unit-III

Trees part-II

Red_Black

- A Red-Black tree is a high balanced binary search tree in which every node is coloured with either black or red.
- A Red-Black tree statisfy the following properties.

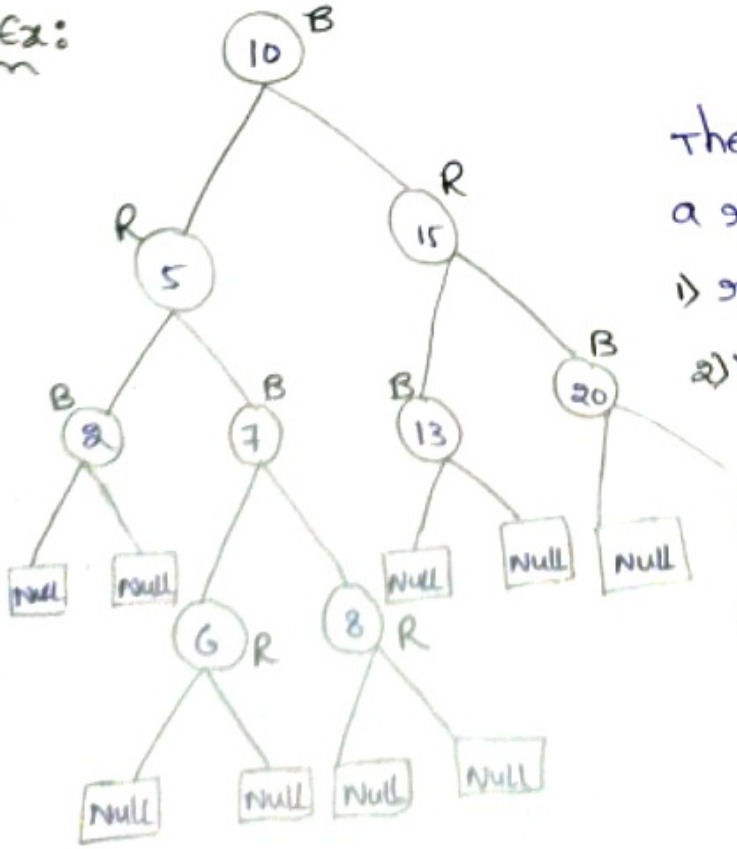
1) The root node and external node always black node.

2) Red condition → No two red node can occurs consecutively on the path from the root node to an external node (or)

- If the node is red then childrens are black

3) Black condition → The no. of black nodes on the path from the root node to an external node must be some for all path

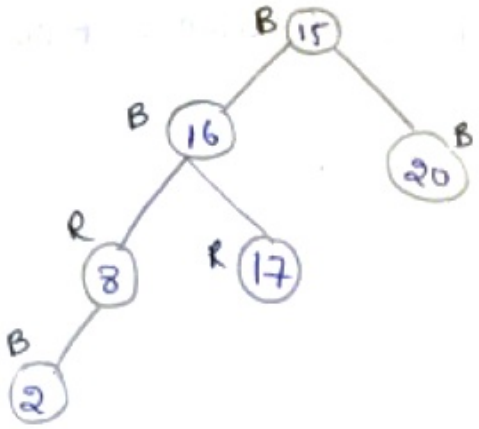
Ex:



Tree

The above diagram is a red black tree, where

1) root node is black
2) no two consecutive reds
3) The number of black nodes are same for all paths



AVL tree v/s Red-black tree (2m)

- The AVL tree are more balanced when compare to Red-black But they make as more rotation during insertion and deletion
- If your Application involves many frequent insertion and deletion then red-black tree should be prefer
- If the insertions and deletions are less frequent and seach is more frquent operation the AVL tree should be pefered over Red_black tree

operations on Red-black tree

The following are operations performed on red black tree

1) Insertion (5m)
2) deletion (5m)
3) search (5m)

1) Insertion: while inserting anew node, new node always inerted as a red node, the when tree is not empty

* After insertion of anew node check whether check the tree is violating the properties of Red-block

tree If it voilates, then we have to balance following operations (or) mechanisms

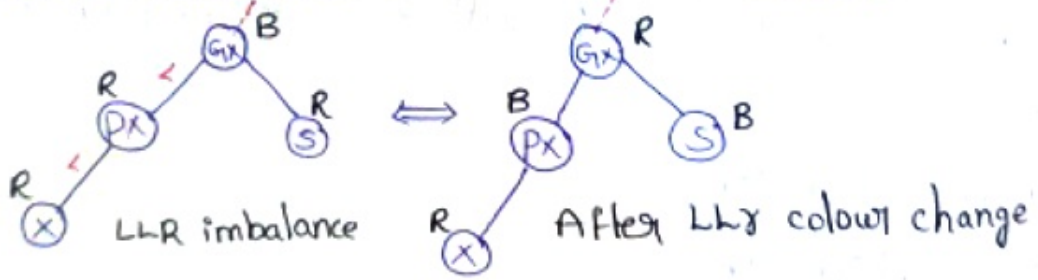
1) Recolour mechanism
2) Rotation mechanism

1) Recolour mechanism

* the following are imbalances
  - LLR imbalance
  - LRR imbalance
  - RRR imbalance
  - RLR imbalance

LLR imbalance

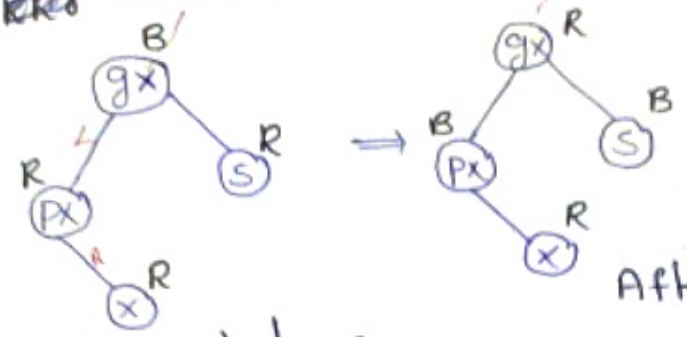
2) Tree is rebalanced by colour changing of nodes
3) If the new node x is inserted as a left child of parent Px, which is left child of Grand parent gx and the colour of other child of the Grand parent gx is red then it is an LLR type inbalance
3) LLR imbalance show in below variable



LRR imbalance

2) Tree is rebalanced by colour changing of nodes
3) If the new node x is inserted as a right child of parent Px, which is left child of grand parent (Gx) and the colour of other child of grand parent (Gx) is red then it is an LRR imbalance

• LRR imbalance is shown in below figure



LRR imbalance

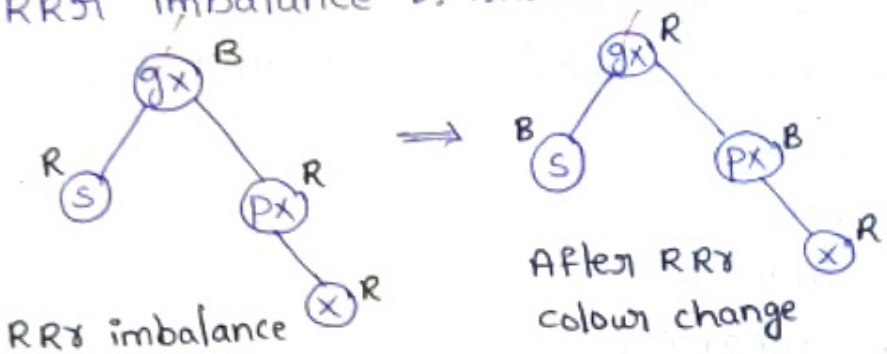
After LRR color change

RRR imbalance

• Tree is rebalanced by colourchanging of nodes

• If the node x is inserted as a right child of Parent Px, which is right child of grand parent (Gx) and the colour of other child of grand parent (Gx) is red then RRR imbalance

• RRR imbalance is shown in the below figure



RRR imbalance

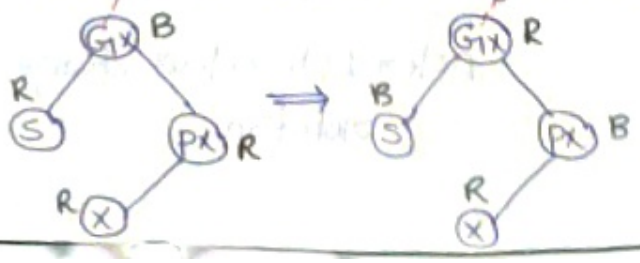
After RRR colour change

RLR imbalance

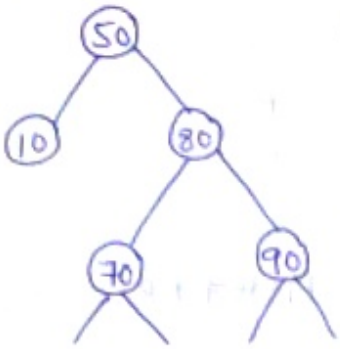
• The tree is rebalanced by colour changing of node

• If the node x is inserted has a right child of parent Px, which is left child of grand parent Gx and the colour of other child of Grand parent Gx is red then RLR imbalance.

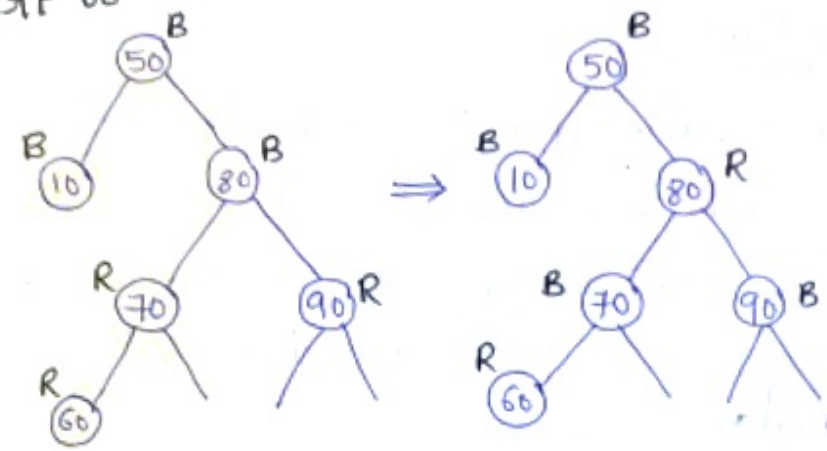
• RLR imbalance shown in given below



Example:



Insert 60



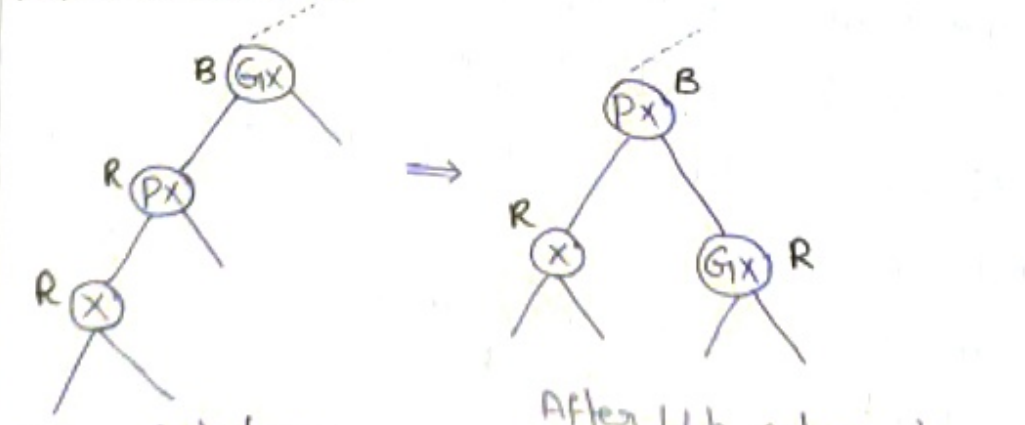
Rotation mechanism

The following are imbalances for rotation

- LLb imbalance
- LRb imbalance
- RRb imbalance
- RLb imbalance

- To setrait the above imbalances Rotation mechanism is used

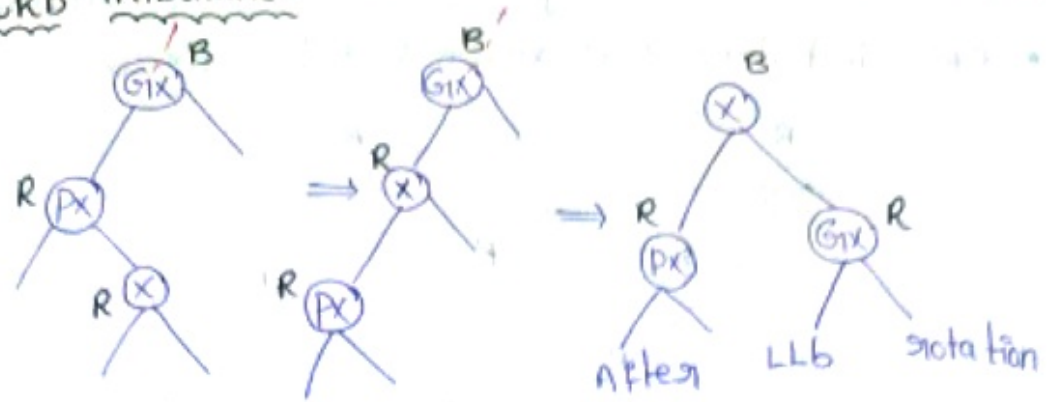
LLb imbalance:



LLb imbalance

After LLb colour-change rotation

LRb imbalance



LLb imbalance

- If the new node x is a inserted as a left child of parent Px, which is left child of grand parent Gx and the colour of other child of the grand parent Gx is black or NULL
- Then it is an LLb imbalance
- Tree is rebalanced by using rotation mechanism
- LLb imbalanced is shown in below figure

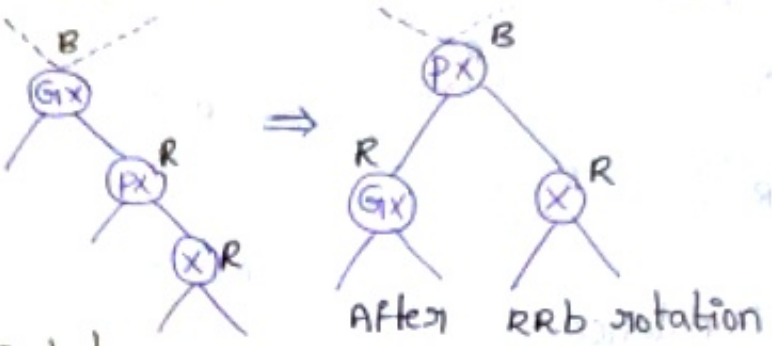
LRb imbalance

- If the new node x is a inserted as a right child of parent Px, which is left child of grand parent Gx and the colour of other child of the grand parent Gx is black or NULL
- Then it is an LRb imbalance
- Tree is rebalanced by using rotation mechanism.
- LRb imbalanced is shown in below figure

RRb imbalance:

- If the new node x is inserted as a right child of parent Px, which is right child of grand parent Gx and the colour of other child of the grand parent Gx is black or NULL
- Then it is an RRb imbalance

- Tree is rebalanced by using rotation mechanism
- RRb imbalanced is shown in below figure

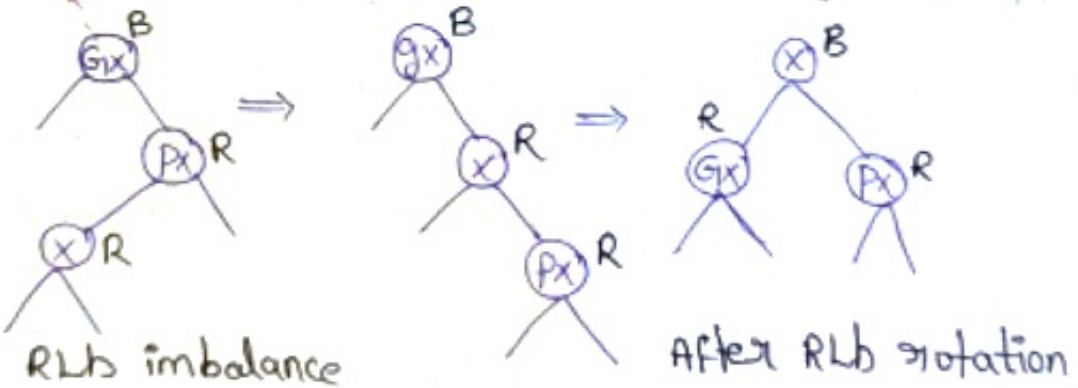


RRb imbalance

After RRb rotation

RLb imbalance:

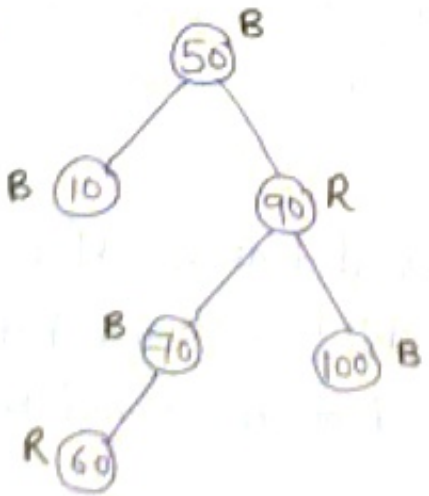
- If the new node x is inserted as a left child of parent Px, which is right child of grand parent Gx and the colour of other child of the Grand Parent Gx is Black or Null then it is an RLb imbalance
- Tree is rebalanced by using rotation mechanism
- RLb imbalance is shown in below figure



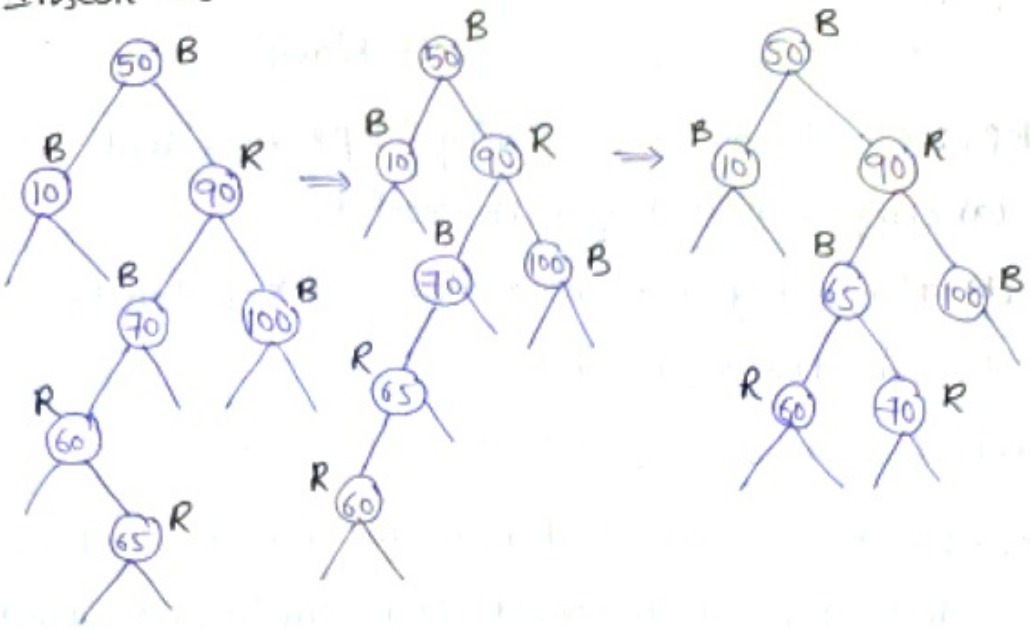
RLb imbalance

After RLb rotation

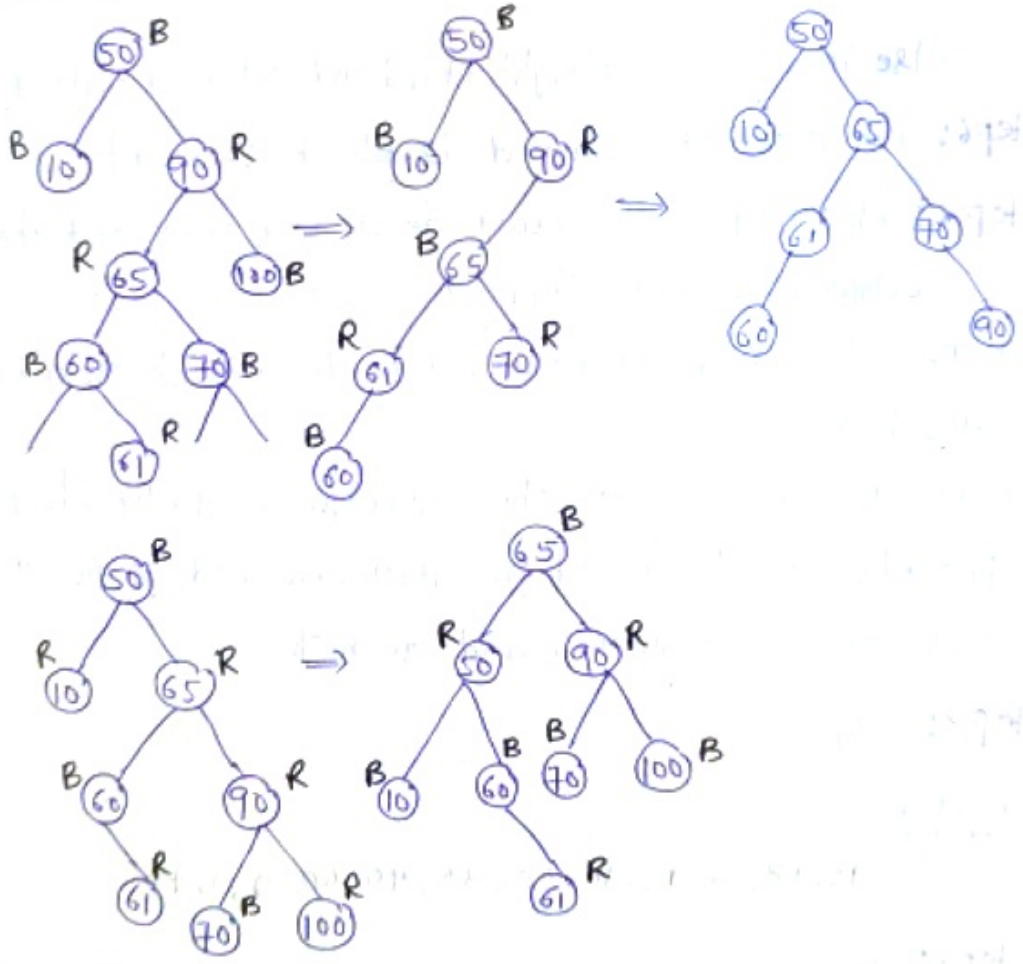
Ex:



Insert 65



Insert 61



Insertion algorithm

step 1: start

step 2: create a new node with given value and set its left and right to null

step 3: If the tree is empty, then insert new node as a root node and colours it black

step 4: Else, Repeat the following steps until leaf

(a) compare new key with root key

(b) If new key is greater than root key, then traverse through the right sub tree

(c) Else traverse through the left subtree

step 5: After reaching leaf node, insert the new node as a left child if the new node is smaller or equal to the left node

Else insert it as a right child and coloured with Red

step 6: The parent of new node is black then exit

step 7: else if, parent of new node is red, then check the colour of parent sibling of new node

(a) If colour is black or null then, do suitable rotation and recolour

(b) If the colour is red then recolour and also check if Parents parent (grand parent) of new node is not root node then recolour it and recheck

step 8: Stop

Example:

10, 18, 7, 15, 16, 30, 25, 40, 60, 2, 1, 70

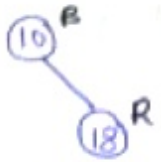
step 1: Insert 10

~~Here~~ tree is empty. So, new node 10 is created and left and right child is null will be inserted and colour as Black.

(10)B

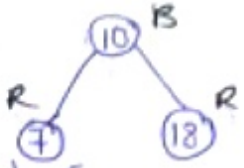
Step2: Insert 18

Here tree is not empty 18 node will be inserted to right of 10 and coloured as red

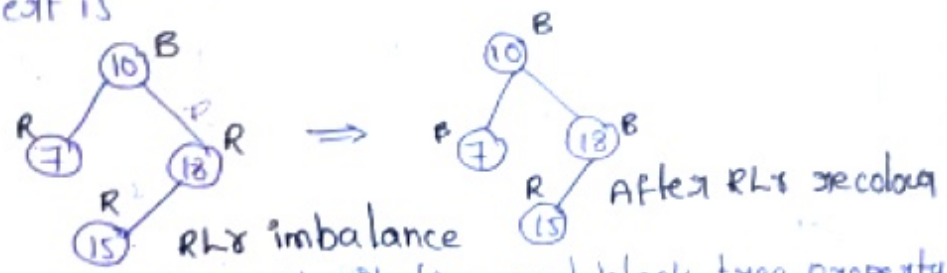


Step3: Insert 7

Here tree is not empty 18 node will be inserted to left of 10 and coloured as red



Step4: Insert 15

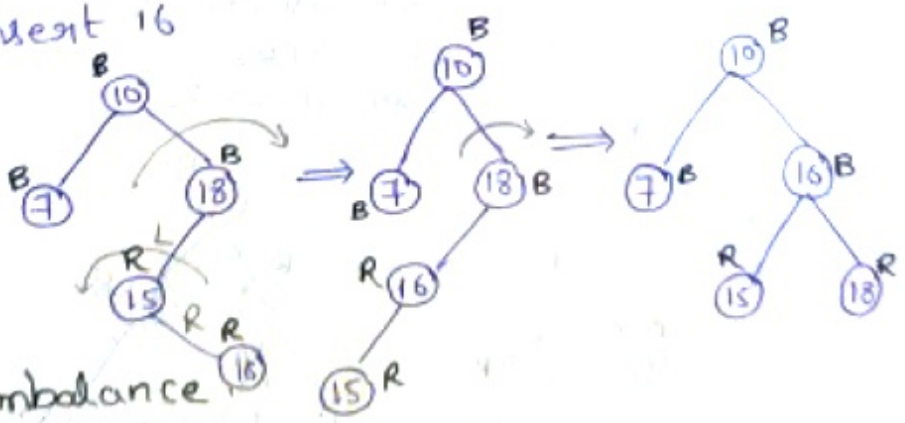


After insertion of 15 it vilating red black tree property i.e. no two adjacent red nodes

RLR imbalance can be removed by changing the colour of parent node 18 and sibling node 7 to black

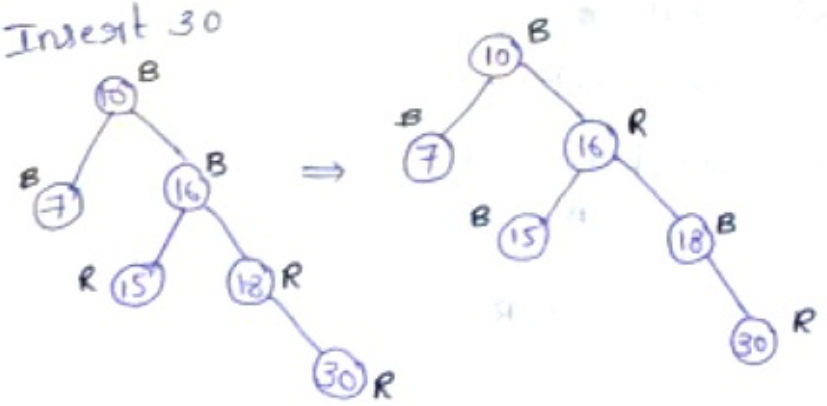
Note: Here we are not changing the colour of grand parent because it is a root node.

Step5: Insert 16

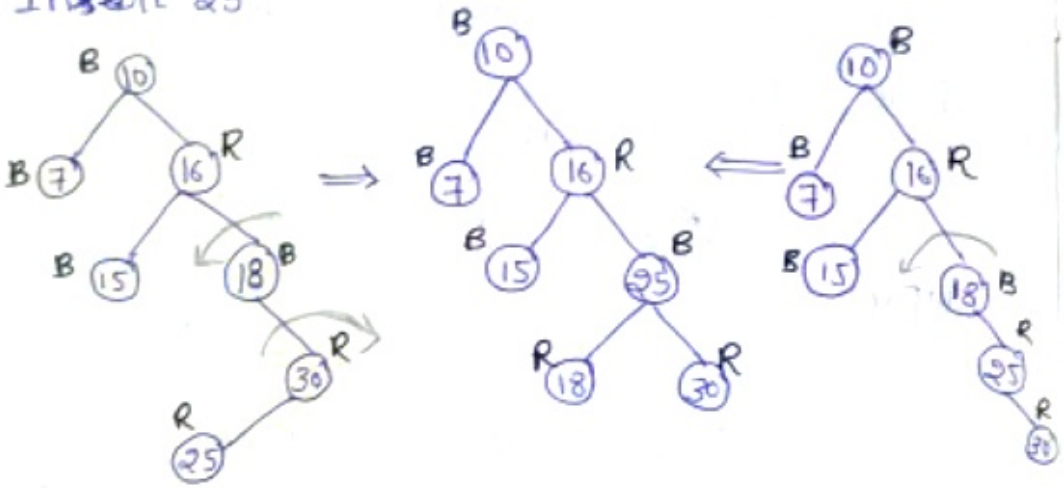


After insertion of 16 it vilating red block tree property and it having LRb imbalance so perform LR rotation and recolour

step 6: Insert 30

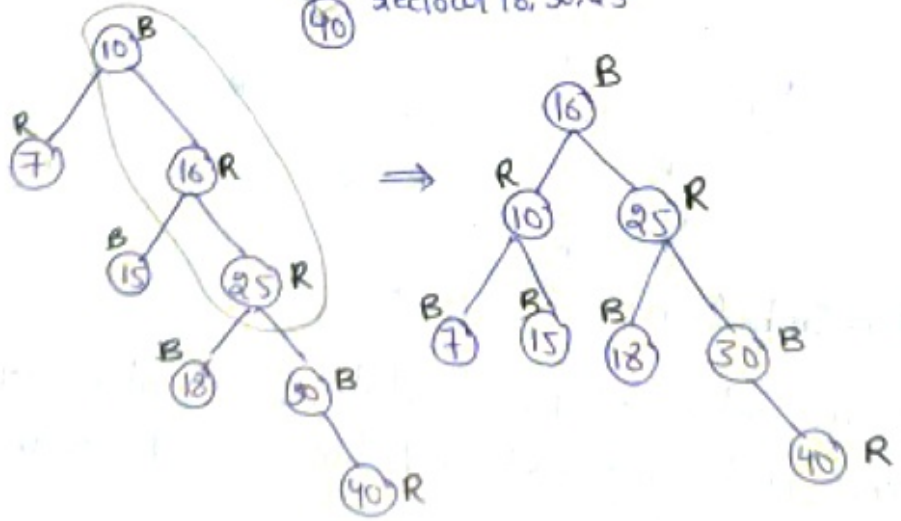


step 7: Insert 25

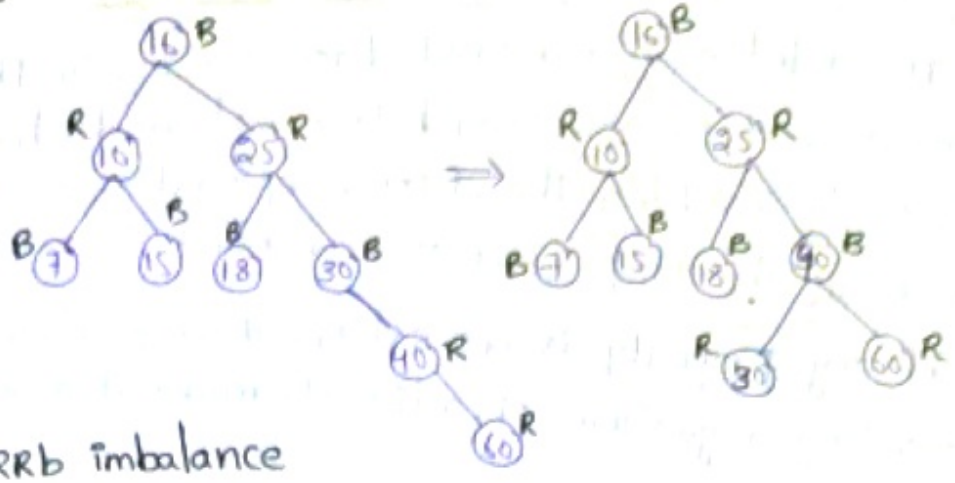


step 8: Insert 40

After insertion of 40 it is RRR recolour 18, 30, 25

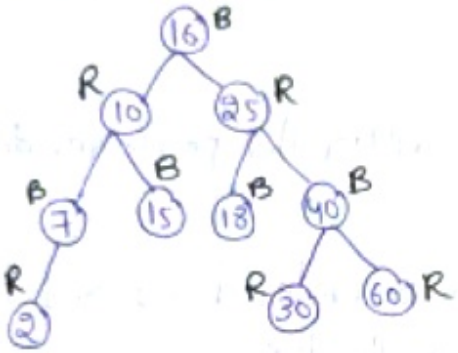


step 9: Insert 60

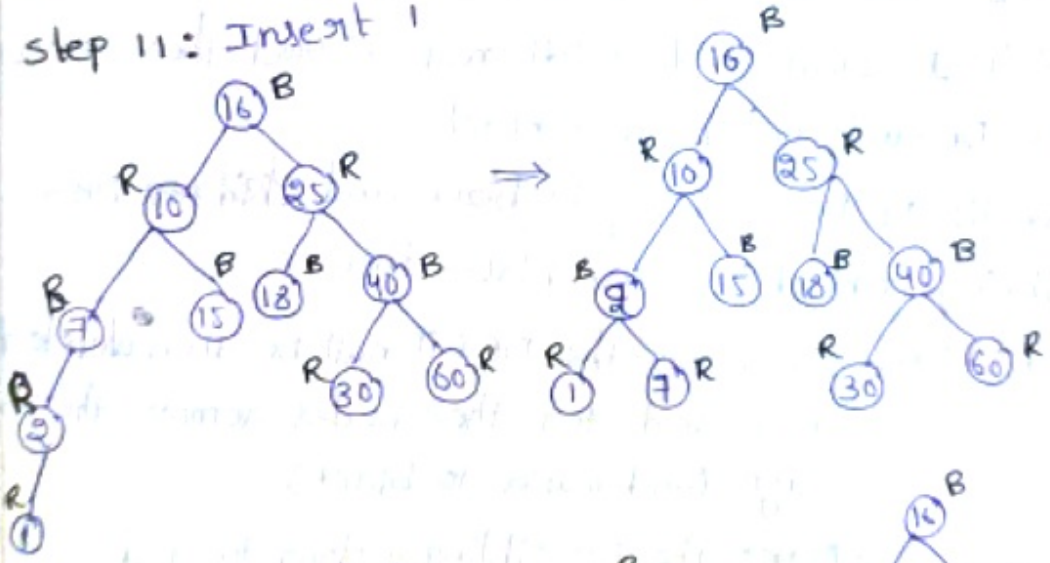


RRb imbalance

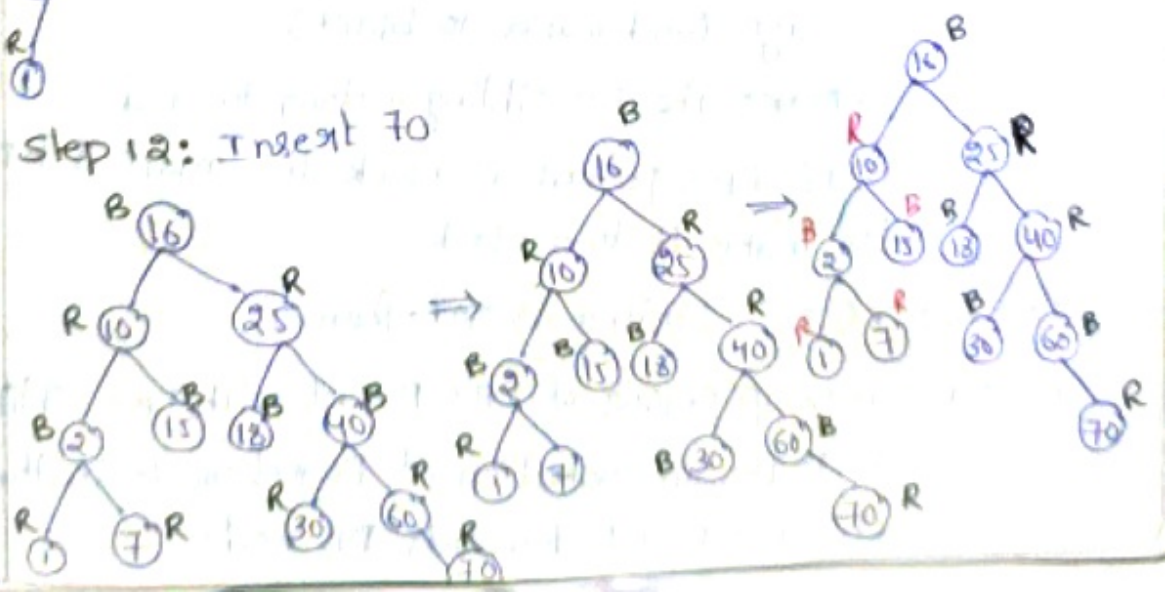
step 10: Insert 2



step 11: Insert 1



step 12: Insert 70



Deletion operation for Red-black tree

- The deletion in red block tree similar to the deletion operation in Binary search tree but nodes have colour property so after the deletion operation you need to check all the properties of red black tree
- If any property is not satisfied, we perform the following algorithm operation to make it a red black tree

Algorithm

step1: start

step2: perform BST deletion after that perform deletion check the following cases

step3: case i: If the node to be deleted is a red leaf node then just remove it from the tree

case ii: If double black (db) node is root then remove the DB and root becomes black

case iii: If DB's sibling is black and ~~children~~ DB's siblings children's are black then

Action: 1) Remove the DB (if null DB then delete the node and for other nodes remove the DB's sign (and make to black)

2) Make the DB sibling colour to red

3) If DB's parent is black then make it to DB else make it to a Black

case iv: If DB's sibling is red then

Action: 1) swap colour of DB's parent with DB's sibling

2) Perform rotation at parent node in the direction of towards DB node

3) check which case is applied and perform that Action

case v: If DB's sibling is black,

a) If DB's sibling child which is far from the DB's black and

b) If DB's sibling child which is near to DB is red then,

Action:

1) swap colour of sibling with siblings red child

2) perform rotation at sibling node in the direction opposite to DB node

3) Apply case 6

case vi: If DB's sibling is black, and DB's sibling far child is red then

Action: 1) swap colour of DB's parent with DB's sibling

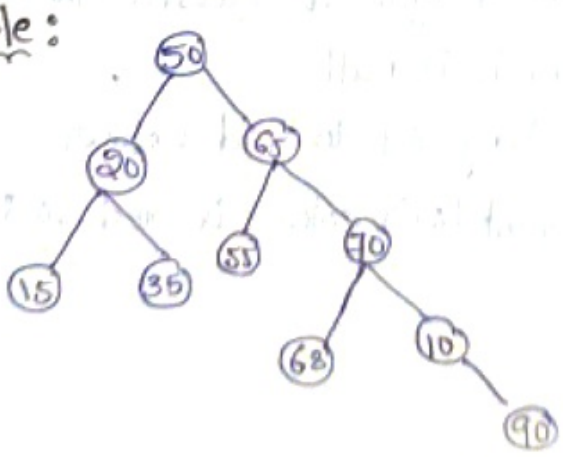
2) perform rotation at DB's parent in the direction of DB

3) Remove DB sign and make the node to normal black node

4) change colour of DB's sibling far red child to black

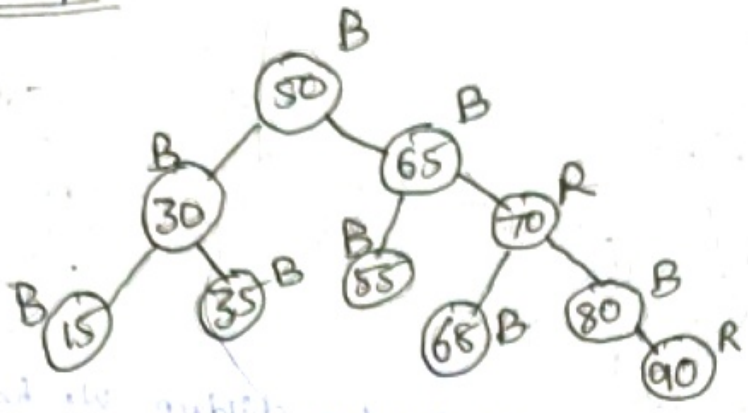
step 4: stop

Example:

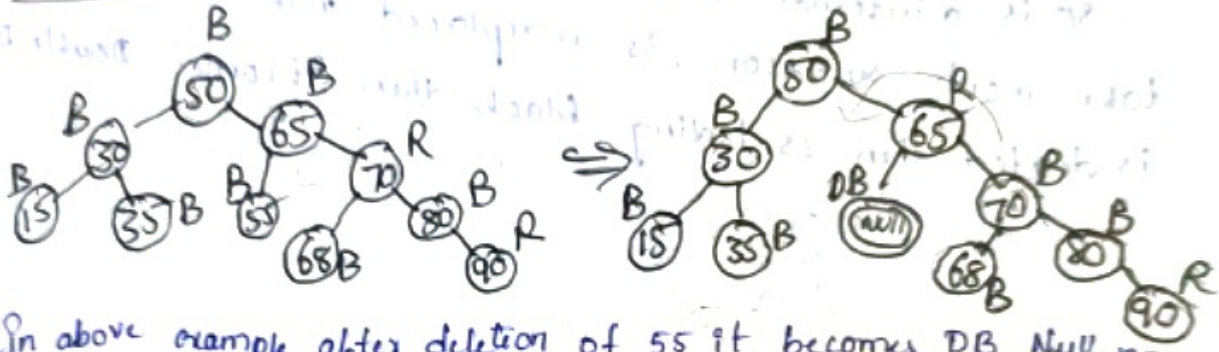


55, 30, 90, 80, 50, 35, 15

Examples: Delete 55, 30, 90, 80, 50, 35, 15



Delete 55:

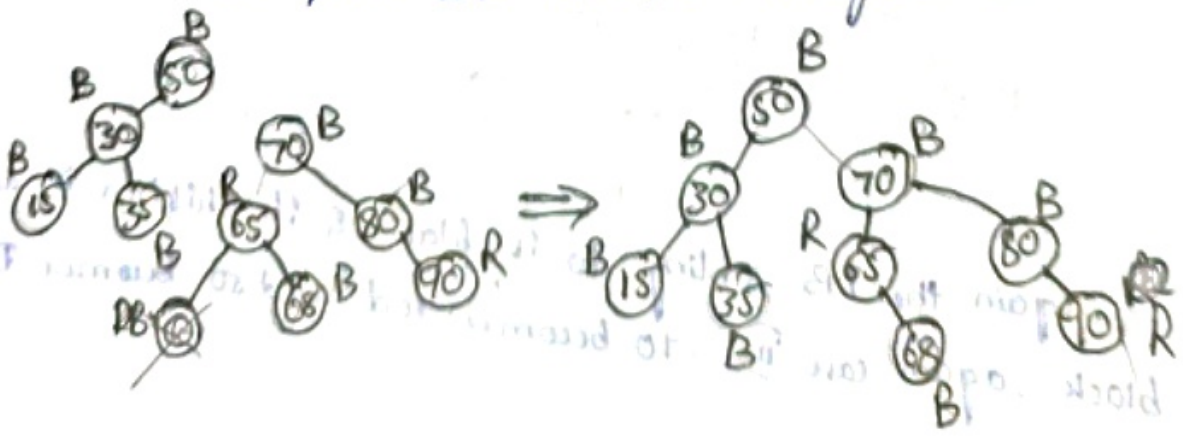


In above example, after deletion of 55 it becomes DB. Null, now will check the cases for this DB where it applys case 4 that is DB's sibling color is 'Red'.

1) Swap the color of DB's parent (65) with sibling (70) [ie B to R]
After swap 65 becomes red and to become black

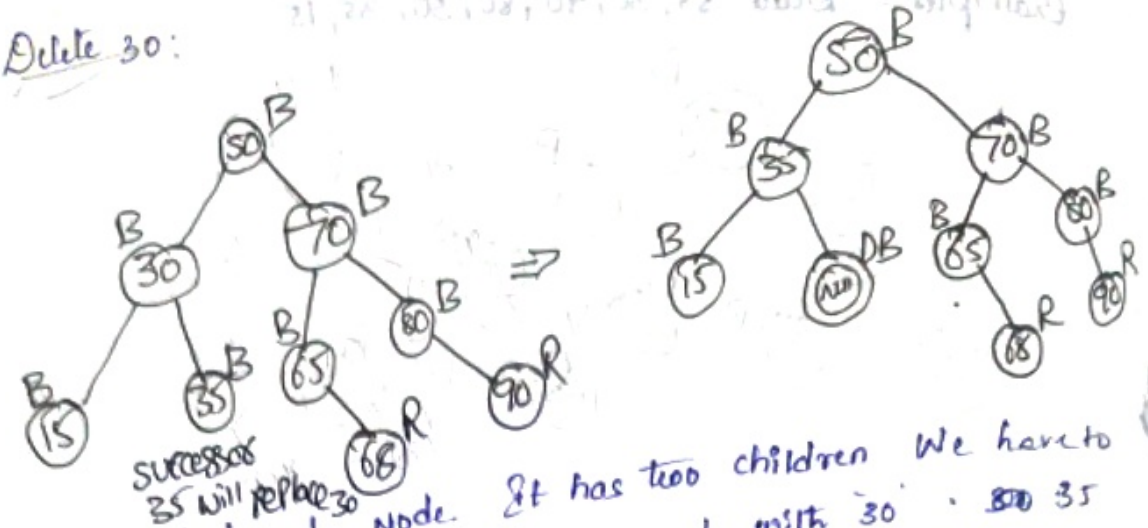
2) perform rotation at DB's parent (65) towards DB node ie.
RR-Rotation

3) After applying case 4 again it satisfies the case 3
- Remove DB which is null
- Make the DB sibling 68 to Red &
- Here DB's parent color is red, so it changes to Black

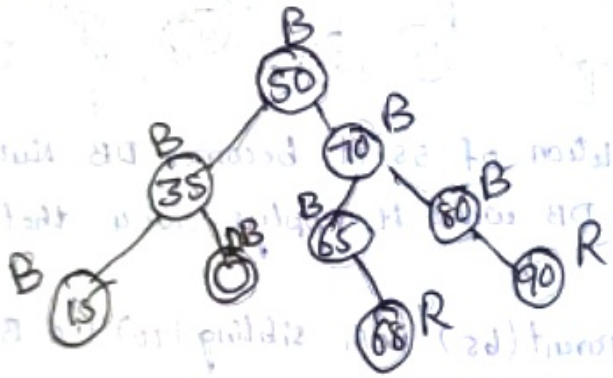


19/11

steps: Delete 30:

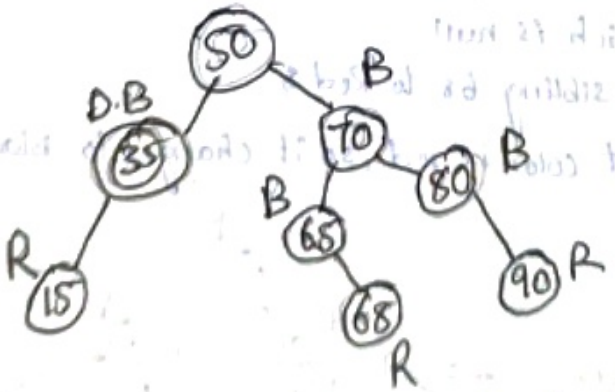


· 30 is a internal Node. It has two children We have to take inorder successor. 35 is replaced with 30 · 35 is deleted then 35 having black then it comes Double Black

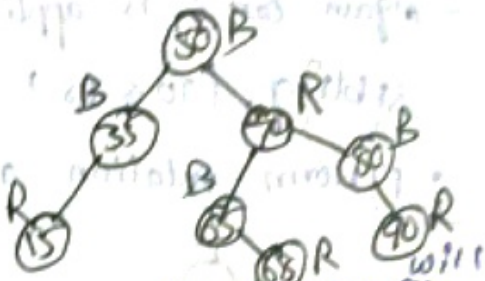


Here DB's sibling is black & children is black then case-3 is applied.

Node is will becomes red & node 35 will become DB

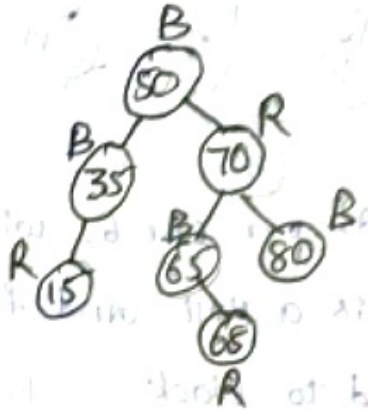


Again the DB sibling 70 is black & its children is also black. again case III. 70 becomes red and so becomes DB.



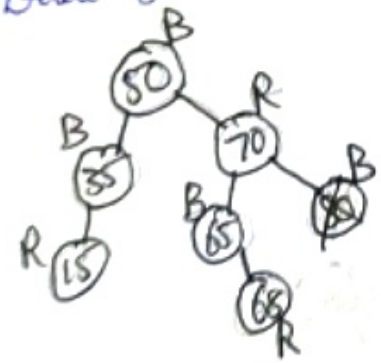
Atlast as per case 5 double black is root node. So it will remove single black from root node

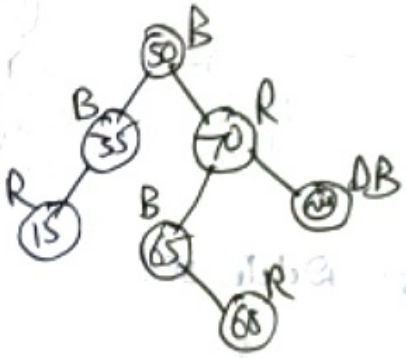
step 3: Delete 90.



90 is leaf red Node so we can delete 90

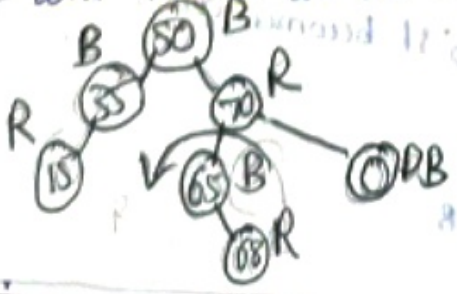
step 4:
Delete 80

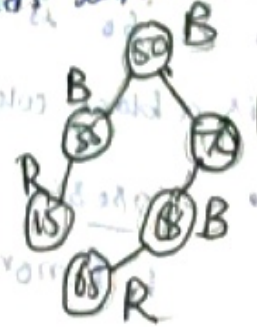




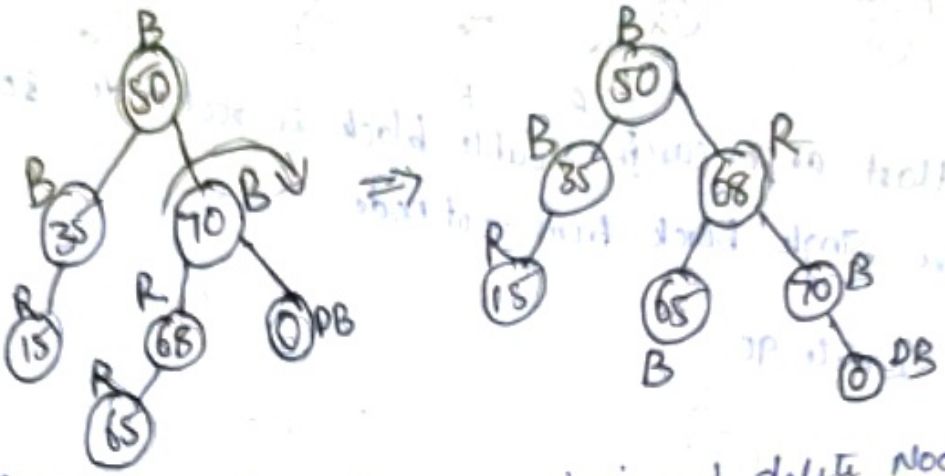
After deletion of 80, it becomes double Black where sibling is black and its far child is black and near child is red so case 5 is applied

- As per case 5, 65 becomes 'red' & 68 become 'black'. And also we will rotate at 65 towards left side

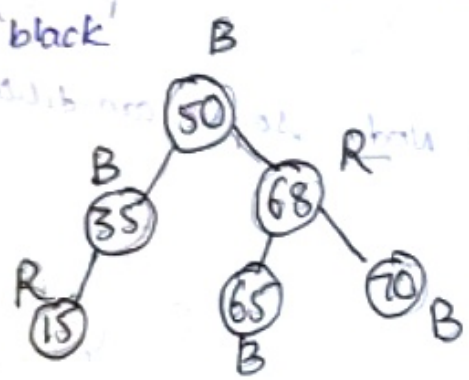




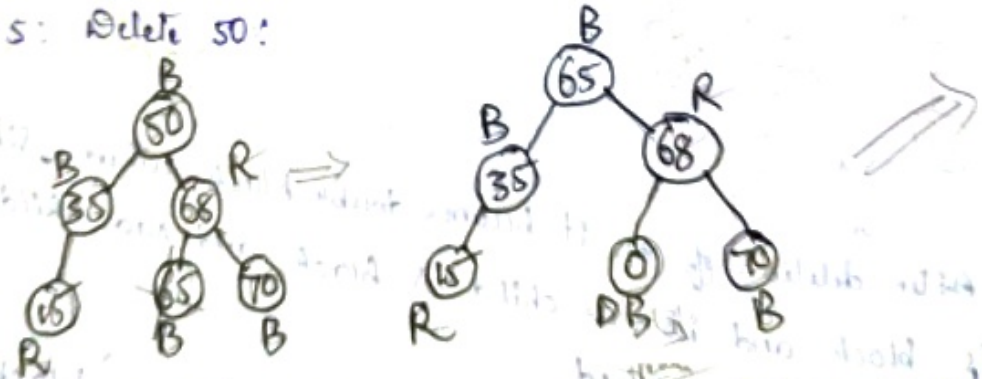
- Again case 6 is applied Swap the color of DB's Parent & sibling, (70 & 68)
- perform rotation at 70 towards right side i.e DB side



As per case 6, will remove DB and delete Node because it is a Null. And finally change the color of sibling '65' and to 'black'



Step 5: Delete 50:



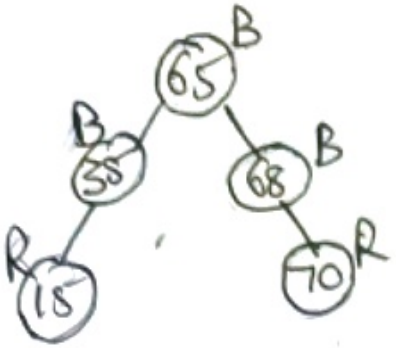
Here 50 is a root node. So we are taking inorder successor i.e 65 is replaced with 50 and delete the 65. but 65 is a black color so it becomes DB. and case 3 is applied.

Case 3:

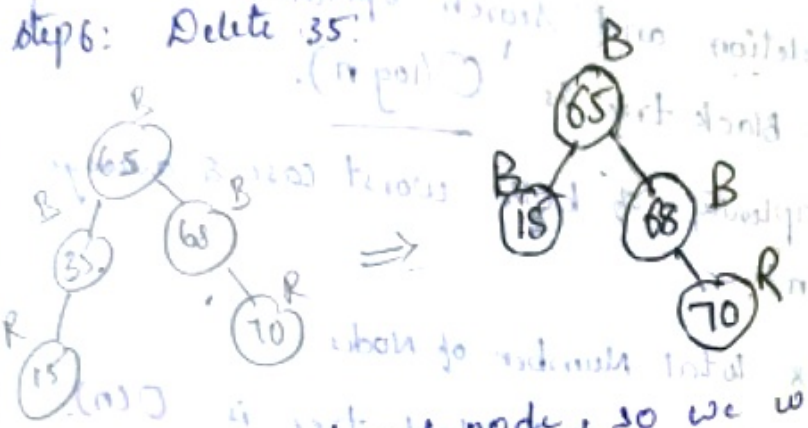
1. Remove DB

2. change the sibling 70 to 'red'

3. Parent 68 is 'red' then so it becomes 'black'

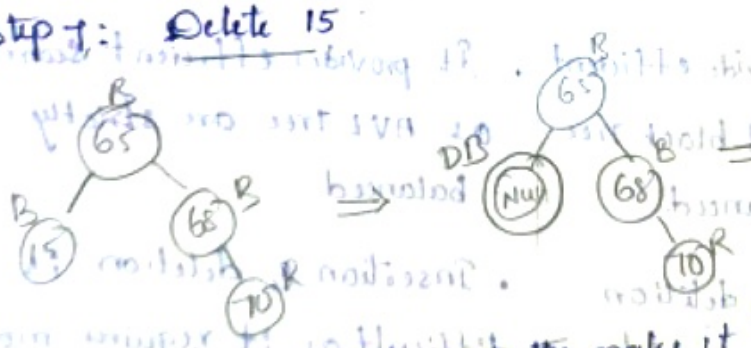


step 6: Delete 35:



Here 35 is internal node, so we will replace with inorder precedor '15' & will delete 15 where 15 is a red leaf node, case '1' is applied. so we can delete it directly

step 7: Delete 15



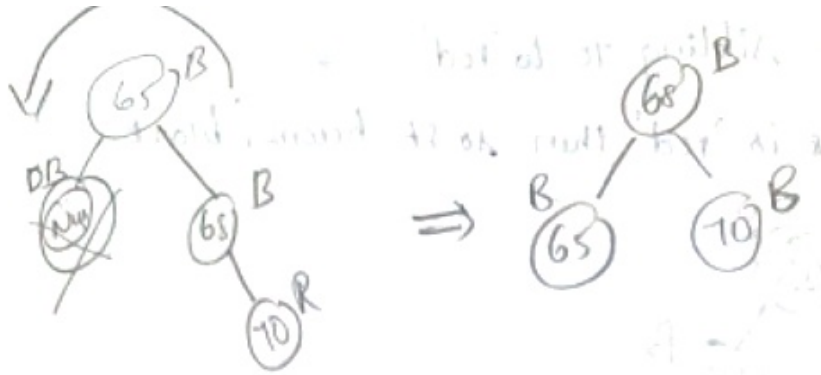
Here '15' is black and the make it to 'DB' and case '6' is applied.

⇒ 1. swap the color of 65, and 68

2. perform rotation at 65 towards DB (i.e left side)

3. Remove DB

4. change the color of for child 70 to Black



→ end →

Time and space complexity (2M)

- For insertion, deletion and search operation, the time Complexity of Red-Black tree is $O(\log n)$.
- The time complexity of both worst case & average case is $O(\log n)$.

Where n is total Number of Nodes

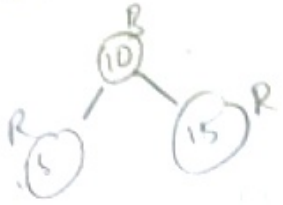
- The space complexity of Red Black tree is $O(n)$.

Worst or average case is same

Differences between Red-Black Tree & AVL Tree (2.07.5)

Red Black Tree	AVL Tree
• It does not provide efficient searching as Red-black Tree are roughly balanced.	• It provides efficient searching as AVL Tree are strictly balanced
• Insertion and deletion operation is easier as it requires less No. of rotations.	• Insertion & deletion is difficult as it requires more No. of rotations.
• The nodes are either Red or Black in color	• The nodes have no colors
• It does not contain any balancing factor to balance the height of the tree	• It contains the balancing factor to maintains balance the height difference in the height of tree.

- Mostly used for insertion & deletion operations
- Example for Red black tree
  Draw tree



- Mostly used for searching operation.
- Example for AVL tree
  Draw tree

Advantages of Red-Black Tree:

1. Red-black Trees are useful when we need insertions & deletions relatively frequent.
2. Red-black Trees are self balancing so these operations guarantees O(logn).
3. It has comparatively low constants in wide range of scenerio's.

Disadvantages of Red-Black Tree:

1. Relatively complicated to implement
2. Red-Black Tree is not strictly balanced in comparision to AVL Tree

Applications of Red-Black Tree: (2M) | 5M

1. Red-Black Tree is used to implement the finite maps
2. Red-Black Trees are used to implement the Java libraries
3. Red-Black Tree is used while building the Linux kernel.
4. It is used in MySQL for Table Indexing
5. It is used to implement the cpu scheduling algorithm.
6. Red Black Tree is used in Computation Jiometry | geometry data structures.

UNIT - III	**Trees Part-II**	8 Hrs

Trees Part-II

Red-Black Trees, Splay Trees, Applications.

Hash Tables: Introduction, Hash Structure, Hash functions, Linear Open Addressing, Chaining and Applications.

Red black tree Refer from class notes

SPLAY TREE

1.1. Definition(2 M):

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. So that if we access the same item again then it takes only O(1) time.

1.2. What is mean by splay operation and splay element:

- Every operation (insertion, deletion or search) on a splay tree performs the splaying operation.
- Splaying an element, is the process of bringing it to the root position by performing suitable rotation operations.
- The insertion operation first inserts the new element as it inserted into the binary search tree, after insertion the newly inserted element is splayed so that it is placed at root of the tree.
- The deletion operation first splays the element to be deleted there by bringing it to the root and then perform the deletion operation.
- The search operations in a splay tree is search the element using binary search tree process then splay the searched element so that it placed at the root of the tree.
- To splay an element to the root, rotations those are similar to that of an AVL tree.

Splaying(2 Marks):

- The splay tree moves a node x to the root of the tree by performing series of single and double tree rotations. Each double rotations moves x to its grandparent's place and every single rotation moves x to its parent's place. We perform these rotations until x reaches to the root of the tree. This process is called *splaying*.
- There are two types of single rotations and four types of double rotations. Each of them is explained in detail below.

Rotations in Splay Tree(5 Marks):

- There are 6 types of rotations are performed on Splay Trees. These are:
    1. Zig Rotation
    2. Zag Rotation
    3. Zig - Zig Rotation
    4. Zag - Zag Rotation
    5. Zig - Zag Rotation
    6. Zag - Zig Rotation

Zig Rotation

Zig is a single rotation. We do zig rotation on node x if x is a left child and x does not have a grandparent (i.e. x's parent is a root node). To make the zig rotation, we rotate x's parent to the right. This is illustrated in figure 1.

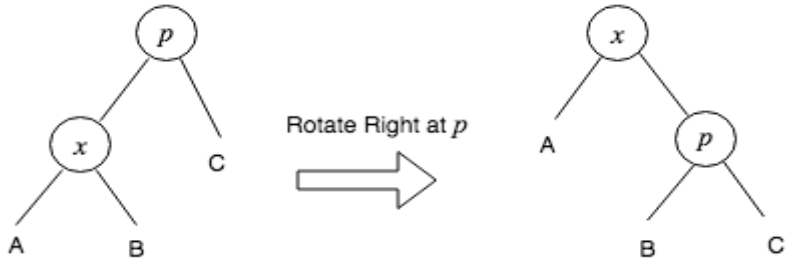


Figure 1: A zig rotation

Zag Rotation

Zag rotation is a mirror of zig rotation. We do zag rotation on node x if x is a right child and x does not have a grandparent. To make the zag rotation, we perform a left rotation at x's parent node. This is illustrated in figure 2.

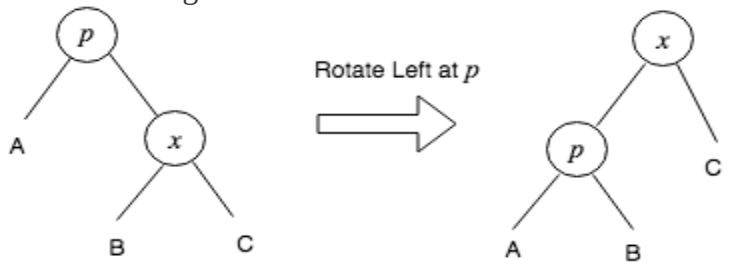


Figure 2: A zag rotation

Zig-Zig Rotation

Zig-Zig is a double rotation. We do a zig-zig rotation on x when x is a left child and x's parent node is also a left child. The zig-zig rotation is done by rotating x's grandparent node to the right followed by right rotation at x's parent node.

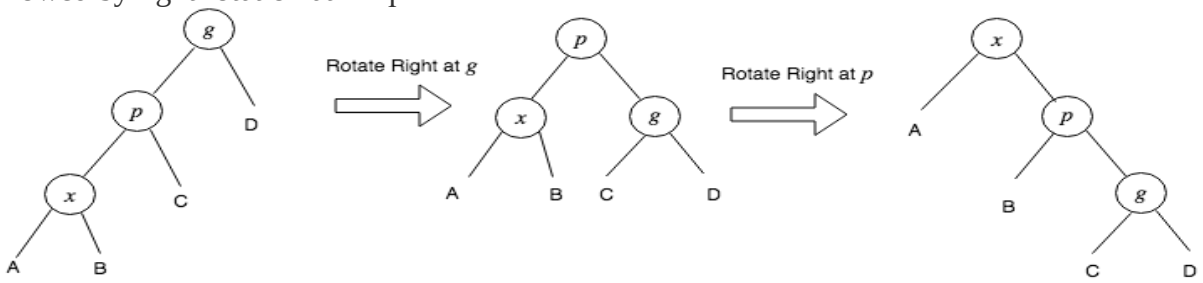


Figure 3: A zig-zig rotation

Zag-Zag Rotation

Zag-Zag rotation is a mirror of zig-zig rotation. We do zag-zag rotation on x if x is a right child and x's parent is also a right child. To make the zag-zag rotation, we first do a left rotation at x's grandparent and then do the left rotation at x's parent node. Figure 4 illustrates this.

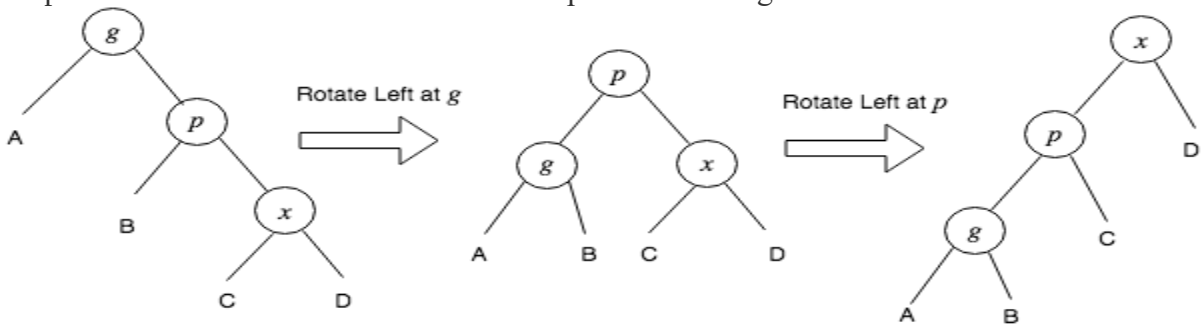


Figure 4: A zag-zag rotation

Zig-Zag Rotation

Zig-zag rotation is also a double rotation. We perform zig-zag rotation on x when x is a right child and x's parent is a left child of grand parent. Zig-zag rotation is done by doing a left rotation at x's parent node followed by right rotating x grandparent (new parent) node. This is illustrated in figure 5.

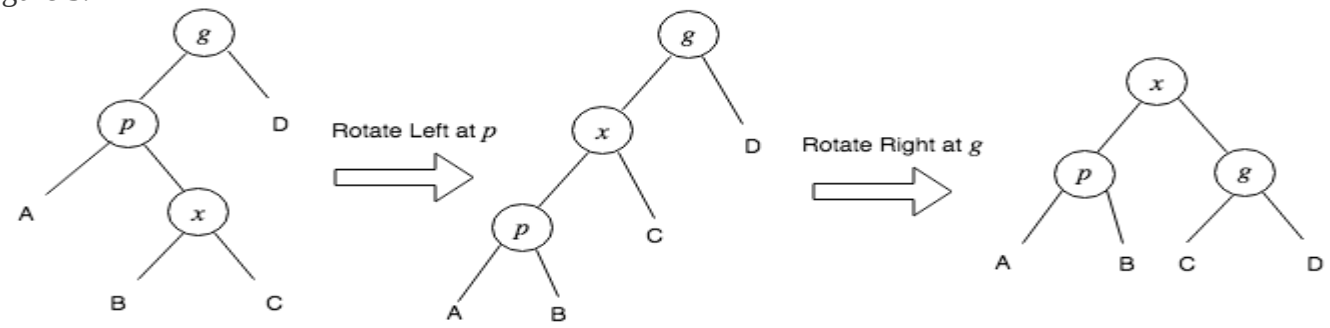
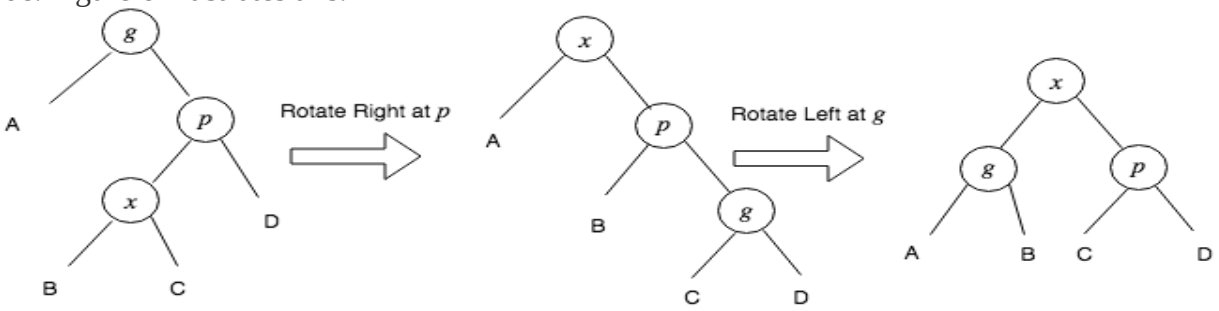


Figure 5: A zig-zag rotation

Zag-Zig Rotation

The last rotation is the zag-zig rotation. It is a mirror of zig-zag rotation. To do zag-zig rotation on node x, we do the right rotation at x's parent node and left rotation at x grandparent (new parent) node. Figure 6 illustrates this.



Operations on Splay Trees:

- The following operations are performed on splay trees.
    1. Searching**(5 Marks)**
    2. Insertion**(5 Marks)**
    3. Deletion**(5 Marks)**

SEARCH

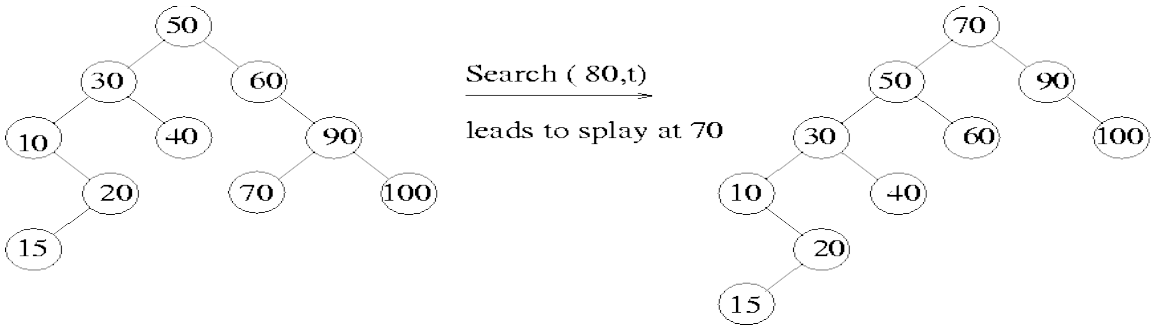
The search operation in a splay tree is similar to the search operation in ordinary search operations. The difference is we splay the node after the search operation.

If item *i* is in tree *t*, return a pointer to the node containing *i*; otherwise return a pointer to the null node.

- Search down the root of *t*, looking for *i*
- If the search is successful and we reach a node *x* containing *i*, we complete the search by splaying at *x* and returning a pointer to *x*
- If the search is unsuccessful, i.e., we reach the null node, we splay at the last non-null node reached during the search and return a pointer to null.
- If the tree is empty, we omit any splaying operation.

Example of an unsuccessful search: See Figure 4.23.

Figure 4.23: An example of searching in splay trees



Insertion: (5 Marks)

Insertion operation in Splay tree

- In the ***insertion*** operation, we first insert the element in the tree and then perform the splaying operation on the inserted element.

Algorithm for Insertion operation

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**

In the above algorithm, Once the insertion is completed, splaying would be performed

Algorithm for Splaying operation

```
SPLAY(x)
  while x.parent ## NIL
    if x.parent.parent == NIL
      if x == x.parent.left
        // zig rotation
        RIGHT-ROTATE(x.parent)
      else
        // zag rotation
        LEFT-ROTATE(x.parent)
    else if x==x.parent.left and x.parent == x.parent.parent.left
      // zig-zig rotation
      RIGHT-ROTATE(x.parent.parent)
      RIGHT-ROTATE(x.parent)
    else if x==x.parent.right and x.parent == x.parent.parent.right
      // zag-zag rotation
      LEFT-ROTATE(x.parent.parent)
      LEFT-ROTATE(x.parent)
    else if x==x.parent.right and x.parent == x.parent.parent.left
      // zig-zag rotation
      LEFT-ROTATE(x.parent)
      RIGHT-ROTATE(x.parent)
```

else

// zag-zig rotation
RIGHT-ROTATE(x.parent)
LEFT-ROTATE(x.parent)

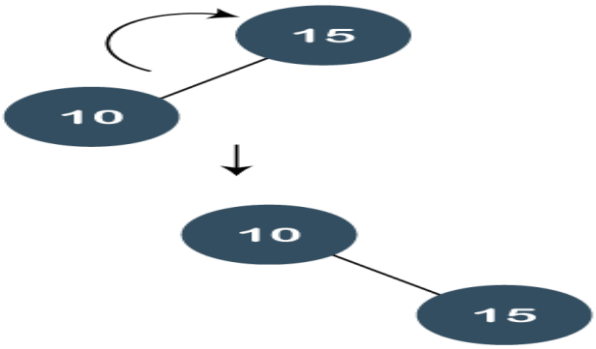
Example:

15, 10, 17, 7

Step 1: First, we insert node 15 in the tree. After insertion, we need to perform splaying. As 15 is a root node, so we do not need to perform splaying.

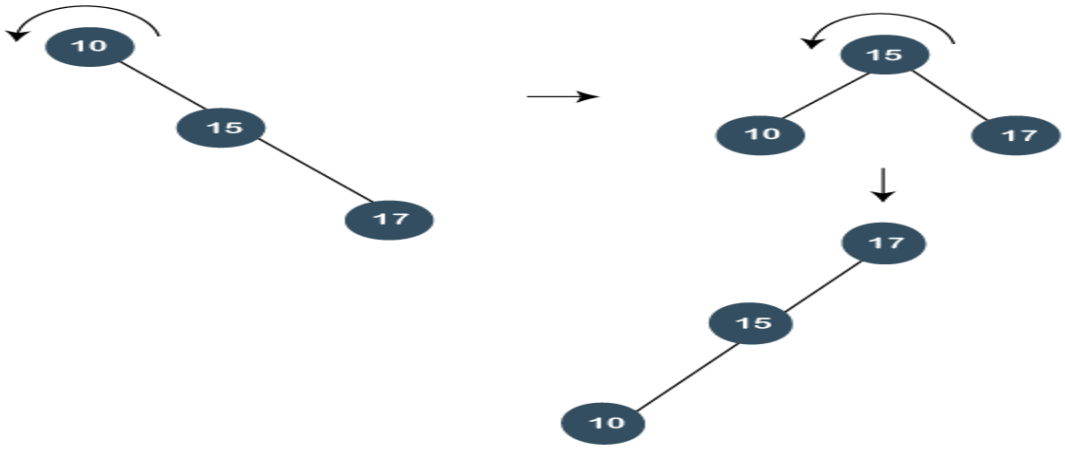
Step 2: The next element is 10. As 10 is less than 15, so node 10 will be the left child of node 15, as shown below:

Now, we perform ***splaying***. To make 10 as a root node, we will perform the right rotation, as shown below:



Step 3: The next element is 17. As 17 is greater than 10 and 15 so it will become the right child of node 15.

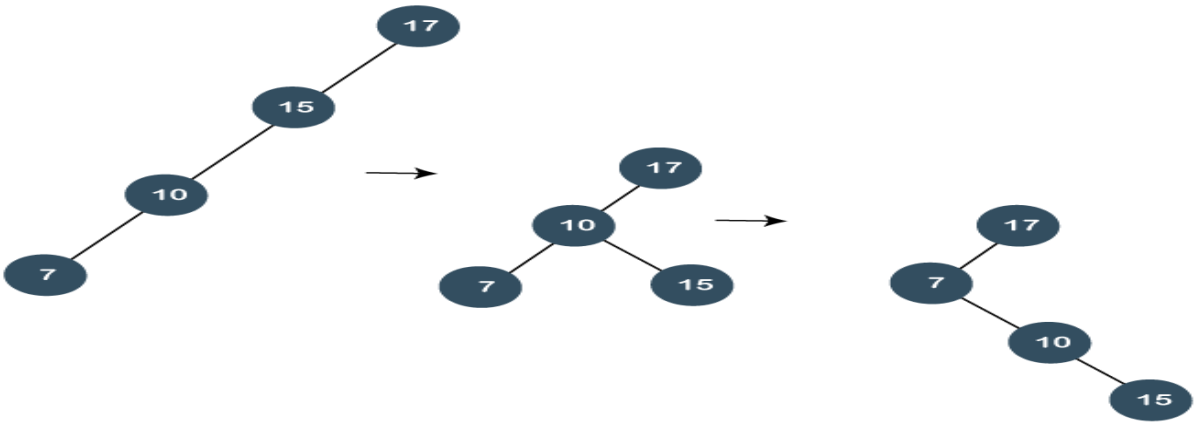
Now, we will perform splaying. As 17 is having a parent as well as a grandparent so we will perform zig zig rotations.



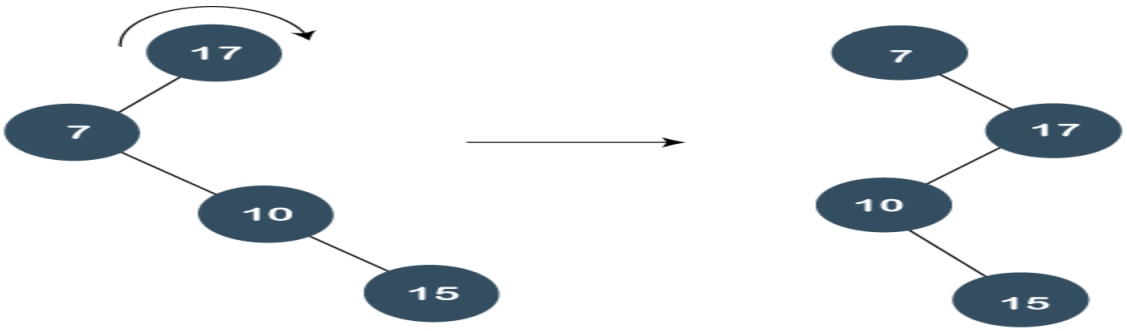
In the above figure, we can observe that 17 becomes the root node of the tree; therefore, the insertion is completed.

Step 4: The next element is 7. As 7 is less than 17, 15, and 10, so node 7 will be left child of 10.

Now, we have to splay the tree. As 7 is having a parent as well as a grandparent so we will perform two right rotations as shown below:



Still the node 7 is not a root node, it is a left child of the root node, i.e., 17. So, we need to perform one more right rotation to make node 7 as a root node as shown below:



Deletion of Splay Tree

- As we know that splay trees are the variants of the Binary search tree, so deletion operation in the splay tree would be similar to the BST, but the only difference is that the delete operation is followed in splay trees by the splaying operation.

Types of Deletions:

There are two types of deletions in the splay trees:

1. Bottom-up splaying
2. Top-down splaying

Algorithm of Delete operation

```
If(root==NULL)
   return NULL
Splay(root, data)
If data!= root->data
        Element is not present
If root->left==NULL
```

```
        root=root->right
else
        temp=root
        Splay(root->left, data)
        root1->right=root->right
        free(temp)
return root
```

- In the above algorithm, we first check whether the root is Null or not; if the root is NULL means that the tree is empty. If the tree is not empty, we will perform the splaying operation on the element which is to be deleted.
- Once the splaying operation is completed, we will compare the root data with the element which is to be deleted; if both are not equal means that the element is not present in the tree. If they are equal, then the following cases can occur:

Case 1: The left of the root is NULL, the right of the root becomes the rootnode.

Case 2: If both left and right exist, then we splay the maximum element in the left subtree. When the splaying is completed, the maximum element becomes the root of the left subtree. The right subtree would become the right child of the root of the left subtree.

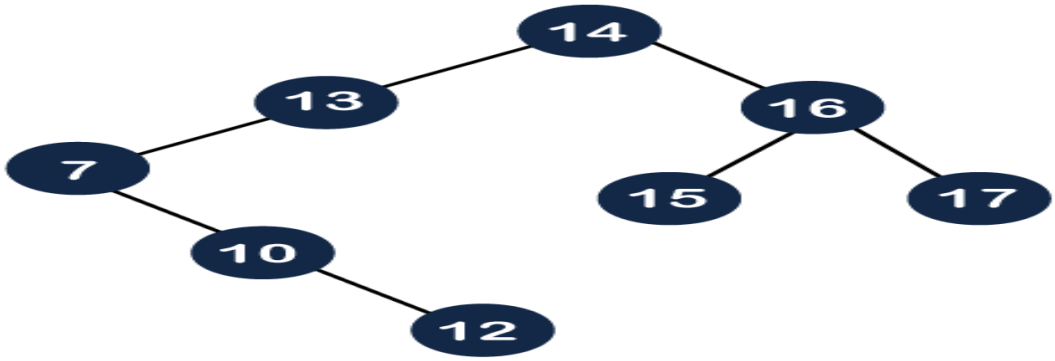
Bottom-up splaying

- In bottom-up splaying, first we delete the element from the tree and then we perform the splaying on the deleted node.

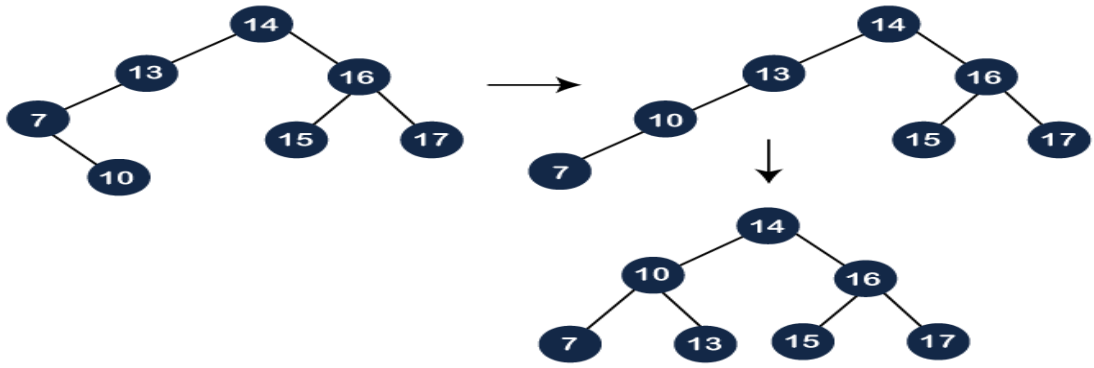
Let's understand the deletion in the Splay tree.

Suppose we want to delete 12, 14 from the tree shown below:

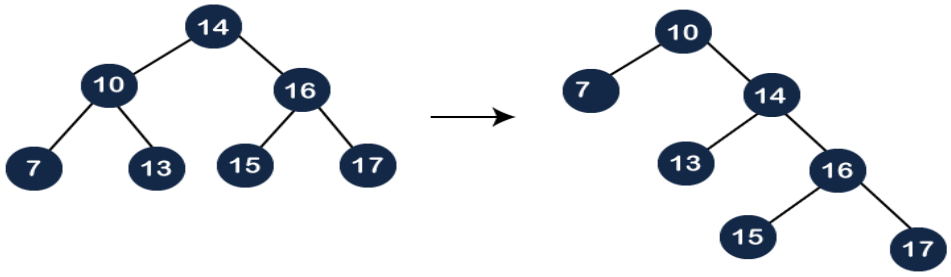
- First, we simply perform the standard BST deletion operation to delete 12 element. As 12 is a leaf node, so we simply delete the node from the tree.



- The deletion is still not completed. We need to splay the parent of the deleted node, i.e., 10. We have to perform ***Splay(10)*** on the tree. As we can observe in the above tree that 10 is at the right of node 7, and node 7 is at the left of node 13. So, first, we perform the left rotation on node 7 and then we perform the right rotation on node 13, as shown below:

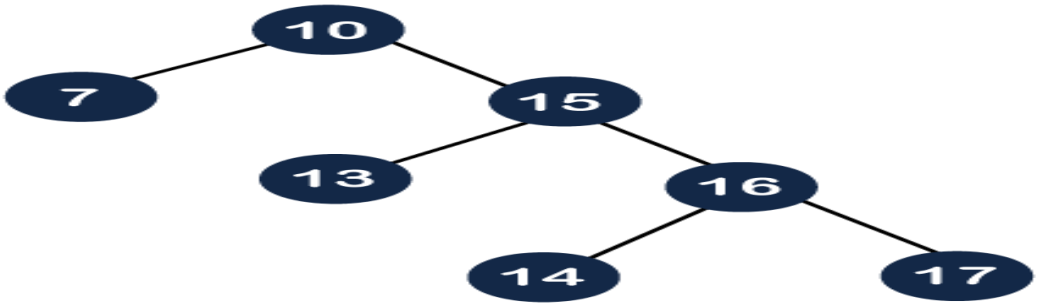


- Still, node 10 is not a root node; node 10 is the left child of the root node. So, we need to perform the right rotation on the root node, i.e., 14 to make node 10 a root node as shown below:

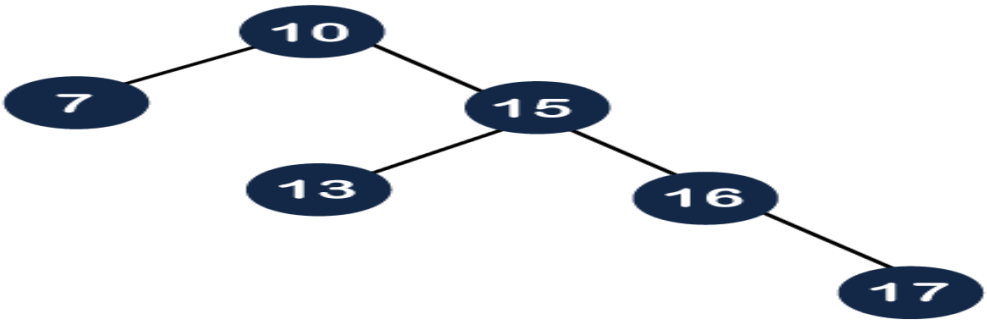


Delete 14:

- Now, we have to delete the 14 element from the tree, which is shown below:As we know that we cannot simply delete the internal node. We will replace the value of the node either using ***inorder predecessor*** or ***inorder successor***.
- Suppose we use inorder successor in which we replace the value with the lowest value that exist in the right subtree. The lowest value in the right subtree of node 14 is 15, so we replace the value 14 with 15. Since node 14 becomes the leaf node, so we can simply delete it as shown below:



- Still, the deletion is not completed. We need to perform one more operation, i.e., splaying in which we need to make the parent of the deleted node as the root node. Before deletion, the parent of node 14 was the root node, i.e., 10, so we do need to perform any splaying in this case.

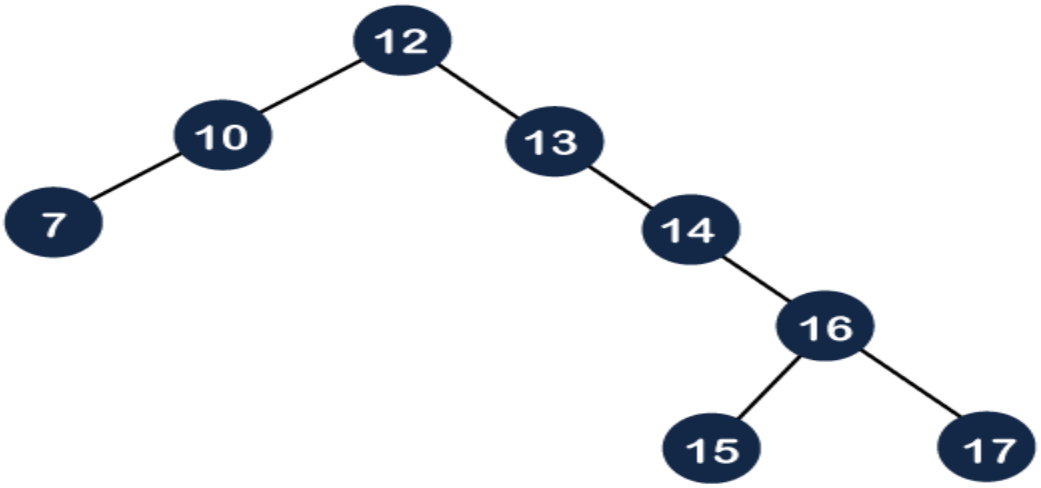


Top-down splaying

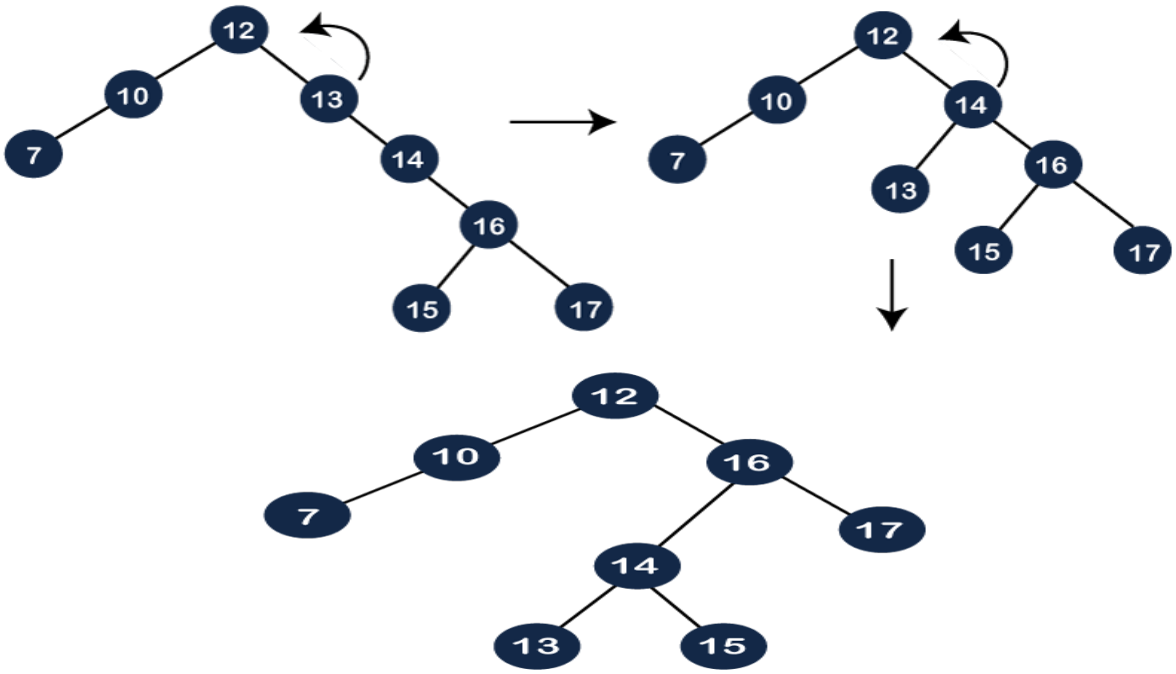
- In top-down splaying, we first perform the splaying on which the deletion is to be performed and then delete the node from the tree. Once the element is deleted, we will perform the join operation.

Let's understand the top-down splaying through an example.

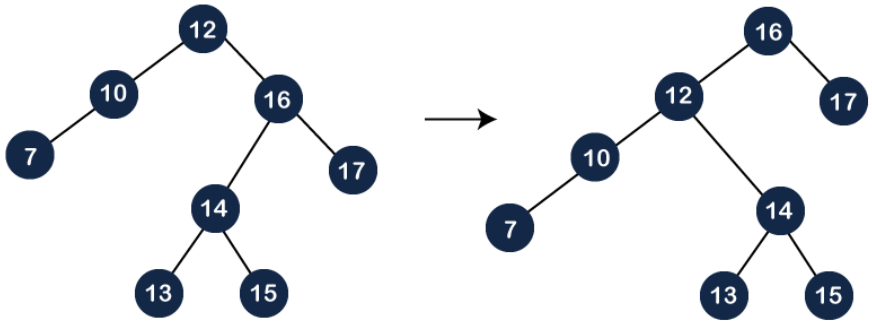
Suppose we want to delete 16 from the tree which is shown below:



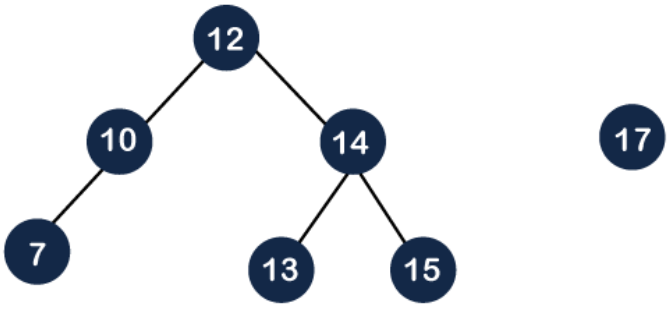
Step 1 Delete 16: In top-down splaying, first we perform splaying on the node 16. The node 16 has both parent as well as grandparent. The node 16 is at the right of its parent and the parent node is also at the right of its parent, so this is a zag zag situation. In this case, first, we will perform the left rotation on node 13 and then 14 as shown below:



- The node 16 is still not a root node, and it is a right child of the root node, so we need to perform left rotation on the node 12 to make node 16 as a root node.



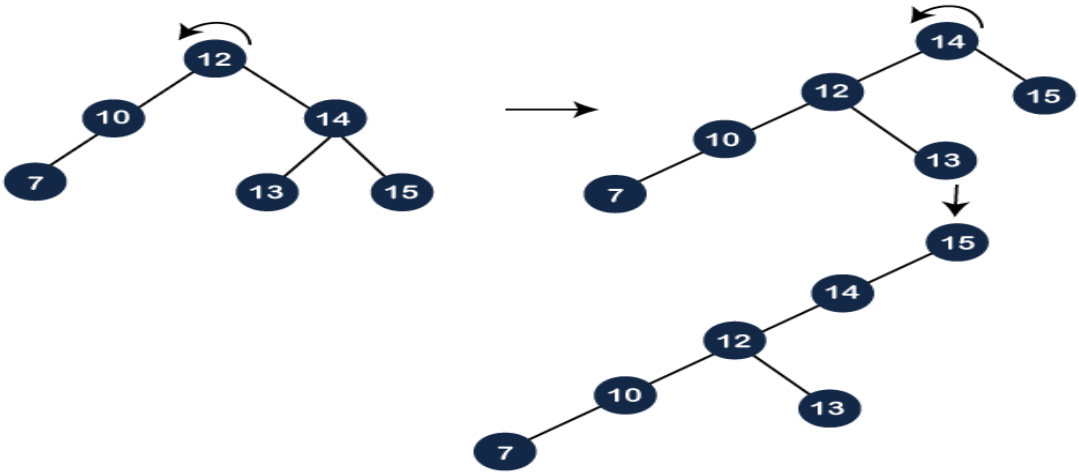
- Once the node 16 becomes a root node, we will delete the node 16 and we will get two different trees, i.e., left subtree and right subtree as shown below:



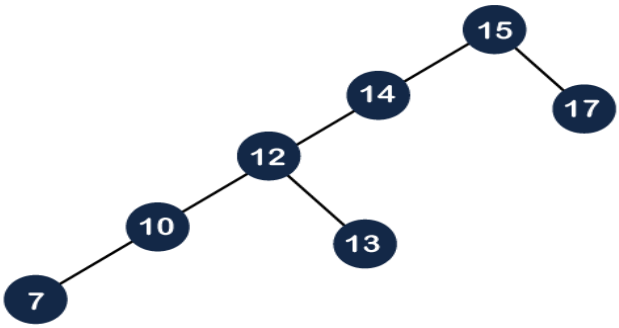
- As we know that the values of the left subtree are always lesser than the values of the right subtree. The root of the left subtree is 12 and the root of the right subtree is 17. The

first step is to find the maximum element in the left subtree. In the left subtree, the maximum element is 15, and then we need to perform splaying operation on 15.

- As we can observe in the above tree that the element 15 is having a parent as well as a grandparent. A node is right of its parent, and the parent node is also right of its parent, so we need to perform two left rotations to make node 15 a root node as shown below:



- After performing two rotations on the tree, node 15 becomes the root node. As we can see, the right child of the 15 is NULL, so we attach node 17 at the right part of the 15 as shown below, and this operation is known as a ***join*** operation.



****END****

Some remarks on amortized analysis of splay trees(5 Marks)

What is Amortized Analysis:

- Analysis of algorithms involves the computation of best, worst and average case time complexities based on the input instances to the algorithm.
- Amortized analysis is different from these analyses in the sense that it estimates the work done by an algorithm over a long sequence of events rather than any single or a specific class of events in isolation. It is the worst case performance of an algorithm over a long sequence of events

amortized analysis of splay trees

The following results hold good with regard to the amortized analysis of splay trees:

(i) The amortized complexity A_i of the splay tree, if step i of its splaying process initiates a zig-zig or a zag-zag step at the specific node u, satisfies the following relation:

$$A_i < 3.\ (r_i(u) - r_{i-1}(u)).$$

(ii) The amortized complexity A_i of the splay tree, if step i of its splaying process initiates a zig-zag or a zag-zig step at the specific node u, satisfies the following relation:

$$A_i < 2 \cdot (r_i(u) - r_{i-1}(u)).$$

(iii) The amortized complexity A_i of the splay tree, if step i of its splaying process initiates a zig or a zag step at the specific node u, satisfies the following relation:

$$A_i < 1 + (r_i(u) - r_{i-1}(u)).$$

(iv) In a binary search tree with n nodes, the amortized cost $C(n)$ of an insertion or search of a specific node with splaying does not exceed $(1 + 3\log_2 n)$ upward moves from the specific node.

(v) In a binary search tree with not more than n nodes, the total complexity of a sequence of w insertions or search operations with splaying, does not exceed $w \cdot (1 + 3\log_2 n) + \log_2 n$.

Applications of Splay Trees(2 Marks):

- Splay trees are suitable for applications with the characteristic that information that is recently retrieved is highly likely to be retrieved in the near future.
- For example, in the case of a university information system, at the beginning of the admission season, those records pertaining to the newly admitted students are highly likely to be accessed over and over again in the first few weeks of their entry. In such an case, it would be a good move to store the records as a splay tree rather than a binary search tree.
- Since a splay tree pushes the recently retrieved records to stay closer to the root
- Maintenance of patient records in a hospital information system, maintenance of records pertaining to seasonal items in a supermarket information system are some examples where splay trees find ideal applications.
- Splay trees have also found applications in data compression, lexicographic search trees and dynamic Huffman coding.

Advantages of Splay Tree(2 Marks)

- In the AVL and Red-Black trees, we need to store some information. Like in AVL trees, we need to store the balance factor, and in the red-black trees, we also need to store one extra bit of information that denotes the color of the node, but in Splay tree we don't need to store
- Splay tree is the fastest type of binary search tree, which is used in a variety of practical applications such as GCC compilers.
- Improve searching by moving frequently accessed nodes closer to the root node. One of the practical uses is cache implementation, in which recently used data is saved in the cache so that we can access the data more quickly without going into memory.

Disadvantages of Splay Tree(2 Marks)

- The main disadvantage of the splay tree is that trees are not strictly balanced, but rather **roughly balanced**. When the splay trees are linear, the time complexity is O(n).

****END****

Hash Table

- (2 Marks)Hash table is one of the most important data structures that use a special function known as a hash function that maps a given value with a key to access the elements faster.
- A Hash table is a data structure that stores some information, and the information has basically two main components, i.e., key and value. The hash table can be implemented with the help of an associative array. The efficiency of mapping depends upon the efficiency of the hash function used for mapping.
- For example, suppose the key value is John and the value is the phone number, so when we pass the key value in the hash function shown as below:

  Hash(key)= index;

  When we pass the key in the hash function, then it gives the index.

  Hash(john) = 3;

  The above example adds the john at the index 3.

Building hash functions

The following are some of the methods of obtaining hash functions:

1. **Folding:** (2 Marks)

   The key is first partitioned into two or three or more parts. Each of the individual parts are combined using any of the basic arithmetic operations such as addition or multiplication. The resultant number could be conveniently manipulated, for example truncated, to finally arrive at the index where the key is to be stored. Folding assures better spread of keys across the hash table.

   ✓ **Example** Consider a six digit numerical key: 719532. We choose to partition the key into three parts of two digits each, (i.e.) 71 | 95 | 32, and merely add the numerical equivalent of each of the parts, (i.e.) 71+ 95+ 32 = 198. Truncating the result yields 98 which is chosen as the index of the hash table where the key 719532 is to be accommodated.

2. **Truncation: (2 Marks)**

   In this method the selective digits of the key are extracted to determine the index of the hash table where the key needs to be accommodated. In the case of alphabetical keys their numerical equivalents may be considered. Truncation though quick to compute, does not ensure even distribution of keys.

   ✓ **Example** Consider a group of six digit numerical keys that need to be accommodated in a hash table with 100 locations. We choose to select digits in position 3 and 6 to determine the index where the key is to be stored. Thus key 719532 would be stored in location 92 of the hash table.

3. **Modular Arithmetic: (2 Marks)**

   This is a popular method and the size of the hash table L is involved in the computation of the hash function. The function makes use of modulo arithmetic. Let k be the numerical key or the numerical equivalent if it is an alphabetical key. The hash function is given by

   H(k) = k mod L

   The hash function evidently returns a value that lies between 0 and L–1. Choosing L to be a prime number has a proven better performance by way of even distribution of keys.

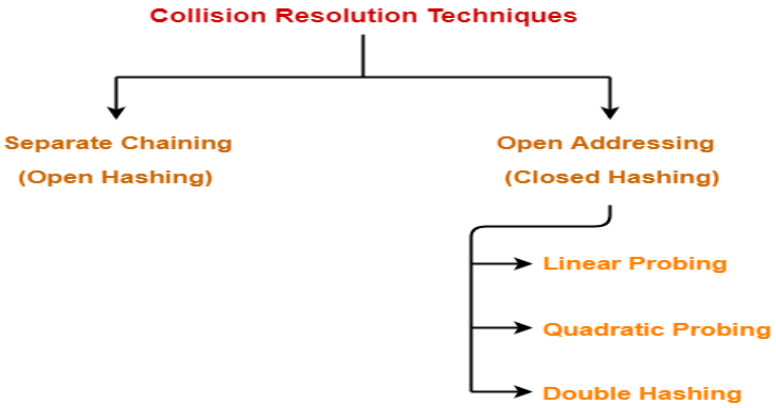
✓ **Example** Consider a group of six digit numerical keys that need to be stored in a hash table of size 111. For a key 145682, H(k) = 145682 mod 111 = 50. Hence the key is stored in location 50 of the hash table.

Collision Resolution Techniques-

- **Hashing** is a well-known searching technique.
- Collision occurs when hash value of the new key maps to an occupied bucket of the hash table.

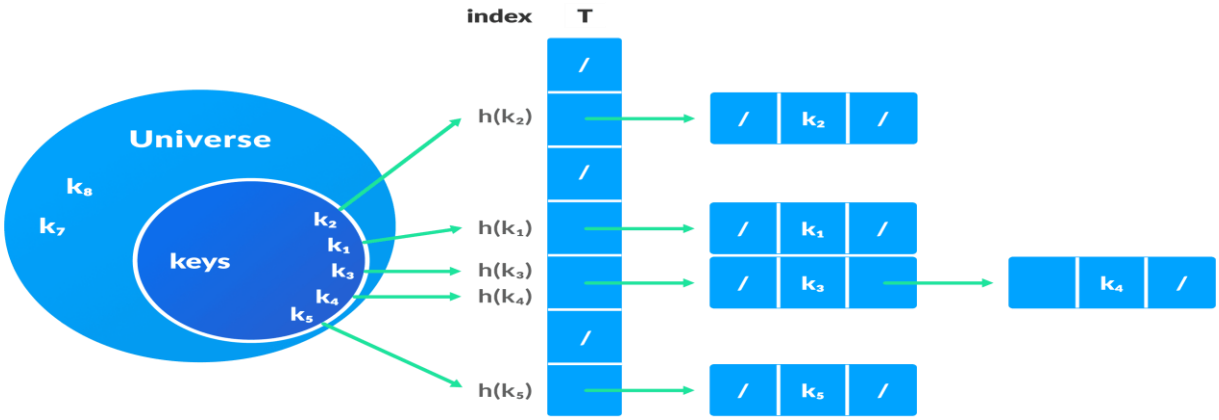
How to handle Collisions?

There are mainly two methods or techniques to handle collision:



Separate Chaining or Linked List:

- It is also called as Closed Addressing, Open Hashing .
- Open Hashing, is the one of the methods used to resolve the collision is known as a chaining method.
- In chaining, if a hash function produces the same index for multiple elements, these elements are stored in the same index by using a linked list which is known as a chain.
- If 'j' is the slot for multiple elements, it contains a pointer to the head of the list of elements. If no element is present, 'j' contains NULL.
- The below figure illustrates collision resolution using chaining.



Example:

Let's use our simple hash function i.e Division method with the following key values: 50, 700, 76, 85, 92, 73, 101.

Solution:

"key mod 7" where 7 is the size of table

Insert 50: 50 mod 7 is 1. So, 50 is inserted in index 1.

Insert 700: 700 mod 7 is 0. So, 700 is inserted in index 0.

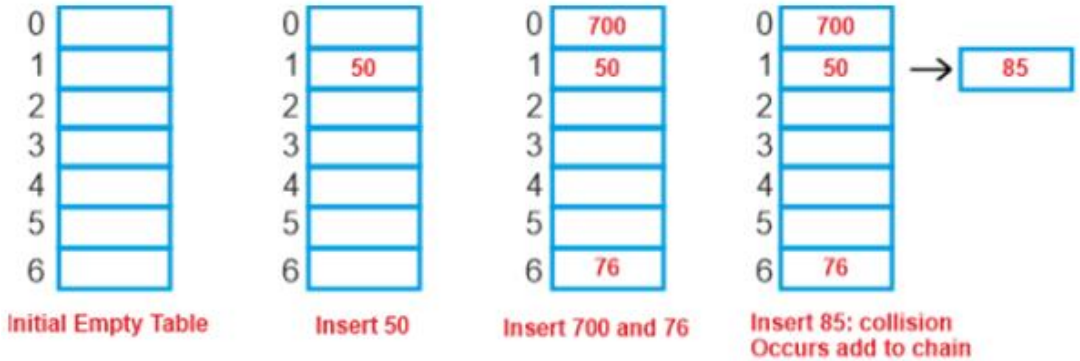
Insert 76: 76 mod 7 is 6. So, 76 is inserted in index 6.

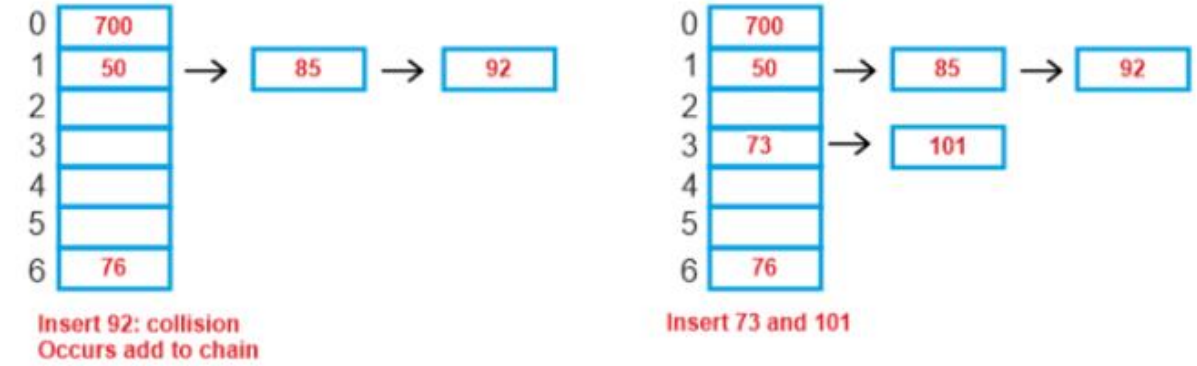
Insert 85: 85 mod 7 is 1. Where index 1 is having element collision occurs. So, it uses linked link to add 85 at same index

Insert 92: 92 mod 7 is 1. Again 92 added to same index 1

Insert 73: 73 mod 7 is 3. So, 73 is inserted in index 3

Insert 101: 101 mod 7 is 3. So, 101 is inserted using chain in index 3





Advantages:

- easy to implement
- We can always add more elements to the chain, thus the hash table never runs out of space.
- less susceptible to load factors or the hash function.
- When it is unclear how many or how frequently keys might be added or removed, it is typically used.

Disadvantages:

- Chaining's cache performance is poor since keys are kept in a linked list. Since everything is stored in the same table, open addressing improves cache speed.
- Space wastage (Some Parts of hash table are never used)
- In the worst situation, search time can become O(n) as the chain lengthens.

- additional space is used for connections.

Load Factor: (2 Marks)

- The load factor of the hash table can be defined as the number of items the hash table contains divided by the size of the hash table. Load factor is the decisive parameter that is used when we want to rehash the previous hah function or want to add more elements to the existing hash table.
- It helps us in determining the efficiency of the hash function i.e. it tells whether the hash function which we are using is distributing the keys uniformly or not in the hash table.

Load Factor= Total elements in hash table/ Size of hash table

Performance of Chaining:

Under the premise that each key has an equal likelihood of being hashed to any table slot, the performance of hashing may be assessed (simple uniform hashing).

m = Number of slots in hash table
n = Number of keys to be inserted in hash table
Load factor α = n/m
Expected time to search = O(1 + α)
Expected time to **delete** = O(1 + α)
Time to insert = O(1)
Time complexity of search insert and **delete** is
O(1) **if** α is O(1)

Linked lists

- Search: O(l) where l = length of linked list
- Delete: O(l)
- Insert: O(l)

Open Addressing or Closed Hashing:

- Open Addressing, which is also known as closed hashing is a technique of collision resolution in hash tables.
- The main idea of open addressing is to keep all the data in the same table to achieve it, we search for alternative slots in the hash table until it is found.
- **Insert(k):** Keep probing until an empty slot is found. Once an empty slot is found, insert k.
- **Search(k):** Keep probing until the slot's key doesn't become equal to k or an empty slot is reached.
- **Delete(k)**: **Delete operation is interesting**. If we simply delete a key, then the search may fail. So slots of deleted keys are marked specially as "deleted". The insert can insert an item in a deleted slot, but the search doesn't stop at a deleted slot.
- Techniques used for open addressing are-
    - Linear Probing
    - Quadratic Probing
    - Double Hashing

Linear Probing:

In linear probing, the hash table is searched sequentially that starts from the original location of the hash. If in case the location that we get is already occupied, then we check for the next location.

The function used for rehashing is as follows: rehash(key) = (n+1)%table-size.

For example, The typical gap between two probes is 1 as seen in the example below:

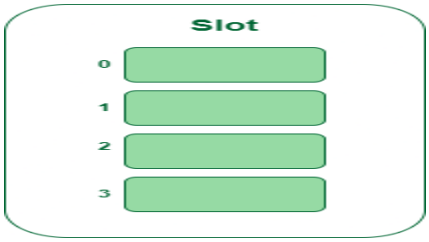
Let **hash(x)** be the slot index computed using a hash function and **S** be the table size

If slot hash(x) % S is full, then we try (hash(x) + 1) % S
If (hash(x) + 1) % S is also full, then we try (hash(x) + 2) % S
If (hash(x) + 2) % S is also full, then we try (hash(x) + 3) % S

And so on..

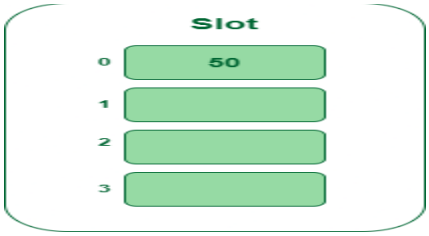
Example: Let us consider a simple hash function as "key mod 5" and a sequence of keys that are to be inserted are 50, 70, 76, 93.

- **Step1:** First draw the empty hash table which will have a possible range of hash values from 0 to 4 according to the hash function provided.



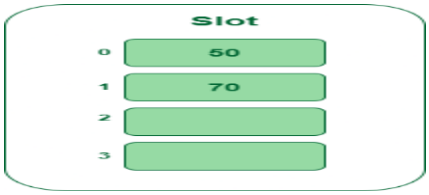
Hash table

- **Step 2:** Now insert all the keys in the hash table one by one. The first key is 50. It will map to slot number 0 because 50%5=0. So insert it into slot number 0.



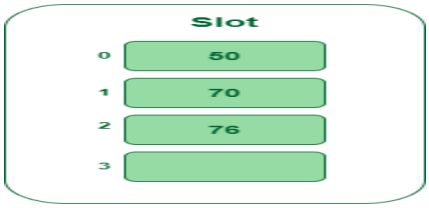
Insert key 50 in the hash table

- **Step 3:** The next key is 70. It will map to slot number 0 because 70%5=0 but 50 is already at slot number 0 so, search for the next empty slot and insert it.



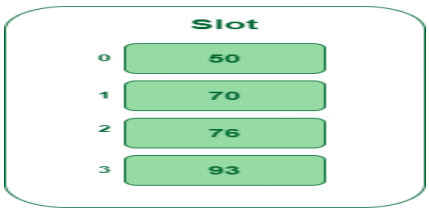
Insert key 70 in the hash table

- **Step 4:** The next key is 76. It will map to slot number 1 because 76%5=1 but 70 is already at slot number 1 so, search for the next empty slot and insert it.



Insert key 76 in the hash table

- **Step 5:** The next key is 93 It will map to slot number 3 because 93%5=3, So insert it into slot number 3.



Insert key 93 in the hash table

Quadratic Probing

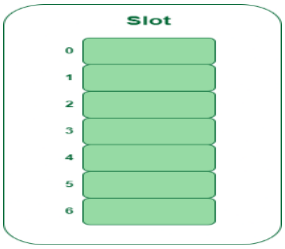
This method is also known as the **mid-square** method. In this method, we look for the **i^2'th** slot in the **i^{th}** iteration. We always start from the original hash location. If only the location is occupied then we check the other slots.

let hash(x) be the slot index computed using hash function.

If slot hash(x) % S is full, then we try (hash(x) + 1*1) % S
If (hash(x) + 1*1) % S is also full, then we try (hash(x) + 2*2) % S
If (hash(x) + 2*2) % S is also full, then we try (hash(x) + 3*3) % S

Example: Let us consider table Size = 7, hash function as Hash(x) = x % 7 and collision resolution strategy to be f(i) = i^2 . Insert = 22, 30, and 50.

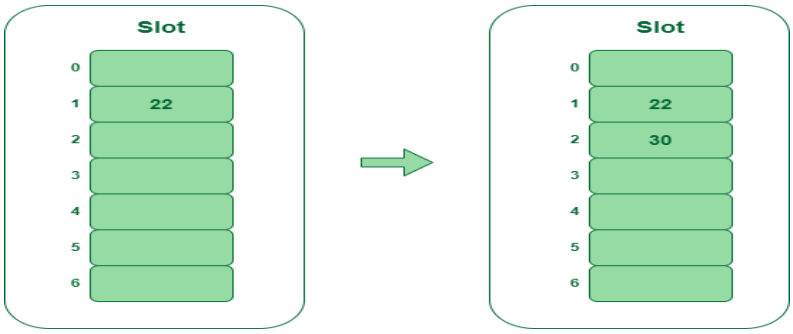
- **Step 1:** Create a table of size 7.



Hash table

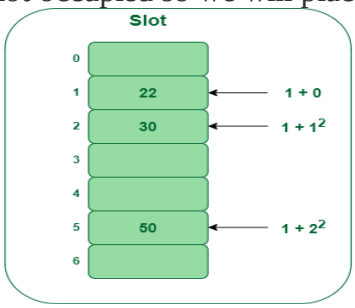
- **Step 2** – Insert 22 and 30
  - Hash(22) = 22 % 7 = 1, Since the cell at index 1 is empty, we can easily insert 22 at slot 1.

- Hash(30) = 30 % 7 = 2, Since the cell at index 2 is empty, we can easily insert 30 at slot 2.



Insert keys 22 and 30 in the hash table

- **Step 3:** Inserting 50
  - Hash(50) = 50 % 7 = 1
  - In our hash table slot 1 is already occupied. So, we will search for slot $1+1^2$, i.e. 1+1 = 2,
  - Again slot 2 is found occupied, so we will search for cell $1+2^2$, i.e.1+4 = 5,
  - Now, cell 5 is not occupied so we will place 50 in slot 5.



Insert key 50 in the hash table

3. Double Hashing

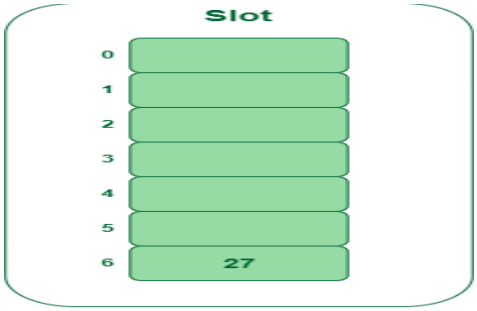
- The intervals that lie between probes are computed by another hash function. Double hashing is a technique that reduces clustering in an optimized way.
- In this technique, the increments for the probing sequence are computed by using another hash function.
- We use another hash function hash2(x) and look for the i*hash2(x) slot in the **i^{th}** rotation.

let hash(x) be the slot index computed using hash function.

*If slot hash(x) % S is full, then we try (hash(x) + 1*hash2(x)) % S*
*If (hash(x) + 1*hash2(x)) % S is also full, then we try (hash(x) + 2*hash2(x)) % S*
*If (hash(x) + 2*hash2(x)) % S is also full, then we try (hash(x) + 3*hash2(x)) % S*
……
……

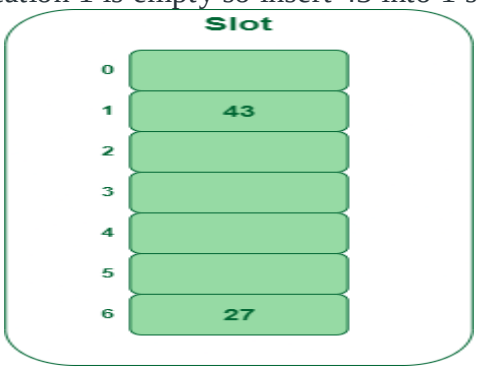
Example: Insert the keys 27, 43, 692, 72 into the Hash Table of size 7. where first hash-function is **h1(k) = k mod 7** and second hash-function is **h2(k) = 1 + (k mod 5)**

- **Step 1:** Insert 27
  - 27 % 7 = 6, location 6 is empty so insert 27 into 6 slot.



Insert key 27 in the hash table

- **Step 2:** Insert 43
    - 43 % 7 = 1, location 1 is empty so insert 43 into 1 slot.

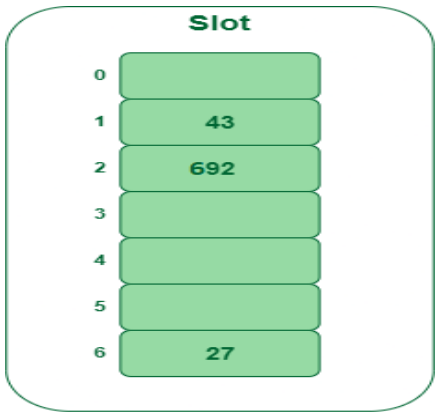


Insert key 43 in the hash table

- **Step 3:** Insert 692
    - 692 % 7 = 6, but location 6 is already being occupied and this is a collision
    - So we need to resolve this collision using double hashing.

      h_{new} = [h1(692) + i * (h2(692)] % 7
      = [6 + 1 * (1 + 692 % 5)] % 7
      = 9% 7
      = 2

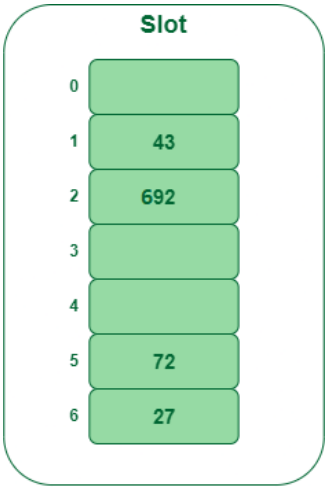
      Now, as 2 is an empty slot,
      so we can insert 692 into 2nd slot.



Insert key 692 in the hash table

- **Step 4:** Insert 72
    - 72 % 7 = 2, but location 2 is already being occupied and this is a collision.
    - So we need to resolve this collision using double hashing.

h_{new} = *[h1(72) + i * (h2(72)] % 7*
= *[2 + 1 * (1 + 72 % 5)] % 7*
= *5 % 7*
= *5,*
Now, as 5 is an empty slot,
so we can insert 72 into 5th slot.



Insert key 72 in the hash table

Performance of Open Addressing:

- Like Chaining, the performance of hashing can be evaluated under the assumption that each key is equally likely to be hashed to any slot of the table (simple uniform hashing)

  m = Number of slots in the hash table
  n = Number of keys to be inserted in the hash table
  Load factor $\alpha = n/m$ (< 1)

  Expected time to search/insert/delete $< 1/(1 - \alpha)$
  So Search, Insert and Delete take $(1/(1 - \alpha))$ time

END

S. No.	Separate Chaining	Open Addressing
1.	Chaining is easier to put into practise.	Open Addressing calls for increased processing power.
2.	Hash tables never run out of space when chaining since we can always add new elements.	Table may fill up when addressing in open fashion.
3.	Chaining is less susceptible to load or the hash function.	To prevent clustering and load factor, open addressing calls for extra caution.

4.	When it is unclear how many or how frequently keys might be added or removed, chaining is typically utilised.	When the frequency and quantity of keys are known, open addressing is employed.
5.	Chaining's cache performance is poor since keys are stored in linked lists.	Since everything is stored in the same table, open addressing improves cache speed.
6.	Space wastage (Some Parts of hash table in chaining are never used).	A slot can be used in open addressing even if an input doesn't map to it.
7.	Chaining requires additional room for links.	Links absent in open addressing

Requirements of good hash function(2m):

- Basic Requirements of Good Hash Function:
    1. Easy to compute: It should be easy to compute and must not become an algorithm in itself.
    2. Uniform distribution: It should provide a uniform distribution across the hash table and should not result in clustering.
    3. Less collisions: Collisions occur when pairs of elements are mapped to the same hash value. These should be avoided

Applications of hashing(2 or 5 m):

- Hashing is a way to add data in any data structure in such a way that it is possible to insert, delete, and scan the simple operations on that data in O (1) time.
- To implement programming languages, file systems, pattern searching, distributed key-value storage, cryptography, etc., hashing is used. There are a number of cases in which the principle of hashing is used.
- There are also other hashing uses, including the hash functions of modern day cryptography. Here are some of these applications:
    1. Message Digest
    2. Password Verification
    3. Data Structures
    4. Compiler Operation
    5. Rabin-Karp Algorithm
    6. Linking File name and path together

- **Message Digest:**
    - A message digest is a fixed size numeric representation of the contents of a message, computed by a hash function. A message digest can be encrypted, forming a digital signature
- **Password Verification:**
    - In password authentication, cryptanalysis hash task is very widely used.
    - Let’s use an example to view this:
        - You type your email and password to validate that the account you are attempting to access belongs to you anytime you use any online service that needs a user username. A hash of the password is

calculating as the password is entered, and is then forwarded to the server for password authentication. The passwords saved on the server are simply the original passwords' calculated hash values. This is required to guarantee that no sniffing is present when the password is transmitted from client to server.

- **Data Structures:**
  - Various programming tongues provide Data Structures found on the hash table. The main principle is to generate a key-value twine where a particular rate is meant to be a key, while for different keys the rate may be the same. This implementation is used in C++ in disordered set & unordered charts, java in HashSet & HashMap, python enum, etc.
- **Compiler Operation**:
  - Programming tongues keywords are refined in a different manner than most identifiers. The accumulator adds all these access. in a collection that is implemented using a hash table to distinguish between the access. of a programming tongue (if, otherwise, for, back, etc.) and other selectors and to accumulate the software successfully.
- **Rabin-Karp Algorithm:**
  - The Rabin-Karp contrivance is one of the most common uses of hashing. This is effectively a cord-searching contrivance used to locate someone set of rules in a cord using hashing. Detecting plagiarism is a realistic implementation of this algorithm. Go by Looking for Patterns or Set 3 (Rabin-Karp Algorithm) to learn more about Rabin-Karp.
- **Linking File name and path together:**
  - We notice two very important components of a file when going through data on our local machine, i.e. file name and file path. The system utilizes a guide (document name, record way), which is actualized utilizing a hash table, to spare the correspondence between document name and document way.