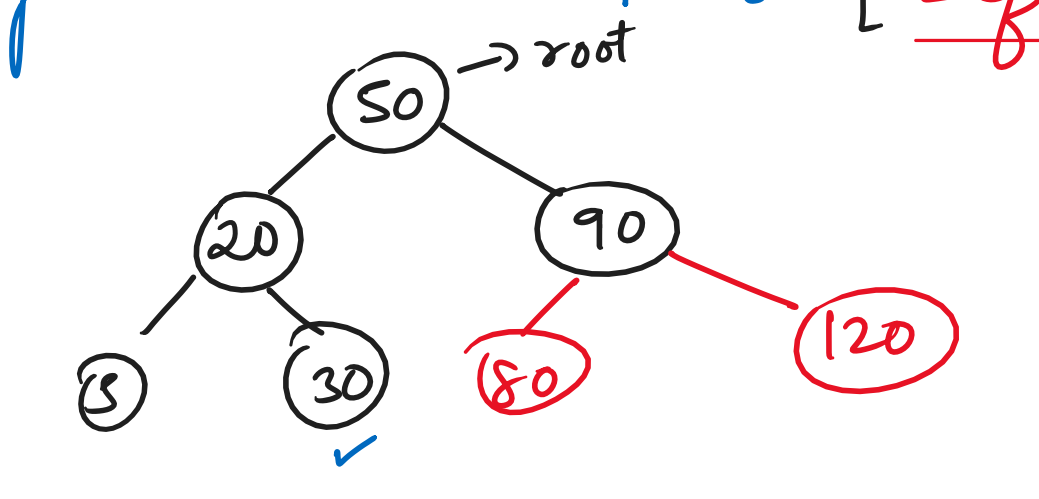
Introduction to Search Trees :→

* Search * Delete
* Insert

* (Binary Search Tree) :→ It is a non-linear data structure containing entities called nodes where each node of the tree follows a unique property.

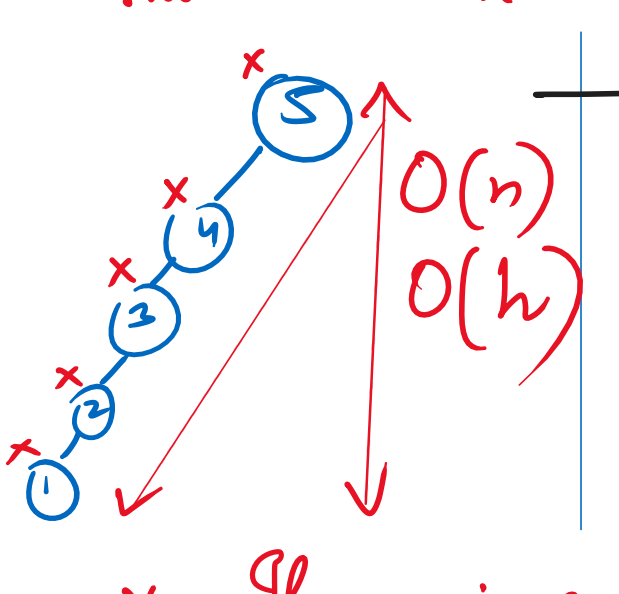
For any node in a BST : [Left < Node < Right]



search (120)
search (5)
TC Search & Insert logN

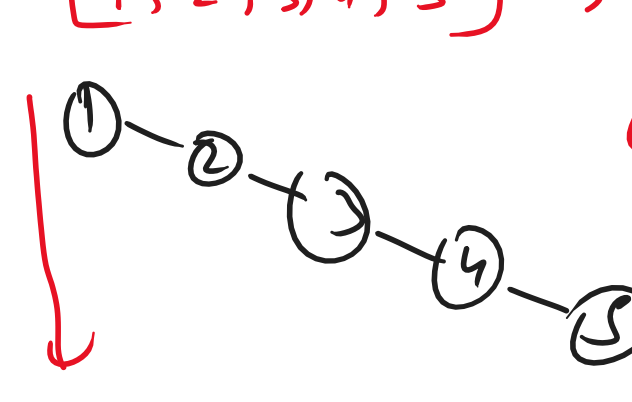
Insert data in a BST :

(1) arr = [5, 4, 3, 2, 1] → Des
(2) arr = [1, 2, 3, 4, 5] → Asc



→ Skewed Tree
RST
LST

O(n)
O(h)



O(n)
O(h)

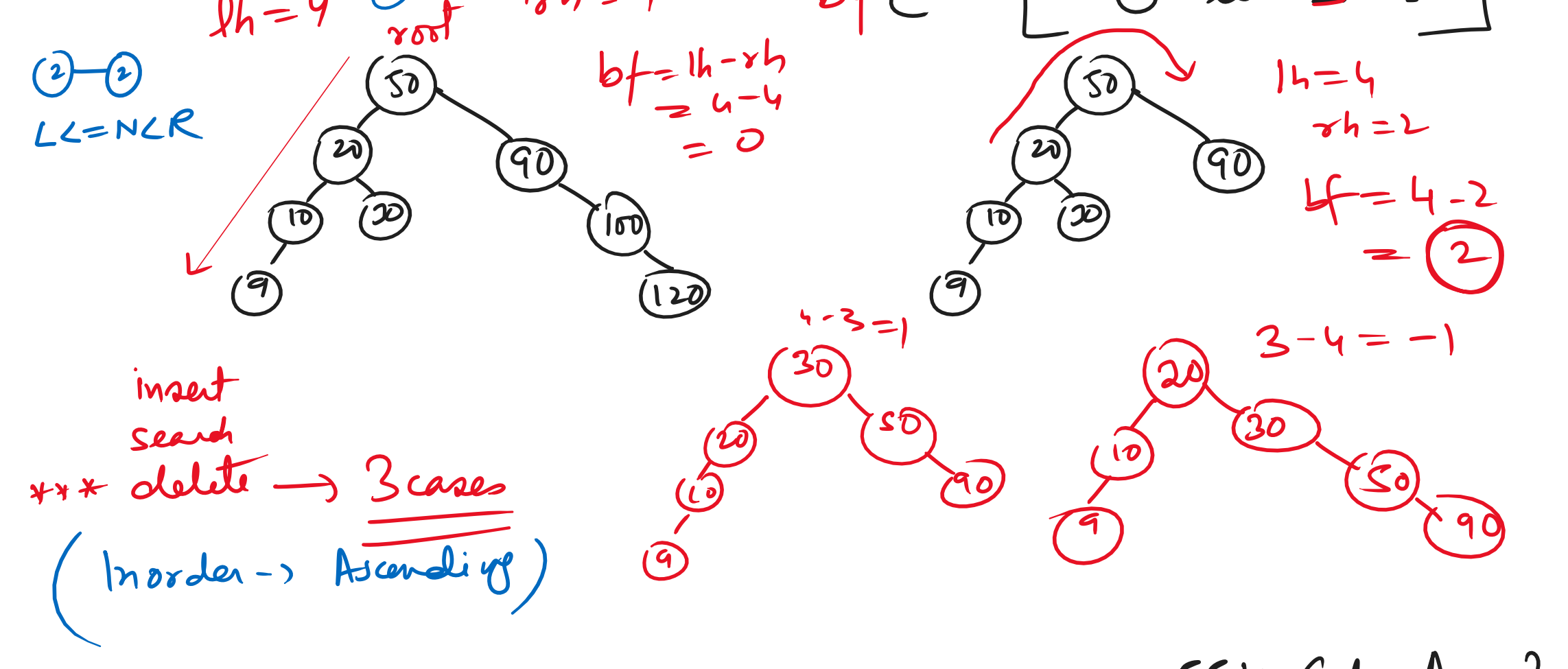
* If we insert sorted data into a BST, it becomes a skewed tree (Right or Left Skewed). The insertion & search time complexity increases from O(log n) to O(n) or O(h) where h is the height of the tree.
This is the biggest drawback of a BST.

For this purpose two new search trees were introduced:
(1) AVL Tree (2) Red Black Tree

In case of BST, the height is not balanced but in case of AVL & Red Black tree, the height is strictly balanced. We perform rotations to balance the height.

Balance Factor (bf) = left height − right height

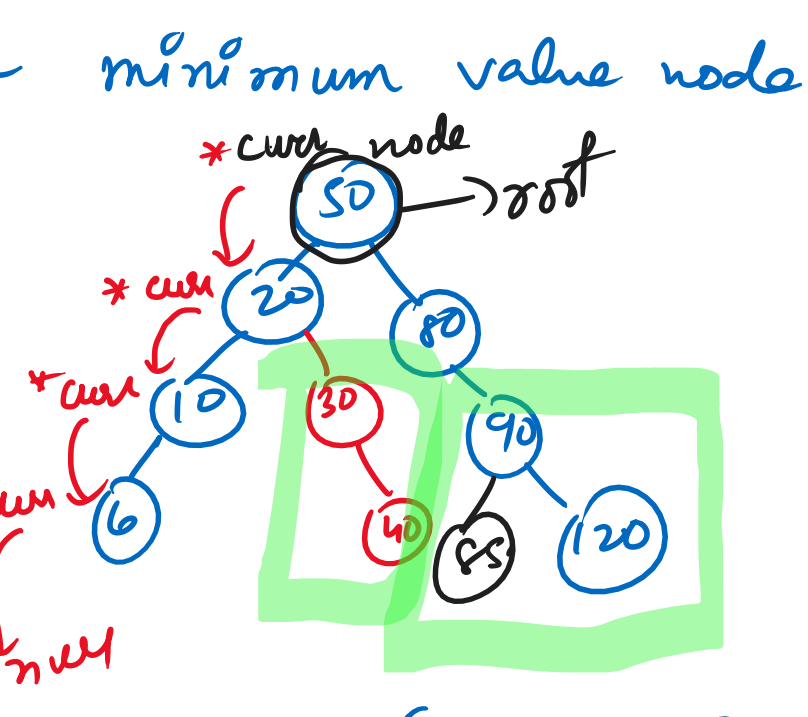
bf ∈ [0 to ± 1]



insert
search
*** delete → 3 cases
(Inorder → Ascending)

SSK Coders Arcade

* Find the minimum value node of a tree :→
(delete)
findMin(node)



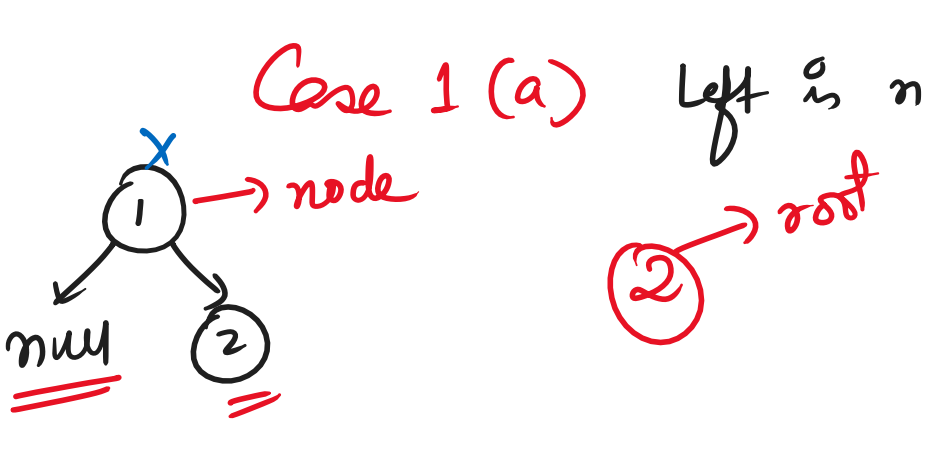
```
TreeNode curr = root;
while (curr != null &&
       curr.left != null)
{
  curr = curr.left;
}
return curr;
```

(Abdul Bari)
(Kunal Kushwaha)
Telusko
Take U Forward (Striver) (C++)
(Tech With Tim)
(Code help)
Love Babbar
(PrepInsta)

* Delete Operation in a BST (Theory + Probable Questions)

Case 1 :→ Node with a single child :→

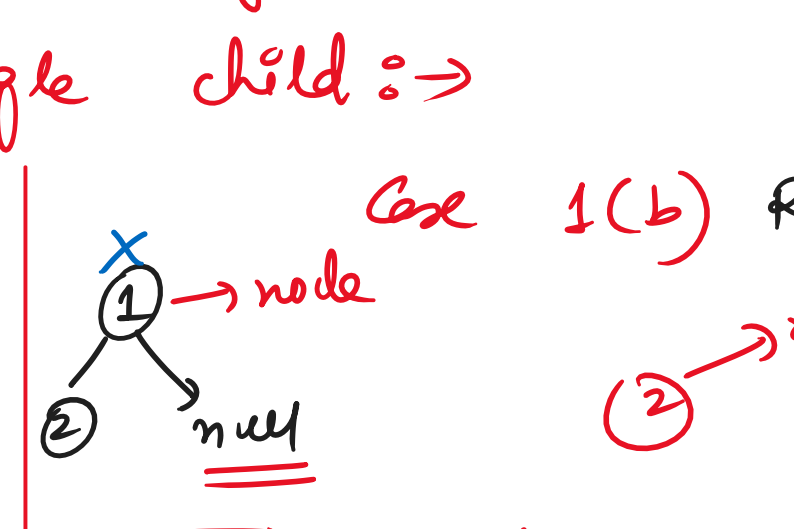
Case 1 (a) Left is null



```
TreeNode temp = node.right;
node = null;
return temp;
```

* node.right becomes new root.

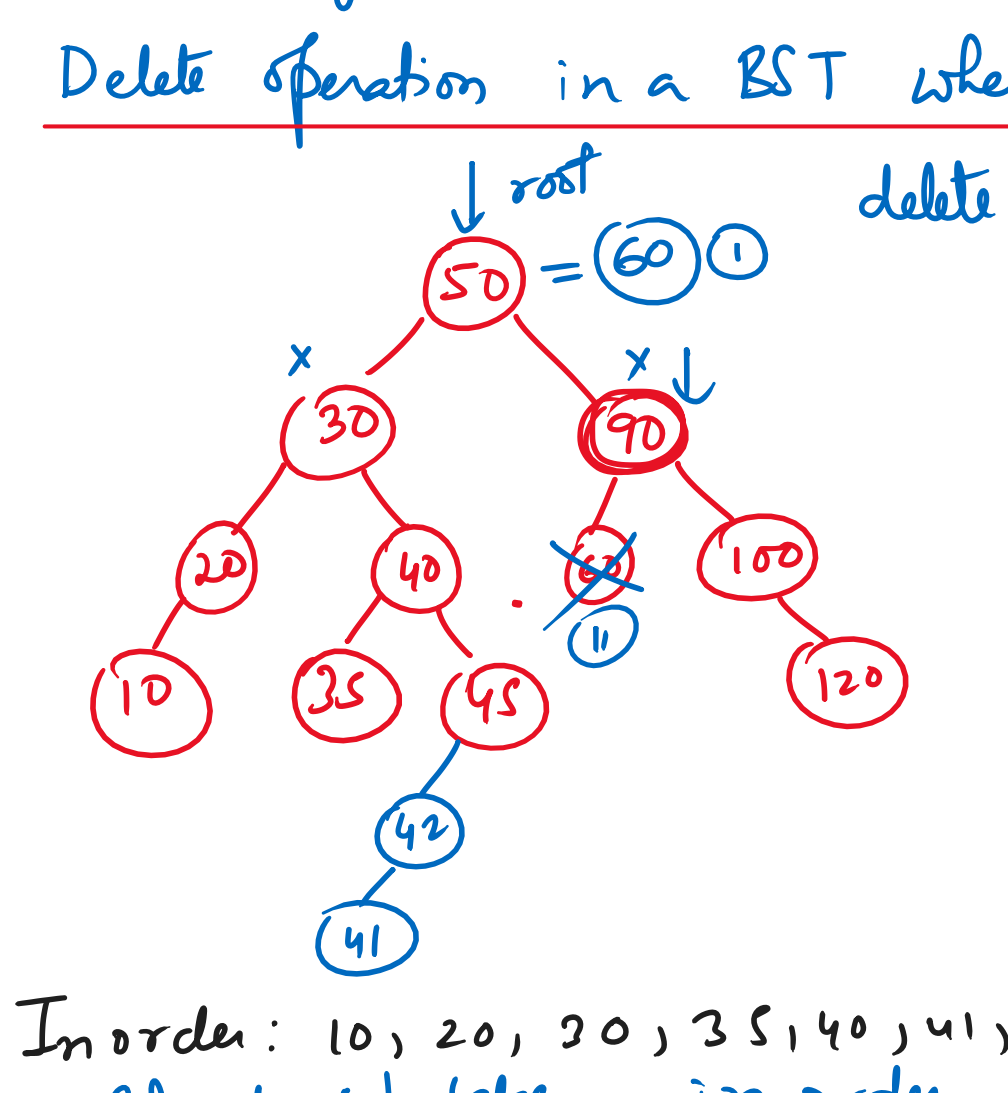
Case 1 (b) Right is null



```
TreeNode temp = node.left;
node = null;
return temp;
```

* node.left becomes new root.

Delete operation in a BST where the root or node has both children :→



delete(root, 50)

```
TreeNode temp =
  findMin(root.right)
  fM(90)
temp = 60;
root.data = temp.data;
root.right = delete(root.right,
                    temp.data);
```

In-order Successor

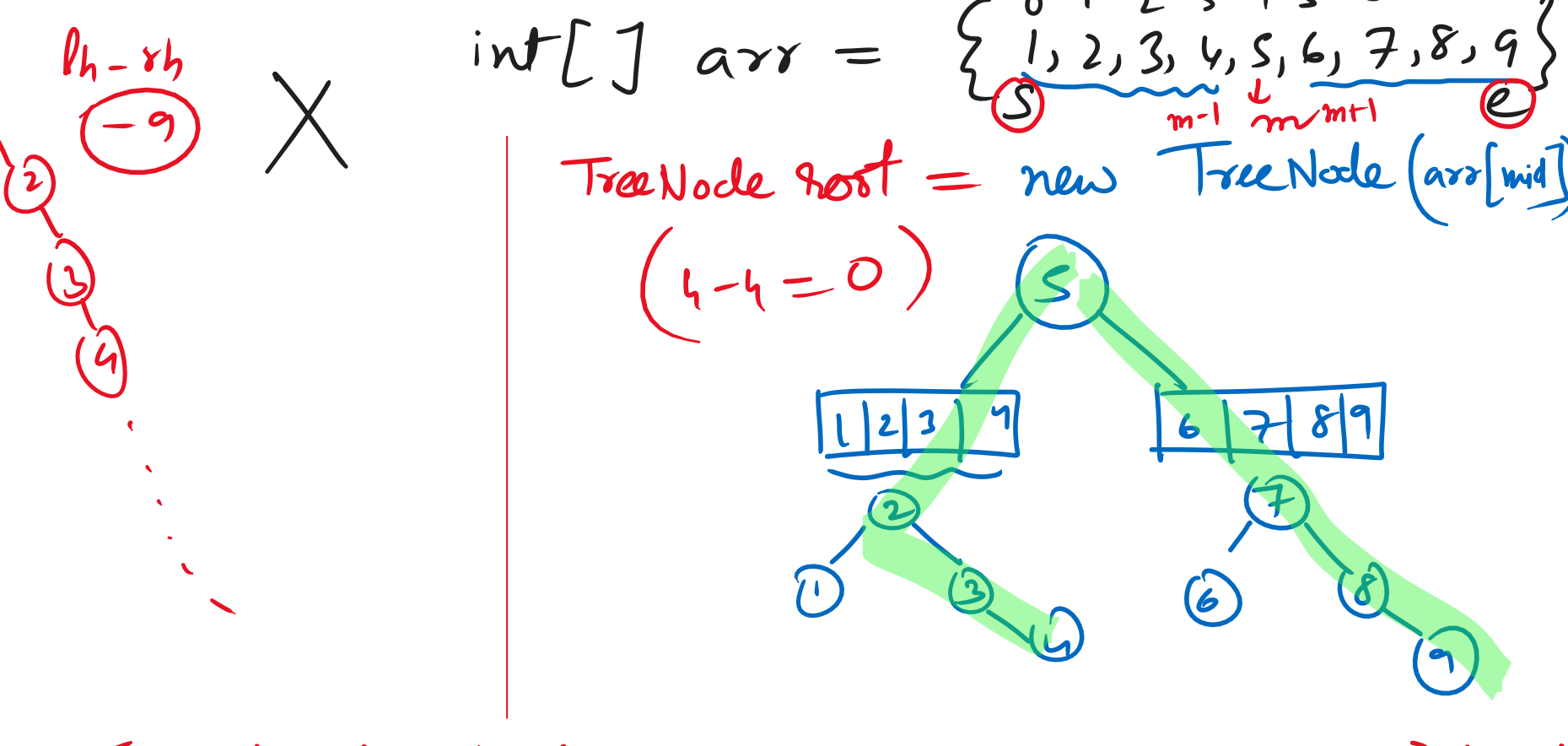
Inorder : 10, 20, 30, 35, 40, 41, 42, 45, (50), 60, 90, 100, 120
We don't take inorder predecessor because of complex restructuring.

* Capgemini | Accenture : Given a sorted array, convert it into a balanced BST :

s > e

int[] arr = {1, 2, 3, 4, 5, 6, 7, 8, 9};
(indices 0 1 2 3 4 5 6 7 8; s, m−1, m, m+1, e)

lh − rh
(−9) X



TreeNode root = new TreeNode(arr[mid]);
(4 − 4 = 0)

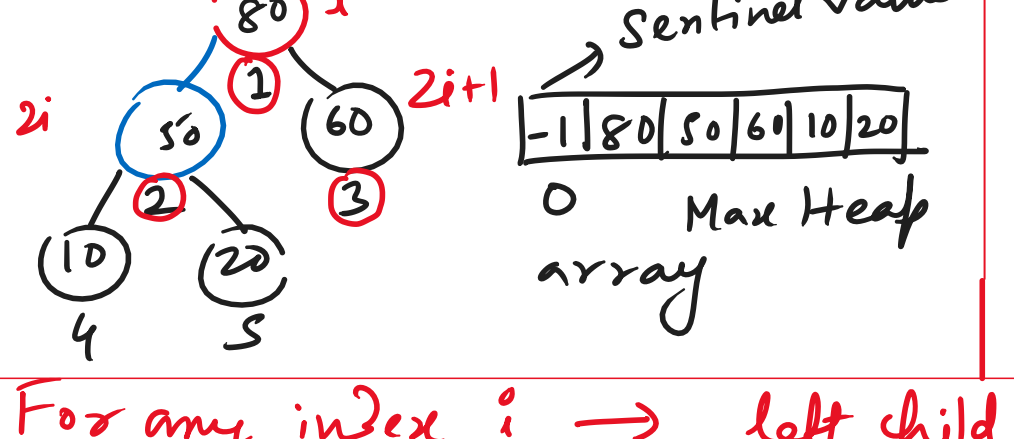
mid = s + (e − s)/2
= (s + e)/2
= 8/2
= (4)

* The Heap Data Structure :→ Heap
*(Complete Binary Tree) (L to R)

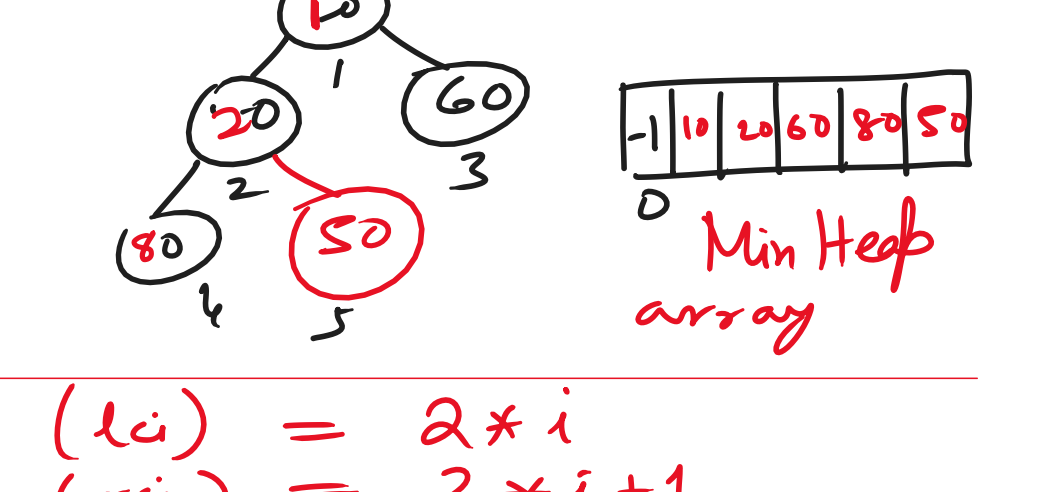
* In the searching algorithms, the least time taken to find a particular key in an array is Big O (log N).

* If some application requires us to find the max/min value in a data in constant time Big O (1), we use the heap data structure. [20, 80, 60, 10, 50]

Max-Heap : (Left < Node > Right)
* Node → Parent is always > children



Min-Heap (Left > Node < Right)
* Node → Parent is always < children



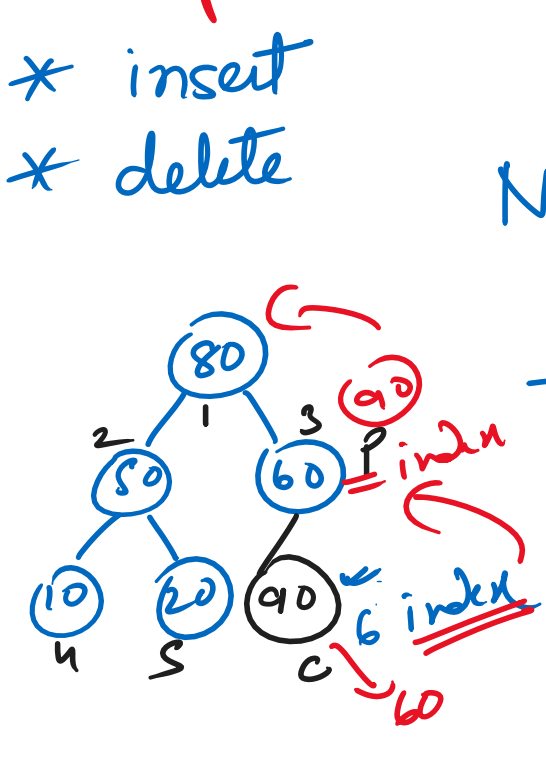
For any index i → left child index (lci) = 2 * i
right child index (rci) = 2 * i + 1
parent index (pi) = i / 2

* insert
* delete

No of leaf nodes : (n nodes) = n/2 + 1

Max Heap :

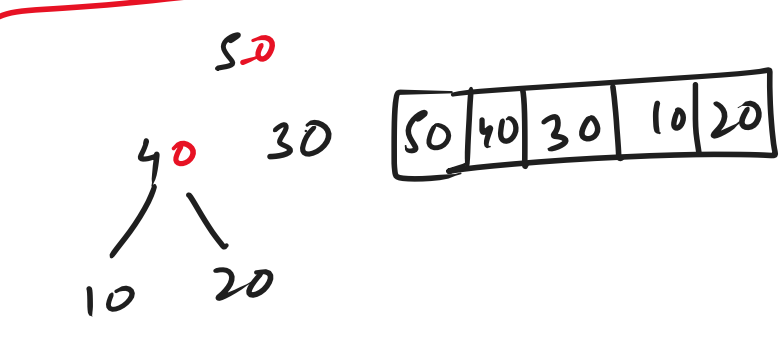
arr = | -1 | 80 | 50 | 60 | 10 | 20 | |
(indices 0 1 2 3 4 5 6) n = 5
↓ size



insert (90) data

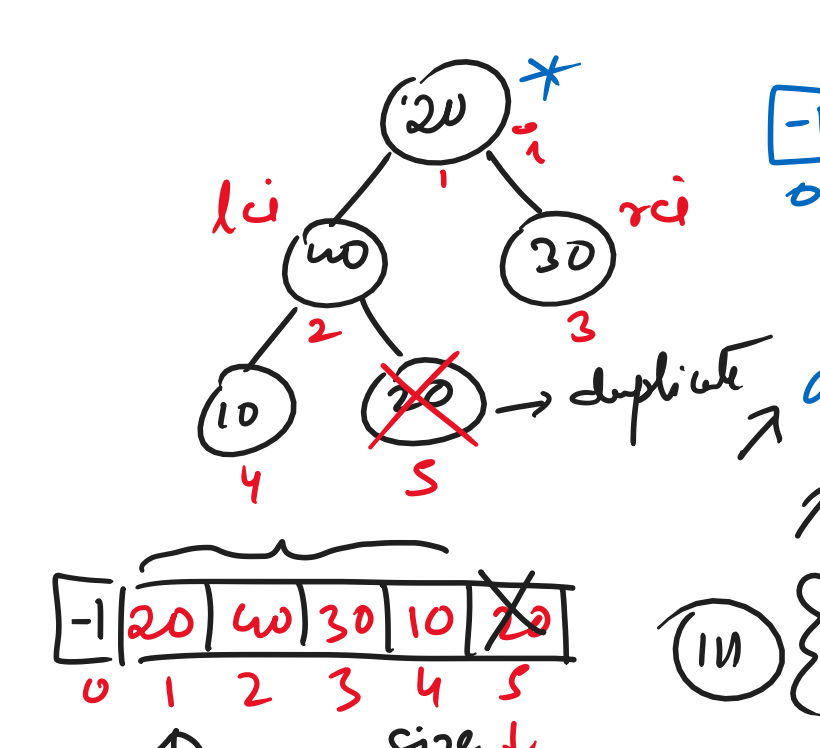
```
size ++
arr[size] = data;   (6)
int index = size;
int parent = index/2;   (3)
while (index > 1) {
  if (arr[index] > arr[parent])
      swap(index, parent);
      index = parent;
}
```

20, 30, 50, 10, 40



[Delete from Heap]

| -1 | 50 | 40 | 30 | 10 | 20 |
(indices 0 1 2 3 4 5; size)



arr[1] = arr[size] { root = 20 }
size − −; the duplicate gets deleted
But we have a problem index = 1
propagate the root to its correct position.

lci = 2×i
rci = 2×i + 1
while (i < size)
swap of l, r

→ * Heapify Method → Convert an array into Heap.
→ * Heap Sort Algorithm → Sort a heap in sorted order.
→ * Problems