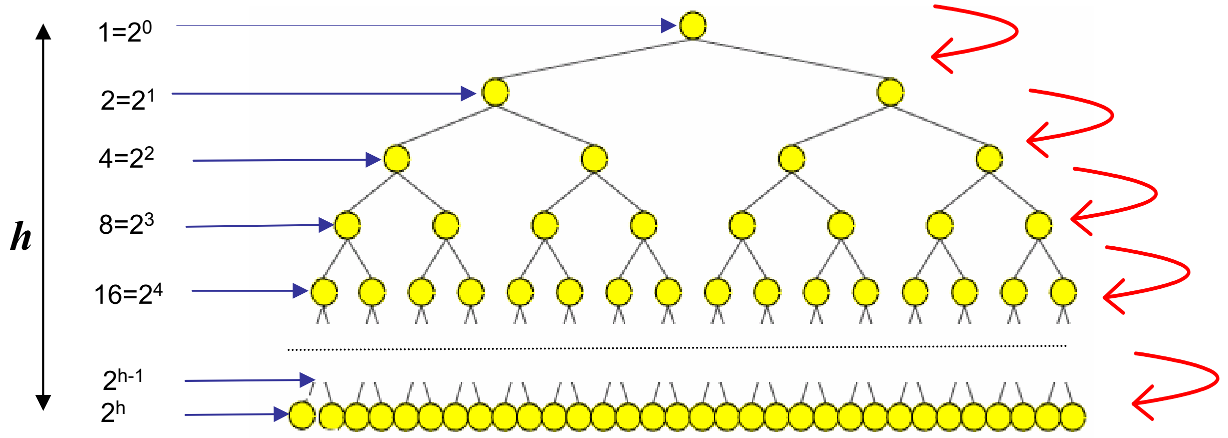
. $T_{fix}(n) = 2c\,(\,lg(n+1) - 1)$ *(closed form)*

Since $\lim_{n\to\infty} T_{fix}(n) / lg(n) = 2c$ (constant), it follows that $\boldsymbol{T_{fix}(n) = \theta(\,lg\ n)}$

Analysis of Heap Building

Worst Case Scenario

We assume that heap is represented by full tree of height *h*, as depicted in the figure



In *worst case of fixing-up the heap, all of the keys descend down to the bottom level* . Further, at each step the offending key is *compared twice* with the left and right children of parental node . Let *c* be the unit cost of comparison. The table summarizes *total cost of comparison of keys at each level*

Level	Number of Nodes	Count of shifting down	Total Cost
0	2^0	h	$2c.h.2^0$
1	2^1	$h-1$	$2c.(h-1).2^1$
2	2^2	$h-2$	$2c.(h-2).2^2$
--	--------	--------	--
i	2^i	$h-i$	$2c.(h-i).2^i$
---	------	------	--
$h-2$	2^{h-2}	2	$2c.2.2^{h-2}$
$h-1$	2^{h-1}	1	$2c.1.2^{h-1}$

Analysis of Heap Building

Worst Running Time

The worst time T_{build} for building the heap is obtained by summing the last column :

$T_{build} = 2ch.2^0 + 2c(h\text{-}1).2^1 + 2c(h\text{-}2).2^2 + \ldots\ldots + 2c.2.2^{h-2} + 2c.1.2^{h-1}$ ………(1)

As shown before, $2^{h+1} = n+1$ and $n = lg(n+1) - 1$, n being the heap size …………….(2)

Using summation notation

$$T_{build} = 2c\sum_{i=0}^{h-1}(h - i)\, 2^i$$

$$= 2ch\sum_{i=0}^{h-1} 2^i \; - \; 2c\sum_{i=0}^{h-1} i\, 2^i \quad \ldots\ldots\ldots\ldots\ldots (3)$$

$\sum_{i=0}^{h-1} 2^i = 2^h - 1$ *(summing the geometric series)*

$\sum_{i=0}^{h-1} i2^i = (h-2)2^h + 2$ *(summing aritho-geometric series* $\sum_{i=1}^{m} i2^i = (m-1)\,2^{m+1} + 2$ *, ref math prelim)*

Substituting into equation (1), and simplifying,

$T_{build} = 2c\, 2^{h+1} - 2ch - 4c$

From result (2), it follows

$T_{build}(n) = 2c[\,(n+1) - (\,lg(n+1) - 1\,) - 2]$ ………(4)

On simplifying (4), it follows that the *worst running time for building heap of size n* is

$T_{build}(n) = 2c(\,n - lg(n+1)\,)$ *(closed form)*

Since , $\lim_{n\to\infty} T_{build}(n) \,/\, n = 2c(\,n - lg(n+1)\,) \,/\, n = 2c$ (positive constant)
it follows that *worst running time* is

$T_{build}(n) = \theta(n)$ *(asymptotic form)*

Analysis of Heap Sorting

Running Time

The heap sorting algorithm uses two procedures : *BUILD-HEAP* and *HEAPIFY*. Thus, total running time $T_{sort}(n)$ to sort an array of size *n* is

$$T_{sort}(n) = \underbrace{T_{build}(n)}_{\text{Time to build heap}} + \underbrace{T_{fix}}_{\text{Time to fix-up heaps of sizes n-1 to 2}}$$

It was shown previously that, in worst case, the BUILD-HEAP procedure takes $\theta(n)$ time.
Thus, $T_{build} = \theta(n)$

Analysis of Heap Sorting

Running Time

The fixing-up procedure is applied repeatedly to sub-tree of sizes $n-1, n-2, \ldots 4,3$

From analysis of heap fixing it follows that running time to fix a heap of size n is given by

$$T_{fix}(n) = 2c(\lg(n+1) - 1)), \quad \ldots\ldots\ldots\ldots\ldots(1)$$

where c cost of comparing keys

In heap sorting algorithm, the fixing-up procedure is applied repeatedly to subtrees of sizes $n-1, n-2 \ldots, 3$. Using result of equation *(1),*

$$\begin{aligned} T_{fix}(n) &= 2c[(\lg(n)-1) + (\lg(n-1)-1) + \ldots + (\lg(4)-1)] \\ &= 2c[\lg((n).(n-1)\ldots\ldots 4) - (n-2)] \\ &= 2c[\lg(n)! - n - \lg(6) + 2] \\ &= 2c[\lg(n!) - n - \lg(6) + 2] \\ &= 2c[\lg(\sqrt{(2\pi n)}\,(n/e)^n) - n - \lg(6) + 2] \quad \textit{(Using Stirling's approximation for n!)} \\ &= c[\lg(n) + 2n\lg(n) - 2n\lg(e) - 2n] + k, \textit{ where k is sum of constant terms} \end{aligned}$$

Now $\displaystyle\lim_{n\to\infty} T_{fix}(n) / n\lg(n) = [c(\lg(n) + 2n\lg(n) - 2n\lg(e) - 2n) + k] / n\lg(n)$

$$= 2c \text{ (constant)}$$

Therefore, $T_{fix}(n) = \theta(n\lg(n))$

$$\begin{aligned} T_{sort}(n) &= T_{build}(n) + T_{fix}(n) \\ &= \theta(n) + \theta(n\lg n) \\ &= \theta(n\lg n) \qquad \textit{(ignoring lower order term n)} \end{aligned}$$

- The *worst running time* of heap sort is $\theta(n\lg n)$