MCSE501L/Data Structures and Algorithms
Module 5: Advanced Trees

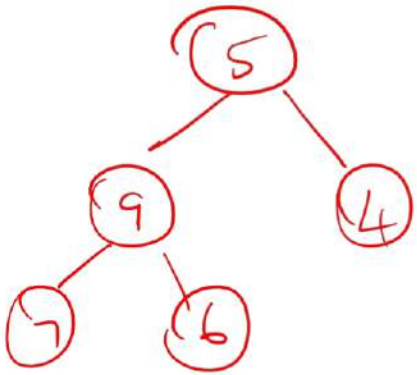
Syllabus: Module 5

Threaded binary trees, Leftist trees, Tournament trees, 2-3 tree, Splay tree, Red-black trees, Range trees.

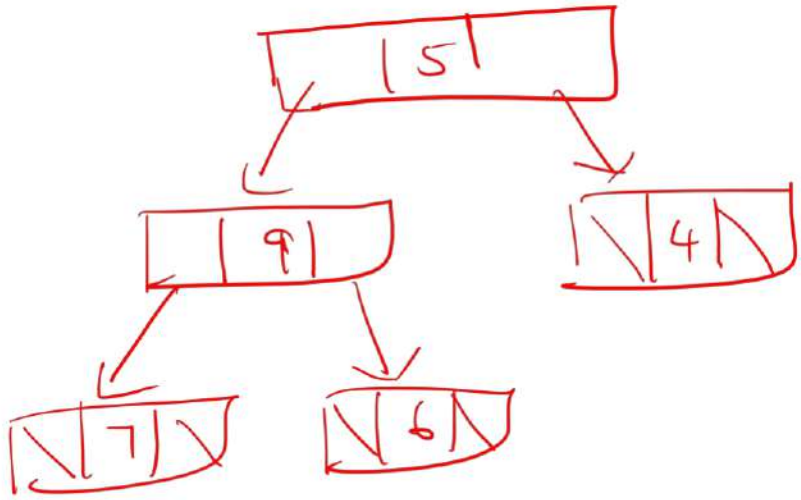
Threaded Binary Tree

Binary Tree & its problem

Example :-



linked list representation



No. of nodes = n = 5

No. of null pointers = n+1 = 6

Threaded Binary Tree

- Hence, instead of storing null pointer, we can make the null pointer to point to some node in the tree
- Based on what we store in the address field, the threaded tree can be classified into left threaded or right threaded binary tree
- In left threaded, store the address of predecessor node and in right threaded, store the address of successor node.
- A third type called fully (doubly) threaded binary tree stores address of both predecessor and successor node in the left and right respectively

Threaded Binary Tree

- Threaded Binary Tree is a binary tree where in the address fields will either hold the address of predecessor node or successor node

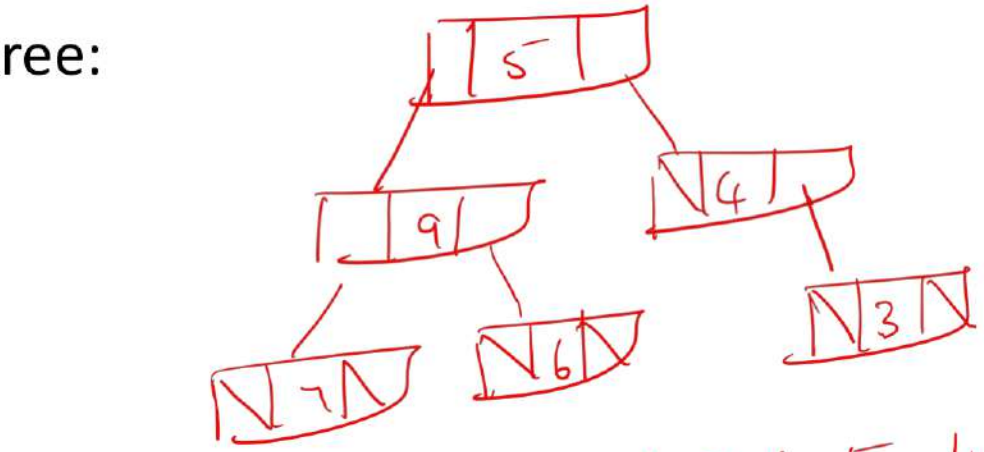
Threaded Binary Tree

- Need to know about the tree traversal to understand the predecessor node or successor node
- Based on the traversal also, it can be classified as
  - –Inorder threaded binary tree
  - –Preorder threaded binary tree
  - –Post order threaded binary tree

  And also in each of these categories, further we have 3 types:
  - Left
  - Right
  - Fully

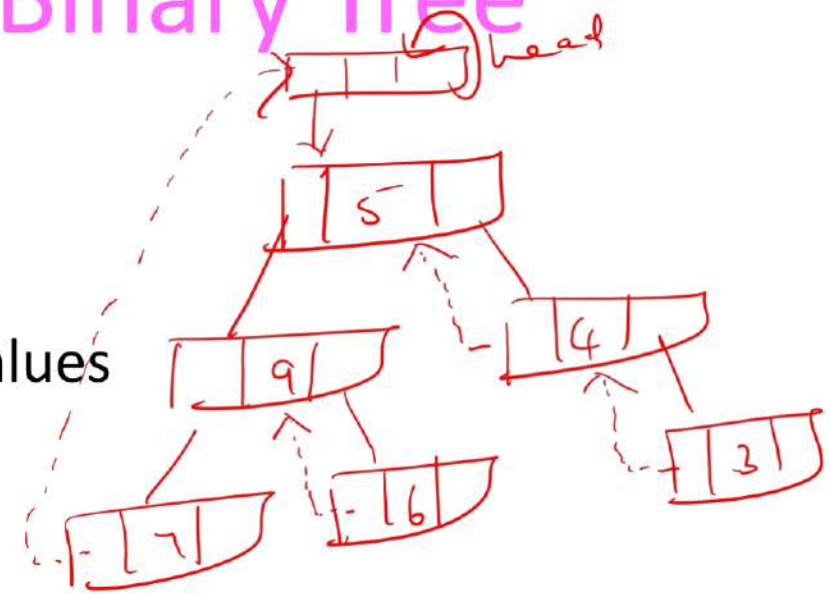
Threaded Binary Tree

Inorder threaded binary tree:



Inorder traversal: 7 9 6 5 4 3

Threaded Binary Tree



Inorder threaded binary tree:

1. Left threaded binary tree:

Just change only the left null values of node 7,6,3,4 to point to its Predecessor

For 6,3,4, the predecessors are available

But for 7, no predecessor is available.

In this case, create a head node whose left pointer points to root and right pointer points to itself.

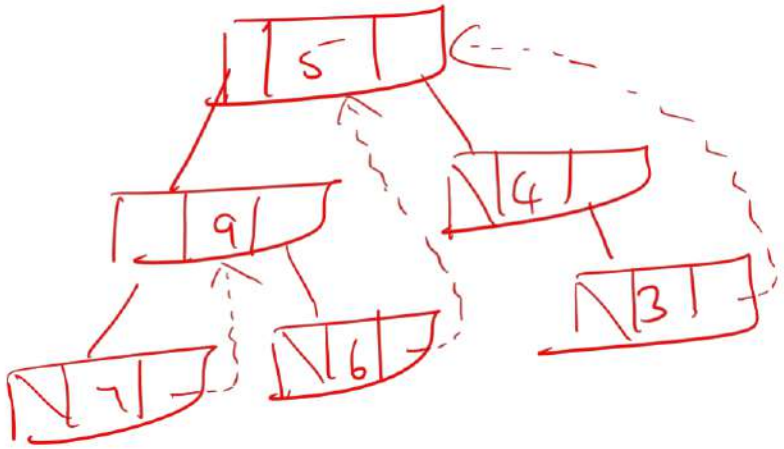
Now, left of node 7 can be made point to head

Threaded Binary Tree

Inorder threaded binary tree:

2. Right threaded binary tree:

Just change only the right null values of node 7,6,3 to point to its successor



For 7,6 the successors are available

But for 3, no successor is available.

In this case, right of node 3 can be made point to head

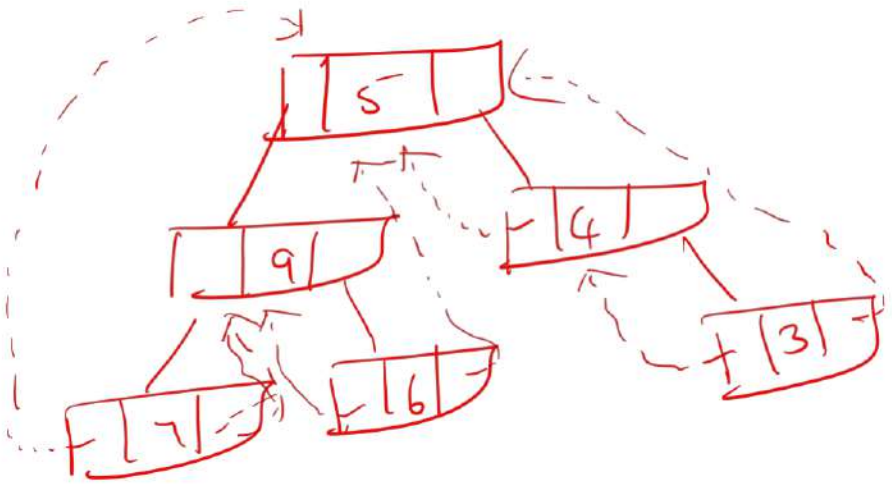
Threaded Binary Tree

Inorder threaded binary tree:

3. Fully threaded binary tree:

Have both left and right points to predecessor

and successor respectively



Threaded Binary Tree

- ***Threaded Binary Tree*** makes use of NULL pointers to improve its traversal process.
- In a threaded binary tree, NULL pointers are replaced by references of other nodes in the tree.
- These extra references are called as ***threads***.
- Types – Single threaded (left or right threaded) and double/fully threaded

Threaded Binary Tree

Structure of a Threaded Binary Tree

A node in a threaded binary tree has the following structure:

- **Data**: Stores the value.
- **Left pointer**: Points to the left child or to the in-order predecessor if it's a thread.
- **Right pointer**: Points to the right child or to the in-order successor if it's a thread.
- **Left thread flag**: Indicates whether the left pointer is a thread or a normal pointer.
- **Right thread flag**: Indicates whether the right pointer is a thread or a normal pointer

Threaded Binary Tree

Structure of a Threaded Binary Tree

```
struct node
{
int data;
struct node *left;
struct node *right;
int lflag;
int rflag;
}
```

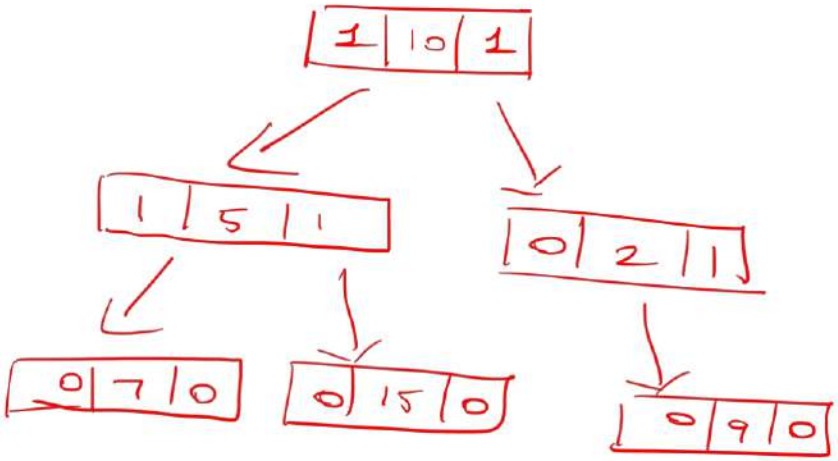
lflag/rflag:

0→ no children (thread)

1→ there is a children

Threaded Binary Tree

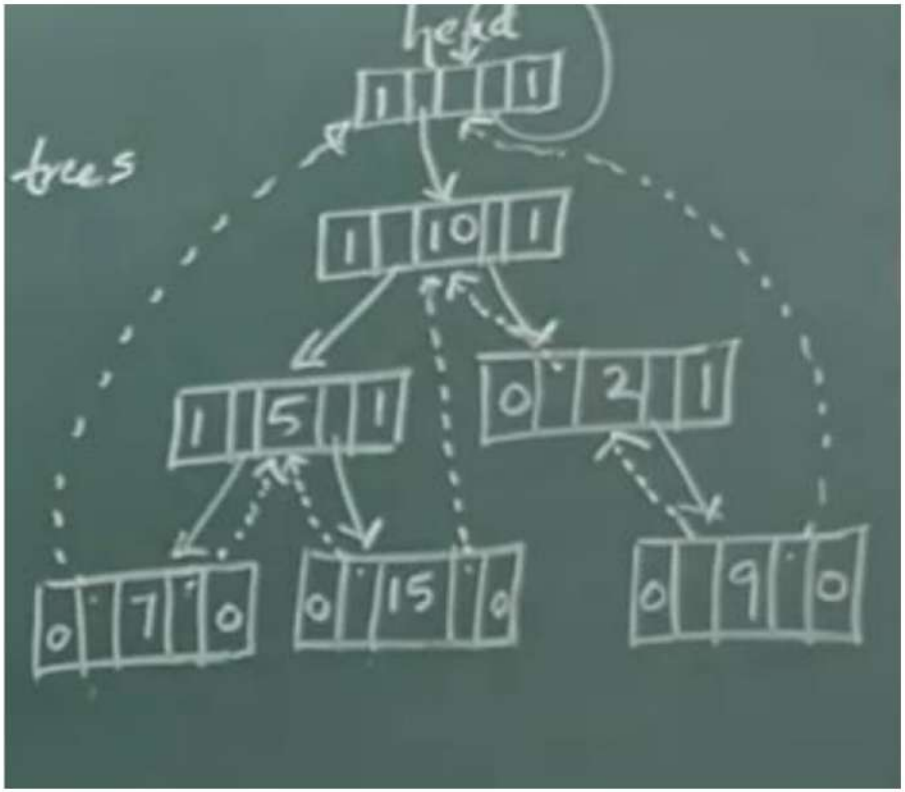
Example:



Threaded Binary Tree

Traversal:

```
void inorder(node *head)
{
node *temp=head->left;
while(temp!=head)
{
  while (temp->lflag!=0)
  {
    temp=temp->left;
   }
   printf("%d",temp->data);
      while(temp->rflag==0)
      {
        temp=temp->right;
          if(temp==head)
                return;
         printf("%d",temp->data);
     }
     temp=temp->right;
}
}
```



Threaded Binary Tree

Traversal:

Inorder traversal→ go to
the leftmost part of tree

```
while (temp-> lflag!=0)
{
temp=temp->left;
}
```

and print it's data:

```
printf("%d",temp->data);   // initially 7 here
```

Then,

```
while(temp->rflag==0)
{
temp=temp->right;     //here is it node 5
if(temp==head)        // not a head
   return;
printf("%d",temp->data);   // here it is 5
}
```

Then,

```
temp=temp->right;    //here it is node 15
```

Threaded Binary Tree

- Time complexity:

Operation	Time Complexity (Balanced Tree)	Time Complexity (Unbalanced Tree)
In-order Traversal	O(n)	O(n)
Searching	O(log n)	O(n)
Insertion	O(log n)	O(n)
Deletion	O(log n)	O(n)

Threaded Binary Tree

Space Complexity

- A threaded binary tree stores additional pointers (threads) for nodes. However, this is more space-efficient compared to using stacks or recursion for in-order traversal.
- The **space complexity** for storing the tree is **O(n)**, similar to a regular binary tree.

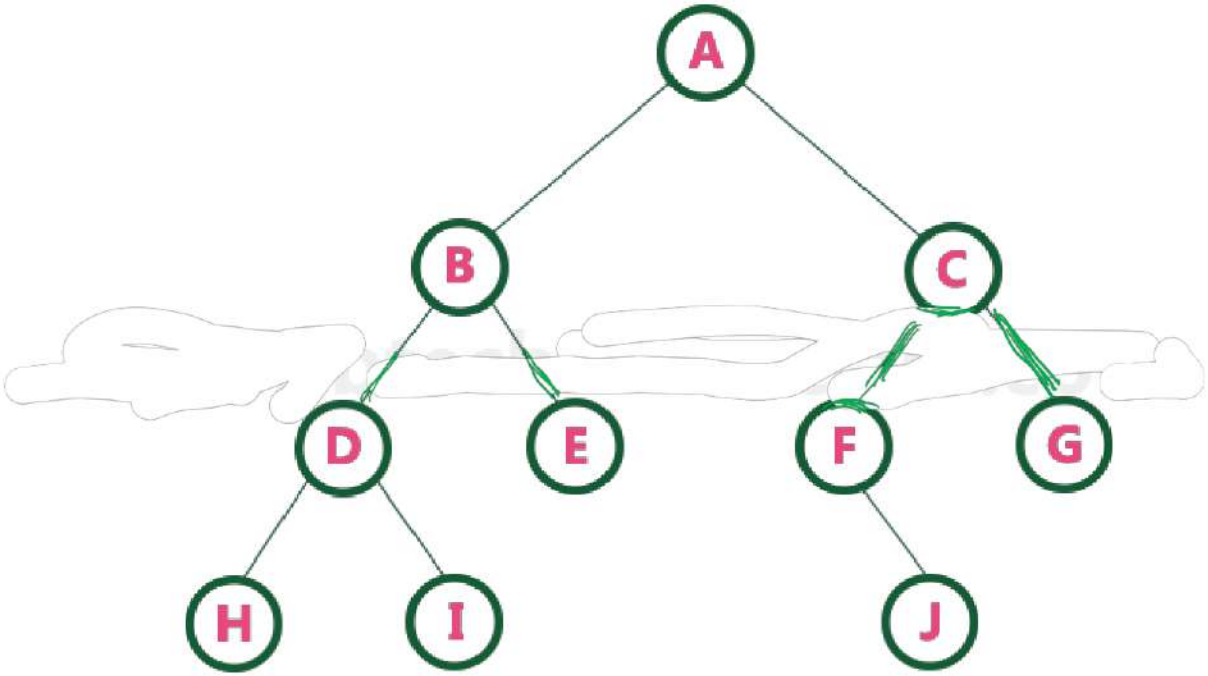
Threaded Binary Tree

- Threaded Binary Tree is also a binary tree
- in which
  - all left child pointers that are NULL (in Linked list representation) points to its in-order predecessor
  - all right child pointers that are NULL (in Linked list representation) points to its in-order successor.

Threaded Binary Tree

If there is no in-order predecessor or in-order successor, then it points to the root node.

Threaded Binary Tree

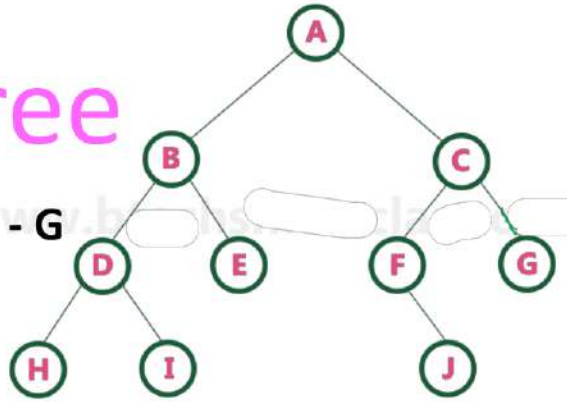


To convert the example binary tree into a threaded binary tree, first find the in-order traversal of that tree...

In-order traversal of above binary tree...H - D - I - B - E - A - F - J - C - G

Threaded Binary Tree

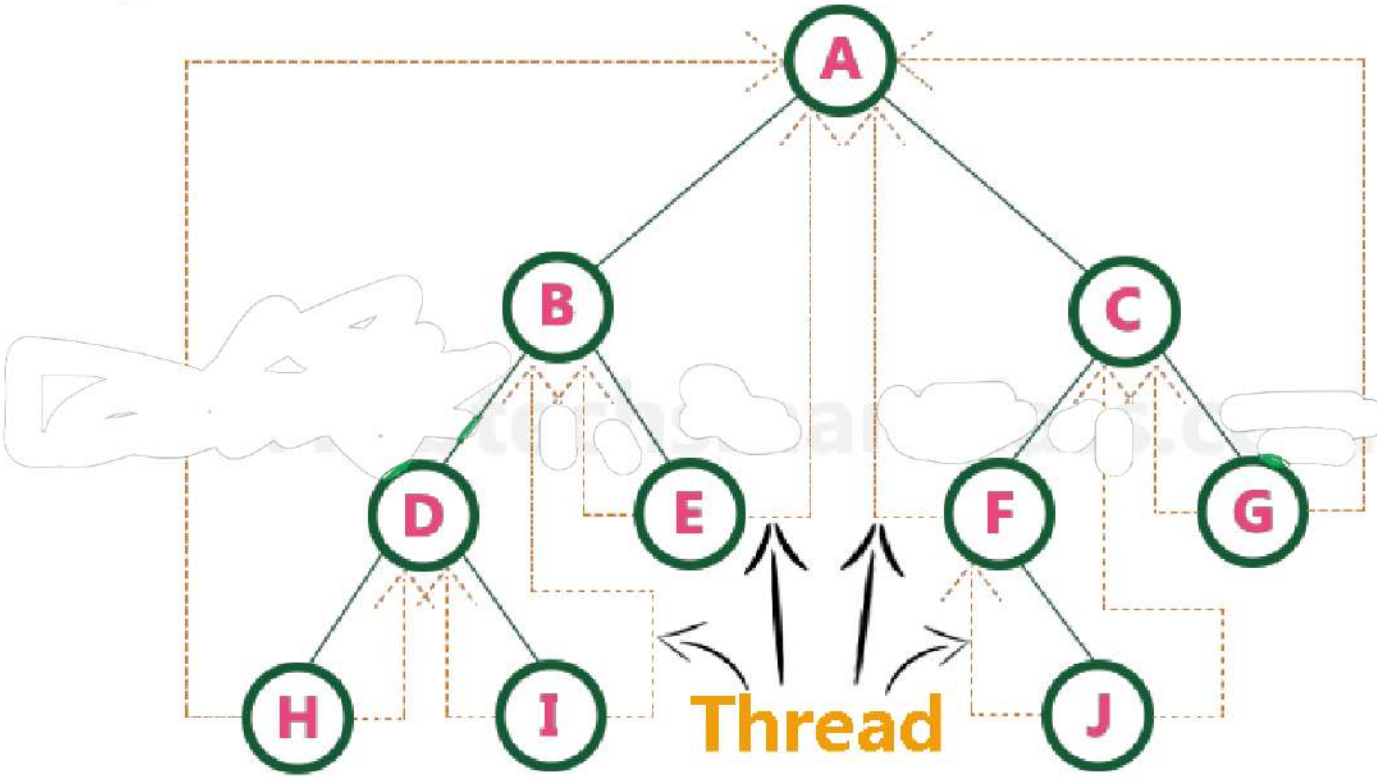
In-order traversal of above binary tree...H - D - I - B - E - A - F - J - C - G



- When we represent the example binary tree using linked list representation, nodes **H, I, E, F, J** and **G** left child pointers are NULL.
- This NULL is replaced by address of its in-order predecessor respectively (I to D, E to B, F to A, J to F and G to C), but here the node H does not have its in-order predecessor, so it points to the root node A.
- Nodes **H, I, E, J** and **G** right child pointers are NULL.
- These NULL pointers are replaced by address of its in-order successor respectively (H to D, I to B, E to A, and J to C), but here the node G does not have its in-order successor, so it points to the root node A.

Threaded Binary Tree

- Example binary tree is converted into threaded binary tree as follows.
- In the figure, threads are indicated with dotted links.



Advantages of Threaded Binary Tree

- In general, Inorder traversal of binary search tree is either be done using recursion or using iterative method with an auxiliary stack.
- The idea of the threaded binary trees is to make inorder traversal faster and do it without stack and without recursion
- Without recursion, it will not take more memory and computational time
- It facilitates forward and backward traversal from a node

Disadvantages of Threaded Binary Tree

- Structure is complex
- Insertion and deletion are more complex

Applications of Threaded Binary Tree

- To traverse large datasets efficiently
- In systems where tree traversal is frequently performed, such as compiler design (e.g., syntax trees), and the in-order traversal is essential for operations like expression evaluation, threaded binary trees can provide a performance boost.

Leftist trees

Leftist tree

A leftist tree is a type of binary tree used to implement priority queues efficiently.

It differs from a regular binary heap by maintaining a structural property based on the concept of null path length (NPL) or S-vlaue.

This structure ensures efficient merging of trees, a key operation in priority queues.

Leftist trees

- A Leftist Tree is a type of binary tree that is **skewed to the left (heavier left subtree)**, meaning that it maintains a special property called the "leftist property."

- This property ensures that the **shortest path from the root to a leaf (the null path length, NPL) is always on the right subtree (as left subtree is heavy).**

- This skewness **allows for efficient merging of trees**, making leftist trees particularly useful in implementing priority queues.

Leftist tree

Key Properties of Leftist Trees:

1. **Heap Order Property**: The root of each subtree contains the smallest key (in a min-leftist tree) or the largest key (in a max-leftist tree), similar to a heap.
2. **Leftist Property**: For every node, the null path length of the left child is greater than or equal to the null path length of the right child. This ensures that the right subtree is always more shallow, allowing efficient merging of trees.

   [Null Path Length (NPL) or S-value: For any node, the null path length is the shortest distance from the node to a leaf (external node). This ensures that the left subtree is always "heavier" than the right subtree.]

Leftist trees

- Heap order and leftist properties:

  1. key(i) >= key(parent(i))

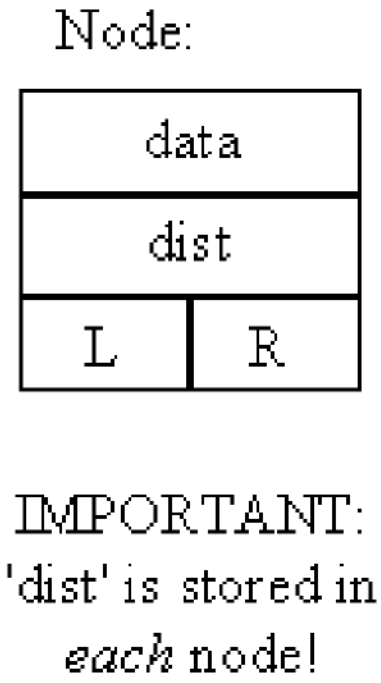
  – The root contains the minimum key. As with array heaps, we could have the maximum at the top, simply by changing this property and modifying the code accordingly.

  2. dist(right(i)) <= dist(left(i))

  – The shortest path to a descendant external node is through the right child.

Leftist trees

- Here's a leftist tree node representation:



Leftist trees

- Need to understand S-value:
  - S-value/S(x) **(or rank or distance)**
  - S-value is the shortest path length (or minimum distance) from any node to any external node

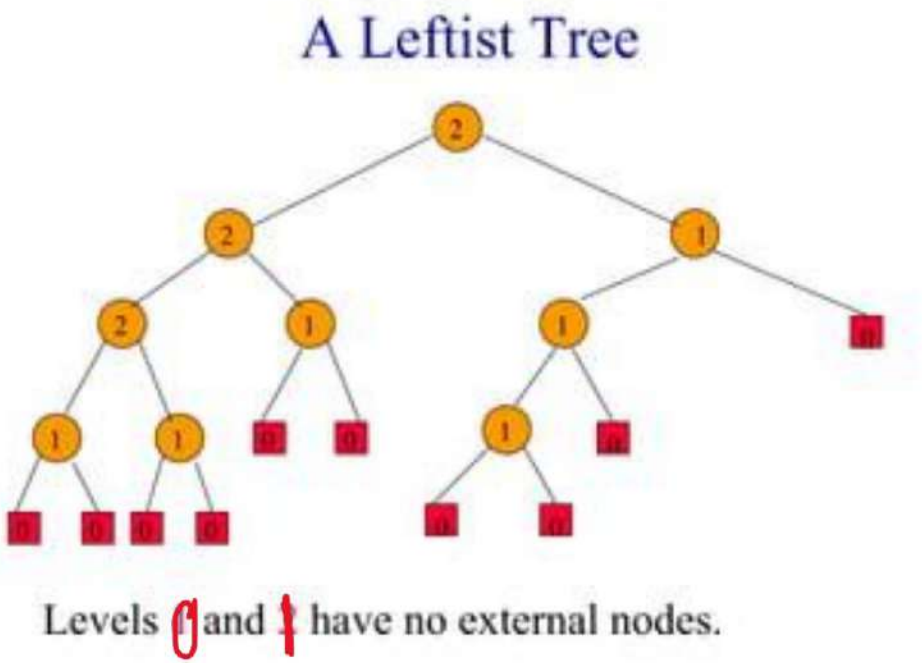
Leftist trees

- For each node, we store the distance to the closest leaf in the subtree anchored at that node.
- Let us call this value **s-value or dist or rank**.
- Compared to a binary heap, which is always a complete binary tree, a leftist heap strives to be extremely imbalanced.

Leftist trees

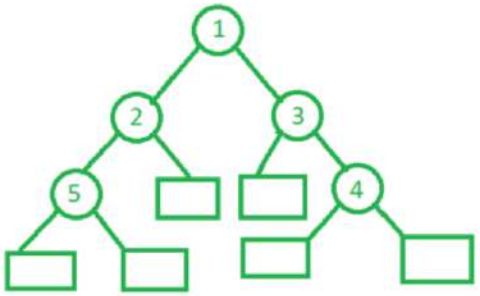
- Consider an extended tree which has external nodes
- Extended tree representation (In this representation, a null child is considered as external or leaf node).

To make it look like every node has 2 children, add null nodes or external nodes, represented in squares.
All other circle nodes are considered as internal nodes



Leftist trees

- Extended binary tree is a type of binary tree in which all the null sub tree of the original tree are replaced with special nodes called external nodes whereas other nodes are called internal nodes

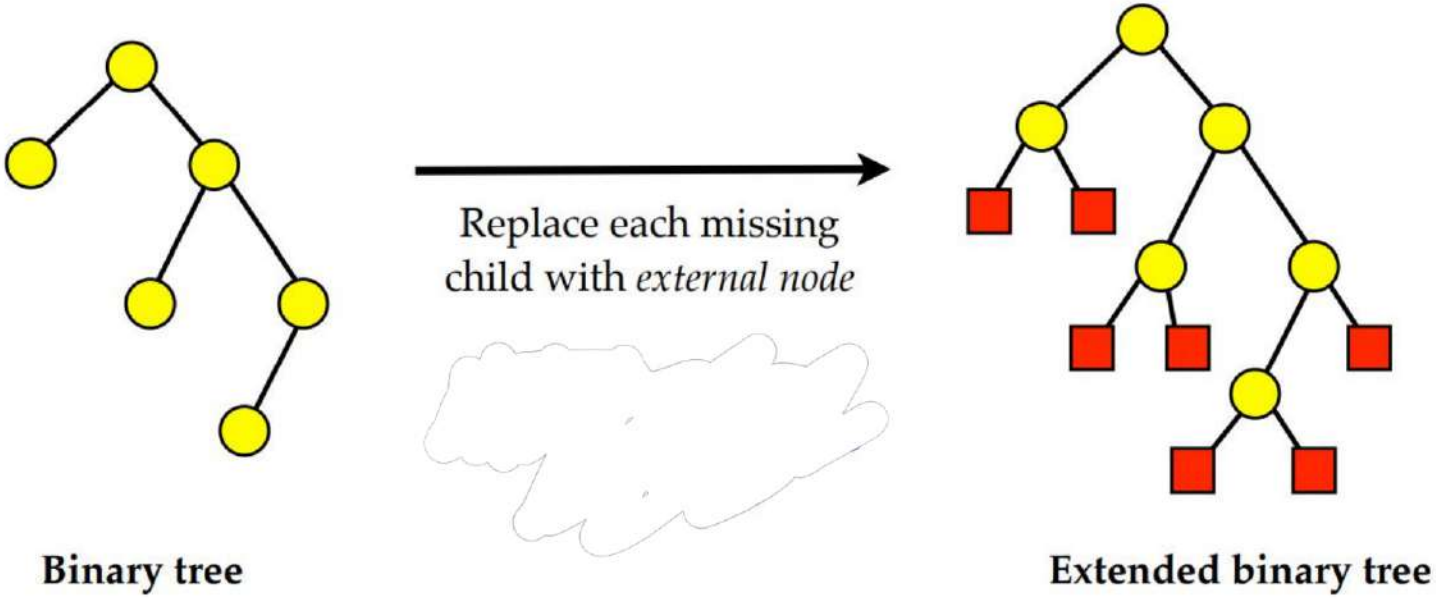


Extended Binary Tree

Here the circles represent the internal nodes and the boxes represent the external nodes.

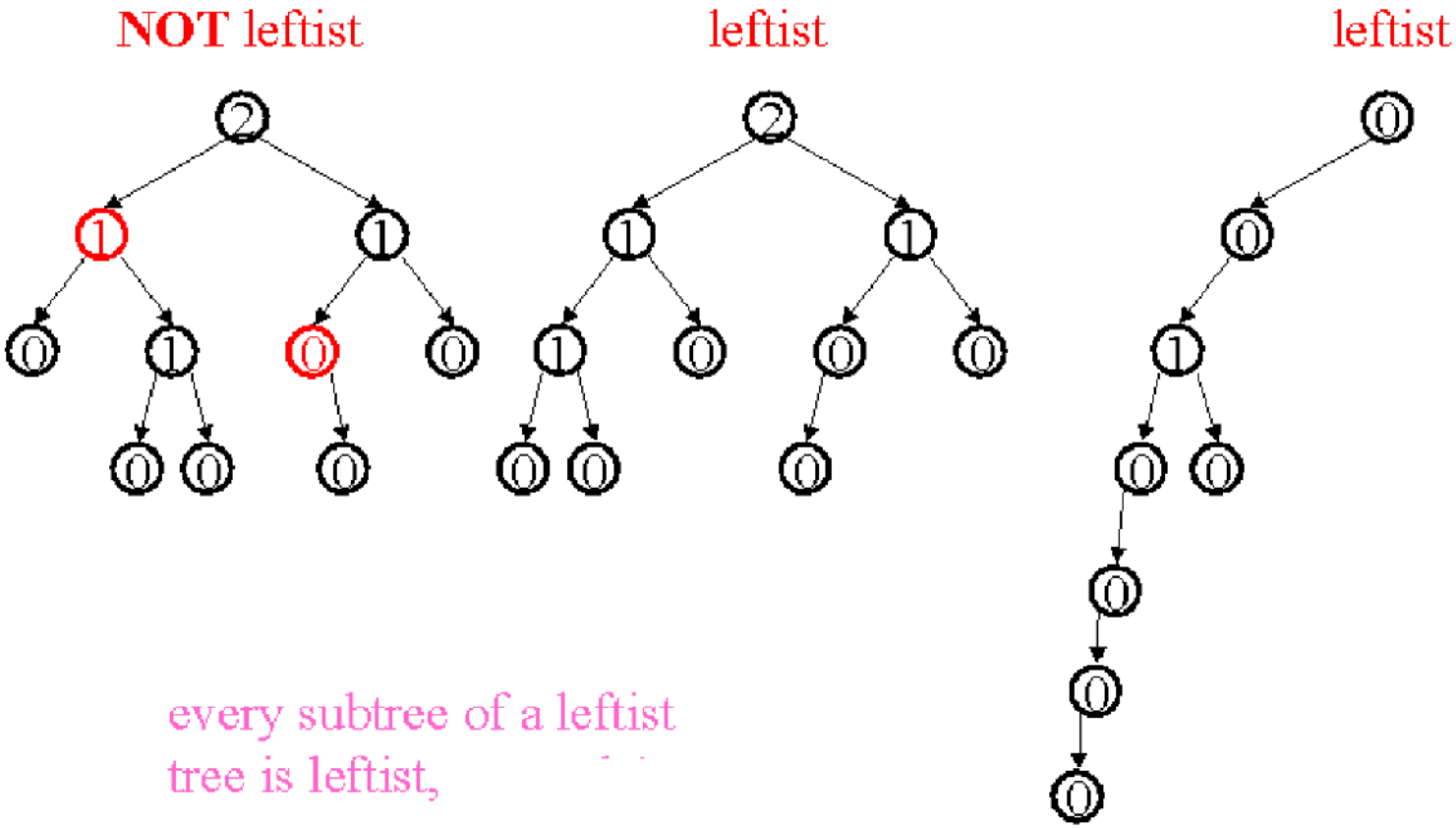
Leftist trees

Extended Binary Trees



Leftist tree in an extension of extended binary tree

Leftist tree examples



every subtree of a leftist tree is leftist,

Leftist trees

- Every node has a structure:
  - Data
  - Left pointer
  - Right pointer
  - S-value

```
struct LeftistNode {
int data;
int s_value;
struct LeftistNode* left;
struct LeftistNode* right;
};
```

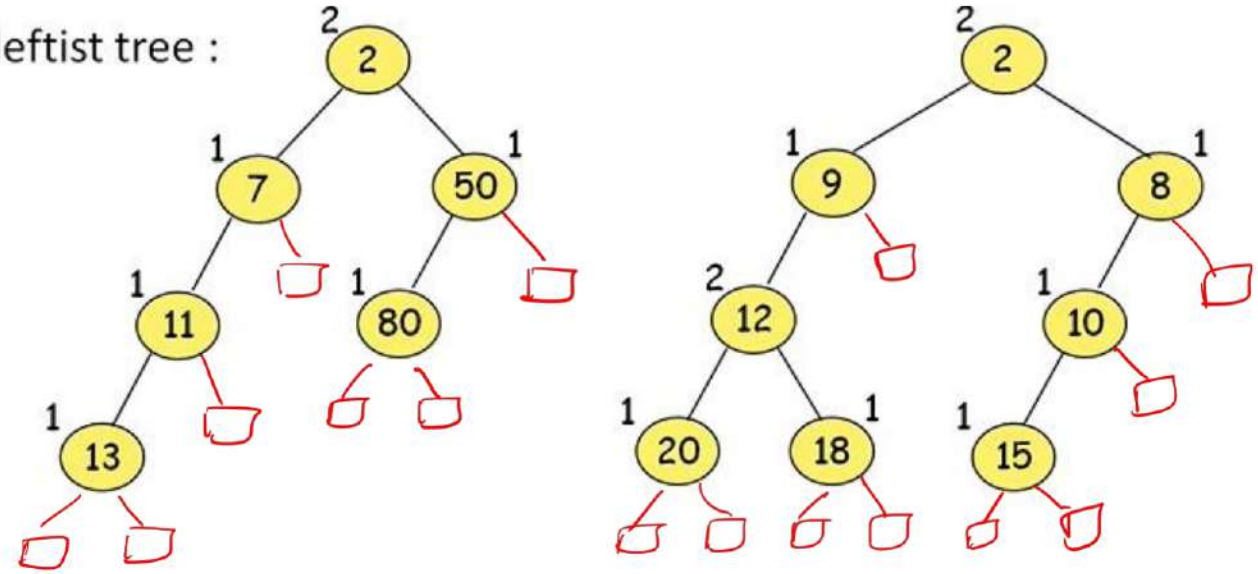
Leftist trees

- S-value of any internal node:
  - S(x)= min(S(left child(x)), S(right child(x)))+1
- S-value of any external node:
  - S(x)=0

Leftist trees

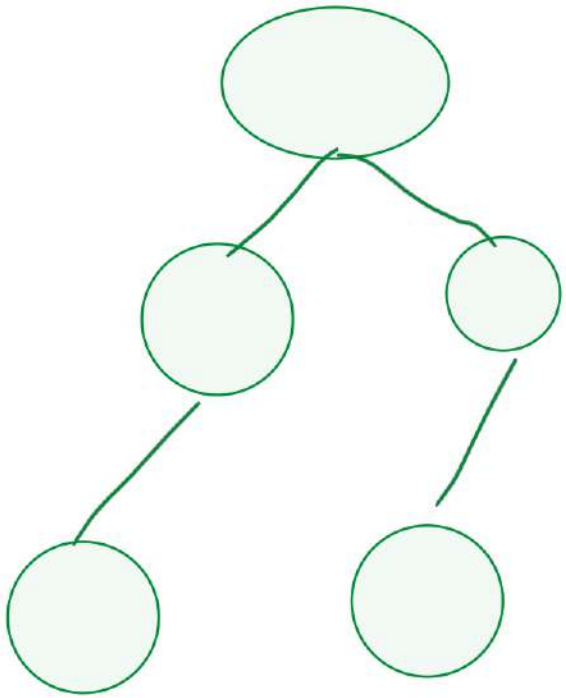
- Given a tree, find out whether it is a min leftist tree

Example of min leftist tree :



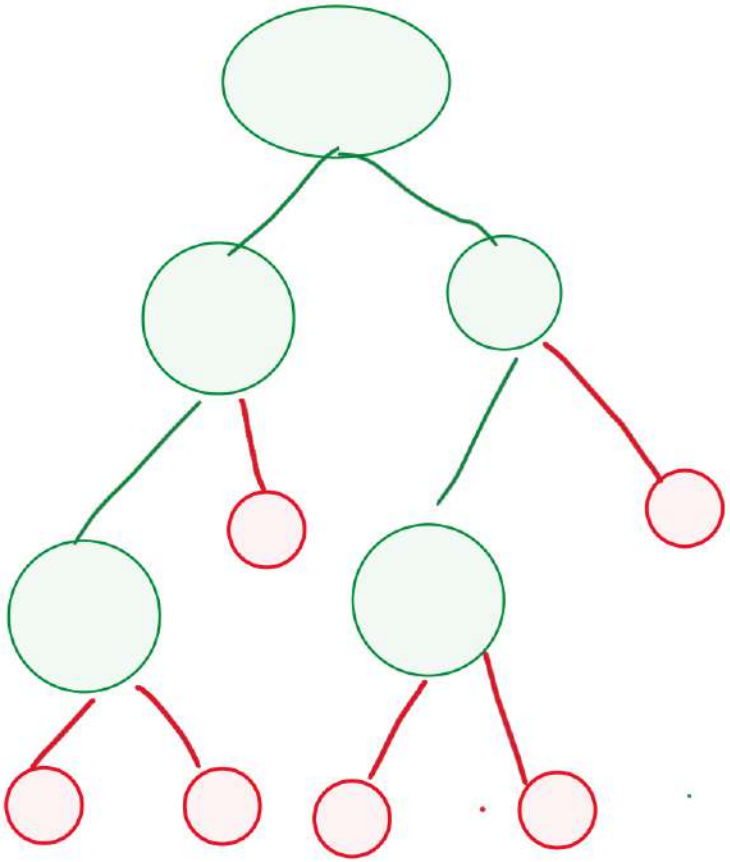
Leftist trees

Whether the given tree is a leftist?



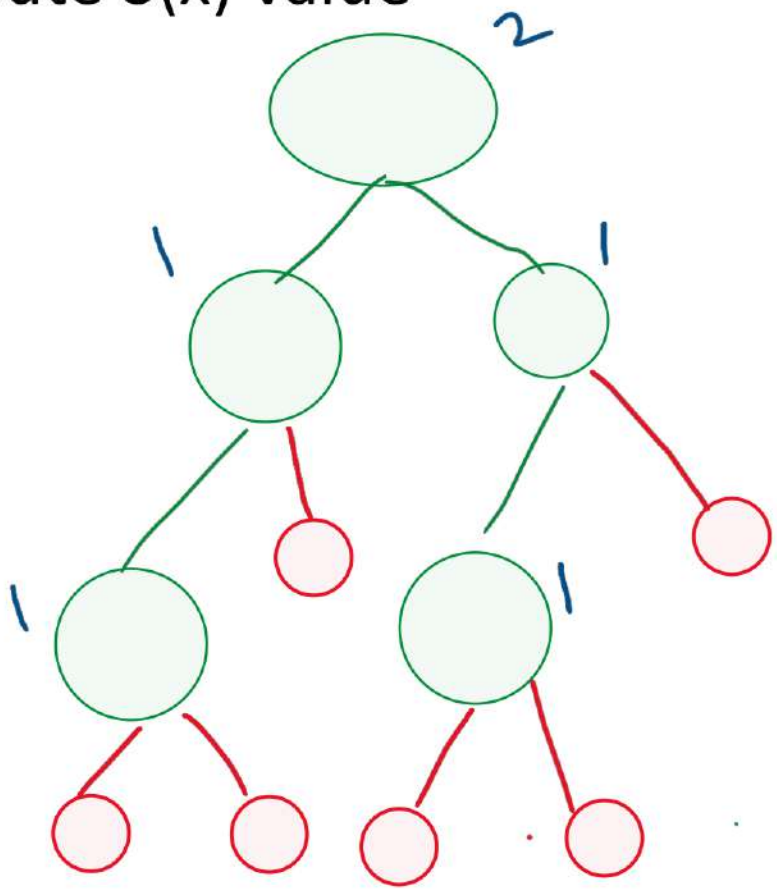
Leftist trees

Draw the external nodes (extended tree)



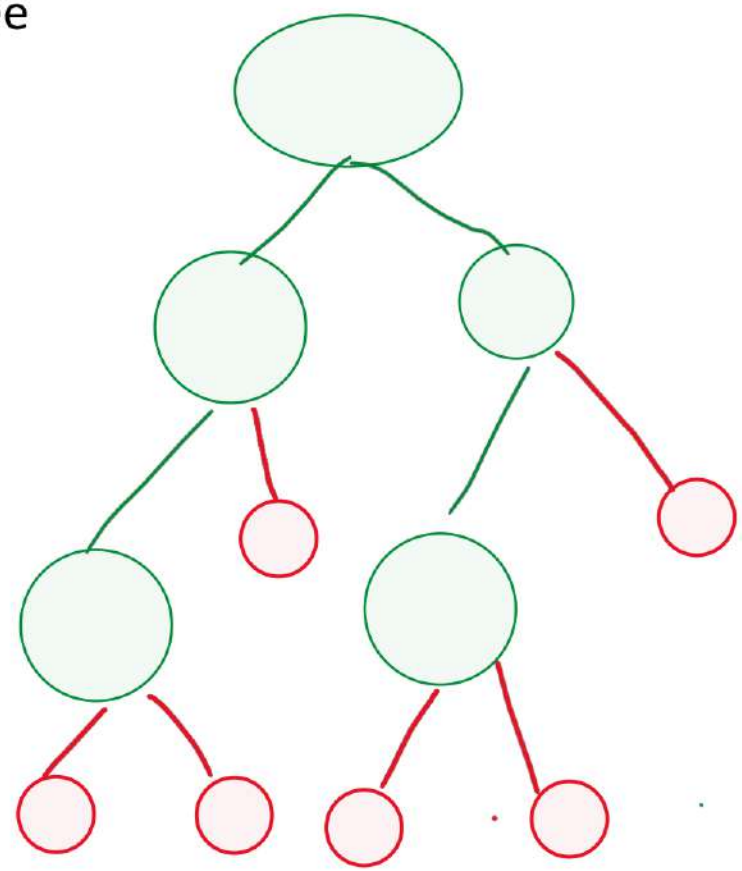
Leftist trees

Compute S(x) value



Leftist trees

It's a leftist satisfying the property, S(left(x)) >=S(right(x)) though it is not a complete binary tree



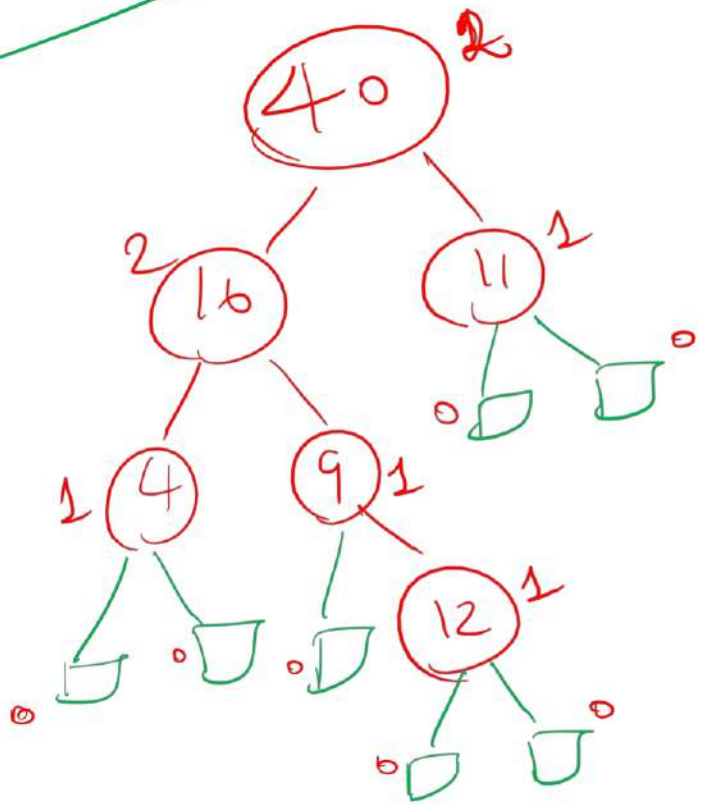
Why Leftist trees

Use:

- **Scenario:**
- Imagine you're managing a priority queue where tasks have different priorities. You need to frequently merge different queues together, like combining tasks from different departments.
- **Without a Leftist Tree:**
- Using a simple binary heap, merging two heaps (priority queues) could involve rebuilding the entire structure, which could take linear time relative to the number of tasks, making it inefficient for large datasets.
- **With a Leftist Tree:**
- The leftist tree allows for an efficient merge by keeping the tree shallow on the left side. When you merge two leftist trees (priority queues), you only need to merge the right subtrees and swap nodes as necessary to maintain the leftist property. This makes the operation much faster.

Leftist trees (not min leftist)

S-value



node x

data :
* left
* right
S(x)

S(9) = shortest path from node 9 to any external node

9 - 12 - left external node

9 - 12 - right external node

9 - left external node

This is the shortest path ∴ S(9) = 1

Leftist trees

(or) for any internal node x

$$s(x) = \min\left(s(\text{left}(x)), s(\text{right}(x)) \right) + 1$$

eg for $x = 40$

$$= \min(2, 1) + 1$$

$$= 1 + 1$$

$$= 2$$

Leftist trees

x → internal node

Leftist tree

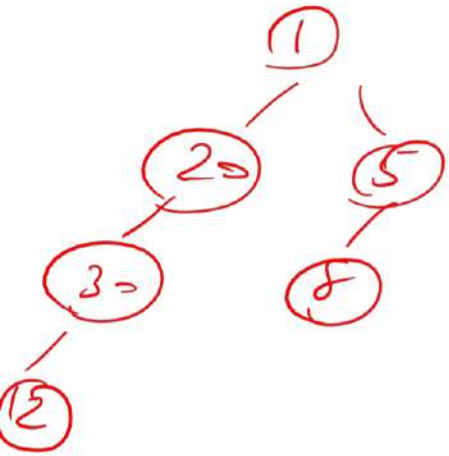
↳ binary tree

↳ for all x,

$$s(left(x)) \geq s(right(x))$$

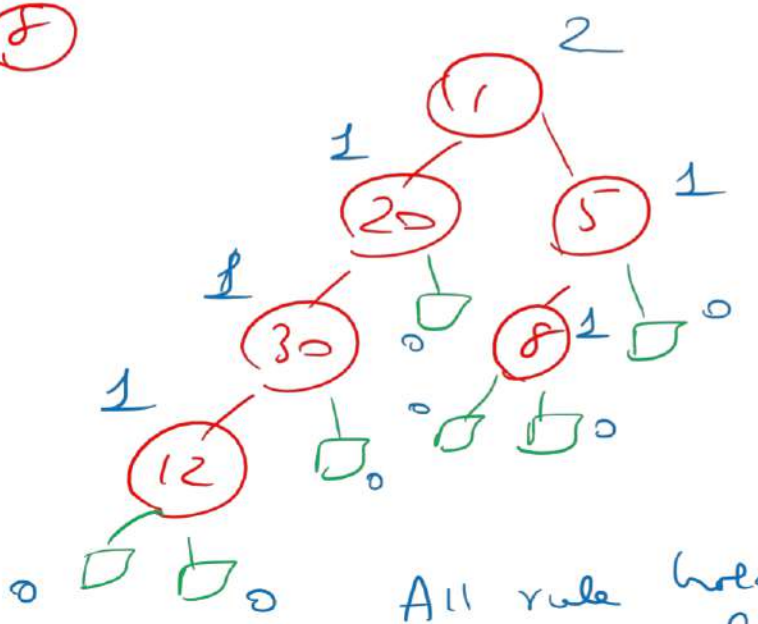
Leftist trees

Find out whether it is a leftist tree or not



Rule 1 : binary tree

Rule 2 : $s(\text{left}(x)) \geq s(\text{right}(x))$



→ check rule 2 for all x (internal nodes)

eg

$s(\text{left}(30)) = 1$ → rule 2

$s(\text{right}(30)) = 0$

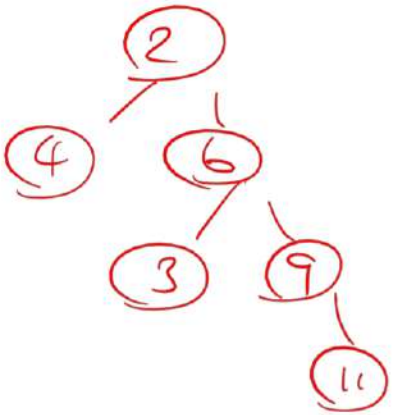
$(> 0$

All rule holds

∴ it is a leftist tree

Leftist trees

Find whether the tree is leftist ?



Ans : NO

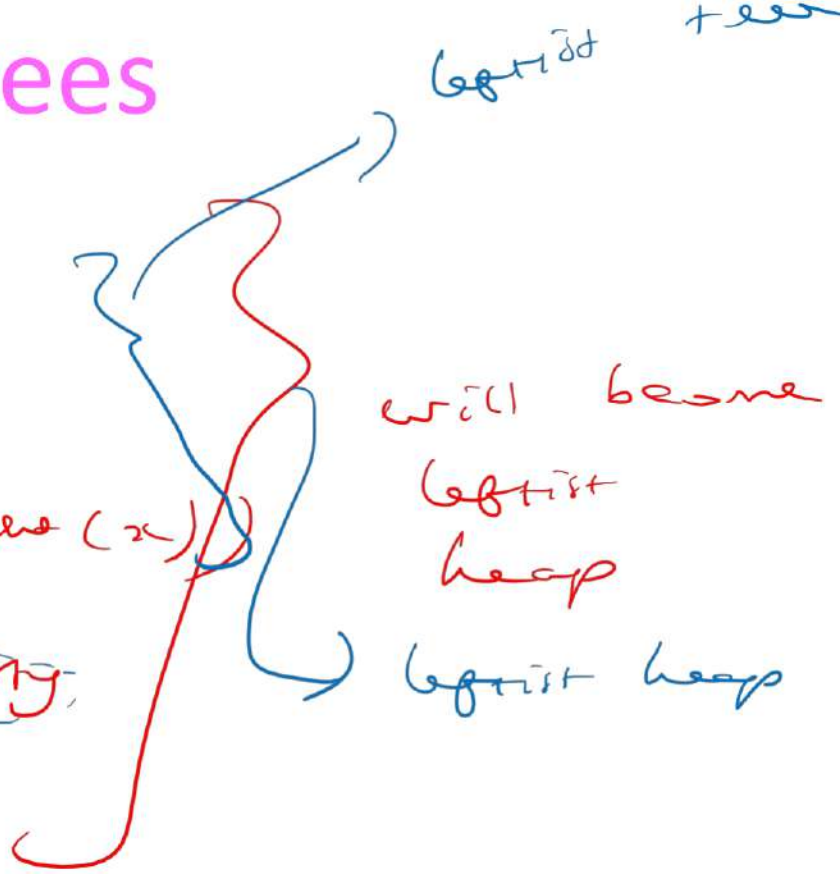
Leftist trees

→ binary tree

→ for all x,

$$s(left(x)) \geq s(right(x))$$

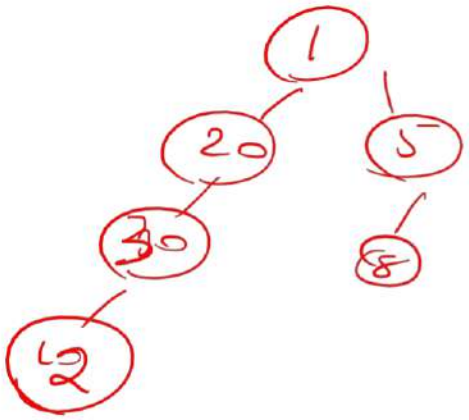
→ heap ordering property



Note

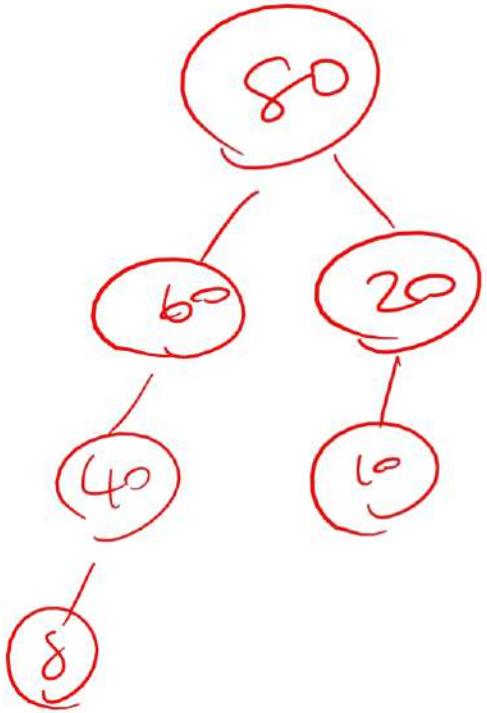
leftist tree are not complete binary tree as it doesnot follow structural property and the leftist heap is also not a complete binary tree (just follows heap order property, not the structural property)

Leftist trees



→ It is a leftist tree but not a leftist heap as it doesn't follow heap order (max heap) property (or) min heap

Leftist tree



→ leftist heap (max heap)

→ all properties are satisfied

Note :-

leftist heap = leftist tree + heap order property

Leftist trees

- **Operations**
  - The main operations performed on a leftist tree include insert, extract-min and merge.
  - The insert operation simply adds a new node to the tree. Simply, create a new node and merge it with the existing leftist tree.
  - The extract-min operation removes the root node and updates the tree structure to maintain the leftist property. The deletion of the minimum element (in a min-heap) is done by removing the root and merging its left and right subtrees.
  - The merge operation combines two leftist trees into a single leftist tree by linking the root nodes and maintaining the leftist property.

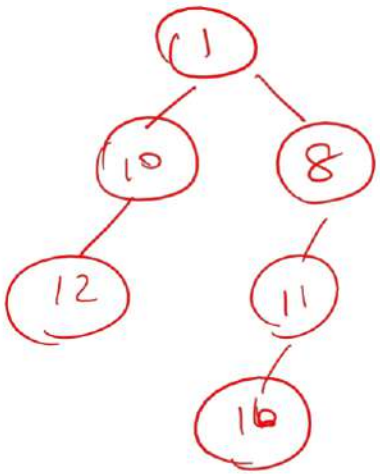
Leftist merge/meld

Exercise:

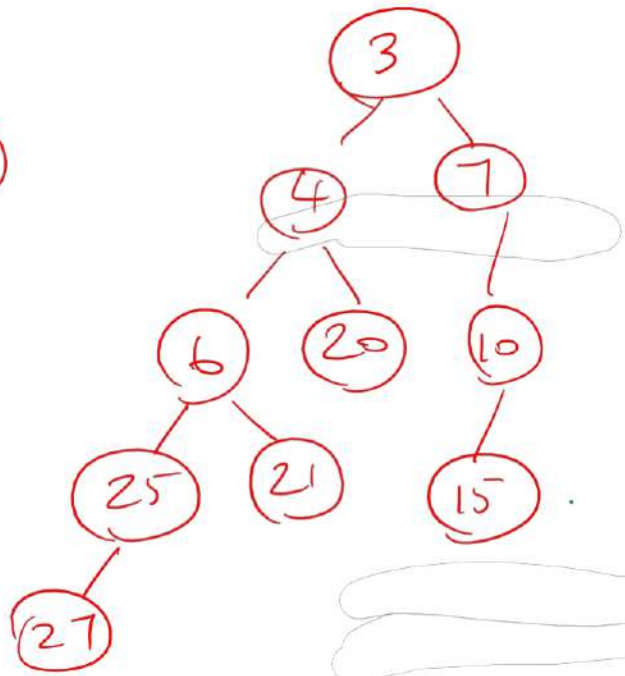
Check whether the give two trees are leftist and perform merge operations

Consider the following two binary trees:

Tree A:

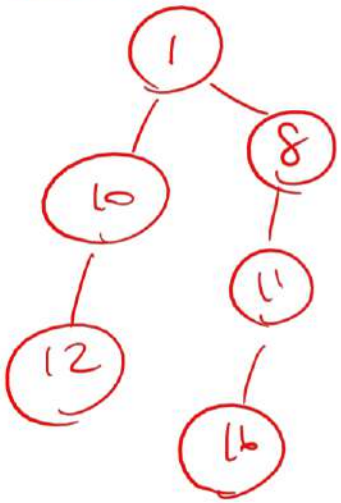


Tree B:

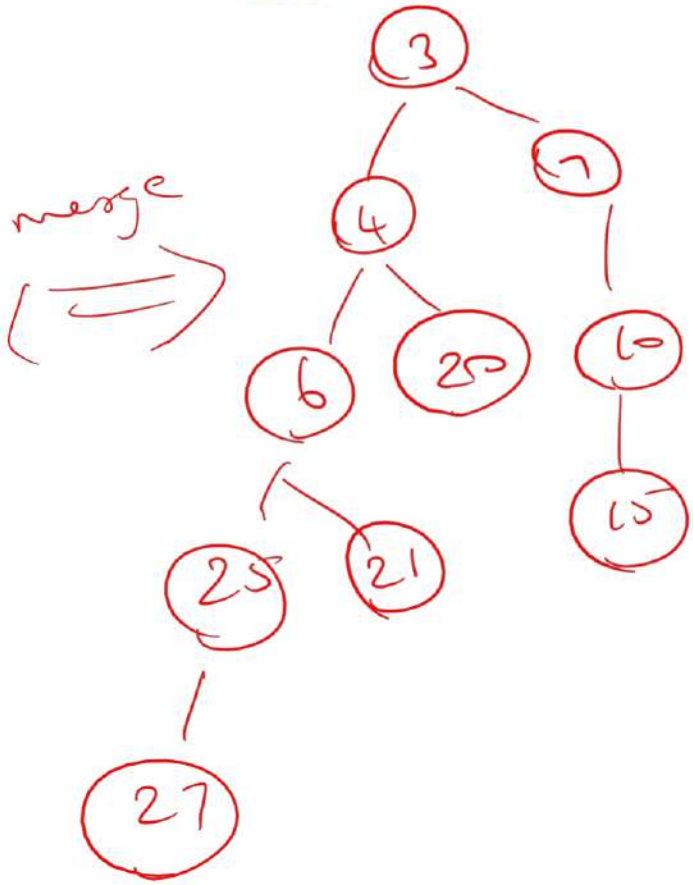


Leftist trees-merge (min heap)

Tree A :



Tree B :

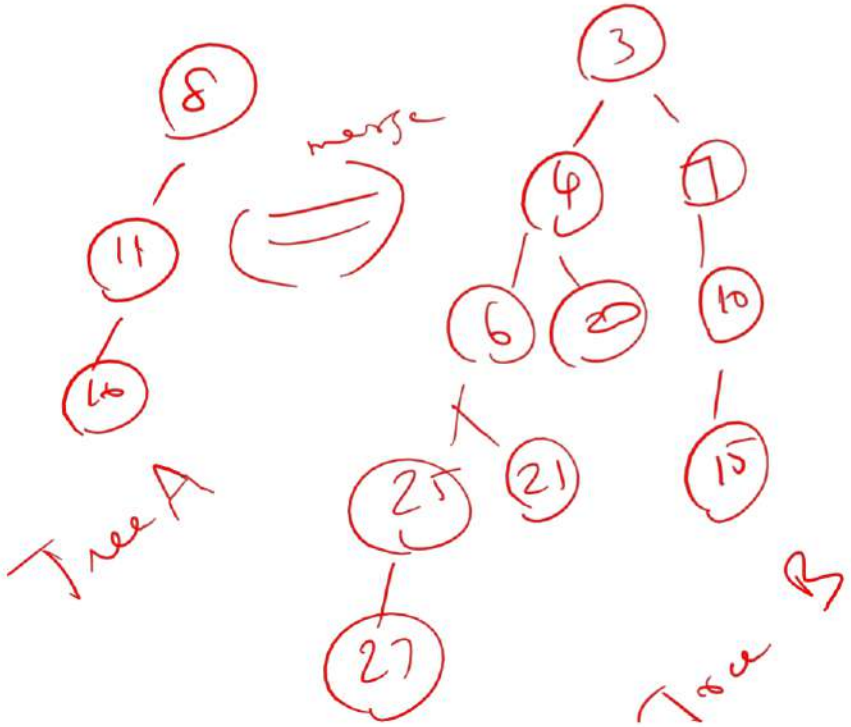


merge

note: Ensure that they are leftist heap

Leftist trees-merge

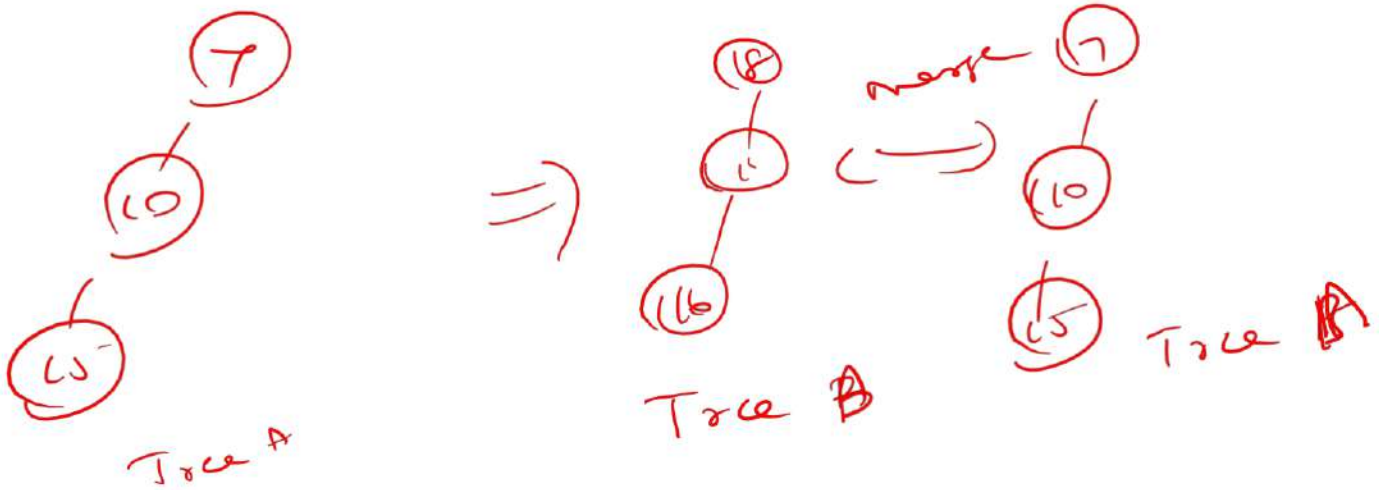
Find the tree with smallest root (tree A) and take it's right subtree and merge with tree B



Repeat the steps.

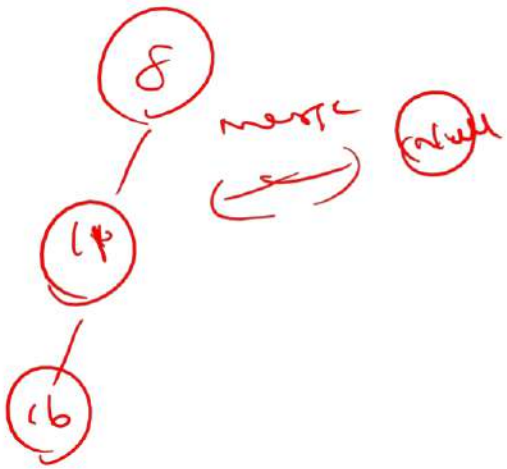
Leftist trees-merge

Take smallest root (Tree B) & take its right subtree



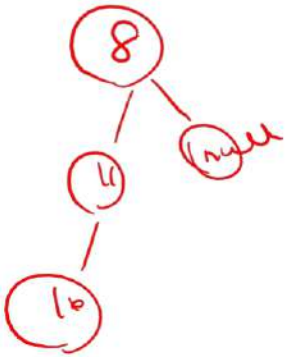
Leftist trees-merge

Take smallest root (Tree A) & it's right sub tree it's null.

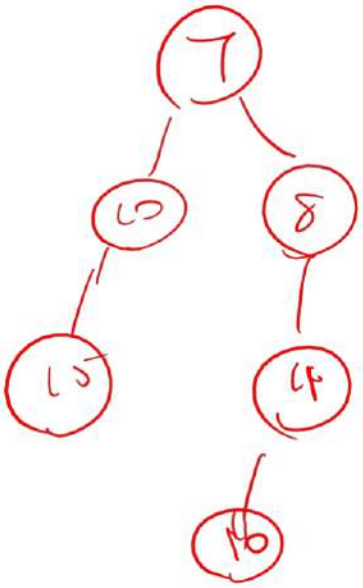


returns when null encountered

Attach it to previous tree and check leftist property

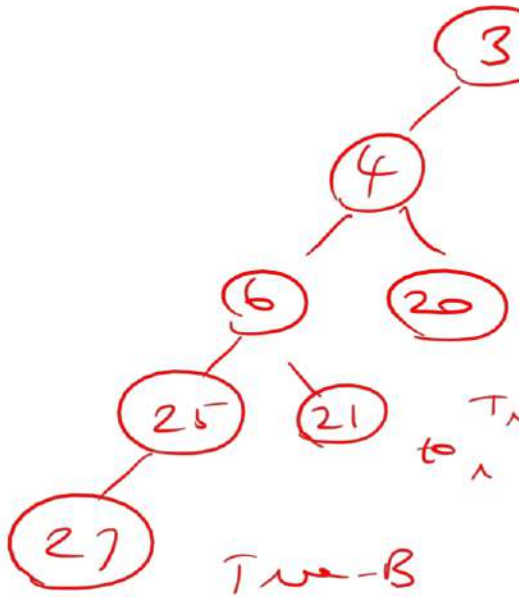


Leftist trees-merge



Tree-A

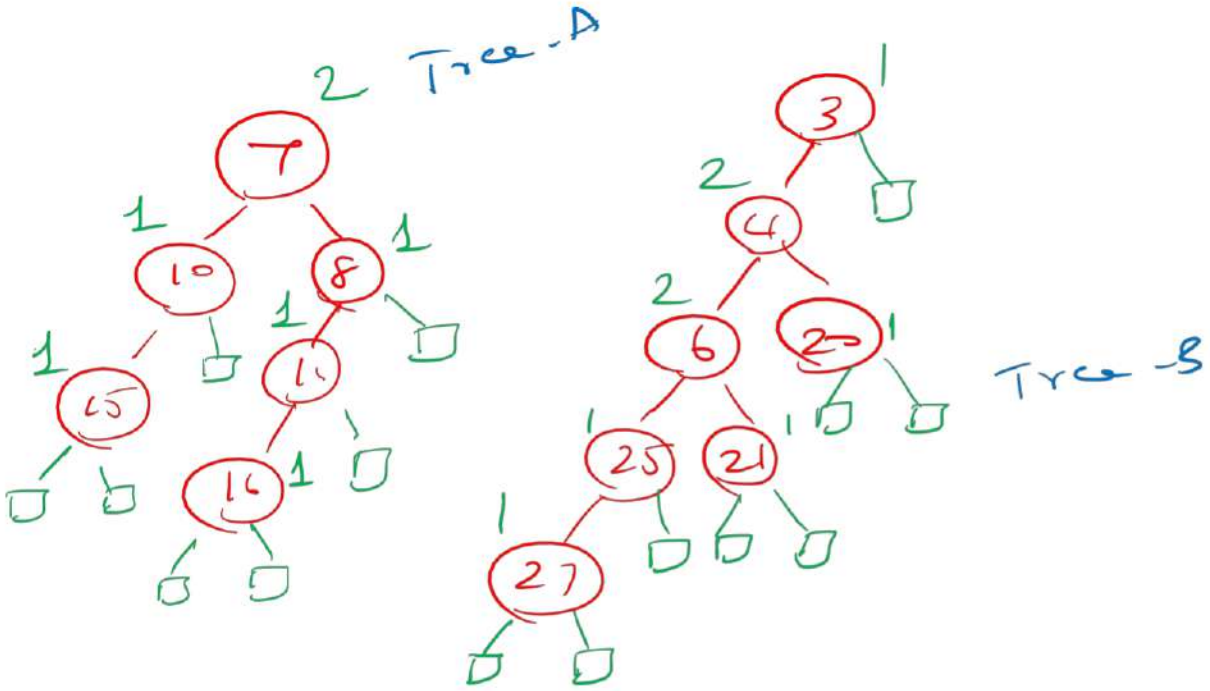
and attach it to the tree



Tree-B

but, before attaching tree A (Tree-B (to the right of 3)) to ^, calculate its S-value & also calculate Tree-B S-value from node 4

Leftist trees-merge



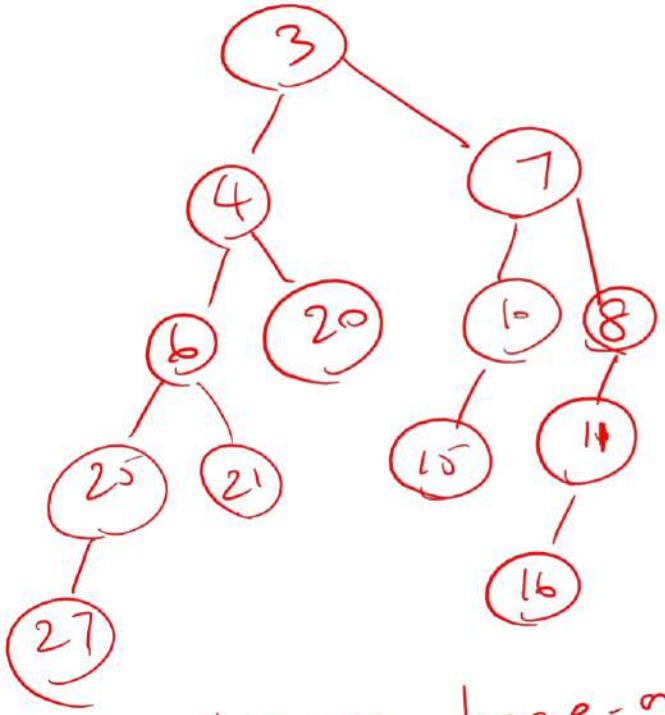
The S-Value of node (7) is 2 & node (4) is 2, here property is not violated.

($S(left(x)) \geq S(right(x))$

here attach Tree A to the right of Tree B's node (3)

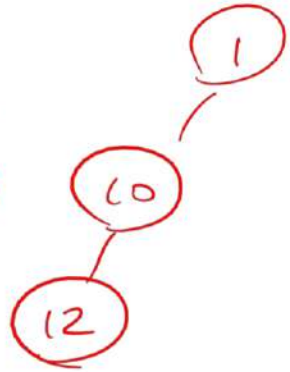
Leftist trees-merge

After merging



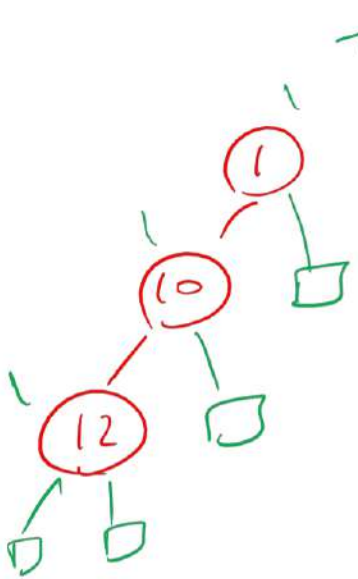
Also follows heap-order Property.

again merge it with

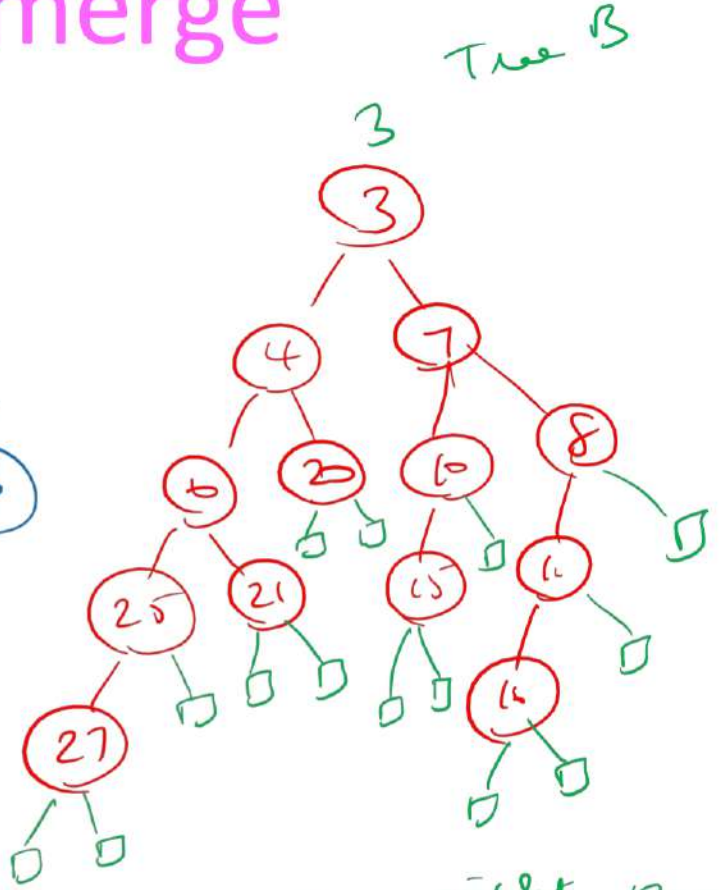


As usual, calculate s-values of both trees before merging

Leftist trees-merge



and also we know the S_value of node (3) is 3

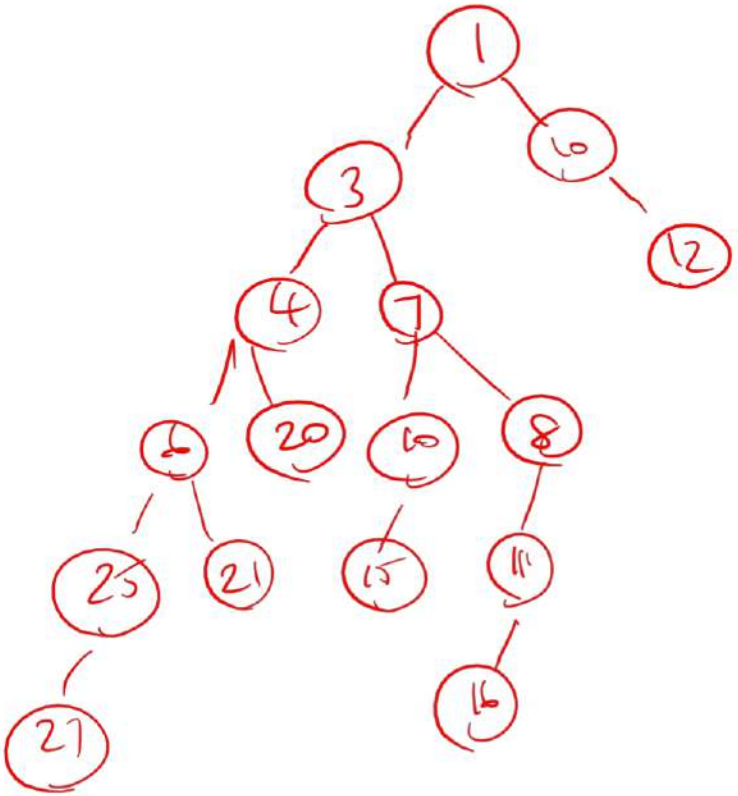


So, if we attach Tree B to the right of Tree A node (1) it will violate the property (S(left(x)) ≥ S(right(x)))

So swap the left & right child of node (1)

Leftist trees-merge

Final merged tree



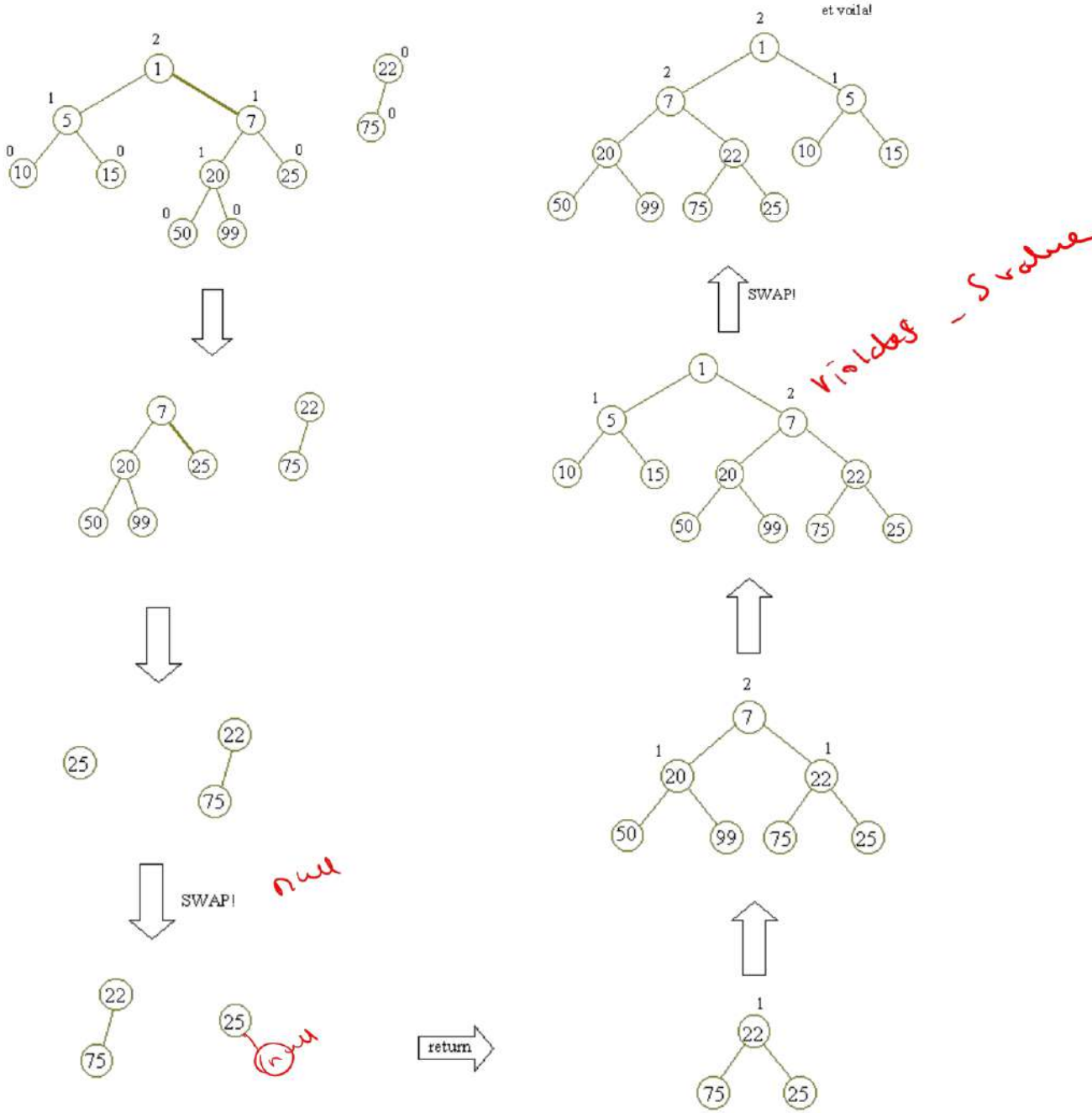
Leftist tree

- **Operations:**

Merge: The key operation in leftist trees. When merging two trees, we:

1. Compare the roots, attach the smaller (min-leftist tree) root as the new root.
2. Recursively merge the right subtrees of both trees.
3. After merging, if the leftist property is violated (left subtree's NPL (s-value) is less than the right subtree's NPL), swap the subtrees.
4. Update the NPL of the new root.

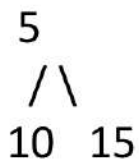
Merging has a time complexity of O(logn), which is efficient due to the tree's leftist property.


et voila!
SWAP!
SWAP!
return

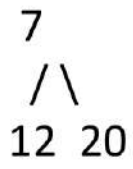
Leftist trees

- Merging the Two Min Leftist Trees

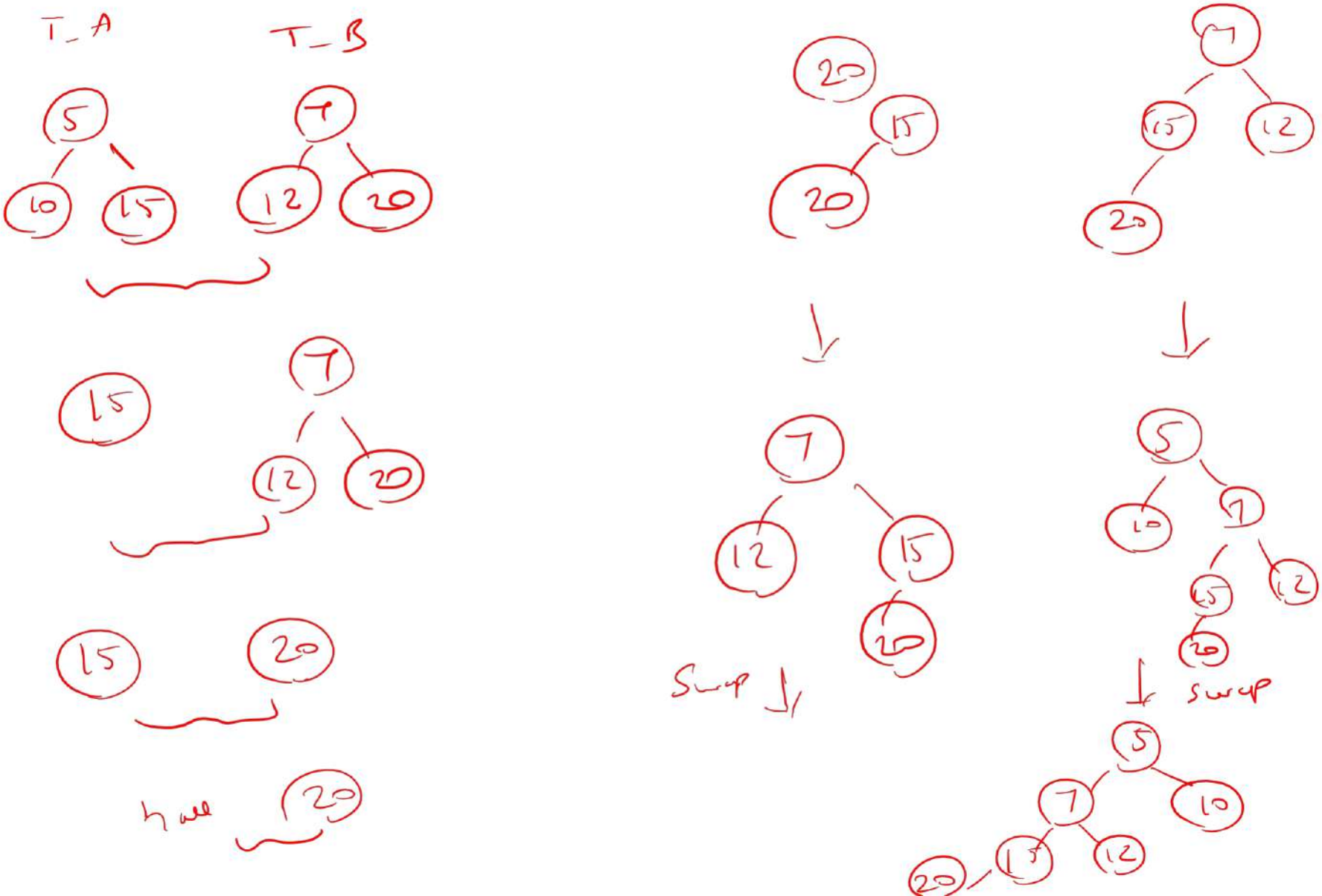
Tree 1:



Tree 2:



Leftist trees



Leftist trees

- Time complexity
  - Insert(), deleteMin(), findMin() and Merge() on a Leftist Heap is O(log n).
  - This is because the height of a Leftist Heap is always O(log n).

Leftist trees

Advantages

- **Efficient extract-min operation:** The extract-min operation has a time complexity of O(log n), making it one of the most efficient data structures for this operation.
- **Efficient merging:** The merge operation has a time complexity of O(log n), making it one of the fastest data structures for merging two binary heaps.
- **Simple implementation:** The leftist tree has a relatively simple implementation compared to other binary heap data structures, such as Fibonacci heaps.

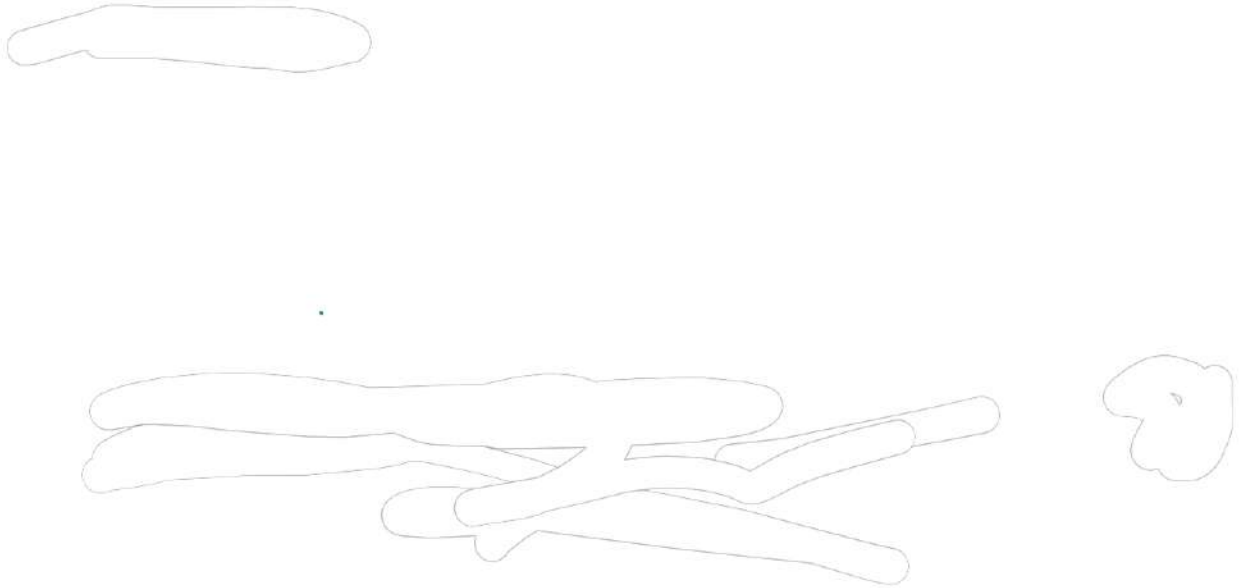
Leftist trees

Disadvantages

- **Slower insert operation:** The insert operation in a leftist tree has a time complexity of O(log n), making it slower than other binary heap data structures, such as binary heaps.
- **Increased memory usage:** The leftist tree uses more memory than other binary heap data structures, such as binary heaps, due to its requirement for the maintenance of null path length values for each node.

Leftist merge/meld

Refer to PDF for worked out example



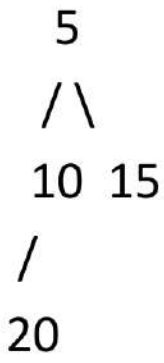
Leftist merge/meld

Exercise:

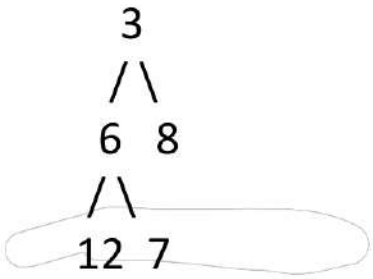
Check whether the give two trees are leftist and perform merge operations

Consider the following two binary trees:

Tree A:

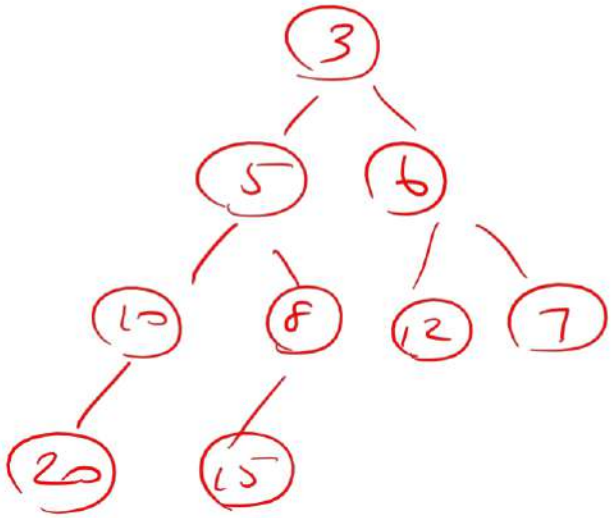


Tree B:



Leftist merge/meld

Final answer



?

Leftist merge/meld

1. Determine if each tree is a leftist tree. For each tree, verify if it satisfies the leftist property and heap order property (min-heap). Specifically, check if:

- The left subtree of every node has an NPL (null path length) greater than or equal to the NPL of the right subtree.
- The tree maintains the min-heap property (depending on the type of leftist tree).

2. Merge the two trees into a single leftist tree. Assume that both trees are min-leftist trees. Provide the resulting merged tree.

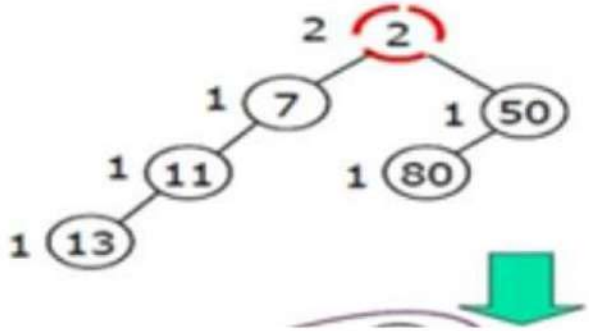
Leftist merge/meld

Exercise:

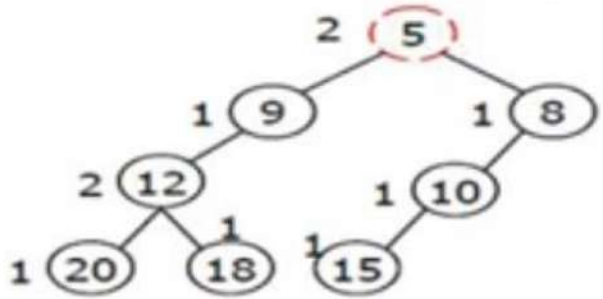
Check whether the give two trees are leftist and perform merge operations

Consider the following two binary trees:

Tree A:

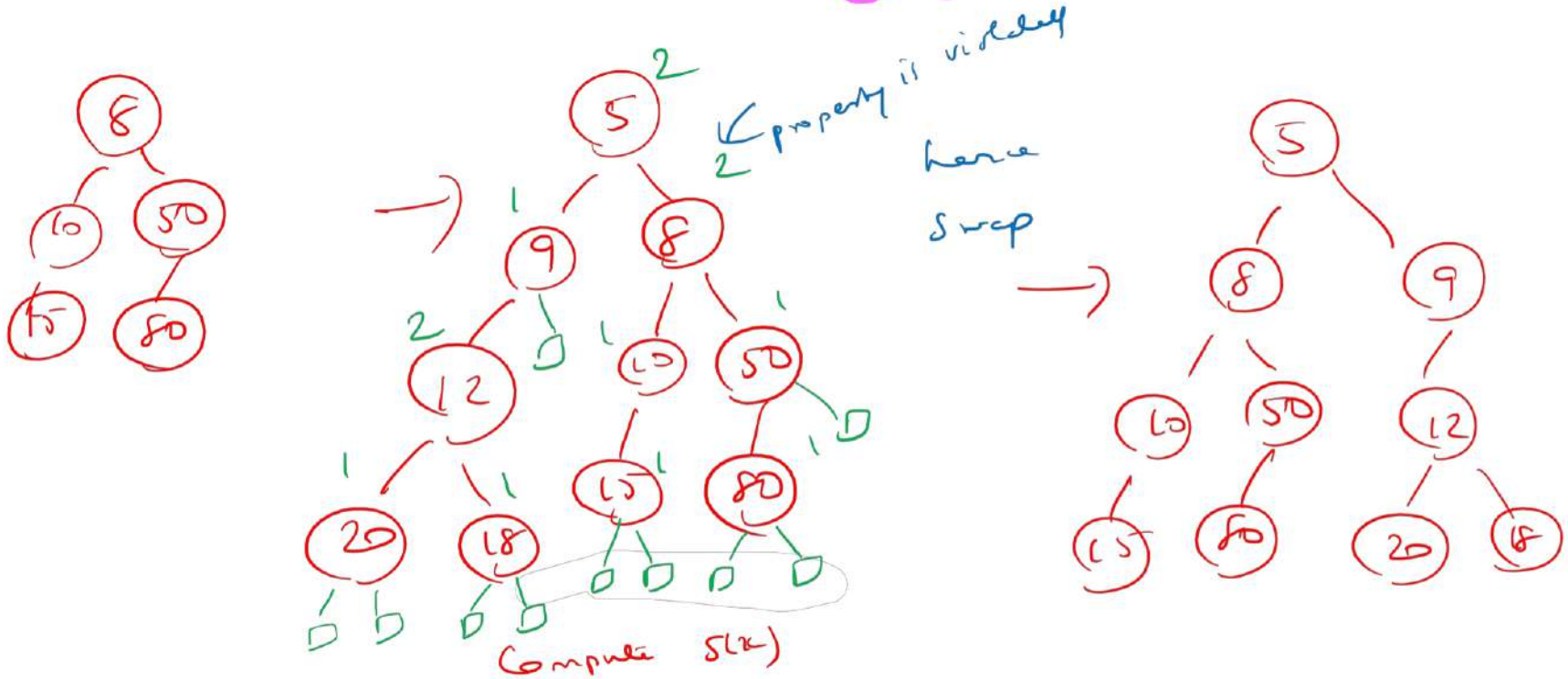


Tree B:

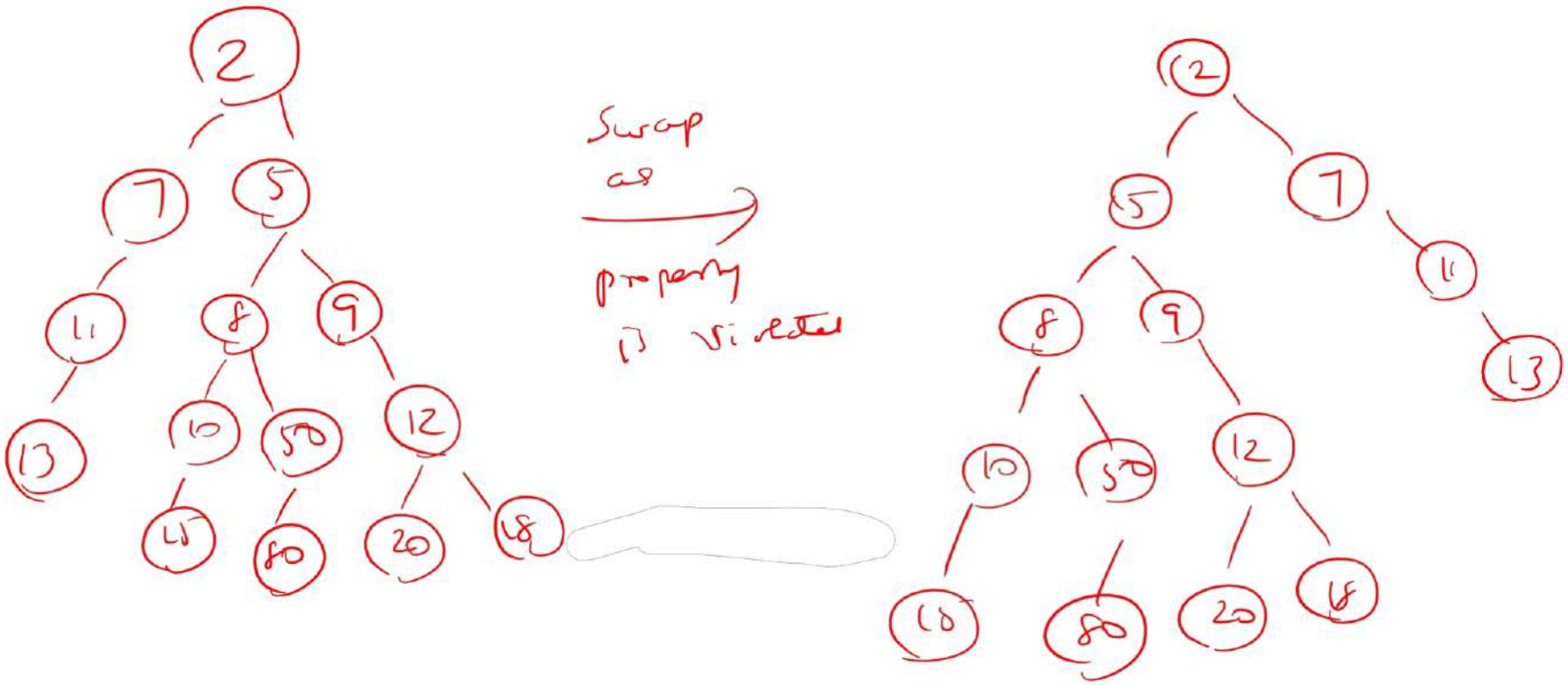


Leftist merge/meld

Leftist merge/meld



Leftist merge/meld



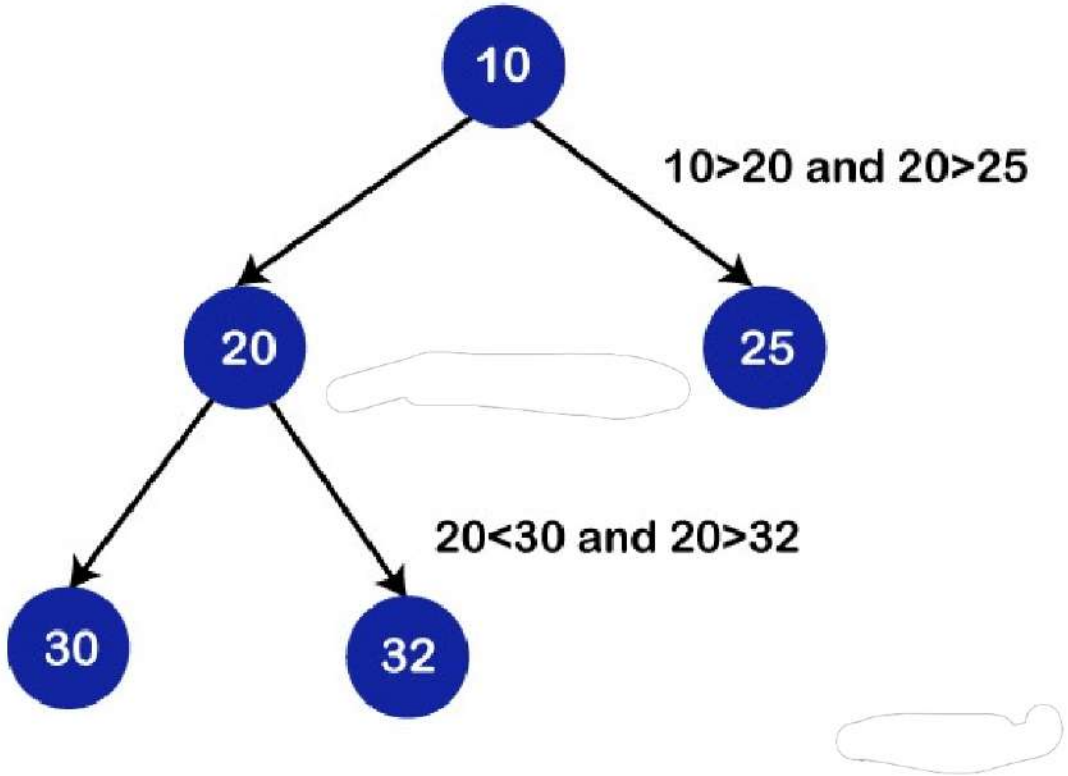
Leftist heap

Difference between Leftist Tree and Leftist Heap:

- **Leftist Tree:** Refers to the binary tree structure that maintains the leftist property but may not necessarily follow the heap order property.
- **Leftist Heap:** A specific type of leftist tree that also maintains the heap property, typically a min-heap, where the root is the smallest element, and each node is smaller than its children.

Leftist heap

It's a leftist satisfying the property, S(left(x)) >=S(right(x)) and for every node i, it's value is lesser than its' descendants in case of **leftist min heap**



Tournament tree

Tournament tree

A tournament tree is a type of binary tree used to model competitions, such as sports tournaments or sorting processes.

It is especially useful for finding the minimum or maximum element in a list, often used in parallel algorithms and games.

Tournament tree

The tournament tree is an application of the heap data structure.

It is a binary heap; hence the tournament tree is also a complete binary tree and satisfies the heap-order property.

A tournament tree comes from the idea of a tournament or a competition.

If there are n players in a tournament, then the tournament tree will have n-leaf nodes which can also be called external nodes, and it'll have n-1 internal nodes.

Tournament tree

Structure:

- A tournament tree is a complete binary tree, that is most efficiently sorted by using the array-based binary tree
- Each internal node represents the winner of a match between its two children.
- The leaves of the tree represent the initial participants (or elements in an array), and the root represents the overall winner.

Tournament tree

Levels:

- The tree has $\log_2(n)$ levels if there are n participants.
- Each level represents a round in the tournament.
- The height of the tree is O(log n).

Tournament tree

Winner Tree:

- The tree is built by pairing elements and comparing them to determine the winner, which moves up to the next round (internal node). The overall winner is found at the root of the tree.

Loser Tree:

- A variation where the tree keeps track of the losers instead of the winners. This can be used to efficiently find the second smallest (or largest) element.

Tournament tree

Applications:

1. Sorting:

Tournament trees can be used to implement selection sort, where the smallest element is repeatedly selected and removed from the tree.

2. Parallel Processing:

In parallel algorithms, tournament trees can be used to reduce the number of comparisons needed to find the minimum or maximum element.

Tournament tree

Example:
Given the elements [7, 3, 5, 2, 6, 8, 1, 4], a tournament tree would compare these elements in pairs:

- **First round:** (7 vs 3), (5 vs 2), (6 vs 8), (1 vs 4).

- **Second round:** Winners from the first round are compared: (3 vs 2), (6 vs 4).

- **Final round:** The winners from the second round are compared to find the overall winner.

Tournament tree

Building the Tree:

- The tree is built from the bottom up, starting with the leaves. Each match outcome is recorded in an internal node, and the winner advances to the next round.

Updating the Tree:

- If there's a change in the result of a match, only the affected nodes (from the leaf to the root) need to be updated, which makes it efficient.

Querying the Tree:

- You can determine the winner of the tournament by simply reading the root node.
- To determine the outcome of specific matches, you trace the path from the leaf to the root.

Tournament tree

Time Complexity:

- Building the tournament tree takes O(n), and finding the winner takes O(log n).

- Finding the second-best element can also be done in O(log n) by tracing the path from the root to the leaf.

Tournament tree

Difference from Other Trees:

- Unlike binary search trees, tournament trees are primarily focused on finding maximum or minimum elements rather than supporting operations like search, insertion, and deletion efficiently.
- This structure is essential in algorithms that require repeated selection of the smallest or largest element, providing an efficient and organized way to handle such tasks.

Tournament tree

- A tournament tree is a form of complete binary tree in which each node denotes a player.

- Tournament tree is a complete binary tree with n external nodes and n – 1 internal nodes. The external nodes represent the players, and the internal nodes are representing the winner of the match between the two players.

- It is also referred to as a Selection tree.

Tournament tree

Properties:

1. The value of every internal node is always equal to one of its children.
2. A tournament tree can have holes. The tournament tree having nodes less than 2 ^ (n+1) -1 contains holes. The hole represents a player or team's absence and can be anywhere in the tree. Trees with a number of nodes not a power of 2, contain holes.
3. Every node in a tournament tree is linked to its predecessor and successor, and unique paths exist between all nodes.
4. It is a type of binary heap (min or max heap). Or we can say it is the application of heaps.
5. The root node represents the winner of the tournament.
6. To find the winner or loser of the match, we need N-1 comparisons.

Tournament tree

Types of tournament trees:

There exist a loser and a winner in every match. So, there are two methods to represent both ideas:

1. Winner Tree
2. Loser Tree

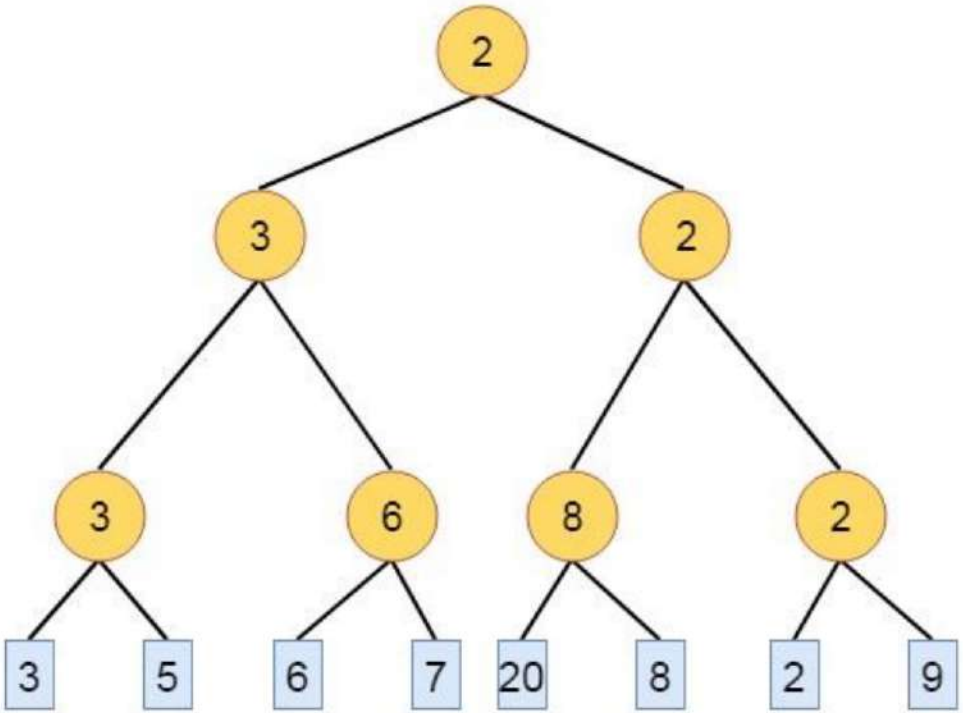
Tournament tree

Winner tree

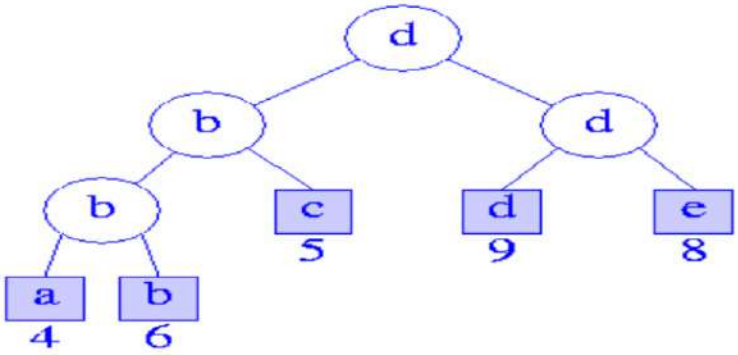
- In a tournament tree, when the internal nodes represent the winner of the match, the tree obtained is referred to as the winner tree.
- Each internal node stores either the smallest or greatest of its children, depending on the winning criteria.
- When the winner is the smaller value then the winner tree is referred to as the ***minimum winner tree (looks like min heap),*** and when the winner is the larger value, then the tree is referred to as the ***maximum winner tree (max heap)***.

Tournament tree

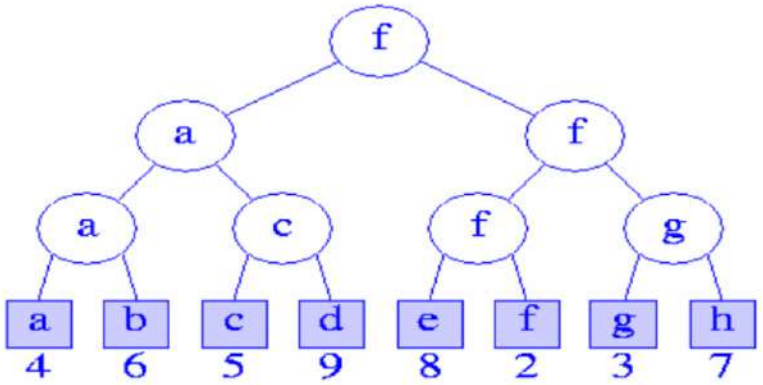
- **Example** – Suppose there are some keys, 3, 5, 6, 7, 20, 8, 2, 9



Tournament tree



The player with the larger value wins



The player with the smaller value wins

Tournament tree

The tournament's winner is always the smallest or the greatest of all the players or values and can be found in O(1).

Tournament tree

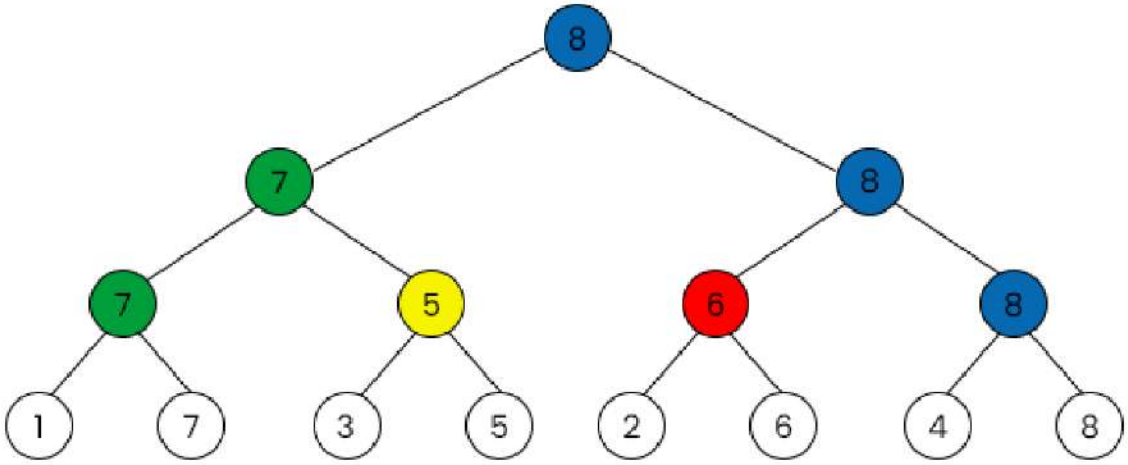
Qn: Eight players participate in a tournament where a number represents each player, from 1 to 8. The pairings of these players are given below:

- Group A: 1, 3, 5, 7 (Matches: 1 - 7 and 3 - 5)
- Group B: 2, 4, 6, 8 (Matches: 2 - 6, and 4 - 8)
- Winning Criteria - The player having the largest value wins the match. Represent the winner tree (maximum winner tree) for this tournament tree.

Tournament tree

The winner tree (maximum winner tree) is represented in the figure:

Here, the tree formed is a max heap, the value of all the parents is either equal to or larger than its children, and the winner is 8, which is the largest of all the players.



Maximum winner Tree

We can observe from this that every tournament tree is a binary tree, but the reverse is not true because, in the tournament tree, the winner is always equal to one of its children, but in binary heaps, the parent is not always equal to its children.

Tournament tree

Tournament tree Vs Heap:

A tournament tree will always be a heap, but a heap is not always a tournament tree because, in a tournament tree, the value of each node must be the value of its left or right child, but in a heap, a node can have any value lesser than it's left and right child.

Tournament tree

loser tree

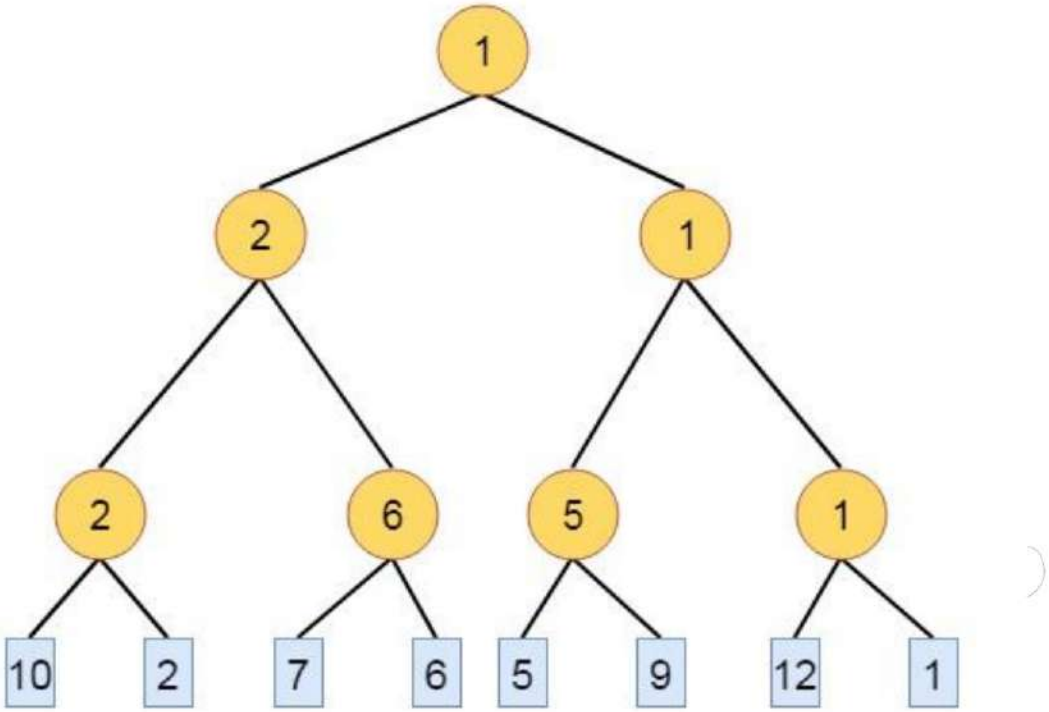
- In a tournament tree, when the internal nodes are used to represent the loser of the match between two, then the tree obtained is referred to as the loser tree. When the loser is the smaller value then the loser tree is referred to as the ***minimum loser tree,*** and when the loser is the larger value, then the loser tree is referred to as the ***maximum loser tree***.

- But in this overall winner is stored at tree[0].

- It is also called the minimum or maximum loser tree. The same idea is also applied here, the loser (or parent) is always equal to one of its children, and the loser is always the greatest or smallest of all the players and can be found in O(1).

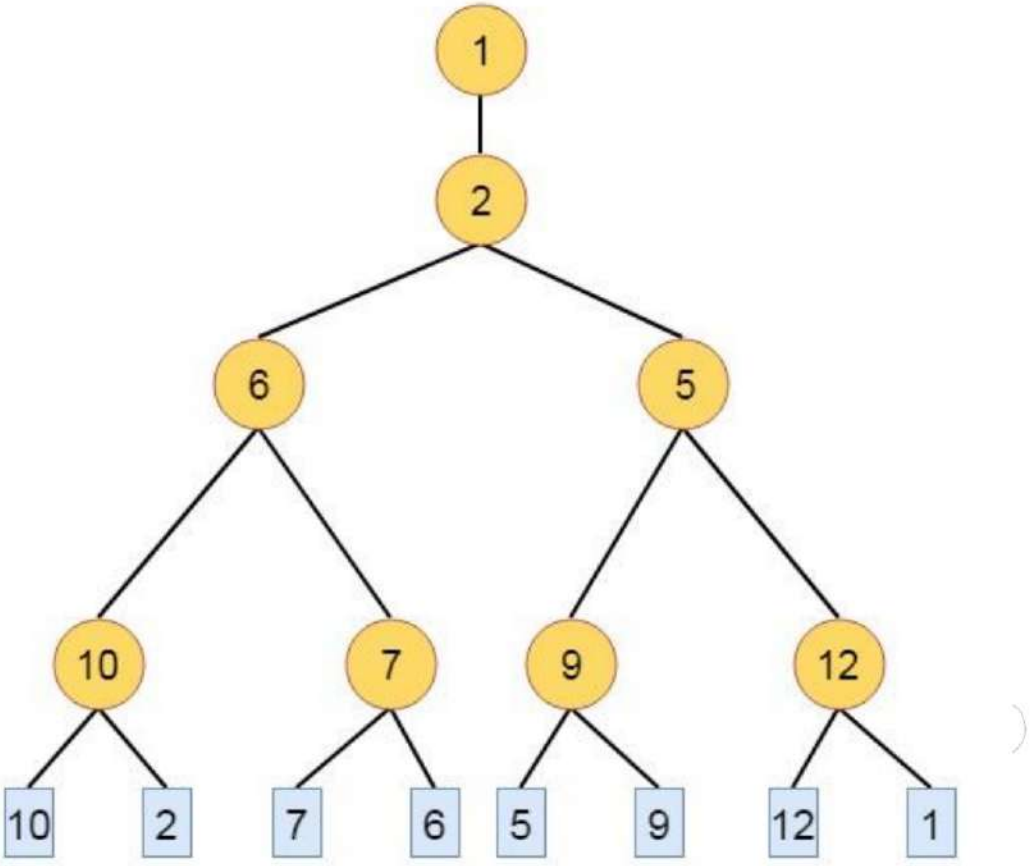
Tournament tree

- **Example** – To form a looser tree, we have to create winner tree at first.
- Suppose there are some keys, 10, 2, 7, 6, 5, 9, 12, 1. So we will create minimum winner tree at first.



Tournament tree

Now, we will store looser of the match in each internal node.



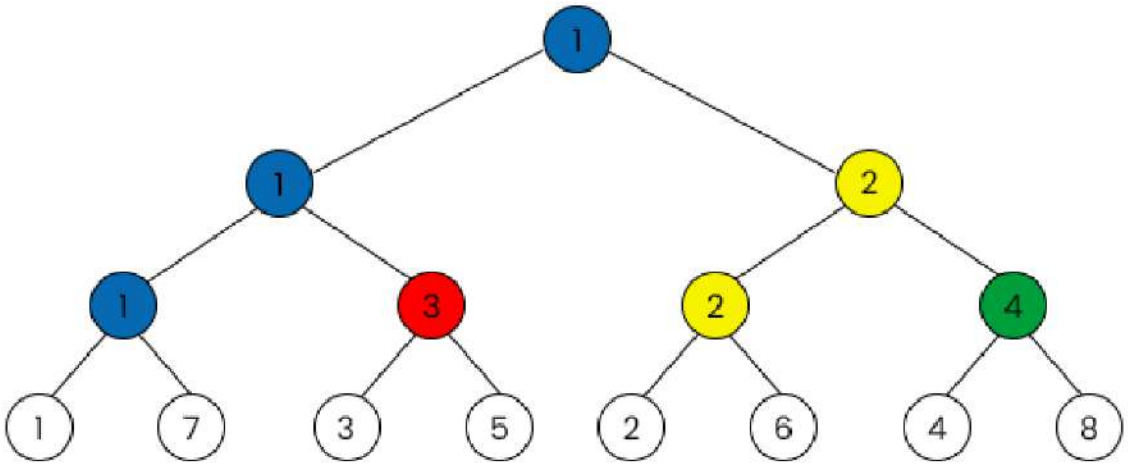
Tournament tree

Qn: Eight players participate in a tournament where a number represents each player, from 1 to 8. The pairings of these players are given below:

- Group A: 1, 3, 5, 7 (Matches: 1 - 7 and 3 - 5)
- Group B: 2, 4, 6, 8 (Matches: 2 - 6, and 4 - 8)

Tournament tree

Winning Criteria - The player having the largest value wins the match. Represent the loser tree (minimum loser tree) for this tournament tree.



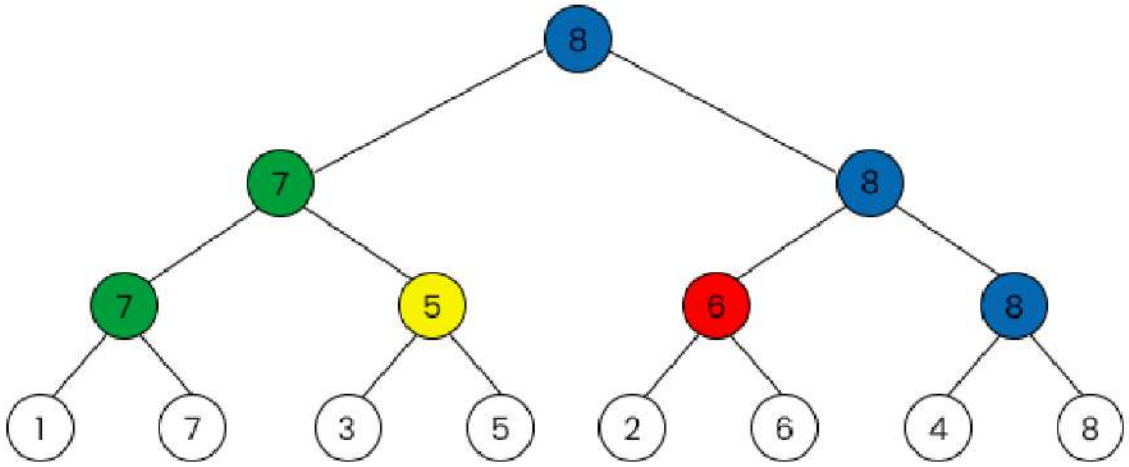
Minimum Loser Tree

Here, the tree obtained is a min-heap, the value of the parent is either smaller or equal to its children, and 1 is the overall loser of the tournament.

Tournament tree

Finding the second-best player:

- In the winner tree, the winner is found at the top of the tree. Now, the problem is to find the second-best player in the tournament. Here, we need to note that the second-best player is only beaten by the best player. To find the second-best player, we need to check the last two games (or comparison) with the best player. Let us see the above idea with an example.



Maximum winner Tree

Here, 8 is the tournament winner, and when we look for the second-best player, we find 7 is the second-best player.

Construction Steps:

Initialization:

- Create a list of leaf nodes where each leaf represents a player or competitor.

Build Tree from Leaves:

- While there are more than one node in the list:

    1. **Pair Nodes**:
        - Group the nodes in pairs. If there's an odd number of nodes, the last node advances automatically.
    2. **Create Matches**:
        - For each pair of nodes, create a new internal node to represent the match. The winner of this match is determined by the comparison or result.
    3. **Update List**:
        - Replace the pairs of nodes with the new internal nodes created from the matches. If there was an odd number of nodes, include the unpaired node in the next round.
    4. **Repeat**:
        - Repeat the pairing, matching, and updating steps until only one node remains, which will be the root of the tournament tree.

Construct Tree:

- The final single node is the root of the tournament tree. It contains the winner of the tournament.

Tournament tree

In the context of **tournament trees**, **holes** refer to unutilized positions within the tree structure when the number of participants (or elements) isn't a power of two.

When this occurs, certain internal nodes may not have a full set of children, leading to "holes" in the tree.

Example Dataset:

Consider a dataset with 5 elements:

[7, 3, 5, 2, 6]

Tournament tree

1. Building the Winner Tree

- Given that the number of elements is 5, which isn't a power of 2, the tournament tree will have some unutilized spots (holes).

Steps:

- **Level 1** (Leaves):

[7, 3, 5, 2, 6]

Since a complete binary tree with 5 leaves needs to fill up to the nearest power of 2, we'll consider the next power of 2, which is 8. We need 3 additional "holes" to complete the tree.

Level 2:

Compare the elements:

- Compare 7 vs 3 → Winner: 7
- Compare 5 vs 2 → Winner: 5
- 6 has no direct pair to compete with, so it moves up.
- The resulting winners at Level 2: [7, 5, 6, hole]

Tournament tree

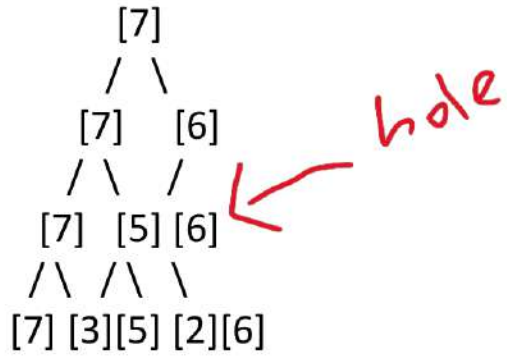
Level 3:

- Compare the winners:
  - Compare 7 vs 5 → Winner: 7
  - 6 automatically moves up.
- The resulting winners at Level 3: [7, 6]

Final Round:

- Compare 7 vs 6 → Winner: 7

- The Winner Tree looks like this:



Tournament tree

Presence of Holes:

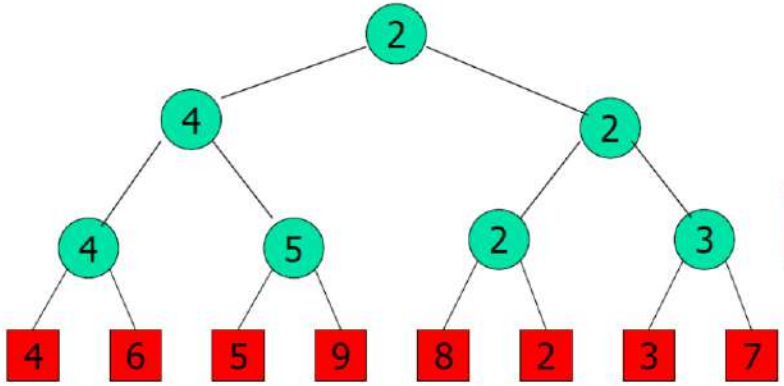
- Notice that one position in the tree is a "hole" because there weren't enough elements to fill every slot.
- The presence of this hole occurs at the second last level, where 6 moves up without a competitor.

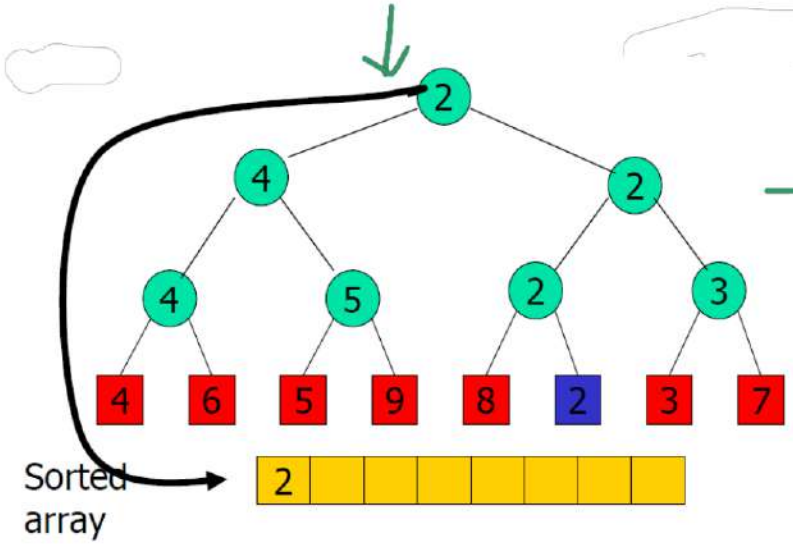
Implications of Holes

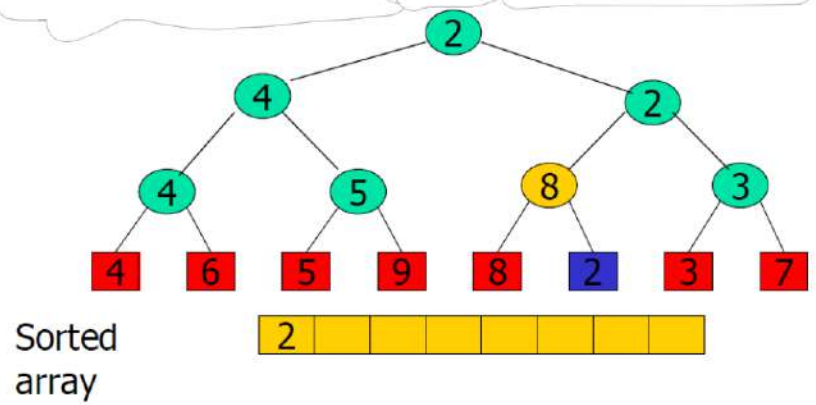
- **Incomplete Matches**: Holes result in elements moving up the tree without competition, which can slightly skew the process by which the winner is determined.
- **Efficiency**: While the presence of holes can simplify the tree's construction (since some elements advance without competition), they may also introduce inefficiencies in certain algorithms that rely on full competition between elements.

Tournament tree-Sorting

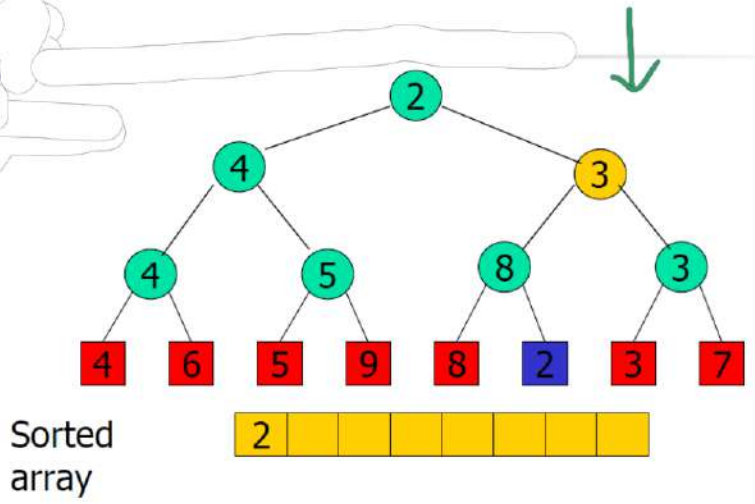
Min Winner Tree



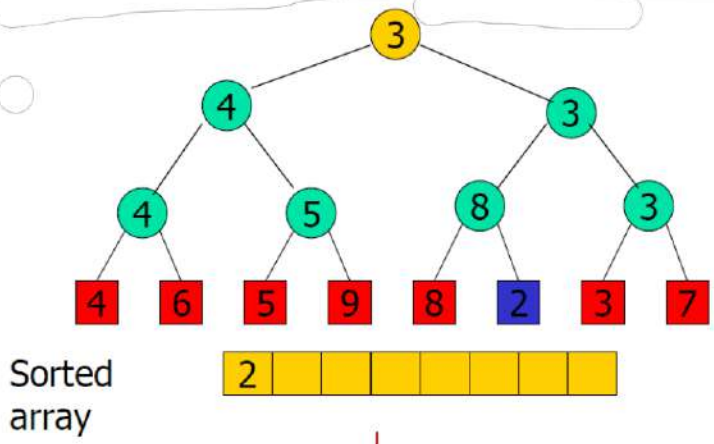


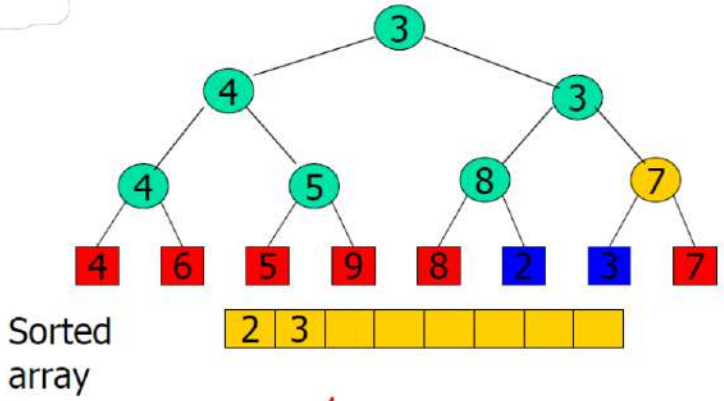


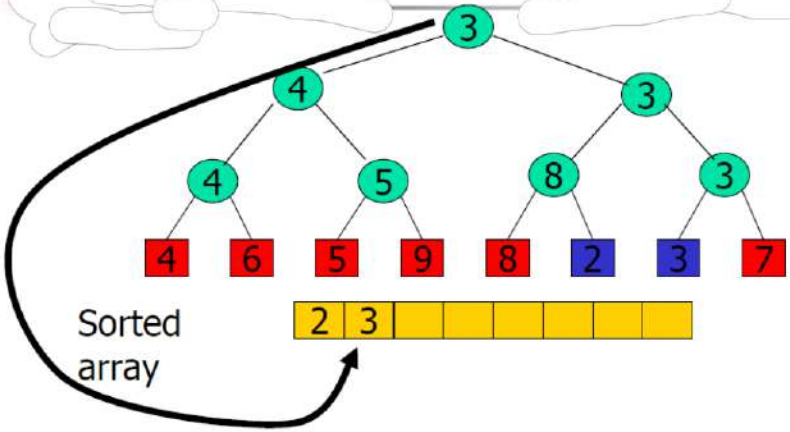
- Restructuring starts at the place where "2" is removed

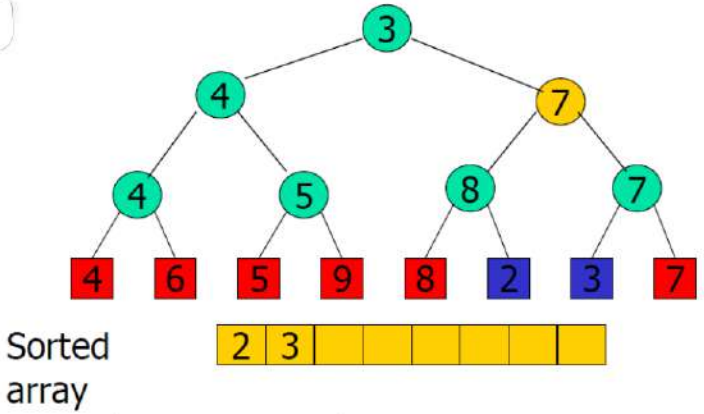


Tournament tree



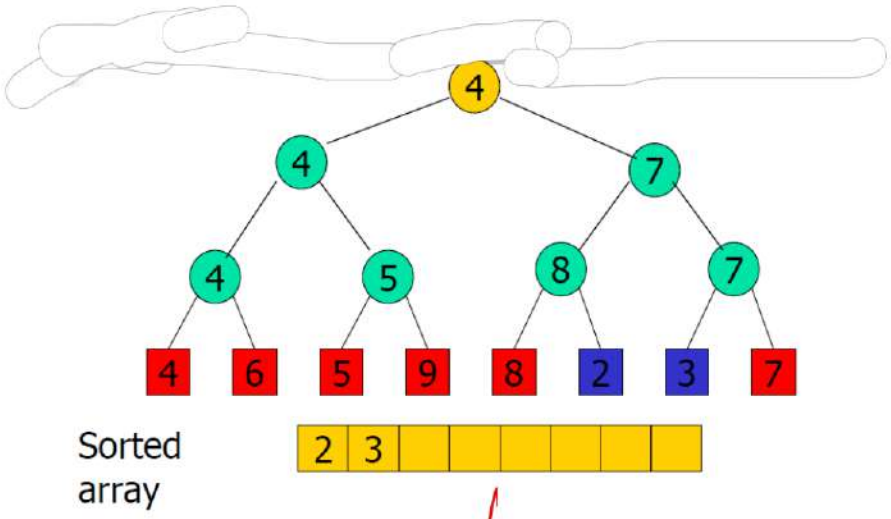


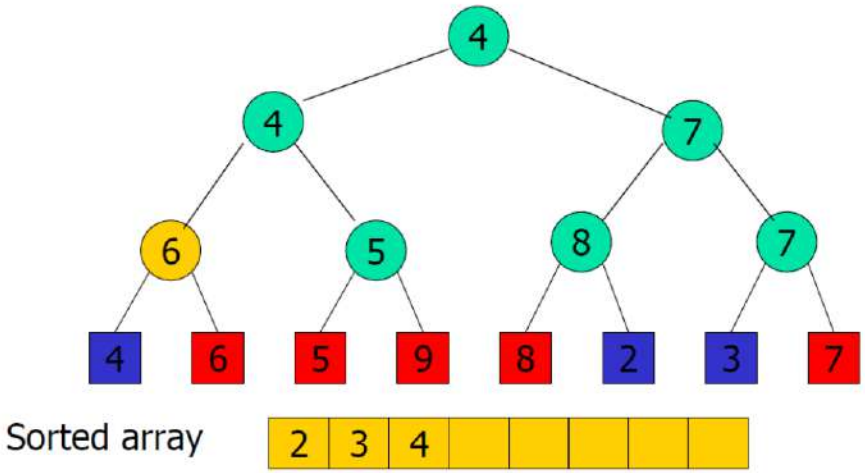


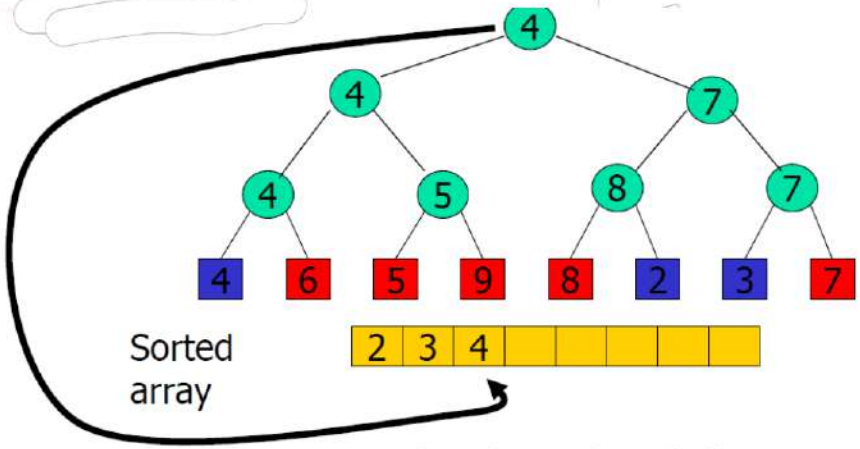


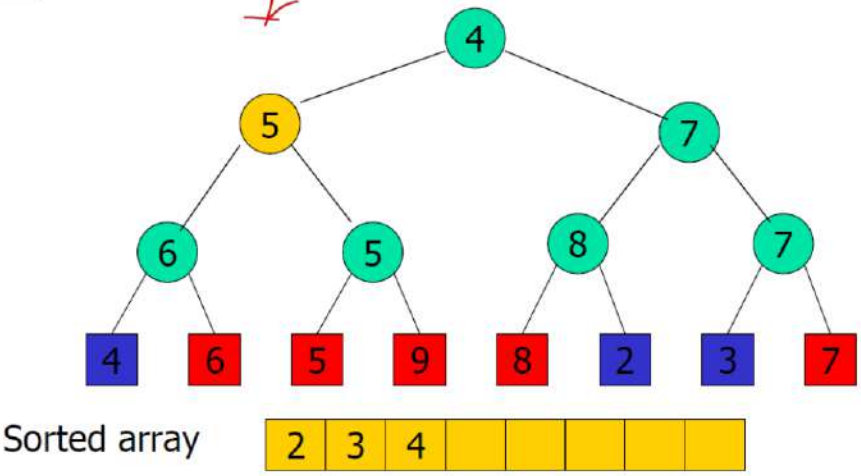
- Restructuring starts at the place where "3" is removed

Tournament tree



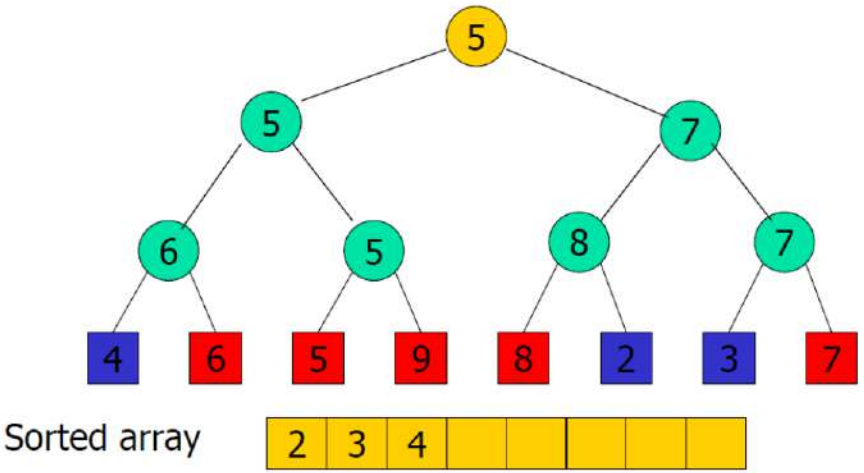






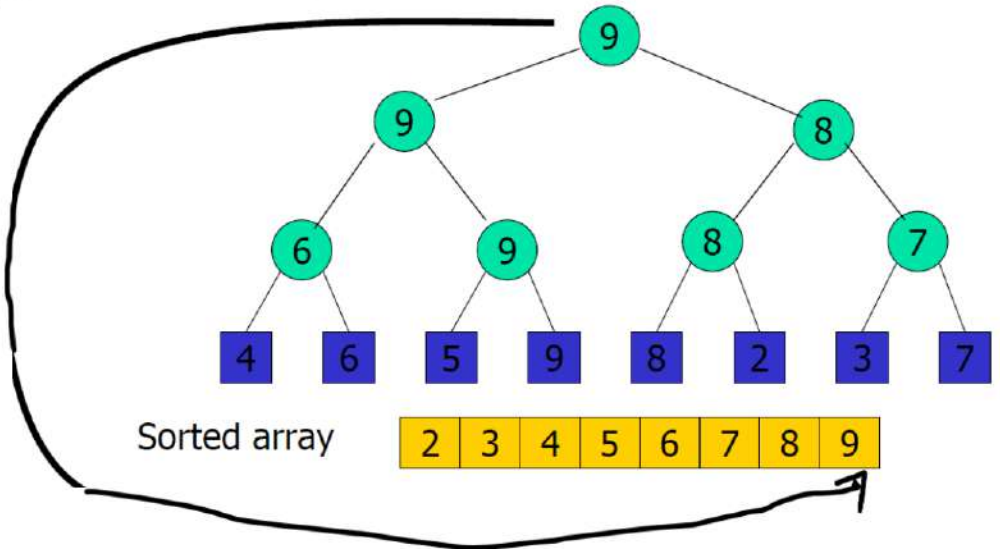
- Restructuring starts at the place where "4" is removed

Tournament tree



Continue the process until all elements are present in the array in sorted order

The final tree and array



2-3 tree

2-3 tree

- A **2-3 tree** is a height balanced binary search tree used to maintain sorted data in a way that allows for efficient search, insertion, and deletion operations.
- It is a type of simple example of a **B-tree of order 3** and is particularly useful because it maintains balance automatically.
- Not a Binary tree

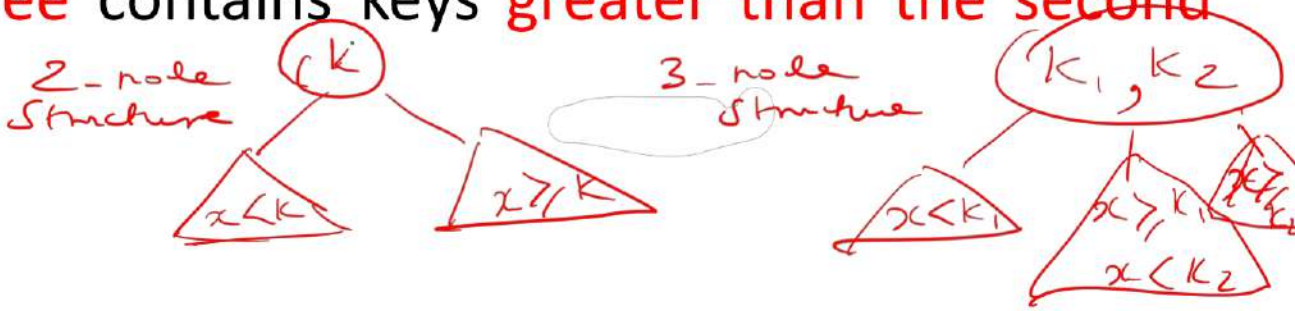
Why 2-3 tree

- Like other Trees such as AVL trees, Red Black Tree, B tree, **2-3 Tree is a height balanced tree.**
- Easier to balance the height
- Easier access than Binary Search Trees
- Used in file systems and databases

2-3 tree

Structure of a 2-3 Tree

- **2-nodes**: A node with two children and one data element. It divides the children into two subtrees, each subtree holding keys either all less than or all greater than the node's key.
- **3-nodes**: A node with three children and two data elements. It divides the children into three subtrees: (BST property)
  - The left subtree contains keys less than the first element.
  - The middle subtree contains keys between the first and second elements.
  - The right subtree contains keys greater than the second element.

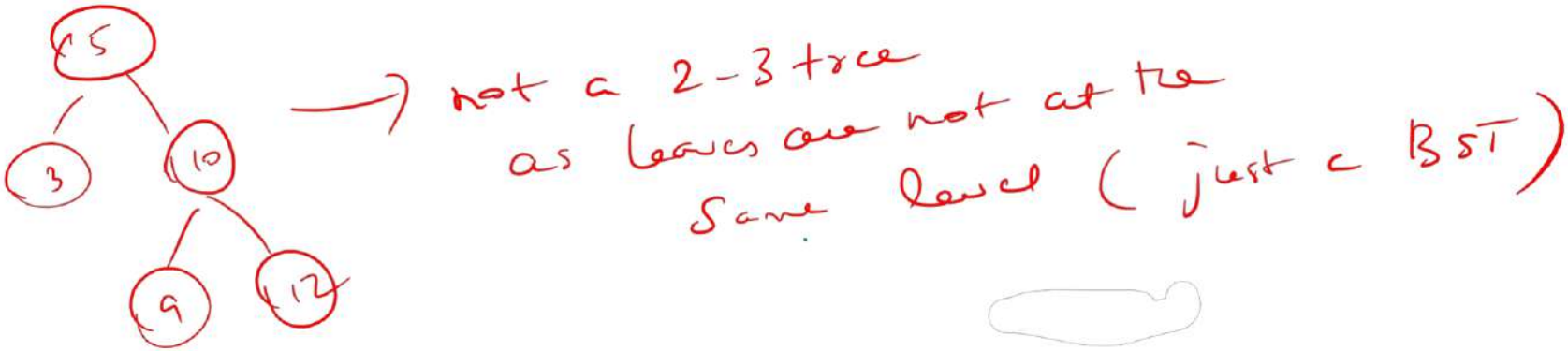


2-3 tree

Properties

1. **Balanced Tree** : All leaves are at the same depth/level.
2. **Node Keys** : A 2-node has one key, and a 3-node has two keys.
3. **Search Operation**: Searching in a 2-3 tree takes O(log n) time because the tree is balanced.

4. Insertion and Deletion: These operations are also performed in O(log n) time. The tree may split (if node overflows) or merge nodes (if node underflows) to maintain balance during these operations.



2-3 tree

Properties of 2-3 tree:

- Nodes with two children are called 2-nodes. The 2-nodes have one data value and two children
- Nodes with three children are called 3-nodes. The 3-nodes have two data values and three children.
- Data is stored in sorted order.
- Each node can either be leaf, 2 node, or 3 node.
- Always insertion is done at leaf.

2-3 tree

Operations:

- Search
- Insert
- Delete

2-3 tree

Insertion algorithm:

1.Start at the Root:
•Begin the insertion process at the root and traverse down the tree to the appropriate leaf node.

2.Search for the Correct Position:
•Search down the tree using comparisons to find the appropriate leaf where the new key should be inserted.
•Keep moving down until you reach a leaf node, maintaining the rule that all leaves must be at the same level.

3. Insert into a Leaf Node:
•If the leaf node is a **2-node (one key)**, insert the new key into it and transform it into a **3-node (two keys)**.
•If the leaf node is already a **3-node (two keys)**, we need to split it.

4. Splitting a 3-node:
•If the node is a **3-node**, you insert the new key, and now it has three keys temporarily.
•To handle this, split the node into two separate **2-nodes**:
    •The **middle key** becomes the parent (or is passed up to the parent node).
    •The two smaller and larger keys remain in the two nodes, maintaining the properties of the 2-3 tree.
•If this split occurs at the root, create a new root with the middle key, and the two split nodes become its children. This grows the tree in height.

5. Propagate the Split Upward (if needed):
•If the parent node is also a **3-node**, the split process propagates upward.
•Continue splitting nodes up the tree until you either reach the root or find a parent node that can accommodate the new middle key without needing to split.

6. Rebalancing:
•The tree rebalances itself through the splitting and promotion of keys. If the root splits, the tree grows by one level.

2-3 tree

Suppose we want to insert the following sequence of keys into an initially empty 2-3 tree:

10, 20, 30, 40, 50

Step 1: Insert 10

- The tree is empty initially. When we insert 10, it becomes the root of the tree as a 2-node.

[10]

2-3 tree

Step 2: Insert 20

- The current root is a 2-node with key 10. We insert 20 into this node, making it a 3-node.

[10 | 20]

2-3 tree

Step 3: Insert 30

- The current node is a 3-node with keys 10 and 20. A 3-node can't hold more than two keys, so we need to split this node.
- The middle key, 20, moves up to become the new root.
- The left child becomes the 2-node [10], and the right child becomes the 2-node [30].

[10 | 20 |30]

[20]

**/ **

[10] [30]

2-3 tree

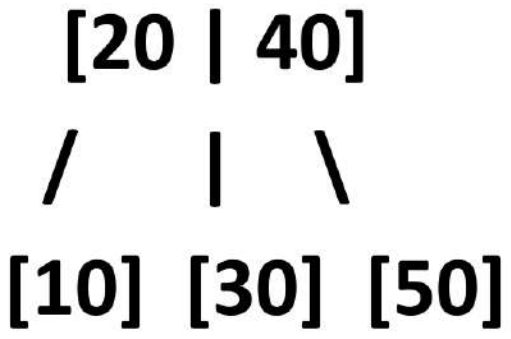
Step 4: Insert 40

- Start at the root. Since 40 is greater than 20, we move to the right child, which is a 2-node [30].
- Insert 40 into this 2-node to make it a 3-node [30 | 40].

2-3 tree

Step 5: Insert 50

- Start at the root. Since 50 is greater than 20, we move to the right child [30 | 40].
- The right child is a 3-node, so inserting 50 will require splitting this node.
- The middle key, 40, is pushed up to the root [20], making the root a 3-node [20 | 40].
- The left child of 40 becomes [30], and the right child becomes [50].

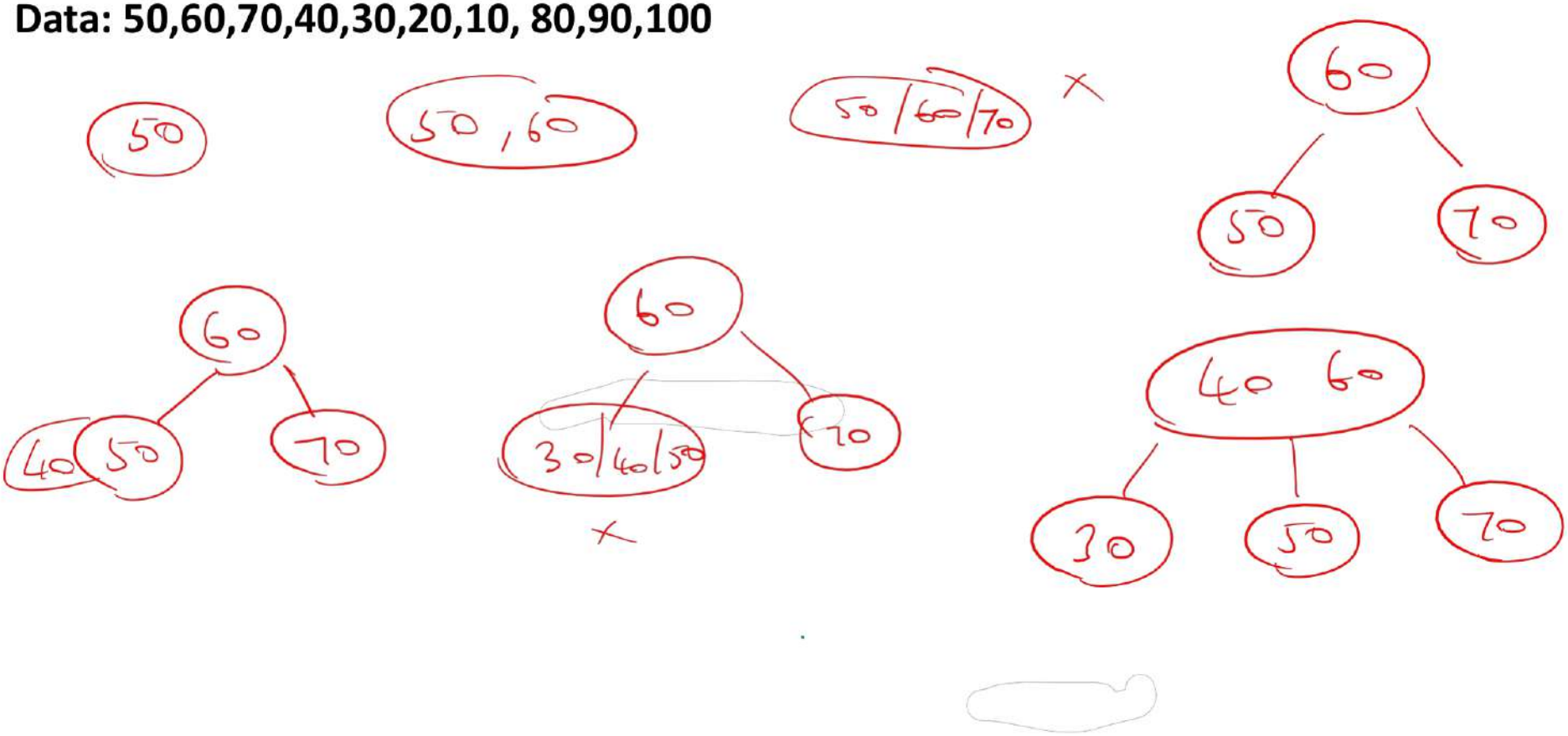


Each insertion either adds a key to an existing node or splits a node if it overflows, maintaining the balanced structure of the 2-3 tree throughout the process.

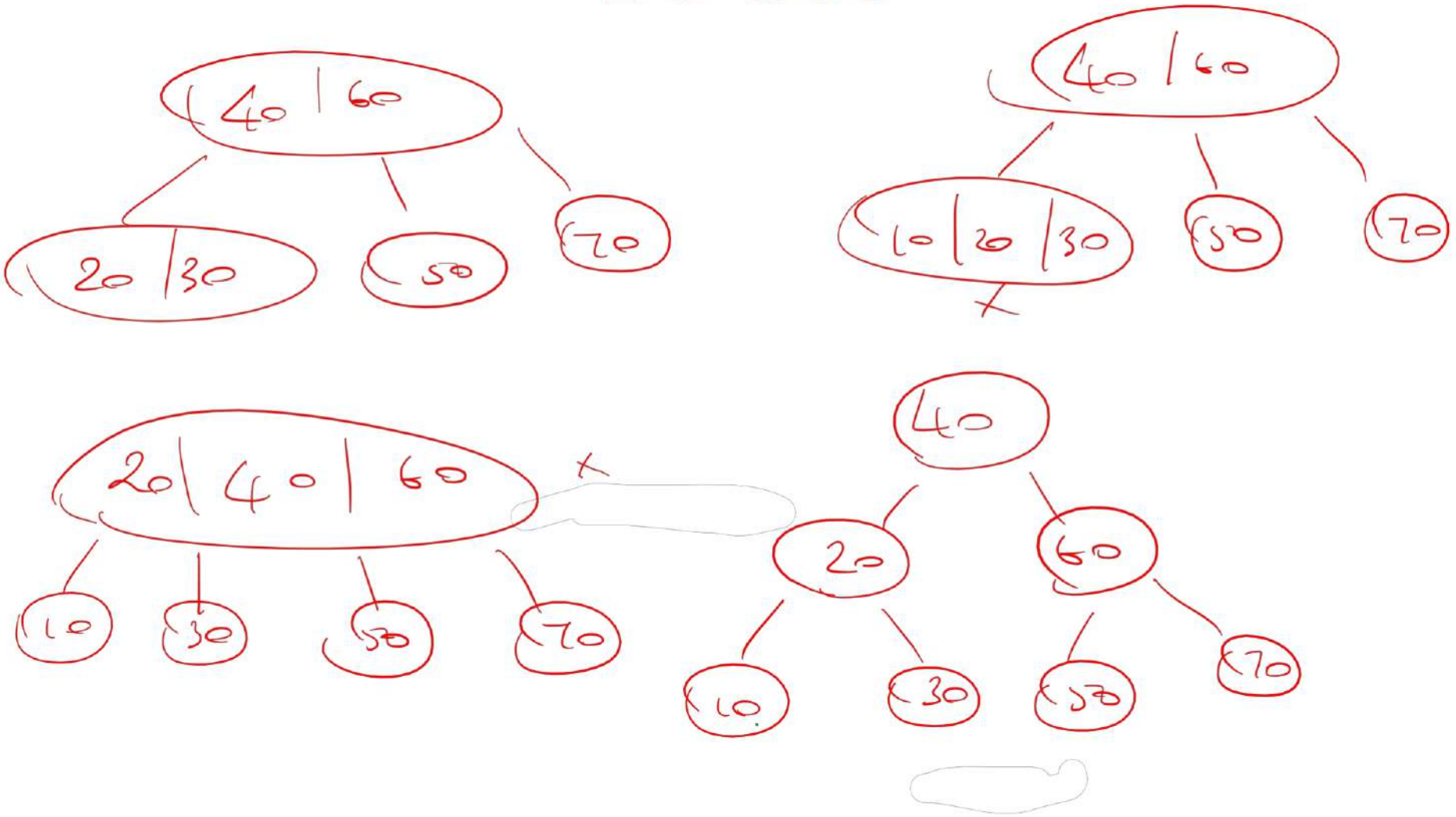
2-3 tree

Insertion:

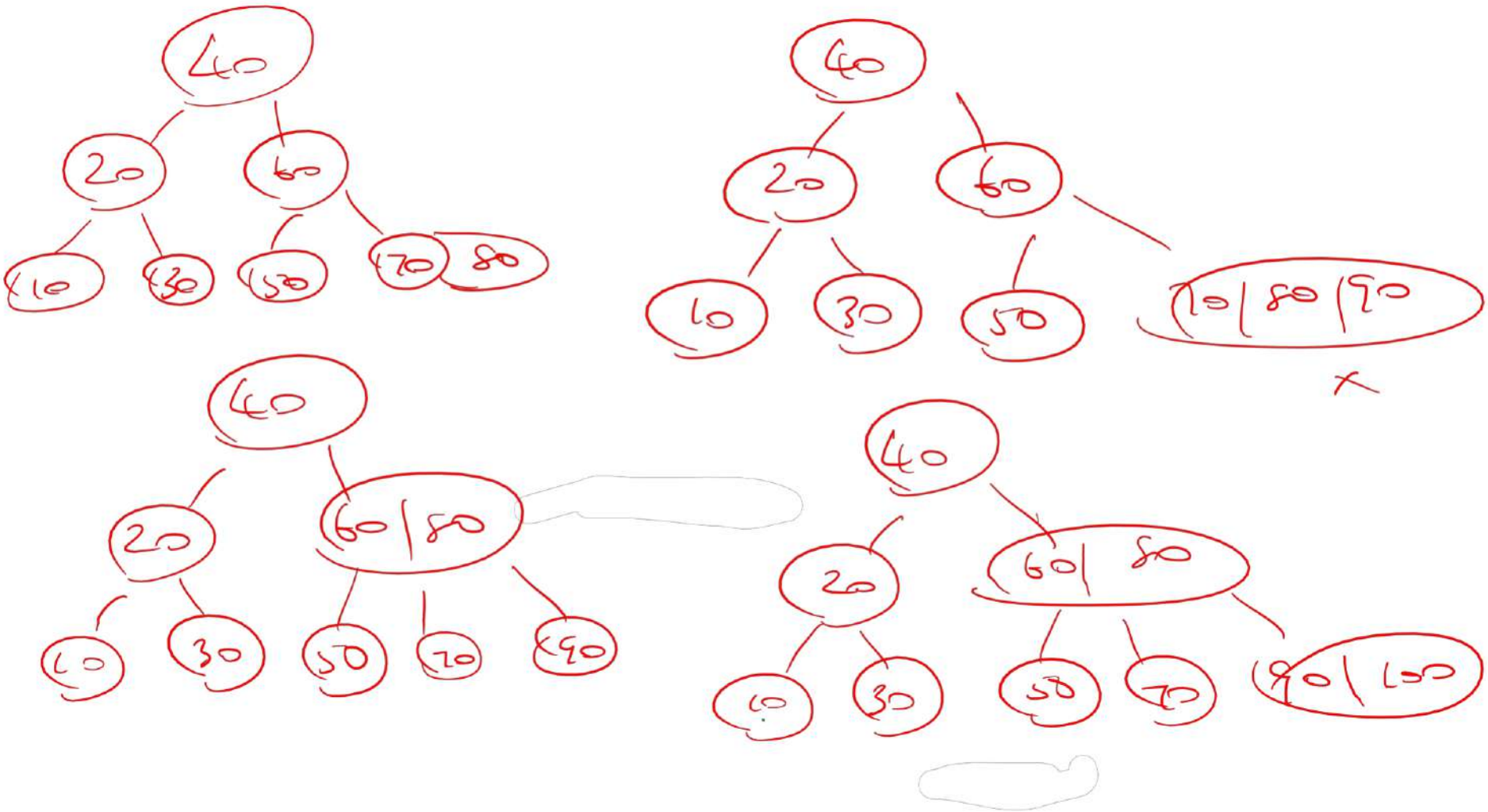
Data: 50,60,70,40,30,20,10, 80,90,100



2-3 tree


40 | 60
20 | 30
50
70
40 | 60
10 | 20 | 30
50
70
20 | 40 | 60
10
30
50
70
40
20
60
10
30
50
70

2-3 tree


40
20
60
10
30
50
70
80
40
20
60
10
30
50
70|80|90
x
40
20
60|80
10
30
50
70
90
40
20
60|80
10
30
50
70
90|100

2-3 tree

Search Algorithm:

Step 1: Start at the Root

Begin at the root node of the 2-3 tree.

Step 2: Compare the Target Key with the Keys in the Current Node

If the current node is a **2-node** (contains one key, say k1):

- If the target key k is **equal to k1**, the search is successful, and you have found the key.
- If the target key k is **less than k1**, move to the **left child**.
- If the target key k is **greater than k1**, move to the **right child**.

If the current node is a **3-node** (contains two keys, say k1 and k2):

- If the target key k is **equal to k1** or **k2**, the search is successful.
- If the target key k is **less than k1**, move to the **left child**.
- If the target key k is **between k1 and k2**, move to the **middle child**.
- If the target key k is **greater than k2**, move to the **right child**.

Step 3: Recursively Search in the Appropriate Child Subtree

After determining which child to move to, recursively repeat the comparison process in the child subtree.

Continue this process until you either:

- **Find the key** in one of the nodes, or
- **Reach a leaf node**.

Step 4: Handle the Case of a Leaf Node

If you reach a leaf node and do not find the target key, the search is unsuccessful, and the key is not present in the tree.

2-3 tree

Delete:

In Deletion Process for a specific value:

- To delete a value, it is replaced by its in-order successor or predecessor and then removed.
- If a node is left with less than one data value then two nodes must be merged together.
- If a node becomes empty after deleting a value, it is then merged with another node.

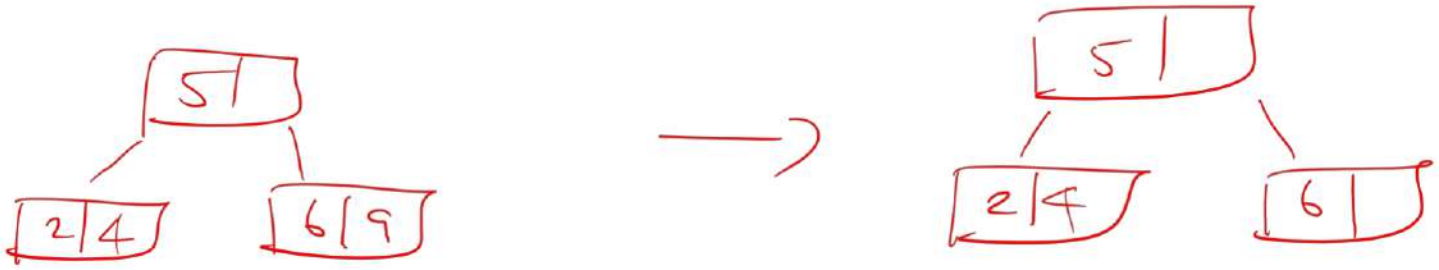
2-3 tree

Deletion:

- All 4 cases

2-3 tree

Case 1 :- Delete 9 (leaf)

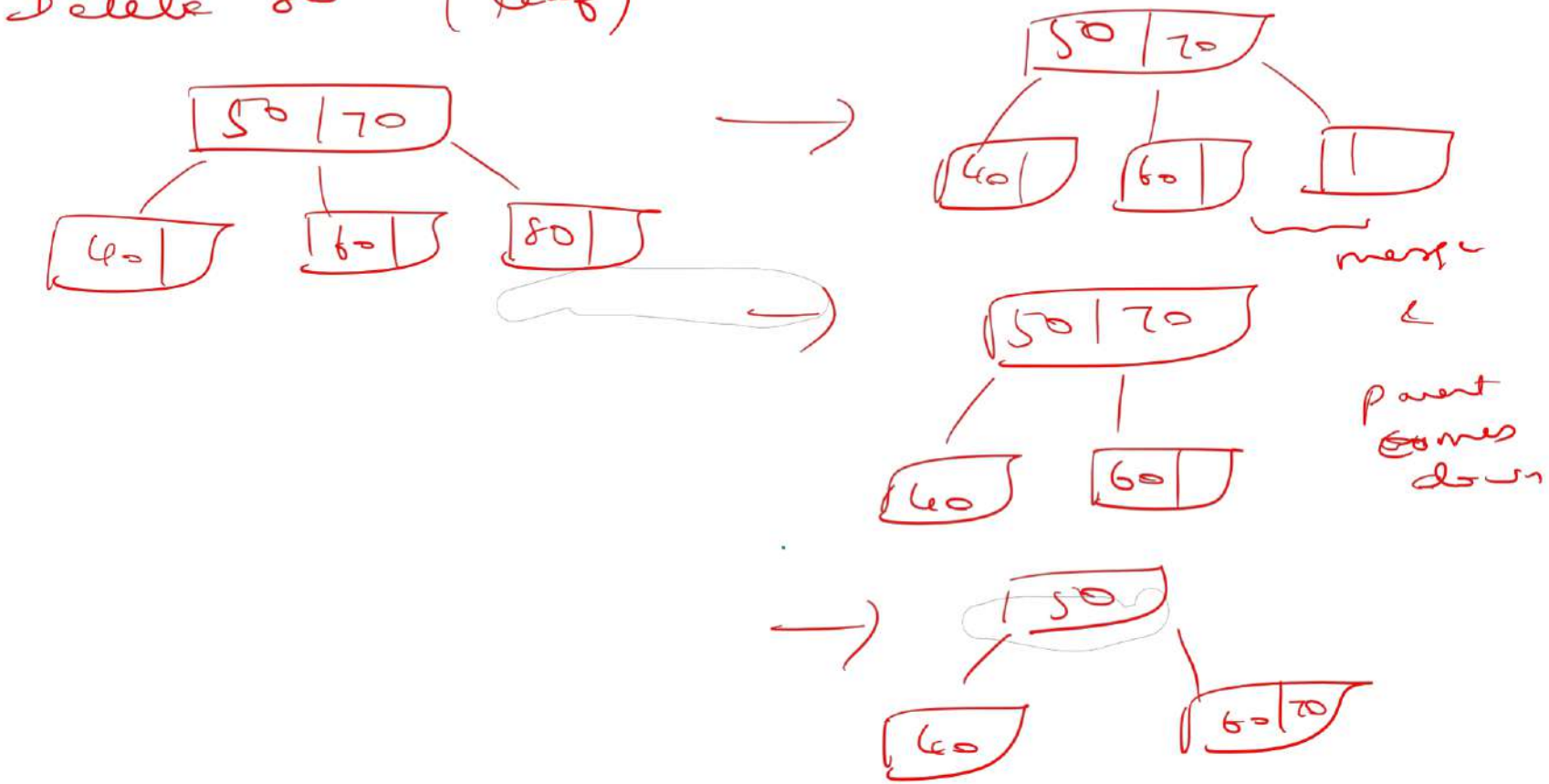


2-3 tree

case 2 :-

Delete and Merge

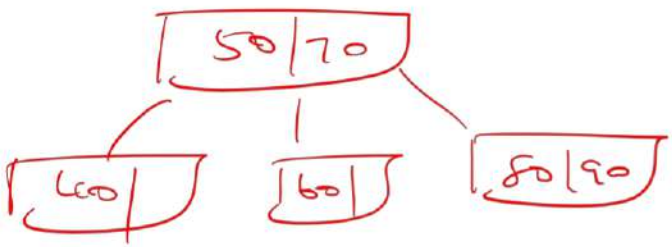
Delete 80 (leaf)

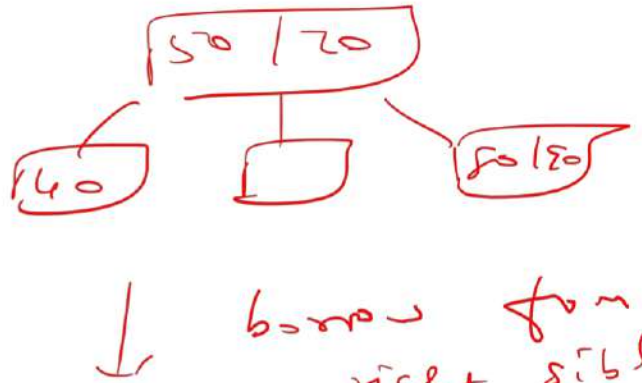


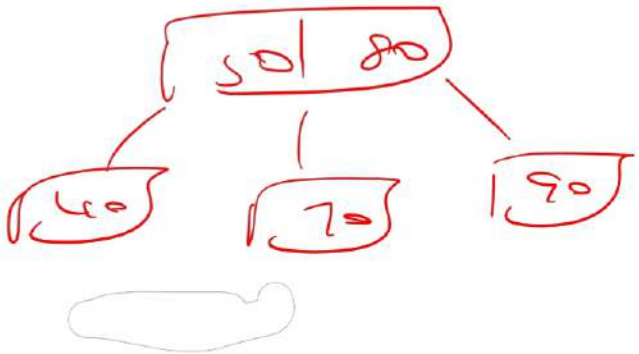
2-3 tree

Case 3 :- Borrowing

Delete 60 (leaf)

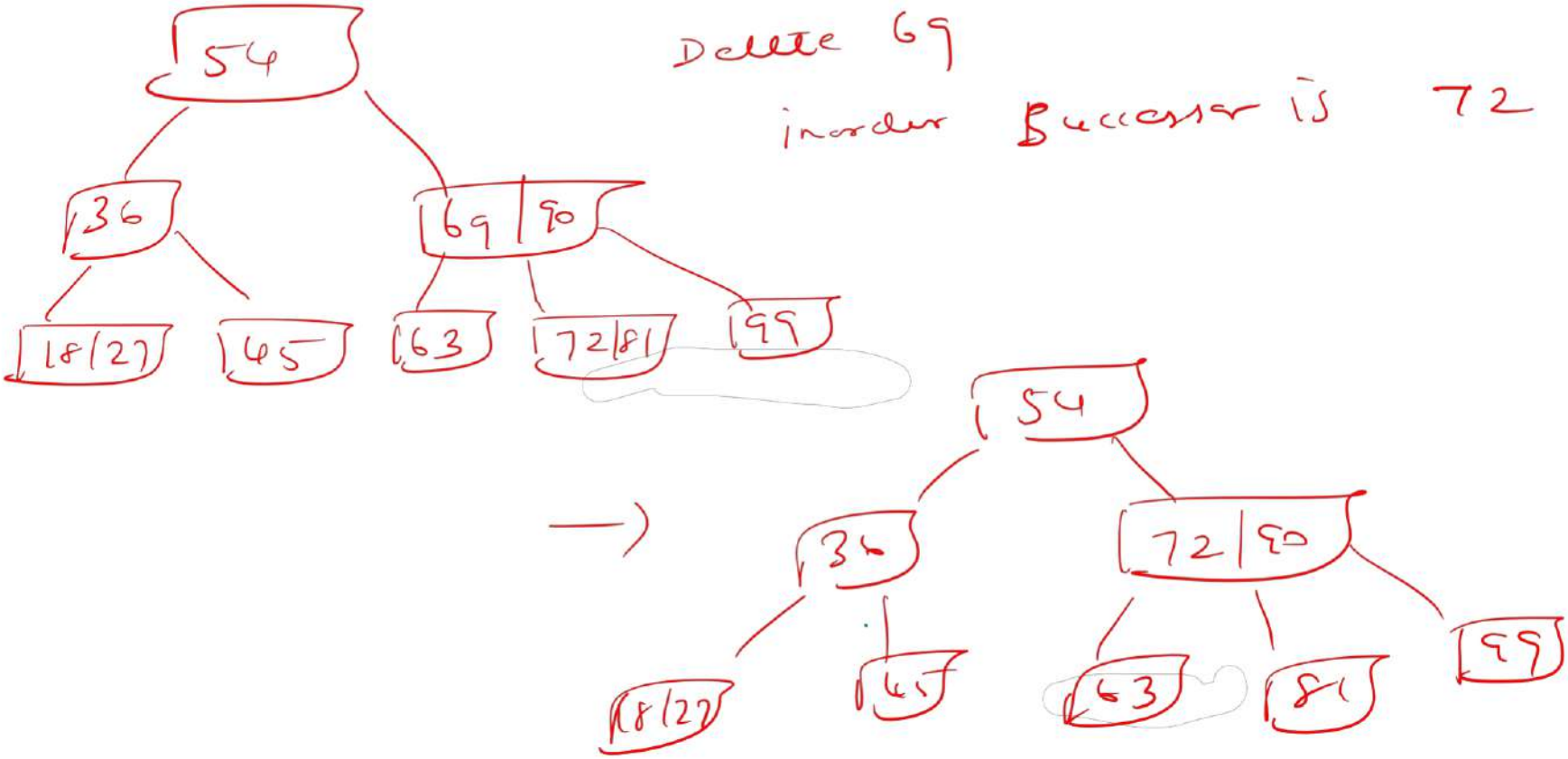






2-3 tree

Case 4 :- Deleting a key in internal node
(Replace with inorder successor)



2-3 tree

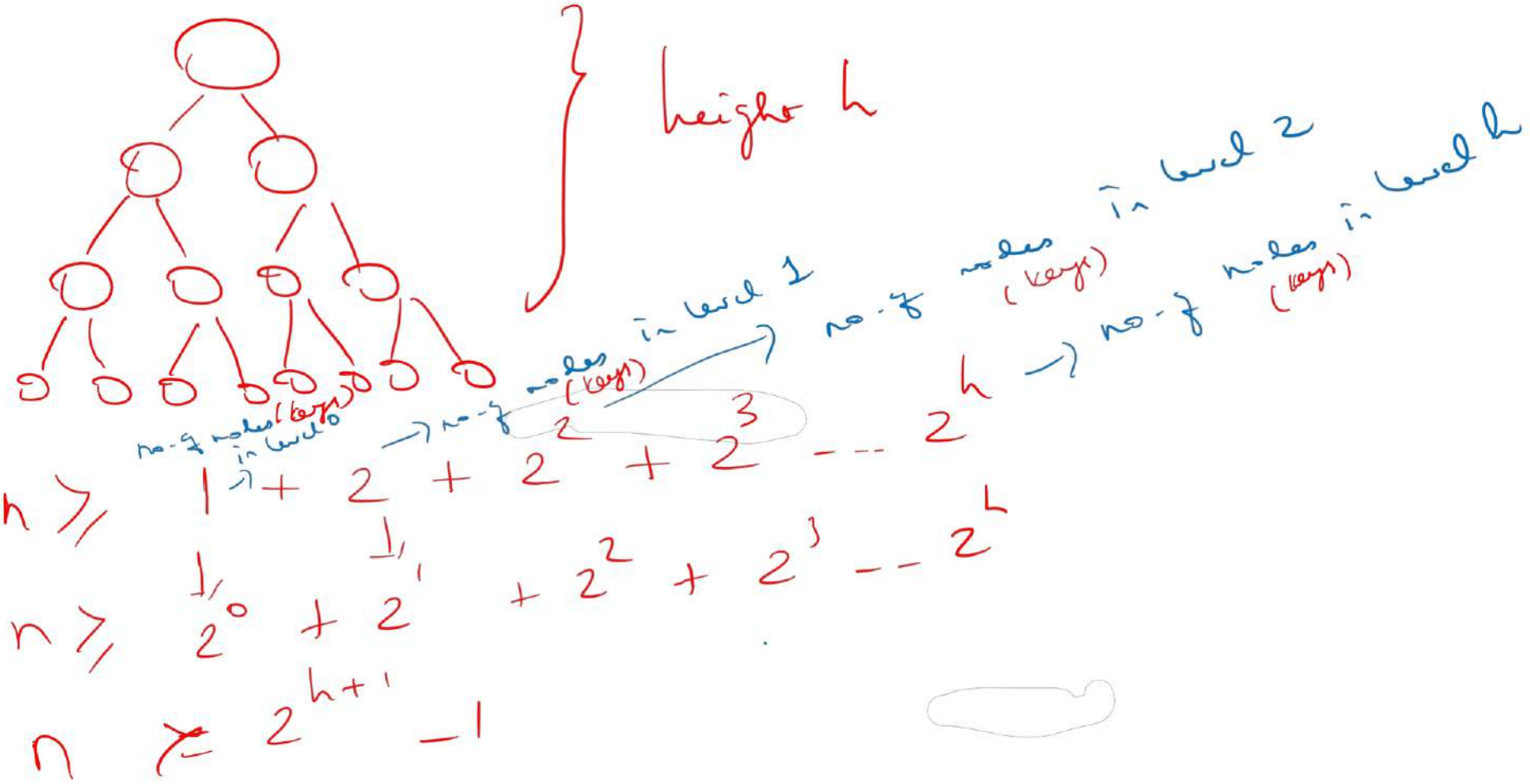
Time complexity:

The time complexity of search/insert/delete is O(log n) .

2-3 tree

2 node structure

Time complexity:



2-3 tree

Add 1 to both sides of inequality

$$n+1 \geq 2^{h+1} - 1 + 1$$

$$n+1 \geq 2^{h+1}$$

Take $\log_2$ on both sides

$$\log_2(n+1) \geq \log_2\left(2^{h+1}\right)$$

$\therefore \log_2 2 = 1$

$$\log_2(n+1) \geq (h+1)\log_2 2$$

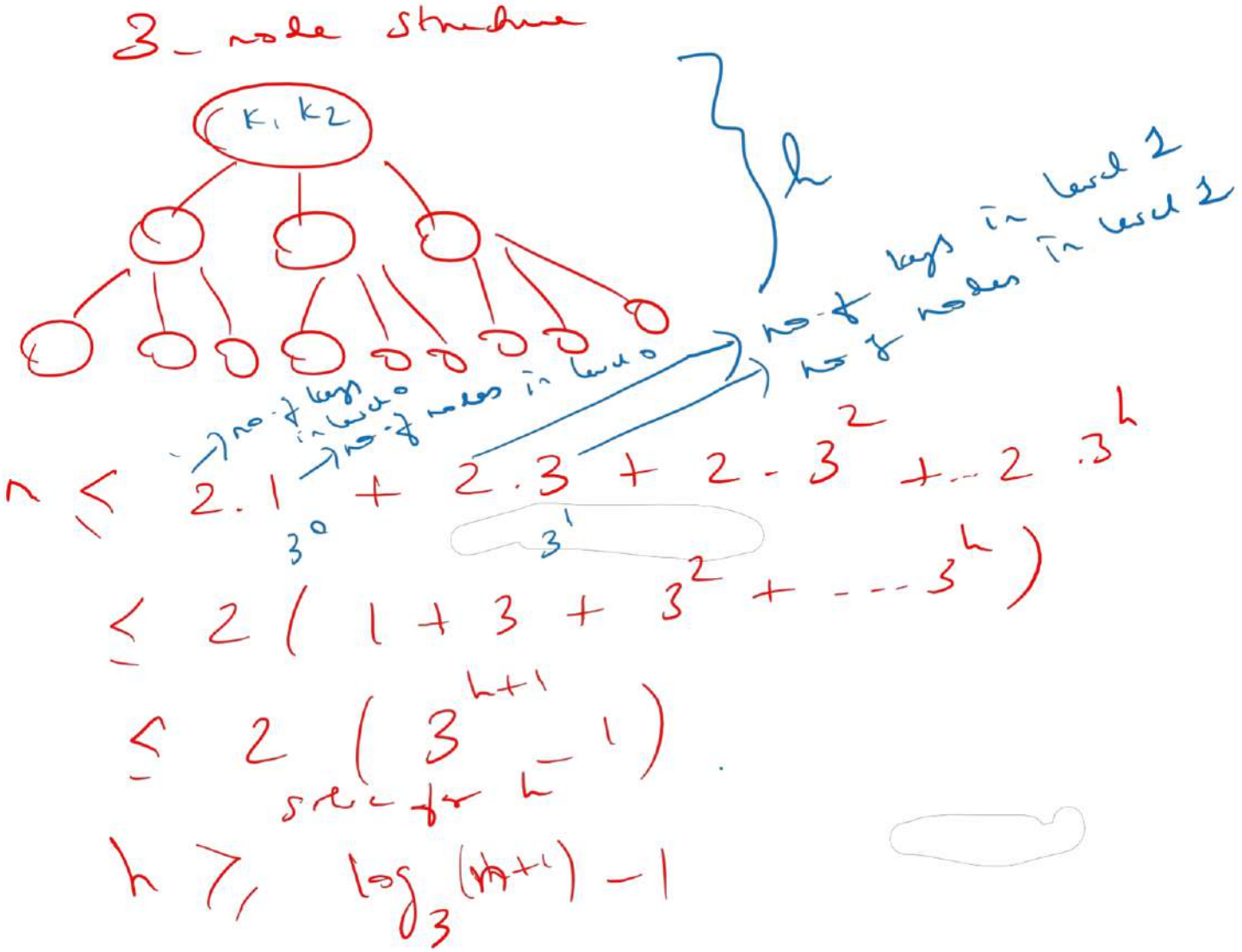
$$\log_2(n+1) \geq (h+1).$$

solve for h

$$\log_2(n+1) - 1 \geq h$$

$$\therefore h \leq \log_2(n+1) - 1$$

2-3 tree


3-node structure
k1, k2
h
no. of keys in level 0
no. of nodes in level 0
no. of keys in level 1
no. of nodes in level 1
n ≤ 2.1 + 2.3 + 2.3^2 + ... 2.3^h
3^0
3^1
≤ 2(1 + 3 + 3^2 + ... 3^h)
≤ 2(3^{h+1} − 1)
solve for h
h ≥ log_3(n/2 + 1) − 1

2-3 tree

B-Tree vs 2-3 tree:

•**2-3 Tree**:
•It is a specific type of B-Tree where each node has either 2 or 3 children. Nodes can be either 2-nodes (with 1 key and 2 children) or 3-nodes (with 2 keys and 3 children).

•**B-Tree**:
•A B-Tree is a generalization of a 2-3 tree. It is a self-balancing tree data structure that maintains sorted data and allows searches, sequential access, insertions, and deletions in logarithmic time.
•A B-Tree of order m (also known as m-ary tree) can have a maximum of m-1 keys and m children. This generalizes the concept where m can be greater than 3, unlike the 2-3 tree.

•**2-3 Tree**: Simpler, with fixed node structures. It's a good introductory example of balanced trees.
•**B-Tree**: More general, scalable, and adaptable, making it suitable for large-scale applications like databases and filesystems.

Splay tree

Splay tree

- Understanding AVL tree is required

Splay tree

- A **splay tree** is a self-adjusting binary search tree with the additional property that recently accessed elements are quick to access again.

- Not a strictly balanced binary search tree (like AVL)

- This is achieved through a process called **splaying**, which moves an accessed node to the root of the tree through a series of tree rotations. That is, whenever an operation like insertion, deletion, search is performed on the tree, rotation (splaying) will be done

Splaying (to spread wide apart)

Splay tree

Key Concepts

1. **Self-Adjusting**: The tree reorganizes itself whenever a node is accessed, making frequently accessed nodes quicker to access in the future.
2. **Binary Search Tree (BST) Property**: Like any BST, the left child of a node contains values smaller than the node, and the right child contains values larger.
3. **Splaying**: When a node is accessed (via search, insertion, or deletion), the splay operation brings it to the root of the tree using rotations. This ensures that future operations on that node (or nodes near it) are faster.

Splay tree

Splay Tree Operations

1. **Search**: When searching for a node, the node is splayed to the root if found. If not found, the last accessed node (where the search terminated) is splayed to the root.
2. **Insertion**: A new node is inserted as in a standard BST, and then the node is splayed to the root.
3. **Deletion**: To delete a node, first splay the node to the root. Then, if it has two children, split the tree into two trees, one containing all elements smaller and the other containing all elements larger. Finally, merge these two trees.

Splay tree

Time Complexity

- **Amortized Analysis**: The amortized time complexity of basic operations (search, insert, delete) in a splay tree is O(log n).
- **Worst-Case**: In the worst case, a single operation can take O(n) time, but over a sequence of operations, the average time per operation is O(log n).

Splay tree

Advantages of Splay Trees

- **No Need for Explicit Balancing**: Unlike AVL or Red-Black trees, splay trees don't require explicit balancing after insertions and deletions.
- **Efficiency for Locality of Reference**: Splay trees are particularly efficient in scenarios where the same elements are accessed repeatedly.

Splay tree

Disadvantages of Splay Trees

- **Worst-Case Performance**: For certain access patterns, the performance of splay trees can degrade to O(n) per operation.
- **Complexity of Implementation**: Splay trees are more complex to implement compared to simpler binary search trees.

Splay tree

Use Cases

- **Caching**: Splay trees are useful in applications where elements are accessed with a non-uniform frequency, as they tend to keep frequently accessed items near the root.
- **Data Compression Algorithms**: Used in the implementation of the move-to-front heuristic, which is part of the Burrows-Wheeler transform.

Splay tree

Search:

Even if we perform search, we need to restructure the tree

Splay tree

Splay Tree is a self - adjusted Binary Search Tree in which every operation on element rearranges the tree so that the element is placed at the root position of the tree.

In a splay tree, every operation is performed at the root of the tree. All the operations in splay tree are involved with a common operation called "Splaying".

Splaying an element, is the process of bringing it to the root position by performing suitable rotation operations.

Splay tree

In a splay tree, splaying an element rearranges all the elements in the tree so that splayed element is placed at the root of the tree.

Every operation on splay tree performs the splaying operation.

For example, the insertion operation first inserts the new element using the binary search tree insertion process, then the newly inserted element is splayed so that it is placed at the root of the tree.

The search operation in a splay tree is nothing but searching the element using binary search process and then splaying that searched element so that it is placed at the root of the tree.

Splay tree

In splay tree, to splay any element we use the following rotation operations.

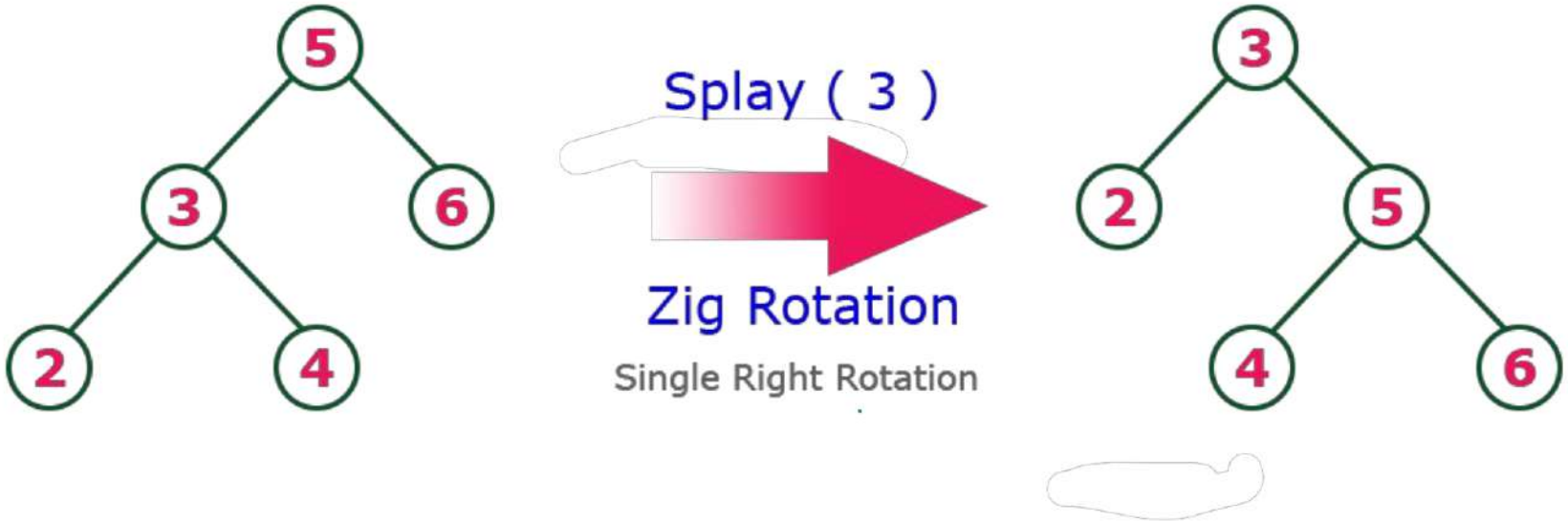
- **Rotations in Splay Tree**

1. Zig Rotation
2. Zag Rotation
3. Zig - Zig Rotation
4. Zag - Zag Rotation
5. Zig - Zag Rotation
6. Zag - Zig Rotation

Splay tree

- **Zig Rotation**

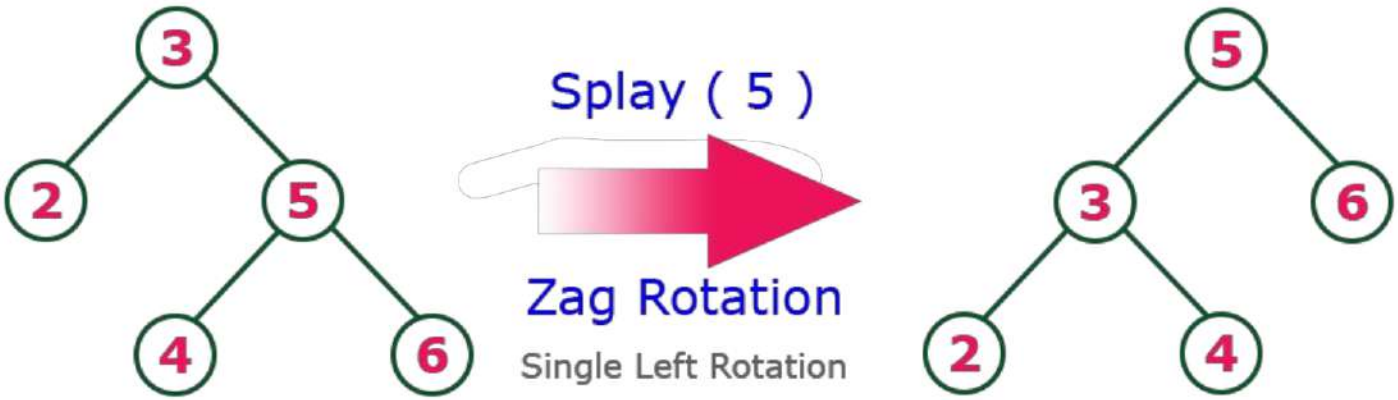
The **Zig Rotation** in splay tree is similar to the single right rotation in AVL Tree rotations. In zig rotation, every node moves one position to the right from its current position.



Splay tree

- **Zag Rotation**

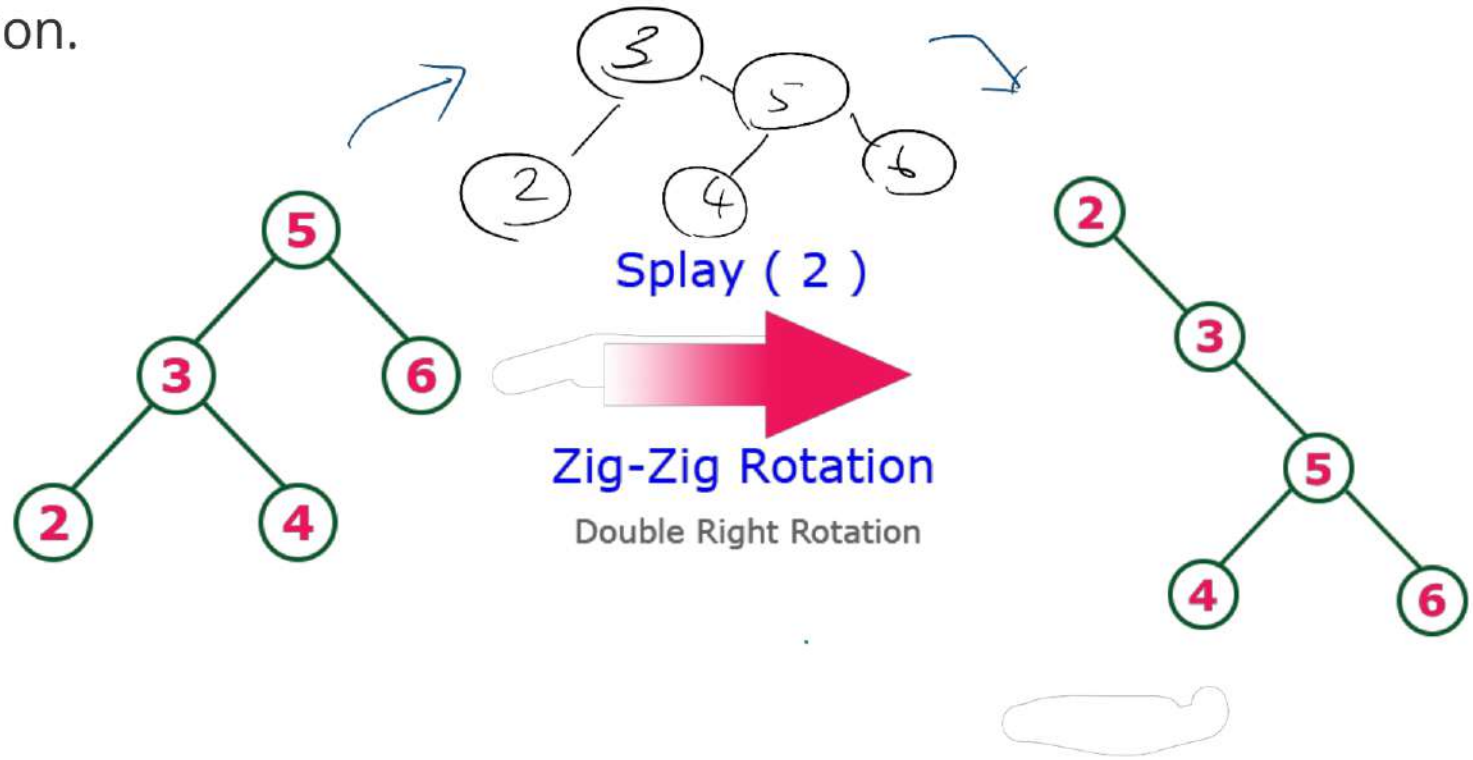
The **Zag Rotation** in splay tree is similar to the single left rotation in AVL Tree rotations. In zag rotation, every node moves one position to the left from its current position



Splay tree

- **Zig-Zig Rotation**

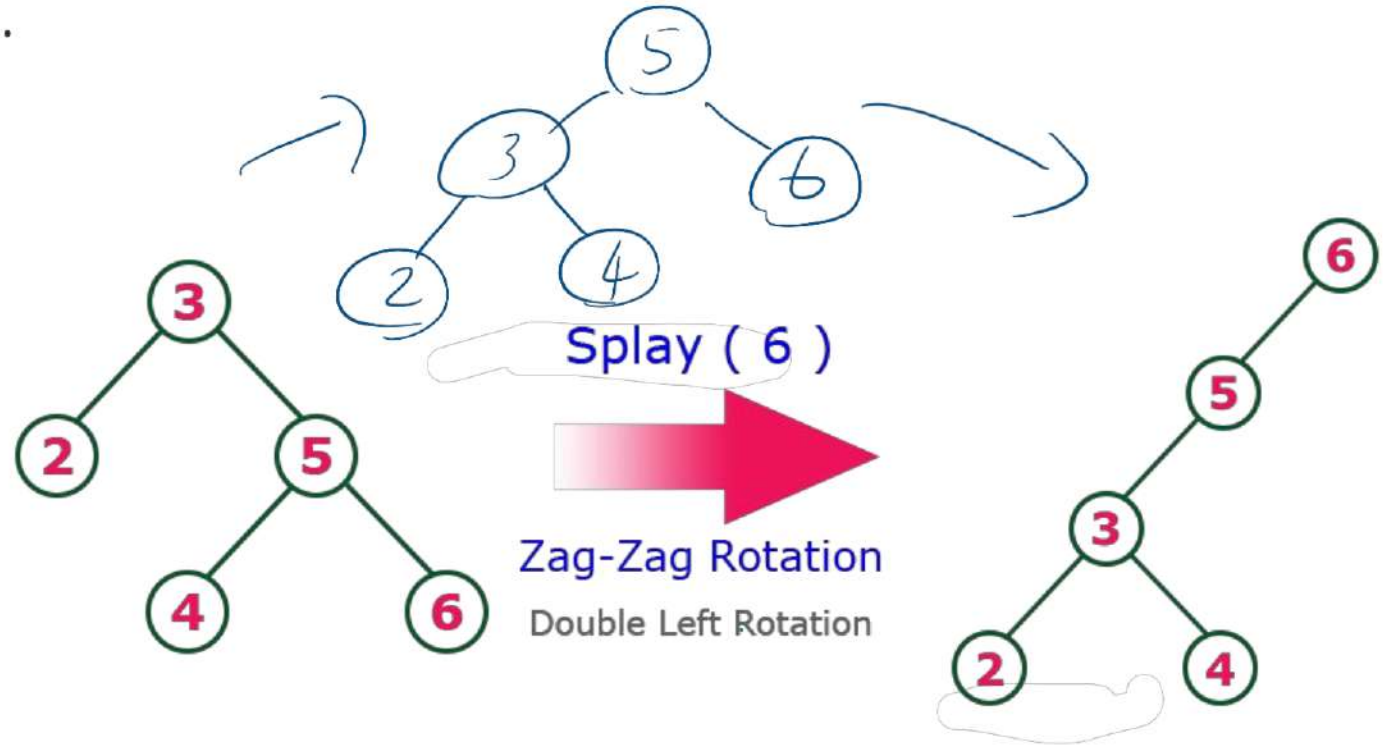
The **Zig-Zig Rotation** in splay tree is a double zig rotation. In zig-zig rotation, every node moves two positions to the right from its current position.



Splay tree

- **Zag-Zag Rotation**

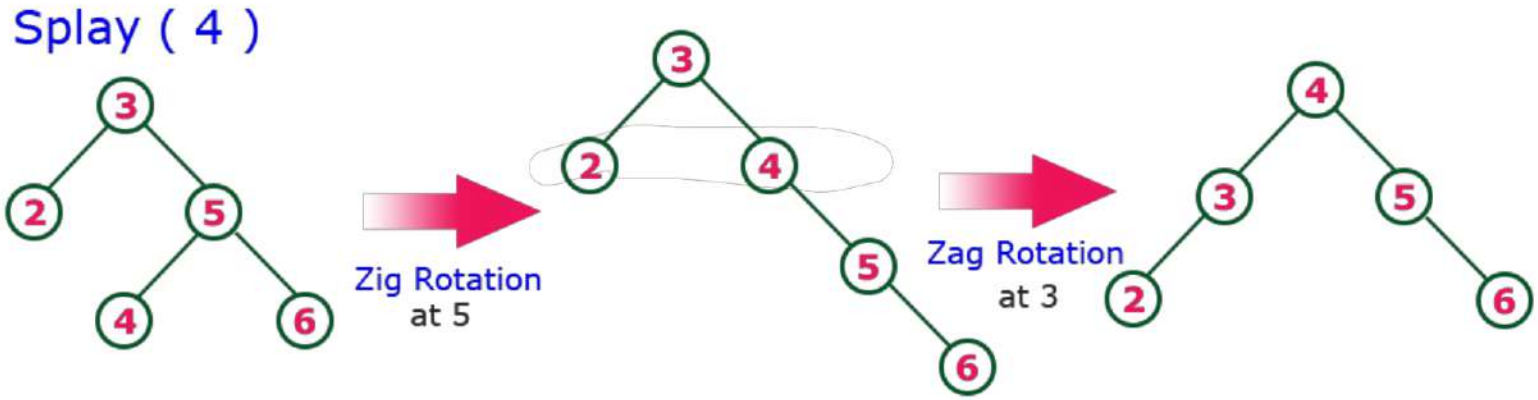
The **Zag-Zag Rotation** in splay tree is a double zag rotation. In zag-zag rotation, every node moves two positions to the left from its current position.



Splay tree

- **Zig-Zag Rotation**

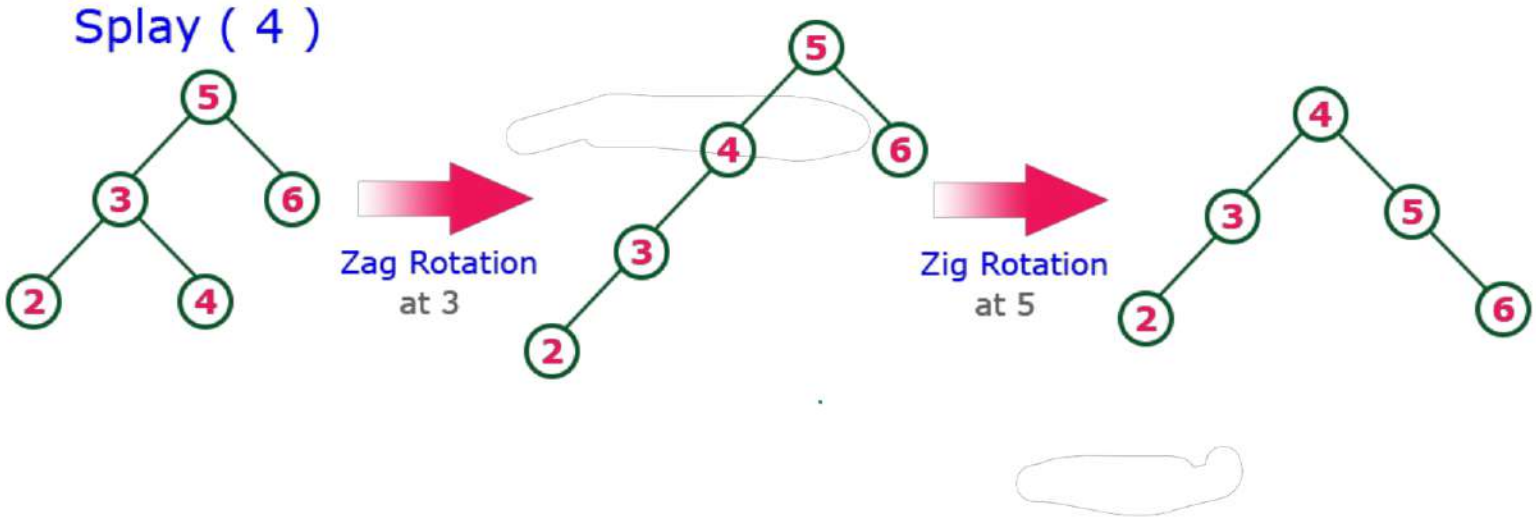
The **Zig-Zag Rotation** in splay tree is a sequence of zig rotation followed by zag rotation. In zig-zag rotation, every node moves one position to the right followed by one position to the left from its current position.



Splay tree

- **Zag-Zig Rotation**

The **Zag-Zig Rotation** in splay tree is a sequence of zag rotation followed by zig rotation. In zag-zig rotation, every node moves one position to the left followed by one position to the right from its current position.



Splay tree

Every Splay tree must be a binary search tree but it is need not to be a balanced tree.

After every operation, the recently accessed key become the root

So, it is useful in scenario where constant access time is required particularly like cache memory, routing table, virtual memory

Splay tree

- **Insertion Operation in Splay Tree**

The insertion operation in Splay tree is performed using following steps...

Step 1 - Check whether tree is Empty.

Step 2 - If tree is Empty then insert the **newNode** as Root node and exit from the operation.

Step 3 - If tree is not Empty then insert the newNode as leaf node using Binary Search tree insertion logic.

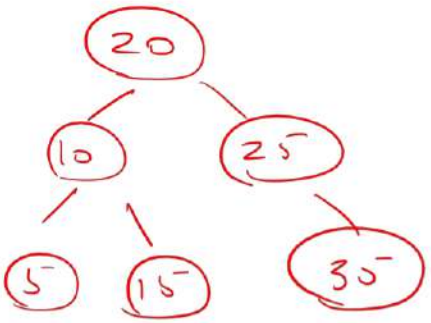
Step 4 - After insertion, **Splay** the **newNode**

- **Deletion Operation in Splay Tree**

The deletion operation in splay tree is similar to deletion operation in Binary Search Tree. But before deleting the element, we first need to **splay** that element and then delete it from the root position. Finally join the remaining tree using binary search tree logic.

Splay tree

- **Search Operation in Splay Tree**



Search Key : 10

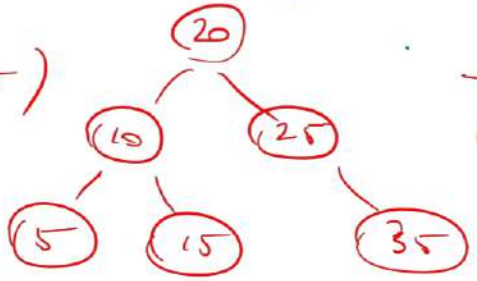
Reference node is root : 20

After performing operation, do rotation (splaying)

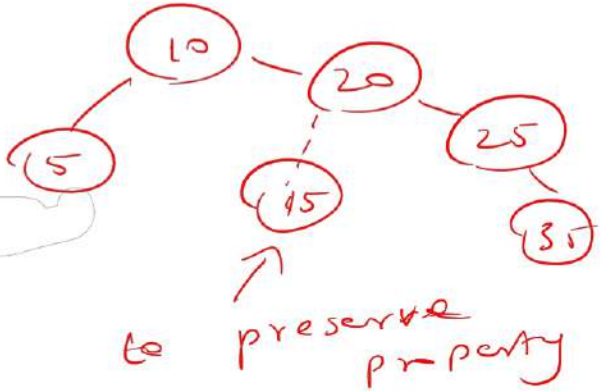
How to make '10' to be present in root.

For that, perform right rotation

(i) Zig (right)

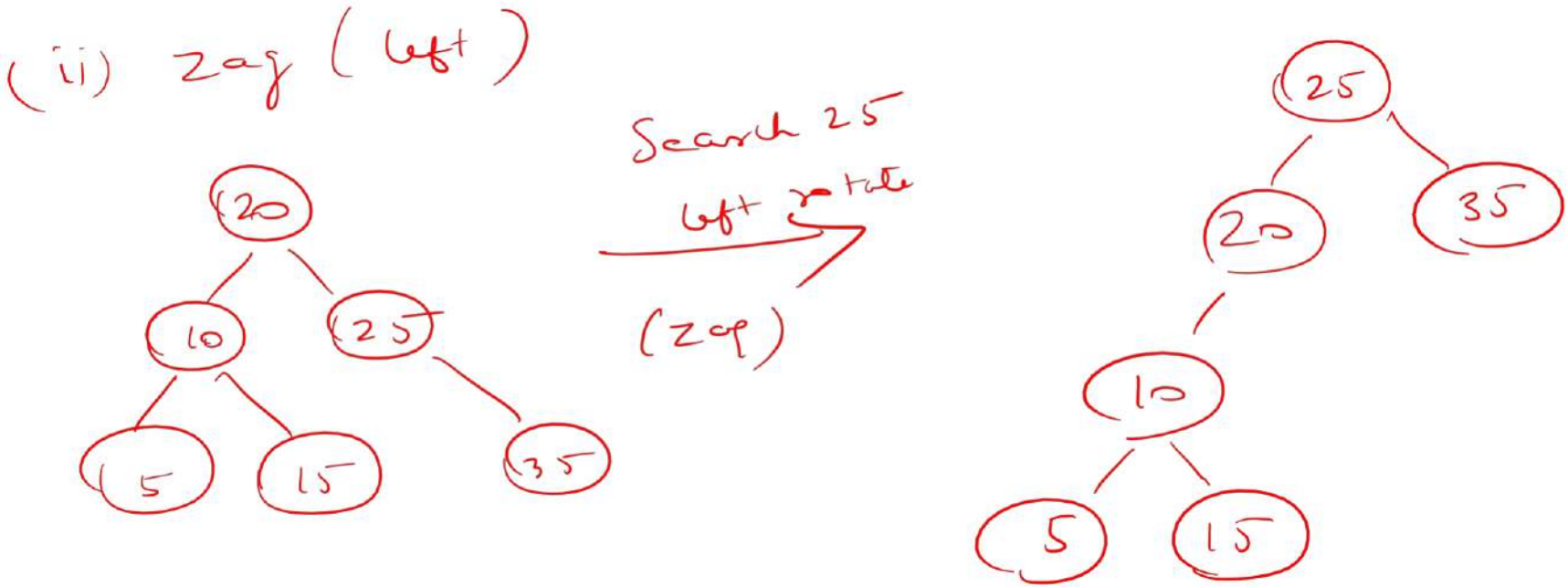


Right rotate (zig)



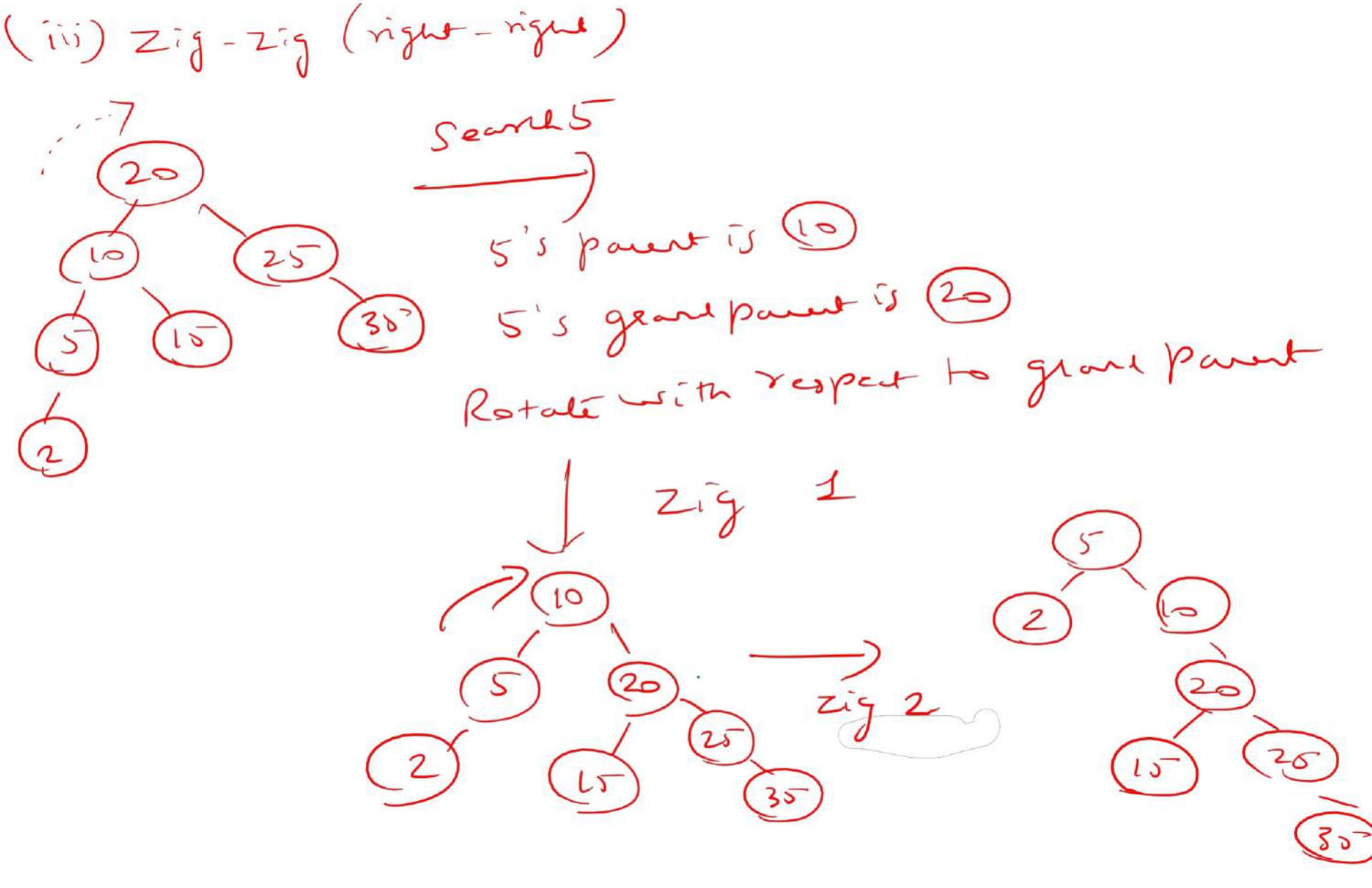
Splay tree

- **Search Operation in Splay Tree**



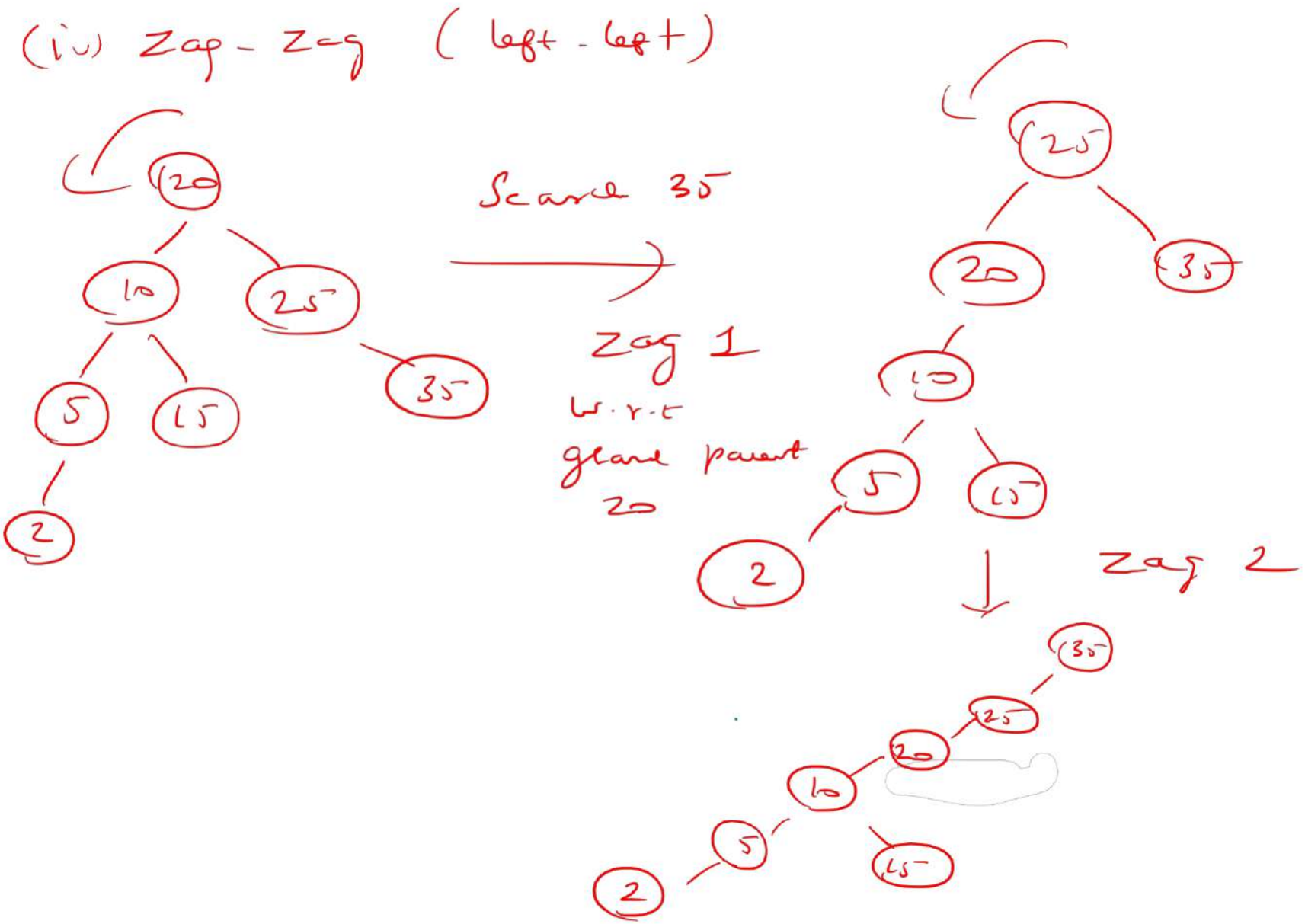
Splay tree

- **Search Operation in Splay Tree**



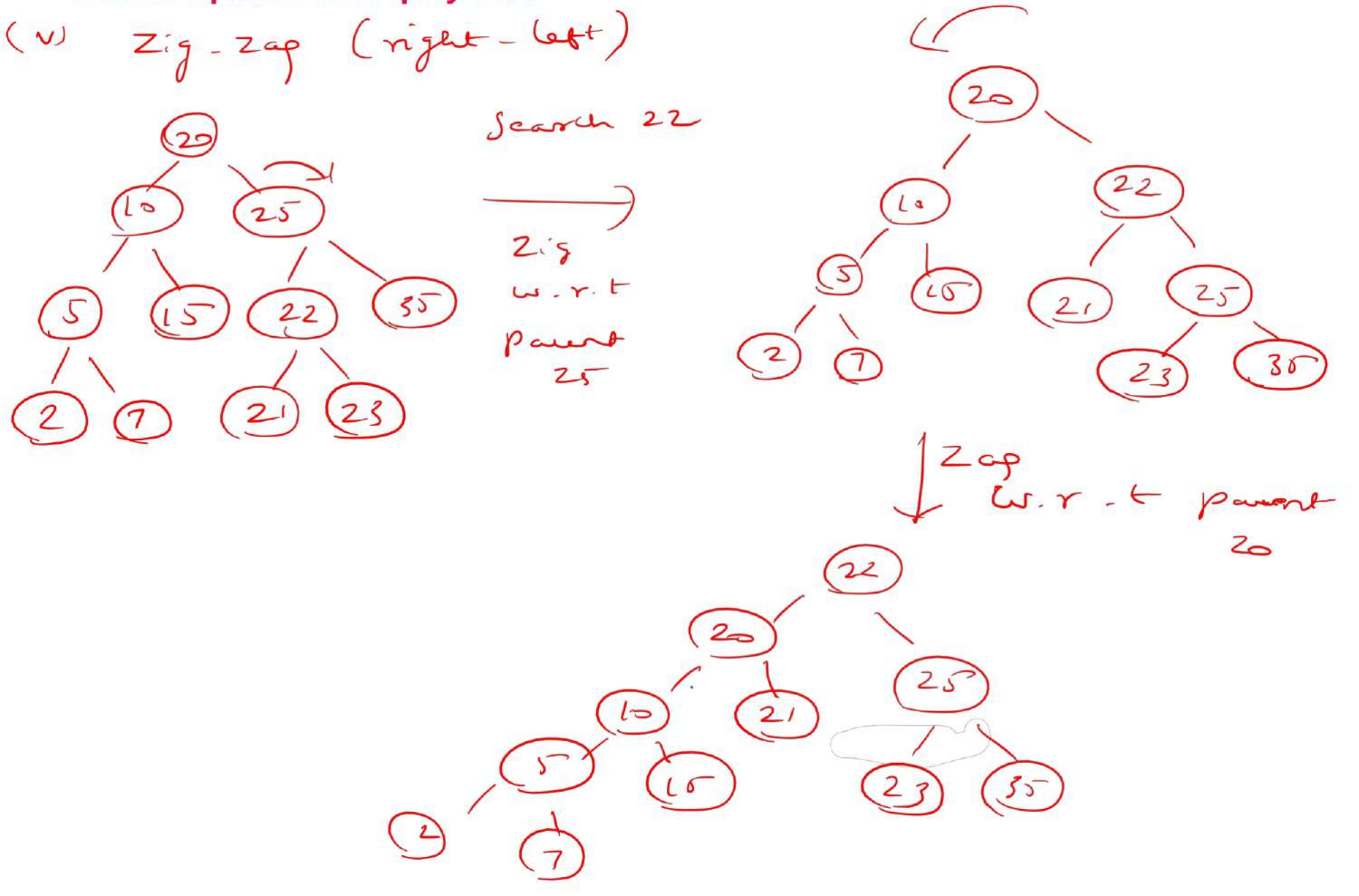
Splay tree

- **Search Operation in Splay Tree**



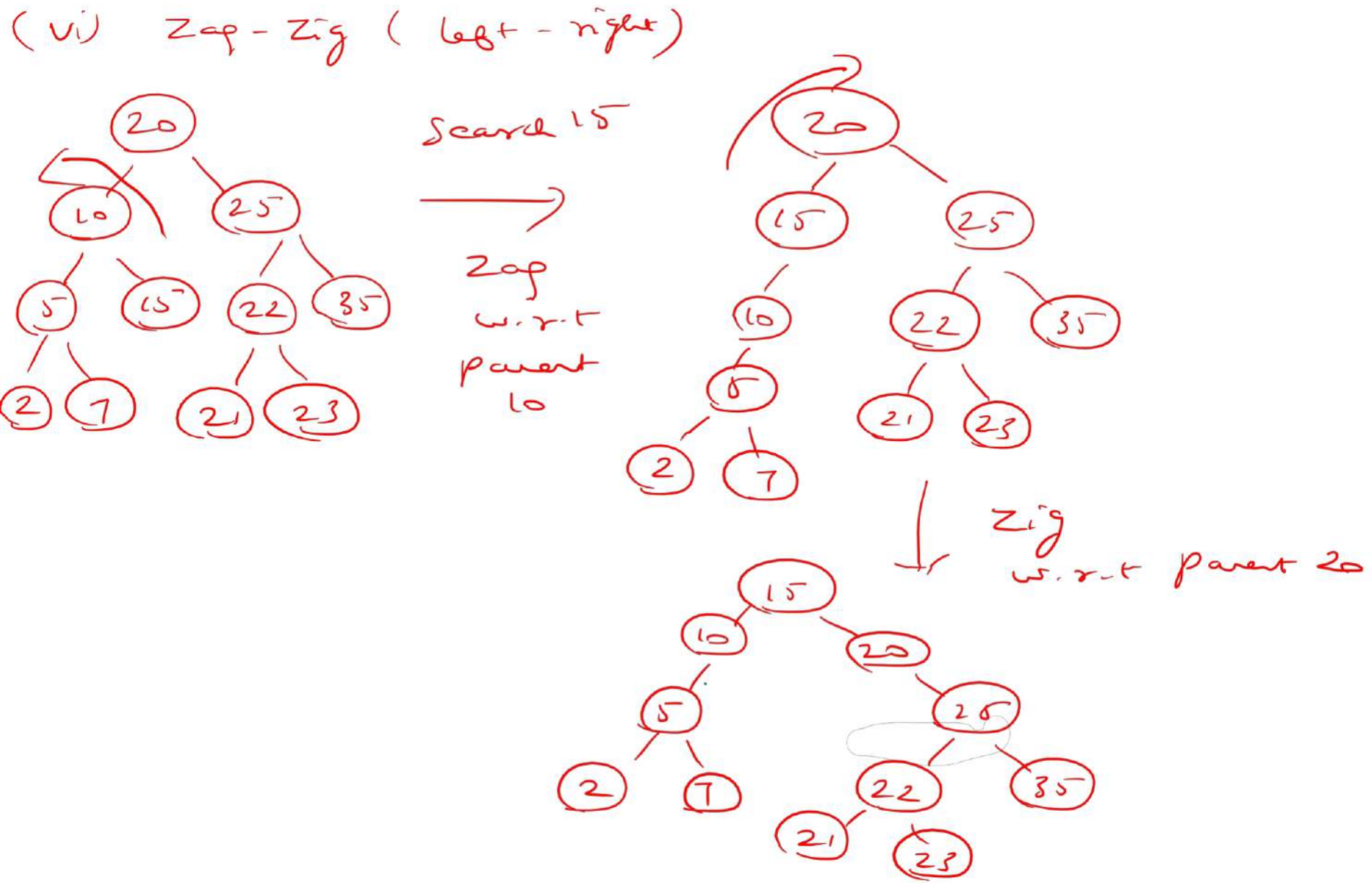
Splay tree

- **Search Operation in Splay Tree**



Splay tree

- **Search Operation in Splay Tree**



Splay tree

- **Insertion Operation in Splay Tree**

Data : 25, 10, 30, 5, 20, 2

After performing the operation, make sure that recently accessed Key / node becomes the root.

If we look at the data, the last element 2 will become the root.

Insert 25 :

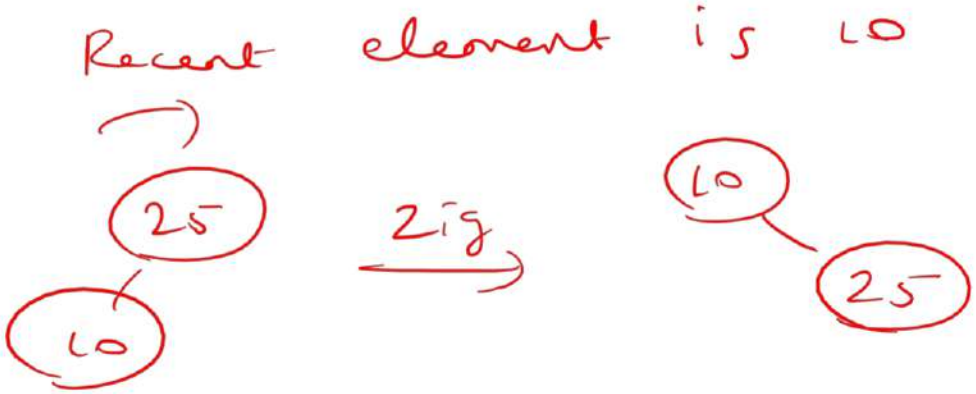
BST for 25

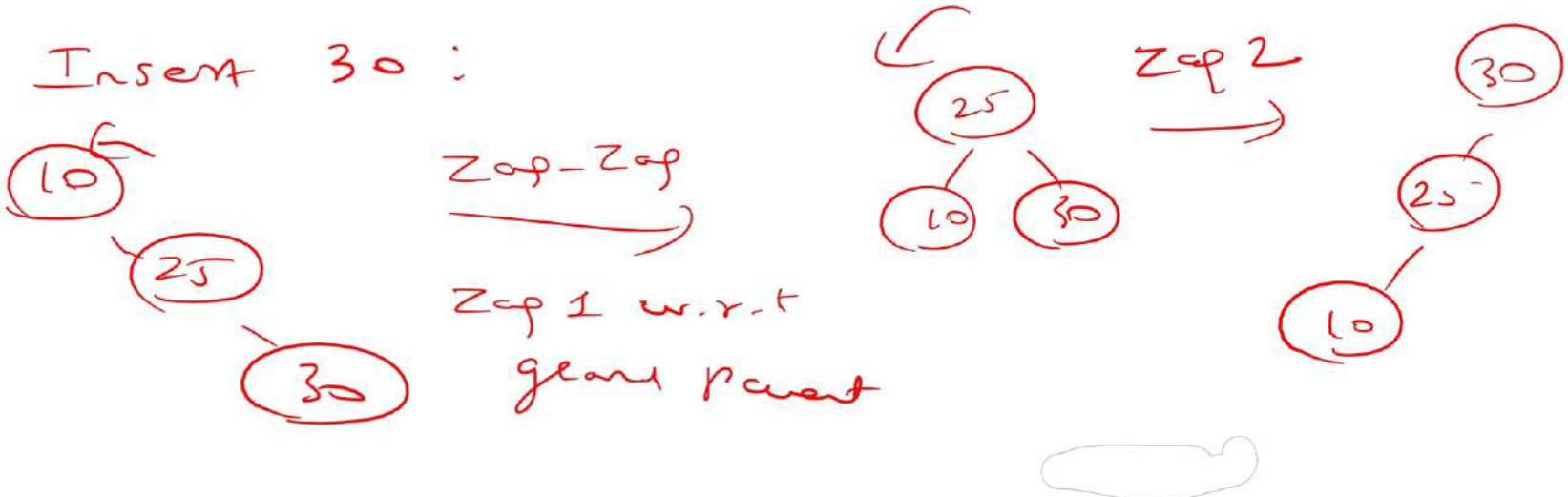
Insert 10 :

BST for 10

Splay tree

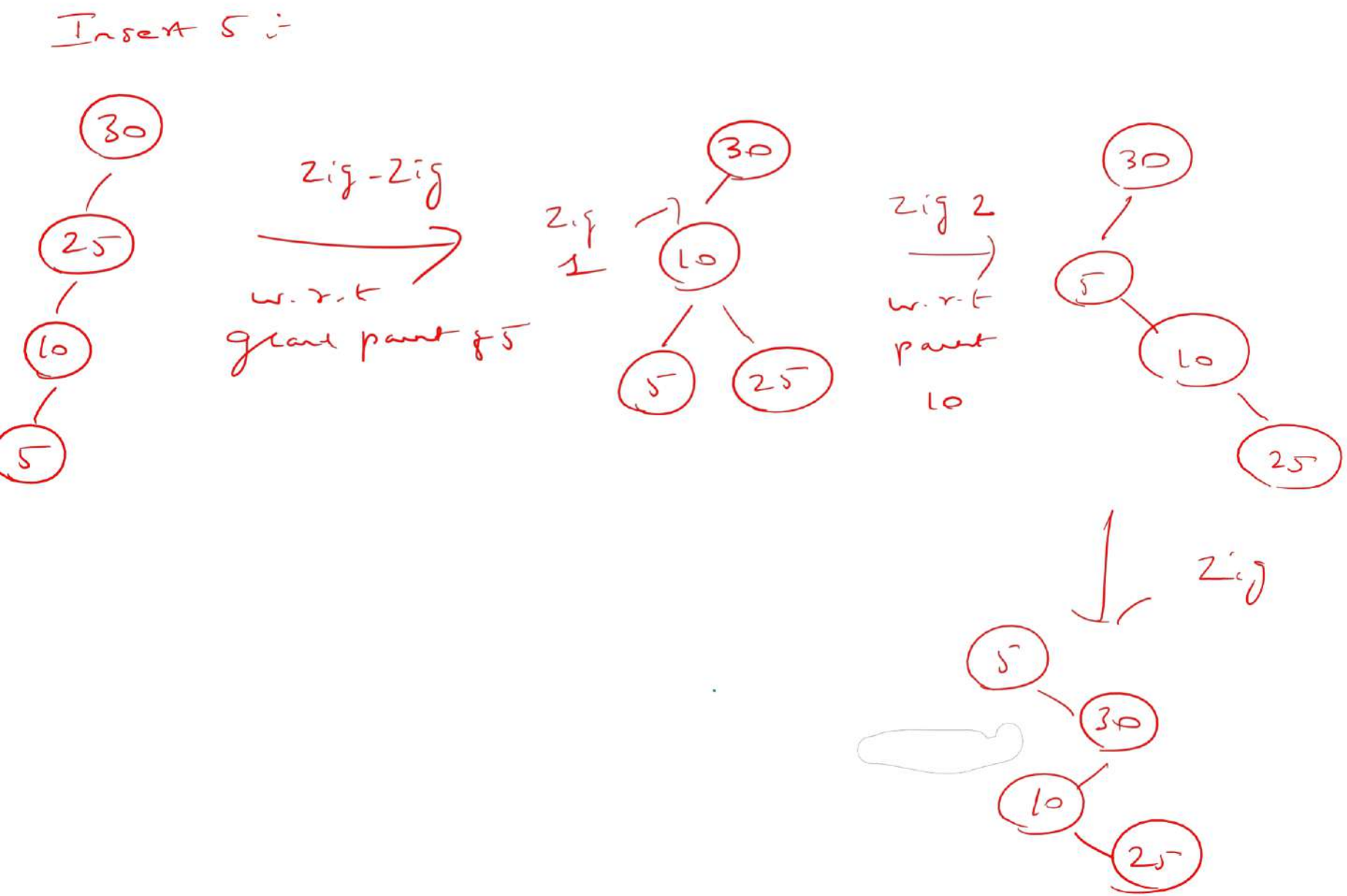
- **Insertion Operation in Splay Tree**





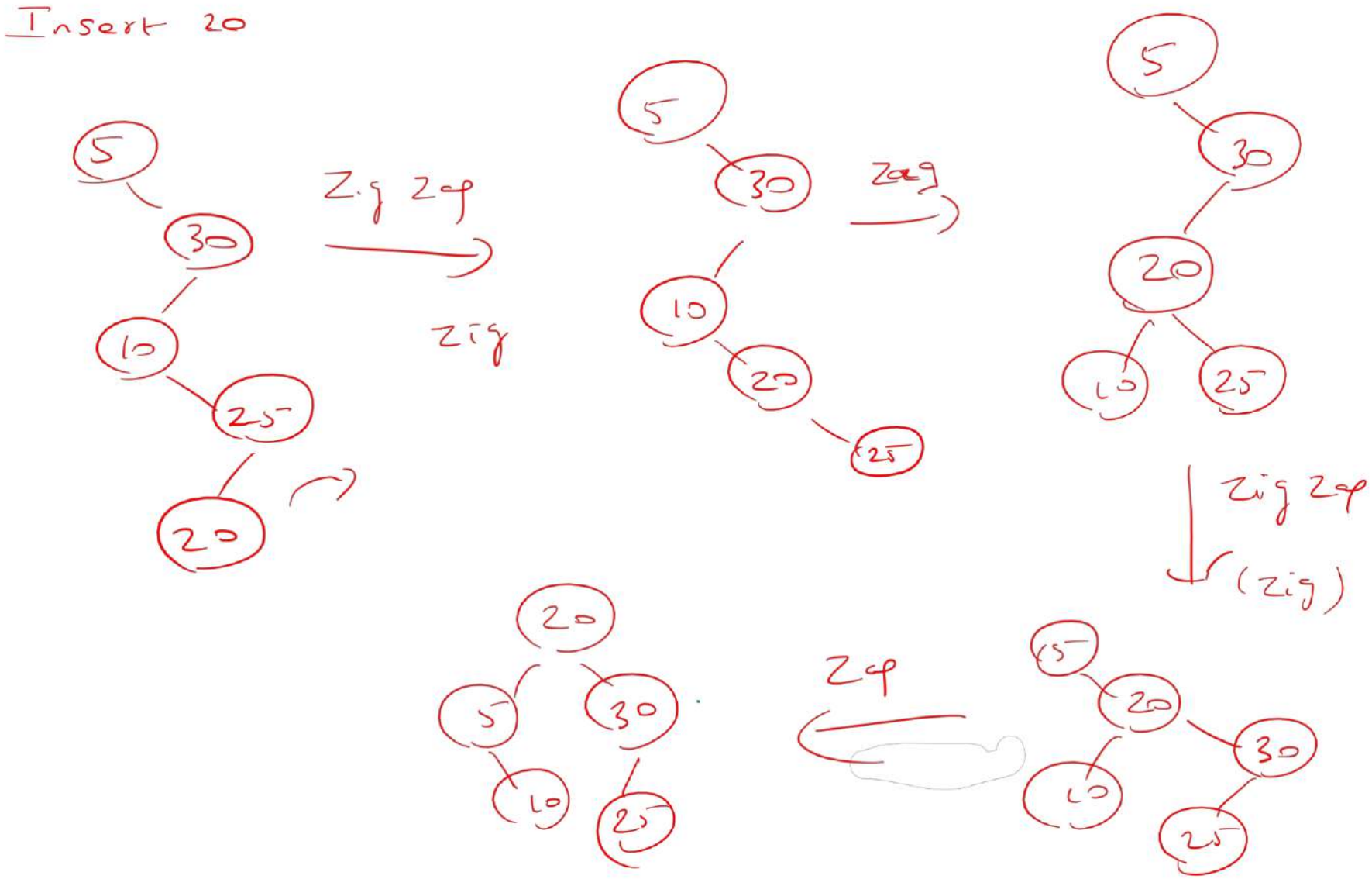
Splay tree

- **Insertion Operation in Splay Tree**



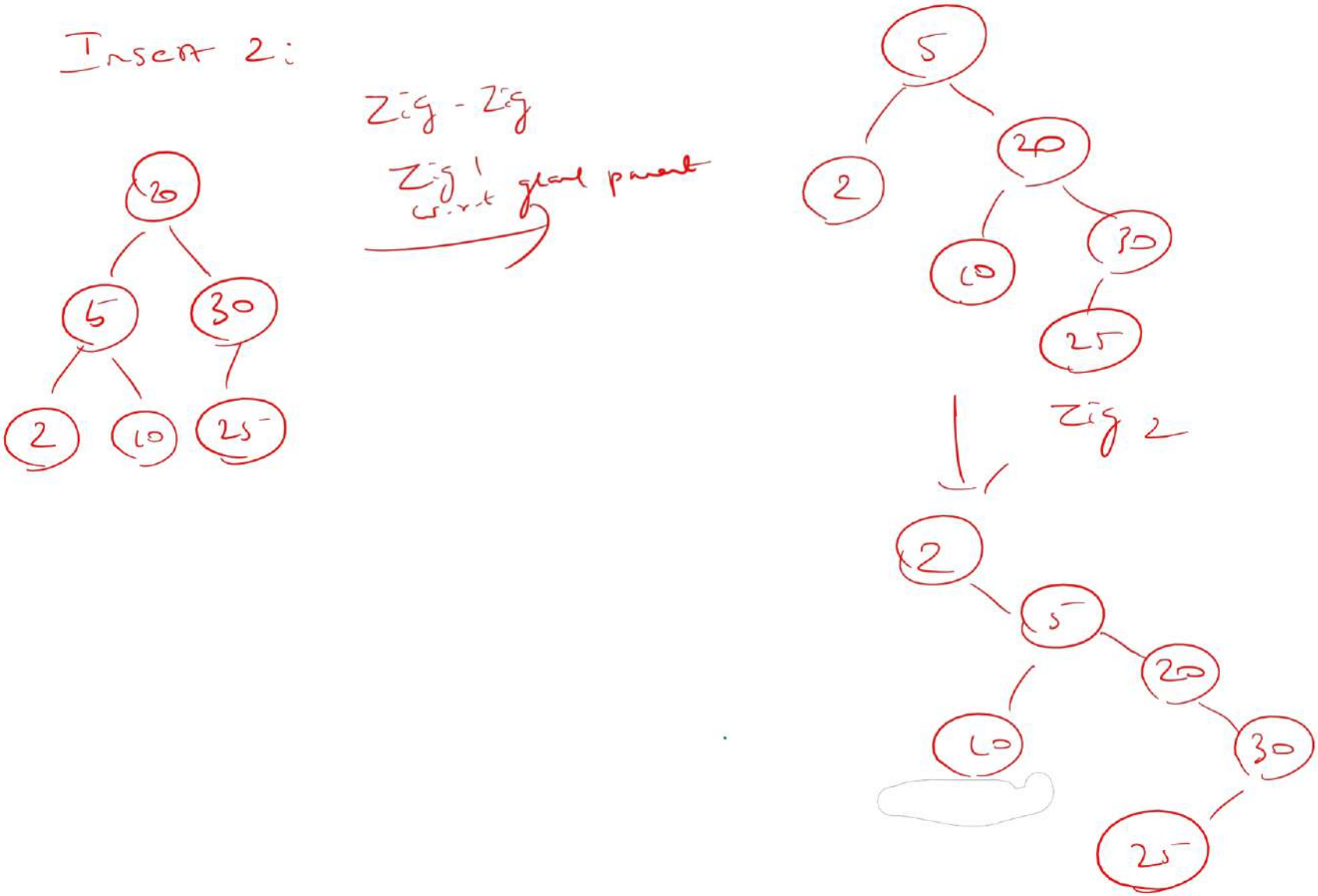
Splay tree

- **Insertion Operation in Splay Tree**



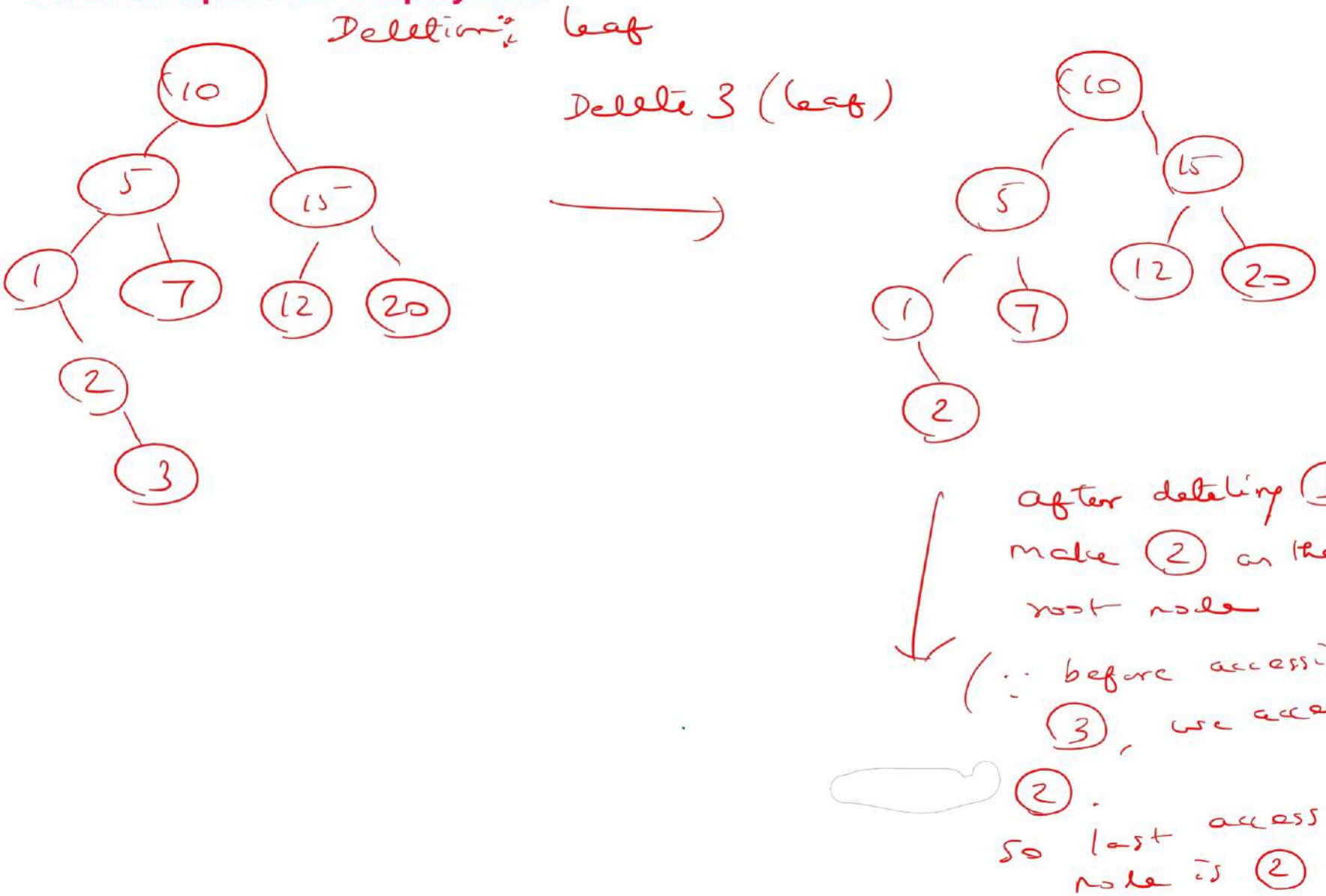
Splay tree

- **Insertion Operation in Splay Tree**



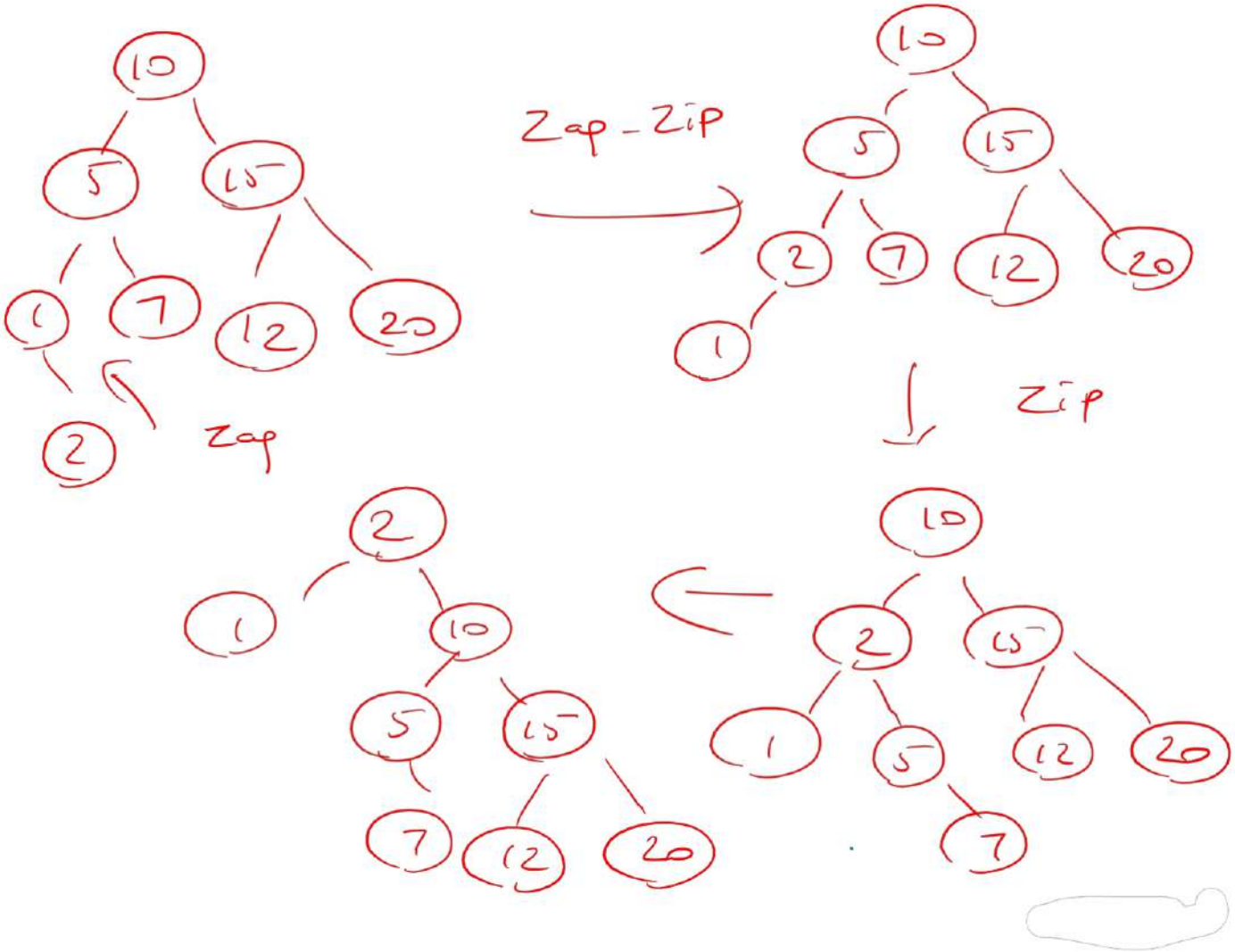
Splay tree

- **Deletion Operation in Splay Tree**



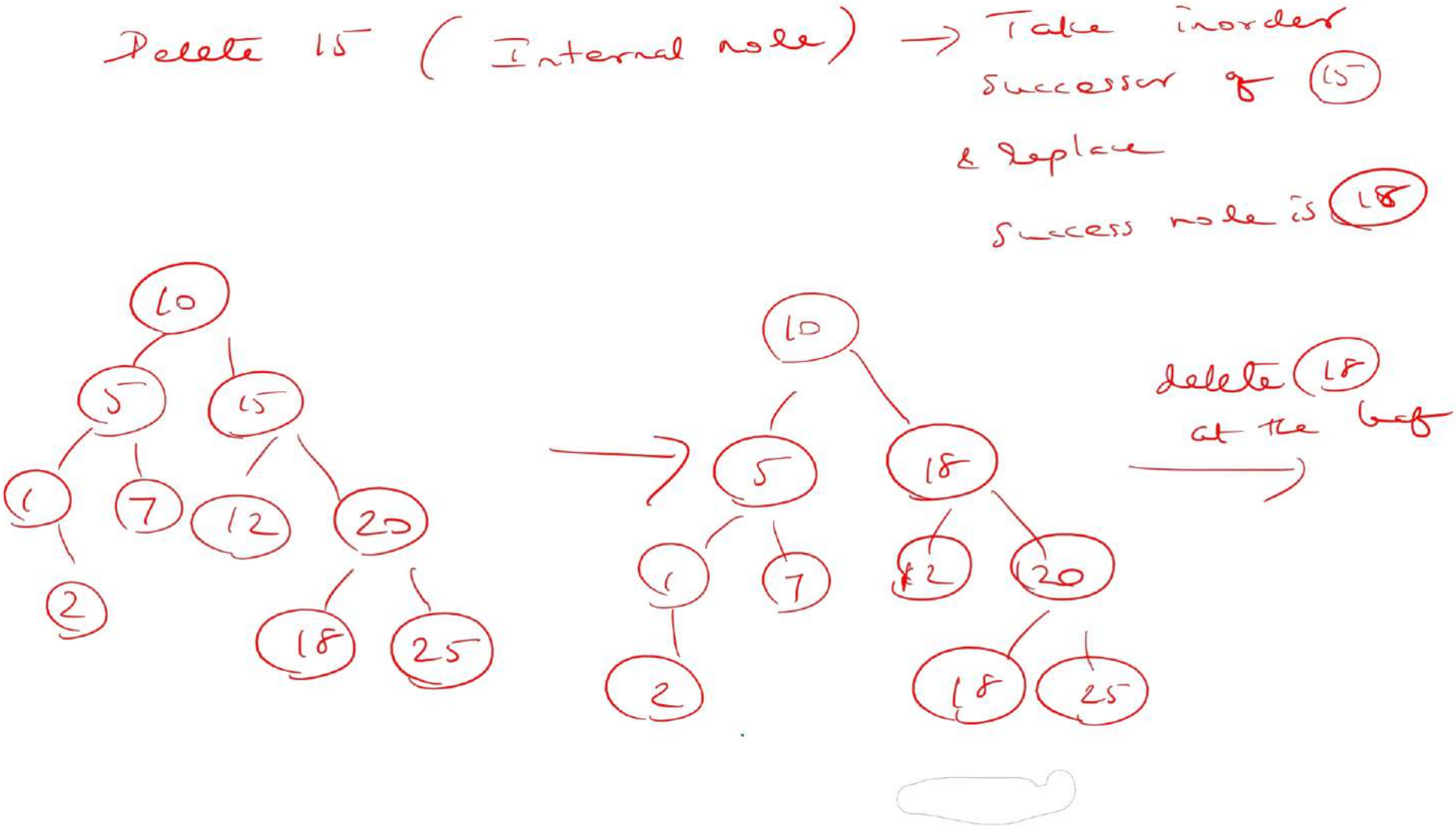
Splay tree

- **Deletion Operation in Splay Tree**



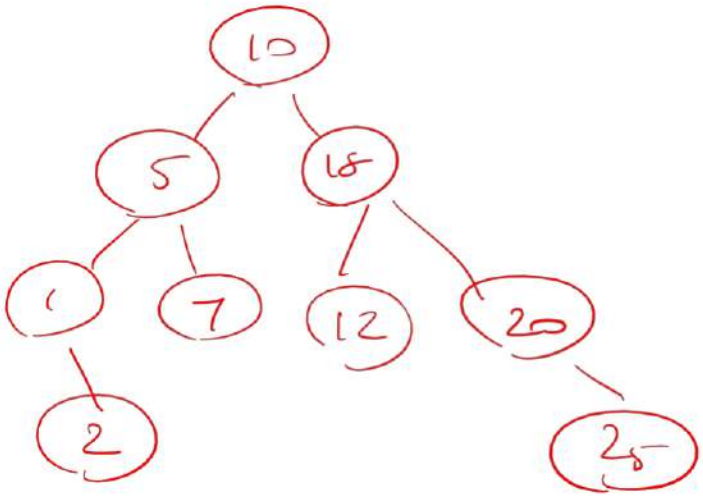
Splay tree

- **Deletion Operation in Splay Tree**



Splay tree

- **Deletion Operation in Splay Tree**



not supposed to perform any rotation as 10 must be the root

Red-black tree

Red-black tree

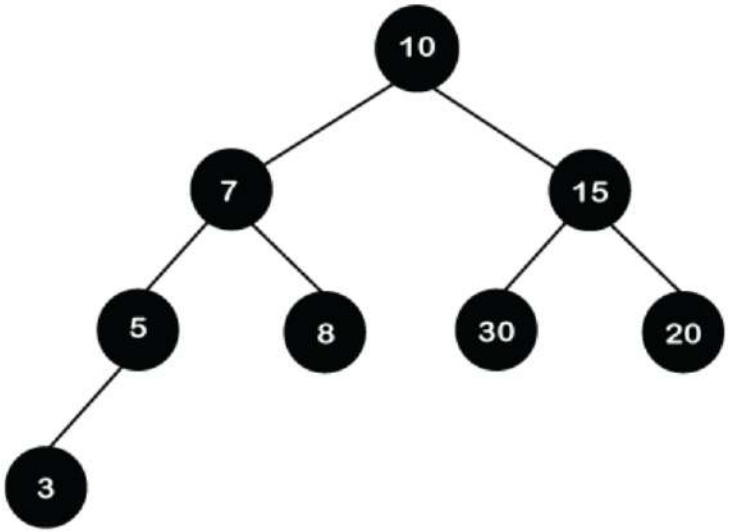
- A red-black tree is a type of self-balancing binary search tree.
- It maintains its balance through specific rules that ensure that the tree remains approximately balanced, which provides good performance for operations like insertion, deletion, and lookups.

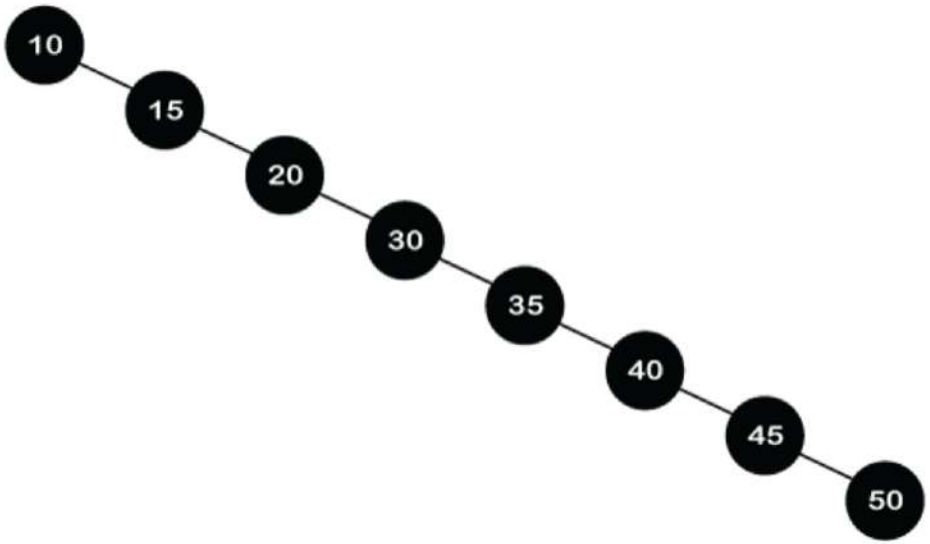
Red-black tree

- Each node in the Red-black tree contains an extra bit that represents a color to ensure that the tree is balanced during any operations performed on the tree like insertion, deletion, etc.

Red-black tree

- Different scenarios of a binary search tree





if we want to search the 80. We will first compare 80 with the root node. 80 is greater than the root node, i.e., 10, so searching will be performed on the right subtree. Again, 80 is compared with 15; 80 is greater than 15, so we move to the right of the 15, i.e., 20. Now, we reach the leaf node 20, and 20 is not equal to 80. Therefore, it will show that the element is not found in the tree. After each operation, the search is divided into half. The above BST will take O(logn) time to search the element.

If we want to search the 80 in the tree, we will compare 80 with all the nodes until we find the element or reach the leaf node. So, the above right-skewed BST will take **O(N)** time to search the element.

The first one is the balanced BST, whereas the second one is the unbalanced BST.

Red-black tree

- A balanced tree takes less time than an unbalanced tree for performing any operation on the tree.
- Therefore, we need a balanced tree, and the Red-Black tree is a self-balanced binary search tree.

Red-black tree

AVL vs Red-black

- Why do we require a Red-Black tree if AVL is also a height-balanced tree. The Red-Black tree is used because the AVL tree requires many rotations when the tree is large, whereas the Red-Black tree requires a maximum of two rotations to balance the tree. The main difference between the AVL tree and the Red-Black tree is that the AVL tree is strictly balanced, while the Red-Black tree is not completely height-balanced. So, the AVL tree is more balanced than the Red-Black tree, but the Red-Black tree guarantees $O(\log_2 n)$ time for all operations like insertion, deletion, and searching.

- Insertion is easier in the AVL tree as the AVL tree is strictly balanced, whereas deletion and searching are easier in the Red-Black tree as the Red-Black tree requires fewer rotations.

- Every AVL tree can be a Red-Black tree if we color each node either by Red or Black color. But every Red-Black tree is not an AVL because the AVL tree is strictly height-balanced while the Red-Black tree is not completely height-balanced.

Red-black tree

- The node is either colored in **Red** or **Black** color. Sometimes no rotation is required, and only recoloring is needed to balance the tree.

Red-black tree

Properties of Red-Black tree

- It is a self-balancing Binary Search tree. Here, self-balancing means that it balances the tree itself by either doing the rotations or recoloring the nodes.
- This tree data structure is named as a Red-Black tree as each node is either Red or Black in color. Every node stores one extra information known as a bit that represents the color of the node. For example, 0 bit denotes the black color while 1 bit denotes the red color of the node. Other information stored by the node is similar to the binary tree, i.e., data part, left pointer and right pointer.
- In the Red-Black tree, the root node is always black in color.
- In a binary tree, we consider those nodes as the leaf which have no child. In contrast, in the Red-Black tree, the nodes that have no child are considered the internal nodes and these nodes are connected to the NIL nodes that are always black in color. The NIL nodes are the leaf nodes in the Red-Black tree.
- If the node is Red, then its children should be in Black color. In other words, we can say that there should be no red-red parent-child relationship.
- Every path from a node to any of its descendant's NIL node should have same number of black nodes.

Red-black tree

Properties of Red-Black tree

- Every node is either red or black
- Root is black
- Every NULL node is black
- If a node is red, then both it's children are black
- Every path from root to any leaf node contains same number of black nodes

Red-black tree

Insertion in Red Black tree

- **The following are some rules used to create the Red-Black tree:**

1. If the tree is empty, then we create a new node as a root node with the color black.
2. If the tree is not empty, then we create a new node as a leaf node with a color red.
3. If the parent of a new node is black, then insert new node and exit.
4. If the parent of a new node is Red, then we have to check the color of the parent's sibling of a new node.

- 4a) If the color is Black, then we perform rotations and recoloring.
- 4b) If the color is Red then we recolor the node. We will also check whether the parents' parent of a new node is the root node or not; if it is not a root node, we will recolor and recheck the node.

Red-black tree

Insertion Rule

- If tree is empty – insert new node as black
- If tree is not empty – insert new node as red
- If parent of new node is black, insert new node (red) as per BST rule and exit(ie, carry on with next iteration)
- If parent of new node is red, then check the parent's sibling
  - Sibling of parent is black or NULL
    - (i) rotate (AVL rotation)
    - (ii) recolor – parent
      - grand parent
  - Sibling of parent is red
    - (i) recolor – parent
      - parent's sibling
    - (ii) check if grand parent of new node is root node or not
      if not, then recolor grandparent and recheck

Red-black tree

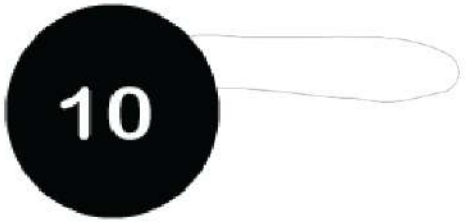
R-R conflict (Red-Red conflict) : It's between parent and child (not between children)

Red-black tree

- **Insertion**

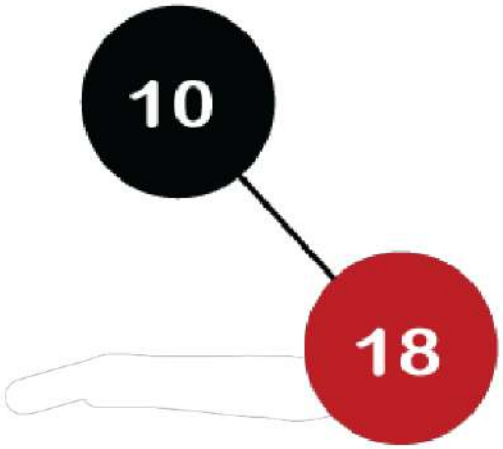
10, 18, 7, 15, 16, 30, 25, 40, 60

Step 1: Initially, the tree is empty, so we create a new node having value 10. This is the first node of the tree, so it would be the root node of the tree. The root node must be black in color, which is shown below:



Red-black tree

- **Step 2:** The next node is 18. As 18 is greater than 10 so it will come at the right of 10 as shown below.

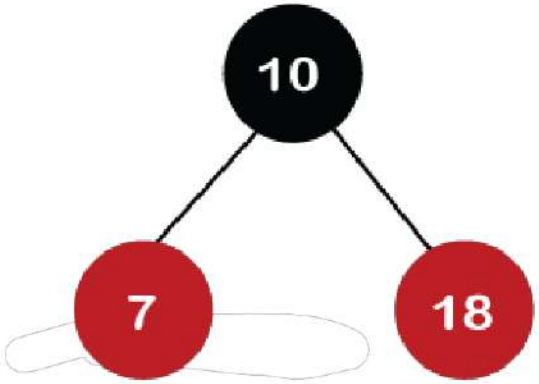


We know the second rule of the Red Black tree that if the tree is not empty then the newly created node will have the **Red** color. Therefore, node 18 has a Red color.

Now we verify the third rule of the Red-Black tree, i.e., the parent of the new node is black or not. In the above figure, the parent of the node is black in color; therefore, it is a Red-Black tree.

Red-black tree

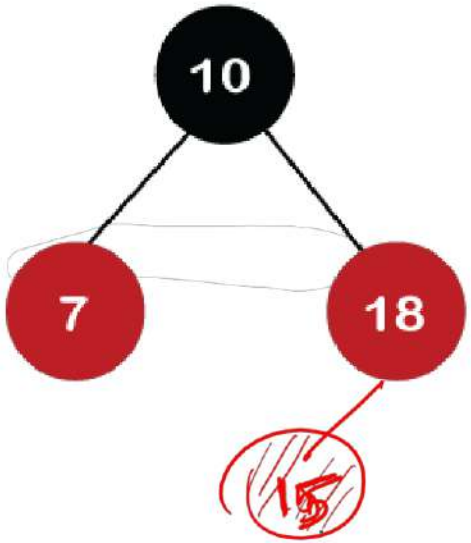
Step 3: Now, we create the new node having value 7 with Red color. As 7 is less than 10, it will come at the left of 10 as shown below.



Now we verify the third rule of the Red-Black tree, i.e., the parent of the new node is black or not. As we can observe, the parent of the node 7 is black in color, and it obeys the Red-Black tree's properties.

Red-black tree

Step 4: The next element is 15, and 15 is greater than 10, but less than 18, so the new node will be created at the left of node 18. The node 15 would be Red in color as the tree is not empty.

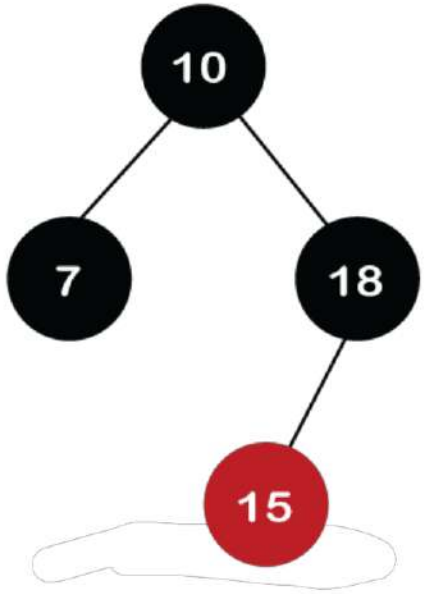


Red-black tree

The above tree violates the property of the Red-Black tree as it has Red-red parent-child relationship. Now we have to apply some rule to make a Red-Black tree.

The rule 4 says that ***if the new node's parent is Red, then we have to check the color of the parent's sibling of a new node.*** The new node is node 15; the parent of the new node is node 18 and the sibling of the parent node is node 7. As the color of the parent's sibling is Red in color, so we apply the rule 4a. The rule 4a says that we have to recolor both the parent and parent's sibling node. So, both the nodes, i.e., 7 and 18, would be recolored as shown in the below figure.

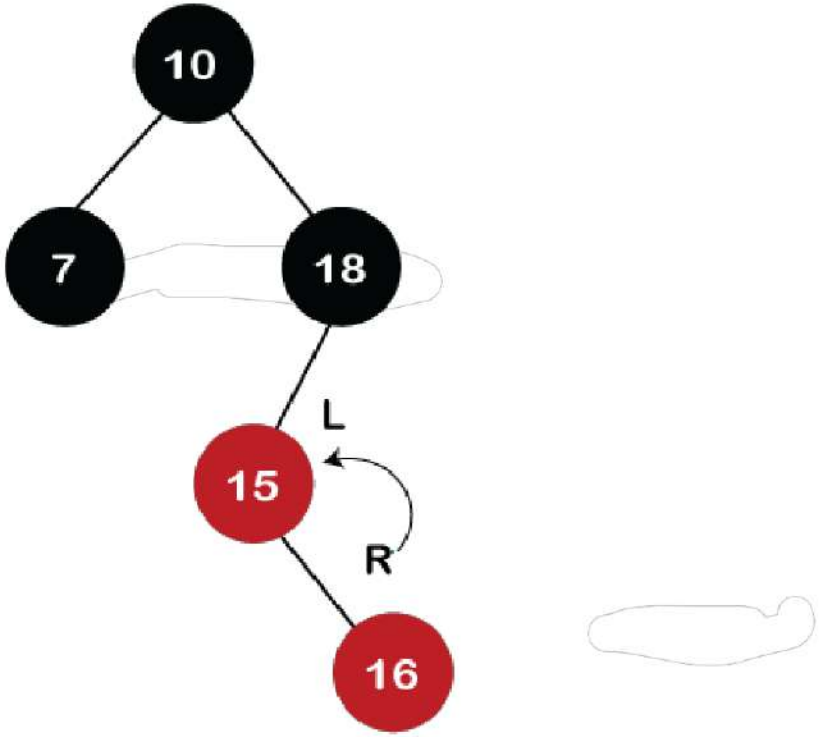
Red-black tree



We also have to check whether the parent's parent of the new node is the root node or not. Here, the parent's parent of a new node is the root node, so we do not need to recolor it.

Red-black tree

Step 5: The next element is 16. As 16 is greater than 10 but less than 18 and greater than 15, so node 16 will come at the right of node 15. The tree is not empty; node 16 would be Red in color, as shown in the below figure:



Red-black tree

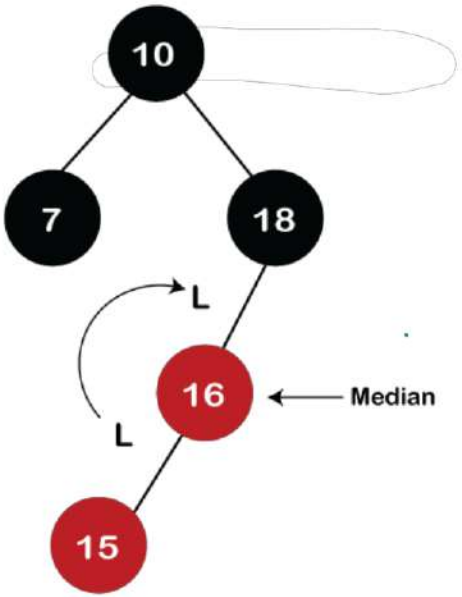
In the above figure, we can observe that it violates the property of the parent-child relationship as it has a red-red parent-child relationship.

We have to apply some rules to make a Red-Black tree.

Since the new node's parent is Red color, and the parent of the new node has no sibling, so rule **4a** will be applied. The rule **4a** says that some rotations and recoloring would be performed on the tree.

Red-black tree

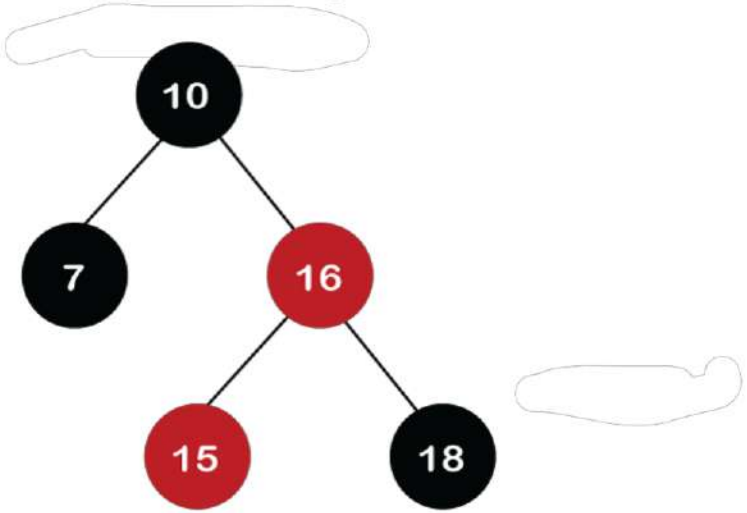
Since node 16 is right of node 15 and the parent of node 15 is node 18. Node 15 is the left of node 18. Here we have **an LR** relationship, so we require to perform two rotations. First, we will perform left, and then we will perform the right rotation. The left rotation would be performed on nodes 15 and 16, where node 16 will move upward, and node 15 will move downward. Once the left rotation is performed, the tree looks like as shown in the below figure:



Red-black tree

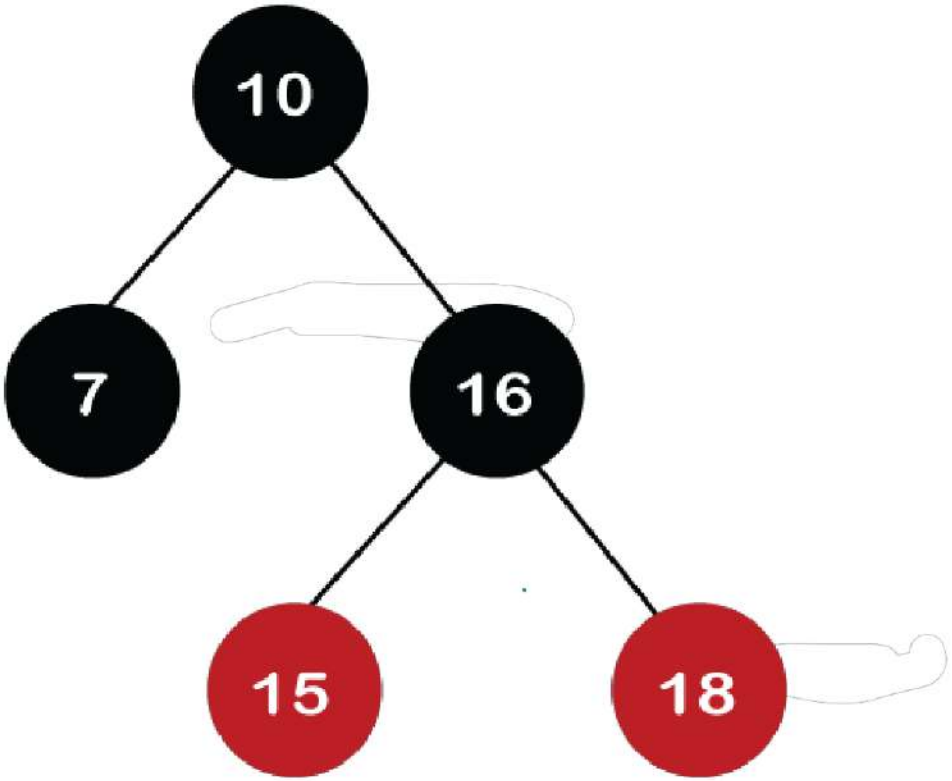
In the above figure, we can observe that there is **an LL** relationship. The above tree has a Red-red conflict, so we perform the right rotation.

When we perform the right rotation, the median element would be the root node. Once the right rotation is performed, node 16 would become the root node, and nodes 15 and 18 would be the left child and right child, respectively, as shown in the below figure.



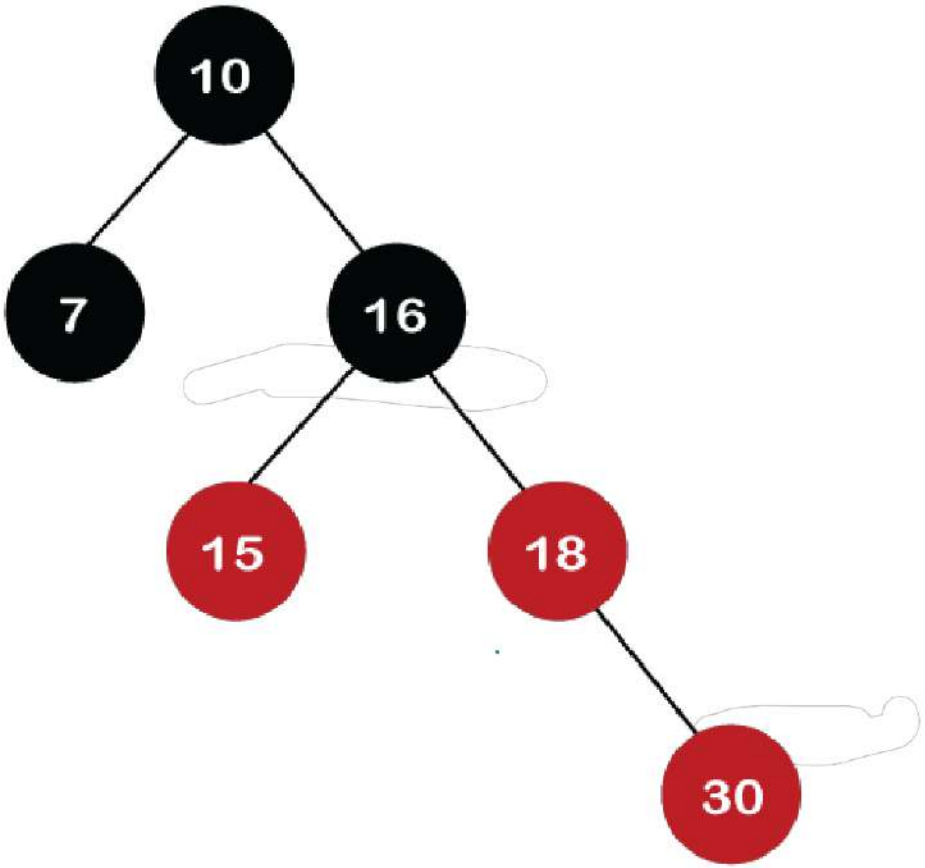
Red-black tree

After rotation, node 16 and node 18 would be recolored; the color of node 16 is red, so it will change to black, and the color of node 18 is black, so it will change to a red color as shown in the below figure:



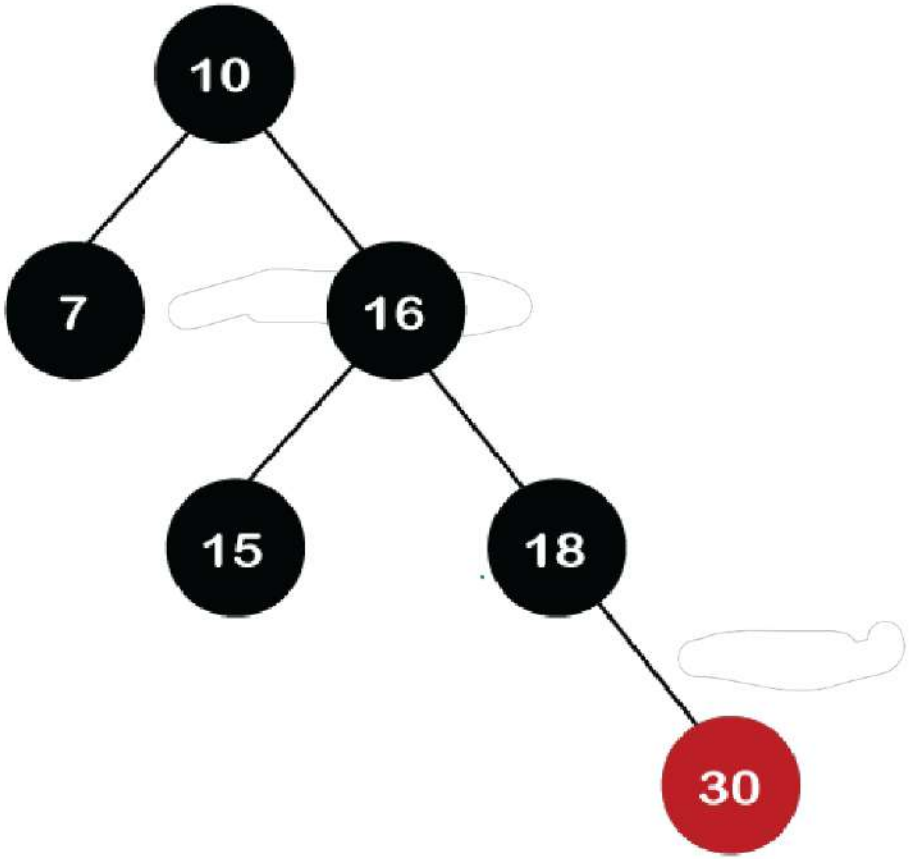
Red-black tree

Step 6: The next element is 30. Node 30 is inserted at the right of node 18. As the tree is not empty, so the color of node 30 would be red.



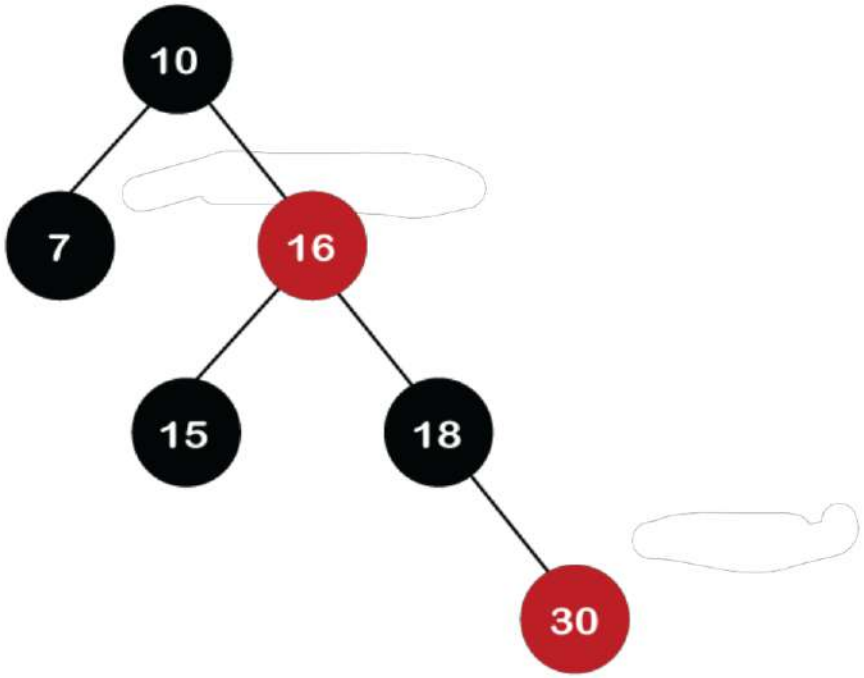
Red-black tree

The color of the parent and parent's sibling of a new node is Red, so rule 4b is applied. In rule 4b, we have to do only recoloring, i.e., no rotations are required. The color of both the parent (node 18) and parent's sibling (node 15) would become black, as shown in the below image.



Red-black tree

We also have to check the parent's parent of the new node, whether it is a root node or not. The parent's parent of the new node, i.e., node 30 is node 16 and node 16 is not a root node, so we will recolor the node 16 and changes to the Red color. The parent of node 16 is node 10, and it is not in Red color, so there is no Red-red conflict.

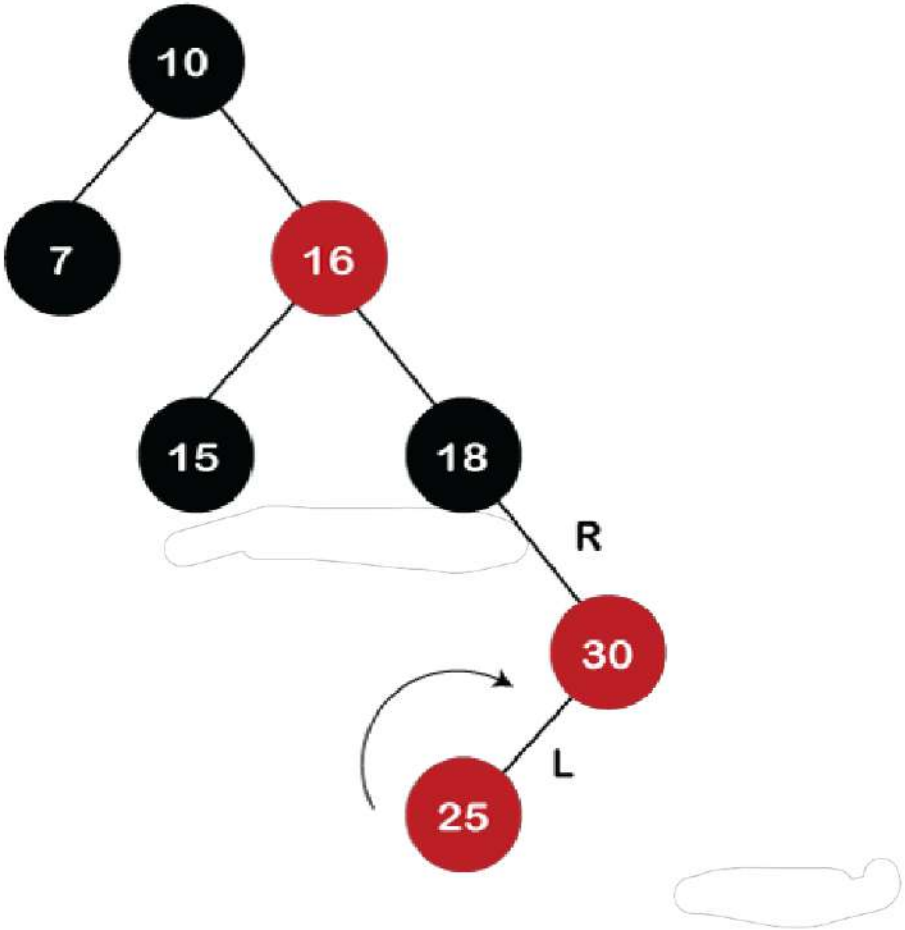


Red-black tree

Step 7: The next element is 25, which we have to insert in a tree. Since 25 is greater than 10, 16, 18 but less than 30; so, it will come at the left of node 30. As the tree is not empty, node 25 would be in Red color. Here Red-red conflict occurs as the parent of the newly created is Red color.

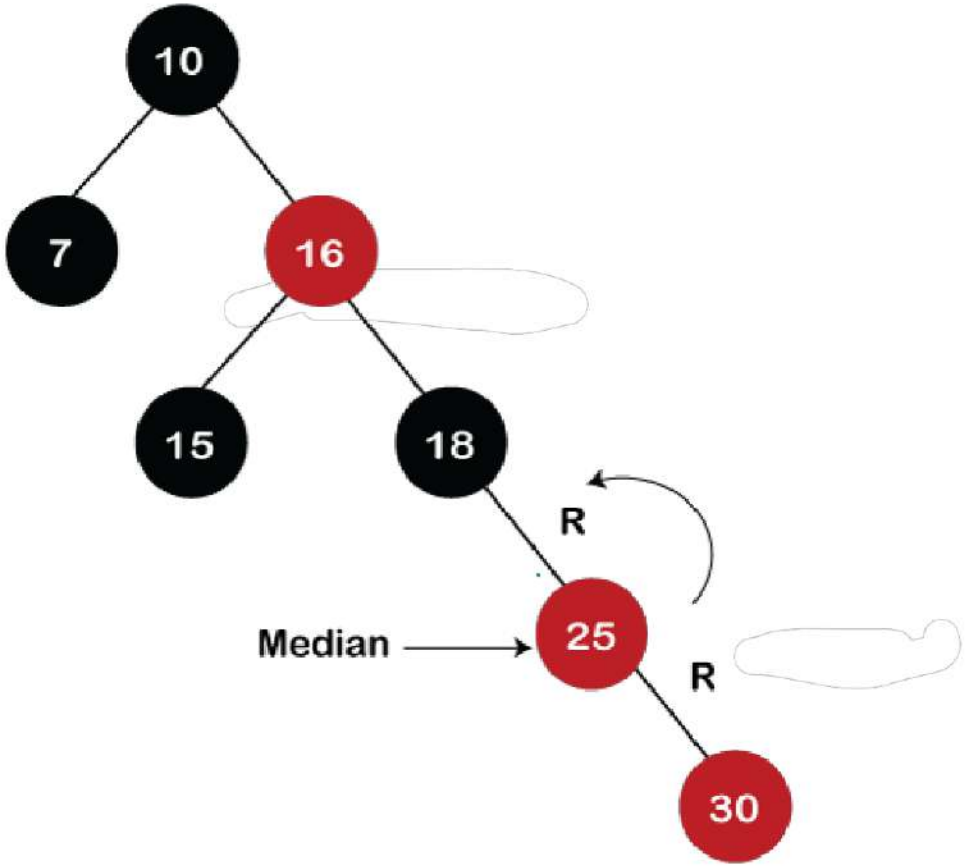
Since there is no parent's sibling, so rule 4a is applied in which rotation, as well as recoloring, are performed. First, we will perform rotations. As the newly created node is at the left of its parent and the parent node is at the right of its parent, so the RL relationship is formed. Firstly, the right rotation is performed in which node 25 goes upwards, whereas node 30 goes downwards, as shown in the below figure.

Red-black tree



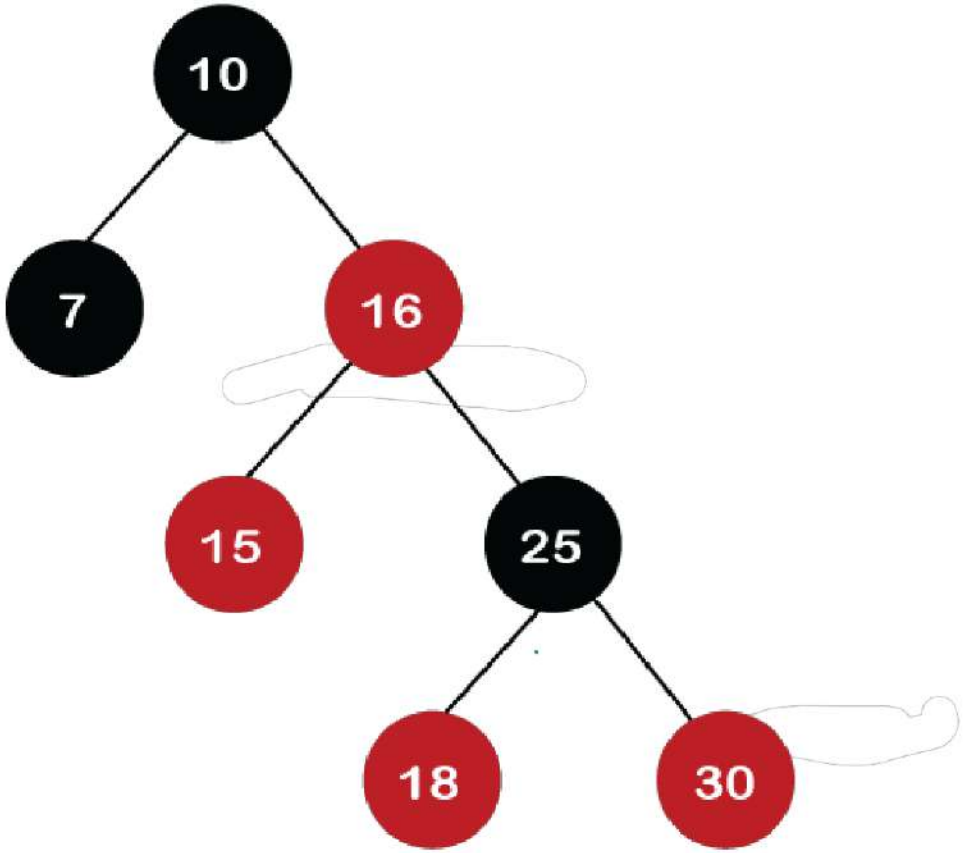
Red-black tree

After the first rotation, there is an RR relationship, so left rotation is performed. After right rotation, the median element, i.e., 25 would be the root node; node 30 would be at the right of 25 and node 18 would be at the left of node 25.



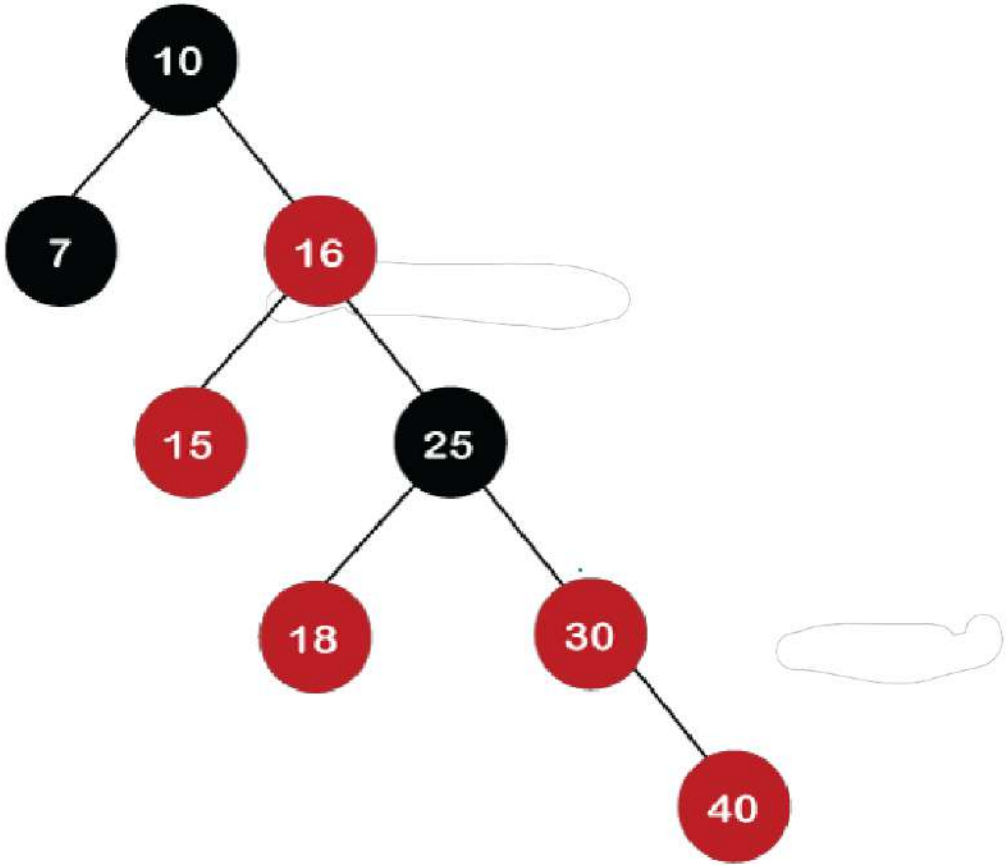
Red-black tree

Now recoloring would be performed on nodes 25 and 18; node 25 becomes black in color, and node 18 becomes red in color.



Red-black tree

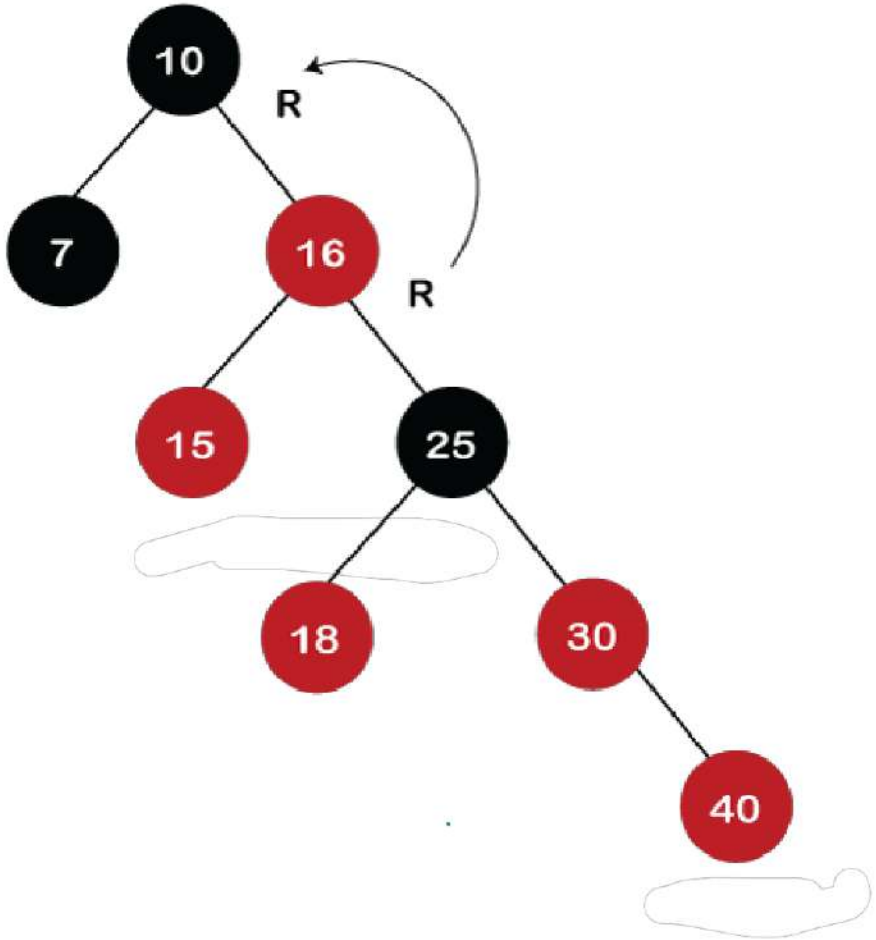
Step 8: The next element is 40. Since 40 is greater than 10, 16, 18, 25, and 30, so node 40 will come at the right of node 30. As the tree is not empty, node 40 would be Red in color. There is a Red-red conflict between nodes 40 and 30, so rule 4b will be applied.



Red-black tree

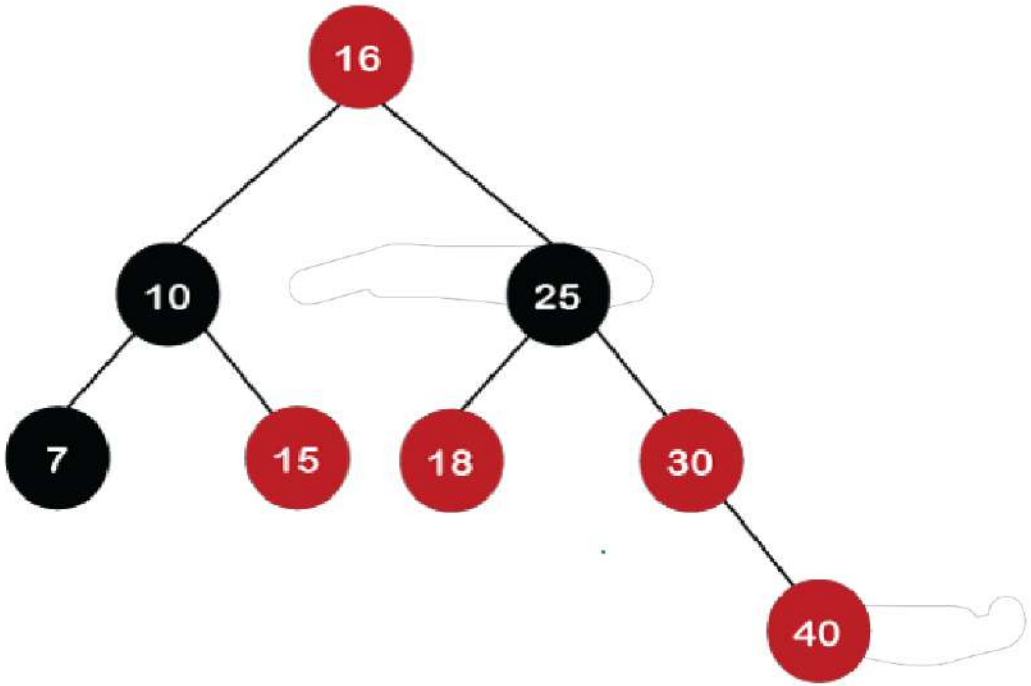
- As the color of parent and parent's sibling node of a new node is Red so recoloring would be performed. The color of both the nodes would become black, as shown in the below image.
- After recoloring, we also have to check the parent's parent of a new node, i.e., 25, which is not a root node, so recoloring would be performed, and the color of node 25 changes to Red.
- After recoloring, red-red conflict occurs between nodes 25 and 16. Now node 25 would be considered as the new node. Since the parent of node 25 is red in color, and the parent's sibling is black in color, rule 4a would be applied. Since 25 is at the right of the node 16 and 16 is at the right of its parent, so there is an RR relationship. In the RR relationship, left rotation is performed. After left rotation, the median element 16 would be the root node, as shown in the below figure.

Red-black tree

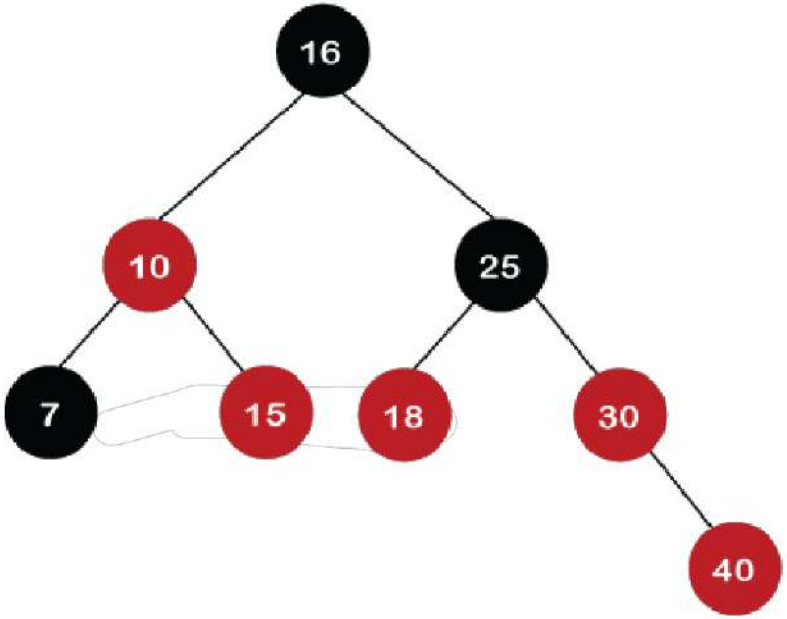


Red-black tree

After rotation, recoloring is performed on nodes 16 and 10. The color of node 10 and node 16 changes to Red and Black, respectively as shown in the below figure.



Red-black tree

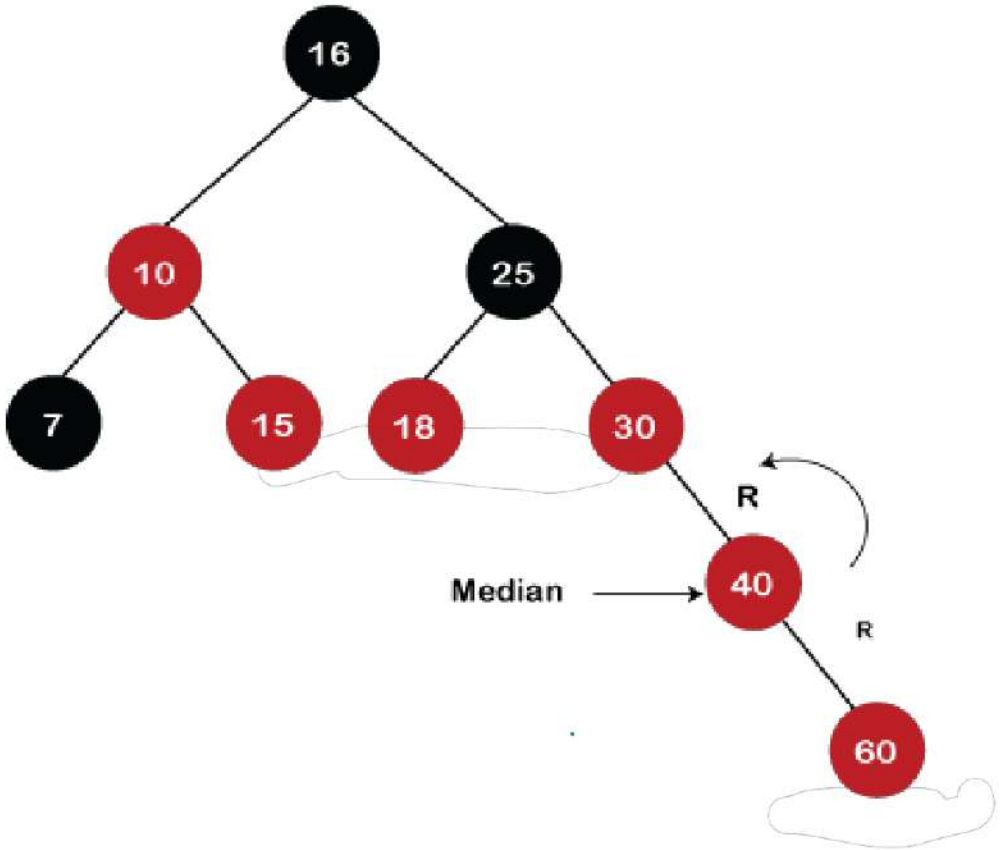


Red-black tree

Step 9: The next element is 60. Since 60 is greater than 16, 25, 30, and 40, so node 60 will come at the right of node 40. As the tree is not empty, the color of node 60 would be Red.

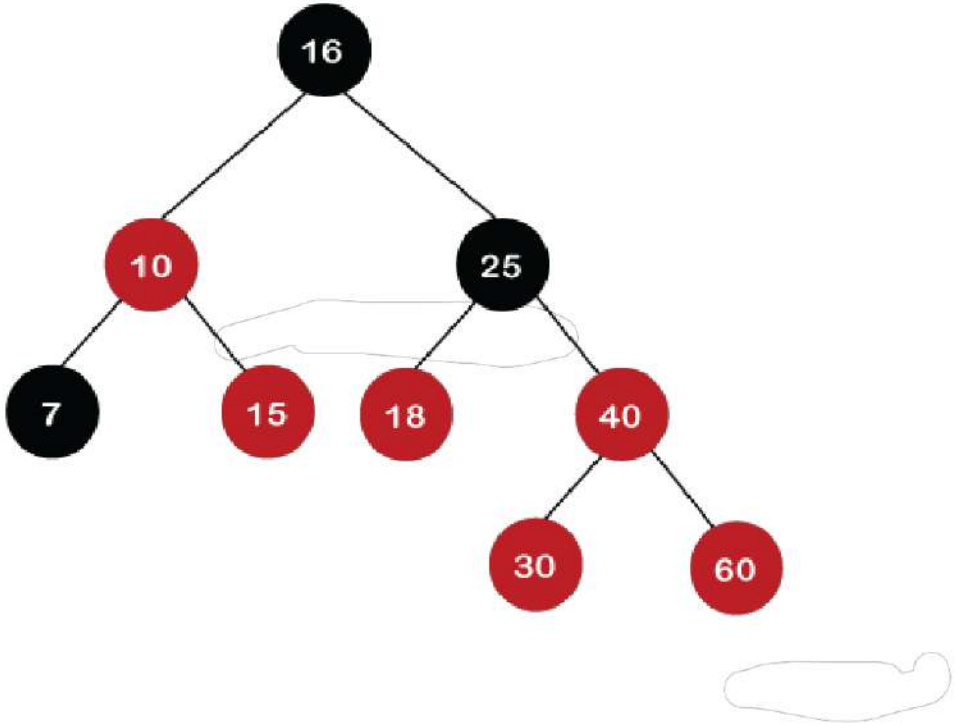
As we can observe in the above tree that there is a Red-red conflict occurs. The parent node is Red in color, and there is no parent's sibling exists in the tree, so rule 4a would be applied. The first rotation would be performed. The RR relationship exists between the nodes, so left rotation would be performed.

Red-black tree



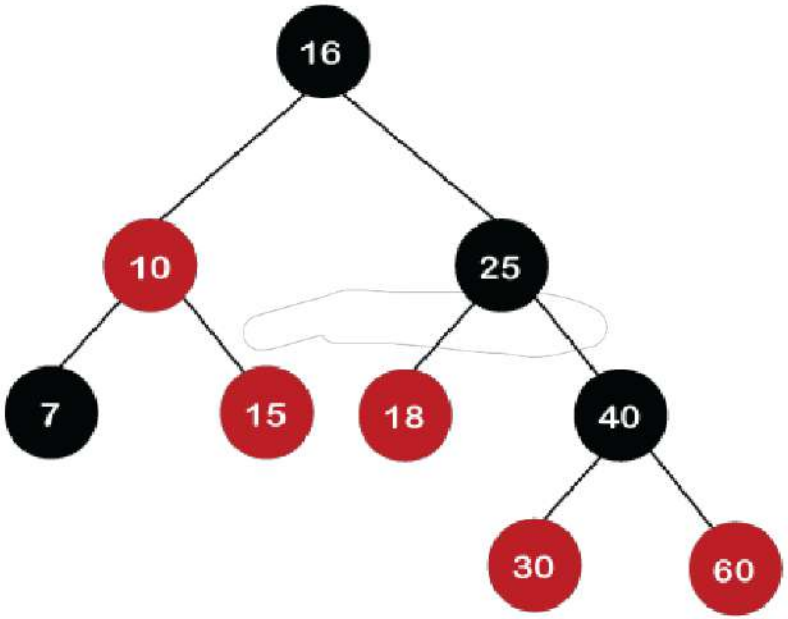
Red-black tree

When left rotation is performed, node 40 will come upwards, and node 30 will come downwards, as shown in the below figure:



Red-black tree

After rotation, the recoloring is performed on nodes 30 and 40. The color of node 30 would become Red, while the color of node 40 would become black.



The above tree is a Red-Black tree as it follows all the Red-Black tree properties.

Red-black tree

Insertion Rule

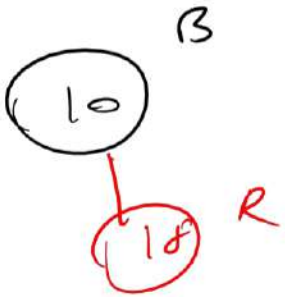
- If tree is empty – insert new node as black
- If tree is not empty – insert new node as red
- If parent of new node is black, insert new node (red) as per BST rule and exit(ie, carry on with next iteration)
- If parent of new node is red, then check the parent's sibling
  - Sibling of parent is black or NULL
    - (i) rotate (AVL rotation)
    - (ii) recolor – parent
      - grand parent
  - Sibling of parent is red
    - (i) recolor – parent
      - parent's sibling
    - (ii) check if grand parent of new node is root node or not
      if not, then recolor grandparent and recheck

Red-black tree

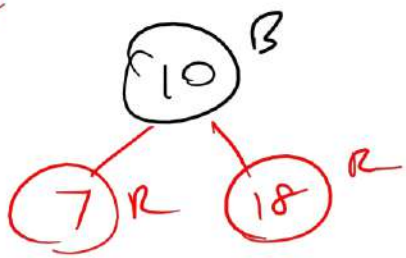
Insert 10, 18, 7, 15, 16, 30, 25

10

18

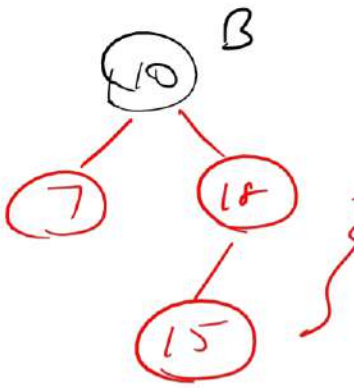


7



Red-black tree

15



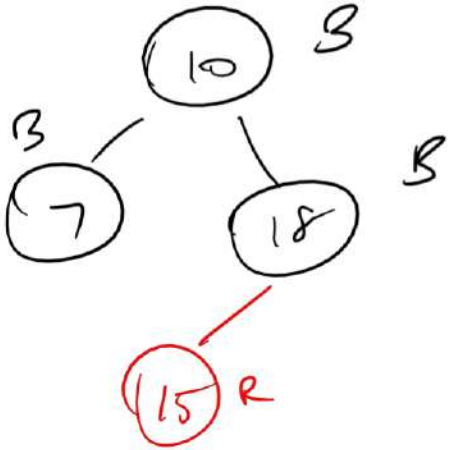
R-R conflict

new node : 15 (R)

Parent : 18 (R)

Parent's sibling : 7 (B)

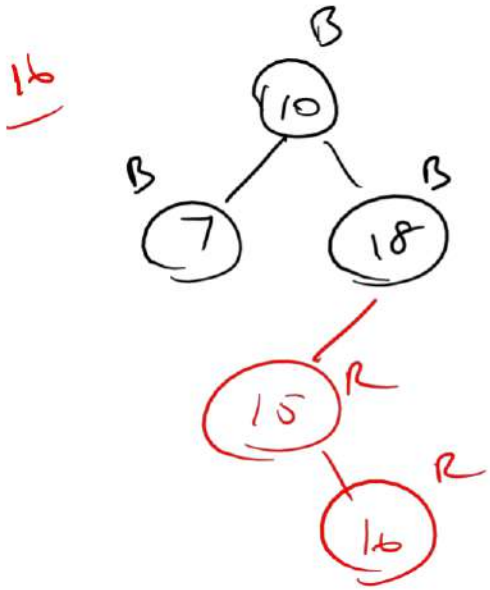
(i) Recolor parent & parent's sibling



(ii) grandparent : 10 (B)

It is root node, so no recoloring!

Red-black tree



newnode : 16

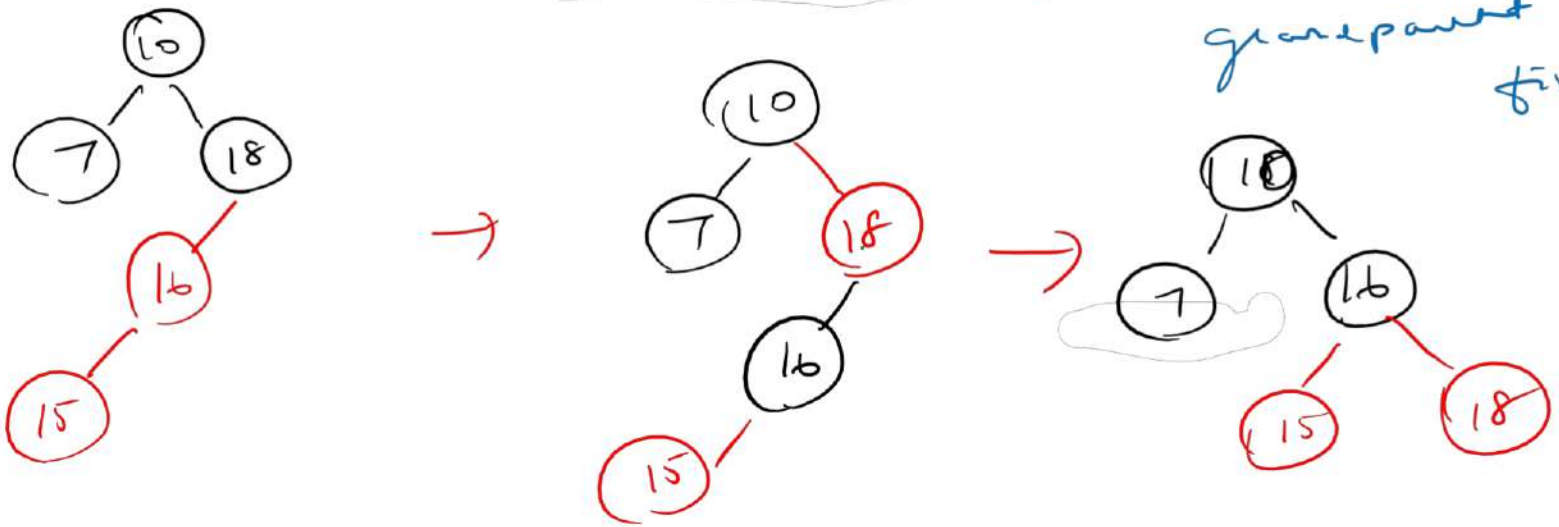
Parent : 15

Sibling of parent : NULL

(i) Rotate (AVL)

(ii) Recolor parent & grandparent (after first rotate)

LR rotate



Red-black tree

In every step, we can check the number of black nodes from root to leaf to see whether they have the same

No. of black nodes

Q 10 - 7 → 2

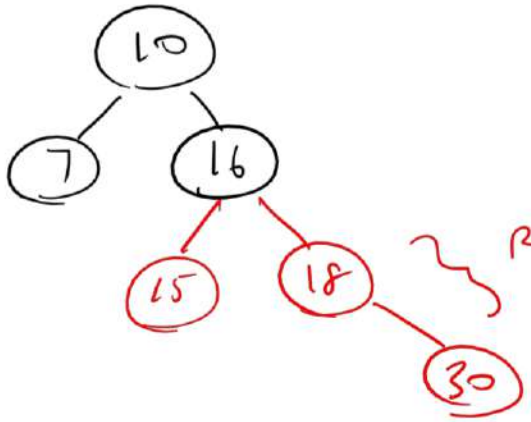
10 - 10 - 15 → 2

10 - 16 - 18 → 2

So it is balanced

Red-black tree

30



R-R conflict

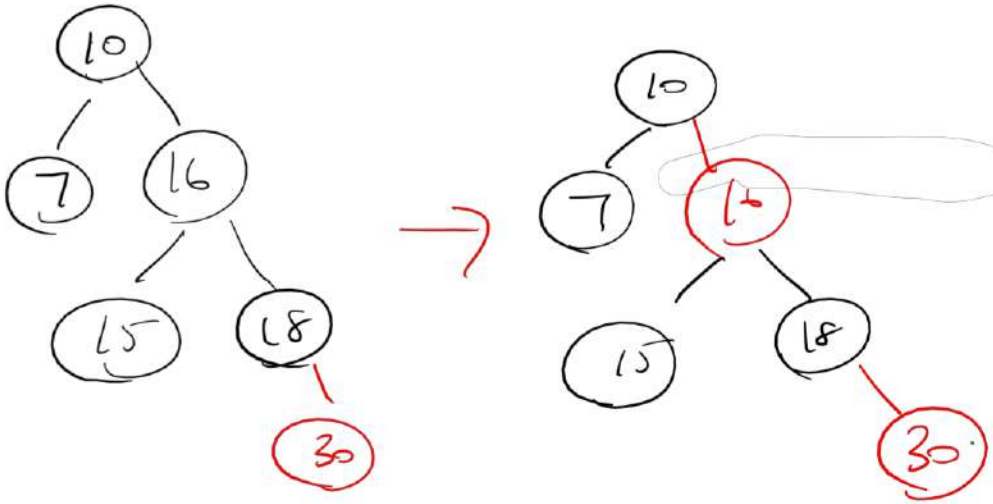
new node : 30 (R)

Parent : 18 (R)

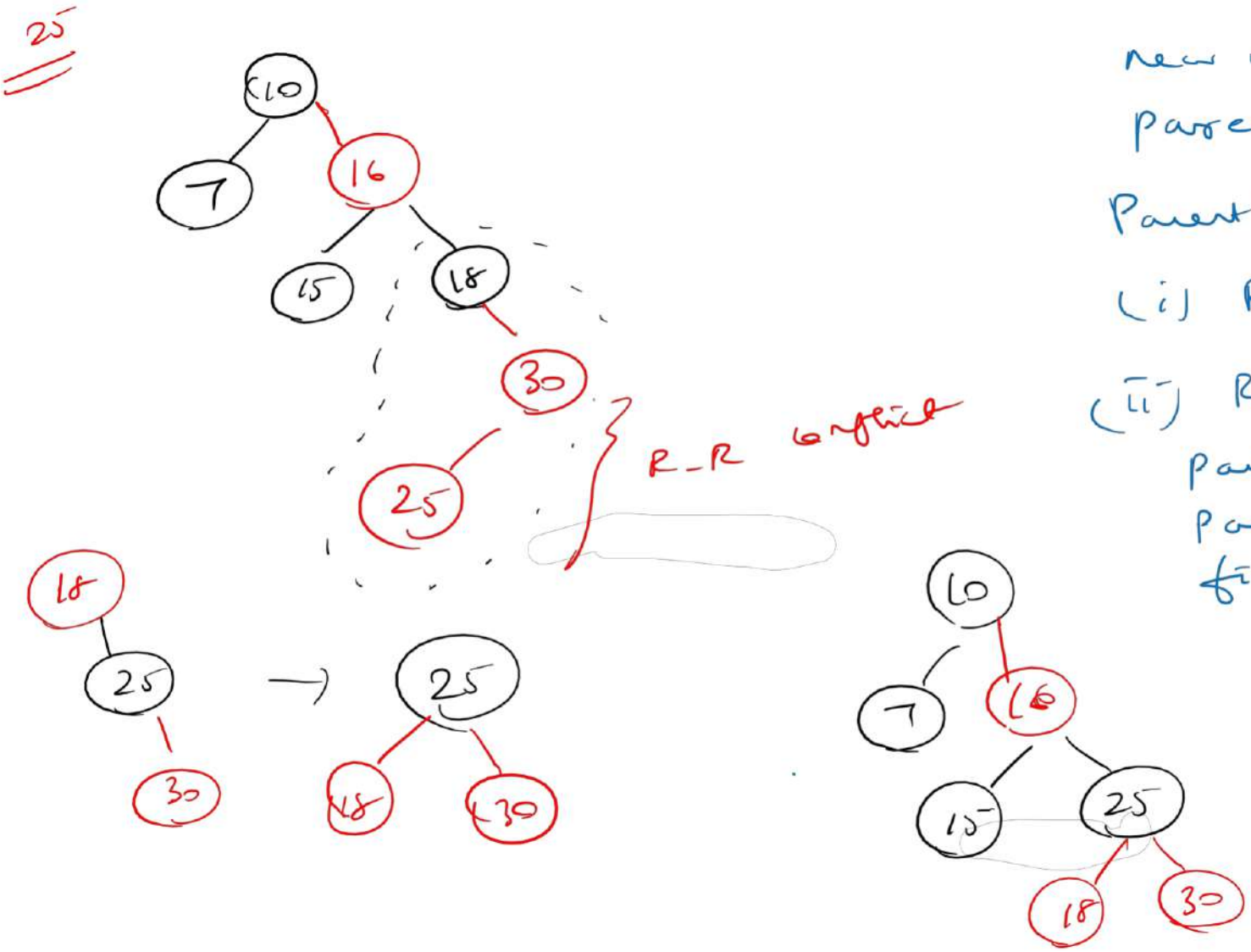
Parent's sibling : 15 (R)

i) Recolor parent & Parent's sibling

ii) It's grandparent 16 is not root, so recolor it



Red-black tree



New node : 25

Parent : 30

Parent's sibling : NULL

(i) Rotate

(ii) Recolor

Parent & grand parent after first rotation

Red-black tree

Properties

1. **Node Color**: Each node is either red or black.
2. **Root Property**: The root is always black.
3. **Red Property**: Red nodes cannot have red children (i.e., no two red nodes can be adjacent).
4. **Black Height Property**: Every path from a node to its descendant leaves must have the same number of black nodes.
5. **Leaf Nodes**: All leaf nodes (NIL nodes) are black.

Red-black tree

Operations

- **Insertion**: When inserting a new node, you initially add it as a red node. After insertion, you may need to perform rotations and color changes to maintain the red-black properties.
- **Deletion**: Removing a node involves adjusting the tree to maintain its properties, which may involve rotations and color changes similar to insertion.
- **Searching**: Searching in a red-black tree is similar to searching in a regular binary search tree, with the added benefit that the tree remains balanced.

Red-black tree

Algorithms

- **Rotations**: Rotations (left and right) are used to maintain the balance of the tree. They help in adjusting the structure without violating the binary search tree properties.
- **Fix-Up**: After insertion or deletion, the tree may need to be rebalanced using a series of color changes and rotations to restore its properties.

Red-black tree

Time Complexity

- **Search**: O(log n)
- **Insertion**: O(log n)
- **Deletion**: O(log n)

Range tree

Range tree

- A Range Tree is designed to efficiently handle range queries, particularly in multidimensional space.
- It allows you to quickly find all points (or records) that lie within a given range.
- Range trees are commonly used in computational geometry and spatial databases for tasks like searching within a rectangle in 2D space, or higher-dimensional generalizations.

Range tree

Dimensions:

•Range trees are used for handling multidimensional data. A 1D range tree handles queries in a single dimension, while a 2D range tree handles two dimensions, and so on.

Range tree

Range Query:

- A range query involves searching for all points that lie within a given rectangular (or hyperrectangular) region.
- The query is processed in stages, corresponding to each dimension.

Range tree

Range tree:

A Range tress is a balanced BST in which at any node, the left sub-tree will have the values less than or equal to the node value and in the right tree the value is greater than the node value and with this the tree is traversed and the range query is addressed

Range tree

Range tree:

Range Trees can only be employed for range queries.

Two properties:

1. Values are stored only in leaf nodes
2. All other nodes contain the highest value of its' left subtree

1D Range tree

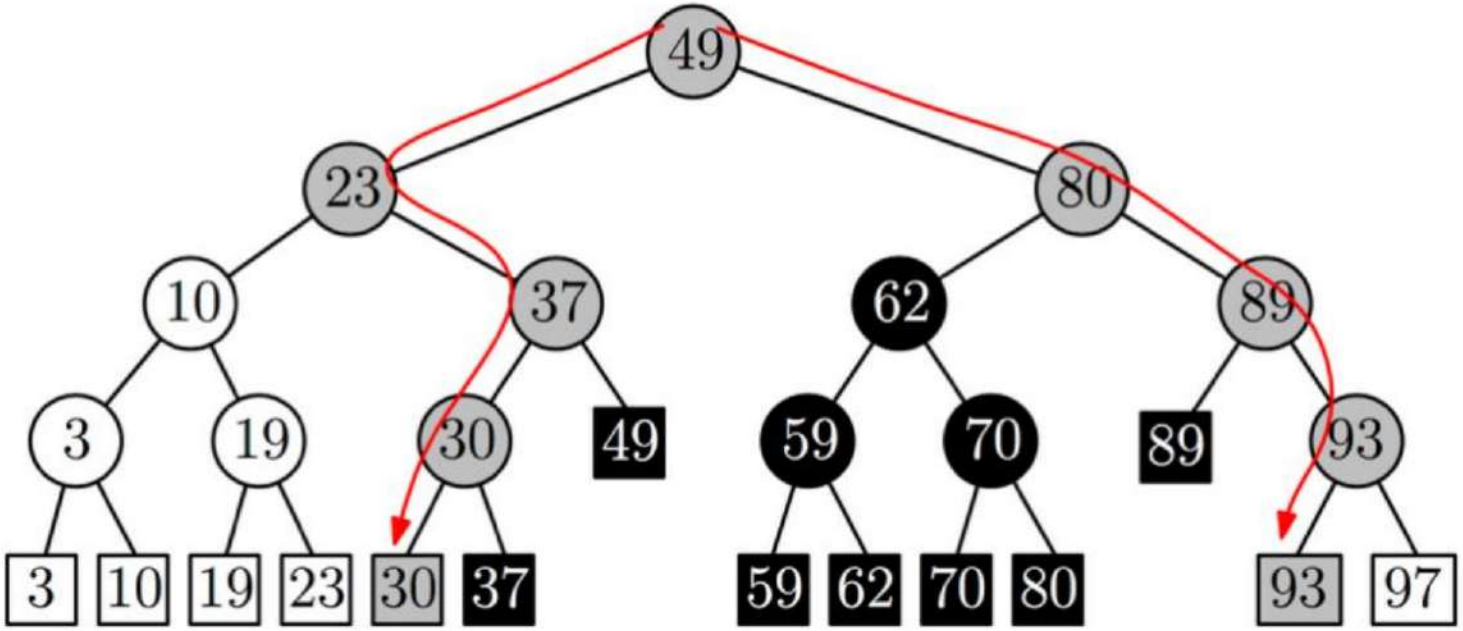
1D

- P is a set of points on the real line
- A range query is an interval

Range tree

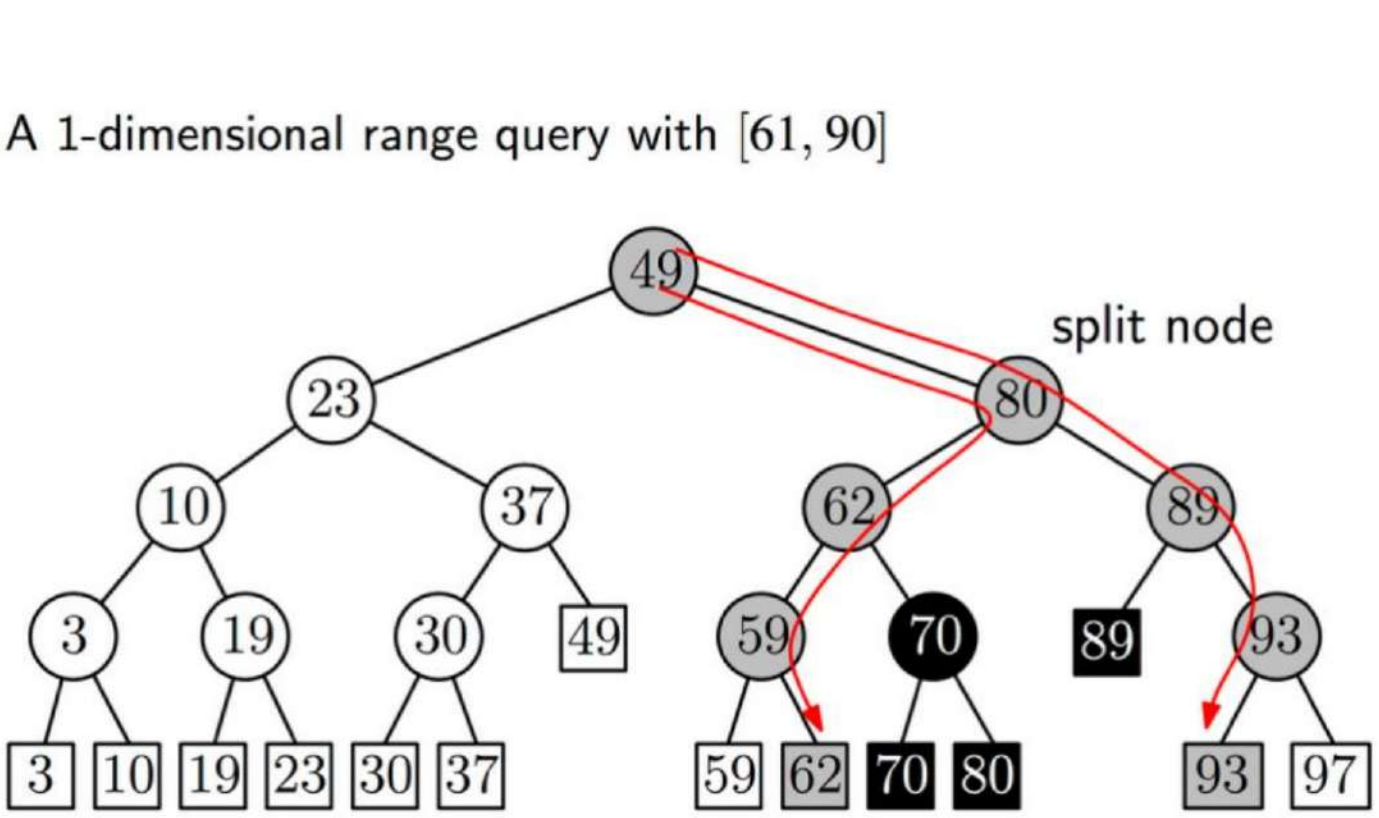
1D: different visual

A 1-dimensional range query with $[25, 90]$



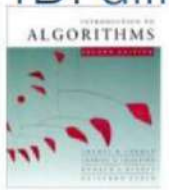
Range tree

1D: different visual



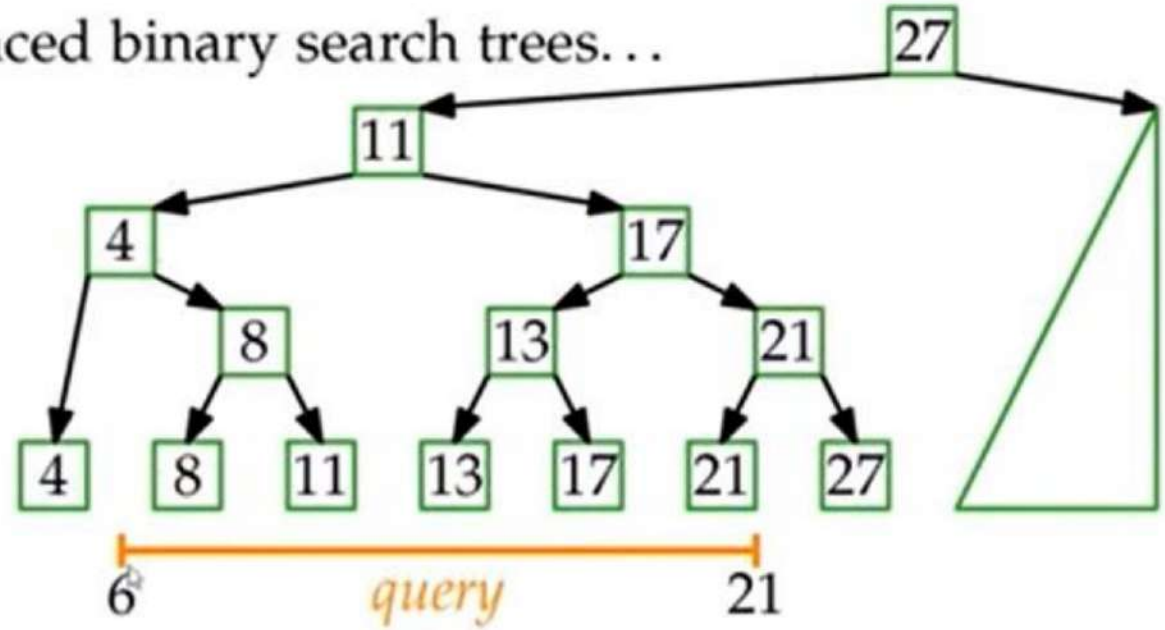
Range tree

1D: different visual



Range tree

Solution: balanced binary search trees...



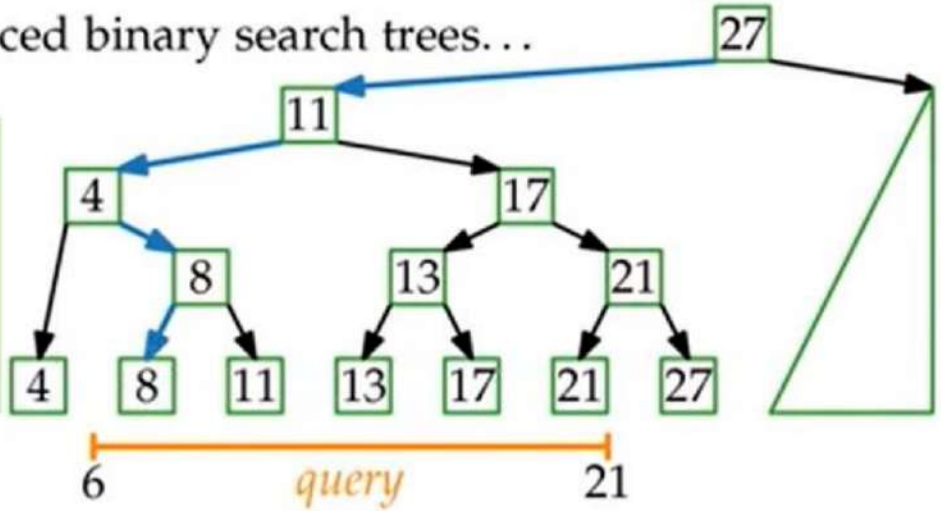
– keys only in leaves

– inner nodes store max. of their left subtrees

Range tree

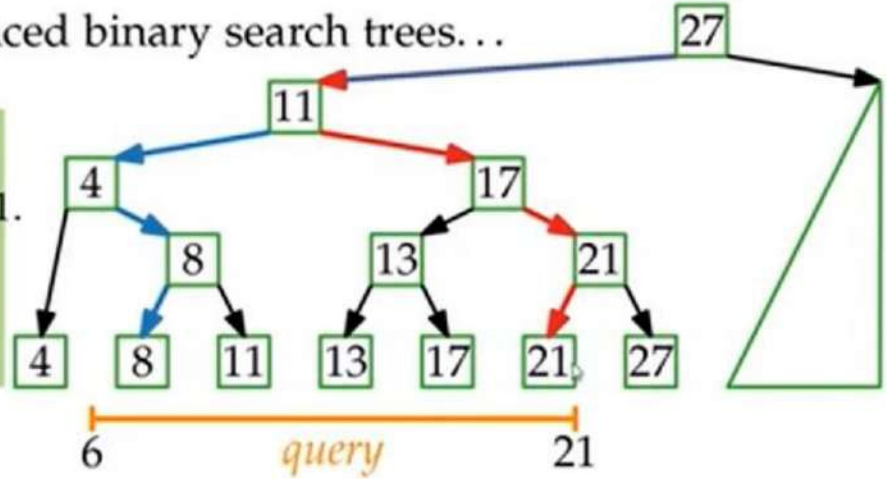
Solution: balanced binary search trees...

1. Search $x = 6$.



Solution: balanced binary search trees...

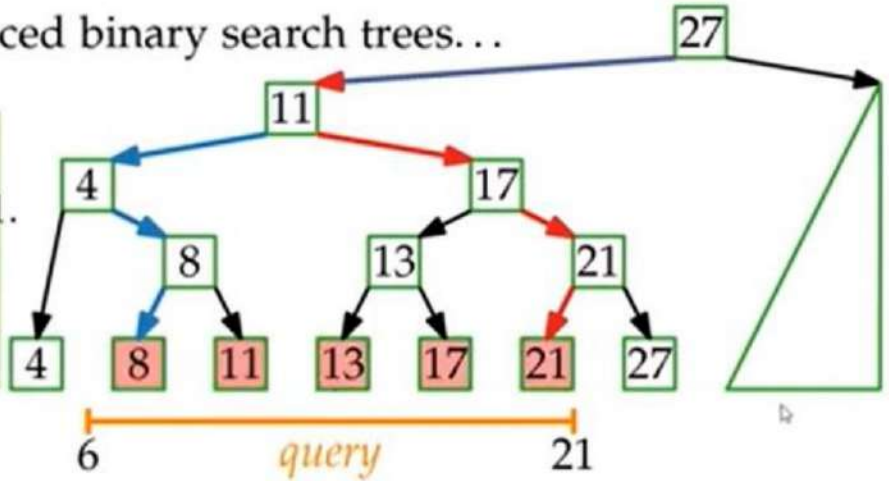
1. Search $x = 6$.
2. Search $x' = 21$.



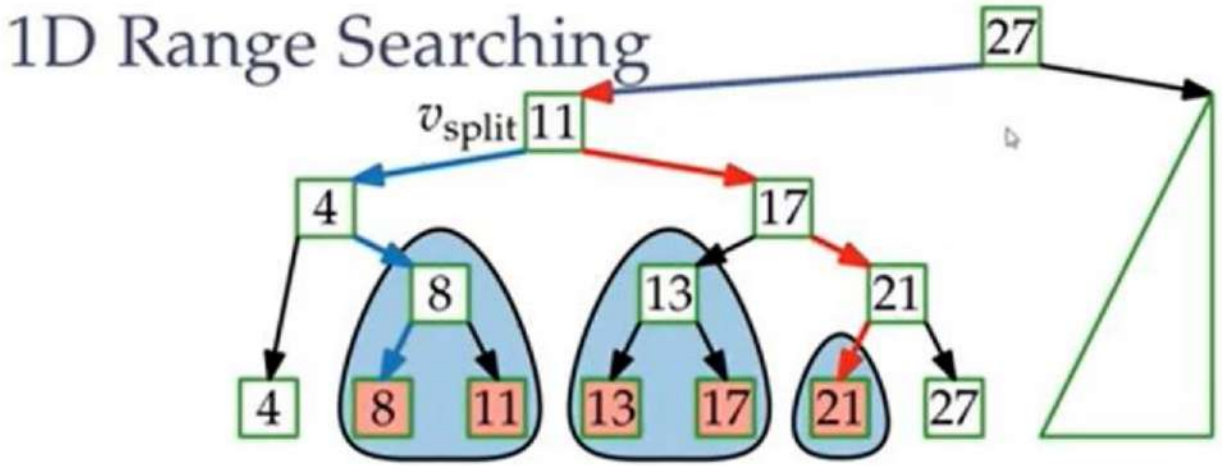
Range tree

Solution: balanced binary search trees...

1. Search $x = 6$.
2. Search $x' = 21$.
3. Return all leaves 'inbetween".



Range tree



Observe: The result of a query is the disjoint union of at most $2h$ *canonical subsets*, where

- $h \in O(\log n)$ is the tree height,
- a canonical subset is an interval that contains all points stored in a subtree.

Range tree

Applications:

- **Geographic Information Systems (GIS)**: Finding all points (e.g., cities) within a certain latitude and longitude range.
- **Database Systems**: Range queries in multidimensional databases.
- **Computer Graphics**: Determining objects within a certain view or selection area.