Book

A Simplified Approach

to

Data Structures

Prof.(Dr.)Vishal Goyal, Professor, Punjabi University Patiala

Dr. Lalit Goyal, Associate Professor, DAV College, Jalandhar

Mr. Pawan Kumar, Assistant Professor, DAV College, Bhatinda

Shroff Publications and Distributors

Edition 2014

HEAP

DEFINITION

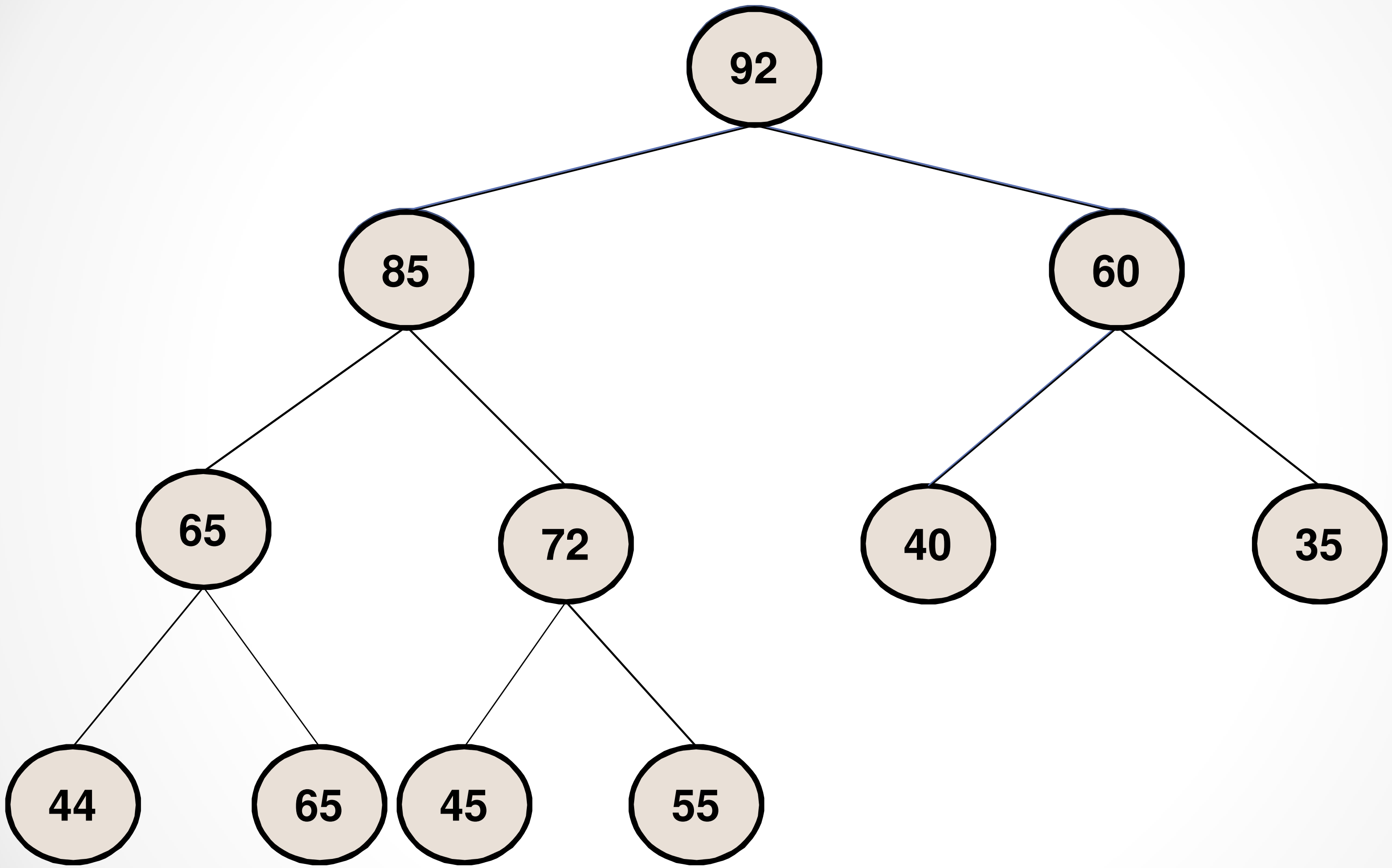
- A **Heap** is a complete binary tree with **n** elements which is maintained in the memory using a linear array.
- It is a very important data structure which can be used efficiently to sort given list of elements.
- There are two types of heaps:
  - If the value present in any node is greater than all its children then such a tree is called as the **max-heap** or **descending heap**.
  - In **min-heap** or **ascending heap** the value present at any node is smaller than all its children.

CHARACTERISTICS

- A Heap is a binary tree which satisfy following characteristics :
  - The binary tree should be almost complete that is all the leaf node should be at $\mathbf{k^{th}}$ or $\mathbf{(k+1)^{th}}$ level.

  - The value at any node is larger or equal to the value at each of its two children.

HEAP



A HEAP WITH 12 NODES(MAX HEAP)

Memory Representation of Heap

- Array representation is very space efficient for maintaining complete or almost complete binary tree.
- As heap is almost complete binary tree ,so heap can be stored in linear array efficiently.

92	85	60	65	72	40	35	16	44	65	45	55

Array representation of heap shown in last slide.

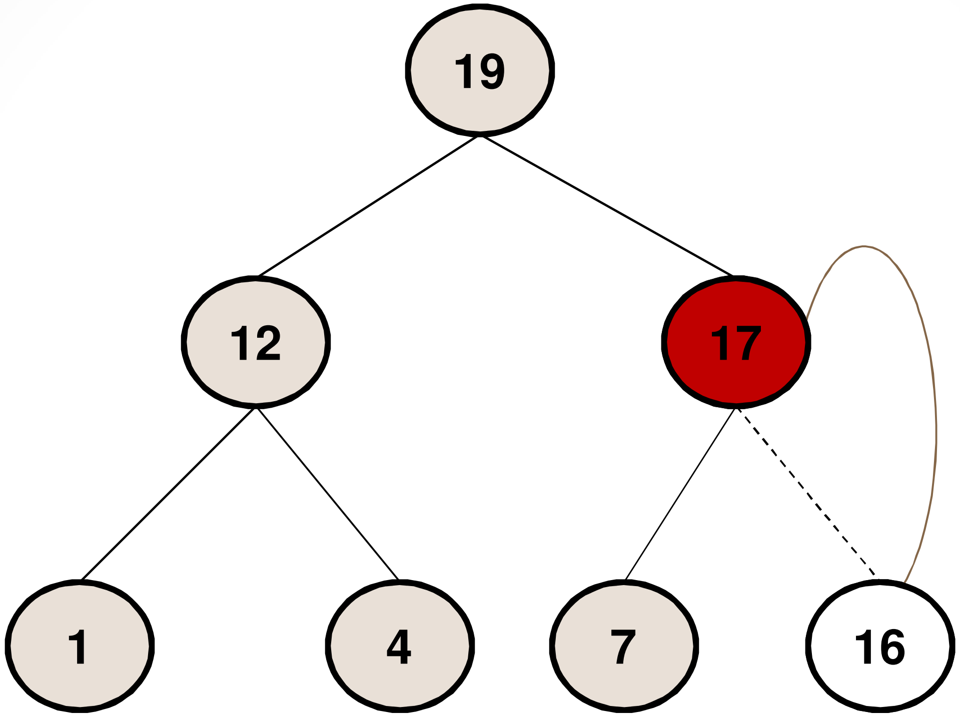
OPERATION ON HEAP

- The two fundamental operation performed on heap are:
  - Inserting an element into a Heap
  - Creating a Heap
  - Deleting an element from Heap

Inserting an element into a Heap

- Consider an array **H** which is a heap and we have a data element **New** that we want to insert into the heap.
  - The data element **New** will be inserted at the end of array **H** , so that **H** is still a complete binary tree
  - After the insertion of **New** element at the end of heap **H** , **H** may not still be a heap . Then newly inserted element **New** will rise up to its appropriate position so that tree again becomes a heap.

Consider a heap **H** of size 6 as shown below. We want to insert an element 17 into this heap.



Inserting new element 17 at the end of the heap.

As 17 is greater than 16, so swapping takes place.

A[1]	A[2]	A[3]	A[4]	A[5]	A[6]	A[7]
19	12	16	1	4	7	17

ALGORITHM

Step1: Set **n=n+1** And **Pos=n**

Step2: Repeat Steps 3 and 4

While H[Pos/2]<=New AND Pos/2>=1

Step3: Set **H[Pos]=H[Pos/2]**

Step4: Set **Pos=Pos/2**

[End Loop]

Step5: **H[Pos]=New**

Step6: Exit

CREATING A HEAP

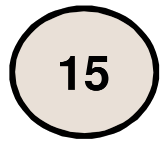
- While creating a Heap **H**, we **insert** the given **element** into the heap **sequentially** one by one.
- The **size** of the heap **increases** as an element is inserted into the heap.
- To create a heap of size **n,** the **nth** element is placed into the existing heap of size **n-1.**
- The new is first placed at the end of the array such that the constraint of the **almost complete tree** is maintained.

CONTINUED

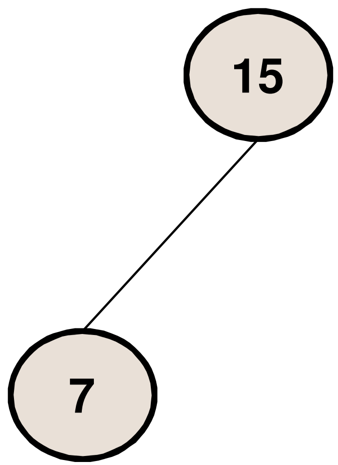
- Then the **new** inserted element is **compared** with its **parent element** and if **>** its parent element then it is interchanged with its parent child and this interchanging is done until either the parent element is greater or the root of the tree is reached.
- Suppose, we want to create a heap **H** from the given list of numbers:

15	**7**	**10**	**2**	**20**	**15**	**8**

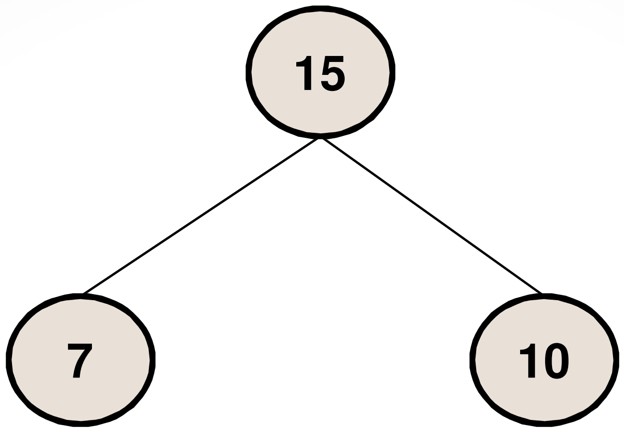
STEP1:
INSERT 15



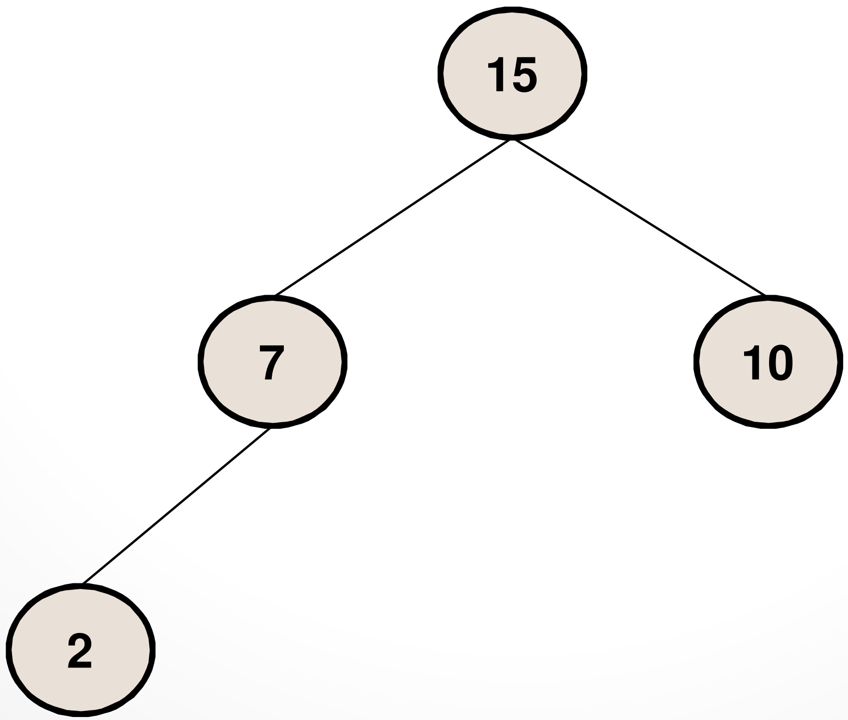
STEP2:
INSERT 7



STEP3:
INSERT 10



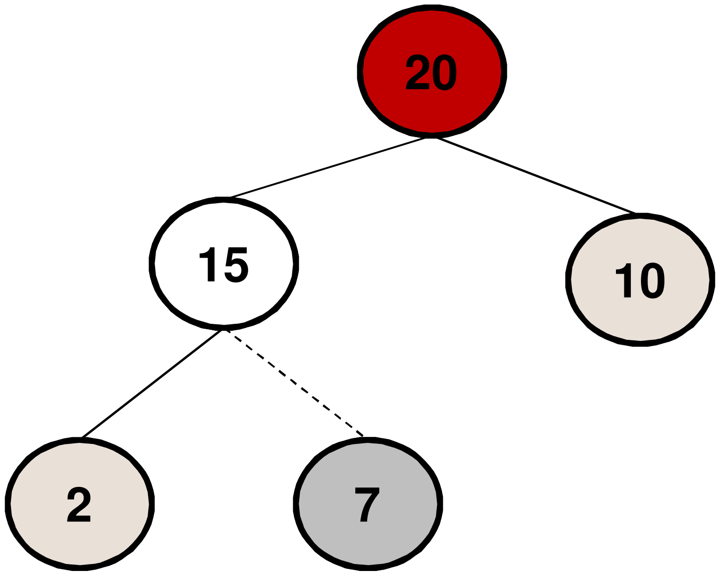
STEP4:
INSERT 2



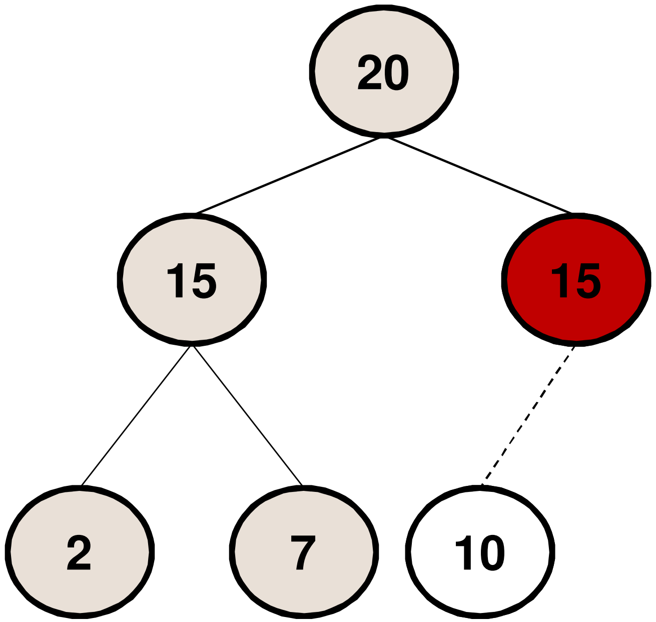
STEP5:
INSERT 20

As 20 is greater than 7.So they are swapped.

20 is greater than 15.so swap these two nodes.

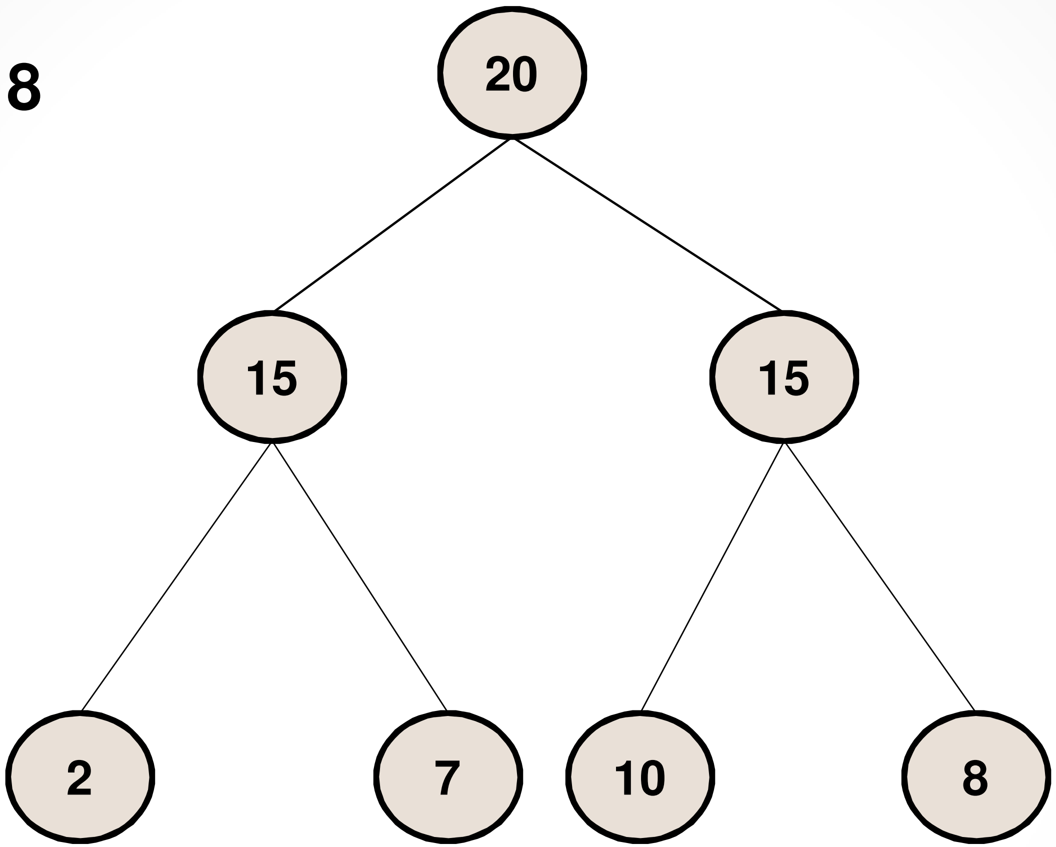


STEP6:
INSERT 15



As 15 is greater than 10.So they are swapped.

STEP7:
INSERT 8



Heap is created.

DELETING AN ELEMENT FROM HEAP

- In a heap an element is always deleted from the root. Consider a heap of **n** elements which is maintained in an array **H.** The deletion operation can be accomplished as:
- First of all, we will store the root element of the heap i.e. **H[1]** into a variable ***item.***
- We replace the root element of the heap with the last element of the heap i.e. **H[n]** and decrease the size of the array by 1. At this stage the array **H** is a complete binary tree but not necessarily a heap.

CONTINUED

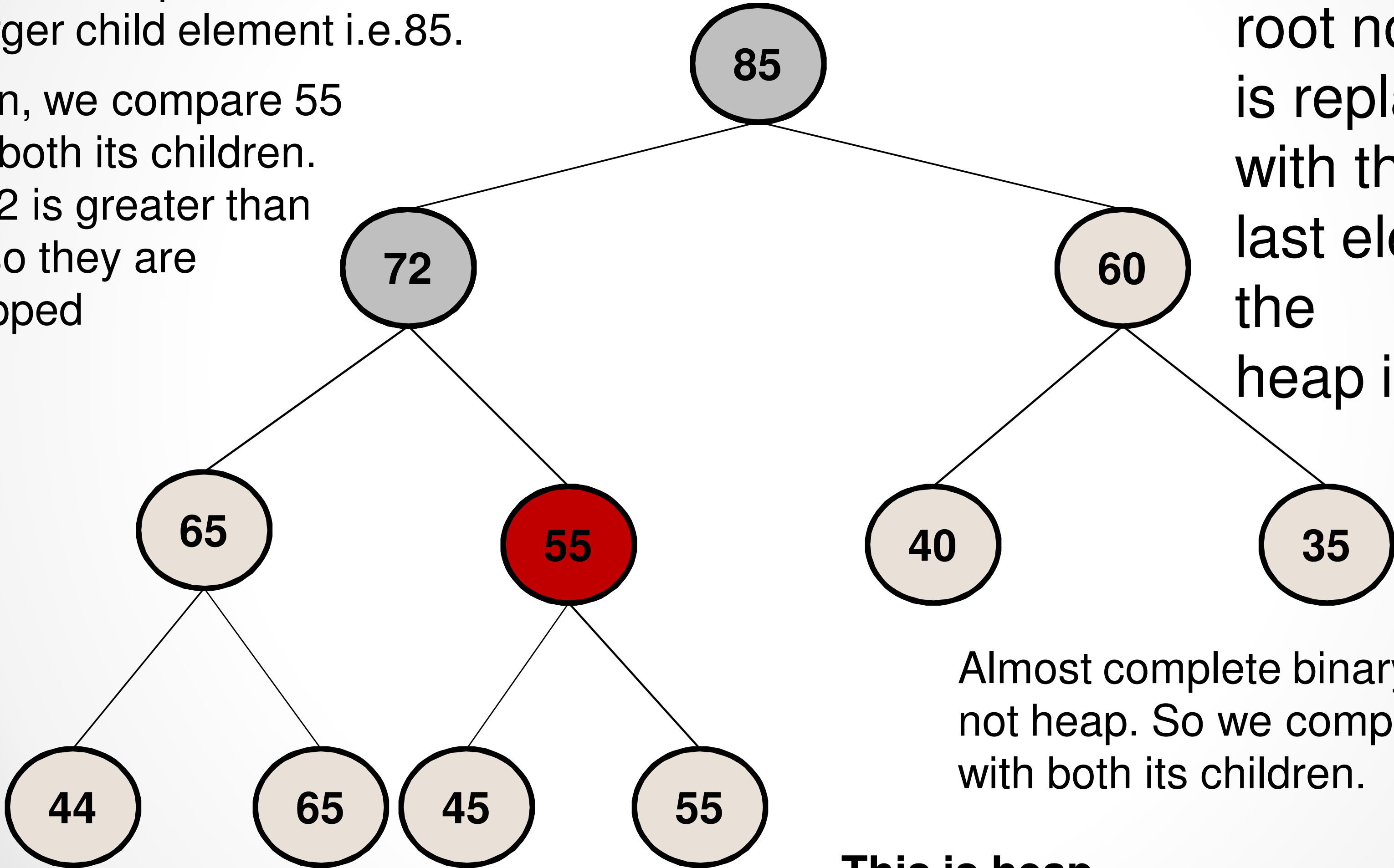
- We move the root element of the tree down after comparing and exchanging it with its child element such that **H** is finally a heap. The rule is if any of the child element is greater than the root element then we exchange it with the larger child element.

Consider a heap given below:

Only 92can be deleted.
For this the root node is replaced with the last element of the heap i.e. 55

As 55 is smaller than 85, it will be replaced with the larger child element i.e.85.

Again, we compare 55 with both its children. As 72 is greater than 55, so they are swapped



Almost complete binary tree but not heap. So we compare 55 with both its children.

This is heap.

ALGORITHM

- **DeleteItem(H,n)**
- **Step 1:** Set **Pos=1**
- **Step 2:** Set ***Item*=H[Pos]**
- **Step 3:** Set ***Temp* =H[n] And n=n-1**
- **Step 4:** Set **Left=2*Pos And Right=2*Pos+1**
- **Step 5:** Repeat steps 6 to 8 While **Right<=n**
- **Step 6:** If ***Temp*>= H[Left] *And* *Temp*<=H[Right]**

  Set **H[Pos]=*Temp***

  return

  [End if]

Continued

- **Step 7:** If **H[Left]>=H[Right]** then
  Set **H[Pos]=H[Left] And Pos=Left**
  Else
  Set **H[Pos]=H[Right] And pos=right**
  [End if]
- **Step 8:** Set **Left=2*Pos And Right = 2*Pos+1**
  [End While]
- **Step 9:** If **Left=n and *Temp* <H[Left] then**
  Set **H[Pos]=H[Left] and Pos=left**
  [End If]
- ***Step 10:*** Set **H[pos]=*Temp* and return *Item***

Heap Sort

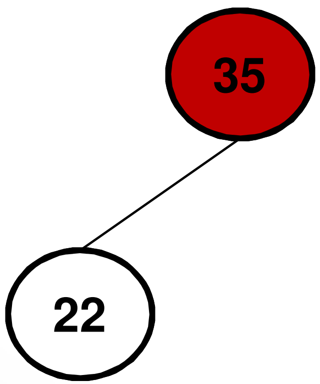
- Firstly the given unsorted list is converted into a max heap, because of the order property of the max heap that root node always contains the largest element of the list.
- The heap sort extracts the elements from the heap one at a time by deleting the root of the heap.
- The process continues until no more elements are left in the heap.
- The deleted elements are placed at the appropriate place in the array, which will be sorted at the end of the heap sort.

Consider an unsorted array A of size 8 shown below:

22	**35**	**17**	**8**	**13**	**44**	**5**	**28**
A[1]	A[2]	A[3]	A[4]	A[5]	A[6]	A[7]	A[8]

Step1:

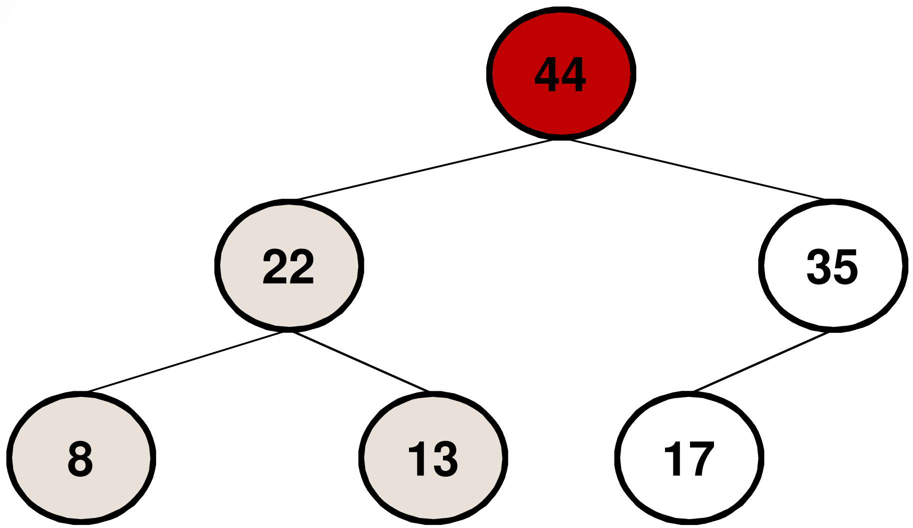
Step2:



Compare the new element with its parent node and interchange the positions if required. do the comparisons until the root is reached or the parent is greater

Step3:

Compare 44 with its parent. As 44 is greater than 17 so they are swapped.

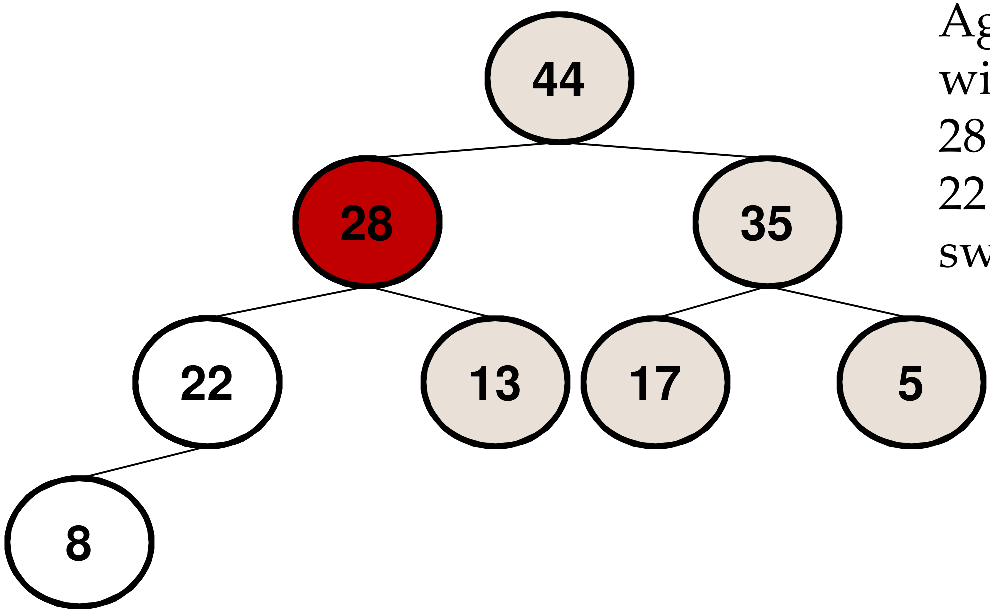


Again Compare 44 with its parent. 44 is greater than 35 so they are swapped.

Step4:

Compare 28 with its parent. As 28 is greater than 8 so they are swapped.

Again Compare 28 with its parent. As 28 is greater than 22 so they are swapped.



Max heap

The sorted array is:

44	28	35	22	13	17	5	8
A[1]	A[2]	A[3]	A[4]	A[5]	A[6]	A[7]	A[8]

Heap

Step1:

35	28	17	22	13	8	5	44
A[1]	A[2]	A[3]	A[4]	A[5]	A[6]	A[7]	A[8]

Heap

Sorted list

Delete largest element 44 & place in the last position.
Re-heap the remaining elements.

Step2:

28	22	17	5	13	8	35	44
A[1]	A[2]	A[3]	A[4]	A[5]	A[6]	A[7]	A[8]

|← Heap →|← Sorted list →|

Delete element 35 & place in the second last position. Re-heap the remaining elements.

Step3:

22	13	17	5	8	28	35	44
A[1]	A[2]	A[3]	A[4]	A[5]	A[6]	A[7]	A[8]

|← Heap →|← Sorted list →|

Delete element 28 & place in the 3rd last position. Re-heap the remaining elements

Step4:

A[1]	A[2]	A[3]	A[4]	A[5]	A[6]	A[7]	A[8]
17	13	8	5	22	28	35	44

Heap

Sorted list

Delete element 28 & place in the sorted list. Re-heap the Remaining elements . Similarly for rest of the elements.

Step5:

A[1]	A[2]	A[3]	A[4]	A[5]	A[6]	A[7]	A[8]
13	5	8	17	22	28	35	44

Heap

Sorted list

Step6:

8	5	13	17	22	28	35	44
A[1]	A[2]	A[3]	A[4]	A[5]	A[6]	A[7]	A[8]

Heap

Sorted list

Step7:

5	8	13	17	22	28	35	44
A[1]	A[2]	A[3]	A[4]	A[5]	A[6]	A[7]	A[8]

Sorted list

ALGORITHM

- The first four steps of the algorithm will convert the unsorted array **A** into a heap and steps 5 through 8 will sort the repeatedly deleting the root of the heap.
- **Step 1**: Repeat Steps 2 to 4 for **j=1 to n-1**
- **Step 2**: Set **Pos =j+1 And Temp=H[pos]**
- **Step 3:** Repeat While **H[Pos/2]<=Temp And Pos/2**

  Set **H[Pos]=H[Pos/2]**

  Set **Pos= Pos/2**

  [End loop]

CONTINUED

- **Step 4:** Set **H[Pos]=Temp**

  [End Loop]
- **Step 5:** Repeat Steps 6 and 7 For **k=n to 2**
- **Step 6:** Set **Item = deleteItem(H,k)**
- **Step 7 :**Set **H[k]=Item**

  [End loop]
- **Step 8:** Exit

CONCLUSION

- The primary advantage of the heap sort is its efficiency. The execution time efficiency of the heap sort is **O(n log n).** The memory efficiency of the heap sort, unlike the other n log n sorts, is constant, O(1), because the heap sort algorithm is not recursive.
- The heap sort algorithm has two major steps. The first major step involves transforming the complete tree into a heap. The second major step is to perform the actual sort by extracting the largest element from the root and transforming the remaining tree into a heap.