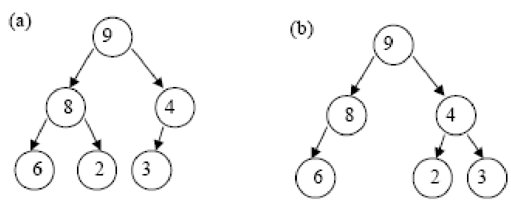
Heap tree:

- A binary tree is a heap tree if it is an almost complete binary tree and has following properties:
  - it is empty *or*
  - the key in the root is larger than or equal to either child and both subtrees have the heap property .(max-heap)

  (Heap property (max-heap): Key in the root is larger than or equal to either child)

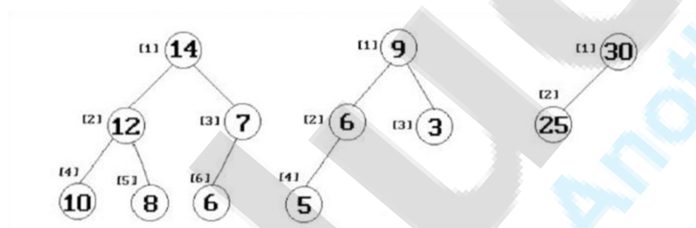


(a) is a heap.

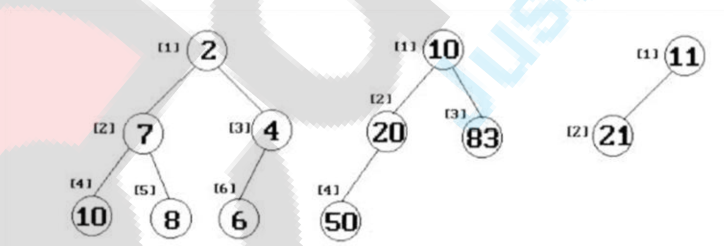
(b) is not a heap as it is not complete

(c)is complete but does not satisfy the second property defined for heaps.

Two kinds of binary heaps:



Max-heaps

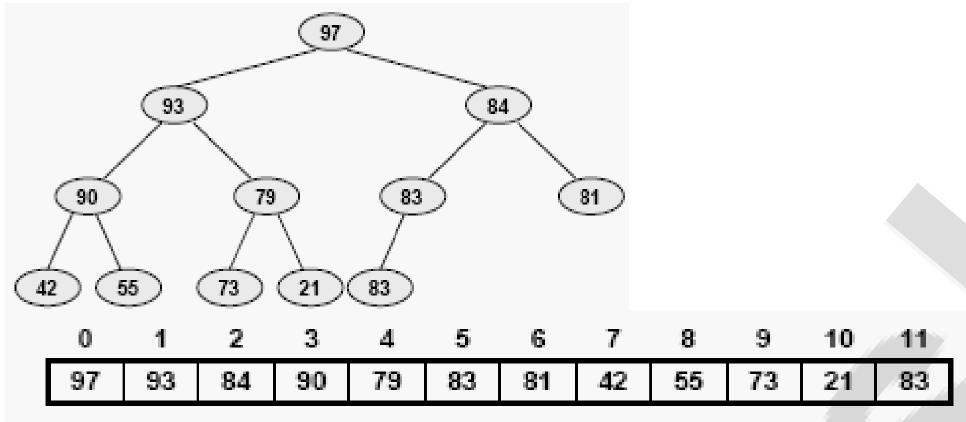


Min-heaps

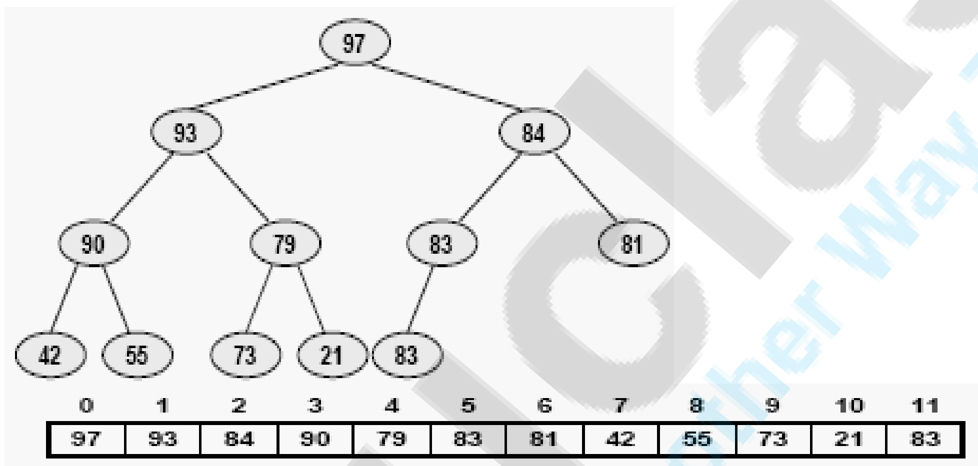
Array Implementation of Heap:

If a node is stored at index k , and elements are stored from index 0 then

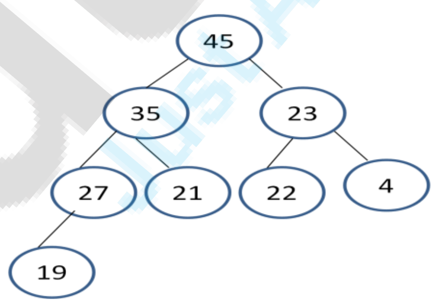
- Left child at index 2k+1
- Right child at index 2k+2



If child is at i position, parent will be at (i – 1)/2



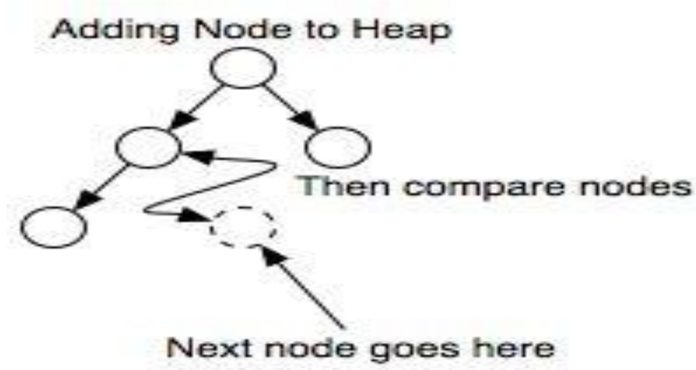
- Q. Draw array implementation of following heap tree.



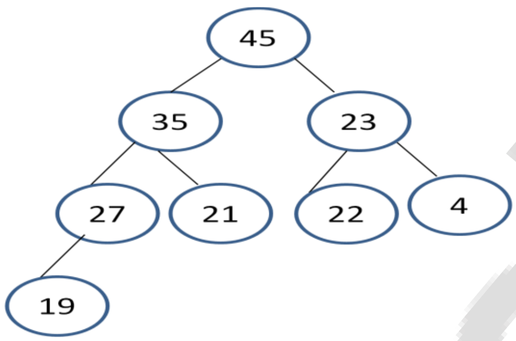
Creation of heap tree:

Inserting node into heap tree:

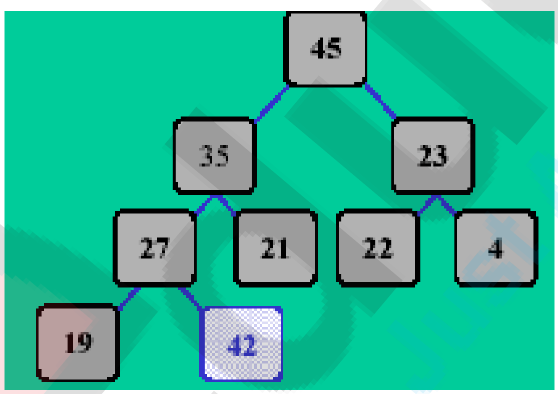
- The insertion algorithm consists of two steps
  - Insert the new node in end (the new last node)
  - Restore the heap-order property (Reheap Up/ Upheap)



- For eg. Insert 42 in following heap tree.

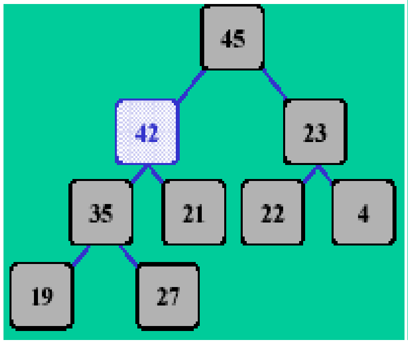


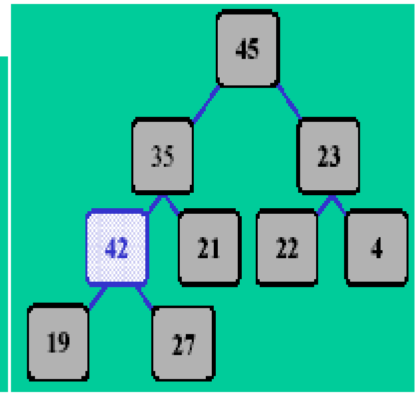
- Put the new node in the next available slot.



Reheap Up :

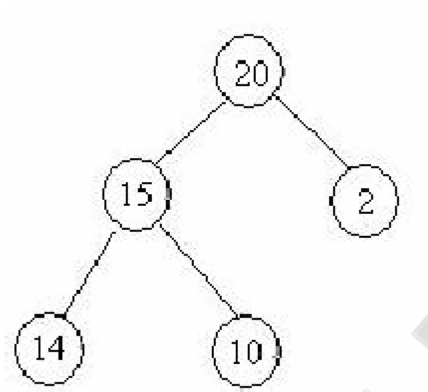
- Push the new node upward, swapping with its parent until the new node reaches an acceptable location. i.e. one of the following conditions must be satisfied.
  - The parent has a key that is >= new node, or
  - The node reaches the root

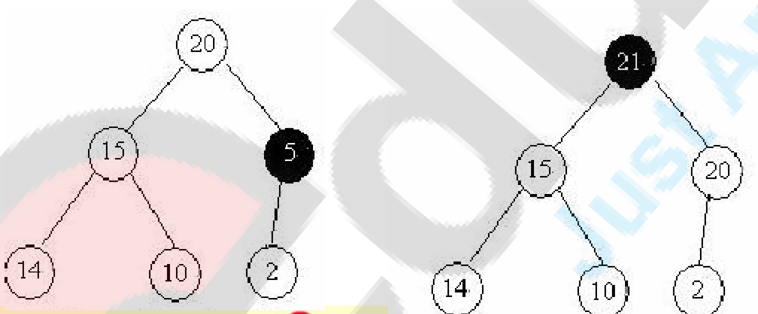


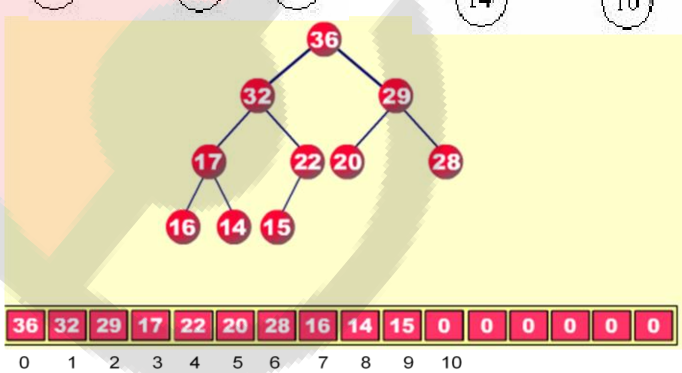


Ques1. Insert 5 into heap

Ques2. Insert 21 into heap

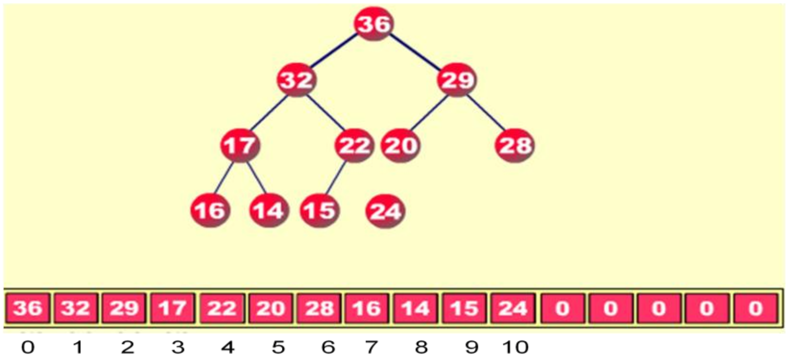






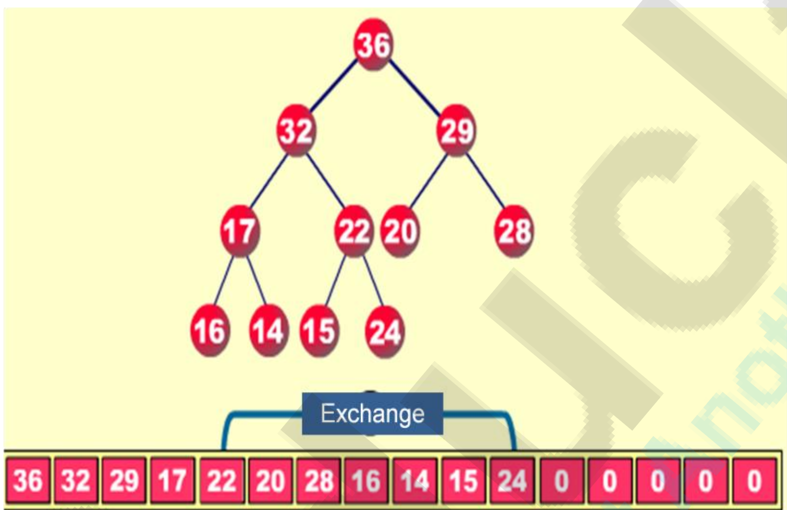
last = 9

Insert 24 at last position



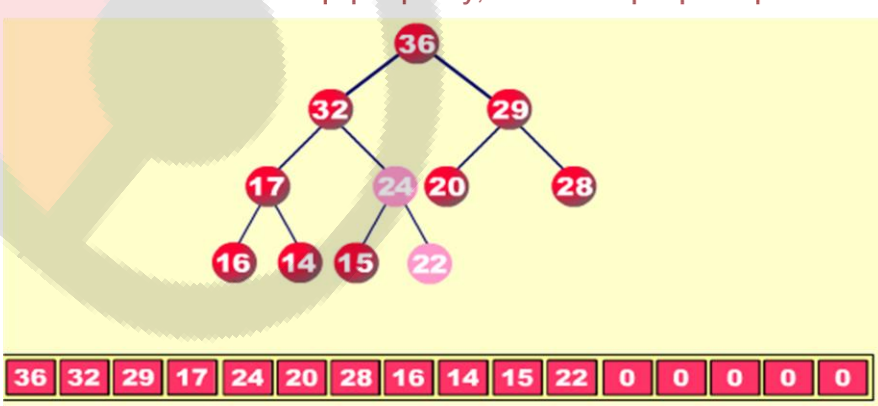
last = last + 1
heap[last] = data
reheapUp (heap, last)

Reheap Up



parent = (newNode – 1)/2
if(heap[newNode] > heap[parent])
swap(newNode, parent)
reheapUp(heap, parent)

Now satisfies heap property, so reheap up stops here



Insertion: ALGORITHM:

Algorithm InsertHeap (heap <array of data type>, last<index>, data<datatype>)

Pre: heap is an array of data working as heap, last is index of last element in heap, data is data to be inserted in heap

Return : returns true if data inserted, false otherwise

1. if (heap full)
    1. return false
2. last = last+1
3. heap[last] = data
4. reheapUp (heap, last)
5. return true

Reheap Up:

Algorithm reheapUp (heap <array of data type>,newNode<index>)

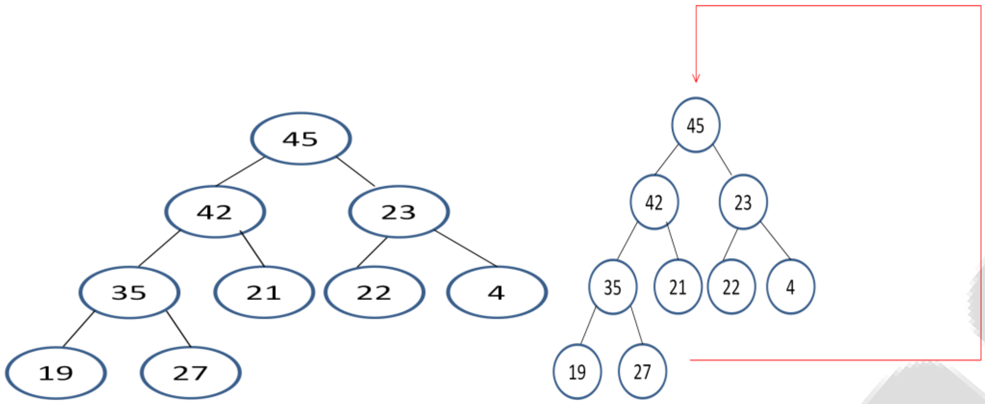
Pre: heap is an array of data working as heap, newNode is index of new element inserted in heap

Return : new node placed at proper position

1. if (newNode not zero)
    1. parent = (newNode-1)/2
    2. if (heap[newNode] > heap[parent])
        1. swap(newNode, parent)   //exchange elements at newNode and parent index
        2. reheapUp (heap, parent)
2. return

Deletion of node from heap tree :

- **Perform deletion operation on following heap tree.**

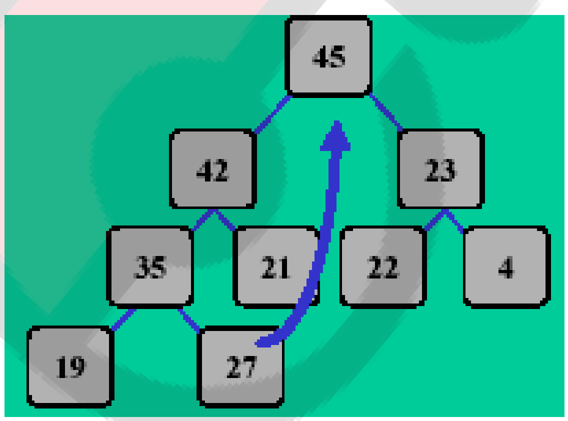


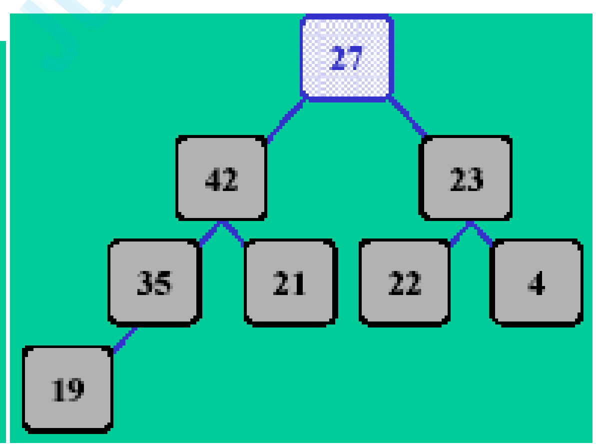
Deletion:

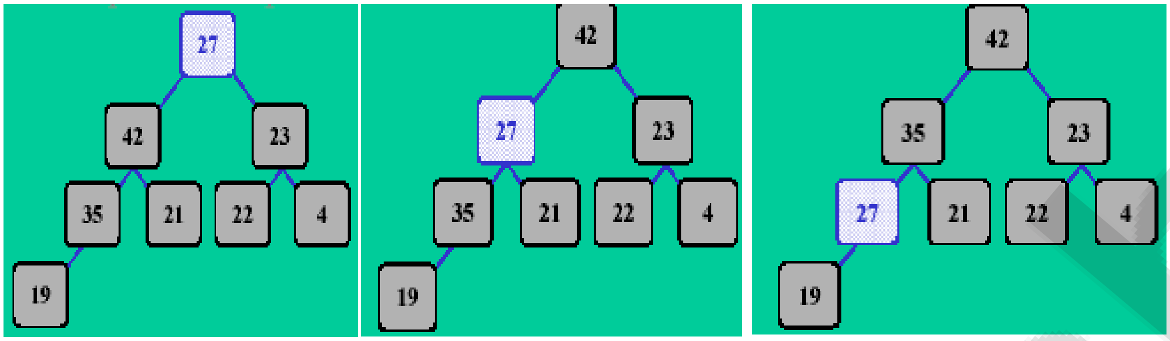
- **The deletion algorithm consists of three steps**
  - Remove root and replace it with the key of the last node (say x)
  - Remove x
  - Restore the heap-order property (Reheap Down/ Down Heap).

Reheap Down:

- Push the out-of place node downward, swapping with its larger child until the new node reaches an acceptable location, i.e.
  - For children, all have keys <= the out-of-place node
  - The node reaches the leaf.
- **Deletion - Removing the Top of a Heap : Move the last node onto the root.**

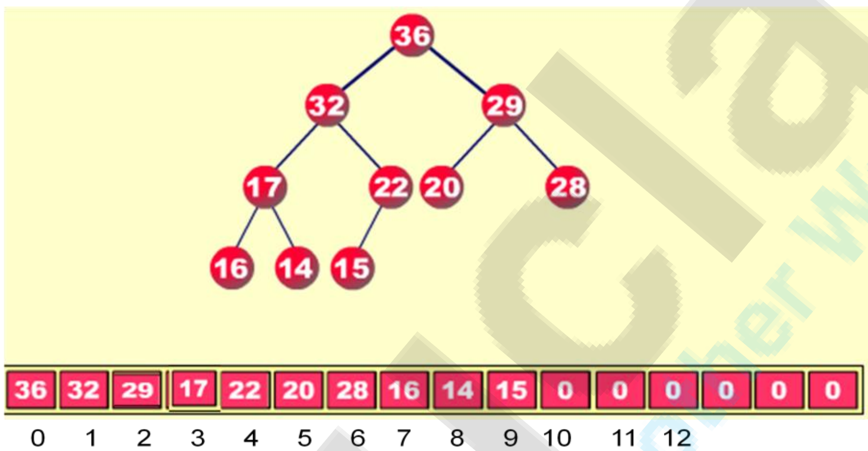






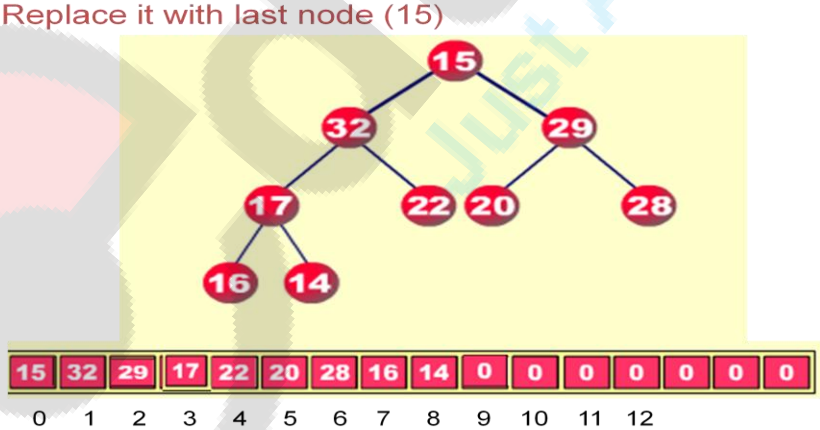
Deletion: heap tree

Remove (36) root



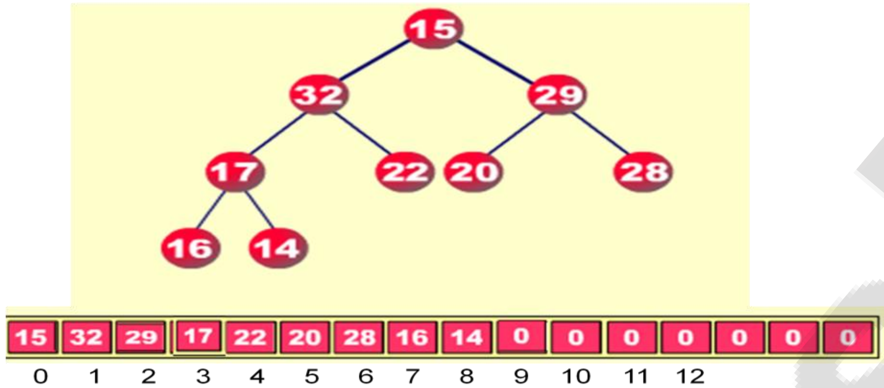
data = heap[0]

Replace it with last node (15)



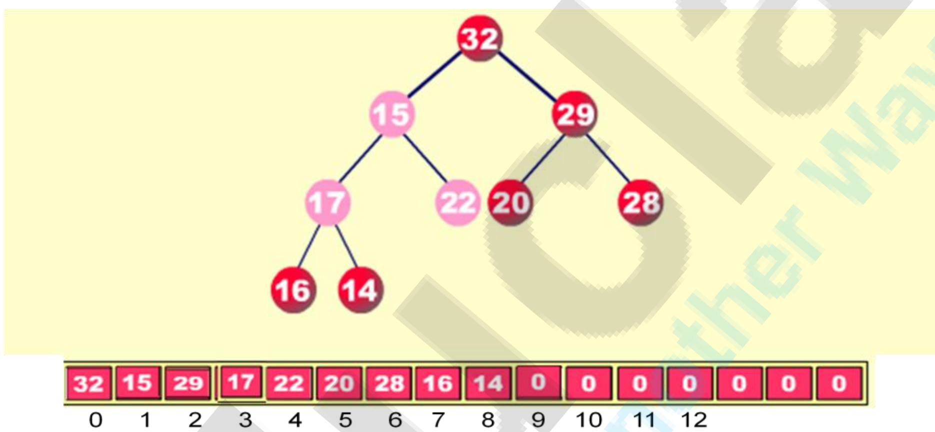
heap[0] = heap[last]
last = last -1

Reheap Down



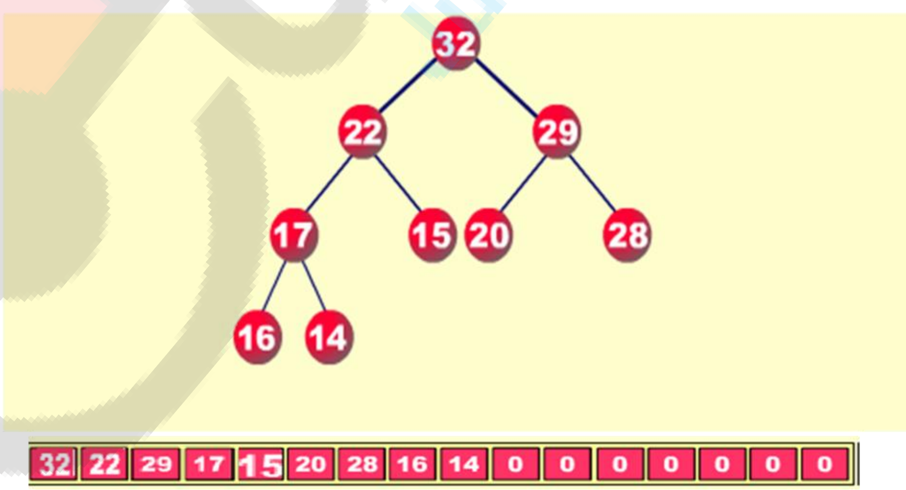
Compare both children of 15, that are 32 & 29. Exchange 15 with larger child that is 32

reheapDown(heap, 0, last)



Compare both children of 15 - 17 & 22. Exchange 15 with 22 as it is larger child

Now satisfies heap property, so reheap down stops here.



Deletion:

Algorithm DeleteHeap (heap <array of data type>, last<index>, data<datatype>)

Pre: heap - array of data working as heap, last - index of last element in heap, data - data deleted from heap is stored in it

Return : returns true if data deleted, false otherwise

1. if (heap empty)
    1. return false
2. data = heap[0]
3. heap[0] = heap[last]
4. last = last -1
5. reheapDown (heap, 0, last)
6. return true

Reheap Down:

Algorithm reheapDown (heap <array of datatype>, root <index>, last<index>)

Pre: heap is an array of data working as heap, root of heap or subheap**, last** is the index of last element in existing   heaptree

Return : heap restored

1. if (root * 2 + 1 <= last)   // check if left child exists   //(i.e. atleast one child exists)
    1. leftkey = heap[root * 2 + 1]
    2. if (root * 2 + 2 <= last)   // check if right child exists
        1. rightkey = heap[root * 2 + 2]
    3. else
        1. rightkey = lowkey   // low key:- some very low value ex. -1
    1. if (leftkey > rightkey)   // Find which is larger child
        1. largechildkey = leftkey
        2. largechildindex = root * 2 + 1
    2. else
        1. largechildkey = rightkey

2. largechildindex = root * 2 + 2

3. if (heap[root] < heap[largechildindex])

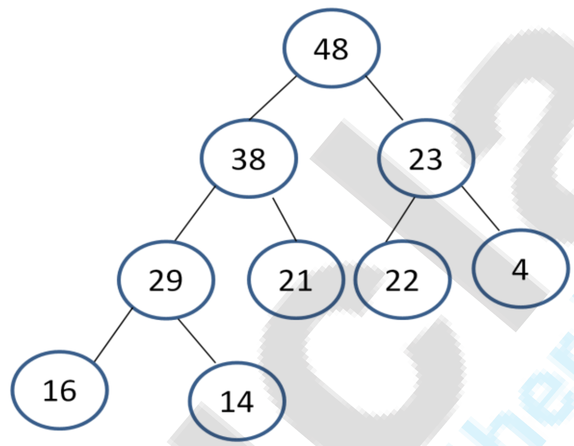
// if parent < child, exchange parent and child

1. swap(root, largechildindex)

2. reheapDown (heap, largechildindex, last)

2. return //no child

Question : Deletion of heap tree : Perform two deletion operations on following heap tree.



Heap sort:

- Build heap tree using data in given array. (Buildheap) i.e. insert element and perform reheap up operation.
- Continuously delete topmost element and perform reheap down operation.
- Then the resultant array will be sorted array.

Algorithm heapSort (heap <array of datatype>, last<index>)

Pre: heap is an array of data working as heap, last is index of last element in array

Return : array gets sorted

1. index = 1
    1. while (index <= last)
        1. reheapUp (heap, index)
        2. index = index + 1
2. lastdata = last
3. while (lastdata > 0)
    1. exchange (heap, 0, lastdata)
    2. lastdata = lastdata - 1
    3. reheapDown(heap, 0, lastdata)
4. return