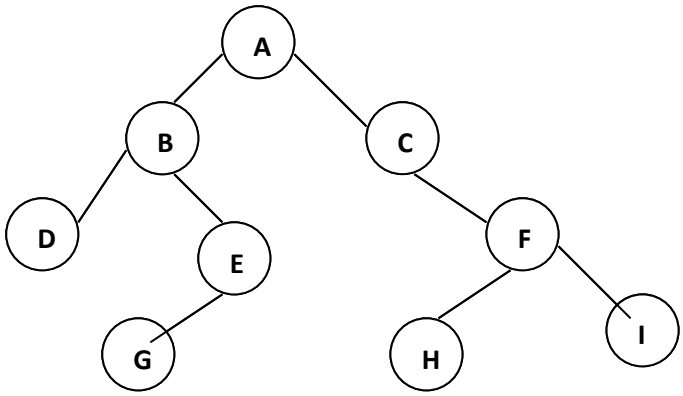
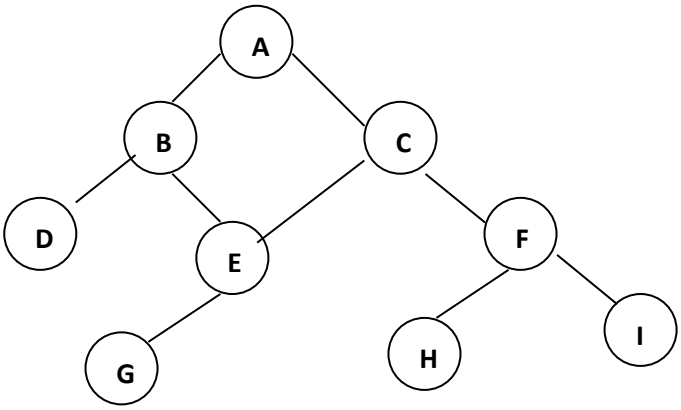
Unit – III

BINARY TREES

A binary tree is a finite set of elements that is either empty or is partitioned to tree disjoint subsets. The first subset contains a single element called the root of the tree. The other two subsets are themselves binary trees, called the left and right subtrees of the original tree. A left or right subtree can be empty. Each elements of a binary tree is called a node of the tree.



Binary tree (a)



Not binary tree (b)

If ‘A’ is the root of a binary tree and ‘B’ is the root of its left or right subtree, then ‘A’ is said to be the father of ‘B’ and ‘B’ is said to be the left or right son of ‘A’ . ‘A’ node that has no sons is called a leaf. Node n1 is an *ancestor* of need n2 if n1 is either the father of n2 or the father of some ancestor of n2. ‘A’ node n2 is a left *descendant* of need n1 if n2 is either the left son of n1 or a descendant of the left son of n1. A right descendant may be similarly defined. Two nodes are bothers if they are left and right sons of the same father.

Although natural trees grow with their roots in the ground and their leaves in the air, computer scientists almost universally portray tree data structures with the root at the top and the leaves at the bottom. The direction form the root to the leaves is “down” and the opposite direction is “up”. Going form the leaves to the root is called “climbing” the tree, and fooling form the root to leaves is called “descending” the tree.

If every non-leaf node in a binary tree has nonempty left and right subtree, the tree is termed a *strictly binary tree.* A strictly binary tree with ‘n’ leaves always contains 2n-1 nodes.

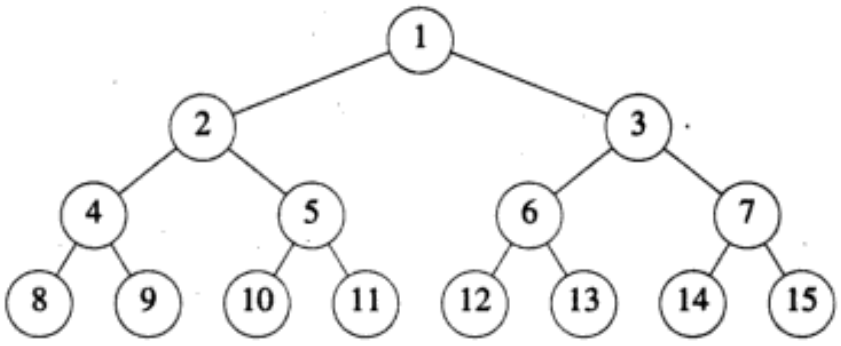
The level of a node in a binary tree is defined as follows: The root of the tree has level 0, and the level of any other node in the tree is one more than the level of its father. The depth of a binary tree is the maximum level of any leaf is the free. This equals the length of the longest path from the root to any leaf. This the depth of the tree of figure (a) is 3. A complete binary tree of depth ‘d’ is the strictly tree all of whose leaves are at level d. Figure (d) illustrates the complete binary tree of depth 3.

If a binary tree contains m nodes at level l , it contains at most 2m nodes at level l+1. Since a binary tree can contain at most one node at level 0, it can contain at most 2^l nodes at level l. A complete binary tree of depth d is the binary tree of depth d that contains exactly 2^l nodes at each level l between 0 and d. The total number of nodes in a complete binary tree of depth d, t_n, equals the sum of the number of nodes at each level between 0 and d. By induction, it can be shown that this sum equals 2^{d+1}-1. Since all leaves in such a tree are at level d, the tree contains 2^d leaves and, therefore, 2^d-1 non-leaf nodes.

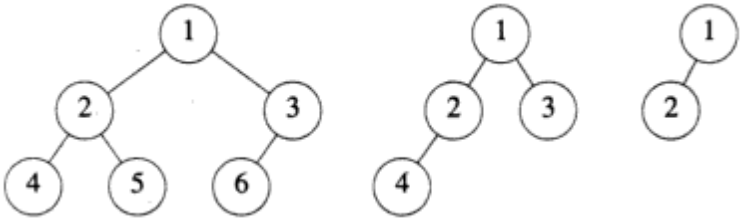
Properties of Binary Trees:

1. The drawing of every binary tree with n elements, n>0 has exactly n-1 edges
2. A binary tree of height h , h>=0 has at least h and at most 2^h-1 elements in it. (height calculated using nodes)
3. The height of a binary tree that contains n, n>=0, elements is at most ***n*** and at least ***log(n+1)***

A binary tree of height h that contains exactly 2^h-1 elements is called a **full binary tree**.



Full binary tree of height 4



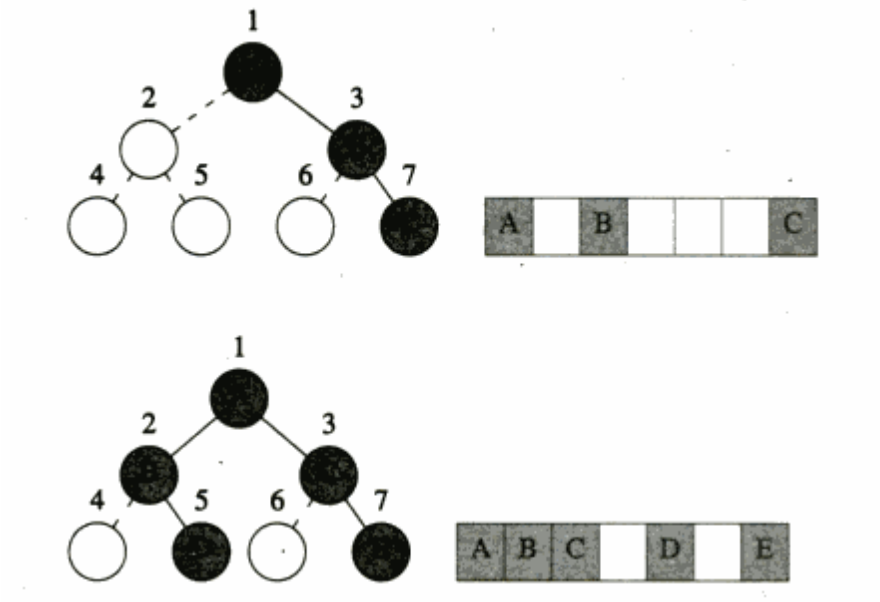
Complete binary trees

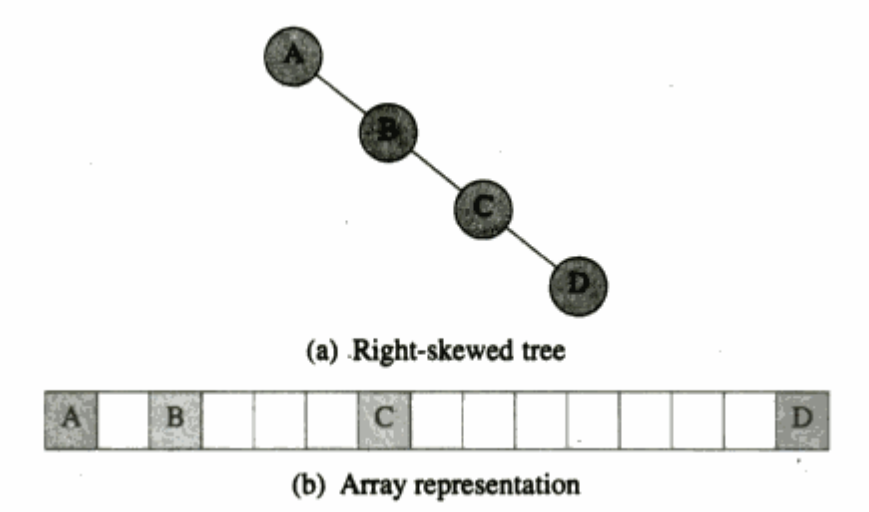
Let I, $1 \leq i \leq n$, be the number assigned to an elements of a complete binary tree. The following are true.

1. If i=1, then this element is the root of the binary tree. If i>1, then the parent of this element has been assigned the number *i/2*
2. If 2i > n , then this element has no left child. Otherwise, its left child has been assigned the number 2i
3. If 2i+1 > n then this element has no right child. Otherwise, its right child has been assigned the number 2i+1.

Representation of Binary Tree

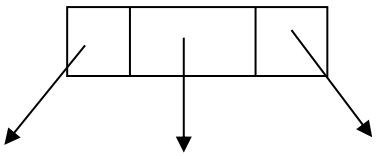
1. Array based Representation: In an array representation, the binary tree is represented in an array by storing each element at the array position corresponding to the number assigned to it.
   If the array index starts with '0'. Left child is stored in 2i+1 index, i is the index of parent node index. Right child is stored in 2i+2 index, i is the index of parent node index.





2.Linked Structure for Binary Tree

A natural way to realize a binary tree T is to use a linked structure. In this approach we represent each node v of T by an object with references to the element stored at v and to the position objects associated with the children



Applications of Binary Trees

A binary tree is a useful data structure when two way decisions must be made at each point in a process. For examples, suppose that we wanted to find all duplicates in a list of numbers. One way of doing this is to compare each number with all those that proceed if. However, this involves a large number of comparisons.

The number of comparisons can be reduced by using a binary tree. The first number in the list is placed in a node that is established as the root of a binary tree with empty left and right subtree. Each successive number in the list in the list is then compared to the number in the root. If it matches, we have a duplicate. It is smaller, we examine the left subtree, if it is larger, we examine the right subtree. If the subtree is empty, the number is not a duplicate and is placed into a new node at that position in the tree. If the subreee is nonempty, we compare the number to the contents of the root of the subtree and entire process is repeated with the subtree.

Another common operation is to traverse a binary tree, that is to pass though the tree, enumerating each of its nodes once. We may simply wish to print the contents of each node

as we enumerate it, or we may wish to process it in some other fashion. In either case we speak of visiting each node as it enumerated.

To traverse a nonempty binary tree in **preorder** (also known as **depth-first order**) we perform the following three operations:

1. Visit the root.
2. Traverse the left subtree in preorder.
3. Traverse the right subtee in preorder.

To traverse a nonempty binary tree in **inorder** (or **symmetric order**)

1. Traverse the left subtree in inorder.
2. Visit the root.
3. Traverse the right subtree in inorder.

To traverse a nonempty binary tree in **postorder**

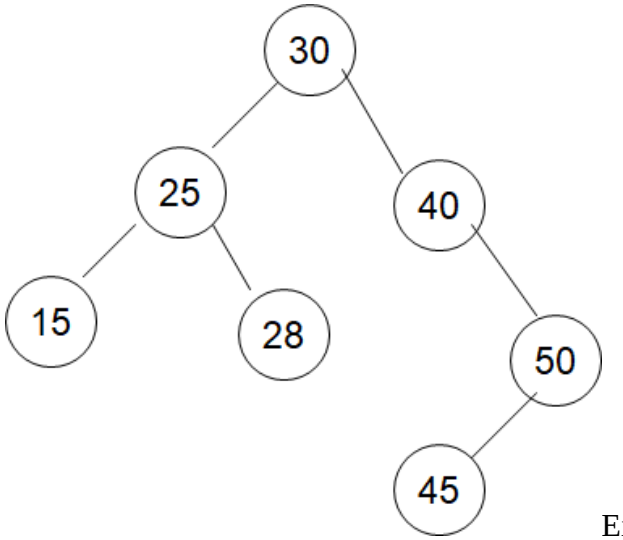
1. Traverse the left subtree in postorder.
2. Traverse the right subtree in postorder.
3. Visit the root.

Binary Search Tree

Definition: A *binary search tree* is a binary tree. It may be empty. If is not empty, then it satisfies the following properties

- Every element has a key and no two elements have the same key
- The keys in the left subtree are smaller than the key in the root.
- The keys in the right subtree are larger than the key in the root.
- The left and right subtrees are also binary search trees.

A binary search tree can support the operations search, insert, and delete.



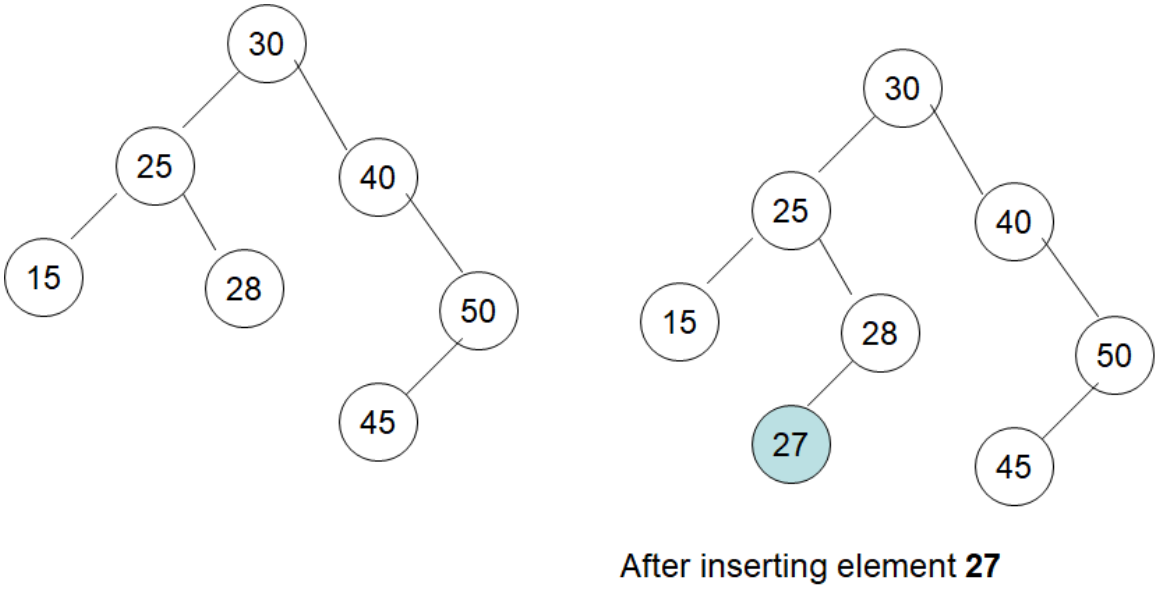
Example of Binary Search Tree

Searching an Element: Suppose we wish to search for an element with key k. We begin at the root. If the root is *null*, the search tree contains no elements and the search is unsuccessful. Otherwise, we compare k with the key in the root. If k is less than the key in the root, then no element in the right subtree can have key value k and only the left subtree is to be searched. If key is larger than the key in the root, only the right subtree needs to be searched. If k equals the key in the root then the search terminates successfully. The time complexity is O(h) where h is the height of the tree being searched.

Algorithm Search (t, x){

```
If (t = 0) then
   return 0;
else if( x = t -> data ) then
    return t;
else if( x < t->data ) then
   return Search( t->leftchild, x);
else
   return Search( t->rightchild, x);
}
```

Inserting an Element: To insert a new element e with key k into a binary search tree, we must first verify that its key is different form those of existing elements by performing a search for an element with the same key as that of e. if the search is unsuccessful, then the element is inserted at the point the search terminated.



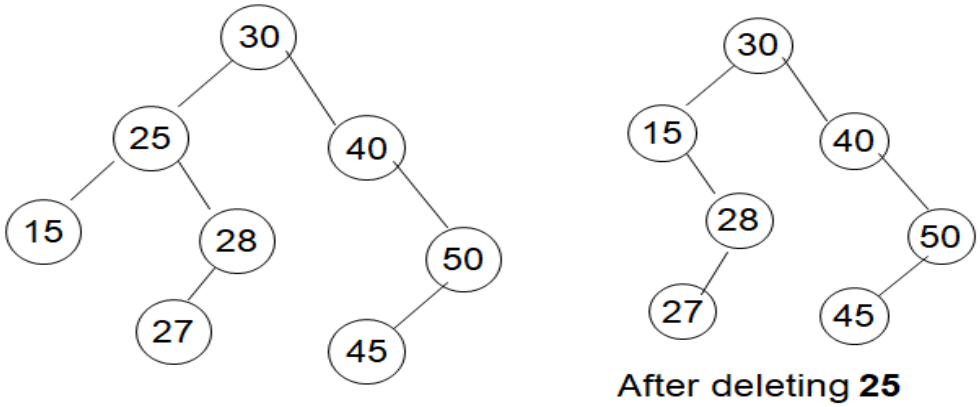
After inserting element **27**

Deleting an Element: For deletion we consider the three possibilities for the node *p* that contains the element to be deleted: 1) *p* is a leaf, 2) *p* has exactly one nonempty subtree, 3) *p* has exactly two nonempty subtrees.

Case(1) is handled by discarding the leaf node. To delete 24 form the tree of Figure(c) , the left child field of its parent is set to *null* and the node discarded. The resulting tree appears in Figure(a)

Next consider the case when the element to be deleted is in a node *p* that has only one nonempty subtree. If *p* has no parent (it is root), node *p* is discarded and the root of its single subtree becomes the new search tree root. If *p* has a parent *pp*, then we change the pointer from *pp* so that is points to *p's* only child and then delete the node *p*.

Finally, to delete an element in a node that has two nonempty subtrees, we replace this element with either the largest element in its left subtree or the smallest element in its right subtree.



After deleting **25**

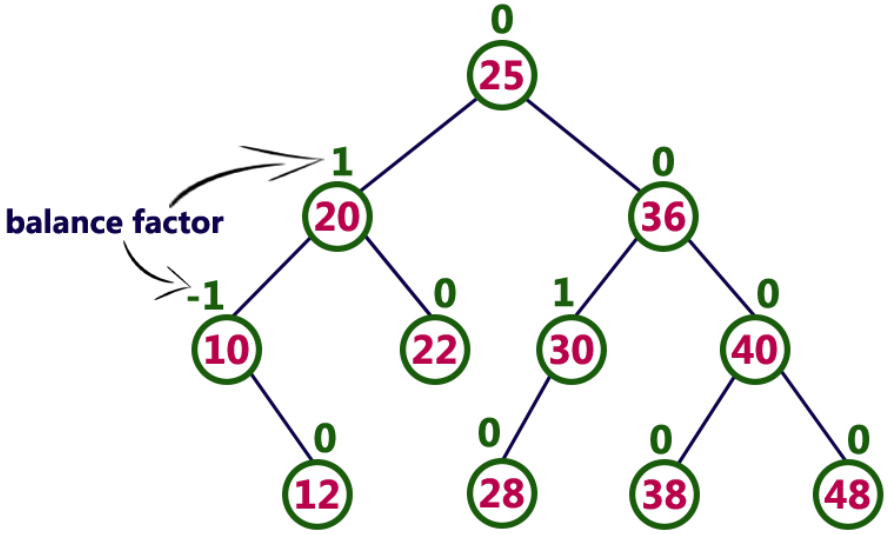
```
/* Binary Search Tree */


#include<stdio.h>
#include<stdlib.h>

struct nodetype{
struct nodetype *left;
int info;
struct nodetype *right;
};
typedef struct nodetype node;
node *root;
node* maketree(int);
void setleft(node*,int);
void setright(node*,int);
void pretrav(node*);
void intrav(node*);
void posttrav(node*);

int main()
{
node *p,*q;
int num;

printf("Enter element :");
scanf("%d",&num);
root=maketree(num);
while(scanf("%d",&num)!=EOF)  /* press Ctrl + Z to end the loop */
{
p=q=root;
while(num!=p->info && q!=NULL)
{
p=q;
if(num < p->info)
q=p->left;
else
q=p->right;
}
if(num==p->info)
printf("%d is duplicate\n",num);
else if(num<p->info)
setleft(p,num);
else
setright(p,num);
```

```
}
printf("\npreorder\n");
pretrav(root);
printf("\npostorder\n");
posttrav(root);
printf("\ninorder\n");
intrav(root);
return 0;
}

node* maketree(int x)
{
node *p;
p=(node*)malloc(sizeof(node));
p->info=x;
p->left=NULL;
p->right=NULL;
return p;
}

void setleft(node *p,int x)
{
if (p==NULL)
printf("void insertion\n");
else if(p->left!=NULL)
printf("invalid insertion");
else
p->left=maketree(x);
}

void setright(node *p,int x)
{
if (p==NULL)
printf("void insertion\n");
else if(p->right!=NULL)
printf("invalid insertion");
else
p->right=maketree(x);
}

void pretrav(node *tree)
{
if(tree!=NULL)
{
printf("%d\t",tree->info);
pretrav(tree->left);
```

```
pretrav(tree->right);
}
}
void intrav(node *tree)
{
if(tree!=NULL)
{
intrav(tree->left);
printf("%d\t",tree->info);
intrav(tree->right);
}
}
void posttrav(node *tree)
{
if(tree!=NULL)
{
posttrav(tree->left);
posttrav(tree->right);
printf("%d\t",tree->info);
}
}
```

AVL Tree

AVL(**Adelson**, **Velski** & **Landis**,) tree is a height-balanced binary search tree. That means, an AVL tree is also a binary search tree but it is a balanced tree. A binary tree is said to be balanced if, the difference between the heights of left and right subtrees of every node in the tree is either -1, 0 or +1. In other words, a binary tree is said to be balanced if the height of left and right children of every node differ by either -1, 0 or +1. In an AVL tree, every node maintains an extra information known as **balance factor**.

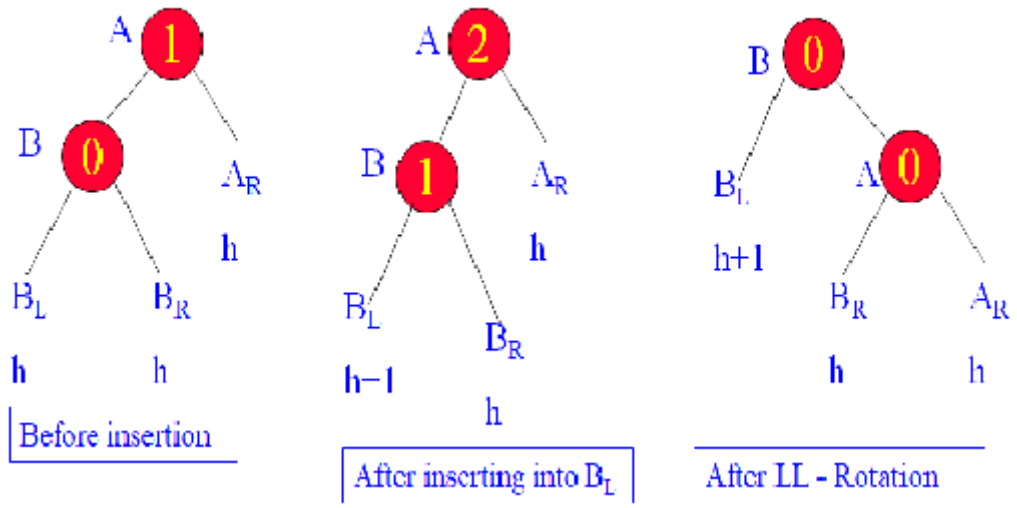


AVL Tree Rotation

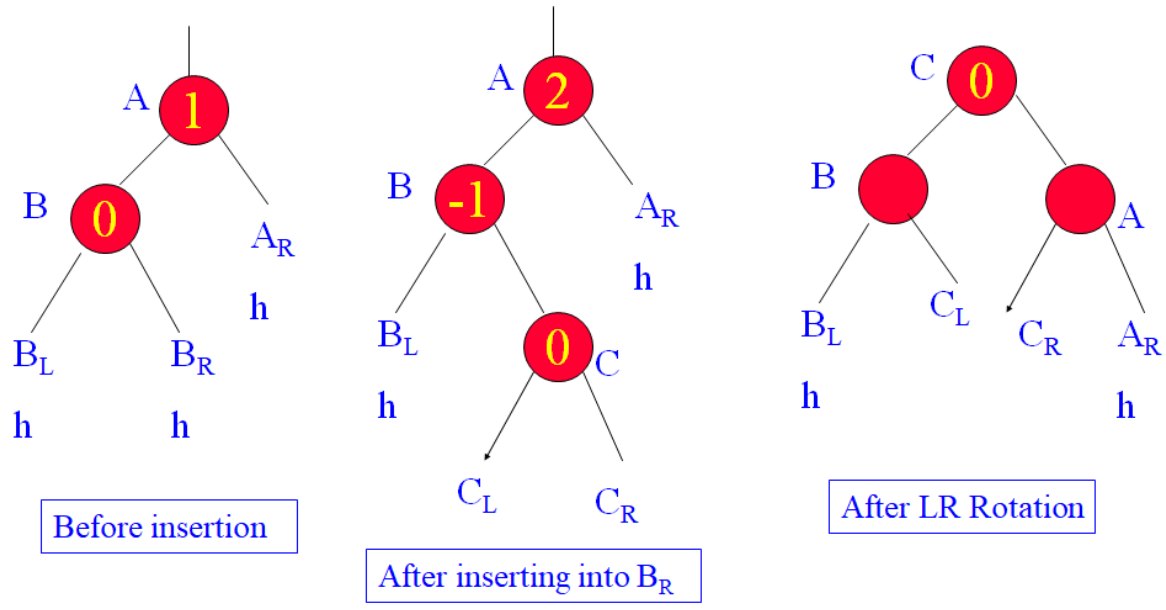
To balance itself, an AVL tree may perform the following four kinds of rotations −

- LL Rotation – new node is in the left subtree of the left subtree of A
- LR Rotation – new node is in the right subtree of left subtree of A
- RR Rotation – new node is in the right subtree of right subtree of A
- RL Rotation – new node is in the left subtree of right subtree of A

LL Rotation:



LR Rotation:



RR- Rotation: It is mirror image of LL Rotation

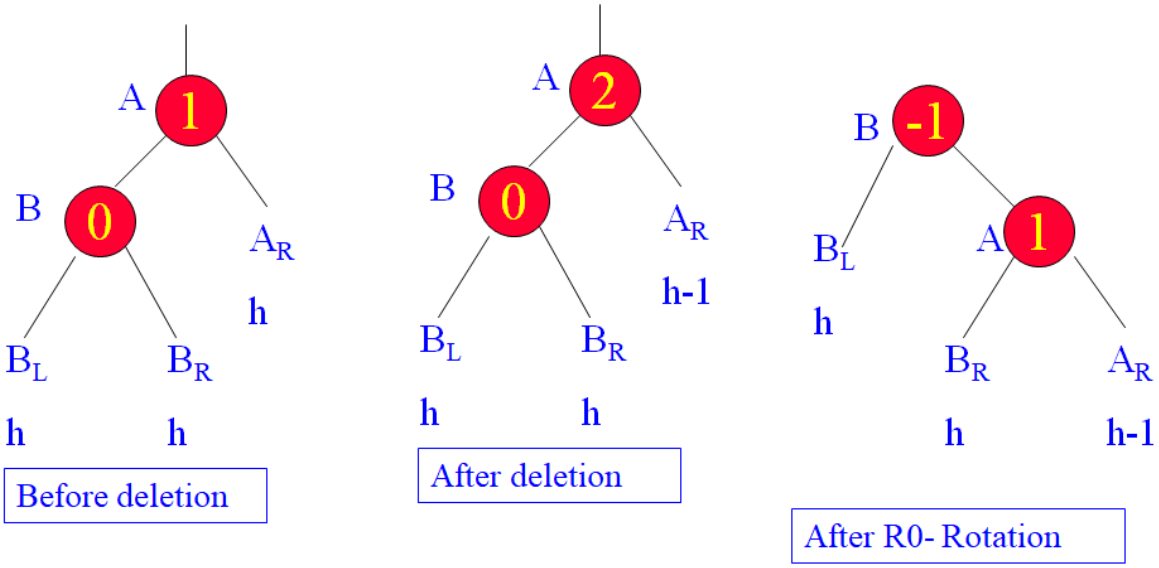
RL-Rotation: It is Mirror image of LR Rotation

Deletion Operation in AVL Tree

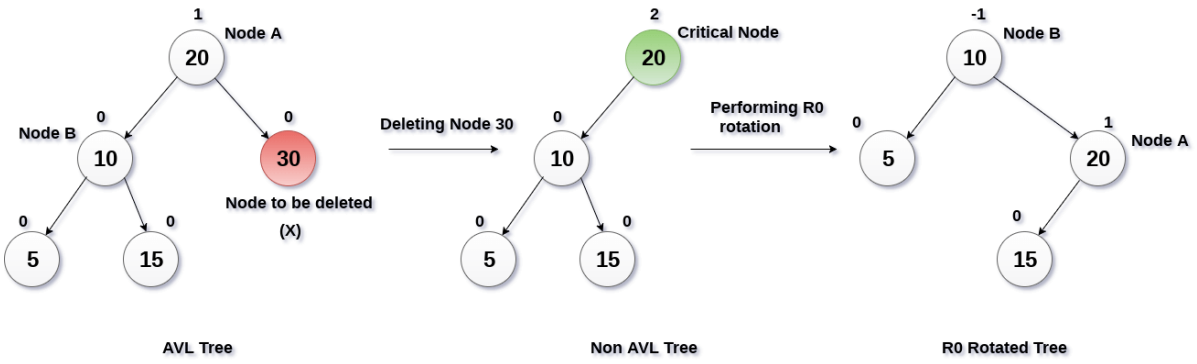
The deletion operation in AVL Tree is similar to deletion operation in BST. But after every deletion operation, we need to check with the Balance Factor condition. If the tree is balanced after deletion go for next operation otherwise perform suitable rotation to make the tree Balanced.

- 6 possible cases for rebalance the AVL tree
  - R0 rotation – node is deleted at right subtree and left subtree balance factor is 0
  - R1 rotation
  - R-1 rotation
  - L0 rotation
  - L1 rotation
  - L-1 rotation

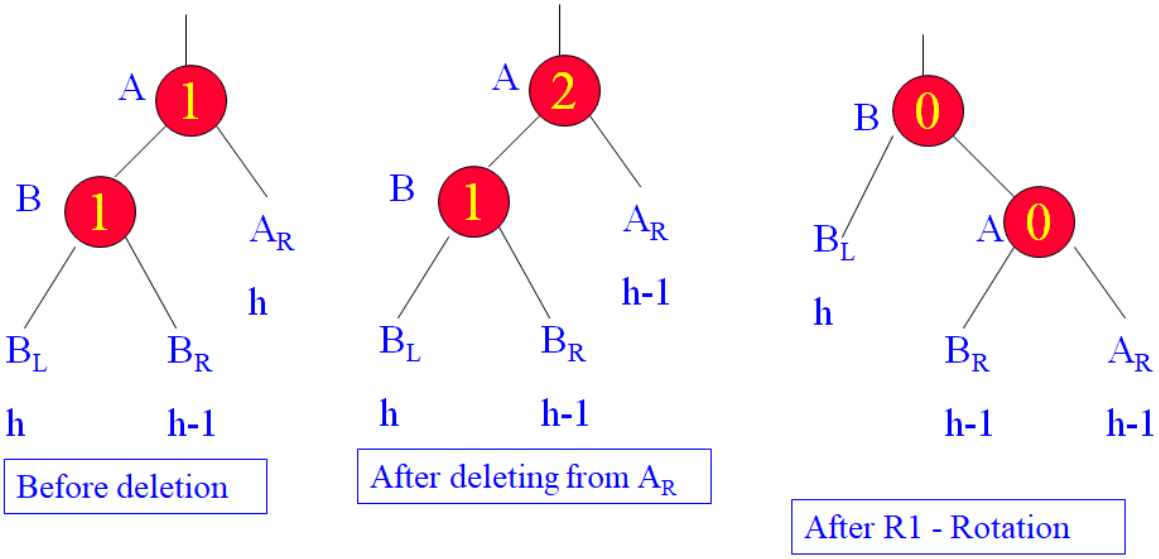
R0 rotation:



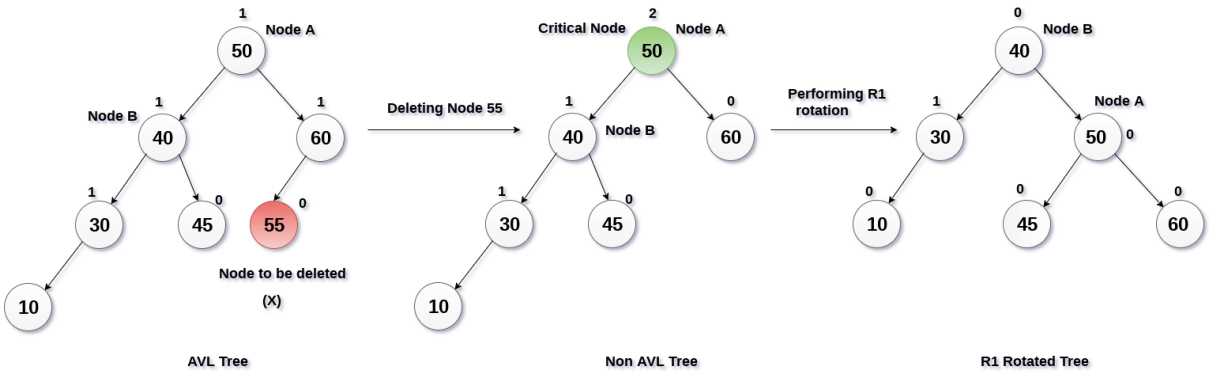
Example:



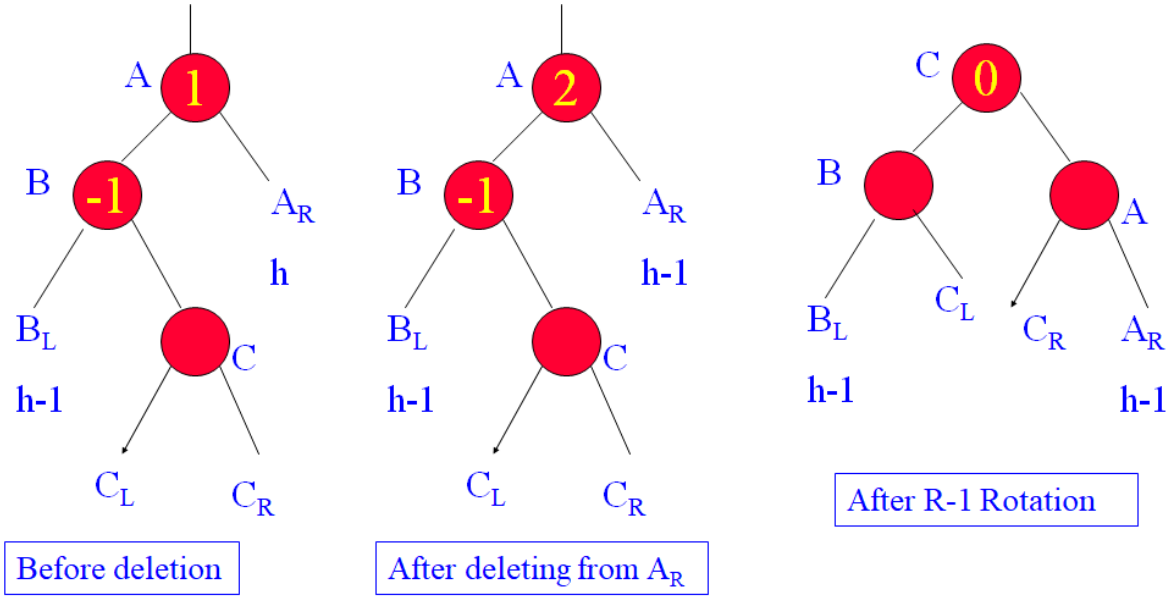
R1 rotation:



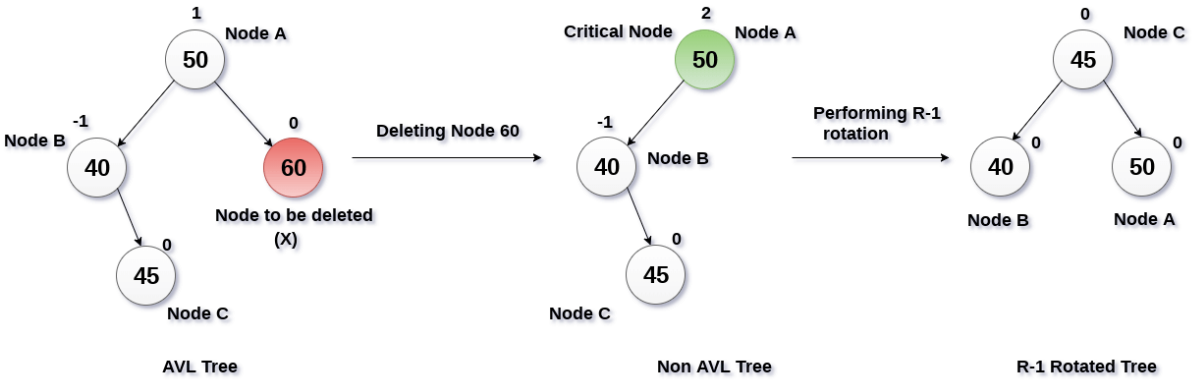
Example:



R-1 rotation:



Example;



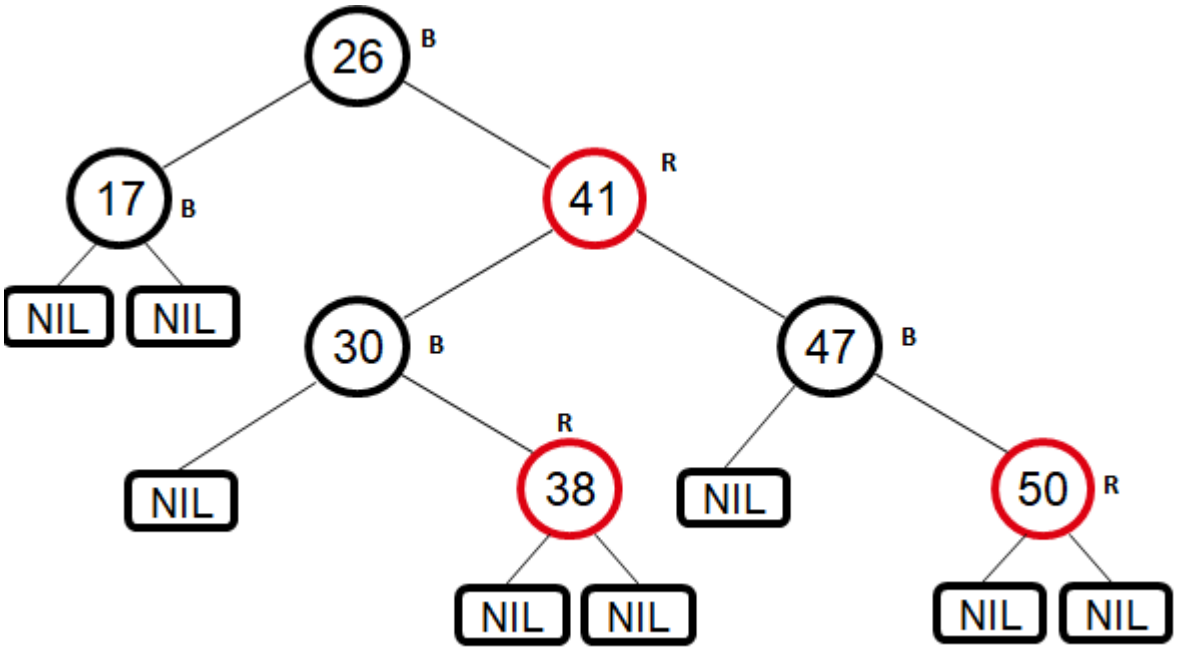
Red -Black Tree

Red - Black Tree is another variant of Binary Search Tree in which every node is colored either RED or BLACK. We can define a Red Black

- "Balanced" binary search trees guarantee an O(lgn) running time
- Red-black-tree

Binary search tree with an additional attribute for its nodes: color which can be **red** or **black**
Constrains the way nodes can be colored on any path from the root to a leaf:

Example of Red-Black Tree



Red-Black Tree Properties:

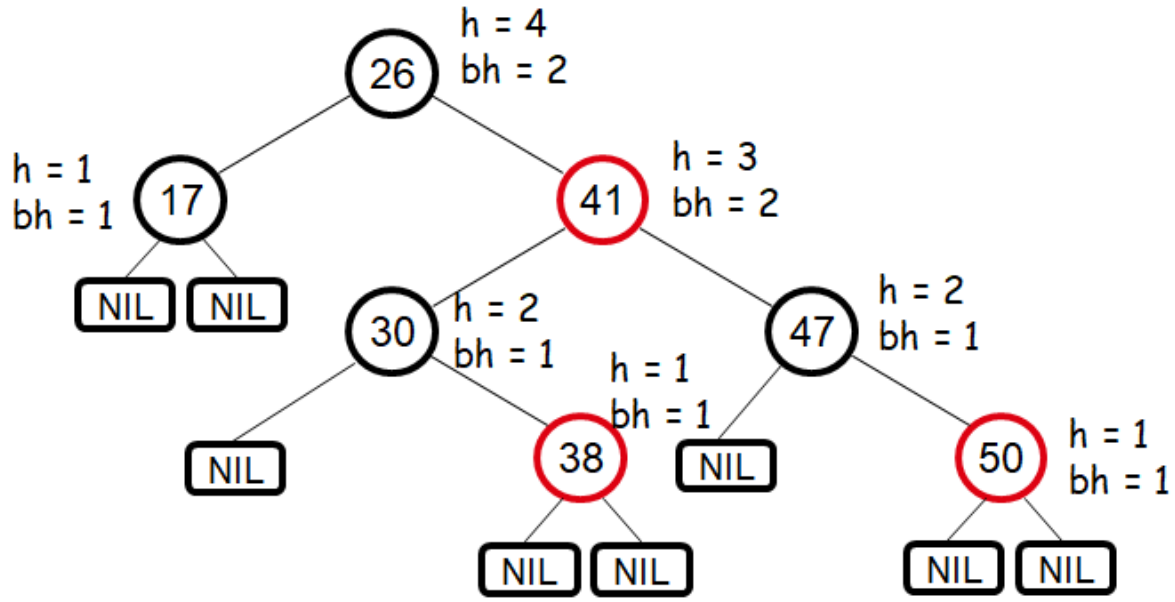
1. Every node is either **red** or **black**
2. The root is **black**
3. Every leaf (NIL) is **black**
4. If a node is **red**, then both its children are **black**

No two consecutive red nodes on a simple path from the root to a leaf

5. For each node, all paths from that node to descendant leaves contain the same number of **black** nodes

Black Height of a Node:

- **Height of a node:** the number of edges in the **longest** path to a leaf
- **Black-height** of a node x: bh(x) is the number of black nodes (including NIL) on the path from x to a leaf, not counting x



Theorem: A red-black tree with n internal nodes has height at most $2\lg(n + 1)$

Proof: Thus at the root of the red-black tree:

$n \geq 2^{bh(root)} - 1$

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

$\log(n+1) \geq h/2$

$h \leq 2 \log(n + 1)$

Thus $h = O(\lg n)$

Red-Black Tree Rotations:

- Operations for re-structuring the tree after insert and delete operations on red-black trees
- Rotations take a red-black-tree and a node within the tree and:

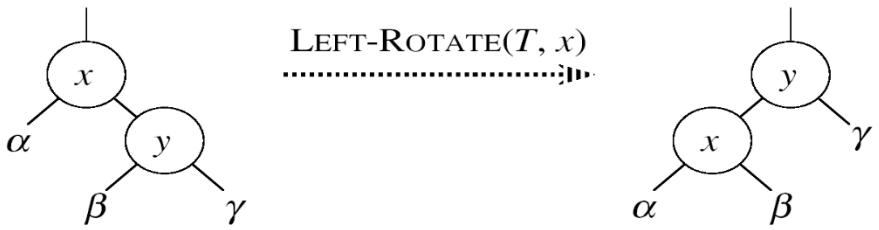
Together with some node re-coloring they help restore the red-black-tree property
Change some of the pointer structure
Do not change the binary-search tree property

- Two types of rotations:
  Left & right rotations

Left Rotation:

- Assumptions for a left rotation on a node x:

The right child of x (y) is not NIL



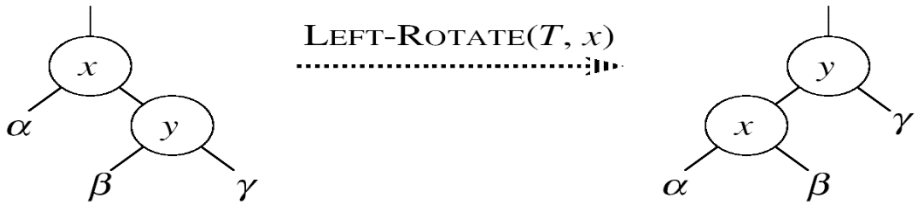
- Idea:

Pivots around the link from x to y
Makes y the new root of the subtree
x becomes y's left child
y's left child becomes x's right child

Right Rotation:

- Assumptions for a left rotation on a node x:

The right child of x (y) is not NIL



- Idea:

Pivots around the link from x to y
Makes y the new root of the subtree

x becomes y's left child
y's left child becomes x's right child

Insert Operation:

- Goal:

Insert a new node z into a red-black-tree

- Idea:

Insert node z into the tree as for an ordinary binary search tree
Color the node **red**
Restore the red-black-tree properties
Use an auxiliary procedure RB-INSERT-FIXUP

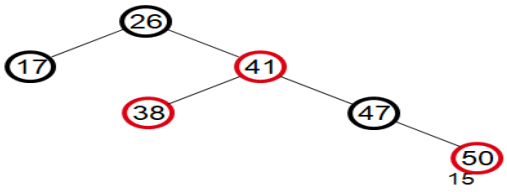
1. Every node is either red or black
2. The root is black
3. Every leaf (NIL) is black
4. If a node is red, then both its children are black
5. For each node, all paths from the node to descendant leaves contain the same number of black nodes

OK!
If z is the root
⇒ not OK

OK!

If p(z) is red ⇒ not OK
z and p(z) are both red

OK!



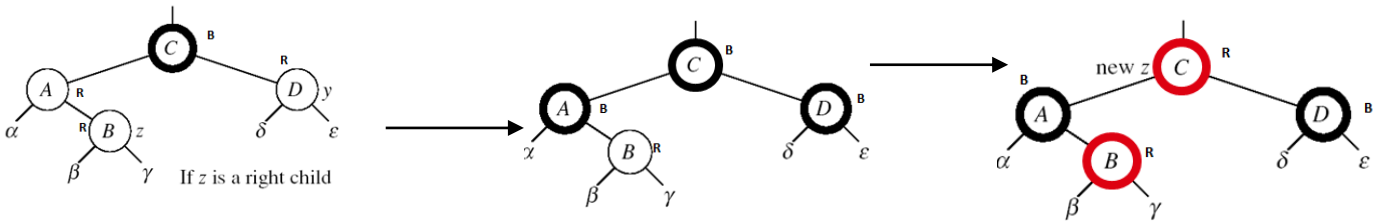
Red-Black Tree Fixup Case-1:

z's "uncle" (y) is **red**

Idea: (z is a right child)

- p[p[z]] (z's grandparent) must be black: z and p[z] are both red
- Color p[z] **black**
- Color y **black**
- Color p[p[z]] **red**
- z = p[p[z]]

Push the **"red"** violation up the tree

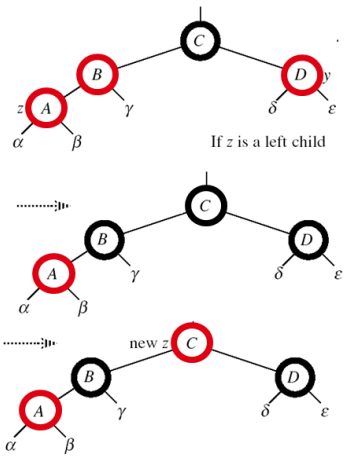


z's "uncle" (y) is **red**

Idea: (z is a left child)

- p[p[z]] (z's grandparent) must be black: z and p[z] are both red
- color p[z] ← **black**
- color y ← **black**
- color p[p[z]] ← **red**
- z = p[p[z]]

Push the **"red"** violation up the tree

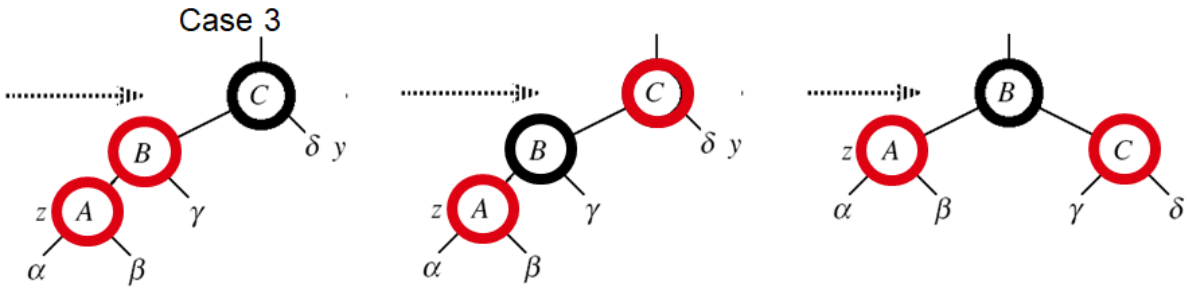


Case 3

- z's "uncle" (y) is **black**
- z is a left child

Idea:

- color p[z] ← **black**
- color p[p[z]] ← **red**
- RIGHT-ROTATE(T, p[p[z]])
- No longer have 2 reds in a row
- p[z] is now black

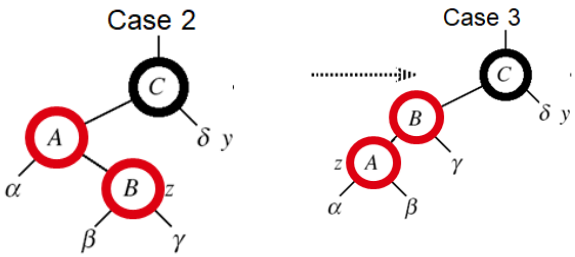


Case 2:

- z's "uncle" (y) is **black**
- z is a right child

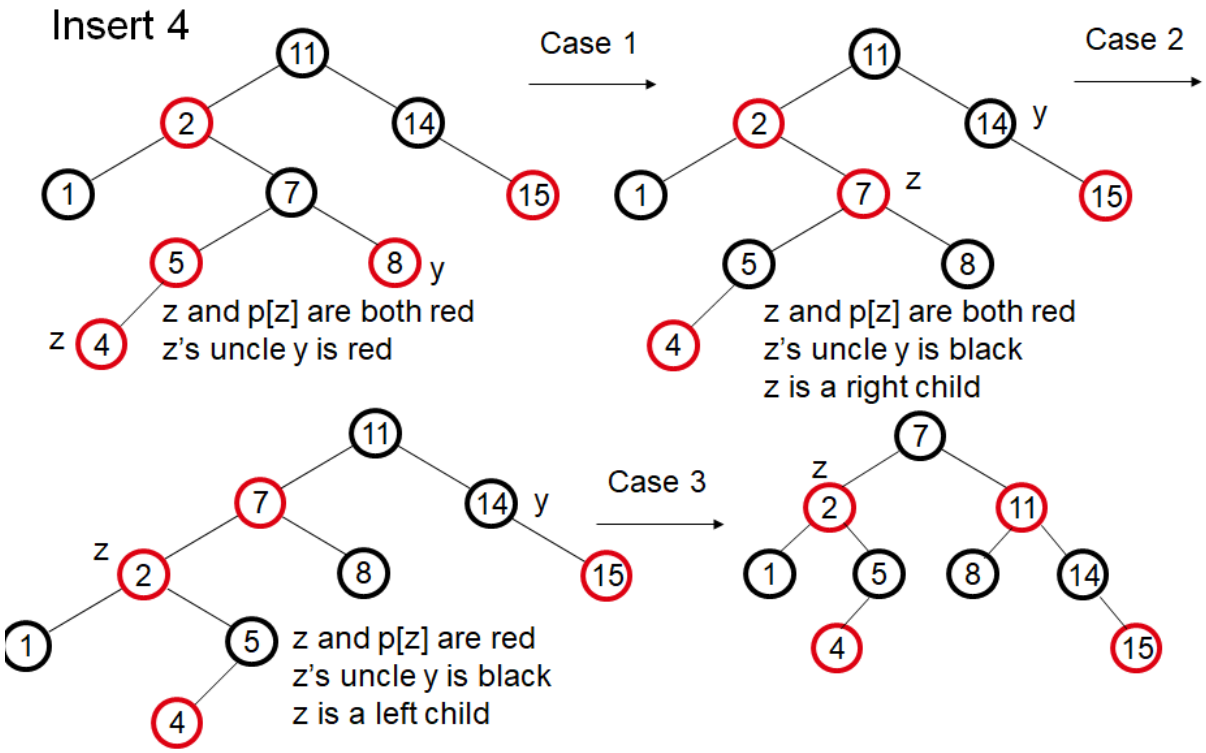
Idea:

- z ← p[z]
- LEFT-ROTATE(T, z)



⇒ now z is a left child, and both z and p[z] are red ⇒ case 3

Example:



- Operations on red-black-trees:

SEARCH	O(h)
PREDECESSOR	O(h)
SUCCESOR	O(h)
MINIMUM	O(h)
MAXIMUM	O(h)
INSERT	O(h)
DELETE	O(h)

- Red-black-trees guarantee that the height of the tree will be O(lgn)

Splay Trees:

Splay tree is another variant of a binary search tree. In a splay tree, recently accessed element is placed at the root of the tree. A splay tree is defined as follows..

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"**.

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

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.

By splaying elements we bring more frequently used elements closer to the root of the tree so that any operation on those elements is performed quickly. That means the splaying operation automatically brings more frequently used elements closer to 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.

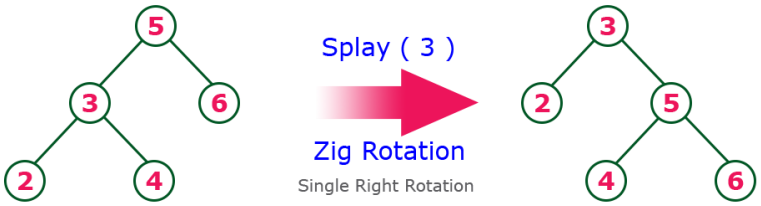
In splay tree, to splay any element we use the following rotation operations...

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

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. Consider the following example...

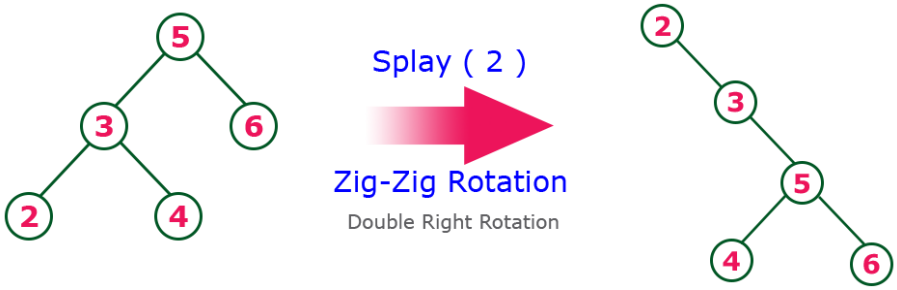


Zag rotation:

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. Consider the following example...

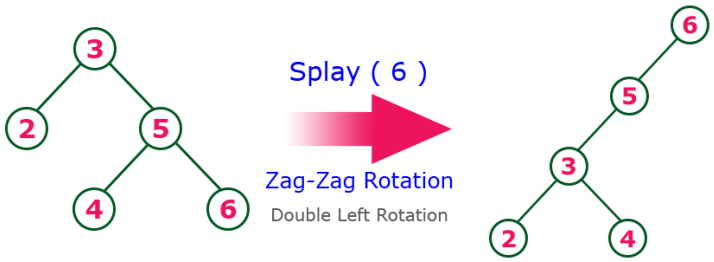
Zig-Zig Rotation:

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. Consider the following example...



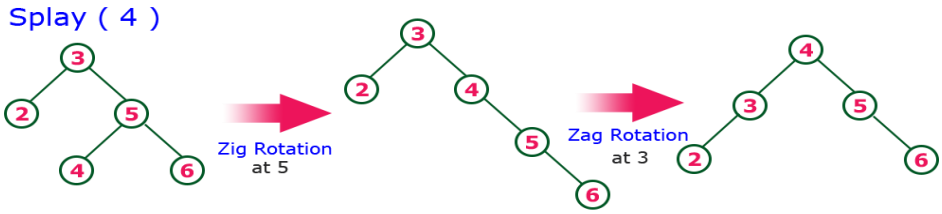
Zag-Zag rotation:

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. Consider the following example...



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. Consider the following example...



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. Consider the following example...