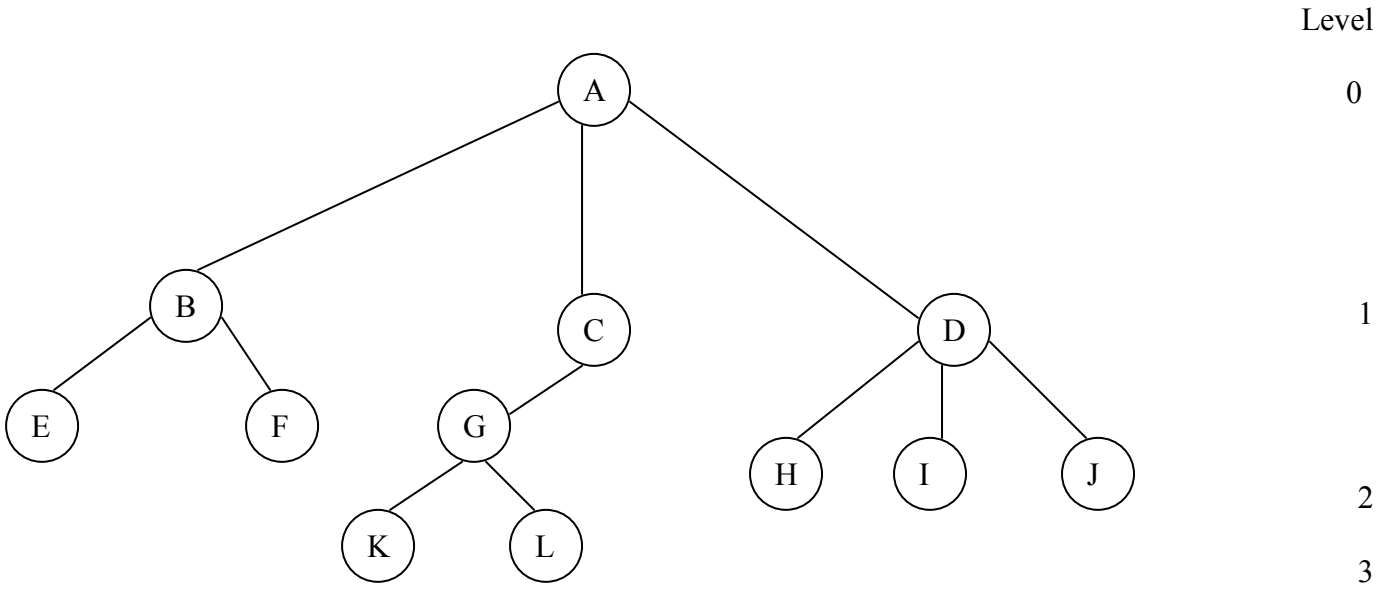
UNIT – III

Non Linear data structures-Trees-Basic Terminology, Binary tree ADT,array and linked representations,iterative traversals,threaded binary trees,Applications-Disjoint-Sets,Union and Find algorithms,Huffman coding,General tree to binary tree conversion, Realizing a Priority Queue using Heap.
Search Trees- Binary Search Tree ADT, Implementation, Operations- Searching, Insertion and Deletion, Balanced Search trees-AVL Trees, Operations – Insertion and Searching,B-Trees, B-Tree of order m,Operations- Insertion,Deletion and Searching,Introduction to Red-BlackTrees, Splay Trees,B*-Trees,B+-Trees(Elementary treatement), Comparison of Search Trees,Trees in java.util.

NON LINEAR DATA STRUCTURES

TREES

A Tree is a data structure in which each element is attached to one or more elements directly beneath it.



Terminology

- The connections between elements are called **branches**.
- A tree has a single root, called **root** node, which is shown at the top of the tree. i.e. root is always at the highest level 0.
- Each node has exactly one node above it, called **parent.** Eg: A is the parent of B,C and D.

- The nodes just below a node are called its **children**. ie. child nodes are one level lower than the parent node.
- A node which does not have any child called **leaf or terminal node**.
- Nodes with at least one child are called **non terminal or external nodes**. Eg: E, F, K, L, H, I and M are leaves.
- The child nodes of same parent are said to be **siblings**.
- A **path** in a tree is a list of distinct nodes in which successive nodes are connected by branches in the tree.
- The **length** of a particular path is the number of branches in that path.
- The **degree** of a node of a tree is the number of children of that node.
- The maximum number of children a node can have is often referred to as the **order** of a tree.
- The **height or depth** of a tree is the length of the longest path from root to any leaf.

1. **Root:** This is the unique node in the tree to which further sub trees are attached. Eg: A
2. **Degree of the node:** The total number of sub-trees attached to the node is called the degree of the node.Eg: For node A degree is 3. For node K degree is 0
3. **Leaves:** These are the terminal nodes of the tree. The nodes with degree 0 are always the leaf nodes. Eg: E, F, K, L,H, I, J
4. **Internal nodes:** The nodes other than the root node and the leaves are called the internal nodes. Eg: B, C, D, G
5. **Parent nodes:** The node which is having further sub-trees(branches) is called the parent node of those sub-trees. Eg: B is the parent node of E and F.
6. **Predecessor:** While displaying the tree, if some particular node occurs previous to some other node then that node is called the predecessor of the other node. Eg: E is the predecessor of the node B.
7. **Successor:** The node which occurs next to some other node is a successor node. Eg: B is the successor of E and F.
8. **Level of the tree:** The root node is always considered at level 0, then its adjacent children are supposed to be at level 1 and so on. Eg: A is at level 0, B,C,D are at level 1, E,F,G,H,I,J are at level 2, K,L are at level 3.
9. **Height of the tree:** The maximum level is the height of the tree. Here height of the tree is 3. The height if the tree is also called depth of the tree.
10. **Degree of tree:** The maximum degree of the node is called the degree of the tree.

BINARY TREES

Binary tree is a tree in which each node has at most two children, a left child and a right child. Thus the order of binary tree is 2.

A binary tree is either empty or consists of
a) a node called the root
b)left and right sub trees are themselves binary trees.

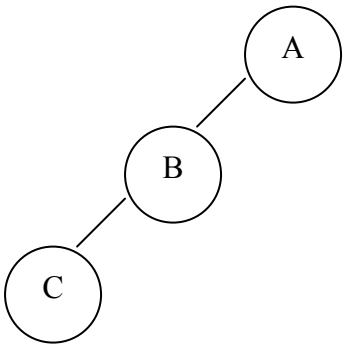
A binary tree is a finite set of nodes which is either empty or consists of a root and two disjoint trees called left sub-tree and right sub-tree.

In binary tree each node will have one data field and two pointer fields for representing the sub-branches. The degree of each node in the binary tree will be at the most two.

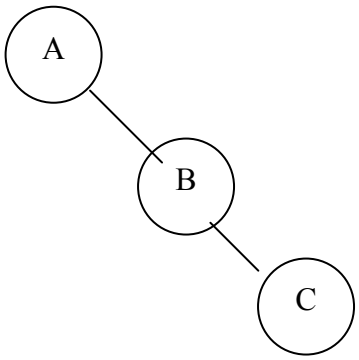
Types Of Binary Trees:

There are 3 types of binary trees:

1. **Left skewed binary tree:** If the right sub-tree is missing in every node of a tree we call it as left skewed tree.

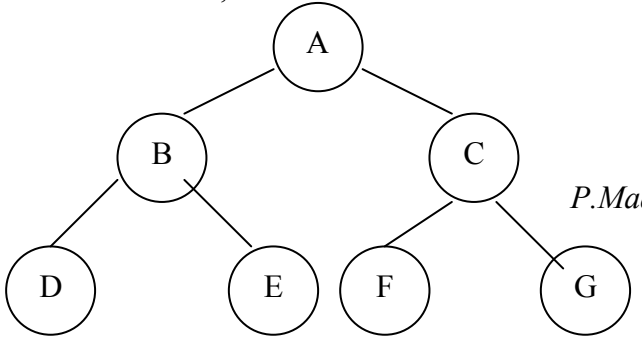


2. **Right skewed binary tree:** If the left sub-tree is missing in every node of a tree we call it is right sub-tree.



3. **Complete binary tree:**

The tree in which degree of each node is at the most two is called a complete binary tree. In a complete binary tree there is exactly one node at level 0, two nodes at level 1 and four nodes at level 2 and so on. So we can say that a complete binary tree depth d will contain exactly 2^l nodes at each level l, where l is from 0 to d.



Note:

1. A binary tree of depth n will have maximum 2^n-1 nodes.
2. A complete binary tree of level *l* will have maximum 2*l* nodes at each level, where *l* starts from 0.
3. Any binary tree with n nodes will have at the most n+1 null branches.
4. The total number of edges in a complete binary tree with n terminal nodes are 2(n-1).

Binary Tree Representation

A binary tree can be represented mainly in 2 ways:

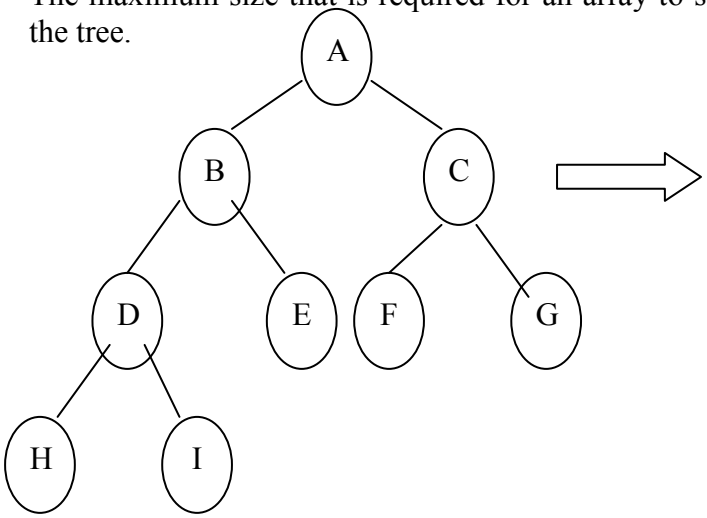
a) Sequential Representation
b) Linked Representation

a) Sequential Representation

The simplest way to represent binary trees in memory is the sequential representation that uses one-dimensional array.

1) The root of binary tree is stored in the 1 st location of array
2) If a node is in the j th location of array, then its left child is in the location 2J and its right child in the location 2J+1

The maximum size that is required for an array to store a tree is 2^{d+1}-1, where d is the depth of the tree.



POSITION	ARRAY
0	A
1	B
2	C
3	D
4	E
5	F
6	G
7	H
8	I
.	.
.	.
.	.
.	.
.	.

Advantages of sequential representation:

The only advantage with this type of representation is that the direct access to any node can be possible and finding the parent or left children of any particular node is fast because of the random access.

Disadvantages of sequential representation:

1. The major disadvantage with this type of representation is wastage of memory. For example in the skewed tree half of the array is unutilized.
2. In this type of representation the maximum depth of the tree has to be fixed. Because we have decide the array size. If we choose the array size quite larger than the depth of the tree, then it will be wastage of the memory. And if we choose array size lesser than the depth of the tree then we will be unable to represent some part of the tree.
3. The insertions and deletion of any node in the tree will be costlier as other nodes have to be adjusted at appropriate positions so that the meaning of binary tree can be preserved.

As these drawbacks are there with this sequential type of representation, we will search for more flexible representation. So instead of array we will make use of linked list to represent the tree.

b) Linked Representation

Linked representation of trees in memory is implemented using pointers. Since each node in a binary tree can have maximum two children, a node in a linked representation has two pointers for both left and right child, and one information field. If a node does not have any child, the corresponding pointer field is made NULL pointer.

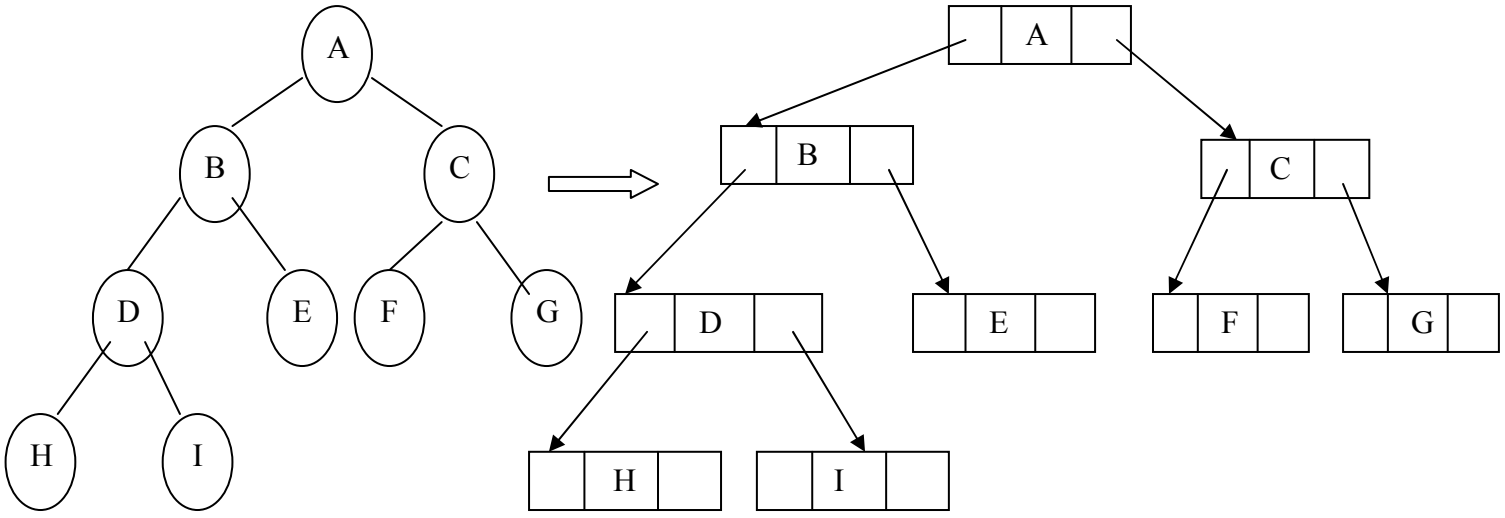
In linked list each node will look like this:

Advantages of linked representation:

1. This representation is superior to our array representation as there is no wastage of memory. And so there is no need to have prior knowledge of depth of the tree. Using dynamic memory concept one can create as much memory(nodes) as required. By chance if some nodes are unutilized one can delete the nodes by making the address free.
2. Insertions and deletions which are the most common operations can be done without moving the nodes.

Disadvantages of linked representation:

1. This representation does not provide direct access to a node and special algorithms are required.
2. This representation needs additional space in each node for storing the left and right sub-trees.



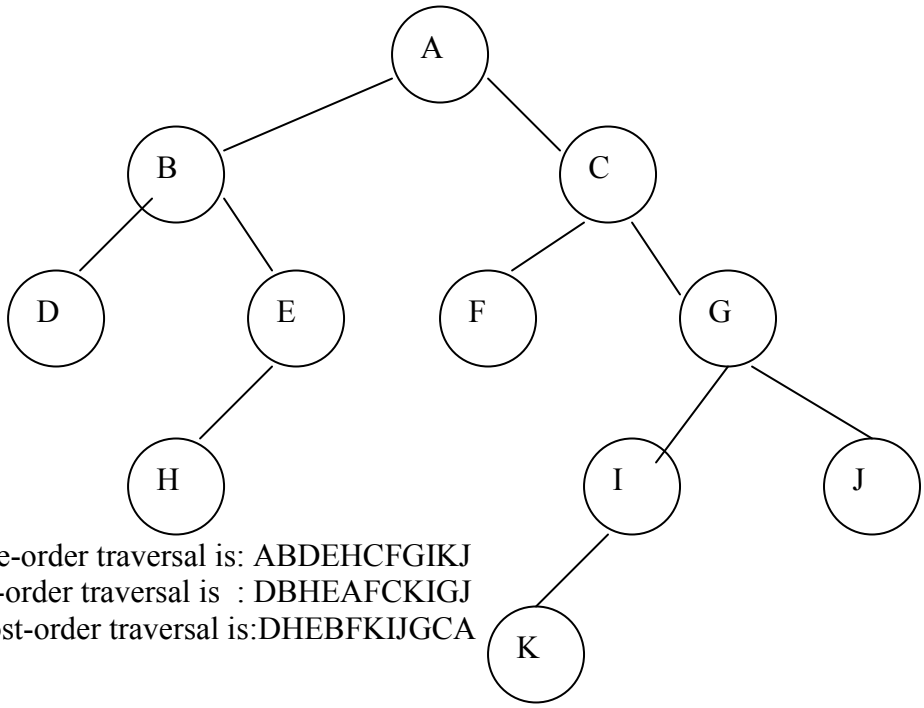
Linked Representation

TRAVERSING A BINARY TREE

Traversing a tree means that processing it so that each node is visited exactly once. A binary tree can be traversed a number of ways.
The most common tree traversals are

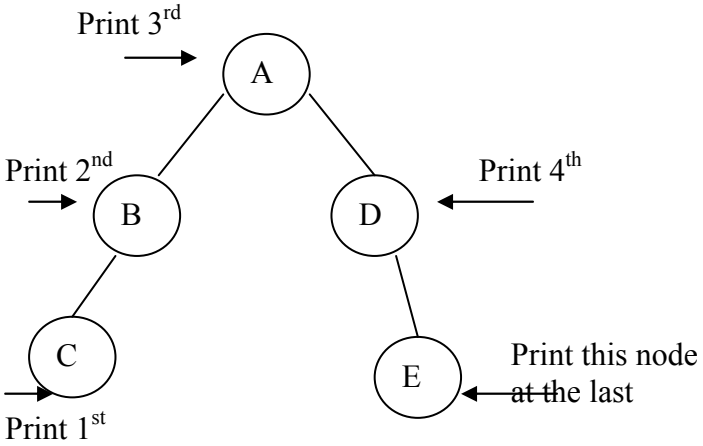
- in-order
- pre-order and
- post-order

Pre-order	1.Visit the root 2.Traverse the left sub tree in pre-order 3.Traverse the right sub tree in pre-order.	Root | Left | Right
In-order	1.Traverse the left sub tree in in-order 2.Visit the root 3.Traverse the right sub tree in in-order.	Left | Root | Right
Post-order	1.Traverse the left sub tree in post-order 2.Traverse the right sub tree in post-order. 3.Visit the root	Left | Right | Root



The pre-order traversal is: ABDEHCFGIKJ
The in-order traversal is : DBHEAFCKIGJ
The post-order traversal is:DHEBFKIJGCA

Inorder Traversal:



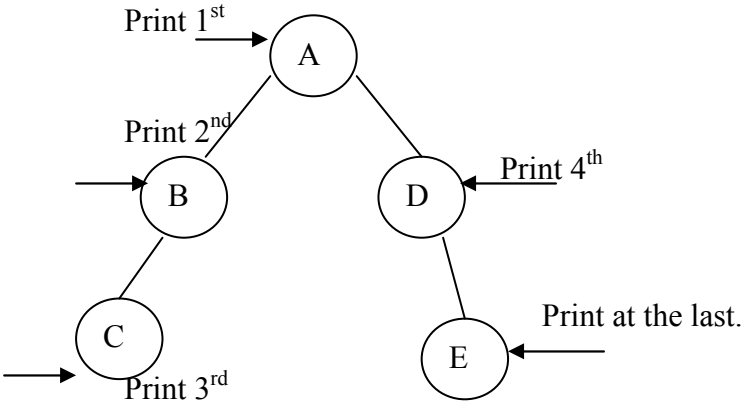
C-B-A-D-E is the inorder traversal i.e. first we go towards the leftmost node. i.e. C so print that node C. Then go back to the node B and print B. Then root node A then move towards the right sub-tree print D and finally E. Thus we are following the tracing sequence of Left|Root|Right.

This type of traversal is called inorder traversal. The basic principle is to traverse left sub-tree then root and then the right sub-tree.

Pseudo Code:

```
template <class T>
void inorder(bintree<T> *temp)
{
if(temp!=NULL)
{
inorder(temp.left);
System.out.println("temp.data";
inorder(temp.right);
}
}
```

Preorder Traversal:

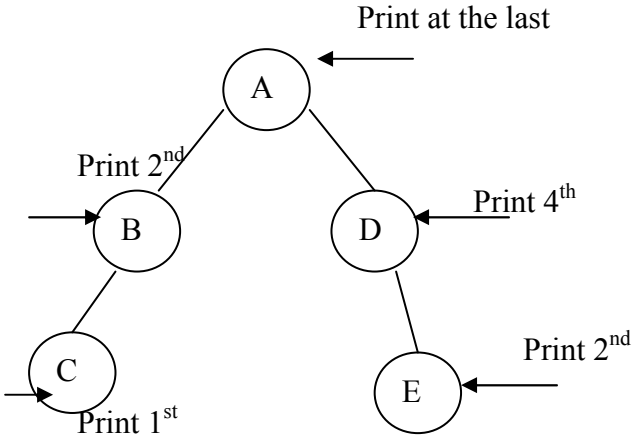


A-B-C-D-E is the preorder traversal of the above fig. We are following Root|Left|Right path i.e. data at the root node will be printed first then we move on the left sub-tree and go on printing the data till we reach to the left most node. Print the data at that node and then move to the right sub-tree. Follow the same principle at each sub-tree and go on printing the data accordingly.

Pseudo Code:

```
template <class T>
void inorder(bintree<T> *temp)
{
if(temp!=NULL)
{
System.out.println("temp.data";
preorder(temp.left);
preorder(temp.right);
}
}
```

Postorder Traversal:



From figure the postorder traversal is C-D-B-E-A. In the postorder traversal we are following the Left|Right|Root principle i.e. move to the leftmost node, if right sub-tree is there or not if not then print the leftmost node, if right sub-tree is there move towards the right most node. The key idea here is that at each sub-tree we are following the Left|Right|Root principle and print the data accordingly.

Pseudo Code:

```
template <class T>
void inorder(bintree<T> *temp)
{
if(temp!=NULL)
{
postorder(temp.left);
postorder(temp.right);
System.out.println(”temp.data”;
}
}
```

CREATING A BINARY TREE FROM A GENERAL TREE

The process of converting the general tree to a binary tree is as follows:

* use the root of the general tree as the root of the binary tree

* determine the first child of the root. This is the leftmost node in the general tree at the next level

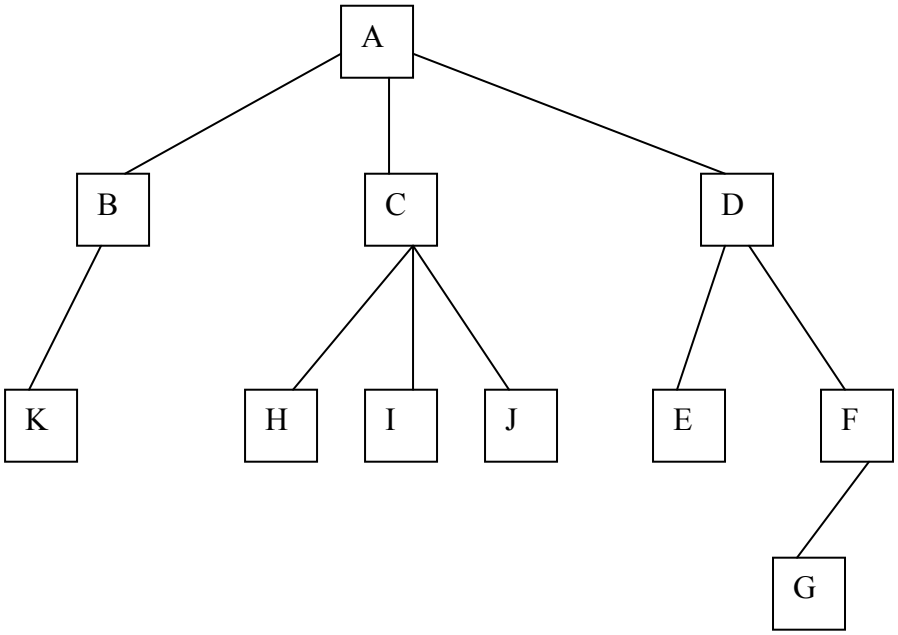
* insert this node. The child reference of the parent node refers to this node

* continue finding the first child of each parent node and insert it below the parent node with the child reference of the parent to this node.

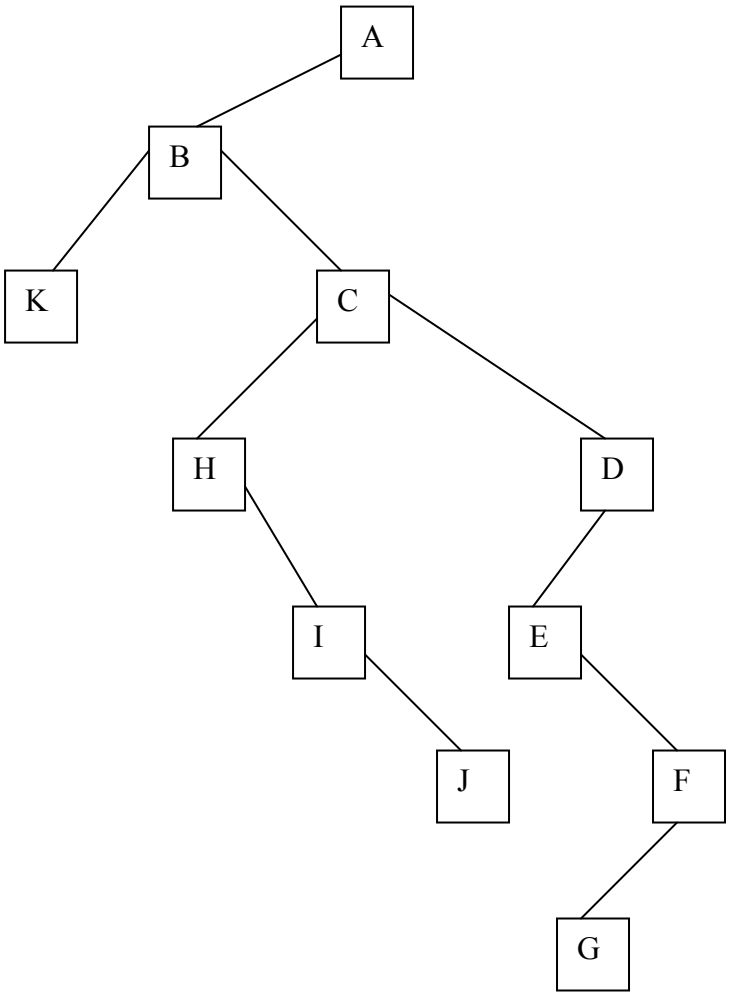
* when no more first children exist in the path just used, move back to the parent of the last node entered and repeat the above process. In other words, determine the first sibling of the last node entered.

* complete the tree for all nodes. In order to locate where the node fits you must search for the first child at that level and then follow the sibling references to a nil where the next sibling can be inserted. The children of any sibling node can be inserted by locating the parent and then inserting the first child. Then the above process is repeated.

Given the following general tree:



The following is the binary version:



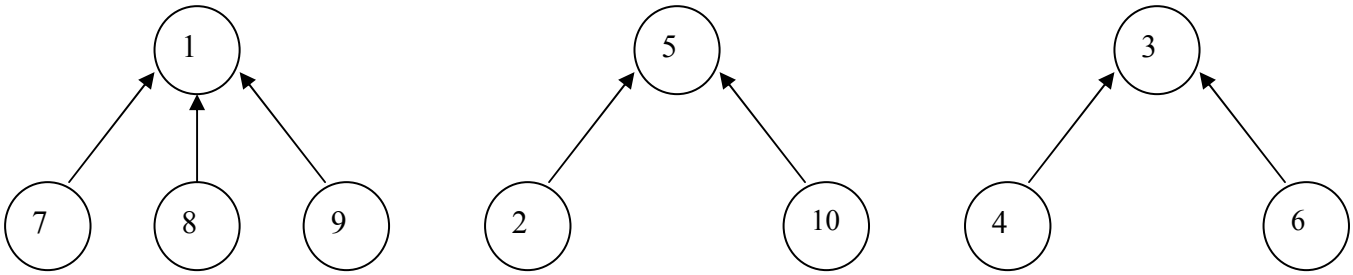
DISJOINT SET OPERATIONS

INTRODUCTION:

This is a use of forests in the representation of sets. Assume the elements of the sets are the numbers 1, 2, 3,……,n. These numbers might be indices into a symbol table in which the names of the elements are stored. We assume that the sets being represented are pair wise disjoint(i.e. if S_i and S_j, $i \neq j$, are two sets, then there is no element that is in both S_i and S_j).

For example, when n=10, the elements can be partitioned into three disjoint sets, $S_1=\{1, 7, 8, 9\}$, $S_2=\{2, 5, 10\}$ and $S_3=\{3, 4, 6\}$

Fig. shows one possible representation for these sets. In this representation, each set is represented as a tree.



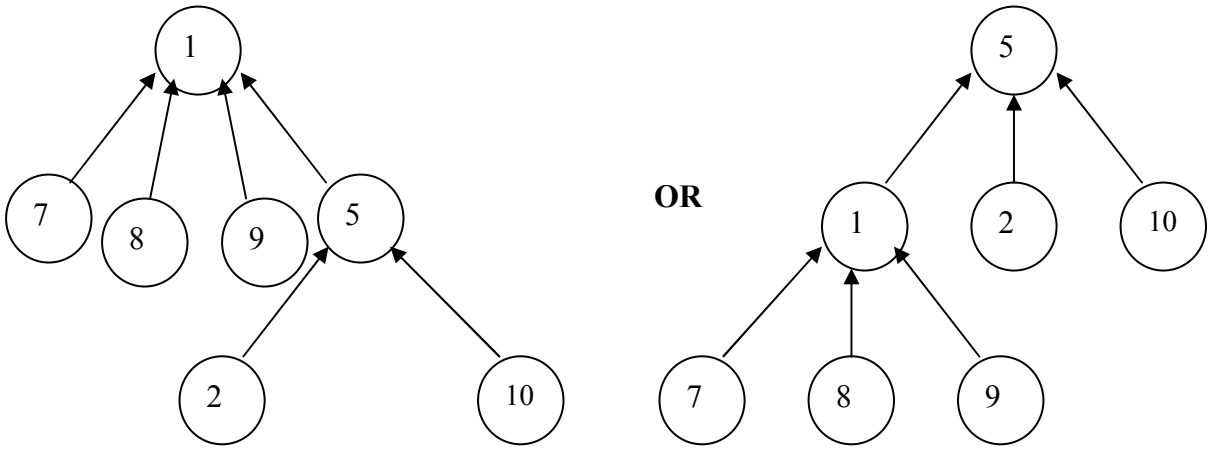
S_1 S_2 S_3

The operations perform on these sets are:

1. **Disjoint set union :** If S_i and S_j are two disjoint sets, then their union S_i U S_j = all elements x is in S_i or S_j . Thus S_1 U S_2 = {1, 7, 8, 9, 2, 5, 10}. Since we have assumed that all sets are disjoint, we can assume that following the union of S_i and S_j,the sets of S_i and S_j do not exists independently; i.e., they are replaced by S_i U S_j in the collection of sets.
2. **Find(*i*):** Given the element I, find the set containing i. Thus, 4 is in set S_3, and 9 is in set S_1.

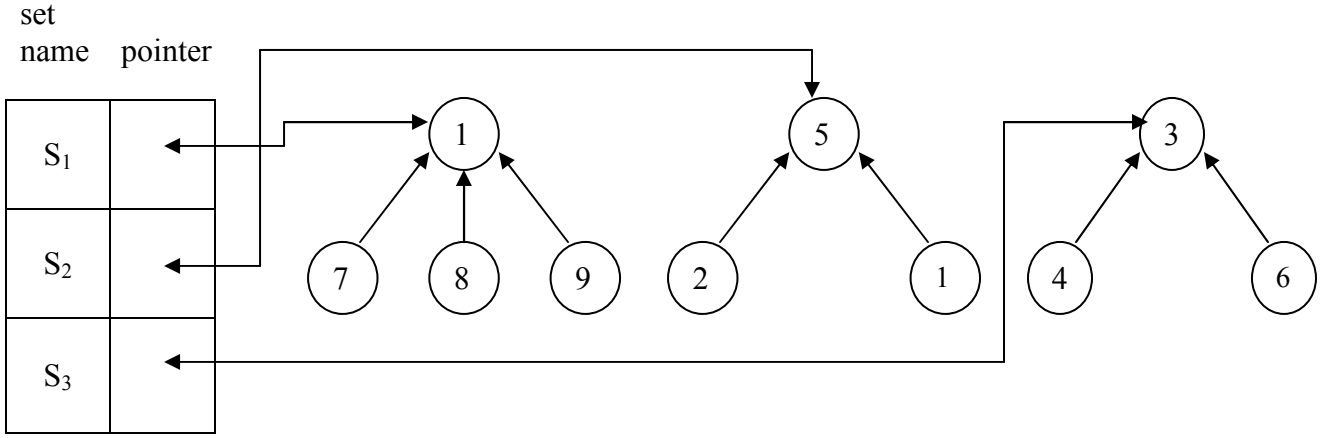
UNION AND FIND OPERATIONS

Suppose we wish to obtain union operation of S_1 and S_2. Since we have linked the nodes from children to parent, make one of the trees a sub tree of the other. S_1 U S_2 could then have one of the representations of fig.



S_1 U S_2 S_1 U S_2

To obtain the union of two sets, all that has to be set the parent field of the roots to the other root. This can be accomplished easily if, with each set name, we keep a pointer to the root of the tree representing that set. In addition, each root has a pointer to the set name, then to determine which set an element is currently in, we follow parent links to the root of its tree and use the pointer to the set name. The data representation for S_1, S_2, S_3 may then take the form as



In presenting the union and find algorithms, we ignore the set names and identify sets by the root of the trees representing them. The transition to set names is easy. If we determine element I is in a tree with root j, and j has a pointer to entry k in the set name table, then the set name is just name[k]. If we wish to unite sets S_i and S_j then we wish to unite the trees with roots FindPointer(S_i) and FindPointer(S_j). Here FindPointer is a function that takes a set name and determines the root of the tree that represents it. This is done by an examination of the [set name, pointer] table. In many applications the set name is just the element at the root. The operation of Find(i) now becomes: Determine the root of the tree containing element i. The function Union(I, j) requires two trees with roots I and j be joined. Also to simplify, assume that the set elements are the numbers 1 through n.

Since the set elements are numbered 1 through n, we represent the tree nodes using an array p[1:n], where n is the maximum number of elements. The ith element of this array represents the tree node that contains element i. This array element gives the parent pointer of the corresponding tree node. The root nodes have a parent of -1.

i	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]
p	-1	5	-1	3	-1	3	1	1	1	5

We can implement Find(i) by following the indices, starting at I until we reach anode with parent value -1. For eg; Find(6) starts at 6 and then moves to 6s parent, 3. Since p[3] is negative, we have reached the root. The operation union(I,j) is equally simple. We pass in two trees with roots I and j. Adopting the convention that the first tree becomes a sub tree of the second, the statement p[i] := j; accomplishes the union.

```
Algorithm SimpleUnion(i, j)
  {
  p[i] := j;
  }

Algorithm SimpleFind(i)
  {
  while(p[i]≥0) do i := p[i];
  return i;
  }
```

The time taken for a union is constant, the n-1 unions can be processed in time O(n). Each find requires following a sequence of parent pointers from the element to be found to the root. Since the time required to process a find for an element at level i of a tree is O(i), the total time needed to process the n finds is

$$O\left(\sum_{i=1}^{n} i\right) = O(n^2)$$

PRIORITY QUEUES

DEFINITION:

A priority queue is a collection of zero or more elements. Each element has a priority or value.

- → Unlike the queues, which are FIFO structures, the order of deleting from a priority queue is determined by the element priority.
- → Elements are removed/deleted either in increasing or decreasing order of priority rather than in the order in which they arrived in the queue.

There are two types of priority queues:

- Min priority queue / Ascending priority queue
- Max priority queue / Descending priority queue

Min priority queue: Collection of elements in which the items can be inserted arbitrarily, but only smallest element can be removed.

Max priority queue: Collection of elements in which insertion of items can be in any order but only largest element can be removed.

- → In priority queue, the elements are arranged in any order and out of which only the smallest or largest element allowed to delete each time.
- → The implementation of priority queue can be done using arrays or linked list. The data structure **heap** is used to implement the priority queue effectively.

APPLICATIONS:

1. The typical example of priority queue is scheduling the jobs in operating system. Typically OS allocates priority to jobs. The jobs are placed in the queue and position of the job in priority queue determines their priority. In OS there are 3 jobs- real time jobs, foreground jobs and background jobs. The OS always schedules the real time jobs first. If there is no real time jobs pending then it schedules foreground jobs. Lastly if no real time and foreground jobs are pending then OS schedules the background jobs.
2. In network communication, the manage limited bandwidth for transmission the priority queue is used.
3. In simulation modeling to manage the discrete events the priority queue is used.

ABSTRACT DATA TYPE(ADT):

Various operations that can be performed on priority queue are-

1. Find an element
2. Insert a new element
3. Remove or delete an element

The abstract data type specification for a max priority queue is given below. The specification for a min priority queue is the same except that top and pop, find and remove the element with minimum priority.

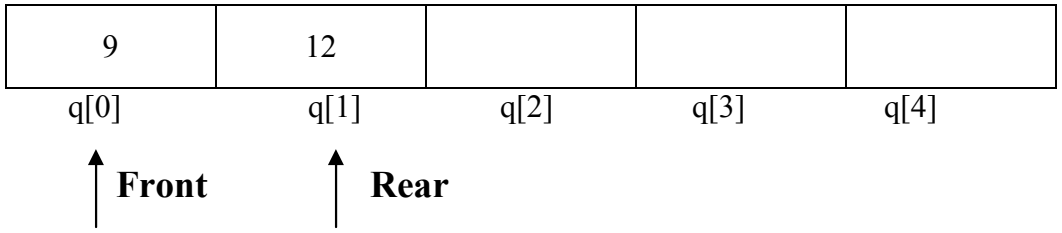
```
Abstract data type maxPriorityQueue
{
Instances
        Finite collection of elements, each has a priority
Operations
        create();
        insert();
        delet();
        display();
}
```

IMPLEMENTATION OF PRIORITY QUEUE USING A LINEAR LIST

There are two major operations that can be performed on priority queue and those are insert and delete. These operations can be explained by following examples:

INSERTION OPERATION:

While implementing the priority queue we will apply a simple logic. That is while inserting the element we will insert the element in the array at the proper position.
For example:



And now if an element 8 is to be inserted in the queue then it will be at 0^{th} location as:

8	9	12		

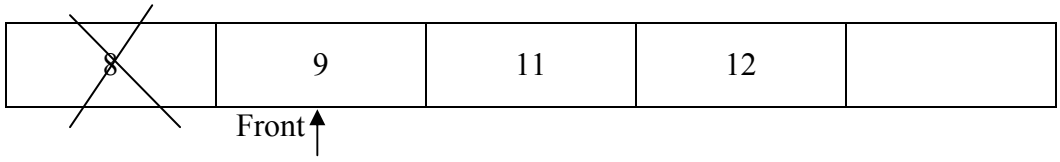
If the next element comes as 11 then the queue will be-

8	9	11	12	

```
void insert(int q[], int rear, int front)
{
int item, j;
cin>>item;
if(rear = max-1) cout<<"Queue overflow";
else
if(front == -1)
front ++;
j=rear;
while(j>0 && item<q[j])
{
q[j+1] = q[j];
j--;
}
q[j+1] = item;
rear = rear+1;
return rear;
}
```

DELETION OPERATION:

In deletion operation we are simple removing the elements from at the front.
eg:



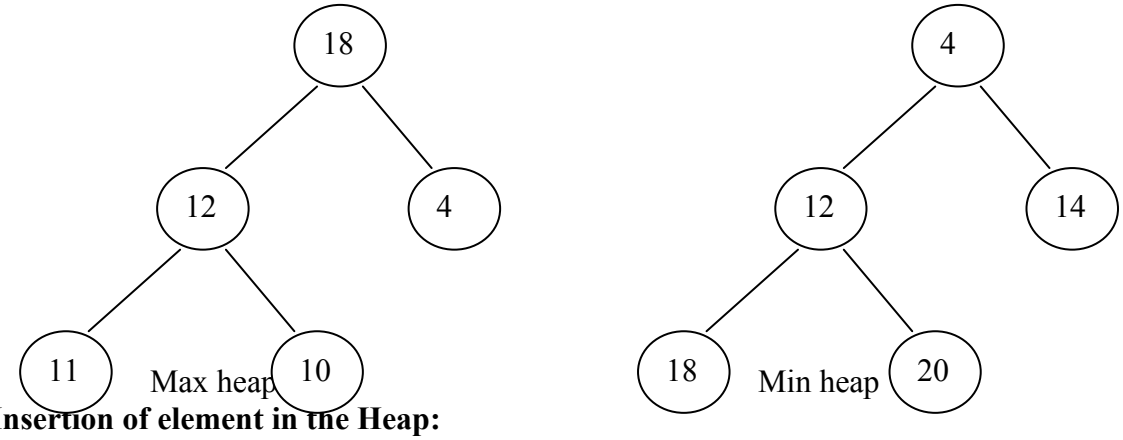

```
void del(int q[],int front)
{
int item;
if(front == -1 || front>rear) cout<<"Queue underflow";
item = q[front];
cout<<item<<"is deleted";
front ++;
retun;
}
```

HEAPS

Heap is a **complete binary tree** or an **almost complete binary tree** in which every **parent node** be either greater or less than its child nodes.

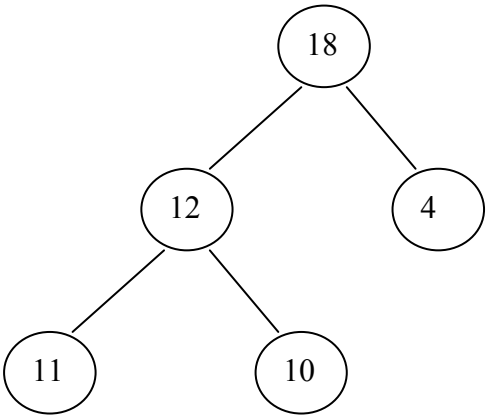
Heap is a tree data structure denoted by either a max heap or a min heap.

A max heap is a tree in which value of each node is greater than or equal to value of its children nodes.
A min heap is a tree in which value of each node is less than or equal to value of its children nodes.

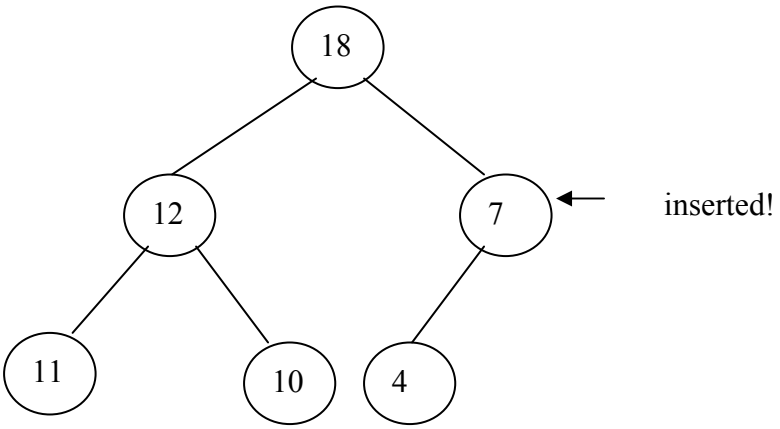


Insertion of element in the Heap:

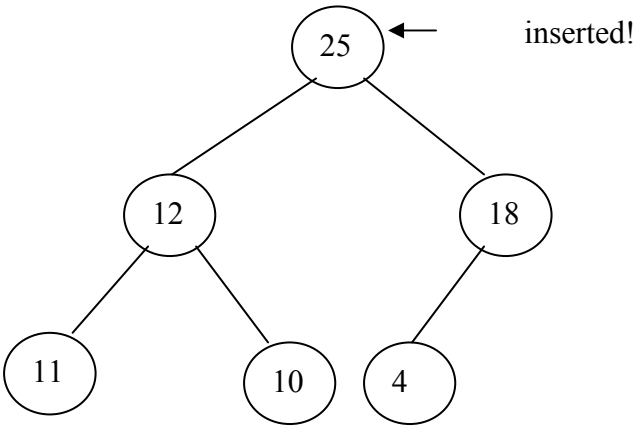
Consider a max heap as given below:



Now if we want to insert 7. We cannot insert 7 as left child of 4. This is because the max heap has a property that value of any node is always greater than the parent nodes. Hence 7 will bubble up 4 will be left child of 7.
Note: When a new node is to be inserted in complete binary tree we start from bottom and from left child on the current level. The heap is always a complete binary tree.



If we want to insert node 25, then as 25 is greatest element it should be the root. Hence 25 will bubble up and 18 will move down.



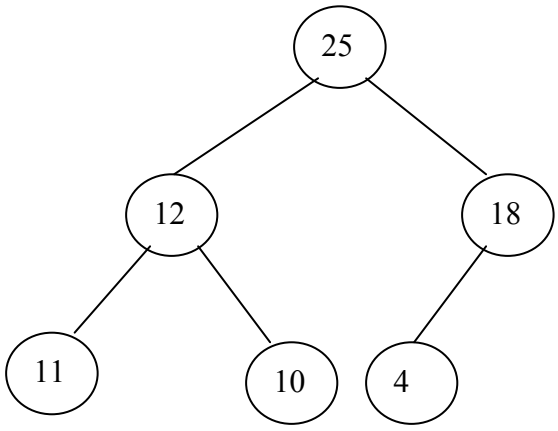
The insertion strategy just outlined makes a single bubbling pass from a leaf toward the root. At each level we do Θ(1) work, so we should be able to implement the strategy to have complexity O(height) = O(log n).

```
void Heap::insert(int item)
{
int temp;         //temp node starts at leaf and moves up.
temp=++size;
while(temp!=1 && heap[temp/2]<item)        //moving element down
{
H[temp] = H[temp/2];
temp=temp/2;  //finding the parent
}
H[temp]=item;
}
```

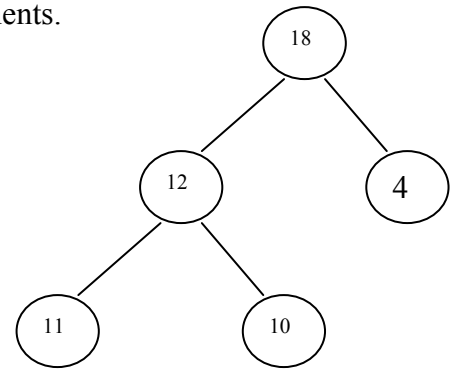
Deletion of element from the heap:

For deletion operation always the maximum element is deleted from heap. In Max heap the maximum element is always present at root. And if root element is deleted then we need to reheapify the tree.

Consider a Max heap

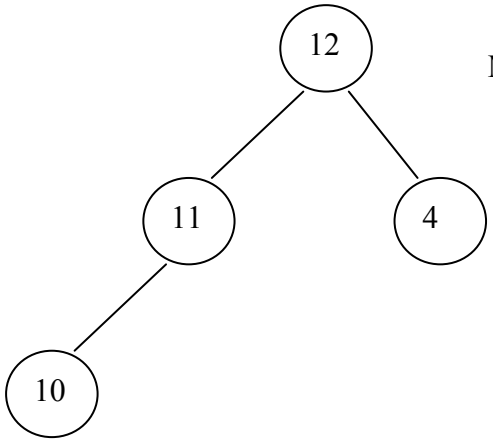


Delete root element:25, Now we cannot put either 12 or 18 as root node and that should be greater than all its children elements.



Now we cannot put 4 at the root as it will not satisfy the heap property. Hence we will bubble up 18 and place 18 at root, and 4 at position of 18.

If 18 gets deleted then 12 becomes root and 11 becomes parent node of 10.



Make tree a complete binary tree.

Thus deletion operation can be performed. The time complexity of deletion operation is O(log n).

1. Remove the maximum element which is present at the root. Then a hole is created at the root.
2. Now reheapify the tree. Start moving from root to children nodes. If any maximum element is found then place it at root. Ensure that the tree is satisfying the heap property or not.
3. Repeat the step 1 and 2 if any more elements are to be deleted.

```
void heap::delet(int item)
{
int item, temp;
if(size==0)
cout<<"Heap is empty\n";
else
{
item=H[size--];
temp=1;
child=2;
while(child<=size)
{
if(child<size && H[child]<H[child+1])
child++;
if(item>=H[child])
break;
H[temp]=H[child];
temp=child;
child=child*2;}
```

H[temp]=item;}

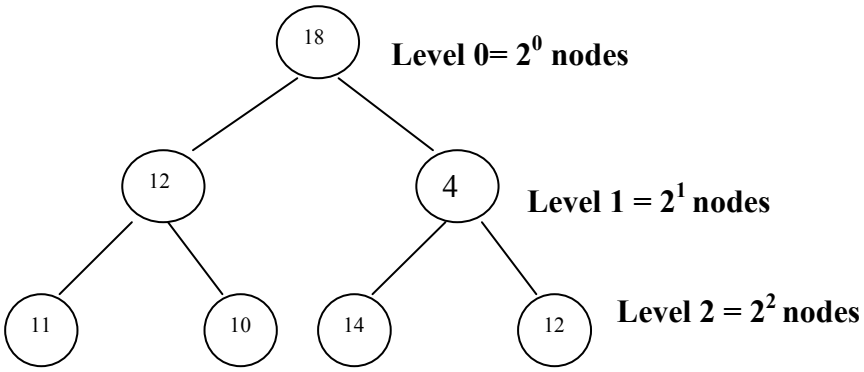
Parent being greater or lesser in heap is called parental propery. Thus heap has two important properties:

Heap → It should be a complete or an almost complete binary tree

Heap → It should satisfy parental property

COMPLETE BINARY TREE:

The complete binary tree is a binary tree in which all leaves are at the same depth or total number of nodes at each level i are 2^i.

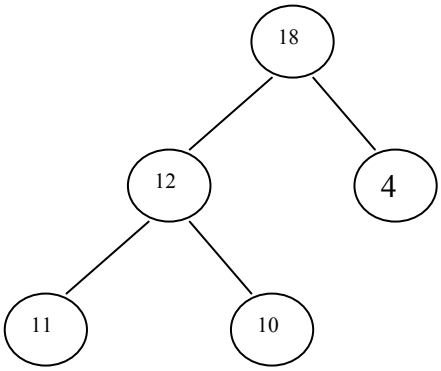


ALMOST COMPLETE BINARY TREE:

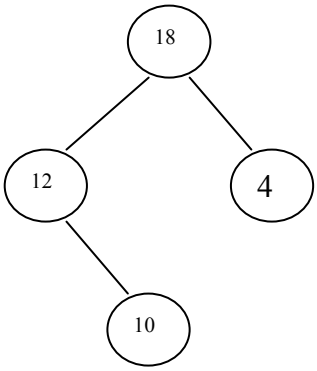
The almost complete binary tree is a tree in which –

i Each node has a left child whenever it has a right child. That is there is always a left child, but for a left child there may not be a right child.

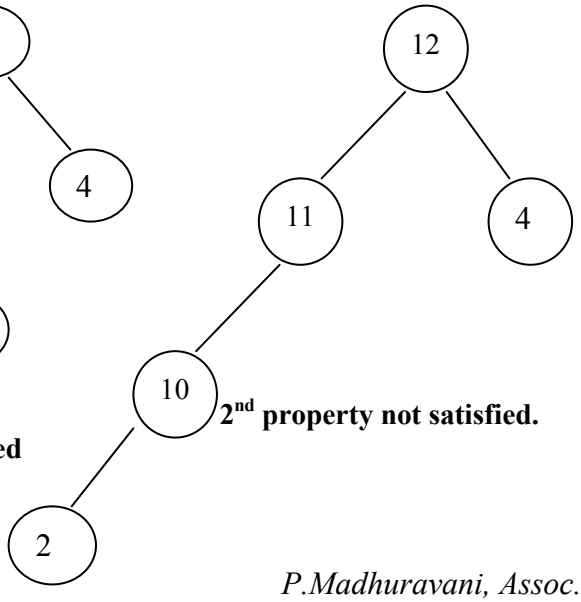
ii The leaf in a tree must be present at height h or h-1. That means all the leaves are on two adjacent levels.



Almost complete binary tree



1st property not satisfied



2nd property not satisfied.

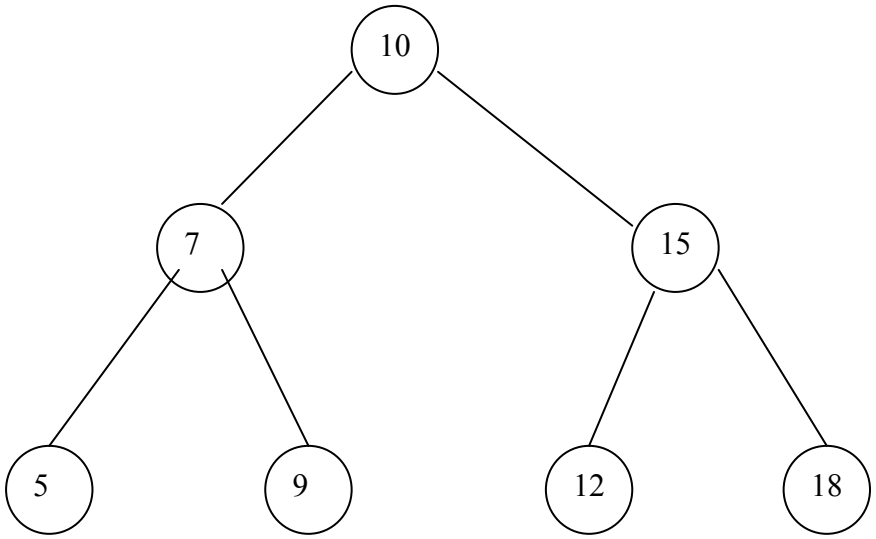
Applications Of Heap:

1. Heap is used in sorting algorithms. One such algorithm using heap is known as heap sort.
2. In priority queue implementation the heap is used.

SEARCH TREES

BINARY SEARCH TREE

In the simple binary tree the nodes are arranged in any fashion. Depending on user's desire the new nodes can be attached as a left or right child of any desired node. In such a case finding for any node is a long cut procedure, because in that case we have to search the entire tree. And thus the searching time complexity will get increased unnecessarily. So to make the searching algorithm faster in a binary tree we will go for building the binary search tree. The binary search tree is based on the binary search algorithm. While creating the binary search tree the data is systematically arranged. That means values at left sub-tree < root node value < right sub-tree values.



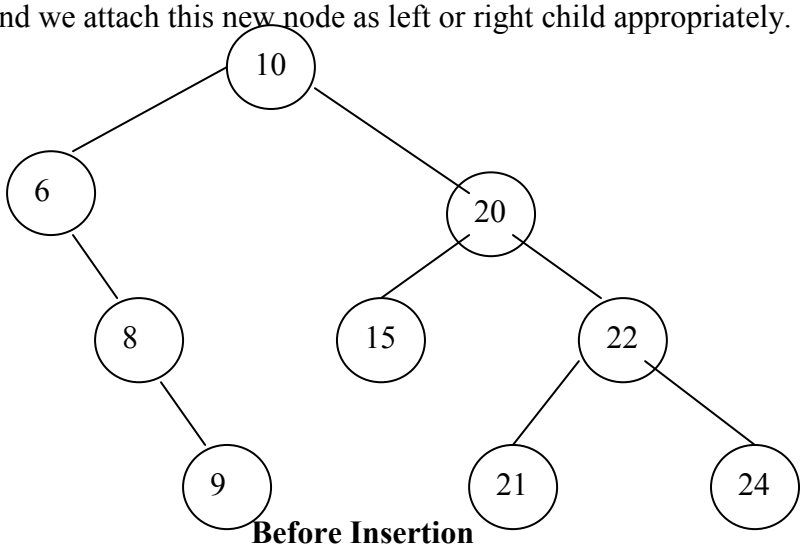
Operations On Binary Search Tree:

The basic operations which can be performed on binary search tree are.

1. Insertion of a node in binary search tree.
2. Deletion of a node from binary search tree.
3. Searching for a particular node in binary search tree.

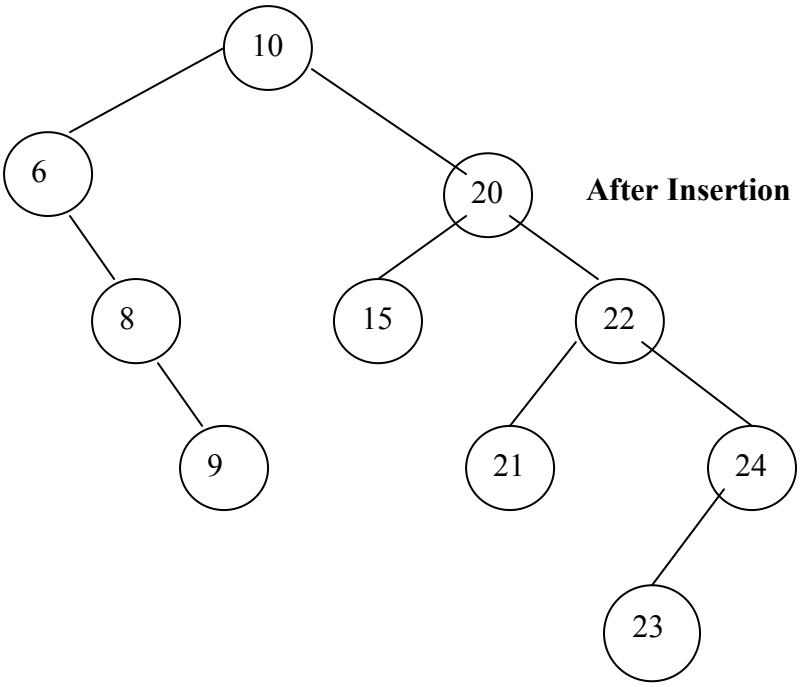
Insertion of a node in binary search tree.

While inserting any node in binary search tree, look for its appropriate position in the binary search tree. We start comparing this new node with each node of the tree. If the value of the node which is to be inserted is greater than the value of the current node we move on to the right sub-branch otherwise we move on to the left sub-branch. As soon as the appropriate position is found we attach this new node as left or right child appropriately.



Before Insertion

In the above fig, if we wan to insert 23. Then we will start comparing 23 with value of root node i.e. 10. As 23 is greater than 10, we will move on right sub-tree. Now we will compare 23 with 20 and move right, compare 23 with 22 and move right. Now compare 23 with 24 but it is less than 24. We will move on left branch of 24. But as there is node as left child of 24, we can attach 23 as left child of 24.



After Insertion

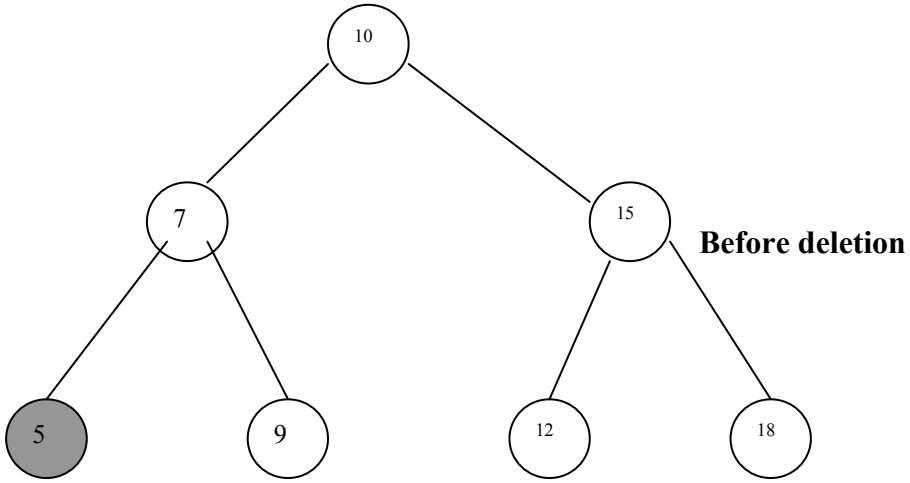
Deletion of a node from binary search tree.

For deletion of any node from binary search tree there are three cases which are possible.

i. Deletion of leaf node.
ii. Deletion of a node having one child.
iii. Deletion of a node having two children.

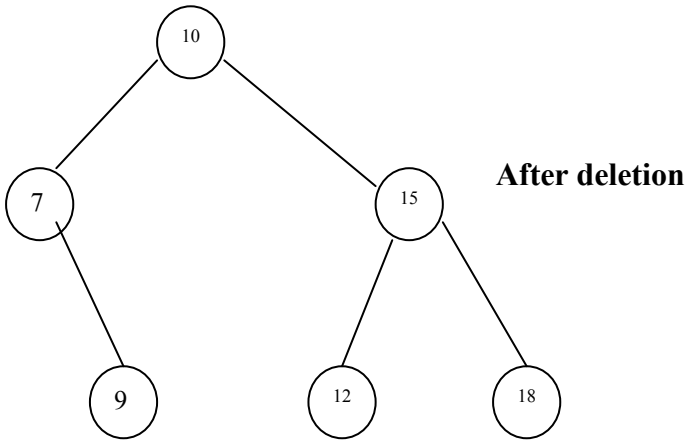
Deletion of leaf node.

This is the simplest deletion, in which we set the left or right pointer of parent node as NULL.



Before deletion

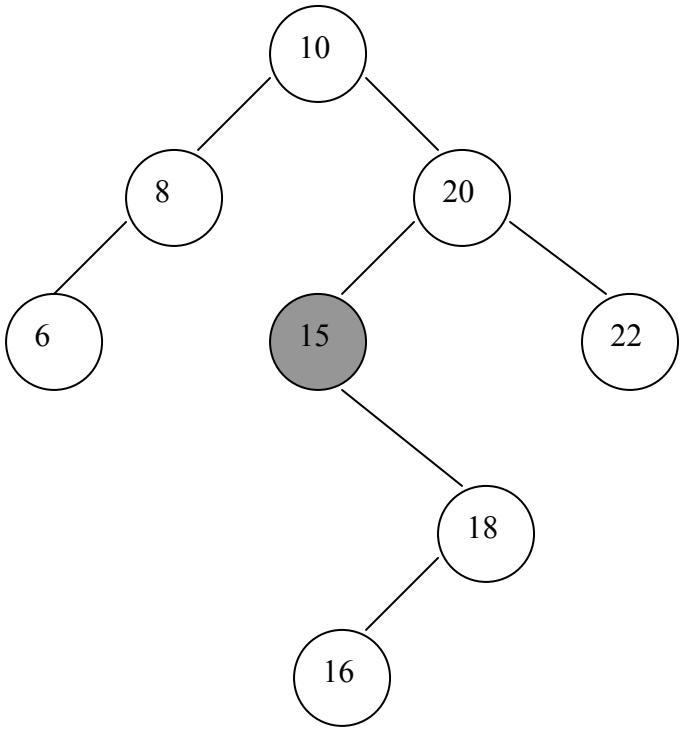
From the above fig, we want to delete the node having value 5 then we will set left pointer of its parent node as NULL. That is left pointer of node having value 7 is set to NULL.



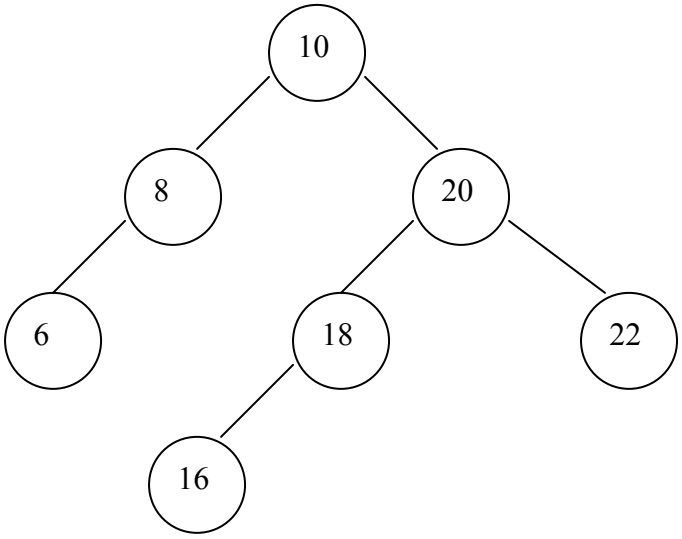
After deletion

Deletion of a node having one child.

To explain this kind of deletion, consider a tree as given below.

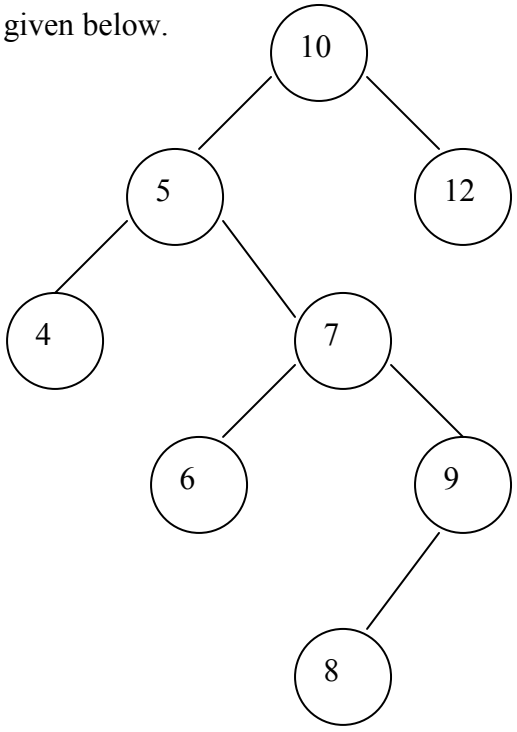


If we want to delete the node 15, then we will simply copy node 18 at place of 16 and then set the node free.

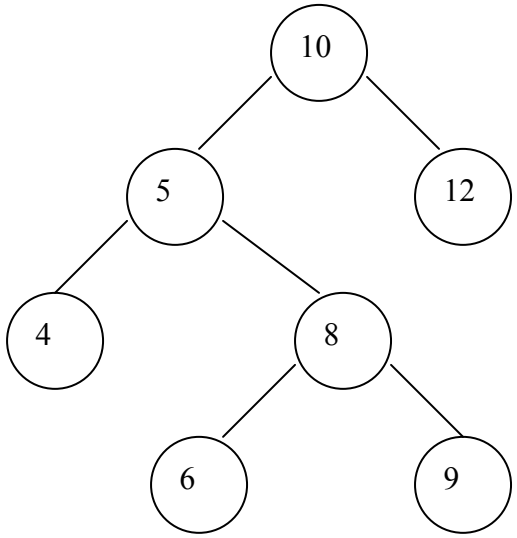


Deletion of a node having two children.

Consider a tree as given below.

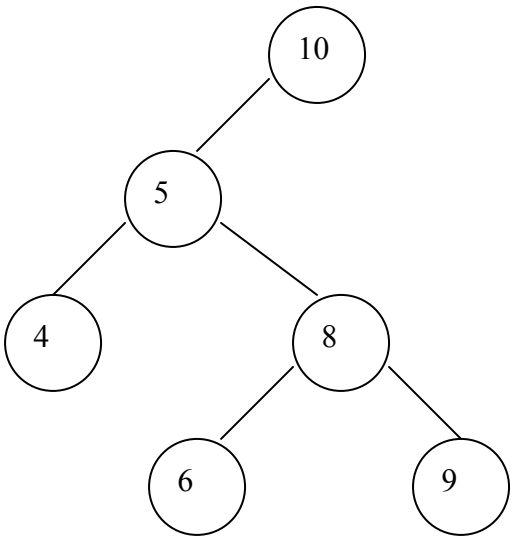


Let us consider that we want to delete node having value 7. We will then find out the inorder successor of node 7. We will then find out the inorder successor of node 7. The inorder successor will be simply copied at location of node 7.
That means copy 8 at the position where value of node is 7. Set left pointer of 9 as NULL. This completes the deletion procedure.



Searching for a node in binary search tree.

In searching, the node which we want to search is called a key node. The key node will be compared with each node starting from root node if value of key node is greater than current node then we search for it on right sub branch otherwise on left sub branch. If we reach to leaf node and still we do not get the value of key node then we declare "node is not present in the tree".



In the above tree, if we want to search for value 9. Then we will compare 9 with root node 10. As 9 is less than 10 we will search on left sub branch. Now compare 9 with 5, but 9 is greater than 5. So we will move on right sub tree. Now compare 9 with 8 but 9 is greater than 8 we will move on right sub branch. As the node we will get holds the value 9. Thus the desired node can be searched.

BST Operations:

Structure of a node:

```
typedef struct bst
{
T data;
struct bst *left, *right;
} node;
node *root, *New, *temp, *parent;

public:
bintree()
{
root = NULL;
}
```

Creation:

```
void create(node *New)
{
        New=new node;
        New.left=NULL;
        New.right=NULL;
        System.out.println("Enter the element\n";
                if(root==NULL)
                root=New;
        else
                insert(root,New);
}
```

Insertion:

```
void insert(node *root,node *New)
{
        if(New.data<root.data)
        {
                if(root.left==NULL)
                        root.left=New;
                else
                        insert(root.left,New);
        }
        if(New.data>root.data)
        {
                if(root.right==NULL)
                        root.right=New;
                else
                        insert(root.right,New);
        }
}
```

Deletion:

```
void delet()
{
        int key;
        System.out.println("enter the element to delete\n");
        if(key==root.data)
        {
                bintree();
        }
        else
                del(root,key);
```

```
}

/*-------------------------------------------------------------------------
This function is for deleting a node from binary search tree. There exists
three possible cases for deletion of a node
-------------------------------------------------------------------------*/

void del(node *root,int key)
{
        node *temp_succ;
        if(root==NULL)
                System.out.println("Tree is not created\n";
        else
        {
                temp=root;
                search(temp,key,parent);
                if(temp.left!=NULL&&temp.right!=NULL)
                {
                        parent=temp;
                        temp_succ=temp.right;
                        while(temp_succ.left!=NULL)
                        {
                                parent=temp_succ;
                                temp_succ=temp_succ.left;
                        }
                        temp.data=temp_succ.data;
                        temp.right=NULL;
                        System.out.println("Now deleted it\n";
                        return;
                }

                /*Deleting a node having only one child*/
                /*The node to be deleted has left child*/

                if(temp.left!=NULL&&temp.right==NULL)
                {
                        if(parent.left==temp)
                                parent.left=temp.left;
                        else
                                parent.right=temp.left;
                        temp=NULL;

                        System.out.println("Now deleted it\n";
                        return;
                }
```

```
                /*The node to be deleted has right child*/

                if(temp.left==NULL && temp.right!=NULL)
                {
                        if(parent.left==temp)
                                parent.left=temp.right;
                        else
                                parent.right=temp.right;
                        temp=NULL;

                        System.out.println("Now deleted it\n");
                        return;
                }

                /*Deleting a node which is having no child*/

                if(temp.left==NULL && temp.right==NULL)
                {
                        if(parent.left==temp)
                                parent.left=NULL;
                        else
                                parent.right=NULL;
                        System.out.println("Now deleted it\n";
                        return;
                }
        }
}
```

Traversal:

```
void display()
{
        if(root==NULL)
                System.out.println("Tree is not created\n");
        else
        {
                System.out.println("The tree is\n");
                inorder(root);
        }
}

void inorder(node *temp)
{
        if(temp!=NULL)
        {
                inorder(temp.left);
```

```
                System.out.println(" "+temp.data;
                inorder(temp.right);
        }
}
```

THREADED BINARY TREE

During binary tree creation, for leaf nodes there is no further sub trees, we just set the left and right fields of leaf nodes as NULL. Since NULL value is put in the left and right fields of the node it is just wastage of the memory. So to avoid NULL values in the node we just set the threads which are actually the links to the predecessor and successor nodes.

There are three types of threading is possible:

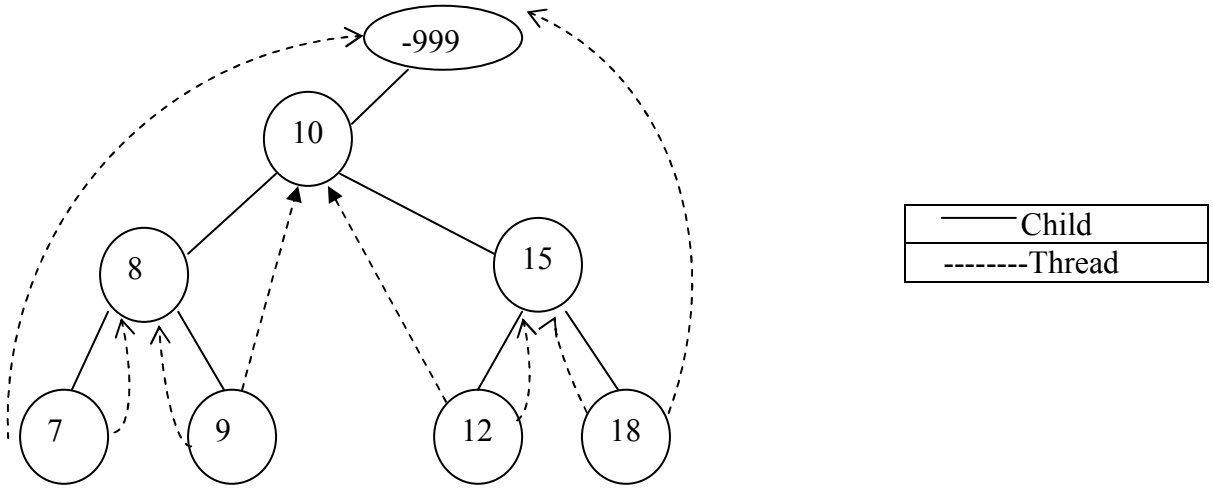
1. Inorder threading
2. Preorder threading
3. Postorder threading.

Structure of the node is:

```
class thread
{
int data;
int lth, rth;
}*left,*right;
```

Inorder threading:

The basic idea in inorder threading is that the left thread should point to the predecessor and the right thread points to the successor. Here we are assuming the head node as the starting node and the root node of the tree is attached to left of head node.
Eg:



AVL TREES

Adelsion Velski and Lendis in 1962 introduced binary tree structure that is balanced with respect to height of sub trees. The tree can be made balanced and because of this retrieval of any node can be done in O(log n) times, where n is total number of nodes. From the name of these scientists the tree is called AVL tree.

Definition:

An empty tree is height balanced if T is a non empty binary tree with T_L and T_R as its left and right sub trees. The T is height balanced if and only if

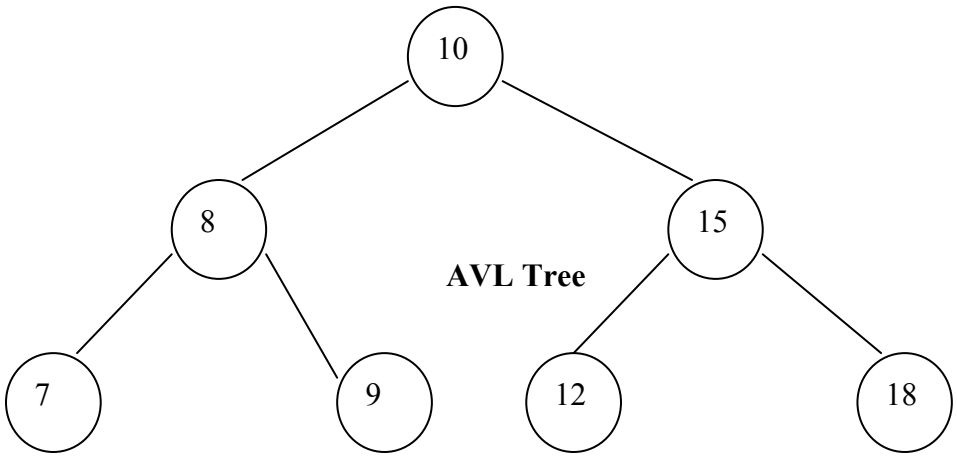
i. T_L and T_R are height balanced.
ii. h_L-h_R <= 1 where h_L and h_R are heights of T_L and T_R.

The idea of balancing a tree is obtained by calculating the balance factor of a tree.

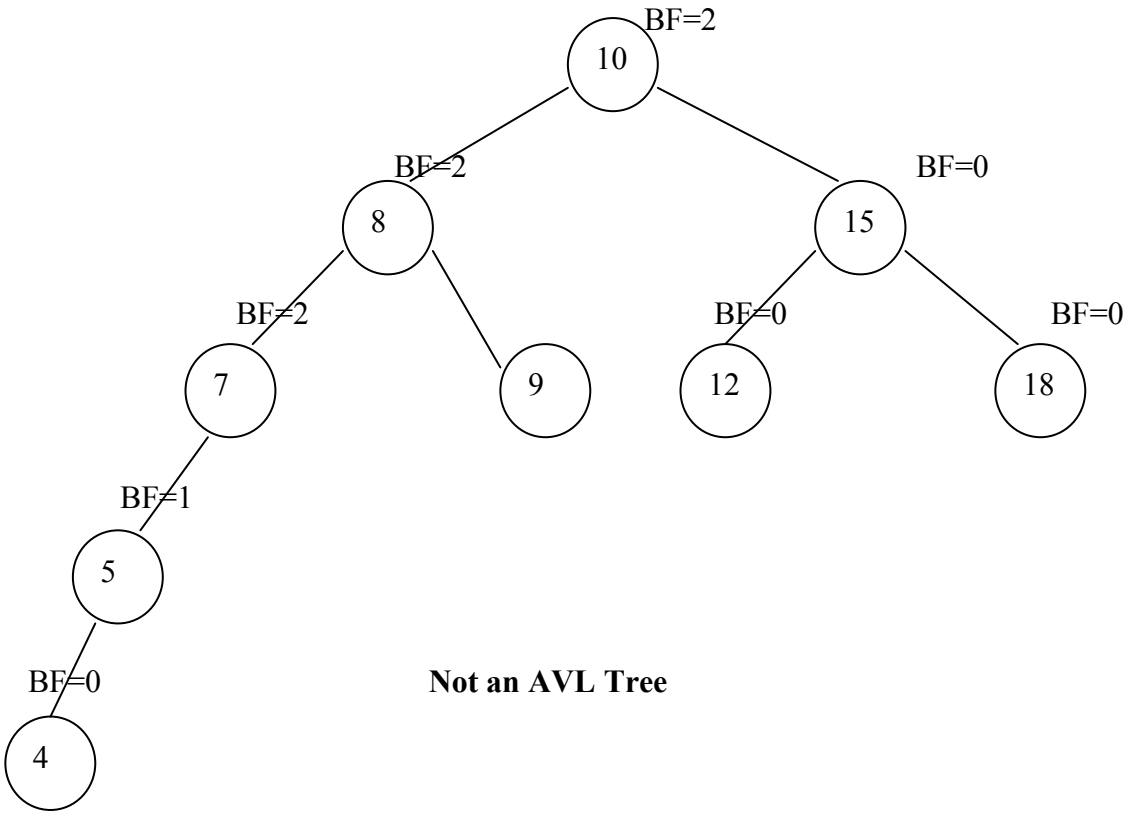
Definition of Balance Factor:

The balance factor BF(T) of a node in binary tree is defined to be h_L-h_R where h_L and h_R are heights of left and right sub trees of T.

For any node in AVL tree the balance factor i.e. BF(T) is -1, 0 or +1.



AVL Tree



Not an AVL Tree

Height of AVL Tree:

Theorem: The height of AVL tree with n elements (nodes) is O(log n).

Proof: Let an AVL tree with n nodes in it. Nh be the minimum number of nodes in an AVL tree of height h.

In worst case, one sub tree may have height h-1 and other sub tree may have height h-2. And both these sub trees are AVL trees. Since for every node in AVL tree the height of left and right sub trees differ by at most 1.

Hence

$N_h = N_{h-1}+N_{h-2}+1$

Where N_h denotes the minimum number of nodes in an AVL tree of height h.

$N_0=0$ $N_1=2$

We can also write it as

$N > N_h = N_{h-1}+N_{h-2}+1$

$> 2N_{h-2}$

$> 4N_{h-4}$

.
.
.

$> 2^i N_{h-2}{}^i$

If value of h is even, let i = h/2-1

Then equation becomes

$N > 2^{h/2-1}N_2$

$= N > 2^{(h-1)/2}x4 \quad (N_2 = 4)$

$= O(\log N)$

If value of h is odd, let I = (h-1)/2 then equation becomes

$N > 2^{(h-1)/2} N_1$

$N > 2^{(h-1)/2} x\ 1 \quad (N_1 = 1)$

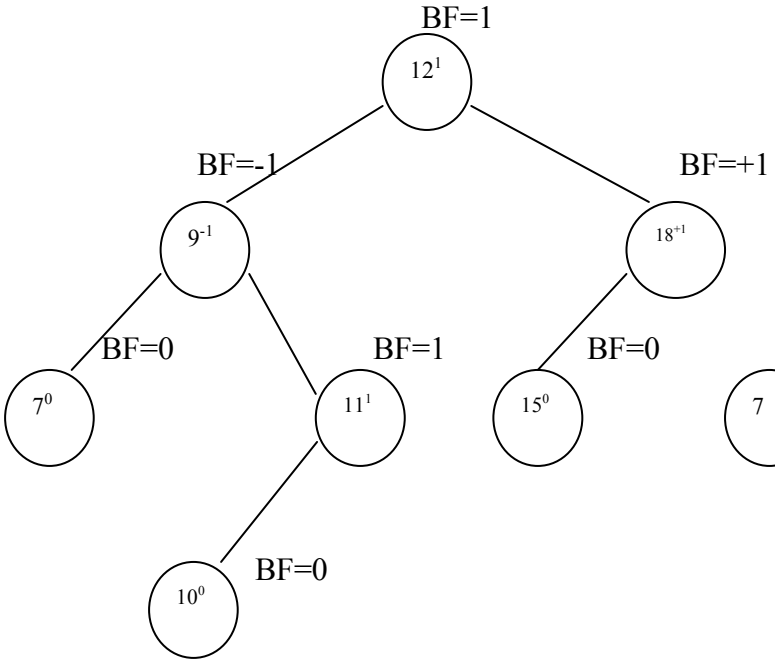
$H = O(\log N)$

This proves that height of AVL tree is always O(log N). Hence search, insertion and deletion can be carried out in logarithmic time.

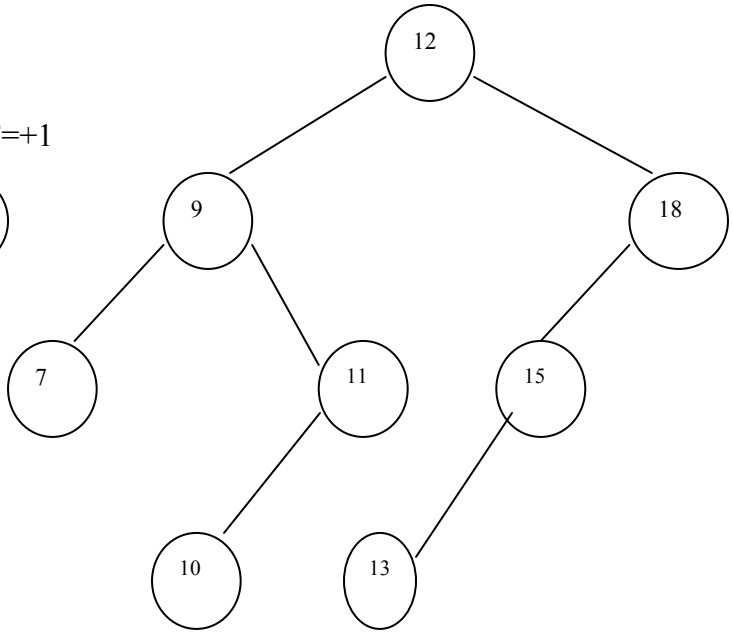
Representation of AVL Tree

- The AVL tree follows the property of binary search tree. In fact AVL trees are basically binary search trees with balance factors as -1, 0, or +1.
- After insertion of any node in an AVL tree if the balance factor of any node becomes other than -1, 0, or +1 then it is said that AVL property is violated. Then we have to restore the destroyed balance condition. The balance factor is denoted at right top corner inside the node.

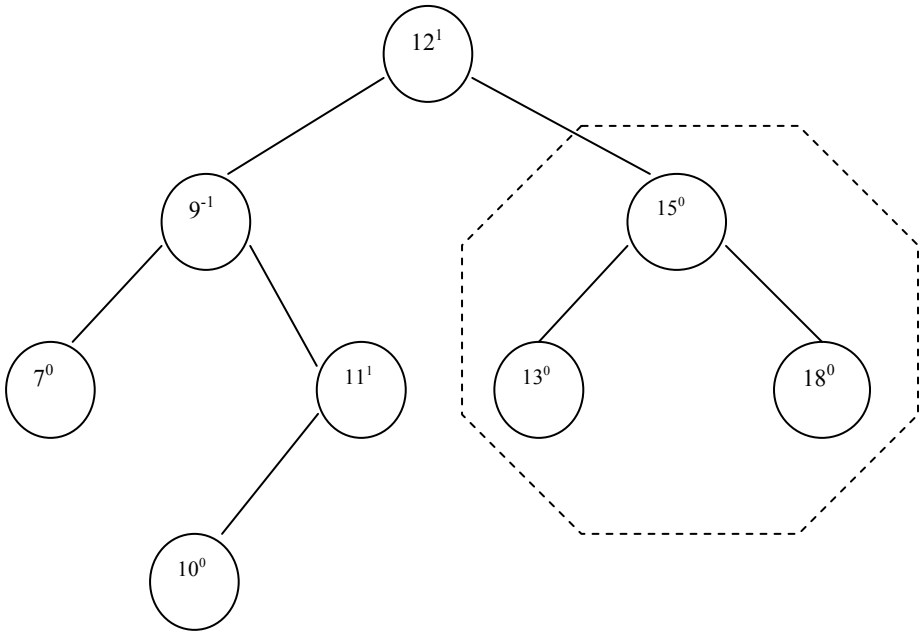
For example:



Original AVL Tree



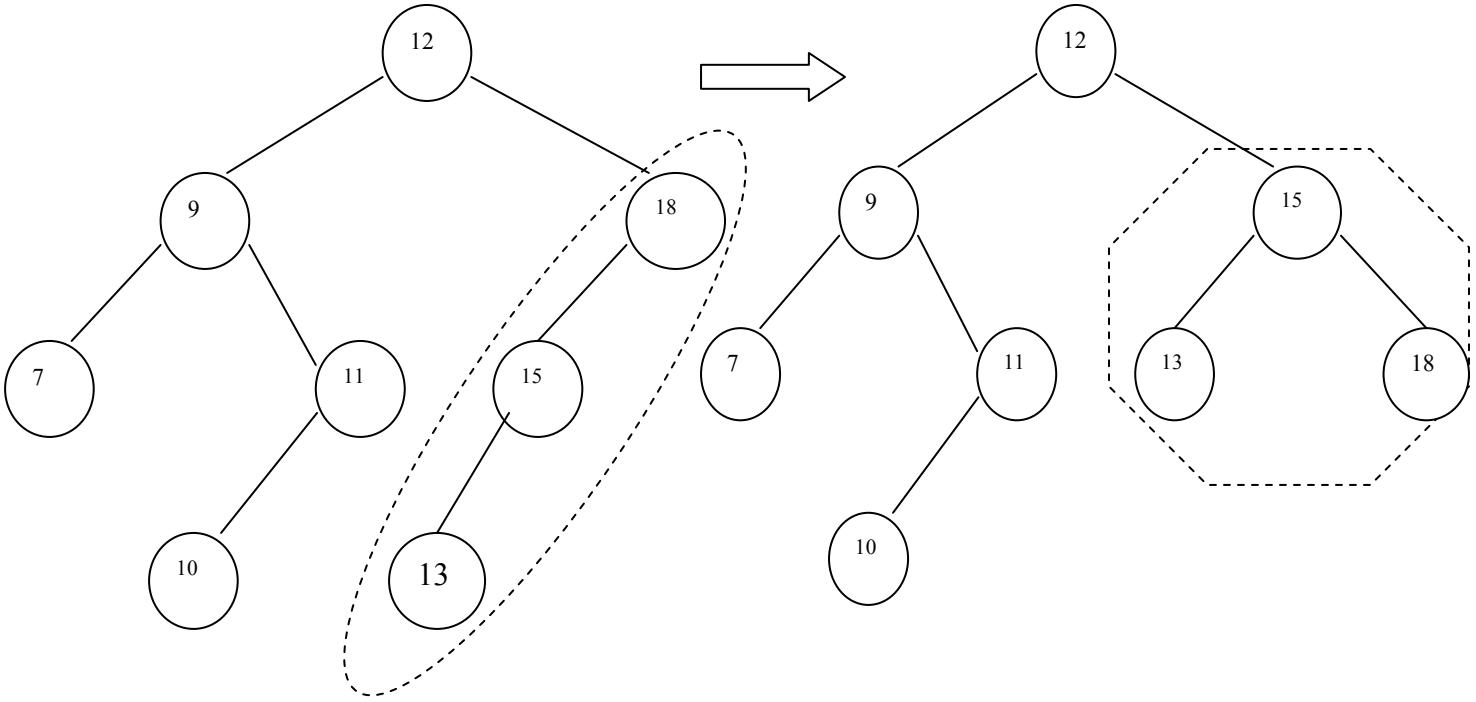
Insert 13 Property violated



Restoring AVL Property

- After insertion of a new node if balance condition gets destroyed, then the nodes on that path(new node insertion point to root) needs to be readjusted. That means only the affected sub tree is to be rebalanced.
- The rebalancing should be such that entire tree should satisfy AVL property.

In above given example-



Nodes 18, 15, 13 are to be adjusted

By adjusting 15 the entire Tree satisfies AVL property

INSERTION

There are four different cases when rebalancing is required after insertion of new node.

1. An insertion of new node into left sub tree of left child. (LL).
2. An insertion of new node into right sub tree of left child. (LR).
3. An insertion of new node into left sub tree of right child. (RL).
4. An insertion of new node into right sub tree of right child.(RR).

There is a symmetry between case 1 and 4. Similarly symmetry exists between case 2 and 3.

Some modifications done on AVL tree in order to rebalance it is called rotations of AVL tree.

There are two types of rotations:

Single rotation
- Left-Left(LL rotation)
- Right-Right(RR rotation)

Double rotation
- Left-Right(LR rotation)
- Right-Left(RL rotation)

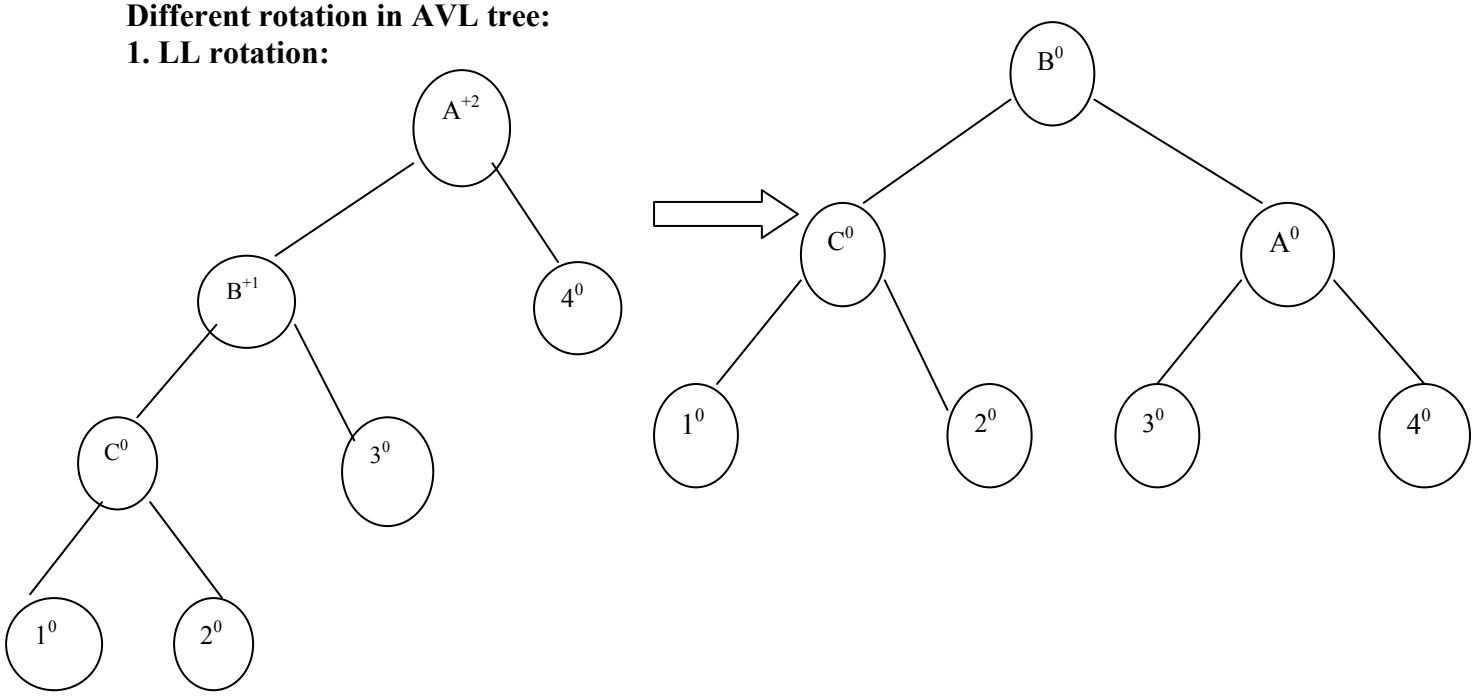
Insertion Algorithm:

1. Insert a new node as new leaf just as an ordinary binary search tree.
2. Now trace the path from insertion point(new node inserted as leaf) towards root. For each node 'n' encountered, check if heights of left (n) and right (n) differ by at most 1.
   a) If yes, move towards parent (n).
   b) Otherwise restructure by doing either a single rotation or a double rotation.

Thus once we perform a rotation at node 'n' we do not require to perform any rotation at any ancestor on 'n'.

Different rotation in AVL tree:

1. LL rotation:

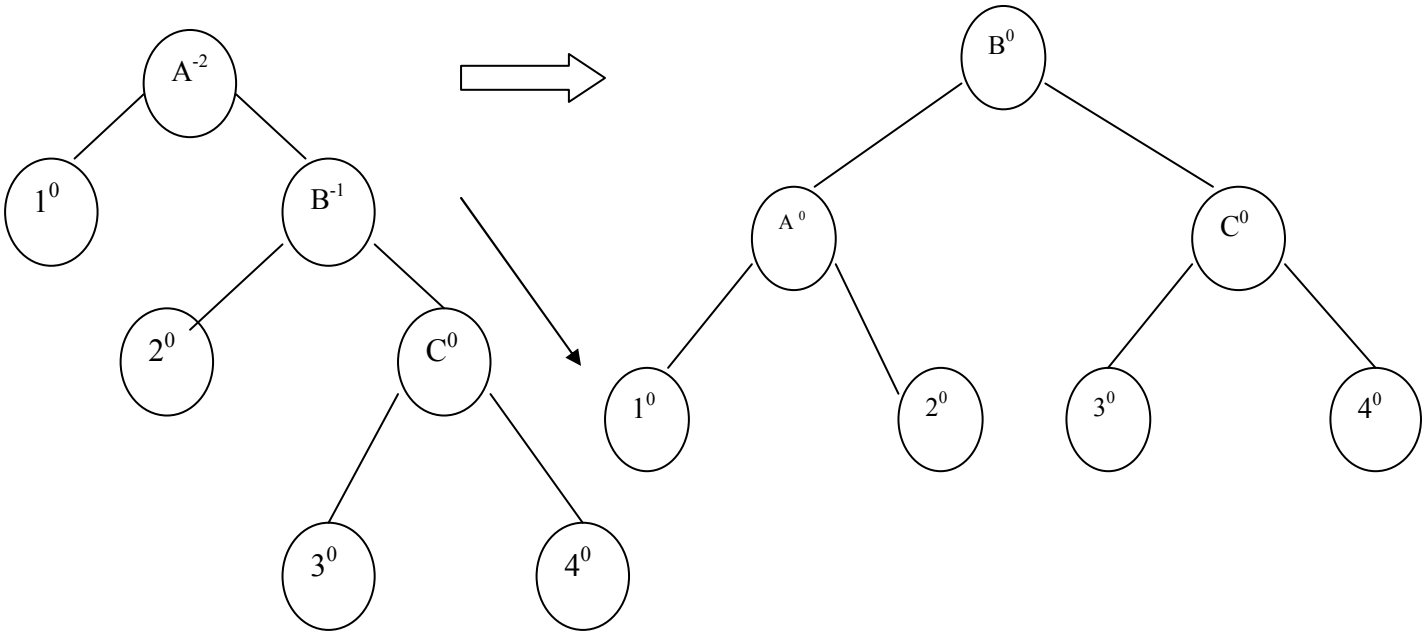


When node '1' gets inserted as a left child of node 'C' then AVL property gets destroyed i.e. node A has balance factor +2.

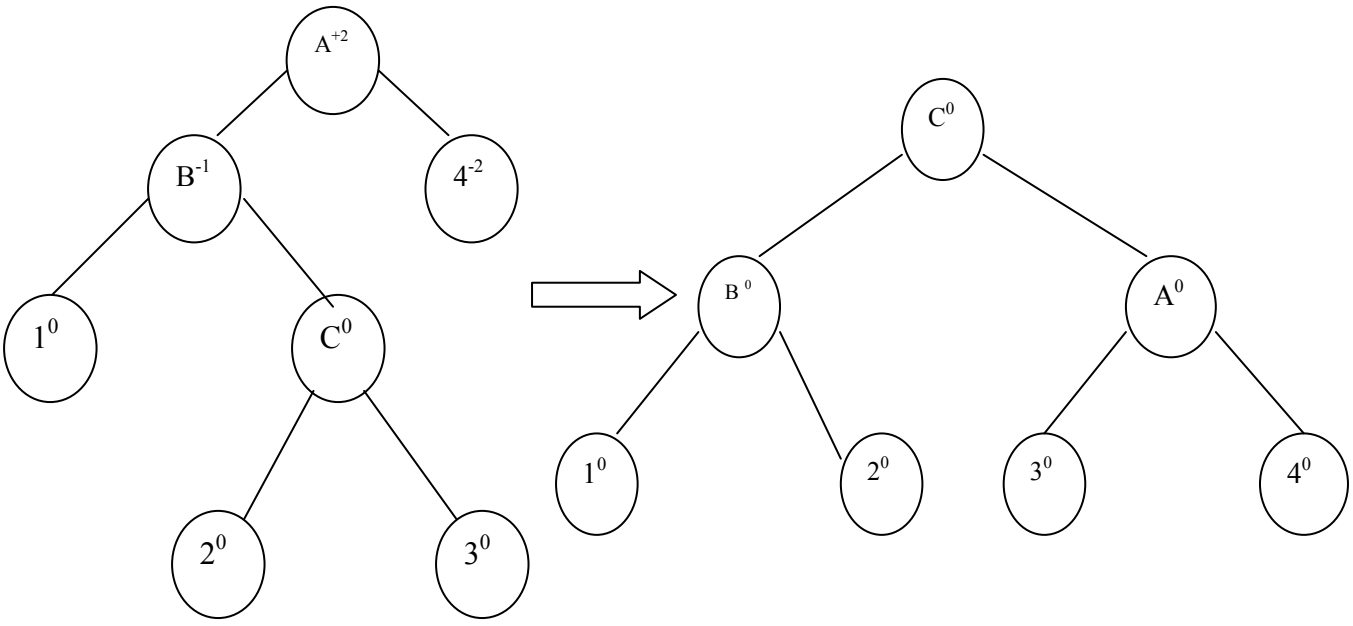
The LL rotation has to be applied to rebalance the nodes.

2. RR rotation:

When node '4' gets attached as right child of node 'C' then node 'A' gets unbalanced. The rotation which needs to be applied is RR rotation as shown in fig.

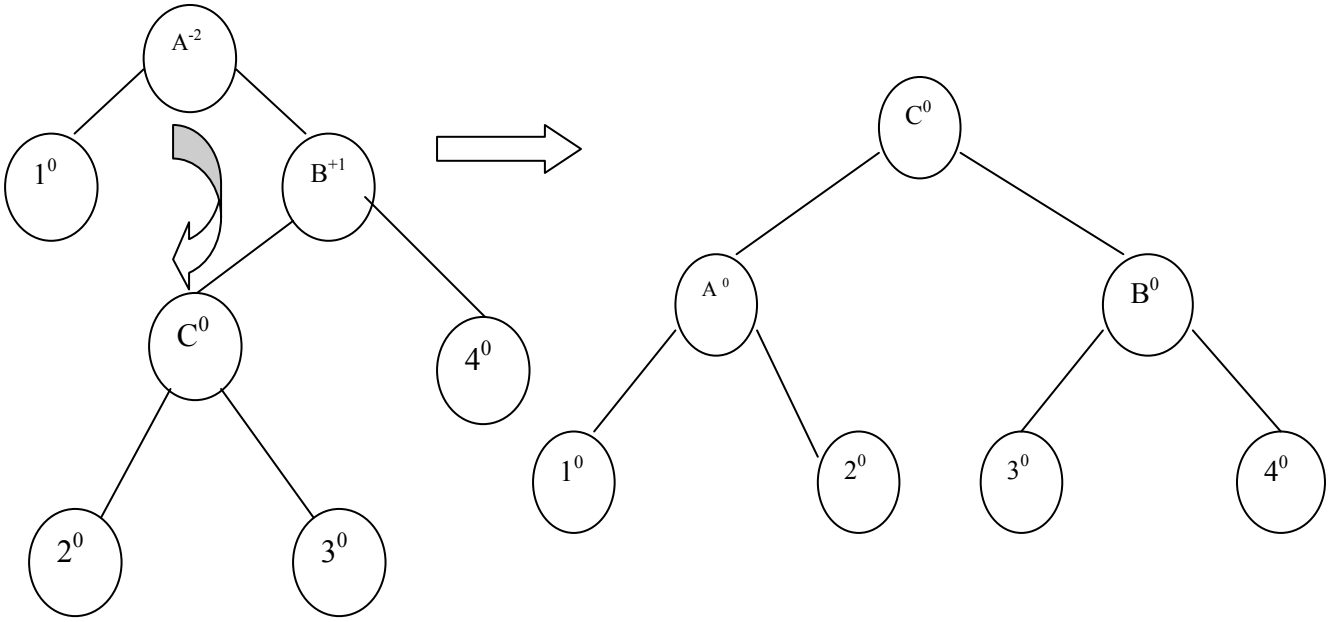


3. LR rotation:



When node '3' is attached as a right child of node 'C' then unbalancing occurs because of LR. Hence LR rotation needs to be applied.

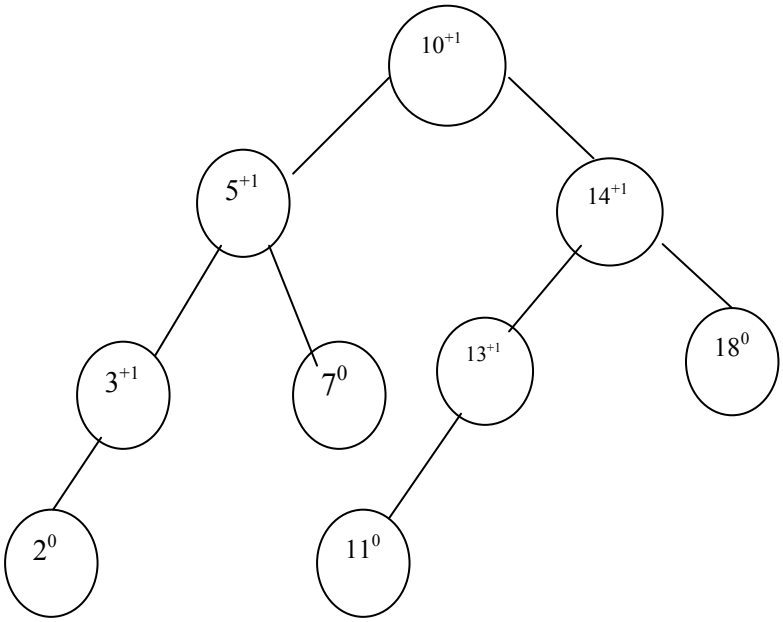
4. RL rotation



When node '2' is attached as a left child of node 'C' then node 'A' gets unbalanced as its balance factor becomes -2. Then RL rotation needs to be applied to rebalance the AVL tree.

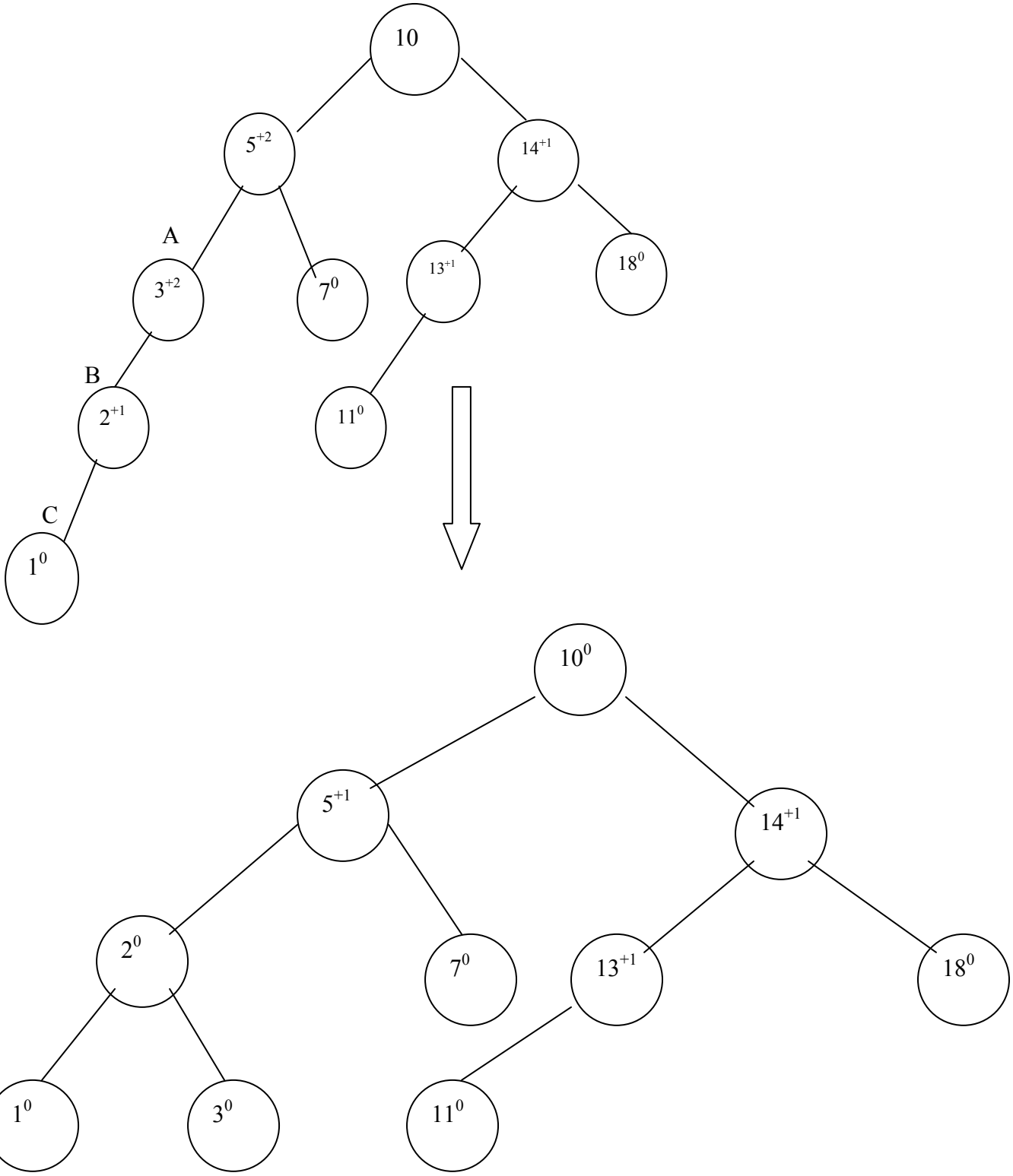
Example:

Insert 1, 25, 28, 12 in the following AVL tree.



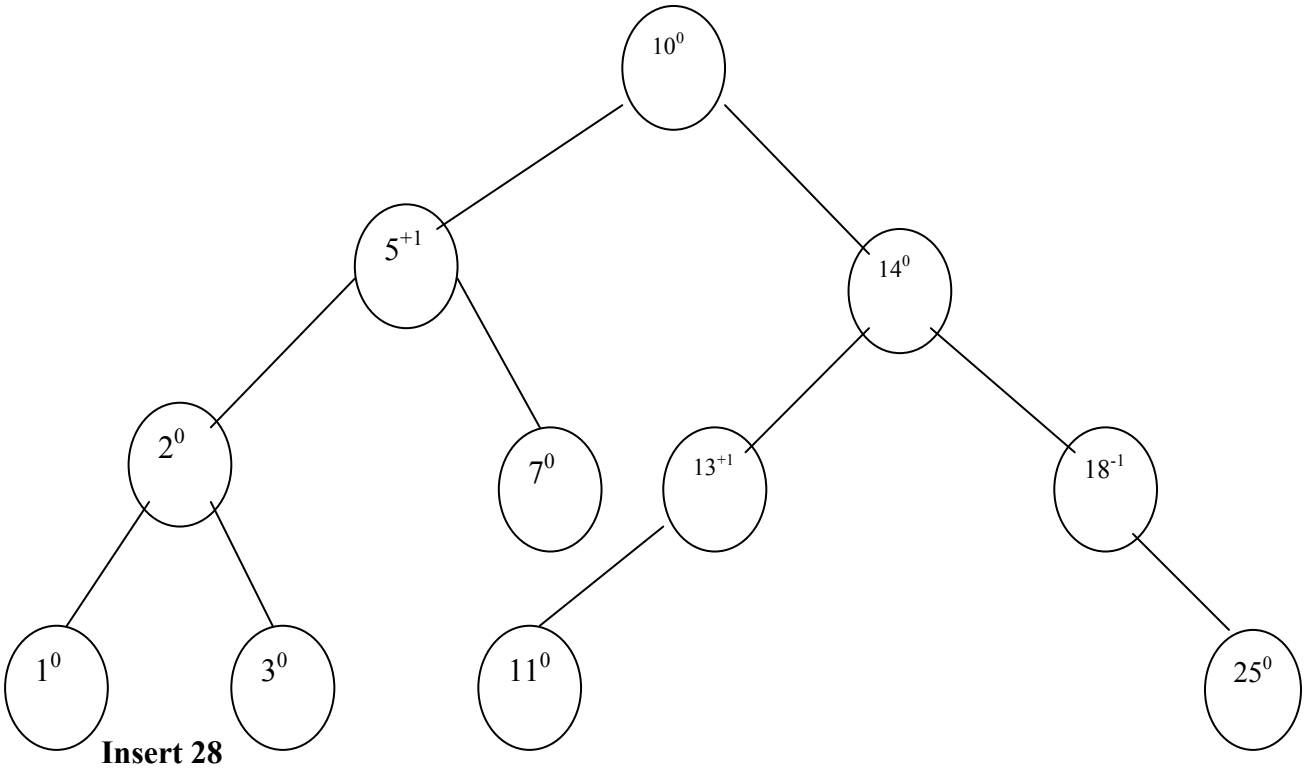
Insert 1

To insert node '1' we have to attach it as a left child of '2'. This will unbalance the tree as follows. We will apply LL rotation to preserve AVL property of it.



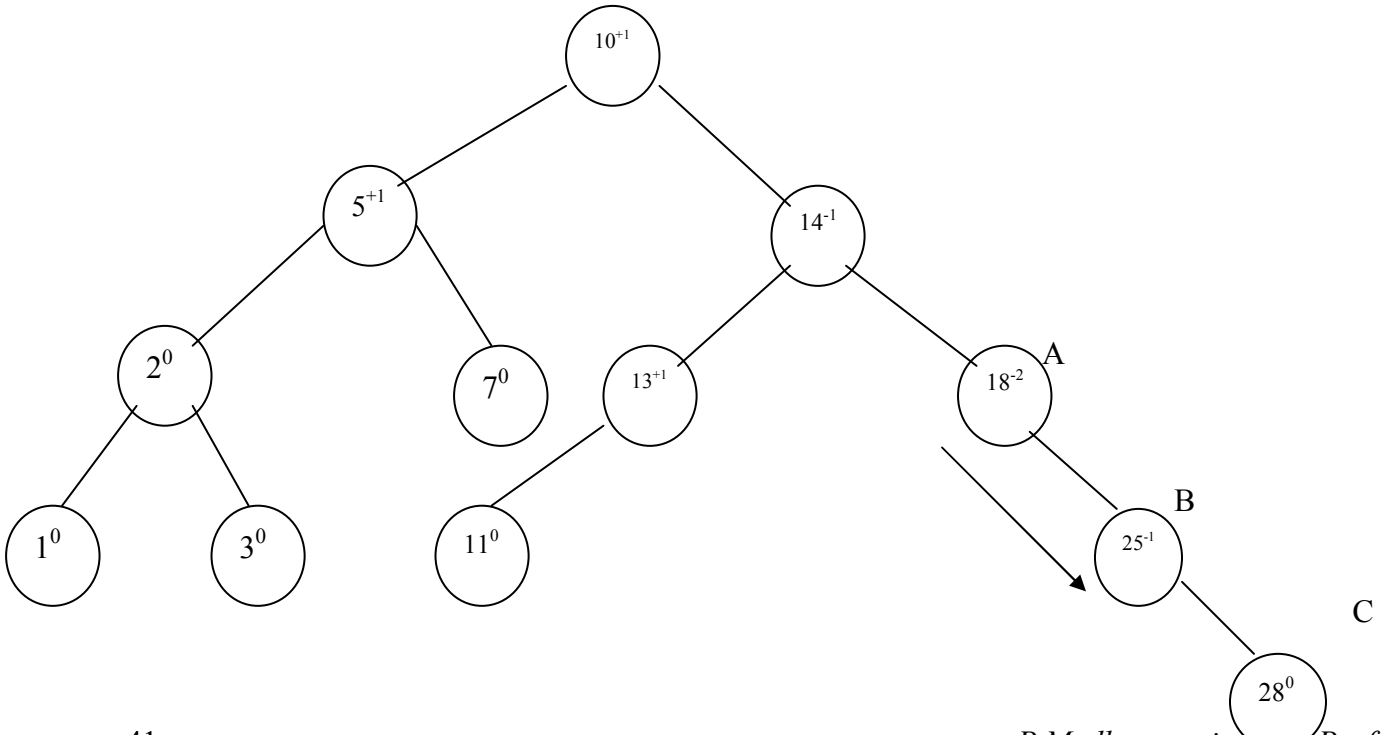
Insert 25

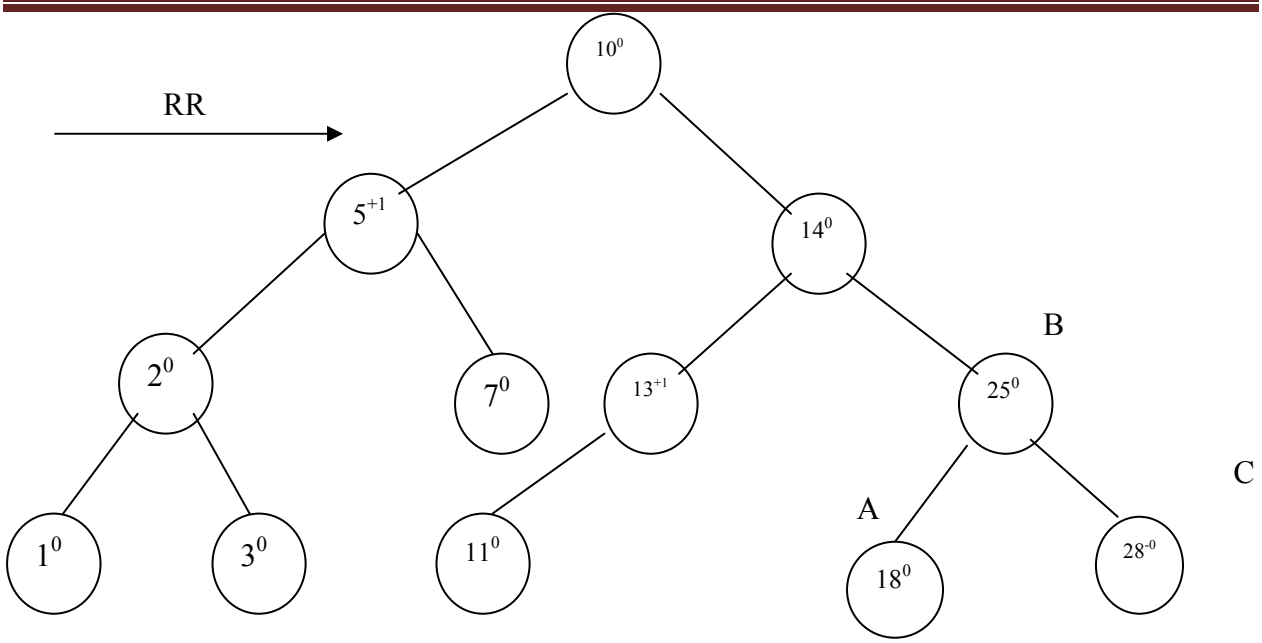
We will attach 25 as a right child of 18. No balancing is required as entire tree preserves the AVL property.



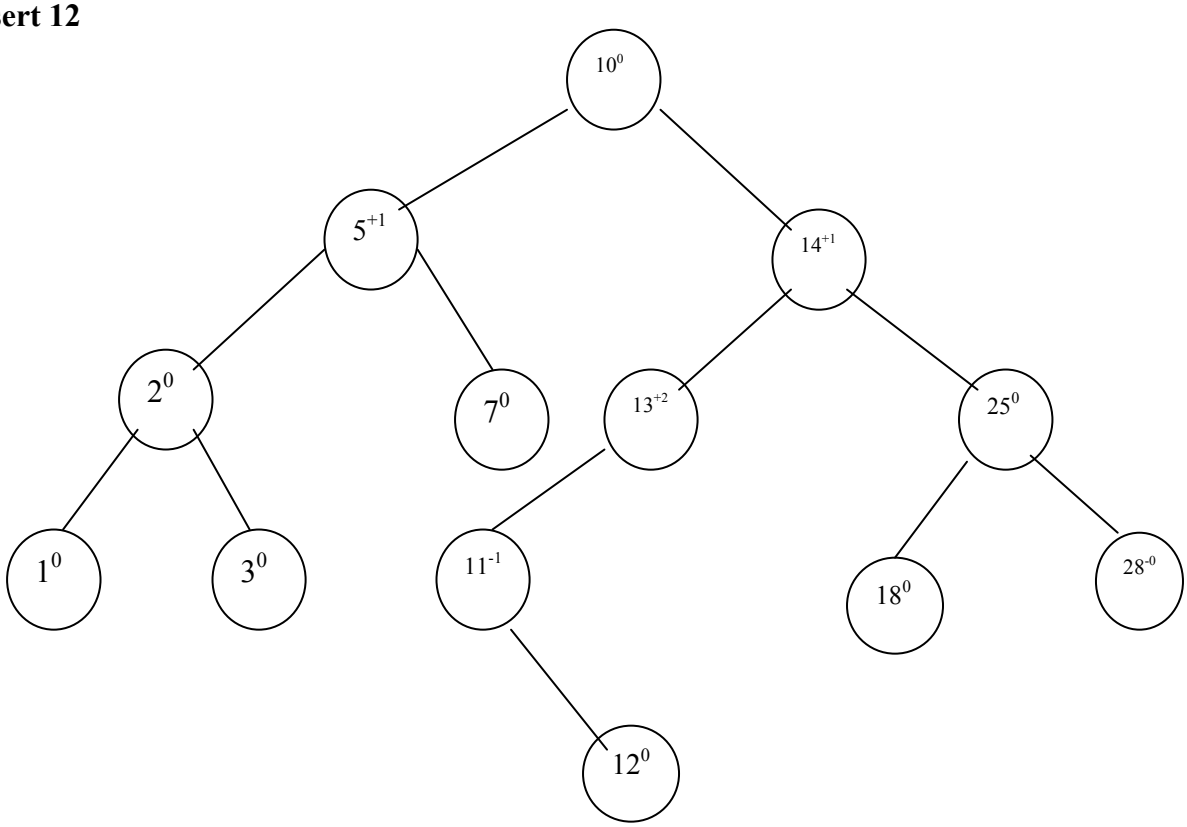
Insert 28

The node '28' is attached as a right child of 25. RR rotation is required to rebalance.

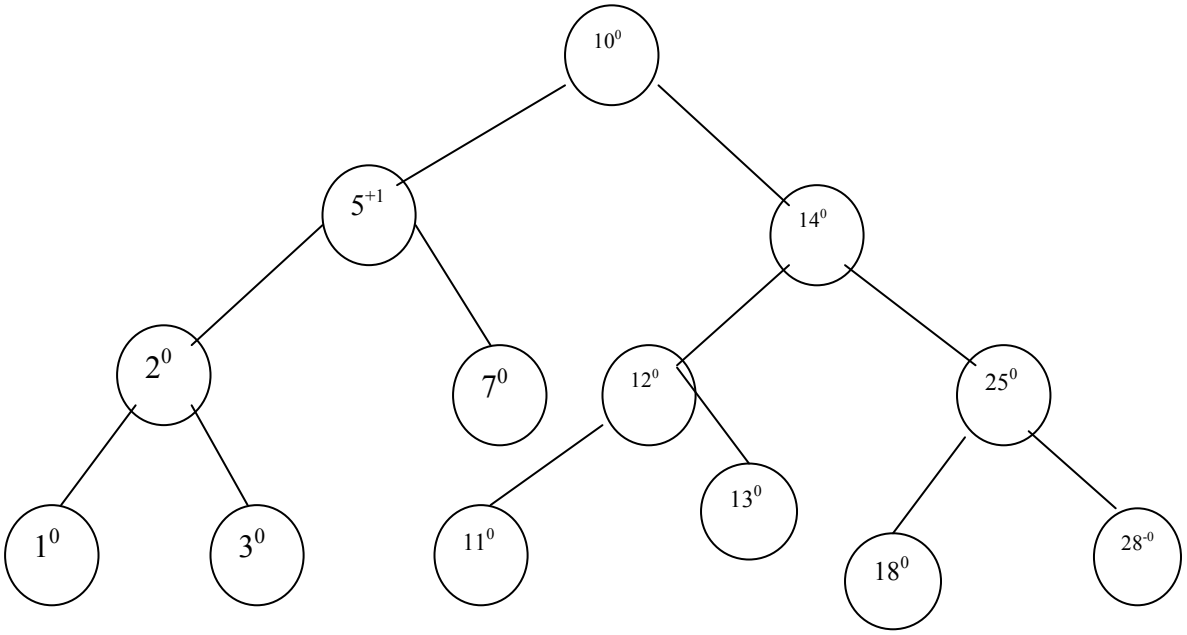



RR
10^0
5^{+1}
14^0
2^0
7^0
13^{+1}
25^0
1^0
3^0
11^0
18^0
28^{-0}
A
B
C

Insert 12


10^0
5^{+1}
14^{+1}
2^0
7^0
13^{+2}
25^0
1^0
3^0
11^{-1}
18^0
28^{-0}
12^0

To rebalance the tree we have to apply LR rotation.



Thus by applying various rotations depending upon direction of insertion of new node the AVL tree can be restructured.

DELETION

Even after deletion of any particular node from AVL tree, the tree has to be restructured in order to preserve AVL property. And thereby various rotations need to be applied.

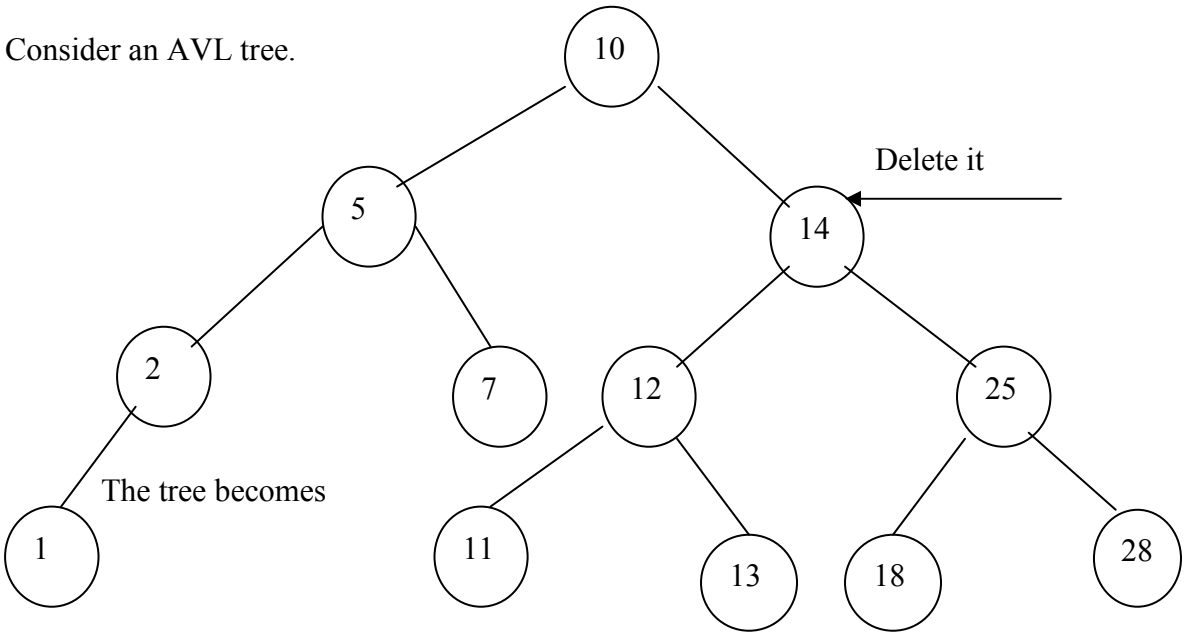
Algorithm for deletion:

The deletion algorithm is more complex than insertion algorithm.

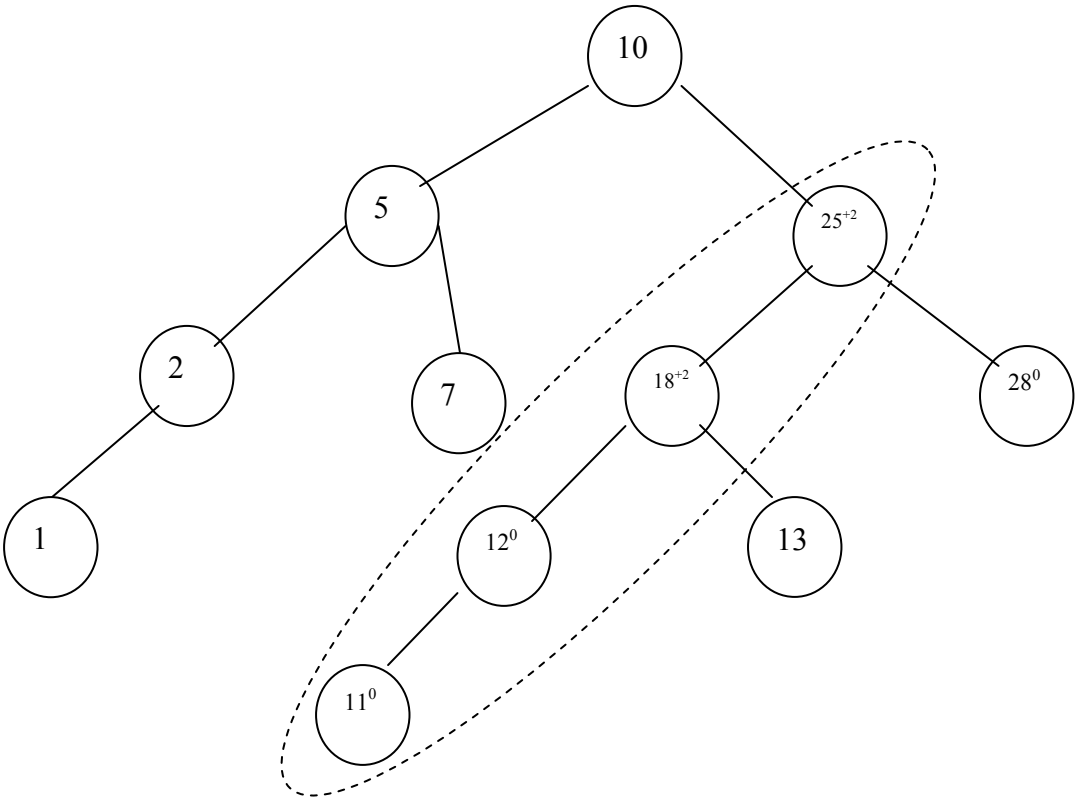
1. Search the node which is to be deleted.
2. a) If the node to be deleted is a leaf node then simply make it NULL to remove.
   b) If the node to be deleted is not a leaf node i.e. node may have one or two children, then the node must be swapped with its in order successor. Once the node is swapped, we can remove this node.
3. Now we have to traverse back up the path towards root, checking the balance factor of every node along the path. If we encounter unbalancing in some sub tree then balance that sub tree using appropriate single or double rotations.

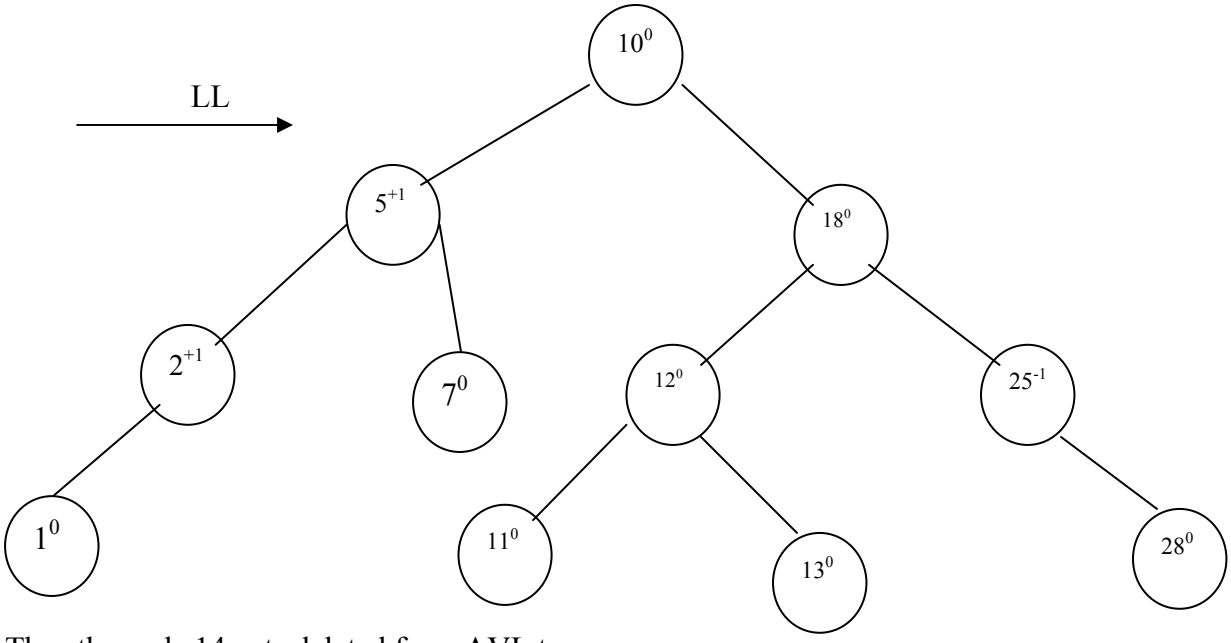
The deletion algorithm takes O(log n) time to delete any node.

Consider an AVL tree.



The tree becomes





Thus the node 14 gets deleted from AVL tree.

SEARCHING

The searching of a node in an AVL tree is very simple. As AVL tree is basically binary search tree, the algorithm used for searching a node from binary search tree is the same one is used to search a node from AVL tree.

The searching of a node from AVL tree takes O(log n) time.

B TREES

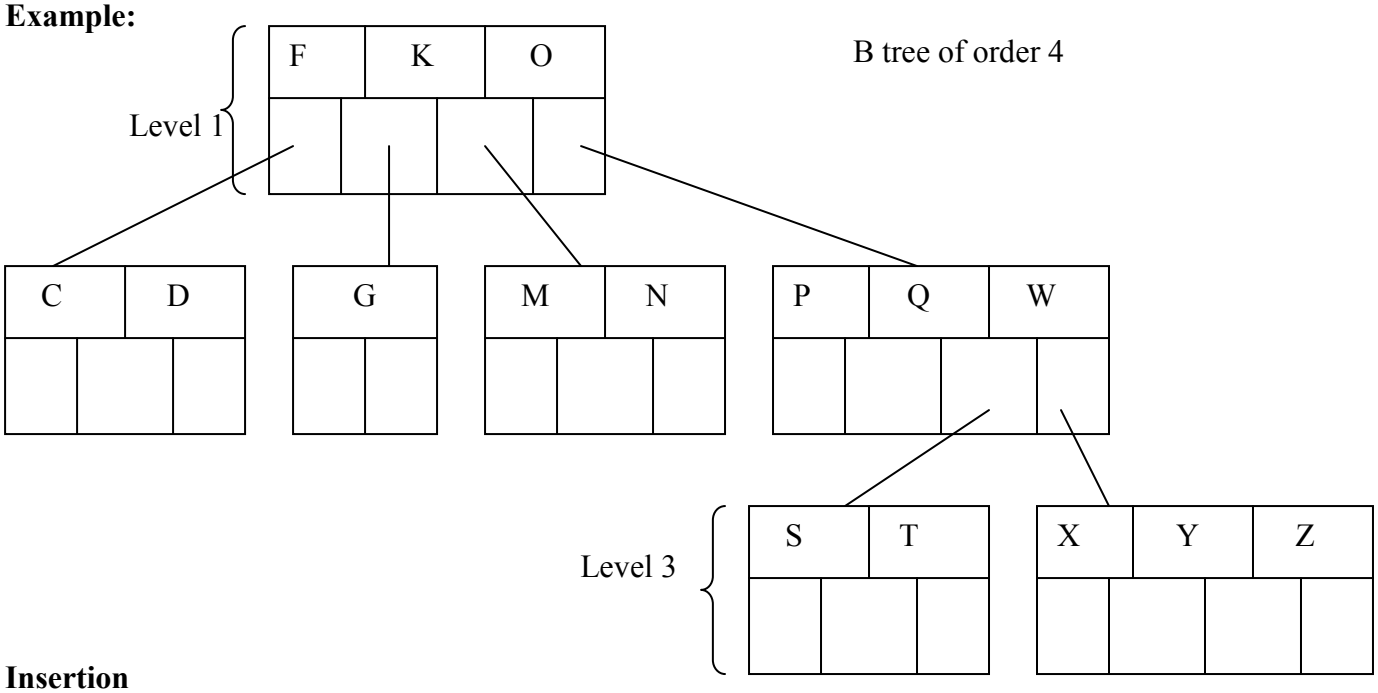
- Multi-way trees are tree data structures with more than two branches at a node. The data structures of m-way search trees, B trees and Tries belong to this category of tree structures.
- AVL search trees are height balanced versions of binary search trees, provide efficient retrievals and storage operations. The complexity of insert, delete and search operations on AVL search trees id O(log n).
- Applications such as File indexing where the entries in an index may be very large, maintaining the index as m-way search trees provides a better option than AVL search trees which are but only balanced binary search trees.
- While binary search trees are two-way search trees, m-way search trees are extended binary search trees and hence provide efficient retrievals.
- B trees are height balanced versions of m-way search trees and they do not recommend representation of keys with varying sizes.
- Tries are tree based data structures that support keys with varying sizes.

Definition:

A B tree of order m is an m-way search tree and hence may be empty. If non empty, then the following properties are satisfied on its extended tree representation:

i. The root node must have at least two child nodes and at most m child nodes.
ii. All internal nodes other than the root node must have at least |m/2 | non empty child nodes and at most m non empty child nodes.
iii. The number of keys in each internal node is one less than its number of child nodes and these keys partition the keys of the tree into sub trees.
iv. All external nodes are at the same level.

Example:



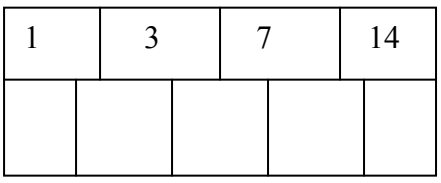
Insertion

For example construct a B-tree of order 5 using following numbers.

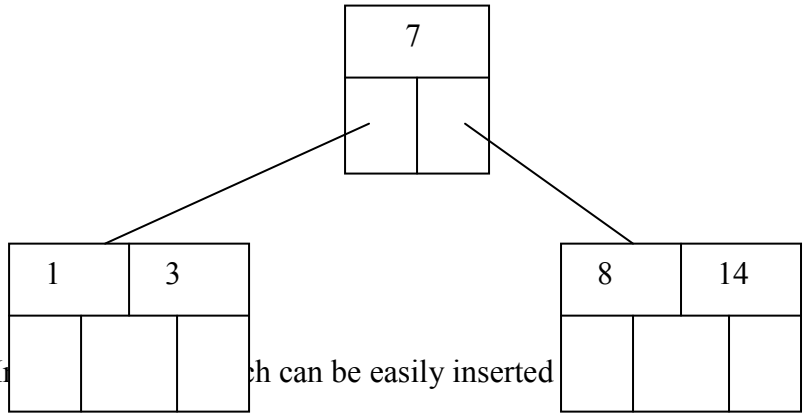
3, 14, 7, 1, 8, 5, 11, 17, 13, 6, 23, 12, 20, 26, 4, 16, 18, 24, 25, 19

The order 5 means at the most 4 keys are allowed. The internal node should have at least 3 non empty children and each leaf node must contain at least 2 keys.

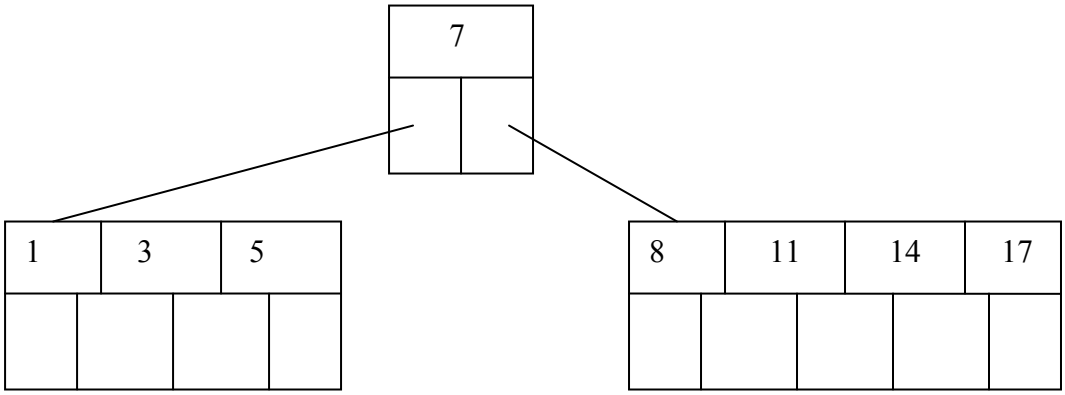
Step 1: Insert 3, 14, 7, 1



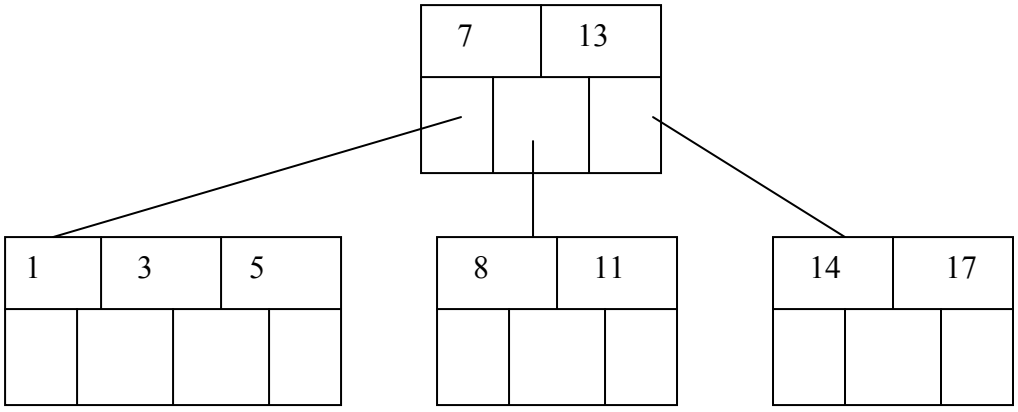
Step 2: Insert 8, Since the node is full split the node at medium 1, 3, 7, 8, 14



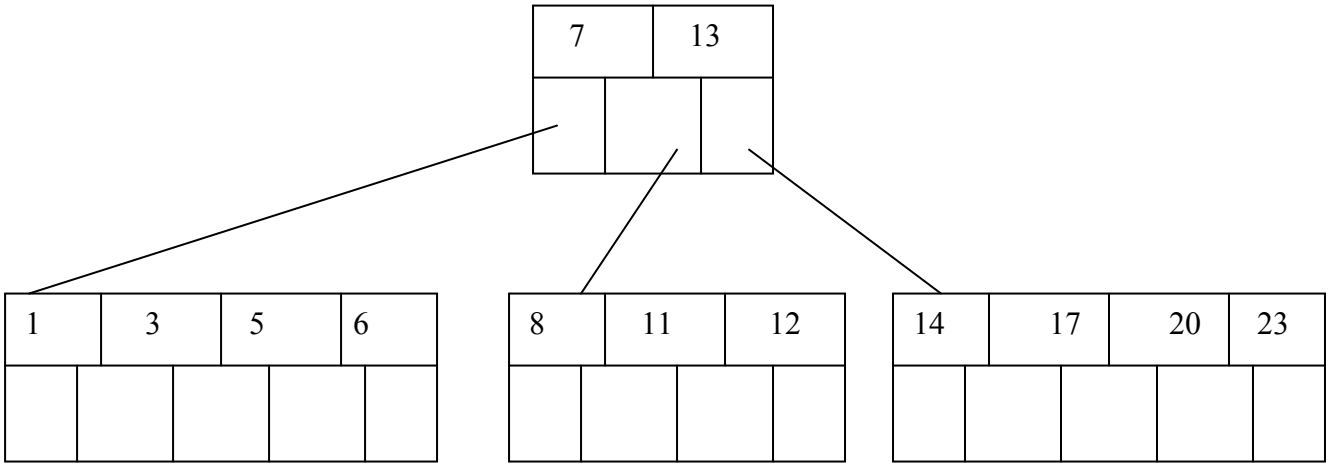
Step 3: I ch can be easily inserted



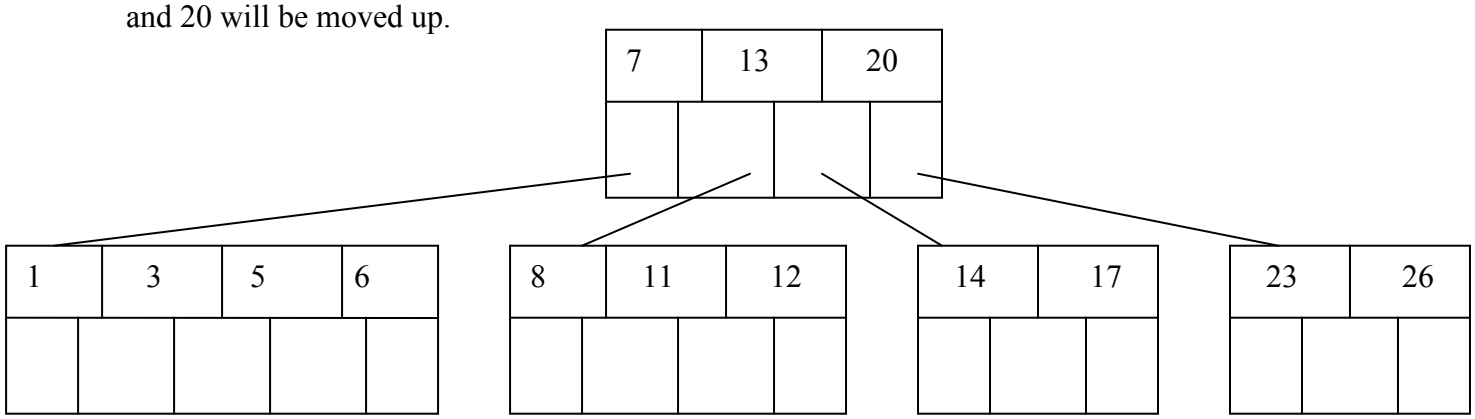
Step 4: Now insert 13. But if we insert 13 then the leaf node will have 5 keys which is not allowed. Hence 8, 11, 13, 14, 17 is split and medium node 13 is moved up.



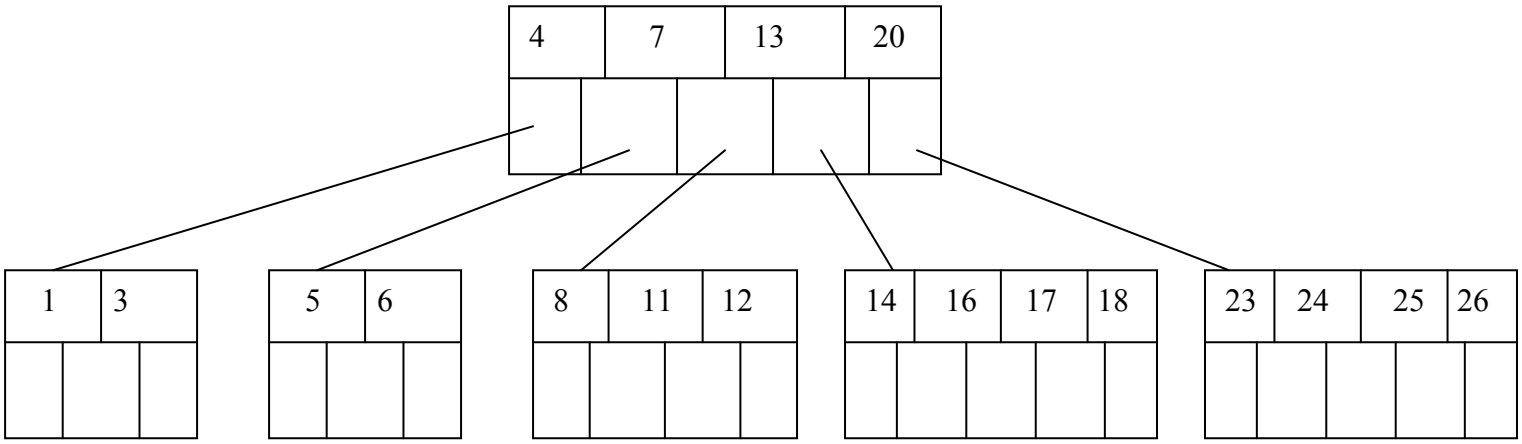
Step 5: Now insert 6, 23, 12, 20 without any split.



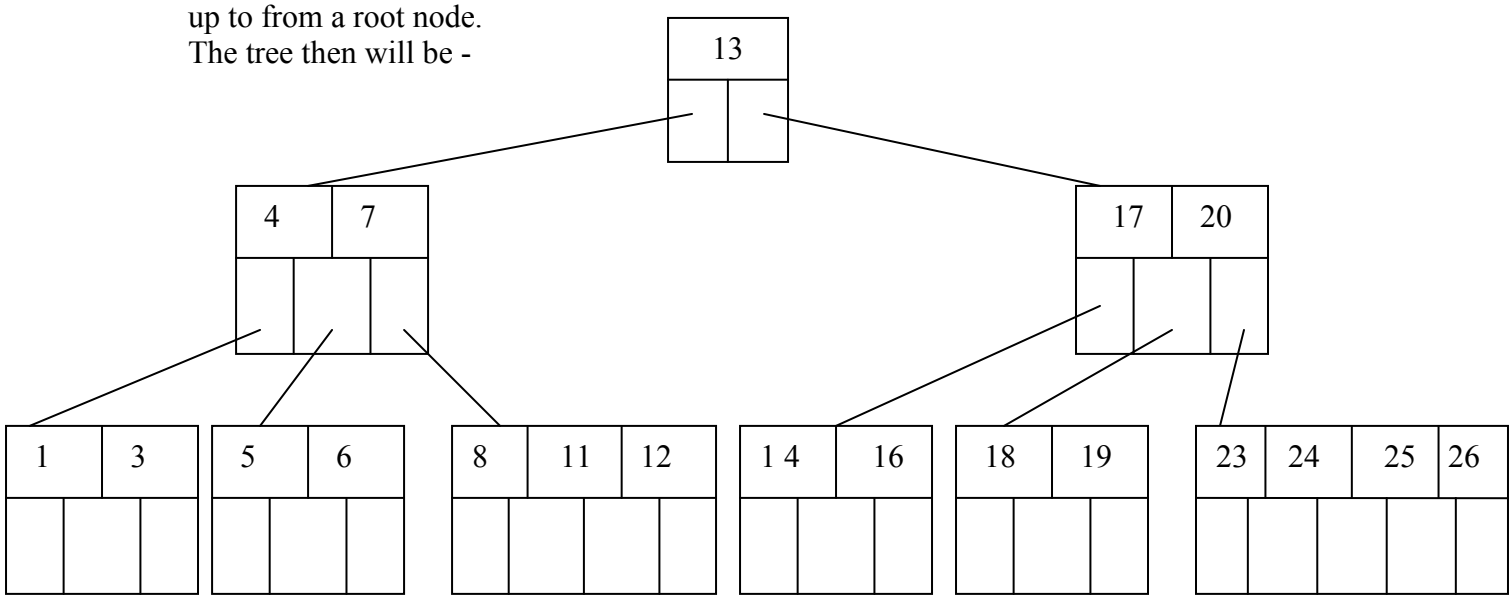
Step 6: The 26 is inserted to the right most leaf node. Hence 14, 17, 20, 23, 26 the node is split and 20 will be moved up.



Step 7: Insertion of node 4 causes left most node to split. The 1, 3, 4, 5, 6 causes key 4 to move up. Then insert 16, 18, 24, 25.

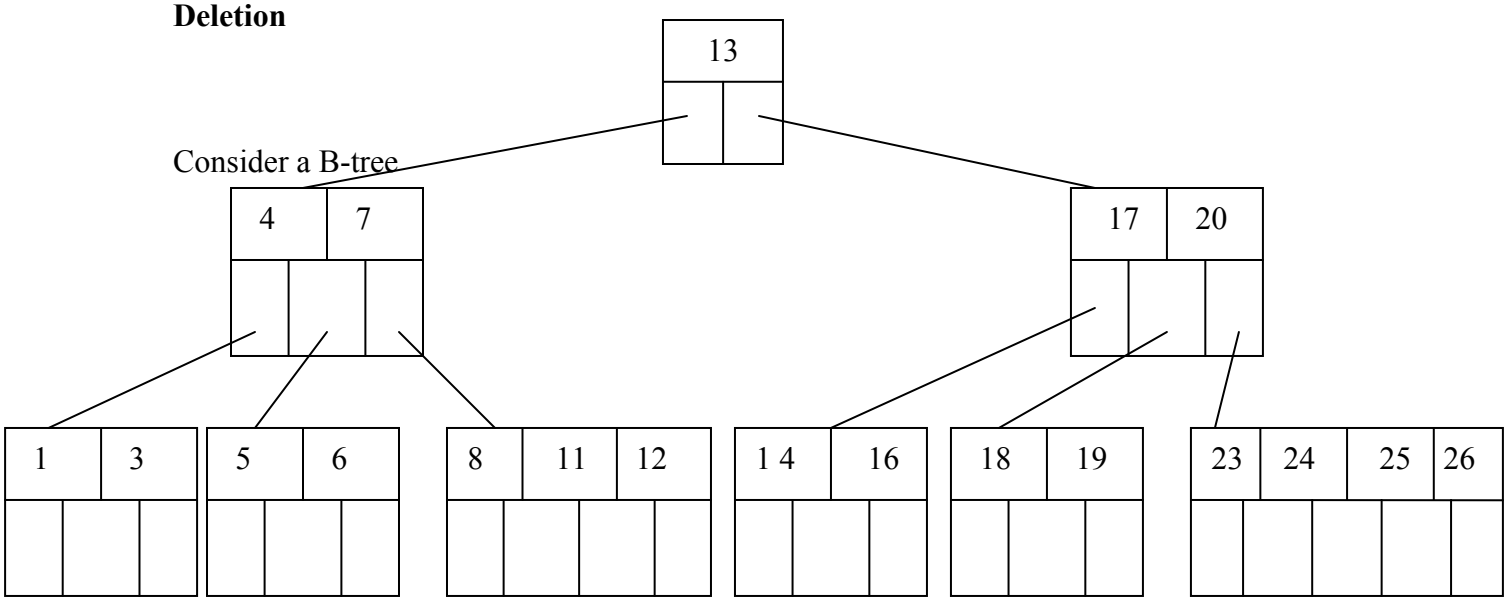


Step 8: Finally insert 19. Then 4, 7, 13, 19, 20 needs to be split. The median 13 will be moved up to from a root node.
The tree then will be -

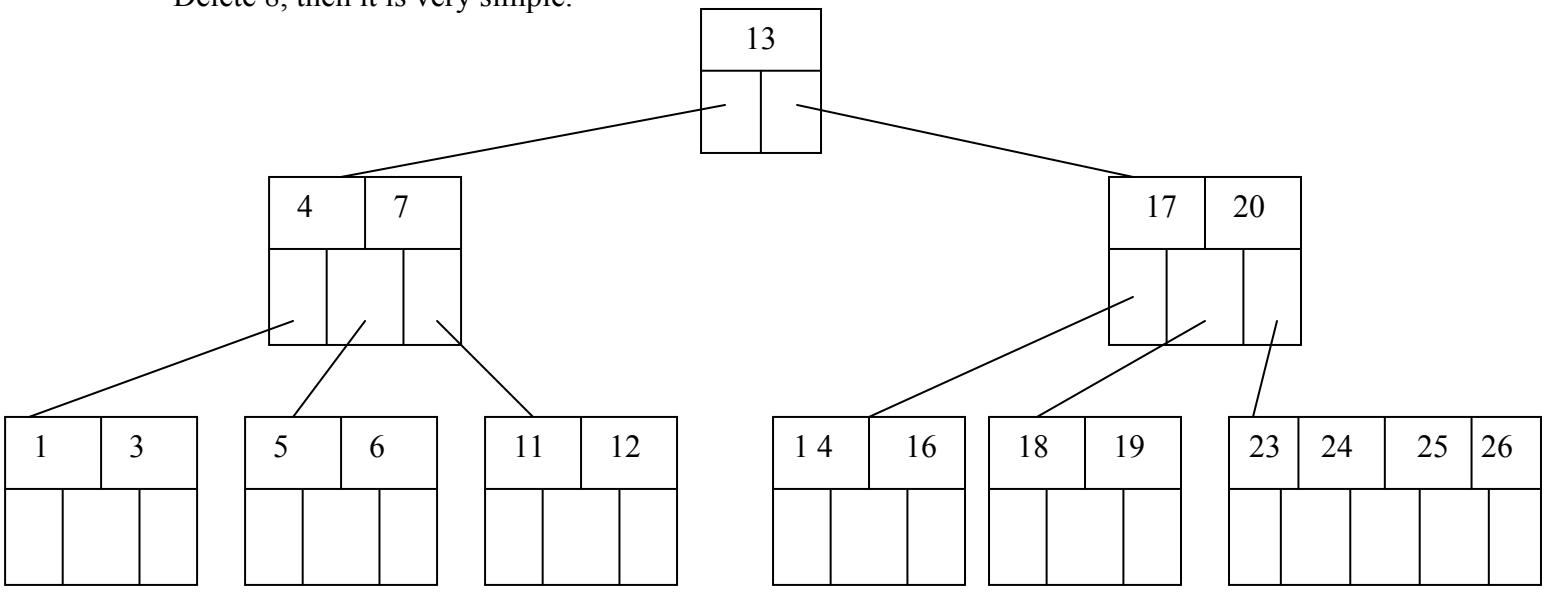


Deletion

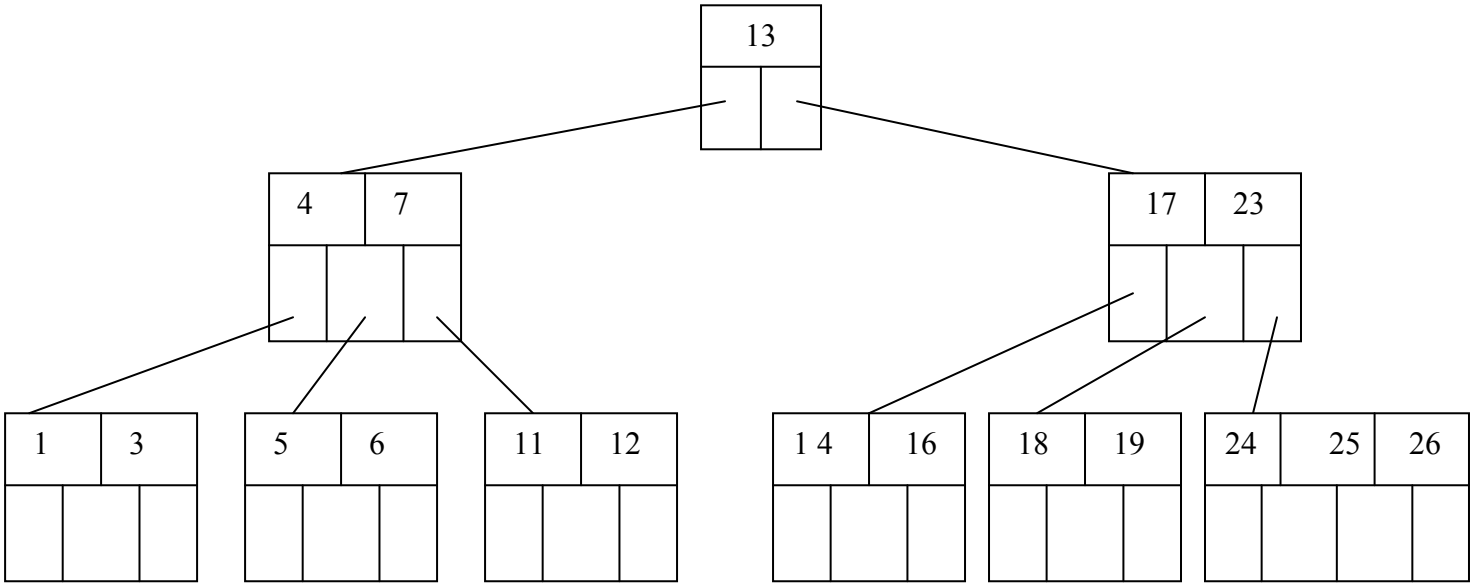
Consider a B-tree



Delete 8, then it is very simple.

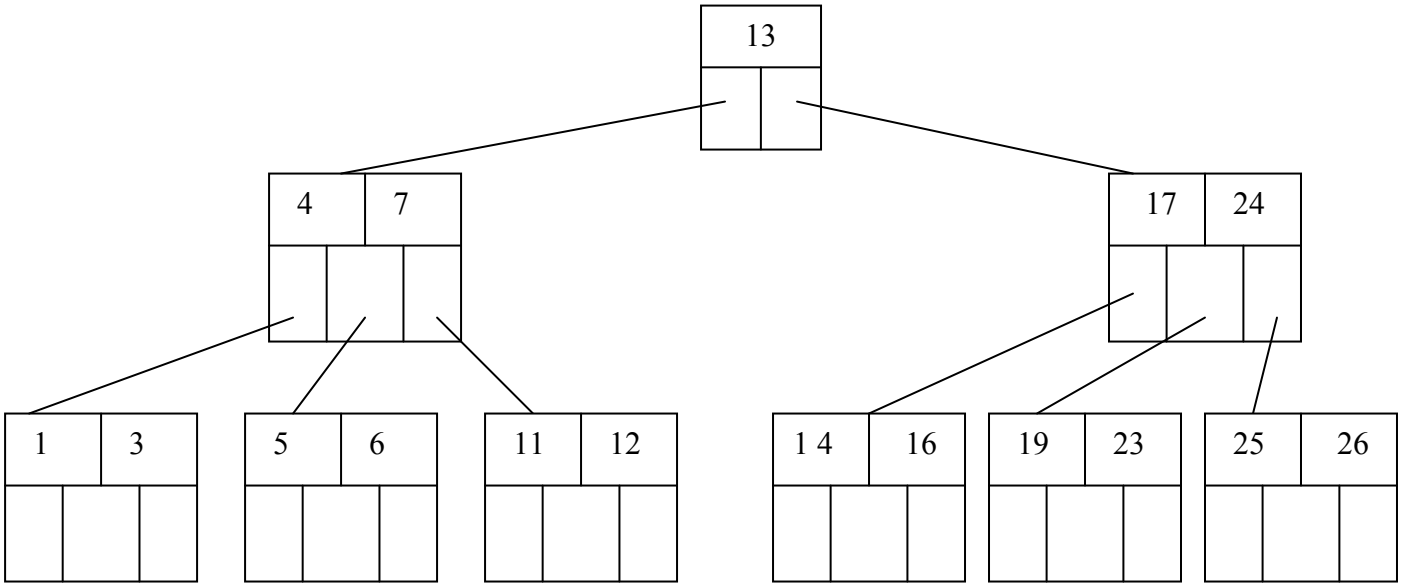


Now we will delete 20, the 20 is not in a leaf node so we will find its successor which is 23, Hence 23 will be moved up to replace 20.

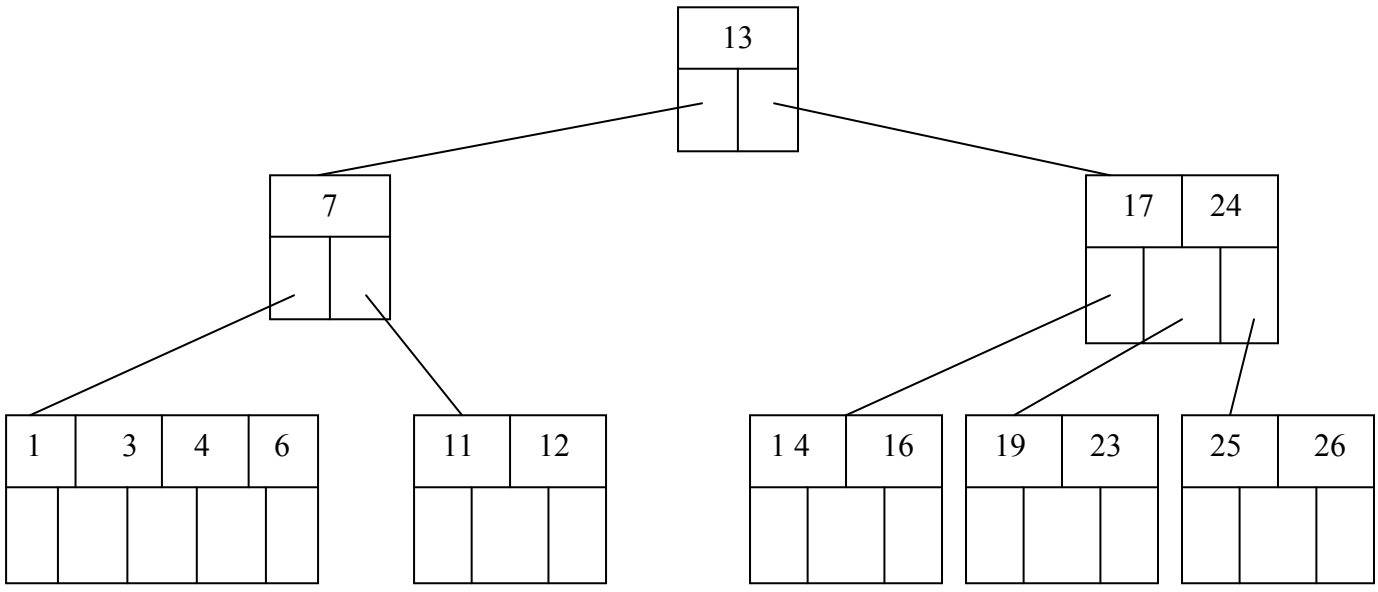


Next we will delete 18. Deletion of 18 from the corresponding node causes the node with only one key, which is not desired (as per rule 4) in B-tree of order 5. The sibling node to immediate

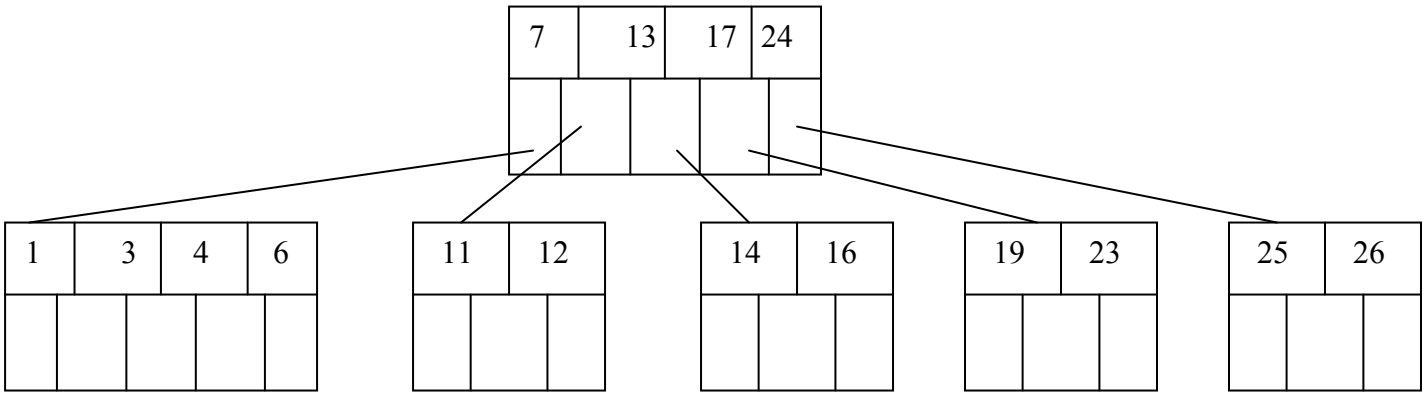
right has an extra key. In such a case we can borrow a key from parent and move spare key of sibling up.



Now delete 5. But deletion of 5 is not easy. The first thing is 5 is from leaf node. Secondly this leaf node has no extra keys nor siblings to immediate left or right. In such a situation we can combine this node with one of the siblings. That means remove 5 and combine 6 with the node 1, 3. To make the tree balanced we have to move parent's key down. Hence we will move 4 down as 4 is between 1, 3, and 6. The tree will be-

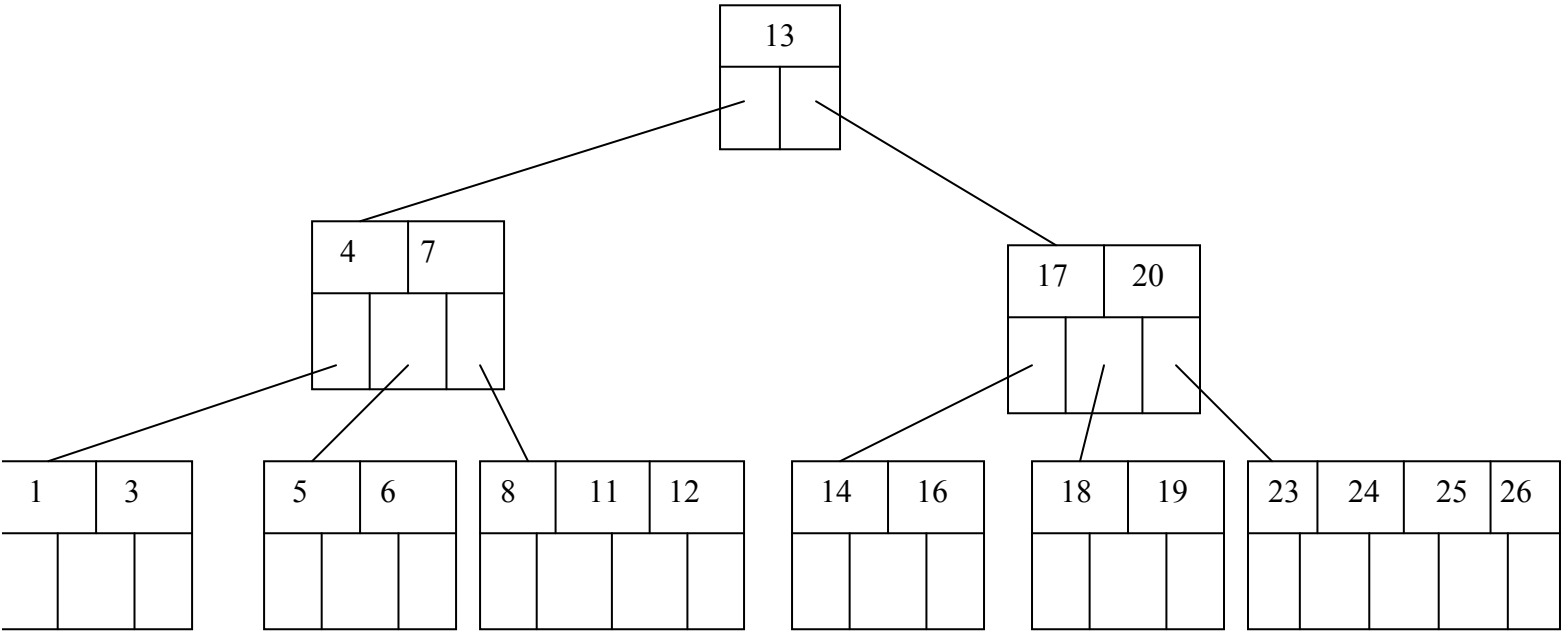


But again internal node of 7 contains only one key which not allowed in B-tree. We then will try to borrow a key from sibling. But sibling 17, 24 has no spare key. Hence we can do is that, combine 7 with 13 and 17, 24. Hence the B-tree will be



Searching

The search operation on B-tree is similar to a search to a search on binary search tree. Instead of choosing between a left and right child as in binary tree, B-tree makes an m-way choice. Consider a B-tree as given below.



If we want to search 11 then

i. 11 < 13 ; Hence search left node

ii. 11 > 7 ; Hence right most node

iii. 11 > 8 ; move in second block

iv. node 11 is found

The running time of search operation depends upon the height of the tree. It is O(log n).

Height of B-tree

The maximum height of B-tree gives an upper bound on number of disk access. The maximum number of keys in a B-tree of order 2m and depth h is

$1 + 2m + 2m(m+1) + 2m(m+1)^2 + \ldots + 2m(m+1)^{h-1}$

$$= 1 + \sum_{i=1}^{h} 2m(m+1)^{i-1}$$

The maximum height of B-tree with n keys

$$\log_{m+1} \frac{n}{2m} = O(\log n)$$

Red-Black Tree

Definition:

Red-Black tree is a binary search tree in which every node is colored with either red or black. It is a type of self balancing binary search tree. It has a good efficient worst case running time complexity. The Red-black tree satisfies all the properties of binary search tree in addition to that it satisfies following additional properties-

1. The root and the external nodes are always black nodes.
2. [Red Condition] No two red nodes can occur consecutively on the path from the root node to an external node.
3. [Black Condition] The number of black nodes on the path from the root node to an external node most be the same for all external nodes.

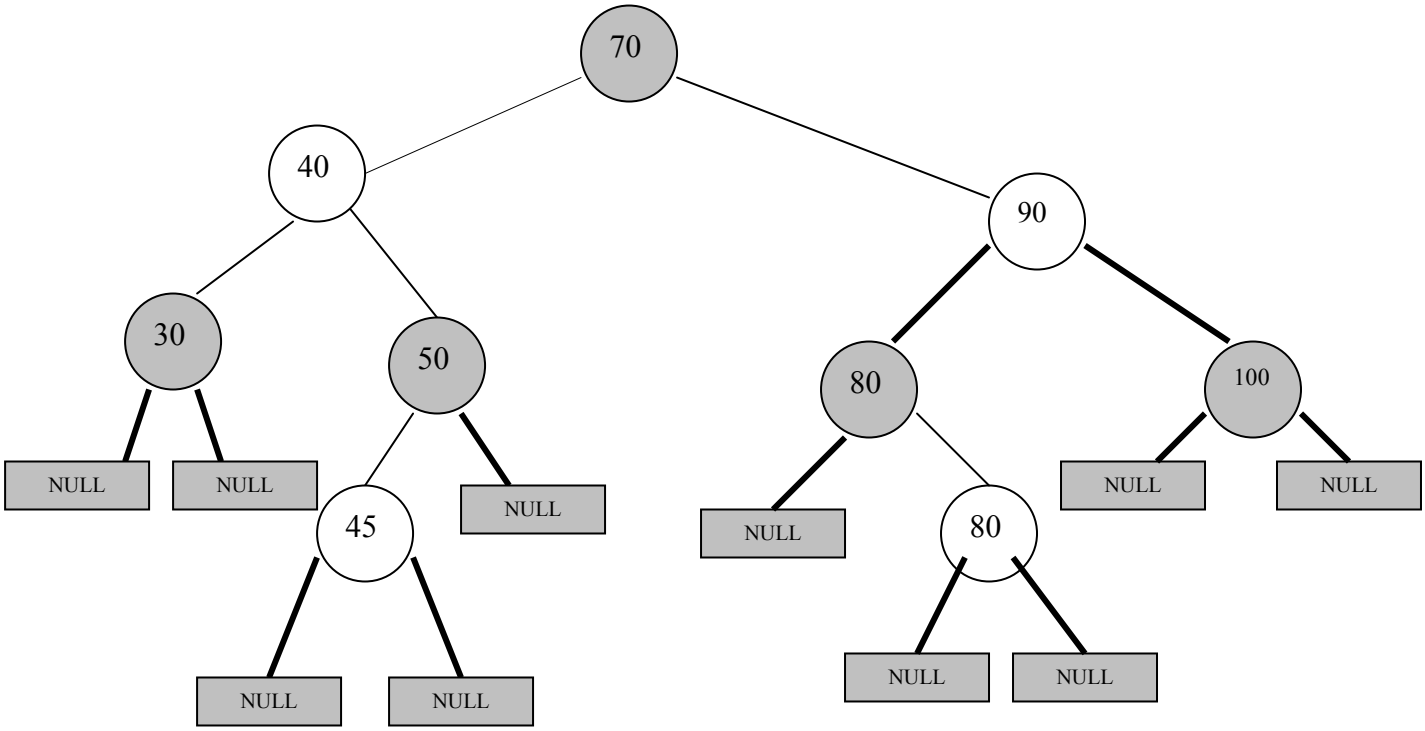
Since the color of the node is same as the color of the edge from which the node emanates, the Red and Black Conditions may be expressed in terms of the edges as well. The Red Condition could be alternatively defined as no two red pointers or edges can occur consecutively on a path from the root node to an external node. The black condition could be redefined as all paths from

the root node to external nodes must have the same number of black pointers. Besides the above mentioned conditions, all pointers linking internal nodes with the external nodes must be black.

The number of black nodes or edges on the path from a node to an external node is called the rank of the node. The rank of all external nodes is 0.

Representation:

While representing a red black tree color of every node and pointer colors are shown. The leaf nodes are simply NULL nodes.



The Red-Black tree shown in above given figure has black nodes that are shaded in black and un shaded nodes are Red nodes. Similarly the leaves are black and all are NULL pointers only. The pointers to black node are black pointers which are shown by thick lines remaining are red pointers. But in practice, we explicitly mention the color of nodes. The above given Red-Black tree follows all the properties of Red-Black tree-

1. It is a binary search tree.
2. The root node is black.
3. The children of red node are black.
4. No root to external node path has two consecutive red nodes (e.g. 70-90-80-88-NULL).

5. All the root to external node paths contain same number of black nodes (including root and external node).

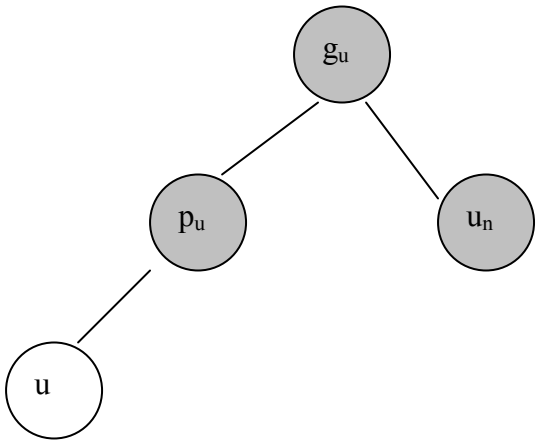
For e.g. : consider path 70-40-30-NULL and 70-90-80-88-NULL in both these paths 3 black nodes are there. Similarly other paths are checked.

Insertion:

- Every new node which is to be inserted is marked red.
- Not every insertion causes imbalancing but if imbalancing occurs then that can be removed depending upon the configuration of tree before new insertion made.
- To understand insertion operation, let us understand the configuration of tree by defining following roles.

Let u is newly inserted node.

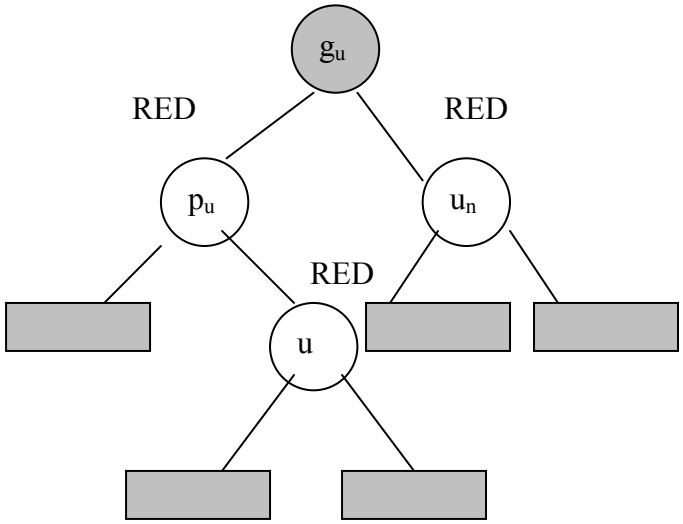
p_u is the parent node of u.
g_u is grand parent of u and parent node of p_u
u_n is an uncle node of u i.e. it's a right child of g_u.



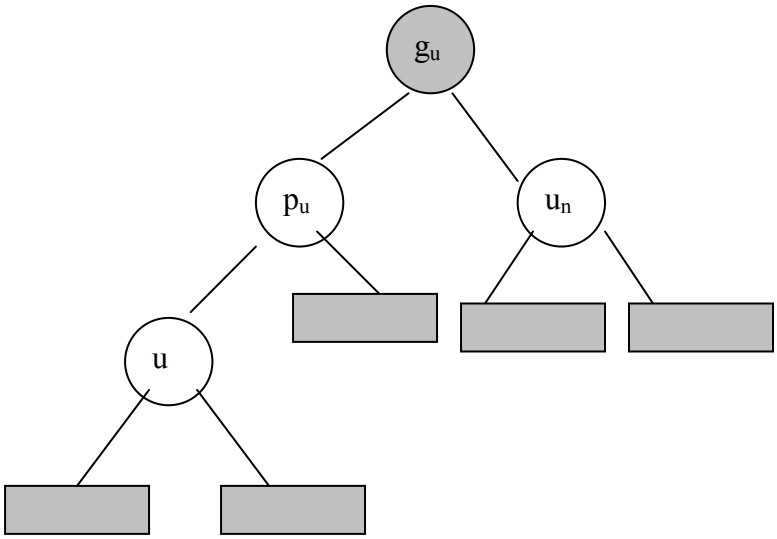
The tree is said to be imbalanced if properties of Red-Black tree are violated.

- When insertion occurs, the new node is inserted in already balanced tree. If this insertion causes any imbalancing then balancing of the tree is to be done at two levels:
  -- at grand parent level i.e. g_u
  -- at parent level i.e. p_u
- The imbalacing is concerned with the color of grand parent's child i.e. uncle node. If uncle node is red then there are four cases.
  1. LR_r
  2. LL_r
  3. RR_r
  4. RL_r

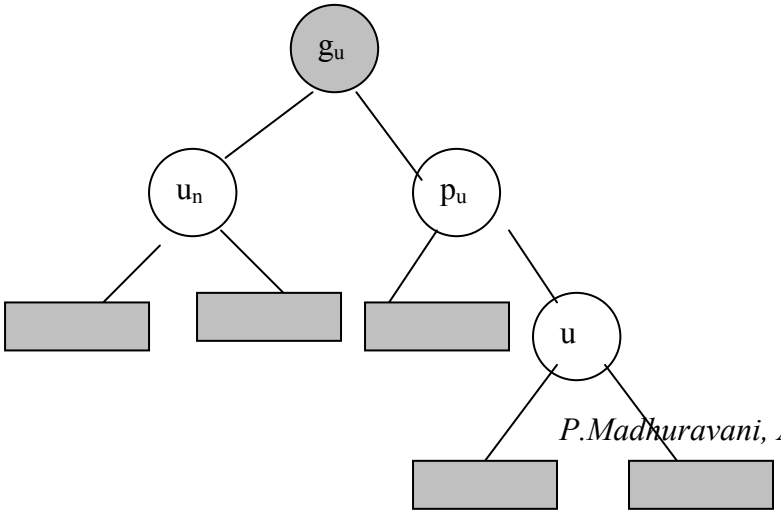
1. LR_r imbalance – The left child of g_u is p_u and u is right child of p_u and u_n node(uncle node) is red.



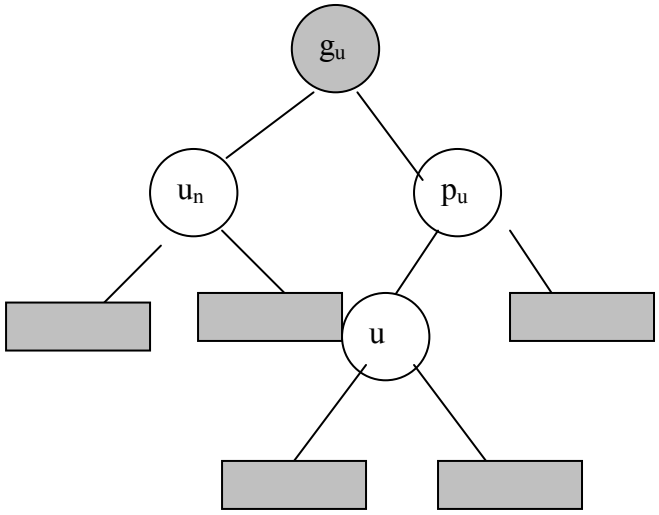
2. LL_r imbalance – The node p_u is a left child of g_u and u is inserted as left child of p_u. Node u_n is red.



3. RR_r imbalance – The right child of node g_u is node p_u and u is inserted as right child of p_u. The u_n node is red.



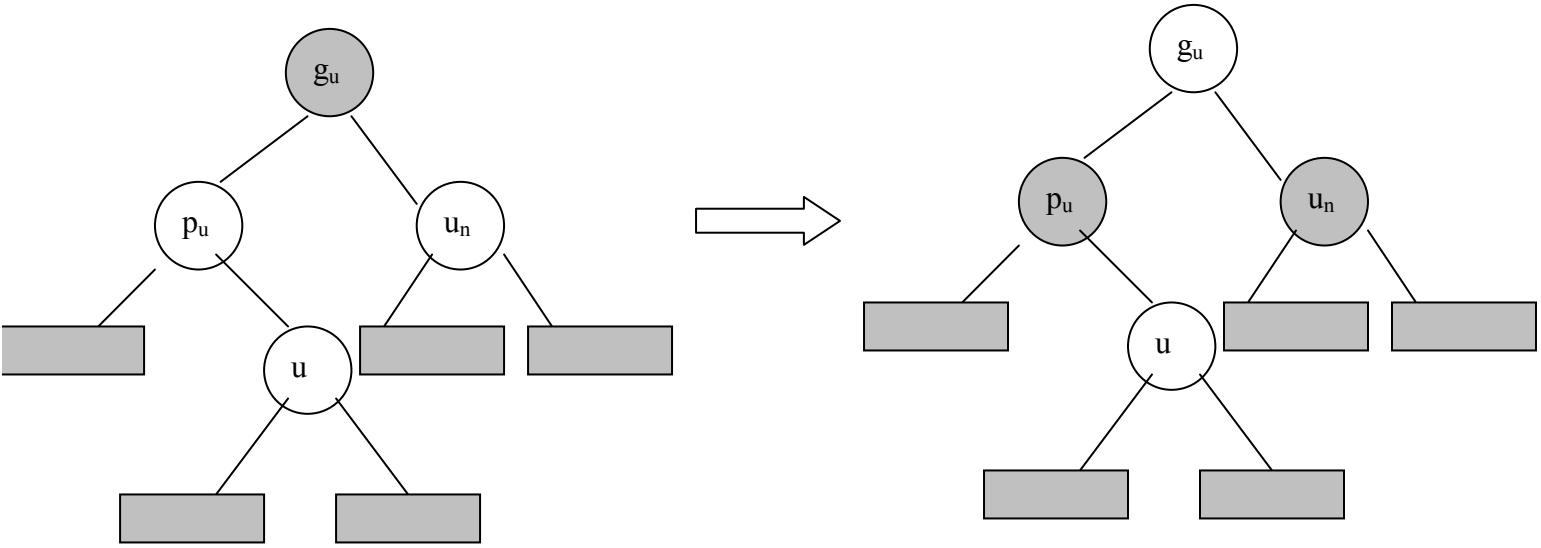
4. RL_r imbalance – The node p_u is right child of g_u and u is inserted as a left child of p_u. The uncle node u_n is red.



To remove these imbalancing rotations are not required. Simply by changing the colors required balancing can be obtained.

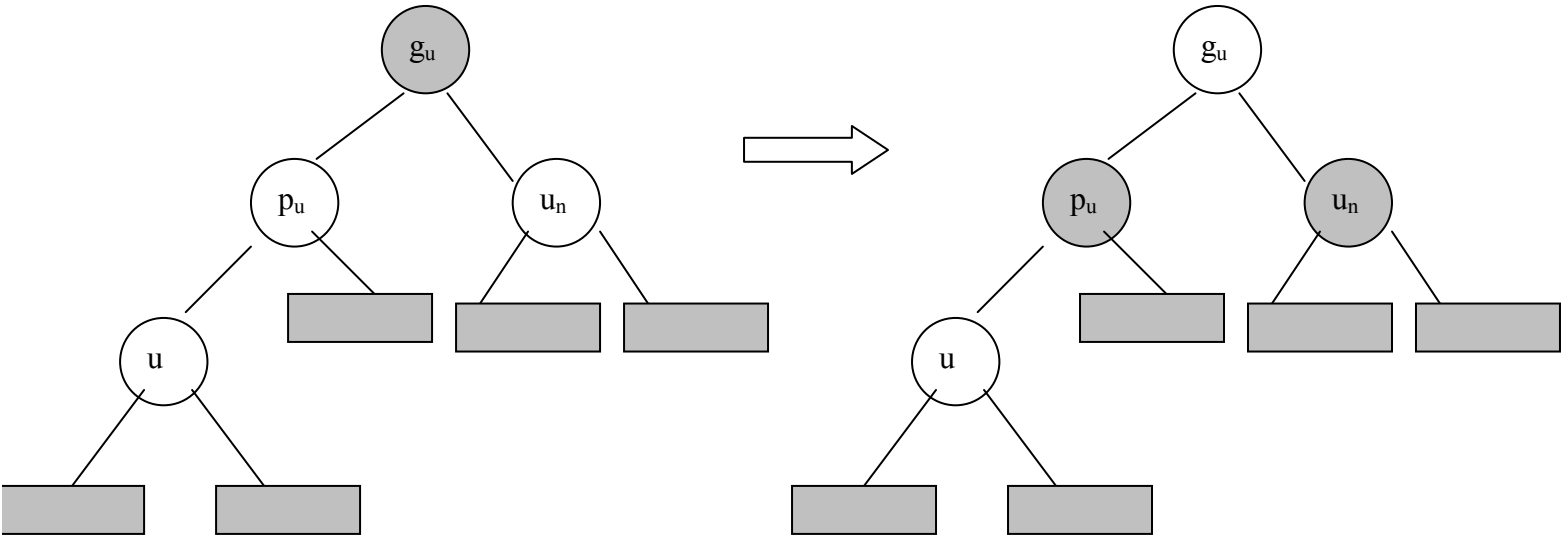
1. Removal of LR_r imbalancing

Before color change note that if g_u in given figures is root then there should not be any color change of g_u. (Because root is always black). But if g_u happens to be red then the rebalancing can be done as -



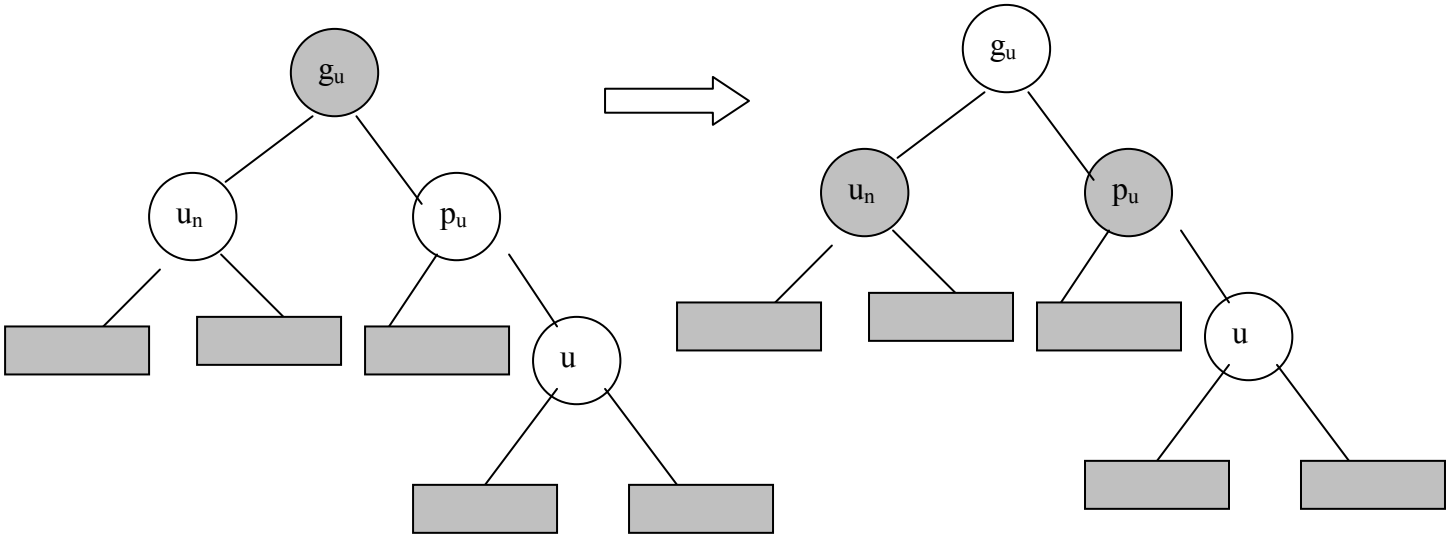
1. Change color of p_u from red to black.
2. Change color of un from red to black.
3. Change color of gu from black to red provided g_u is not a root node.

2. Removal of LL_r imbalancing



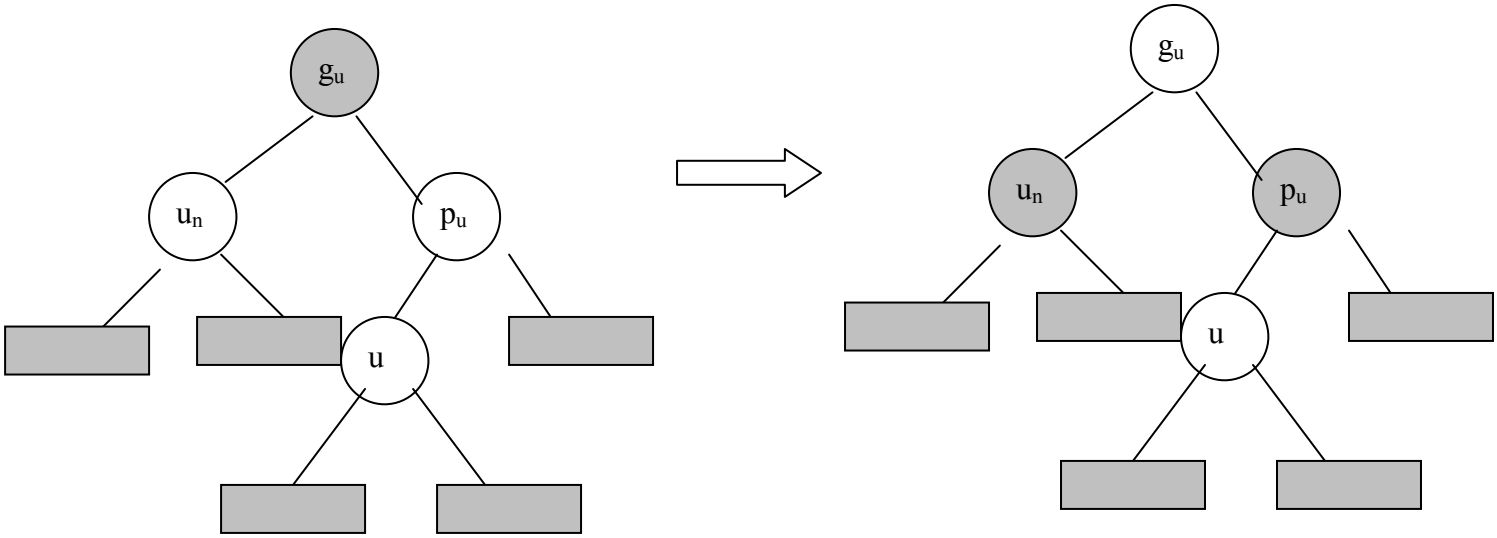
1. Change color of p_u from red to black.
2. Change color of un from red to black.
3. Change color of gu from black to red when g_u is not a root node.

3. Removal of RR_r imbalancing



1. Change color of p_u from red to black.
2. Change color of un from red to black.
3. Change color of gu from black to red when g_u is not a root node.

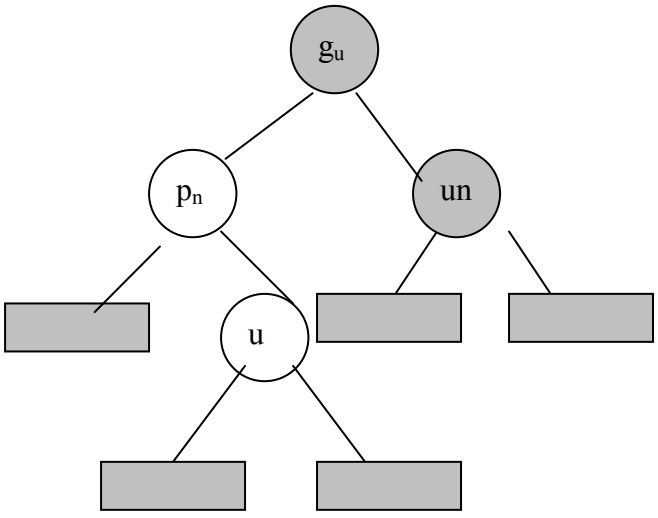
4. Removal of RL_r imbalancing



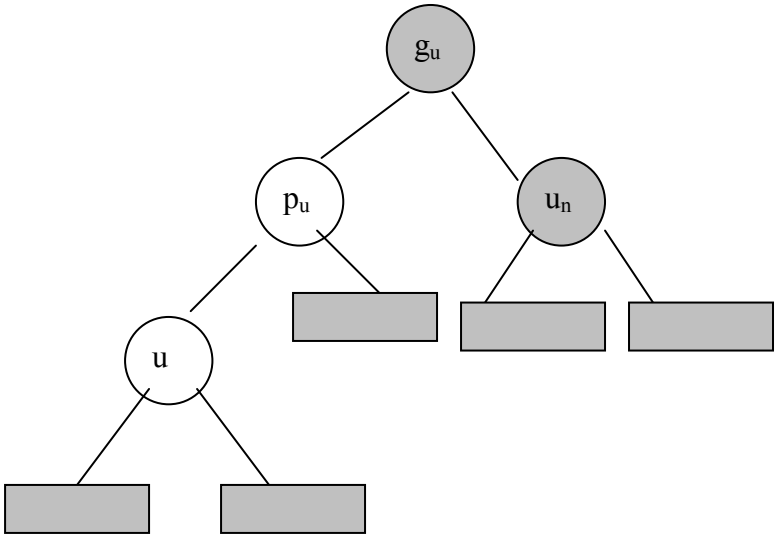
1. Change color of p_u from red to black.
2. Change color of un from red to black.
3. Change color of gu from black to red when g_u is not a root node.

Now when other child of g_u i.e. uncle node u_n is black then there arises four cases.

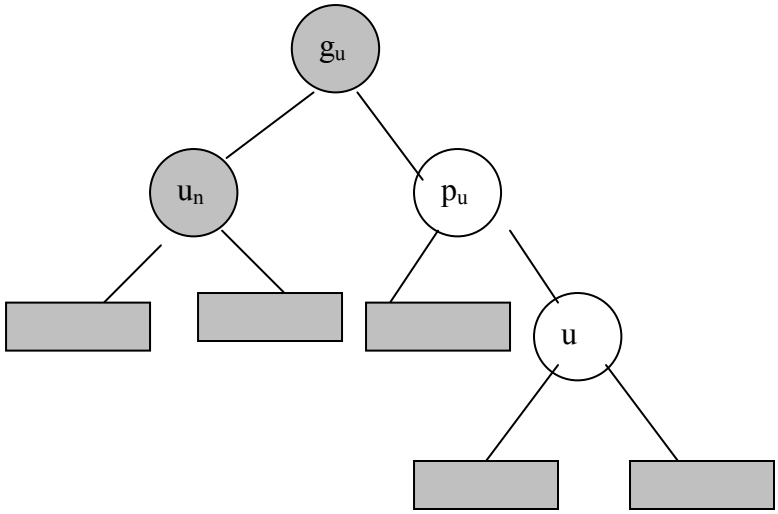
1. LR_b imbalancing: The p_u node is attached as a left child of g_u and u is inserted as a right child of p_u. The node u_n is black.



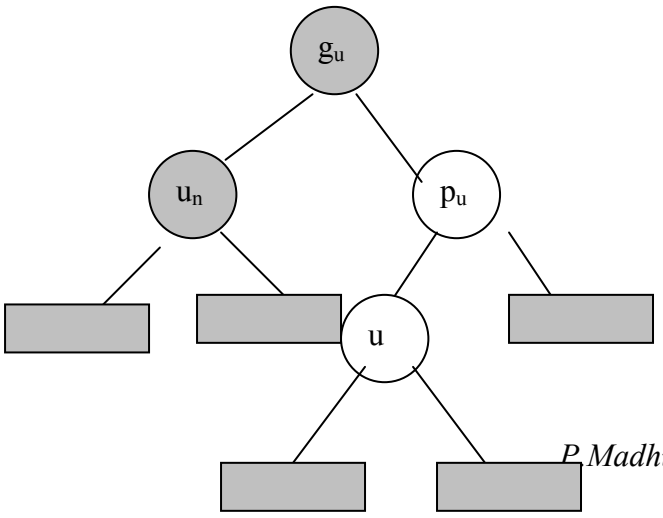
2. LL_b imbalancing : The node p_u is a left child of g_u and u node is a left child of p_u. The node u_n is black.



3. RR_b imbalancing : The node p_u is a right child of g_u and node u is a right child of p_u. The node u_n is black.



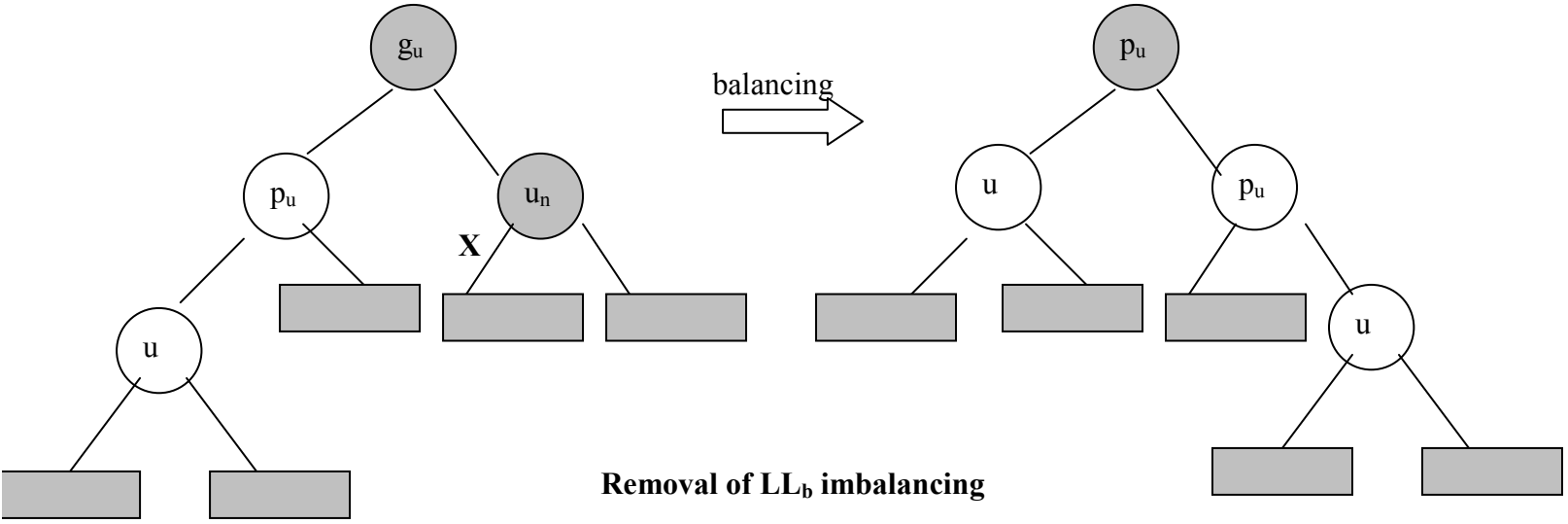
4. RL_b imbalancing : The node p_u is a right child of node g_u and node u is a right child of p_u. The node u_n is black.



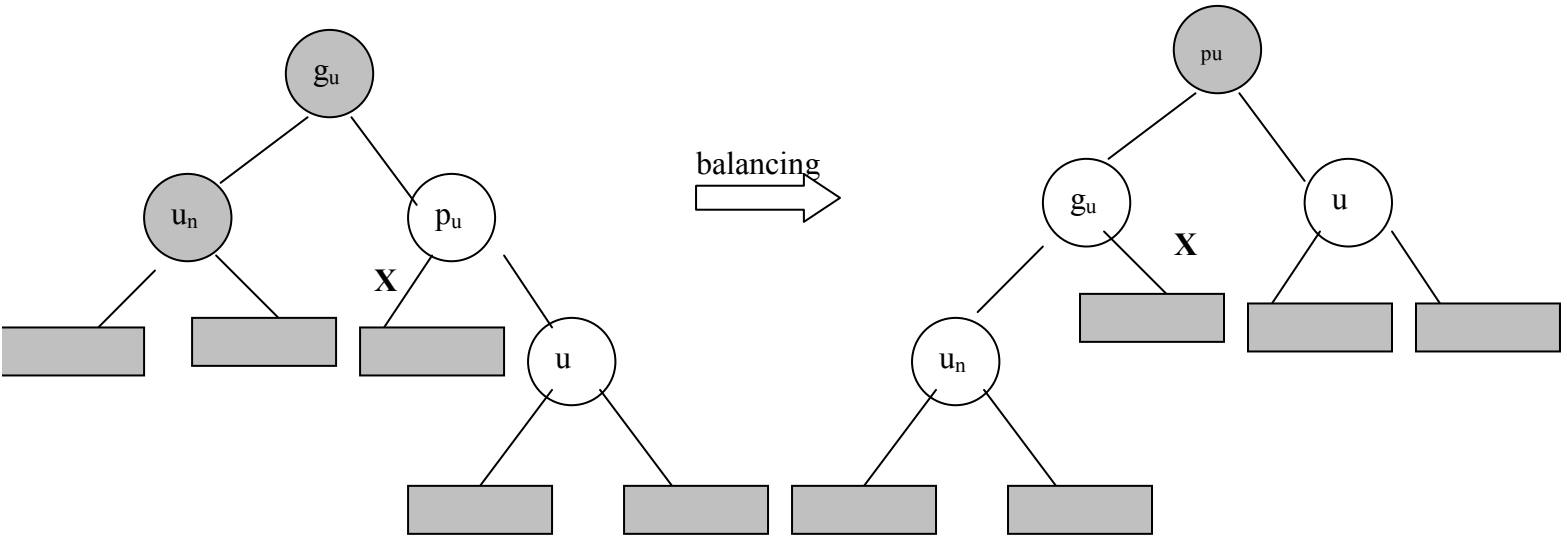
As u node gets inserted rebalancing must be performed.
-LL_b and RR_b cases require single rotation followed by recoloring.
-LR_b and RL_b cases require double rotation followed by recoloring.

Removing LL_b and RR_b imbalances

1. Apply single rotation of p_u about g_u.
2. Recolor p_u to black and g_u to red.



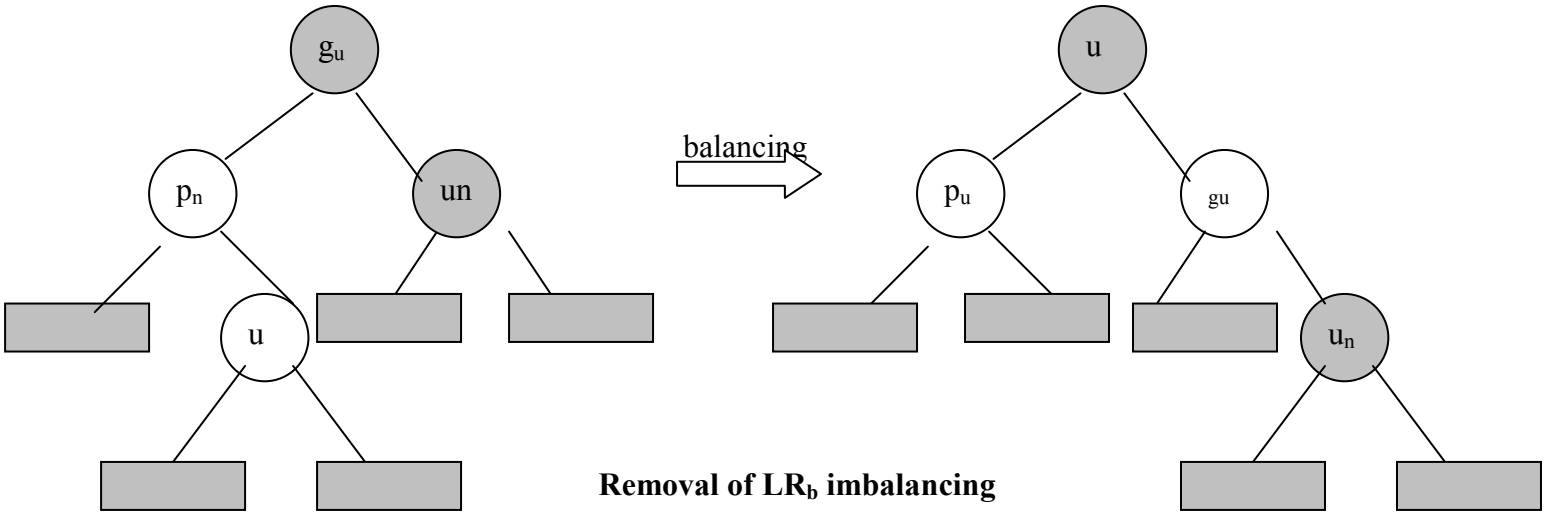
Removal of LL_b imbalancing



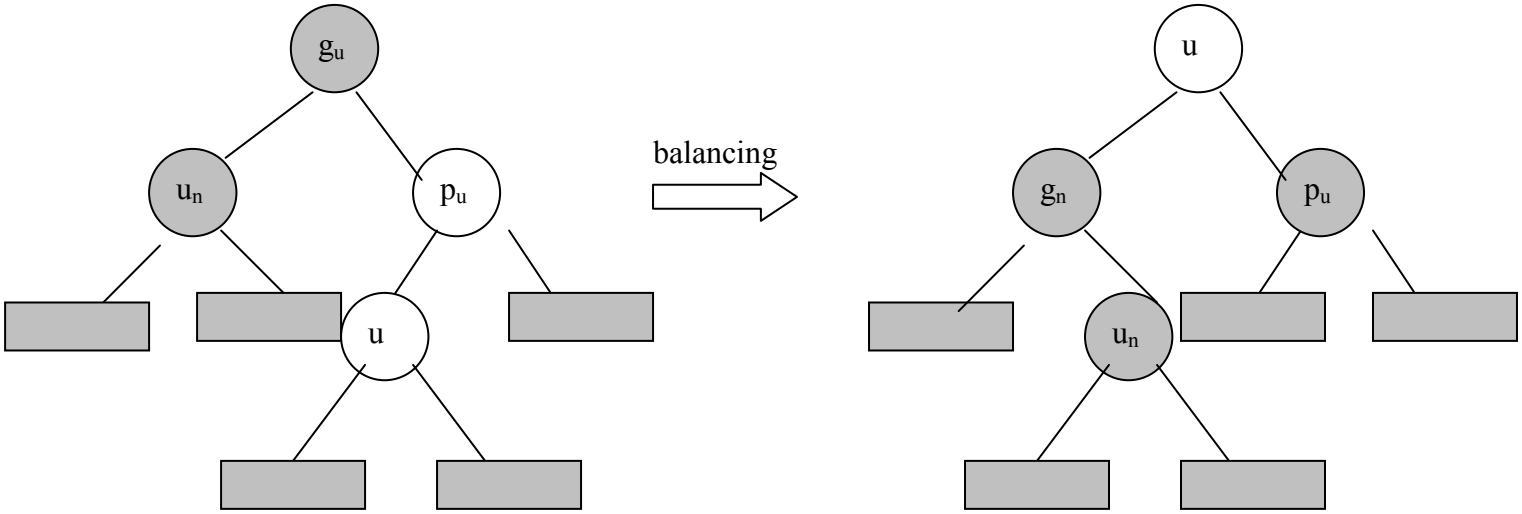
Removal of RR_b imbalancing

Removing LR_b and RL_b imbalances

1. Apply double rotation of u about p_u followed by u about g_u.
2. For LR_b recolor u to black and recolor p_u and g_u to red.
3. For RL_b recolor p_u to black.



Removal of LR_b imbalancing



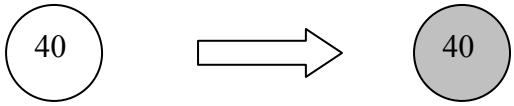
Removal of RL_b imbalancing

Example for insertion of elements in Red-Black tree

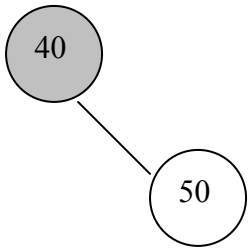
Insert key sequence is as given below.
40, 50, 70, 30, 42, 15, 20, 25, 27, 26, 60, 55

Construct Red-Black tree

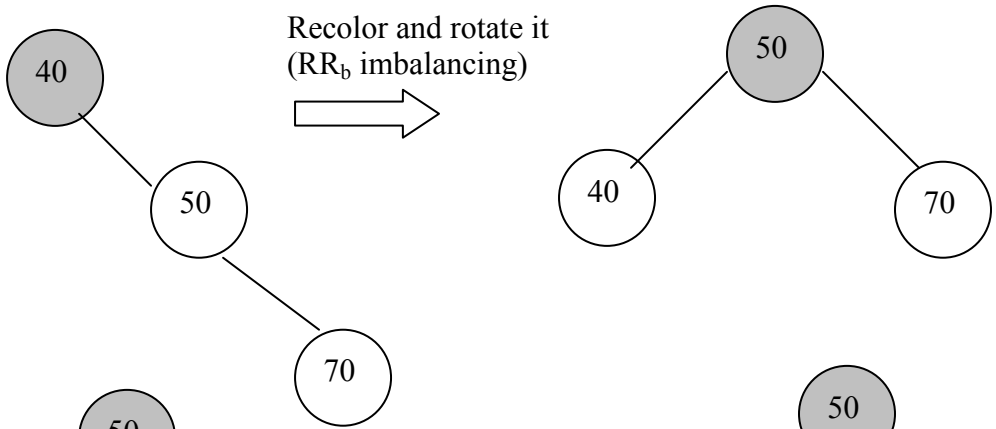
Initially insert node 40 with color red. Recolor this root node to black.



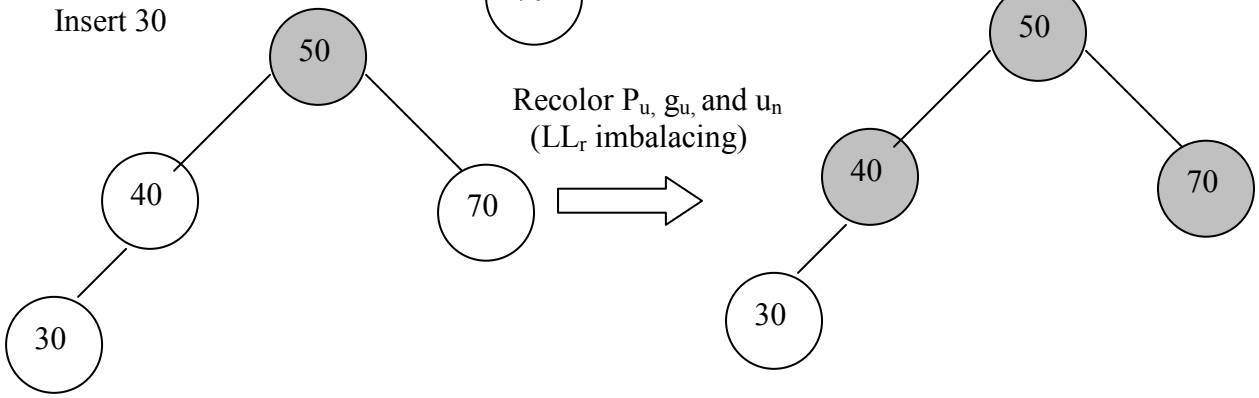
Insert 50



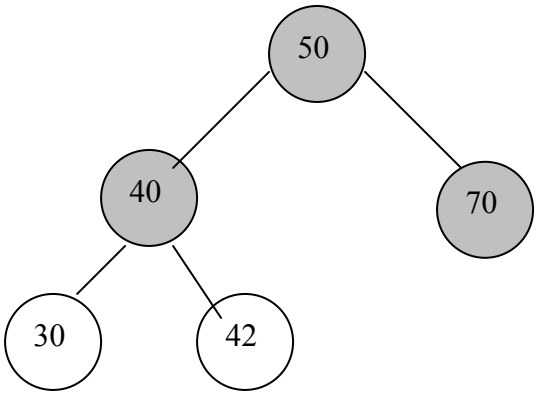
Insert 70



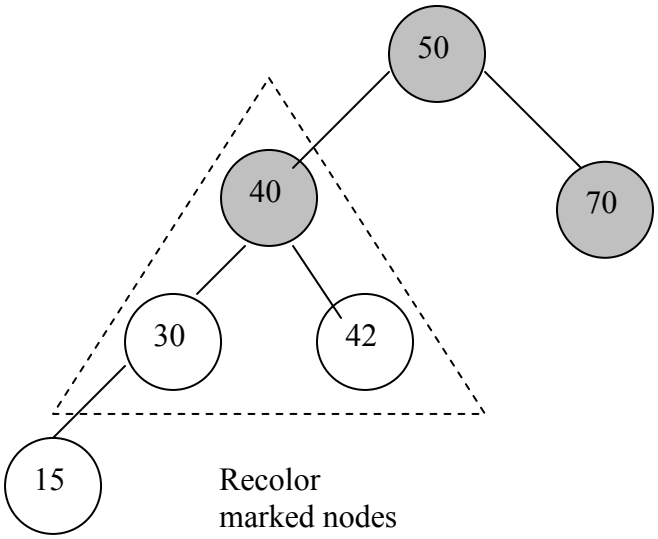
Insert 30



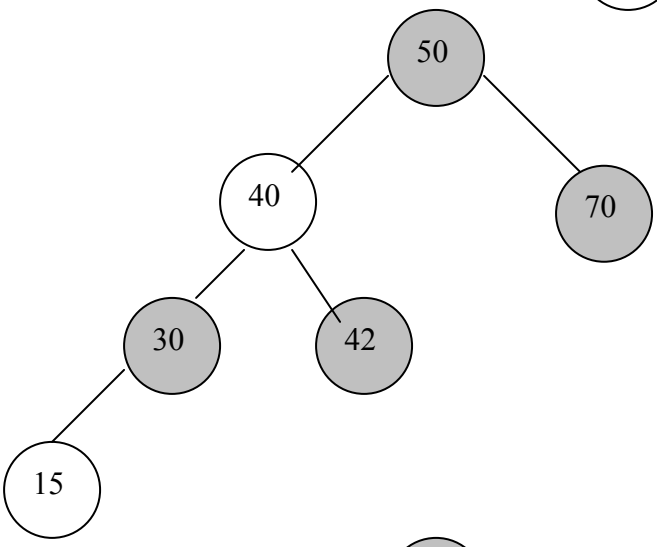
Imsert 42



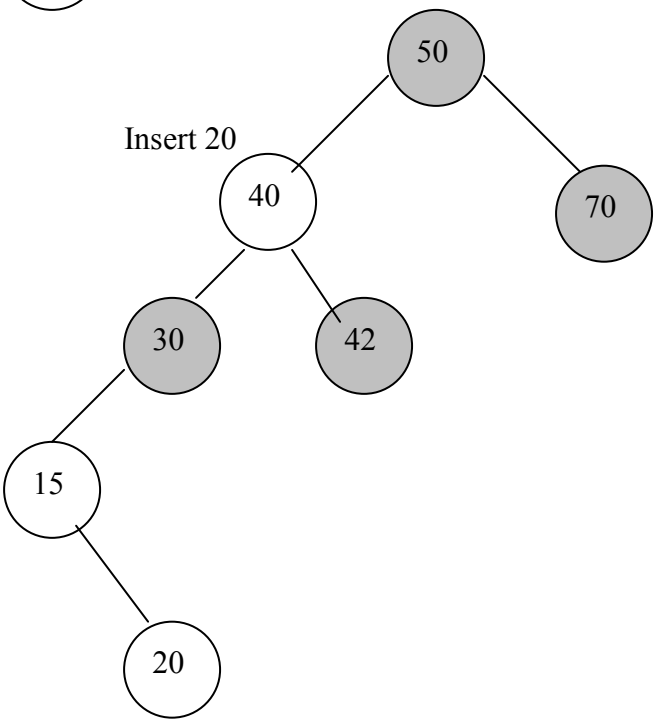
Insert 15



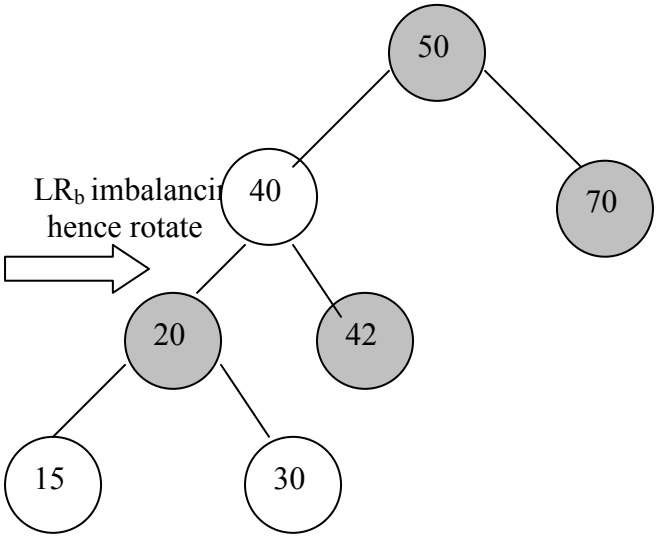
Recolor
marked nodes
(LL_r imbalancing)



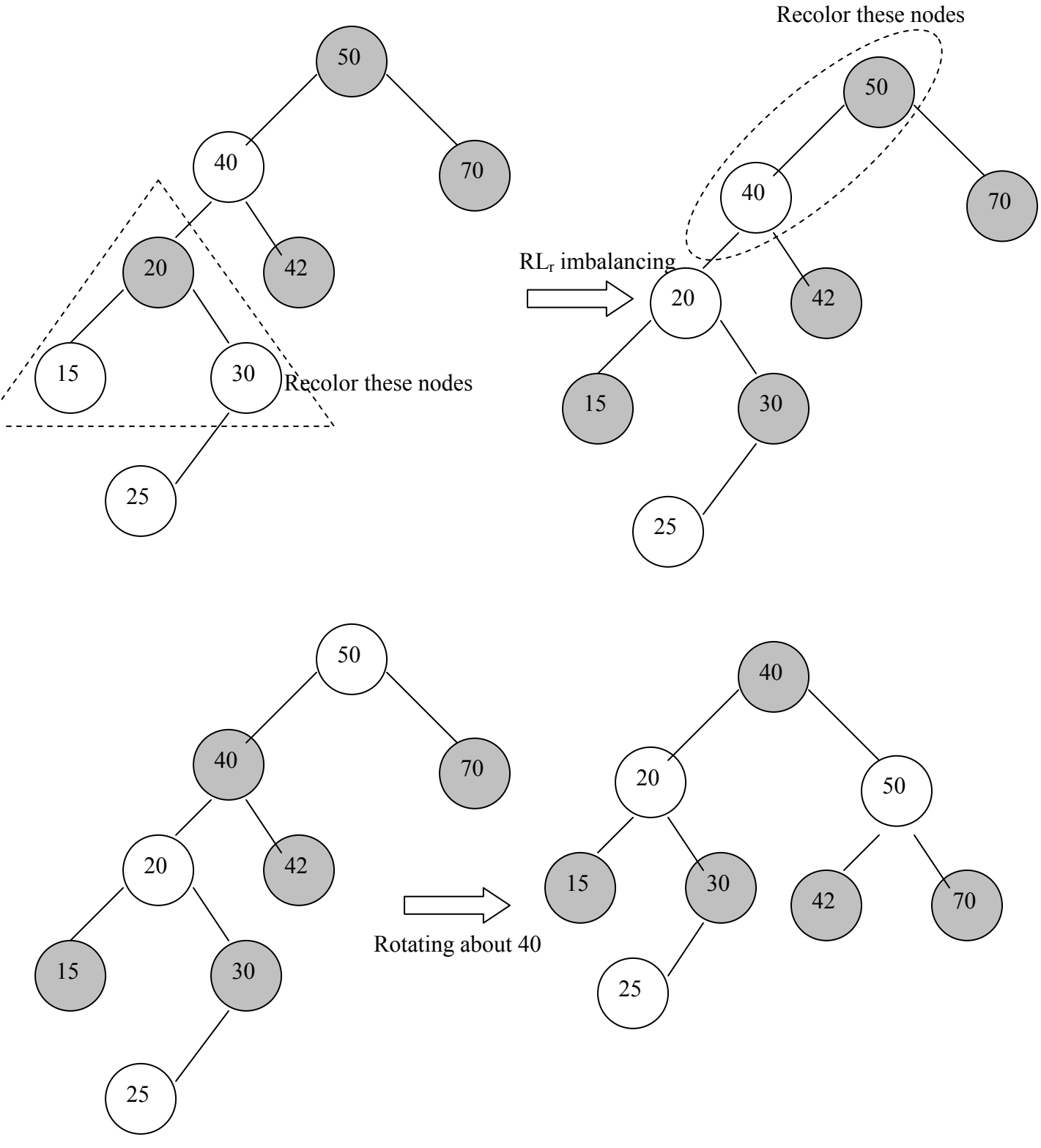
Insert 20



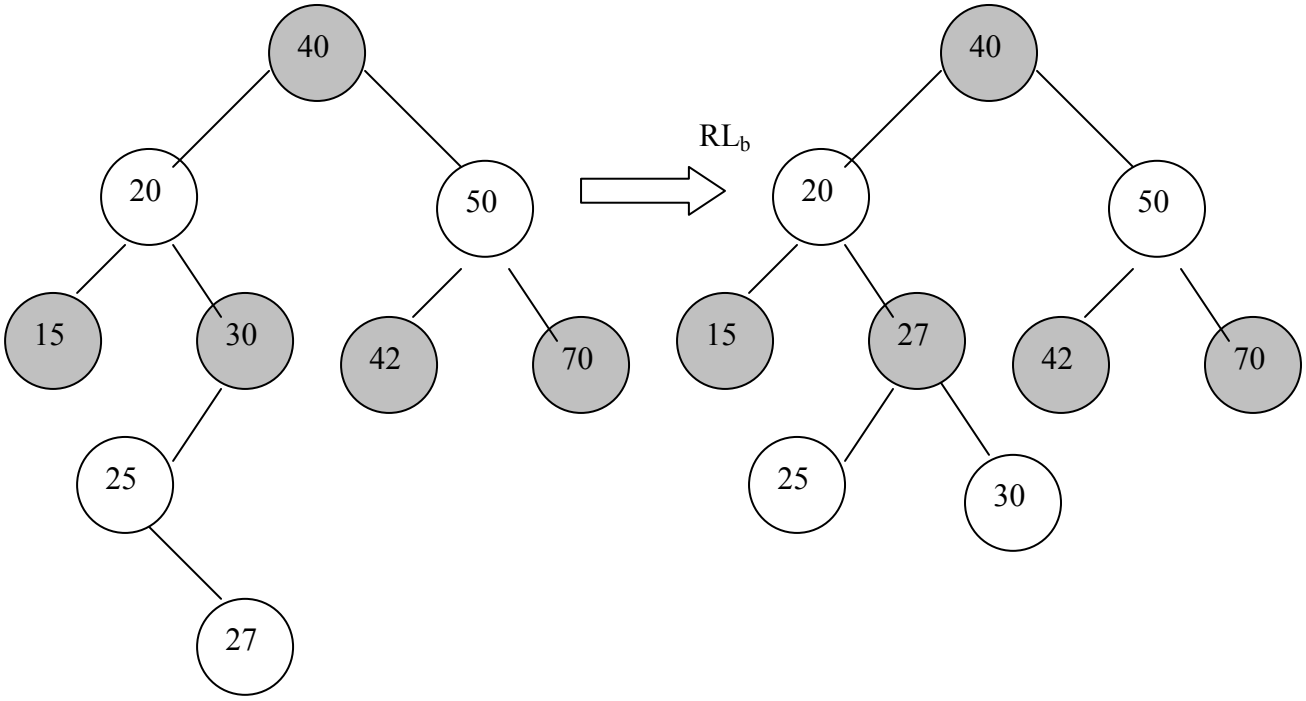
LR_b imbalanci
hence rotate



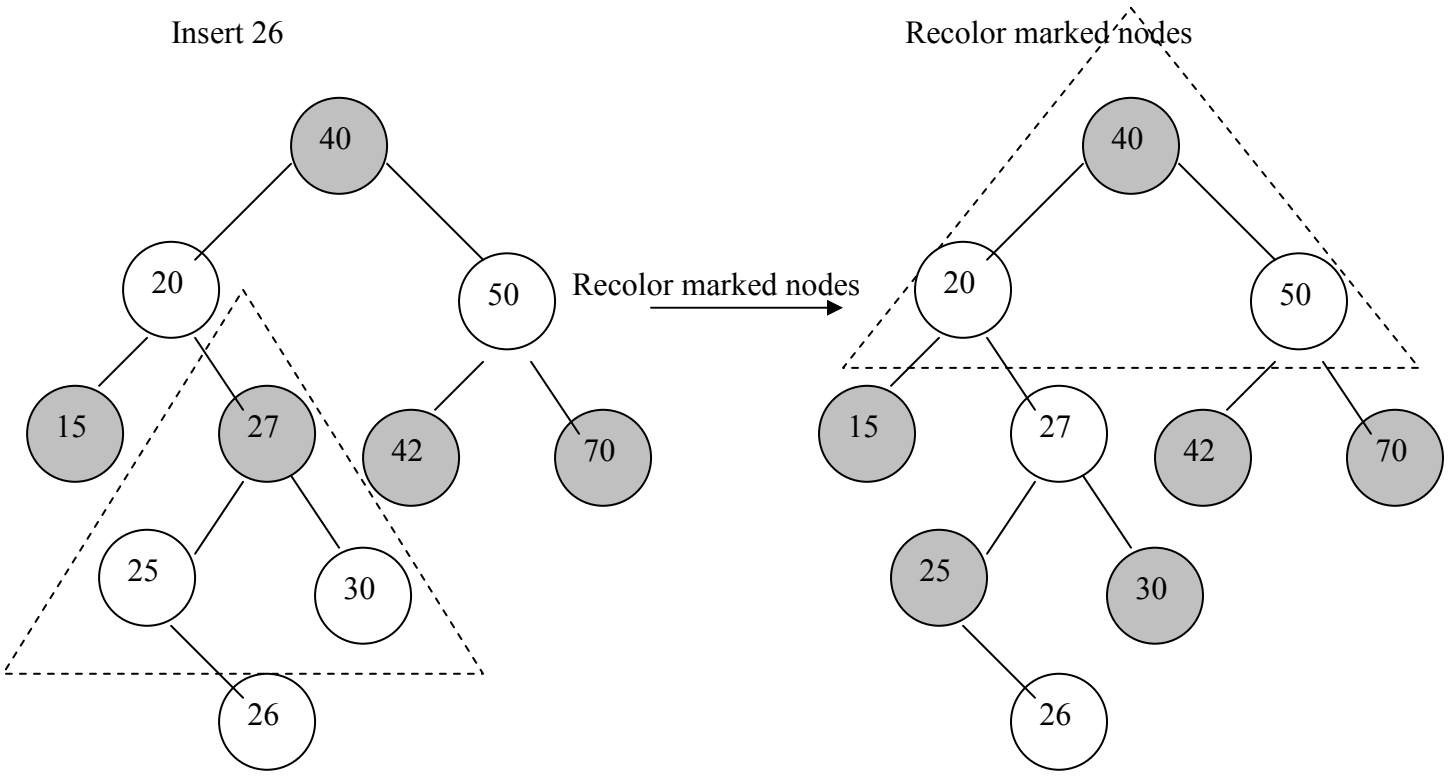
Insert 25

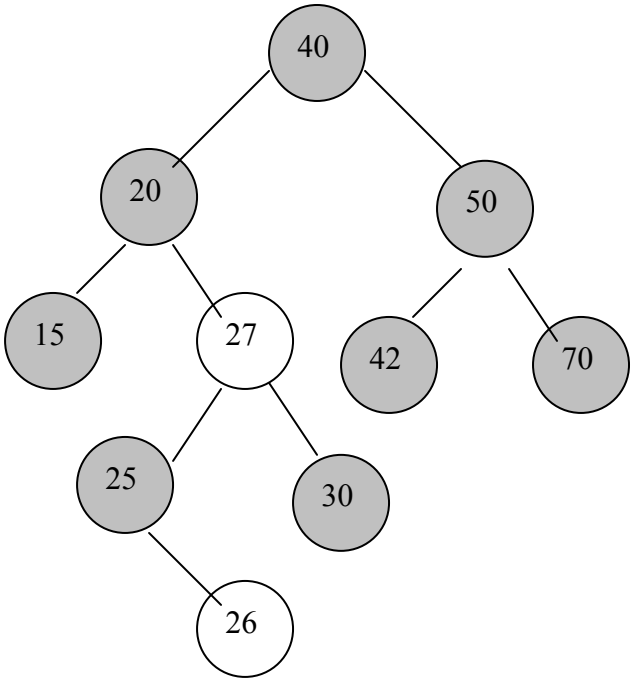

Recolor these nodes
50
40
70
20
42
15
30
25
Recolor these nodes
RL$_r$ imbalancing
50
40
70
20
42
15
30
25
50
40
70
20
42
15
30
25
Rotating about 40
40
20
50
15
30
42
70
25

Insert 27

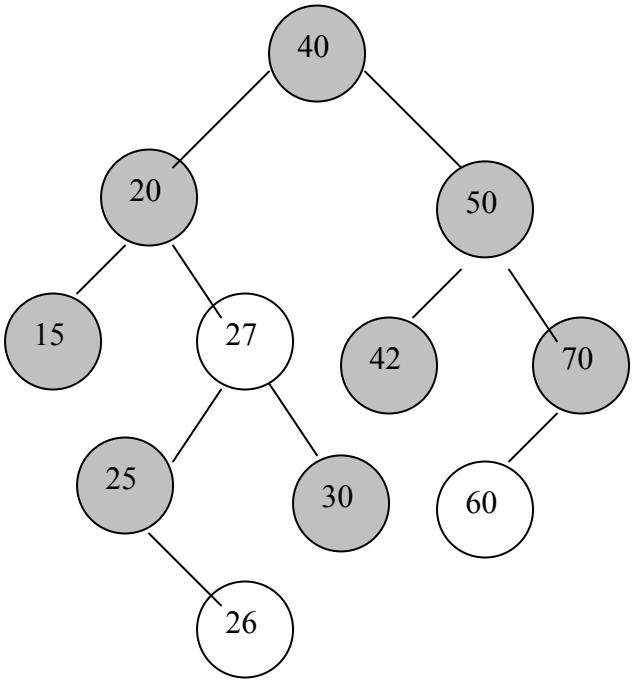


Insert 26

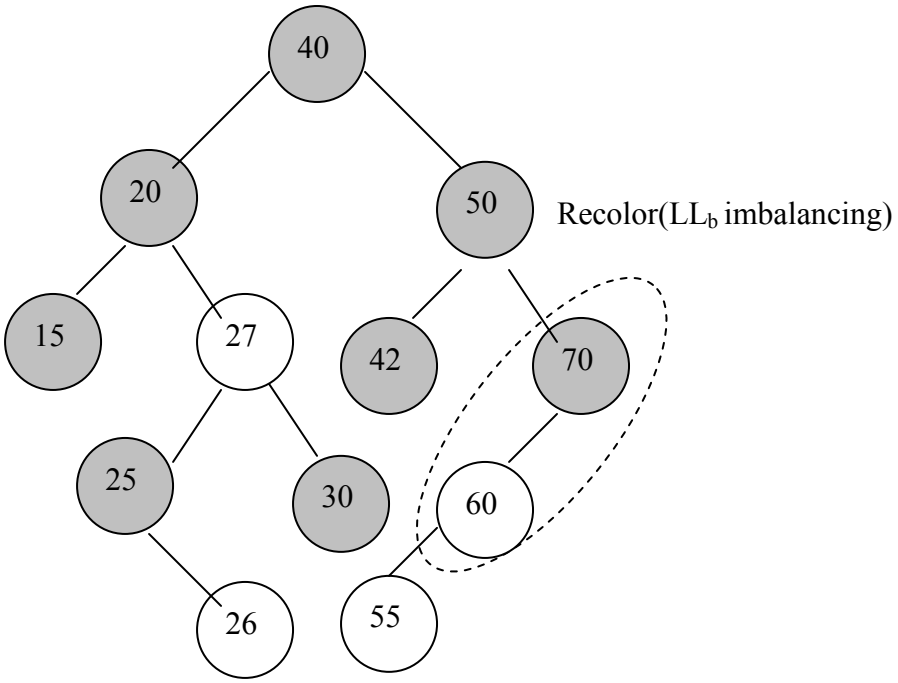




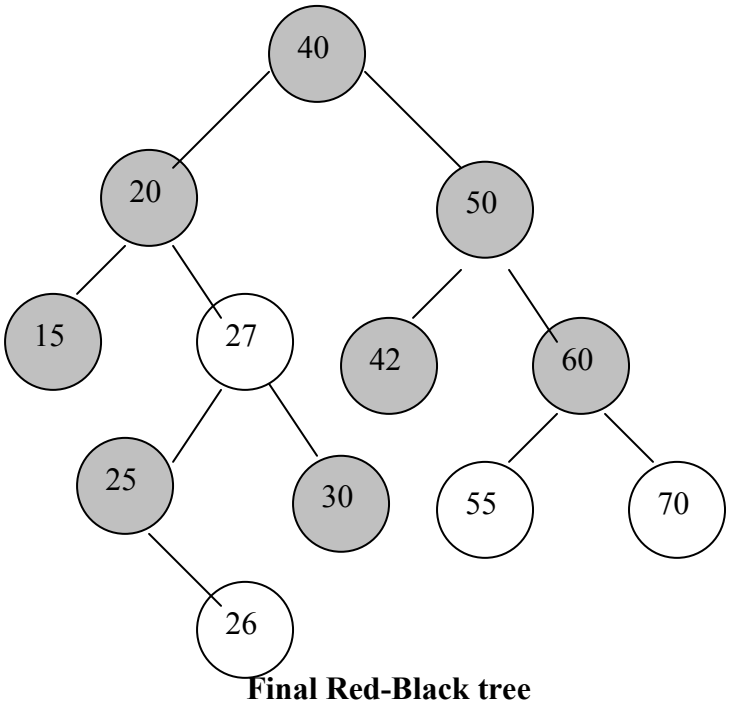
Insert 60



Insert 55



Recolor(LL_b imbalancing)



Final Red-Black tree

Deletion

Deletion of a node from Red-Black tree is just similar to deletion of a node from BST. But color changes and rotations may be required to rebalance the tree after deletion occurs depending upon original structure of Red-Black tree. There are different cases that should be considered while deleting a node 'y'.

Let P_y is a parent of node y and V is its sibling.

1. X_b imbalancing

If V is black then imbalancing of type L_b or R_b. It is also known as X_b imbalancing. Here X ie either 'L' or 'R'.

2. X_r imbalancing

If V node is red then imbalancing is either L_r or R_r. It is then known as X_r imbalancing. Here X is either 'L' or 'R'.

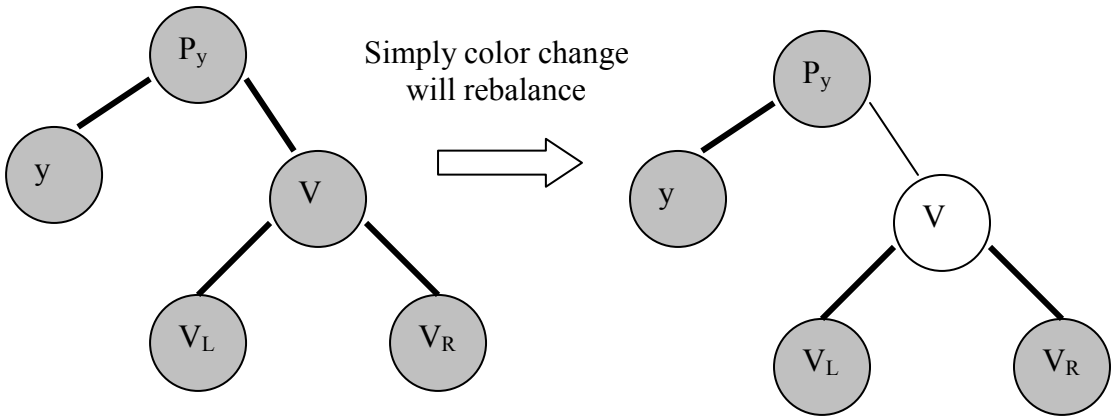
Removing X_b imbalancing:

The X_b imbalancing has three cases, based on number of children of V node.

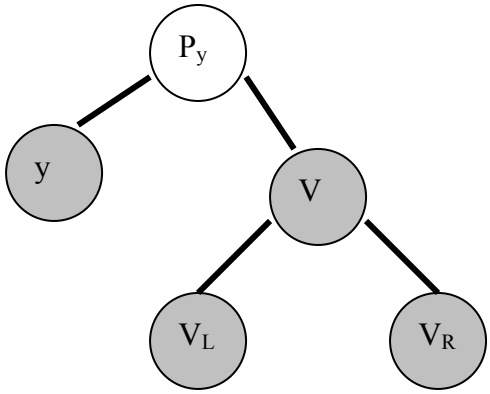
Case 1:

Rb_0 – y is right child of P_y, V is black with no red child.
Lb_0 – y is left child of P_y, V is black with no red child to it.

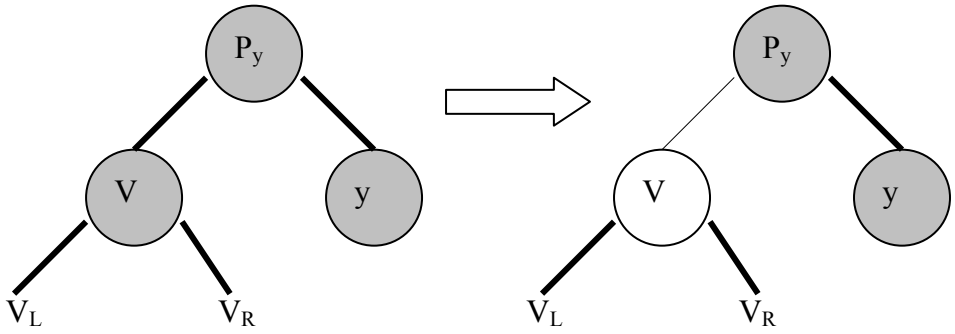
Think pointers indicate that successor node is black and thin pointers indicate that node following it is red.



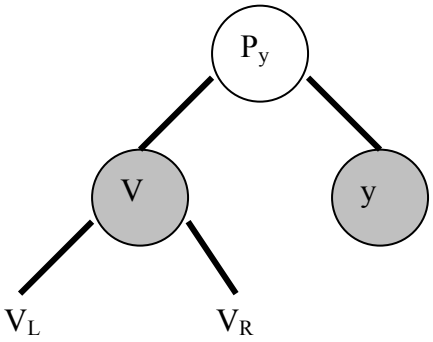
OR



Removal of Lb_0 rebalancing



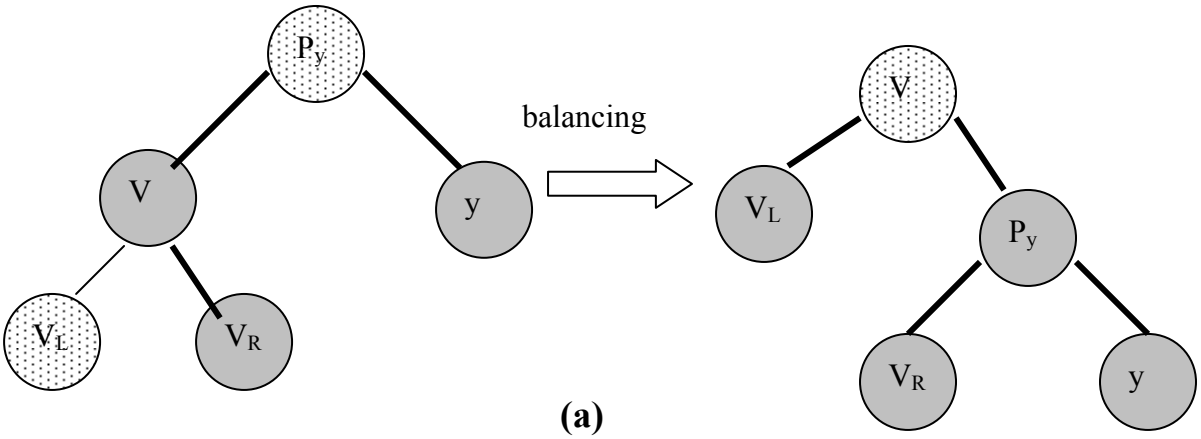
OR



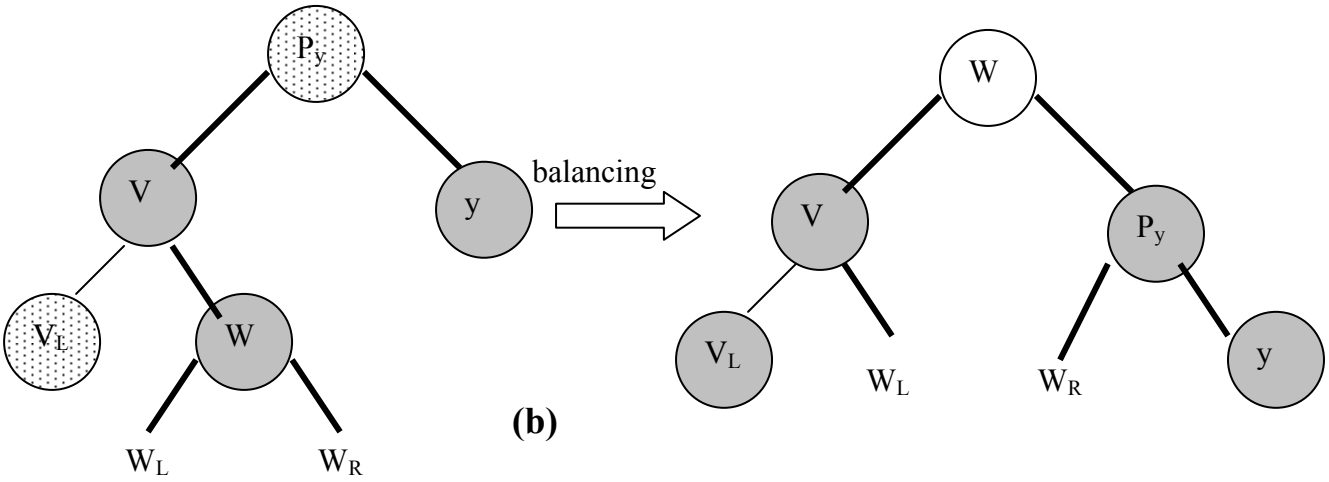
Removal of Rb_0 rebalancing

Case 2:

Rb_1 – y is right child of P_y, V is black node having one red child.
Lb_1– y is left child of P_y, V is black node having one red child.
The dotted node indicates that node may be red or black and is unchanged by rotation.

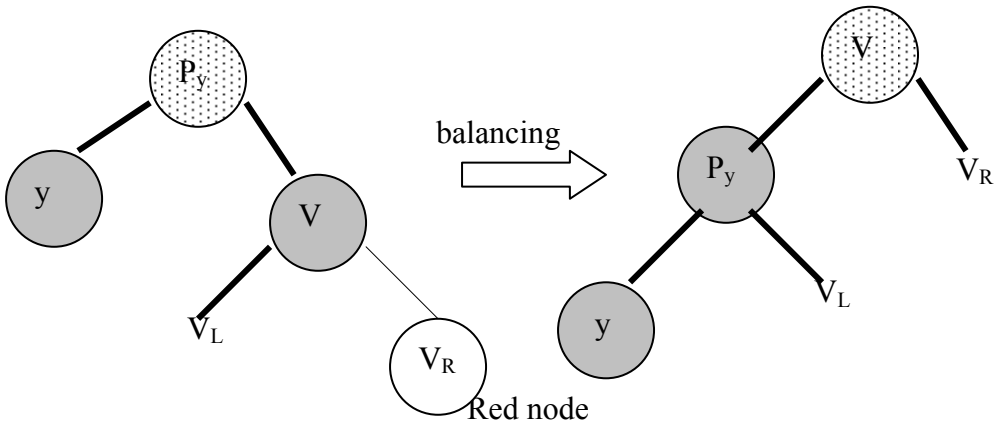


(a)

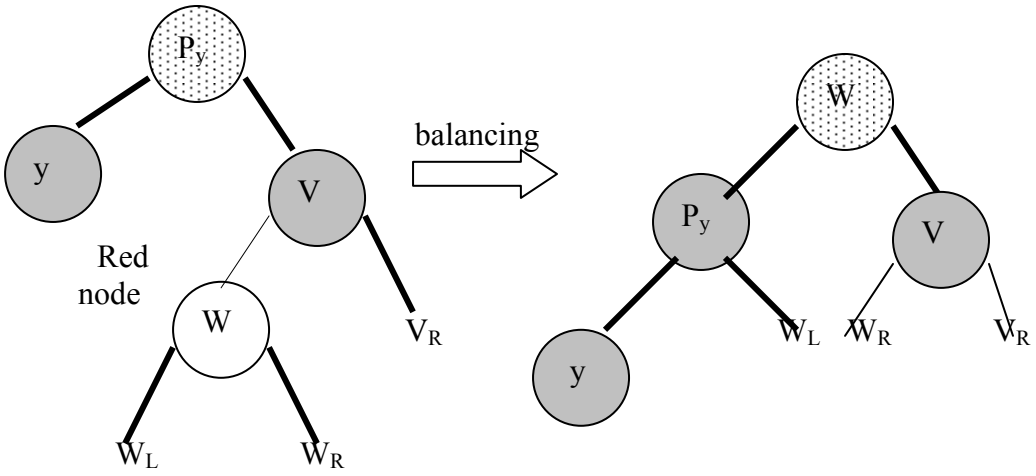


(b)

Removal of Rb_1 imbalancing



(a)

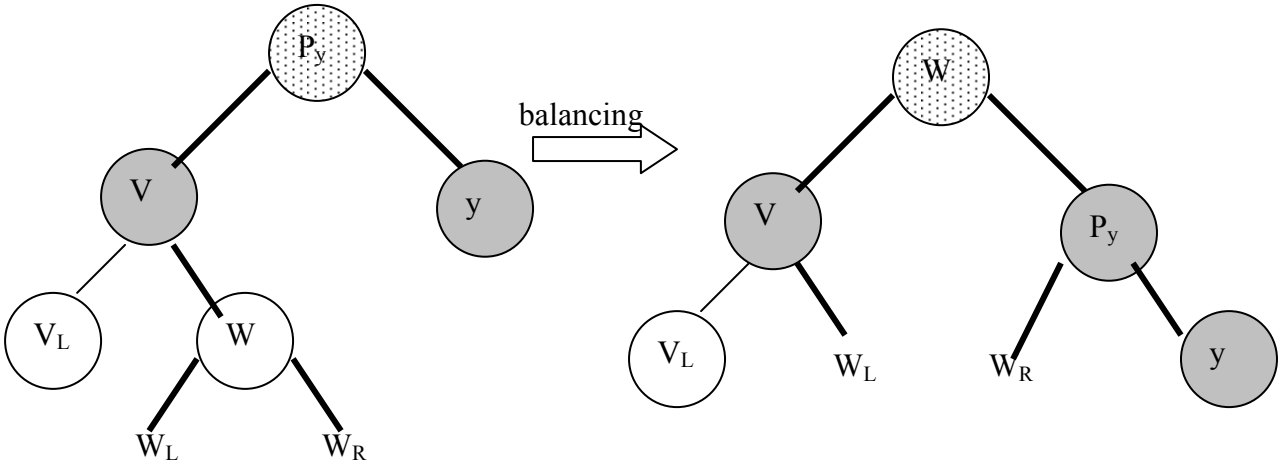


(b)

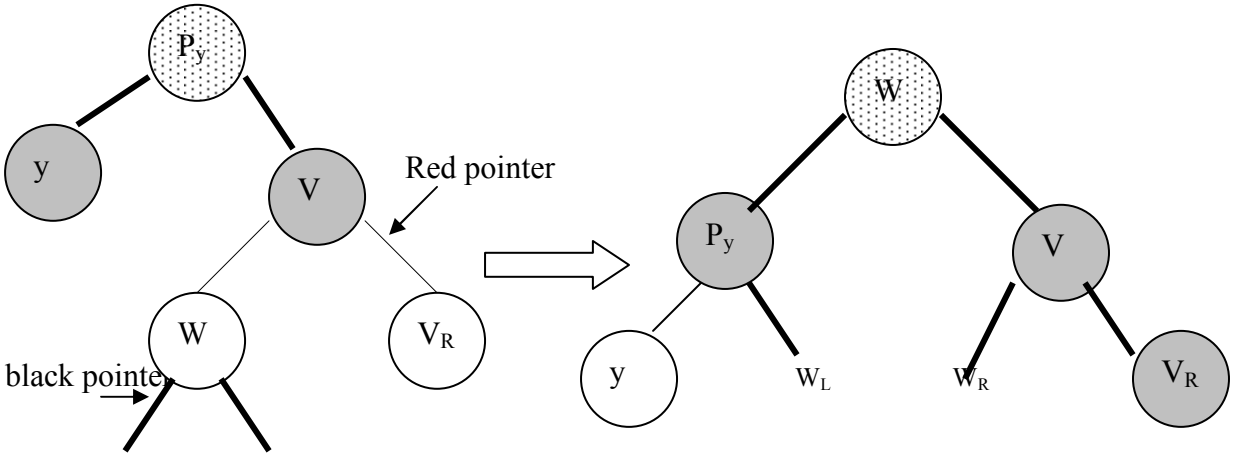
Removal of Lb_1 imbalancing

Case 3:

Rb_2 – y is child of P_y, V is black node with two red children.
Lb_2 – y is left child of P_y, V is black node with two red children.



Removal of Rb_2 imbalancing



Removal of Lb_2 imbalancing

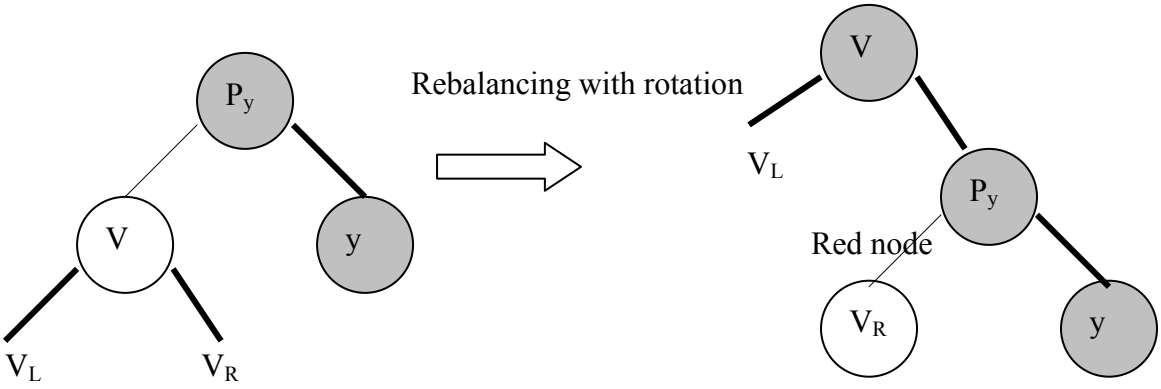
Removing X_r imbalancing:

In X_r imbalancing again there are three cases. The V node is a red node may be having no red children/one red child/two red children. The three cases are as follows:

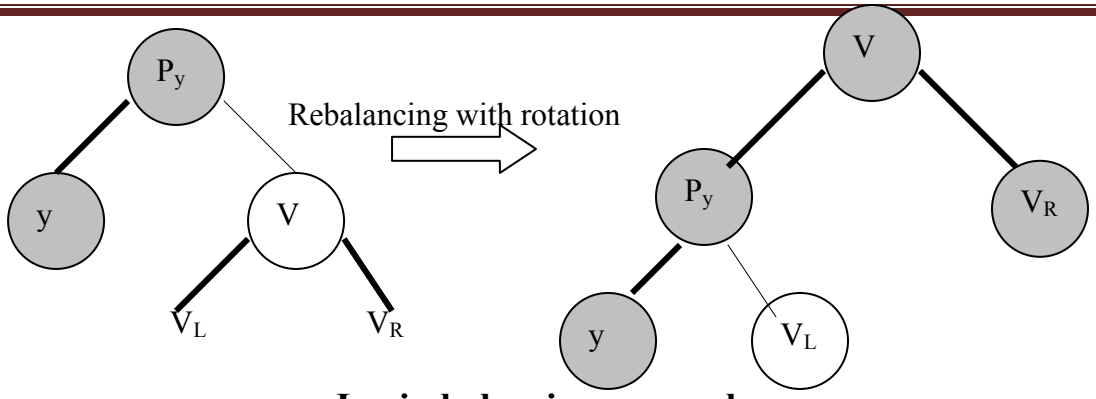
Case 1:

Rr0 – y is right child of P_y, V is red node with no red child.

Lr0 – y is left child of P_y, V node is red with no red child.



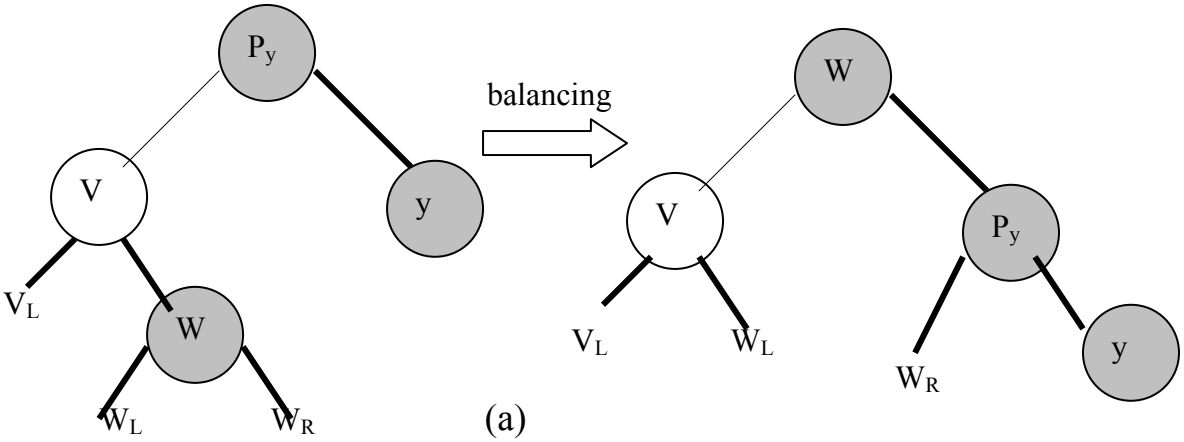
Rr0 imbalancing removal



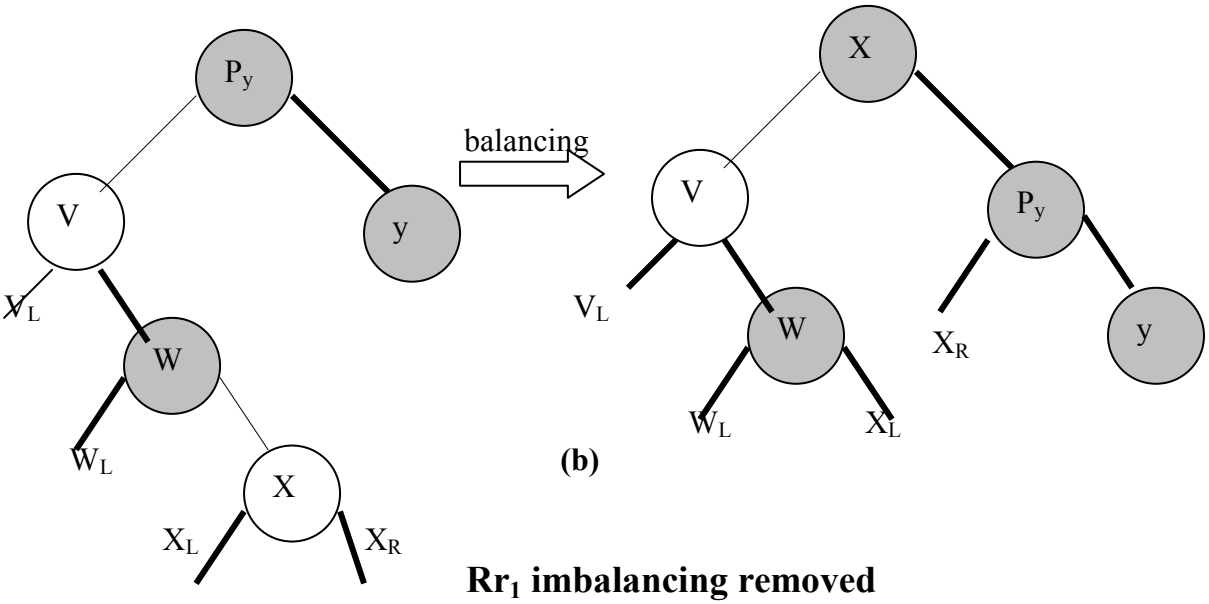
Lr_0 imbalancing removal

Case 2 and 3:

Rr_1 - y is a right child of P_y, V is red node with one red child.
Lr_1 – y is left child of P_y, V is red node with one red child.
Rr_2 – y is right child of P_y, V is red node with two red children.
Lr_2 – y is left child of P_y, V is red node with two red children.

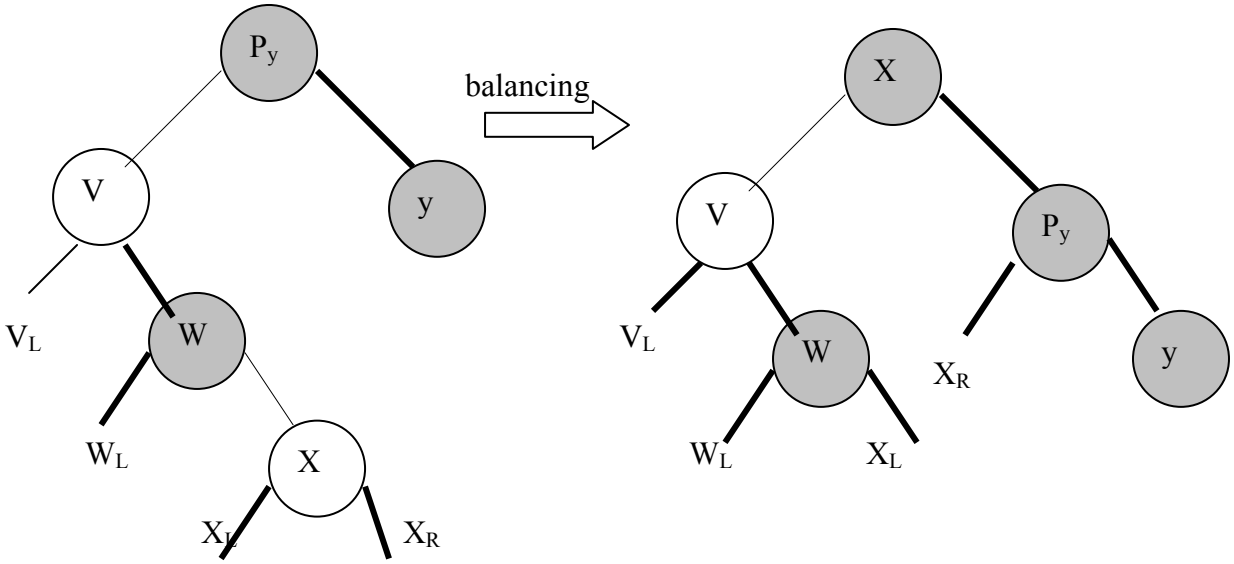


(a)

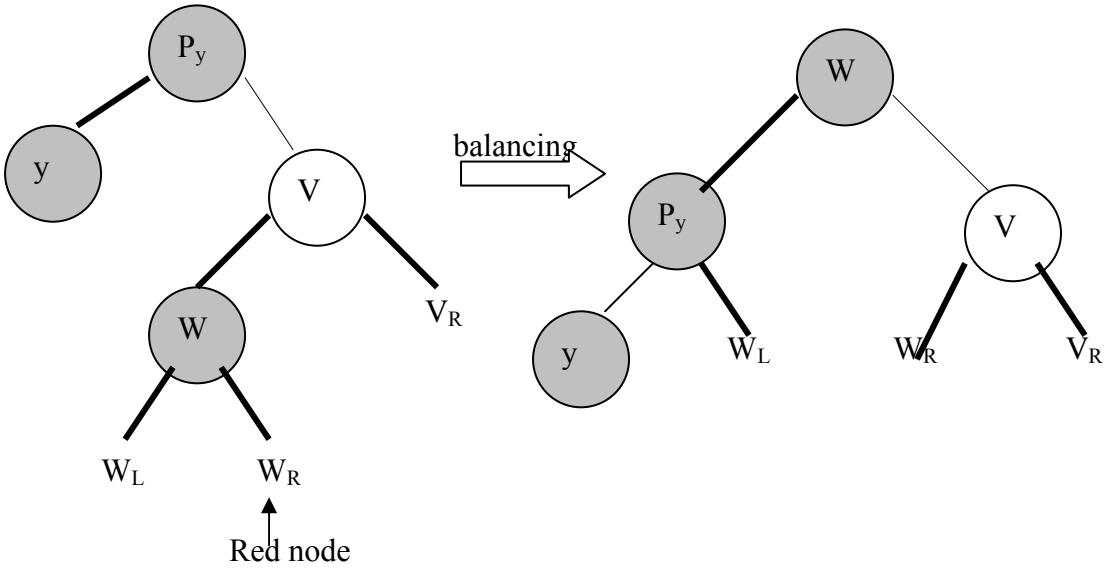


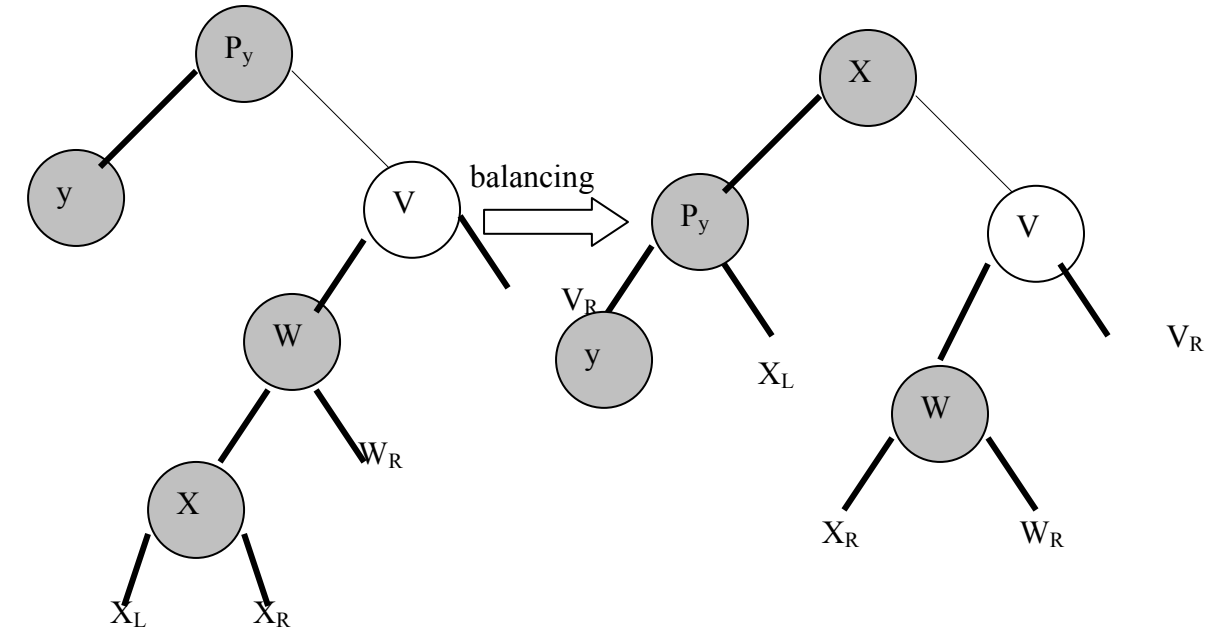
(b)

Rr_1 imbalancing removed

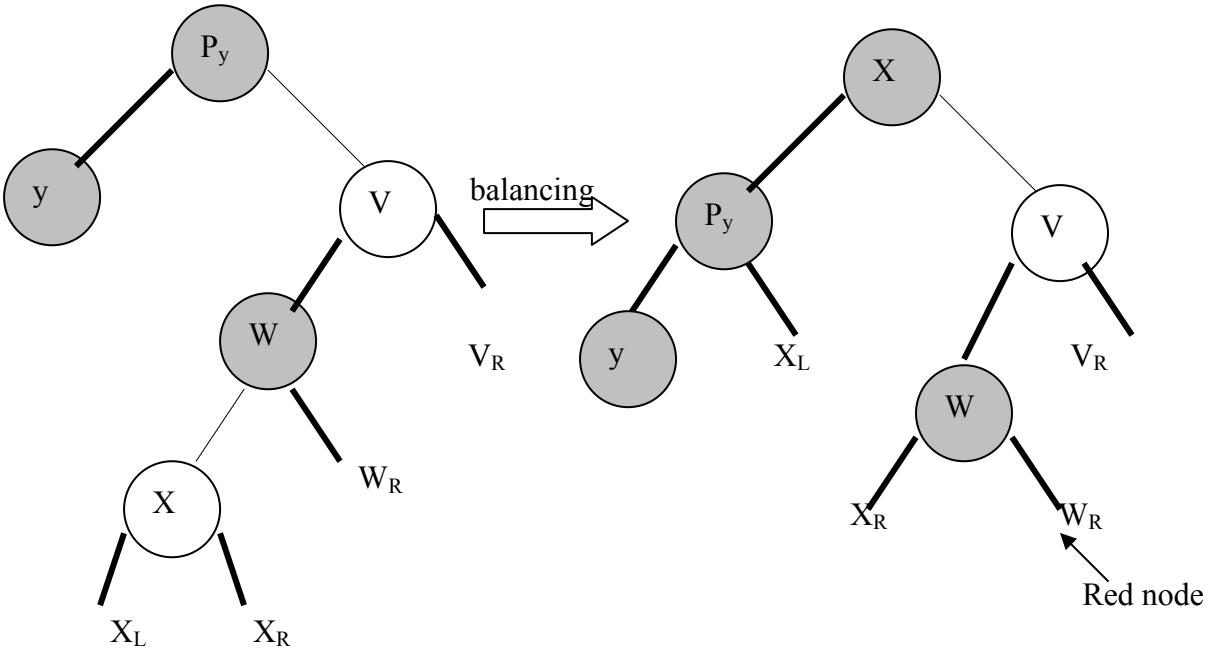


Rebalancing Rr_2 imbalancing





Rebalancing Lr_1 imbalancing



Rebalancing Lr_2 imbalancing

Searching

Searching a desired node in Red-Black tree is exactly similar to searching a node in binary search tree. The time complexity for searching is O(log n).

SPLAY TREES

A splay tree is a self balancing binary search tree with no explicit balance condition. The splay tree has a property that recently accessed elements can be accessed quickly.

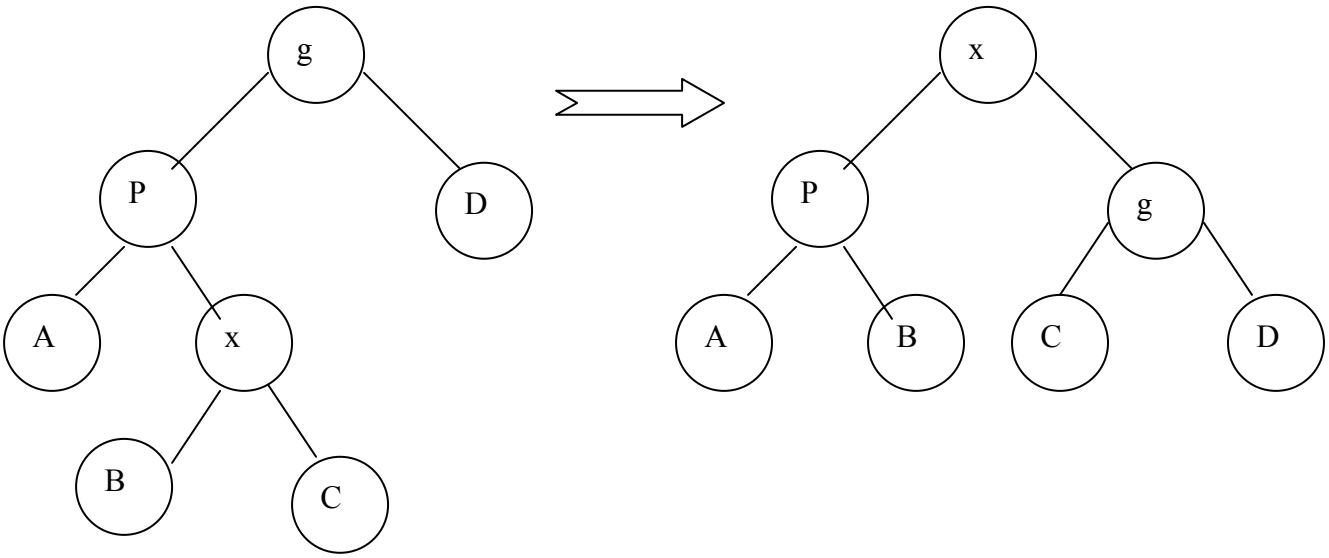
All normal operations that are performed on binary search tree are performed on splay tree. But there is a special operation called splaying is performed on splay tree.

Splaying means arranging the elements of a tree in such a way that the most recently accessed node will be placed as root of the tree. Various cases that arise are

1. Zig-zag step
2. Zig-zig step
3. Zig step

These are basically rotations applied for the node x. Zig means left and zag means right. Let 'P' be the parent node of 'x' and 'g' be the grand parent of x.

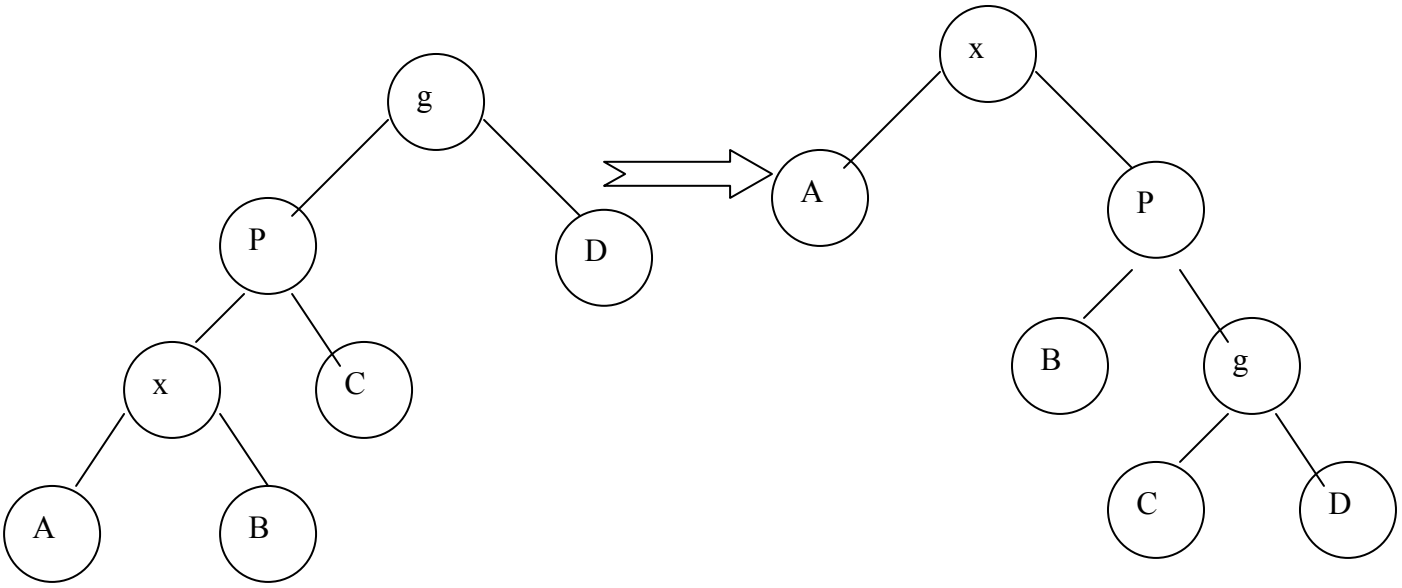
1. Zig-zag case (LR step)



When x is a right child of node P and P is a left child of node g. Then the rearrangement is made for most recently accessed node x.

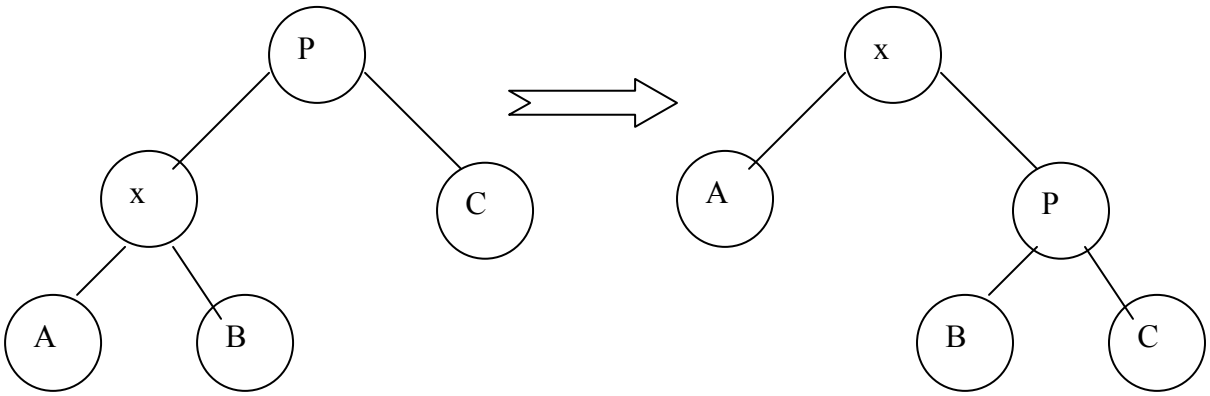
The x becomes root, P becomes its left child and g becomes the right child. The Zig-zag step is equivalent to rotation about x.

2. Zig-zig case (LL step)



The P node becomes right child of x and g becomes right child of P, where elements get rearranged in zig-zig step.

3. Zig step(L step)



When x is attached as a left child of P node then the rearrangement that is needed is of zig type. In such a rearrangement P becomes right child of x node.

Splay Operations

Various operations supported by splay tree are

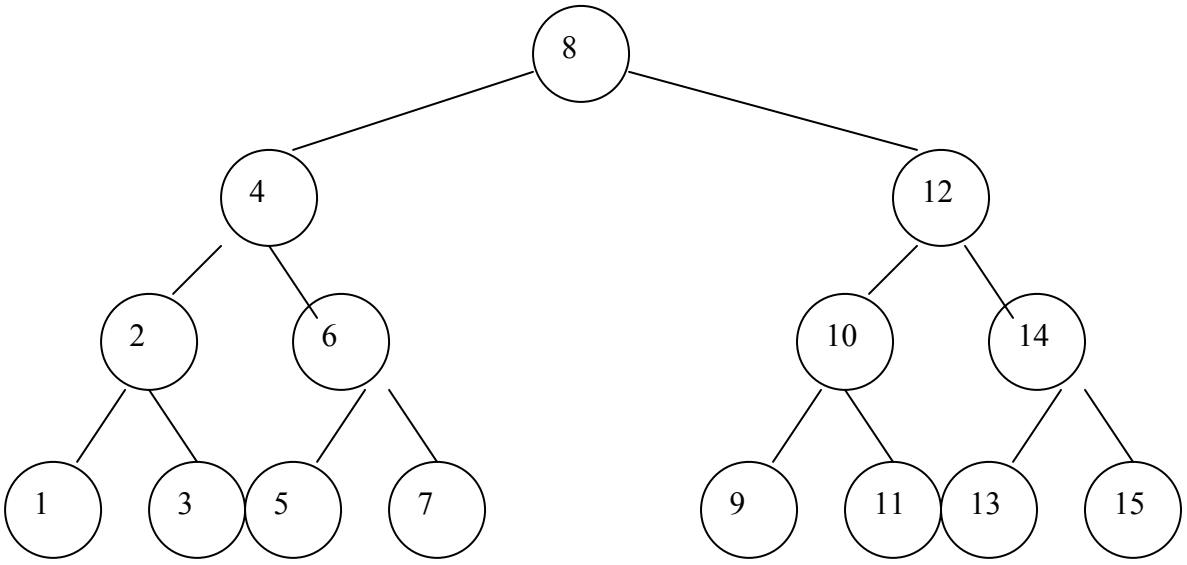
1. Splay
2. Find
3. Insert
4. Delete

The splay operation starts at splay node. The splay node is basically root of the binary search tree.

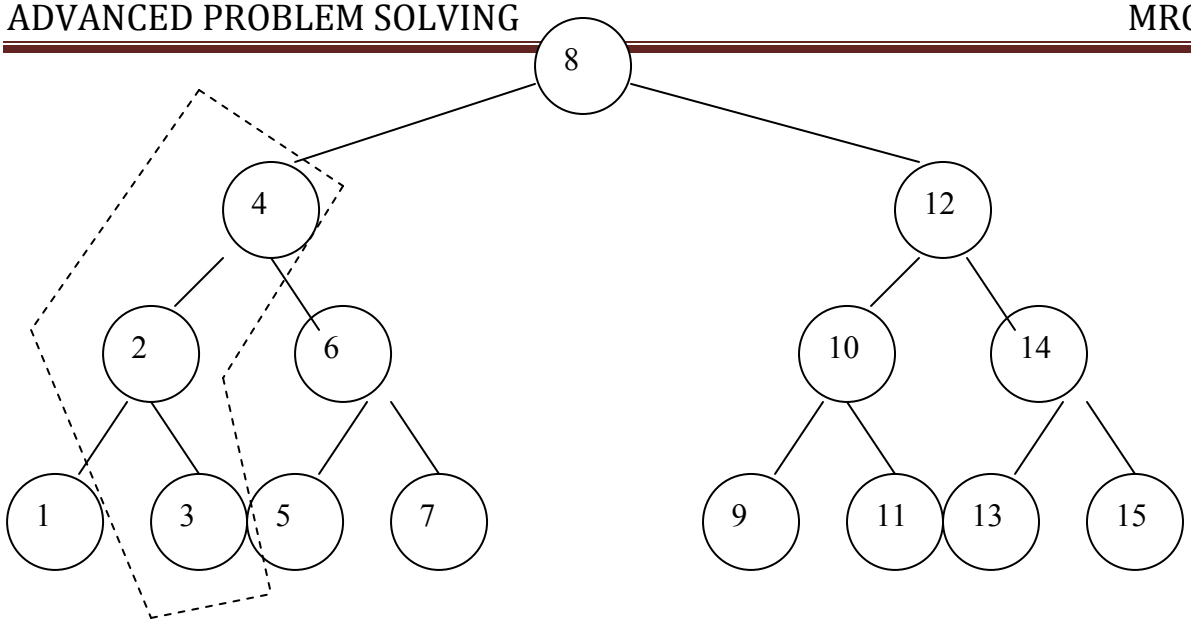
During splay operation the splay node that is been selected is at the deepest level. To move this node at root the sequence of splay steps are performed. When splay node reaches to root position the sequence of splay step is empty. The splay steps involve various rotations such as zig-zag, zig, zig-zig.

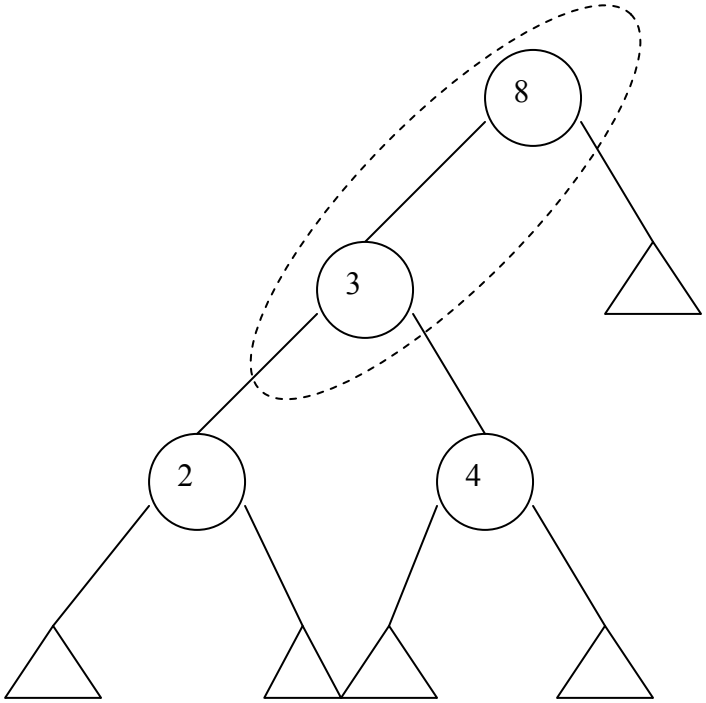
For example:

Consider a tree as given below:

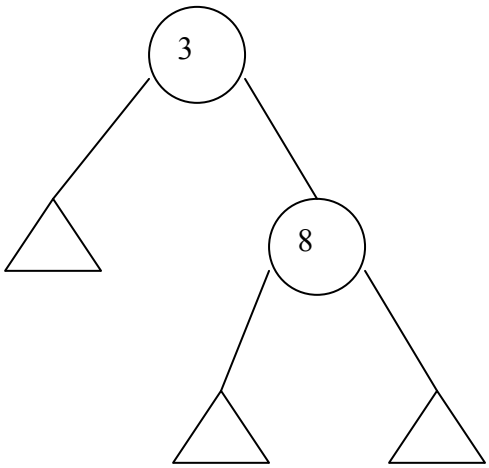


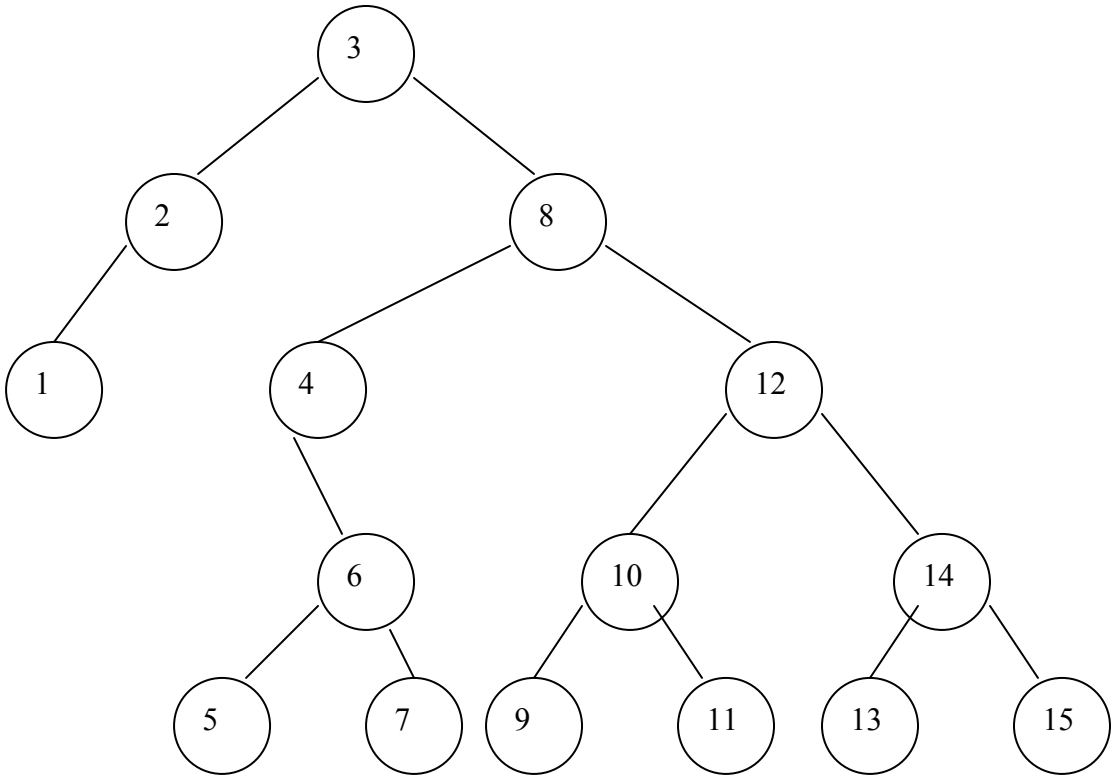
Now if we want to find node '3'. Then focus of splay operation consists of node being splayed its parent and its grand parent.
The node involved in splay operation indicate zig-zag or LR rotation.
After performing this splay operation the tree becomes-
The marked nodes direct the zig or L rotation. Hence the splay tree becomes



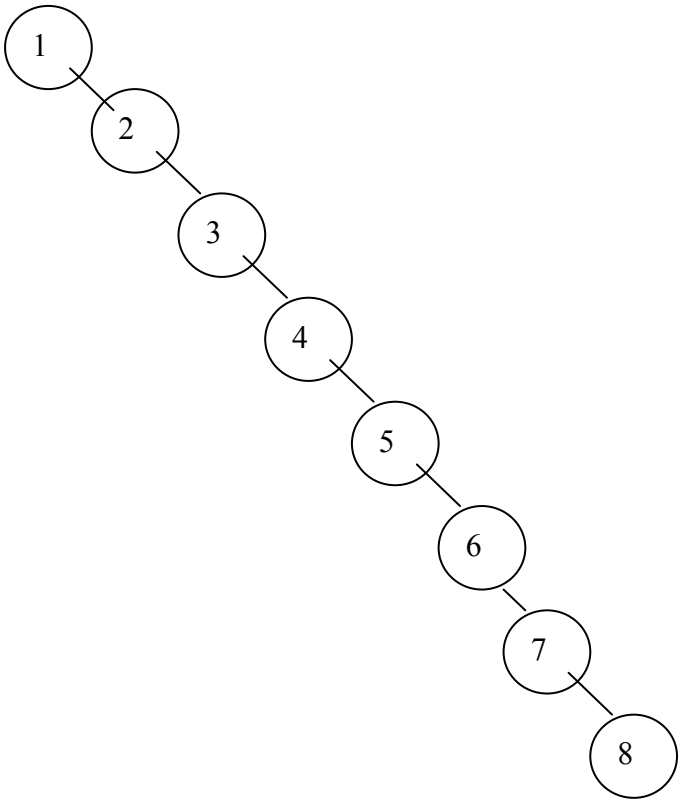


The tree is

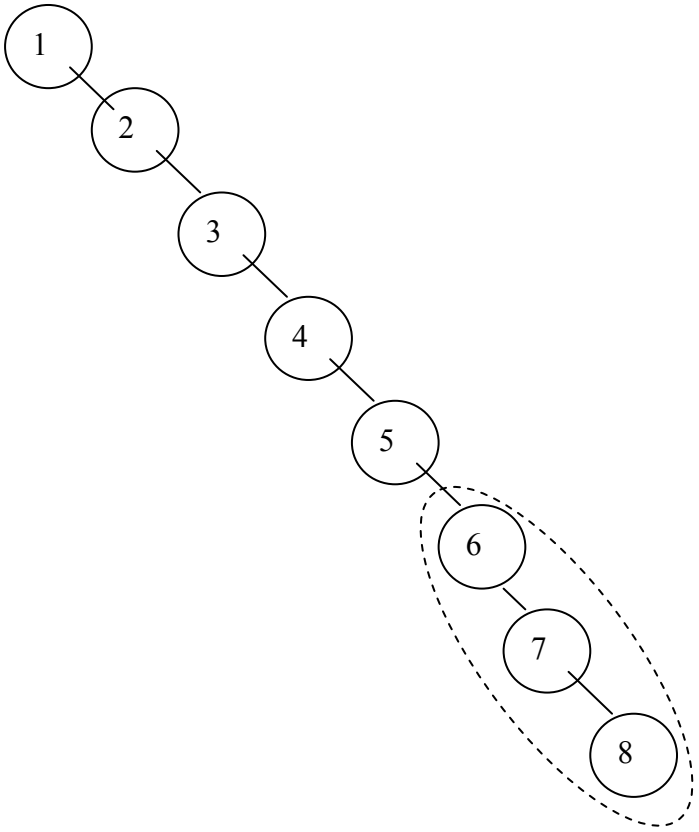




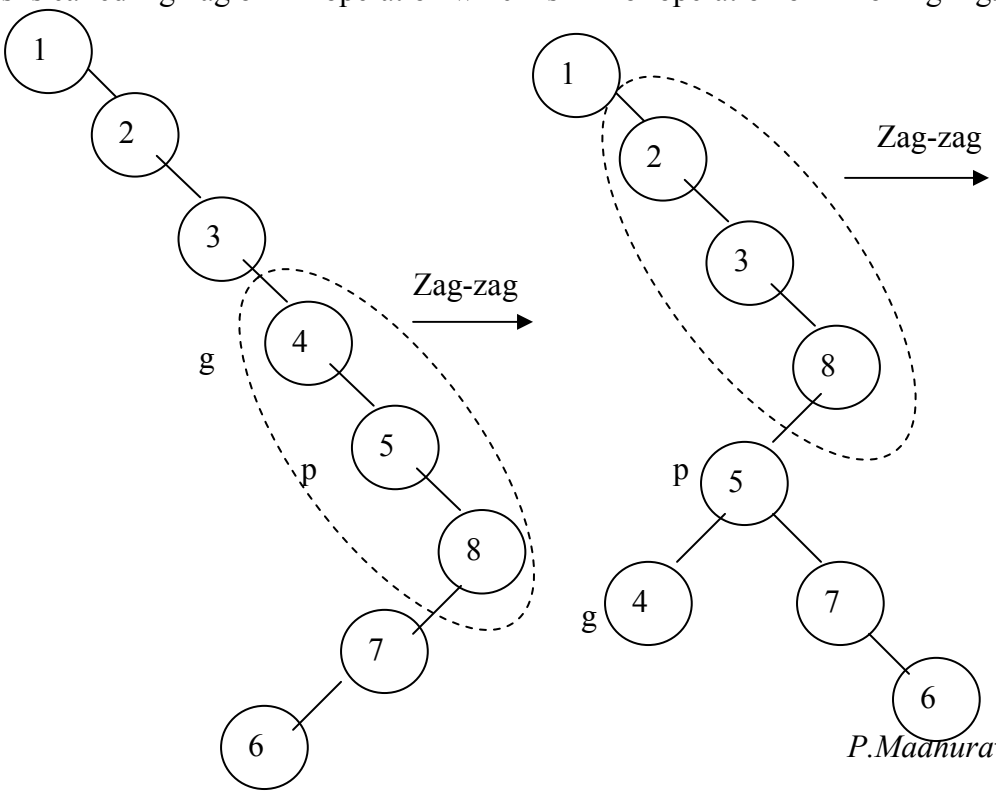
Consider another example-

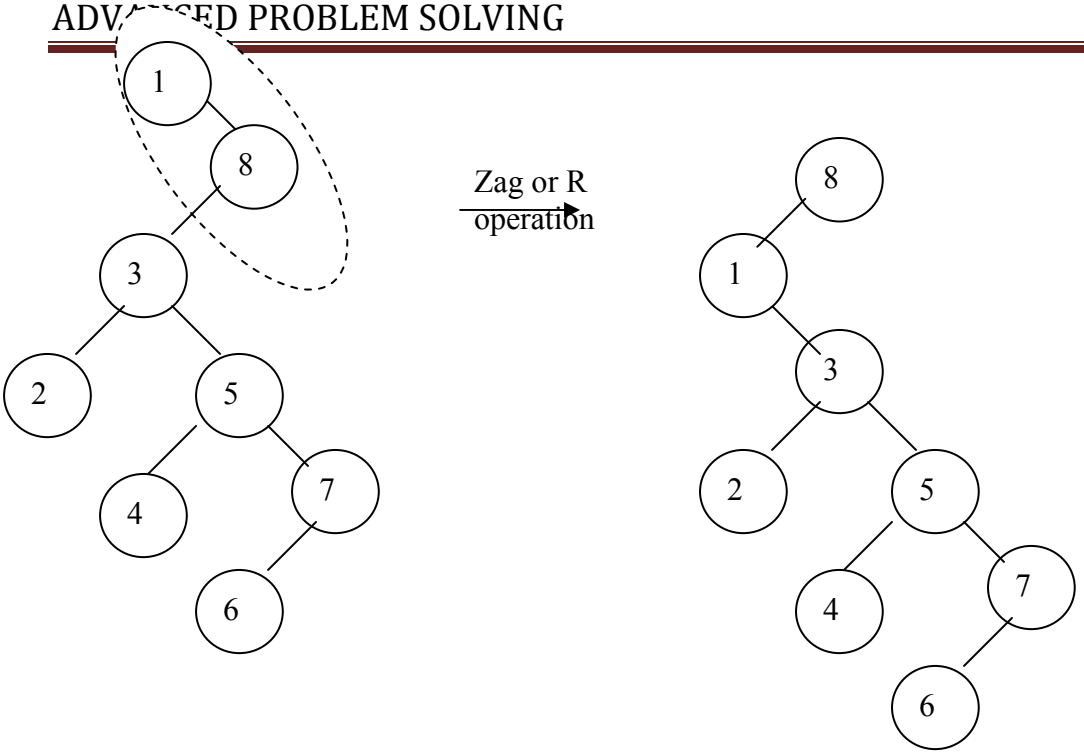


Now if splay node is 8 then the sequence of operations performed are –



This is called zig-zag or RR operation which is mirror operation of LL or zig-zig.





Amortized complexity –

The time required to perform a sequence of (related) operations is averaged over all the operations performed, such time is called amortized running time. Amortized complexity guarantees the average performance of each operation in the worst case.

Theorem: The amortized cost of a splay operation on node x is O(log n)

Proof: The amortized cost of a splay operation on node x is basically the sum of amortized cost of splay steps performed on x, which involves

$$\text{amortized cost} = \sum_i \text{cost}(\text{splay} = \text{step}_i)$$

$$\leq \sum (3(r^{i+1}(x) - r^i(x)) + 1$$

$$= 3(r(\text{root}) - r(x)) + 1$$

The 1 is added from last r or 1 splay step (if necessary). If we set w(x) = 1 for all nodes in tree, then r(root) = log n

We will have

$$\text{amortized cost} \leq 3(\log n) + 1$$

$$= O(\log n)$$

Hence all the splay operations find, insert, delete have amortized complexity as O(log n).