MAHAVEER INSTITUTE OF SCIENCE AND TECHNOLOGY
(AN UGC AUTONOMOUS INSTITUTION)
Approved by AICTE, Affiliated to JNTUH, accredited by NAAC with 'A' Grade Recognized Under Section 2(f)of UGC Act 1956,
ISO 9001:2015 Certified
Vyasapuri, Bandlaguda, Post: Keshavgiri, Hyderabad- 500 005, Telangana, India.
https://www.mist.ac.in E-mail:principal.mahaveer@gmail.com, Mobile: 8978380692

ESTD : 2001

Department of Computer Science and Engineering
(R22)

DATA STRUCTURES

B. Tech II YEAR – I SEM

Prepared by

Mrs.M.Kavyasree
(Assistant Professor)
Department of CSE

CS302PC: DATA STRUCTURES
B.Tech. II Year I Sem.

Prerequisites: Programming for Problem Solving

Course Objectives

- Exploring basic data structures such as stacks and queues.
- Introduces a variety of data structures such as hash tables, search trees, tries, heaps, graphs.
- Introduces sorting and pattern matching algorithms

Course Outcomes

- Ability to select the data structures that efficiently model the information in a problem.
- Ability to assess efficiency trade-offs among different data structure implementations or combinations.
- Implement and know the application of algorithms for sorting and pattern matching.
- Design programs using a variety of data structures, including hash tables, binary and general tree structures, search trees, tries, heaps, graphs, and AVL-trees.

UNIT - I

Introduction to Data Structures, abstract data types, Linear list – singly linked list implementation, insertion, deletion and searching operations on linear list, Stacks- Operations, array and linked representations of stacks, stack applications, Queues- operations, array and linked representations.

UNIT - II

Dictionaries: linear list representation, skip list representation, operations - insertion, deletion and searching.

Hash Table Representation: hash functions, collision resolution-separate chaining, open addressinglinear
probing, quadratic probing, double hashing, rehashing, extendible hashing.

UNIT - III

Search Trees: Binary Search Trees, Definition, Implementation, Operations- Searching, Insertion and
Deletion, B- Trees, B+ Trees, AVL Trees, Definition, Height of an AVL Tree, Operations – Insertion,
Deletion and Searching, Red –Black, Splay Trees.

UNIT - IV

Graphs: Graph Implementation Methods. Graph Traversal Methods.

Sorting: Quick Sort, Heap Sort, External Sorting- Model for external sorting, Merge Sort.

UNIT - V

Pattern Matching and Tries: Pattern matching algorithms-Brute force, the Boyer –Moore algorithm, the
Knuth-Morris-Pratt algorithm, Standard Tries, Compressed Tries, Suffix tries.

TEXT BOOKS:

1. Fundamentals of Data Structures in C, 2 nd Edition, E. Horowitz, S. Sahni and Susan Anderson Freed, Universities Press.
2. Data Structures using C – A. S.Tanenbaum, Y. Langsam, and M.J. Augenstein, PHI/Pearson Education.

REFERENCE BOOK:

1. Data Structures: A Pseudocode Approach with C, 2 nd Edition, R. F. Gilberg and B.A.Forouzan,

Cengage Learning.

27/1/19

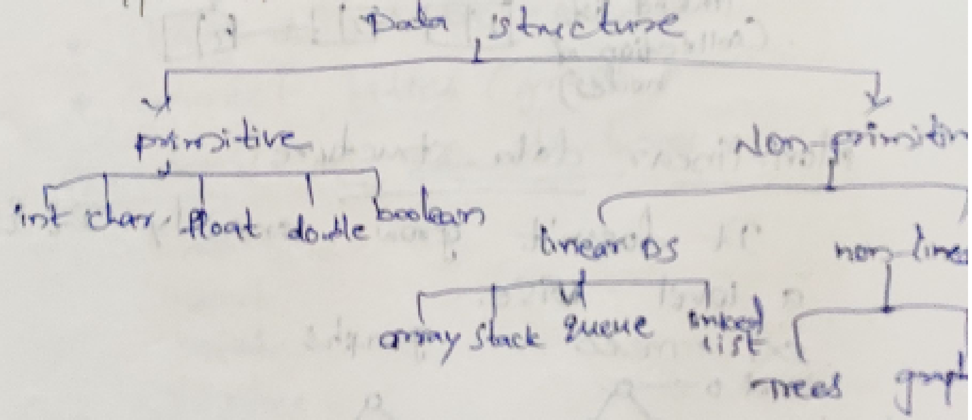
Introduction

* Data structure is data organization, management & storage format that enables efficient access and modification.
* Data Structure is a way in which data is stored on a computer.

Need of Data structure:

* Data structure allow data to be stored in specific manner.
* It allows efficient data search and ritival.
* Specific data structures are decided to work for specific problems. So that it can be accessed and worked in a appropriate manner.
* It allows to manage large amount of data. such as large database & Indexing services.

Types of DS.



primitive data type

1) primitive data type(or) structures are pre-defined data types

2) All these data types are supported by all programming languages.

Ex: int, char, float etc.

3) these are linear in

non-primitive data type

1) these structures are not pre-defined

2) these can be implemented with the help of primitive data-types.

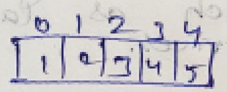
Ex: Arrays, stacks, queues etc.

* these are linear & non-linear in nature

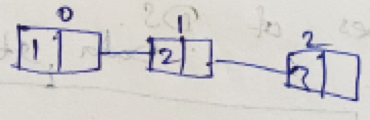
linear datastructure

It grows linearly. Different operating which can be performed in a linear way.

Ex: Array



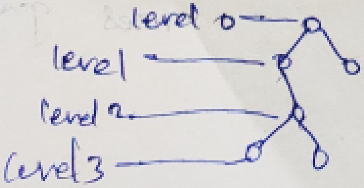
linked list (Collection of nodes)

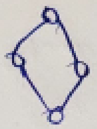


Non-linear data structure

It doesn't grow linearly. It is a level wise.

ex: Trees graphs





Sequential data structure

It utilises memory which is alloted sequentially nothing but Sequential data structure.

Ex: Array 1D, 2D, 3D

Char A[20]

Non-sequential DS

It doesn't utilise memory in Sequential.

Memory alloted is Scattered. in the memory space.

this data structures are nothing but non-sequential DS.

Ex: linked list, trees, graphs.

Applications

* Recursive function calls. (Stack)
* printer in network (queues)
* Stores the data (files)
* Connect cities (graph)

Operations on Data structures

Create

Insert

Delete

Display

Search

modify (or) update.

Static and Dynamic representation

static representation:

Do you have a fixed size.

- * Use when the size of the data is known by the programmer.
- * memory used is fixed. so no control over the structure is needed to proven issues with the structure using too much memory.
- * Memory allocated to the structure can be reused or redirected.

Dynamic representation

- * Doesn't have fixed size.
- * Uses a heap which is a section of memory which can be increased or decreased in size required.
- * Useful when implementing a data structure where the size of data is not known to be stored by the programmer.

Strings I/o in c programming.

Read and write strings in C programming

1) printf, scanf

2) puts, gets

```
Syntax scanf ("%s", str1);    printf ("%s", str1);
       gets (str1);           puts (str1);
```

```
main()
{
 char name[20];
printf("Enter your name");
Scanf("%s", name);
printf("%s", name);
return 0;
}
```

```
main()
{
 char name[20];
printf("Enter your name");
gets(name);
puts(name);
return 0;
}
```

String Handling functions

Length of string

```
void main()
{
 int i;
char str1[10];
 printf("Enter name");
gets(Str1);
for(i=0; str[i]!=\0; ++i)(or) i= strlen(str1);
  printf("the length of string is %d", i);
return 0;
}
```

Compare of strings

```
void main()
{
 int i;
char str1[10], str2[10];
 printf("Enter names");
  gets(str1);
  gets(str2);
for(i=0; str[i]!=\0 && str2[i]!=\0 ; i++)
{
if(str1[i]< str2[i])
printf("both are not equal");
else if (str1[i] != str2[i])
 printf("both are not equal");
else
  printf(" both are equal");
 }
 return 0;
```

(or)

```
if (strcmp(str1, str2)==0)
printf("str1, str2 are equal
else
printf("str1, str2 are not equal
 return 0;
```

String copy

```
   void main()
  {
  int i;
char str1[10], str2[10]
printf ("Enter name");
gets(str1);

for (i=0; str1[i]! = '\0'; i++)
  {
  str2[i] = str1[i]
  }
 str2[i] = '\0';
  printf(str2);
   return 0;
   }
```

(or)

```
Strcpy(str2,str1);
printf (str2);
```

string reverse

```
void main()
{
int i=0;
Char str1[10],str2[10];
printf ("Enter names");
  gets (str1);
j= strlen(str1)-1;
while (i<j) or(strlen(str1))
 {
 Str2 [i] = str1[j]
    i++;
    j--;
   }
  puts(str2);
  return 0;
  }
```

(or) str2 = Strrev(str1)
puts(str2);

```
i<j
0 6
1 5
2 4
3 3
4 2
5 1
6 0
```

string contatination

```
void main()
{
int  i=0;
char str1[20],str[20];
printf ("Enter names");
   gets(str1);
    gets (str2);
```

(or) Strcat(str1,str2);
puts(Str1);

```
a= strlen(str1); } or for(i=0; str1[i]!=\0;i++)
for ( j=0, i=a+1; str2[j]!= \0;i++,j++)
{
str1[i] = str2[j];
}
  str1[i] = '\0';
puts(str1);
return (0); }
```

~~Stacks~~

linked list

→ write a program to print the given string.

```
#include<stdio.h>
#include<string.h>
int main()
 {
  char str1[10];
  printf("enter str:");
  scanf ("%s", str1);
   printf ("string is:%s",str1);
   return 0;
  }
output:
  enter str: Ankitha
   String is: Ankitha
```

→ write a program to handle the program using gets and puts

```
#include<stdio.h>
#include<string.h>
int main()
 {
  char str1[10];
  printf ("enter string:");
   gets (str1);
   puts (str1);
   return 0;
 }
```

```
output
 enter string:Ankitha
  Ankitha
```

→ write a program to print the length of String.

```
#include<stdio.h>
#include<string.h>
int main()
{
 int i;
 char str1[10];
 printf("Enter string:");
 gets(str1);
  i=strlen(str1);
  printf("length of string: %d",i);
 return 0;
 }
```

```
output
Enter string: ankitha
 length of string: 7
```

→ Write a program to compare two strings.

```
#include<stdio.h>
#include<stdio.h>
int main()
 {
  char  str1[10], str2[10];
  printf("Enter string1:");
   gets(str1);
  printf("Enter String2:");
   gets(str2);
   if (strcmp(str1,str2)==0)
    printf("both are equal");
    else
    printf("both ae not equal");
   return 0;
  }
```

```
output
enter string1: ankitha
Enter String2: ankitha
 both are equal
```

→ write a program to copy a string.

```
#include<stdio.h>
#include<string.h>
 int main()
 {
  char str1[10],str2[10];
   printf ("enter string1:");
    gets(str1);
    strcpy (str2,str1);
     puts(str2);
   return 0;
   }
```

output:

```
enter string1:ankitha
ankitha
```

→ write a program to concate two strings

```
#include<stdio.h>
#include<string.h>
 int main()
 {
   char str1[10],str2[10];
  printf("enter string1:");
    gets(str1);
     printf("enter string2:");
     gets(str2);
     strcat(str1,str2);
      puts(str1);
      return 0;
  }
```

output:

```
enter string1: mittapally
enter string2: ankitha
mittapallyankitha
```

→ write a program to reverse a given string

```
#include<stdio.h>
 #include<string.h>
 int main()
 {
 char str1[10];
 printf("enter string1:");
  gets(str1);
   strrev(str1);
    puts(str1);
   return 0;
  }
```

output:

```
enter string1: ankitha
 ahtikna
```

→ write a program to find the average of four numbers using arrays

```
#include <stdio.h>
 ~~int~~ int main()
 {
  int a[10], n=0,i,sum=0, avg=0;
  printf ("enter size of array:");
  scanf ("%d", &n);
   for (i=0;i<n;i++)
    {
     scanf ("%d' &a[i]);
    }
   for (i=0;i<n;i++)
    {
     sum=sum +a[i];
    }
    avg= sum/n;
    printf ("avg is: %d" ,avg);
    return 0;
  }
```

output

```
enter size of array: 4
 1
 3
 5
 7
avg is: 4
```

→ write a program to access student details using structure.

```
#include<stdio.h>
#include< string.h>
 struct student
 {
  int rollno;
  float marks;
  char name[10];
 };
 void main()
 {
  struct student s1;
  s1.rollno=14;
  s1.marks=50.5;
   strcpy (s1.name,"ankitha");
   printf ("details of student:");
   printf ("\n rollno:%d" , s1.rollno);
   printf ("\n marks:%f" , s1.marks);
   printf ("\n name:%s" , s1.name);
  }
```

output:

```
details of student:
rollno: 14
marks: 50.500000
name: ankitha
```

→ write a program to access student details using union.

```
#include<stdio.h>
#include <string.h>
union student
 {
  int rollno;
  float marks;
  char name[10];
 };
 void main()
 {
   union student s1;
   printf("details of student:");
   s1.rollno = 14;
    printf("\n rollno: %d", s1.rollno);
    s1.marks=50.0;
   printf("\n marks: %f",s1.marks);
    strcpy(s1.name, "ankitha");
     printf("\n name: %s",s1.name);
  }
```

output:

```
details of student:
rollno :14
 marks: 50.000000
  name: ankitha
```

→ write program to emplement enumerated datatype.

```
#include<stdio.h>
enum week{sunday, monday, tuesday,
    wednesday thursday, friday, saturd
int main()
{
  enum week today;
  today=thursday;
  printf("day %d",today+1);
  return 0;
  }
```

output

```
day 5
```

Creating a node:

```
Snode* createnode( int val)
 {
  newnode=(snode *)malloc (sizeof (snode));
   if(new node==null)
   {
    printf ("memory is not allocated");
    return 0;
   }
   else
  {
  newnode → value=val;
   newnode → next=null;
  return newnode;
  }
```

Inserting in beginning

```
 void insert_node_first()
 {
  int val;
  printf ("Enter value");
  scanf ("%d", &val);
 newnode=create_node (val);
 if (first==last && first==null)
  { first=last=newnode;
   first → next=null;
   last → next=null;
  } else {
      temp=first;
      first=newnode
      first → next=temp; }
                                  printf ("Inserted
                                  node in beginning");
                                  }
```

Inserting in the ending:

```
void insert_node_last()
 {
 int val;
 printf("Enter value");
 scanf("%d",&val);
 newnode=create_node(val);
 if(first==last && last==null)
   {
    first=last=newnode;
   first->next=null;
     last->next=null;
   }
  else{
   last->next=newnode;
   last=newnode;
   last->next=null;
   }
 printf("Inserted in the ending");
 }
```

Inserting at position (pos):

```
void insert_node_pos()
{
int pos, val, cnt=0, *prev, ptr;
printf("Enter value");
scanf("%d" &val);
new node=create_node(val);
printf("Enter position to be inserted");
 scanf("%d", &pos);
 ptr=first;               }
while(ptr!=null)          } to detect
{                         } no. of elements
 ptr=ptr->next;           }
  cnt++;                  }
 }
if(pos==1)
{
  if(first==last && first==null)
   {
    first=last=newnode;   }
    first->next=null;     } If no
    last->next=null;      } element in list
    }
   else
   {
    temp=first;           } first
    first = newnode;      } position
    pfirst->next=temp;    }
   }
  . printf("Inserted");
 else if ( pos>1 && poss <cnt)
  {
    ptr=first;
```

```
for (i=0; i<pos; i++)
  {
   prev=ptr;              } To remember
   ptr=ptr→next;          } previous & next
                            elements
  }
  prev→next=newnode;      } keeping
   newnode→next=ptr;      } element
   printf ("Inserted");   } in particular
 }                          position
else
{
 printf ("not in given range");
 }
 }
}
```

Display

```
  void desplay()
   {
    if (first==null)
     printf ("no elements");
    else
     {
      temp=first;
       while (temp→next! =null)
       {
        printf ("%d" ,temp→value);
        temp=temp→next;
       }
     }  (or) for (temp=first; temp→next!=null;
                                 temp=temp→next)
   }        printf ("%d"; temp→value);
```

Deletion

```
void  del_pos()
{
if ( first==null)
 {
  printf("no elements");
 }
else
  {
  printf("Enter value to be deleted");
   scanf ("%d" ,& val);
  }
  ptr=first;
  while (ptr ->next!=null)
   {
   if(ptr->value!=val)
     {
      prev=ptr;
      ptr=ptr ->next;
      cnt++;
     }
   else if (ptr ->value==val)
     {
      prev->next=ptr->next;
      }
      free(ptr);
     printf(" deleted");
     break;
    else{
      printf (" no element to be deleted!!");
      }
   }
}
```

update 0-1, 2, 3, 4, 5
↓
$\boxed{6.}$

Stacks

overflow [full element

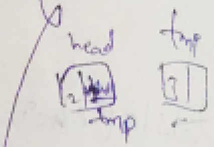
underflow [no element

```
int stack(10), choice, n, top, x, i;
 void push();
  void pop();
  void display();
  int main()
  {
  top = -1;
  printf("Enter your ~~choice~~ stack size");
   scanf("%d", &n);
  do
  {printf("1.push, 2.pop, 3.display, 4.exit");
  printf("Enter your choice");
   scanf("%d", &choice);
   switch(choice)
    {
     case 1:
      {push();
       break;
      }
      case 2:
       {
        pop();
        break;
       }
      case 3:
       {
        display();
        break;
       }
       case 4:
       {
        printf("exit");
        break;
       }
default
 {
  printf("\n Invalid input");
  }
 }
}while(choice!=4);
 return 0;
 }
void push()
 {
  if(top>=n);
  {
  printf("stack is overflow");
  }
 else
  {
 printf("Enter element");
 scanf("%d",&x);
  top++;
  stack[top]=x;
  }
 }
void pop()
{
 if(top<=-1)
  {
 printf("stack is underflow");
  }
 else
 {
 printf(" %d is deleted from stack", stack[top]);
  top--;
  }
 }
```

```
void display()
  {
 if(top<=-1)
  {
 printf("no elements");
 }
 else
 {
for(i=0;i<=top;i++);
printf("%d",stack[i]);
}
}
```

```
struct node
{
int data;
struct node *node;
} → void init(struct node *head)
void push();  {  head=null;
 void pop();    }
void display();
int main()
{
struct node *head =NULL
```

push



so,
head→next=tmp;
tmp=head;

head→next=null;

Queues

```
void delete()
{
if( front<=-1)
 {
 pf("Queue is underflow");
 }
 else{
  pf("%d is deleted element", queue(front));
  front++;
  top--;
 }
```

```
insert()
delete()
display()
```

```
front=-1
rear
~~top~~=-1
 void insert ~~push~~()
 {
 int x;
 if(rear ~~top~~==max)
 printf("Queue is overflow");
 else
 {
  if(front==-1)
   front=0;
  printf("Enter x");
   scanf("%d",x);
   rear ~~top~~=rear ~~top~~+1;
   queue[rear ~~top~~]=x;
  }
 }
```

```
void delete()
 {
  if(front==-1||front
          >rear ~~top~~)
  printf("Queue is
          underflow")
  else(
  printf("%d is deleted
         from the queue,
          queue(front))
 front=front+1;
  }
 }
```

```
void display()
{
if(front==-1||front>rear ~~top~~)
printf("no elements");
 else
  {
 for(i=front;i<=rear;
         i++)
  {
  printf("%d",queue(i));
  }
 }
}
```

Infix notation

The operators go in between the operands ('A' and 'B') is called infix notation.

Ex: A*B, A+B

Prefix notation:

* Instead of saying 'A plus B', we could say "add A, B" and write "+AB"
* this is prefix notation.

Postfix notation

Another alternative is to put the operators after the operands as in "AB+" called postfix.

parenthesis

→ Evaluate 2+3*5

first:

+ (2+3)*5 = 5*5 = 25

* first: 2+(3*5) = 2+15 = 17

→ Infix notation requires parenthesis

prefix

→ +2 *3 5 = +2 *3 5
= +2 15 = 17

→ * + 2 3 5 = * + 2 3 5
= * 5 5
= 25

No parenthesis needed

postfix

2 3 5 * +
2 15 + = 17

2 3 + 5 * = 2 3 + 5 *
= 5 5 *
= 25

No parenthesis is needed.

→ parenthesis are required for infix

((A+B) * (C−E)) / (F+G))

AB+ CE −*/(FG+)

→ AB+CE−*FG+/

output

AB+CE−*FG+/

stack ((+) * (−)) / (+))

Infix to postfix

```
char stack[20];
 int top=-1;
void push(char x)
 {
 stack[++top)=x;
 }
 void pop()
 {
 if(top==-1)
  printf("no elements");
 else
 {
  return stack(top--);
  }
 int priority(char x)
  {
  if(x=='(')
  return 0;
    if(x=='+'||x=='-')
    return 1;
    if(x=='*'||x=='/')
   return 2;
  }
  void main()
  {
  char exp[20];
   char *e,x;
 printf("Enter expression");
 scanf("%s",exp);
  e=exp;
```

```
while (*e!='\0')
 {
 if (isalnum(*e))
  printf("%c",*e);
 else if(*e='(')
  push(*e);
  else if (*e=')')
   while (a=pop()!='(')
    printf("%c",a);
   else
   {
   while (priority(stack[top]>=priority(*e)))
    printf("%c", pop());
     push(*e);
    }
     e++;
   }
  while(top!=-1)
   {
   printf("%c",pop());
   }
  }
```

post-fix evaluation

→ $\underline{2\ 3\ *}\ 4\ +$

6 4 +

10

→ 16 * 5 + 4 * 7 - infix

$\underline{16\ 5\ *}\ 4\ 7\ *\ +$ - postfix

$80\ \underline{4\ 7\ *}\ +$

80 28 +

108

27/1/19

Introduction

* Data structure is data organization, management, & storage format that enables efficient access and modification.

* Data structure is a way in which data is stored on a computer.

Need of Data structure:

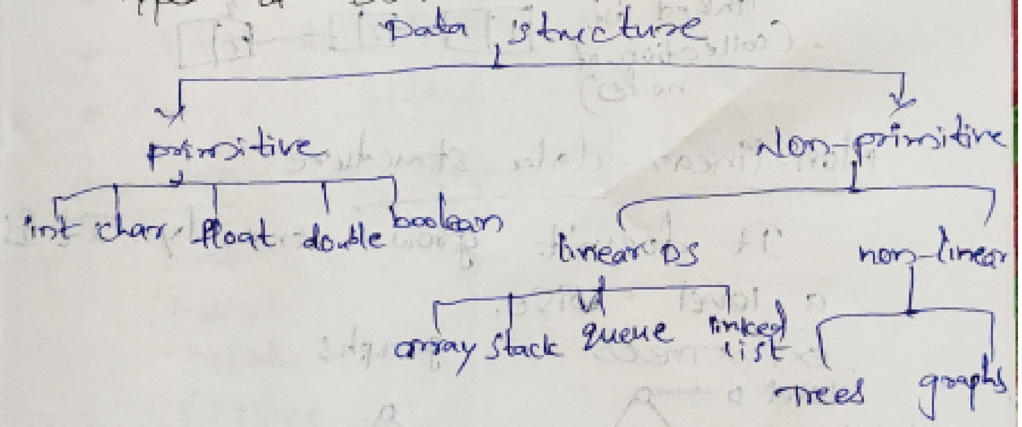
* Data structure allow data to be stored in specific manner.

* It allows efficient data search and ritrival.

* specific data structures are decided to work for specific problems. So that it can be accessed and worked in a appropriate manner.

* It allows to manage large amount of data. such as large database & Indexing services.

Types of DS.

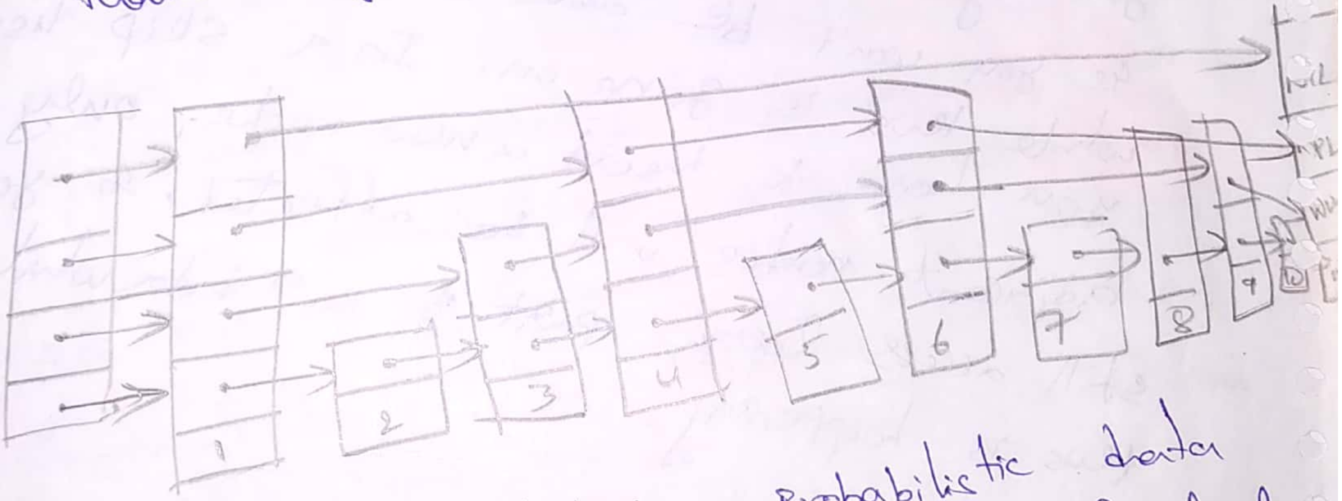


UNIT - II

Skip list in data structure:

A skip list is a data structure that is used for storing a sorted list of items with a help of hierarchy of linked lists that connects increasingly sparse subsequence of the items. A skip list allows the process of items look up in efficient manner.

The skip list data structure skip over many of the items of the full list in one step that's why it is known as skip list.

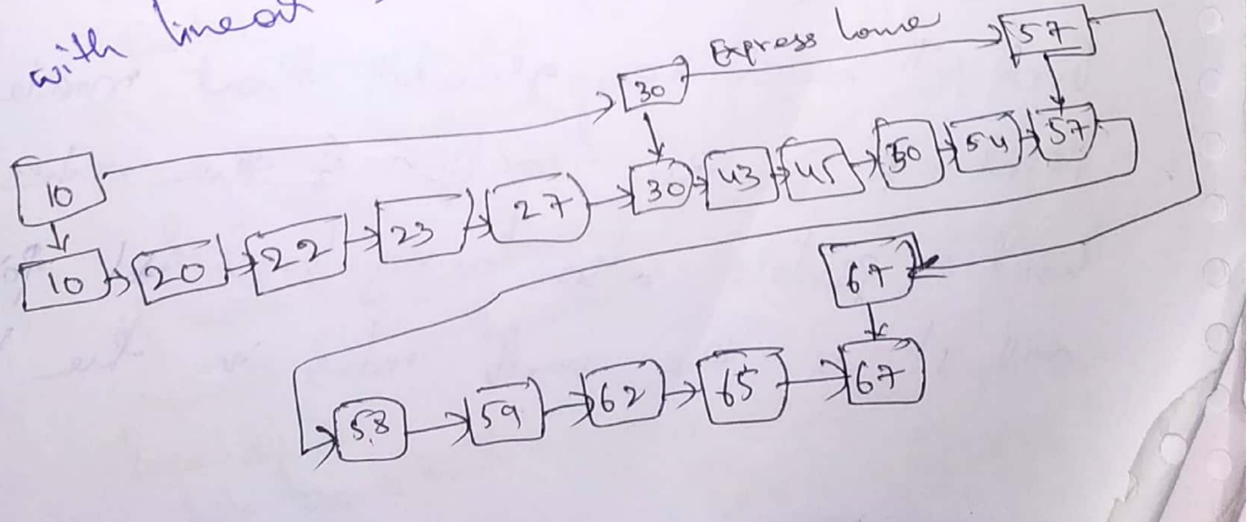


The skip list is a probabilistic data structure that is build upon the general ideal of a linked list. The skip list uses probability to build linked list. Each additional layer of links contains fewer elements but no new elements.

Example: You can think about the skip list like a subway system. There's one train that stops at every single stop. However, there is also an express train. This train doesn't visit any unique stops, but it will stop at fewer stops. This makes the express train an attractive option if you know where it stops.

Skip lists are very useful when you need to be able to concurrently access your data structure. Imagine a red-black tree, an implementation of the binary search tree. If you might have to rebalance the entire thing so you won't be able to access your data while this is going on. In a skip list if you have to insert a new node, only the adjacent nodes will be affected, so you can still access large part of your data while this is happening.

The idea is simple, we create multiple layels so that we can skip some nodes. See following example list with 16 nodes and two layels. The upper layel works as an "express lane" which connects only main outer stations, and the lower layer works as a "normal lane" which connects every station. suppose we want to search for 50 we start from first node of "express lane" we keep moving on "express lane" till we find a node whose next is glater than 50. once we find such a node (30 is the node in following example) on "express lane", we move to "normal lane" using pointel from this node, we linearly search for 50 on "normal lane". In following example we start from 30 on "normal lane" and with linear search, we find 50.



Deleting an element from the skip list!

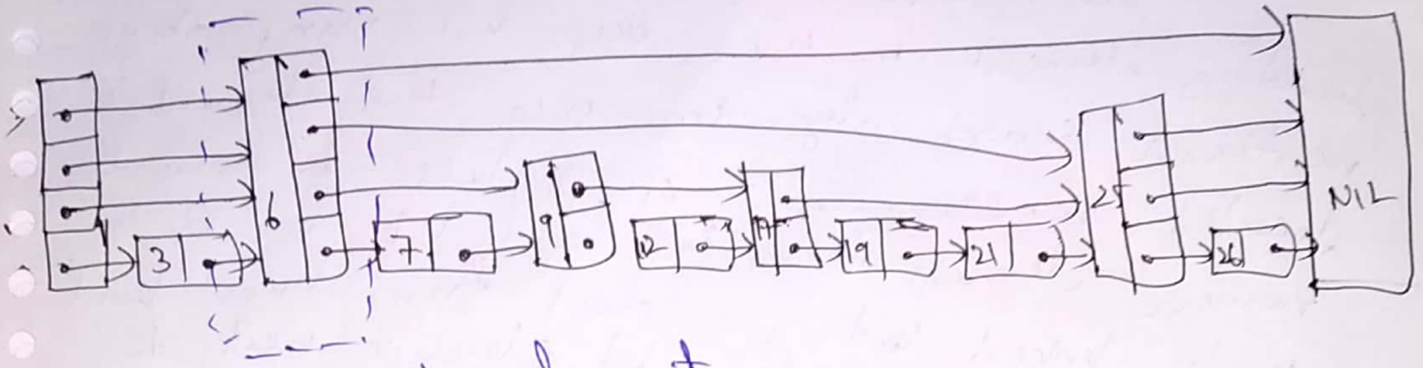
Deletion of an element k is preceded by locating element in the skip list using above mentioned search algorithm. once the element is located, rearrangement of pointers is done to remove element form list just like we do in singly linked list. we start from lowest level we do rearrangement until element next to update[i] is not k.

After deletion of element ther could be level with no elements, so wie will remove these level as well by decrementing the level of skip list.

Following is the code for deletion:

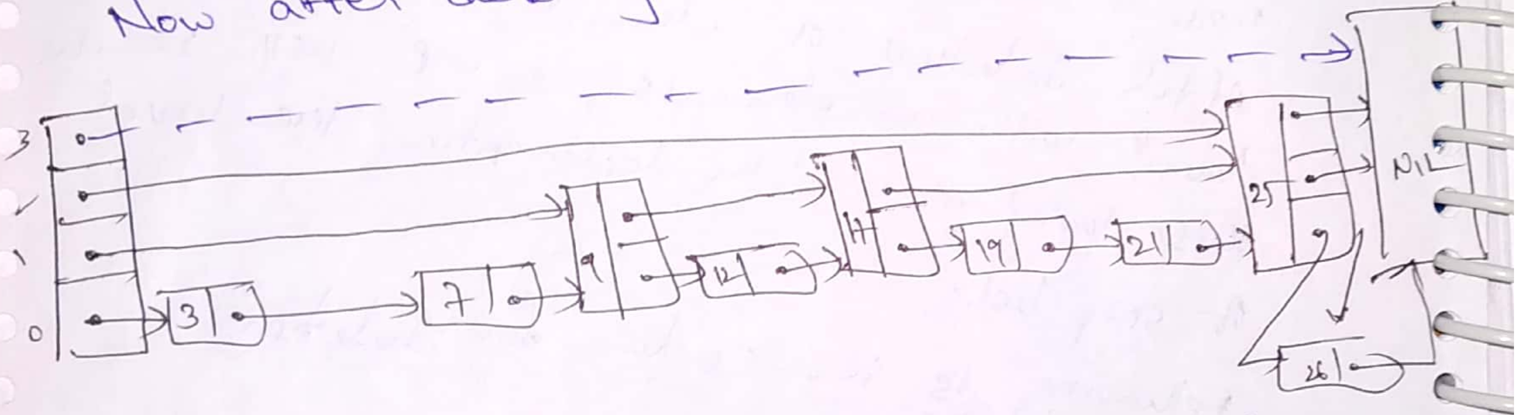
```
Delete(list, searchkey)
 local update[0... maxlevel+1]
 x:=list -> header
 for i:= list -> level down to 0 do
  while x -> forward[i] -> key forward[i]
  update[i]:=x
  x:=x -> forward[0]
  if x -> key= searchkey then
     for i: =0 to list -> level do
     if update[i] -> forward[i] ≠ x then break
     update [i] -> forward[i]: = x -> forward[i]
  free(x)
  while list -> level >0 and list -> header -> forward [list -> level]=NIL do
   list -> level: =list -> level -1
```

Consider this example where we want to delete element 6.

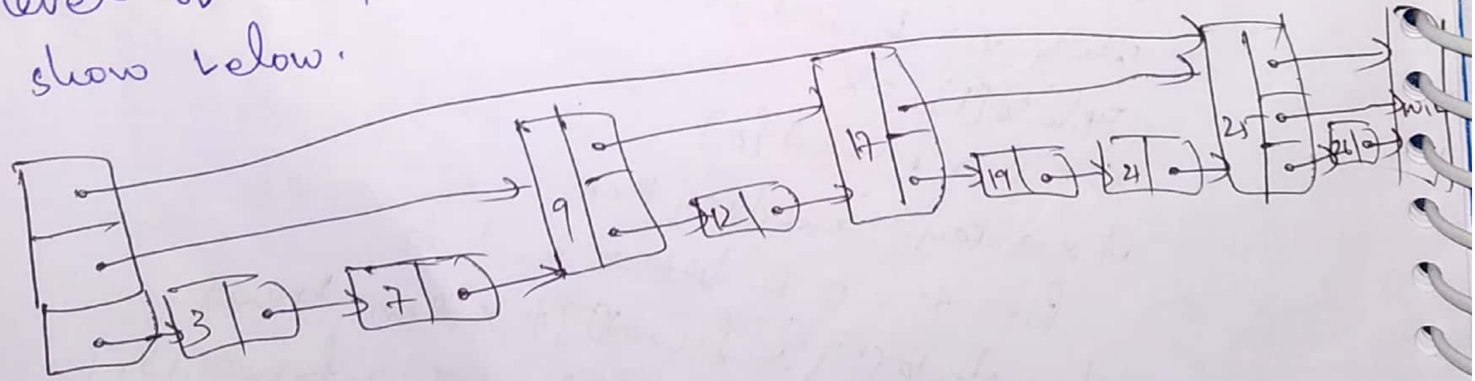


deleted element.

Now after deleting element 6 from skip list



Now Here at level 3, there is no element (allow in blue) after deleting element 6. So we will decrement level of skip list by 1. Now level will be '2' show below.



searching an element in skip list:

searching an element is very similar to approach for searching a spot for inserting an element in skip list. The basic idea is if

(1) key of next node is less than search key then we keep on moving forward on the same level.

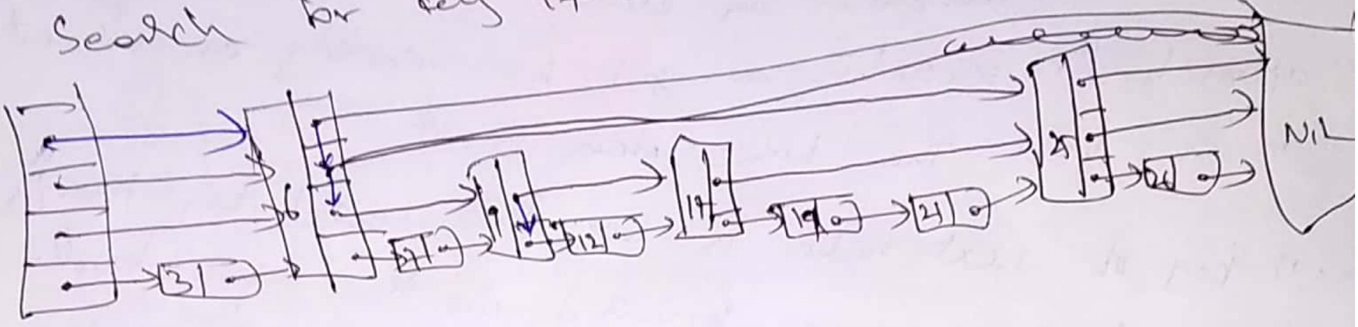
(2) key of next node is greater than the key to be inserted then we store the pointer to current node i at update[i] & move one level down node & continue our search.

At the lowest level (0), if the element next to the rightmost element (update[0]) has key equal to the search key, then we have found key otherwise failure.

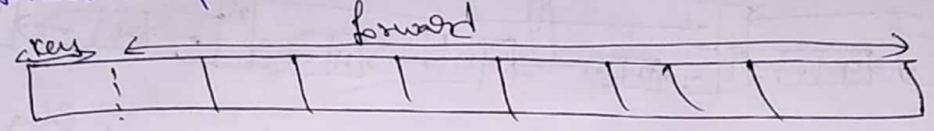
Following is the code for searching element

```
search(list, searchkey)
x := list -> header.
-- loop invariant : x -> key level downto 0 do
while x -> forward [i] -> key forward [i]
x := x -> forward [0]
if x -> key = search key then return x-value
else return failure.
```

Consider this example where we want to Search for key 17 -



Node structure: Each node carries a key and a forward array carrying Pointers to nodes of a different level. A level i node carries i ~~forwa~~ forward pointers indexed through 0 to i.



Node.

Inseltion in skip list:

we will start from highest level in the list we compare key of next node of cullent node with the key to be inselted.

Basic Ideaa is if -

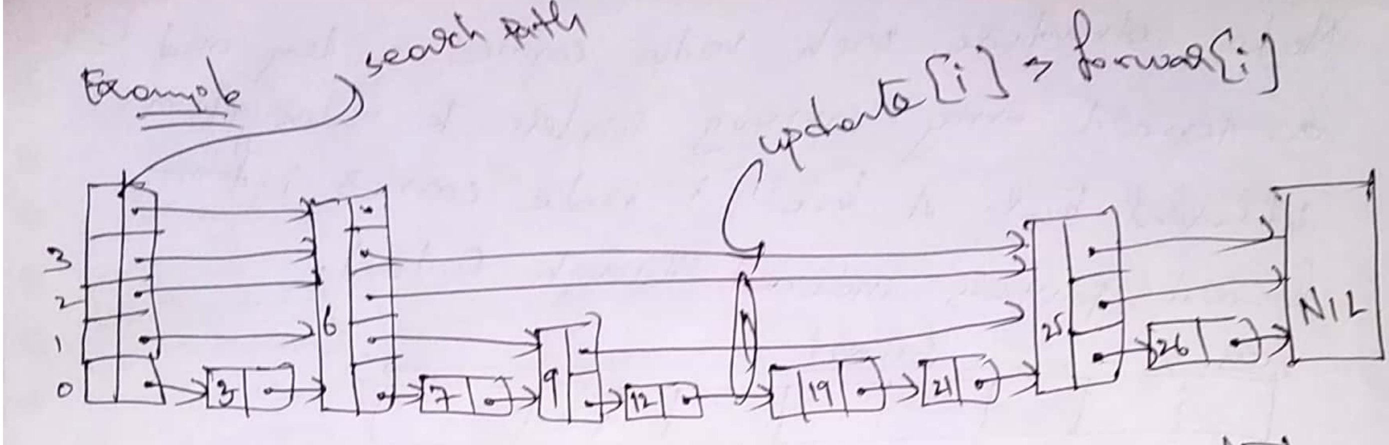
(1) key of next node is less than key to be inselted then we keep on moving forward on the same level.

(2) key of next node is greater than the key to be inselted then we store the pointer to cullent node i at update [i] we move one level down we continue our search.

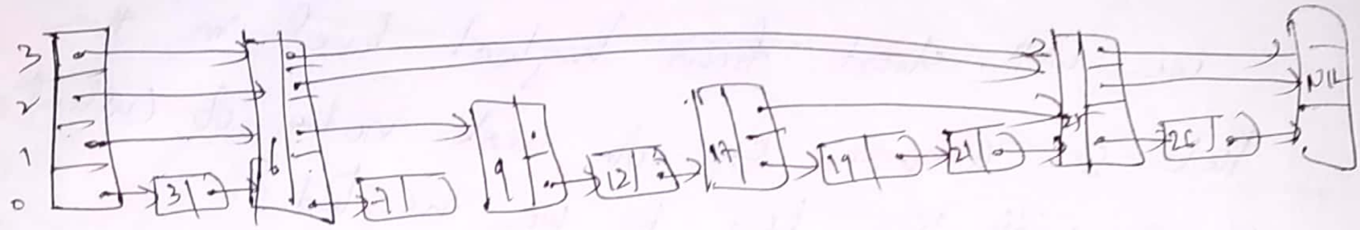
At the level 0, we will definitely find a position to inselt given key. Following is

Here update [i] holds the Pointer to node at level i from which we moved down to level i-1 we Pointer of node left to inseltion Position at level 0. Consider this example where we want to insert key 17.

Example



original list 17 to be updated



level	0	1	2	3
Pointed	12	6	6	6

update array

skip list

Type	list
Invented	1989
Invented by	W. pugh.

Time complexity in big O notation.

Algorithm	Avelage	worst cas
space	o(n)	o(nlogn)
search	o(logn)	o(n)
Insert	o(logn)	o(n)
Delete	o(logn)	o(n)

Applications of skip list

* skip list are used in distributed applications. In distributed systems, the nodes of skip list represented the computer systems and Pointers represent n/w connection

* skip list are used for implementing highly scalable concurrent priority queues with less lock contention

What is Hashing:

Hashing is a technique for pelforming almost constant time in case of insertion deletion and find opelation

Hashing is a technique that is used to uniquely identify a specific object from a group of similar objects.

mapping key must be simple to compute we must help in identifying the associated records.

Function that help us in genelating such type of key is termed as Hash function.

Need of Hashing: Hashing is the process of mapping large amount of data item to a smaller table with the help of a hashing function. To essence of hashing is to facilitate the next level searching method when compared with the linear or binary search

The advantage of this searching method is its efficiency to hand vast amount of data items in a given collection (i.e collection size) Due to this hashing process, the result is a Hash data structure that can stores or retrieve data items in an avelage time disregard to the collection size.

Hagh Table: Hagh table support one of the most efficient type of searching. Fundamentally a hagh table consists of an allray in which data is accessed via a special index called key.

Application of Hagh table:

* Database system
* symbol table in compiler
* Tagged buffer etc.

Hash table:-

Hash table is a data structure which stores data in an associative manner. In a hash table, data is stored in an array format, where each data value has its own unique index value. Thus, it becomes a data structure in which insertion and search operations are ~~data~~ very fast irrespective of the size of the data

Hash Table uses an array as a storage medium and uses hash technique to generate an index where an element is to be inserted or is to be located from

Hashing:

Hashing is an important Data structure which is designed to use a special function called the Hash function which is used to map a given value with a particular key for faster accesss of elements. The efficiency of mapping depends of the efficiency of the hash function used.

let a hash function H(x) maps the value x at the index x%10 in an Array.

For example if the list of value is [11, 12, 13, 14, 15] it will be stored at positions {1, 2, 3, 4, 5} in the array or Hash table respectively.

Hashing Data structure

list = {11, 12, 13, 14, 15}

$H(x) = [x \% 10]$ X = key

$H(x) = 11 \% 10$

10) 11 (1
10
1

Then '11' is placed in the place of '1' (index)

0	1	2	3	4	5
	11				

$H(x) = 12 \% 10$

10) 12 (1
10
2

0	01	2	3	4	5
·	11	12			

$H(x) = 13 \% 10$

10) 13 (1
10
3

0	1	2	3	4	5
	11	12	13		

$H(x) = 14 \% 10$

10) 14 (1
10
4

0	1	2	3	4	5
	11	12	13	14	

$H(x) = 15 \% 10$

10) 15 (5
10
5

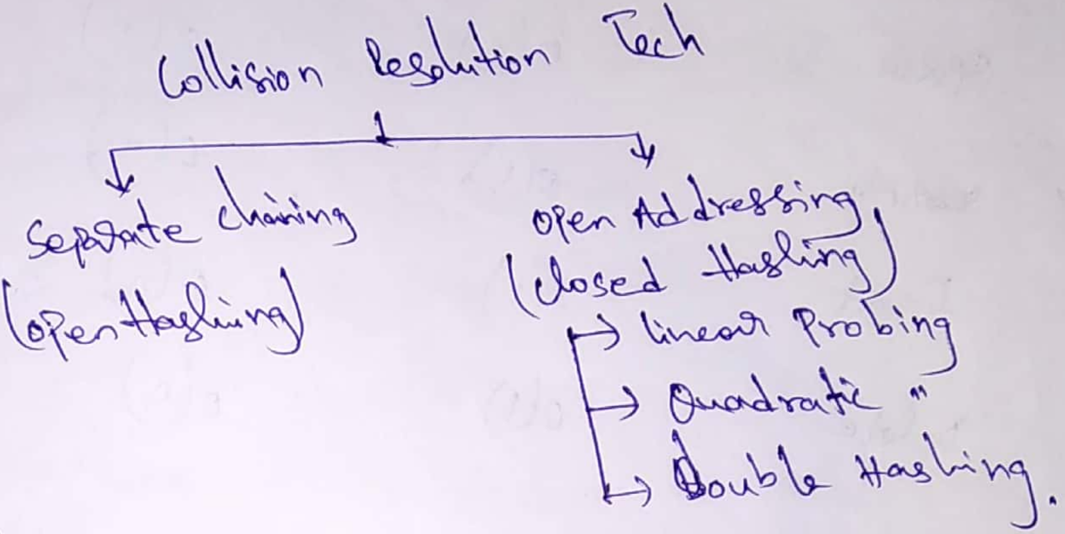
0	1	2	3	4	5
	11	12	13	14	15

Time Complexity in big O notation

Algorithm	Avrage	wast case
space	o(n)	o(n)
search	o(1)	o(n)
Insert	o(1)	o(n)
Delete	o(1)	o(n)

Collision Resolution:- Two key mapping to the same location in the hash table is called "Collision". Collision can be reduced with a selection of a good hash function

Collision resolution techniques are callasified as



*Separate chaining and open Addressing.

Separate Chaining:- To handle the collision.

* This technique creates a linked list to the slot for which collision occurs.
* The new key is then inserted in the linked list.
* These linked lists to the slots appear like chains
* That is why, this technique is called as Separate Chaining

Practic Problem Based on separate chaining

Problem:- using the hash function 'key mod 7' insert the following sequence of keys in the hash table.

50, 700, 76, 85 92, 73 and 101.

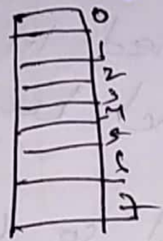
use separte chaining technique for collision resolution The given sequence of key will be inserted in the hash table as.

Scanned by CamScanner

Step-01:- Draw an empty ~~haish~~ table.

* for the given hash function, the possible range of hash value is [0, 6]

* So, draw an empty hash table consisting of 7 buckets as

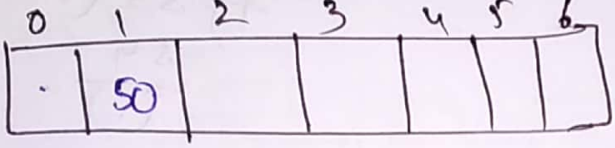


Step-02:- Insert the given keys in the hash table one by one

* The first key to be inserted in the hash table = 50

* Bucket of the hash table to which key 50 maps = 5mod 7 = 1

* so, key 50 will be inserted in bucket of the hash table as



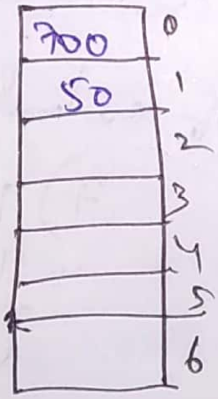
7) 50 (7
49
(1)

Step-03: The next key to be inserted in the hash table = 700

* Bucket of the hash table to which key 700 maps = 700 mod 7 = 0

* So key 700 will be inserted in bucket-0 of the hash table as

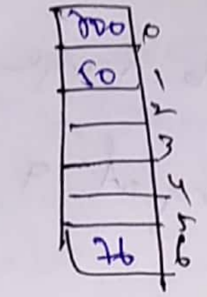
7) 700 (100
700
(0)



Step-04:- The next key to be inserted in the hash table = 76

* Bucket of the hash table to which key 76 maps = 76 mod 7 = 6

* so, key 76 will be inserted in bucket-6 of the ~~tash~~ table



7) 76 (10
70
6

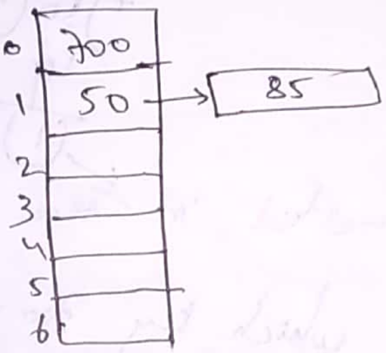
Step-05 :- The next key to be inserted in the hash table = 85

* Bucket of the hash table to which key 85 maps = 85 mod 7 = 1

* since bucket-1 is already ocupied, so collision occurs

* separate chaining handles the collision by creating a linked list to bucket = 1

* so, key 85 will be inserted in bucket-1 of the hash table



7) 85 (12
84
1

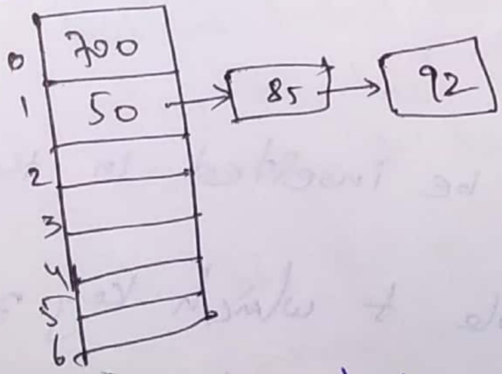
Step-06 :- The next key to be inserted in the hash table = 92

* Bucket of the hash table to which key 92 maps = 92 mod 7 = 1

* since bucket-1 is already ocupied, so collision occurs.

* separate chaining handle the collision by creating a linked list to bucket-1

* so, key 92 will be inserted in bucket-1 of the hash table as

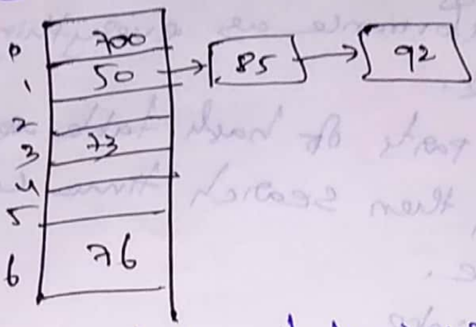


7) 92 (13
91
1

Step-07 :- The next key to be inserted in the hash table = 73

* Bucket of the hash table to which key 73 maps = 73 mod 7 = 3

* so, key 73 will be inserted in bucket-3 of the hash table as



7) 73 (10
70
3

step-08 The next key to be inserted in the hash table = 101

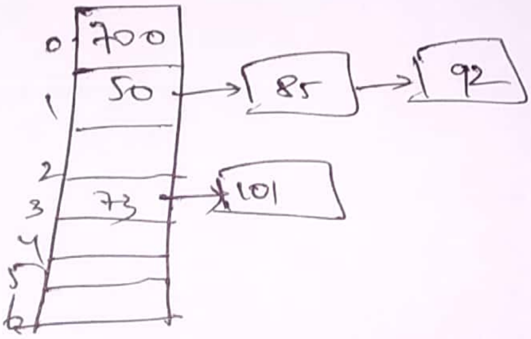
* Bucket of the hash table to which key 101 maps = 101 mod 7 = 3

* Since bucket-3 is already occupied, so collision occurs

* separate chaining handles the collision by creating a linked list to bucket-3

* so, key 101 will be inserted in bucket-3 of the hash table as

7) 101 (14
98
3



Advantage :- The biggest advantage of separate chaining is its collision avoidance capabilites. This means that many data items may be hashed with the same keys creating long link chains

* simple to implement

* Hash table never fills up, we can always add more elements to the chain.

* less sensitive to the hash function or load factors

* It is mostly used when it is known how many & how frequently key may inserted or deleted

Disadvantages:- Cache performance of chaining is not good as key are stored using linked list. open addressing Provides better Cache performance as everything is stored in the same table.

* wastage of Space (some parts of hash table are never used)
* If the chain becomes long, then search time can become O(n) in the worst case.
* uses extra space for links

Time Complexity:- This approach is O(N) where N is the size of the sting.

Difference b/w separate chaining (open Hashing) and open Addressing (closed Hashing.

Chaining	open Addressing
* elements can be stored at outside of the table	* In open addressing elements should be stored inside the table only
* In chaining at any time the no. of elements in the hash table may greater than the size of the hash table	* In open addressing the no. of element present in the hash table will not exceed to no. of indices in hash table.
* In case of deletion ~~chaing~~ chaining is the best method	* If deletion is not required Only inserting & searching is required open addressing is better.
* chaining requires more space.	* open addressing requires less space than chaining.

Hashing Name	Hash Function
Hashing with chaining	$h(k) = k \bmod n$
linear Probing	$h(k,i) = (h'(k) + i) \bmod m$
	$h'(k) = k \bmod m$
Quadratic Probing	$h(k,i) = (h'(k) + c_1 i + c_2 i^2) \bmod m$
	$h'(k) = k \bmod m$
	$h'(k) = k \bmod m$
	c_1 & c_2 are constant
Double Hashing	$h(k,i) = (h_1(k) + i.h_2(k)) \bmod m$
	$h_1(k) = k \bmod m$
	$h_2(k) = k \bmod m$
	Here m is slightly lesser than m (sy (m-1) or m-2))

Open addressing or closed hashing: Is a method of collision resolution in hash tables. with this method a hash collision is resolved by Probing, or searching through alternate locations in the array until either ~~that~~ the target record is found, or an unused array slot is found, which indicates that there is no such key in the table. well known probe sequences include.

Insert: (k): key Probing until an empty slot is found. once an empty slot is found, insert k.

search(k): keep Probing until slot's key doesn't become equal to k or an empty slot is reached

Delete(k): Delete operation is interesting, if we simply delete a key, then search may fail. so slots of deleted keys are marked specially as "deleted".

Insert can insert an item in a deleted slot, but the search doesn't stop at a deleted.

Probing (try

Linear Probing:-

* when collision occurs, we linearly Probe for the next slot(bucket).
* we keep Probing until an empty bucket is found.

In linear probing, we linearly Probe for next slot. For example, typical gap between two Probes is 1 as taken in below example also.

Advantage:- It is easy to compute.

Disadvantage:- The main problem with linear probing is clusting.

* many consecutive elements from groups.
* Then, it takes time to search an element or to find an empty bucket.

Time Complexity:- Worst time to search an element in linear probing is O(table size).

* Even if there is only one element present and all other elements are deleted
* Then, "deleted" marker present in the hash table makes search the entire table.

Alg :-

* use an array of linked list
  - → LinkedList[ ] Table ;
  - → Table = new LinkedList(N), where N is the table size
* Define load Factor of Table as
  - → λ = no. of keys / size of the table
  - (~~*~~ can be more than 1)
* still need a good hash function to distribute keys evenly.
* for search and updates.

Advantages:

* Simple to implement.
* Hash table never fills up, we can always add more elements to the chain.
* less sensitive to the hash function or load factors
* It is mostly used when it is unknown how frequently keys may be inserted or deleted.

Disadvantages:

* cache performance of chaining is not good as keys are stored using a linked list. open addressing provides better cache performance as everything is stored in the same table.
* wastage of space (some parts of hash table are never used).
* If the chain becomes long, then search time can become O(n) in the worst case
* uses extra space for links.

Linear Probing:

The idea:

- * Table remains a simple array of size N
- * on insert(x), compute (f(x) Mod N, if the cell is full, find another by sequentially searching for the next available slot.
  - * Go to f(x)+1, f(x)+2 etc...,
- * on find(x), compute f(x) Mod N, if the cell doesn't match, look else where.
- * linear Probing function can be given by
  → $h(x,i) = (f(x) + i) \bmod N$ (i=1,2 ....)

Clusteling:- The main Problem with linear probing is clusteling, many consecutive elements from groups and it starts taking time to find a free slot or to search an element.

when home address is occupied, go to the next address (current address+1)

In this method, when a collision occurs, the record is placed in the next available bucket. If an empty bucket is not found till the end of the table, then again search for available bucket is done from start of the table upto the home bucket. In other words, when searching for a empty bucket, the table is considered to be circular.

For example:-

let us take the hash function $f(k) = k \% D$

3, 9, 8, 6, 4

$f(3) = 3 \% 5 = 3$

0	1	2	3	4
			3	

Here '3' is insteled in place 3rd place in hash table.

$f(9) = 9 \% 5 = 4$

0	1	2	3	4
			3	9

Here '9' is going to inserted in the place of 4th place in hash table.

$f(8) = 8 \% 5 = 3$

0	1	2	3	4
8			3	9

Here '8' is inserted in the place of

$f(6) = 6 \% 5 = 1$

0	1	2	3	4
8	6		3	9

$f(4) = 4 \% 5 = 4$

0	1	2	3	4
8	6	4	3	9

Example:

Cells $h_0(x)$, $h_1(x)$, $h_2(x)$ .... are tried in succession where

$h_i(x) = (\text{hash}(x) + f(i)) \bmod \text{TableSize}$ with $f(0)=0$

The function f is the collision resolution strategy.

36, 18, 72, 43, ~~6~~93, 47, 40, 76, 55

Hash key = key % table size

Step1:-

36 % 8

$$8\overline{)36}(4$$
$$\underline{32}$$
$$\boxed{4}$$

0	
1	
2	
3	
4	
5	
6	
7	

Now '36' is inserted in the space of 4th place Now Hash table is

				36			
0	1	2	3	4	5	6	7

Step 2:-

18 % 8

$$8\overline{)18}(2$$
$$\underline{16}$$
$$\boxed{2}$$

0	
1	
2	18
3	
4	36
5	
6	
7	

Step 3: The next key to be inserted in the hash table = 72.

* Bucket of the hash table to which key 72 maps = 72 mod 8

8)72(9
72
(0)

72 % 8 = 0

Now hash Table

0	1	2	3	4	5	6	7
72		18		36			

Step 4:- The next key to be inserted in the hash table = 43

* Bucket of the hash table to which key 43 maps = 43 mod 8

8)43(5
40
(3)

43 % 8 = 3

0	1	2	3	4	5	6	7
72		18	43	36			

Step 5:- The next key to be inserted in the hash table = 93

* Bucket of the hash table to which key 93 maps = 93 mod 8

8)93(11
88
(5)

93 % 8 = 5

0	1	2	3	4	5	6	7
72		18	43	36	93		

Step 6:- The next key to be inserted in the hash table = 47

* Bucket of the hash table to which key 47 maps = 47 mod 8

8)47(5
40
(7)

47 % 8

0	1	2	3	4	5	6	7
72		18	43	36	93		47

Scanned by CamScanner

Step 7:- The next key to be inserted in the hash table = 40

* Bucket of the hash table to which key 40 maps = 40 mod 8

* since bucket (hash table) is already occupied, so collision occurs

* when collision occurs, the record is placed in the next available bucket (hash table)

8)40(5
40
(0)

Then 0th position is already occupied with some data it searches Next empty position in the hash table. Now 1st position is empty it placed (inserted) in the 1st position.

0	1	2	3	4	5	6	7
72	40	18	43	36	93	20	47

Step 8:- The next key to be inserted in the hash table = 76

* Bucket of the hash table to which key 76 maps = 76 mod 8

* Since bucket (hash table) is already occupied, so collision occurs.

* when collison occurs the record is inserted in the next available bucket (hash table)

8)76(9
72
(4)

The 4th position is already occupied with some data it searches Next empty Position in the hash table. Now the 6th position is empty it inserted in the 6th position.

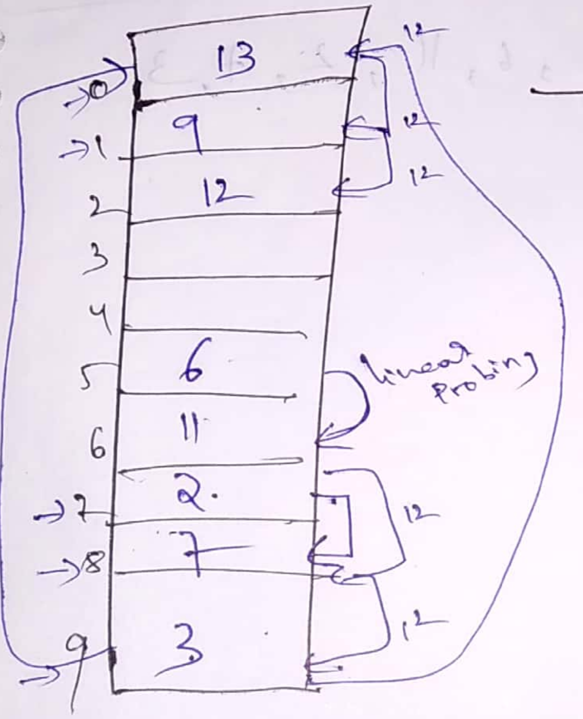
0	1	2	3	4	5	6	7
72	40	18	43	36	93	76	47

Scanned by CamScanner

Linear probing Example:- 3, 2, 9, 6, 11, 13, 17, 12

use Division method & open addressing to Store these Values

$\boxed{h(k) = 2k+3}$ $\boxed{m=10}$



key	location(u)	Probes
3	((2x3)+3)% 10 = 9	1
2	((2x2)+3)% 10 = 7	1
9	((2x9)+3)% 10 = 1	1
6	((2x6)+3)% 10 = 5	1
11	((2x11)+3)%10 = 5	2
13	((2x13)+3)% 10 = 9	2
7	((2x7)+3)% 10 = 7	2
12	((2x12)+3)% 10 = 7	6

Step 1:- 3 → (2x3)+3 ⇒ 6+3 ⇒ 9 % 10 = 9

Step 2:- 2 → (2x2)+3 ⇒ (4+3)% 10 ⇒ 7 % 10 = 7

Step 3:- 9 → ((2x9)+3)% 10 ⇒ (18 +3)% 10 ⇒ 21 % 10 ⇒ 1

10)21
20
(1)

Step 4:- 6
[(2x6)+3]% 10
[12+3] % 10
15 % 10 ⇒ 5
10)15
10
5
Collision

Step 5:-
11 → [(2x11)+3]% 10
[22+3]% 10
25% 10 = 5
10)25
20
5
Collision occured

Step 6:-
13 → [(2x13)+3]% 10
3·(36 +3) % 10
39 % 10 = 9
10)39
30
9
(collision occured linear search

Step :7:- 7→ $[(2\times7)+3]\%10$

$[14+3]\%10$

$17\%10 = 7$

$10)\overline{17}(1$
10
$\overline{7}$

Collision occured

Step 8:12 → $[(2\times12)+3]\%10$

$[24+3]\%10$

$27\%10$

$= 7$

Order:- 13, 9, 12, –, –, 6, 11, 2, 7, 3, Quadratic Probing:— This method of resolving collision uses the following formula.

Quadratic probing is an open addressing scheme in computer programming for resolving collision in hash tables. when an incoming data's hash value indicates it should be stored in an already occupied slot or bucket. Quadratic probing operates by taking the original hash index and adding successive values of an arbitrary quadratic polynomial until an open slot is found.

Quadratic Probing is similar to linear probing. The difference is that if you were to try to insert into a space that is filled you would first check $1^2=1$ element away then $2^2=4$ elements away then $3^2=9$ element away then $4^2=16$ elements away and so on.

With linear probing we know that we will always find an open spot if one exists (it might be a long search but we will find it). However this is not the case with quadratic probing unless you take care in the choosing of the table size. For example consider what ~~happ~~ would happen in the following situation.

Table size is 16. First 5 pieces of data that all hash to index 2.

* First piece goes to index 2.
* First piece goes to 3 ((2+1) % 16)
* Second piece goes to 6 ((2+4) % 16)
* Third " " " ((2+9) % 16)
* Fourth " " " "

* fifth piece doesn't get inserted belonge (2+16) % 16 == 2 which is full so we end up back where we started and we haven't searched all empty spots.

In order to guarantee that you quadratic probes will hit every single available sporty eventually, your table size must meet these requirements.

* Be a prime number.
* never be more than half full (even by one element)

$$h(x, i) = (h(x) + i^2) \bmod m$$

where m is the hash table size

and $i = 0, 1, 2, \text{----} m-1$

Disadvantage:- Can suffer from secondary clusteing. If two keys have the same initial probe position then their probe sequences are the same. Insertion sometimes fails although the table still has free fields.

```
Void insert (key , r[ ])
{ int n;
   int i, last ;
    i = hashfunction(key) ; /* Computes h(x) */
    last = (i+m-1) % m ;
    while (i! = last && ! empty (r[i]) && !deleted(r[i])
                    && r[i]!= key )
      i = (i*i+1) % m;
     if (empty (r[i] || deleted (r[i]))
      r[i] = key ; /** insert here ** /
      else Error ; /** ~~Error~~ table full or key already
                     in table **/
    }
```

Advantages:- Compared to linear probing access becomes inefficient at a high load factor.

~~Example:-~~

Alg:-

* Another open addressing method.

* Resolve collision by examining certain cells (1, 4, 9, ...) away from the original probe point

* Collision policy:

Define $h_0(x), h_1(x), h_2(x), h_3(x) \ldots$

where $h_i(x) = (hash(x) + i^2) \bmod size$

* Caveat:

* may not find a vacant cell!

* Table must be less than half full $(\lambda < 1/2)$

* linear probing always finds a cell).

Example:- Assume a table has 10 slots . using Q.P insert the following elements in the given order 89, 18, 49, 58 and 69 are inserted into a hash table.

0	49
1	
2	
3	69
4	
5	
6	
7	58
8	18~~89~~
9	89

Now

step 1:-

10) 89 (8
80
$\boxed{9}$

then 89 is occupied 9th position in the hash table.

step 2:-

10) 18 (1
10
$\boxed{8}$

Now 18 is occupied by '08' position is the hash table

Step 3:-

10) 49 (4
40
$\boxed{9}$

Now 49 is occupied by '49' in the Position 9th place but 9th position already occupied by '89' Now

Collision occured.

Insert at $9 + 1^2 = 9+1 \Rightarrow 10 \Rightarrow 1+0 = 0.$

0th Position 49 is occliped.

||ly already 8th position is occupied

Step 4:-

10) 58 (5
50
$\boxed{8}$

Now Insert 58=8 is occupied he next 3 locations are occupied so $8+3^2 = 8+9 = 17 \Rightarrow 7$ location

Step 5:- 69 % 10 = 9

10) 69 (6
60
$\boxed{9}$

2 attempts $- 2^2 = 4$ spots

$9 + 2^2 \Rightarrow 13 = 3$ or 4 places.

.

.

Disadv:- Time required to square the probe number

$(n+1)^2 = n^2 + 2n + 1$ increment factor.

Example: solution in Quadeatic probing

Itelatively check

$(hash(key) + i^2)$ Mod N

$h(x) = x \bmod 10$

key = (2, 12, 22, 32)

Step 1:- hash = (2, 2, 2, 2)

0	1	2	3	4	5	6	7	8	9
		2							

Step 2:

10) 12 (1
10
$\boxed{2}$

To store 10 value in table we have to apply formula $(hash(key) + i^2)$ Mod N

Now hash(key) is '2'

$2 + 1^2 = 2 + 1 = 3$ Now 10 is going to store in 3rd position.

0	1	2	3	4	5	6	7	8	9
		2	12						

Step 3:- Now 22 value again it is pointing to the location 2nd then again we have to apply Quadratic probing

10) 22 (2
20
2

Now check $2+1^2=3$ false

$2+2^2 = 2+4 = 6$ Now 22 is going to store in the position '6th'

0	1	2	3	4	5	6	7	8	9
		2	12			22			

Step 4:- Now 32 value again it is pointing to the 2nd location. ~~The~~ 2nd position is not empty. apply

10) 32 (3
30
2

Now

* $2+1^2 = 3$ false * $2+2^2 = 2+4 = 6$ false.

* $2+3^2 = 2+9 \Rightarrow 11$

10) 11 (1
10
1

11 is eaual to 1.

0	1	2	3	4	5	6	7	8	9
	32	2	12			22	.		

Step 5:- Now we want add '42' then we will

10) 18 (1
10
8

$2+1^2=3$ $2+4^2 = 2+16$

$2+2^2=6$ $= 18$

$2+3^2=11$

0	1	2	3	4	5	6	7	8	9
	32	2	12			22		42	

up to n-1 we can go do

Step 6:-

0	~~1~~	2	3	4	5	6	7	8	9
	32	2	12			22	82	42	8

10) 27 (2
29
2

Step 7:-

To add 62

10) 62 (6
60
(2)

$2+3^2 = 11 \% 10 = 1$

$2+4^2 = 18 \% 10 = 8$

$2+5^2 = 27 \% 10 =$

$2+10^2 = 2+100 = 102 \% 10 = 2$

'62' we can not add in to the hash table after doing all iteration

Secondary clusteling :- unable to add an element in Quadratic probing, becauses the index calculated is never empty

This occurs in Quadratic Probing

There are empty space but still we can not enter in the Quadradratic probing

3, 2, 9, 6, 11, 13, 7, 12 i = 0 to (m-1)

h(k) = 2k+3 m = 10

0	13
1	9
2	
3	12
4	
5	6
6	11
7	2
8	7
9	3

Key	location(u)	Probe
3	((2×3)+3) % 10 = 9	1
2	[(2×2)+3] % 10 = 7	1
9	[(2×9)+3] % 10 = 1	1
6	[(2×6)+3] % 10 = 5	1
11	[(2×11)+3] % 10 = 5 = 6	2
13	(13×2)+3] % 10 = 0	2
7	[(2×7) + 3 % 10] = 7	2
12	[(2×12) + 3 % 10] = 3	5

In Quadratic probing $(u+i^2) \% m$

Step 1:

h(k) = 2k+3

= (2×3+3)

=> 6+3 => 9 % 10

=>

Step 2:

h(k) = 2(2) + 3

= 4+3 => 7

= 7 % 10

Step 3:

9 = 2(9) + 3

= 18+3

= 21 % 10

10)21(2
20
1

= 1

Step 4:

6 = [(2)(6)+3]

= 12+3

= 15 % 10 = 5

10)15(1
10
5

Step 5:

(11)(2)+3

22+3

25 % 10

10)25(2
20
5

collision Here.

If collision occurs use the Quadratic Probing

- Insert k_i at first free location from $(u+i^2)\% m$

-where $i=0$ to $m-1$

Now

step 5:- Now $u=5$

$11 \rightarrow (5+0^2)\%10$

$5\%10$

5 already occupied

$(5+1^2)\%10$

$(5+1)\%10$

$6\%10$

6

Step 6:- $13 \rightarrow [(2\times13)+3]\%10$

$26+3=29=9$

Now $(u+i^2)\%10$

$(9+0)\%10$

$9\%10=9$

$(9+1)\%10=0$

Step 7:- $7 \Rightarrow ((2\times7)+3)\%10=7$

$(u+i^2)\%m$

$7+0^2\%10$

$7\%10=7$

$7+i^2\%10$

$7+1\%10$

$8\%10=8$

Step 8:- $[(2\times12)+3]\%10=7$

$(u+i^2)\%m \Rightarrow 7+0=7\%10=7$

$7+1=8\%10=8$

$(7+2^2)\%10$

$(7+4)\%10$

$11\%10=1$

$(7 + 3^2) \% 10$

$(7+9) \% 10$

$16 \% 10 \Rightarrow 6$

$(7+4^2) \% 10$

$(7+16) \% 10$

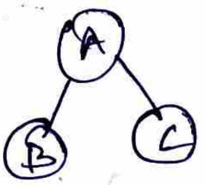
$23 \% 10$

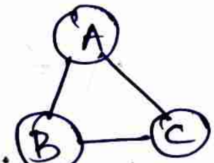
$= 3$

Order 13 , 9, –, 12 , – , 6 , 11, 2, 7, 3

UNIT- III

Tree: A tree is a collection of nodes Connected by directed (or undirected) edges. A tree is a non linear data structure.



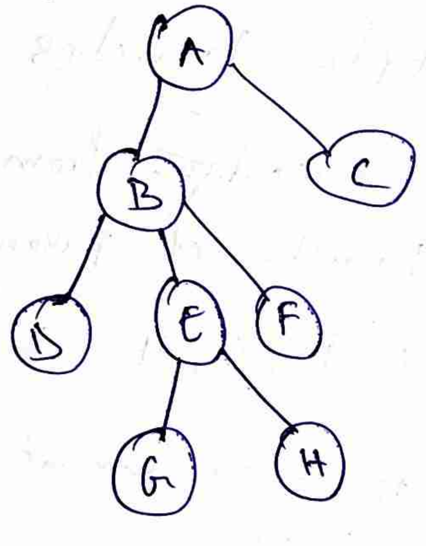


Not a tree bez closed.

Elements - Nodes

we are going to represent by Nodes

Node: Element of a tree is called Node



Simple tree

A can have no. of nodes.

A, B, C, D, E, F G, H are Elements or nodes

Root Node:- starting Node of tree called root node.

In ~~[illegible]~~ example 'A' is a root node
Tree will have only one root

Edge: edge is a link or connection b/w two nodes.

For tree N-nodes it will be having (N-1) edges.

N = No. of nodes

= 8 nodes

E = 7 edges (N-1) => (8-1) => 7

Parent :- Node with branches from top to bottom

In example: A, B, E are parent nodes.

Parent or Node can have multiple branches

Child :- Node with ~~bron~~ edge from bottom to top. (or) Branches of parent.

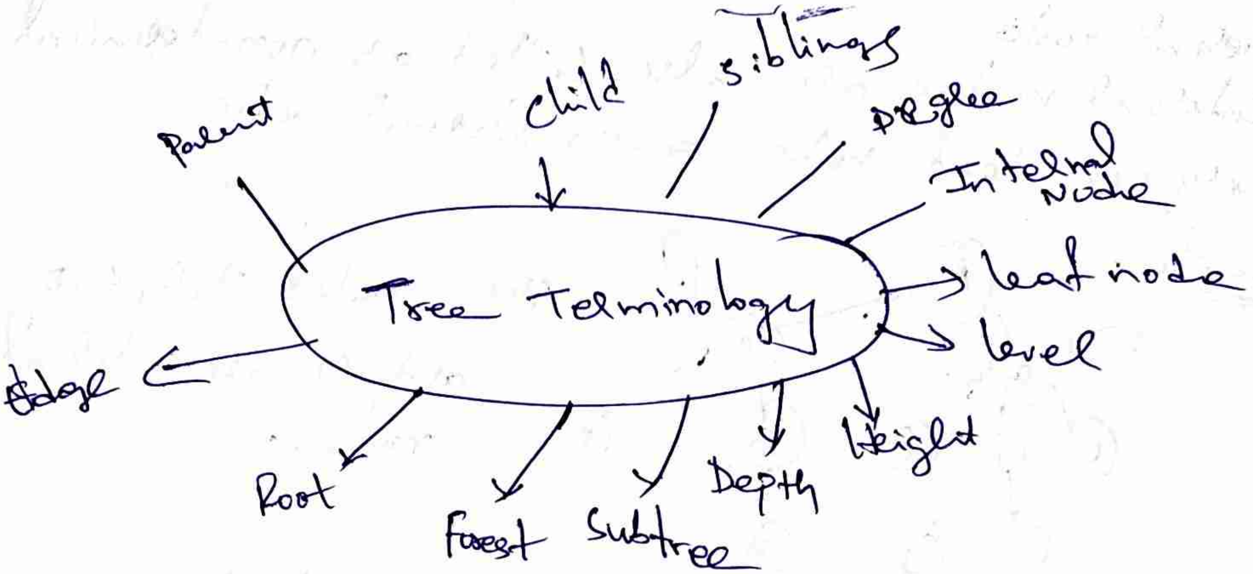
In example B, C, D, E, F, G, H → childs.

siblings :- Child nodes of same parent node

In example
B, C - are siblings
D, E, F - " "
G, H - " "

Leaf :- Node with out child node.

In example :- C, D, F, G, H

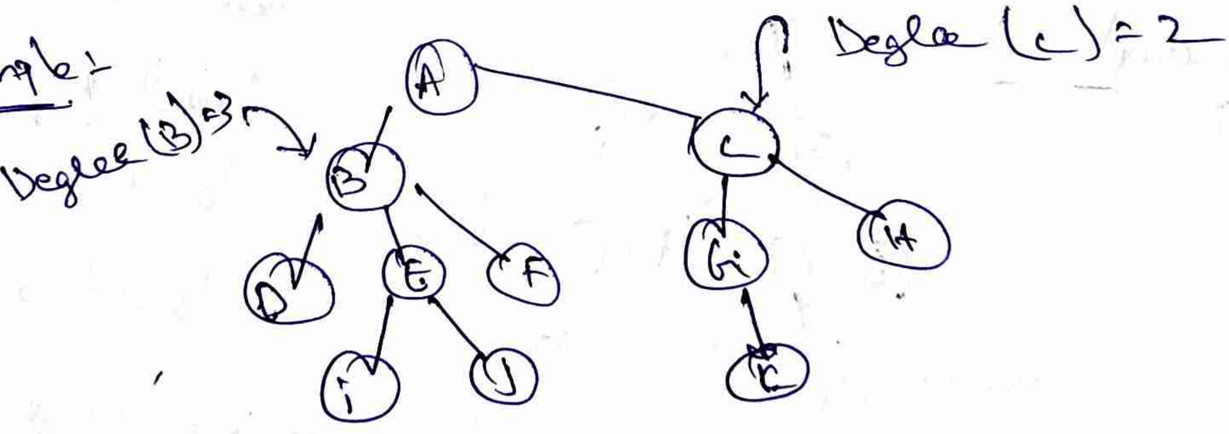


Degree :-

-> Degree of a node is the total no. of children of that node

-> Degree of a tree is the highest degree of a node among all the nodes in the tree.

Example :-



Here :-

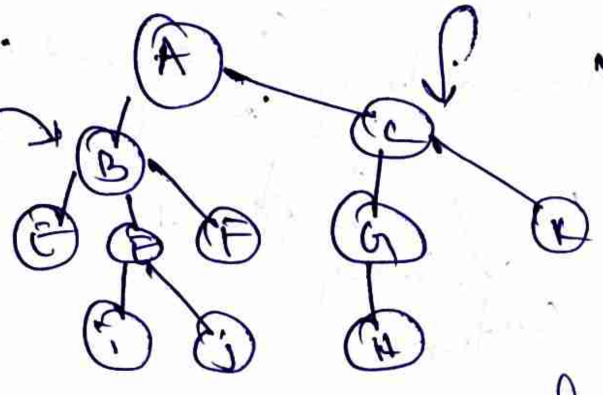
* Degree of node A = 2
" " " B = 3
" " " C = 2
" " " D = 2
" " " E = 2
" " " F = 0
" " " G = 1

Degree of node H = 0
" " " i = 0
" " " J = 0
" " " K = 0

Internal Node:-

* The node which has at least one child is called as an internal node.
* Internal nodes are also called as non-terminal node.
* Every non-leaf node is an internal node.

Example.

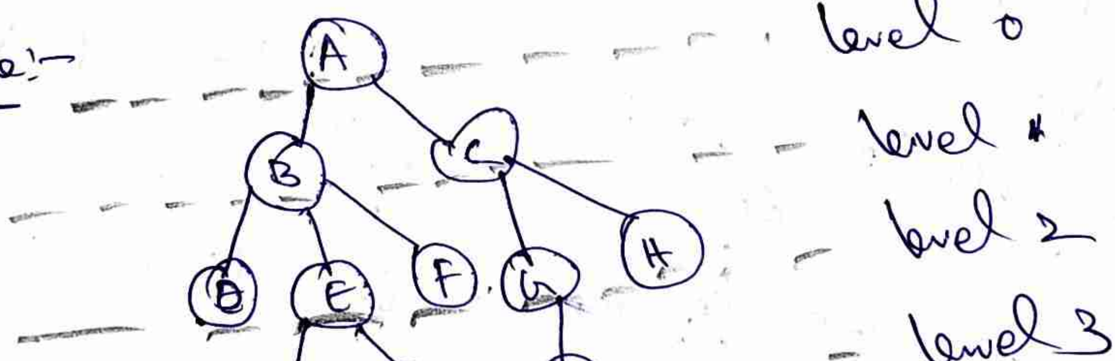


Here nodes A, B, C, E and G are internal node.

Level:- In a tree, each step from top to bottom is called as level of a tree.

* The level count starts with 0 we increments by 1 each level or step.

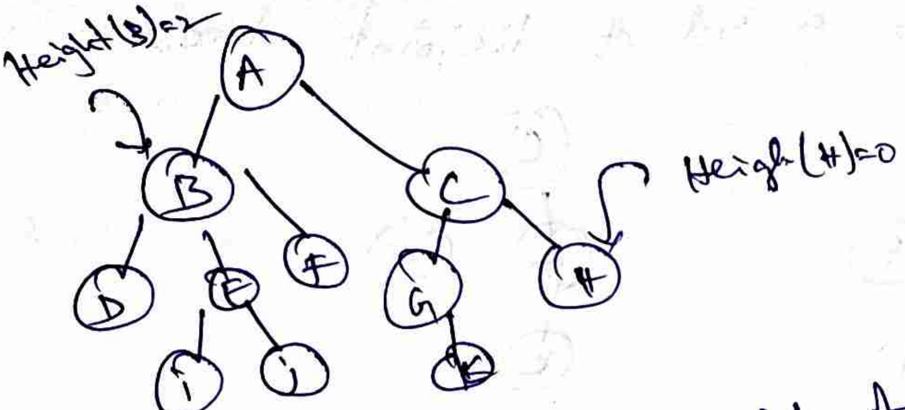
Example:-



Height:- Total no. of edges that lies on the longest path from any leaf node to a particular node is called as height of that node.

* Height of a tree is the height of root node.
* Height of all leaf nodes = 0

Example



Here:-

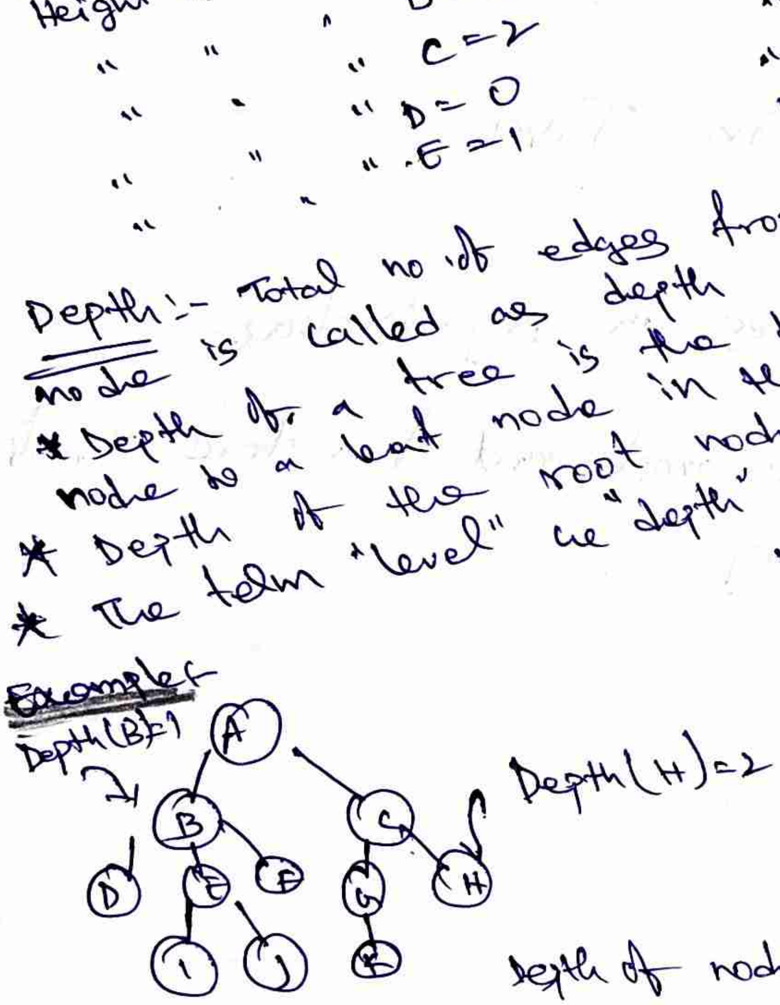
Height of node A =~~2~~3
" " " B =2
" " " C=2
" " " D= 0
" " " E =1

Height of node F = 0
" " " G =1
" " " H =0
" " " I =0
" " " J =0
" " " K =0

Depth:- Total no. of edges from root node to a particular node is called as depth of that node.

* Depth of a tree is the total no. of edges from root node to a leaf node in the longest path.
* Depth of the root node = 0
* The term "level" ie "depth" are used interchangeably.

Example:

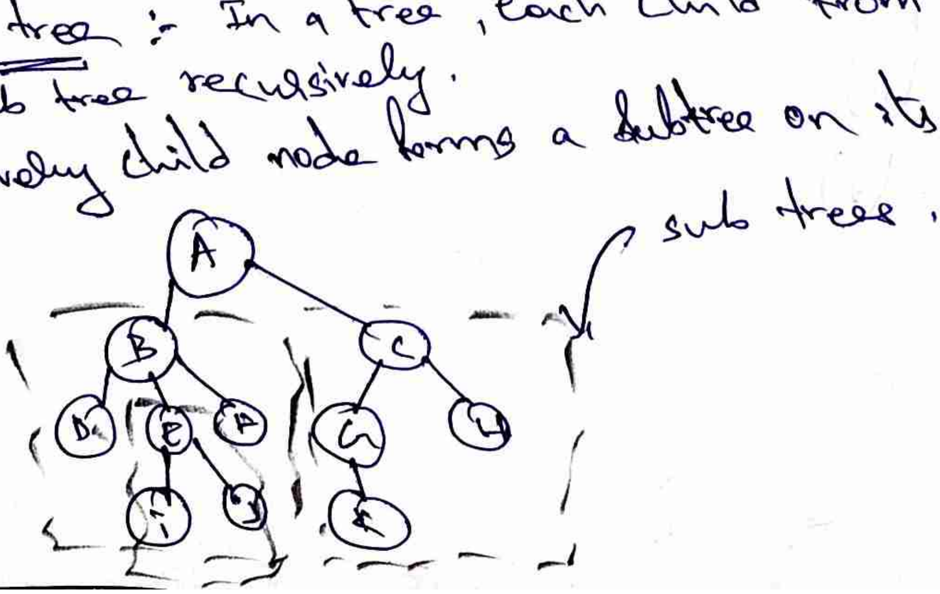


depth of node K=3

Here * Depth of node A =0
" " " B =1
" " " C =1
" " " D = 2
" " " E = 2
" " " F =2
" " " G =2
" " " H =2
" " " I =3
" " " J =3

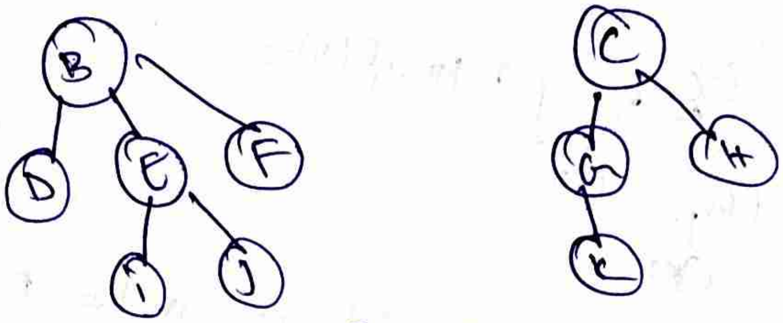
Sub tree :- In a tree, each child from a node forms a sub tree recursively.

* Every child node forms a subtree on its parent node.



Forest:- A forest is a set of disjoint trees.

Example:



forest

Application of trees :-

- * class hierarchy in Java
- * File system.
- * ~~Ex~~ storing hierarchies in organizations.

Tree ADT :- whatever the implement of a tree is, its interface is the following.

- * root ()
- * size()
- * isempty()
- * Parent(v)
- * children(v)
- * isInternal(v)
- * isExternal(v)
- * isRoot()

Binary Tree :- A binary tree is a tree data structure where each node has up to two child nodes, creating the branches of the tree. The two children are usually called the left and right nodes. Parent nodes are nodes with children, ~~other~~ while child nodes may include reference to their parents.

Explains Binary Tree: A binary tree is made up of at most two nodes, often called the left or right nodes, and a data element. The topmost node of the tree is called the root node, and the left or right pointers direct to smaller subtrees on either side.

Binary trees are used to implement binary search trees and binary heaps. They are also often used for sorting data as in a heap sort.

Binary Tree Representation in C: A tree is represented by a pointer to the topmost node in tree. If the tree is empty, then value of root is NULL.

A Tree node contains following parts.

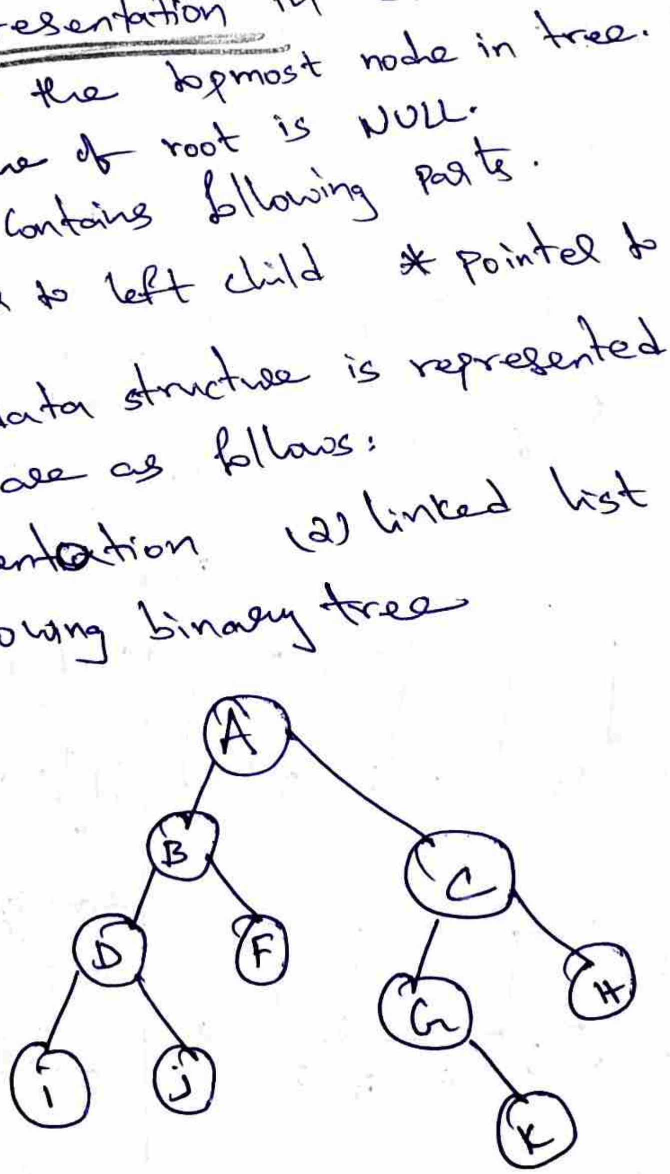
* Data * pointer to left child * pointer to right child

Binary tree data structure is represented using two method

Those Methods are as follows:

1) Array Representation (2) linked list Representation

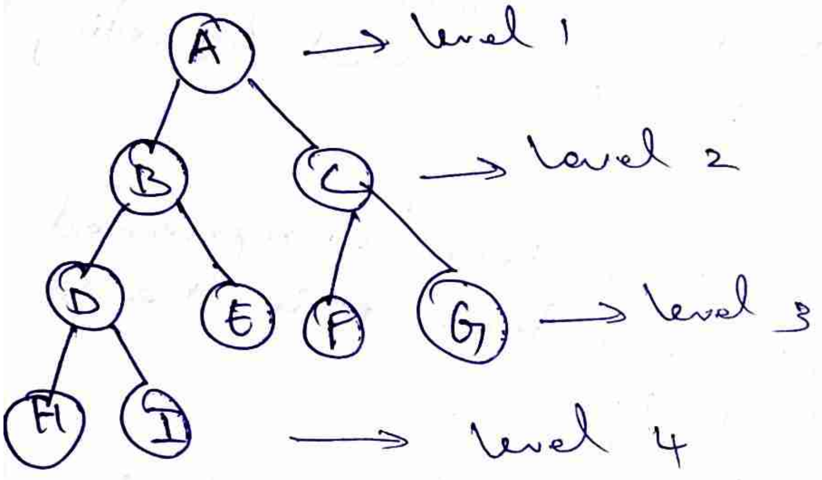
Consider the following binary tree



(1) **Array Representation of Binary Tree:** In array representation of a binary tree, we use one-dimensional array (1-D Array) to represent a binary tree. Consider the above example of a binary tree and it is represented as follows...

A	B	C	D	F	G	H	I	J	-	-	-	K	-	-	-	-	-

To represent a binary tree of depth 'n' using array representation, we need one dimensional array with a maximum size of $2n+1$.



case I

0	1	2	3	4	5	6	7	8

→ we have start from root ~~Node~~ we are going fill the array by level by level and from left to right

step 2

A								
0	1	2	3	4	5	6	7	8

step3 :-

A	B	C						
0	1	2	3	4	5	6	7	8

step 4:-

A	B	C	D					
0	1	2	3	4	5	6	7	8

~~first~~ repeat all the step's complete tree levels up to level 4.

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

In array representation we can not find root node and child node which is root node and which node is child to find that we have

* if a node is at ith index ..
* left child would be at - [(2*i) +1]
* right child would be at - [(2*i) +2]

To find of parent

Parent would be at = $\left(\frac{(i-1)}{2}\right)$

taking
Example i=4

E is present at 4th Position Now we have find Parent of 'E'

Parent would be at = $\left(\frac{(i-1)}{2}\right)$ Now i=4

$= \left(\frac{4-1}{2}\right) \Rightarrow \frac{3}{2} \Rightarrow 1.5 \Rightarrow 1$

Now check in arr at 1st Postion will find Parent of E'

	B			E				
0	1	2	3	4	5	6	7	8

To find out left child - [(2*i) +1]

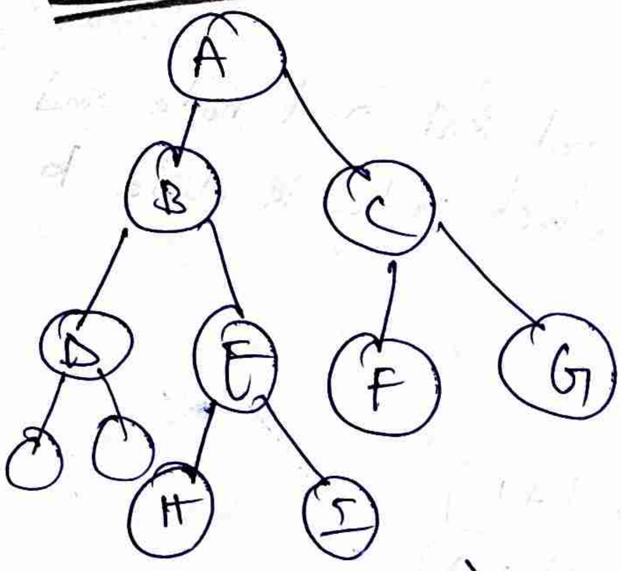
$\Rightarrow$ [(2*4) +1] $\Rightarrow$ 8+1=9

No left child for E.

To find out right child - [(2*i) +2] $\Rightarrow$ (2*4)+2 $\Rightarrow$ 10

No right child for E.

for Example:-



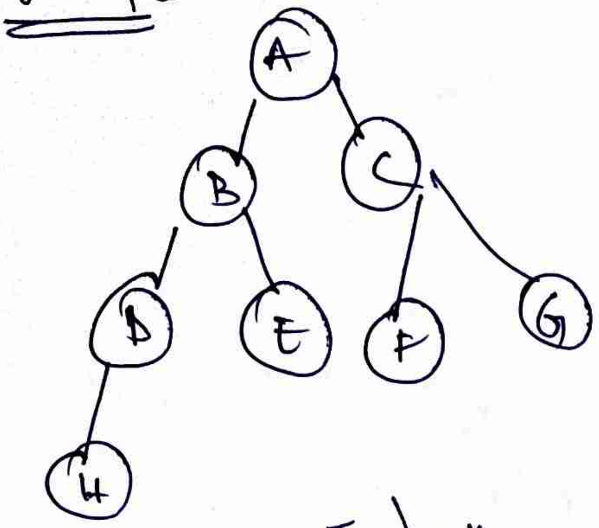
In this example for 'D' No child that y we have give nodes (empty nodes] as shown in the example then only before formulas can be apply.

Now array Index

A	B	C	D	E	F	G	-	-	H	I	
0	1	2	3	4	5	6	7	8	9	10	11

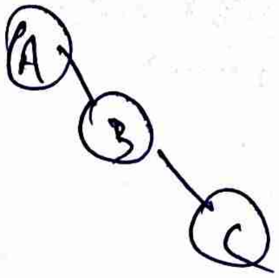
Now we can find out.

Example:-

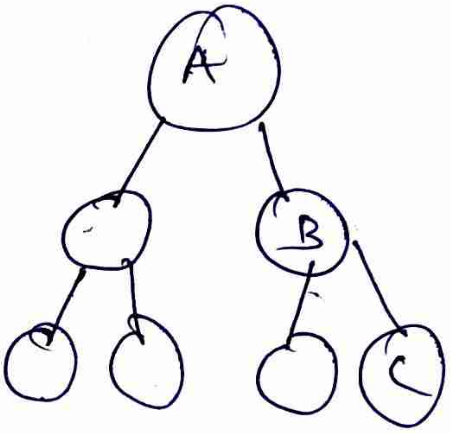


Now array Index

A	B	C	D	E	F	G	H
0	1	2	3	4	5	6	7



Now we have to do Complete Binary tree

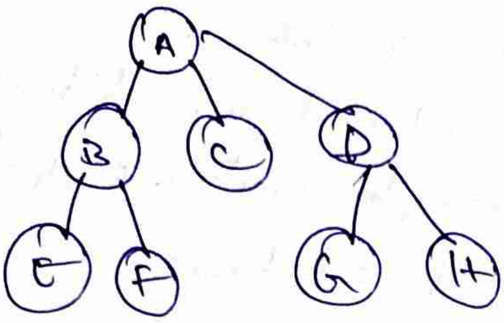


Now array Index

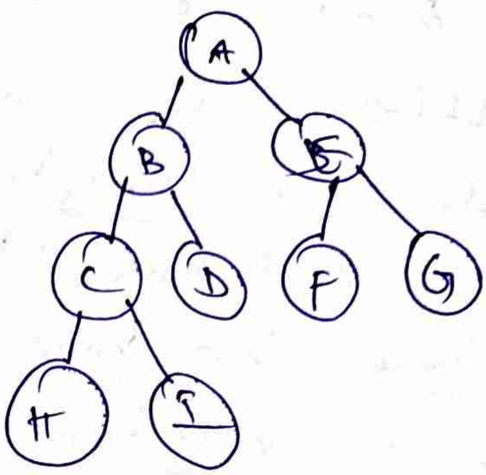
A	-	B	-	-	-	C	~~C~~
0	1	2	3	4	5	6	~~7~~

wast of space.

Binary Tree :- Every node in a tree should have atmost 2 childrens.



Not a Binary tree



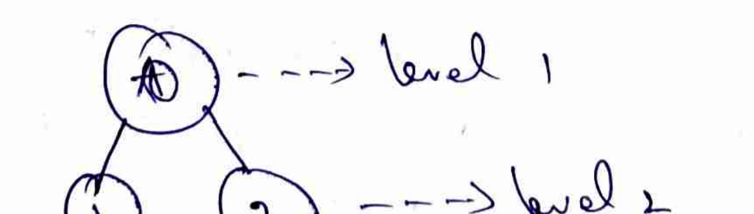
Binary Tree

Every node has '2' childrens.

Different types of Binary Trees:

- * Full Binary tree (strictly Binary tree)
- * Almost Complete " " (In Complete " ")
- * Perfect Binary tree (Complete " ")

- * left skewed " "
- * Right " " "



$2^{3-1} \Rightarrow 2^{2} \Rightarrow 4$ Here 3 is level

2^{i-1} in level i how many nodes = 2^{i-1}

Height :- '3' we are starting from 1.

$2^{3} - 1 = 8 - 1 \Rightarrow 7$

$\boxed{2^{h} - 1}$

The difference b/w binary tree & tree is Each element in binary tree has at most two sub trees (one or both of those sub trees may be empty). Each element in a tree can have any no. of sub trees.

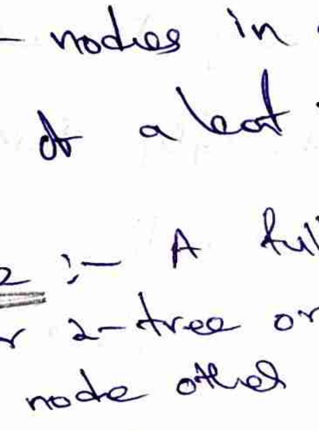
Properties of binary tree:

The maximum no. of nodes at level 'i' of a binary tree is 2^{i-1} (level of a root node is considered as 1)

Maximum no. of nodes in a binary tree of height 'h' is $2^{h}-1$ (Height of a leaf node is considered as 1).

Full binary tree :- A full binary tree (some times proper binary tree or 2-tree or strictly binary tree is a tree in which every node other than the leaves has two children

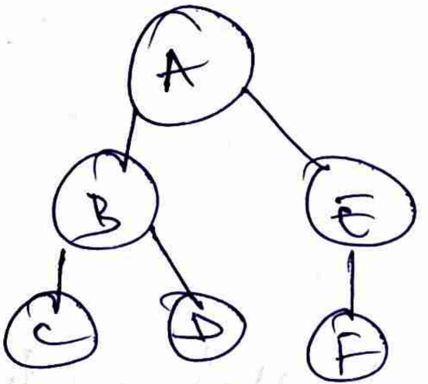
Example:



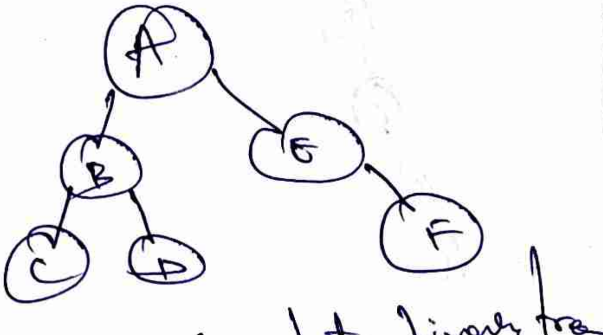
In this fig '0' having '2' childs and '1' having '2' " but '2' has no child still It is known as full binary tree bez having '2' for childs.

It is known as full binary tree.

In Complete binary tree :- A complete binary tree is filled at each depth from left to right in other word A complete binary tree is a binary tree in which every level, except possibly the last is completely filled and all nodes are as far left as possible



Complete binary tree



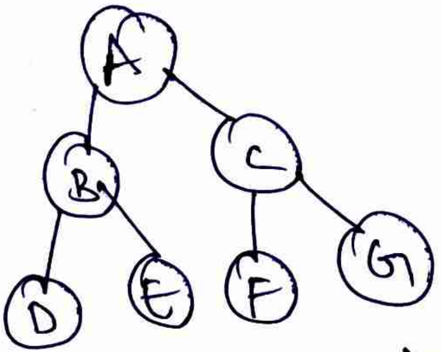
Not Complete binary tree

Perfect binary tree :- A binary tree with all leaf nodes at the same depth. All internal nodes have degree 2

0th level - $2^0 = 1$ node - A

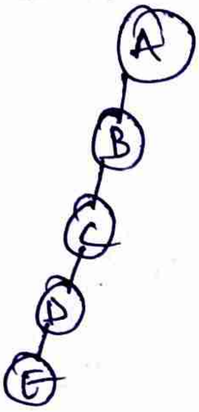
1st level - $2^1 = 2$ nodes - B, C

2nd level - $2^2 = 4$ nodes - D, E, F, G

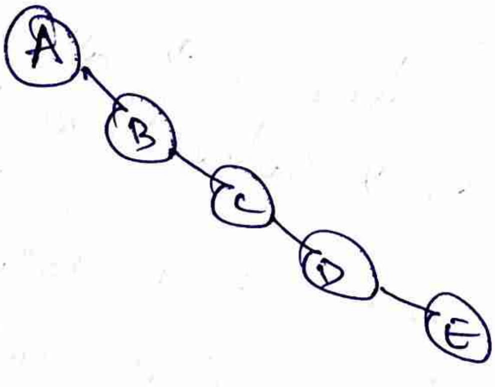


In this perfect binary tree the condition must be stated Each level there must be $\boxed{2^{L} \text{ nodes}}$, L-level.

* left Skewed Binary : Every node should have only left children



* Right Skewed Binary : Every node should have only right children



Binary tree Travelsals :-
(Inorder, Preorder and post order)

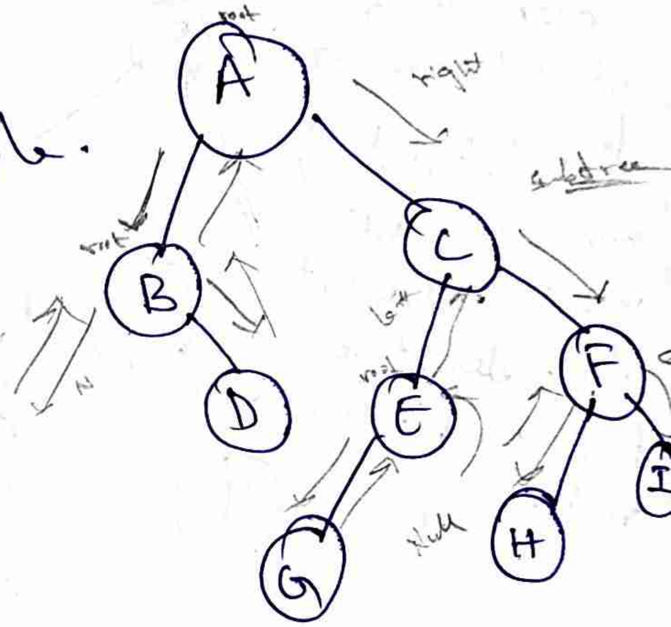
Inorder :- Left - Root - Right

Preorder :- Root - left - Right

Postorder :- left - Right - Root.

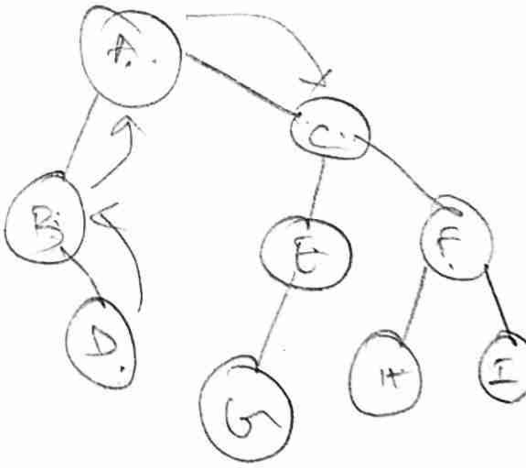
Inorder :- For give example.

B D A G E C H F I



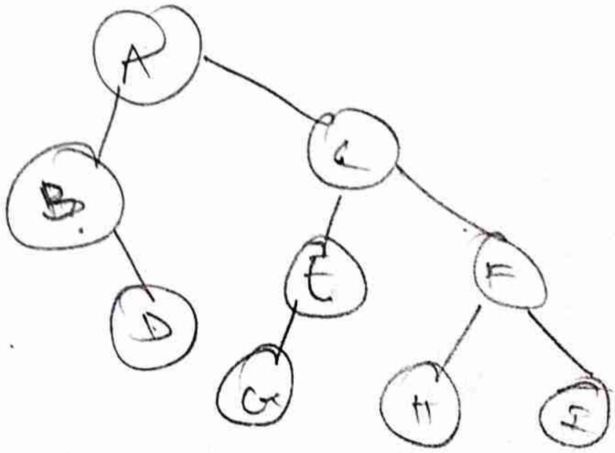
Preorder :-

A B D C E G F H I



Post order

D B G E H I F C A



Construct a Binary tree from Post order & Inorder

Post order :- 9, 1, 2, 12, 7, 5, 3, 11, 4, 8 (L Right Root)

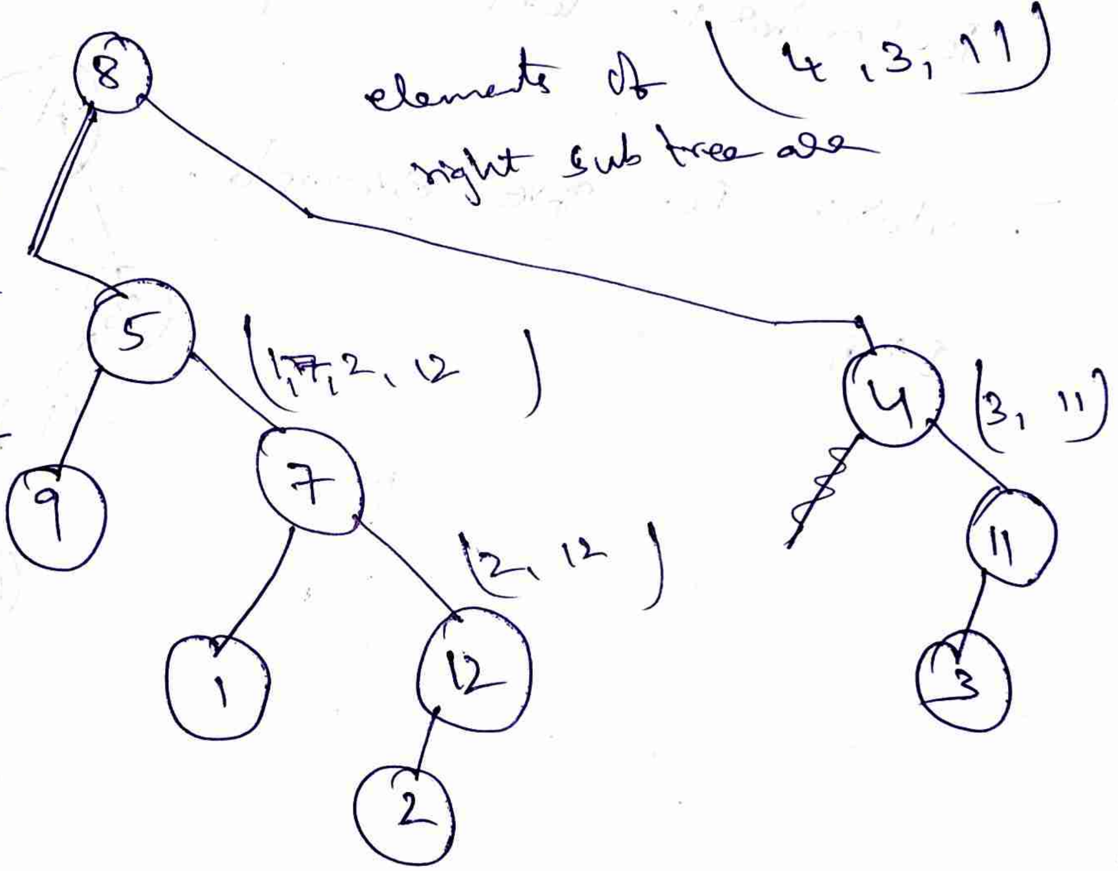
root ↓ 5, root ↓ 8

Inorder :- 9, 5, 1, 7, 2, 12, 8, 4, 3, 11 (L Root Right)

left ↑ root left ↑ root ↑ ↑ root ↑ right

element of left sub tree are (9, 5, 1, 7, 2, 12)

elements of right sub tree are (4, 3, 11)



scan right to left in post order in that which element is first that is the root.

Construct Binary tree form given preorder & Post order

(Root L R)

Pre order : F B A D C E G I H

Post order : A C E D B H I G F

left subtree (L R Root)

righty

(A, C, E, D) B are left sub tree

root

Pre - B A D C E

Post: A C E D B

left Right

(H I G B) are right subtree

Pre - G I H

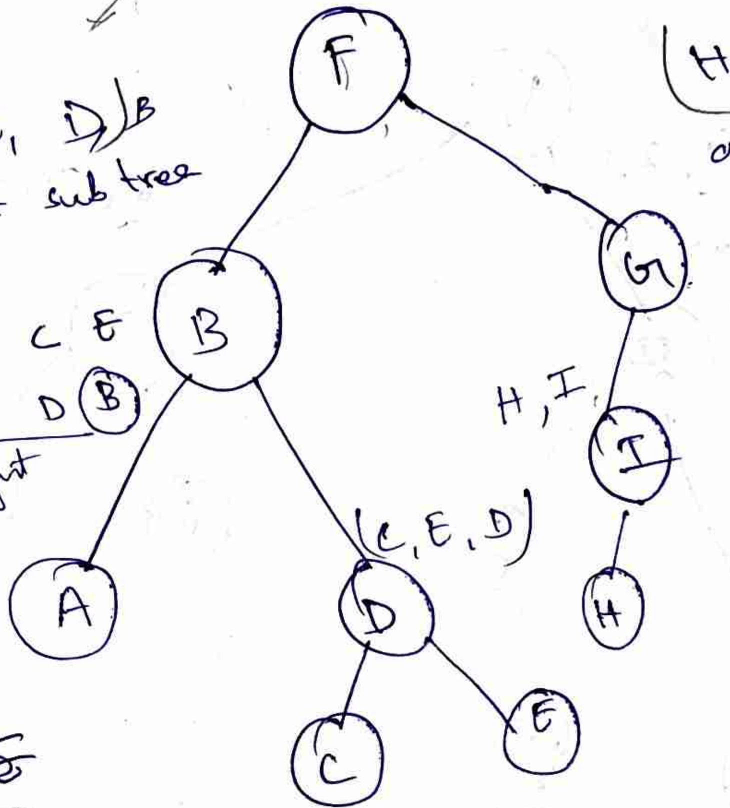
Post H I G

L R

Pree : I H

Post - H, I

left



Pre - D ~~E~~ E

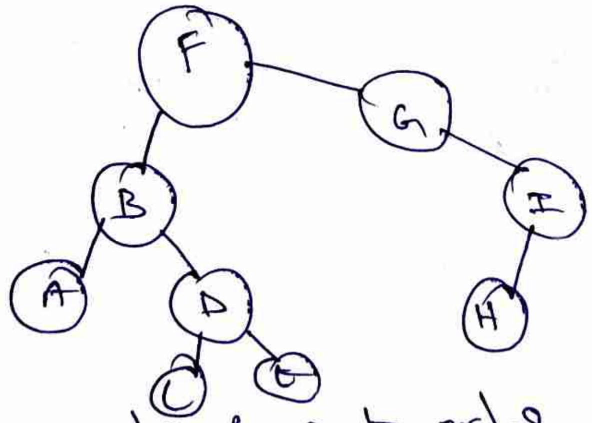
Post - C E D

Right

F Root left Right

left Right Root

Trick :-



For preorder & post order we can not constru

U-quick Binary tree

Construct a Binary tree from Preorder & Inorder

root, 1st

Preorder : 1, 2, 4, 8, 9, 10, 11, 5, 3, 6, 7

left (Root, 2, Right)

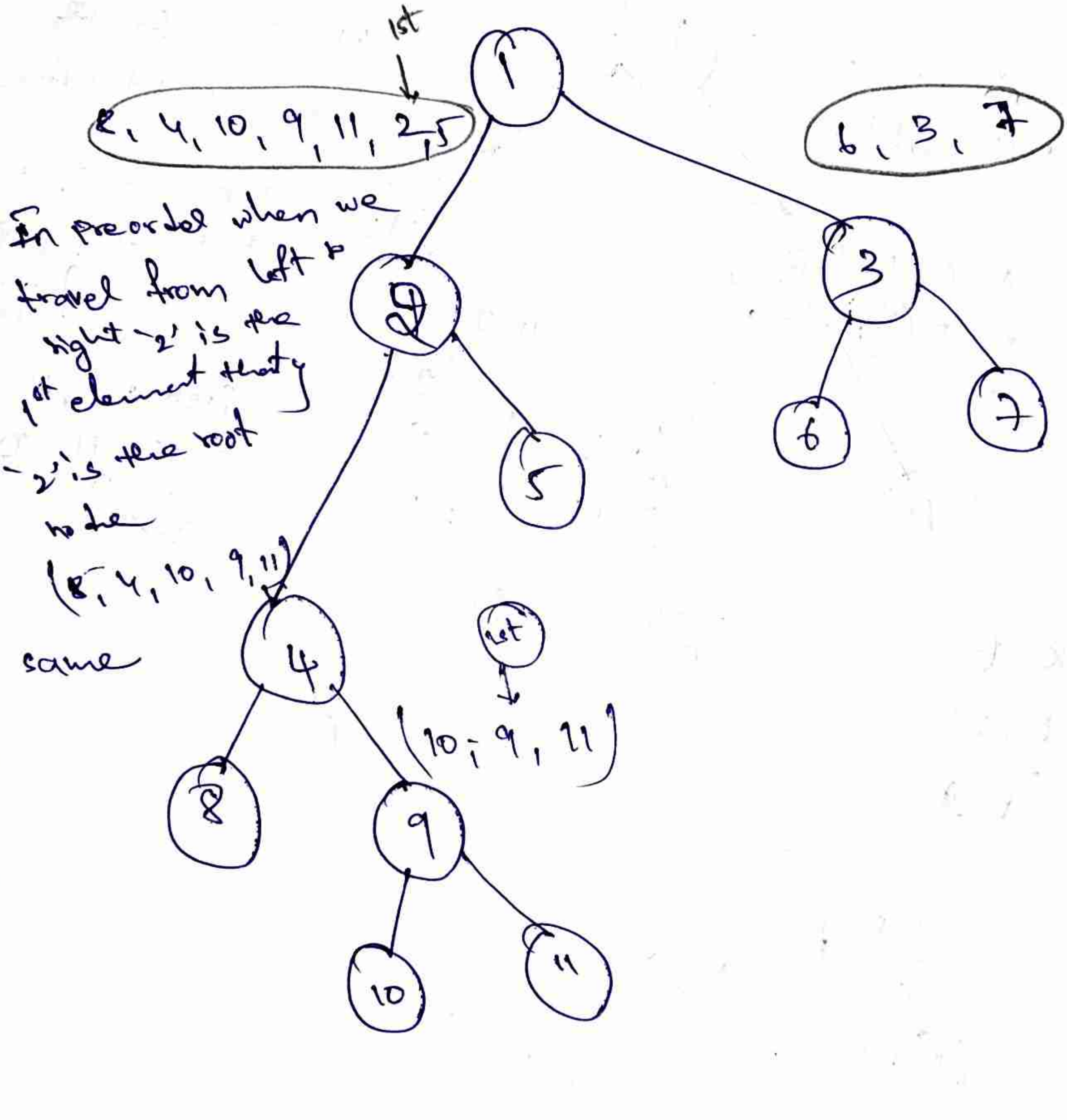
left root right

L r

Inorder : 8, 4, 10, 9, 11, 2, 5, 1, 6, 3, 7

Left Right

(L, Root, Right)



In preorder when we travel from left to right '2' is the 1st element that y '2' is the root node

same

Tree Travelsals

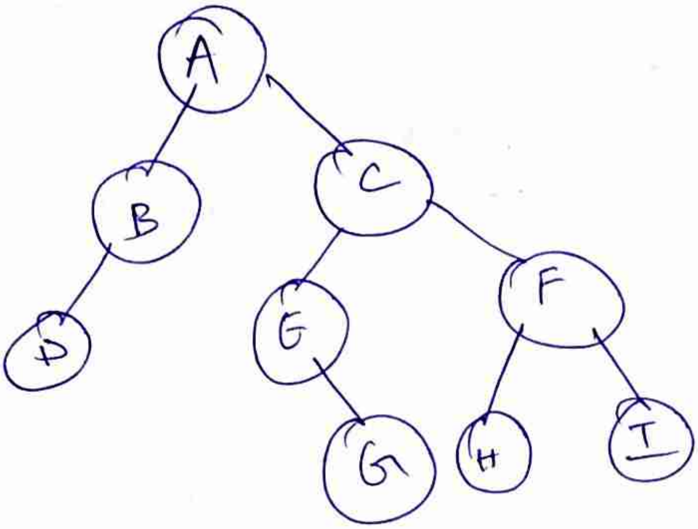
Inorder :- (Left-Root-Right)

D B A E G C H F I

Pre order :-

Root - left - Right

A B D C E G F H I



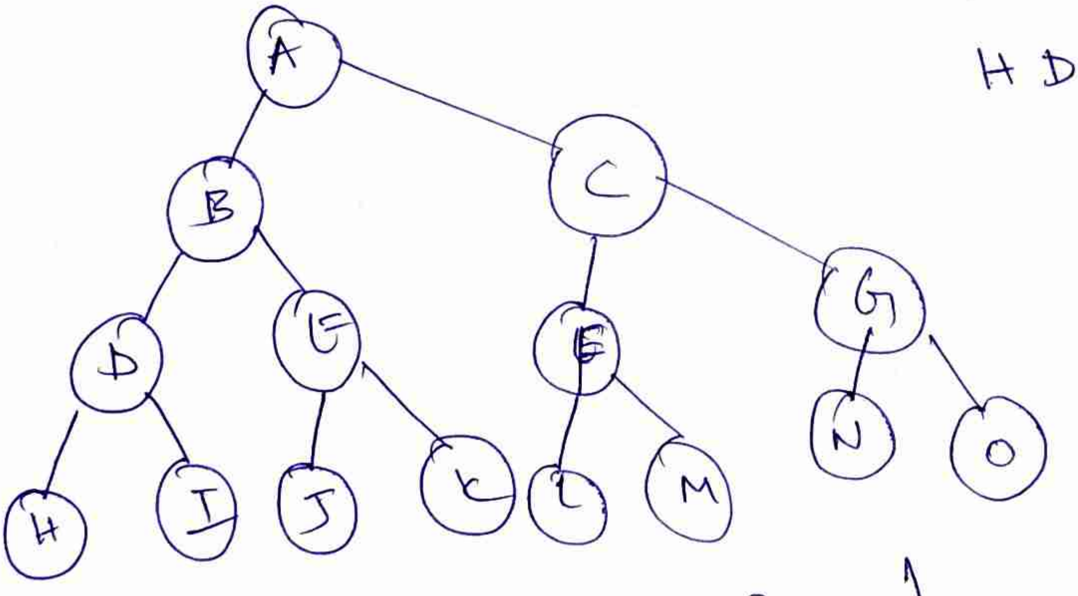
Post order left - Right - Root

D B G E H I F C A

Inorder :- (L - Root - R)

H D I B J E K

A L F M C N G O



Pre order :- (Root - left - Right)

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

Post order :- (left - Right - Root)

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

What is searching:-

- * searching is the process of finding a given value position in a list of values.
- * It decides whether a search key is present in the data or not
- * It is the algorithmic process of finding a particular item in a collection of items.
- * It can be done on internal data structure or on external data structure.

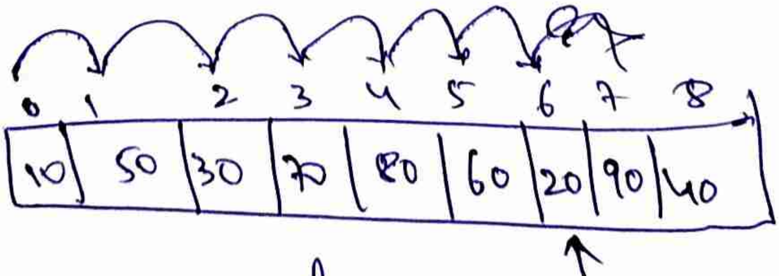
Searching Techniques :-

* sequential search   * Binary search.

sequential search :- In this the list or array is traversed sequentially and every element is checked. For example linear search.

This method can be performed on a sorted or an unsorted list (usually array). In case of a sorted list searching starts from 0th element and continues until the element is found from the list or element whose value is greater then (assuming the list is sorted in ascending order), the value being searched is reached.

Example



0	1	2	3	4	5	6	7	8
10	50	30	70	80	60	20	90	40

6th postion 20 is find.

20 is searching element

In sequential or linear way it is going to ~~set~~ search.

* start from the leftmost element of arr[ ] and one by one Compare x with each element of arr[ ]

* If x matches with an element return the index

* If x doesn't match with any of element, return -1

A linear search scans one item at a time, without juming to any item.

* The worst case complexity is O(n), sometimes known as O(n) search

* Time taken to search elements keep increasing as the no. of elements are increased.

Difference b/w trees and binary trees

Tree.	Binary tree
* Each element in a tree can have any no. of subtrees	* Each element in a binary tree has at most two subtrees
* The subtrees in a tree are unordered	* The subtrees of each element in a binary tree are ordered. (i.e we distinguish b/w left & right subtrees).

Difference b/w linear search & binary search.

linear search	binary search
* The element are in random order	* The element are sorted order
* worst case time complexity $O(n)$	* worst case time complexity $O(\log_2 n)$
* Access is slow	* Access is faster.
* Single & multidimensional Array is sorted used	* only Single dimentional Array is sorted used

linear search:-

0	1	2	3	4	5	6	7
15	5	20	35	2	42	67	17

data to search = 42

n=8

① element is present :-

② Not present

The searching element starts from 0th index we going to cheking each the index one by one if it is maches then it returns index or else it checks next position up to ~~index~~ value or data maches

To seach '42' in the array it checks 0th position 15 is not equal to 42 then next
1 " 5 " " " " " " "
or value maches
it contious up to data or value maches in array

5th index 6th postion 42 is present it display. data is present in the index value 5th. it returns '42' is ~~8~~ element is present.

```
for (i=0 ; i<n; i++)
{
   if (a[i] = = data)
    {
       Printf ("element found at index %d", i);
    } bread;

}

if (i==n)
  {
     printf ("element not found");
```

Time Complexity :-

Best case : O(1)

wast case O(n)

linear searching :- A linear search is the most basic type of searching algorithm. A linear search sequentially moves through your collection (or data structure) looking for a matching value. In other words, it looks down a list, one item at a time without jumping.

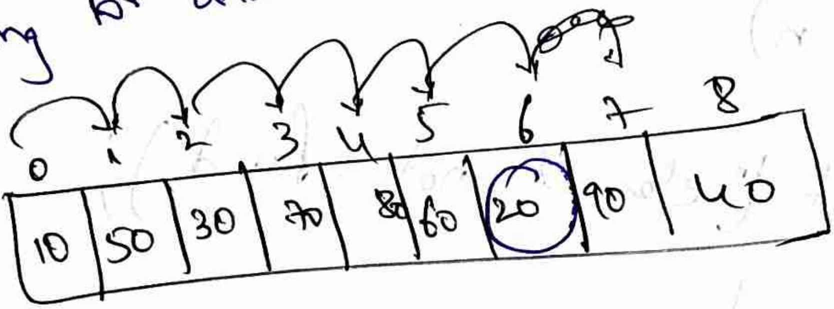
linear search : steps on how it works :-

Here is simple approach is to do linear search.

* start from the leftmost element of array and one by one compare the element we are searching for with each element of the array.

* If there is a match b/w the element we are searching for and an element of the array return the index.

* If there is no match b/w the element we are searching for and an element of the array return -1



0	1	2	3	4	5	6	7	8
10	50	30	70	80	60	20	90	40

find 20

$-1+1 = 1$

$1+1 = 2$ - - - - - 5

Binary search :-

0	1	2	3	4	5	6	7	8	9
5	9	17	23	25	45	59	63	71	89

* In binary searching tech the array ~~show~~ should be in sorted. un sorted array can not apply binary search tech. In above example data is ~~be~~ sorted.

searching : 59

left	right	mid	
0	9	$0+9/2 = 4$	$59 \neq 25$

In index 4 that data is 25 that means data is greater that 25 By that Now we should search the element or data right side.

In this 3 case are there

caseI: data==a[mid]

caseII: data<a[mid]

caseIII: data>a[mid]

left	right	mid
5	9	$5+9/2 \Rightarrow \frac{14}{2} = 7$

Now case is less that 63 Now we can say that before 63 is data is present. The data is present left of mid

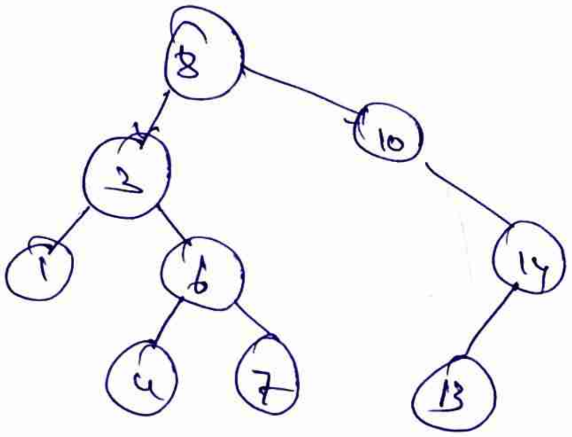
Binary search Tree: Is a node based binary tree data structure which has the following properties:

* The left subtree of a node contains only nodes with keys lesser than the node's key.

* The right subtree of a node contains only nodes with keys greater than the node's, key.

* The left & right subtree each much also be a binary search tree

Example:-



The above properties of Binary search tree provide an ordering among key so that the operations like search, minimum and maximum can be done fast. if there is no ordering, then we may have to compare every key to search a given key.

Searching a key:- To search a given key in Binary search tree, we first compare it with root, if the key is present at root, we return root. if key if greater than root's key, we recur for right subtree of root node. Otherwise we recur for left subtree.

Basic Operation:- Following are the basic operations of a tree.

Search:- searches an element in a tree.

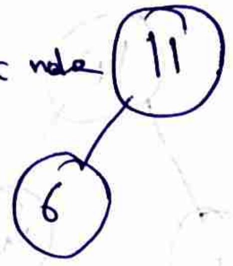
Insert:- Inserts an " " " "

Example:-

11, 6, 8, 19, 4, 10, 5, 17, 43, 49, 3,

step 2:-

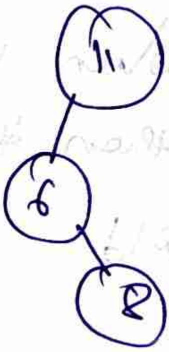
Compare with root node 6 is less of 11 that is left said of '11'



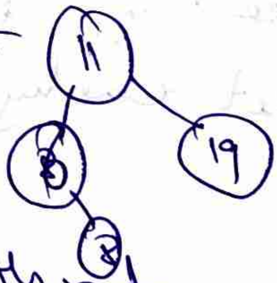
Step 3:-

Compare with root node '8' then 8 is less of of 11 ~~[illegible]~~ then it is left said Now it should be compare with '6' Now here 8 is greater then of '6' then it is place at right of '6'.

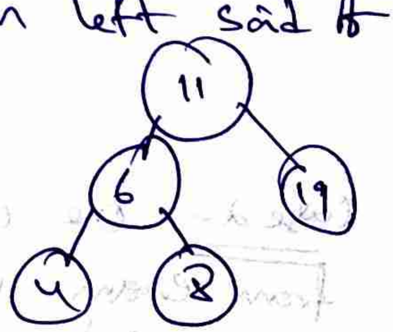
Now



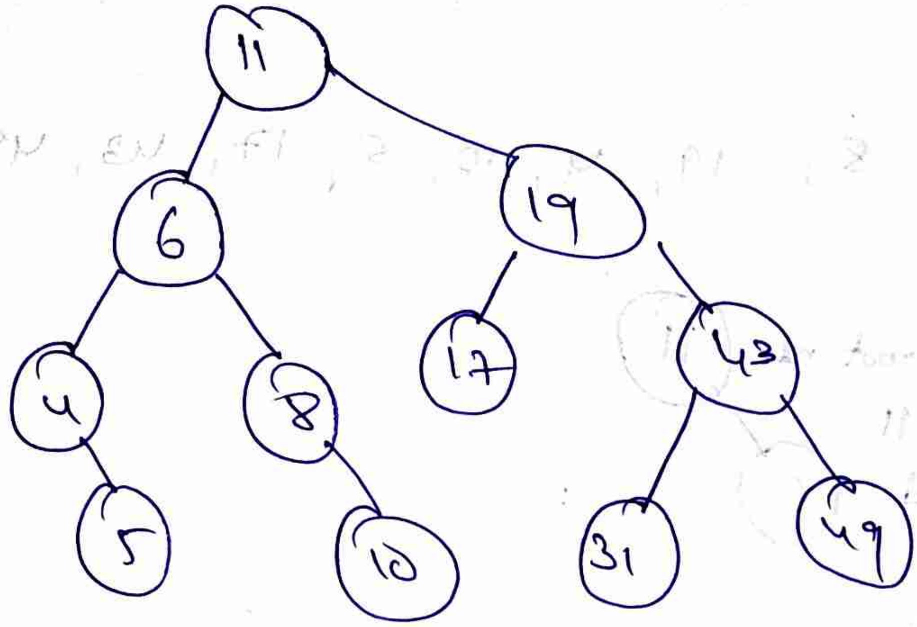
Step 4:- Now next element is 19 compare with ~~root~~ node 19 is greater then of '11' then right of '11' Now.



Step 5:- Now next element is '4' compare with root node 4 is less then left said again it should compare with '6'. ~~less~~ 4 is less then of '6' then left said of 6.



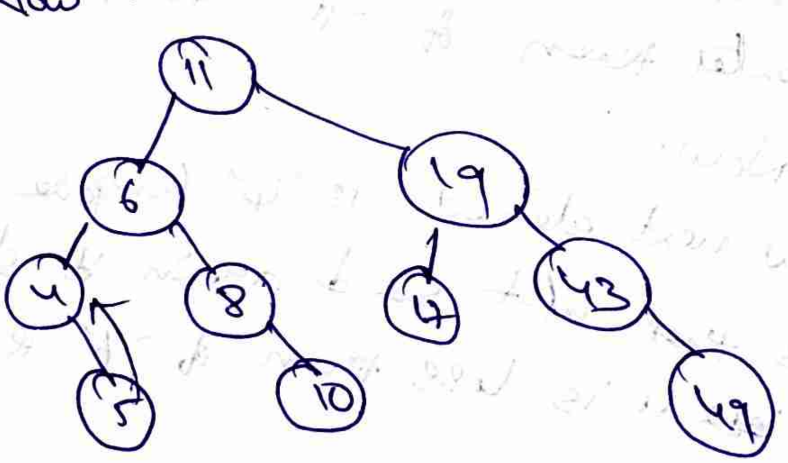
Step 6:- Contious the all steps up to complete ~~of~~ tree.



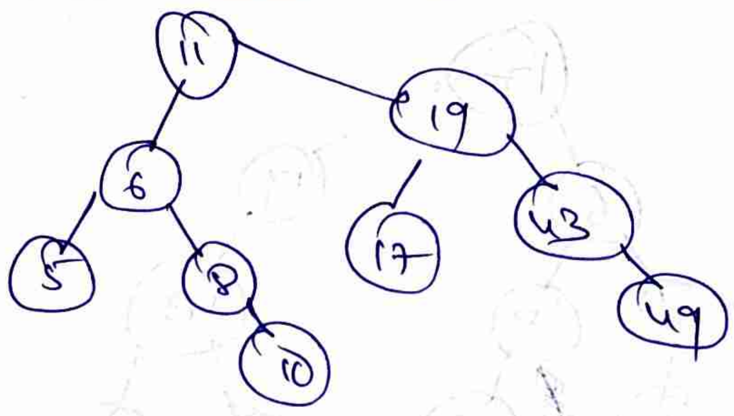
Deletion !- when we are deleting from binary search tree then there could be three cases:

* 'O' zero child
* one child
* two children.

Case 1:- 'O' zero child we are going to delete from tree. '31' is No child then from tree 31 can delete easily Now tree becomes.



Case 2:- one child we are going to delete from tree from exam '4' is having only one child '5' then 4 is deleted and 5 is replaced with '4' Now tree becomes

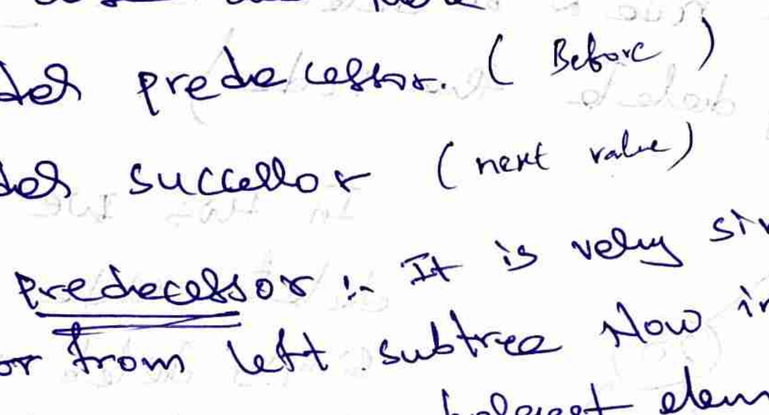


Case-3:- when deleteing 'two' children we are going to delete '11'. '11' is having left sub tree and right sub tree. NOW '11' is replaced by which number for this two case are there.

→* Inorder predecessor. (Before)

→* inorder succesor (next value)

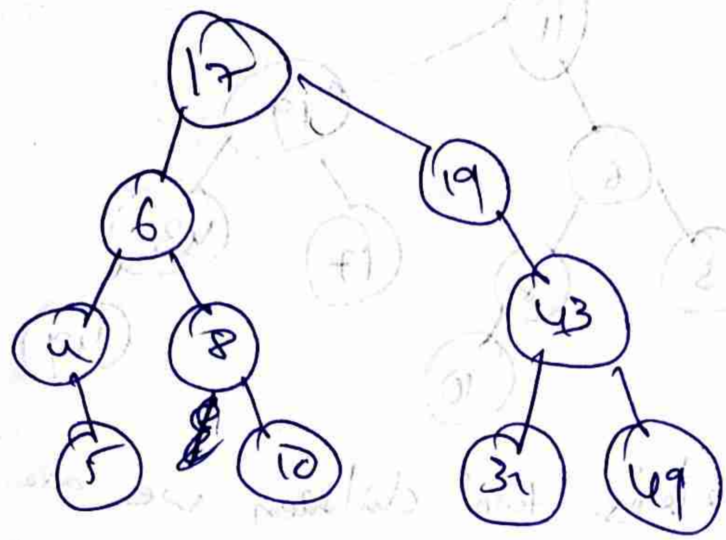
Inorder predecessor :- It is very simple, the largest number ~~for~~ from left subtree Now in fig 6, 5, 8, 10 from that '10' is the largest element then we can replace 11 with '10' after replacing '10' the tree becomes.



inorder predessor of 11 is → 10(left)

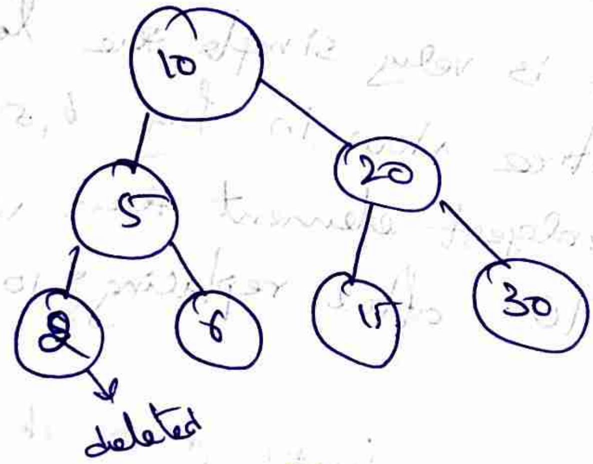
inorder sucessor of 11 is → 17(right)

Inorder succesor :- It is also very simple ~~fast~~ to find.. Smallest number from right sub tree. Now in fig 19, 17, 43, 49. from that '17' is the Smallest number (element) then we can replace 11 with (or 10 with) 17 after replacing 17 the tree becomes.

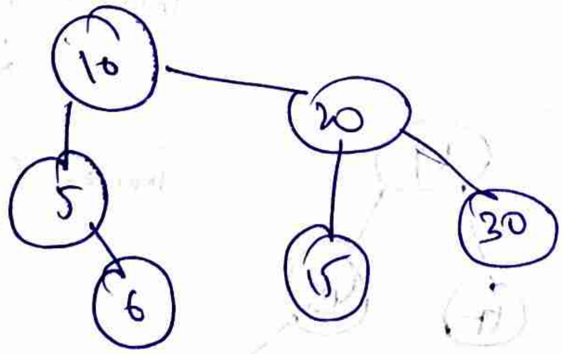


Example 2:- Deletion in Binary search Tree (B s T)

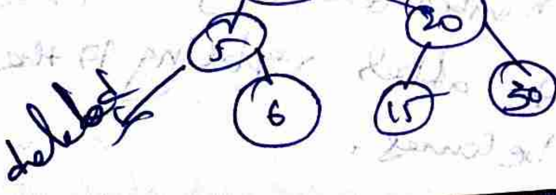
Case 1:- If the node to be delete is a leaf node
↓
No child
(simply delete the node)



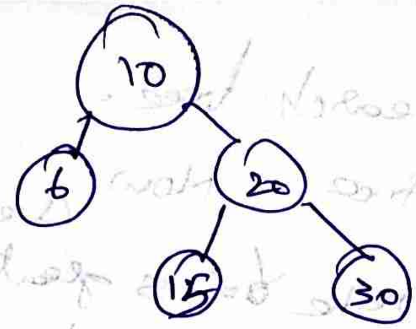
In this we want to delete Node Number '2' simply delete the node Now tree becomes



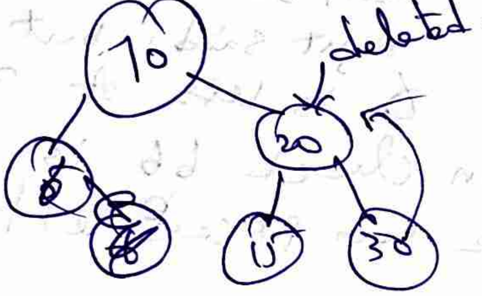
Case 2:- If the node to be deleted has only one child (copy child to node we delete child)



In this we want to delete node number '5' for '5' one child is here so copy child to node we delete child. now tree becomes



Case 3:- If the node to be deleted has two children nodes. find Inorder successor of node then Copy content of inorder successor of node ie delet inorder successor.

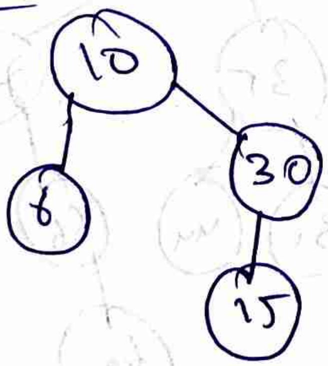


In this we want to delet Node Number '20' for '20' '2' childrens are there

Inorder :- Left root Right 6. 10 15, 20 30

Inorder successor is for '20' '30' is the inorder successor.

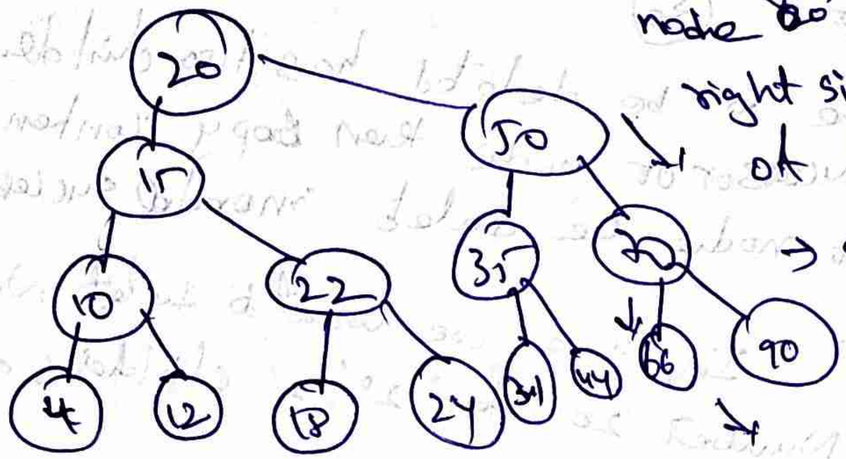
Now:-



Example:

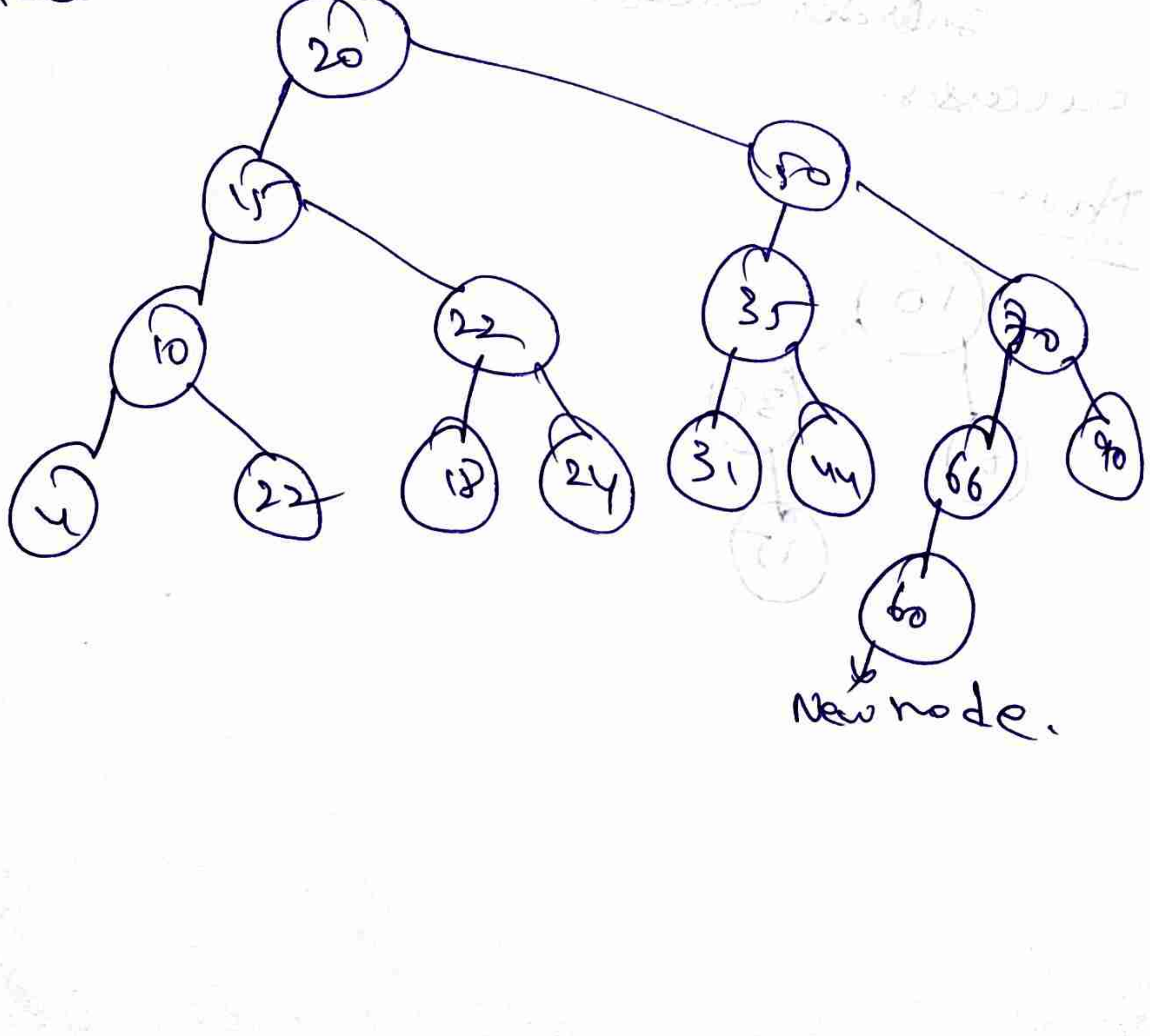
Insertion in Binary search tree.

Insert 60 in the tree Now check with root node '60' is greater then 20 right side. Now check right of tree. Now



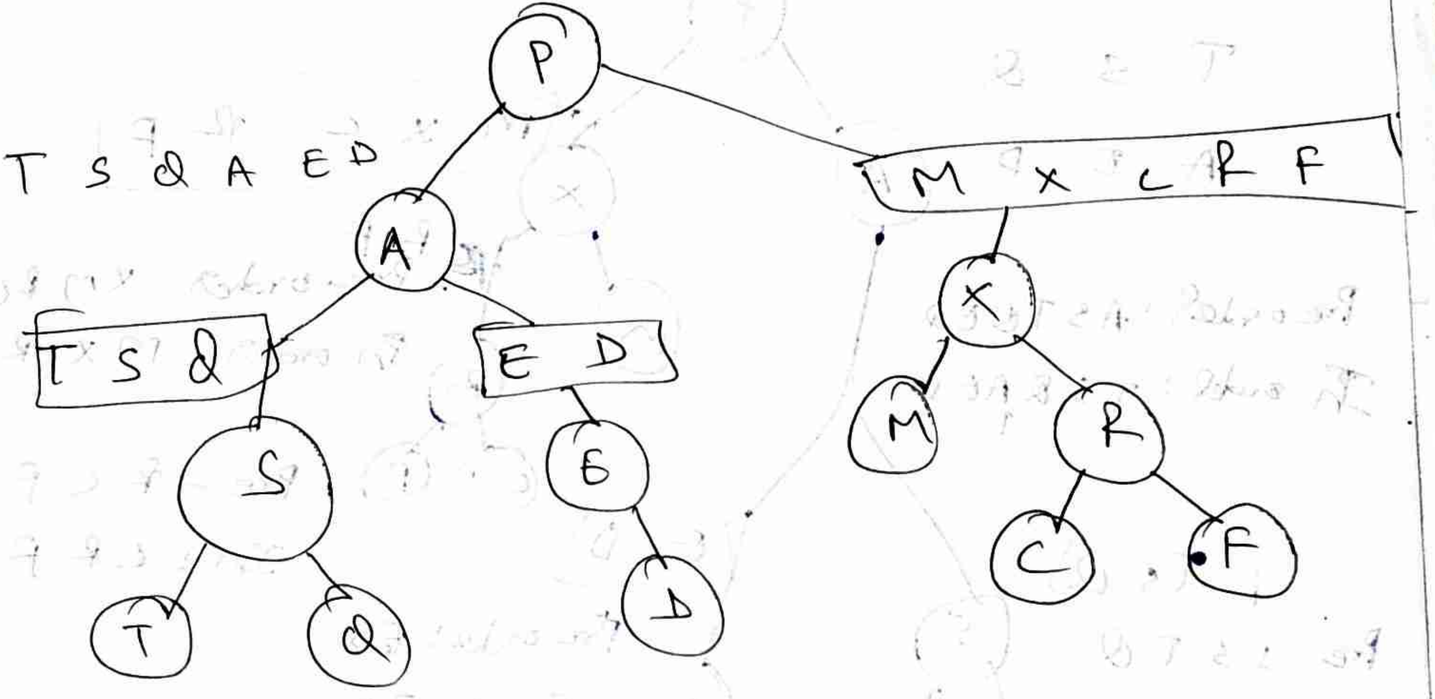
→ again 60 is greater then 50 Now again right said; but 60 is less of 70 then left said of '70' again check 66 is greater then of '60' Now we can placed left said of '66'.

Now tree is



Post order: T Q S D E A M C F R X $\boxed{P}$

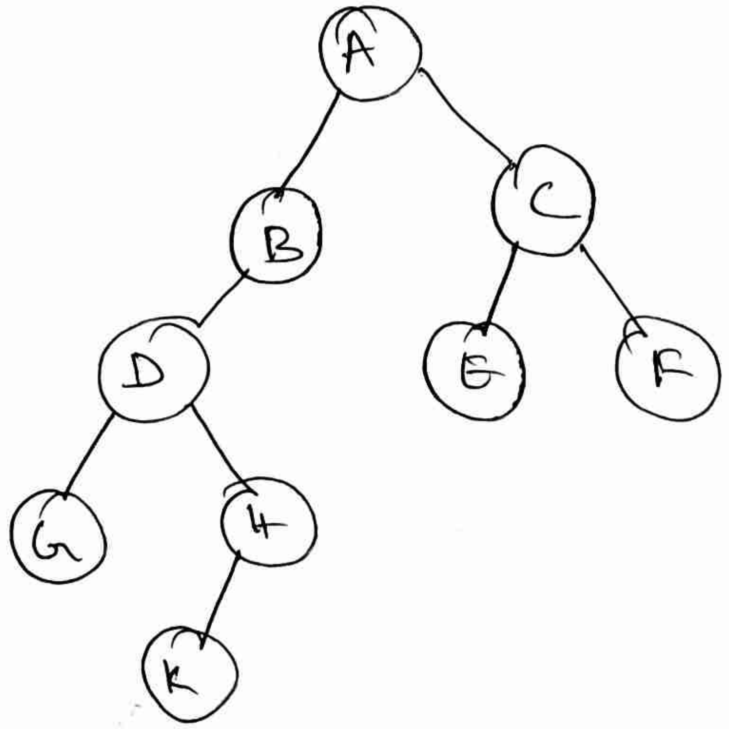
Inorder: T S Q $\boxed{A}$ E D $\boxed{P}$ M X C R F



Preorder = A B D G H K C E F

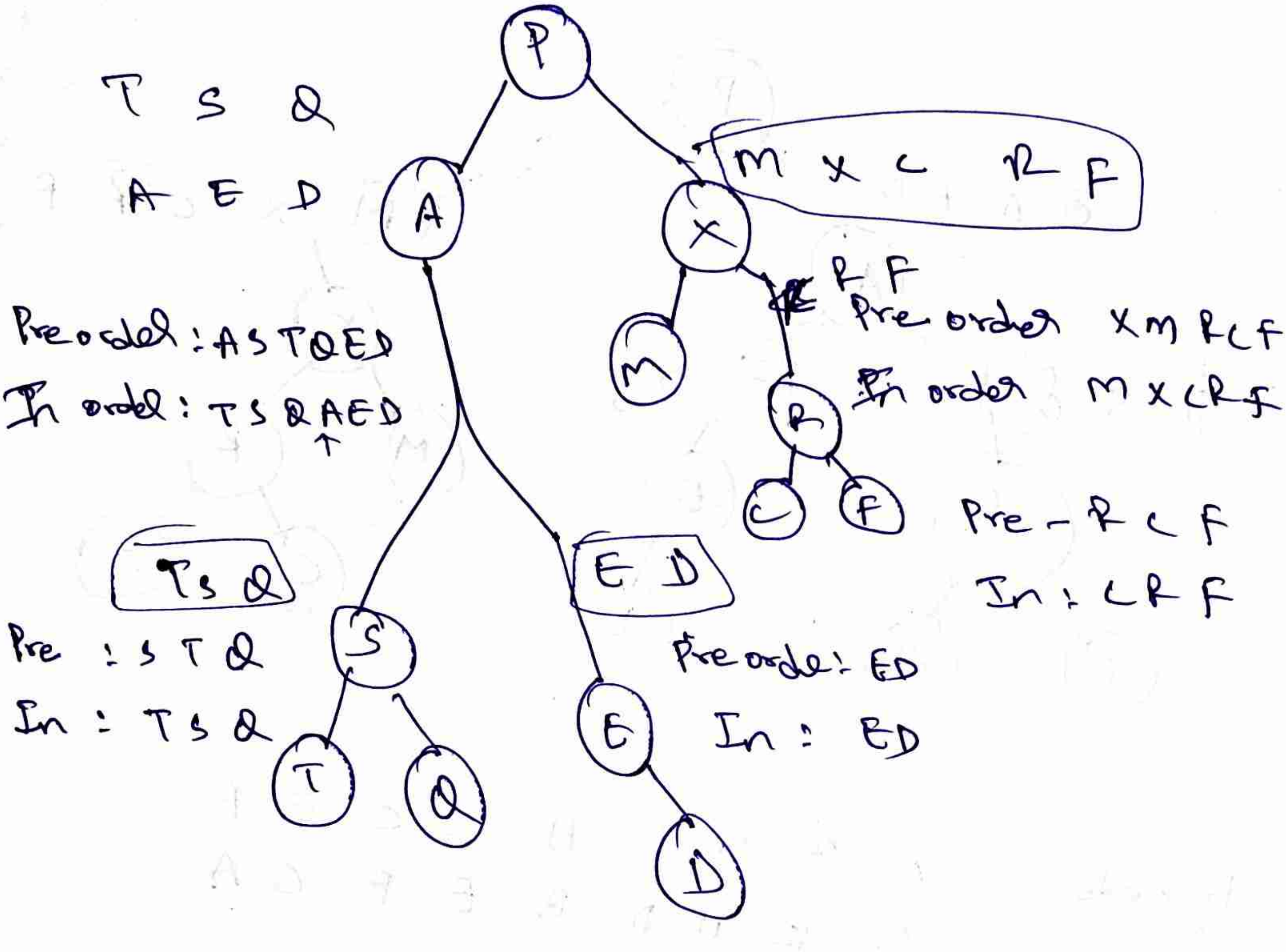
Post order = G K H D B E F C A

(i)



Preorder: P A S T Q E D X M R C F

Inorder: T S Q A E D P M X C R F



T S Q

A E D

M X C R F

~~R~~ R F

Preorder XM RCF

Inorder MXCRF

Preorder: ASTQED

Inorder: TSQAED

Pre - RCF

In: CRF

TSQ

ED

Pre : STQ

In : TSQ

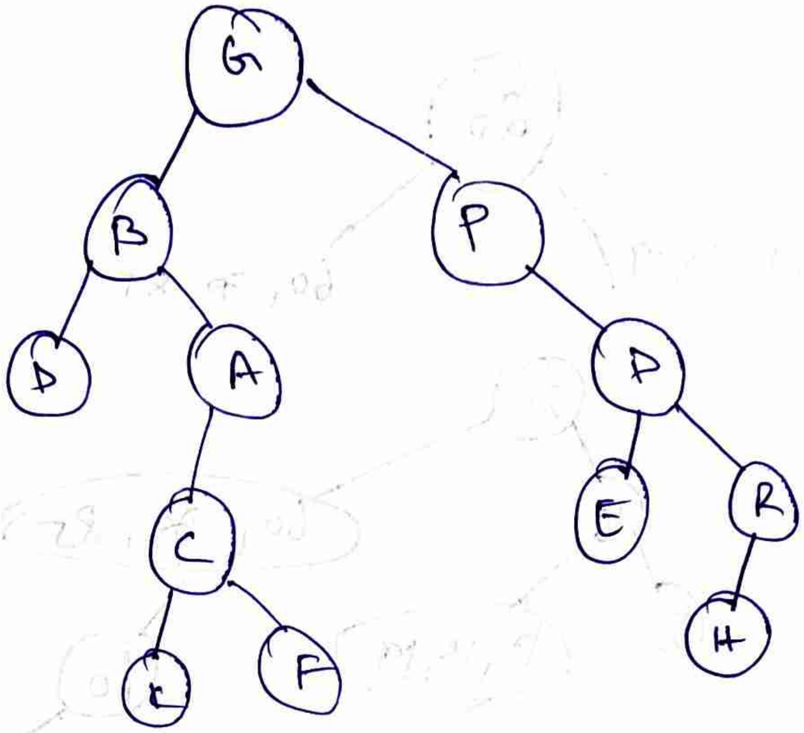
Preorder: ED

In : ED

Example1

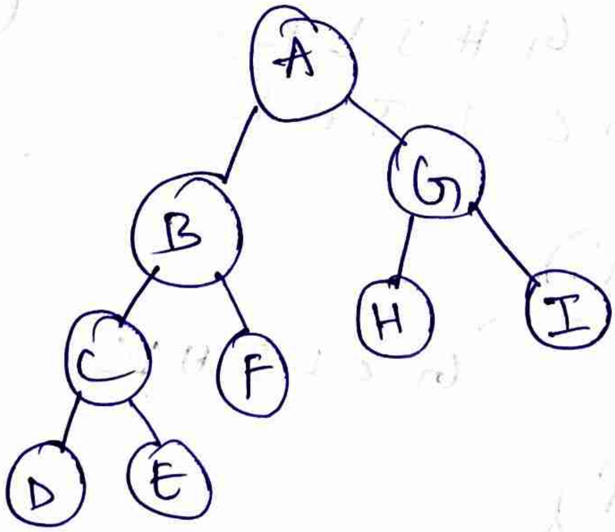
Pre order : G B Q A C K F P D E R H

In orde : Q B K C F A G P E D H R



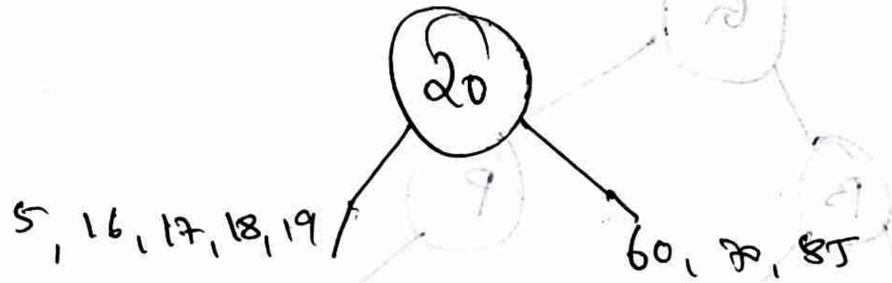
Post : D E C F B H I G A

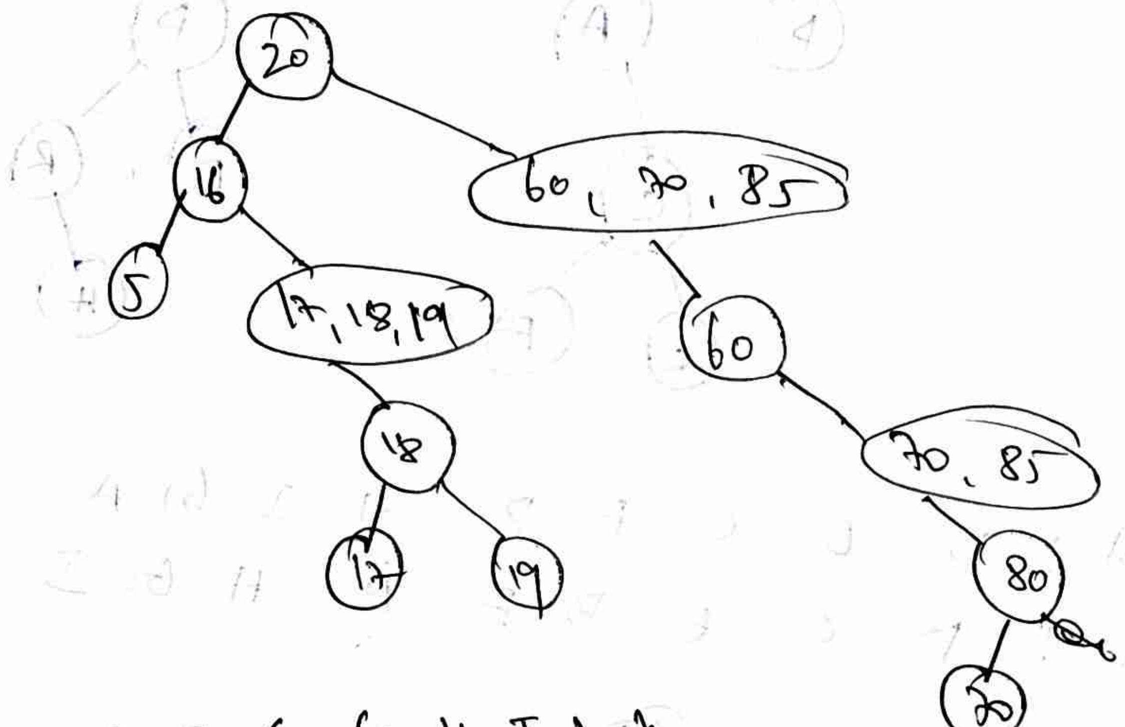
In : D C E B F A H G I



Pre : 20, 16, 5, 18, 17, 19, 60, 85, 70

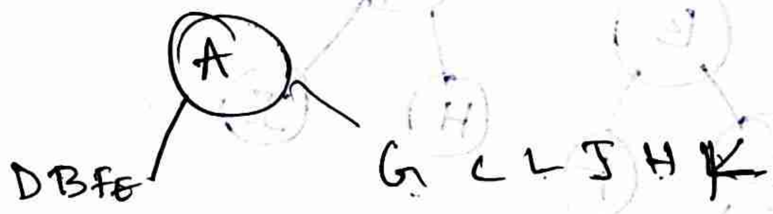
In : 5, 16, 17, 18, 19, 20, 60, 70, 85

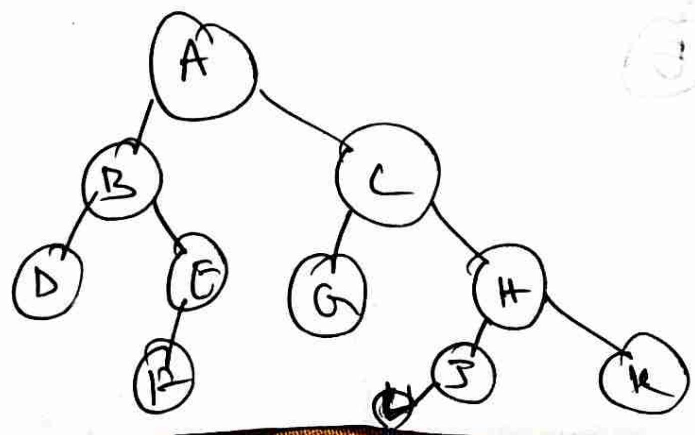




Pre : A B D E F C G H J L K

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





AVL Tree :- AVL Tree were introduced by Adelson-Velskii and Landis (hence the acronym AVL). An AVL tree is a binary tree that is balanced in accordance to the height of the subtree. In the worst case, the height of an AVL tree is O(logn), where 'n' is the no. of nodes in a tree.

In a non-empty binary tree denoted by 's' contains two subtree. i.e., left subtree (S_L) and right subtree (S_R) then s is said to be an AVL tree if it satisfies the following properties.

(i) The left and right subtree are AVL trees

(ii) The diff b/w the ht of the left & right subtree is less than or equal to 1 i.e.,

$$|H_l - H_r| \leq 1$$

where H_l is ht of the left subtree S_l &

H_r " " " " right " S_r.

AVL tree properties :- The properties of AVL tree are the follows

(1) An AVL tree consisting of 'n' elements or nodes which is of ht O(logn)

(2) An AVL tree can be constructed for each value of n, where n≥0.

(3) The search complexity of an AVL tree of n-elements is O(logn).

(4) The insertion of an element in an n-element AVL search tree results in an n+1 element AVL tree and time complexity for such an insertion is O(logn).

(5) The deletion of an elements in an n-element AVL search tree results in an n-1 element AVL tree & time complexity for such a deletion is O(logn)

List out various **rotations of AVL tree** :-

The different types of rotations that can be performed after inserting or deleting an element are.

(1) LL (Left - Left) rotation

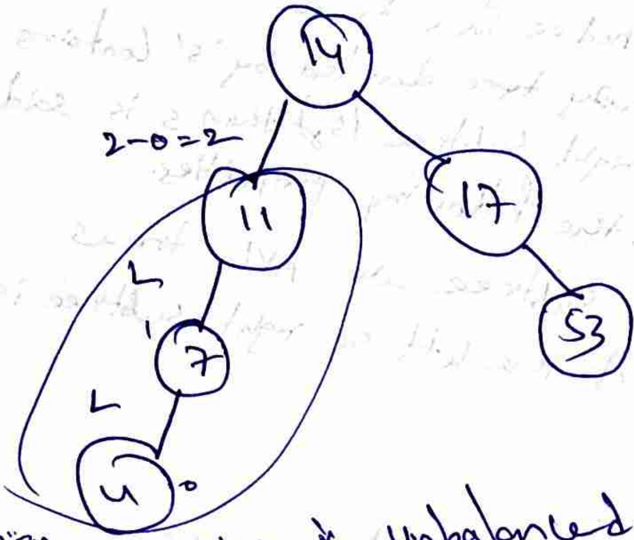
(2) RR (Right - Right) "

(3) LR (Left - Right) rotation

(4) RL (Right - Left) rotation

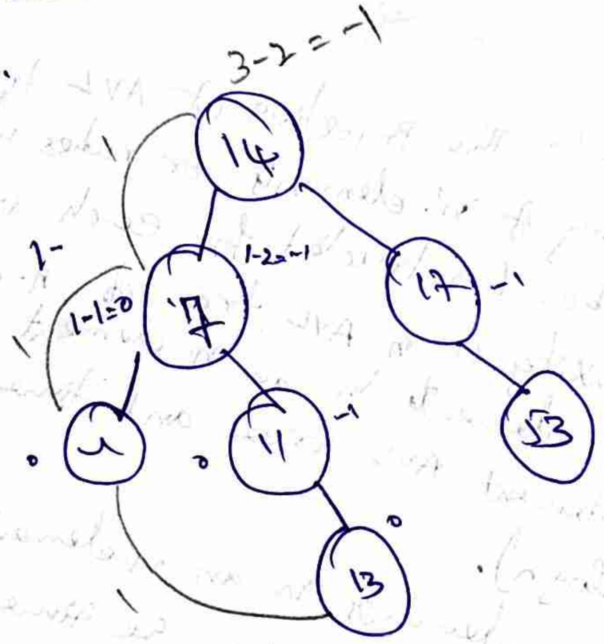
Construct AVL tree by inserting the folloing data:

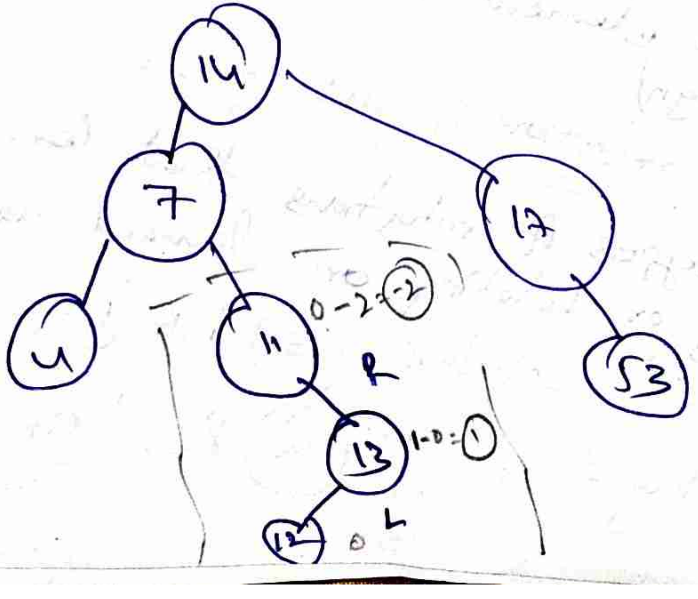
14, 17, 11, 7, 53, 4, 13, 12, 8, 60, 19, 16, 20.



LLRotation Now is unbalanced. we have to make balanced tree.

Now tree.

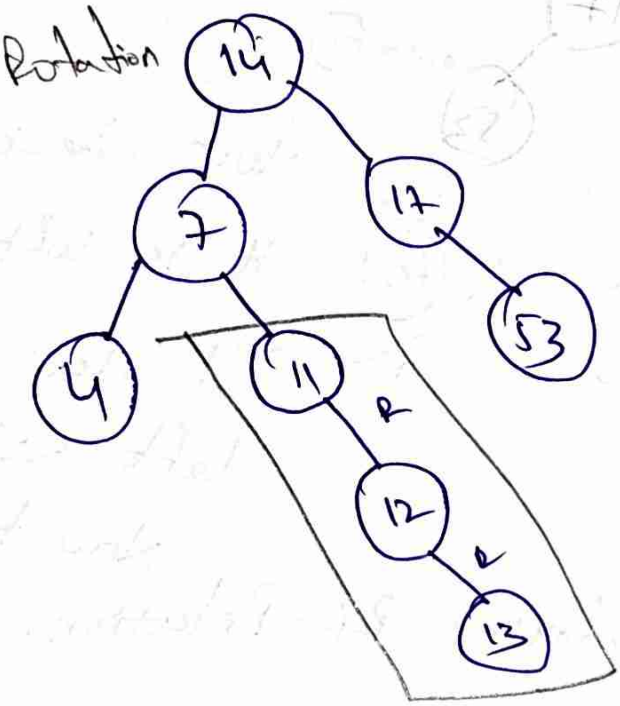




Here Right left rotation in AVL balanced ~~tree~~ should be there.

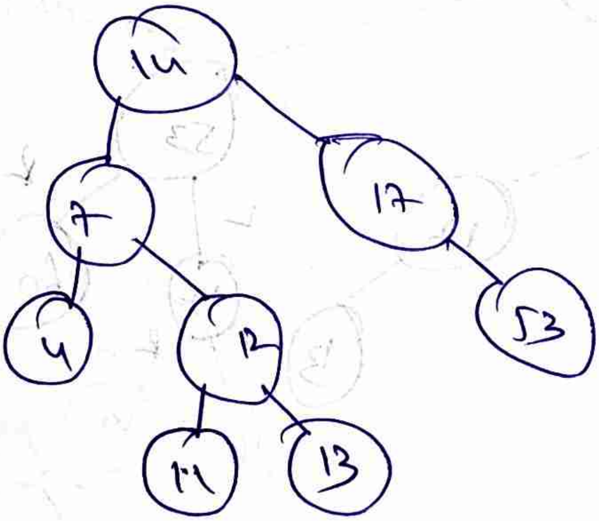
2- Rotation will be there 1st Right rotation next left rotation.

Right Rotation

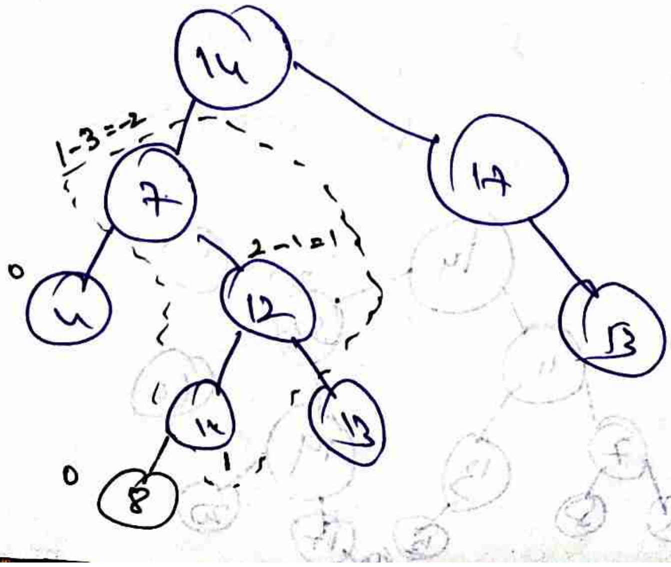


RR - Rotation still unbalanced.

Left Rotation

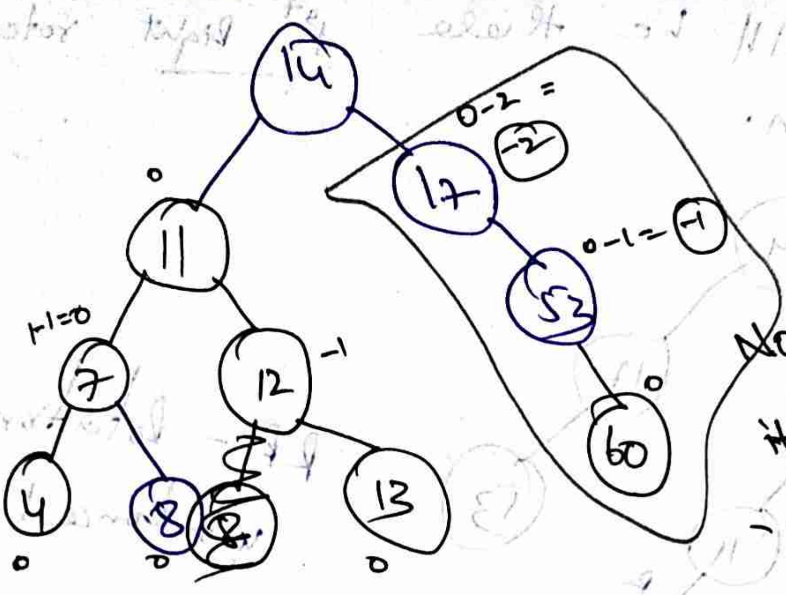


Insert '8' in AVL tree.



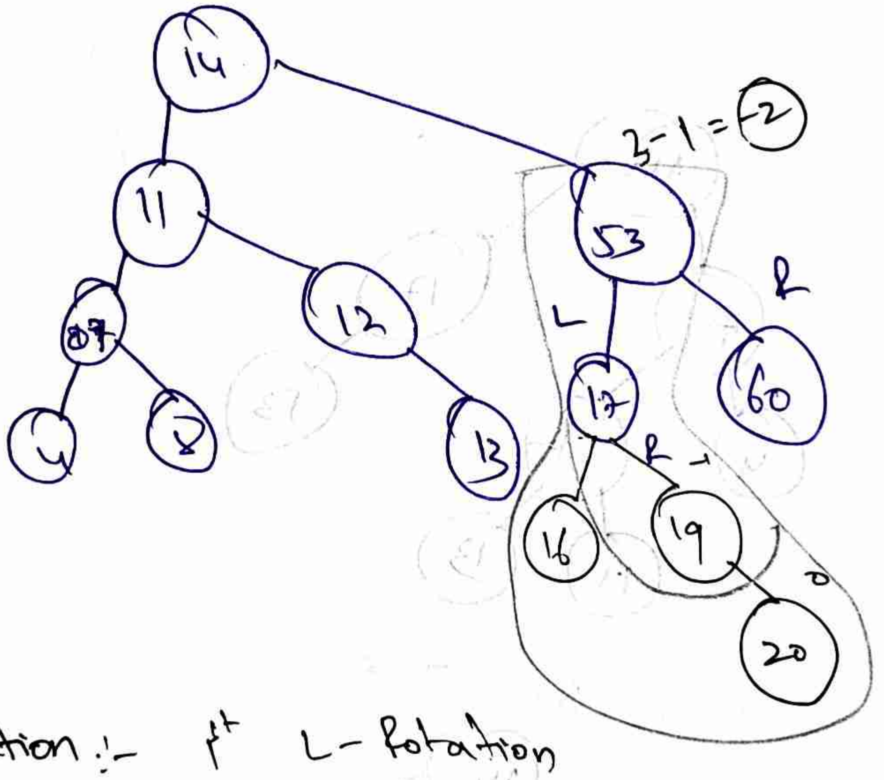
7, $\boxed{11,}$ 12

11- should Root.



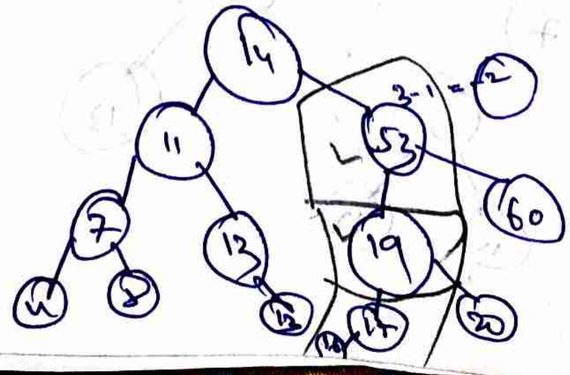
Now check '8' it is left of '11' Now check left said of '11' Now tree

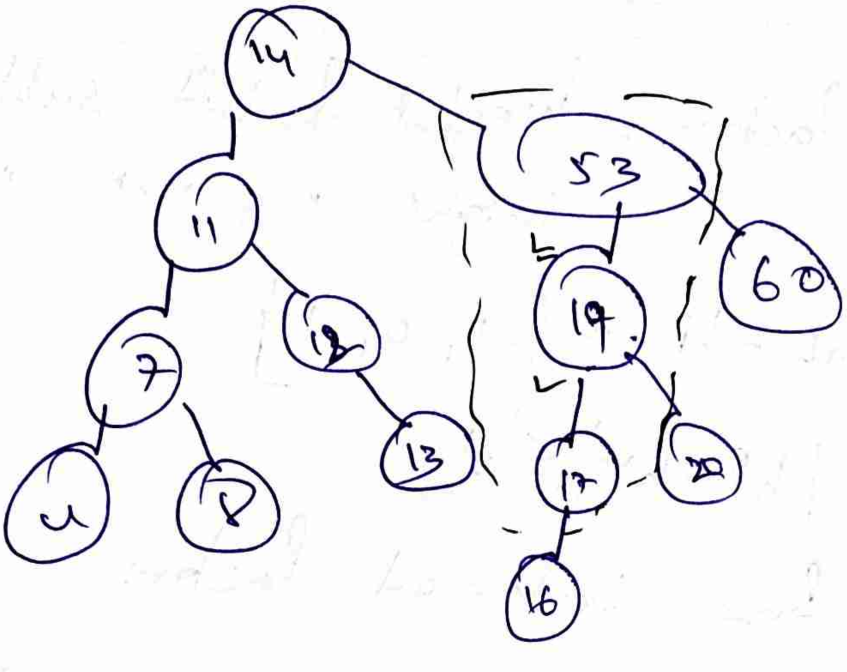
Now 17 is unbalaced RR-Rotation.



LR-Rotation :- 1st L-Rotation
2nd R-Rotation.

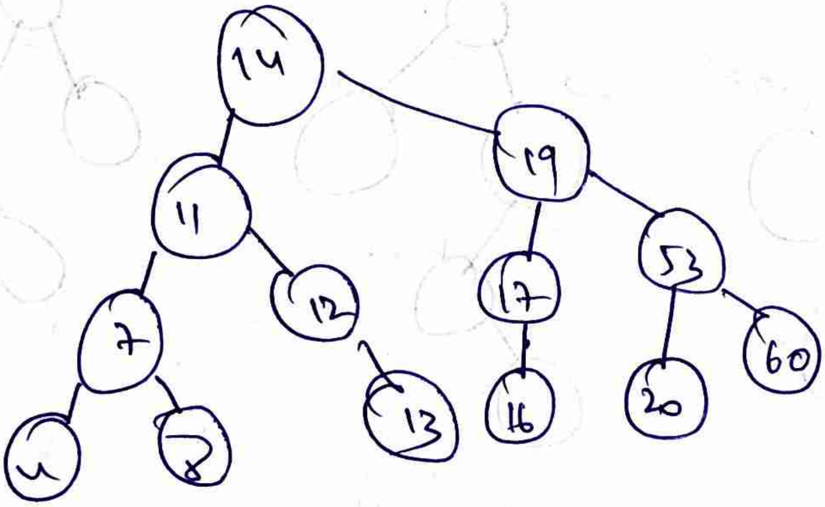
Now:-





Now LL
Rotation

Now

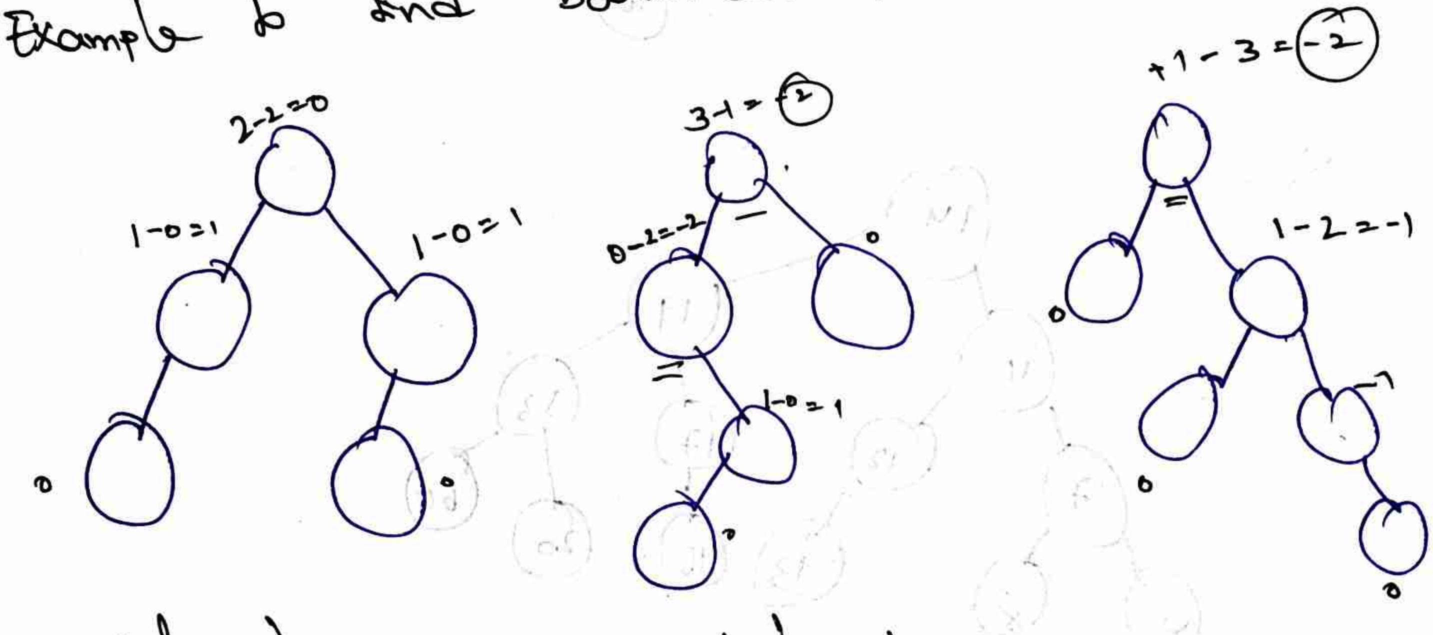


Balance factor = hieght ofe left subtree -
height " right "

$bf = h_L - h_R = \{-1, 0, 1\}$

$bf = |h_l - h_r| \leq 1$

Example to find balanced factor!

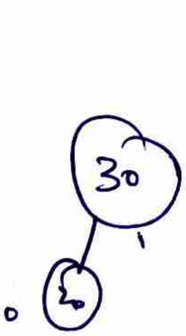


balanced.

Inbalanced

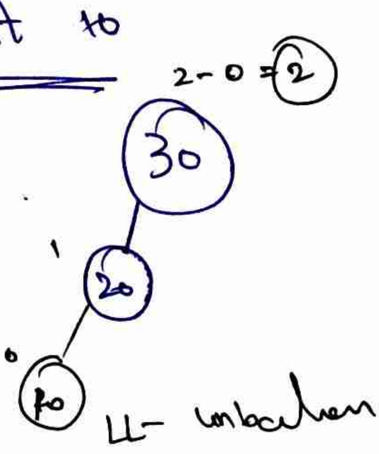
Example:-

Initially



balanced

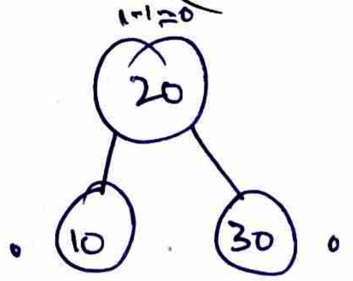
Insert 10



LL- unbalanced

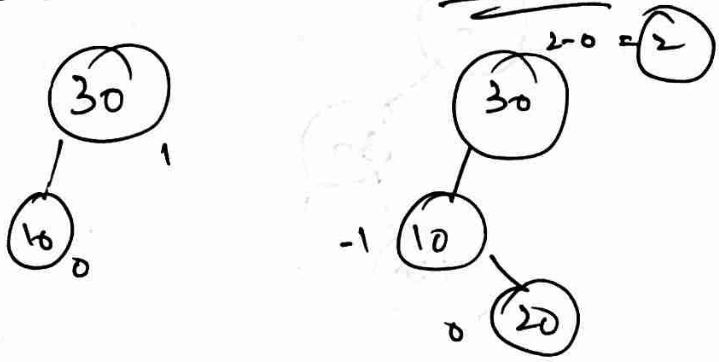
Now Rotation we have to LL- rotation

After Rotation :- (LL- Rotation).

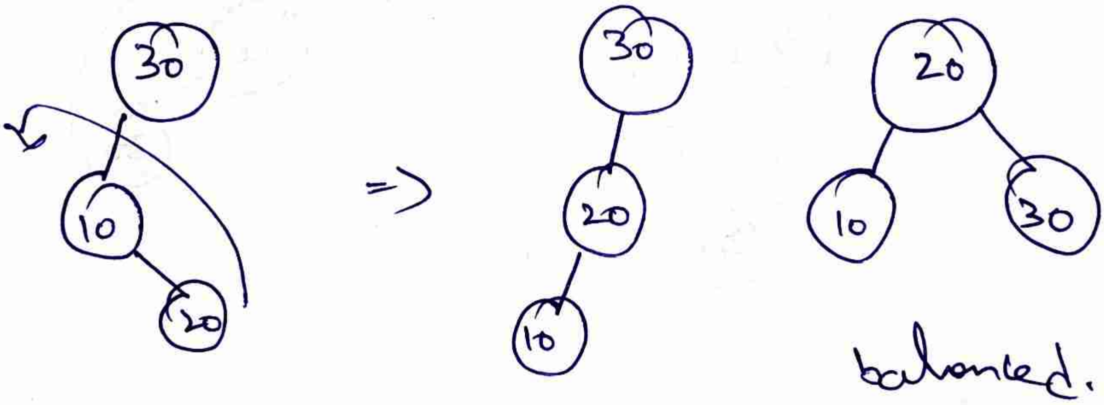


Initially

Insert 20



LR- inbalance.

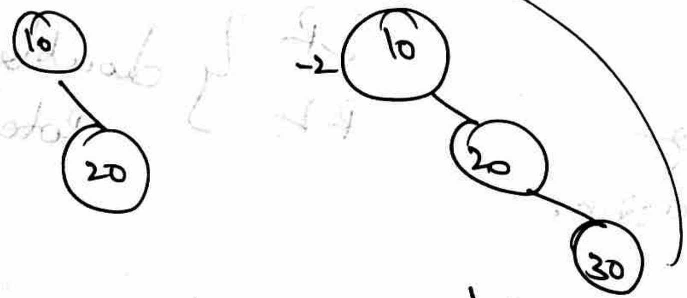


balanced.

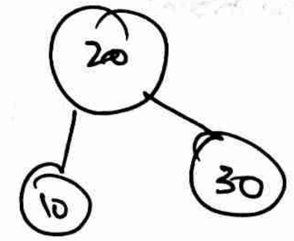
double rotation. LR rotation.

Initially

Insert 30

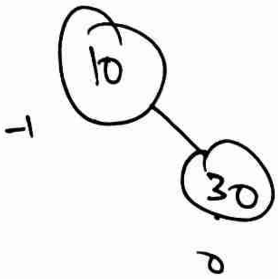


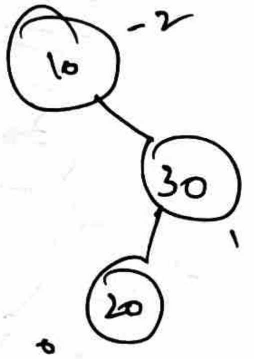
RR-rotation.



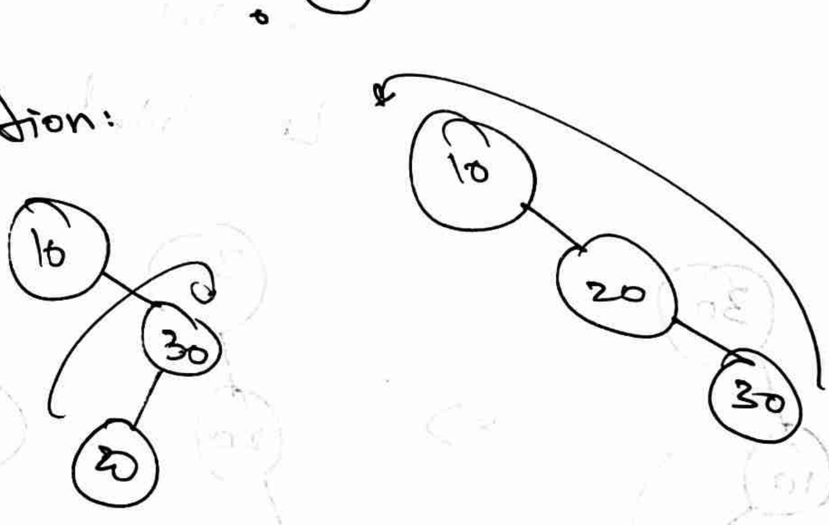
Initially

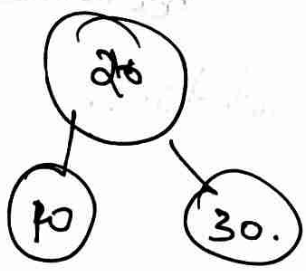
Insert 20





RL- Rotation:

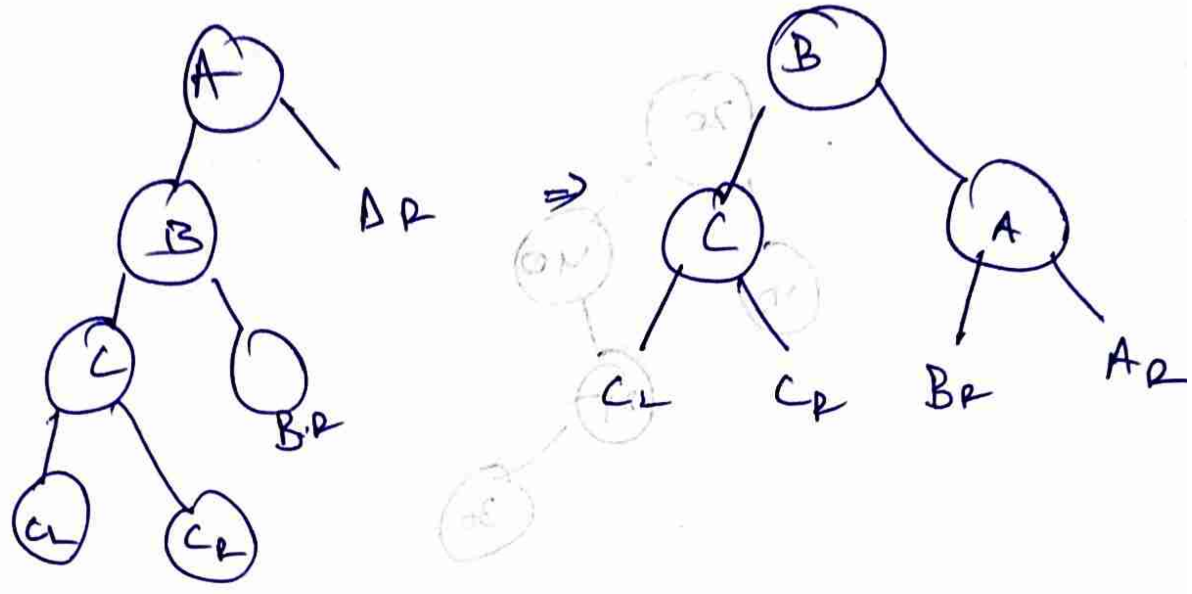




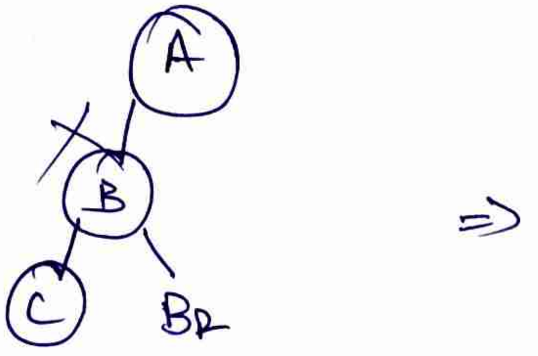
LL } Single Rotation.
RR

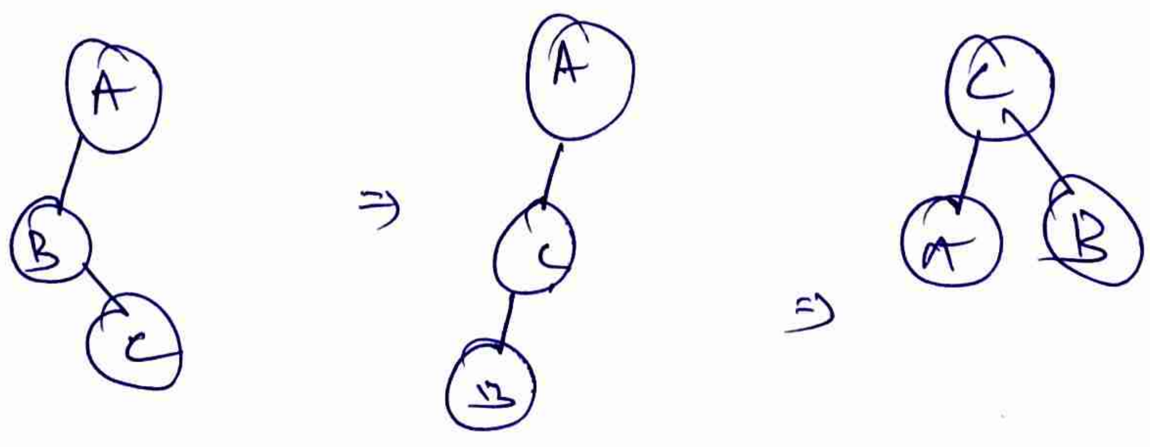
LR } double Rotation
RL

LL- Rotation:

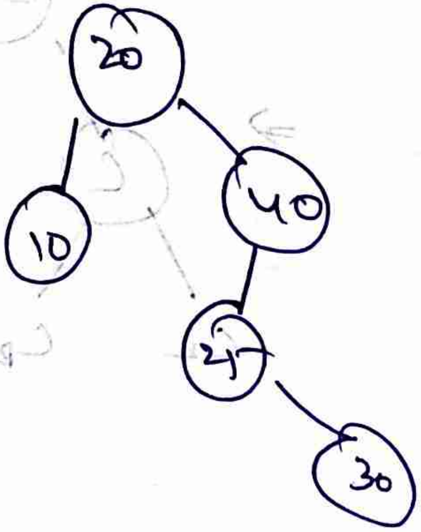


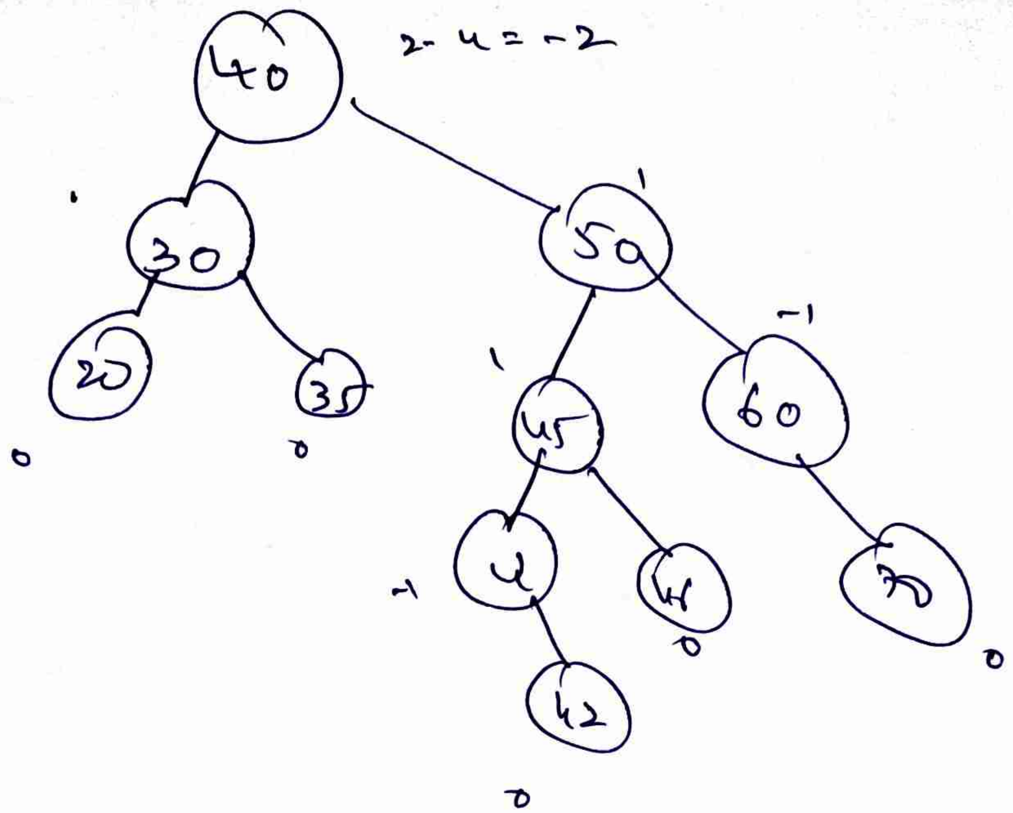
LR Rotation





Key : 40, 20, 10, 25, 30, 22, 50

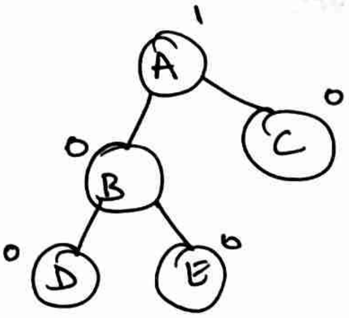




R LLR

O(logn)

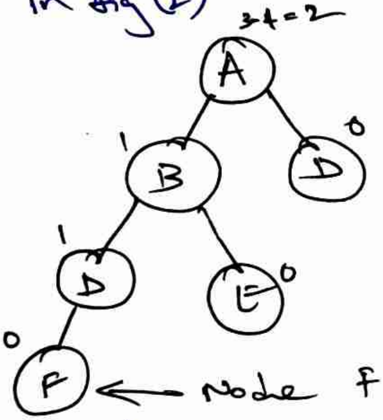
LL Rotation:- LL Rotation in a single rotation that can be applied when a node is inselted in the left subtree of the left child of a node. In this, rotation is pelformed in a clockwise direction. Consider the following AVL tree.



fig(1)

Balanced tree befor insertion

In fig(1), a node 'f' is inselted in the left subtree of left child of node A. This shown in fig(2)



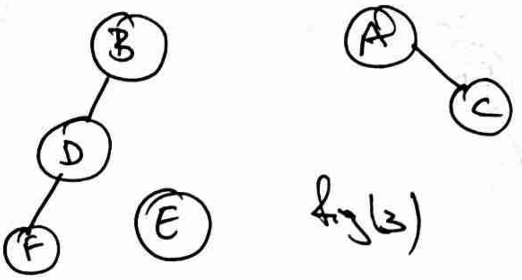
fig(2)

← Node f is inselted

Imbalanced Tree After inseltion

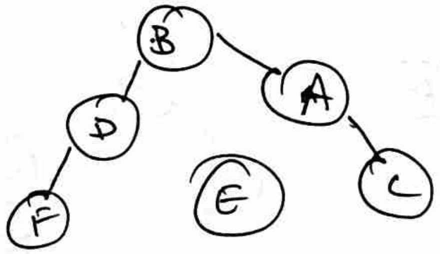
After the inseltion, the tree belomes imbalanced bez node A has a balance factor 2. Thus to rebalance the tree in accordance to the balance factors {-1, 0, +1}, the folloing opelation must to pelformed

(i) The root of the subtree in which the node F is inselted i.e node B in made as the new root node. This is shown in fig(3).



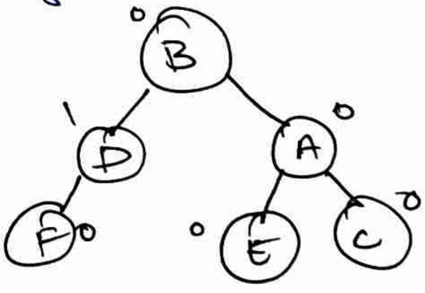
fig(3)

(ii) The original root node i.e A is made as the right subchild of the new root node B. This is shown in fig(4)

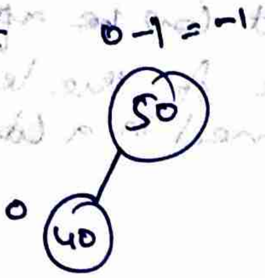


fig(4)

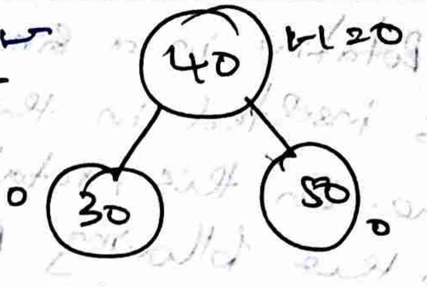
(iii) The right child of node B i.e, E is made as the left subchild of A, wheeas the right child of A i.e, C remains unchanged. This is shown in fig(5)



step1:-

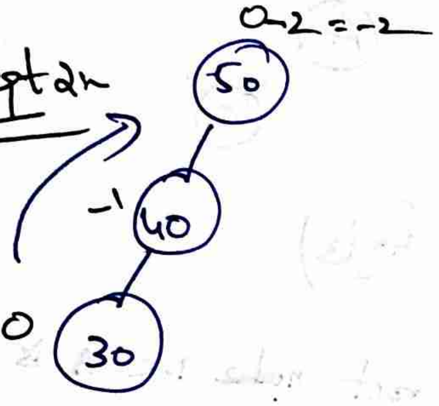


~~step~~ step 3:-

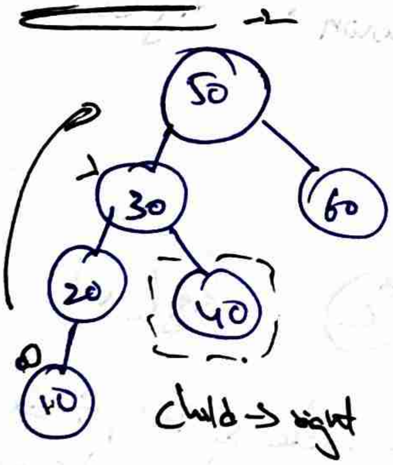


balanced AVL tree

step 2:-

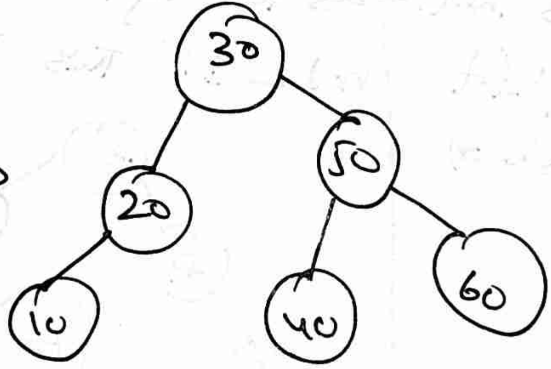


Examples:-



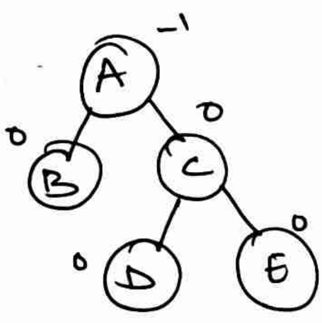
child -> right

=>



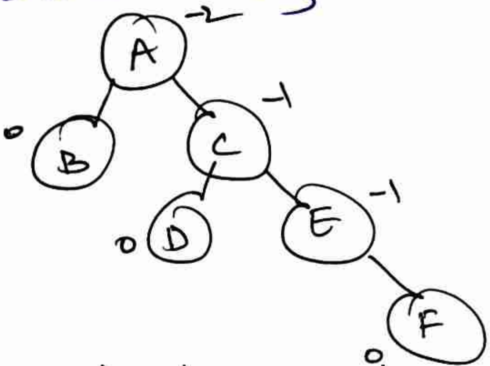
Parent -> left

RR- Rotation :- RR- Rotation is also a single rotation that can be performed when a node is inserted in the right subtree of the right child of a node. In this, the rotation is performed in an anti-clockwise direction. Consider the following AVL tree.



Balanced tree before insertion

In the fig, a node F is inserted in the right subtree of the right child of a node A. This is shown in fig

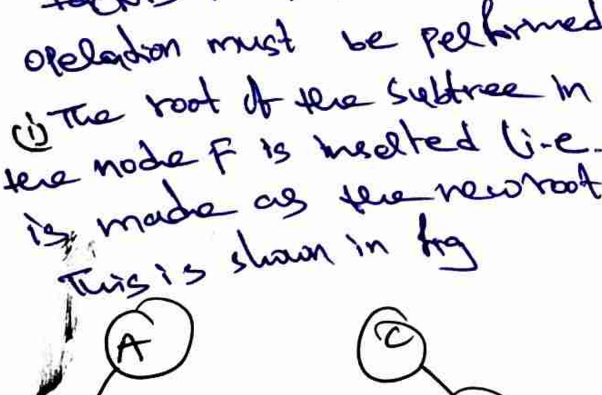


Node F is inserted

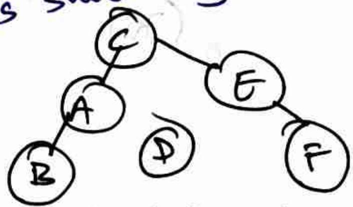
Imbalanced Tree after inserted.

After insertion the tree becomes imbalanced bcz node A has a balanced factor of -2. Thus to rebalance the tree in accordance to the balance factors. -1, 0, +1 the following operation must be performed.

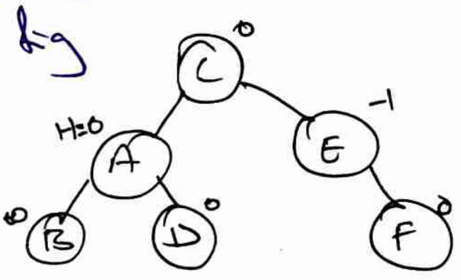
(i) The root of the subtree in which the node F is inserted (i.e.., C) is made as the new root node. This is shown in fig



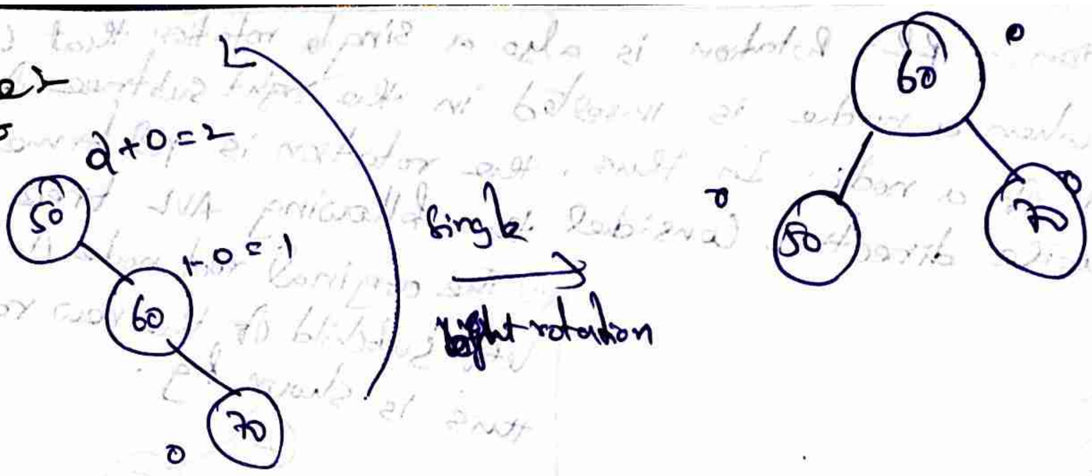
(ii) The original root node A, is made left subchild of the new root node C. This is shown fig.



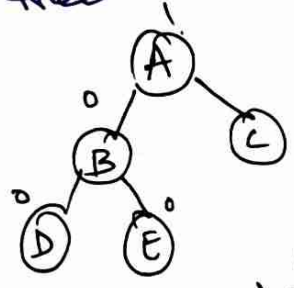
(iii) The left child of node C (i.e D) is made as the right subchild of A, whereas the left child of A (i.e., B) remains unchanged. This is shown in fig



Example:

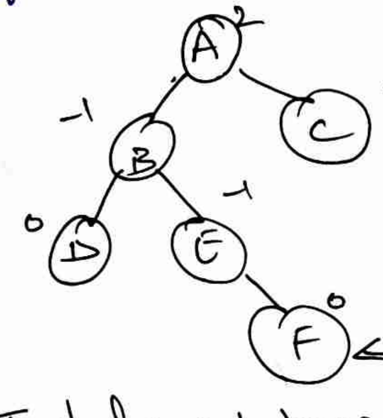


LR Rotation:- LR rotation is a double rotation that can be Pelformed when a node is inserted in the right subtree of the left child of a node. In this type of rotation, RR rotation followed by LL rotation are pelformed. Considers the following AVL tree.



Balanced tree before Insertion

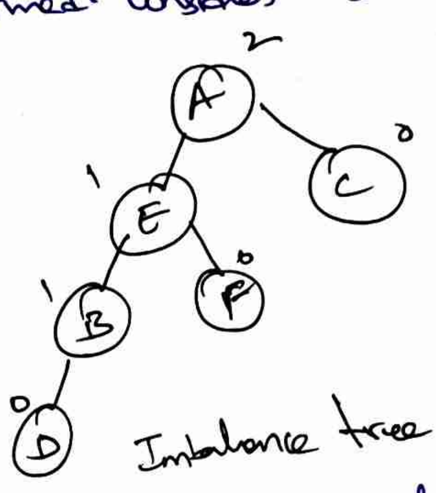
In the fig above a node F is inserted in the right subtree of left child of the node A. This is shown in fig



Imbalanced tree after insertion.

After insertion, the tree becomes imbalanced because node A has a balance factor 2. Thus to rebalance the tree in accordance to the balance factors -1, 0, +1, the following operation must be formed.

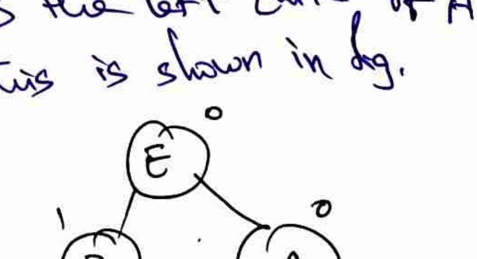
(i) Initially, RR rotation must be Pelformed by rotating, the parent of the inserted node (i.e.. E) in an anti-clockwise direction i.e making it as the root of the subtree. Thus B, D becomes the left child i.e F becomes the right child of node E. This is shown in fig.



Imbalance tree after RR Rotation

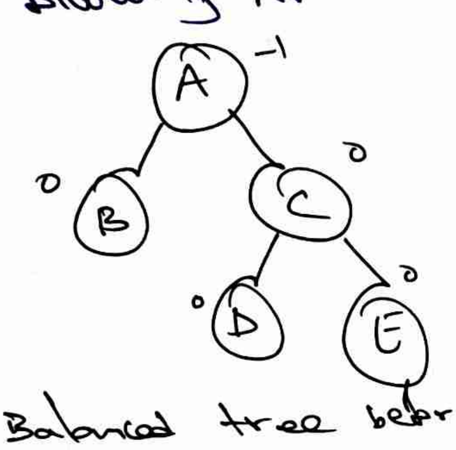
(ii) The tree shown in fig is also an imbalanced tree bez of the balance factor 2 at node A. Now, LL rotation is to be performed in a clockwise direction so as to make the tree balanced.

(iii) In LL rotation, the parent of the inserted node (i.e E)'s made as the ~~new~~ root node of the tree. The original root A is made as the right child of E i.e node F is made as the left child of A. This is shown in fig.



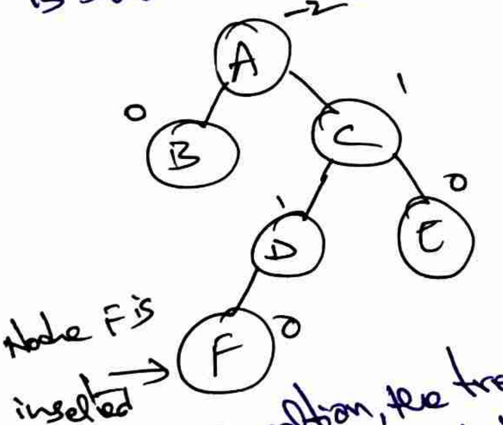
Balanced Tree after LL-Rotation.

RL Rotation :- RL rotation is also a double rotation that can be pelformed when a node is inselted in the left subtree of the right child of a node. In this type of rotation, LL rotation followed by the RR-rotation are pelformed. Consider the following AVL tree.



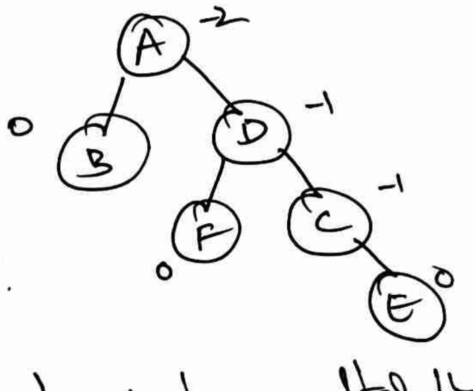
Balanced tree befor insertion.

In the above fig, a node F is inserted into the left subtree of the right child of node A. This is shown in fig.



After the insertion, the tree becomes imbalanced bez node A has a balance factor -2. Thus to rebalanced the tree in accordance to the balance factors -1, 0, +1 the following operation must be pelformed

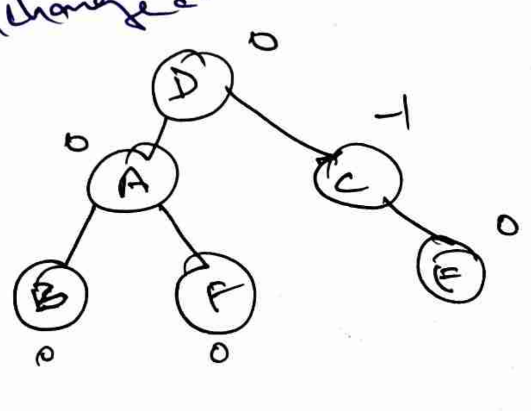
(i) Initially LL rotation must be pelformed by rotating the parent of the inselted node (i.e D) in a clockwise direction ie making it as the root of the subtree. Thus C we E bez the right child and F becomes the left child of node D. This is shown in fig.



Imbalanced tree after LL Rotation

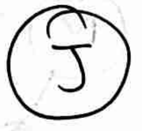
(i) The tree shown fig is an imbalanced tree bez of the balance factor 2 at node A. Now, RR rotation is to be pelformed in an anti-clockwise direction so as to make the tree balanced.

(iii) In RR rotation, the parent of the inselted node (i.e D) is made the new root node of the tree. The original root A is made as the left child of D and the node F is made as the right child of A. However, the node B remains unchanged. This shown in fig.



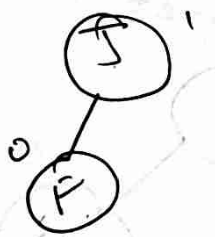
Construct AVL tree for the list {J, F, M, A, N, K, L, A, S, O, P, D}

Step1:- Initrally, the AVL tree is empty, construct an AVL tree by inseeting J into to the empty tree.



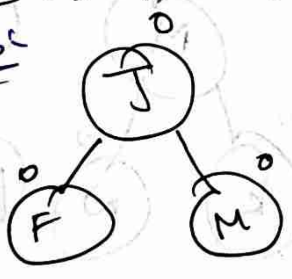
Insert (J)

step2:- Then, a new element (i.e, F) is inserted into an AVL tree. This item is inserted into the left subtree of node J, causing the left subtree J to glow in height. After insertion, the balance factor of node J is 1.



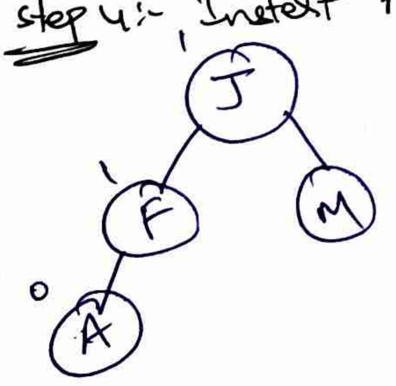
The subsequent insertion are as follows,

step 3:

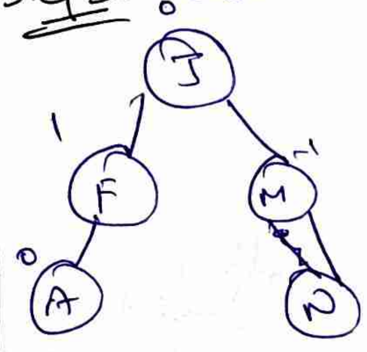


(Insert M)

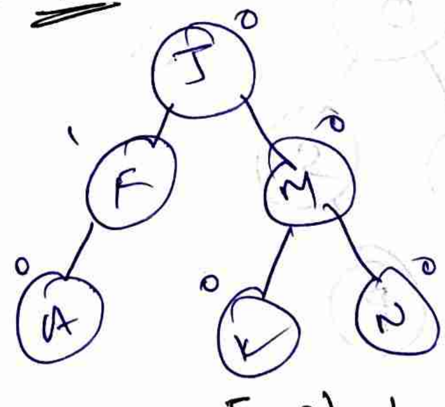
step 4:- Insert A



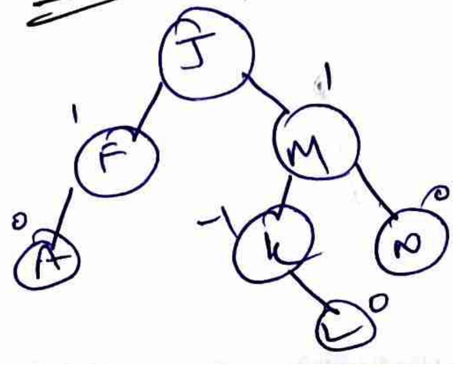
Step 5:- Insert N



step 6:- Insert K

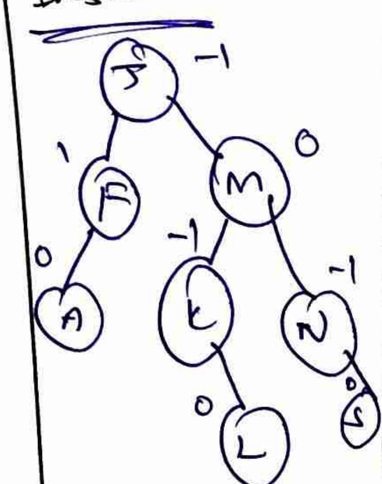


step 7:- Insert L

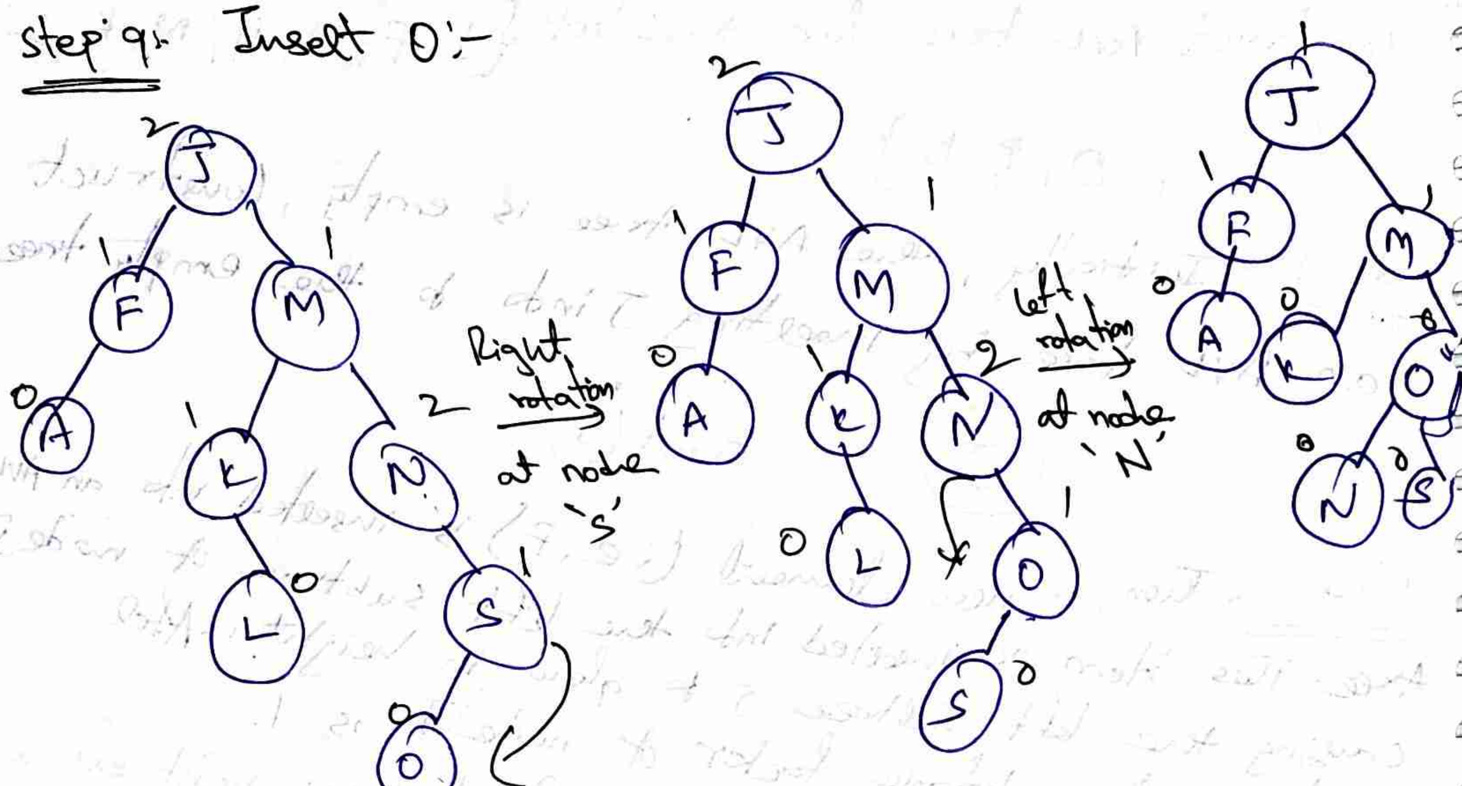


step 8:- Next element to insert is A, since this element already exits in the AVL tree it is rejected ~~step~~ next 'S', is inserted into the AVL tree

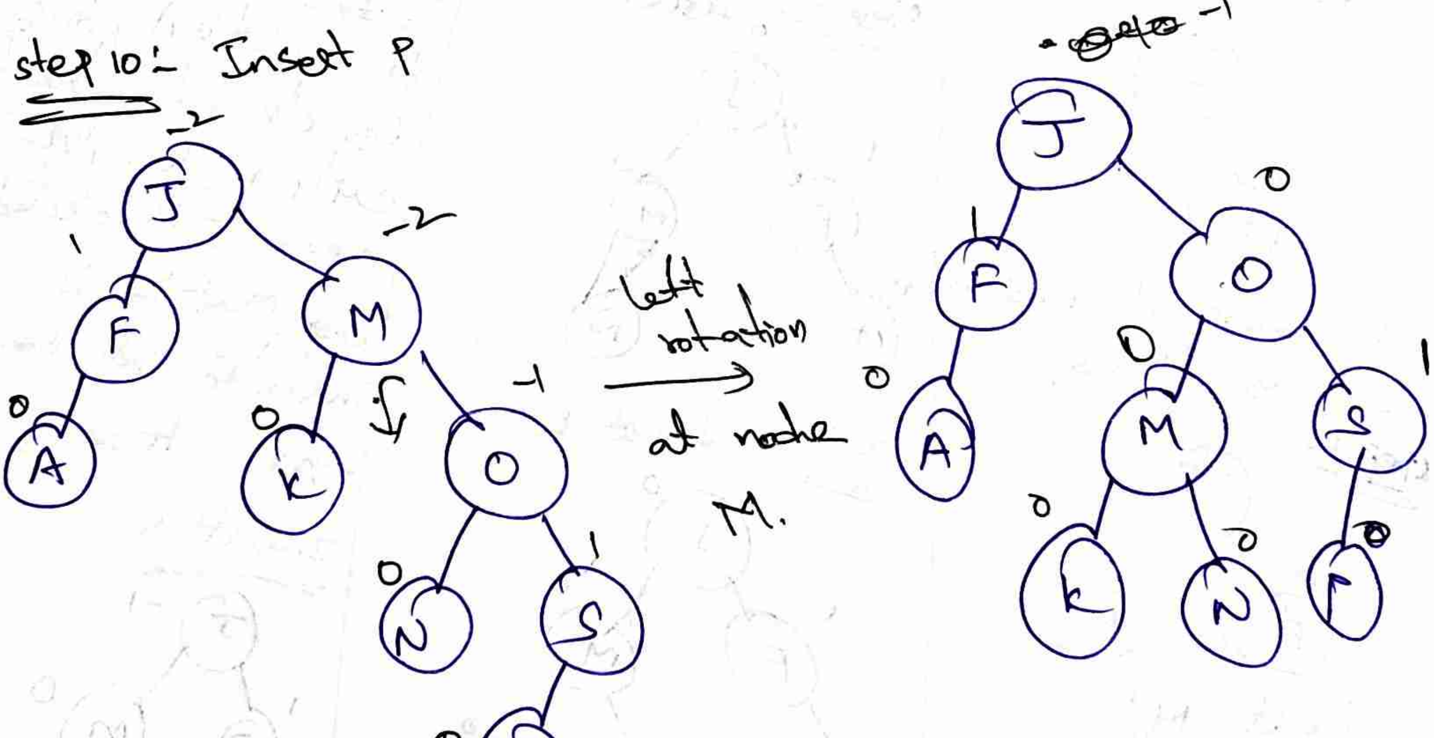
Insert S



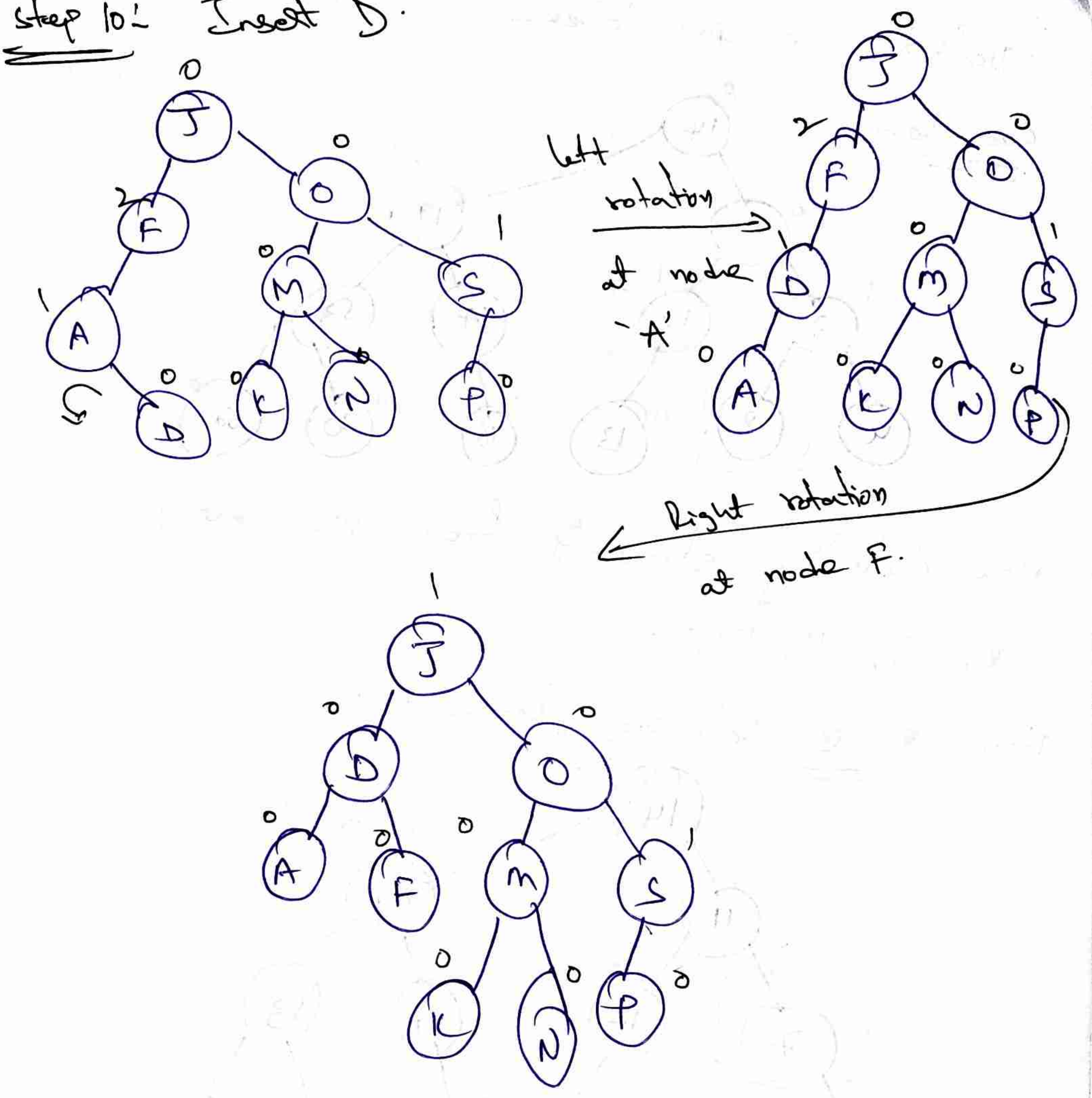
step 9:- Insert O:-



step 10:- Insert P

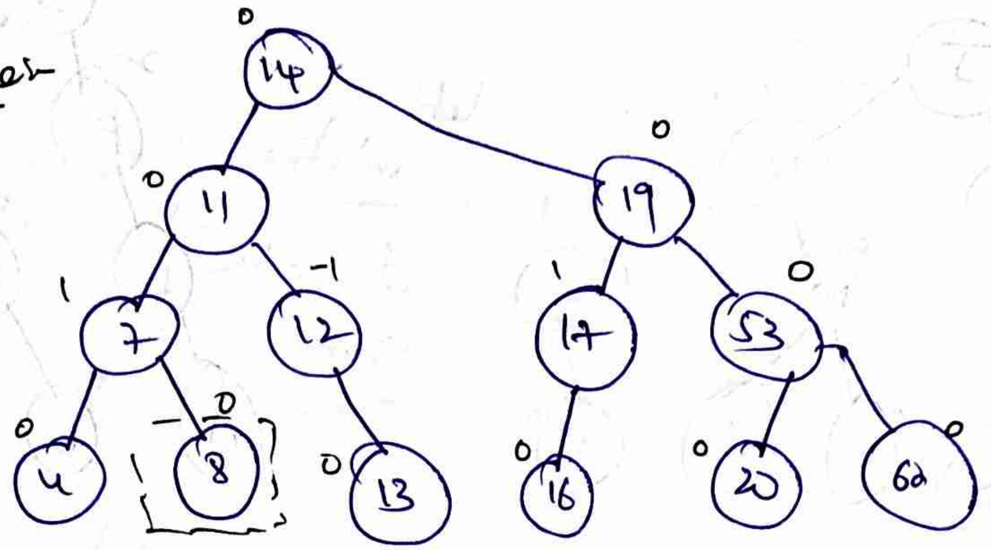


Step 10:- Insert D.



Delection in AVL tree

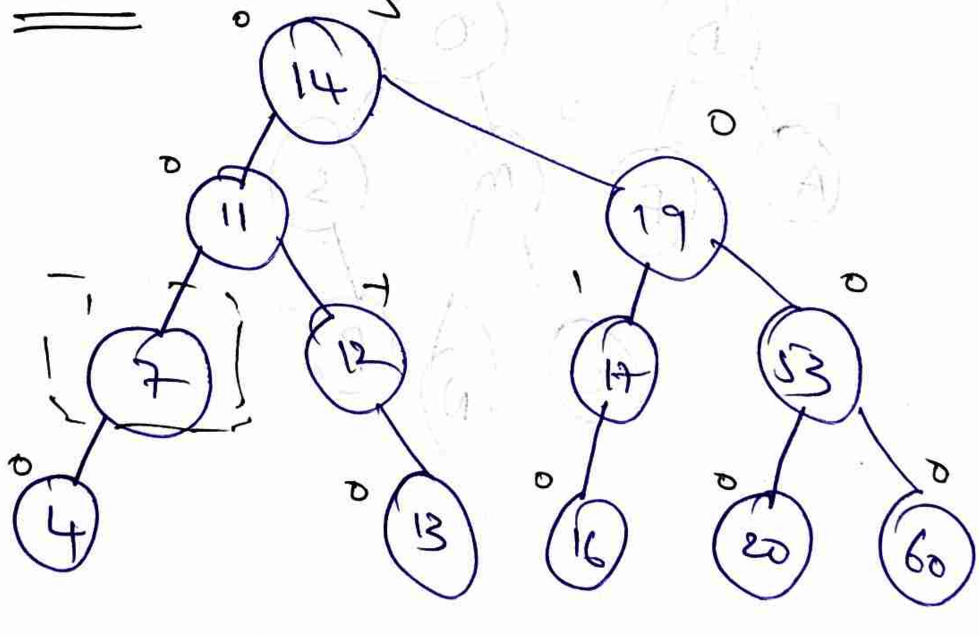
Examples:-



Now deleting elements from AVL ale:

8, 7, 11, 14, 17

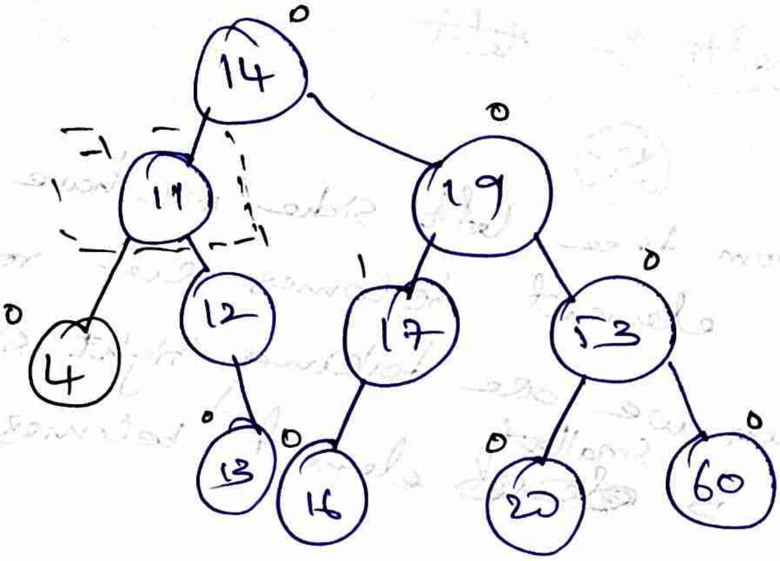
Now 8 is deleting elements.



Balanced AVL tree.

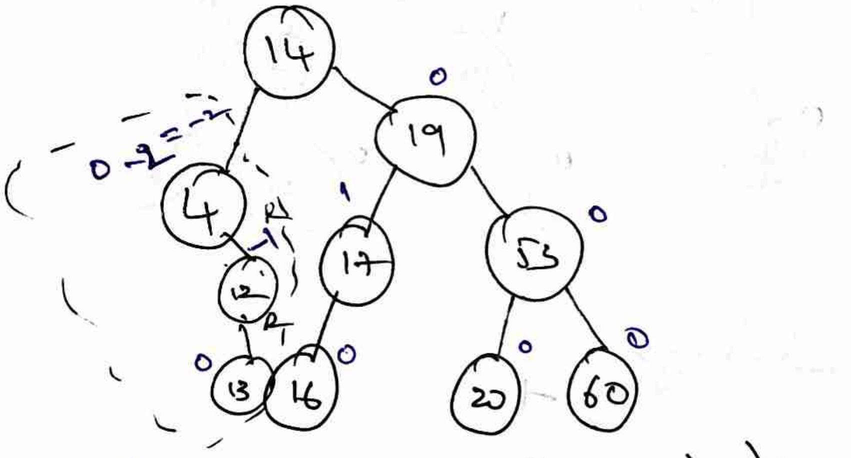
Now deleting 7 :

when we ale deleting '7'. In '7' plance '4' is replaced in the place 7.

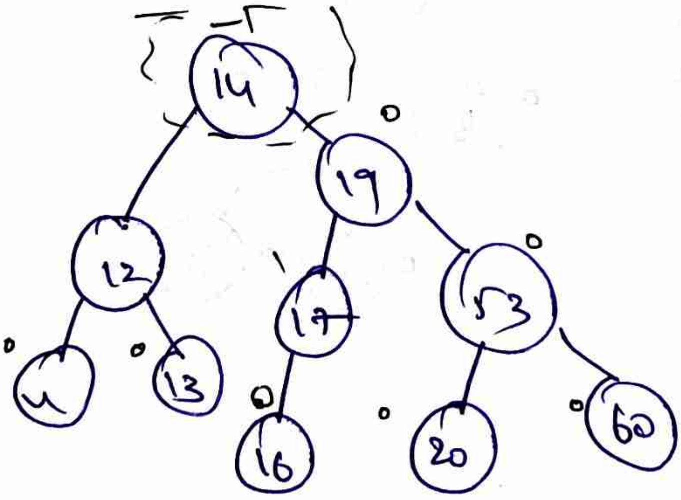


'11' deleting elements :

For '11' has two childrens are there it is replaced left or right.



unbalance AVL RR-Rotation



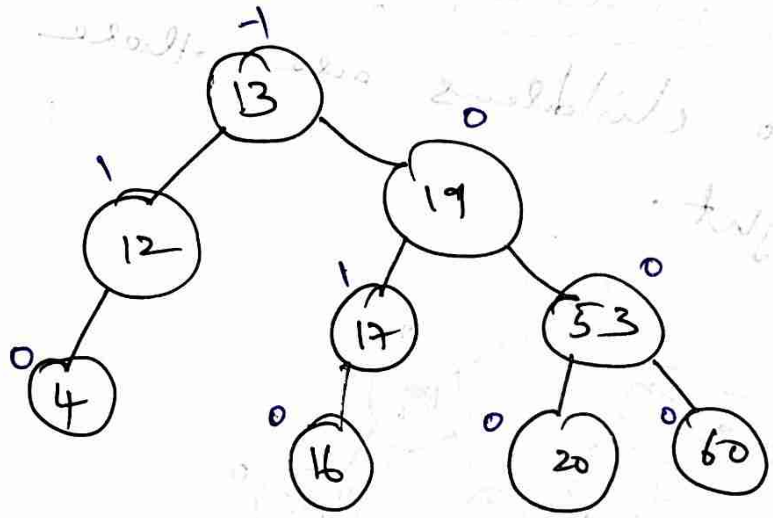
Now 14 - deleting :- ~~left~~

~~13~~

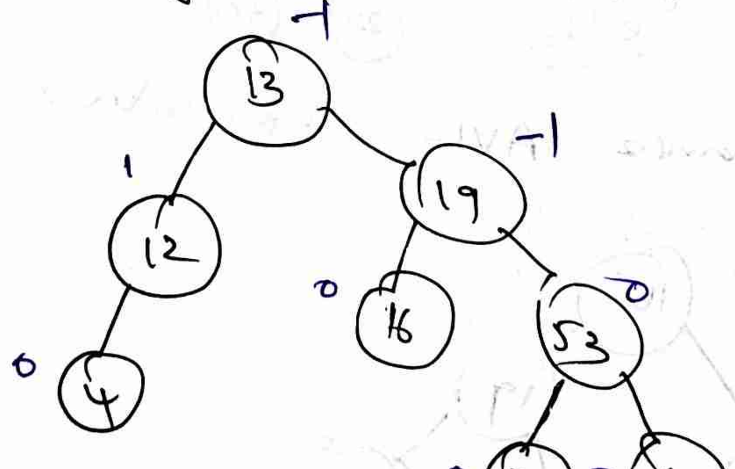
Case 1 :- from tree left side we have see which is largest element becomes the root.

Case 2 :- when we are taking right side we have see which is ~~[illegible]~~ smallest element becomes the root.

Now tree becomes



Now 17 - deleting :-

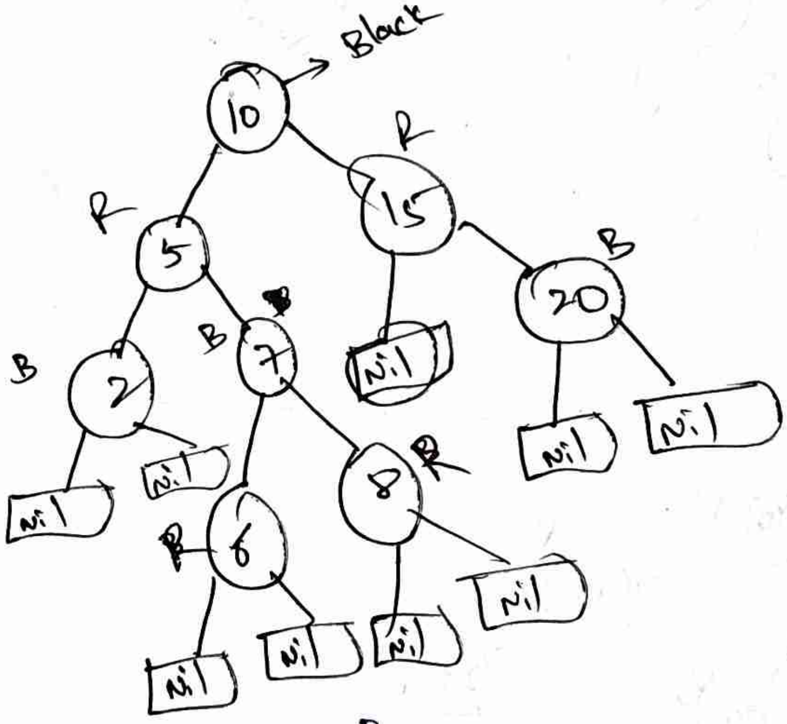


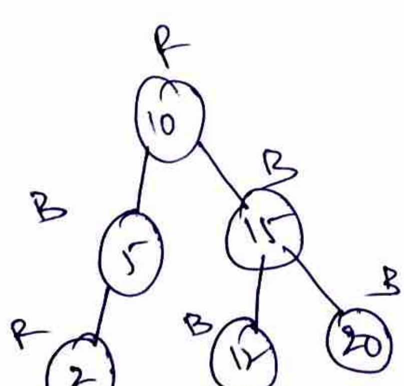
Searching in AVL :- search opelation in an AVL tree is Pelformed exactly same as in an unbalanced binary search tree and thus takes O(logn) time, since an AVL tree is always kept balanced. No special provisions are required as the tree's Structure is not modified by search opelation

Induction to Red-Black tree:

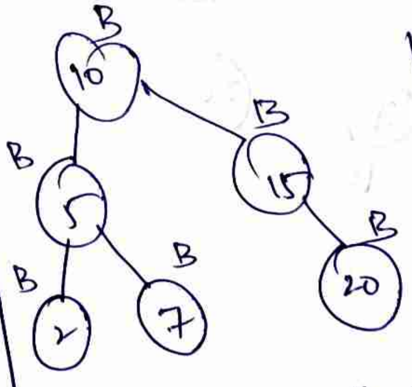
- * It is a self balancing BST.
- * Every node is either Black or Red
- * Root is always Black
- * Every leaf which is Nil is Black
- * if node is Red then its children are Black
- * Every path from a node to any of its descendent Nil node has same no of Black node

Example

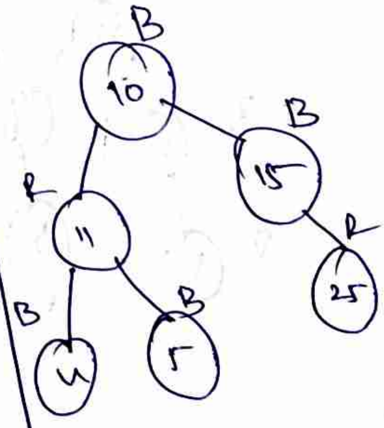




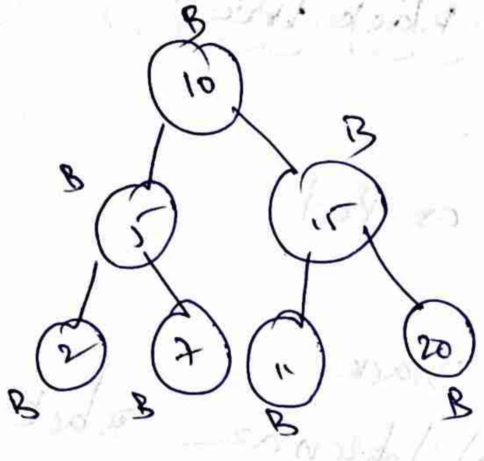
Root is in Red. then its is not red-Black tree



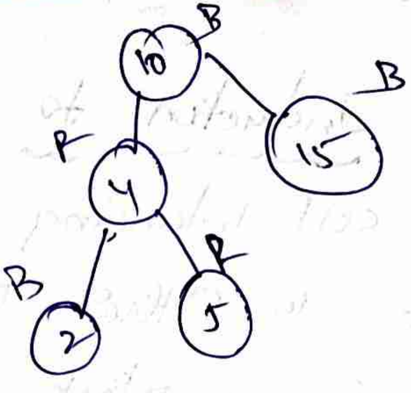
Black node Path is not there



It is not Bst that's y it is not RBT

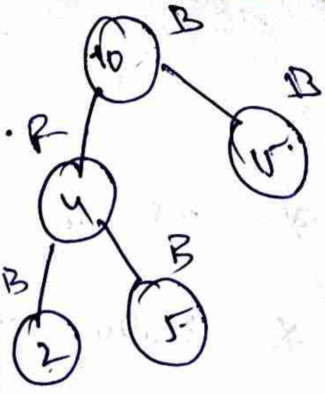


It is RBT

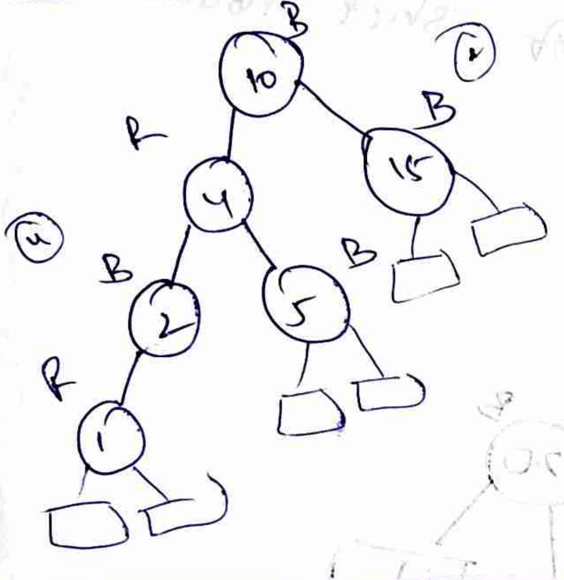


It is nt RBT

two Red nodes

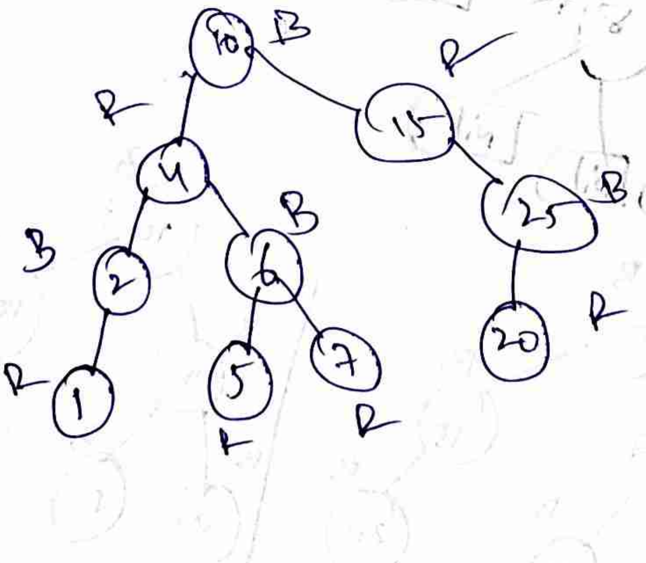


It is RBT



2x2=4

Ton or more we cant extend the tree.



Insertion in Red Black Tree:

* Root = Black

* No two adjacent red nodes

* Count no. of black nodes in each path.

* If tree is empty, Create new node as root node with color Black

* If tree is not empty, Create new node as leaf node with color Red

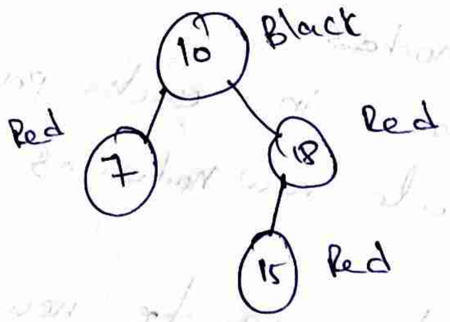
* If Parent of new node is black then exit

* " " " " Red " check the color of Parents sibling new node.

* if color is black or null then do suitable rotation & recolor.

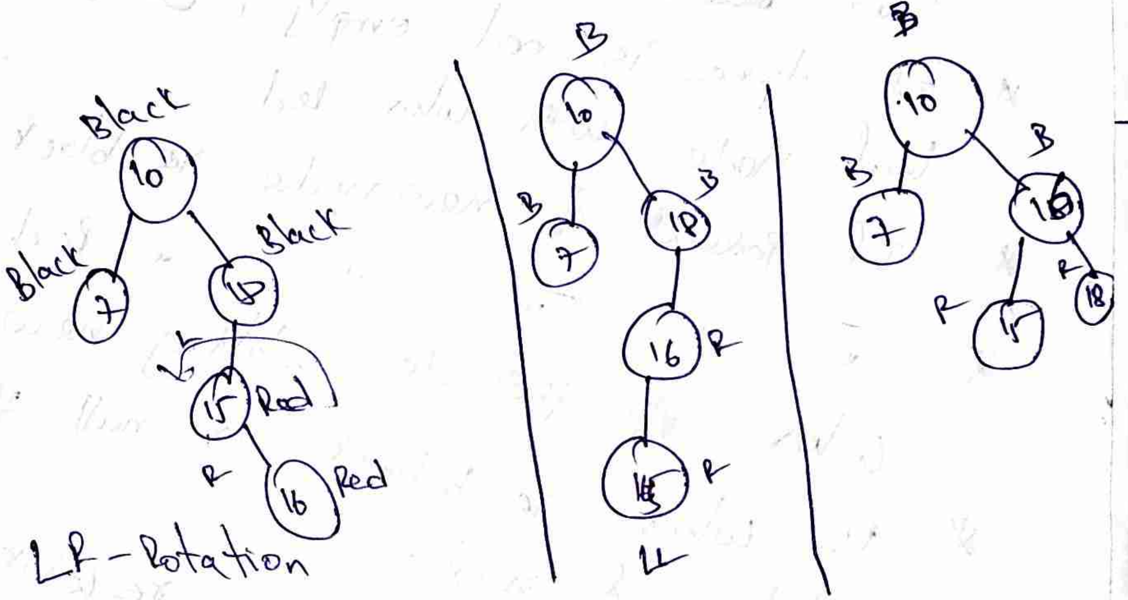
* if color is Red then recolor & also check if Parents parent of newnode is not root node then recolor it & recheck.

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

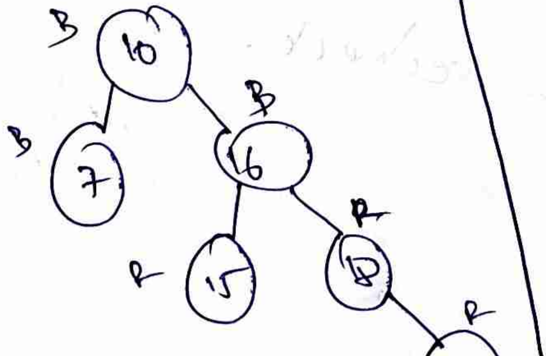


Red - Red relation

Now Recolor

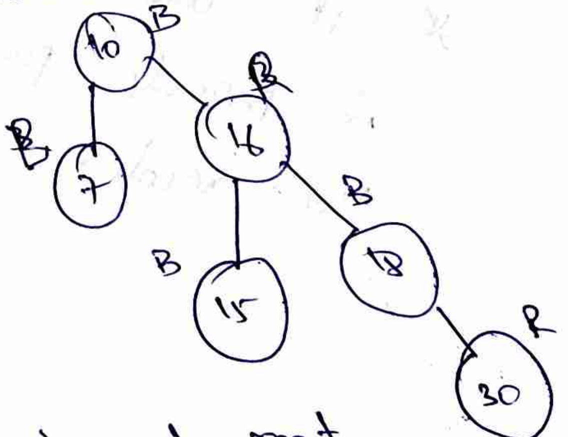


Insert :- 30



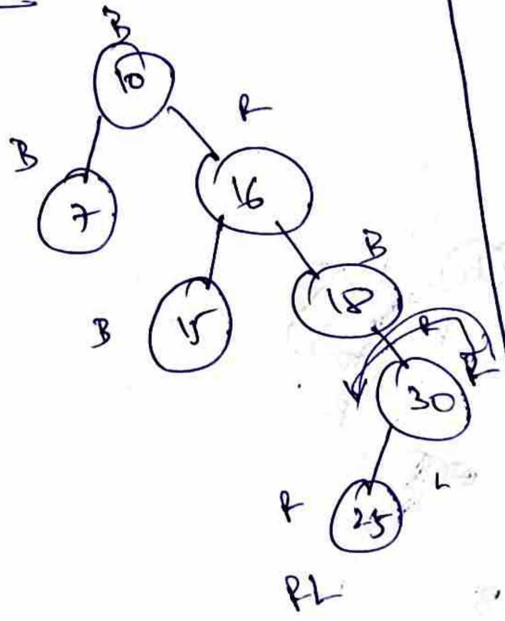
Red - Red

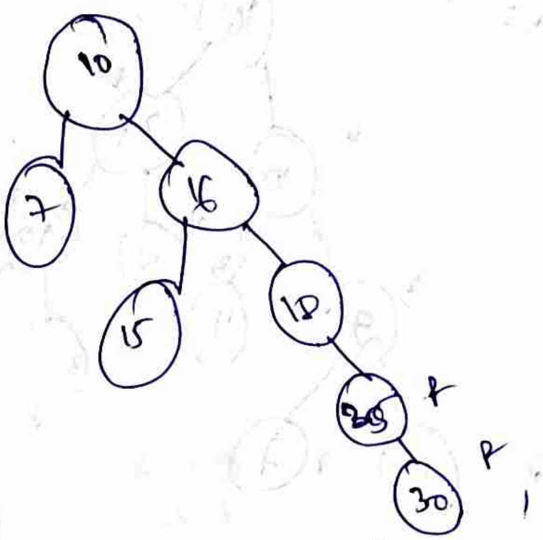
re- colour

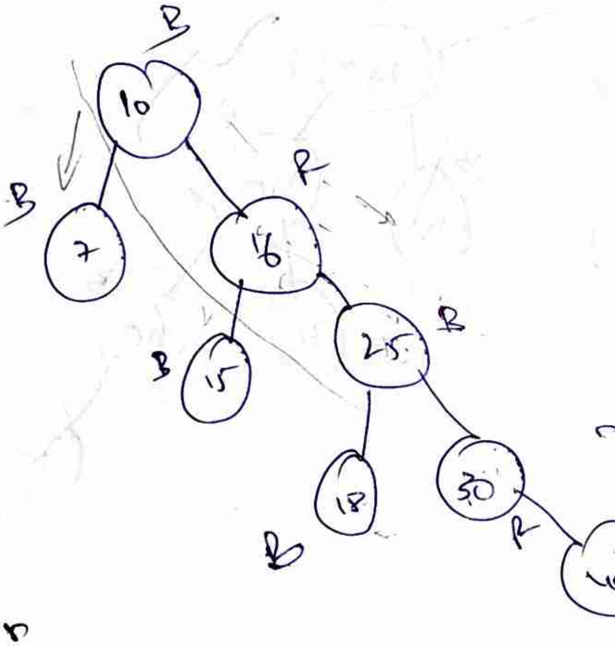


16 is not root
we can change re color
of 6 Now

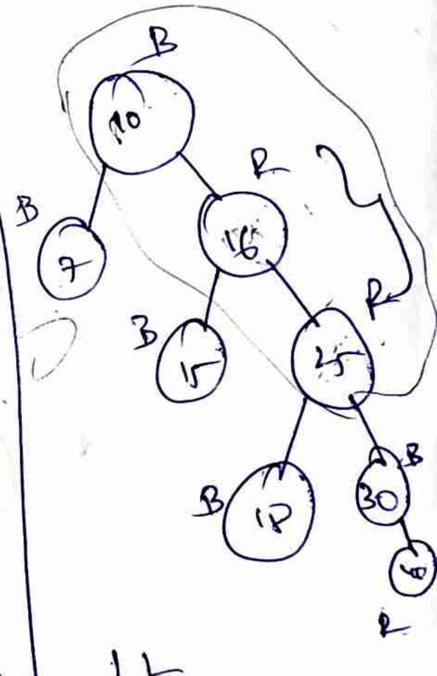
Insertion 25







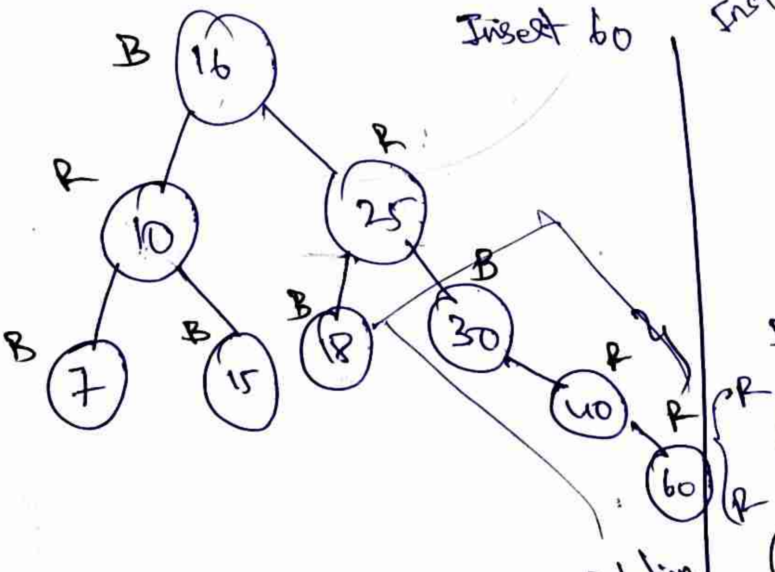
Insert 40



LL

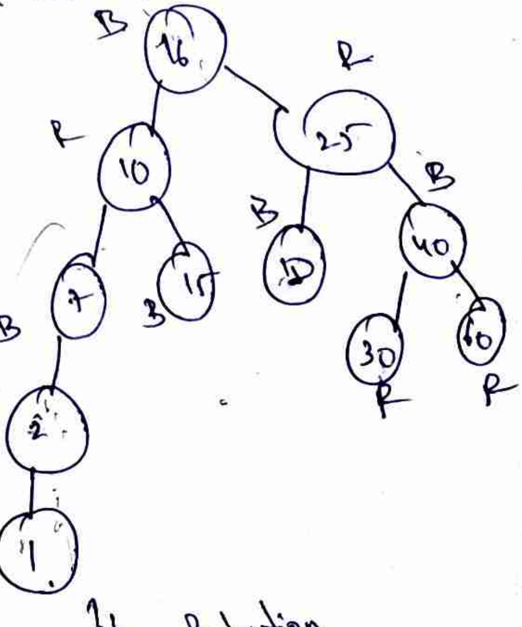
Recolor

Insert 60



RR-Rotation

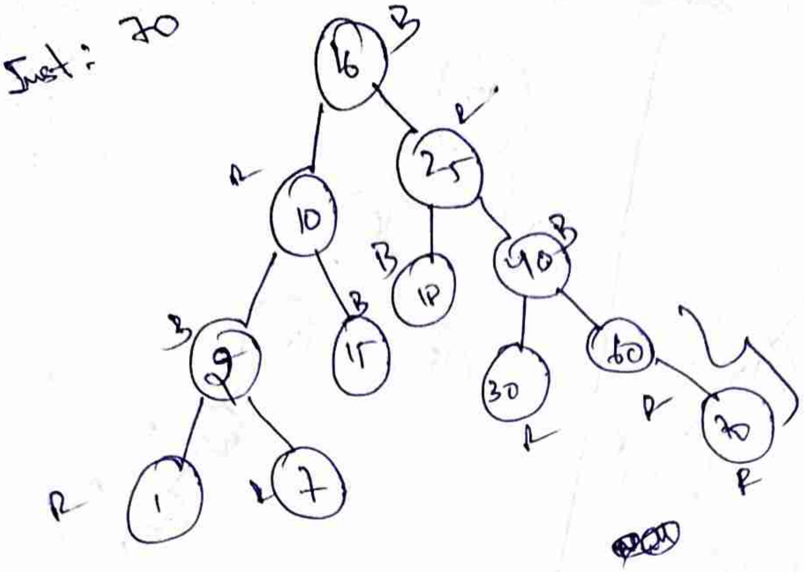
Inst 2 &1



LL- Rotation

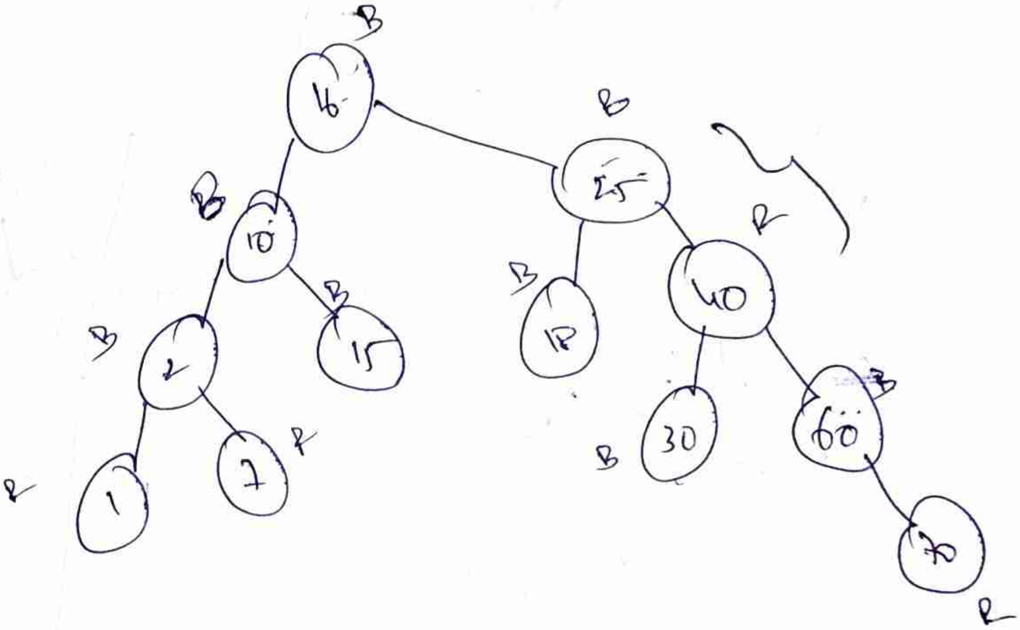
Dele

Inst : 70

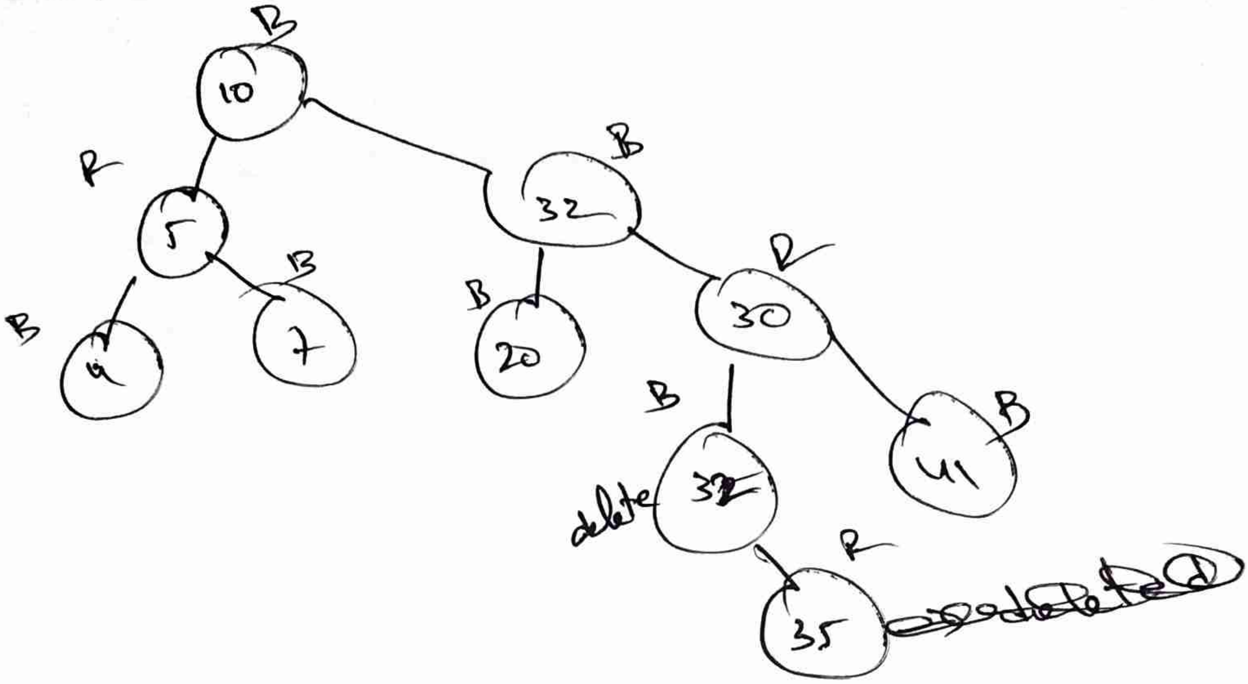


B

⇒

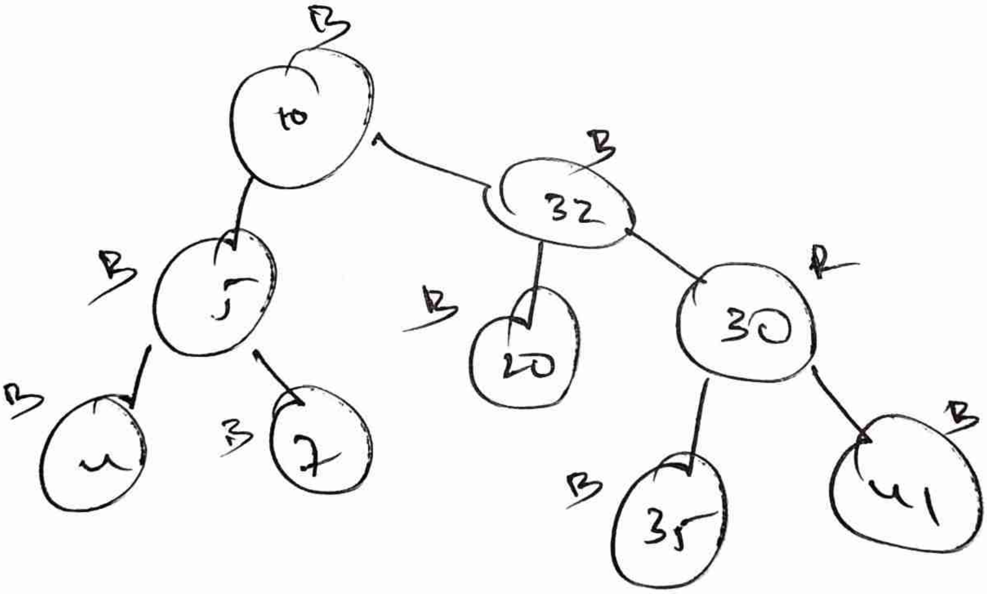


Delection:-



B
10
R
5
B
32
B
4
B
7
B
20
R
30
B
delete
32
B
41
R
35
~~deleted~~

⇒



B
10
B
32
B
5
B
20
R
30
B
4
B
7
B
35
B
41

Red-Black trees: A red black tree is a type of self-balancing binary search tree, typically used to implement associative arrays. It is complex but has good worst-case running time for its operation and is efficient in practice. It can search, insert or delete in O(log n), where 'n' is the no. of elements in the tree.

Each node has a color attribute, the value of which is either red or black. In addition to the ordinary requirements imposed on binary search trees, we make the following additional requirements of a valid red-black tree.

* Every node is coloured either red or black.
* The root node is coloured black.
* Every leaf nil node (known as external node) is coloured
* Both children of every red node are black.
* All paths from any given node to its leaf nodes contain the same no. of black node.

Example :— RBT (Red - Black - Tree)

Self balanced tree. { AVL tre, BST, RBT

* Root Node should be Black

Root → Black

* New node should be Red

New → Red.

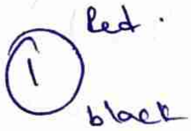
* No. of Black in each path should be equal.

* R → R x.

1, 2, 5, 8, 7, 4 elements.

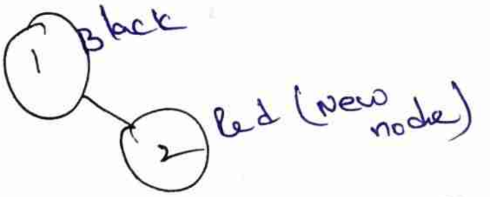
step1:-

New Node ! Insert '1'



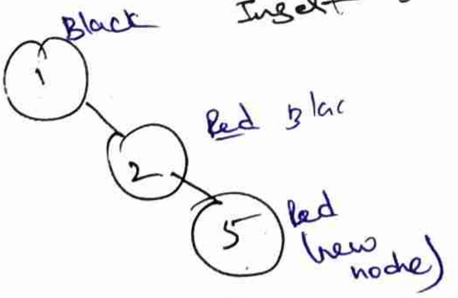
It is also root node we have to give black Node

step2:-



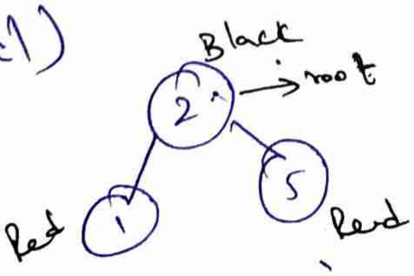
Insert 5

step3:-

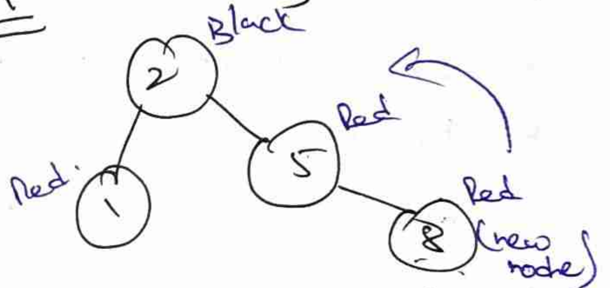


R-R relation ship Condition false (fail)

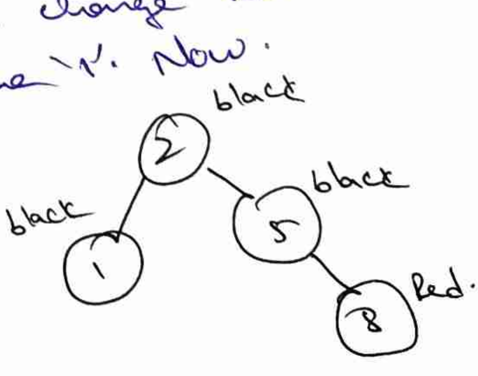
Now



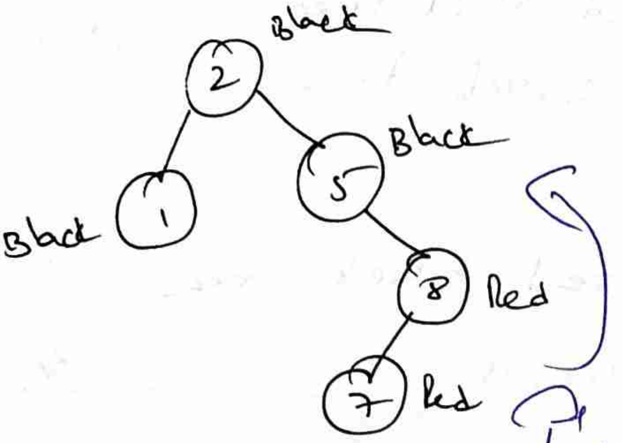
step4:- Inserting 8

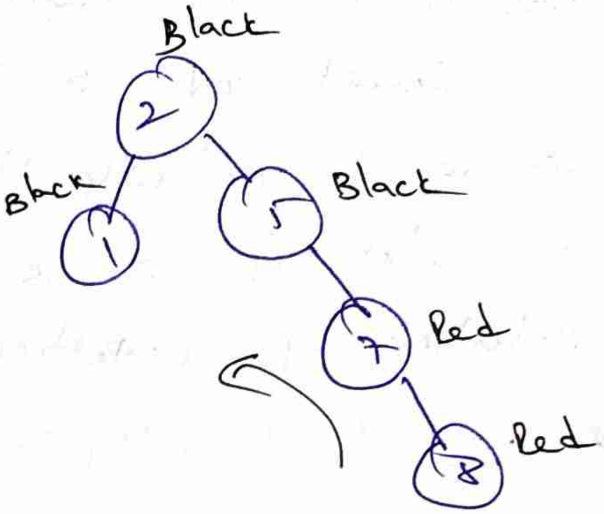


Now R-R (Red - Red) we have to change the colours of 5 ce 8 node because (Red - Red) should be there. Now when we are changing '5' node to black we must be change the colour of node '1'. Now.

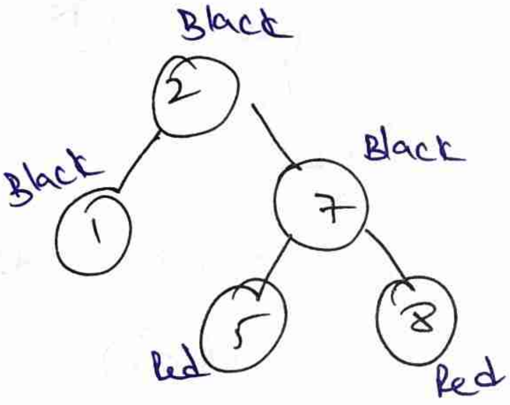


Step 5:- Inserting 7

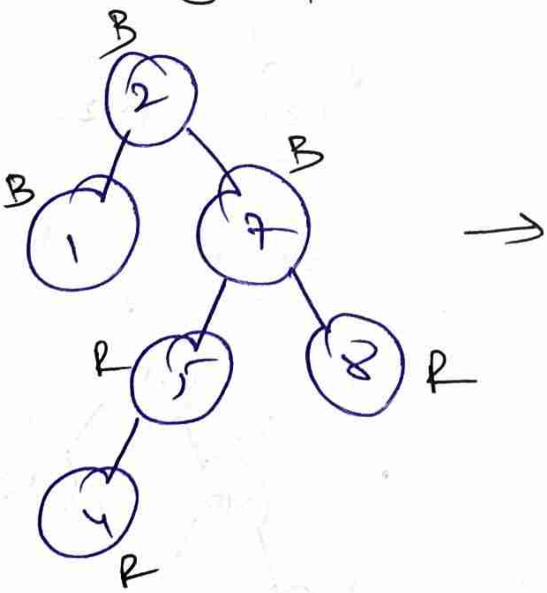


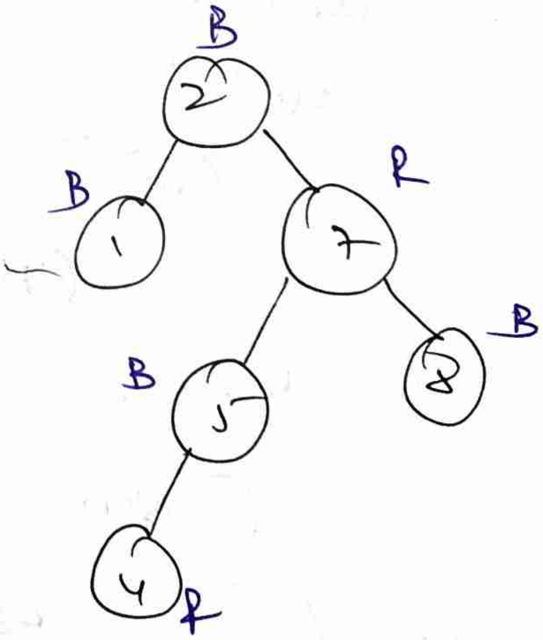


right rotation after first left rotation



Step 6:- Inserting 4





Insertion in Red black tree:-

Insert node z in red black tree.

→ as in ordinary binary search tree.

→ color of z is red.

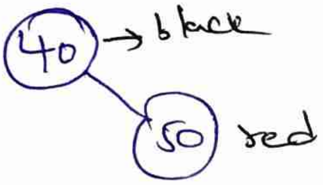
Violation of properties of red black tree

~~Case 1:~~ Example:- 40, 50, 70, 30, 42, 15, 20, 25, 27, 26, 80, 55

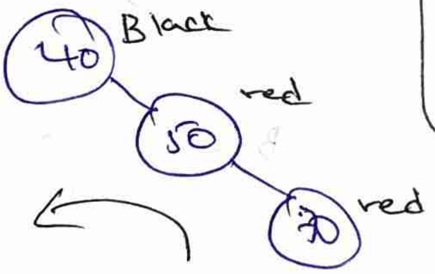
Step 1:- Insert 40

→ root is black.

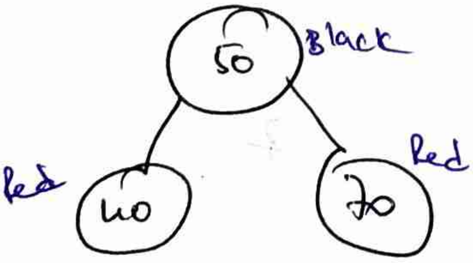
Step 2:- Insert 50



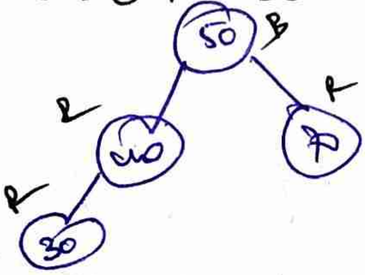
Step 3:- Insert 70

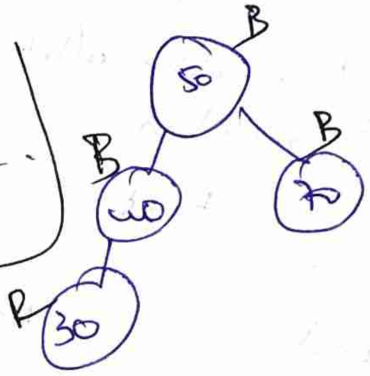


RR imbalancing Simple Rotation

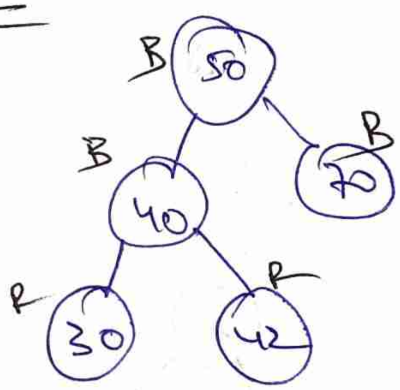


Step 4:- Insert 30

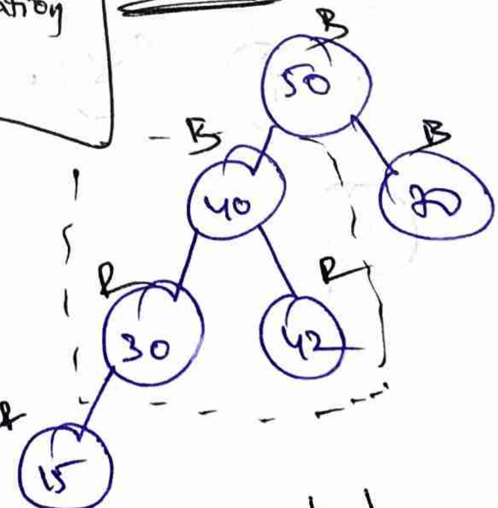




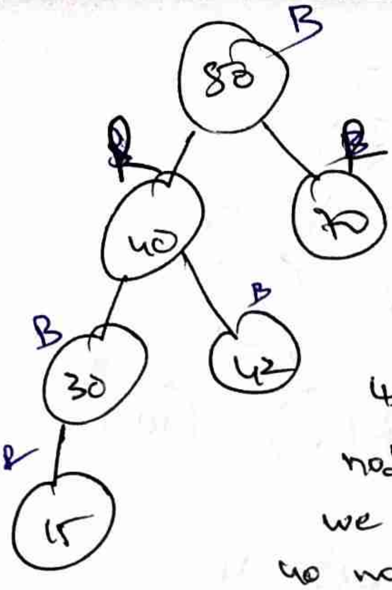
Step 5:- Insert 42



Step 6:- Insert 15

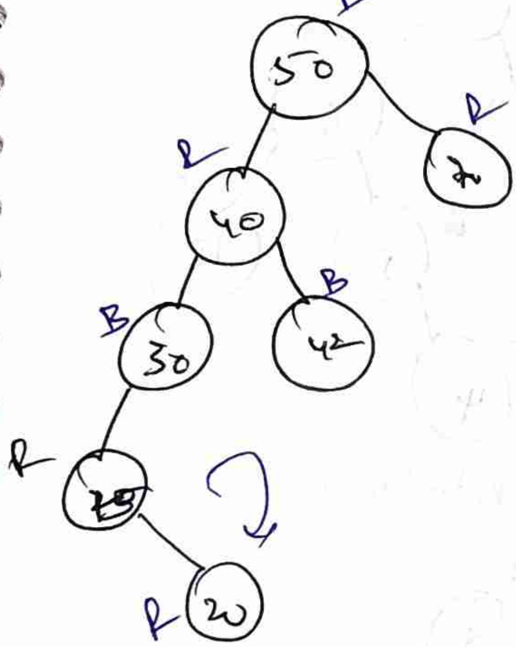


LL rotation.

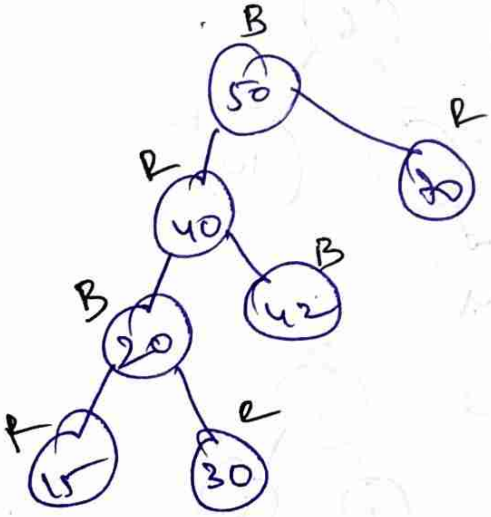


40 is not root node that's y we can make 40 node as Red.

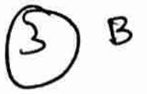
Step:- Insert 20



LR Imbalanced

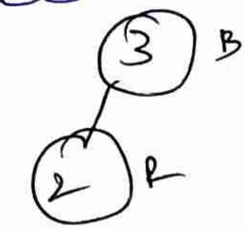


Example 3, 2, 5, 6, 10, 4, 8, 9

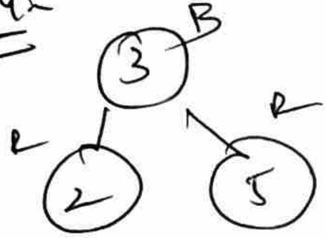


step 2:

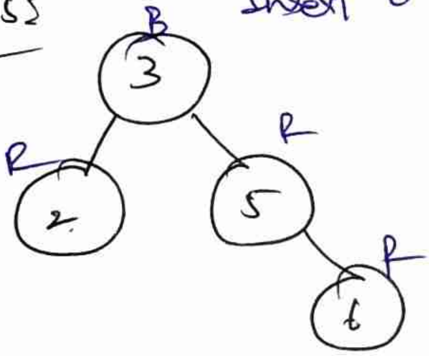
Insert 2

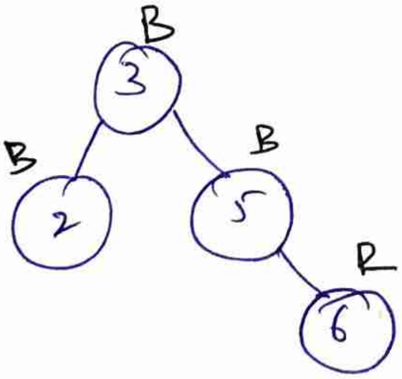


step 4: Insert 5

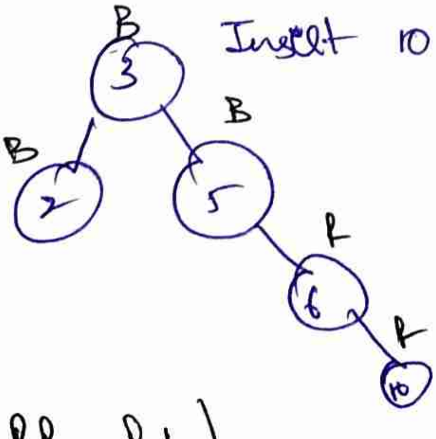


step 5: Insert 6



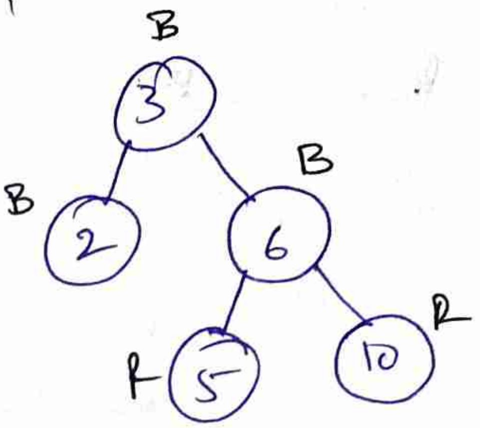


Step 6: Insert 10

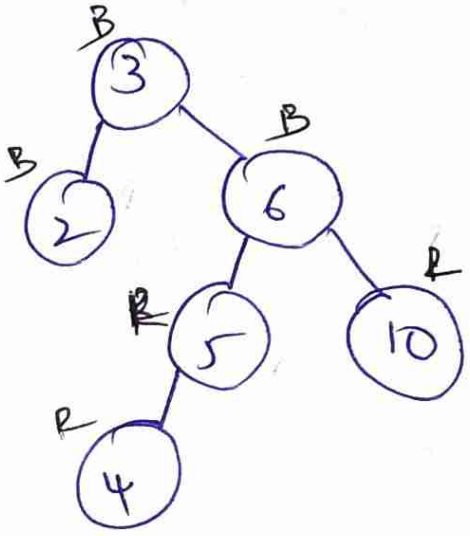


RR Rotation

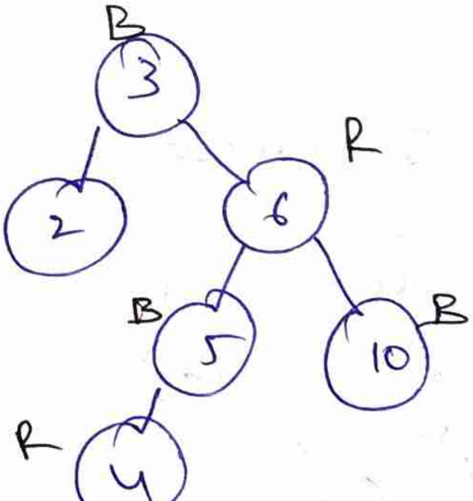
~~Step~~ :



Step 7: Insert 4

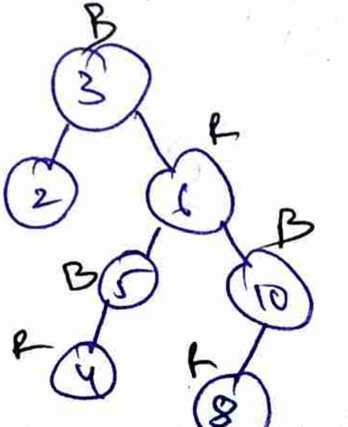


LL Rotation

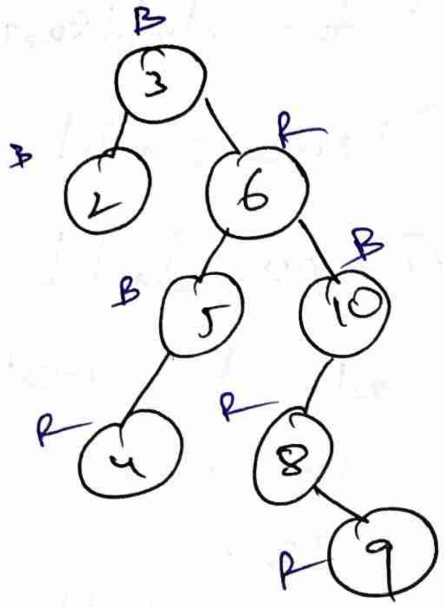


step 8:

Insert 8

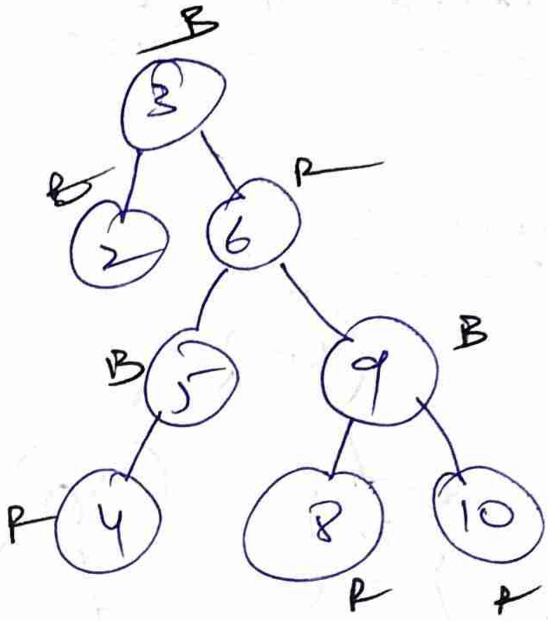


Step9: Insert 9



LR-Rotation

=>



Deletion in RBT

① two children

② one child

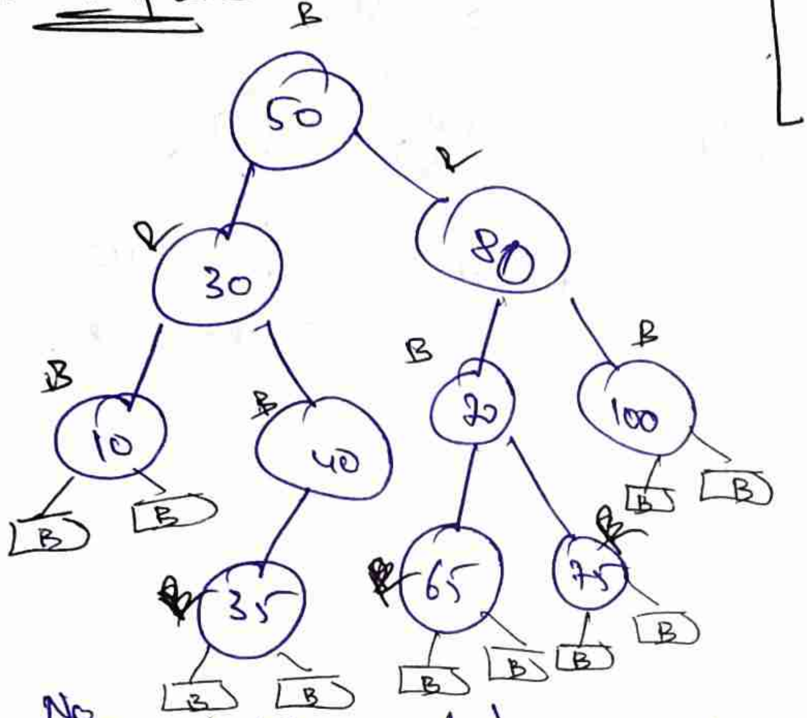
③ no child

only two categoir be there in RBT

-> one child

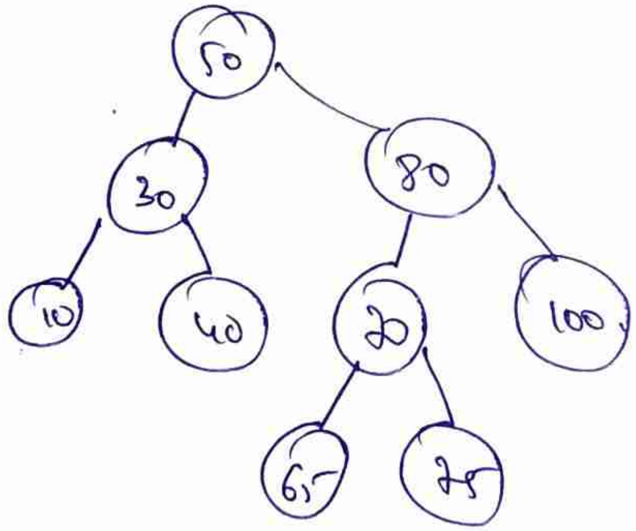
-> no child

Example:

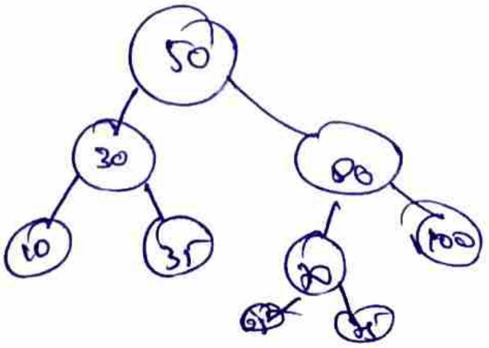


No ~~one~~ child deletion. 35 or 65 . or 75

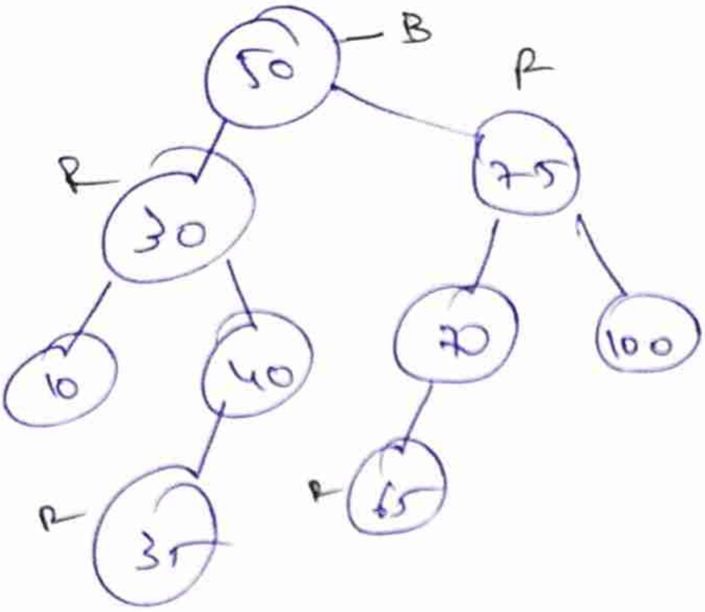
Now 35 we are going to delet.



one child 40 is one child Now.



two children '80'



Splay tree:- A splay tree Con be defined as a self balancing tree with an extra unusual Propelty using which recently accessed elements Con be accessed quickly. The opelations of splay tree like inseltion, deletion and Sealching Consumes o(logn) time Complexity. The pelformance of splay trees for a non-uniform sequence is good Compared to other self balancing search trees like AVL & red-black tree.

Splaying is defined as the Common basic opelation that is used to pelform all the opelations on a binary search tree. The splaying opelation on a tree for a specific elements makes that element as root of the tree

Advantages:-

* It is easy to implement Compared to other self-balancing binary search trees like AVL tree and red-black tree
* It is not required to store any book keeping data which leads to less Memory requirements.
* A pelsistent version of splay tree Con be created to allow for old & latest versions even after the update which Consumes O(logn) space for each update.
* The working of splay trees with similar nodes is good Compared to other self balancing tree

Disadvantages:-

* In Case of uniform access, the pelformance of splay trees is worst Compared to other type of balanced trees
* The sequential access of elements of a sorted tree makes the tree unbalanced.

Rotations:

① zig rotation - Single right rotation.

② zag rotation single left "

③ zig-zig rotation double right "

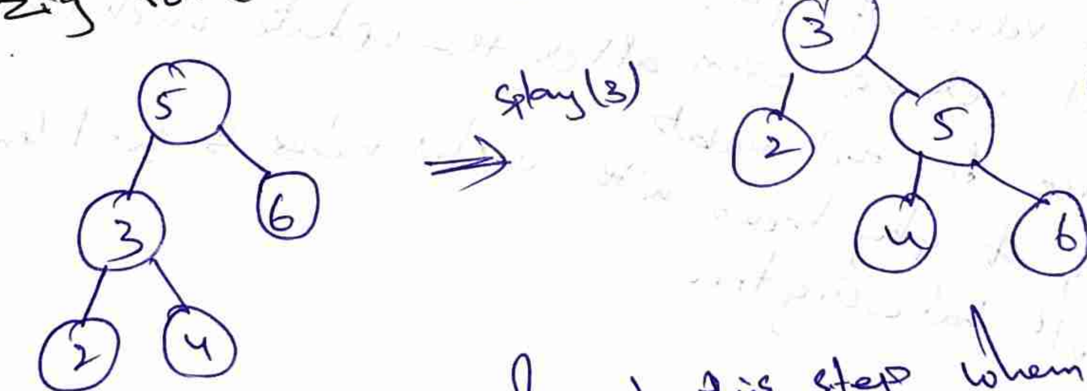
④ zag-zag " " left "

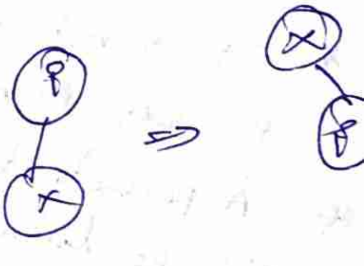
⑤ zig-zag " Right followed by left

⑥ zag-zig " left " " right.

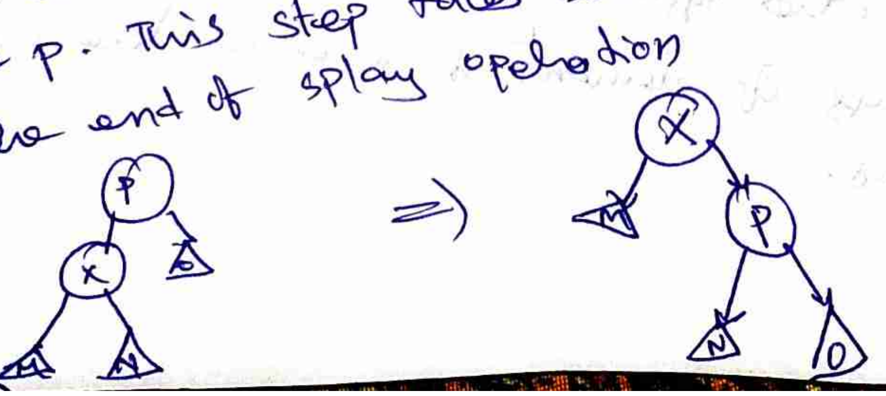
Examples

zig rotation :- Single right rotation.

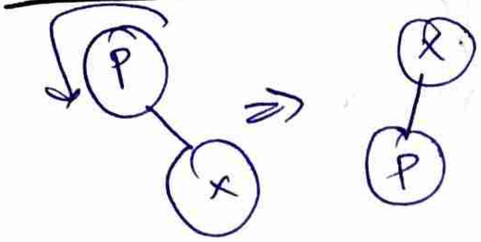


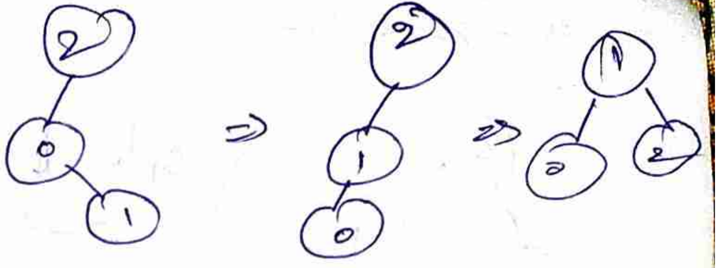


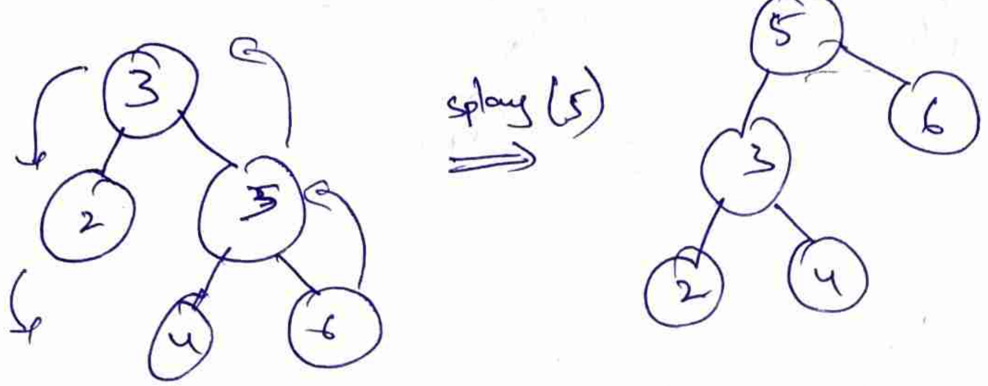
The splay operation performed this step when P is a root node. In this step, the tree is rotated on the edge b/w x and P. This step faces some issues which can be solved at ~~the~~ end of splay operation



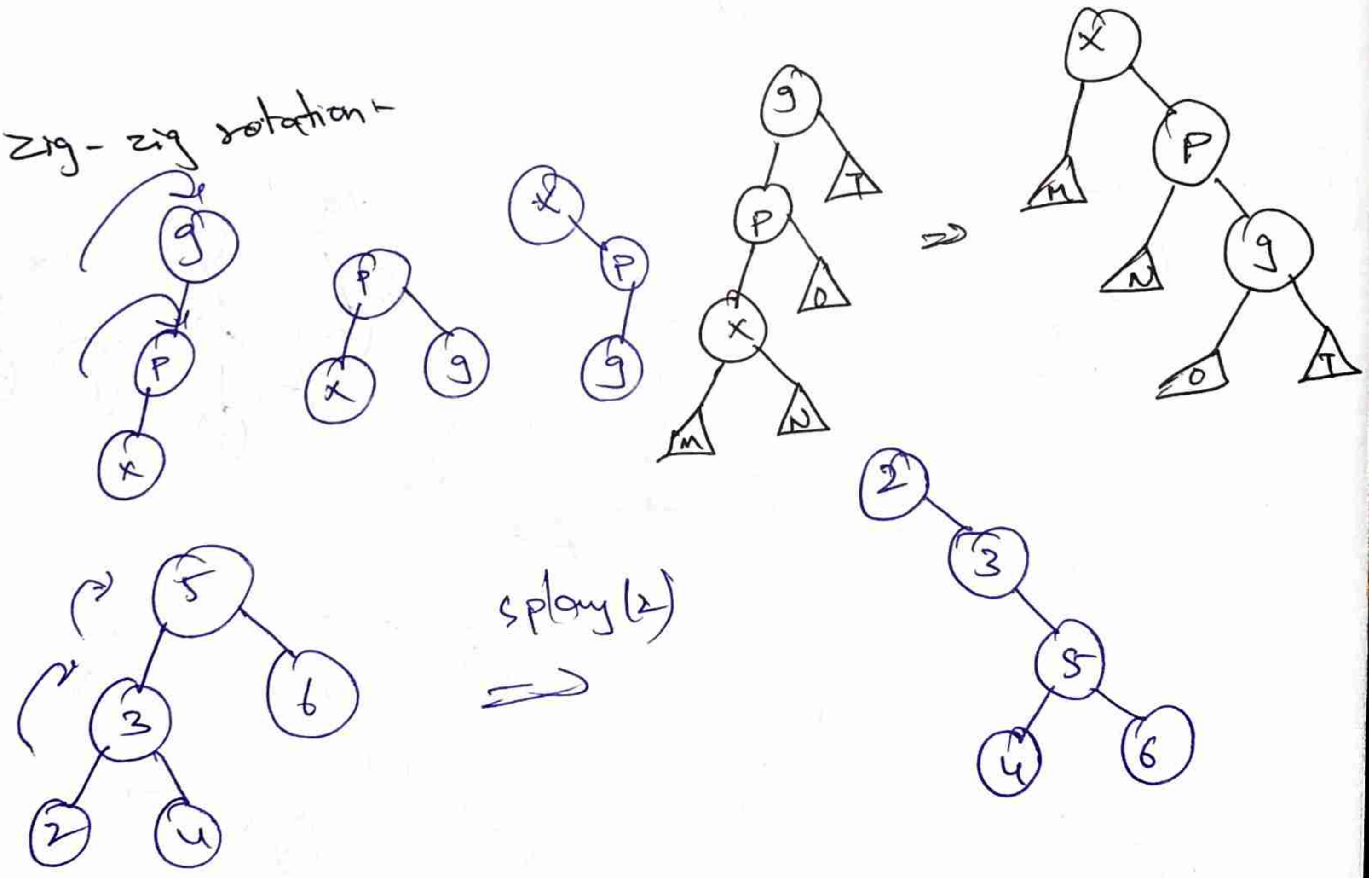
Zag rotation:-



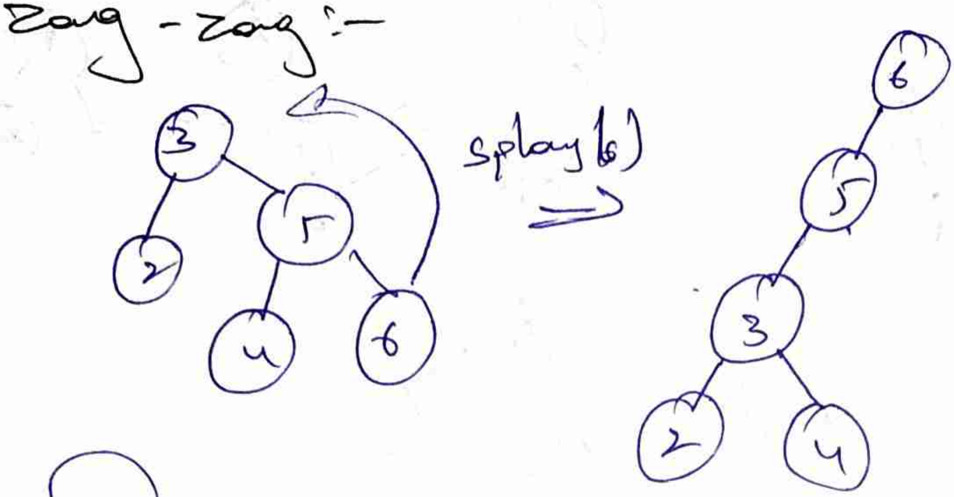


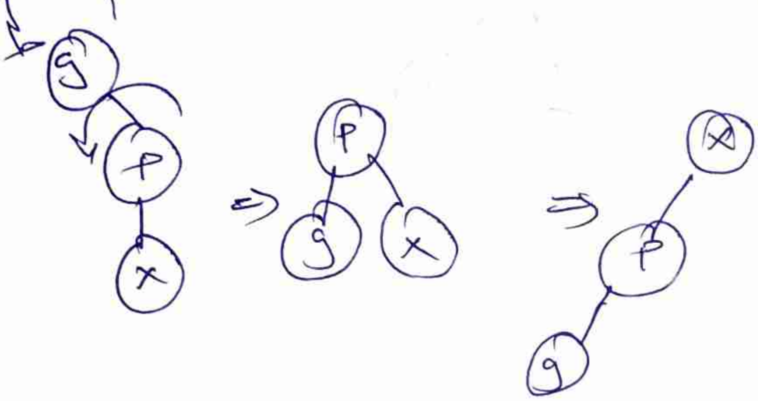


Zig- zig rotation:-

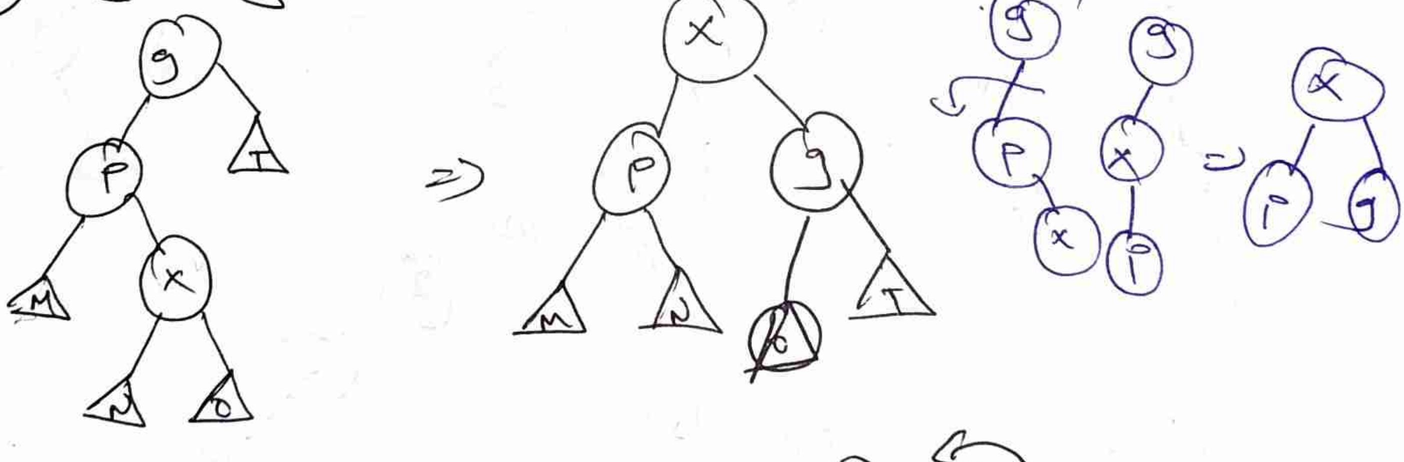


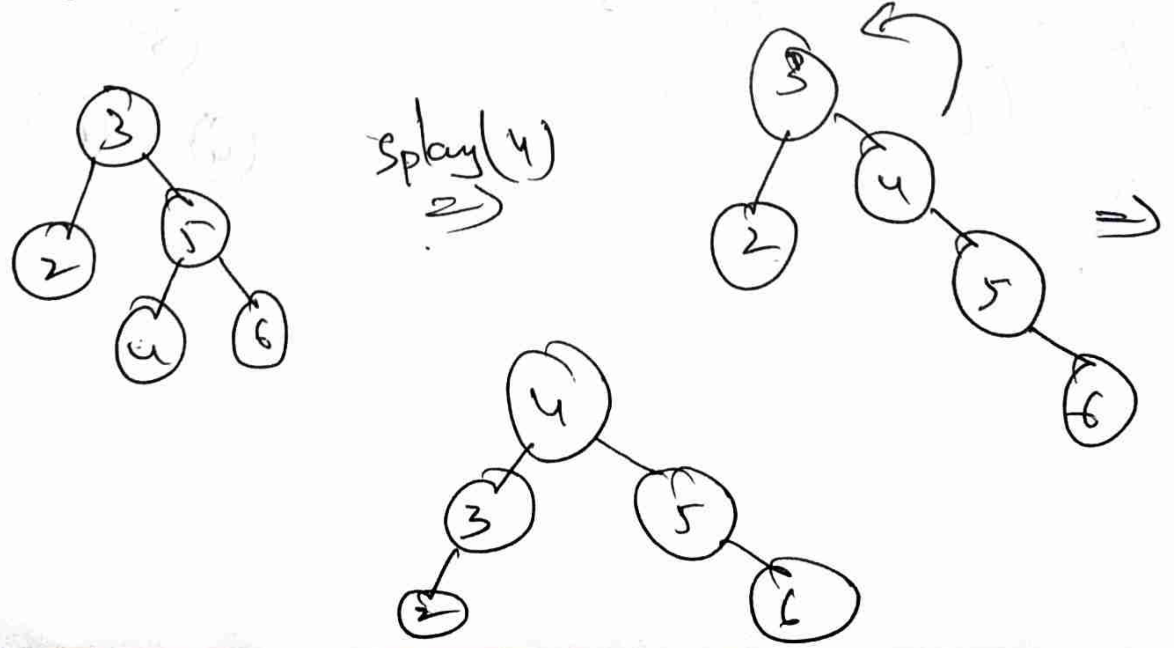
Zag - Zag :-



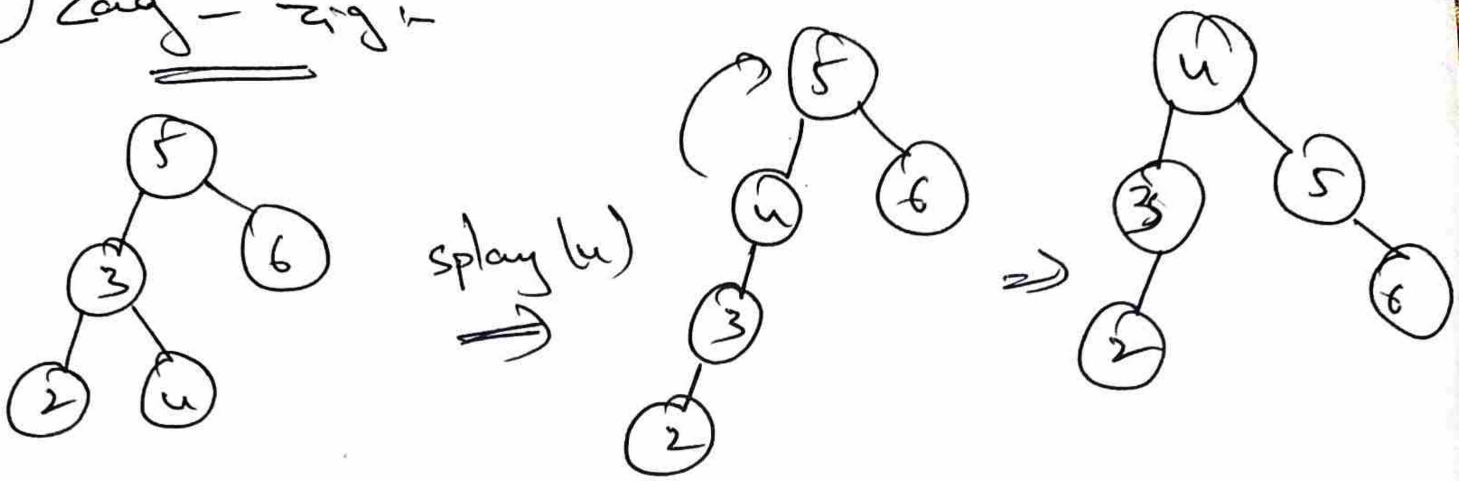


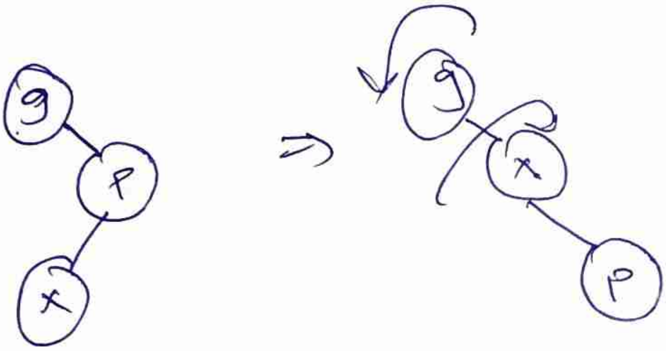
(5) Zig - Zag :-

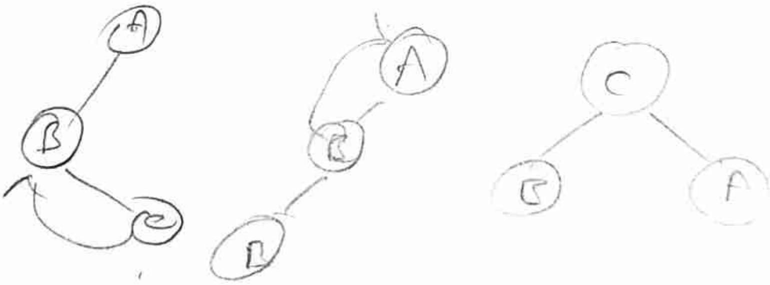




(6) Zag – zig :-







Graphs UNIT - 4

graphs:- A graphs is defined as $G=(V,E)$ wheles

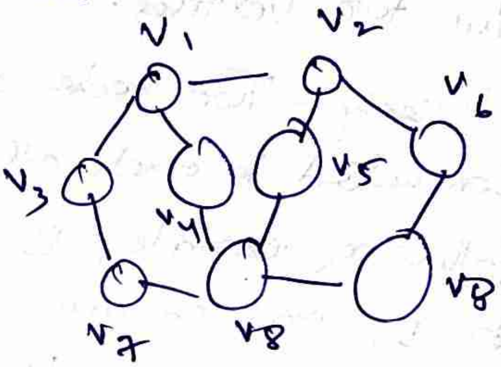
(i) V is the set of elements called nodes or vertices or points

(ii) E is the set of edges of the graphs identified with a uniaue pair (U,V) of nodes.

Here (U,V) pair denotes that there is an edge from node U to node V.

In graphs no rules in connections. A graph G is an ordeled pair of a set V of veltices and a set E of edges

$G = (V, E)$



ordeled pair:

$(a,b) \neq (b,a)$ if $a \neq b$

unordeled pair:

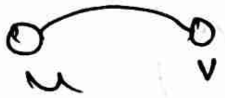
$\{a,b\}$

$V=\{v_1, v_2, v_3, v_4, v_5, v_6, v_7, v_8\}$

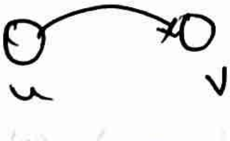
A graphs consists of a finite set of vertices & set of edges which connect a pair of nodes

How do represent an edge?

undirected edges

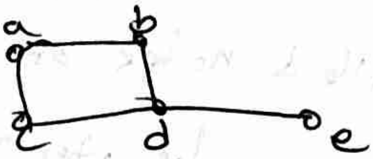


directed edge



The links that connect the vertices are called edge

The interconnected objects are represented by Points termed as vertices



$V = \{a, b, c, d, e\}$

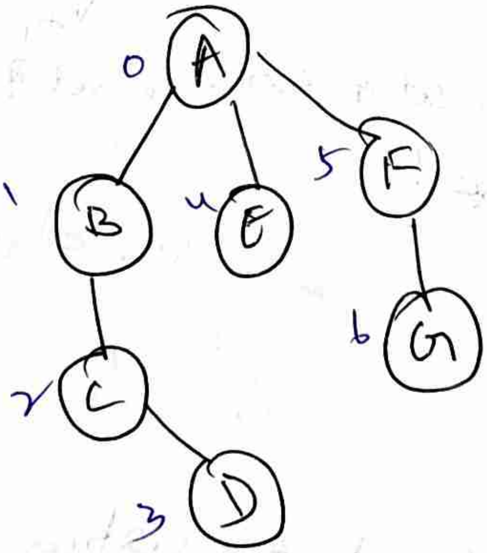
$E = \{ab, ac, bd, cd, de\}$

Vertex:- Each node of the graph is represented as a vertex. In the following example, the labeled circle represents vertices.

Edge:- Edge represents a path b/w two vertices or a line b/w two vertices.

Adjacency:- Two node or vertices are adjacent if they are connected to each other through an edge. In the following example, B is adjacent to A, C is adjacent to B, and so on.

Path:- Path represents a sequence of edges b/w the two vertices In the following example, ABCD represents a path from A to D

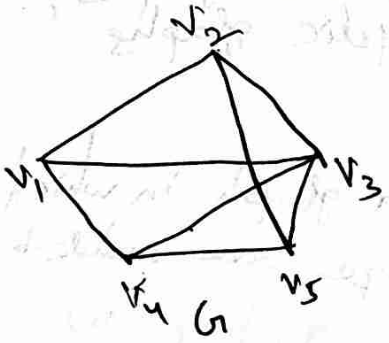


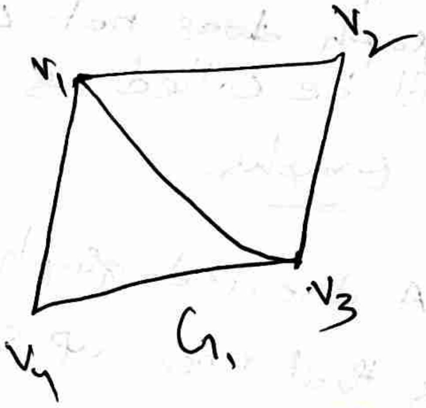
Complete Graph:- If a vertex contains edges to all the vertices from it, then the graph is called complete graphs.

Subgraphs :- Consider two graphs G and G_1, say G_1 is a subgraph of G if,

(i) All the vertices and all the edges of G_1 are in G.

(ii) Each edge of G_1 has the same end vertices in G as in G_1. A subgraph is a graph which is a part of another graph.

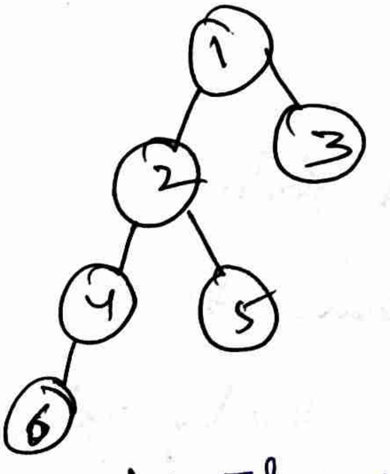


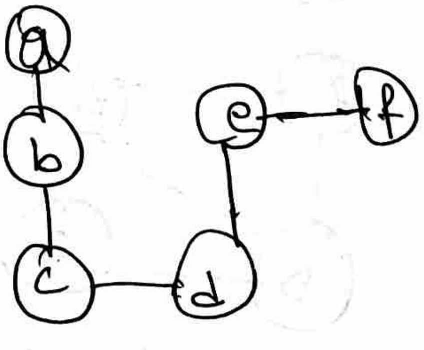


In can be observed that all the vertices and edges of graph G_1 are in graph G and also that every edge in G_1 has the same end vertices in G as in G_1. so it can be concluded that G_1 is a subgraph of G.

Tree :- A tree is defined as a finite set of one or more elements with one elements designated as root and the other elements are divided into trees are called subtrees.

fig below illustrates some of the examples of trees.





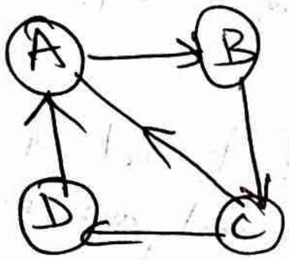
Cycle:- It can ~~not~~ be defined as a circuit in which the terminal vertex doesn't appear as an internal vertex and no internal vertex is repeated. A circuit is a closed walk where no edge appears more than once.

Parallel edges:- If a pair of velfices contains more than one edge then the edges are called as parallel edges The glaph will be called as multiglaph in such cases.

Acyclic Graph:- If there is path contains edges starting from a veltex and ending at the same veltex, then this path is called as cycle. The glaph will be called as cyclic glap. If a glaph does not contain any glaphs, then such glaph will be called as a cyclic glaphs.

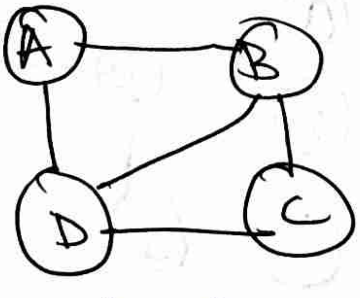
Different Types of Graphs:-

(a) **Directed Graph:-** A directed glaph is a glaph in which the pair of velfices that make up an edge are orddet In such glaph, the order of velfices representing an edge is important



Directed glaph.

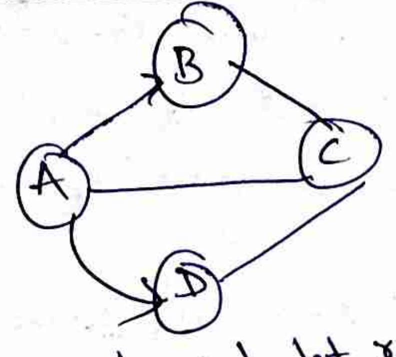
(b) **Undirected Graph:-** In an undirected glaph, the order of pair of velfices is not important.



undirected glaph.

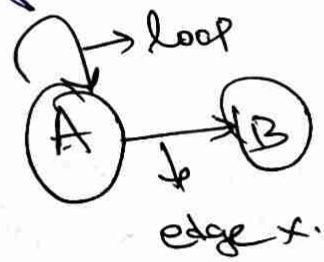
Here (A, B) and (B, A) represent the same edge.

(c) **mixed Graph:-** A glaph in which some edges are directed and some edges are undirected is known as a mixed glaph.



Let (V,E) be a glaph and let x ∈ E be a directed edge associated with the pair of node (U,V). The edge x is said to be initiating or originating in node U and telminating or ending in node V. The node U and V ale also called initral node and telminal nodes of edge x.

An edge of a glaph which connects to itself is called a loop.



In the above glaph, for the edge x, A is the initial node and B is the telminating node.

Graph ADT :- A graph is a data structure that consists of set of nodes (vertices) and set of arc (edges). Every edge present in the graph is indicated by a pair of vertices.

The various graph ADT operations are as follows.

(1) create()

(2) insert Vertex()

(3) delete vertex()

(4) insert Edge()

(5) deleteedge()

(6) boolean isEmpty()

(7) list Adjacent()

(1) Create()! - This Method is used to create an empty graph. The created graph does not contain any vertices and edges.

(2) insertvertex' graph, ve1) :- This Method is used to insert a new vertex ve1 into the graph. The inserted vertex does not contain any adjacent edges.

(3) delete vertex (graph, ve1) :- This Method is used to delete an existing vertex ve1 from the graph. All the adjacent edges that are connected with vertex ve1 are also being deleted.

(4) insertEdge (graph, ve2, ve3)

This Method inserts an edge e1 b/w the vertices ve2 & ve3.

(5) deleteEdge (graph, ve2, ve3):

This Method deletes an edges e1 existing b/w the vertices ve2 & ve3.

(6) Boolean-isEmpty (graph) : This Method checks whether the graph is empty or not. If empty, return true, otherwise returns false.

(7) list Adjacent (graph, vel) : This method returns all the respective edges that are adjacent to the vertex vel.

The program for implementing graph ADT is as follows,

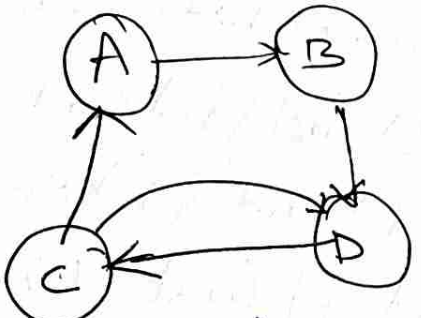
```
Class GraphADT
{
  Public:
   Virtual ~GraphADT( )
   bool isEmpty( ) Const { return vertices = 0 };
   int Numberof vertices ( ) Cost { return vertices };
   int NumberOfEdges ( ) Const { return edge }
   virtual in Degree(int p) Const = 0;
   virtual bool existEdge (int p, int q) Const = 0;
   virtual Void insertVertex (int q) - 0;
   virtual Void insertEdge(int p, int q) = 0;
   Virtual Void ~~[illegible]~~ deleteVertex (int q) = 0;
   virtual void deleteEdge(int p, int q) = 0;

  Private:
     int vertices;
     int edges;

  };
```

Connected and Non Connected graph :- In a graph if there exist a path b/w every pair of vertices then the graph is known as connected graph. In a connected graph, it is possible to traverse from one node to another node. on the other hand if no path exist b/w any pair of vertices then the graph is known as non-connected graph.

Example:-



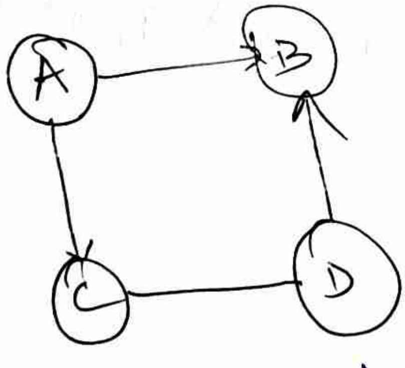
Connected Graph.

The graph is connected becouse there is path from each vertex to every other vertex.

Path A to B : A → B

A to C :- A → B → D → C

A to D : A → B → D



The graph is non-connected becouse there is no path from A to D, there is path from D to any other node.

Difference b/w connected and Non Connected Graph:

A directed graph is said to be strongly connected if there exist a path from every vertex to every other vertex. on the other hand, a directed graph is said be weakly connected if two or more vertices in the graph are not connected.

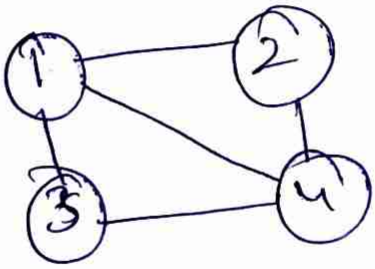
An undirected graph is called a connected graph, if every node in the graph can be reached from any other node. Graphically, a connected undirected graph consist of single connected component which is a connected sub graph.

Adjacency matrix:- An adjacency matrix $A = (a_{ij})$ of graph G is defined as,

$$a_{ij} \begin{cases} 1 & \text{if } v_i \text{ is adjacent to } v_j \\ 0 & \text{otherwise.} \end{cases}$$

In case of directed graph "v_i is adjacent to v_j" means that there is a directed edge from v_i to v_j.

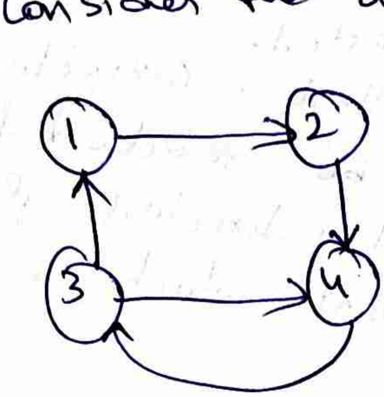
Example:- Consider the undirected graph given below.



Adjacency matrix is

$$A = \begin{array}{c|c|c|c|c|} & 1 & 2 & 3 & 4 \\ \hline 1 & 0 & 1 & 1 & 1 \\ \hline 2 & 1 & 0 & 0 & 1 \\ \hline 3 & 1 & 0 & 0 & 1 \\ \hline 4 & 1 & 1 & 1 & 0 \\ \hline \end{array}$$

Example:- Consider the directed graph shown below,



Here a_{ij} has 1 when there is an edge from V_i to V_j

The adjacency matrix A shows the no. of paths of length 1 b/w any pair of vertices (V_i, V_j). The adjacency matrix A^2 shows the no. of paths of length 2 b/w any pair of vertices.

Example:- Consider the adjacency matrix for the the graph given in example.

Graph traversal Methods! The various types of ~~graph~~ graph traversal methods are as follows

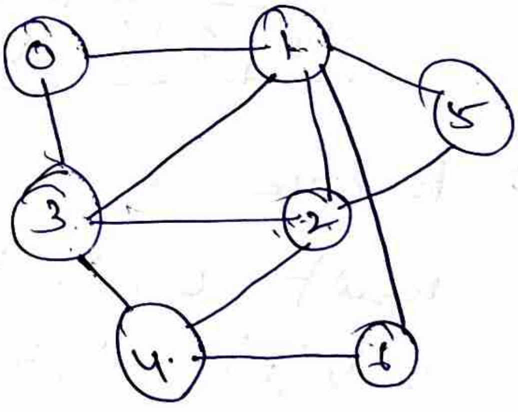
Two types :-

BFS (level-order) : Breadth First Traversal

DFS : Depth First traversal

Breadth First Traversal: In this method traversal is started from given vertex v and all the nodes adjacent to v from left to right are visited Data structure queue is used to keep track of all the adjacent nodes. The first node visited is the first node whose successors are visited.

Example:-



1. state[0] = 1
Push it into queue

0	

2. while queue is no empty.

i) Remove 0.
state[0] = 2 [0 is visited]

Result : 0

Now '0' is deleted from queue then '0' is visiting 2 node that two nodes going to inserted into the queue. Now queue becomes.

ii) insert nodes adjacent to 0 i.e., 1 && 3 to queue

1	3	

state[1] = 2
state[3] = 2

for[1] 0, 3, 2, 5 && 6 unvisited vertes.

3: i) Remove 1

Result:- 0 1 (deleted elements from queue)

state[1] = 0, 3, 2, 5 & 6

But 0, 3 are visited nodes only unvisited are 2, 5 & 6.

3	2	5	6	

(ii) Remove 3

Result:- 0, 1, 3

State[3] = ~~0, 1, 4, 2~~ 0, 1, 4, 2

state[3] = 4 statee[3] = 4.

But 0, 1, 2 are visited nodes

2	5	6	4	

4: Remove 2

Result 0, 1, 3, 2

State[2] = 1, 3, 4, 5

visited = 1, 3, 4, 5 ~~Alon~~

No need to insert in to the queue.

5. Remove 5

Result 0, 1, 3, 2, 5

state[5] = 1, 2

No need to insert in to the queue.

6. Remove 6 Result 0, 1, 3, 2, 6

state[6] = 2 (1 & 4)

No need to inserted

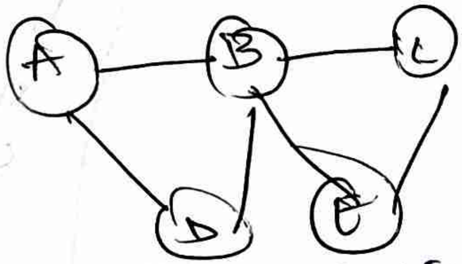
7. Remove 4

Output: 0, 1, 3, 2, 5, 6, 4

state[4] = 3, 6, 2, visited.

no need to inserted.

Example 2:-



1. state[A] = 1

Push it into queue | A |

2. while queue is not empty,

(i) Remove A

state[A] = 3 {A} is visited}

(ii) Insert nodes adjacent to A i.e

D & B to queue

| D | B |

state[B] = 2

state[D] = 2

3(i) Remove D

state[D] = 3 {D is visited}

| D | B |

(ii) No node adjacent to D are in ready state

4. (i) Remove B state[B] = 3 {B is visited}

(ii) Insert E, C which are adjacent to B and are in ready state | E | C |

5. (i) Remove E,

state[E] = 3 {E is visited}

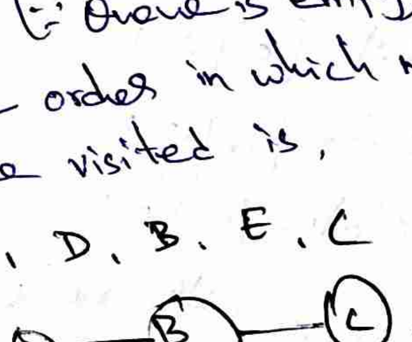
(ii) Remove C,

state[C] = 3 {C is visited}

stop (∵ Queue is empty)

The order in which nodes are visited is,

A, D, B, E, C

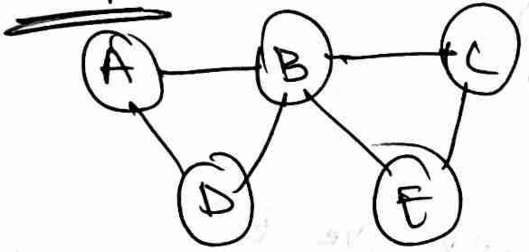


The main difference in BFS traversal from DFS the use of queue instead of a stack.

Hence in the example D is removed before B i.e its adjacent nodes are inserted.

Depth First Traversal :- In this method, traversal is started from a given vertex v is the graph. This vertex is marked as visited and any of the unvisited vertex adjacent to v is visited. Then neighbours of v is visited. This process is continued until no new node can be visited. Now this is backtracked to visit unvisited vertices if any left. stack is used to keep track of all nodes adjacent to a vertex

Example: Consider the graph.



let node A be the starting node

1. state[A] = 1
   state [B] = 1
   state [c] = 1
   state [D] = 1
   state [E] = 1

[Initializing all the nodes to ready]

(2) Begin with node A. Push it onto ~~statact~~ stack.

(3) while stack ≠ empty

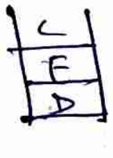
(i) POP A state[A] = 3 [A is visited]

(ii) Push nodes adjacent to A onto stack i.e make their state ~~to~~ waiting.

state[B] = 2
state[D] = 2

(4)(i) POP B
state[B] = 3 [B is visited]

(ii) Push nodes adjacent to B in stack
state [C] = 2
state [E] = 2



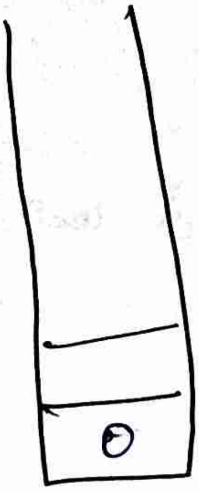
5 (i) POP C
state[C] = 3 [c is visited]

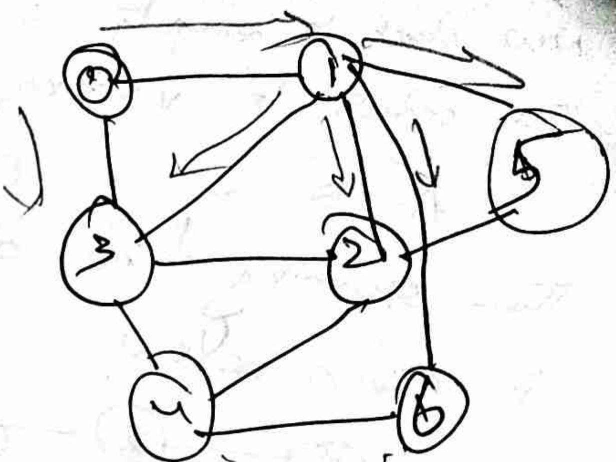
(ii) node adjacent to c are B and E but they are not in ready state. so they are not pushed onto stack.

6 (i) POP E
state[E] = 3 [E is visited]

(ii) Nodes adjacent to E are B and C. They are not in ready state so they are not pushed onto stack

(iii) POP D
state [D] = 3 [D is visited]
No nodes adjacent to D are in ready state. so they are not pused onto stack.



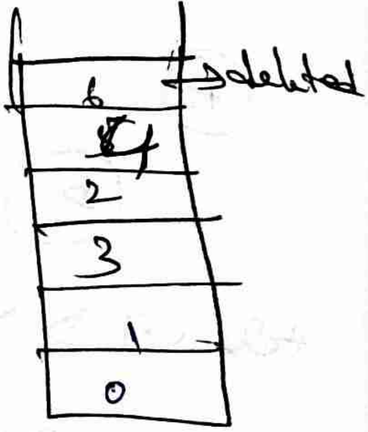


Result: 0

any one adjacent should be inserted in to stack.

depth first traversal. (depth first search).

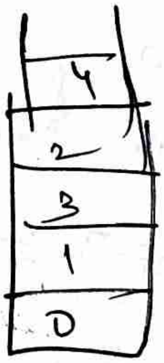
For '0' 1 & 3 are adjacent vertex any one we have to take.



Result : 0 1 3 2 4 6

for '6' all are visived vertex

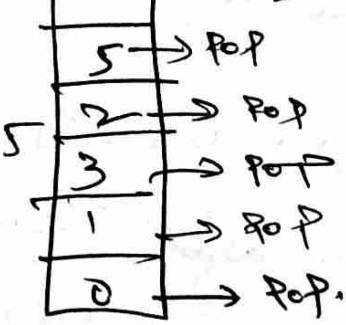
back track.



For '4' unvisited vertices.

deleted '4'

Results 0 1, 3, 2, 4, 6, 5



Now stack is empty.

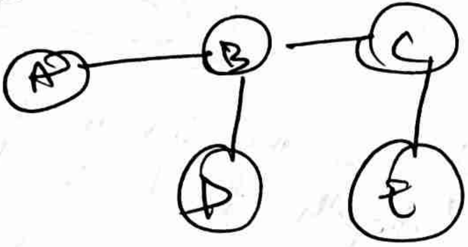
The order 0, 1, 3, 2, 4, 6, 5

Now stack is empty

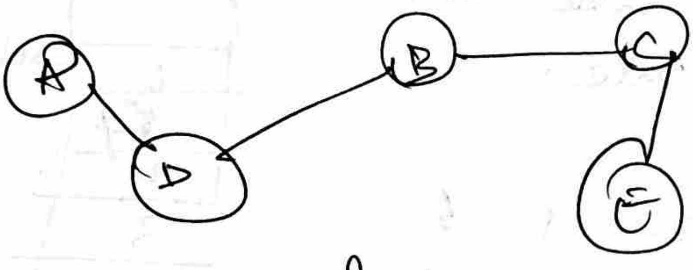
The order of visiting the nodes is

A B C E D

The spanning tree obtained using depth first search is given below



some other possible depth first traversal is



Here (i) D is first successor of A & B is next successor.

(ii) C is first successor.

Algorithm DFT :-

(1) Initialize all the nodes to ready state and stack to empty.
state[V]=1 [∵ 1 indicates Ready state]

(2) [Begin with any arbitrary node 'S' in graph, push it onto stack and change its state to waiting].
State[S] = 2 [∵ 2 indicates waiting state]

(3) Repeat through step 5 while stack is not empty

(4) [Pop node N of stack and mark the status of node to be visited] state[N]=3 [∵ 3 indicates visited]

5. [Push all nodes w adjacent to N into stack and mark their status as waiting]

state[w] = 2

6. If the graph still contains nodes which are in ready state goto step 2.

7. Return

Algorithm Breadth- First [v]

① [Initialize all nodes to ready state]

state[v] =1 [Here V represents all nodes of graph]

② [place stating node 's' in queue and change its state to waiting] state[s] = 2

③ Repeat through step 5 until queue is not empty.

④ [remove a node N from queue and change its status to visited]

state [N] = 3

⑤ [add to queue all neighbours w of 'N' which are in ready state and change their status to waiting state].

state[w] = 2

⑥ Return.

Heap sort :- Heap sort is a comparison based sorting technique based on Binary Heap data structure. It is similar to selection sort where we first find the maximum element and place the maximum element at the end. we repeat the same process for remaning element.

Example :- max Heap

15	20	7	9	30

Heap sort steps :-

Heap is tree based data structure.

* const. Heap tree → Desecding - max heap
→ Ascending min "

* Delete root node ie replace it with last leaf node of tree

* Heapify tree

* Reapeat step 2 ie 3 until heap remaing with single elements.

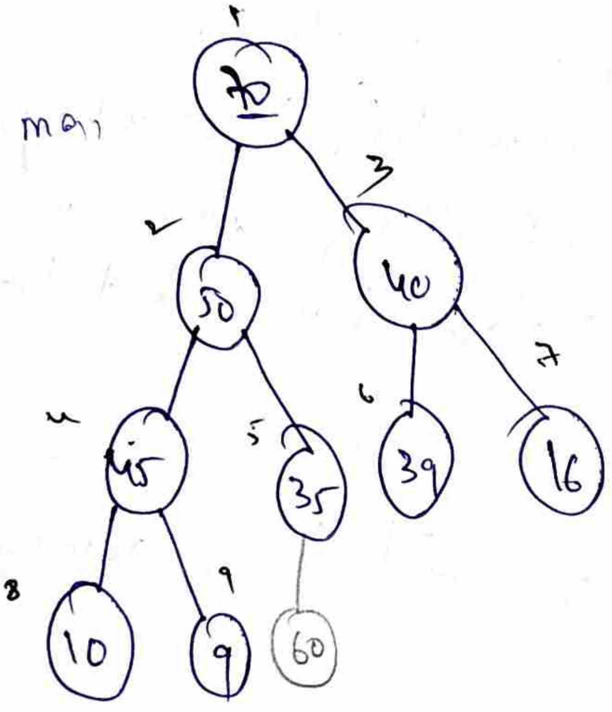
Heap is Complete Binary tree or allmoust complete Binary tree

Max heap

* for every node i, the value of node is less then or equal to its parent value

$A[\text{Parent}(i)] \geq A[i]$

{except root node]



A	70	50	40	45	35	39	16	10	9	60
	0	1	2	3	4	5	6	7	8	

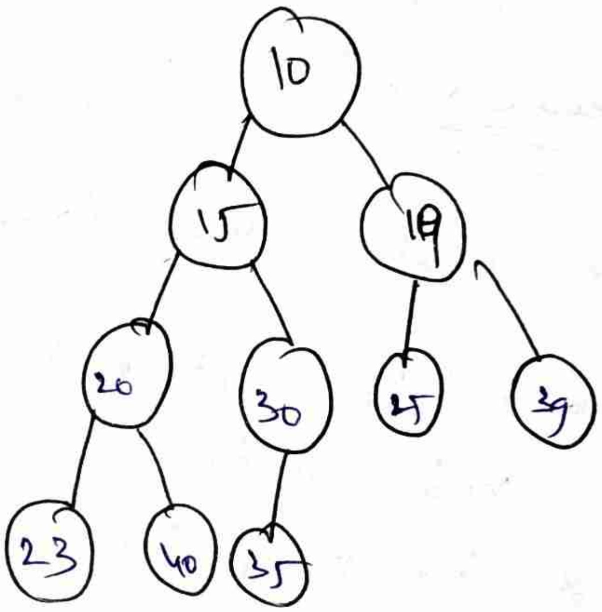
Insert 60

It should be in an Complete binary tree. alway we should insert the data from leaf node. we should not insert data from root.

min heap

for every node i, the value of node is greater then or equal to its parent value

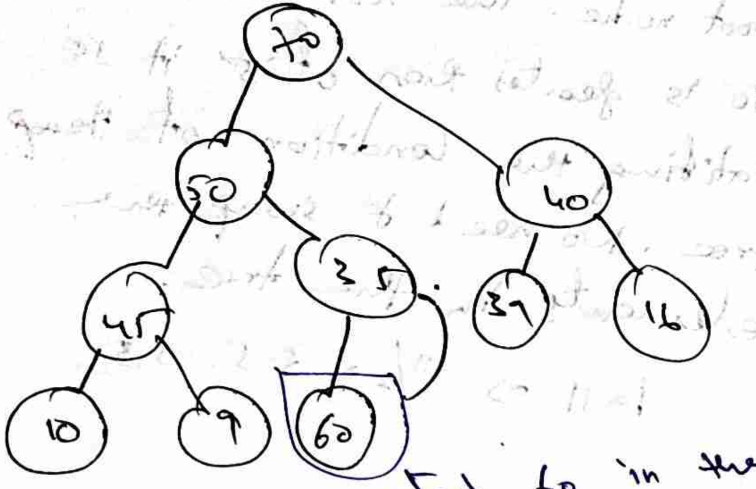
$A[\text{Parent}[i] \leq A[i]$



	1	2	3	4	5	6	7	8
A	10	15	19	20	30	25	39	23

40	35
9	10

Now tree becomeing

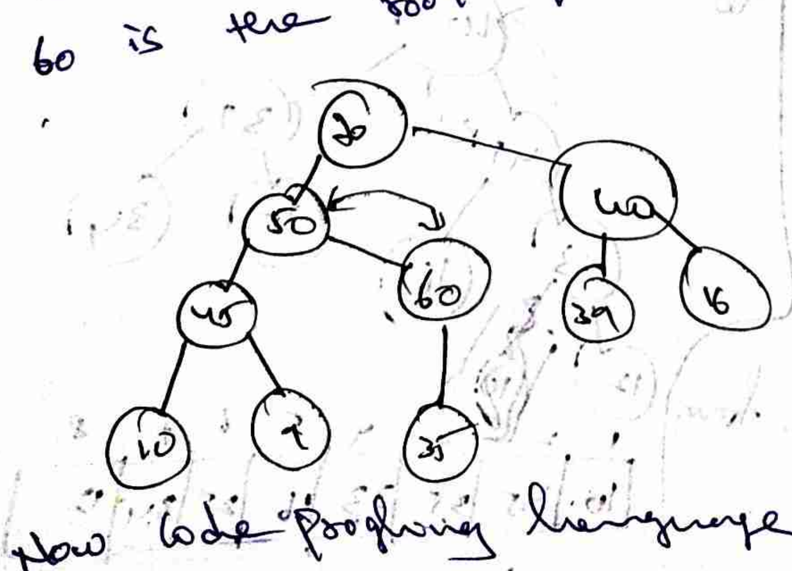


Here we inserted 60 in the left Position but not right bez it should staify the contion of complete binary tree

Now size of alley

1	2	3	4	5	6	7	8	9	10
70	50	40	45	35	39	16	10	9	60

In Heap the condition is Palent should be greater then child but now in our tree 60 is child of '35' but 35 is less then 60 that's we should swap 35 & 60 then '35' is child of 60. 60 is the root of '35' when we swap



Now code programing language

Now 60 is in 10th index
i=10
To find out Palent of '60' formula $i/2 = 10/2 = 5$

Now check in index '5' In index '5'. '35' is data then for 60 '35' is the Palent.

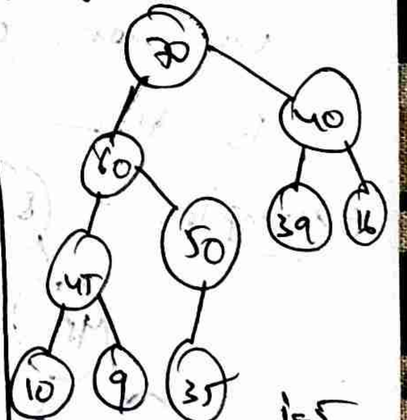
In array

1	2	3	4	5	6	7	8	9	10
70	50	40	45	60	39	16	10	9	35

$i/2$

Compare '60' with Palent.

For '60' palent element is '50' again check heap contion false again swap is required



i=5

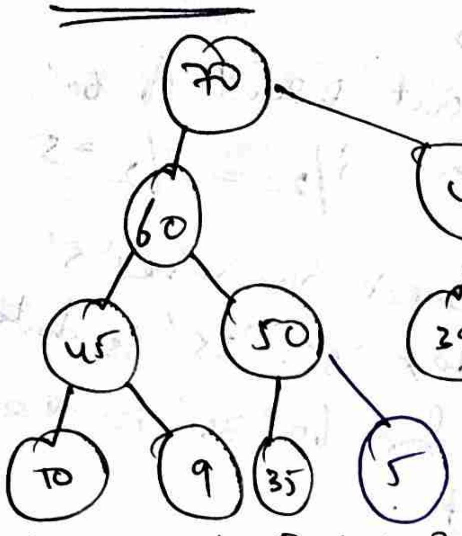
$i/2 = 5/2 = 2.5$

1	2	3	4	5	6	7	8	9	10
70	60	40	45	50	39	16	10	9	35

$i=2 \Rightarrow 2/2=1$

70 is compared with 60. it is true

Insert : 5



1	2	3	4	5	6	7	8	9	10	11
70	60	40	45	50	39	16	10	9	35	5

Now 5 is compared with the root node. The root node is '50' '50' is greater than of '5' it is satifing the Condition of Heap tree. No need to swap the elements in the tree.

i=11 ⇒ 11/2 = 5.5 ⇒ 5

Compare may be.. depended on the height of the tree. The height complete binary tree always log(n). The time taken to insedent any element in Max heap

O(logn)

~~[illegible]~~Deleting the data from tree:

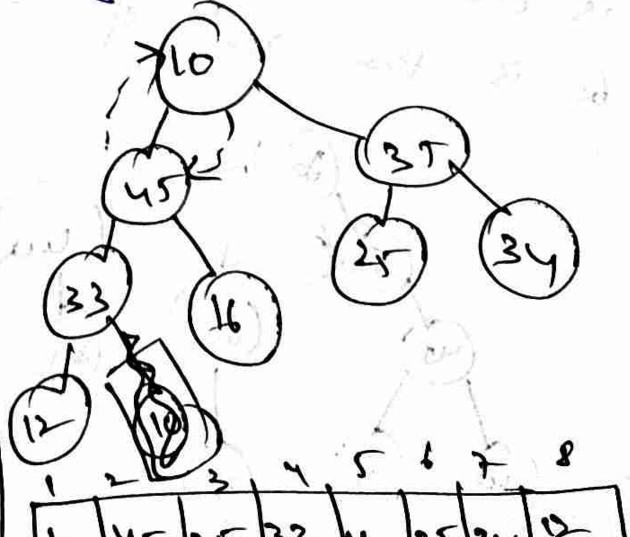
* we Can not delete any data from tree (in Heap). that we can only delete the root node that is the condition

Example:

1	2	3	4	5	6	7	8	9
50	45	35	33	16	25	34	12	10

only 50 can be delete from tree bcz 50 is root node

when we are deleting 50 from tree then the root becomes '10'. In Heap the last element in the index of array is '10' : Then '10' becomes '10'

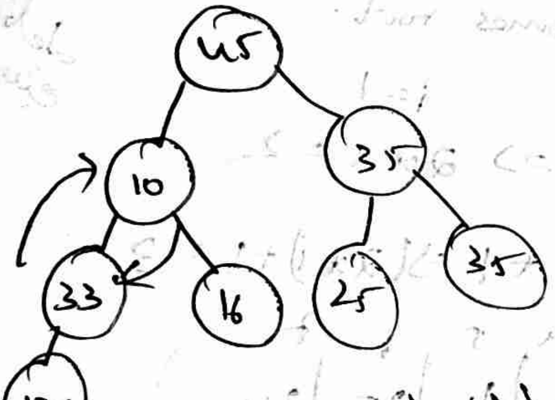


1	2	3	4	5	6	7	8
10	45	35	33	16	25	34	12

Now we should check the tree wheather it is standing the condition Heap tree or not.

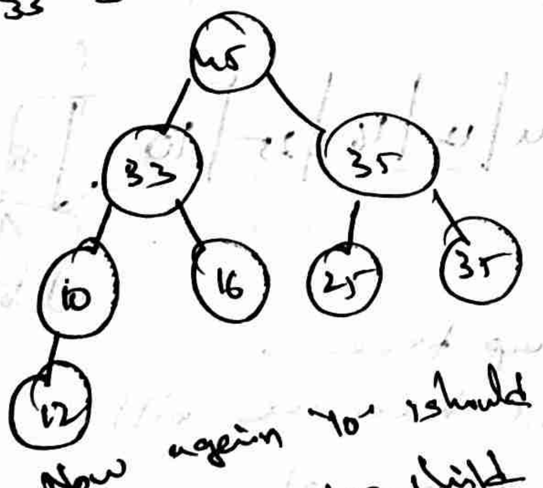
Our Heap tree Condition is parent element should be greater then child element.

Now we should compare '10' with the child nodes '10' has two child one is 45 and 35. from both child '45' is greater then 35 & 10. 45 should becomes root swap is required. Now tree becomes



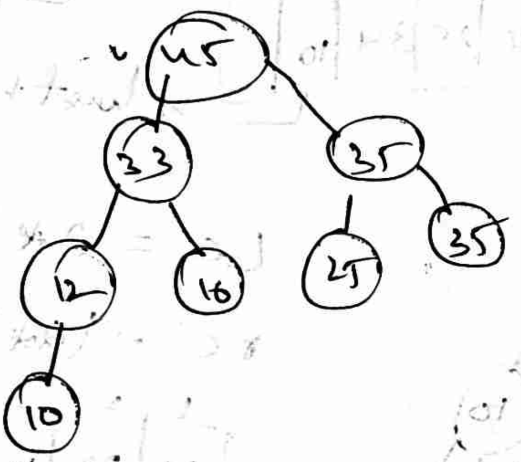
Now agein '10' should compare with child Now same agein swaping is req.

Now '33' is root



Now agein '10' should compare with the child Now agein swoping is req the root.

12 > 3 replaceing 10 with 12 ie 12 with 10



left child = 2*i = 2*1 = 2

right child = (2*i) +1 = 2*1+1 ⇒ 2+1 = 3

1	2	3	4	5	6	7	8
10	45	35	33	16	25	34	12

1	2	3	4	5	6	7	8
45	10	35	33	16	25	34	12

left child = 2*i ⇒ 2*2 = 4

right child = (2*2) +1 ⇒ 4+1 = 5

1	2	3	4	5	6	7	8
45	33	35	10	16	25	34	12

i = 4

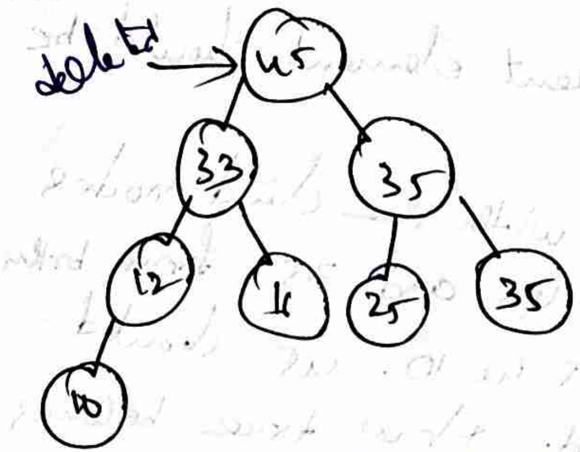
left child = 2*4 ⇒ 8

right child = (2*4)+1 ⇒ 8+1 = 9

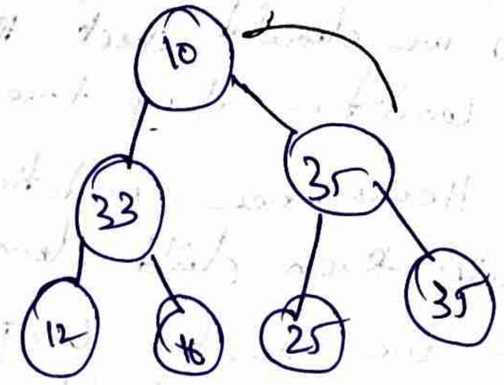
1	2	3	4	5	6	7	8	
45	33	35	12	16	25	34	10	50

deleted

deleted element '45'.



=>

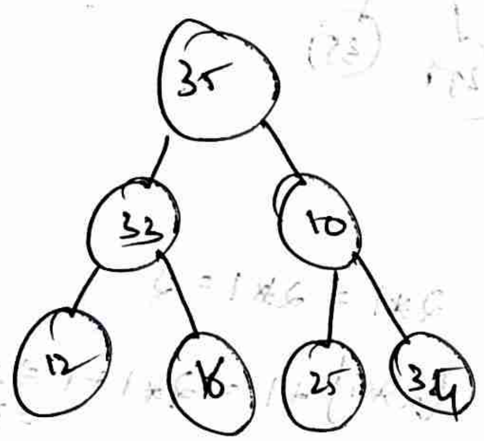


1	2	3	4	5	6	7	8
45	33	35	12	16	25	34	10

last becomes root.

1	2	3	4	5	6	7	
10	33	35	12	16	25	34	45

deleted element



i=1

LC = 2*i => 2*1 = 2

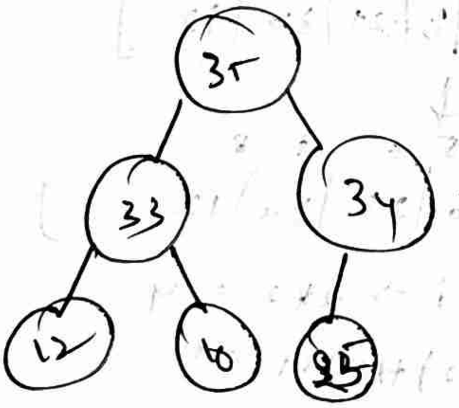
rc = (2*i) + 1 => (2*1)+1 = 3.

1	2	3	4	5	6	7
35	33	10	12	16	25	34

i = 3

lc = 2*3 = 6

rc = (2*3)+1 => 6+1 = 7



35	33	34	12	16	25	10	45	50

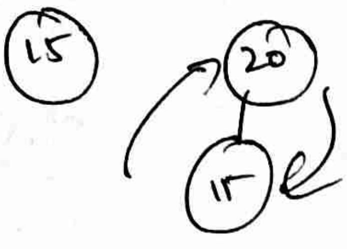
deleted

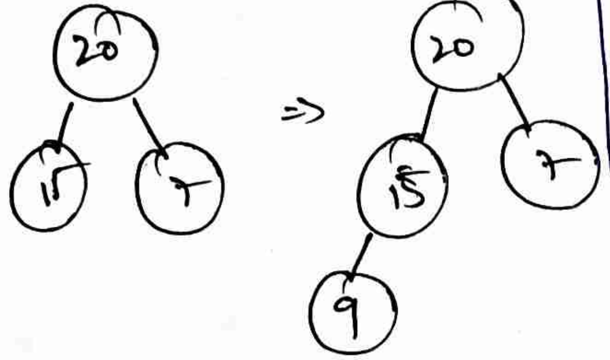
To sort the elements in the Heap tree. we should delete all the element then we will get sorted data in ~~ab~~ above example.

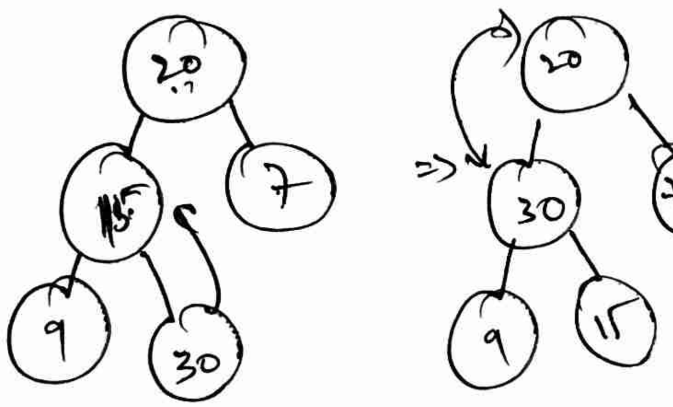
45 & 50 we deleted both in sorted order.

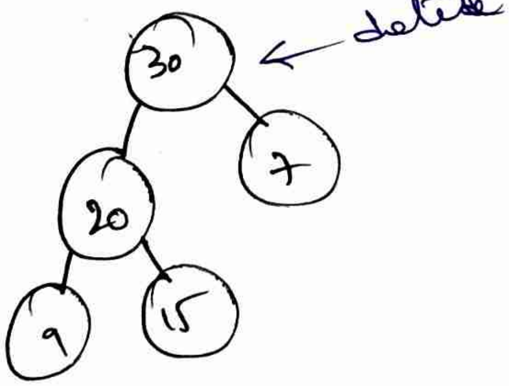
Examples

1	2	3	4	5
15	20	7	9	30









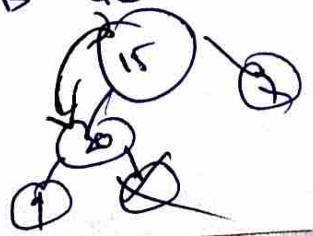
delete

1	2	3	4	5
30	20	7	9	15

A =

delet the data then we get sorted data.

30 is deleted.

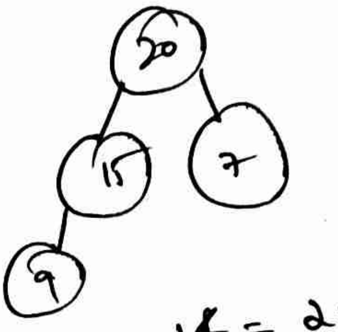


1	2	3	4	
15	20	7	9	30

free space

Next => condition of Heap (max)

swapping

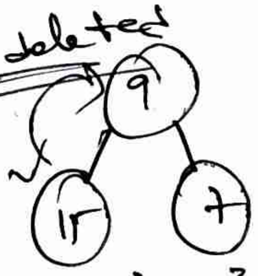


i=1 => lc = 2 * 1 => 2

rc = (2 * 1)+1 => 3

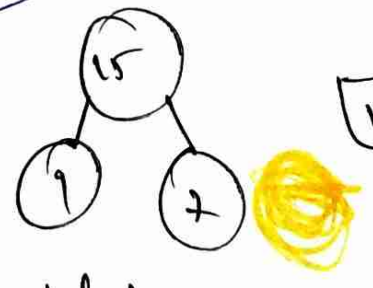
20	15	7	9	30

20 deleted



1	2	3		
9	15	7	20	30

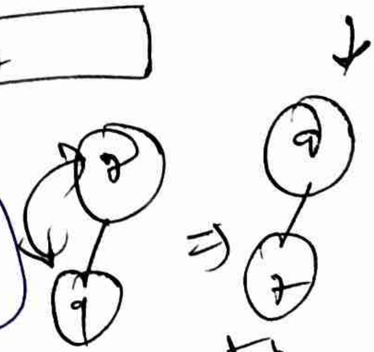
sorted data



15	9	7

15 deleted

9	7



7	9	15	20	30

sorted

7 deleted

7	9	15	20	30

dated 7

Merge Sort:- Merge sort is an

External sorting process:- External sorting process is used when the no. of elements (or records) to be sorted are in large number, such that all of them cannot be accommodated in the internal memory of computer. Therefore, these files containing ~~Large~~ huge records to be sorted are stored on external storage devices i.e external sorting process is applied.

The external sorting process performs the following steps,

✱) Bring few records (from external storage) into the main memory.

✱ Apply internal sort algorithm on those records to generate "runs"

✱ write "runs" onto the external storage devices.

✱ Merge the "runs" generated in the steps above.

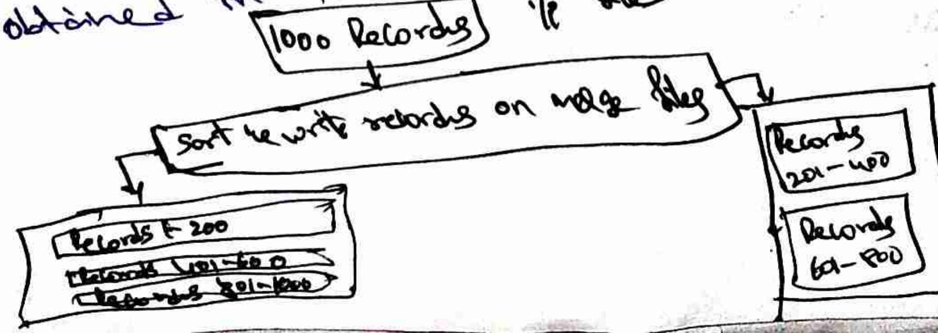
✱ Repeate steps (ii), (iii) & (iv) until all runs are merged i.e a single "run" which ✱ is a sorted file of records are left out.

Example:- Consider a file containing 1000 records. But main memory can accommodate only 200 records at a time. Therefore external sorting technique is applied as follows,

✱ Read the first 200 records from the i/p file, sort them & write them to ano/p merge file (say merge A).

✱ Read another 200 records, sort them, & write them to an alternate merge file (say merge B).

✱ Again another 200 records are read from i/p file, sorted & written to merge file merge A. This process is repeated until all records are read & sorted. The figure below shows the situation that is obtained in the sort phase.



There are few application that deal with very large i/ps. But the main memory cannot have enough space to store such large inputs. A solution for this problem is to use external sorting algorithms. The design of these external sorting algorithm is meant to deal with very large inputs.

External sorting depends much on the storage device being used, unlike internal sorting. A tape is a mass storage device and is the most restrictive one. Because, to access an element, the tape is wound to its corresponding location Thus, for an efficient use of tapes, its elements can be accessed in any direction of the tape, but in seauental order.

Examples of external sorting algorithm that use tapes as their storage devices are.

* multiway merge

* Polyphase merge.

Multiway Merge :- A multiway merge is an n-way merge that uses additional tapes and makes sorting of i/p more simpler by minimizing the total no. of passes.

let T_{x1}, T_{x2}, T_{x3} ----- T_{xn}, T_{y1}, T_{y2}, T_{y3} --- T_{yn} be the tapes. out of these 2n tapes, half of them will work as i/p tapes i.e the other half will work as o/p tapes. The x&y tapes can work as i/p or o/p tapes depending on the algorithm

let T_{x1} work as i/p tape, then the data i/p will be placed on T_{x1}. 75 88 1 80 5 29 8 90 15 61 32 69 7

T_{x1},

T_{x2}

,

T_{xn},

T_{y1}

T_{y2}

Graph Representation:-

- * Adjacence matrix
- * Incidence matrix
- * Adjacence list.

Adjacence matrix:- A Graph G = (V, E) where V = {0, 1, ---- n-1} can be represented using two dimensional array of size n×n.

int adj [20] [20] can be used to store a graph with 20 vertices

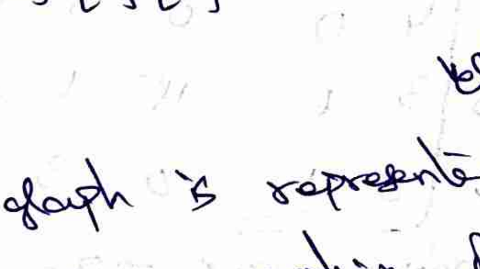
⇒ adj [i] [j] = 1, indicates presence of edge b/w two vertices i & j

adj [i] [j] = 0 absence of edge b/w two vertices i & j

→ A graph is represented using square matrix

→ Adjacency matrix of an undirected graph is always Symmetric matrix i.e an edg(i,j) implies edge(j,i)

→ Adjancency matrix of a directed graph is never Symmetric matrix adj [i] [j] = 1 indicates a directed edge from vertex i to j.

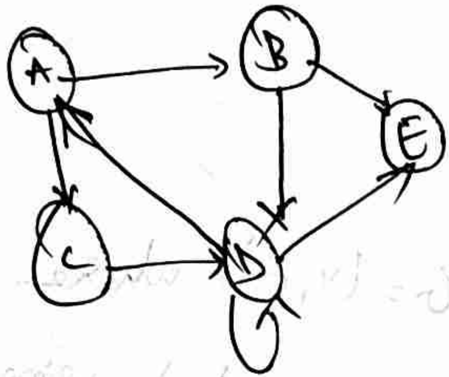


(A, B, C, D, E) vertex

$$
\begin{array}{c|ccccc}
\text{vertex} \rightarrow & A & B & C & D & E \\
\hline
A & 0 & 1 & 1 & 1 & 0 \\
B & 1 & 0 & 0 & 1 & 1 \\
C & 1 & 0 & 0 & 1 & 0 \\
D & 1 & 1 & 1 & 1 & 1 \\
E & 0 & 1 & 0 & 1 & 0
\end{array}
$$

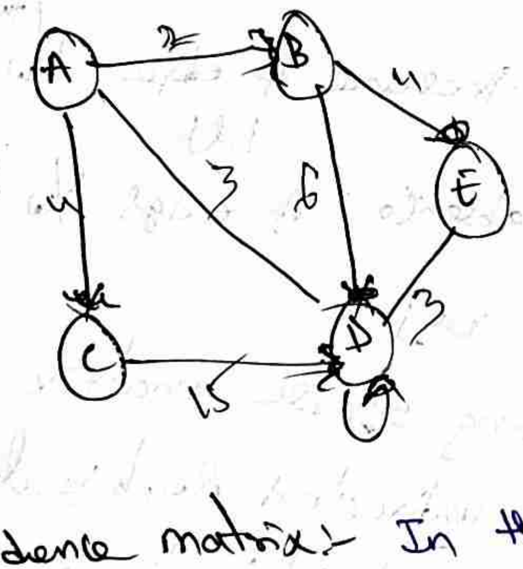
↑ vertex

Directed graph representation:-



$$\begin{array}{c|ccccc} & A & B & C & D & E \\ \hline A & 0 & 1 & 1 & 0 & 0 \\ B & 0 & 0 & 0 & 1 & 1 \\ C & 0 & 0 & 0 & 1 & 0 \\ D & 1 & 0 & 0 & 1 & 1 \\ E & 0 & 0 & 0 & 0 & 0 \end{array}$$

weighted graph representation:-
undirected graphs



$$\begin{array}{c|ccccc} & A & B & C & D & E \\ \hline A & 0 & 2 & 4 & 3 & 0 \\ B & 2 & 0 & 0 & 6 & 11 \\ C & 4 & 0 & 0 & 15 & 0 \\ D & 3 & 6 & 15 & 1 & 3 \\ E & 0 & 11 & 0 & 3 & 0 \end{array}$$

Incidence matrix:- In this matrix, rows represents vertices

" columns " edges

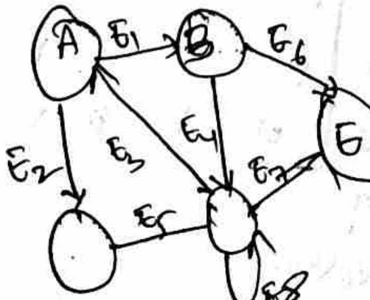
This matrix is filled with either 0 or 1 or -1

Here 0 represented row edge is not connected to column vertex

1 " " " " " outgoing edge to column vertex

-1 " " " " " incoming edge to column vertex

Ex

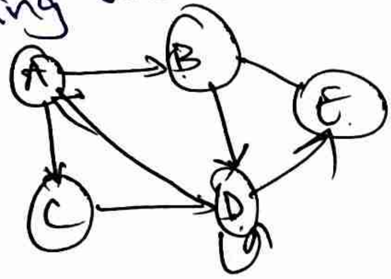


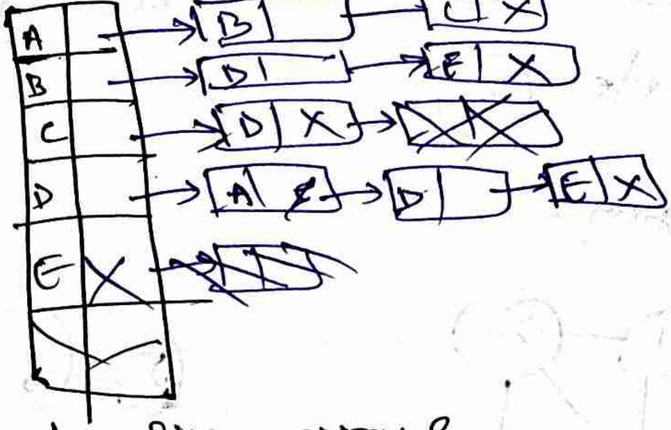
$$\begin{array}{c|cccccccc} & E_1 & E_2 & E_3 & E_4 & E_5 & E_6 & E_7 & E_8 \\ \hline A & 1 & 1 & -1 & 0 & 0 & 0 & 0 & 0 \\ B & -1 & 0 & 0 & 1 & 0 & 1 & 0 & 0 \\ C & 0 & -1 & 0 & 0 & 1 & 0 & 0 & 0 \\ D & 0 & 0 & 1 & -1 & -1 & 0 & 1 & 1 \\ G & 0 & 0 & 0 & 0 & 0 & -1 & -1 & 0 \end{array}$$

↑
vertices

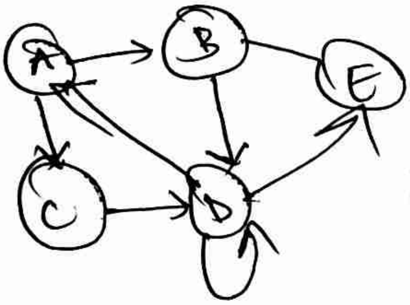
Adjacency list:- In this representation, every vertex of graph contains list of its adjacent vertices.

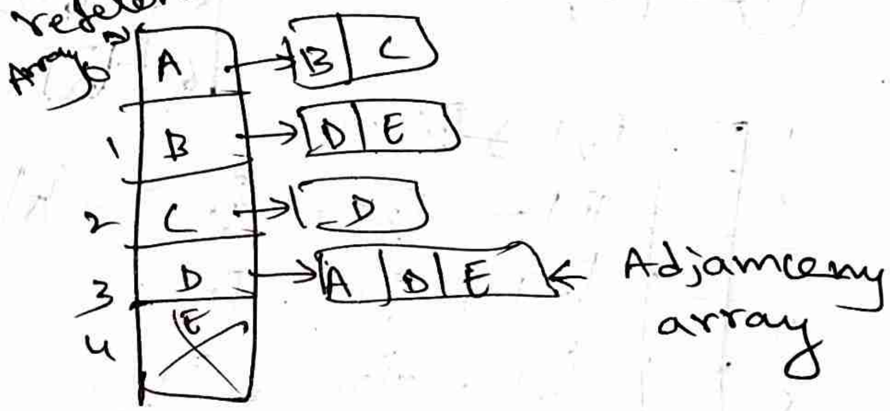
Example:- (1) Consider directed graph representation. implemented using linked list





* It can also be implement using arrays.

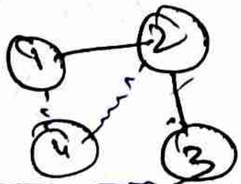


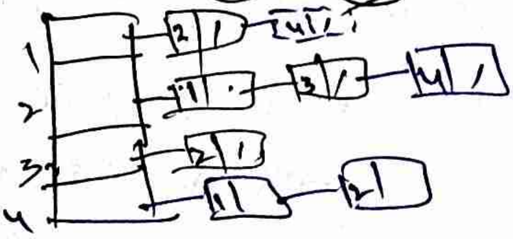


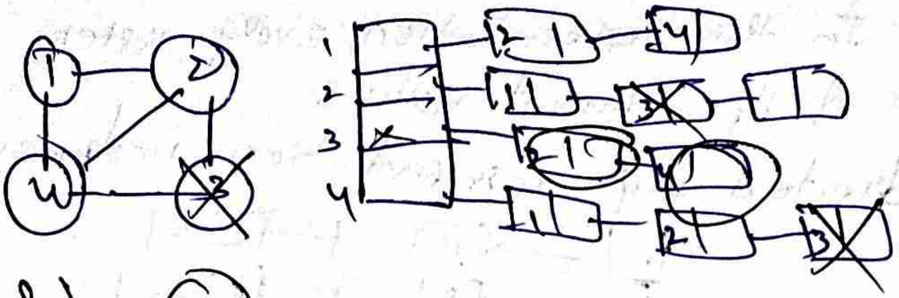
operation on graph:-

* Incertion (Adding vertices / edges)
* Delection (Removing " ")
* merging (merging the graph)
* Travelsal (To touch all the vertices)

graph:- we want to add (4)

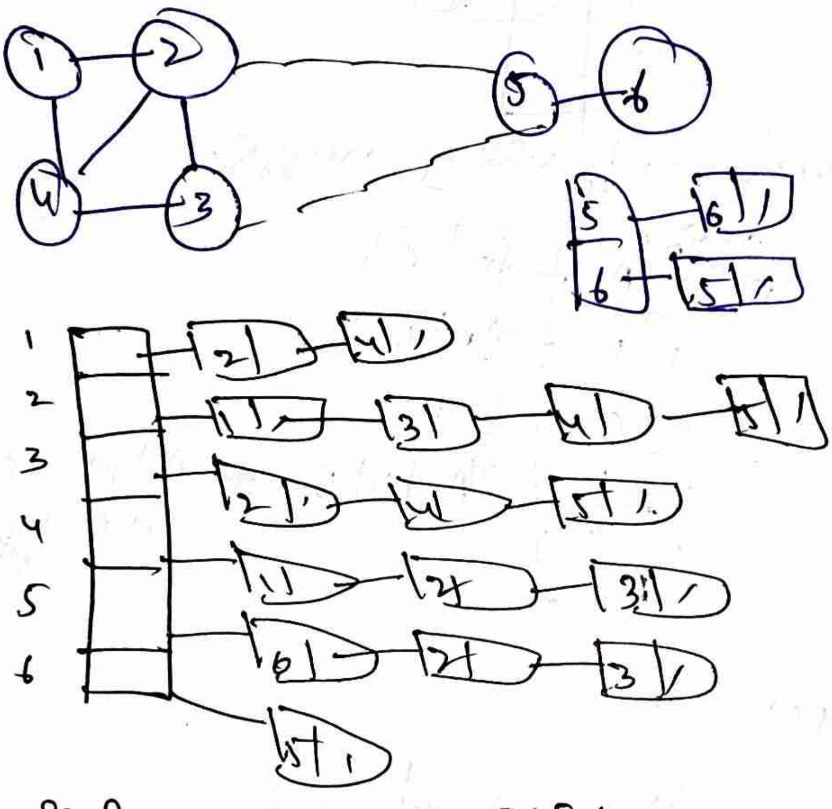






Delete ③

Merging :-



Traversal :- BFS or DFS.

UNIT - V

Pattern matching Alg:

In this we try to check ewether the given patter in present in the Existing string (or) not.

For this we learn regarding 3 Alg.

1. Brute Force

2. Knuth - moris prat

3. Booyer - moore Alg.

Brute Force Alg:

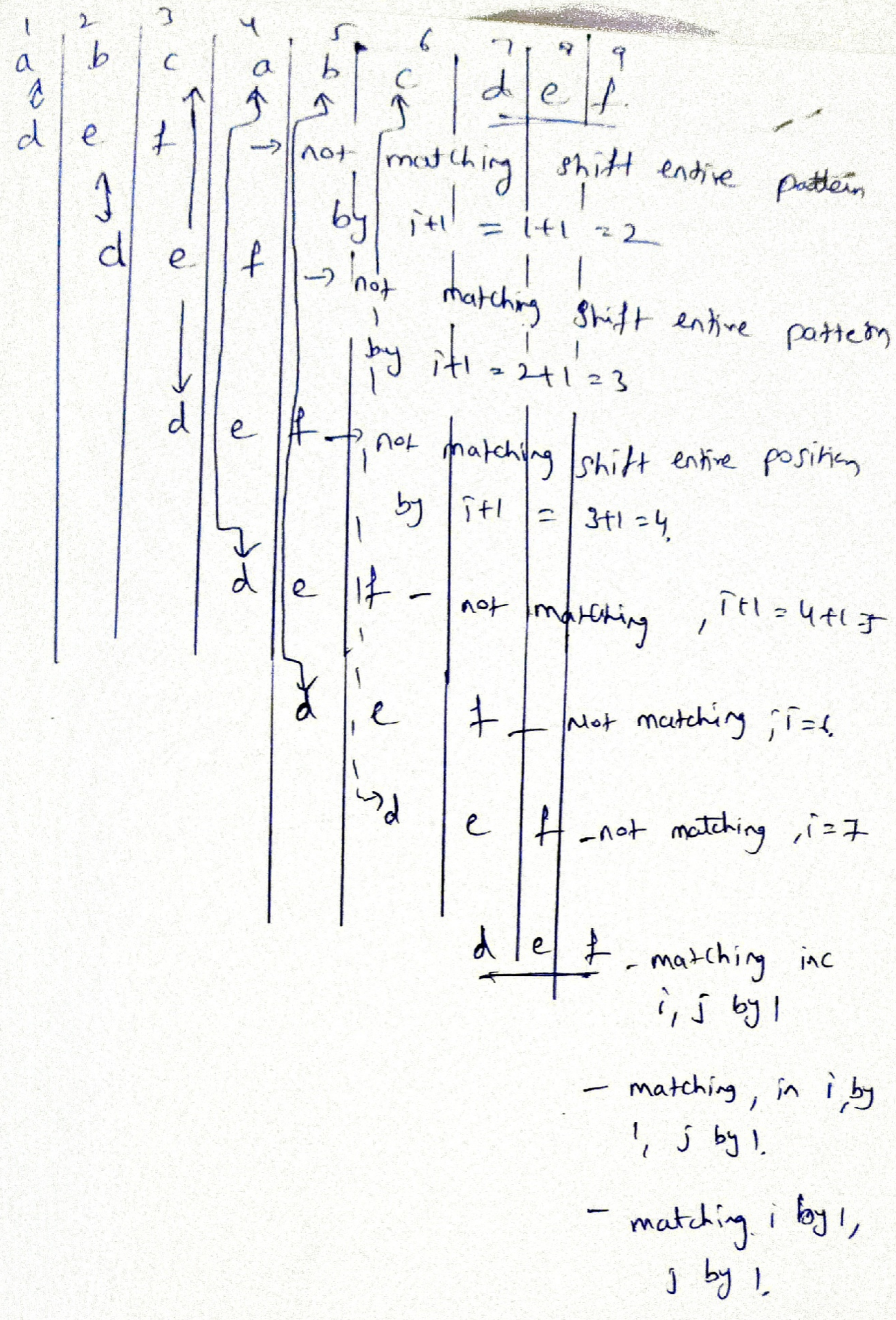
It is simplest of all the algorithms.

1) ~~It~~ compare the string (i) with pattern (j)

2) If i is not matching with j, then shift entire patter by i+1 and j=0.

3) If i matching with j, then increment i & j.

Let us see with an example.



→ not matching shift entire pattern by i+1 = 1+1 = 2

→ not matching shift entire pattern by i+1 = 2+1 = 3

→ not matching shift entire position by i+1 = 3+1 = 4

– not matching , i+1 = 4+1 = 5

– not matching , i = 6

– not matching , i = 7

– matching inc i, j by 1

– matching , in i, by 1, j by 1.

– matching i by 1, j by 1.

Pattern found,

Brute force pattern matching

LS length of original string = 10
LP " " " patter. = 4

Man = (LS-LP+1) = 10-4+1
= 6+1 = 7. time we can match the pattern.

```
void brute(s,p)
  {
    ls = length(s);
    LP = length(p);
    man = (LS-lp+1);
 for(i=1; i<=max; i++)
   {
     flag = true;
     for(j=1; j<=lp && flag==true ; j++)
      {
        ~~if(p[j] ≠ s[j+1])~~ if(P[j] ≠ s[j+i-1
           {
             flag = false;
           }
        }
      if (flag == true)
         {
           return i;
      }}
   }. return 0;
```

Knuth- Morris pratt algorithm:

try i c # a d
find b a b

	1	2	3	4	5	6	7	8
string:	a	b	c	d	e	f	g	h

Patten

present alg.

a

Disadv of Brute force: alphabets are compared Repeatedly whenever there is a mismatch, backtracking of i is done more frequently

Eg:

a	b	c	d	a	b	c	a	b	c	d	f
1	2	3	4	5	6	7	8	9	10	11	12.

Pattern

a	b	c	d	f
1	2	3	4	5.

Now lets trace.

i	i	i	i	i							
a	b	c	d	a	b	c	a	b	c	d	f
"	"	"	"	#							
a	b	c	d	f							
j	j	j	j	j							

then move j to 1 & i to $\overset{i=2}{i+1.}$, again compare.

	i	i									
a	b	c	d	a	b	c	a	b	c	d	f
	#										
	a	b	c	d	f						
	j										

②

```
        i  i
  a  b  c  d  a  b  c  a  b  c  d  f
        #
        a  b  c  d  f
```

```
           i
=>  a  b  c  d  a  b  c  a  b  c  d f
           #
           a  b  c  d f
```

```
                i  i  i  i
=>  a  b  c  d  a  b  c  a  b c  d  f
                "  "  "  #
                a  b  c  d  f
                   j  j  j
```

Again move to j to 1 & index again back to 6.

```
                     i
=>  a  b  c  d  a  b  c  a  b c  d f
                     #
                     a  b  c  d f
```

```
                        i
=>  a  b  c  d  a  b  c  a  b  c  d f
                        #
                        a  b  c  d f
```

```
                              i
=>  a  b  c  d  a  b  c  a  b  c  d  f
                              "  "  "  "  "
                              a  b  c  d  f
```

Stop.

There is a lot of wotte of time each there is a mismatch at last position we are moving entire patten back along with the main string.

→ Brute force – Takes more no of comparis

To avoid backtracking of i , and to get less compuision , we go for knuth-moris prat algorithm. Write x tab

1. We have to create a pi table for pattern.

let us learn with the examples.

P1 : a b c d a b e a b f

Steps : Write the index of every alphabet, if the alphabet is repeating elsewhere in the string just put the index of the alphabet, Remaining all 0's.

	1	2	3	4	5	6	7	8	9	10
P1 :	a	b	c	d	a	b	e	a	b	f
	0	0	0	0	1	2	0	1	2	0

	1	2	3	4	5	6	7	8	9	10	11
P2 :	a	b	c	d	e	a	b	f	a	b	c
	0	0	0	0	0	1	2	0	1	2	3

	1	2	3	4	5	6	7	8	9	10
P3 :	a	a	b	c	a	d	a	a	b	e
	0	0	0	0	1	0	1	2	3	0

	1	2	3	4	5	6	7	8	9
P4 :	a	a	a	a	b	a	a	c	d
	0	1	2	3	0	1	2	0	0

Write the patter with pi table in a proper ③ -table and start comparing from i = 1 & j = i+1, by taking j as '0'.

Step 3: if i & j are matching increment i, increment j,

Step 4: if i & j are matching, move back j to the index based on pie table. compare i with j.

Step 5: if j at starting position and cannot be moved back further then increment 'i' by 1.

Step 6. Repeat step 3, 4, 5 until the process on string reaches the end.

Let us see with an example

→ String: a b a b c a b c a b a b a b d

String:

a	b	a	b	c	a	b	c	a	b	a	b	a	b	d.
1	2	3	4	5	6	7	8	9.	10	11	12	13	14	15

Pattern

1	2	3	4	5
a	b	a	b	d
0	0	1	2	0.

Now lets start tracing

i
a b a b c a b c a b a b a b d.
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15

j°

a	b	a	b	d
0	0	1	2	0
1	2	3	4	5

=> Compare i with j+1. , a & a matching move i & j

i
=> a b a b c

j

a	b	a	b	d
0	0	1	2	0

=> compare i with j+1, matching, inc i & j

i
a b a b c — i with j+1

j
a b a b d — matching, inc i & j

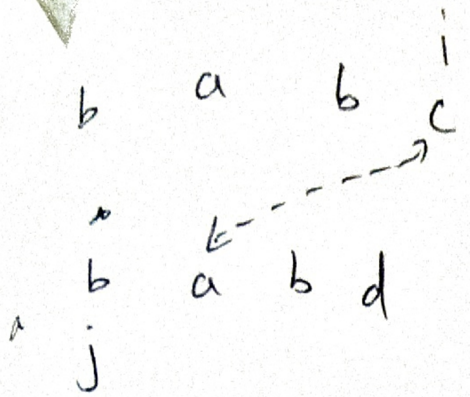
i
=> a b a b c — matching, inc i & inc j

j
a b a b d

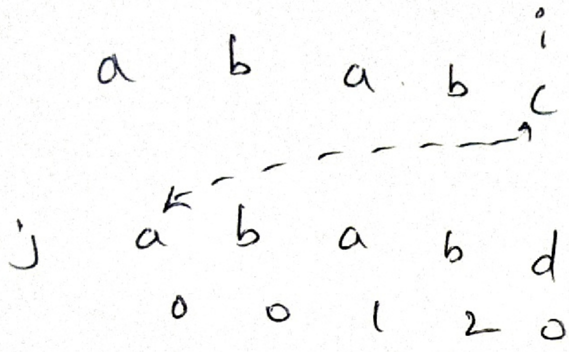
i
a b a b c

j
a b a b d
0 0 1 2 0

=> Not matching, move j back to index.

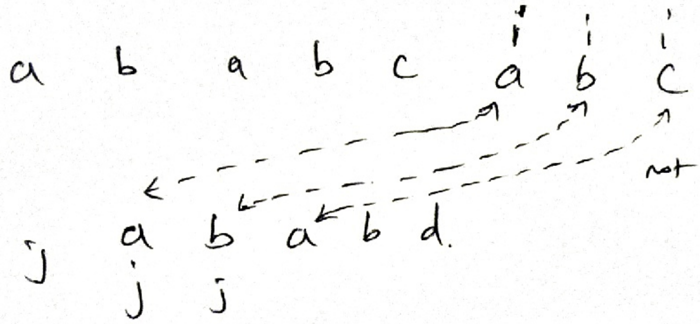


Compare i with j+1, not matching, move j back to index position.



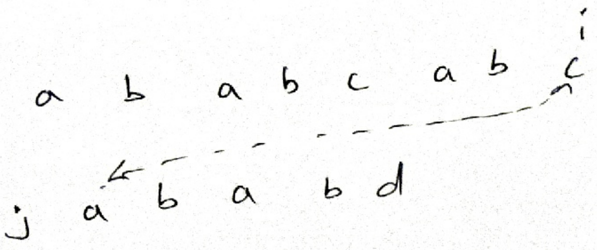
compare i with j+1, not matching & j cannot be move further then increment i

⇒



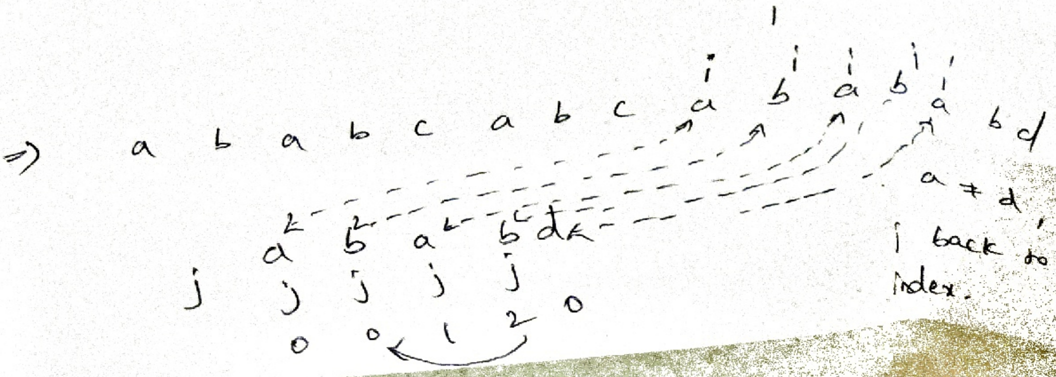
not matching

⇒ Not matching move j back to index position.

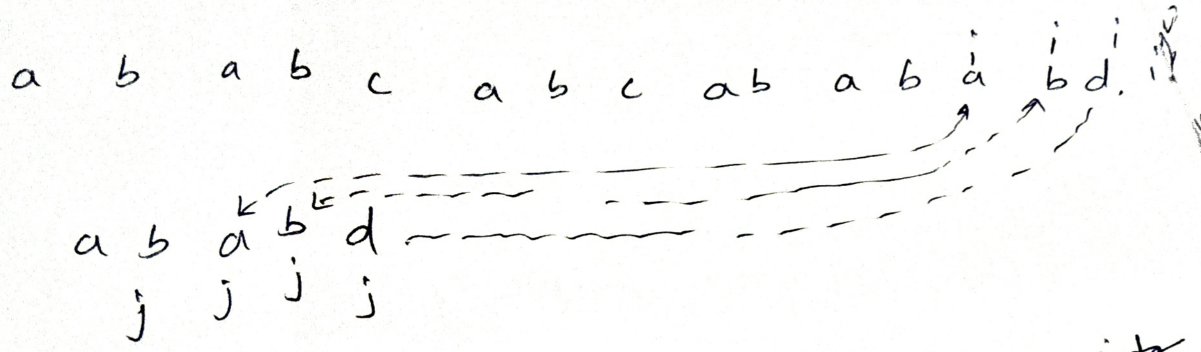


not matching, j cannot be moved back further, increment

⇒



a ≠ d, move j back to index position.



⇒ End of pattern, patten found, print the ~~index position~~

of pattern found.

Booyer-moore Pattern matching Algo

Boyer - More Algorithm It is the most efficient Algorithm among all the remaining Alg.

→ In this we try to compare from Right to left

→ We calculate ~~Bad~~ Bad - match table. for pattern.

Bad - match table.

1- Identity all the distinguished alphabets.

2, Draw them in a table

3. calculate shift of all the alphabets.

4. If we get shift of 2 common alphabets, take the minimum value.

5. Always the shift of last alphabet is total no of alphebets (count of alphabets)

The formula for shift is

$$\boxed{\text{Shift} = \text{length} - \text{index} - 1.}$$

Now let us calculate the bad-match table for the given pattern.

Note:- should also consider spaces.

AT – THAT

1. Identify all the distinguished alphabets.
↓
who are repeating write them only once.

0 1 2 3 4 5 6.
→ A T – T H A T

→ Total length = 7.

→ Distinguished alphabets.

Pattern	A	T	–	H	*
Shift	1	3	4	2	7.

2. Calculate shift of every alphabet
shift = Totallength – index – 1

shift (A) – 7 – 0 – 1 = 6

shift (T) – 7 – 1 – 1 = 5

shift(–) – 7 – 2 – 1 = 4

shift (T) – 7 – 3 – 1 = 3

shift (H) – 7 – 4 – 1 = 2

shift(A) — 7 − 5 − 1 = 1.

shift(T) — Total length = 7.

→ For shift(A) we have 6, 1 ⇒ As 1 is minimum, we take shift(A) = 1

→ shift(T) = 5, 3, minimum = 3.

→ shift(–) – only one – take one

→ shift(H) – only one – take one

→ shift(*) – Another alphabet – Total length = 7

Steps for Boyer-more Alg

1. Calculate the Bad-match table.
2. Compare starting from Right to left
3. calculate $d = \max(\underbrace{\text{shift}(C)}_{\text{bad character}} - \underbrace{K}_{\text{No of Character matched}}, 1)$

If the first character is not matched

4. Shift the entire pattern, to d value

4. If pattern ~~pat~~ matching dec j and dec i until matched.

Let us see with an example

WHI,CH-FINALLY-H,ALTS--AT-T

① $d = \max(\text{shift}(c) - k, 1)$

$= \max(\text{shift}(F) - 0, 1)$

$= \max(7 - 0, 1) = \max(7, 1) = 7.$

Shift (F) character not present in pattern so, take 7 value

so shift entire pattern to 7th position.

② $d = \max(\text{shift}(-)_{0} - 0, 1)$

$= \max(4 - 0, 1)$

$= \max(4, 1) = 4.$

shift 4 locations again,

③ $d = \max(\text{shift}(S) - k, 1)$

$\max(7 - 0, 1) = \max(7, 1) = 7$

④ $d = \max(\text{shift}(H) - k, 1)$

$= \max(2 - 0, 1) = \max(2, 1) = 2$

S - W H I C H - F I N A L L Y - H A L T S - - A T - T H A T.

P - A T - T H A T
0 1 2 3 4 5 6

1) Not matching calculate

d = 7

2) A T - T H A T
0 1 2 3 4 5 6

Not matching, cal

d = 4

3) A T - T H A T
0 1 2 3 4 5 6

Not matching, cal

d = 7

4) A T - T H A T
0 1 2 3 4 5 6.

Not matching, cal

d = 2

A T - T H A T

matching

Note: Starting counting after 1 index of insertion.

THIS - IS - A - TEST. - $

TEST - pattern.

→ Calculate Bad match table.

0	1	2	3.
T	E	S	T

→

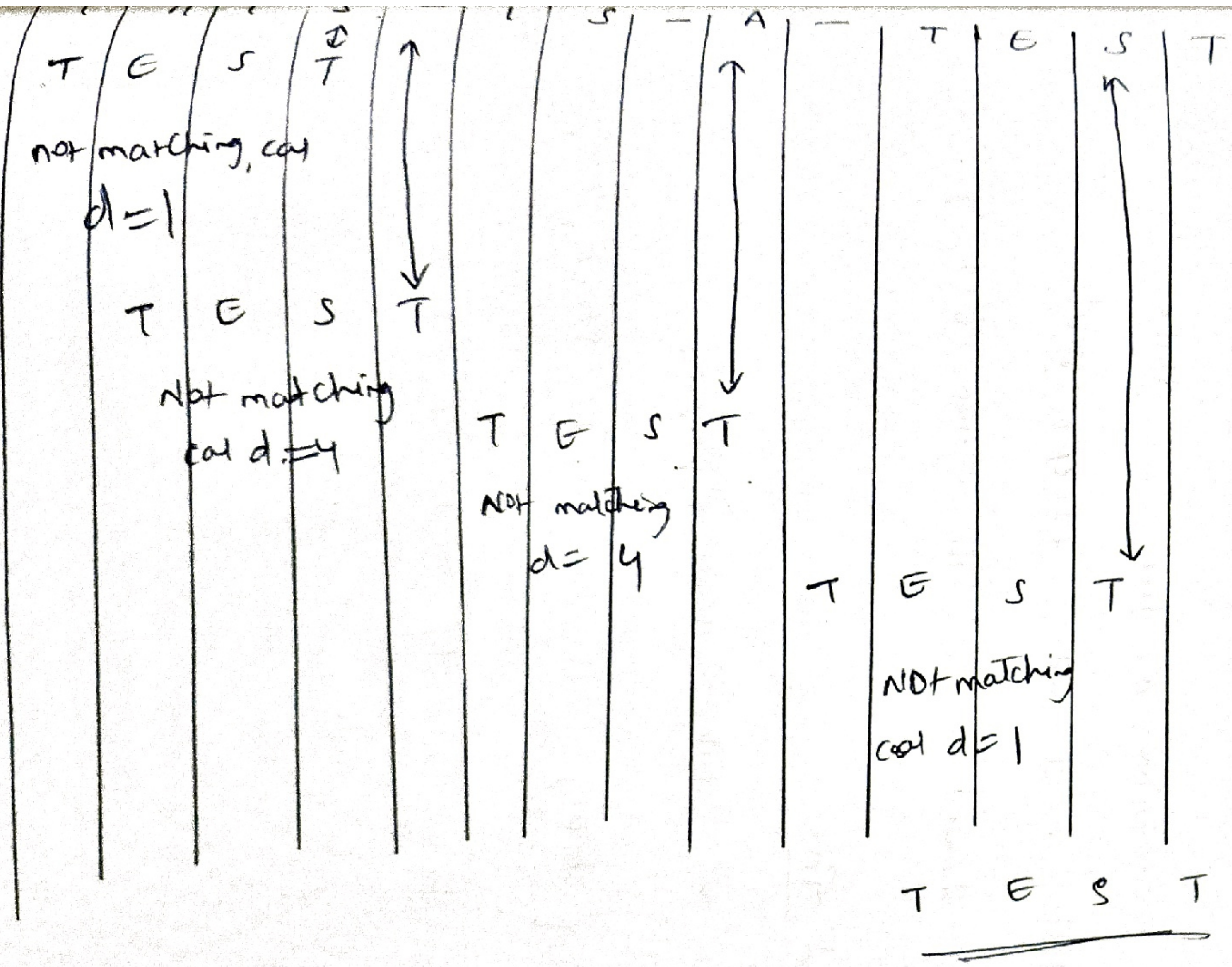
P	T	E	S	*
shift	3	2	1	4

shift(T) - $4-0-1=3$

shift(E) - $4-1-1=2$

shift(S) - $4-2-1=1$

shift(*) - Total length = 4.



① d = max(shift( ) − k , 1)

= max (shift(S) − 0,1)

= max(1,1) = 1.

② d = max (shift(−) − k,1)

= max (~~9~~ 4−0,1) = ~~9~~ 4

shift T location ahead.

③ d = max(shift(A) − k, 1)

= max (4−0, 1) = 4

④ d = max (shift (S) − k, 1)

= max(1 −0, 1) = 1

matching

Standard Tries, compressed Tries, suffix Tries.

Tries: Trying to store a string in the form of tree.

→ It stores set of strings.

→ For storing string we make ~~make~~ use of

1) Standard Trie

2) compressed Trie

3) suffix Trie.

→ For constructing a tree, we take by storing a letter in a node, except root.

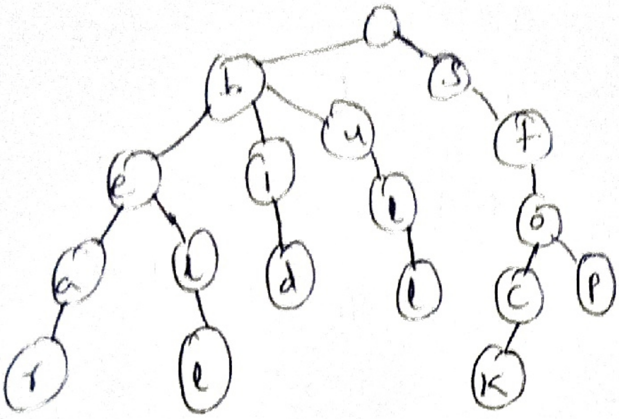
→ letter should be in the given list of string.

Standard Trie

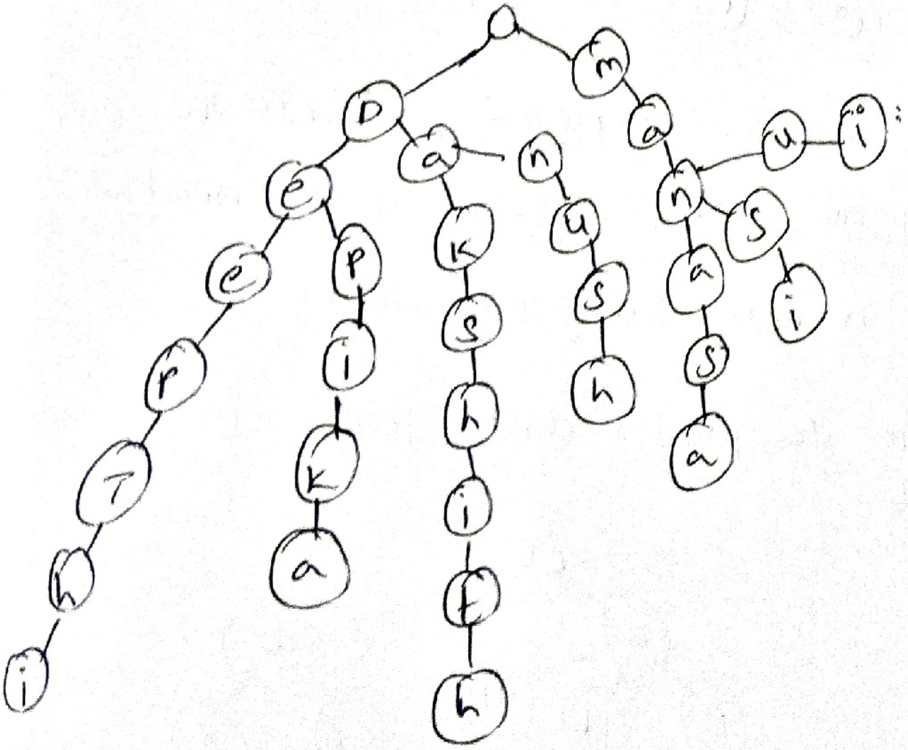
→ let us take an example

S = { bear, bell, bid, bull, stock, stop }

As we have already node for b, e, so we add l, l node to same b



Eg: { Deepthi, Oepika, Dakshith, Danush, Manasa, mansi, manvi }.

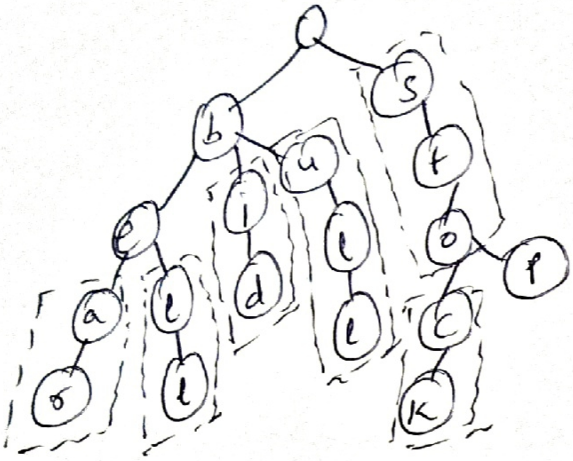


→ Rep set of words from root to leaf, rep the set of words in a set.

Compressed Trie: Rep a standard trie in a more compact (or) more portable fashion.

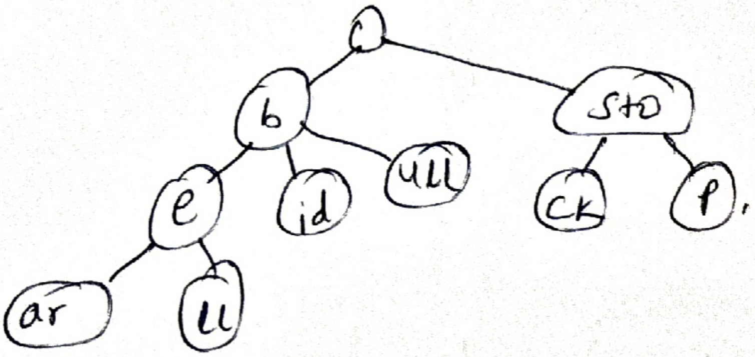
Ex: { bear, bell, bid, bull, stock, stop }.

→ lets again construct the standard Trie.



→ Where we can compress - compress the node having one child (as there is no need to create a node having only child)

⇒ Write the parent & child together in a larger node.



→ The above is the compressed Trie.

Representation of Compressed Trie

0 1 2 3 4

s[0] = b e a r

s[1] = b e l l

s[2] = b i d

s[3] = b u l l

s[4] = s t o c k

s[5] = s t o p.

0, 0, 0 → States that in String s[0], in location [0], starts in location [0], ends.

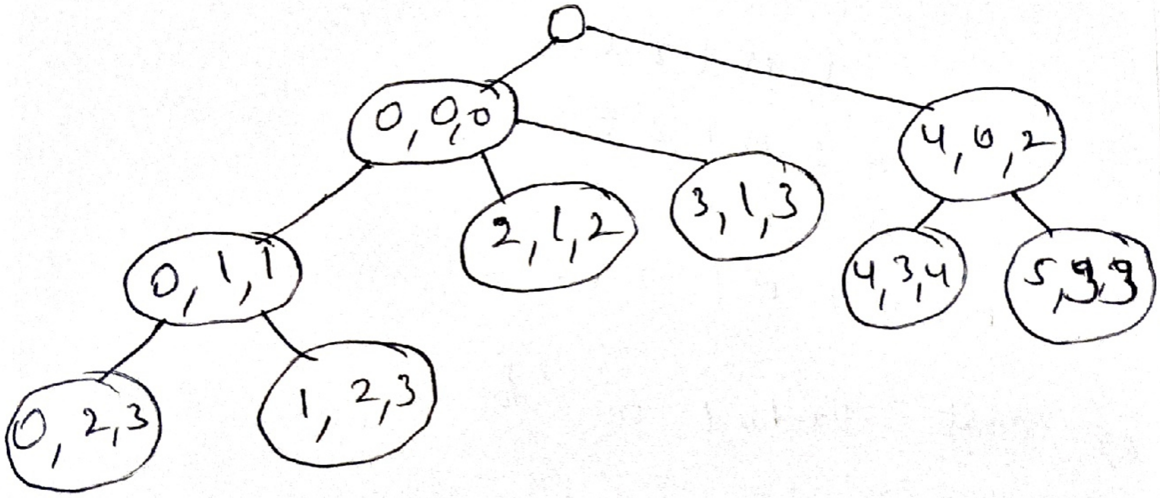
We take, 3 variables.

i, j, k.

i - In which string the alphabet is

j - Where does the ~~[illegible]~~ prefix starts

k - where does the ~~And~~ prefix end.



Suffix Trie : For a given set of strings. we first 1) write all possible suffixes

2) construct it's standard tree

3) construct it's compressed trie.

4) Represent the suffix trie.

~~let us take the same examp~~

let us take an example.

MINIMIZE.

→

1) Write it's suffix

E

ZE

IZE

MIZE

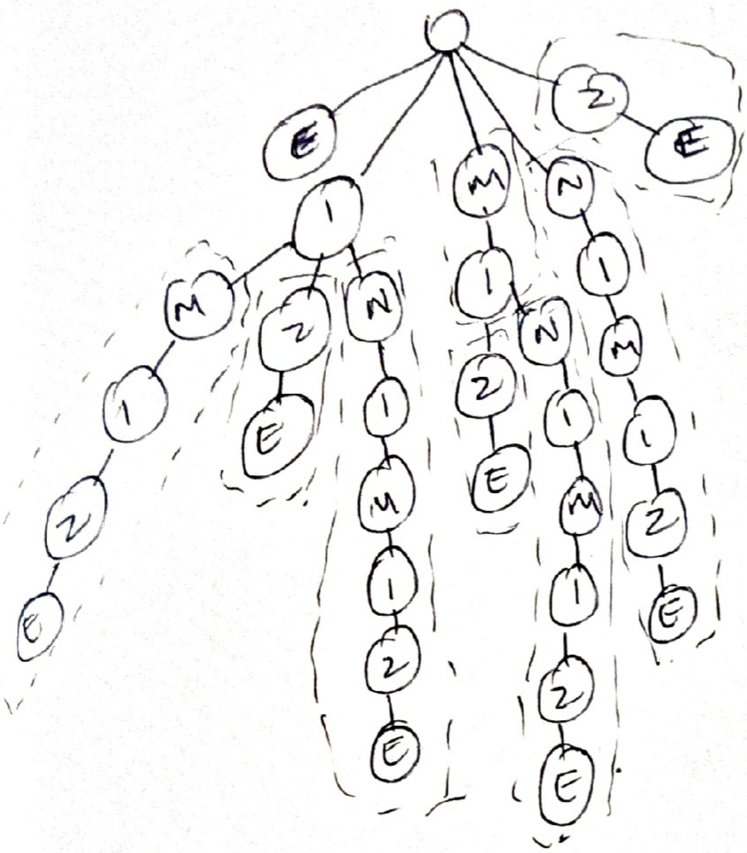
IMIZE

NIMIZE

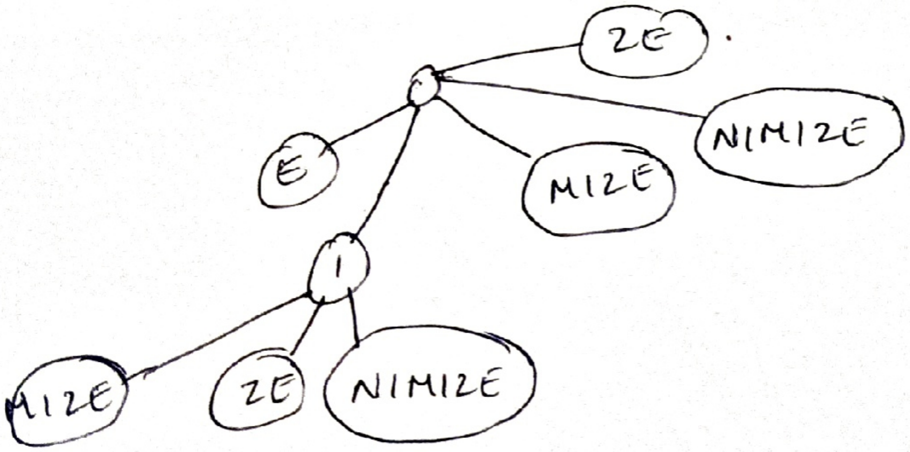
INIMIZE

MINIMIZE.

2) create Standard trie for all above suffix.



3) construct it compressed Trie.



4) Representation of suffix tree

j, k

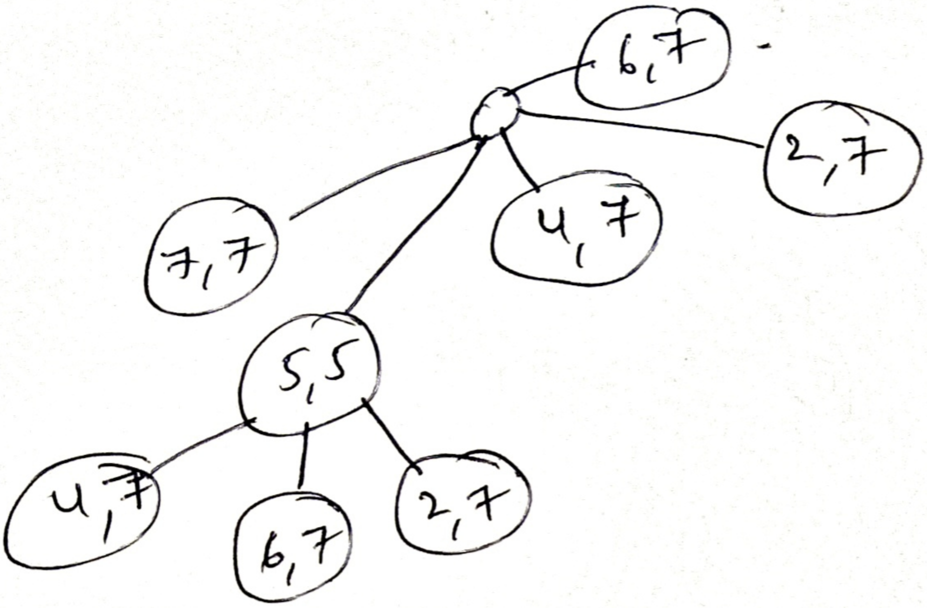
j – starting of suffix.

k – end of suffix.

1

M I N I M I Z E

0 1 2 3 4 5 6 7



The above is the suffix tree.