Tree

CSC-114 Data Structure and Algorithms

Slides credit: Ms. Saba Anwar, CUI Lahore

Outline

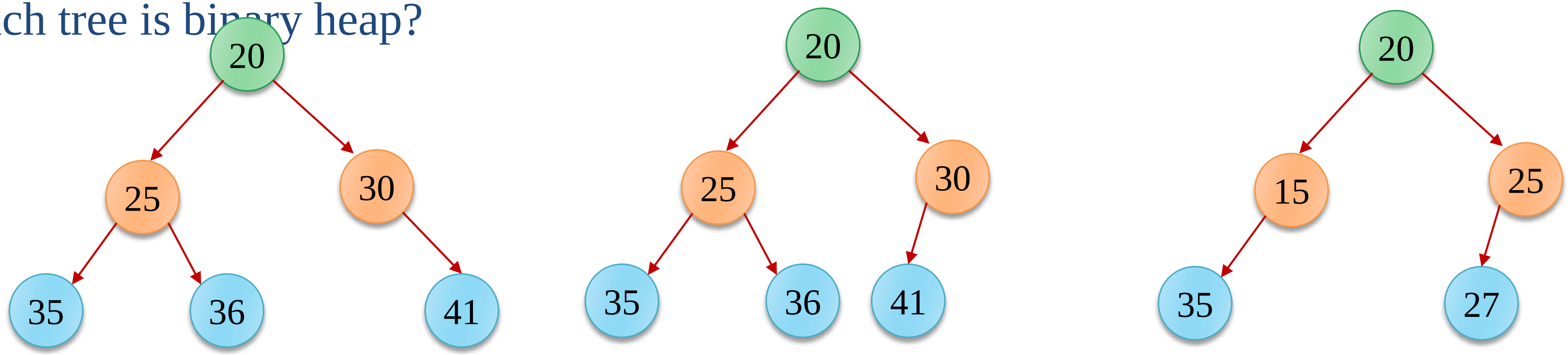
- Binary Tree Variations
  - Binary Heap Tree
    - Max Heap
    - Min Heap
  - Insertion
  - Deletion

Binary Heap

A binary tree which holds two properties:

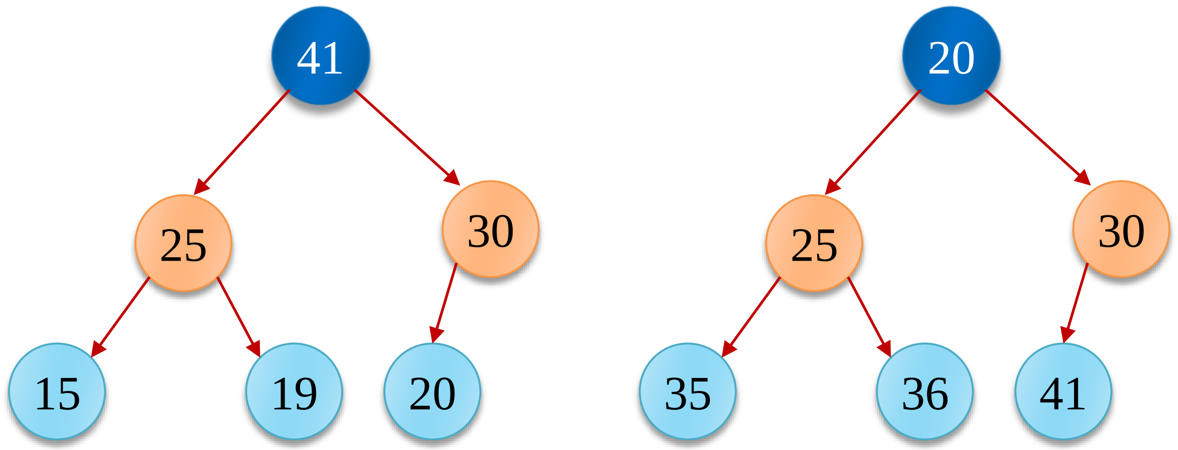
- Heap Order Property:
  - Min-Heap Property: Every node is smaller than or equal to each of its children
  - Max-Heap Property: Every node is larger than or equal to each of its children
- Shape Property:
  - Tree is **complete**.
    - A tree that is full at all levels except last level, and nodes are filled from left to right

Which tree is binary heap?



Finding Min

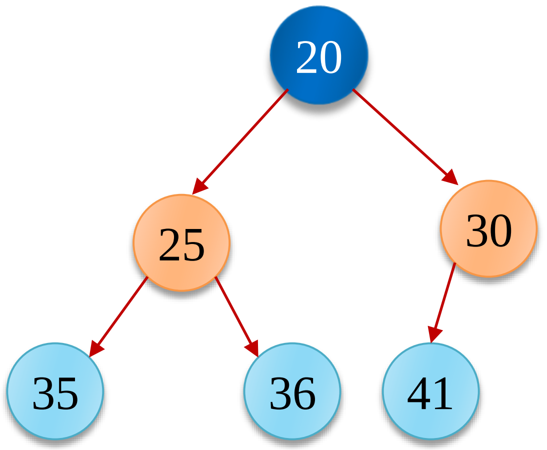
- Finding minimum or maximum?
  - It will always be root node with max or min value depending upon it is min heap or max heap.
  - So constant time required
- What about removal?



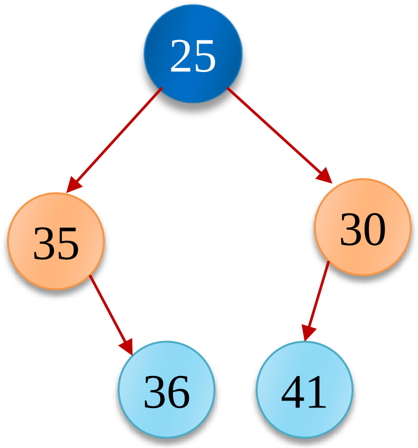
Deletion

Deleting minimum?

- Root node will be removed
- Which node will be next root node?
  - It must be next minimum that is 25
  - Then which node will come at place of 25?
    - Again, the minimum in sub tree of 25



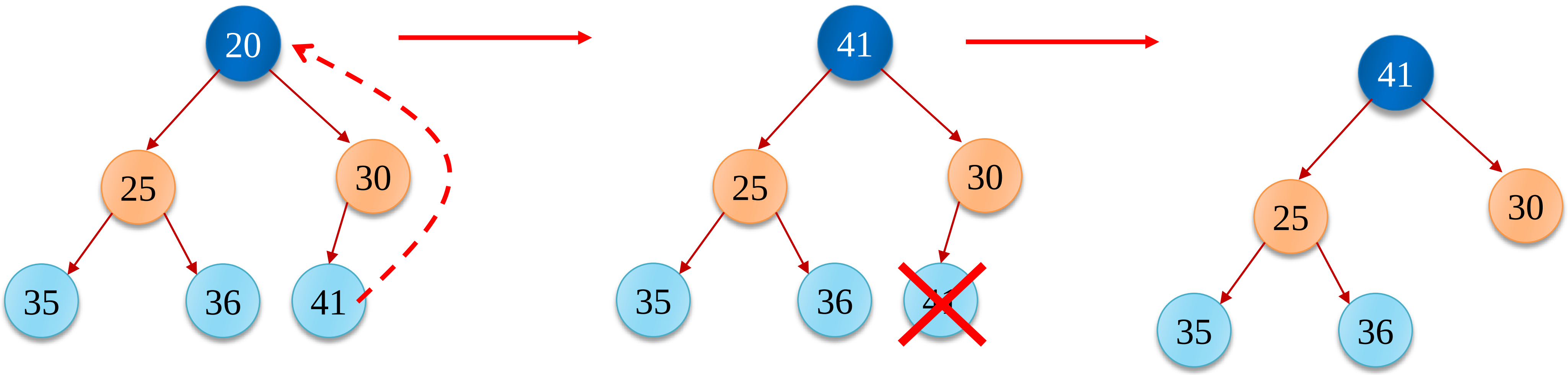
- Is this heap tree any more?



Deletion

Deleting minimum?

What if we replace root node with **last** inserted node?

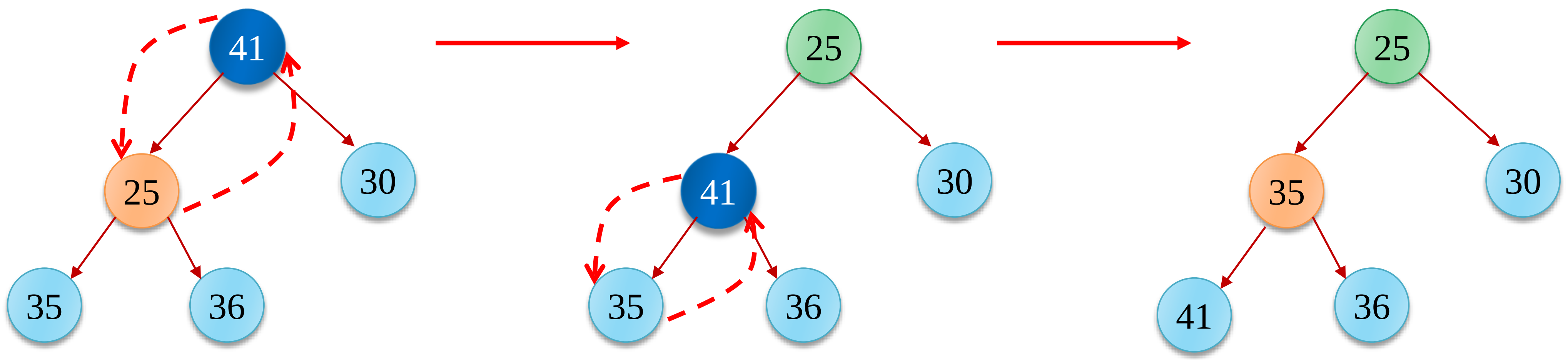


Now the completeness property is reserved

But another problem is created, what is that?

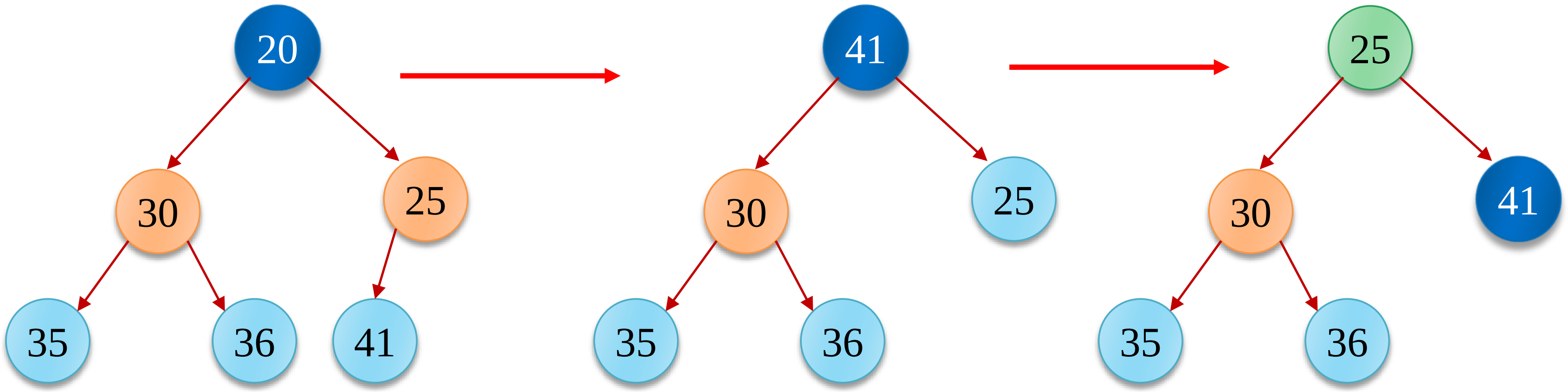
Deletion

- We need to perform another process to maintain heap order
  - After replacing root node, check if it is greater than its children, swap with appropriate child.
    - Repeat this process till leaf or parent node becomes smaller than its children
    - This process is called **Heapify-Down or Down Heap**



Heapify-Down

Example-2



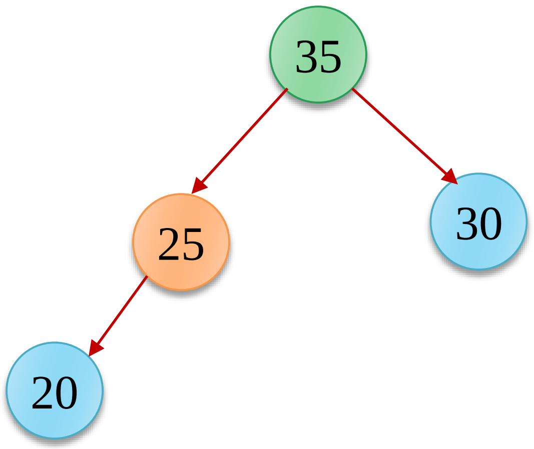
Deletion involves following steps:

1. Replace root node with last inserted node, to maintain shape
2. Heapify-Down process, to maintain heap order

Insertion

Insertion of a new node

- Always insert from left to right to maintain shape property of left completeness
  - Let say starting from root node how to decide that we should go left or right?
  - Always remember where you inserted last node

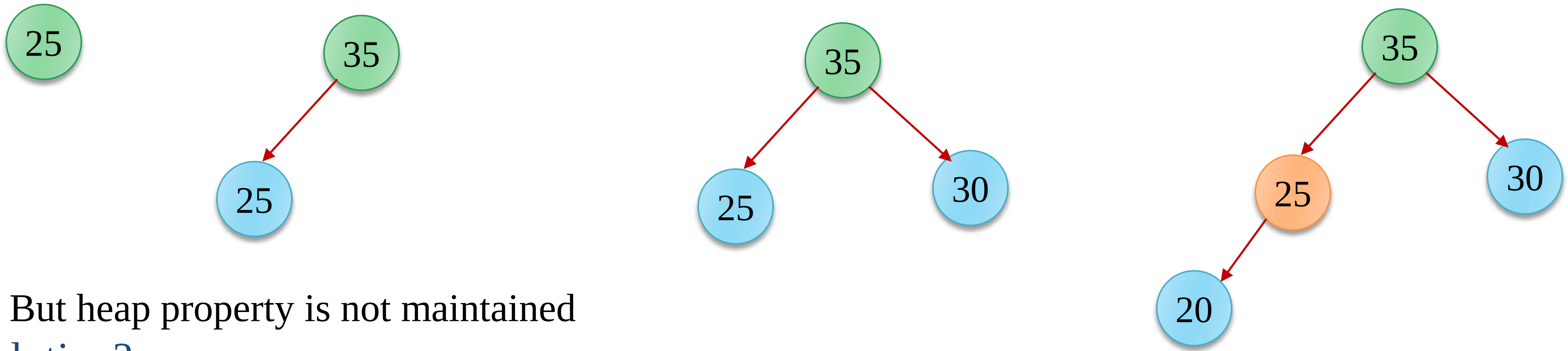


  - But heap property is not maintained
- Solution?
  - Repeatedly check if parent is larger than node, then swap the node with parent
  - **Process is called Heapify-Up or Up Heap**

Insertion

Insertion of a new node

- Always insert from left to right to maintain shape property of left completeness
  - How?
  - Always remember where you inserted last node



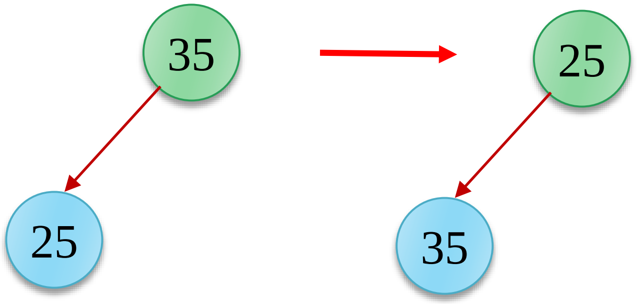
  - But heap property is not maintained
- Solution?
  - Repeatedly check if parent is larger than node, then swap the node with parent
  - **Process is called Heapify-Up or Up Heap**

Heapify-Up

Insert 35

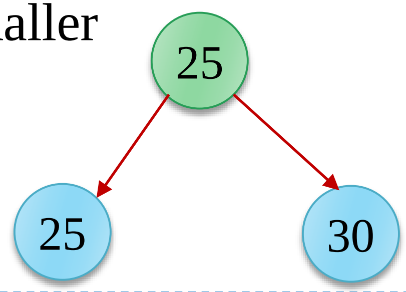
Insert 25

Swap with parent if parent is larger

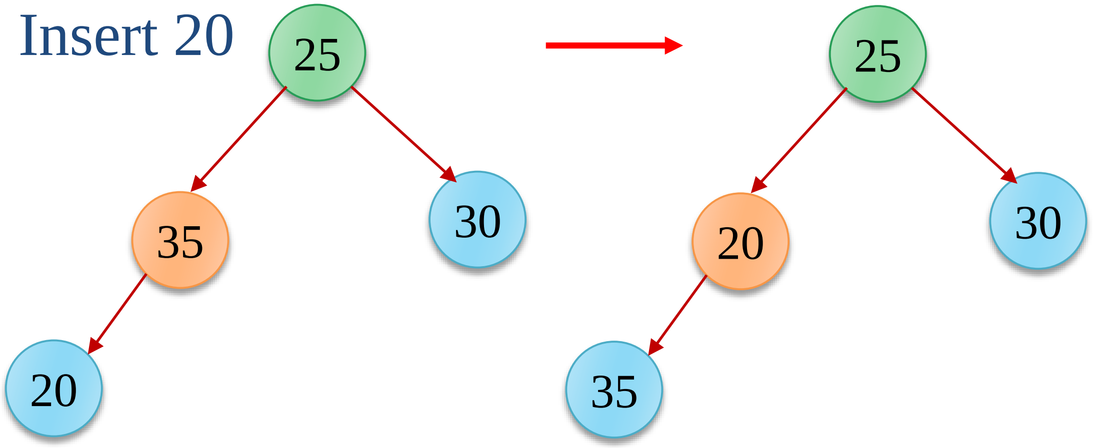


Insert 30

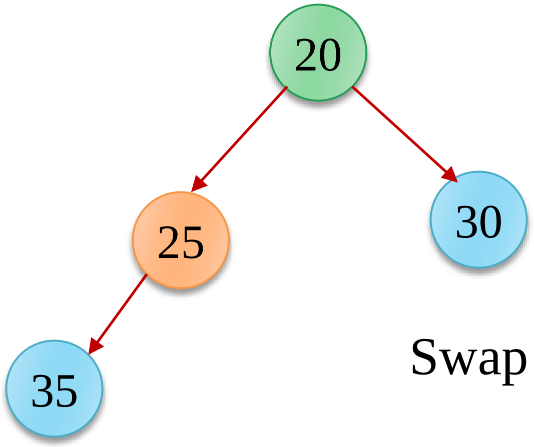
Parent is already smaller



Insert 20



Swap with 35

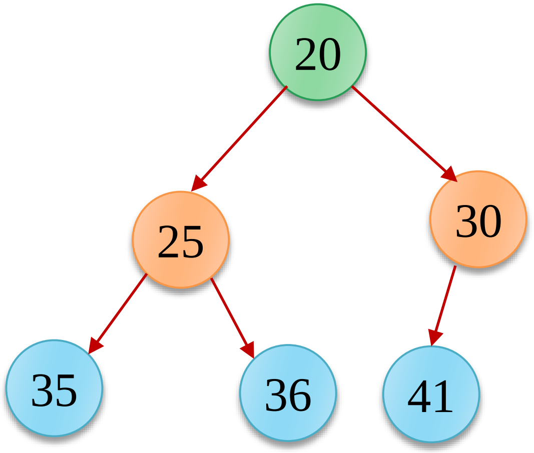


Swap with 25

Insertion

Insert 36

Insert 41



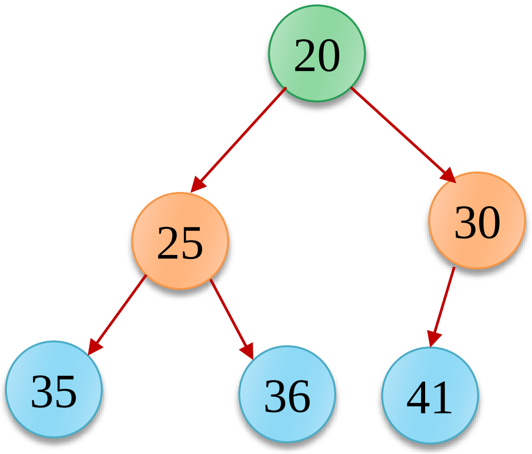
Insertion involves two steps:

1. Inserting node at correct position using last node, to maintain left completeness
2. Heapify-Up process, to maintain heap order

Implementation

Using Linked Memory Allocation

- Maintain two nodes:
  - Root
  - Last node



Using Array

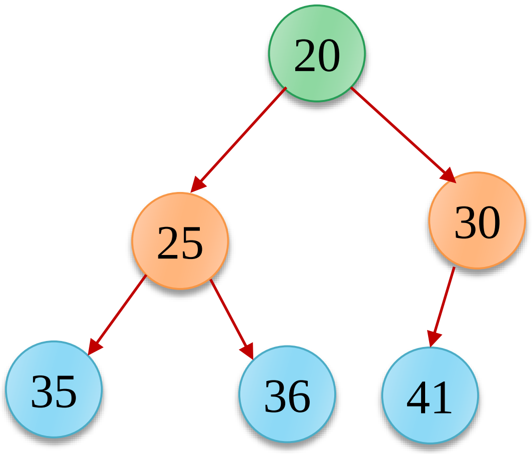
- Children of node at location k
  - Left -> 2K+1
  - Right -> 2K+2
- Parent of a node located at k

  (k-1)/2 (consider integer division)

0	**1**	**2**	**3**	**4**	**5**	**6**
20	25	30	35	36	41	null

Using Array

- Root is always 1st index
- Last node's index is = size-1
- No need of functions for left, right, parent
  - Just do calculation
- Deletion is always replacing root with last
  - Then Heapify-Down
- Insertion is always at end of current nodes
  - And then Heapify-Up

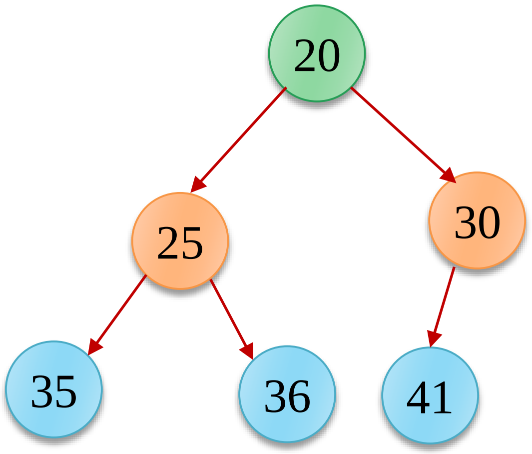


0	1	2	3	4	5	6
20	25	30	35	36	41	null

Using Linked Memory Allocation

Need to maintain last node?

- In Deletion
  1. Root is replaced with Last node
  2. Last node is updated
  3. And then Heapify-Down
- In Insertion
  1. Last node used to find correct location for new node
  2. Node is inserted
  3. Last node is updated
  4. And then Heapify-Up

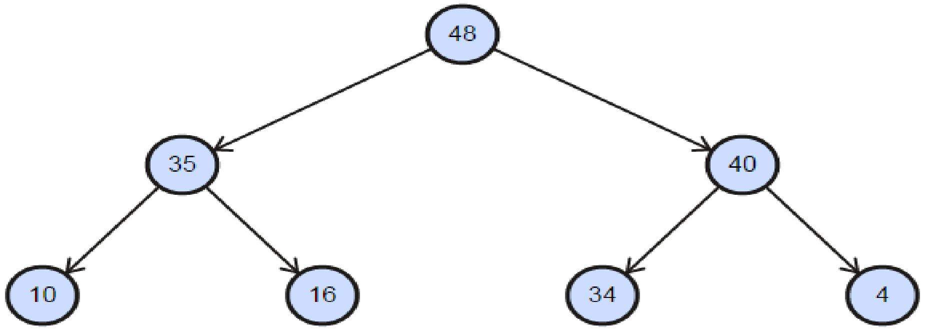


0	1	2	3	4	5	6
20	25	30	35	36	41	null

Example

See the figure, it’s a max heap

Here last node is 4

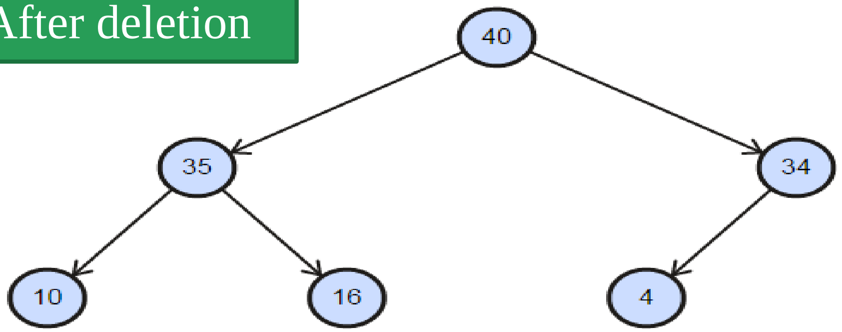


0	1	2	3	4	5	6
48	35	40	10	16	34	4

If we delete 48

- Which node will become last node now?
  - In array?
  - In Linked allocation?

After deletion



0	1	2	3	4	5	6
40	35	34	10	16	4	null

Example

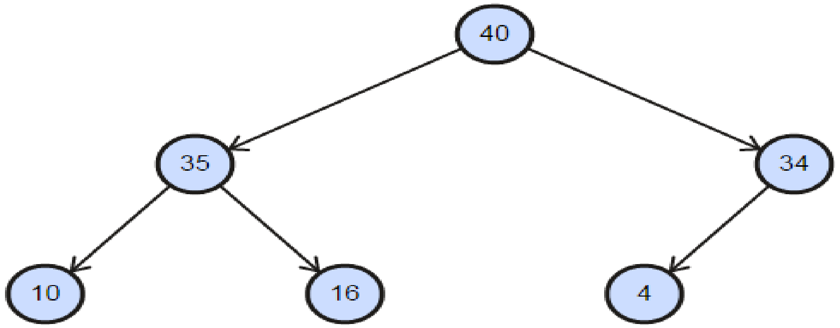
Last node is 4

Insert 50

- Which node will become last node now?
  - In array?
  - In Linked allocation?

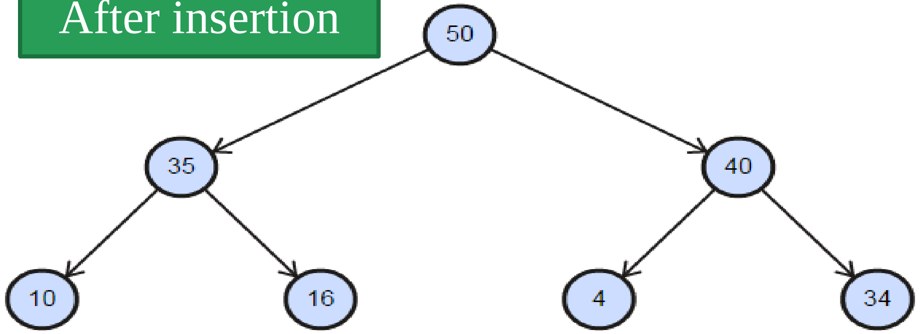
Perform few more insertions and deletions

- It will give you good understanding of last node



0	1	2	3	4	5	6
40	35	34	10	16	4	null

After insertion

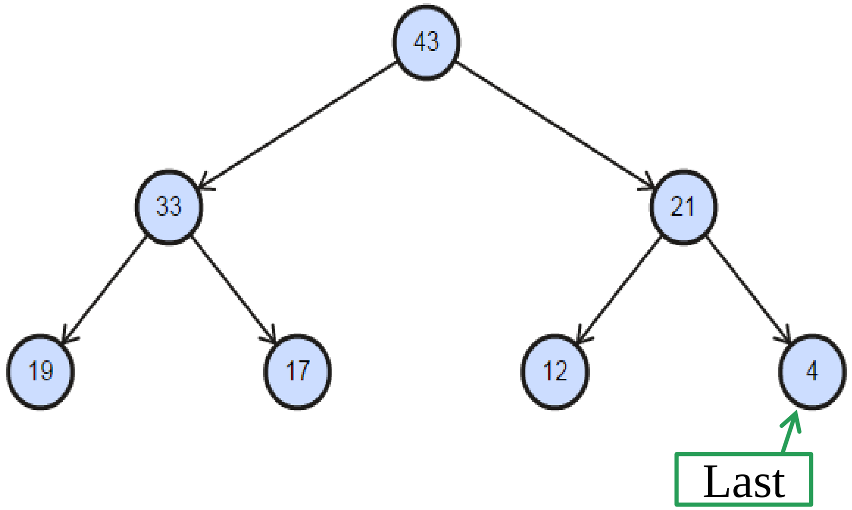


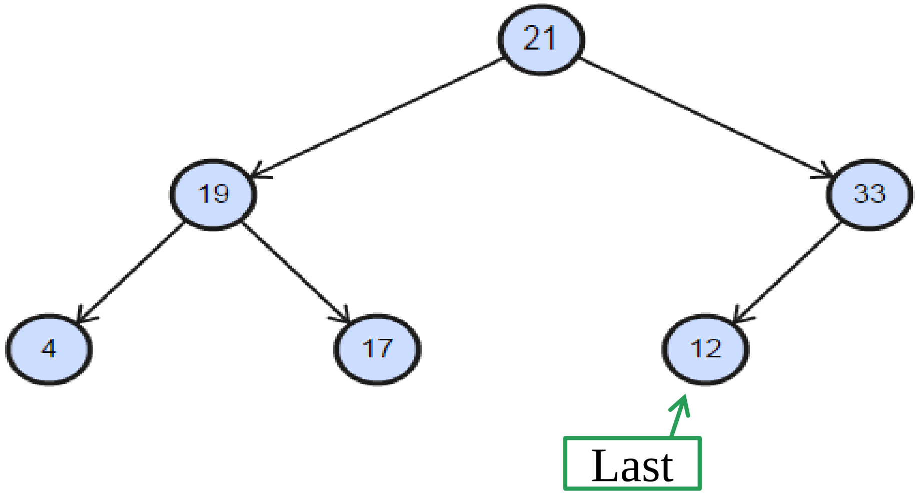
0	1	2	3	4	5	6
50	35	40	10	16	4	34

Deletion: Linked Implementation

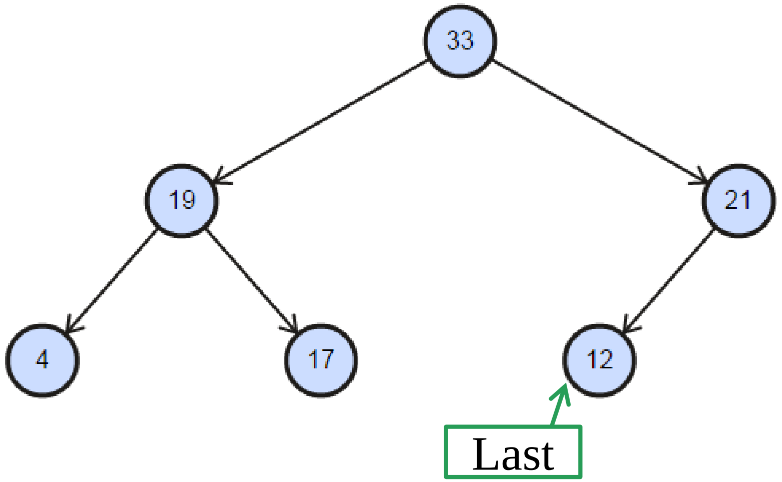
Delete

- Last is right node
- Last= sibling





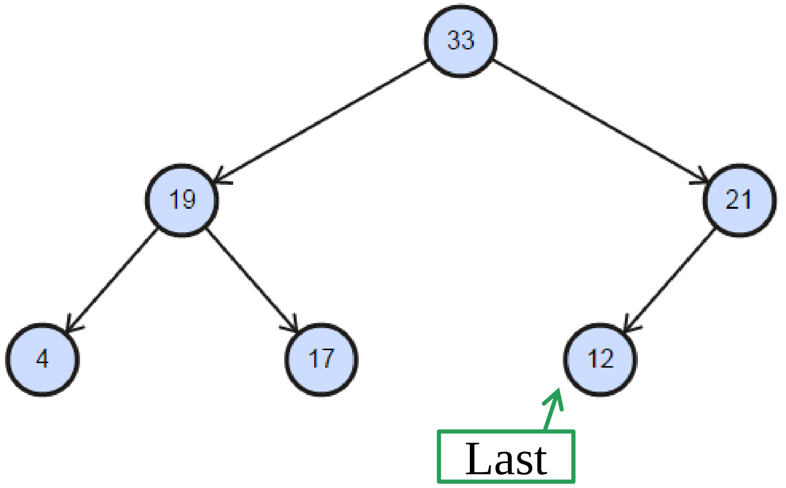
Heapify-Down

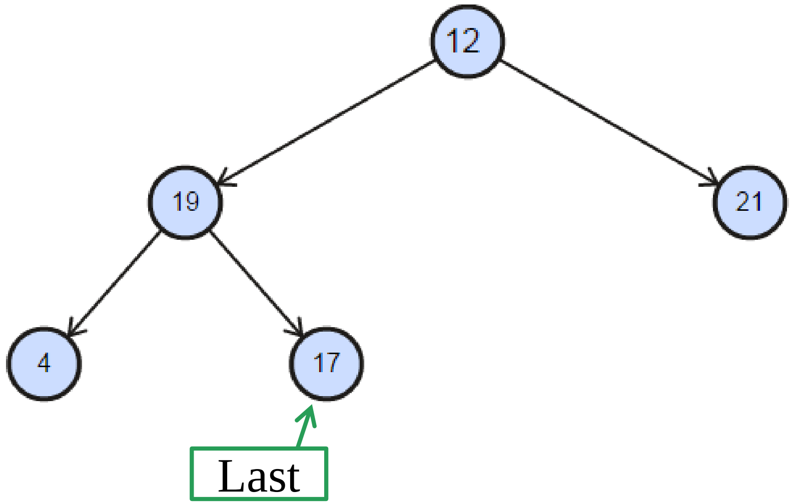


Deletion: Linked Implementation

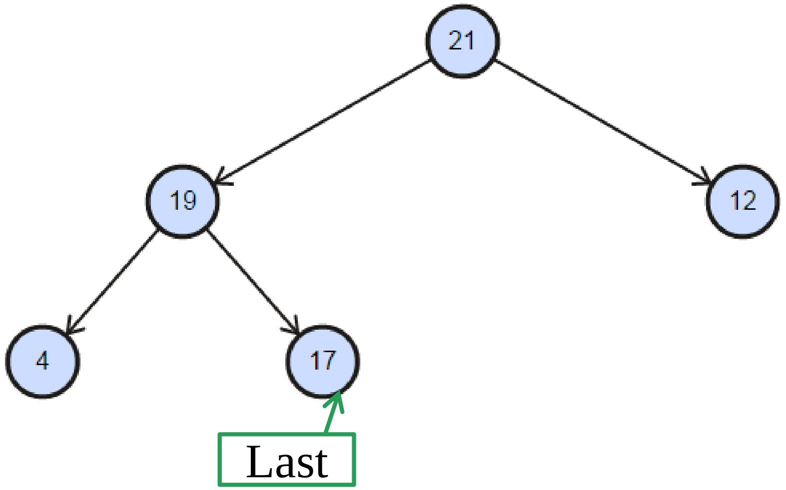
Delete

- Last is left node
- go up, until root or node is right
- 21 is right
- last= most right of sibling→ 17



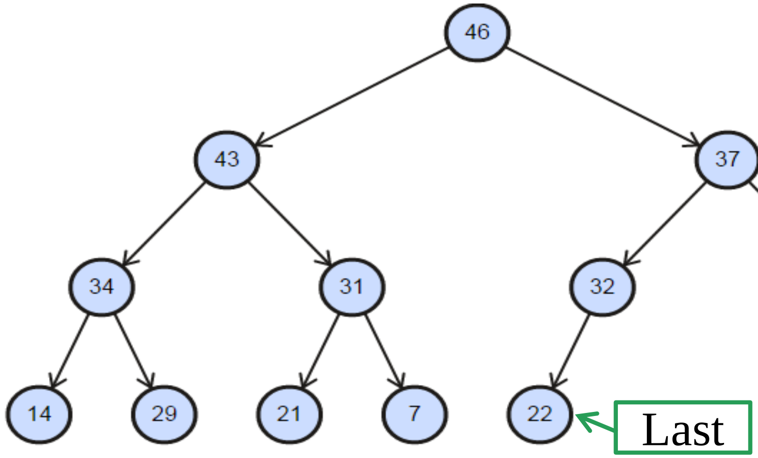


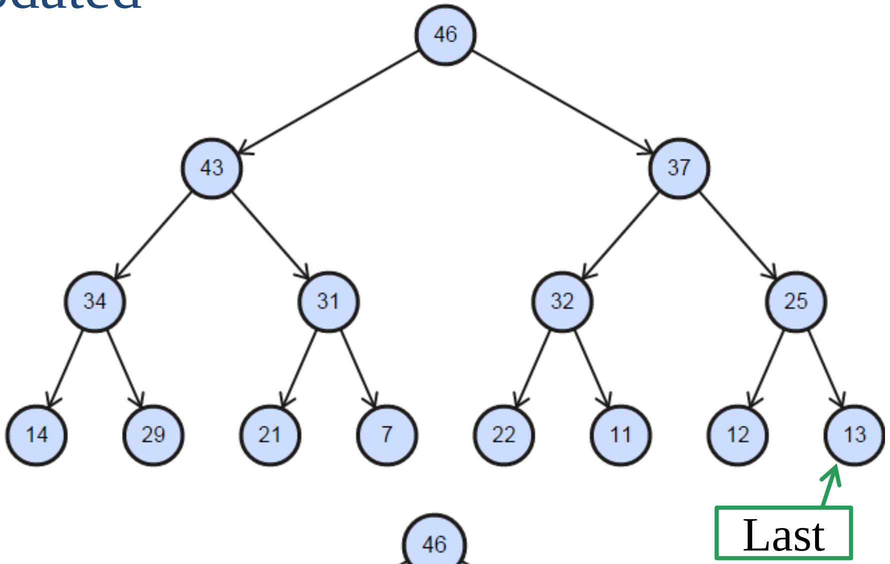
Heapify-Down

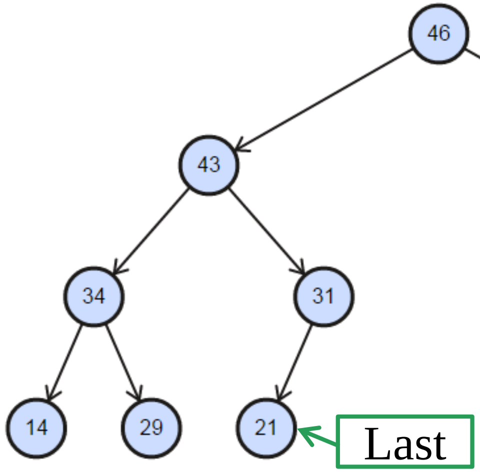


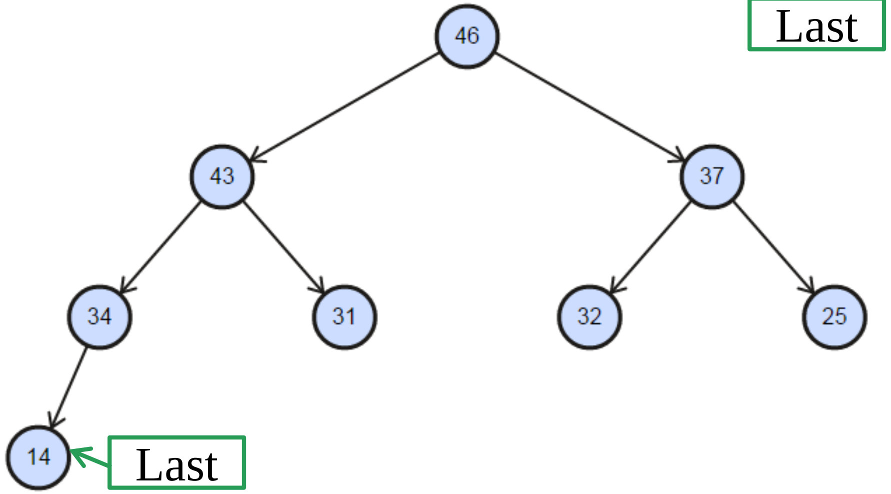
Deletion: Linked Implementation

Perform delete in each figure to see how last is updated









Deletion: Linked Implementation

1. Replace root with last node
2. Update last node
   1. Find parent of last node
   2. If last is right node

      last=parent.left

      parent.right=null
   3. Else If last is left node

      Parent.left=null

      Repeatedly move up in hierarchy to parent nodes of parent until you reach root node or a node that is right node

      If root node

      grand parent is left child

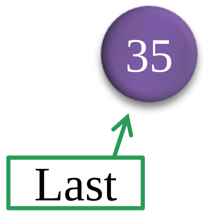
      Last= most right node of root

      Else //its right node

      Last= most right of sibling of this node
3. Perform Heapify-Down process

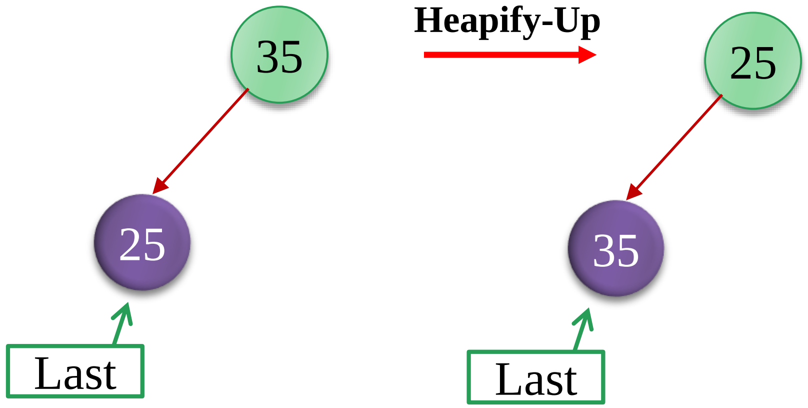
Insertion: Linked Implementation

Insert 35



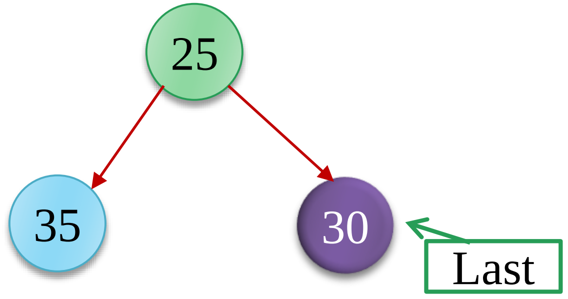
Insert 25

- Last node is itself a root node
- New node will become left node
  - Heapify-up(last)
- Now 35 is last node



Insert 30

- Last node is a left node
- New node will become right node
- No need of Heapify-Up
- Now 30 is last node

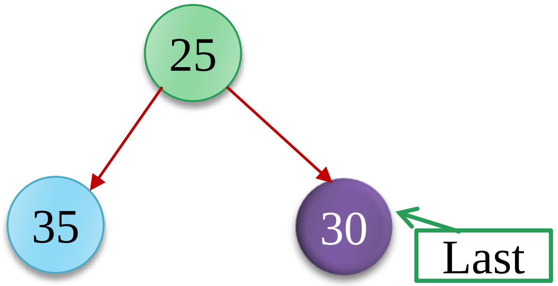


Insertion- Linked Implementation

Insert 20

Last node is right child

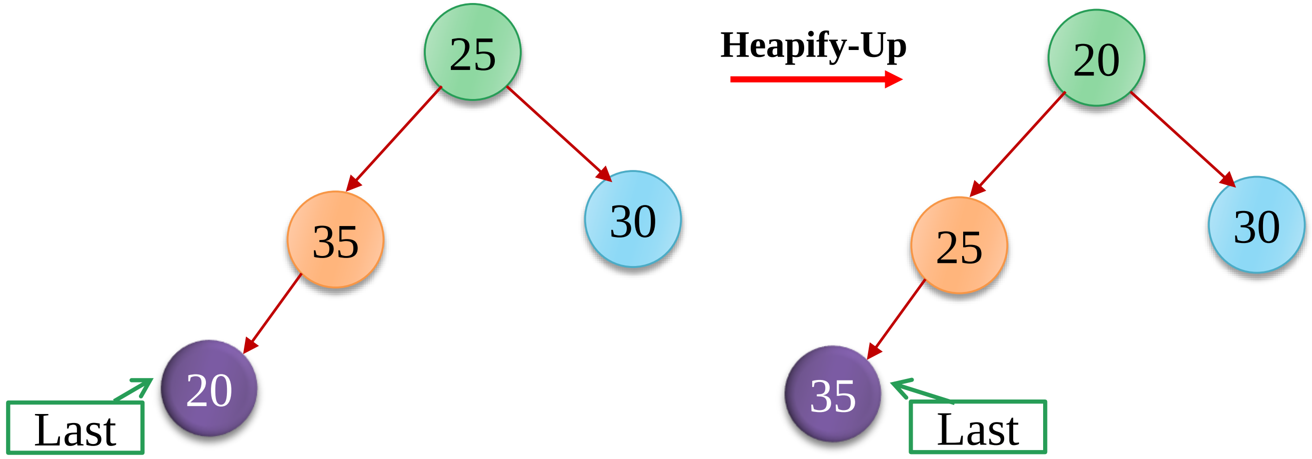
Go up until you reach a node that is either left or root node



25 is root, go to left until you reach leaf node that is 35

Insert new node as left node

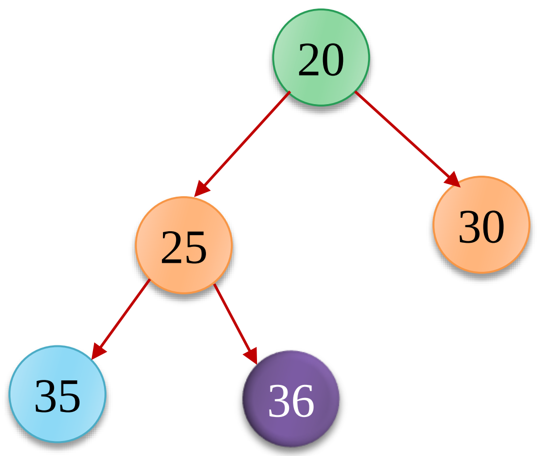
Now 35 is last node



Insertion- Linked Implementation

Insert 36

Last node is a left node
New node will become right node
No need of Heapify-Up
Now 36 is last node

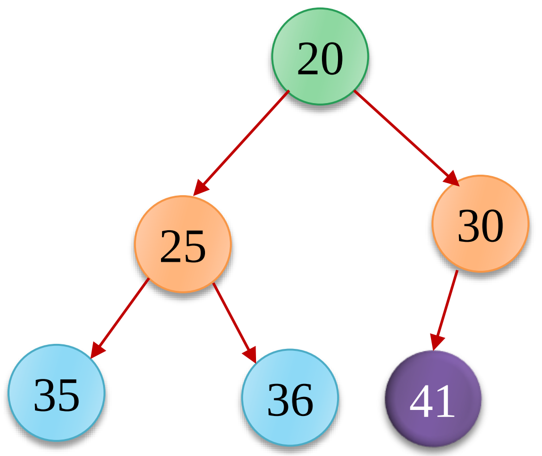


Insert 41

Last node is right child

Go up until you reach a node that is either left or root node

25 is left child, go to its sibling that is 30
Insert new node as left node
No need of Heapify-Up
Now 41 is last node



How much time it will take to find insertion location?

Insertion: Linked Implementation

1. If **last** node is root node

   new node is left child

2. If **last** node is left node

   new node is right node of parent of **last**

3. Otherwise, repeatedly go to parent nodes of **last** until you reach a node that is either root node or a left child of its parent.

   If node is root node

   then simply go to its left until you reach leaf node

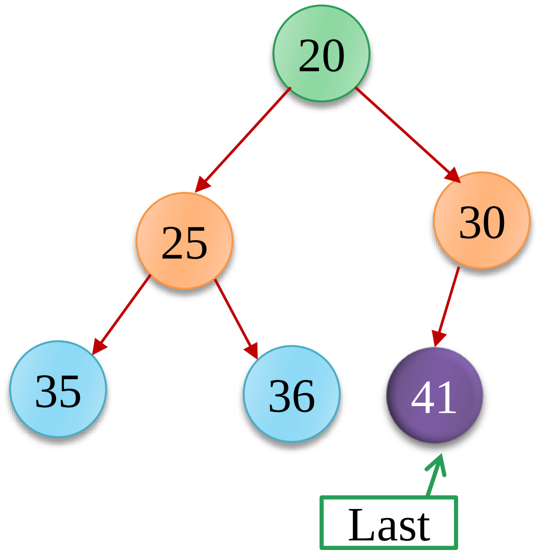
   - If node is left child

     then go to its sibling node

     new node is left child of this node // (sibling will be a leaf node)

   last=new node //update last

   Perform Heapify_Up process



Applications

Priority Queue

Heap data structure is mainly as priority queue, often they are used as synonyms

Highest priority is always on top.

Data Structure	FindMin	Add	Remove
Unsorted Array	O(n)	O(1)	O(n)
Sorted Array	O(1)	O(n)	O(1)
Unsorted List	O(n)	O(1)	O(n)
Sorted List	O(1)	O(n)	O(1)
Heap	O(1)	O(log n)	O(log n)

Time complexity of binary tree depends upon height of tree which is equivalent to logN. Where N is total number of nodes.

What can be heap tree's worst case?

Applications

Heapsort

- Heap tree can also be used to sort a list of numbers
- How?
  - Build the heap tree from given list
  - Reconstruct list by repeatedly doing the following:
    - Remove min and put in list until tree becomes empty
- Time Complexity? O(nlogn)
  - Remove min?
  - Total number of nodes?