95-00

33

DRP

Binary & Other trees

In this chapter we study two basic varieties:

* General trees (or simply trees)
* Binary trees.

Two applications of trees are developed.

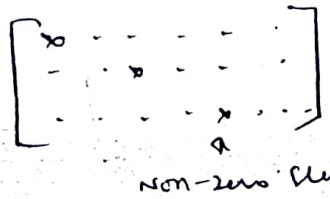
* placement of signal boosters in a tree distribution network.
* online equivalence problem. This problem is also known as the union/find problem.

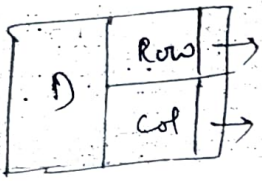
This chapter covers the following topics:

1. Tree & binary tree terminology such as height, depth, level, root, leaf, child, parent & sibling.
2. Formula/array based & linked representations of binary trees.
3. The four common ways to traverse a binary tree:
   - → preorder
   - → inorder
   - → postorder &
   - → level order.

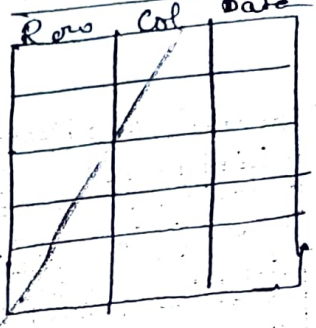
Trees: Trees are non-linear data structures. The data structures we have discussed so far (linked lists, Stacks, Queues) are generally not suitable for the representation of hierarchical data.

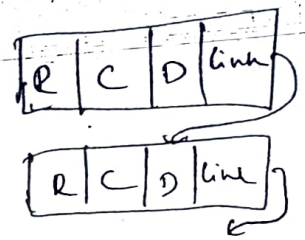
Sparse Matrix

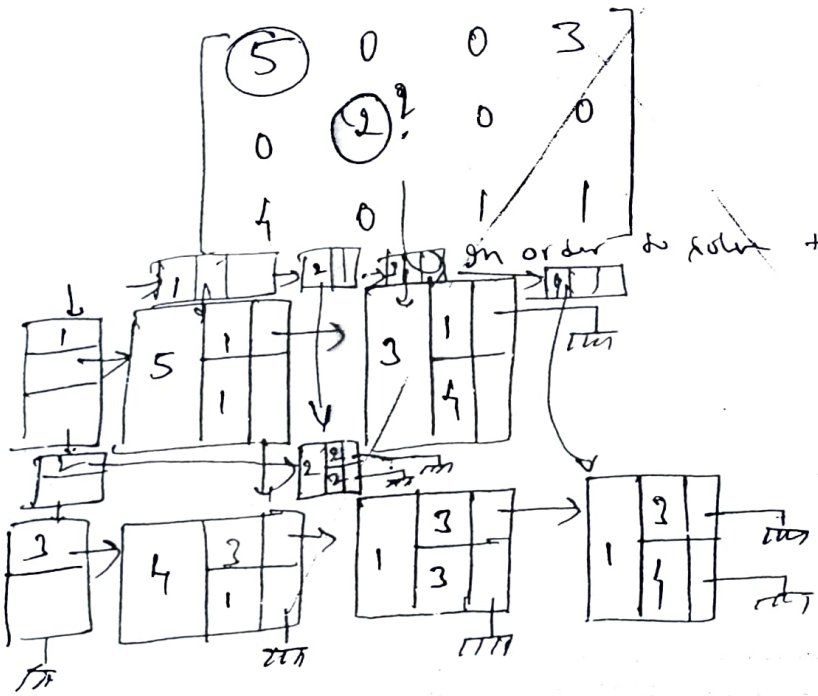




How to Store



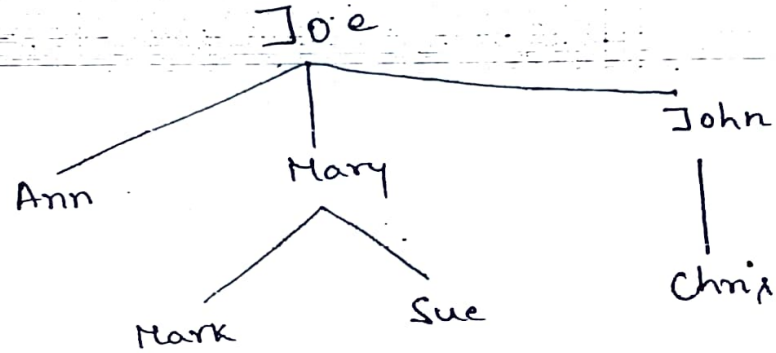




In order to solve this will have a header pointer

In hierarchical data we have an ancestor-descendant, Superior - Subordinate or Similar relationship among the data Elements.

The figure below shows the descendants of Joe arranged in a hierarchial manner, beginning with Joe at the top of the hierarchy.



Joe's children (Ann, Mary & John) are listed next in the hierarchy & a line or Edge Joins Joe. his children. Ann has no children, while Mary has two & John has one. Mary's children are listed below her & John's child is listed below him. There is an Edge b/w each parent & her children.

From this hierarchial representation, it is easy to identify Ann's Siblings, Joe's descendants, Chris's ancestors & so on.

Definition: A tree 't' is a finite nonempty set of Elements. One of these Elements is called the root & the remaining Elements (if any) are Partitioned into trees which are called the Subtrees of t.

Let us see how this definition relates to Examples of hierarchial data.

In the descendants of Joe Example, the data set { Joe, Ann, Mary, Sue, John, Chris }. So,

- → n = 7
- → The root of the collection is Joe.
- → The remaining elements are partitioned into the three disjoint sets { Ann }; { Mary, Mark, Sue }; & { John, Chris }
- → { Ann } is a tree with Single element, i root is Ann.
- → The root of { Mary, Mark, Sue } is Mary & that of { John, Chris } is John.
- → The remaining elements of { Mary, Mark, S are partitioned into disjoint sets { Mark } & { Sue }, which are both Single-Elemen (Sub) trees, & the remaining element of { John, Chris } is also a Single-Element Subtree.

When drawing a tree, Each element is represented as a node. The tree root is drawn at the top & its subtrees are drawn below.

Each subtree is similarly drawn with its root at the top & its subtrees below.

The Edges in a tree connect an element node & its children nodes.

For Example Ann, Mary & John are the children of Joe, & Joe is their parent.

Children Ann, Mary

children of the same parent are called Siblings. Ann, Mary & John are Siblings in the tree but Mark & Chris are not.

In a tree, elements with no children are called leaves. So Ann, Mark, Sue & Chris are the leaves of the tree. The tree root is the only tree element that has no parent.

Another commonly used tree term is level. By definition, the tree root is at level 1; its child (if any) are at level 2; their children (if any) are at level 3; & so on.

The degree of an element is the no. of children it has. The degree of a leaf is zero.

The degree of a tree is the maximum of its element degrees.

Binary Trees

Definition: A binary tree 't' is a finite (Possibly empty) collection of elements. When the binary tree is not empty, it has a root element and the remaining elements (if any) are partitioned into two binary trees, which are called the 'left' & right subtrees of t.

The essential differences b/w a binary tree & a tree are:

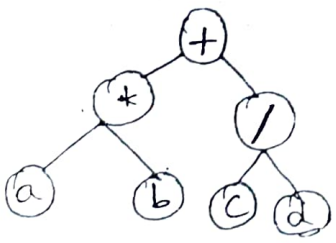
* A binary tree can be empty, whereas a tree cannot.
* Each element in a binary tree has exactly two subtrees (one or both of these subtrees may be empty). Each element in a tree can have any number of subtrees.
* The subtrees of each element in a binary tree are ordered. That is we distinguish b/w the left & the right subtrees.
The subtrees in a tree are unordered.

Properties of Binary Trees

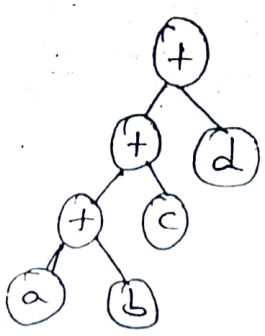
Property:1 The drawing of every binary tree with n elements, $n>0$, has exactly $n-1$ edges.

Proof: Every element in a binary tree (except root) has exactly one parent.
There is exactly one edge b/w each child & its parent. So the no of edges is $n-1$.

The height or depth of a binary tree is the no of levels in it.



Height = 3



Height = 4

Property 2: A binary tree of height h, $h \geq 0$, has at least h & at most $2^h - 1$ elements in it.

Proof: Since there must be at least one element at each level, the no of elements is at least h. At each element can have at most two children the no of elements at level i is at most 2^{i-1}, $i \geq 1$. For $h=0$, the total no of elements is 0, which equals $2^0 - 1$. For $h>0$, the no of elements cannot exceed

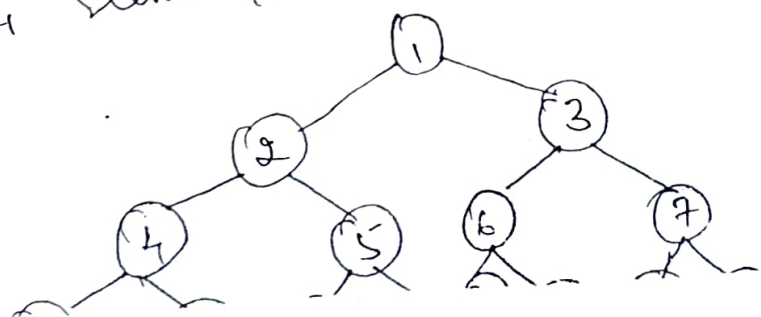
$$\sum_{i=1}^{h} 2^{i-1} = 2^h - 1$$

complete binary tree

Property 1: The height of a binary tree that contains n, $n \geq 0$, elements is at most n at least $\lceil \log_2 (n+1) \rceil$.

Proof: Since there must be at least one element at each level, the height cannot exceed n. From property 2 w.k.t a binary tree of height i can have no more than $2^i - 1$ elements. So $n \leq 2^h - 1$. Hence $h \geq \log_2 (n+1)$. Since h is integer, we get $h \geq \lceil \log_2 (n+1) \rceil$.

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

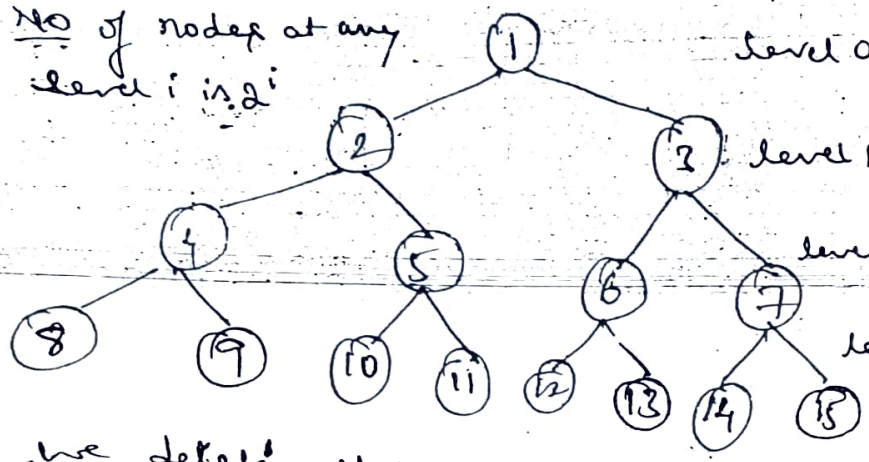


Full binary of height

Property 4: Let i, $1 \le i \le n$, be the no. a... to an Element of a Complete binary tree.

(Complete binary tree:

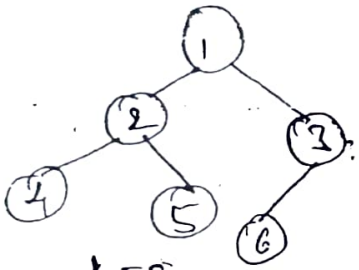
Defn: No of nodes at any level i is 2^i



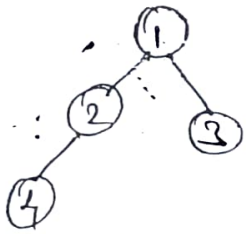
NOTE:
Strictly binary tree
If the outdegree of every node in a tree is either 0 or 2.

If we delete the k Elements ($k \ge 0$) numbered $2^1 - i$ ($1 \le i \le k$) for any k. The resulting binary tree is called a Complete binary tree.

The height of a complete binary tree that contains n elements is $\lceil \log_2 (n+1) \rceil$



$k=9$



$k=11$

$k=13$

Complete binary trees

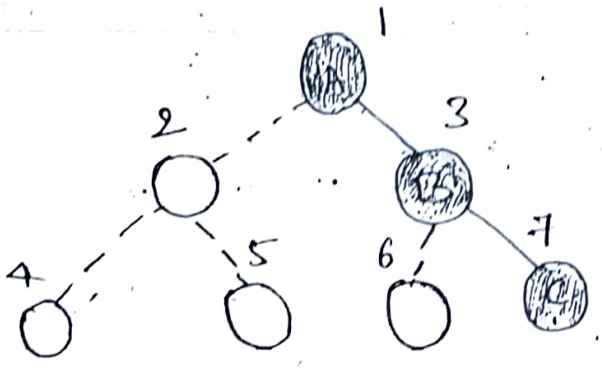
The following are true

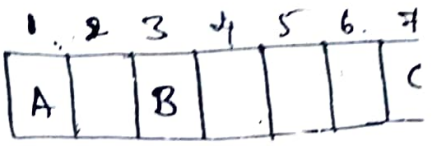
1. If $i=1$, then this Element is the root of the binary tree. If $i>1$, then the parent of this Element has been assigned the number $\lfloor i/2 \rfloor$
2. If $2i > n$, then this Element has no left child. Otherwise its left child has been assigned the number $2i$.
3. If $2i+1 > n$, then this Element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

Representation of Binary trees

1. Formula Based Representation:

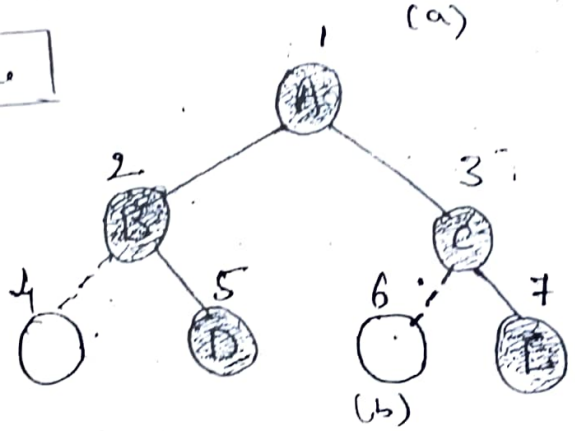
The binary tree to be represented is regarded as a complete binary tree with some missing elements. Figure below shows two sample binary trees.

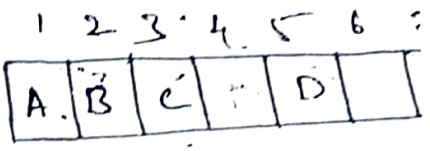




(a)

Fig: Reference





(b)

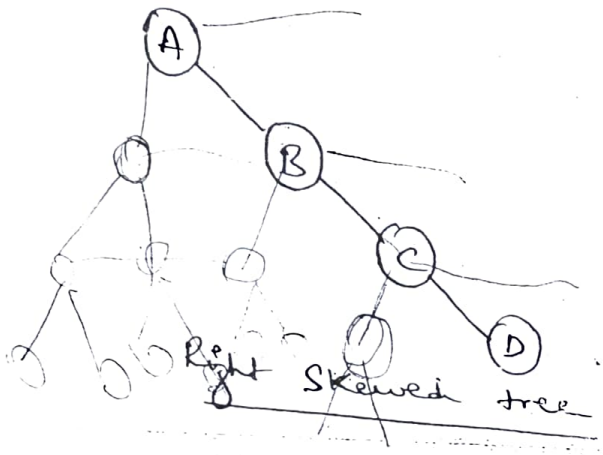
The first binary tree has 3 elements (A, B & C) a Second has five elements (A, B, C, D & E). Neither is complete. unshaded circles represen missing elements. All elements (including the missing one) are numbered as shown.

In a formula based representation, the binary tr is represented in an array by storing each elemen at the array position corresponding to the numbe assigned to it. Formula based representations for

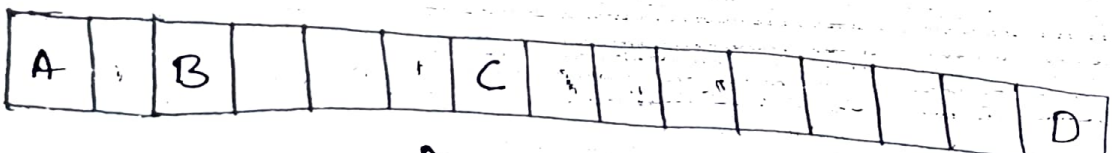
As can be seen, this representation is quite wasteful of space when many elements are missing. In fact, a binary tree that has n elements may require an array of size up to $2^n - 1$ for its representation.

This maximum size is needed when each element (except the root) of the n-element binary tree is the right child of its parent. Figure below shows such a binary tree with four elements.

Binary trees of this type are called Right-Skewed binary trees.



Right Skewed tree



Array representation

The formula based representation is useful only when the no of missing elements is small.

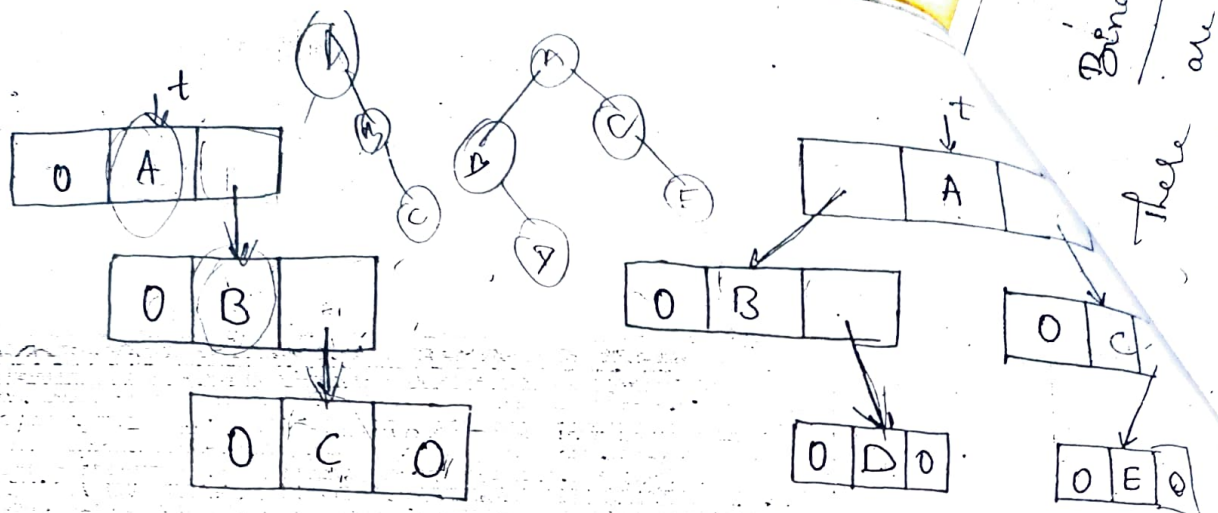
Linked Representation

The most popular way to represent a binary tree is by using links or pointers.
Each element is represented by a node that has exactly two link fields.
One link field is Leftchild & the other is Rightchild. In addition to these two link fields, each node has a field named data.
This node structure may be defined as a C++ template class as in program shown

```
template <class T>
class BinaryTreeNode
{
  private:
  T data;
  BinaryTreeNode<T> *Leftchild,
                    *Rightchild;

  Public:

BinaryTreeNode(int e, BinaryTreeNode *l,
               BinaryTreeNode *r)
  { data = e;
    Leftchild = l;
    Rightchild = r;
  }
};
```



(a) Linked representation (b)

Fig: References is shown above.

Linked representation for the binary trees in

Each Edge in the drawing of a binary tree is represented by a pointer from the parent node to the child node. This pointer is placed in the appropriate link field of the parent node.

Since an n-Element binary tree has Exactly n-1 Edges, we are left with $2n-(n-1)=n+1$ link field that have no value. These link fields are set to Zero as shown.

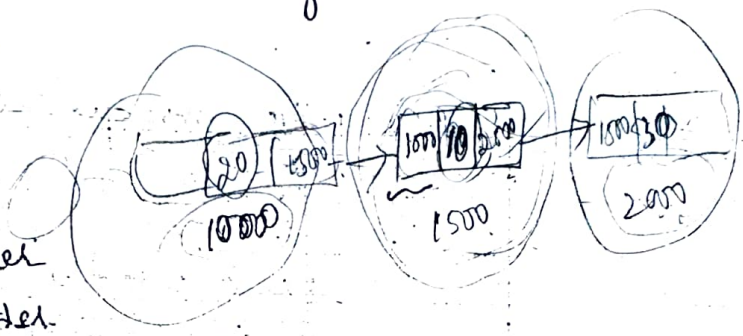
A variable 't' is used in the fig (a) & (b) of Linked representation to keep track of the root of the binary tree.

We can access all the nodes in a binary tree t by starting at the root & following Leftchild & Rightchild links.

Binary Tree Traversal

There are four common ways to traverse a binary tree:

* Preorder
* Inorder
* Postorder
* Level order



In the first three traversal methods, the left subtree of a node is traversed before the right subtree.

The difference among the three orders comes from the difference in the time at which a node is visited

→ In the case of a preorder traversal each node is visited before its left & right subtree are traversed.

→ In an Inorder traversal, the root of each subtree is visited after its left subtree has been traversed but before the traversal of its right subtree begins.

→ In a postorder traversal, each root is visited after its left & right subtrees have been traversed.

Program 1

```
Void Preorder (BinaryTreeNode *t)
{
    If (t)
    {
        Visit (t);          // visit tree root
        Preorder (t->Leftchild); // do left subtree
        Preorder (t->Rightchild); // do right subtree
    }
};
```

Preorder traversal

Program 2

```
Void Inorder (BinaryTreeNode *t)
{
    If (t)
    {
        Inorder (t->Leftchild); // do left subtree
        Visit (t);              // visit tree root
        Inorder (t->Rightchild); // do right subtree
    }
};
```

Inorder traversal

Program 3

```
Void Postorder (BinaryTreeNode *t);
{
    If (t)
    {
        Postorder (t->Leftchild); // do left subtree
        Postorder (t->Rightchild); // do right subtree
        Visit (t);              // visit tree root
    }
};
```

Postorder traversal

Binary Search Tree

* A Binary Search Tree is a binary tree that may be empty. A nonempty binary search tree satisfies the following properties.

1. Every element has a key (or value) & no two elements have the same key. ∴ all keys are distinct.
2. The keys in the left subtree of the root are smaller than the key in the root.
3. The keys in the right subtree of the root are larger than the key in the root.
4. The left & right subtrees of the root are also binary search trees.

* The number inside a node is the element key.
* The tree shown in the following figure(a) is not a binary search tree even though it satisfies properties 1, 2 & 3. The right subtrees fails to satisfy property 4.
* This subtree in fig(a) is not a binary search tree, as its right subtree has a key value (22) that is smaller than the key value in the right subtree's root (25

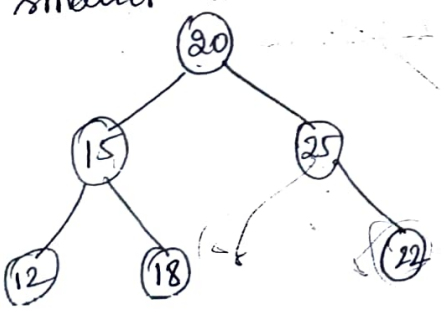


fig (a)

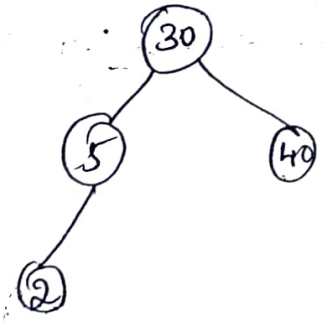


fig (b)

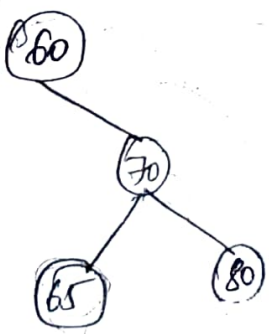


fig (c)

* the binary trees of fig (b) & (c) are search trees.

Binary Search Tree operations

1 Searching :- To search for a pair with the given key.

Step 1 :- If the root is NULL the search tree contains no pairs & the search is successfull.

Step 2 :- Compare key with the key in the root, If the key value is less than the key in the root then no pair in the right subtree can have the key & only left subtree has to be searched.

Step 3 :- If the key is larger than the key in the root, only the right subtree needs to be searched.

Step 4 : If the key equals the key in the root then the search terminates successfully.

2) Inserting an Element

Step 1 : To insert a new pair into a binary tree, first determine whether its key is different from those of existing pairs by performing a search for the key.

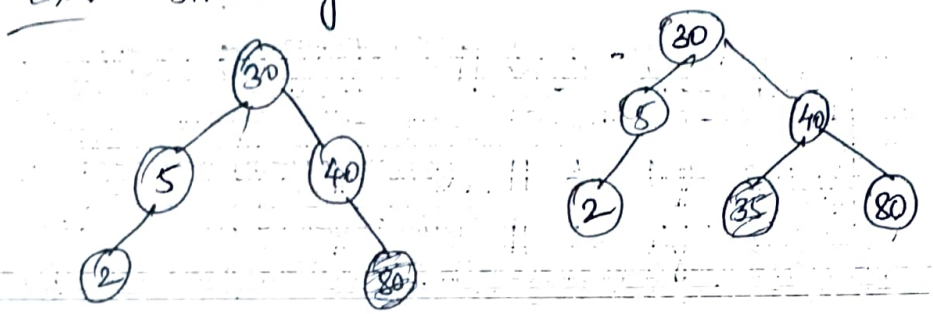
step 2 : If the search is successfull, replace the old value associated with the first pair with the value of the record.

Step 3 :- If the search is unsuccessfull, then the new pair is inserted as a child

are examined during the search.

2) Deleting an Element.

Ex:- Inserting 80 into the tree.

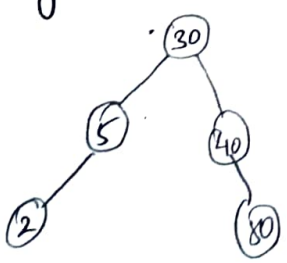


3) Deleting an Element.

* The 3 possibilities for the node p that contains the pair that is to be removed.
(i) p is a leaf (ii) p has exactly one non empty subtree (iii) p has exactly two non empty subtrees.

Case i: It is handled by discarding the leaf node & if the discarded leaf was also the tree root the root is set to NULL.

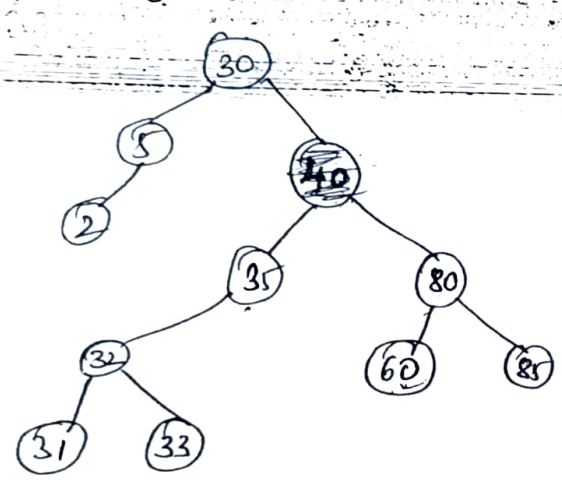
Case ii:- If p has no parent (it is the root), the root of its single subtree becomes the new search tree root. If p has parent then change the pointer from parent so that it points to p's only child.



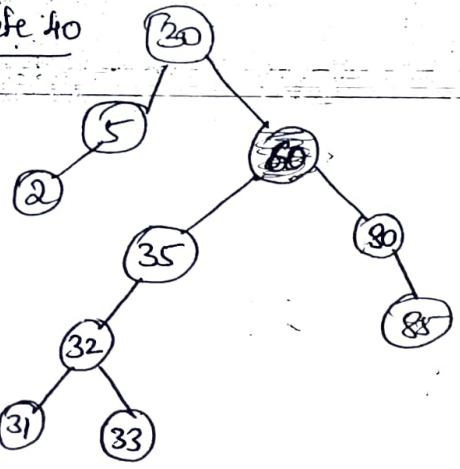
To delete 5, make root 30 to points to the 5's only child 2

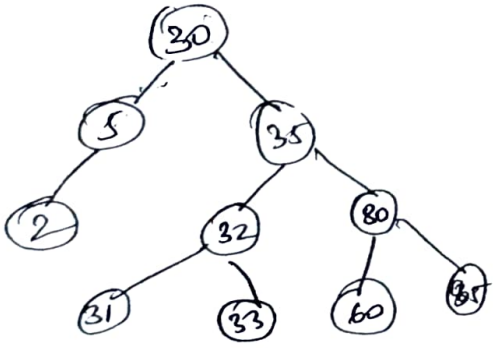
Case iii: To remove a pair in a node that has two non empty subtrees, replace this pair with either the largest pair in its left subtree or the smallest pair in its right subtree.

* The replacing pair is removed from its original node.

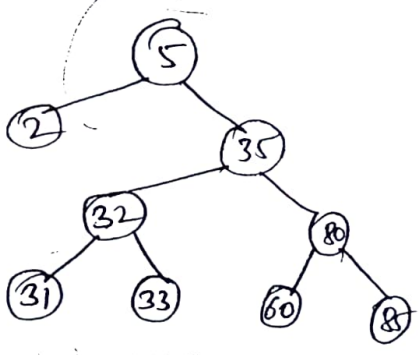


To delete 40





To delete 30



+ + + a b c d

a + b + c + d

a b + c + d +

/ + - a + x y * + b * c a

- a + x + y / + b * c * a

a - x y + + b + c a * * /

Balanced Search Tree

* Balanced Tree structure ~~are~~ tree structure whose height is $O(\log n)$.
* The performance for the search, insert & delete operations of a search tree is $O(\log n)$.
* One of the more popular balanced ~~Trees~~ known as AVL Tree. [Adelson-Velskey-Landies]

Definition :- An empty binary tree is an AVL tree. If T is a non-empty binary tree with T_L & T_R as its left & right subtrees, then T is an AVL tree ① if T_L & T_R are AVL trees & ② $|h_L - h_R| \leq 1$ where h_L & h_R are the heights of T_L & T_R respectively.

* AVL tree is a self-balanced ~~tree~~ binary search Tree.
* Every AVL search tree is a binary search tree but all the binary search Trees need not to be AVL trees.

Ex :-

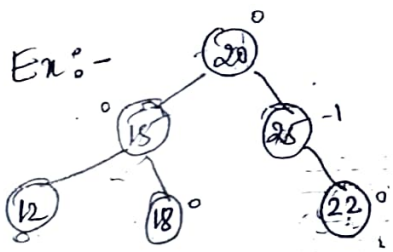


fig (a)

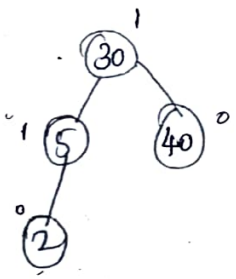


fig (b)

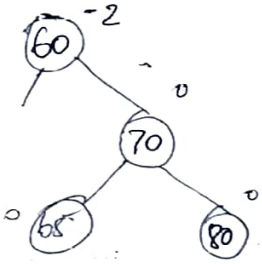


fig (c)

* AVL Search Tree is a binary search Tree & that is also an AVL Tree.
* fig (a) & (b) trees are AVL trees & fig (c) is not
* Tree (a) is not an AVL search tree as it is not a binary search tree.

* AVL search tree represents a dictionary perform each operation in logarithmic time.

* The height of an AVL tree with n elements & nodes is O(log n). It takes O(log n) time for search operation.

* Insert operation – O(log n) time.

* Delete – O(log n) time.

Representation of an AVL tree

* AVL trees are represented by using the linked representation scheme for binary trees.

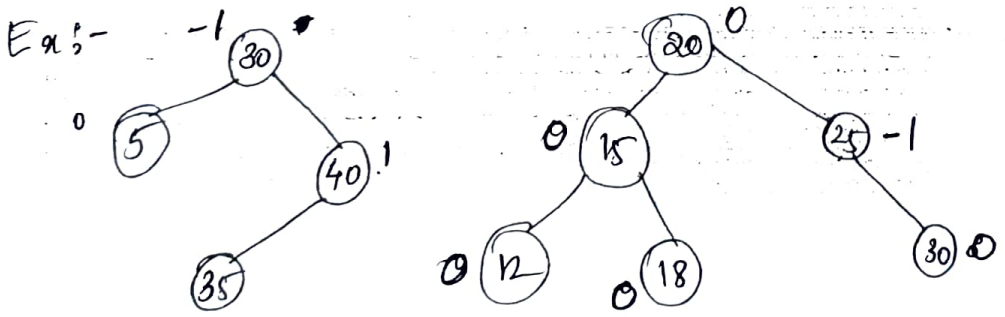
* To facilitate insertion & deletion, a balance factor bf is associated with each node.

* The bf(x) of a node x is defined as,

height of left subtree of x – height of right subtree of x.

* The permissible balance factors are -1, 0, +1.

Ex:-



The number outside each node is its bf

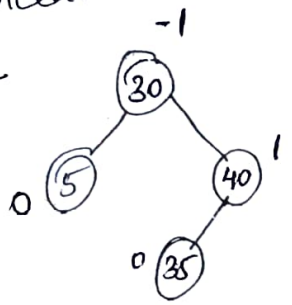
Searching an AVL Search Tree.

Searching an AVL tree is similar to binary search Tree. Since the height of an AVL tree with n Elements is $O(\log n)$, the search time is $O(\log n)$

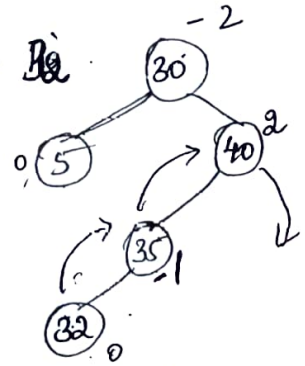
2) Inserting into an AVL Search Tree

* In AVL tree, after performing every operation like, insertion, & deletion, we need to check the balance factor (bf) of every node in the tree.
* If every node satisfies the bf condition then we conclude the operation, otherwise we must make it balanced.

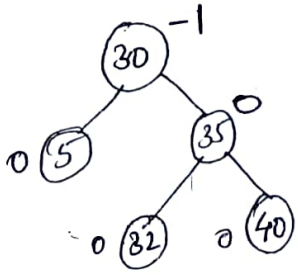
Ex:-



Insert 32

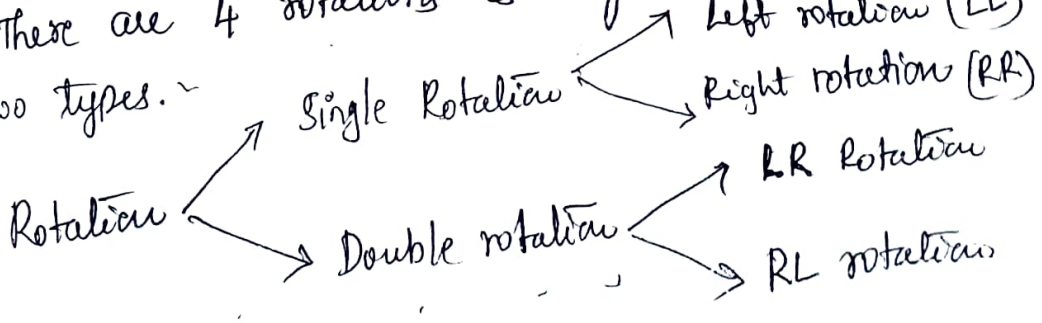


imbalanced AVL Tree



Rebalancing

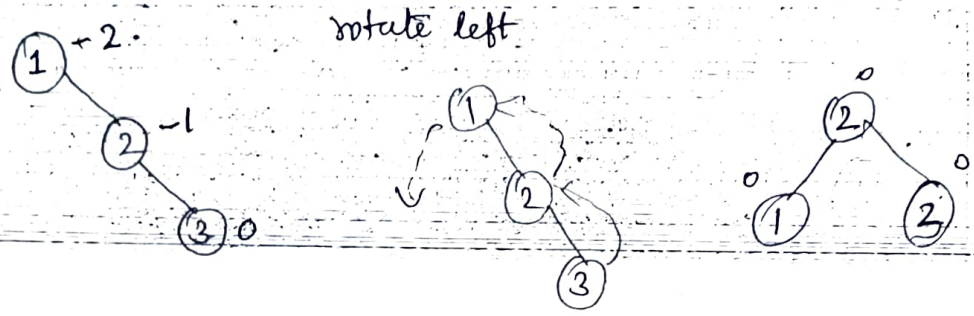
* Rotation operations make tree balanced whenever the tree is becoming imbalanced, due to any operation.
* There are 4 rotations & they are classified into two types.



(i) Single left rotation (LL)

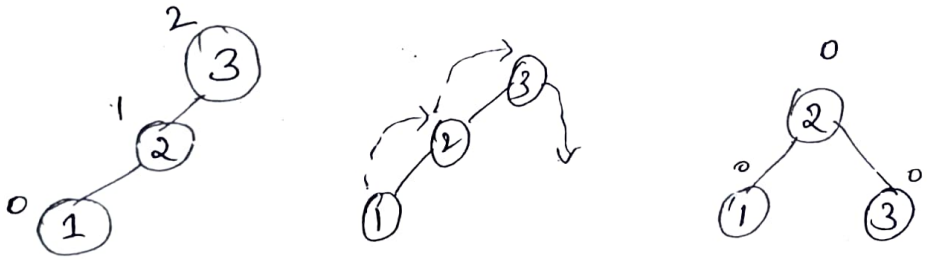
* In LL rotation every node moves one position to left from the current position.

Ex:- Insert 1, 2 & 3.



(ii) Single right rotation (RR)

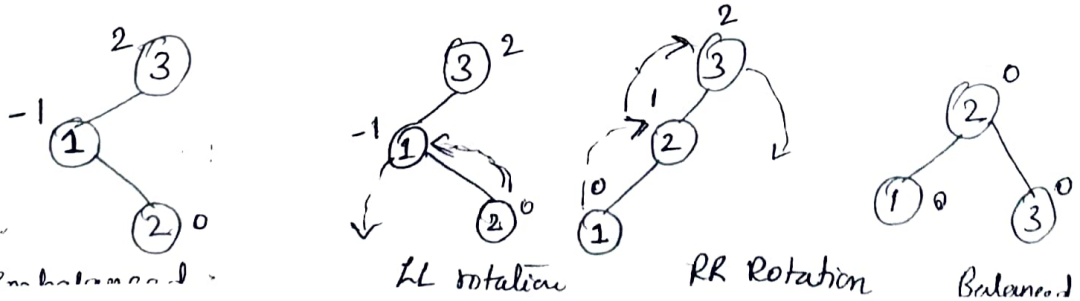
Ex:- Insert 3, 2 & 1



(iii) Left-right (LR) Rotation.

* The LR rotation is combination of single left rotation followed by single right rotation.
* In LR rotation first every node moves one position to left then one position to right from the current position.

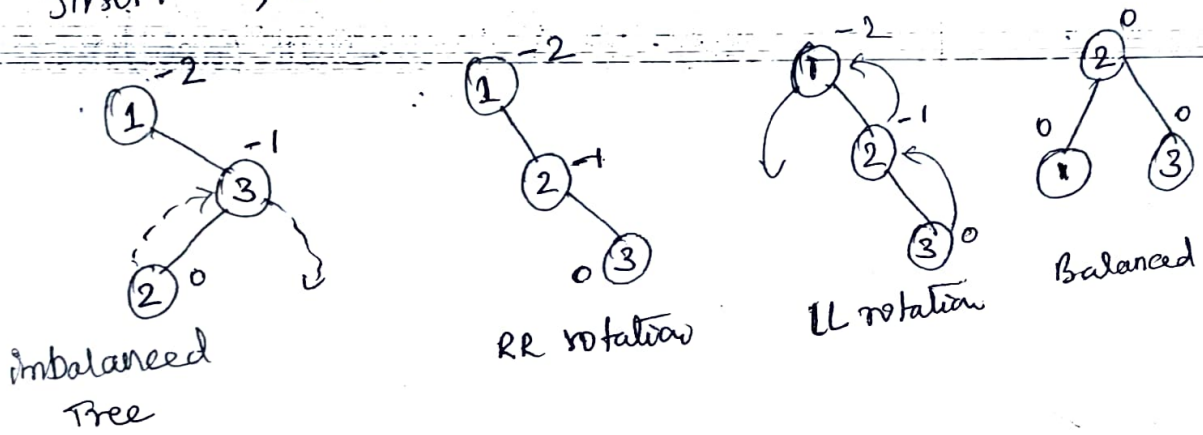
Ex:- Insert 3, 1 & 2



Right left rotation (RL)

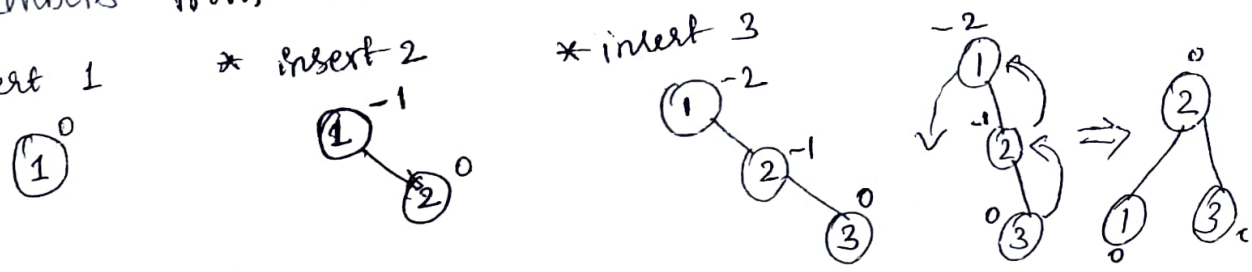
* The RL rotation is combination of single right rotation followed by single left rotation.
* In RL, first every node moves one position to right then one position to left from the current position.

Ex:- Insert 1, 3 & 2

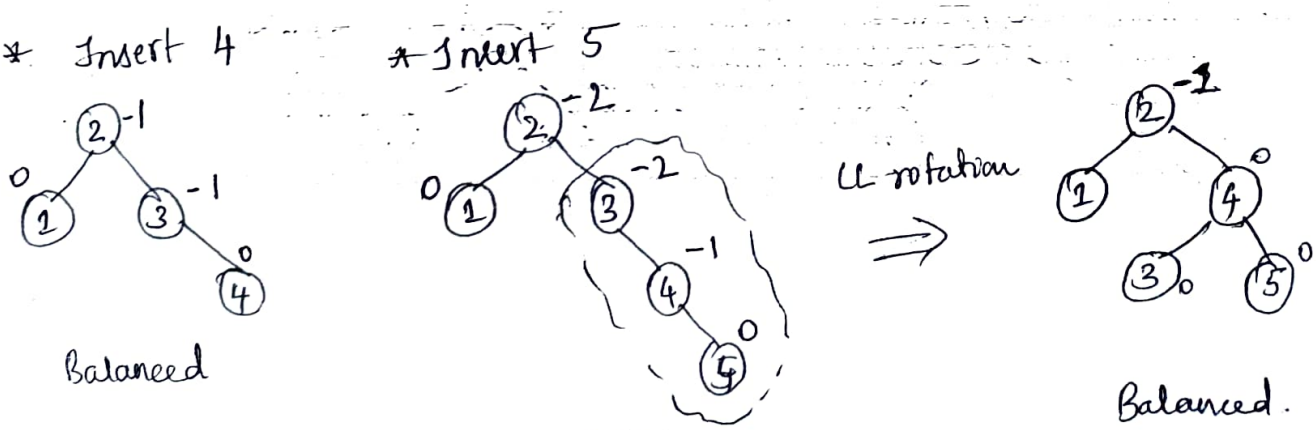


Example :- Construct an AVL tree by inserting numbers from 1 to 8.

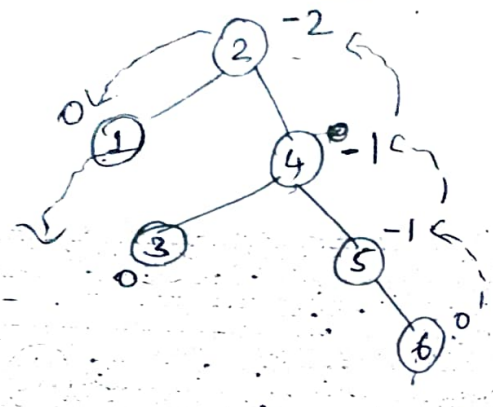
* insert 1
* insert 2
* insert 3



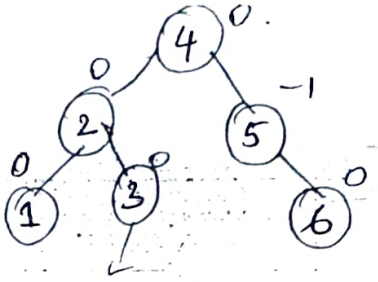
* Insert 4
* Insert 5



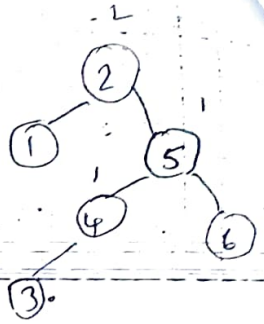
* Insert 6.



LL rotation at 2

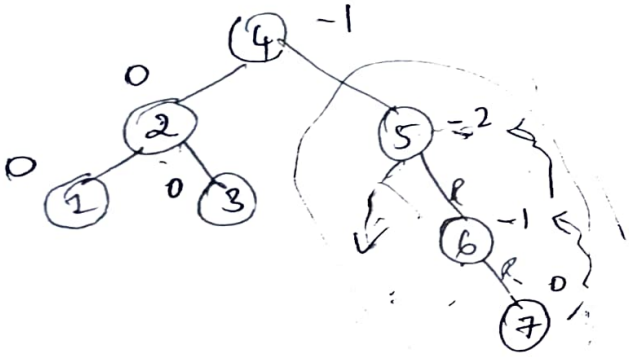


Becomes right child of 2

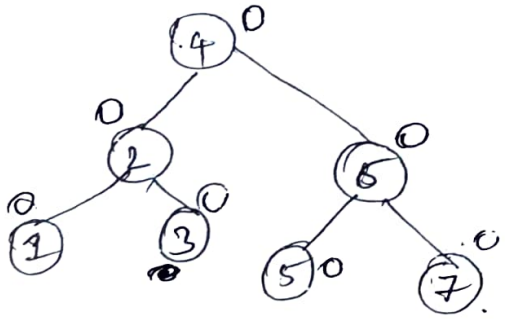


Balanced

* Insert 7.

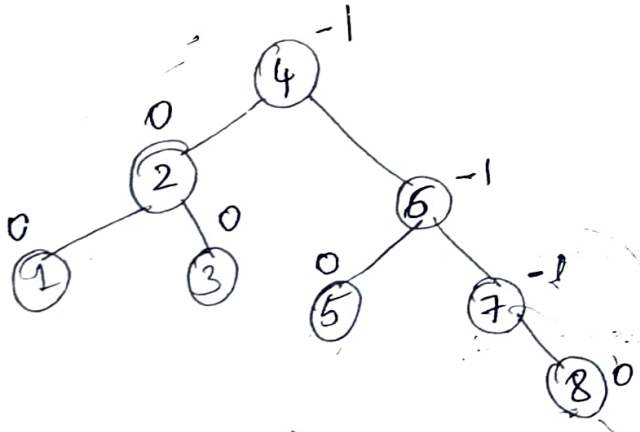


RL rotation



Balanced

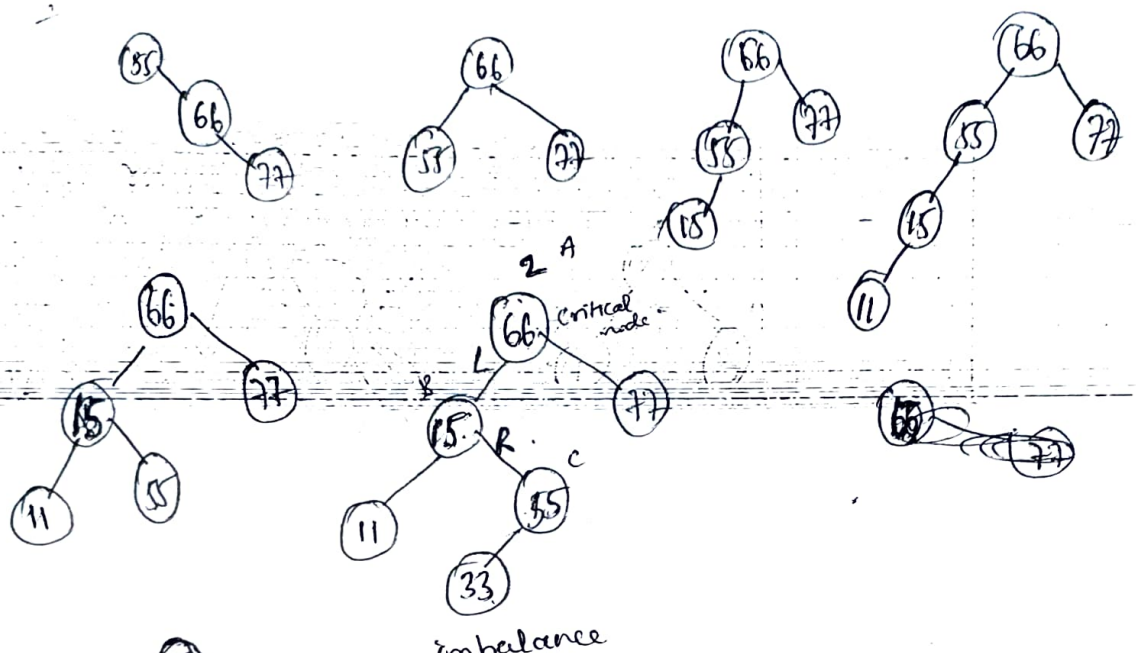
* Insert 8.

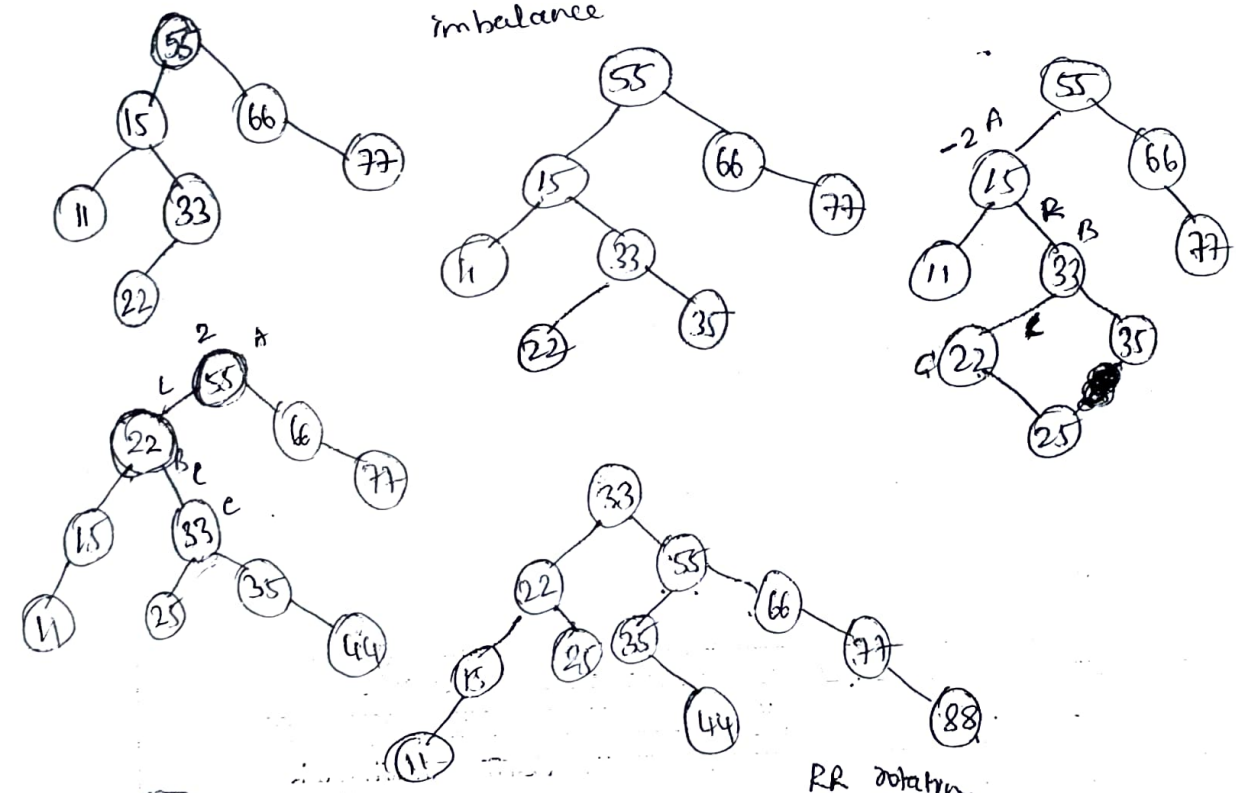


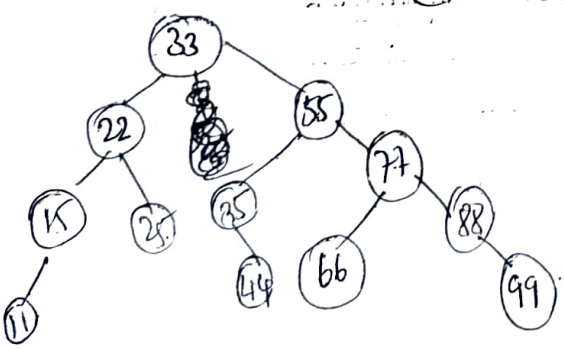
Balanced Tree.

construct a AVL tree for the following numbers

55 66 77 15 11 33 22 35 25 44 88 99







* It takes $(O \log_2 n)$ time for operation.

Final AVL tree.

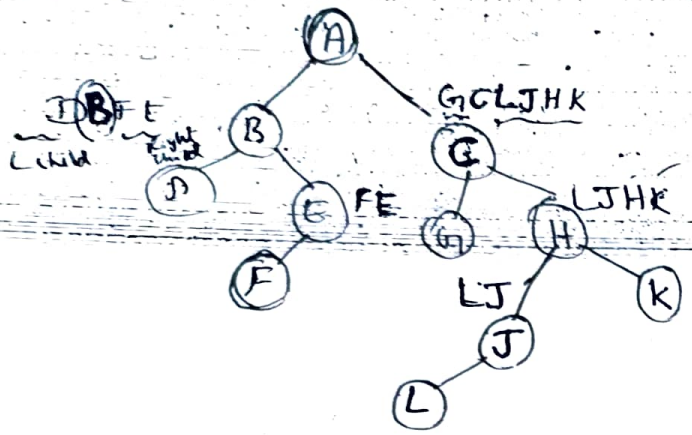
Pre order : A B D E F C G H J L K
In order : D B F E A G C L J H K

Root

left Subtree

Root

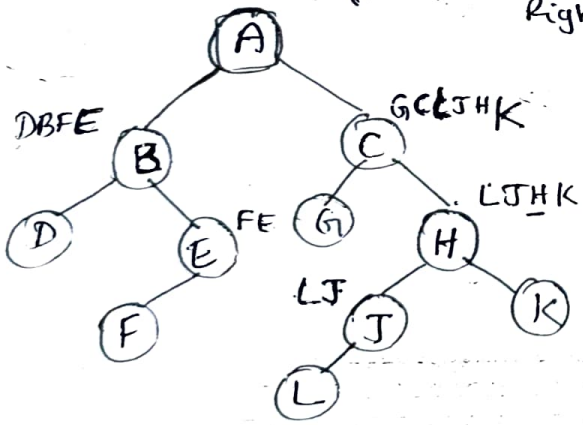
Right Subtree



Post order : D F E B G L J K H C A
In order : D B F E A G C L J H K

left

Right



Pre order : A B D G H K C E F
Post order : G K H D B E F C A

A

Pre order : Root, L, R
Post order : L R Root

Root node

n_1 = Predecessor of root node in Post order.
n_2 = Successor of — " — Pre-order.

Case 1 :- If $n_1 = n_2$ tree is not unique.

Case 2 :- If $n_1 \neq n_2$ n_1 = right child
n_2 = left —"—

First set appears after n_2 & before n_1 in pre-order.

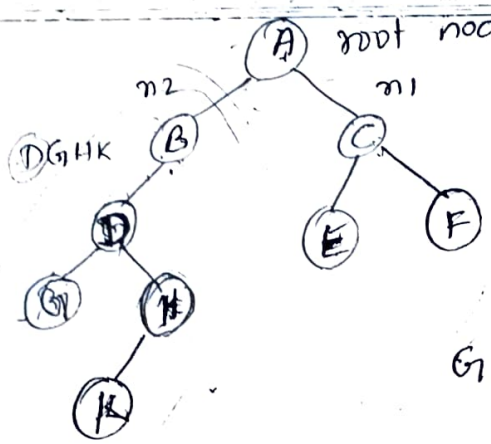
First set is DGHK

Second set is EF

Preorder: A B D G H K C E F

postorder: G K H D B E F C A

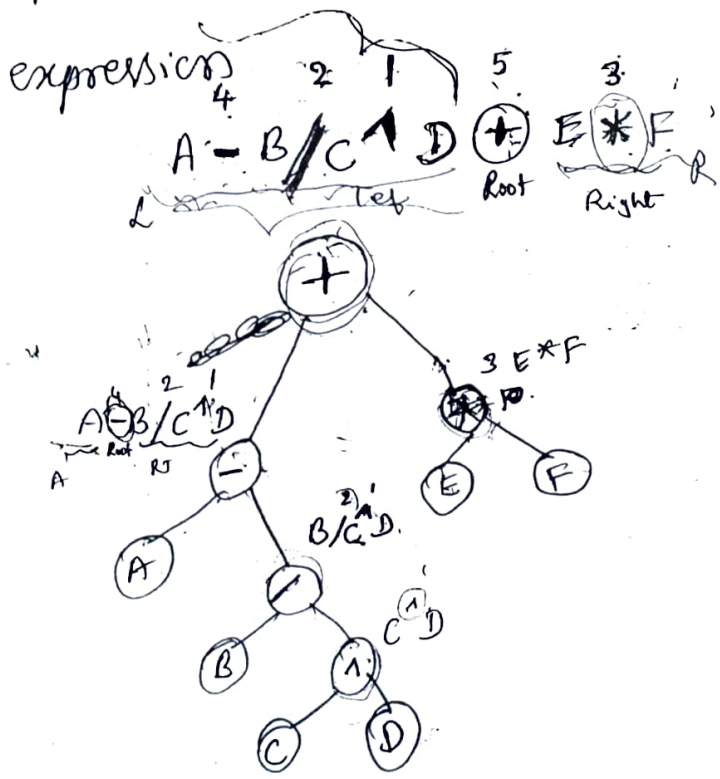
inorder: G D K H B A E C F



GDKH

BODMAS

* How to construct a Binary tree given the expression

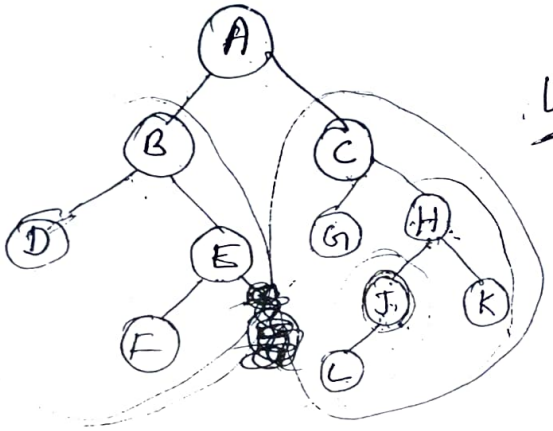


Priority 1) ↑ R-R
2) * / : L-R
3) + - : R-R

~~(A+B*C)+((d*e+f)*g)~~

To construct a tree given pre-order & post order traversals.

Pre-order : A B D E F C G H J L K
post-order : D F E B G L J K H C A



Verify

Pre-order
A B D E F C G H J L K

Post-order
D F E B G L J K H C A

inorder : D B F E A G C L J H K

L Root R

$n_1 = B$
$n_2 = C$

case 1 : $n_1 = n_2$
that node becomes either left root node & right — u —

case 2 : $n_1 \neq n_2$
$B \neq C$
left child
right child.

Find first let

Pre-order B D E F
Second let C G H J L K

Post order D F E B
G L J K H C

Pre-order : E F
Post-order : F E

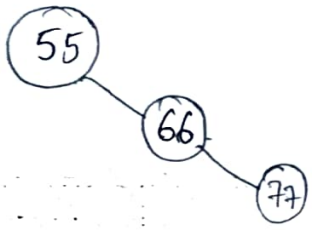
E F
F E

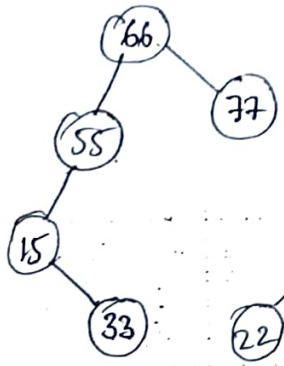
Pre-order : H J L K
Post order : L J K H

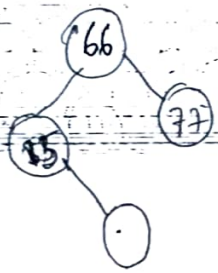
Pre-order J L K
Post-order : L J K

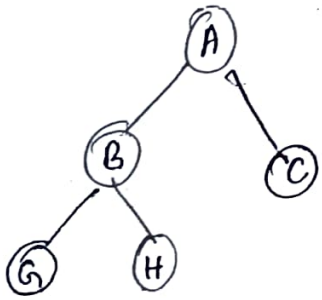
Construct an AVL Tree for following ... 3.6

example of data, 55 66 77 15 11 33 22 35 25





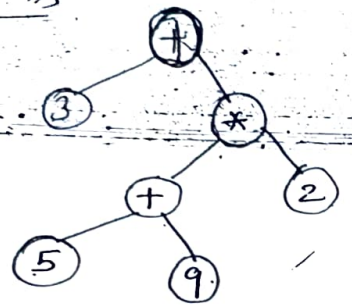




Expression Tree :- Expression tree is a binary Tree in which each internal node corresponds to Operator & each leaf node corresponds to operand.

Ex:- Given: $3+((5+9)*2)$

The tree is



* Inorder traversal of expression tree produces infix expression given postfix expression.
* Pre-order traversal gives pre-fix expression.

Construct a binary tree given the expression

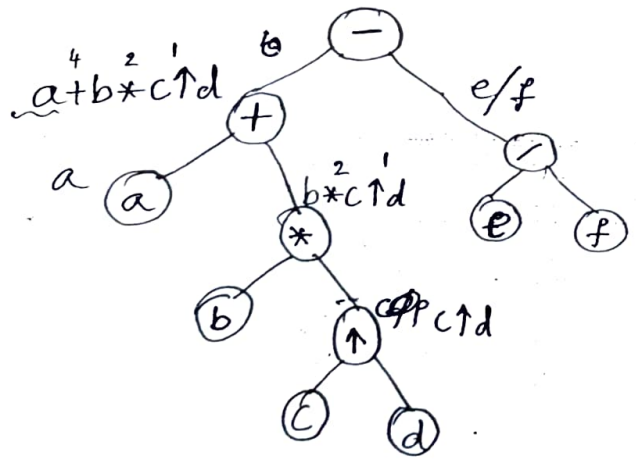
$a+b*c\uparrow d-e/f$

4 2 1 5 3

Left tree — root — Right tree

Make the operator which evaluates last as root.

Priority		Associativity
1.	$\uparrow$	R-L
2.	*, /	L-R
3.	+, -	L-R



Ex:- 2) $*b+4**a*c$

preorder traversal of the tree

$-+a*b\uparrow cd/ef$

$abcd\uparrow*+ef/-$

* If we traverse a tree in-order we get infix expression
* ——— " ——— pre-order ——— " - prefix ——— "
* ——— " ——— post-order ——— " — postfix ——— "

Construction of expression tree given post fix (postorder) expression.

→ Reading postfix expression.

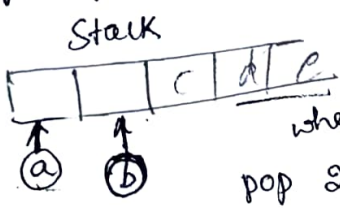
- one symbol at a time
- If symbol is

operand : create a node & push pointer into stack.

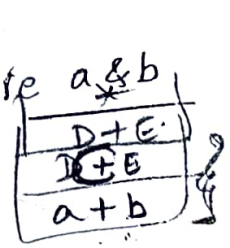
operator : pop 2 pointers & form a new tree with operator as root node & operand as left & right children Then push pointer to this new tree on stack.

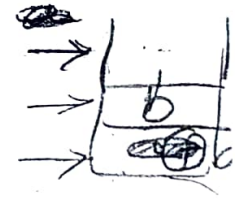
Example :- construct a binary tree given postfix expression. ab + cde + * *

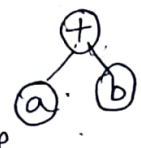
step ① Stack



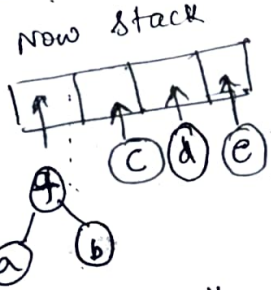
when + comes pop 2 pointers from stack i.e a & b



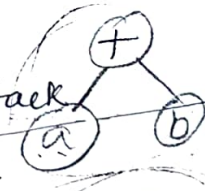




Step ② Now stack

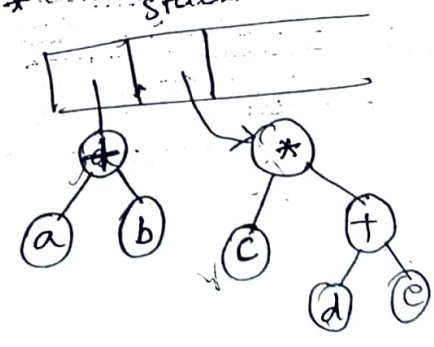


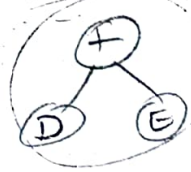
(operator) when + comes pop e & d from stack



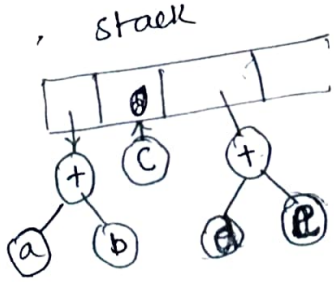
Now * comes pop 2 pointers.

Step ④ stack

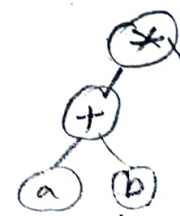




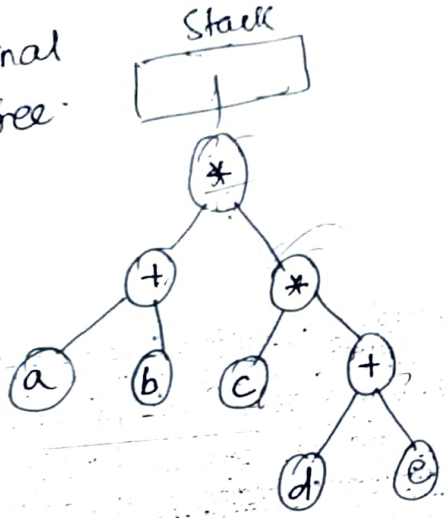
Step ③



Step ⑤ Now * comes then we need to pop 2 pointers.



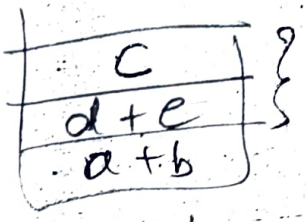
* Final Tree.

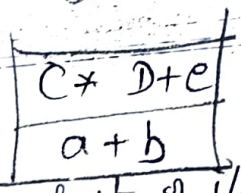


$O(1) < O(\log n) < O(n)$

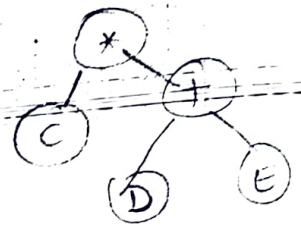
$O(n \log n) < O(n^2)$

$O(2^n)$





C * (D+e



independent of i/p size.

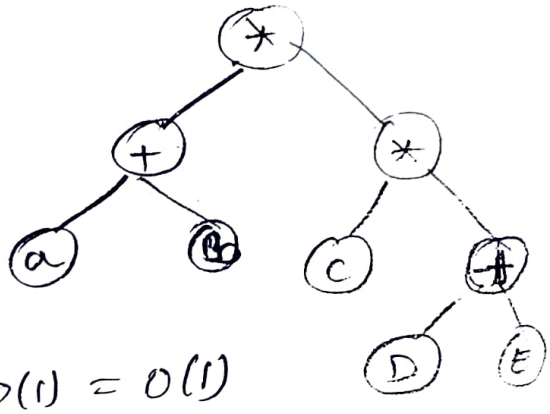
```
x = 5 + (15 * 20);
```

$O(1)$

```
x = 5 + (15 * 20);
y = 15 - 2;
print x + y;
```

Total time = $O(1) + O(1) + O(1) = O(1)$

$= 3 * O(1)$




```
for x in range (0, n);
    print x.
```

$N * O(1) = O(N)$

```
y = 5 + (15 * 20);        O(1)
for x in range (0, n);    O(N)
    print x;
```

Total $= O(1) + O(N) = O(N)$.

```
for x in range (0, n)
    for y in range (0, n):
        print x * y; // O(1)
```

$O(N^2)$.

$O(1)$
$O(N)$
$O(N^2)$

Tree Traversal

* Traversal is a process to visit all the nodes of a tree & may print their values also.
* Because all nodes are connected via edges we always start from the root node.
* we can not randomly access a node in a tree.

There are 3 ways which we use to traverse a tree.

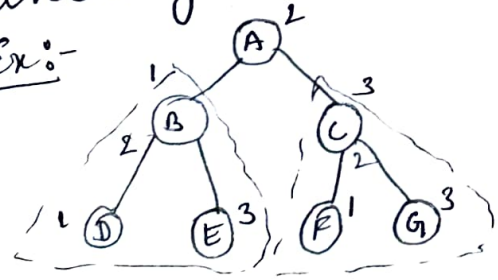
- In-order
- Pre-order
- Post-order
- Level-order

* Traversal a tree is also used to search or locate a given item or key in the tree. & to get all values in it.

(i) In-order Traversal

* In this method left tree is visited first then the root & later the right sub-tree.
* Every node may represent a subtree itself.
* If a binary tree is traversed in-order the output will produce sorted key values in an ascending order.

Ex:-

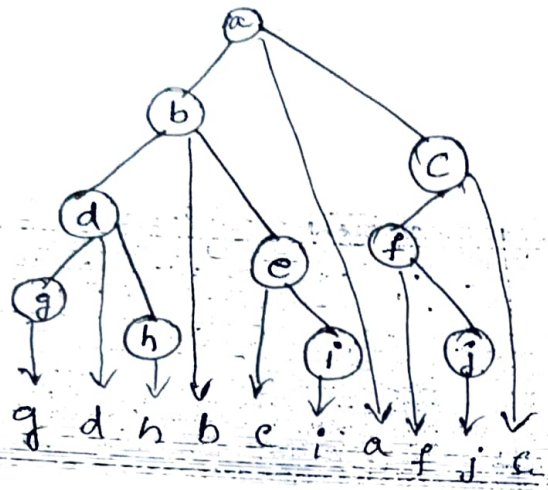


Inorder traversal

D → B → E → A → F → C → G

* 1. Visit the left subtree using inorder
2. Visit the root
3. Visit the right subtree using inorder

Example 2



In order traversal.

→ g d h b e i a f j c

Pre-order for same example

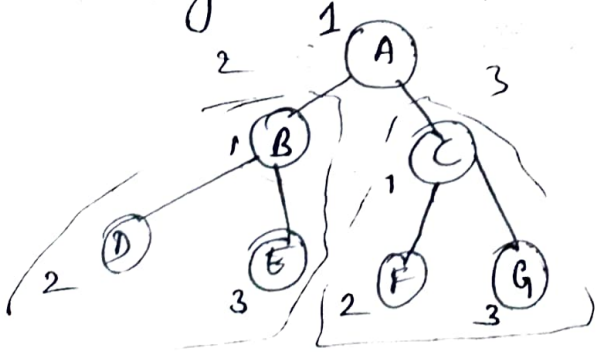
a b d g h e i c f j

Function [Recursive]

```
Public static void inorder(binaryTreeNode t)
{   if (t != null)
    {   inorder(t.leftchild);
        visit(t);
        inorder(t.rightchild);
    }
}
```

Pre-order traversal

* In this traversal method, the root node is visited first, then the left subtree & finally the right subtree.



Pre-traversal.

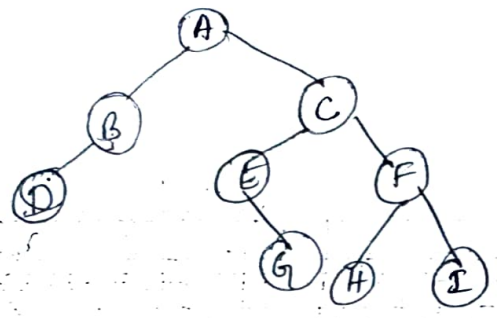
A → B → D → E → C → F → G

i) Visit root node

ii) Visit left subtree using pre-order

iii) Visit right subtree using pre-order

example 2 :-



Pre-order traversal

A → B D C E G F H I

* Pre-order search is also called backtracking.

Function Recursive

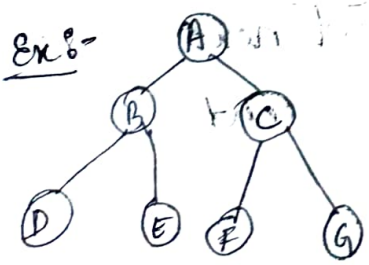
```
public static void preorder (BinaryTreeNode t)
{ if (t != null)
  { visit (t);
    preorder (t.leftchild);
    preorder (t.rightchild);
  }
}
```

Post-order Traversal.

* In a post order traversal each root is visited after its left & right subtrees have been traversed.

1. Visit the left subtree using postorder
2. Visit the right subtree " "
3. Visit the node.

Ex:-



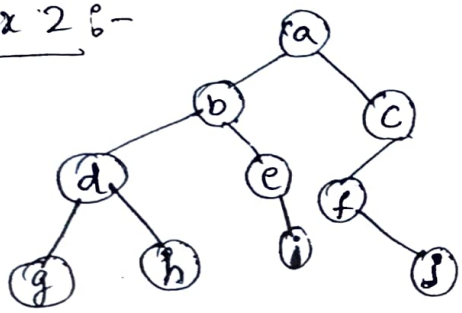
Post-order.

O/p : D E B F G C A

Recursive function.

```
Public static void postorder(binary tree node t)
{ if (t != null)
  { postorder(t.leftchild);
    postorder(t.rightchild);
    visit(t);
  }
}
```

Ex 2 :-



post order

g h d i e b j f c a

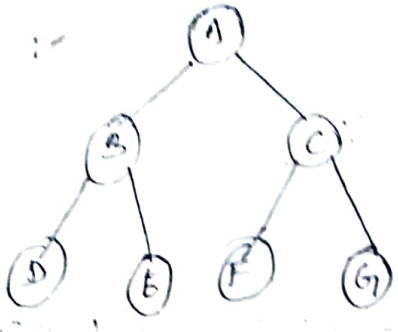
Level order

* In this traversal method, all the nodes are visited according to the levels in which they are.
* First the root node is visited then all the nodes which are in the next level are visited & so on.

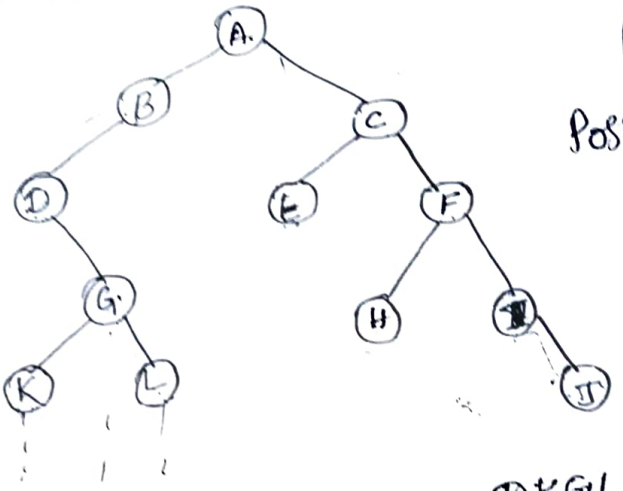
Ex:-
1. Visit root node
2. Visit all nodes in next level from left to right
3. visit all nodes in subsequent level from left to right

Level order traversal

A → B → C → D → E → F → G



In order	In : L Root R Pre : Root, L, R.	Roo of ms



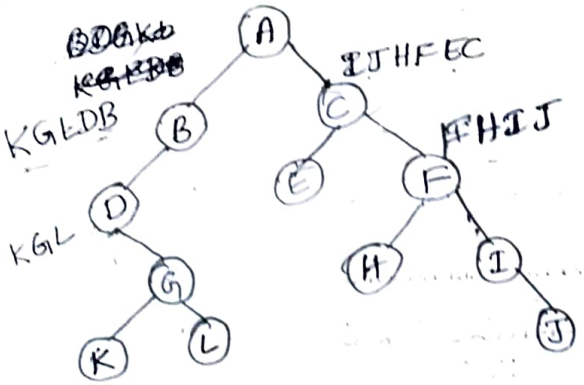
Pre-order :- A B D G K L C E F H I J

Post-order : K L G D B E H J I F C A

E C H F I J

D K G L B A

A B D G K L C E F H I J



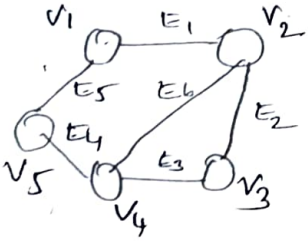
Difference b/w Trees & Graphs.

Graph.

1 Graph is a collection of two sets V & E where V is a finite ~~set~~ non-empty set of of vertices & E is a finite non-empty set of edges.

- Vertices are nothing but the nodes in the graph.
- Two adjacent vertices are joined by edges.
- Any graph is denoted as $G = \{V, E\}$

Ex:-



$G_1 = \{\{V_1\ V_2\ V_3\ V_4\ V_5\}$
$\{E_1\ E_2\ E_3\ E_4\ E_5\ E_6\}\}$

~~Graph is~~

* Graph is a non-linear data structure
* collection of vertices/nodes & edges.

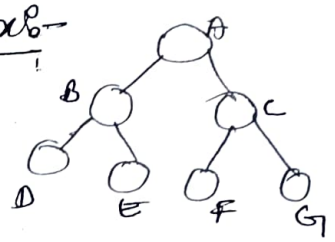
Tree.

1 A Tree is a finite set of one/more nodes such that

(i) There is a specially designed node called root.

(ii) The remained nodes are partitioned into $n >= 0$ disjoint sets $T_1, T_2, T_3 \ldots T_n$ where $T_1, T_2, T_3 \ldots T_n$ is/s called subtrees of the root

Ex:-



A → root node

BDE → left subtree

CFG → Right subtree.

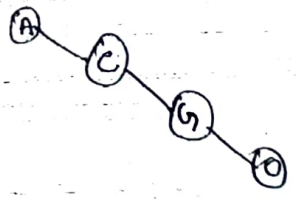
* Tree is a non-linear data structure.
* collection of nodes & Edges

* There is no unique node called root in the graph	* There is a unique node called root
* A cycle can be formed	* There will not be any cycle.
* Applications :- Finding shortest path, in n/w graph is used. colouring of maps, job scheduling etc.	Applications * For game trees, decision trees, the trees are used. sorting & searching
* more complex	* less complex

Skewed Binary Tree

- If a Tree which is dominated by left child node or right child node is said to be a skewed Binary Tree.
- In a skewed binary Tree, all nodes expect one have only one child node. The remaining has no child.

Left skewed Binary Tree.

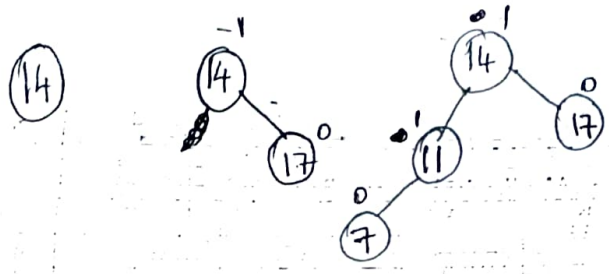


Right skewed Binary Tre

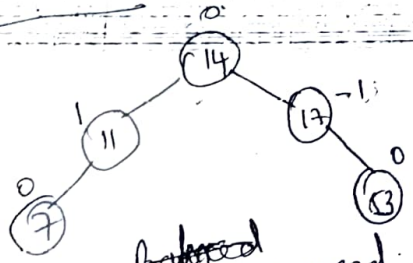
1
2
3
4
5
6 AB
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40

41 – AB
42
43 –
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
prabhanjan

Construct AVL tree by inserting the following data 14, 17, 11, 7, 53, 4, 13, 12, 8, 60.

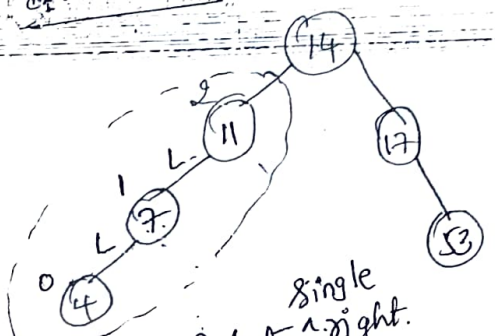


Insert 53

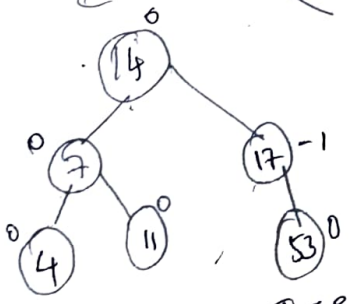


Balanced

Insert 4

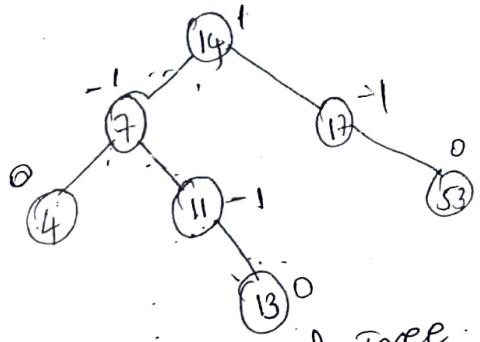


Single Rotate right.



Balanced Tree

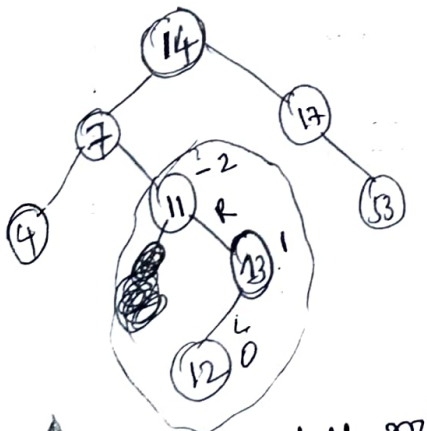
insert 13

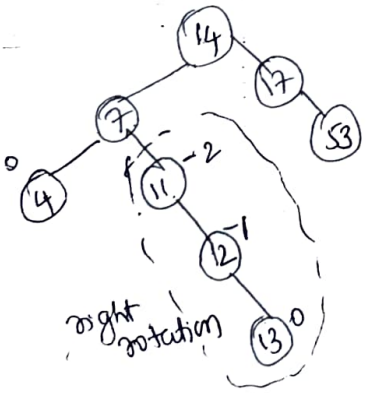


Balanced Tree

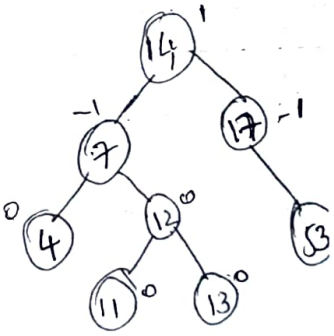
1-2 = -1

Insert 12





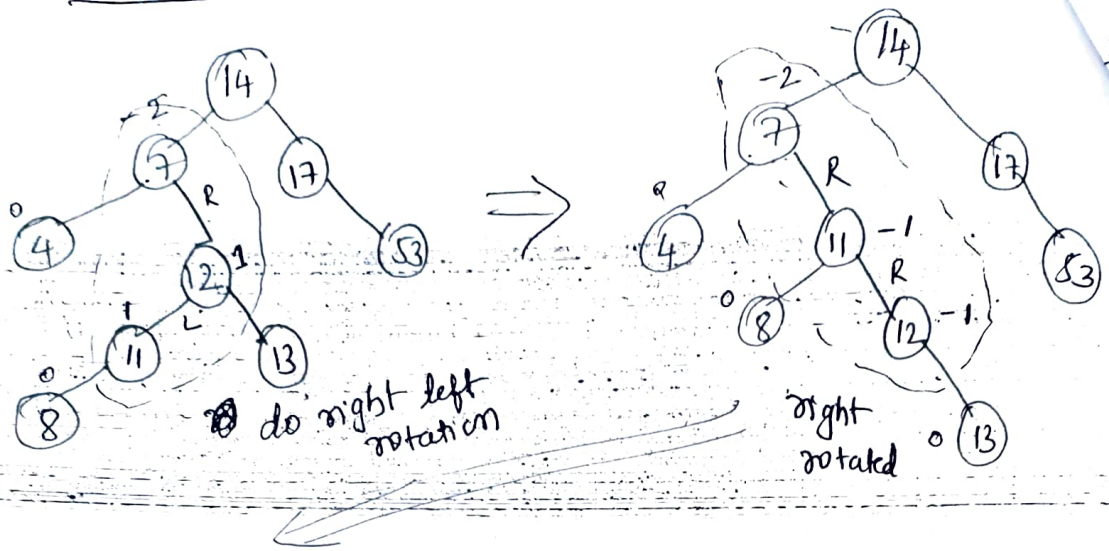
right rotation

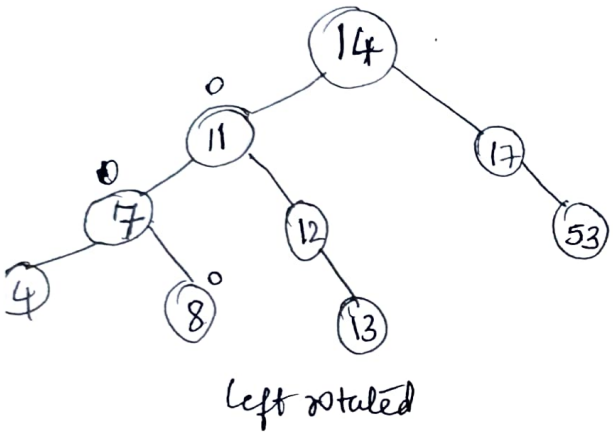


left rotation

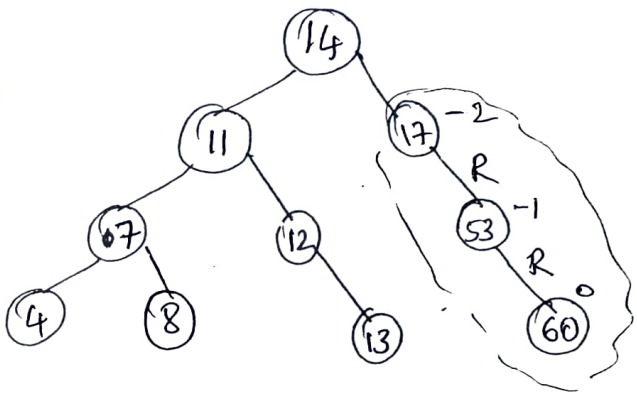
do right left rotation.

Insert 8:

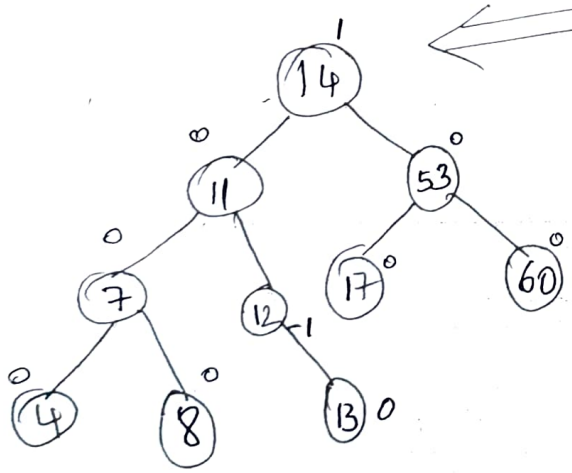




Insert 60



do left rotation



Tree Balanced.

++ Program to check for balanced parenthesis by using stacks.

```
# include <iostream.h>
# using namespace std;
Struct node
 { char data;
   node * next;
 } *p=NULL, *top=NULL, *s=NULL, *ptr;
Void push(char x)
 { p=new node;
   p->data = x;
   p->next = NULL;
   if (top==NULL)
   { top = p;
   }
   else
      { s = top;
        top = p;
        p->next = s;
      } }
char pop()
 } if (top==NULL)
 {cout<<" Underflow!";
  }
  else
  { ptr=top;
    top=top->next;
    return(ptr->data);
    delete ptr;
  }
}
int main()
 { int i;
   char [30], a, y, z;
   cout <<"Enter the expression:\n";
  cin>>e;
for (i=0 ; i<strlen(c); i++)
 { if ((e[i]=='(')|| (e[i]=='{') ||
                     (e[i]=='[')
   { push (c[i])
    }
   else
    { switch(c[i])
     { case')':
          a = pop();
          if (a=='{')||(a=='[']))
          { cout<<" invalid exp!!";
            getch();
          }
          break;
   case '}': y = pop();
      if (y=='[')||(y=='('))
        { cout<<" Invalid exp!!";
          getch();
        { break;
```

```
case ']' : z=pop();
        if ((z=='{') || (z=='('))
        { cout << "Invalid Exp!";
          getch();
        }
        break;
    }
  }
}
  if (top==NULL)
  { cout << "balanced Expn";
  }
Else
  { cout << "String is not valid";
  }
  getch();
}
```

Infix to Postfix conversion

priority

exponential → 3

* / → 2

\+ − → 1

(A+B/C*(D+E−F)

Symbol	stack	postfix
(	(	
A	(+	A
+		AB
B	(+/	
/		ABC
C		

stack	postfix	popped out
*	(+/*	ABC/
(	(+*(	ABC/D
D		
+	(+*(+	
E		
−		
F		